CODIFICA DELLE INFORMAZIONI

Prof. Marco Camurri

Argomenti

- **Sistemi di numerazione**

  conversioni tra i sistemi decimale, binario ed esadecimale

- **Codifica dei numeri interi senza segno**

  codifica binaria, codifica BCD

- **Codifica dei numeri interi con segno**

  modulo e segno, complemento a due, condizioni di overflow

- **Codifica dei numeri frazionari**

  virgla fissa e virgola mobile, standard IEEE-754

- **Codifica dei colori e dell immagini**

  formato RGB, compressione lossy e lossless

- **Codifica dei caratteri**

  ASCII a 7 bit, codiifiche ASCII estese, Unicode (UTF-32,UTF-16,UTF-8)

- **Codifica dei suoni**

Codifica delle informazioni

- Le memorie dei computer sono in grado di memorizzare sequenze di due soli **simboli**, che chiamiamo **0** e **1**
- 0 e 1 sono dette **cifre binarie** o **bit** (binary digit)
- Vogliamo memorizzare informazioni varie: *numeri*, *lettere*, *immagini*, *suoni*, *pagine web, programmi*
- Ci serve un modo per **rappresentare** tutte queste informazioni come sequenze di 0 e 1
- Dobbiamo stabilire una **codifica dei dati**
- Lo stesso insieme di simboli (es. alfabeto latino) può essere rappresentato usando diverse codifiche!

Codifica dei numeri interi senza segno

- Interi senza segno = interi maggiori o uguali a 0
- Un modo naturale per codificare gli interi senza segno è utilizzare la loro **rappresentazione in base due**
- i numeri così rappresentati si dicono **codificati in binario**

- Nota: la **codifica binaria** non è l'unica codifica per numeri interi usata in informatica, ma è sicuramente la più utilizzata negli attuali computer
- Un'alternativa è la **codifica BCD** (binary-coded decimal) che vedremo più avanti

Sistemi di numerazione

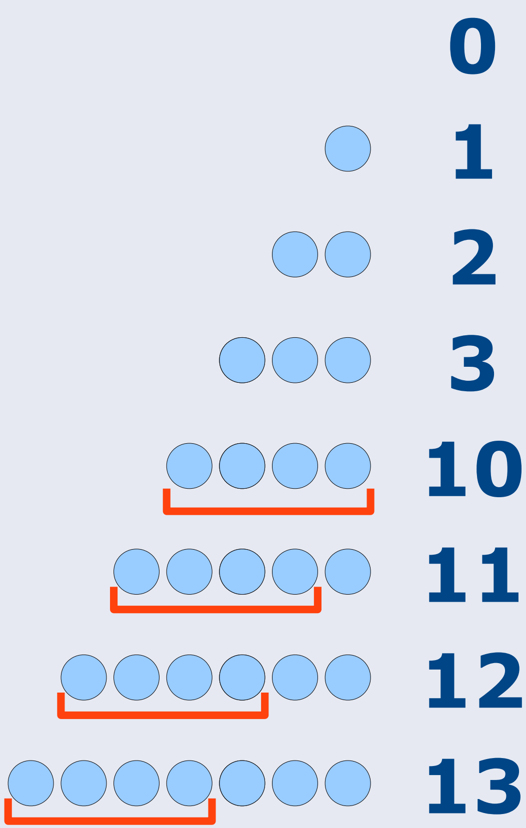
- Tutti sappiamo rappresentare i numeri nel **sistema di numerazione in base 10**

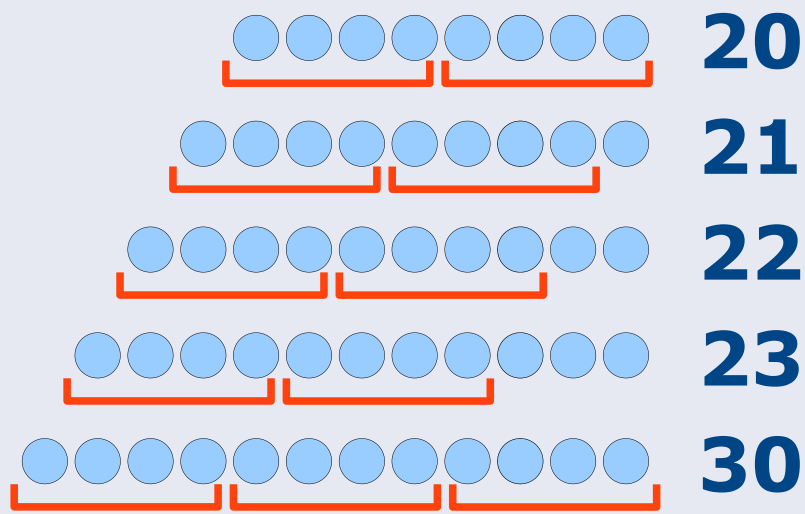
$$\underbrace{0\quad 1\quad 2\quad 3\quad 4\quad 5\quad 6\quad 7\quad 8\quad 9}_{\text{i 10 simboli del sistema decimale (cifre decimali)}}$$

- Pechè usiamo la base 10?

Esempio: contiamo in base 4

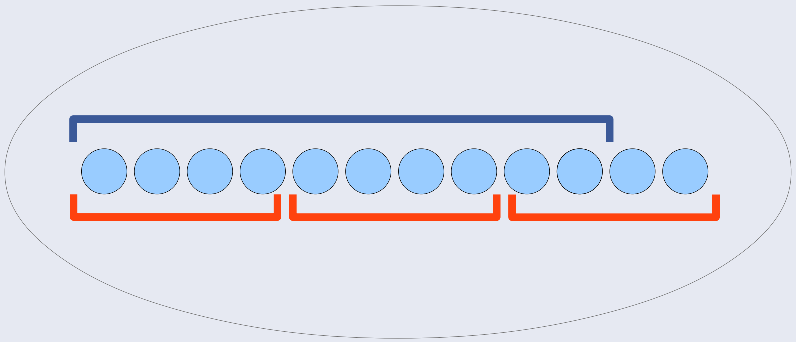
- Se avessimo 2 dita per mano, conteremmo in **base 4** ?
- ..avremmo a disposizione **solo 4 simboli**: **0 1 2 3**





$$12_{(10)} = 30_{(4)}$$

Notazione della base



$$12_{(10)} = 30_{(4)}$$

Attenzione: $12_{(10)}$ e $30_{(4)}$ sono due **modi diversi di rappresentare la stessa quantità** (stesso numero di palline !)

L'indicazione della base è omessa quando la base è 10 oppure se la base è chiara dal contesto

Esercizio

Esprimere tutti i numeri da 0 a 10 nelle basi 3, 4, 5, 6, 7

base 10	base 3	base 4	base 5	base 6	base 7
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	10	3	3	3	3
4	11	10	4	4	4
5	12	11	10	5	5
6	20	12	11	10	6
7	21	13	12	11	10
8	22	20	13	12	11
9	100	21	14	13	12
10	101	22	20	14	13

Sistema di numerazione posizionale in base 10

- Sistema di numerazione **posizionale** = sistema in cui il valore associato a una cifra dipende dalla posizione che essa occupa nella rappresentazione del numero.

Esempio:

$$595 = 5 \times 10^2 + 9 \times 10^1 + 5 \times 10^0$$

posizione 0 (cifra meno significativa)

base

posizione 1

posizione 2 (cifra più significativa)

Numerazione in base 2

- Il sistema di numerazione binario funziona come quello in base 10, ma abbiamo a disposizione due soli simboli!

$$\underbrace{\mathbf{0} \quad \mathbf{1}}$$ cifre binarie (bit)

- Esempio

$$\begin{array}{l} 2^2\ 2^1\ 2^0 \\ 1\ 0\ 1_{(2)} = 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 5_{\mathbf{(10)}} \end{array}$$

Numerazione in base 2

DEC	BIN
0	0
1	1
2	10
3	11
4	100
5	101

DEC	BIN
6	110
7	111
8	1000
9	1001
10	1010
11	1011

DEC	BIN
12	1100
13	1101
14	1110
15	1111
16	10000
...	...

Conversione da base 10 a base N

- Per ottenere la rappresentazione in base N di un numero espresso in base 10 si utilizza un algoritmo basato su **divisioni successive per la base N**.
- Ad esempio, per convertire un numero da base 10 a base 2 occorre eseguire una serie di divisioni per 2 annotando quozienti e resti (vedi esempio)

- Esempio: ottenere la rappresentazione in base 2 del numero $44_{(10)}$.

$$44_{(10)} = \overset{?}{\ldots\ldots\ldots\ldots\ldots}_{(2)}$$

Conversione da base 10 a base 2

$$44_{(10)} = 1\ 0\ 1\ 1\ 0\ 0_{(2)}$$

44	
22	0
11	0
5	1
2	1
1	0
0	1

quozienti **resti**

Il procedimento termina quanto si ottiene 0 nella colonna dei quozienti (e quindi 1 come ultimo resto).

Attenzione a ricopiare i resti nell'ordine giusto: l'ultimo resto corrisponde alla cifra più significativa!!!

Conversioni da sistema binario a decimale

- Per passare dalla rappresentazione in base 2 a quella in base 10, si moltiplica ogni cifra binaria per la potenza di due corrispondente alla posizione della cifra
- Alla cifra meno significativa corrisponde la potenza 2^0, che vale 1.
- Esempio: convertire $101001_{(2)}$ in base 10

$$\begin{array}{cccccc} 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array}_{(2)} = \mathbf{1}\times 2^5+\mathbf{0}\times 2^4+\mathbf{1}\times 2^3+\mathbf{0}\times 2^2+\mathbf{0}\times 2^1+\mathbf{1}\times 2^0$$

$$= 32 + 0 + 8 + 0 + 0 + 1 = \mathbf{41}_{\mathbf{(10)}}$$

Trucchi e scorciatoie in binario..

- I numeri pari finiscono per 0 e i dispari per 1

- Aggiungere uno zero a destra equivale a moltiplicare per 2:

11 = 3   110 = 6   1100 = 12

- Un **1** seguito da N zeri corrisponde a $\mathbf{2^N}$:

1000 = 2^3 10000 = 2^4

- Una sequenza di N **1** consecutivi corrisponde a $\mathbf{2^N-1}$:

111 = 2^3-1 = 7   1111=2^4-1 = 15

Potenze di due

Da sapere!

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$

Unità di misura

1 byte = 8 bit   1 nibble = 4 bit

Kilo	1 KB	=	1024 byte	=	2^{10} byte	circa mille byte
Mega	1 MB	=	1024 KB	=	2^{20} byte	circa 1 milione di byte
Giga	1 GB	=	1024 MB	=	2^{30} byte	circa 1 miliardo di byte
Tera	1 TB	=	1024 GB	=	2^{40} byte	

Esercizio: a quanti KB corrispodono 2^{16} byte ?

Soluzione: $2^{16} = 2^{6} \times 2^{10} = 64$ KB

Range di valori rappresentabili

Range = intervallo

Poichè **N bit** producono **2^N** diverse combinazioni, essi consentono di codificare al più **2^N valori** diversi (numeri, lettere, colori, ...).

- Esempio: configurazioni di 2 bit --> 00 01 10 11
- Con **N bit** è possibile rappresentare, tramite la codifica binaria, tutti i numeri interi senza segno **da 0 a 2^N-1**
- Con **1 byte** (8 bit) è possibile rappresentare tutti i numeri interi **da 0 a 255** (256 valori)
- Con **2 byte** (16 bit) il range si estende **da 0 a 2^{16}-1 = 65535** (2^{16} valori diversi)

Esercizi

Es 1 Convertire i seguenti numeri da base 10 a base 2:

a) 108
b) 250
c) 70
d) 80
e) 95
f) 38
g) 63
h) 126
i) 47
j) 256
k) 2048
l) 513

Es 2 Convertire i seguenti numeri da base 2 a base 10:

m) 1100
n) 10101
o) 00111111
p) 111000
q) 101011
r) 100000000
s) 10001
t) 10111
u) 10000
v) 1111
w) 11111
x) 100000

SOLUZIONI: a) 1101100 b) 11111010 c) 1000110 d) 1010000 e) 1011111 f) 100110
g) 111111 h) 1111110 i) 101111 j) 100000000 k) 100000000000 j) 1000000001
m) 12 n) 21 o) 63 p) 56 q) 43 r) 256 s) 17 t) 23 u) 16 v) 15 w) 31 x) 32

Esercizi risolti

a) $100_{(10)} = ..?.._{(2)}$

Risultato:

$100_{(10)} = 1\ 1\ 0\ 0\ 1\ 0\ 0_{(2)}$

Procedimento:

100	
50	0
25	0
12	1
6	0
3	0
1	1
0	1

100 diviso 2 fa 50 resto 0

50 diviso 2 fa 25 resto 0

25 diviso 2 fa 12 resto 1

mi fermo quando il quoziente è zero

ricopio a partire dal basso (cifra più significativa)

Aritmetica binaria

- Le operazioni in colonna fra numeri binari seguono lo stesso algoritmo già noto per il sistema decimale!
- Per la **somma in colonna** basta ricordare che:
  - **0 + 0 = 0**
  - **1 + 0 = 1**
  - **0 + 1 = 1**
  - **1 + 1 = 0 con riporto di 1**
  - **1 + 1 + 1 = 1 con riporto di 1**

Aritmetica binaria

Esempio

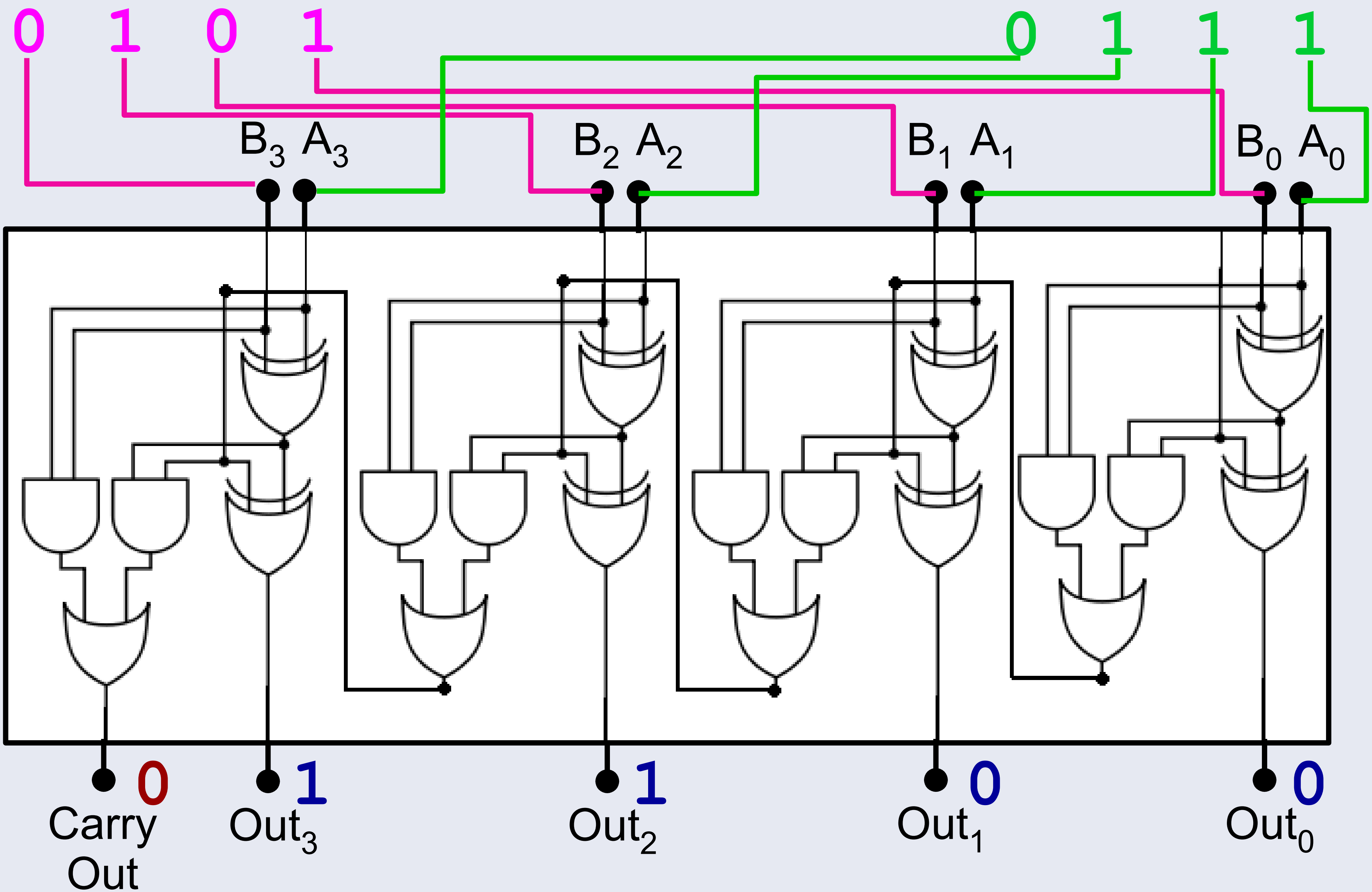
riporti

riporto in uscita (Carry Out)

1	1	0	1	1	1	0	0					
	1	1	0	0	1	1	0	1	+		205	+
	1	1	0	1	1	1	0	0	=		220	=
1	1	0	1	0	1	0	0	1			425	

$$2^8 + 2^7 + 2^5 + 2^3 + 2^0 = 256 + 128 + 32 + 8 + 1 = 425$$

Un circuito sommatore a 4 bit



Codifica BCD

- La **codifica BCD** (**binary-coded decimal**) è un' alternativa alla **codifica binaria** vista in precedenza
- Nella versione *packed* BCD, prevede di rappresentare ogni cifra decimale con 4 bit

Esempio: rappresentare il numero 254 in formato **packed BCD** e in **formato binario**

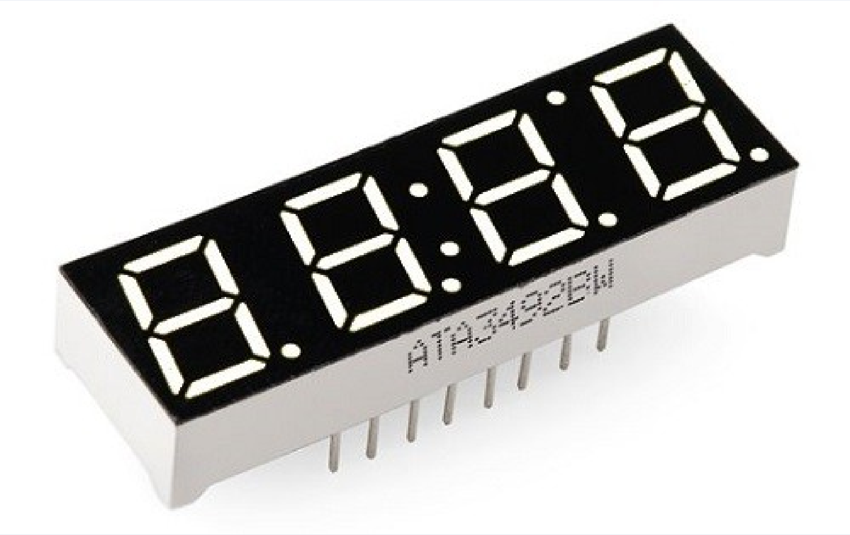
	0 2 5 4	
Codifica BCD	0000 0010 0101 0100	**2 byte**
Codifica binaria	1111 1110	**1 byte**

BCD vs Binary

- Nella codifica BCD solo 10 delle 16 possibili configurazioni di 4 bit sono utilizzate
- A parità di bit utilizzati, quindi, la codifica binaria consente di rappresentare un range più ampio

Codifica	Range
Binaria (2 byte)	**0 … 65535**
BCD (2 byte)	**0 … 9999**

- BCD semplifica la visualizzazione
  dei numeri su display



Esercizi

- **Es 1.** Codifica i seguenti numeri in formato BCD

  a) 96 [1001 0110]

  b) 10 [0001 0000]

  c) 35 [0011 0101]

Il sistema esadecimale (HEX)

- Il sistema di numerazione in base 16 è detto **sistema esadecimale** (*hexadecimal number system*)
- Usa **16 simboli** diversi (cifre esadecimali):

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
										10	11	12	13	14	15

- La cifra **A** rappresenta il numero decimale **10**
- La cifra **B** rappresenta il numero decimale **11**
- La cifra **C** rappresenta il numero decimale **12**
- La cifra **D** rappresenta il numero decimale **13**
- La cifra **E** rappresenta il numero decimale **14**
- La cifra **F** rappresenta il numero decimale **15**

Conversione da Hex a Dec

- Per passare da esadecimale a binario, si moltiplica il valore di ogni cifra per la potenza di 16 corrispondete

Esempio

$$A37E_{(h)} = \underset{?}{\ldots\ldots\ldots\ldots}_{(10)}$$

$$14 \times 16^0 +$$
$$7 \times 16^1 +$$
$$3 \times 16^2 +$$
$$10 \times 16^3 =$$
$$41854$$

A	↔ 10
B	↔ 11
C	↔ 12
D	↔ 13
E	↔ 14
F	↔ 15

Decimale – Binario – Esadecimale

Dec	Bin	Hex
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Dec	Bin	Hex
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1A
27	11011	1B
28	11100	1C
29	11101	1D
30	11110	1E
31	11111	1F

Dec	Bin	Hex
32	100000	20
33	100001	21
..	...	...
48	110000	30
...	...	...
64	1000000	40
65	1000001	41
66	1000010	42
...	...	...
126	1111110	7E
127	1111111	7F
128	10000000	80
...	...	...
253	11111101	FD
254	11111110	FE
255	11111111	FF

Notazioni

- Per indicare che un numero è espresso in base 16 si fa seguire il numero dal **suffisso h**
- In molti linguaggi di programmazione (C/C++, Java, ..) è possibile far precedere il numero dal **prefisso 0x**

  Esempio:

  il numero $\mathbf{42_{(16)}}$ si può indicare come **42h** o **0x42**

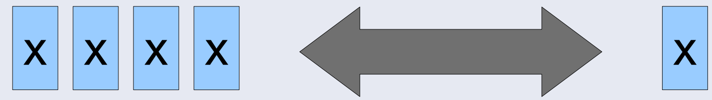
- Nota: 42h è diverso da 42 !!!

  42h = 4*16 + 2 = 64 + 2 = 66

```
int main() {
  int n;
  n = 0x42;
}
```

Perchè gli informatici usano l'HEX ?

BIN ↔ HEX
0000 ↔ 0
0001 ↔ 1
0010 ↔ 2
0011 ↔ 4
0100 ↔ 4
0101 ↔ 5
0110 ↔ 6
0111 ↔ 7
1000 ↔ 8
1001 ↔ 9
1010 ↔ A
1011 ↔ B
1100 ↔ C
1101 ↔ D
1110 ↔ E
1111 ↔ F



4 bit
(nibble)

1 cifra
esadecimale

- Ad ogni possibile configurazione di 4 bit corrisponde esattamente 1 cifra esadecimale
- Con due cifre esadecimali si rappresenta in modo compatto il contenutodi un byte di memoria:

  01101011 ↔ 6B

Immagini della memoria

L'esadecimale è spesso usato per rappresentare il contenuto di una memoria (**memory dump**)

cella di memoria di un byte, contenente il valore 8B

8B	44	24	04
FC	$\boxed{8B}$	48	14
8B	40	18	C2
24	04	B8	6C
68	20	72	47
50	50	E8	D3
04	00	00	00
8D	3D	24	32
C2	00	00	00
75	0C	66	83
3D	E4	31	5C
F5	E9	95	01
66	A7	75	0C
F8	8D	3D	A4

Il contenuto di questo byte è

$$\mathbf{E8} = \underbrace{\mathbf{1110}}_{\mathbf{E}}\,\underbrace{\mathbf{1000}}_{\mathbf{8}}$$

Perchè gli informatici usano l'HEX?

- la notazione esadecimale è più **compatta** di quella decimale e binaria (servono meno cifre per rappresentare lo stesso numero)
- usando una sola cifra esadecimale si rappresentano tutte le 16 possibili configurazioni di 4 bit (mezzo byte), ovvero i valori decimali da 0 a 15
- usando solo due cifre esadecimali si rappresentano tutte le 256 possibili configurazioni di 8 bit (un byte) ovvero i valori da 0 a 255
- poichè 16 è una potenza di 2, le conversioni da base 16 a base 2 e viceversa sono estremamente semplici

Aritmetica in esadicmale

- L'addizione segue lo stesso algoritmo già noto nel sistema decimale!

  Esempi:

- 9 + 1 = A
- A + 1 = B
- A + 2 = C
- C + 3 = F
- E + 5 = 13h

Aritmetica esadecimale

Esempio

```
riporti
 0 0 1 0
   A 0 2 B +
   1 C F 3 =
 ___________
   B D 1 E
```

Osservazioni

- Aggiungere uno zero alla destra di un numero esadecimale equivale a moltiplicare per $16_{(10)}$

  Esempi: Ah = 10 A0h = 10*16 A00h = 10*16*16

- I numeri che terminano per zero sono quindi multipli di 16

Conversioni BIN ↔ HEX

- Per convertire da binario a esadecimale si raggruppano le cifre binarie a gruppi di 4 **<u>partendo da destra</u>** e si sostituisce ogni gruppo con la cifra esadecimale corrispondente

$$\underbrace{1\,1}_{3}\;\underbrace{0\,0\,1\,0}_{2}\;\underbrace{1\,1\,1\,0}_{E}\,{}_{(2)} \quad = \quad 3\,2\,E\,{}_{(h)}$$

- Per convertire da hex a bin si sostiuisce ogni cifra esadecimale con la sequenza di 4 bit corrispondente.

Esercizi

Es 1 Convertire i seguenti numeri da base 2 a base 16:

```
a)  10110111
b)  11011001
c) 111111111
d)    101010
e) 101110101
f)  10000110
```

Es 2 Convertire i seguenti numeri da base 16 a base 2

```
g)  A3
h)  FA
i)  BE
j) ABC
k) D1C
l) FFE
```

Es 3 Convertire i seguenti numeri da base 16 a base 10

```
m)  A3
n)  FA
o)  BE
p) ABC
q) D1C
r) FFE
```

Es 5 Eseguire le seguenti somme

```
s) A8h + 32h
t) E1h + 20h
u) F3h + 17h
v) FFFFh + 1
w) A01Bh + F4h
x) 55h + 4Ch
```

Soluzioni:
a)B7 b)D9 c)1FF d)2A e)175 f)86
g)1010 0011 h)1111 1010 i)1011 1110
j)1010 1011 1100 k)1101 0001 1100
l)1111 1111 1110 m)163 n)250 o)190
p)2748 q)3356 r)4094 s) DAh t) 101h
u)10Ah v) 10000h w) A10Fh x) A1h

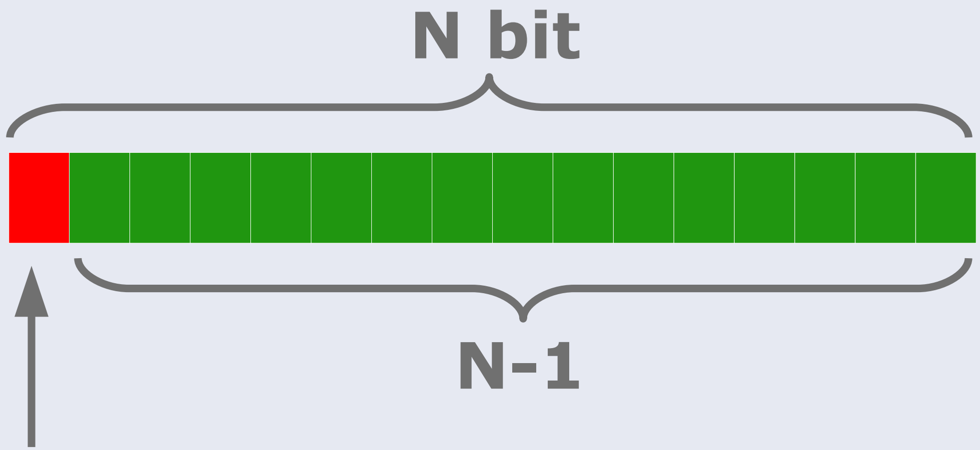
Codifica dei numeri interi con segno

Problema: dati N bit, come possiamo utilizzarli per codificare un insieme di **numeri interi con segno** (negativi, positivi, zero) ?

diverse soluzioni possibili

- codifica **modulo e segno**
- codifiche **eccesso-k**
- codifica in **complemento a uno**
- codifica in **complemento a due**

Rappresentazione modulo e segno



bit di segno:

- **0** se il numero è **positivo**
- **1** se il numero è **negativo**

Dati N bit, si utilizza il bit più significativo per rappresentare il **segno** e gli N-1 bit rimanenti per rappresentare il **valore assoluto** (***modulo***) del numero. Per convenzione, il bit di segno è 1 se il numero è negativo.

Rappresentazione modulo e segno

- **Esempio**: rappresentare il numero **-5** in modulo e segno su 8 bit

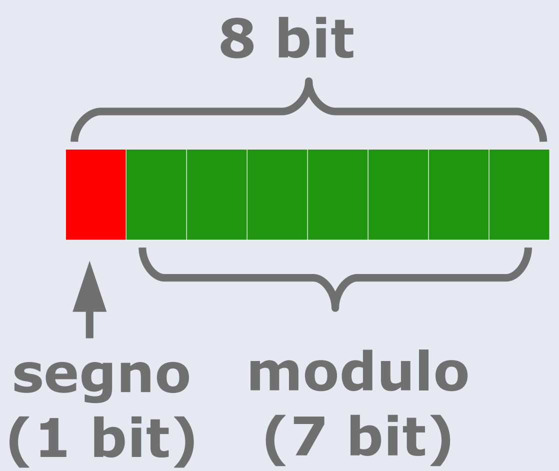
$$\overbrace{\begin{array}{|c|c|c|c|c|c|c|c|}\hline 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline\end{array}}^{\textbf{8 bit}}$$

- Interpretando la stessa sequenza (10000101) come un intero senza segno otterremmo un numero completamente diverso: $1 \times 2^7 + 1 \times 2^2 + 1 =$ **133** !!!
- Una sequenza di bit, di per sè, è solo una sequenza di bit! Conoscendo la codifica utilizzata possiamo interpretarla come un numero.

Modulo e segno: valori rappresentabili

- In modulo e segno su N bit l'intervallo di numeri rappresentabili si estende da $-(2^{N-1} - 1)$ a $+(2^{N-1} - 1)$

Esempio (N=8)



val. minimo	**11111111**	**(-127)**
val. massimo	**01111111**	**(+127)**
zero "negativo"	**10000000**	**(-0)**
zero "positivo"	**00000000**	**(+0)**

problema della doppia rappresentazione dello zero

Doppia rappresentazione dello zero

- Nella codifica modulo e segno, lo stesso numero (zero) è codificato due volte:
  - **10000000** (-0) zero negativo
  - **00000000** (+0) zero positivo

- La doppia rappresentazione dello zero è un problema perchè:

  - occorre tenere conto di questa "anomalia" nei **confronti** (poichè matematicamente +0 = -0)

  - **si spreca una configurazione** (che potrebbe essere usata per estendere di una unità il range dei valori rappresentabili)

Codifiche eccesso-K

- Risolve il problema della doppia rappresentazione dello zero
- Per codificare un numero X con una codifica eccesso-K si somma un **valore costante (K)** al numero da rappresentare (X) e si codifica poi in binario il risultato della somma (X+K).
- La costante K è scelta in modo che il risultato della somma (X+K) sia sempre positivo (mentre X potrebbe essere negativo)
- In questo modo non serve alcun bit "speciale" per rappresentare il segno!
- Se la codifica è su N bit, si utilizza in genere $\mathbf{K=2^{N-1}}$ oppure $\mathbf{K=2^{N-1}-1}$

Codifiche eccesso-K

Procedimento di codifica:

- numero da codificare: X
- calcolo X + K
- codifico (X + K) in binario

Idea: anzichè codificare direttamente X (che potrebbe essere negativo) si somma ad X una costante K, scelta in modo che la somma (X+K) sia sempre positiva, e si codifica in binario il risultato della somma.

Esempio: Codificare il numero **-3** usando la la codifica "**eccesso-127"** a 8 bit

Soluzione X=-3 K=127

-3 + 127 = 124

124 = 01111100

rappresentazione di **-3** in eccesso-127

Esempio: codifica eccesso-127

- Con la codifica "eccesso-127" a 8 bit l'intervallo di valori rapprentabili è: -127 ... +128

Infatti, con **K=127** si ottiene

- Valore minimo: **-127**

  codifica: **-127** + **127** = **0** → **00000000**

- Valore massimo: **+128**

  codifica: **+128** + **127** = **255** → **11111111**

- Unica rappresentazione dello **zero**:

  0 + **127** = **127** → **01111111**

Notare che lo zero "eccesso-127" non coincide con lo zero della codifica binaria senza segno... 00000000

Codifica in complemento a due

- E' il formato adottato internamente dalla maggior parte delle CPU attuali (ALU progettata per operare su numeri in complemento a due)
- Consente di realizzare circuiti hardware particolarmente semplici (unico circuito "sommatore" per somma e sottrazione)
- Unica rappresentazione dello zero: 00000000

Complemento a due

Fissato un certo numero N di bit, la rappresentazione in complemento a due di un numero si ottiene in questo modo:

- se il numero è **positivo**, la codifica in complemento a due coincide con codifica binaria del numero stesso
- se il numero è **negativo**, per ottenere la sua codifica in complemento occorre eseguire tre passaggi:

passo 1] codificare in binario il **valore assoluto** del numero

passo 2] **invertire** tutti i bit (gli 1 diventano 0 e viceversa)

passo 3] **sommare 1** al numero ottenuto al punto precedente, ignorando (scartando) l'eventuale riporto oltre il bit più significativo

Complemento a due

Esempio 1 Trova la rappresentazione in complemento a due a 8 bit del numero $\mathbf{9}_{(10)}$

Soluzione Poichè il numero è positivo, basta calcolare la rappresentazione binaria di 9 e disporla su 8 bit:

$$\mathbf{9}_{(10)} = \mathbf{00001001}_{(c2)}$$

Complemento a due

Esempio 2 Trova la rappresentazione in complemento a due a 8 bit del numero **$-9_{(10)}$**

Soluzione Poichè il numero è negativo servono 3 passi

Passo 1	Rappresento +9 in binario su 8 bit	**00001001**
Passo 2	Inverto tutti i bit (*complemento a uno*)	**11110110 +**
		1 =
Passo 3	Sommo +1 al risultato del passo precedente	**11110111**

rappresentazione di -9

Operazioni di complemento a uno e a due

- data una sequenza di bit, l'operazione di inversione dei bit è detta anche **operazione di complemento a uno**

  esempio: il complemento a uno di **0101** è **1010**

- data una sequenza di bit, l'operazione di inversione dei bit e aggiunta di 1 (scartando l'ultimo riporto se presente) è **detta operazione di complemento a due**

  esempio: il complemento a due di 00001111 è 11110001

- Nota bene: le operazioni di complemento a uno e complemento a due partono da una sequenza di N bit e producono una nuova sequenza di esattamente N bit

Operazioni di complemento a uno e a due

- data una sequenza di bit, l'operazione di inversione dei bit è detta anche **operazione di complemento a uno**

  esempio: il complemento a uno di **0101** è **1010**

- data una sequenza di bit, l'operazione di inversione dei bit e aggiunta di 1 (scartando l'ultimo riporto se presente) è **detta operazione di complemento a due**

  esempio: il complemento a due di **00001111** è
  11110000 + **1** = **11110001**

- Nota bene: le operazioni di complemento a uno e complemento a due partono da una sequenza di N bit e producono una nuova sequenza di esattamente N bit

Proprietà del complemento a due

- Tutti i numeri **positivi** hanno il bit più significativo a **0**
- Tutti i numeri **negativi** hanno il bit più significativo a **1**
- Il bit più significativo (**MSB** = **most significant bit**) funziona quindi come un **bit di segno**
- Effettuare l'operazione di complemento a due su una sequenza di bit equivale a cambiare di segno il numero rappresentato

complemento a due = cambio di segno

Cambio di segno

Verifichiamo che ogni volta che si esegue il complemento a due su una sequenza di bit si cambia il segno del numero rappresentato:

	+9	**-9**	**0**
Sequenza originale	00001001	11110111	00000000
Complemento a uno	11110110	00001000	11111111
Aggiungi 1	11110110 + 1	00001000 + 1	11111111 + 1
Complemento a due	11110111	00001001	1 00000000
	-9	**+9**	**0**

riporto scartato

Esercizio risolto

Esercizio Quale numero è rappresentato in complemento a due dalla sequenza **11110000** ? (esprimi il risultato in base 10)

Soluzione

poichè il bit più a sinistra (MSB) è un 1, il numero sarà sicuramente negativo

per ottenere il suo valore assoluto, gli cambio segno eseguendo l'operazione di complemento a due:

11110000 → 00001111 +1 = 00010000

$00010000_{(2)} = 16_{(10)}$

quindi la sequenza originale rappresenta $\mathbf{-16_{(10)}}$

Aritmetica in complemento a due

3	00000011
+ 2	+ 00000010
5	00000101

(-3)	11111101
+ (-2)	+ 11111110
(-5)	1] 11111011

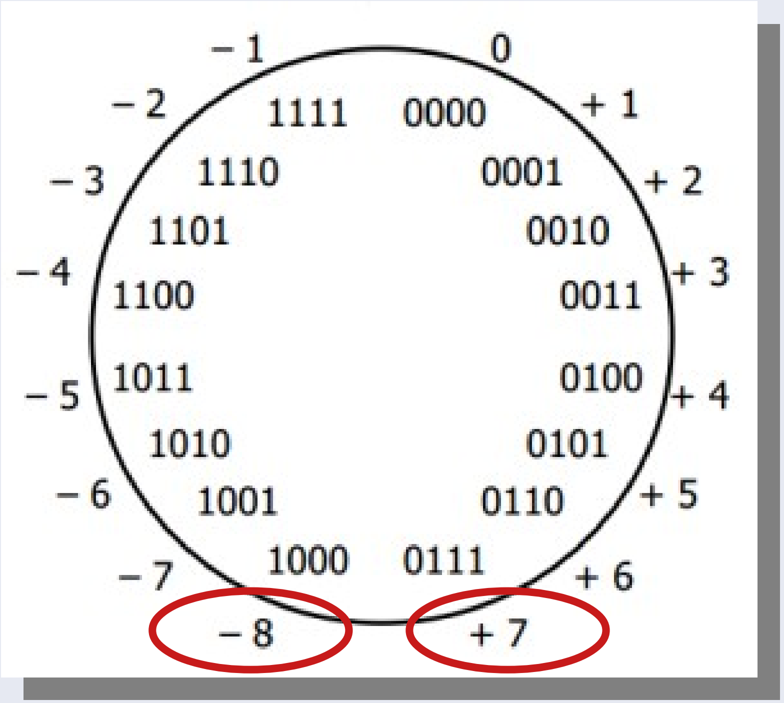
(-5)	11111011
+ 2	+ 00000010
-3	11111101

5	00000101
+ (-2)	+ 11111110
3	00000011

Funziona!

Range di valori rappresentabili

- La codifica in complemento a due a **N** bit consente di rappresentare tutti i numeri interi da -2^{N-1} a $+2^{N-1} - 1$



Esempio:

Qual è il range dei numeri rappresentabili in complemento a due con 4 bit ? e con 8 bit?

Risposta:

4 bit --> [-2^3 … $+2^3-1$] --> [-8 … +7]

8 bit --> [-2^7 … $+2^7-1$] --> [-128 … +127]

Un caso speciale

- In complemento a due, il **numero negativo con valore assoluto massimo** è sempre rappresentato da una sequenza del tipo **1000...000**
- Esempio: con **8 bit** il range è **[-128 ... +127]** e il valore -128 si rappresenta con 1000000
- Questa sequenza è particolare perchè il suo complemento a due è la sequenza stessa!

Sequenza originale	1000000
Comeplemento a uno	0111111
Aggiungo 1	+1
Complememento a due	1000000

Overflow in complemento a due

- Può capitare che il risultato di un'operazione aritmetica tra due numeri a N bit non sia rappresentabile con N bit

  si dice che l'operazione genera **overflow** (*trabbocamento*)

Esempio 1: utilizzando la codifica in complemento a due a 8 bit, l'operazione 120 + 10 produce overflow poichè il risultato (130) eccede il massimo numero rappresentabile (127)

Esempio 2: utilizzando la codifica in complemento a due a 8 bit, l'operazione -80 -60 produce overflow poichè il risultato (-140) è inferiore al minimo numero rappresentabile (-128)

Condizioni di overflow

- E' possibile stabilire se si è verificato overflow osservando direttamente le codifiche in complemento a due, in base ad uno qualsiasi dei seguenti criteri:

Criterio 1. l'operazione di somma in complemento a due produce overflow _se e solo se_ i due addendi hanno segno uguale e il risultato ha segno diverso.

Criterio 2. l'operazione di somma in complemento a due produce overflow _se e solo se_ l'ultimo e il penultimo riporto sono diversi.

Applicazione dei criteri di overflow

Esempio 1: eseguire la somma tra i seguenti numeri in complemento a due a 8 bit e stabilire se si è verificato overflow: 00000110 + 00001100

Soluzione:

		0 0 0 0 1 1 0 0
6		0 0 0 0 0 1 1 0
+ 12	+	0 0 0 0 1 1 0 0
18		0 0 0 1 0 0 1 0

gli ultimi due riporti sono uguali quindi non c'è overflow (criterio 2)

oppure

i numeri sono entrambi positivi e il risultato è positivo, quindi non c'è overflow (criterio 1)

Applicazione dei criteri di overflow

Esempio 2: eseguire la somma tra i seguenti numeri in complemento a due a 8 bit e stabilire se si è verificato overflow: 11111101 + 11111110

Soluzione:

		1 1 1 1 1 1 0 0
(-3)		1 1 1 1 1 1 0 1
+(-2)	+	1 1 1 1 1 1 1 0
-5		1 1 1 1 1 0 1 1

gli ultimi due riporti sono uguali quindi non c'è overflow (criterio 2)

oppure

gli addendi hanno lo stesso segno e il risultato ha lo stesso segno, quindi non c'è overflow (criterio 1)

Applicazione dei criteri di overflow

Esempio 3: eseguire la somma tra i seguenti numeri in complemento a due a 8 bit e stabilire se si è verificato overflow: 01111110 + 00000011

Soluzione:

		0 1 1 1 1 1 1 0
	126	0 1 1 1 1 1 1 0
+	3	+ 0 0 0 0 0 0 1 1
	129	1 0 0 0 0 0 0 1
		=
		-127

risultato matematico errato

gli **ultimi due riporti sono diversi** quindi c'è overflow (criterio 2)

oppure

gli addendi hanno lo stesso segno e il **risultato ha segno diverso** quindi c'è overflow (criterio 1)

Applicazione dei criteri di overflow

Esempio 4: eseguire la somma tra i seguenti numeri in complemento a due a 8 bit e stabilire se si è verificato overflow: 10110000 + 10110000

Soluzione:

gli **ultimi due riporti sono diversi** quindi c'è overflow (criterio 2)

		10110000
(-80)		10110000
+(-80)	+	10110000
-160		01100000
		=
		+96

risultato matematico errato

oppure

gli addendi hanno lo stesso segno e il **risultato ha segno diverso** quindi c'è overflow (criterio 1)

Provare per credere...

Linguaggio C

```c
# include <stdio.h>

int main() {
   char x,y,z;
   x = -80;
   y = -80;
   z = x + y ;
   printf("%d", z);
}
```

```
C:\Users\Me\Desktop\test.exe
96
--------------------------------
Process exited after 0.03446 seconds with retur
n value 0
Premere un tasto per continuare . . .
```

```cpp
# include <iostream>
using namespace std;

int main() {
   char x,y,z;
   x = -80;
   y = -80;
   z = x + y ;
   cout << (int) z;
}
```

Linguaggio C++

Confronto fra codifiche degli interi con segno

	Rappresentazione modulo e segno (N bit)	Rappresentazione in complemento a due (N bit)
Range di valori rappresentabili	$-(2^{N-1}-1)$ .. $2^{N-1}-1$	-2^{N-1} .. $2^{N-1}-1$
Numero di interi diversi	2^N-1	2^N
Codifica dello zero	Doppia rappresentazione (per verificare l'uguaglianza con zero servono due confronti)	Unica rappresentazione dello zero (non esistono lo zero positivo e quello negativo)
Indicazione del segno	Il bit più significativo indica il segno: 1 -> negativo 0 -> positivo	Il bit più significativo indica il segno: 1 -> negativo 0 -> positivo
Vantaggi e svantaggi	Operazioni aritmetiche e di confronto più complesse da implementare in hardware	Operazioni aritmetiche più semplici da implementare in hardware (basta un unico circuito di somma binaria)
Utilizzo	Non utilizzata nelle CPU attuali	Utilizzata nella maggior parte delle CPU attuali

Esercizi

Es 1 Rappresentare in complemento a due a 8 bit i seguenti numeri

```
a) -13
b) -18
c) +20
d) -20
e) +50
f) -100
```

Es 3 stabilire quale numero in base 10 è rappresentato dalle seguenti sequenze in complemento a due

```
m) 11110101
n) 10000000
o) 11111100
p) 11000000
q) 10110110
r) 10111111
```

Es 2 Eseguire le somme usando l'aritmetica in complemento a due a 8 bit, e stabilire se producono overflow

```
g) 00001111 + 00001100
h) 11000100 + 11001110
i) 11000100 + 10111010
j) 01010001 + 00010101
k) 01011010 + 01101110
l) 01111100 + 00000101
```

Soluzioni:

a)11110011 b) 11101110 c)00010100
d)11101100 e)00110010 f)10011100

g) no h) no i) si j) no k) si l) si

m)-11 n) -128 o)-4 p)-64 q)-74 r)-65

http://www.exploringbinary.com/twos-complement-converter/

Numeri frazionari nel sistema binario

- Sappiamo che nei numeri in base 10 le cifre dopo la virgola sono associate a potenze di 10 con esponente negativo:

$$675{,}93_{(10)} = 6\times10^{2}+7\times10^{1}+5\times10^{0}+9\times10^{-1}+3\times10^{-2}$$

- Nel sistema binario la parte frazionaria si interpreta allo stesso modo, utilizzando però potenze di 2:

$$101{,}011_{(2)}=1\times2^{2}+0\times2^{1}+1\times2^{0}+0\times2^{-1}+1\times2^{-2}+1\times2^{-3}$$

$$= 1\times4 + 0\times2 + 1\times1 + 0\times(1/2) + 1\times(1/4) + 1\times(1/8)$$

$$= 4 + 1 + 0{,}25 + 0{,}125 = \mathbf{5{,}375}_{(10)}$$

Numeri frazionari: conversione DEC → BIN

Problema Trovare la rappresentazione binaria del numero $3{,}875_{(10)}$

Soluzione - Per la **parte intera** si utilizza il metodo già noto

$$3_{(10)} = 11_{(2)}$$

- Per la **parte frazionaria** si utilizza l'**algoritmo delle moltiplicazioni per due**

0,875	× 2 =	**1**,75
0,75	× 2 =	**1**,5
0,5	× 2 =	**1**,0
0,0	× 2 =	**0**,0
0,0	× 2 =	**0**,0
...		...

$$3{,}875_{(10)} = 11{,}111000.._{(2)}$$

Numeri frazionari: conversione DEC → BIN

Problema 2 Trovare la rappresentazione binaria del numero $0{,}3_{(10)}$

Soluzione - **parte intera** $0_{(10)} = 0_{(2)}$

- **parte frazionaria**

0,3	x 2 =	**0**,6
0,6	x 2 =	**1**,2
0,2	x 2 =	**0**,4
0,4	x 2 =	**0**,8
0,8	x 2 =	**1**,6
0,6	x 2 =	**1**,2
...		...

da qui in poi si ripete all'infinito la parte cerchiata

$0{,}3_{(10)} = 0{,}0\ 1001\ 1001\ 1001\ 1001\ ..\ _{(2)}$

Osservazione importante

- L'esempio precedente ci mostra che un numero con un numero finito di cifre dopo la virgola nel sistema decimale, può avere infinite cifre dopo la virgola nella rappresentazione binaria!
- Ciò significa che, per rappresentarlo nella memoria di un computer con un numero finito di bit, sarà necessario **troncare** la parte frazionaria !
- il troncamento produce un numero che **approssima** il numero originale, ma può essere diverso dall'originale

Esercizi

- **Es 1. Converti in binario i segenti numeri frazionari espressi in base 10**

  a) 3,125

  b) 10,625

  c) 12,1

- **Es 2. Converti in decimale i seguenti numeri frazionari espressi in base 2**

  d) 11,100110

  e) 101,001101

  f) 1110,0111

Soluzioni:
a) 11.001
b)1010,101
c)1100,00011001100110011...
d)3,59375
e) ...
f) ....

Virgola fissa e virgola mobile

Per codificare nella memoria di un computer un numero binario frazionario sono possibili due approcci:

- rappresentazione in **virgola fissa (*fixed-point*)**
- rappresentazione in **virgola mobile (*floating-point*)**

Virgola fissa

- Si riserva un certo numero di bit **costante** per la parte intera, e un altro numero di bit costante per la parte frazionaria.

Esempio Rappresentare in virgola fissa il numero 9,5 in virgola fissa, usando 1 bit per il segno, 15 bit per la parte intera e 16 bit per la parte frazionaria

Soluzione

$$9{,}5_{(10)} = 1001{,}1_{(2)}$$

0 000000000001001 1000000000000000

Virgola mobile

- la rappresentazione in virgola mobile consente un uso più efficiente dei bit a disposizione
- a parità di bit complessivi, permette un range di numeri molto più ampio
- Si basa sulla rappresentazione esponenziale dei numeri frazionari:

Esempi (in sistema decimale)

$$+412,36 = +4,1236 \times 10^{2}$$

$$-0,00985 = -9,85 \times 10^{-3}$$

Lo standard IEEE-754

- la rappresentazione in virgola mobile definita dallo standard **IEEE-754** è adottata dalla maggior parte delle attuali CPU
- Lo standard prevede due codifiche (o formati):
  - ***IEEE-754 single precision*** (precisione singola)
    - codifica a 32 bit
    - usata per le variabili di tipo **float** in molti linguaggi
  - ***IEEE-754 double precision*** (precisione doppia)
    - codifica a 64 bit
    - usata per le variabili di tipo **double** in molti linguaggi

Codifica IEEE-754

Le codifica in virgola mobile IEEE-754 si basa sulla rappresentazione di numero binario in forma esponenziale normalizzata:

Passo 1 Si rappresenta il numero frazionario in binario

Passo 2 Si **normalizza** il numero, cioè lo si scrive nella forma

$$\underbrace{\pm}_{\text{segno}}\ 1,\underbrace{XXXXX}_{\text{mantissa}} \times 2^{\overbrace{\pm N}^{}}$$

esp.

per qualsiasi numero diverso da 0 è sempre possibile ottenere questa forma spostando la virgola verso destra o verso sinistra di N posizioni:

spostamento verso **destra** ⇒ esponente = **-N**
spostamento verso **sinistra** ⇒ esponente = **+N**

Codifica IEEE-754

Passo 3 Si codifica l'esponente con **eccesso-127** (8 bit) se si utilizza la precisone singola, oppure con **eccesso-1023** (11 bit) se si utilizza la precisione doppia

Passo 4 Si sistemano i bit per segno, mantissa ed esponente, a seconda del formato scelto, secondo questi schemi:

Precisione singola (totale 32 bit):

1 bit per segno (1=negativo, 0=positivo)
8 bit per esponente
23 bit per mantissa

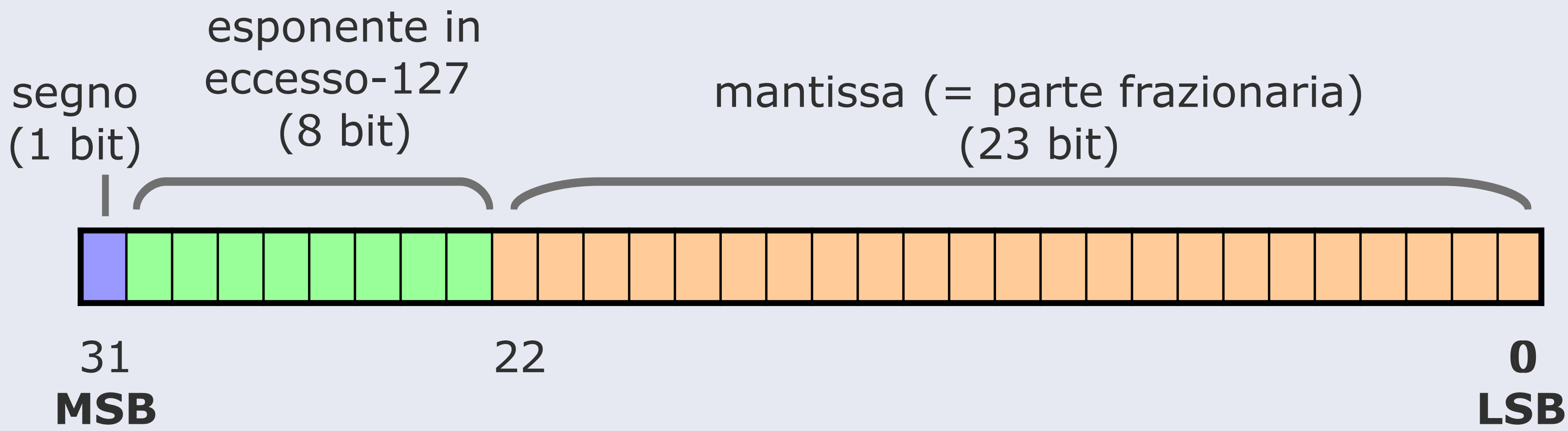
Precisione doppia (totale 64 bit):

1 bit per segno (1=negativo, 0=positivo)
11 bit per esponente
52 bit per mantissa

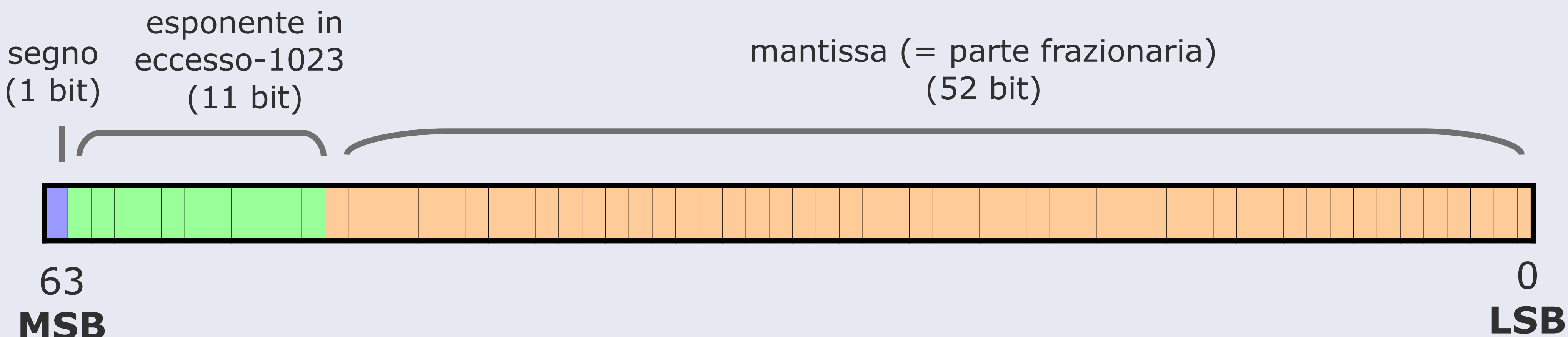
Formati IEEE-754

$\pm\ 1{,}XXXXX \times 2^{\pm N}$

Precisione singola = 32 bit



Precisione doppia = 64 bit



Codifica IEEE-754 – Esempio 1

Esempio Codificare **-12,25** in formato *IEEE-754 single precision*

$$0{,}25 = 0\times 2^{-1} + 1\times 2^{-2} = 0\times 1/2 + 1\times 1/4$$

Passo 1
Codifica binaria

$$-12{,}25_{(10)} = -\,1100{,}01_{(2)}$$

Passo 2
Normalizzazione

$$-\,1100{,}01 = -1{,}\underbrace{100}01\times 2^{+3}$$

virgola spostata di 3 posizioni verso sinistra, quindi esponente positivo +3

Codifica IEEE-754 – Esempio 1

Passo 3

Codifico l'esponente in eccesso-127

$$\underbrace{-}_{\text{segno}}1,\underbrace{10001}_{\text{mantissa}}\times 2^{\overbrace{3}^{\text{esp.}}}$$

3 + 127 = 130 -> 10000010

Passo 4

Posiziono i bit nel formato

$$\underbrace{1}_{\text{segno}}\ \underbrace{10000010}_{\substack{\text{esp + 127}\\ \text{(8 bit)}}}\ \underbrace{10001000000000000000000}_{\substack{\text{mantissa}\\ \text{(23 bit)}}}$$

	1100	0001	0100	0100	0000	0000	0000	0000
HEX	**C**	**1**	**4**	**4**	**0**	**0**	**0**	**0**

Valori speciali e range IEEE-754

- Le sequenze 100...000 e 000..000 rappresentano i valori -0 e +0
- L'esponente 111...111 è riservato per codificare valori speciali (**infinity** e **NaN = Not a Number**)
- L'esponente 000...000 è riservato per i **numeri denormalizzati**
- **Range precisione singola**:

$$\pm 1.18 \times 10^{-38} \;..\; \pm 3.4 \times 10^{38}$$

- **Range precisione doppia**:

$$\pm 2.23 \times 10^{-308} \;...\; \pm 1.80 \times 10^{308}$$

Esercizi

Es 1 **Codifica i seguenti numeri nel formato IEEE-754 precisione singola, rappresentando il risultato in esadecimale**

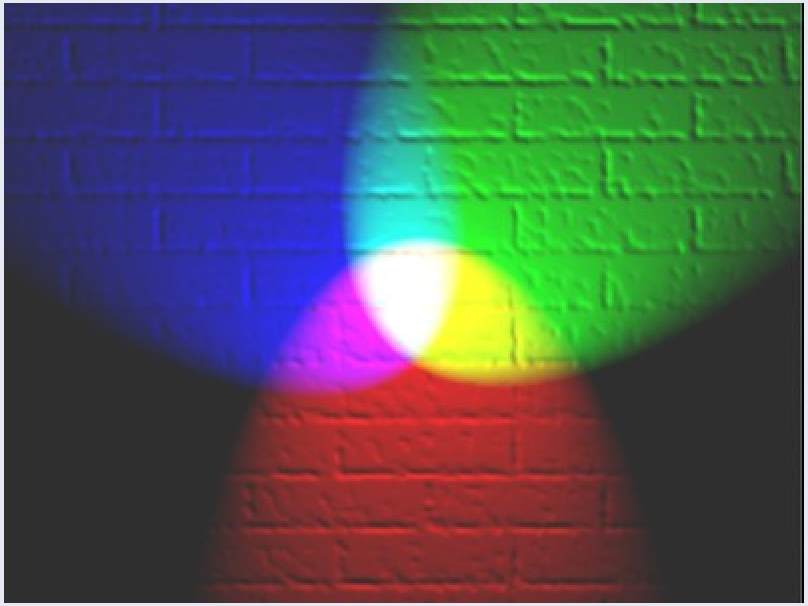
a) 4.25
b) 12.6
c) 0.5
d) 1.0
e) 1.5
f) -4.5
g) -80.2
h) 76.3
i) 8.625
j) 10.1
k) 1024.75
l) 40.3

SOLUZIONI:
a) 40 88 00 00
b) 41 49 99 9A
c) 3F 00 00 00
d) 3F 80 00 00
e) 3F C0 00 00
f) C0 90 00 00
g) C2 A0 66 66
h) 42 98 99 9A
i) 41 0A 00 00
j) 41 21 99 9A
k) 44 80 18 00
l) 42 21 33 33

Nota: per verificare gli esercizi è possibile utilizzare il seguente link:
http://www.h-schmidt.net/FloatConverter/IEEE754.html

Codifica dei colori nel formato RGB

- Si può pensare di produrre qualsiasi colore visibile sullo schermo combinando la luce di 3 sorgenti luminose: una di colore rosso, una verde e una blu.



- Variando l'intensità relativa delle 3 luci si ottengono diversi colori
- Se tutte le luci sono spente, si ottiene il nero
- Se tutte le luci sono accese alla massima intensità, si otterrà il bianco

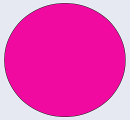
Codifica dei colori nel formato RGB

- Nella codifica RGB un colore si rappresenta tramite **3 valori** (R, G e B) che indicano i livelli di intensità della luce rossa, verde e blu necessari a generarlo.
- I valori R, G e B costituiscno le **tre componenti** del colore.
- Nella **codifica RGB a 24 bit** ogni colore è rappresentato tramite 3 byte (un byte per ogni componente):
  - il primo byte specifica la quantità (livello) di luce rossa (R)
  - il secondo byte specifica la quantità (livello) di luce verde (G)
  - il terzo byte specifica la quantità (livello) di luce blu (B)
- Il valore di ogni componente può variare da **0** (intensità minima) a **255** (intensità massima).

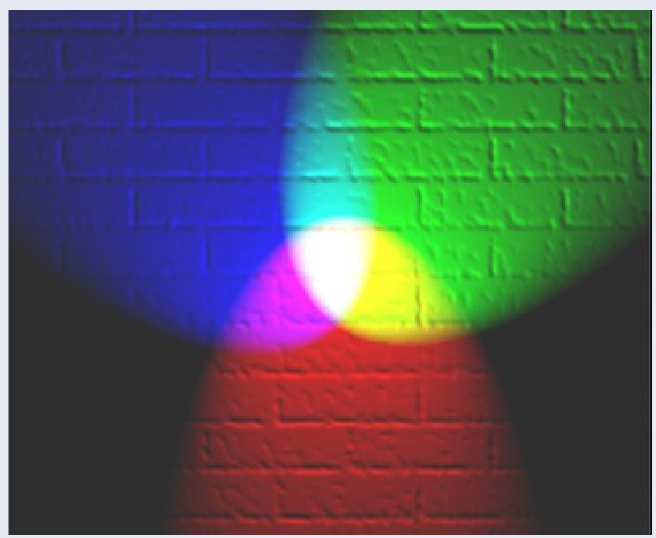
Formato RGB a 24 bit

Esempio di colore in formato RGB a 24 bit:

11110000	00001010	10100000
Red=240	**Green**=10	**Blue**=160
livello di rosso	livello di verde	livello di blu



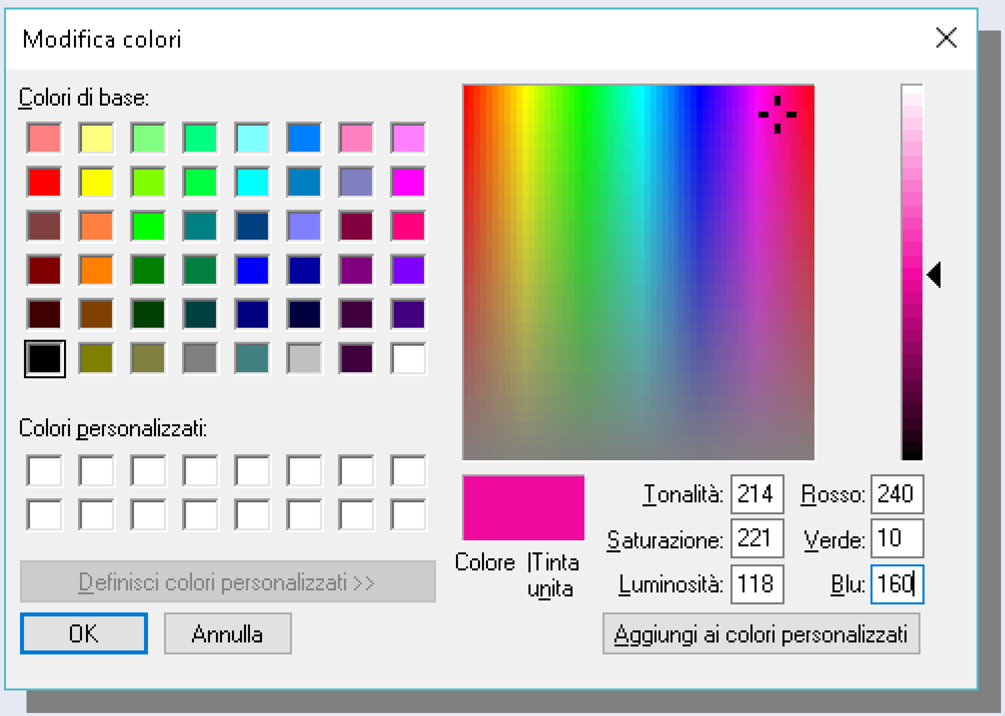
colore risultante



Codifica dei colori nel formato RGB

- I valori R, G e B vengono solitamente espressi in decimale o in esadecimale.
- Il colore precedente si indica quindi così:

rgb(240 , 10, 160) oppure **#F00AA0**



Altri esempi:

00FF00 = verde

FFFF00 = giallo

FF00FF = violetto

000000 = nero

FFFFFF = bianco

Numero di colori rappresentabili

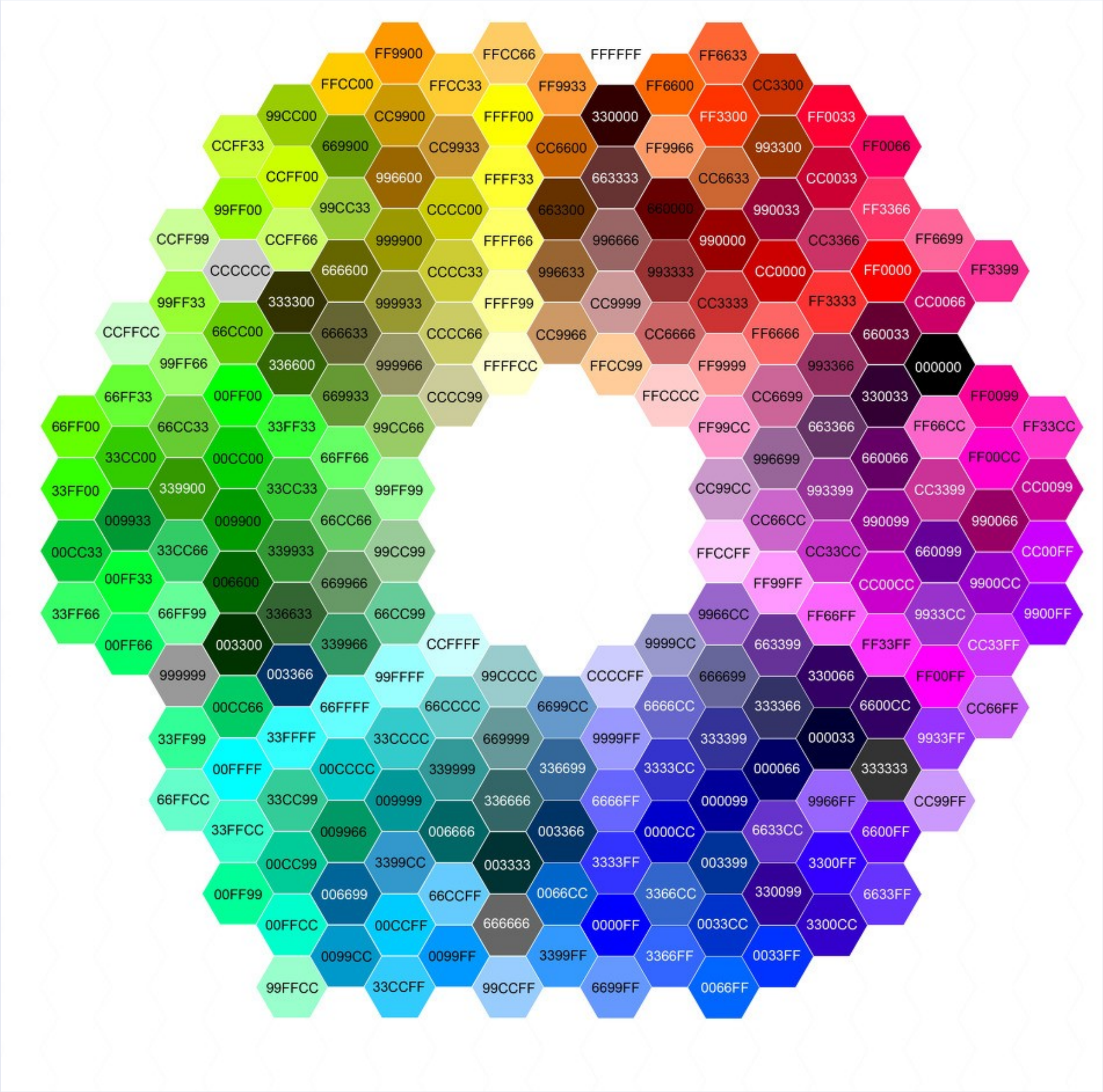
- Poichè ogni componente può assumere 256 diversi livelli, il numero di combinazioni possibili è:

  $256 \times 256 \times 256 =$ circa **16 milioni di colori**

- Ragionamento alternativo: con 24 bit, si possono costruire 2^{24} sequenze diverse di 0 e 1, e quindi si possono rappresentare 2^{24} colori diversi.

  $2^{24} = 2^{4} \times 2^{20} =$ circa **16 milioni di colori**

Alcuni codici colore HEX ...

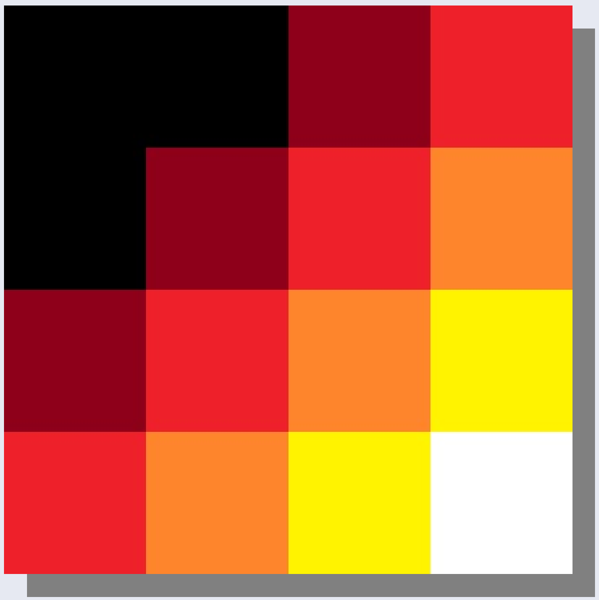


Codifica delle immagini

- Un'immagine digitale è costituita da una griglia rettangolare di punti detti **pixel** (= picture element).
- Ad ogni pixel è associato un colore
- in linea di principio, un'immagine può essere codificata specificando le dimensioni della griglia (larghezza e altezza) e il colore di ogni pixel (secondo un ordine concordato).
- Esempio di immagine 4x4 pixel

Codifica delle immagini

Una possibile codifica è la seguente*:



- il primo byte indica la larghezza
- il secondo byte indica l'altezza
- i byte successivi, letti a gruppi di 3, indicano il colore dei pixel procedendo "per righe" a partire dall'alto.

l'immagine verrebbe codificata nella seguente sequenza di **50 byte**:

```
4,4,  0,0,0,  0,0,0,  136,0,21,  237,28,36,  0,0,0,  136,0,21,
237,28,36,  255,127,39,  136,0,2,  237,28,36,  255,127,39,
255,242,0,  237,28,36,  255,127,39,  255,242,0,  255,255,255
```

* i formati immagine reali sono molto più complessi, ed efficienti!

Codifica dei suoni

- Ciò che percepiamo come suono è una variazione di pressione nel tempo, rilevata dal nostro orecchio
- Un microfono misura questa variazione di pressione e la trasforma in un segnale elettrico con andamento analogo (ottiene cioè un segnale analogico)
- Per codificare il suono in modo digitale, il segnale analogico deve essere campionato e quantizzato

  campionamento: si effetttua una misura del segnale a intervalli di tempo regolari ottendo una serie di campioni. Per ottenere una buona qualità del suono (qualità CD) occorre campionare a **44100 Hz**

  quantizzazione: ogni campione è memorizzato usando un numero un certo numero di bit

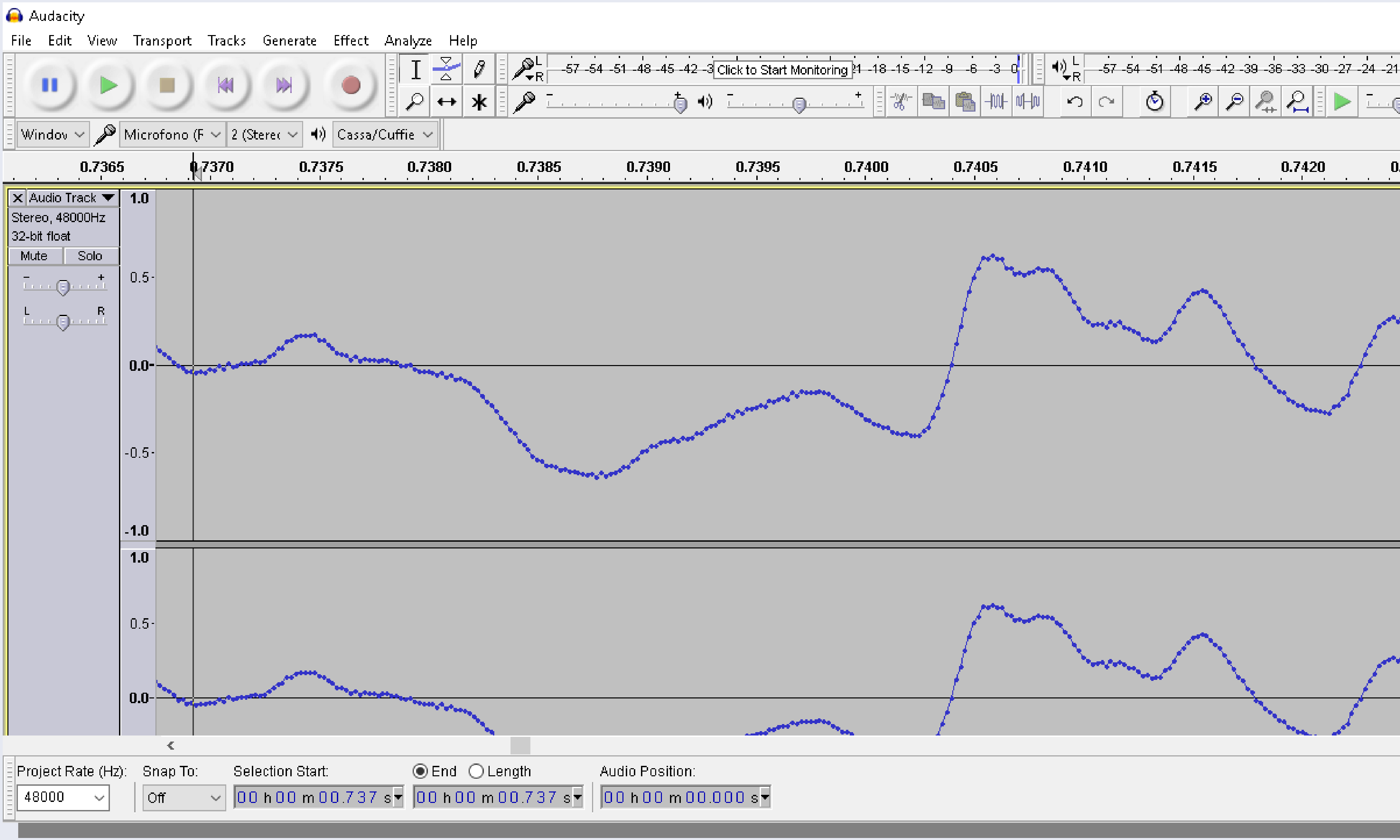
Codifica dei suoni

- Ciò che percepiamo come suono è una variazione di pressione nel tempo, rilevata dal nostro orecchio
- Un microfono misura questa variazione di pressione e la trasforma in un segnale elettrico con andamento analogo (ottiene cioè un segnale analogico)
- Per codificare il suono in modo digitale, il segnale analogico deve essere campionato e quantizzato

  campionamento: si effetttua una misura del segnale a intervalli di tempo regolari ottendo una serie di campioni.

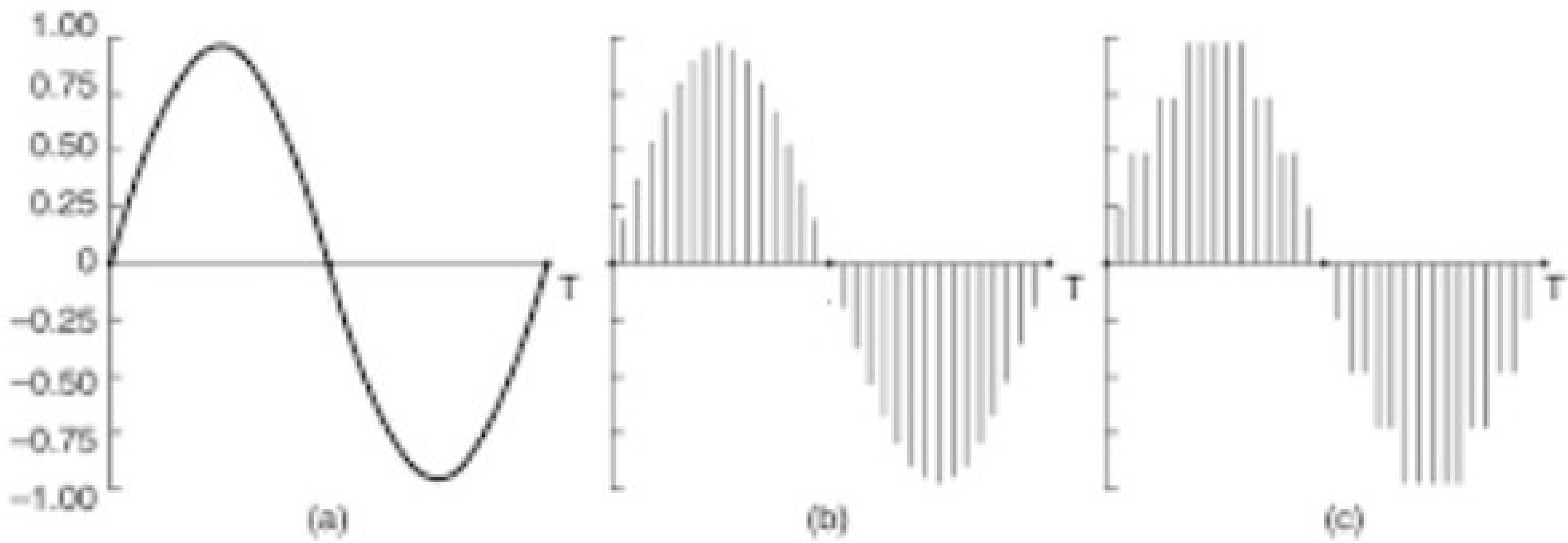
  quantizzazione: ogni campione è memorizzato usando un numero un certo numero di bit. Ciò comporta che il valore del campione venga approssimato con il valore più vicino fra quelli rappresentabili con i bit a disposizione. Esempio: con 4 bit per campione, si avranno a disposizione solo 16 diversi livelli.

Codifica dei suoni



Codifica dei suoni

segnale originale (analogico) **campionamento** **quantizzazione**



Codifica dei suoni

- Una misurazione del segnale in un certo istante è detta **campione**
- **frequenza di campionamento** = numero di campioni presi in un secondo
- la frequenza si misura in **Hertz** (1 Hz = 1 campione al secondo)
- L'audio con **qualità CD** prevede di campionare a **44100 Hz e** quantizzare i campioni usando **16 bit per campione**
- per dare l'illusione di un suono più realistico, si utilizza l'audio stereo
- Il suono stereo prevede la memorizzazione di **due** serie di campioni leggermente diverse: canale destro e canale sinistro

Codifica dei suoni

Problema: quanti byte sono necessari per memorizzare 1 minuto di suono stereo, campionato alla frequenza di 44100 Hz, utilzzando 16 bit per campione e senza compressione?

Soluzione

60 sec × **44100** campioni/sec × **2** byte × **2** canali =

= 10584000 byte = circa 10 MB !!!

Codifica dei caratteri

Argomenti

- Codifica ASCII a 7-bit (US-ASCII)
- Codifiche ASCII estese
    - ISO 8859-1 (Latin 1)
- Standard Unicode
    - Codifica UTF-8
    - Codifica UTF-16
    - Codifica UTF-32

ASCII

- American Standard Code for Information Interchange
- La codifica ASCII originale e' una codifica a **7 bit** (nata per la rete telegrafica, anni '60)
  - con 7 bit si codificano 2^7=128 caratteri
  - caratteri rappresentabili: caratteri di controllo, lettere dell'alfabeto inglese, numeri, segni di interpunzione
  - ad ogni carattere è associato un numero da 0 a 127, detto **codice ascii** del carattere
- Il codice ASCII a 7 bit è noto anche come **ASCII standard**, o **US-ASCII**
- se si rappresentano con 8 bit, tutti i caratteri ASCII standard hanno 0 nel bit più significativo (00000000 .. 01111111)

ASCII standard (7 bit)

Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char
00000000	0	Null	00100000	32	Spc	01000000	64	@	01100000	96	`
00000001	1	Start of heading	00100001	33	!	01000001	65	A	01100001	97	a
00000010	2	Start of text	00100010	34	”	01000010	66	B	01100010	98	b
00000011	3	End of text	00100011	35	#	01000011	67	C	01100011	99	c
00000100	4	End of transmit	00100100	36	$	01000100	68	D	01100100	100	d
00000101	5	Enquiry	00100101	37	%	01000101	69	E	01100101	101	e
00000110	6	Acknowledge	00100110	38	&	01000110	70	F	01100110	102	f
00000111	7	Audible bell	00100111	39	’	01000111	71	G	01100111	103	g
00001000	8	Backspace	00101000	40	(	01001000	72	H	01101000	104	h
00001001	9	Horizontal tab	00101001	41	)	01001001	73	I	01101001	105	i
00001010	10	Line feed	00101010	42	*	01001010	74	J	01101010	106	j
00001011	11	Vertical tab	00101011	43	+	01001011	75	K	01101011	107	k
00001100	12	Form Feed	00101100	44	,	01001100	76	L	01101100	108	l
00001101	13	Carriage return	00101101	45	-	01001101	77	M	01101101	109	m
00001110	14	Shift out	00101110	46	.	01001110	78	N	01101110	110	n
00001111	15	Shift in	00101111	47	/	01001111	79	O	01101111	111	o
00010000	16	Data link escape	00110000	48	0	01010000	80	P	01110000	112	p
00010001	17	Device control 1	00110001	49	1	01010001	81	Q	01110001	113	q
00010010	18	Device control 2	00110010	50	2	01010010	82	R	01110010	114	r
00010011	19	Device control 3	00110011	51	3	01010011	83	S	01110011	115	s
00010100	20	Device control 4	00110100	52	4	01010100	84	T	01110100	116	t
00010101	21	Neg. acknowledge	00110101	53	5	01010101	85	U	01110101	117	u
00010110	22	Synchronous idle	00110110	54	6	01010110	86	V	01110110	118	v
00010111	23	End trans. block	00110111	55	7	01010111	87	W	01110111	119	w
00011000	24	Cancel	00111000	56	8	01011000	88	X	01111000	120	x
00011001	25	End of medium	00111001	57	9	01011001	89	Y	01111001	121	y
00011010	26	Substitution	00111010	58	:	01011010	90	Z	01111010	122	z
00011011	27	Escape	00111011	59	;	01011011	91	[	01111011	123	{
00011100	28	File separator	00111100	60	<	01011100	92	\	01111100	124	|
00011101	29	Group separator	00111101	61	=	01011101	93	]	01111101	125	}
00011110	30	Record Separator	00111110	62	>	01011110	94	^	01111110	126	~
00011111	31	Unit separator	00111111	63	?	01011111	95	_	01111111	127	Del

ASCII standard (7 bit)

Da ricordare:

- i primi 32 caratteri (da 0 a 31) non corrispondono a simboli grafici e sono detti **caratteri di controllo** o **caratteri non stampabili**
- fra i caratteri di controllo ricordare i caratteri per l'**andata a capo**:
  - Codice ASCII **10**: **line feed** (o new line) ('\n' in C)
  - Codice ASCII **13**: **carriage return** ('\r' in C)
- al carattere **spazio** (' ') corrisponde il codice ASCII **32**
- al carattere '**0**' corrisponde il codice ASCII **48**
- al carattere '**A**' corrisponde il codice ASCII **65**
- al carattere '**a**' corrisponde il codice ASCII **97**
- i codici delle lettere minuscole si ottengono sommando 32 al codice della maiuscola corrispondente

Codifiche ASCII estese

- Tra i caratteri ASCII standard non compaiono le lettere accentate usate in Italia (non usate negli USA) così come molte altre lettere e simboli:

  Ğ (Turchia) Ű (Ungheria) Š (Finlandia)

- Idea: **estendere il codice ASCII** portandolo a 8 bit, in modo da utilizzare anche i codici con il bit più significativo a 1 (codici ASCII da 128 a 255)
- Problema: 128 caratteri in più non sono sufficienti per codificare tutti i caratteri che mancano
- Soluzione: definire una serie di codifiche ASCII estese. Ogni codifica estesa copre le necessità di una certa area geografica.
- La codifica ASCII estesa più nota è **ISO-8859-1**, nota anche come **codifica *Latin-1***

Esempi di codifiche ASCII estese

- ISO-8859-1 Latin alphabet part 1 North America, Western Europe
- ISO-8859-2 Latin alphabet part 2 Eastern Europe
- ISO-8859-3 Latin alphabet part 3 Sotheast Europe, Esperanto
- ISO-8859-4 Latin alphabet part 4 Scandinavia/Baltics
- ISO-8859-5 Latin/Cyrillic part 5 Cyrillic
- ISO-8859-6 Latin/Arabic part 6 Arabic alphabet
- ISO-8859-7 Latin/Greek part 7 Greek
- ISO-8859-8 Latin/Hebrew part 8 Hebrew alphabet

ISO-8859-1 (latin-1)

- E' una codifica a 8 bit
- I primi 128 codici coincidono con ASCII a 7 bit
- L'intervallo esteso (da **128** a **255**) è utilizzato per codificare lettere usate dai paesi dell'europa occidentale (Italia, Germania, Spagna, ..)

  Ad esempio, nella codifica latin-1: 224 = à

- Non comprende caratteri usati nei paesi dell'europa orientale, per i quali esiste la codifica **ISO-8859-2 (latin-2)**

  Ad esempio, nella codifica latin-2: : 224 = ŕ

Codifiche errate dei file di testo

Problema: cosa accade se un file di testo, scritto in un paese dell'europa dell'est (ISO-8859-2) viene visualizzato su un computer in Italia (ISO-8859-1) ?

Soluzione: i caratteri ASCII standard (codici tra 0 e 127) rimangono immutati, mentre i caratteri non-ASCII (128..255) vengono visualizzati diversamente!

ISO-8859-2

a = 97

ŕ = 224

01100001
11100000

file di testo

ISO-8859-1

97 = a

224 = à

Unicode: motivazioni

Problemi delle codifiche ASCII estese:

- **errori di visualizzazione** quando un file creato con una codifica viene aperto su un sistema che utlizza una codifica differente
- è complicato combinare caratteri appartenenti a **estensioni diverse nello <u>stesso</u> file**

Soluzione:

è stato svilupato un nuovo standard, denominato **Unicode**, con l'obiettivo di codificare in modo "unico" qualsiasi simbolo utilizzato nel mondo

Come funziona Unicode?

- Unicode associa ad ogni simbolo un numero intero, detto **code-point**

  Esempio **code-point 8734**: **∞**

- Lo standard consente di utilizzare più di 1 milione di codepoint
- I code-point da 0 a 127 sono assegnati in modo da coincidere con ASCII.

  Esempio **code-point 97**: **a**

- Il modo in cui un code-point viene trasformato in una sequenza di bit dipende dalla **codifica Unicode** utilizzata:
  - UTF-8 (Unicode Transformation Format a 8 bit)
  - UTF-16 (Unicode Transformation Format a 16 bit)
  - UTF-32 (Unicode Transformation Format a 32 bit)

UTF-32

- La più **semplice** tra le codifiche unicode è UTF-32
- Codifica a **lunghezza fissa**: ogni code-point è rappresentato in binario su 32 bit (4 byte)

  Esempio:

  in UTF-32, il carattere ∞ è rappresentato come:

  00000000 00000000 00100010 00011110

UTF-32

- La più **semplice** tra le codifiche unicode è UTF-32
- Codifica a **lunghezza fissa**: ogni code-point è rappresentato in binario su 32 bit (4 byte)

Esempio:

in UTF-32, il carattere ∞ è rappresentato come:

00000000 00000000 00100010 00011110

UTF-32

Problemi:

- **enorme spreco di spazio**: se un testo utilizza solo caratteri ASCII, il file codificato in UTF-32 è 4 volte più grande dello stesso file codificato in ASCII
- **non c'è compatibilità con ASCII**: un file che utilizza solo caratteri ASCII standard deve essere convertito per essere portato nel nuovo formato

Soluzione: UTF-8

UTF-8

Idea: utilizzare una **codifica a lunghezza variabile** in cui

- i caratteri più frequenti sono codificati usando un solo byte
- i caratteri meno frequenti sono codificati usando più byte

UTF-8 è una codifica a lunghezza variabile in cui:

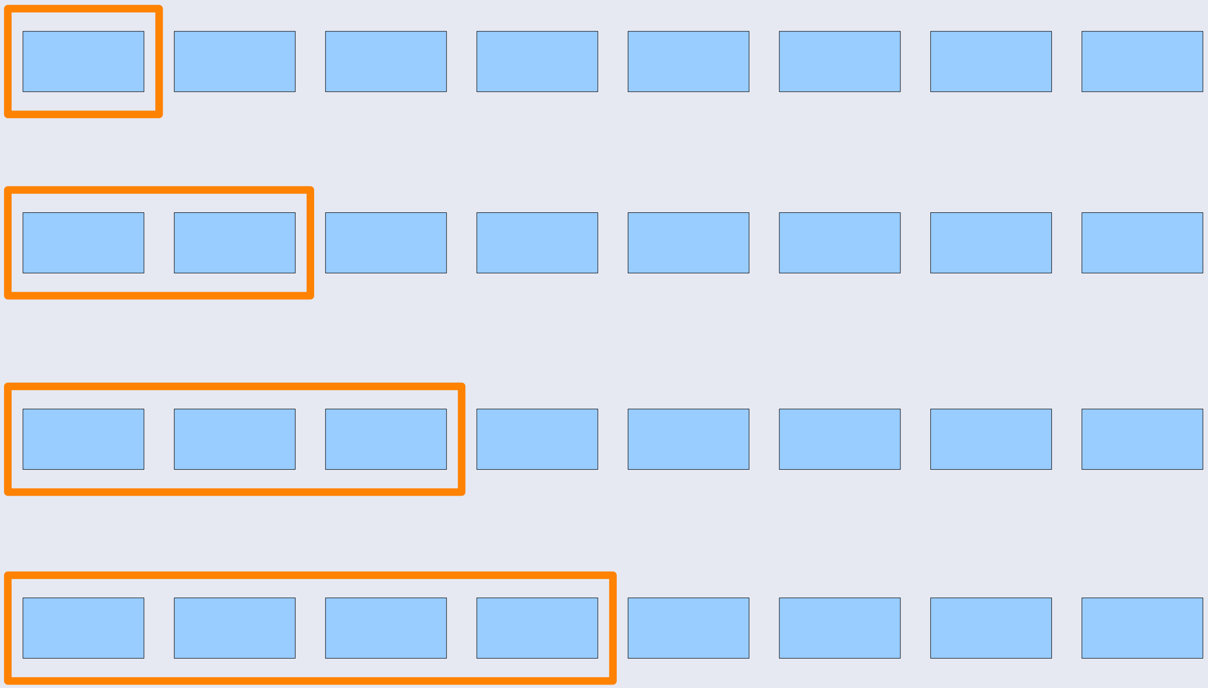
- i caratteri ASCII standard sono codificati con 1 byte
- tutti gli altri caratteri sono codificati utilizzando 2, 3 o 4 byte

In altre parole:

- in UTF-8 un code-point è codificato utilizzando 1, 2, 3 o 4 byte
- I code-point tra 0 e 127 occupano un solo byte

Come funziona UTF-8

Per capire come funziona UTF-8, partiamo dal problema della decodifica: data una sequenza di byte, come si risale alla sequenza di caratteri Unicode?



Come funziona UTF-8

1 byte

0XXXXXXX

Se il byte inizia con 0 allora il codepoint è codificato con un solo byte. I bit XXXXXXX forniscono il valore del codepoint (in binario)

2 byte

110XXXXX	10YYYYYY

Un byte che inizia con 110 indica che il codepoint si estende anche sul byte successivo. Il byte successivo inizia con 10. I bit XXXXXYYYYYY forniscono il valore del codepoint.

3 byte

1110XXXX	10YYYYYY	10ZZZZZZ

4 byte

11110XXX	10YYYYYY	10ZZZZZZ	10TTTTTT

Esempio

Problema Decodifcare la seguente squenza di byte, sapendo che si tratta di testo codificato in UTF-8

01000001 11100010 10001001 10100000 01000010

Soluzione

01000001 11100010 10001001 10100000 01000010

code-point: $1000001_{(2)}$	code-point: $0010001001100000_{(2)}$	code-point: $1000010_{(2)}$
||	||	||
65	8800	66
A	≠	B

UTF-16

- UTF-16 è un compromesso tra UTF-32 e UTF-8
- E' la codifica adottata internamente da molti linguaggi di programmazione per rappresentare caratteri e stringhe (es Java)
- Un code-point in UTF-16 è codificato usando 2 o 4 byte