

# ARCHITETTURA DEGLI ELABORATORI

A.A. 2020-2021

Università di Napoli Federico II

*Corso di Laurea in Informatica*

Docenti

Proff. Luigi Sauro gruppo 1 (A-G)  
Silvia Rossi gruppo 2 (H-Z)



# Esercizi

- Fornire la rappresentazione binaria in complemento a 2 ad 8 bit del numero -15
- Fornire la rappresentazione binaria in complemento a 2 ad 8 bit del numero -109
- Eseguire la somma dei numeri 5 e -32 espressa in completamento a 2 ad 8 bit
- Convertire in esadecimale il numero -53248 espresso in completamento a 2 ad 16 bit

# Binary Coded Decimal

- L'elettronica degli elaboratori è binaria mentre la mente umana è abituata a ragionare in decimale.
- I codici Binary Coded Decimal hanno lo scopo di fornire una naturale rappresentazione binaria del sistema numerico decimale.
- Essendo 10 i simboli da codificare ('0',...,'9') avremo bisogno di 4 bit.
  - Notate che con 4 bit consentono di avere  $2^4=16$  combinazioni che in binario puro corrispondono ai numeri 0,1,2,...9,10,...,15
  - Poiché le combinazioni da 10 a 15 non si usano, la codifica BCD è *ridondante*

# Binary Coded Decimal

Cifra decimale	Cifra BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

i codici 1010, 1011, 1100, 1101, 1110 ,1111 non sono utilizzati (codifica ridondante)

Numeri decimali a più cifre di codificano giustapponendo le codifiche cifra per cifra  
Esempio:

$$23=0010\ 0011_{BCD}$$

Nota che

$$23=00100011_{BCD} \neq 00100011_2 = 35$$

Anche le somme differiscono:

$$1000_{BCD} + 0011_{BCD} = 8 + 3 = 11 = 00010001_{BCD}$$

e non 1011 (che è un codice non utilizzato)

# Binary Coded Decimal

- Naturalmente, la codifica BCD viene usata solo in funzione di interfaccia per rendere comprensibile ad operatori umani i risultati di una elaborazione numerica binaria.
  - Operazione propedeutica alla sua visualizzazione su un display numerico decimale (es. display a sette segmenti).
  - I quattro bit di ciascuna cifra codificata BCD vengono inviati ad un circuito di decodifica che provvederà ad attivare i segmenti della cifra corrispondente.
- E' opportuno sottolineare che, mentre la codifica binaria è il primo passo per approdare ad un sistema numerico che porta poi all'aritmetica binaria, non esiste una aritmetica BCD.

# Numeri con virgola

- Vediamo come rappresentare (approssimazioni di numeri) reali.
- Consideriamo un numero con virgola nella base naturale 10

$$c_{m-1}c_{m-2}\cdots c_0, c_{-1}\cdots c_{-k} = c_{m-1} \cdot 10^{m-1} + c_{m-2} \cdot 10^{m-2} + \cdots + c_0 \cdot 10^0 + c_{-1} \cdot 10^{-1} + \cdots + c_{-k} \cdot 10^{-k}$$

- Per una generica base  $b$  abbiamo la generalizzazione

$$\sum_{i=-k}^{h-1} c_i \cdot b^i$$

# Cambiamenti di base con virgola

- Per il cambiamento di un numero  $x$  da una base  $a$  ad una  $b$  si procede separatamente per la parte intera e per quella frazionaria
- Per la parte intera l'algoritmo chiaramente è quello visto in precedenza
- Per la parte frazionaria in procedimento è l'inverso:
  - si moltiplica la parte frazionaria di  $x$  per  $b$  (entrambi codificati in base  $a$ ). La parte intera  $i$  del risultato sarà un numero da 0 a  $b-1$  che, convertito in base  $b$ , costituisce la prima cifra frazionaria di  $x$  in base  $b$ .
  - La parte frazionaria del risultato  $f$  si moltiplica ancora per  $b$  e la nuova parte intera, convertita in base  $b$ , costituisce la seconda cifra frazionaria.
  - Si procede iterativamente finché la parte frazionaria del risultato è zero o si è raggiunta la precisione desiderata

# Cambiamenti di base con virgola

convertire in binario 0,625

$$0,625 \cdot 2 = 1,250 \quad i=1 \quad f=0,250 \quad c_{-1}=1$$

$$0,250 \cdot 2 = 0,500 \quad i=0 \quad f=0,500 \quad c_{-2}=0$$

$$0,500 \cdot 2 = 1,000 \quad i=1 \quad f=1,000 \quad c_{-3}=1$$

$$0,625 = 0,101_2$$

Convertire  $0,65_8$  in base 7 fino alla 3 cifra significativa

- Convertiamo prima in decimale

$$6 \times 8^{-1} + 5 \times 8^{-2} = 6 \times 0,125 + 5 \times 0,015625 = 0,75 + 0,078125 = 0,828125$$

- Convertiamo  $0,828125$  in base 7:  $0,554_7$

$$0,828125 \times 7 = 5,796875 \quad 5$$

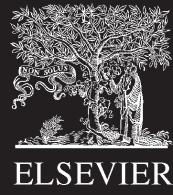
$$0,796875 \times 7 = 5,578125 \quad 5$$

$$0,578125 \times 7 = 4,046875 \quad 4$$

# Numbers with Fractions

Two common notations:

- **Fixed-point:** binary point fixed
- **Floating-point:** binary point floats to the right of the most significant 1



# Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.**1100**

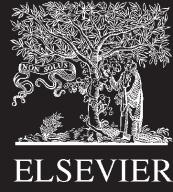
$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Binary point is implied
- The number of integer and fraction bits must be agreed upon beforehand



# Fixed-Point Number Example

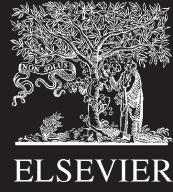
- Represent  $7.5_{10}$  using 4 integer bits and 4 fraction bits.



# Fixed-Point Number Example

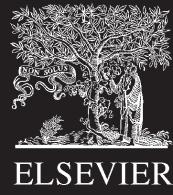
- Represent  $7.5_{10}$  using 4 integer bits and 4 fraction bits.

**01111000**



# Signed Fixed-Point Numbers

- **Representations:**
  - Sign/magnitude
  - Two's complement
- **Example:** Represent  $-7.5_{10}$  using 4 integer and 4 fraction bits
  - **Sign/magnitude:**
  - **Two's complement:**



# Signed Fixed-Point Numbers

- **Representations:**
  - Sign/magnitude
  - Two's complement
- **Example:** Represent  $-7.5_{10}$  using 4 integer and 4 fraction bits
  - **Sign/magnitude:**

11111000

- **Two's complement:**

$$\begin{array}{rcl} 1. +7.5: & 01111000 \\ 2. \text{ Invert bits:} & 10000111 \\ 3. \text{ Add 1 to lsb:} & + & 1 \\ & \hline & 10001000 \end{array}$$



# Rappresentazione in virgola fissa

- Ci occupiamo ora di rappresentare numeri positivi frazionari con parole (binarie) di lunghezza fissata m
- Nella rappresentazione in virgola fissa si suddividono gli m bit in due sottoparole
  - i primi h bit (con  $h < m$ ) sono dedicati alla codifica della parte intera
  - i rimanenti  $k = m - h$  bit rappresentano la parte frazionaria
- Supponiamo di far uso di parole a 32 bit e di dedicare 20 bit per la parte intera e 12 per la parte frazionaria:
  - Massimo intero codificabile:  $2^{19} - 1$
  - Con 12 bit per la parte frazionaria si codificano circa 3 cifre decimali (ricordate  $\log_2 10 = 3,32$ )

# Esercizi

- Fornire la rappresentazione binaria in virgola fissa del numero 7.25 con 4 bit per la parte intera e 4 bit di parte frazionaria
- Riportare in codice esadecimale la rappresentazione binaria in virgola fissa del numero 2.33 con 4 bit per la parte intera e 4 bit di parte frazionaria, trascurando l'eventuale resto
- Fornire la rappresentazione binaria in virgola fissa del numero 55.4121 con 8 bit per la parte intera e 4 bit di parte frazionaria, trascurando l'eventuale resto
- Eseguire la somma  $12.25+5.5$  nella rappresentazione binaria in virgola fissa con 5 bit per la parte intera e 3 per quella frazionaria
- Eseguire la somma  $9.875+10.5$  nella rappresentazione binaria in virgola fissa con 5 bit per la parte intera e 3 per quella frazionaria

# Rappresentazione in virgola mobile

- Nella rappresentazione in virgola fissa disponendo di parole a 64 bit e supponendo di dedicarle tutte per la rappresentazione della parte frazionaria riusciremmo a codificare circa 18 cifre decimali
- Alcune applicazioni scientifiche operano con valori ancora più piccoli, altre invece con valori molto grandi
- In generale, fissare a priori la lunghezza della parte intera  $h$  e di quella frazionaria  $k$  costituisce una scelta rigida
- Per ovviare a queste difficoltà è stata introdotta una rappresentazione detta in virgola mobile

# Rappresentazione in virgola mobile

- Questa sfrutta la rappresentazione di un numero in una data base b:

$$x = (-1)^s m b^e$$

- $s$  determina il segno: 0 positivo, 1 negativo
  - $m$  è detta mantissa
  - $e$  è detto esponente
- Un numero reale è quindi rappresentato da una triple  $(s,m,e)$ . Notate però che vi sarebbero infiniti modi di rappresentare  $x$ . Ad esempio, per 34,67 in base 10:
  - $3467 \times 10^{-2}$
  - **3,467 x 10**
  - $0,3467 \times 10^2$
- Useremo la rappresentazione scientifica in cui  $b > m \geq 1$
- In binario questo vuol dire che la mantissa è sempre del tipo 1,xyz
  - Chiaramente nella rappresentazione della mantissa non sprecherò memoria per rappresentare il primo bit “1” e lo considero sottointeso

# Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation
- For example, write  $273_{10}$  in scientific notation:

$$273 = 2.73 \times 10^2$$

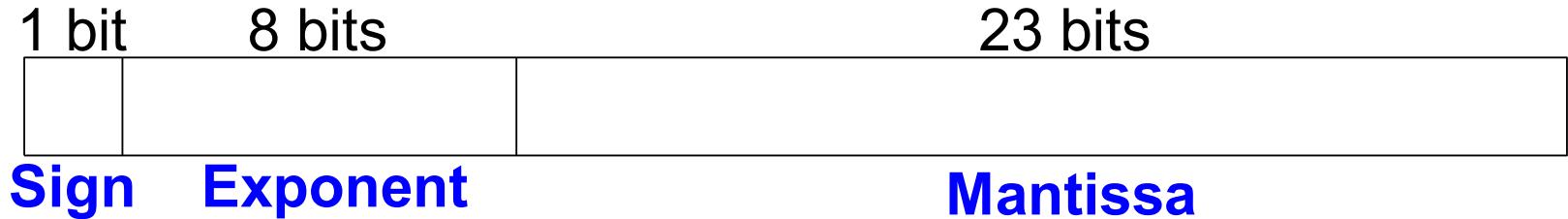
- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

- **M** = mantissa
- **B** = base
- **E** = exponent
- In the example,  $M = 2.73$ ,  $B = 10$ , and  $E = 2$

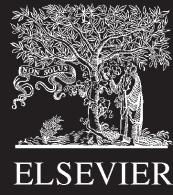


# Floating-Point Numbers



- **Example:** represent the value  $228_{10}$  using a 32-bit floating point representation

We show three versions – final version is called the **IEEE 754 floating-point standard**



# Floating-Point Representation 1

1. Convert decimal to binary (**don't reverse steps 1 & 2!**):

$$228_{10} = 11100100_2$$

2. Write the number in “binary scientific notation”:

$$11100100_2 = 1.11001_2 \times 2^7$$

3. Fill in each field of the 32-bit floating point number:

- The sign bit is positive (0)
- The 8 exponent bits represent the value 7
- The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
Sign	Exponent	Mantissa
0	00000111	11 1001 0000 0000 0000 0000



# Floating-Point Representation 2

- First bit of the mantissa is always 1:
  - $228_{10} = 11100100_2 = \textcolor{red}{1}.\textcolor{blue}{11001} \times 2^7$
- So, no need to store it: *implicit leading 1*
- Store just fraction bits in 23-bit field

1 bit	8 bits	23 bits
Sign	Exponent	Fraction
0	00000111	110 0100 0000 0000 0000 0000



# Floating-Point Representation 3

- *Biased exponent:* bias = 127 ( $01111111_2$ )

- Biased exponent = bias + exponent
  - Exponent of 7 is stored as:

$$127 + 7 = 134 = 0x10000110_2$$

- The IEEE 754 32-bit floating-point representation of  $228_{10}$

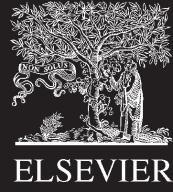
1 bit	8 bits	23 bits
Sign	Biased Exponent	Fraction
0	10000110	110 0100 0000 0000 0000 0000

in hexadecimal: **0x43640000**



# Floating-Point Example

Write  $-58.25_{10}$  in floating point (IEEE 754)



# Floating-Point Example

Write  $-58.25_{10}$  in floating point (IEEE 754)

1. Convert decimal to binary:

$$58.25_{10} = \textcolor{blue}{111010.01}_2$$

2. Write in binary scientific notation:

$$\textcolor{blue}{1.1101001} \times 2^5$$

3. Fill in fields:

Sign bit: **1** (negative)

8 exponent bits:  $(127 + 5) = 132 = \textcolor{blue}{10000100}_2$

23 fraction bits: **110 1001 0000 0000 0000 0000**

1 bit	8 bits	23 bits
Sign	Exponent	Fraction
1	100 0010 0	110 1001 0000 0000 0000 0000

in hexadecimal: **0xC2690000**



# Floating-Point: Special Cases

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000000000000
$\infty$	0	11111111	00000000000000000000000000000000
$-\infty$	1	11111111	00000000000000000000000000000000
NaN	X	11111111	non-zero



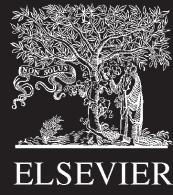
# Floating-Point Precision

- **Single-Precision:**
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127
- **Double-Precision:**
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023



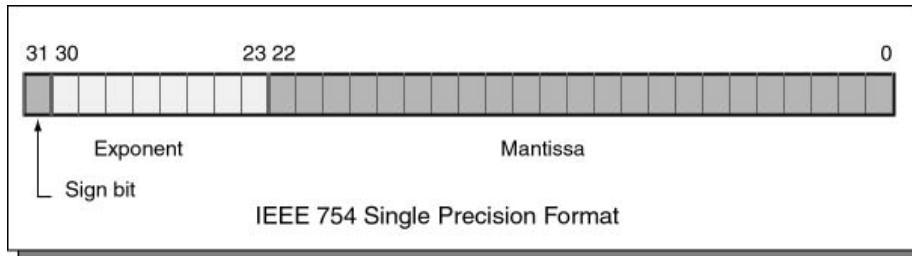
# Floating-Point: Rounding

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
  - Down
  - Up
  - Toward zero
  - To nearest
- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
  - Down: 1.100
  - Up: 1.101
  - Toward zero: 1.100
  - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)



# Standard IEEE 754

- Una rappresentazione largamente adottata è quella dell' Institute of Electrical and Electronical Engineering (IEEE)
- Prevede 4 diversi formati per calcoli in singola o doppia precisione di tipo semplice o esteso (raramente usato)
- Descriveremo i tipi semplici
  - Singola precisione: 32 bit totali, 1 per il segno, 23 per la mantissa e 8 per l'esponente



- Doppia precisione: 64 bit totali, 1 per il segno, 52 per la mantissa e 11 per l'esponente

# Dettagli dello Standard IEEE

- Riguardo all'esponente, il complemento a 2 inverte l'ordine fra positivi e negativi (i numerali associati ai negativi sono maggiori di quelli associati ai positivi)
- Per ovviare a questo difetto si usa nello standard IEEE la “rappresentazione polarizzata”
- Dati  $k$  bit assegnati all'esponente, il più grande esponente rappresentabile è  $P = 2^{k-1} - 1$
- Un valore  $e$  (compreso tra 0 e  $2^k - 1$ ) in rappresentazione polarizzata codifica l'esponente  $e' = e - P$
- Ad esempio sia  $k=3$  ( $P = 2^2 - 1 = 3$ ):

$$e = 0 \rightarrow e' = -3$$

$$e = 4 \rightarrow e' = 1$$

$$e = 1 \rightarrow e' = -2$$

$$e = 5 \rightarrow e' = 2$$

$$e = 2 \rightarrow e' = -1$$

$$e = 6 \rightarrow e' = 3$$

$$e = 3 \rightarrow e' = 0$$

$$e = 7 \rightarrow e' = 4$$

- Quindi ora gli esponenti sono codificati in ordine crescente da  $-3 (000_2)$  a  $4 (111_2)$

# Dettagli dello Standard IEEE

- Per la mantissa si adotta la rappresentazione scientifica  $1,xyz\dots$
- Riassumendo ad  $(s, m, e)$  è associato il numero

$$x = (-1)^s \cdot 1,m \cdot 2^{e-p}$$

- Problema con questa rappresentazione non riesco a rappresentare lo 0. Per questo lo standard IEEE considera delle eccezioni.

# Dettagli dello Standard IEEE

precisione singola

	e=0 ( $00000000_2$ )	e=1,...,254	e=255 ( $11111111_2$ )
m=0	$(-1)^s \times 0$	$(-1)^s \times 1,0 \times 2^{e-127}$	$(-1)^s \infty$
m≠0	$(-1)^s \times 0,m \times 2^{-126}$	$(-1)^s \times 1,m \times 2^{e-127}$	NaN (indefinito)

Per  $m = 0$  ed  $e = 0$ , si hanno due rappresentazioni dello 0, a seconda che  $s$  sia 1 (segno negativo) o 0 (segno positivo).

Le combinazioni che corrispondono ad  $m = 0$  ed  $e = 255$  sono la rappresentazione di  $\pm \infty$ .

Per  $m \neq 0$  ed  $e = 0$  è prevista una rappresentazione per numeri molto vicini allo 0. Ricordate che  $m$  non è vincolato a iniziare con 1. Quindi se usassimo la rappresentazione usuale avrei, assumendo  $s=0$ , otterrei numeri maggiori di  $2^{-127}$ :  $1,m \times 2^{-127} > 2^{-127}$ . Invece per  $m=00\dots01$ :  $0,0\dots01 \times 2^{-126} = 2^{-23} \times 2^{-126} = 2^{-149}$

# Esempi standard IEEE

- Il numero decimale 1021 è rappresentato dalla tripla  $(s, m, e)$ :

$(0; 111\ 1111\ 0100\ 0000\ 0000\ 0000; 1000\ 1000)$

- $s=0$  perché il numero è positivo.
- La rappresentazione binaria di 1021 è:

$$1\ 111\ 1111\ 01 = 1,111\ 1111\ 01 \times 2^9$$

- $m=111\ 1111\ 0100\ 0000\ 0000\ 0000$
- Avendo 8 bit a disposizione per l'esponente,  $P = 2^{8-1} - 1 = 2^7 - 1 = 127$ .
- L'esponente  $e$  si ricava invertendo la relazione di definizione della costante di polarizzazione:

$$e = e' + P = 9 + 127 = 136$$

che, convertito in binario, da 1000 1000

# Esercizi

- Rappresentare nel formato IEEE 754
  - 1.25
  - 10
  - -0.625
  - 1007
  - 3.875

# Operazioni in virgola mobile

- La moltiplicazione fra due numeri  $n_1$  ed  $n_2$  rappresentati in virgola mobile dalle triple  $(s_1, m_1, e_1)$  ed  $(s_2, m_2, e_2)$  ha per risultato il numero rappresentato dalla tripla:  $(s, e, m)$  in cui:
  - $s = 0$  se  $s_1 = s_2$  oppure:  $s = 1$  se  $s_1 \neq s_2$
  - $e = e_1 + e_2$
  - $m = m_1 \times m_2$
  - Dopo l'operazione di solito è necessaria la normalizzazione del risultato
- La divisione si effettua con regole analoghe.
- L'addizione e la sottrazione sono più complesse perché prima di effettuarle bisogna rendere uguali gli esponenti.
  - Durante questa operazione se i numeri sono uno molto grande ed uno molto piccolo, per effetto dello scorrimento delle mantisse per pareggiare gli esponenti, si possono perdere cifre significative.

# Perdita di cifre significative

- Vediamo con un esempio perché si possono perdere cifre significative effettuando una somma. Per semplicità, considereremo una coppia di numeri decimali espressi attraverso una mantissa di 4 cifre ed un esponente di una sola cifra:

$$n_1 = 1,3435 \times 10^3 \text{ ed } n_2 = 1,9970 \times 10^5.$$

- Per effettuare la somma si può portare l'esponente di  $n_1$  da 3 a 5. Siccome ciò equivale a moltiplicare di fattore 100, per mantenere il valore costante occorre contemporaneamente dividere per 100 la mantissa:

$$1,3435 \times 10^3 = 0,013435 \times 10^5$$

- Poiché le cifre della mantissa sono 4, occorre effettuare un arrotondamento. Per difetto otterremmo 0,0134.
- La somma dei due numeri risulta:  $2,0104 \times 10^5$  (invece di  $2,010435 \times 10^5$ )

# Codifica caratteri alfa-numerici

- I calcolatori, nonostante il nome italiano (in francese si chiamano ordinatori) sono spesso utilizzati per manipolare informazioni non numeriche.
- Si parla di caratteri "alfanumerici" per sottolineare che in un testo sono presenti:
  - caratteri alfabetici (a,b,c,d,...)
  - caratteri numerici (0,...,9)
  - segni di punteggiatura (!,?,...)
  - simboli particolari vario tipo (£, &, @, ...)
- I processori moderni non hanno istruzioni specifiche per testi. Quindi, si usano codifiche da testo a numeri interi
- Un testo è una sequenza di caratteri. I codici associano un numero intero ad ogni carattere.

# Codifiche di caratteri

## **1968 ASCII.**

Codice a 7 bit: 95 caratteri stampabili e 33 di controllo.

## **1980 Extended ASCII.**

Varie estensioni a 8 bit, con simboli grafici e lettere accentate.

## **1991 Unicode.**

Codice a 21 bit (1 milione di simboli). Attualmente (v. 9.0) definiti circa 128.000 caratteri! Viene ulteriormente codificato in **UTF-8**.

## **1992 UTF-8.**

Codifica di Unicode a lunghezza variabile (da 1 a 4 byte). Retro-compatibile con ASCII. UTF-8 è la codifica consigliata per XML e HTML.

# ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&			F			f
7	7	[BELL]	39	27	'			G			g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]			5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]			6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

"0": 011 0000

A: 100 0001

a: 110 0001

# Codice ASCII

- Ai primi 32 numerali sono assegnati caratteri di controllo
  - null indica la fine di una stringa
  - carriage return porta il cursore su una nuova riga (andata a capo)
  - horizontal tab è l'usuale carattere tab
- Altro tipo:
  - Bell dovrebbe far suonare un cicalino

# Alcuni caratteri Unicode

<b>codice</b>	<b>carattere</b>	<b>significato</b>
U+0434	Д	Lettera cirillica “de”
U+6328	ب	Lettera araba “ba”
U+0E10	ໂ	Lettera tailandese “tho than”
U+1F63A		“Smiling cat face with open mouth”
U+1F382		Torta di compleanno

# Il principio di UTF-8

- UTF-8 codifica ciascun carattere Unicode con una sequenza lunga da 1 a 4 byte . Il primo byte di un carattere indica quanto è lunga la sequenza:

Primo byte	Byte totali	Bit a disposizione del carattere
0xxx xxxx	1	7
110x xxxx	2	11
1110 xxxx	3	16
1111 0xxx	4	21

- Tutti i byte successivi nella sequenza hanno il formato 10xx xxxx

# Esempio di UTF-8

- Consideriamo il simbolo dell'euro “€”, codice Unicode U+20AC
- E’ un codice di 16 bit, quindi richiede 3 byte in UTF-8
- Vediamo come i 16 bit vengono distribuiti su 3 byte da UTF-8:

$0x20AC = 0010\ 0000\ 1010\ 1100$

- Codifica UTF-8:

1110 0010 1000 0010 1010 1100

3 byte

# Il successo di UTF-8

