Programmazione I

Rappresentare l'informazione

Daniel Riccio
Università di Napoli, Federico II

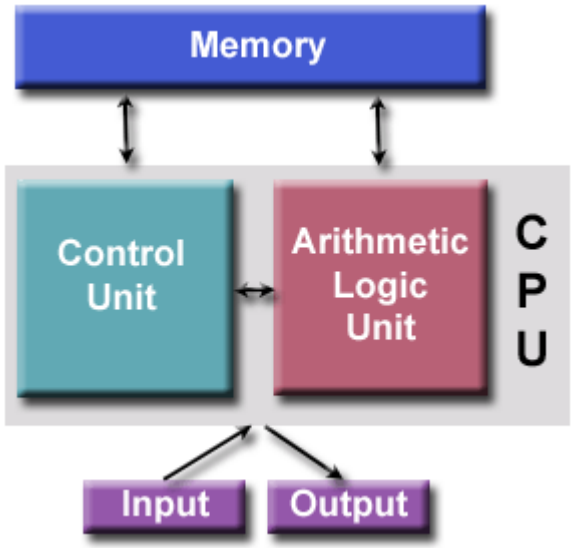
29 settembre 2021

Sommario

- Informazioni
  - Il bit
  - Il sistema binario
  - Rappresentazione di numeri interi
  - Rappresentazione di numeri razionali
  - Il codice ASCII

Il modello di Von Neumann

Gli attuali computer general-purpose sono di tipo **stored program** (o a programma memorizzato), e rispecchiano il modello generale della macchina di **Von Neumann**



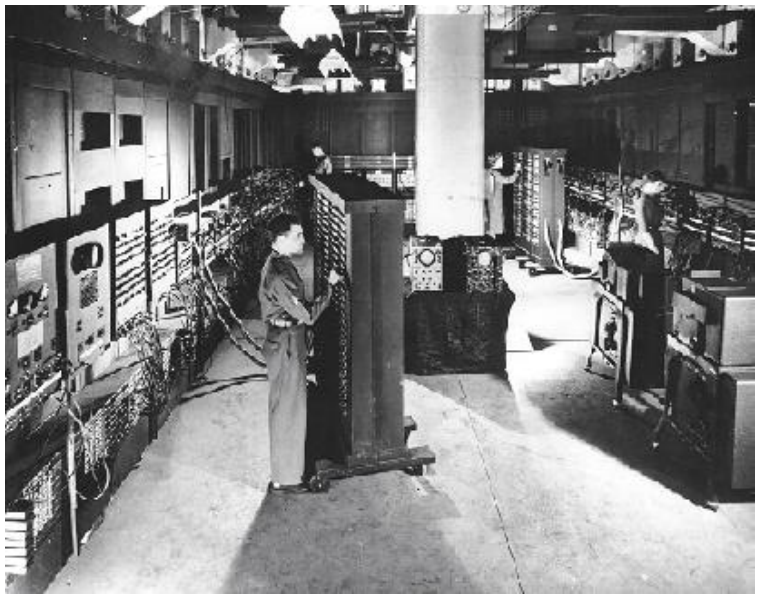
John von Neumann, nato János Lajos Neumann (Budapest, 28 dicembre 1903 – Washington, 8 febbraio 1957), è stato un matematico, fisico e informatico ungherese naturalizzato statunitense

Dal 1948, anno in cui il primo programma girava su un computer all'università di Manchester, la tecnologia ha conosciuto un incessante evoluzione:

- valvole
- transistor e circuiti integrati (IC)
- very large scale integrated (VLSI) circuits

Il primo calcolatore

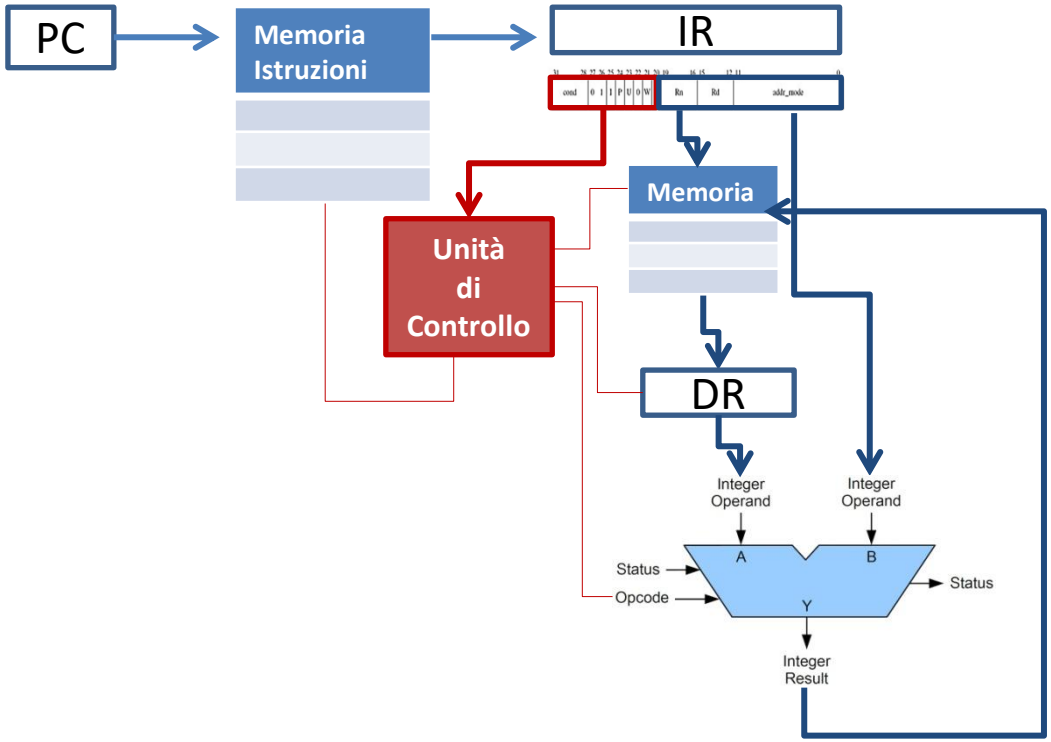
Le **Schede perforate** (**1890**) vengono introdotte da Herman Hollerit per automatizzare la tabulazione dei dati di un censimento

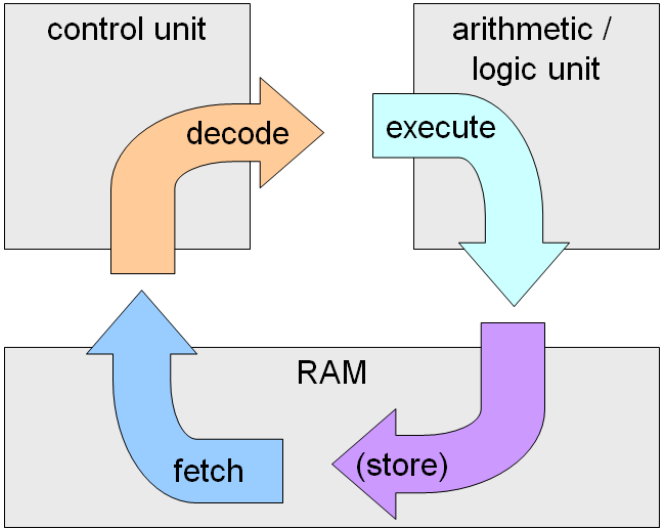


L'**Electronic Numerical Integrator and Calculator** (ENIAC) (**1946**), è considerato il primo calcolatore a valvole general-purpose programmabile progettato da Mauchly & Eckert (Univ. Pennsylvania).
Era programmato tramite inserimento di cavi e azionamento di interruttori

Il processo di esecuzione delle istruzioni

Una istruzione ha un ciclo di vita che consta di quattro fasi principali

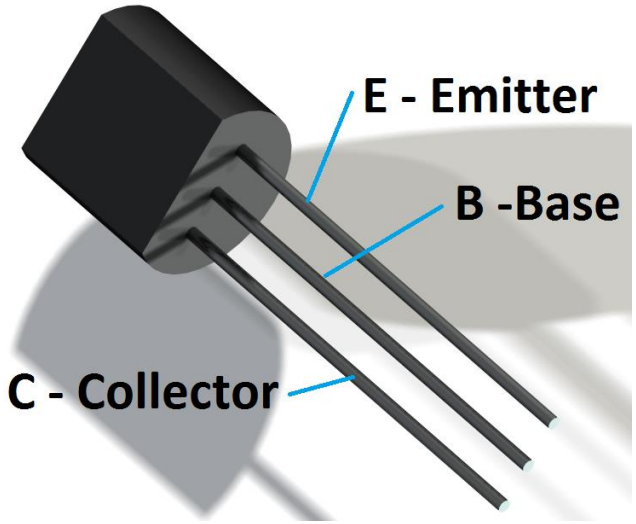


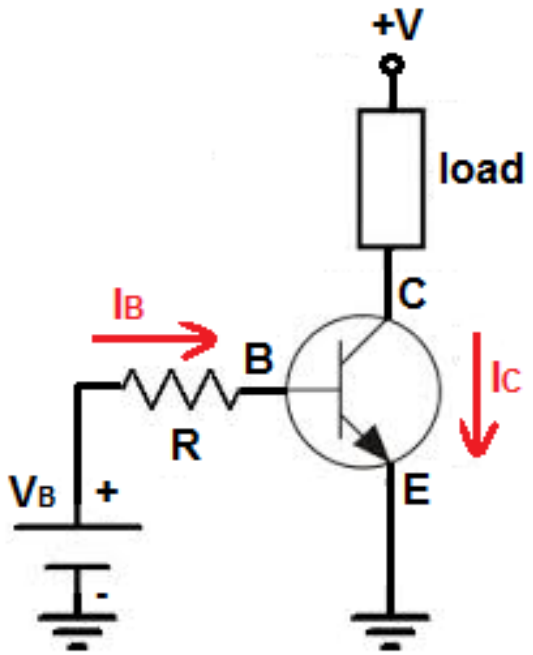


Il transistor

I computer sono una complessa sintesi di apparecchiature che operano a velocità molto elevate

Un moderno microprocessore è costituito da diversi milioni di transistor, ciascuno dei quali può cambiare il suo stato centinaia di milioni di volte in un secondo

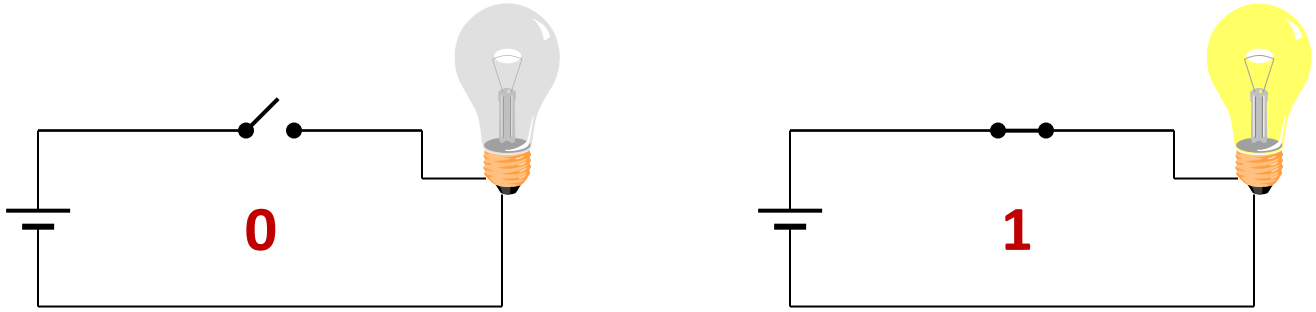




In an NPN transistor, positive voltage is given to the collector terminal and current flows from the collector to the emitter, given there is sufficient base current

Il bit

La tensione all'uscita di un transistor può assumere due stati tensione alta (1) o tensione bassa (0)



La cifra binaria è detta **bit**, dall'unione di due elisioni:

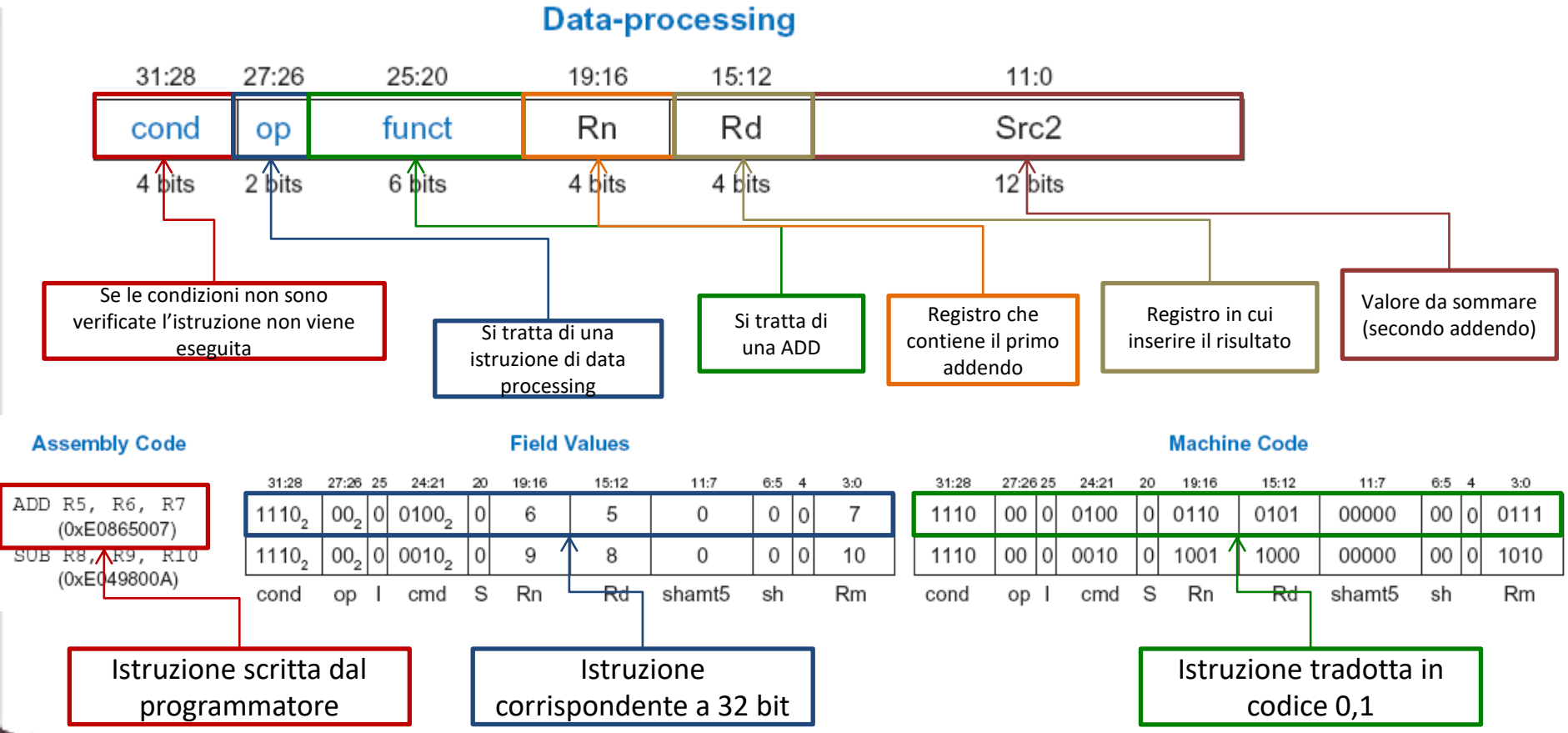
binary dig**it**

I bit estremi di un numero binario si chiamano:

MSB *(Most Significant Bit)* → **1** 0 1 1 1 0 0 1 0 **0** ← LSB *(Least Significant Bit)*

Rappresentazione dei dati

Nel processore ARM una istruzione per l'elaborazione dei dati ha la forma:



I sistemi numerali posizionali

Sistemi di numerazione **posizionali**:

La **base** del sistema di numerazione
Le **cifre** del sistema di numerazione

Il numero è scritto specificando le cifre in ordine ed il suo valore dipende dalla **posizione relativa** delle cifre

Esempio: Il sistema decimale (Base 10)

Cifre : 0 1 2 3 4 5 6 7 8 9

$$5641 = 5\cdot10^3 + 6\cdot10^2 + 4\cdot10^1 + 1\cdot10^0$$

Posizione: 3 2 1 0

Sistemi a base B

La base definisce il numero di cifre diverse nel sistema di numerazione

La cifra di minor valore è sempre lo 0; le altre sono, nell'ordine, 1,2,…,B–1; se B>10 occorre introdurre B–10 simboli in aggiunta alle cifre decimali

Un numero **intero** N si rappresenta con la scrittura $(c_n c_{n-1} \ldots c_2 c_1 c_0)_B$

$$N = c_n B^n + c_{n-1} B^{n-1} + \ldots + c_2 B^2 + c_1 B^1 + c_0 B^0$$

c_n è la **cifra più significativa**, c_0 la **meno significativa**

Un numero **frazionario** N' si rappresenta come $(0,c_1 c_2 \ldots c_n)_B$

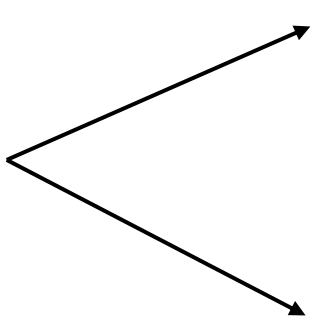
$$N' = c_1 B^{-1} + c_2 B^{-2} + \ldots + c_n B^{-n}$$

Sistemi a base B

Con n cifre in base B si rappresentano tutti i numeri interi positivi da 0 a B^n-1 (B^n numeri distinti)

Esempio: base 10

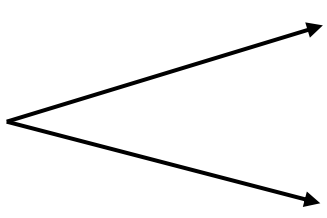
2 cifre: da 0 a $10^2-1 = 99$



$10^2 = 100$ valori

Esempio: base 2

2 cifre: da 0 a $2^2-1 = 3$



$2^2 = 4$ valori

Bit e informazione

Con **n** bit si rappresentano 2^n numeri;

ad esempio con 4 bit:

0000	...	0	1000	...	8
0001	...	1	1001	...	9
0010	...	2	1010	...	10
0011	...	3	1011	...	11
0100	...	4	1100	...	12
0101	...	5	1101	...	13
0110	...	6	1110	...	14
0111	...	7	1111	...	15

Prime 16 potenze di 2:

2^0	...	1	2^9	...	512
2^1	...	2	2^{10}	...	1024
2^2	...	4	2^{11}	...	2048
2^3	...	8	2^{12}	...	4096
2^4	...	16	2^{13}	...	8192
2^5	...	32	2^{14}	...	16384
2^6	...	64	2^{15}	...	32768
2^7	...	128	2^{16}	...	65536
2^8	...	256			

Il sistema Binario (B=2)

La base 2 è la più piccola per un sistema di numerazione

Cifre: 0 1 – **bit** (binary digit)

Esempi:

Forma polinomia

$$(101101)_2 = 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 =$$
$$32 + 0 + 8 + 4 + 0 + 1 = (45)_{10}$$

$$(0,0101)_2 = 0\cdot2^{-1} + 1\cdot2^{-2} + 0\cdot2^{-3} + 1\cdot2^{-4} =$$
$$0 + 0,25 + 0 + 0,0625 = (0,3125)_{10}$$

$$(11,101)_2 = 1\cdot2^1 + 1\cdot2^0 + 1\cdot2^{-1} + 0\cdot2^{-2} + 1\cdot2^{-3} =$$
$$2 + 1 + 0,5 + 0 + 0,125 = (3,625)_{10}$$

Dal bit al byte

Un **byte** è un insieme di 8 bit (un numero binario a 8 cifre)

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

Con un byte si rappresentano i numeri interi fra 0 e $2^8 - 1 = 255$

00000000
00000001
00000010
00000011
……
11111110
11111111

$2^8 = 256$ valori distinti

È l'elemento base con cui si rappresentano i dati nei calcolatori

Si utilizzano sempre dimensioni multiple (di potenze del 2) del byte:
2 byte (16 bit), 4 byte (32 bit), 8 byte (64 bit)…

Dal byte al kilobyte

Potenze di 2

$2^4 = 16$
$2^8 = 256$
$2^{16} = 65536$

$2^{10} = 1024$ (K=Kilo)
$2^{20} = 1048576$ (M=Mega)
$2^{30} = 1073741824$ (G=Giga)

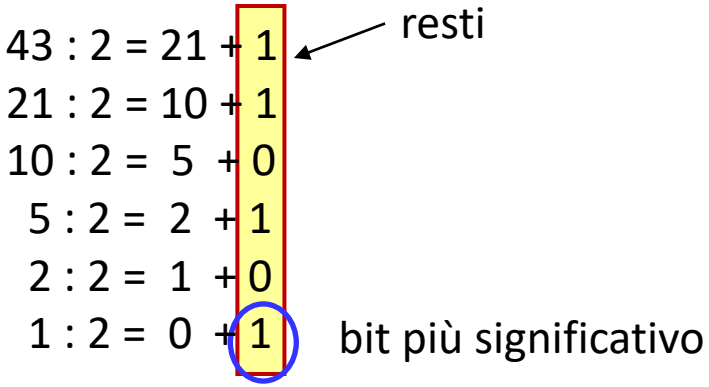
Cosa sono KB (Kilobyte), MB (Megabyte), GB (Gigabyte)?

1 KB = 2^{10} byte = 1024 byte
1 MB = 2^{20} byte = 1048576 byte
1 GB = 2^{30} byte = 1073741824 byte
1 TB = 2^{40} byte = 1099511627776 byte (Terabyte)

Da decimale a binario (numeri interi)

Si divide ripetutamente il numero **intero** decimale per 2 fino ad ottenere un quoziente nullo; le cifre del numero binario sono i resti delle divisioni; la cifra più significativa è l'ultimo resto

Esempio: convertire in binario $(43)_{10}$



$(43)_{10} = (101011)_2$

Da decimale a binario (numeri razionali)

Si moltiplica ripetutamente il numero **frazionario** decimale per 2, fino ad ottenere una parte decimale nulla o, dato che la condizione potrebbe non verificarsi mai, per un numero prefissato di volte; le cifre del numero binario sono le parti intere dei prodotti successivi; la cifra più significativa è il risultato della prima moltiplicazione

Esempio: convertire in binario $(0{,}21875)_{10}$ e $(0{,}45)_{10}$

$0{,}21875 \times 2 = 0{,}4375$
$0{,}4375 \times 2 = 0{,}875$
$0{,}875 \times 2 = 1{,}75$
$0{,}75 \times 2 = 1{,}5$
$0{,}5 \times 2 = 1{,}0$

$(0{,}21875)_{10} = (0{,}00111)_2$

$0{,}45 \times 2 = 0{,}9$
$0{,}90 \times 2 = 1{,}8$
$0{,}80 \times 2 = 1{,}6$
$0{,}60 \times 2 = 1{,}2$
$0{,}20 \times 2 = 0{,}4$ etc.

$(0{,}45)_{10} \approx (0{,}01110)_2$

Da binario a decimale

Oltre all'espansione esplicita in potenze di 2 – **forma polinomia**…

$$(101011)_2 = 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = (43)_{10}$$

…si può operare nel modo seguente: si raddoppia il bit più significativo e si aggiunge al secondo bit; si raddoppia la somma e si aggiunge al terzo bit… si continua fino al bit meno significativo

Esempio: convertire in decimale $(101011)_2$

bit più significativo

1 x 2 = 2 + 0
2 x 2 = 4 + 1
5 x 2 = 10 + 0
10 x 2 = 20 + 1
21 x 2 = 42 + 1 = 43

$$(101011)_2 = (43)_{10}$$

Sistema esadecimale

La base **16** è molto usata in campo informatico

Cifre: 0 1 2 3 4 5 6 7 8 9 A B C D E F

La corrispondenza in decimale delle cifre oltre il 9 è

$A = (10)_{10}$ $D = (13)_{10}$
$B = (11)_{10}$ $E = (14)_{10}$
$C = (12)_{10}$ $F = (15)_{10}$

Esempio:

$$(3A2F)_{16} = 3\times16^3 + 10\times16^2 + 2\times16^1 + 15\times16^0 =$$
$$3\times4096 + 10\times256 + 2\times16 + 15 = (14895)_{10}$$

Dai bit all'hex

Un numero binario di **4n** bit corrisponde a un numero esadecimale di n cifre

Esempio: 32 bit corrispondono a 8 cifre esadecimali

1101	1001	0001	1011	0100	0011	0111	1111
D	9	1	B	4	3	7	F

$(D91B437F)_{16}$

Esempio: 16 bit corrispondono a 4 cifre esadecimali

0000	0000	1111	1111
0	0	F	F

$(00FF)_{16}$

Da esadecimale a binario

La conversione da esadecimale a binario si ottiene espandendo ciascuna cifra con i **4** bit corrispondenti

Esempio: convertire in binario il numero esadecimale **0x**0c8f

Notazione usata in molti linguaggi di programmazione (es. C e Java) per rappresentare numeri esadecimali

0	c	8	f
0000	1100	1000	1111

Il numero binario ha 4x4 =16 bit

Limiti della rappresentazione

- Quando scriviamo sulla carta non ci preoccupiamo quasi mai della **grandezza dei numeri** (a meno di particolari necessità).

- Nelle macchine numeriche un numero deve essere rappresentato in un particolare dispositivo elettronico interno che si chiama **registro** ed è paragonabile ad una cella di memoria.

- Caratteristica fondamentale di questo dispositivo è la sua **dimensione** (numero di bit) stabilita in sede di progetto: ovvero in un elaboratore potremo rappresentare solo una quantità limitata di numeri.

Limiti della rappresentazione

- Ad esempio se il nostro contenitore (registro) è lungo 5 bit:

potremo rappresentare solamente i numeri binari compresi tra **0**

e **31**

Numeri interi con segno

main.c

```c
#include <stdio.h>

int main()
{
    char a=127; // char numero intero a 8 bit (byte)

    printf("Prima -> il valore di a è: %d\n", a);

    a = a + 1;

    printf("Dopo -> il valore di a è: %d\n", a);

    return 0;
}
```

Esecuzione

```
Prima -> il valore di a è: 127
Dopo -> il valore di a è: -128


...Program finished with exit code 0
Press ENTER to exit console.
```

Come è possibile?!

Numeri interi positivi

I numeri interi positivi sono rappresentati all'interno dell'elaboratore utilizzando un multiplo del byte (generalmente 4/8 byte)

Se l'intero si rappresenta con un numero di cifre minore, vengono aggiunti degli zeri nelle cifre più significative

Esempio: 12 viene rappresentato in un byte come…

00001100

Numeri con segno

Per rappresentare numeri con segno, occorre utilizzare un bit per definire il segno del numero

Si possono usare tre tecniche di codifica

- Modulo e segno
- Complemento a 2
- Complemento a 1

Modulo e segno

Il bit più significativo rappresenta il segno: **0** per i numeri positivi, **1** per quelli negativi

Esiste uno zero positivo (00…0) e uno zero negativo (10…0)

Se si utilizzano **n** bit si rappresentano tutti i numeri compresi fra $-(2^{n-1}-1)$ e $+2^{n-1}-1$

Esempio: con 4 bit si rappresentano i numeri fra -7 ($-(2^3-1)$) e $+7$ (2^3-1)

positivi		negativi	
0000	+0	1000	−0
0001	+1	1001	−1
0010	+2	1010	−2
0011	+3	1011	−3
0100	+4	1100	−4
0101	+5	1101	−5
0110	+6	1110	−6
0111	+7	1111	−7

Complemento a 1

- Considerando numeri binari di **n** bit, si definisce **complemento a uno** di un numero A la quantità:

$$\overline{A} = 2^n - 1 - A$$

- Viene anche detto semplicemente **complemento**.

- Regola pratica:
  - il complemento a uno di un numero binario A si ottiene cambiando il valore di tutti i suoi bit (**complementando** ogni bit)
  - Esempio:

$$A = 1011 \rightarrow \overline{A} = 0100$$

Complemento a 2

Il **complemento a 2** di un numero binario $(A)_2$ a **n** cifre è il numero

$$2^n - (A)_2 = 10\underbrace{\ldots\ldots}_{n}0 - (A)_2$$

Il complemento a 2 di un numero si calcola sommando 1 al suo complemento a 1

$A = 1011 \rightarrow \overline{A} = 0100 \rightarrow \overline{\overline{A}} = \overline{A} + 1 = 0101$

complemento a 1

1000	2^3	8
0111 −	$2^3 - 1$	7
0011	A	3
0100	$2^3 - 1 - A$	4

0101	$2^3 - 1 - A + 1$	5

Oppure: **a partire da destra, lasciando invariate tutte le cifre fino al primo 1 compreso, quindi invertendo il valore delle rimanenti**

A = 1 0 1 | 1; A = 0 1 0 | 1

Interi in complemento a 2

I **numeri positivi** sono rappresentati (come) in modulo e segno

I **numeri negativi** sono rappresentati in complemento a 2 $\Rightarrow$ la cifra più significativa ha sempre valore 1

Lo **zero** è rappresentato come numero positivo (con una sequenza di n zeri)

Il campo dei numeri rappresentabili varia da -2^{n-1} a $+2^{n-1}-1$

Esempio: numeri a 4 cifre

0000	+0	1000	−8
0001	+1	1001	−7
0010	+2	1010	−6
0011	+3	1011	−5
0100	+4	1100	−4
0101	+5	1101	−3
0110	+6	1110	−2
0111	+7	1111	−1

Nota: 0111 +7
1000 −8

Numeri interi con segno

main.c

```c
#include <stdio.h>

int main()
{
    char a=127; // char numero intero a 8 bit (byte)

    printf("Prima -> il valore di a è: %d\n", a);

    a = a + 1;

    printf("Dopo -> il valore di a è: %d\n", a);

    return 0;
}
```

Esecuzione

```
Prima -> il valore di a è: 127
Dopo -> il valore di a è: -128


...Program finished with exit code 0
Press ENTER to exit console.
```

a = 127 = 01111111 +

00000001

10000000 = -128 -> a

Addizione binaria

Le regole per l'addizione di due bit sono

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ con riporto di 1

L'ultima regola è... $(1)_2+(1)_2 = (10)_2$ ... $(1+1=2)_{10}$!!

Esempio

```
1  11  1        riporti
01011011+
01011010
--------
10110101
```

```
 91+
 90
----
181
```

Addizione binaria in complemento a 2

Sono utili perché l'operazione di somma algebrica può essere realizzata non curandosi del bit di segno

In complemento a 1 (più semplice da calcolare)…

- Zero ha due rappresentazioni: 00000000 e 11111111
- La somma bit a bit funziona "quasi sempre"

00110 +	(6)
10101 =	(−10)
11011	(−4)

11001 +	(−6)
11010 =	(−5)
10011	(−12)

In complemento a 2…

- Zero ha una sola rappresentazione
- La somma bit a bit funziona sempre

Somma e sottrazione in complemento a 2

- La somma si effettua direttamente, senza badare ai segni degli operandi, come fossero due normali numeri binari.

- La sottrazione si effettua sommando al minuendo il complemento a 2 del sottraendo:

$$A - B \rightarrow A + (-B) \quad \text{ovvero:} \quad A + \overline{\overline{B}}$$

- Esempio:

0 1 0 1 0 +	10 +
1 0 1 0 0 =	- 12 =
1 1 1 1 0	- 2

L'overflow

L'**overflow** si ha quando il risultato di un'operazione non è rappresentabile correttamente con n bit

Esempio: 5 bit → [−16,+15]

14 +	01110 +	−8 +	11000 +
10	01010	−10	10110
24	11000 → −8	−18	101110 → +14

Per evitare l'overflow occorre aumentare il numero di bit utilizzati per rappresentare gli operandi

Punteggio nei vecchi videogame… sorpresa per i campioni!

0111 1111 1111 1111 + 1 = 1000 0000 0000 0000

32767 + 1 = −32768

Numeri con segno

1. Operandi con segno discorde:
   - non si può mai verificare overflow!!!!!

2. Operandi con segno concorde:
   - c'è overflow quando il risultato ha segno discorde da quello dei due operandi

3. In ogni caso, si trascura sempre il **carry** (riporto) oltre il **MSB**

Esempi:

0 1 0 1 +	1 1 1 0 +
0 1 0 0 =	1 1 0 1 =
1 0 0 1	~~1~~ 1 0 1 1 =
overflow!	**carry, risultato OK**

Numeri con la virgola (fixed point)

- Si usa un numero fisso di bit per la parte intera e per quella frazionaria (e non si rappresenta la virgola!)

- Ad esempio (4 + 4 bit, binario puro):

15.9375 = 11111111

0.0625 = 00000001

Numeri con la virgola (fixed point)

- Vantaggi:
  - gli operandi sono **allineati** per cui le operazioni aritmetiche risultano facili ed immediate;
  - la **precisione** assoluta è fissa
- Svantaggi:
  - l'intervallo di valori rappresentati è assai **modesto**
  - la precisione dei numeri frazionari rappresentati molto **scarsa**
- Utilizzo tipico:
  - DSP (Digital Signal Processor)
  - Sistemi digitali per applicazioni specifiche (*special-purpose)*
  - Numeri interi nei calcolatori

Numeri interi (fixed point)

- A causa dell'estrema semplicità che presentano le operazioni aritmetiche in **complemento a 2**, in tutte le macchine numeriche i **numeri interi** vengono rappresentati in questo codice.

- Il numero di bit utilizzati dipende dalla macchina: si tratta generalmente di **16** bit (interi corti) o **32** bit (interi lunghi).

- La rappresentazione è nota col nome di **fixed-point** e il punto frazionario è supposto all'estrema destra della sequenza di bit (parte frazionaria nulla).

Numeri reali

- Le rappresentazioni fin qui considerate hanno il pregio di rappresentare esattamente i numeri (almeno quelli interi) ma richiedono un numero di bit esorbitante quando il numero da rappresentare ha valore elevato.

- La rappresentazione dei numeri frazionari che deriva dai codici precedenti, ovvero in **fixed point**, a causa delle forti approssimazioni che impone è usata raramente.

- Generalmente viene utilizzato un apposito codice noto come **floating point** che consente di rappresentare in un numero limitato di bit grandezze di qualsiasi valore anche se condizionate da approssimazioni più o meno elevate.

Floating point

- E’ basata sul formato esponenziale (*notazione scientifica*)

$$N = \text{mantissa} \cdot \text{base}^{\text{esponente}}$$

  - Ricorda le notazioni:

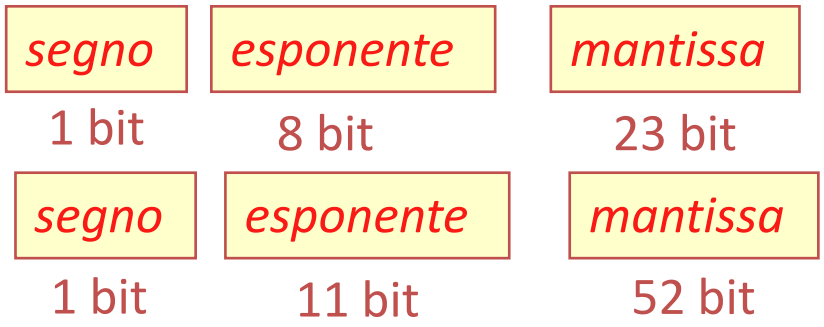
standard	3.5×10^4	3.5E+4
scientifico	0.35×10^5	0.35E+5

- Nei sistemi di elaborazione
  - Base = 2
  - Mantissa ed esponente sono rappresentati in binario

Floating point

- Vantaggi:
  - grande intervallo di valori rappresentabili
  - errore relativo **fisso**
- Svantaggi:
  - operandi **non allineati** per cui le operazioni aritmetiche risultano molto complesse
  - errore assoluto variabile e dipendente dal valore del numero

- E' la rappresentazione (IEEE-P754) utilizzata da tutti i calcolatori elettronici per rappresentare i numeri frazionari ed è stata standardizzata dall'IEEE.



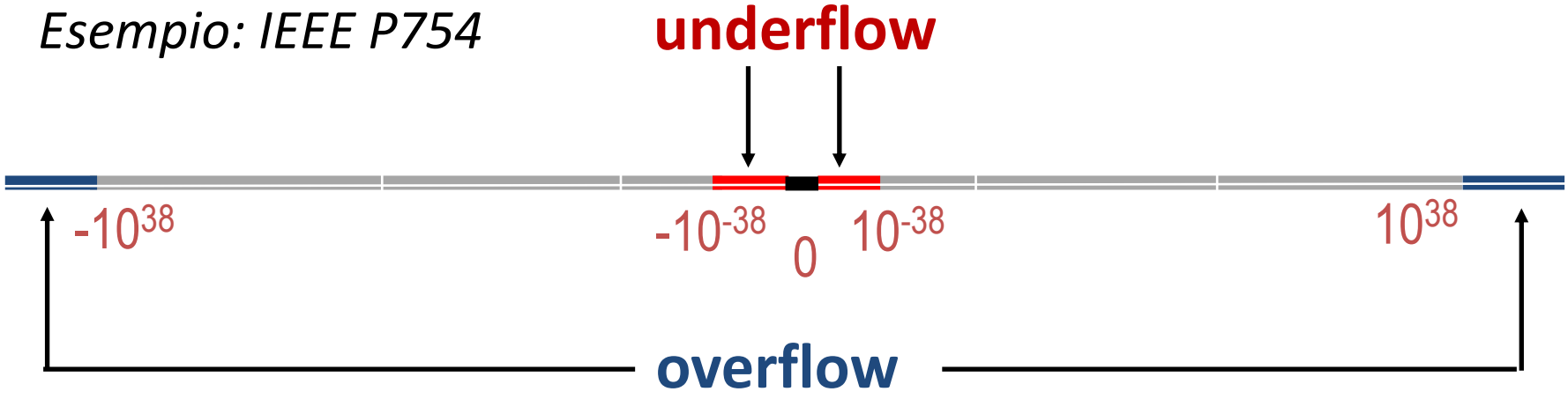
Singola precisione: circa 7 cifre decimali

Doppia precisione: circa 17 cifre decimali

Floating point

- A causa della precisione variabile è possibile avere errori di rappresentazione:
  - numeri troppo grandi: **overflow**
  - numeri troppo piccoli: **underflow**

Esempio: IEEE P754



L’operazione di shift

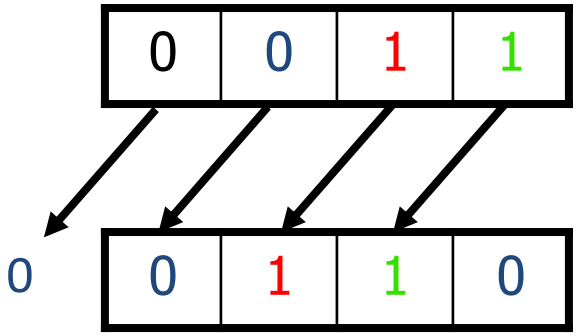
- Equivale ad una moltiplicazione o divisione per la **base**.
- Consiste nel “*far scorrere* ” i bit (a sinistra o a destra) inserendo opportuni valori nei posti lasciati liberi.
- In decimale equivale a moltiplicare (shift a sinistra) o dividere (shift a destra) per **10**.
- In binario equivale a moltiplicare (shift a sinistra) o dividere (shift a destra) per **2**.

L’operazione di shift a sinistra

- Si inserisce come **LSB** un bit a zero

- Equivale ad una **moltiplicazione per due**

0011 « 1 = 0110 ($3 \times 2 = 6$)
0011 « 2 = 1100 ($3 \times 2^2 = 12$)
0011 « 3 = 11000 ($3 \times 2^3 = 24$)



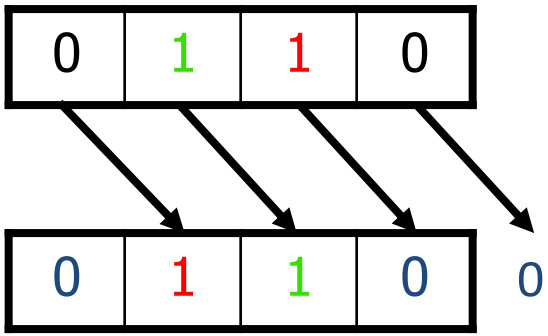
« 1 (shift a sinistra di 1 posizione)

L’operazione di shift a destra

- Si inserisce come **MSB** un bit a zero
- Equivale ad una **divisione** per due

0110 » 1 = 0011 (6 : 2 = 3)

0110 » 2 = 0001 (6 : 4 = 1) troncamento!



» 1 (shift a destra di 1 posizione)

L’aritmetica degli elaboratori

- L’aritmetica **interna** degli elaboratori differisce notevolmente dall’aritmetica classica
- Sebbene le stesse operazioni possano essere realizzate secondo modalità diverse su elaboratori diversi, si riscontrano alcune caratteristiche comuni:

  - Rappresentazione binaria dei numeri
  - Rango finito dei numeri rappresentabili
  - Precisione finita dei numeri
  - Operazioni espresse in termini di operazioni più semplici

Rango finito dei numeri rappresentabili

- Qualunque sia la codifica utilizzata, esistono sempre il più grande ed il più piccolo numero rappresentabile
- I limiti inferiore e superiore del rango di rappresentazione dipendono sia dal tipo di codifica, sia dal numero di bit utilizzati
- Se il risultato di un’operazione non appartiene al rango dei numeri rappresentabili, si dice che si è verificato un overflow (un **underflow**, più precisamente, se il risultato è più piccolo del più piccolo numero rappresentabile)

Precisione finita dei numeri

La **precisione** della rappresentazione di un numero frazionario è una misura di quanto essa corrisponda al numero che deve essere rappresentato

Negli elaboratori, i numeri frazionari sono rappresentati in virgola mobile (**floating–point**), utilizzando un numero finito di bit

È plausibile che un numero reale non ammetta una rappresentazione finita, quindi dovrà essere codificato in maniera approssimata

Negli elaboratori si rappresentano soltanto numeri razionali (fino ad una data precisione)

L’aritmetica degli elaboratori

Operazioni espresse in termini di operazioni più semplici

La maggior parte degli elaboratori non possiede circuiti in grado di eseguire direttamente tutte le operazioni:

- La sottrazione si realizza per mezzo di una complementazione e di un’addizione
- La moltiplicazione si realizza per mezzo di una successione di addizioni e di **shift** (traslazioni)
- La divisione si realizza per mezzo di una successione di shift e sottrazioni

Le operazioni più semplici sono eseguite direttamente da appositi circuiti (in **hardware**); le operazioni più complesse sono realizzate mediante l’esecuzione di successioni di operazioni più semplici, sotto il controllo di programmi appositamente realizzati, e generalmente memorizzati permanentemente (in **firmware**)

Codifica dei caratteri alfabetici

- Oltre ai numeri, molte applicazioni informatiche elaborano caratteri (simboli)
- Gli elaboratori elettronici trattano numeri
- Si codificano i caratteri e i simboli per mezzo di numeri
- Per poter scambiare dati (testi) in modo corretto, occorre definire uno standard di codifica

A	→	01000001
3	→	00110011
$	→	00100100

Codifica dei caratteri alfabetici

- Quando si scambiano dati, deve essere noto il tipo di codifica utilizzato
- La codifica deve prevedere le lettere dell'alfabeto, le cifre numeriche, i simboli, la punteggiatura, i caratteri speciali per certe lingue (æ, ã, ë, è,…)
- Lo standard di codifica più diffuso è il **codice ASCII**, per **American Standard Code for Information Interchange**

Codifica ASCII

- Definisce una tabella di corrispondenza fra ciascun carattere e un codice a **7 bit** (128 caratteri)
- I caratteri, in genere, sono rappresentati con **1 byte** (8 bit); i caratteri con il bit più significativo a 1 (quelli con codice dal 128 al 255) rappresentano un'estensione della codifica
- La tabella comprende sia **caratteri di controllo** (codici da 0 a 31) che **caratteri stampabili**
- I caratteri alfabetici/numerici hanno codici ordinati secondo l'ordine alfabetico/numerico

0 48	A 65	a 97
1 49	B 66	b 98
…….	…….	…….
8 56	Y 89	y 121
9 57	Z 90	z 122
cifre	maiuscole	minuscole

Caratteri di controllo ASCII

- I caratteri di controllo (codice da 0 a 31) hanno funzioni speciali
- Si ottengono o con tasti specifici o con una sequenza **Ctrl+carattere**

Ctrl	Dec	Hex	Code	Nota
^@	0	0	NULL	carattere nullo
^A	1	1	SOH	partenza blocco
……	…	…	……	……
^G	7	7	BEL	beep
^H	8	8	BS	backspace
^I	9	9	HT	tabulazione orizzontale
^J	10	A	LF	line feed (cambio linea)
^K	11	B	VT	tabulazione verticale
^L	12	C	FF	form feed (alim. carta)
^M	13	D	CR	carriage return (a capo)
……	…	…	……	……
^Z	26	1A	EOF	fine file
^[	27	1 B	ESC	escape
……	…	…	……	……
^_	31	1F	US	separatore di unità

Caratteri ACII stampabili

Dec	Hx	Chr	Dec	Hx	Chr	Dec	Hx	Chr	Dec	Hx	Chr	Dec	Hx	Chr	Dec	Hx	Chr
32	20	SPACE	48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
34	22	”	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
39	27	’	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

Nota: il valore numerico di una cifra può essere calcolato come differenza del suo codice ASCII rispetto al codice ASCII della cifra 0 (es. ‘5’–‘0’ = 53–48 = 5)

Tabella ASCII estesa

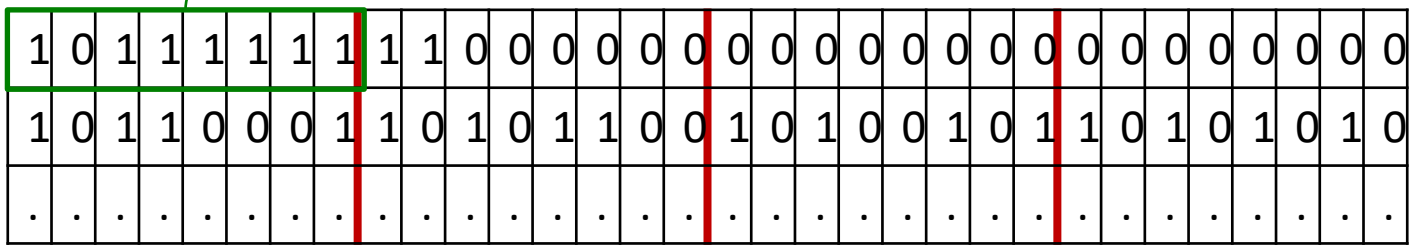
- I codici oltre il 127 non sono compresi nello standard originario

128	Ç	144	É	160	á	176	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	161	í	177	▒	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	162	ó	178	▓	195	├	211	╙	227	π	243	≤
131	â	147	ô	163	ú	179	│	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	164	ñ	180	┤	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	165	Ñ	181	╡	198	╞	214	╓	230	µ	246	÷
134	å	150	û	166	ª	182	╢	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	167	º	183	╖	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	168	¿	184	╕	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	169	_	185	╣	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	170	¬	186	║	203	╦	219	▉	235	δ	251	√
139	ï	156	£	171	½	187	╗	204	╠	220	▄	236	∞	252	_
140	î	157	¥	172	¼	188	╝	205	═	221	▌	237	ϕ	253	²
141	ì	158	_	173	¡	189	╜	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	174	«	190	╛	207	╧	223	▀	239	∩	255	
143	Å	192	└	175	»	191	┐	208	╨	224	α	240	≡		

Dati in memoria

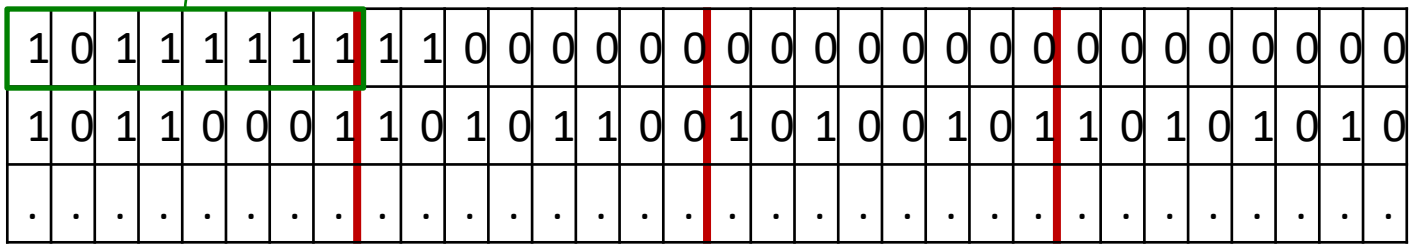
A : 1 byte intero senza segno

191 In esadecimale: **BF**



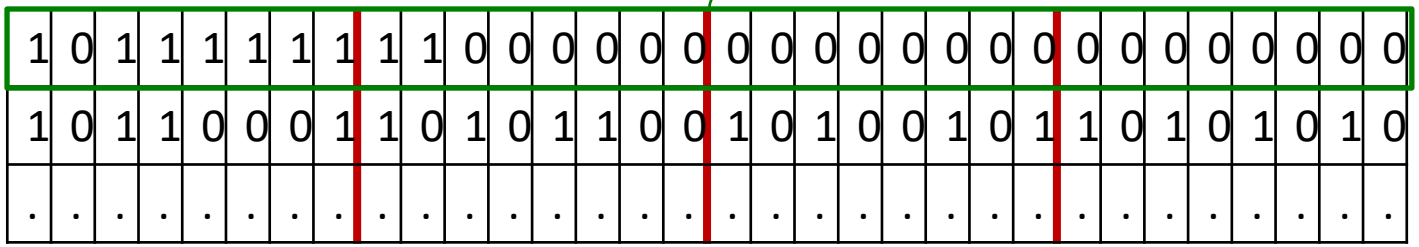
A : 1 byte intero con segno

-65



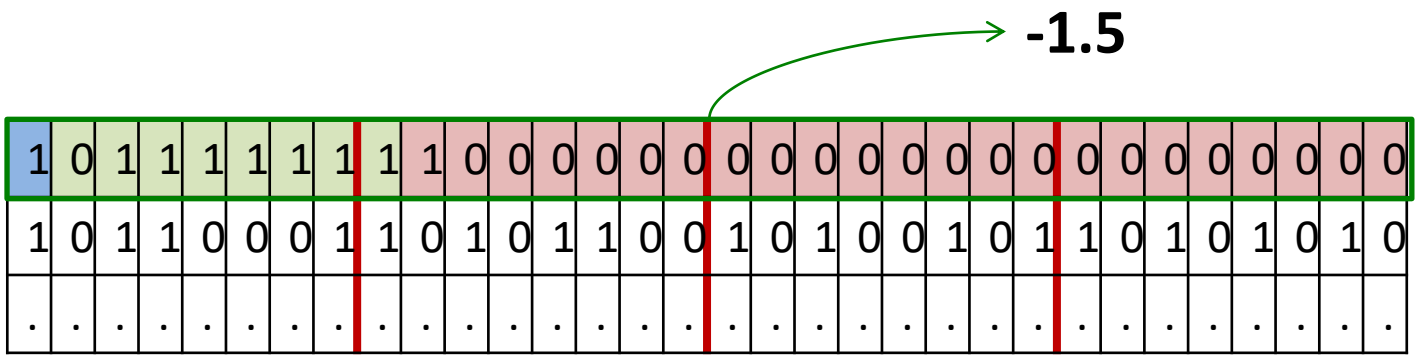
A : 4 byte razionale con segno

-1.5



Dati in memoria

A : 4 byte
intero
con segno



	Sign	Exponent	Mantissa
Value:	-1	2^0	1.5
Encoded as:	1	127	4194304
Binary:	☑	☐ ☑ ☑ ☑ ☑ ☑ ☑ ☑	☑ ☐

Decimal representation: -1.5

Value actually stored in float: -1.5

+1