Fondamenti di Informatica

Ingegneria Clinica

Lezione 19/11/2009

Prof. Raffaele Nicolussi

FUB - Fondazione Ugo Bordoni

Via B. Castiglione 59 - 00142 Roma

Docente	**Raffaele Nicolussi**	*rnicolussi@fub.it* *0654803323*
Lezioni Aula 54 (ex aula 4) Via del Castro Laurenziano, 7	**Lunedì, Giovedì, Venerdì**	**12:00 – 13:30**
Esercitazioni Aula 15 Via Tiburtina, 205	**Giovedì**	**14:00 – 16.30**
Ricevimento:	**Per appuntamento**	**in FUB, per email, per telefono**
Sito web:	**http://w3.uniroma1.it/IngClinFondinf**	

Rappresentazione dei caratteri

- Il byte è stato definito come un gruppo di bit **sufficienti a rappresentare un carattere**.
- I caratteri dell'alfabeto inglese sono 26, quindi **come minimo** devo essere in grado di rappresentare 26 codici distinti.
- Per fare questo non basterebbero 4 bit ($2^4 = 16$)
- 5 bit andrebbero bene, infatti : $2^5 = 32 > 26$

Rappresentazione dei caratteri

- In realtà:
  - rappresentare solo i caratteri dell'alfabeto è limitativo.
  - vorrei poter rappresentare anche le cifre.
  - vorrei poter rappresentare anche la punteggiatura.
  - vorrei poter rappresentare anche qualche simbolo (p.es. aritmetici).
  - vorrei poter aggiungere "caratteri" speciali che marchino p.es. la fine di una riga, l'inizio e la fine di una trasmissione, e svolgano altre funzioni di controllo.
  - vorrei anche distinguere le 26 lettere in maiuscolo dalle stesse in minuscolo.

Rappresentazione dei caratteri (1)

- Codice ASCII (American Standard Code for Information Interchange) usa 1 byte (8 bit) per rappresentare 128 caratteri
- Esempi: 'A' = 01000001, 'a' = 01100001
- Un byte ASCII è spesso interpretato come un numero in base 2, rappresentato poi in base 8, 16 o 10
- 4 gruppi di byte:
- Caratteri di controllo non stampabili (0-31)
- Punteggiatura, spaziatura, simboli, cifre (32-64) (91-96) (123-127)
- Caratteri alfabetici maiuscoli (65-90)
- Caratteri alfabetici minuscoli (97-122)

Rappresentazione dei caratteri (2)

- In totale 128 oggetti : la codifica con 7 bit è sufficiente
  - Mancano molti caratteri di utilità (p.es., lettere accentate)
  - Un byte può rappresentare fino a 256 caratteri
  - Disponibili varie *estensioni* (non standardizzate) del codice ASCII in cui vengono rappresentati 256 caratteri (128 dei quali mantengono la rappresentazione che hanno in ASCII)
- Interpretando i byte di codice ASCII come numeri

  ASCII('a') - ASCII('A') = 32
- *Esistono altri codici per la rappresentazione di caratteri*

Codice ASCII

Byte	Cod.	Char
00000000	0	Null
00000001	1	Start of heading
00000010	2	Start of text
00000011	3	End of text
00000100	4	End of transmit
00000101	5	Enquiry
00000110	6	Acknowledge
00000111	7	Audible bell
00001000	8	Backspace
00001001	9	Horizontal tab
00001010	10	Line feed
00001011	11	Vertical tab
00001100	12	Form Feed
00001101	13	Carriage return
00001110	14	Shift out
00001111	15	Shift in
00010000	16	Data link escape
00010001	17	Device control 1
00010010	18	Device control 2
00010011	19	Device control 3
00010100	20	Device control 4
00010101	21	Neg. acknowledge
00010110	22	Synchronous idle
00010111	23	End trans. block
00011000	24	Cancel
00011001	25	End of medium
00011010	26	Substitution
00011011	27	Escape
00011100	28	File separator
00011101	29	Group separator
00011110	30	Record Separator
00011111	31	Unit separator

Byte	Cod.	Char
00100000	32	Spc
00100001	33	!
00100010	34	"
00100011	35	#
00100100	36	$
00100101	37	%
00100110	38	&
00100111	39	'
00101000	40	(
00101001	41	)
00101010	42	*
00101011	43	+
00101100	44	,
00101101	45	-
00101110	46	.
00101111	47	/
00110000	48	0
00110001	49	1
00110010	50	2
00110011	51	3
00110100	52	4
00110101	53	5
00110110	54	6
00110111	55	7
00111000	56	8
00111001	57	9
00111010	58	:
00111011	59	;
00111100	60	<
00111101	61	=
00111110	62	>
00111111	63	?

Byte	Cod.	Char
01000000	64	@
01000001	65	A
01000010	66	B
01000011	67	C
01000100	68	D
01000101	69	E
01000110	70	F
01000111	71	G
01001000	72	H
01001001	73	I
01001010	74	J
01001011	75	K
01001100	76	L
01001101	77	M
01001110	78	N
01001111	79	O
01010000	80	P
01010001	81	Q
01010010	82	R
01010011	83	S
01010100	84	T
01010101	85	U
01010110	86	V
01010111	87	W
01011000	88	X
01011001	89	Y
01011010	90	Z
01011011	91	[
01011100	92	\
01011101	93	]
01011110	94	^
01011111	95	_

Byte	Cod.	Char
01100000	96	`
01100001	97	a
01100010	98	b
01100011	99	c
01100100	100	d
01100101	101	e
01100110	102	f
01100111	103	g
01101000	104	h
01101001	105	i
01101010	106	j
01101011	107	k
01101100	108	l
01101101	109	m
01101110	110	n
01101111	111	o
01110000	112	p
01110001	113	q
01110010	114	r
01110011	115	s
01110100	116	t
01110101	117	u
01110110	118	v
01110111	119	w
01111000	120	x
01111001	121	y
01111010	122	z
01111011	123	{
01111100	124	|
01111101	125	}
01111110	126	~
01111111	127	Del

Un'estensione del codice ASCII

Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char
10000000	128	Ç	10100000	160	á	11000000	192	+	11100000	224	Ó
10000001	129	ü	10100001	161	í	11000001	193	-	11100001	225	ß
10000010	130	é	10100010	162	ó	11000010	194	-	11100010	226	Ô
10000011	131	â	10100011	163	ú	11000011	195	+	11100011	227	Ò
10000100	132	ä	10100100	164	ñ	11000100	196	-	11100100	228	õ
10000101	133	à	10100101	165	Ñ	11000101	197	+	11100101	229	Õ
10000110	134	å	10100110	166	ª	11000110	198	ã	11100110	230	µ
10000111	135	ç	10100111	167	º	11000111	199	Ã	11100111	231	þ
10001000	136	ê	10101000	168	¿	11001000	200	+	11101000	232	Þ
10001001	137	ë	10101001	169	®	11001001	201	+	11101001	233	Ú
10001010	138	è	10101010	170	¬	11001010	202	-	11101010	234	Û
10001011	139	ï	10101011	171	½	11001011	203	-	11101011	235	Ù
10001100	140	î	10101100	172	¼	11001100	204	¦	11101100	236	ý
10001101	141	ì	10101101	173	¡	11001101	205	-	11101101	237	Ý
10001110	142	Ä	10101110	174	«	11001110	206	+	11101110	238	¯
10001111	143	Å	10101111	175	»	11001111	207	¤	11101111	239	´
10010000	144	É	10110000	176	¯	11010000	208	ð	11110000	240	-
10010001	145	æ	10110001	177	¯	11010001	209	Ð	11110001	241	±
10010010	146	Æ	10110010	178	¯	11010010	210	Ê	11110010	242	‗
10010011	147	ô	10110011	179	¦	11010011	211	Ë	11110011	243	¾
10010100	148	ö	10110100	180	¦	11010100	212	È	11110100	244	¶
10010101	149	ò	10110101	181	Á	11010101	213	i	11110101	245	§
10010110	150	û	10110110	182	Â	11010110	214	Í	11110110	246	÷
10010111	151	ù	10110111	183	À	11010111	215	Î	11110111	247	¸
10011000	152	ÿ	10111000	184	©	11011000	216	Ï	11111000	248	º
10011001	153	Ö	10111001	185	¦	11011001	217	+	11111001	249	¨
10011010	154	Ü	10111010	186	¦	11011010	218	+	11111010	250	·
10011011	155	ø	10111011	187	+	11011011	219	¯	11111011	251	¹
10011100	156	£	10111100	188	+	11011100	220	¯	11111100	252	³
10011101	157	Ø	10111101	189	¢	11011101	221	¦	11111101	253	²
10011110	158	×	10111110	190	¥	11011110	222	Ì	11111110	254	¯
10011111	159	ƒ	10111111	191	+	11011111	223	¯	11111111	255	

Tipo char in C (1)

- L'insieme dei valori che una variabile di tipo char può assumere è un intervallo di interi (estremi dipendono dall'*implementazione* e sono definiti dalle costanti CHAR_MIN e CHAR_MAX nel file "limits.h")
- Questo intervallo contiene gli interi corrispondenti al codice utilizzato per la codifica dei caratteri (*tipicamente*, ASCII)
- Un valore di tipo char è *tipicamente* memorizzato in un byte (in questo caso, l'intervallo di interi è [-128, 127] o [0, 255])

Tipo char in C (2)

Esempi

- char z = 'a' ; dichiara z come char assegnandogli il codice di 'a' (97, se ASCII)
- char beta = 97 ; dichiara beta come char assegnandogli il valore 97 ('a' , se ASCII)
- char cc = '\n' ; dichiara cc come char assegnandogli il codice di newline ('\n' è un carattere)
- *Specifica di conversione* %c in scanf e printf
- Operazioni: un valore di tipo char può essere usato in espressioni aritmetiche

La rappresentazione dei numeri negativi

Ci sono diverse convenzioni per la rappresentazione dei numeri negativi

Prendiamone in considerazione le 2 più usate:

1. ***modulo e segno*** in cui il primo bit viene riservato al segno (1 negativo, 0 positivo) e gli altri al modulo

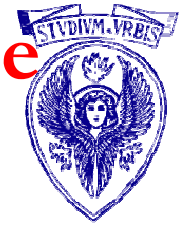
Es.: (con 8 bit)

$$2 = \mathbf{0}000\ 0010 \quad -2 = \mathbf{1}000\ 0010$$

Problemi:

- Lo zero ha due rappresentazioni
  (es.: 0000 0000 e 1000 0000)
- La somma dà risultati scorretti
  (es.: 2 + (-2) = -4)

La rappresentazione dei numeri negativi con modulo e segno



Consideriamo per semplicità solo 4 bit

In modulo e segno:

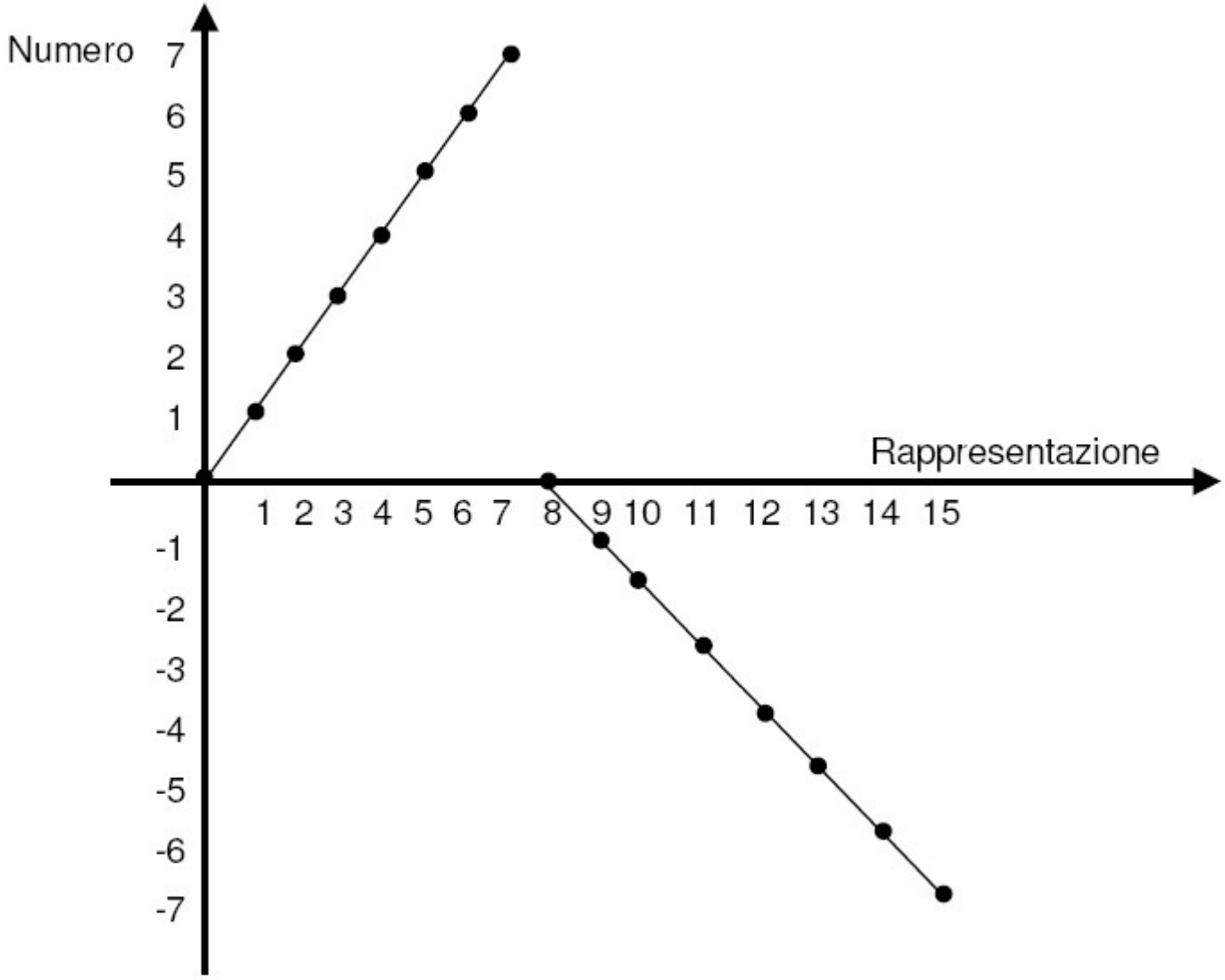
+/-	b_2	b_1	b_0

In valore assoluto:

b_3	b_2	b_1	b_0

Significato in modulo e segno	Rappresentazione binaria	Significato in valore assoluto
+7	0 111	7
+6	0 110	6
+5	0 101	5
+4	0 100	4
+3	0 011	3
+2	0 010	2
+1	0 001	1
+0	0 000	0
-0	1 000	8
-1	1 001	9
-2	1 010	10
-3	1 011	11
-4	1 100	12
-5	1 101	13
-6	1 110	14
-7	1 111	15

La rappresentazione dei numeri negativi: modulo e segno



La rappresentazione dei numeri negativi

- Addizione e sottrazione sono le operazioni di cui si deve disporre per poter realizzare qualsiasi operazione aritmetica più complessa
- Si supponga che il calcolatore abbia una "Unità Aritmetica" che realizzi indipendentemente le due operazioni.

La rappresentazione dei numeri negativi

- Di fronte ad una somma algebrica, il calcolatore dovrebbe:
  - confrontare i due segni
  - se uguali, attivare il circuito di addizione
  - se diversi, identificare il maggiore (in valore assoluto) ed attivare il circuito di sottrazione
  - completare il risultato con il segno corretto
- I passi indicati non sono eseguibili contemporaneamente perché ognuno dipende dai precedenti

La rappresentazione dei numeri negativi (2)

2. ***Complemento a 2*** : il negativo di un numero **X** si ottiene calcolando 2^n**-X**, dove **n** il numero di bit
(**in pratica, basta cambiare tutti gli 1 in 0 e viceversa e aggiungere 1**)

Es.: (con 8 bit)

X=**0000 0010** → (*cambio gli 0 con 1 e viceversa)* X_1= **1111 1101**→ *(aggiungo 1)* c(X) = **1111 1110**

- Risolve i problemi del doppio zero
- La somma dà risultati *corretti*

  (es.: (-5) + (+2) = (-3). Provare)

Complemento a due

- Il **complemento a due** (in inglese **Two's complement**) è il metodo più diffuso per la rappresentazione dei numeri relativi in informatica.
- Esso è inoltre utilizzato come **operazione di negazione** (cambiamento di segno) nei computer che usano questo metodo.
- La sua enorme diffusione è data dal fatto che i circuiti di addizione e sottrazione non devono esaminare il segno di un numero rappresentato con questo sistema per determinare quale delle due operazioni sia necessaria, permettendo tecnologie più semplici e maggiore precisione
  - Si risparmia una IF !

Complemento alla base di un numero

- Nella rappresentazione in complemento alla base *b* con *n* cifre le b^n combinazioni rappresentano numeri positivi e numeri negativi.
- In particolare:
  - le combinazioni da 0 a $b^n - 1$ rappresentano i numeri positivi, rispettando la usuale rappresentazione posizionale;
  - le combinazioni da b^n fino a $b^n - 1$ rappresentano i numeri negativi, con la seguente definizione:
- dato un numero positivo *X*, *il suo corrispondente negativo è dato* da:

$$b^n - X$$

Complemento alla base di un numero

- Nella rappresentazione in complemento alla base *b* con *n* cifre le b^n combinazioni rappresentano numeri positivi e numeri negativi.
- Dato un numero positivo *X*, *il suo corrispondente negativo è dato* da:

$$\boldsymbol{b^n - X}$$

Complemento alla base -1 di un numero

- Nella rappresentazione in complemento alla base -1 con *n cifre* le b^n combinazioni rappresentano numeri positivi e negativi.
- In particolare:
  - le combinazioni da 0 a $b^n - 1$ rappresentano i numeri positivi, rispettando la usuale rappresentazione posizionale;
  - le combinazioni da b^n fino a $b^n - 1$ rappresentano i numeri negativi, con la seguente definizione:
- dato un numero positivo *X*, *il suo corrispondente negativo è dato* da:

$$(b^n-1) - X$$

Complemento alla base

b=2, n=5

Positivi da 0 a 01111

Negativi da 10000 a 11111

X = 01011; trovo -X

1 0 0 0 0 0 -

0 1 0 1 1 =

---

1 0 1 0 1

b=2, n=7

Positivi da 0 a 0111111

Negativi da 1000000 a 1111111

X=0011000; trovo -X

1 0 0 0 0 0 0 0 -

0 0 1 1 0 0 0 =

---

1 1 0 1 0 0 0

Regola pratica (un'altra …)

1. Si parte dal bit meno significativo e si riportano invariati tutti i bit fino al primo bit a 1 compreso
2. Da questo bit in poi si complementano i rimanenti bit (0 → 1,1 → 0)

Complemento alla base -1

X=36, b=10, n=2

- in complemento alla base -1 è: 99 - 36 = 63
- Si ottiene complementando a 9 ogni singola cifra

X=01011, b=2, n=5

- *-X in complemento alla base -1 è: (25 - 1) - X = (100000 - 1) - X =*

11111 - 01011 = 10100

Si ottiene complementando ogni singolo bit (0 → 1, 1 → 0)

Riepilogo

- **Rappresentazione in complemento a 2**:
  - i numeri positivi sono rappresentati dal loro modulo e hanno il bit più significativo (segno) posto a 0.
  - i numeri negativi sono rappresentati dalla quantità che manca al numero positivo per arrivare alla base elevata al numero di cifre utilizzate, segno compreso.
    - Pertanto i numeri negativi hanno il bit del segno sempre a 1
- Metà delle configurazioni sono perciò riservate ai numeri positivi e metà ai numeri negativi
- Discorsi analoghi possono essere fatti per basi diverse da 2: in base 10 un numero è negativo se la prima cifra è >= 5, in base 8 se >= 4, in base 16 se >= 8

Rappresentazione in complemento alla base

- Con **n** bit a disposizione:
  - Il numero minimo rappresentabile è -2^n-1
  - Il numero massimo rappresentabile è $2^{n-1} - 1$
  - -1 è rappresentato da tutti 1 qualunque sia il numero di bit considerato
  - Il numero può essere interpretato considerando il bit più significativo con segno negativo

$$N = - b_3 \bullet 2^3 + b_2 \bullet 2^2 + b_1 \bullet 2^1 + b_0 \bullet 2^0$$

Utilità del complemento alla base

- Con la tecnica del complemento si può utilizzare un solo circuito per effettuare sia l'addizione che la sottrazione (per esempio il circuito dell'addizione)
- Operiamo in base 10 e vogliamo calcolare A – B.
- Si supponga di conoscere il risultato dell'operazione 10 - B (complemento a 10 di B). Allora:

$$\mathbf{A - B = A + (10 - B)}$$

a condizione che si trascuri il riporto

- Analogo discorso con k cifre purché si disponga del risultato dell'operazione 10^k – B (complemento a 10^k). Si ricordi sempre di fissare il numero di cifre
- Se operiamo in base 2, con k cifre:

$$\mathbf{A - B = A + (2^k - B)}$$

a condizione che si trascuri il riporto

- Se si utilizza la tecnica del complemento alla base -1 occorre sommare il riporto al risultato finale

La rappresentazione in virgola fissa dei numeri razionali

- Un numero razionale ha un numero finito di cifre periodiche dopo la virgola (ad esempio 3.12 oppure 3.453)
  - la rappresentazione è solitamente su 4/8 byte
  - con la rappresentazione in *virgola fissa* riservo X bit per la parte frazionaria
  - es : con 3 bit per la parte intera e 2 per quella frazionaria 011.11, 101.01

La rappresentazione in virgola fissa dei numeri razionali (2)

- Convertiamo in base 10 una rappresentazione in virgola fissa
  - es : 101.01 = $1 * 2^2 + 0 * 2^1 + 1 * 2^0 +$
    $0 * 2^{-1} + 1 * 2^{-2} = 4 + 1 + 0.25 = 5.25$

    dove $2^{-1} = 1/2 = 0.5$, $2^{-2} = 1/2^2 = 0.25$

    e in generale $2^{-n} = 1/2^n$

- Viceversa, la conversione in base 2 come quella per i numeri interi, ma con la rappresentazione:

$$c_{N-1} * 2^{N-1} + c_{N-2} * 2^{N-2} \ldots + c_1 * 2^1 + c_0 * 2^0 + c_{-1} * 2^{-1} \ldots$$

Problemi

- Con queste rappresentazioni ho DUE grossi problemi
  - Problema dell'overflow
  - Problema dello spreco di memoria

Problema dell'overflow

- Overflow:

  Se moltiplico o sommo due numeri molto elevati posso ottenere un numero che non è rappresentabile

  - es.: vediamo cosa succede *in base 10* con solo 3 cifre :

    500 + 636 = 1136 risultato 136

  se uso solo 3 cifre non ho lo spazio fisico per scrivere la prima cifra (1) che viene 'persa'.

  - Similmente per il caso in base 2.

Problema dello spreco di memoria

- Spreco dei bit per memorizzare molti '0' quando lavoro con numeri molto piccoli o molto grandi
  - es. vediamo in base 10, con 5 cifre per la parte intera e 2 cifre riservate alla parte frazionaria

10000.00 oppure 00000.02

Soluzione

- La notazione *esponenziale* o *floating point* (*virgola mobile*) risolve entrambi i problemi dell'overflow e dello spreco di memoria
  - i bit vengono usati più efficientemente.

Rappresentazione in virgola mobile

- **Idea** : quando lavoro con numeri molto piccoli uso tutti i bit disponibili per rappresentare le cifre dopo la virgola e quando lavoro con numeri molto grandi le uso tutte per rappresentare le cifre in posizioni elevate
  - questo permette di rappresentare numeri piccoli con intervalli minori fra loro rispetto ai numeri grandi

Rappresentazione in virgola mobile

- Il termine **numero in virgola mobile** (in inglese **floating point**) indica il metodo di rappresentazione dei numeri razionali (e di approssimazione dei numeri reali) e di elaborazione dei dati usati dai processori per compiere operazioni matematiche.
- Si contrappone all'aritmetica intera o in virgola fissa.
- In informatica viene usata solitamente in base 2;
  - in questo caso può essere considerata l'analogo binario della notazione scientifica in base 10.
- L'uso di operazioni aritmetiche in virgola mobile è ad oggi il metodo più diffuso per la gestione di numeri reali.

Mantissa

- Un numero in virgola mobile N, è costituito nella sua forma più semplice da due parti:
  - *un campo di* ***mantissa*** M*;*
  - *un campo di* ***esponente*** E.
- con il seguente significato

$$N = M * B^E \qquad \text{(B base)}$$

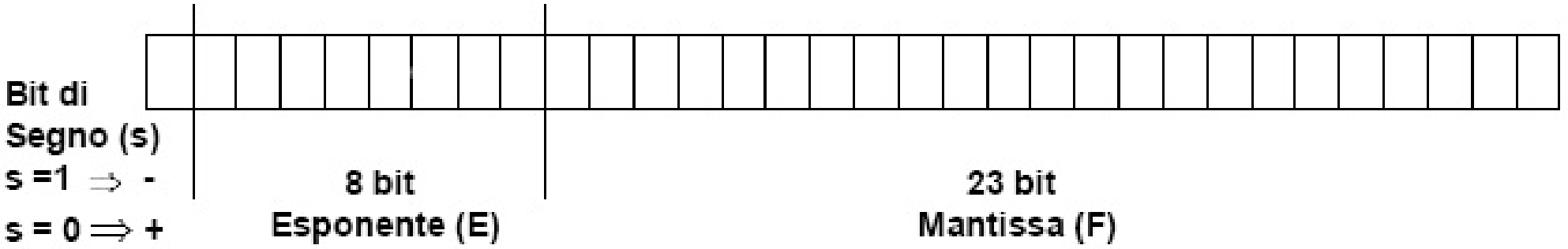
Mantissa

- Questo metodo di scrittura permette di rappresentare un larghissimo insieme numerico all'interno di un determinato numero di cifre, cosa che la virgola fissa non concede.
- Un numero è caratterizzato dal valore *b*, che costituisce la **base** della notazione in cui è scritto il numero, e la quantità *p* di cifre presenti nella mantissa, detta **precisione**.
- La mantissa di un numero scritto con questo metodo si presenta quindi nella forma **±d.ddd...ddd** (una quantità di *p* cifre **d** comprese tra 0 e *b*-1).
- Se la prima cifra della mantissa non è zero, il numero è definito *normalizzato*.

Notazione Scientifica (normalizzata)

- Notazione scientifica (singola cifra a sinistra della virgola)
  - 4,6664456
  - $0{,}34 \times 10^3$
- Notazione scientifica normalizzata (cifra $\neq 0$)
  - $0{,}3432 \times 10^2$ → **NO**
  - $12{,}234 \times 10^2$ → **NO**
  - $1{,}34 \times 10^2$ → **Normalizzato**

Rappresentazione in virgola mobile



- Numero in virgola mobile:
  - $(-1)^s \times F \times 2^E$
- $1{,}101 \times 2^2$ significa:
  - $(-1)^s \times (1+(1\times2^{-1})+(0\times2^{-2})+(1\times2^{-3})) \times 2^2$
- Considerazioni
  - Mantissa molto grande: maggior accuratezza
  - Esponente molto grande: maggior intervallo di numeri rappresentabili

Problemi con l'uso della virgola mobile

- In generale, questo tipo di numeri si comporta in modo molto simile ai numeri reali.
- Tuttavia ciò porta spesso i programmatori a non considerare l'importanza di un'adeguata analisi numerica sui risultati ottenuti.
- Ci sono molte incongruenze tra il comportamento dei numeri in virgola mobile in base 2 impiegati nell'informatica e quello dei numeri reali, anche in casi molto semplici (ad esempio la frazione 0,1 che non può essere rappresentata da nessun sistema binario in virgola mobile).
- Per questo non è un formato impiegato ad esempio in campo finanziario.

Cause principali di errore nel calcolo in virgola mobile

- **arrotondamento**
  - *numeri non rappresentabili (ad esempio 0,1);*
  - *arrotondamento di operazioni aritmetiche (es.: 2/3=0,666667);*
- **assorbimento** *(es.: $1\times10^{15} + 1 = 1\times10^{15}$);*
- **cancellazione** *(es.: sottrazione di due numeri molto vicini);*
- **overflow** (con *segnalazione di risultato infinito);*
- **underflow** *(dà come risultato 0, un numero subnormale o il più piccolo numero rappresentabile);*
- **operazioni impossibili** *(es.: radice quadrata di un numero negativo diverso da zero, danno come risultato Nan);*
- **errori di arrotondamento***: a differenza della virgola fissa, l'impiego del dithering sulla virgola mobile è pressoché impossibile.*

Per ridurre overflow ed underflow

- Per ridurre overflow ed underflow
  - Doppia precisione

Proprietà aritmetica in virgola mobile

- *l'aritmetica in virgola mobile non è* ***associativa****: in generale, per i numeri in virgola mobile,*

$$(x + y) + z \neq x + (y + z)$$
$$(x \cdot y) \cdot z \neq x \cdot (y \cdot z)\ ;$$

- *l'aritmetica in virgola mobile non è* ***distributiva****: in generale,*

$$x \cdot (y + z) \neq (x \cdot y) + (x \cdot z).$$

- In definitiva, **l'ordine in cui vengono eseguite più operazioni in virgola mobile può variarne il risultato**

Proprietà aritmetica in virgola mobile (2)

- Questo è importante per l'analisi numerica, in quanto due formule matematicamente equivalenti possono dare risultati diversi, uno anche sensibilmente più accurato dell'altro
- Per esempio, nella maggior parte delle applicazioni in virgola mobile:
  - $1{,}0 + (10^{100} + -10^{100})$ dà come risultato 1,0
  - $(1{,}0 + 10^{100}) + -10^{100}$ dà come risultato 0,0

Mantissa : esempi

1. in base 10, con 3 cifre per la mantissa e 2 cifre per l'esponente riesco a rappresentare il numero

$$349\ 000\ 000\ 000 = 3.49 * 10^{11}$$

con la coppia (3.49,11) perché M = 3.49 ed E = 11

Mantissa : esempi

2. in base 10, con 3 cifre per la mantissa e 2 per l'esponente riesco a rappresentare

$$0.000\ 000\ 002 = 2.0 * 10^{-9}$$

con la coppia (2.0,-9) perché M = 2.0 ed E = -9

- sia 0.000 000 002 che 349 000 000 000 non sono rappresentabili in virgola fissa usando solo 5 cifre decimali
- Lo stesso si verifica nel caso binario

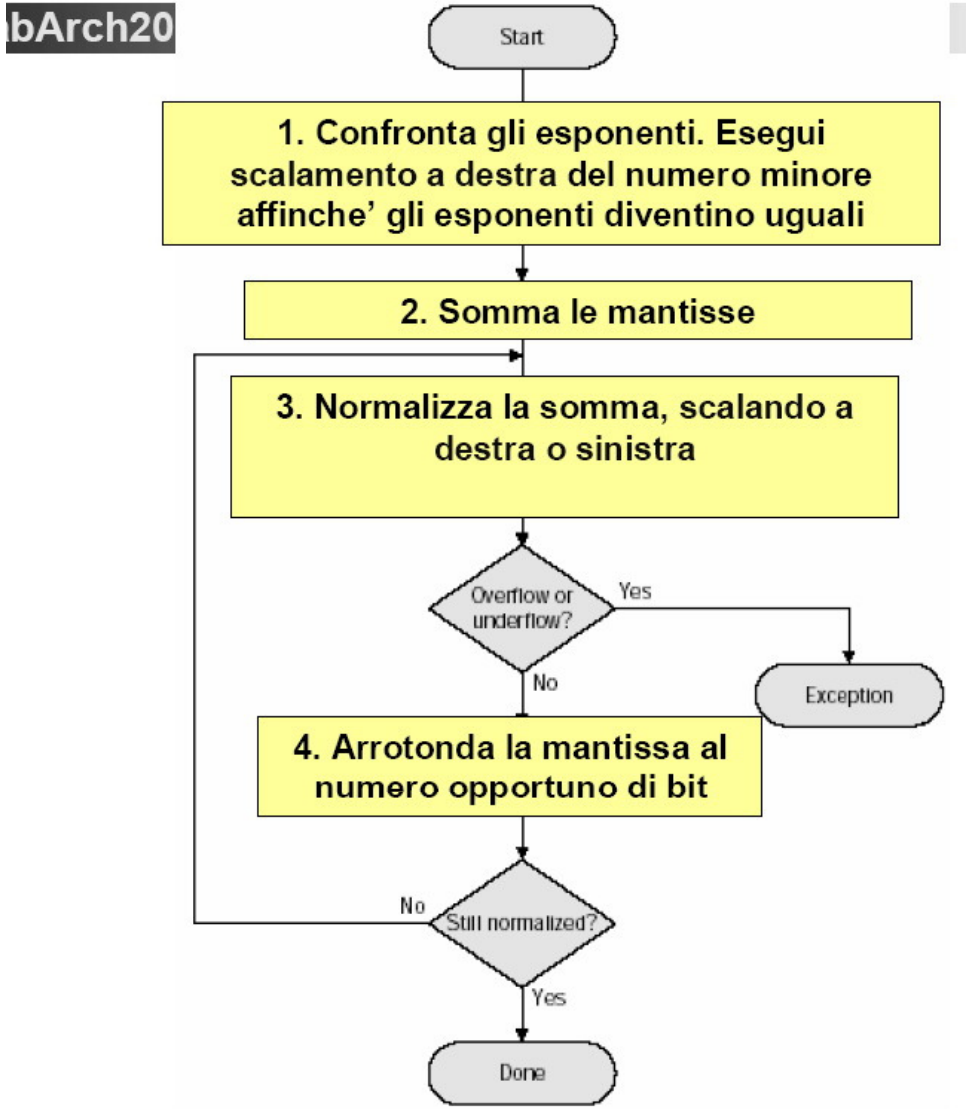
Somma (decimale) in virgola mobile

- Supponiamo di voler sommare questi due numeri: $9{,}452 \times 10^{-1} + 7{,}342 \times 10^{-2}$
- Assumiamo che i numeri siano in notazione scientifica normalizzata
- Assumiamo di poter gestire solo quattro cifre di mantissa

Somma (decimale) in virgola mobile (2)

- **Step1**: bisogna rendere gli esponenti uguali, rendendo il più piccolo uguale al più grande
  - $7{,}342 \times 10^{-2}$ deve essere espresso in termini di 10^{-1}
  - quindi: $0{,}7342 \times 10^{-1} \rightarrow 0{,}734 \times 10^{-1}$
- **Step2: Somma delle mantisse**
  - $9{,}452 \times 10^{-1} + 0{,}734 \times 10^{-1} = 10{,}186 \times 10^{-1}$
- **Step3: Rinormalizzare (se necessario)**
  - $10{,}186 \times 10^{-1} \rightarrow 1{,}0186 \times 10^{0}$
- **Step4: Arrotondamento mantissa**
  - $1{,}0186 \times 10^{0} \rightarrow 1{,}019 \times 10^{0}$

Somma (decimale) in virgola mobile (3)



Operazioni Binarie

- 0 + 0 = 0 con riporto 0
- 0 + 1 = 1 con riporto 0
- 1+ 0 = 1 con riporto 0
- 1+1=0 con riporto 1

Operazioni binarie

$$10110101 + 1000110 = 11111011$$

$$00110011 + 00111000 = 01101011$$

Operazioni binarie (cont.)

1101 x	10011 x
11 =	10 =
1101	00000
1101	10011
100111	100110

Esercizi

- Eseguire le seguenti operazioni direttamente in binario, convertire in decimale e verificare il risultato:
  - 110000 + 1001010;
  - 1001010 + 1111111 + 10;
  - 001001 x 111.

Esercizio

- Realizzare un programma C che converta un intero decimale in binario attraverso la tecnica delle divisioni successive
- Algoritmo base
  1. Leggi il numero da convertire
  2. Calcola le varie cifre binarie del numero convertito attraverso un ciclo
  3. Stampa il numero convertito
- Problemi:
  - Il numero mi viene all'incontrario …

Sub – problema

- Dato un numero scomposto nelle sue cifre e fornite dalla meno significativa a quella più significativa, memorizzarlo in una variabile e stamparlo
  - Input da tastiera:
    - 4 2 7 2 9 7 4 0 7
  - Output a video: 704792724

- Possibile soluzione: costruisco il numero finale moltiplicando ogni cifra per un multiplo di 10 che mi consenta di posizionarla nella giusta posizione

Algoritmo

- Come si fa?
  - Moltiplicatore = 1, numero_finale =0;
  - CICLO
    - Trovo la cifra_attuale;
    - Numero_finale = numero_finale + cifra_attuale * moltiplicatore
    - Moltiplicatore = moltiplicatore * 10;
  - FINE CICLO

Soluzione completa

```
#include <stdio.h>

int main(void)

{
  int numero,resto, bit, risultato=0;
  int moltiplicatore = 1;

  printf("Inserisci un numero decimale: ");
  scanf("%d", &numero);

  while (numero>0)
  {
      bit = numero % 2;
      numero = numero / 2;

      risultato= risultato + (moltiplicatore * bit);

      moltiplicatore = moltiplicatore * 10;

  }

  printf("Il numero in binario e': %d\n", risultato);
  system ("Pause");

  return 0;
}
```

```
int numero,resto, bit, risultato=0;
int moltiplicatore = 1;
```

```
while (numero>0)
  {
      bit = numero % 2;
      numero = numero / 2;

      risultato= risultato +
       (moltiplicatore * bit);

      moltiplicatore =
              moltiplicatore * 10;

  }
```