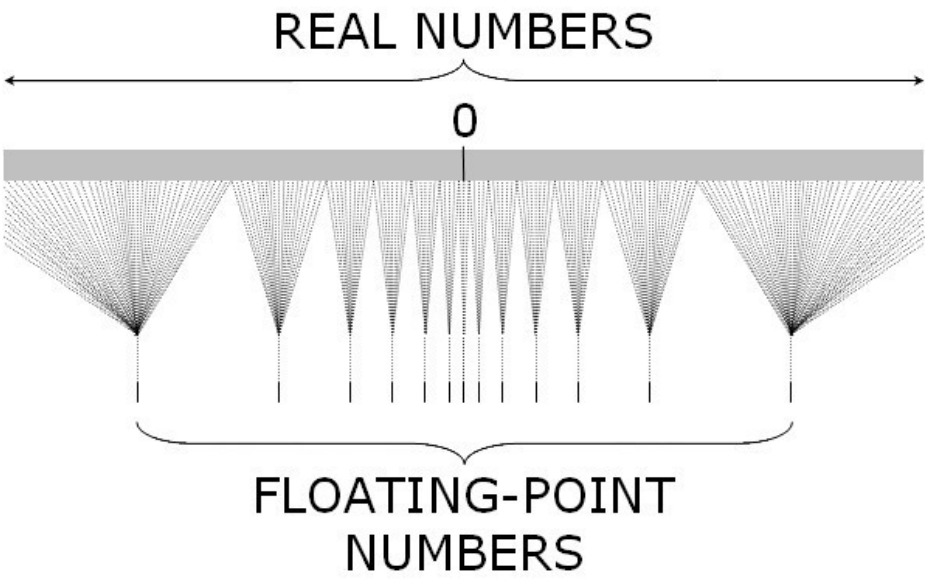
Rappresentazione dell'Informazione



Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char
0	0x00	000	0000000	NUL	32	0x20	040	0100000	space	64	0x40	100	1000000	@	96	0x60	140	1100000	`
1	0x01	001	0000001	SOH	33	0x21	041	0100001	!	65	0x41	101	1000001	A	97	0x61	141	1100001	a
2	0x02	002	0000010	STX	34	0x22	042	0100010	"	66	0x42	102	1000010	B	98	0x62	142	1100010	b
3	0x03	003	0000011	ETX	35	0x23	043	0100011	#	67	0x43	103	1000011	C	99	0x63	143	1100011	c
4	0x04	004	0000100	EOT	36	0x24	044	0100100	$	68	0x44	104	1000100	D	100	0x64	144	1100100	d
5	0x05	005	0000101	ENQ	37	0x25	045	0100101	%	69	0x45	105	1000101	E	101	0x65	145	1100101	e
6	0x06	006	0000110	ACK	38	0x26	046	0100110	&	70	0x46	106	1000110	F	102	0x66	146	1100110	f
7	0x07	007	0000111	BEL	39	0x27	047	0100111	'	71	0x47	107	1000111	G	103	0x67	147	1100111	g
8	0x08	010	0001000	BS	40	0x28	050	0101000	(	72	0x48	110	1001000	H	104	0x68	150	1101000	h
9	0x09	011	0001001	TAB	41	0x29	051	0101001	)	73	0x49	111	1001001	I	105	0x69	151	1101001	i
10	0x0A	012	0001010	LF	42	0x2A	052	0101010	*	74	0x4A	112	1001010	J	106	0x6A	152	1101010	j
11	0x0B	013	0001011	VT	43	0x2B	053	0101011	+	75	0x4B	113	1001011	K	107	0x6B	153	1101011	k
12	0x0C	014	0001100	FF	44	0x2C	054	0101100	,	76	0x4C	114	1001100	L	108	0x6C	154	1101100	l
13	0x0D	015	0001101	CR	45	0x2D	055	0101101	-	77	0x4D	115	1001101	M	109	0x6D	155	1101101	m
14	0x0E	016	0001110	SO	46	0x2E	056	0101110	.	78	0x4E	116	1001110	N	110	0x6E	156	1101110	n
15	0x0F	017	0001111	SI	47	0x2F	057	0101111	/	79	0x4F	117	1001111	O	111	0x6F	157	1101111	o
16	0x10	020	0010000	DLE	48	0x30	060	0110000	0	80	0x50	120	1010000	P	112	0x70	160	1110000	p
17	0x11	021	0010001	DC1	49	0x31	061	0110001	1	81	0x51	121	1010001	Q	113	0x71	161	1110001	q
18	0x12	022	0010010	DC2	50	0x32	062	0110010	2	82	0x52	122	1010010	R	114	0x72	162	1110010	r
19	0x13	023	0010011	DC3	51	0x33	063	0110011	3	83	0x53	123	1010011	S	115	0x73	163	1110011	s
20	0x14	024	0010100	DC4	52	0x34	064	0110100	4	84	0x54	124	1010100	T	116	0x74	164	1110100	t
21	0x15	025	0010101	NAK	53	0x35	065	0110101	5	85	0x55	125	1010101	U	117	0x75	165	1110101	u
22	0x16	026	0010110	SYN	54	0x36	066	0110110	6	86	0x56	126	1010110	V	118	0x76	166	1110110	v
23	0x17	027	0010111	ETB	55	0x37	067	0110111	7	87	0x57	127	1010111	W	119	0x77	167	1110111	w
24	0x18	030	0011000	CAN	56	0x38	070	0111000	8	88	0x58	130	1011000	X	120	0x78	170	1111000	x
25	0x19	031	0011001	EM	57	0x39	071	0111001	9	89	0x59	131	1011001	Y	121	0x79	171	1111001	y
26	0x1A	032	0011010	SUB	58	0x3A	072	0111010	:	90	0x5A	132	1011010	Z	122	0x7A	172	1111010	z
27	0x1B	033	0011011	ESC	59	0x3B	073	0111011	;	91	0x5B	133	1011011	[	123	0x7B	173	1111011	{
28	0x1C	034	0011100	FS	60	0x3C	074	0111100	<	92	0x5C	134	1011100	\	124	0x7C	174	1111100	|
29	0x1D	035	0011101	GS	61	0x3D	075	0111101	=	93	0x5D	135	1011101	]	125	0x7D	175	1111101	}
30	0x1E	036	0011110	RS	62	0x3E	076	0111110	>	94	0x5E	136	1011110	^	126	0x7E	176	1111110	~
31	0x1F	037	0011111	US	63	0x3F	077	0111111	?	95	0x5F	137	1011111	_	127	0x7F	177	1111111	DEL

Prof. Ivan Lanese

Rappresentazione dell'Informazione

- I calcolatori elaborano informazioni di varia natura:
  - Testi, immagini, suoni, filmati, …
- Come abbiamo visto le memorie dei calcolatori digitali contengono solo valori binari
  - E' quindi necessaria una opportuna codifica
- Studieremo le codifiche di numeri e caratteri
  - In particolare vedremo le tecniche per codificare
    - i numeri interi
    - i numeri con parte decimale tramite la codifica "floating point"
    - caratteri tramite le codifiche ASCII e UNICODE

Sistemi di numerazione posizionali

- Data una codifica in base *b*, è possibile rappresentare i numeri interi tramite sequenze di cifre tra 0 … *b-1*
  - Sia $d_k\ d_{k-1} \ldots d_1\ d_0$ un numero codificato in base *b*, allora tale numero corrisponderà a:
    - $\sum_{i=0..k} d_i \times b^i$
    - Esempio: in binario $1011 = 1\times2^3 + 1\times2^1 + 1\times2^0 = 11$
  - In informatica le codifiche posizionali principali sono:
    - Codifica binaria: base 2 (cifre 0,1)
    - Codifica ottale: base 8 (cifre 0,1,2,3,4,5,6,7)
    - Codifica esadecimale: base 16 (cifre 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F con le lettere A…F che rappresentano rispettivamente 10…15)

Numerazione posizionale nelle principali basi

Binario 1 1 1 1 1 0 1 0 0 0 1

$$1\times2^{10}+1\times2^{9}+1\times2^{8}+1\times2^{7}+1\times2^{6}+0\times2^{5}+1\times2^{4}+0\times2^{3}+0\times2^{2}+0\times2^{1}+1\times2^{0}=$$
$$=1024+512+256+128+64+0+16+0+0+0+1=2001$$

Ottale 3 7 2 1

$$3\times8^{3}+7\times8^{2}+2\times8^{1}+1\times8^{0}=$$
$$=1536+448+16+1=2001$$

Decimale 2 0 0 1

$$2\times10^{3}+0\times10^{2}+0\times10^{1}+1\times10^{0}=$$
$$=2000+0+0+1=2001$$

Esadecimale 7 D 1

$$7\times16^{2}+13\times16^{1}+1\times16^{0}=$$
$$=1792+208+1=2001$$

Figura A.2 Il numero 2001 in binario, ottale, decimale ed esadecimale.

Conversione di base

- Due sequenze di simboli in due diverse codifiche sono equivalenti se rappresentano il medesimo numero
  - Convertire una sequenza (in una codifica) vuol dire trovare l'equivalente sequenza in una codifica diversa
- Convertire da binario in ottale (o esadecimale) e viceversa è facile:
  - corrispondenza fra 3 (o 4) cifre binarie e cifre ottali (o esadecimali)

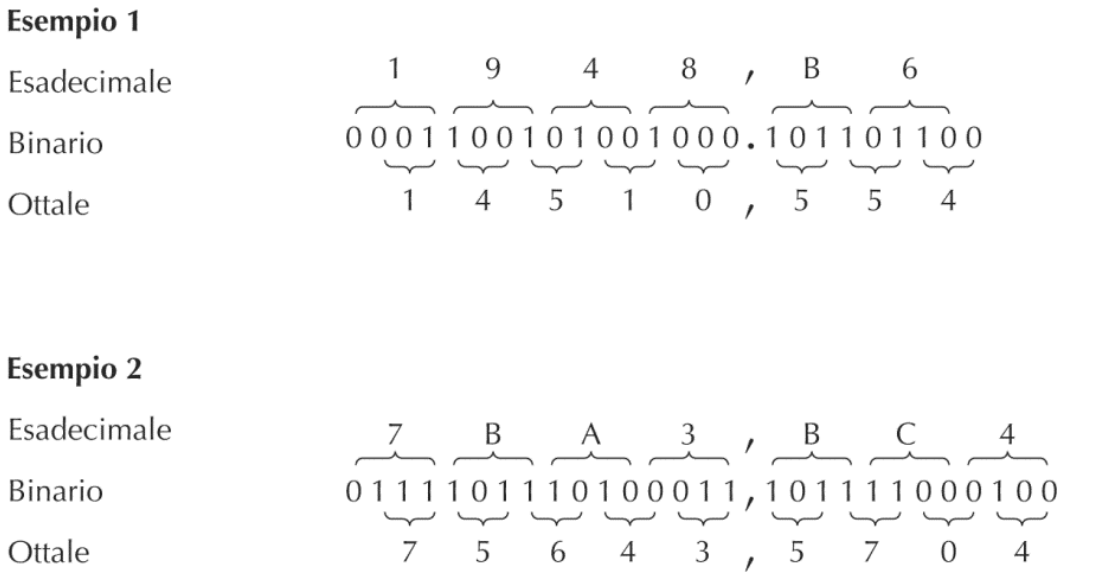


Figura A.4 Esempi di conversione da ottale a binario e da esadecimale a binario.

Conversione da binario a decimale

- Oltre alla tecnica generale per il calcolo del numero espresso tramite una base *b*, è possibile usare la tecnica delle moltiplicazioni successive
  - Partendo da sinistra e da un accumulatore uguale a 0, per ogni cifra si moltiplica il valore dell'accumulatore per 2 e si aggiunge la cifra considerata

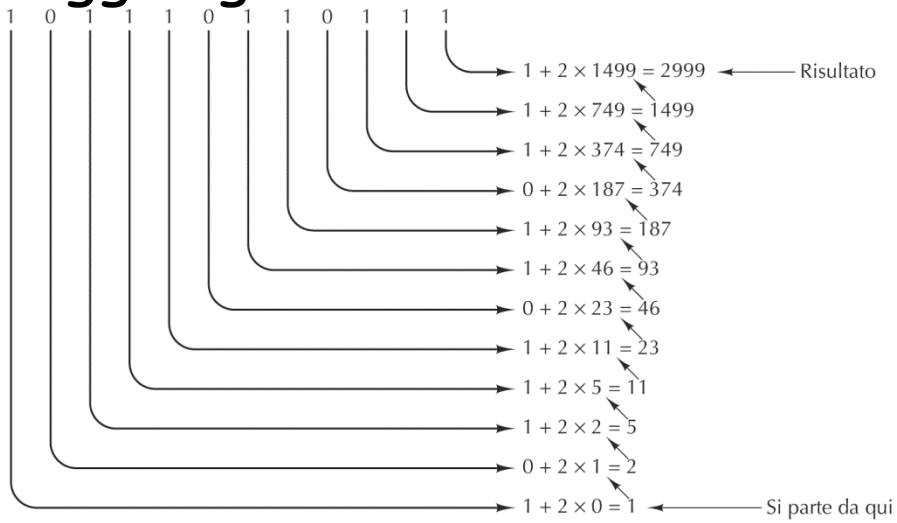


Figura A.6 La conversione del numero binario 101110110111 in decimale mediante raddoppiamenti successivi, a partire dal basso. Ogni riga si ottiene raddoppiando l'elemento della riga precedente e sommandogli il bit corrispondente. Per esempio, 749 è due volte 374 più il bit 1 che si trova in corrispondenza della riga di 749.

Conversione da decimale a binario

- Nella conversione da decimale a binario si usa la tecnica inversa, detta delle divisioni successive
  - Si divide ripetutamente per due e si considerano i resti (*0* o *1*) in ordine inverso di generazione

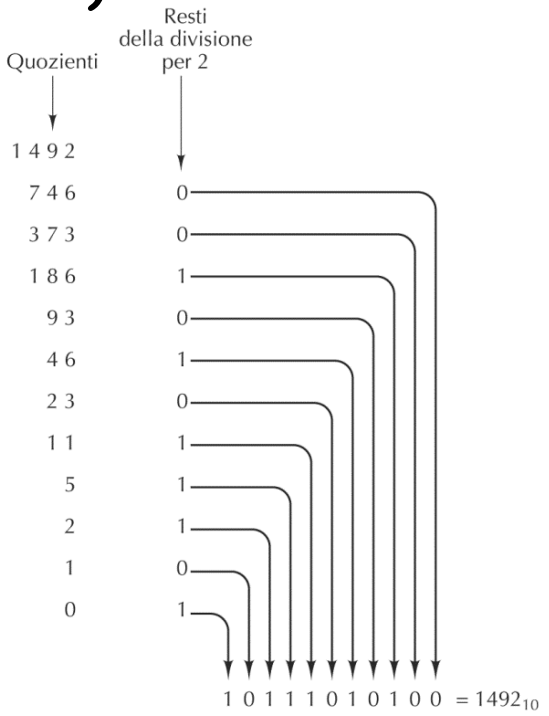


Figura A.5 La conversione del numero decimale 1492 in binario mediante dimezzamenti successivi, partendo dall'alto e procedendo verso il basso. Per esempio, 93 diviso 2 fa 46 con resto 1, riportati nella riga successiva.

Numeri binari negativi

- Per codificare in binario i numeri interi con segno, si possono usare varie tecniche:
  - Modulo e segno: si usa il bit più a sinistra come segno (0 coincide con +, 1 coincide con -):
    - Esempio usando 8 bit: 00000110=6, 10000110=-6
  - Complemento a 1: il bit più a sx indica il segno, ma se il numero è negativo il modulo viene complementato
    - Esempio usando 8 bit: 00000110=6, 11111001=-6
  - Complemento a 2: come per il complemento a 1, ma se il numero è negativo dopo il complemento si aggiunge 1
    - Esempio usando 8 bit: 00000110=6, 11111010=-6

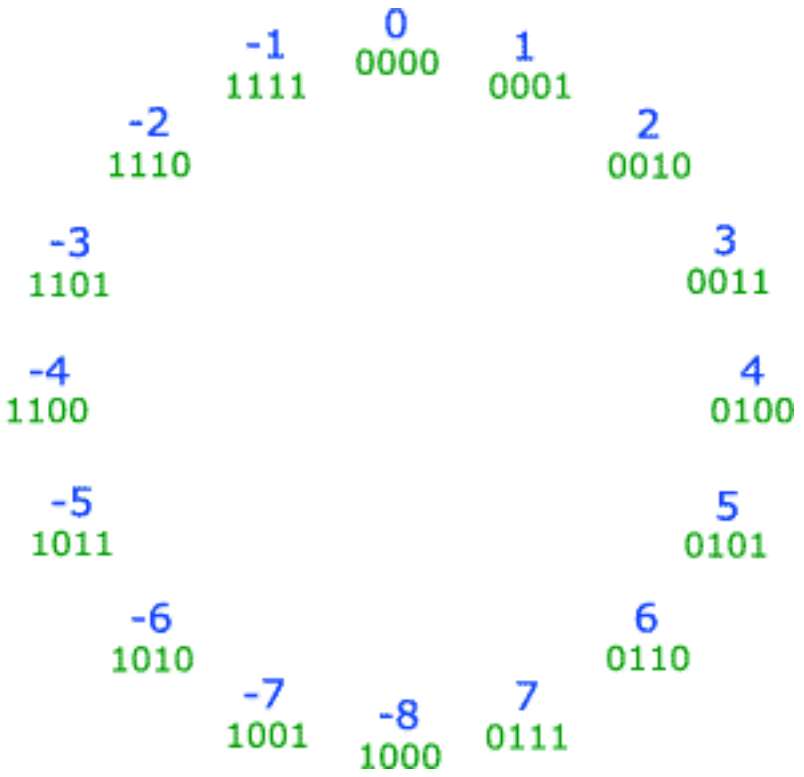
Codifica a complemento a 2

- Vediamo meglio la codifica a complemento a 2:
  - Consideriamo il numero negativo $-n$
  - Sia $b_{k-1}\ldots b_0$ la sua codifica in complemento a 2 con k bit
  - Se interpreto $b_{k-1}\ldots b_0$ come numero binario, ottengo un numero positivo m che coincide con $-n$ modulo 2^k
    - Esempio usando 8 bit:
      in complemento a 2, sappiamo che 11111010=-6;
      11111010 come numero binario coincide con 250;
      ma -6 mod 2^8 = -6 mod 256 = 250
  - In generale, data $b_{k-1}\ldots b_0$ in complemento a 2, il numero corrispondente è $-b_{k-1}\times 2^{k-1} + \sum_{i=0..k-2} b_i \times 2^i$

Codifica a complemento a 2 (esempio)

- Consideriamo di usare codifica a complemento a 2 utilizzando 4 bit:

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



Numeri rappresentabili

- Con k bit si possono rappresentare al più 2^k numeri diversi
- Se interessano solo i numeri naturali (non negativi) con k bit si rappresentano i numeri nell'intervallo $[0..2^k-1]$
  - Esempio: con 8 bit rappresentiamo [0..255]
- Se usiamo modulo e segno, o complemento a 1, rappresenteremo i numeri nell'intervallo $[-2^{k-1}+1.. 2^{k-1}-1]$
  - Esistono due codifiche per lo 0
  - Esempio: con 8 bit rappresentiamo [-127..127]
- In complemento a 2 rappresenteremo i numeri nell'intervallo $[-2^{k-1}.. 2^{k-1}-1]$
  - Esempio: con 8 bit rappresentiamo [-128..127]

Codifica “in eccesso”

- Un altro modo per rappresentare i numeri nell'intervallo $[-2^{k-1}.. 2^{k-1}-1]$ consiste nel sommare 2^{k-1} alla rappresentazione binaria del numero stesso, così si sposta l'intervallo ai numeri non negativi $[0.. 2^{k}-1]$
  - Esempio:
    0000…0 rappresenta il numero -2^{k-1}
    1111…1 rappresenta il numero $2^{k-1}-1$
    1000…0 rappresenta il numero 0
- Questa tecnica prende il nome di codifica in eccesso
  - La decodifica si ottiene applicando la decodifica standard e poi sottraendo 2^{k-1} al numero ottenuto

Somma binaria

- E' possibile eseguire somme binarie fra 2 numeri seguendo la tabella sottoriportata con somma e riporto per la somma di coppie di bit

Addendo	0 +	0 +	1 +	1 +
Addendo	0 =	1 =	0 =	1 =
Somma	0	1	1	0
Riporto	0	0	0	1

Figura A.8 Tabellina della somma binaria.

- In generale per tener conto del riporto dobbiamo sommare 3 bit alla volta (2 degli addendi e 1 del riporto)

Somma binaria in complemento a 1 e a 2

- In complemento a 1, il riporto finale viene sommato
- In complemento a 2, il riporto viene scartato

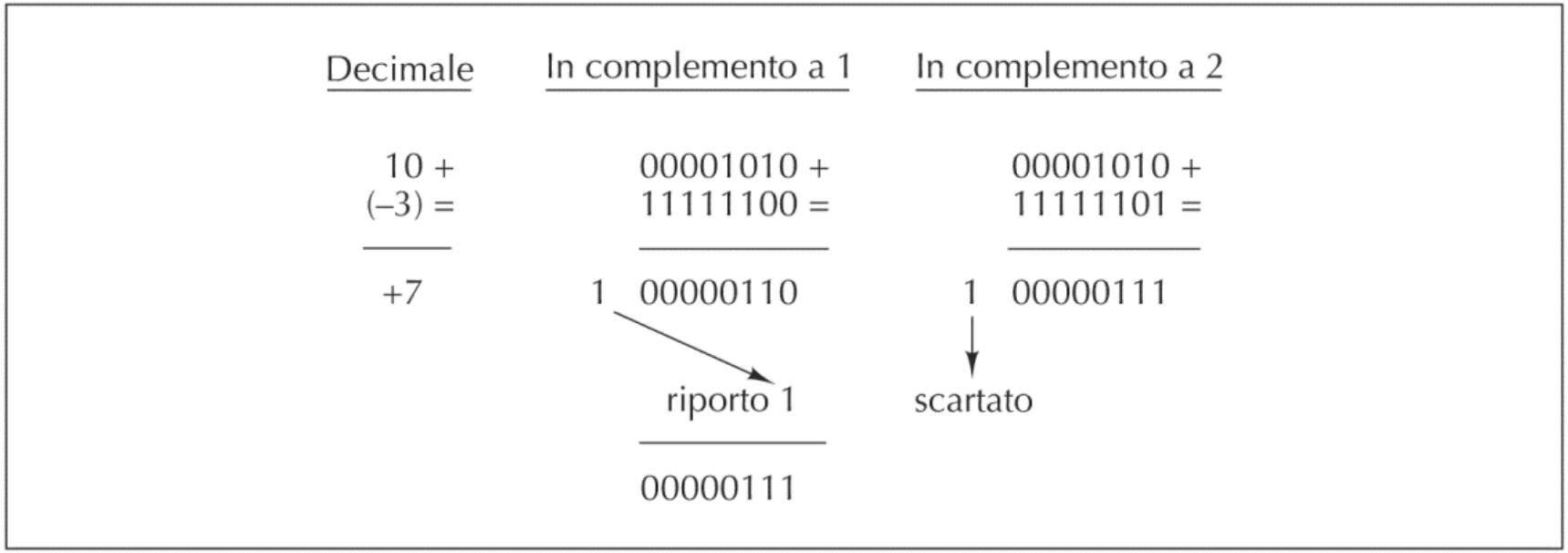


Figura A.9 Somma in complemento a uno e in complemento a due.

Somma binaria in complemento a 1 e a 2 (continua)

- Quando avviene un overflow?
  - Se gli addendi hanno segno opposto non c'è overflow
  - Se gli addendi hanno lo stesso segno, ma il risultato ha un segno diverso c'è overflow

Esempi:

01111111+00000001=10000000 (overflow)

10000000+11111111=01111111 in complemento a 2 (overflow)

10000000+11111111=10000000 in complemento a 1 (no overflow in quanto 11111111 = 0)

Rappresentazione di numeri con la virgola

- L'usuale tecnica di rappresentazione dei numeri con virgola fissa (prima della virgola le unità-decine-centinaia.., dopo la virgola i decimi-centesimi-millesimi…) non è sempre efficace
  - Ad esempio, un calcolo astronomico potrebbe riguardare la massa di entità come l'elettrone (9×10^{-28}) o il sole (2×10^{33})
  - Con la tecnica usuale (detta virgola fissa) servirebbero 33 cifre a sinistra della virgola e 28 a destra della virgola
  - Per questo motivo nei calcolatori (e nei testi scientifici) si usa la codifica a virgola mobile (vedi prossime slide)

Codifica a virgola mobile (floating point)

- L'idea alla base della codifica floating point è rappresentare un numero con la virgola *n* tramite altri due numeri *f* ed *e* tali che:
  - $n = f \times 10^e$
  - *f* viene detto frazione (o mantissa)
  - *e* viene detto esponente (o caratteristica)
- Esempi:

$$\begin{array}{lcll} 3,14 & = & 0,314 \times 10^1 & = & 3,14 \times 10^0 \\ 0,000001 & = & 0,1 \times 10^{-5} & = & 1,0 \times 10^{-6} \\ 1941 & = & 0,1941 \times 10^4 & = & 1,941 \times 10^3 \end{array}$$

Esempio di ipotetica codifica a virgola mobile

- Immaginiamo di usare un'ipotetica codifica che usa
  - per la frazione un numero (con aggiunta di segno) a tre cifre uguale a 0 oppure compreso fra 0,001 e 0,999
  - per l'esponente un numero (con aggiunta di segno) compreso fra 0 e 99
- Si riescono a rappresentare numeri solo negli intervalli 2 e 6 dell'immagine (oltre al numero 0)

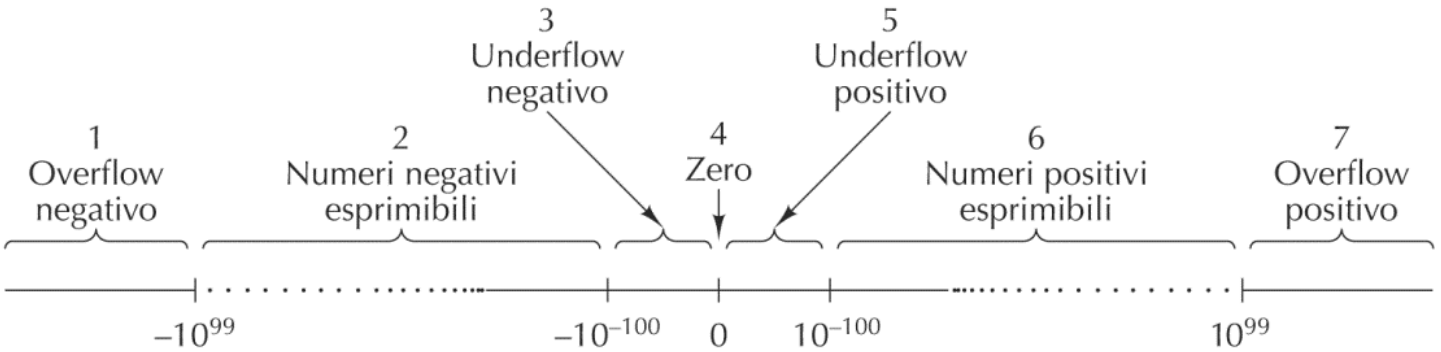


Figura B.1 La rappresentazione *R* ripartisce la retta reale in sette regioni.

Overflow, Underflow, e precisione finita

- Operazioni su numeri con la virgola mobile (solitamente eseguite da hardware dedicato) possono generare due tipi di errore:
  - Overflow: numero in valore assoluto troppo grande (aree 1 e 7 dell'immagine precedente)
  - Underflow: numero in valore assoluto troppo piccolo (aree 3 e 4 dell'immagine precedente)
- Inoltre, anche nelle aree 2 e 6, esistono un'infinità di numeri reali che non possono essere rappresentati in modo preciso causa la "precisione finita" dei calcolatori
  - In ogni area è possibile rappresentare non più di 999 × 199 numeri diversi (999 sono le possibili frazioni diverse e 199 i possibili esponenti diversi)

Esempi concreti

- Nei calcolatori di solito non si considera base 10, ma base 2, 4, 8 o 16
- Si usa una frazione minore di 1
- Inoltre, si tende a normalizzare la frazione:
  - La cifra più significativa non può essere uguale a 0
- Nella prossima slide si mostrano 4 diverse rappresentazioni floating point del numero 432:
  - base 2 non normalizzato
  - base 2 normalizzato
  - base 16 non normalizzato
  - base 16 normalizzato

Esempi concreti (continua)

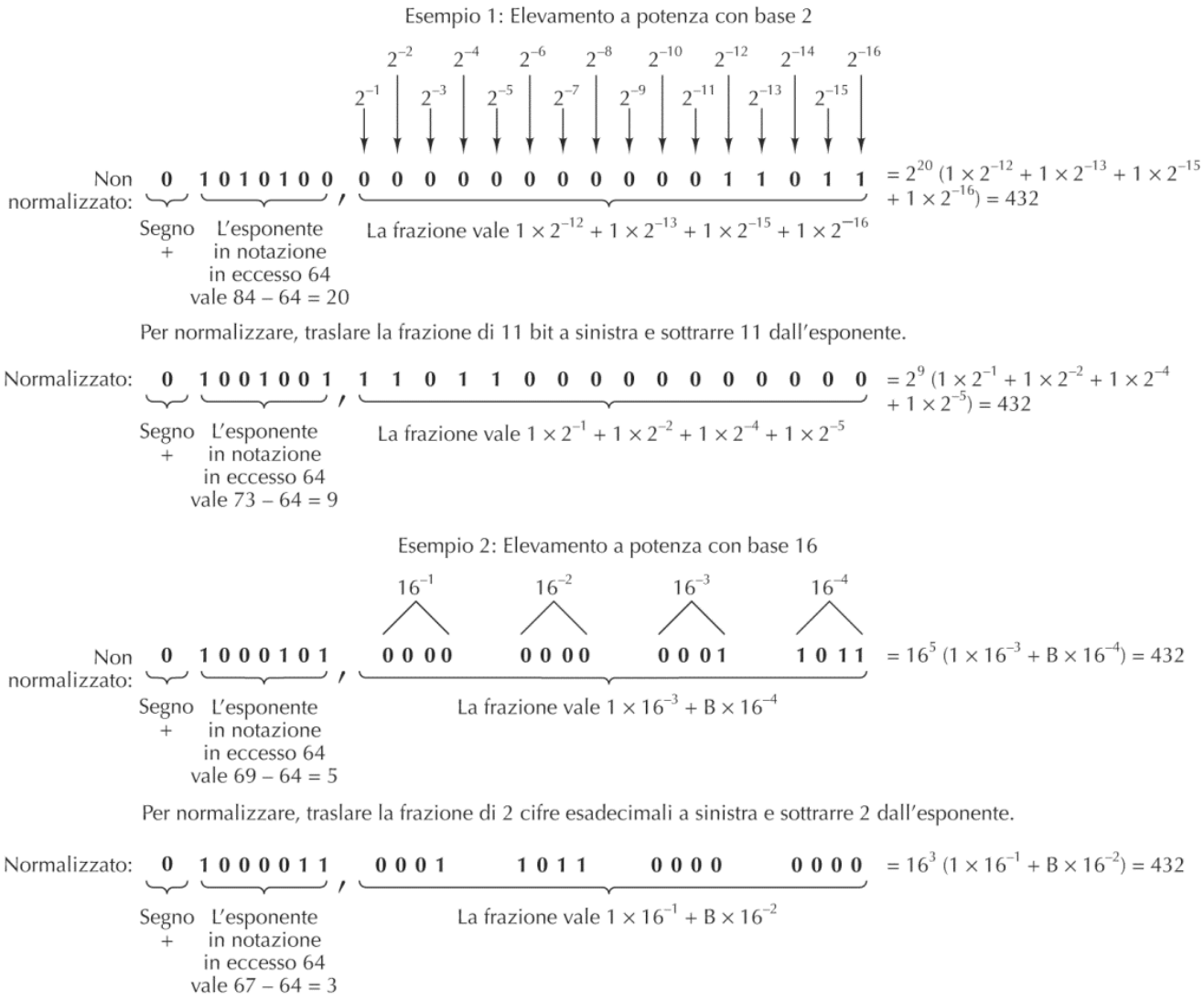


Figura B.3 Esempi di numeri in virgola mobile normalizzati.

Lo standard IEEE 754

- Il più usato standard per rappresentare numeri floating point
- Definisce diversi formati inclusi single (BINARY32) e double (BINARY64) precision
- Es. BINARY32:
  - binario
  - 1 bit di segno
  - 8 bit di esponente
  - 23 bit di mantissa
  - codifiche speciali per infiniti e NaN (Not a Number)

Rappresentazione dei caratteri (nei testi)

- Una codifica per caratteri largamente diffusa è la codifica ASCII: American Standard Code for Information Interchange
- Usa 7 bit per i principali simboli alfabetici anglosassoni e per alcuni caratteri speciali (necessari in passato nei terminali/telescriventi)

Esa	Nome	Significato	Esa	Nome	Significato
0	NUL	Nullo	10	DLE	Uscita trasmissione (Data Link Escape)
1	SOH	Inizio intestazione (Start Of Heading)	11	DC1	Controllo periferica 1
2	STX	Inizio testo (Start Of Text)	12	DC2	Controllo periferica 2
3	ETX	Fine testo (End Of Text)	13	DC3	Controllo periferica 3
4	EOT	Fine trasmissione (End Of Transmission)	14	DC4	Controllo periferica 4
5	ENQ	Interrogazione (Enquiry)	15	NAK	Riconsocimento negativo (Negative AcKnowledgement)
6	ACK	Riconoscimento (ACKnowledgement)	16	SYN	Annulla (SYNchronous Idle)
7	BEL	Campanello (BELL)	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Tabulazione orizzontale (Horizontal Tab))	19	EM	Fine supporto (End of Medium)
A	LF	Riga nuova (Line Feed)	1A	SUB	Sostituisci (SUBstitute)
B	VT	Tabulazione verticale (Vertical Tab)	1B	ESC	Esc (ESCape)
C	FF	Avanzamento carta/nuova pagina (Form Feed)	1C	FS	Separatore di file (File Separator)
D	CR	Ritorno a capo (Carriage Return)	1D	GS	Separatore di gruppi (Group Separator)
E	SO	Disinserzione (Shift Out)	1E	RS	Separatore di record (Record Separator)
F	SI	Inserzione (Shift In)	1F	US	Separatore di unità (Unit Separator)

Esa	Car	Esa	Car	Esa	Car	Esa	Car	Esa	Car	Esa	Car
20	Spazio	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	“	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Figura 2.44 Caratteri ASCII.

Codifica UNICODE

- ASCII tratta solo i simboli dell'alfabeto anglosassone
- Per codificare altri alfabeti, si rende necessaria una codifica che usa una quantità superiore di bit
- A tal scopo è nata la codifica UNICODE
  - Usa 16 bit
  - Compatibile con ASCII (se si mettono a 0 i primi 9 bit i restanti 7 possono essere usati come in ASCII)
  - Si usano intervalli contigui di codici per i diversi alfabeti (es. latino, greco, cirillico, armeno, ebraico,..)
  - Ci sono codici non ancora assegnati per poter rappresentare in futuro caratteri al momento non considerati

Codifica UTF-8

- UNICODE ha alcuni limiti:
  - Usa sempre 16 bit anche per caratteri ASCII per i quali ne sarebbero sufficienti 7
  - E' oramai vicino all'esaurimento dei possibili codici
- UTF-8 Unicode Transformation Format può dinamicamente occupare da 1 a 4 byte a seconda dell'informazione da codificare
  - I bit iniziali indicano il formato specifico e la quantità di bit utilizzati
  - Se il bit iniziale è 0, i restanti 7 contengono una codifica ASCII
- Altre codifiche esistono (UTF-16, ...)

Codici correttori

- Indipendentemente dal tipo di dati memorizzati, occasionalmente le memorie sono soggette ad errori sia durante le operazioni di lettura che durante le operazioni di scrittura
- Analogamente ci possono essere errori trasmettendo i dati
- Per proteggersi, alcune memorie utilizzano dei codici di rilevazione (ed in alcuni casi anche correzione) di errori
- Se una parola consiste di m bit, si aggiungono r bit di controllo ottenendo una "parola di codice" a $n=m+r$ bit
- Un codice è un meccanismo atto a determinare gli r bit di controllo relativi ad ogni parola di m bit

Distanza di Hamming

- La distanza di Hamming tra due sequenze di bit è il numero di bit rispetto ai quali le due parole differiscono: 001101 e 011100 hanno distanza 2 (differiscono per 2 bit)
- La distanza di Hamming di un codice è la minima distanza di Hamming tra "parole di codice" (cioè le parole corrette che non hanno subito errori)
- Regola generale:
  - Per rilevare *d* bit errati è necessario un codice con distanza di Hamming maggiore o uguale a *d+1*
  - Per correggere *d* bit errati è necessario un codice con distanza di Hamming maggiore o uguale a *2d+1*

Esempi di codici

- Uno dei codici più semplici è il cosiddetto "bit di parità"
  - un unico bit di controllo (*r=1*), che è scelto in modo che il numero di bit "1" nella "parola di codice" sia pari
  - Il codice ha distanza di Hamming uguale a *2*
- Il codice può essere usato per rilevare singoli bit errati
  - basta controllare se i bit "1" nella parola sono pari o no
- Inventiamo un nuovo codice con le seguenti "parole di codice" di lunghezza 10:
  - 0000000000 - 0000011111 - 1111100000 - 1111111111
  - distanza di Hamming uguale a 5 ed è possibile correggere fino a 2 bit errati

Un limite strutturale

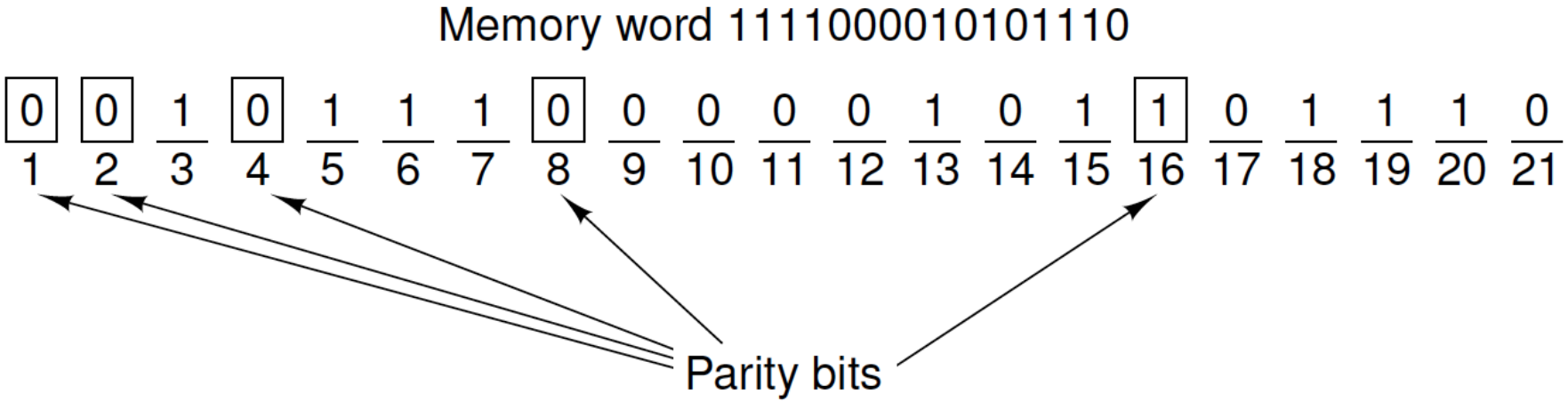
- Supponiamo che un codice con *m* bit di dati e *r* bit di controllo sia in grado di correggere tutti i possibili errori su singolo bit
  - ciascuna delle 2^m "parole di codice" necessita di *n+1* parole ad essa dedicate (con *n=m+r*):
    - 1 per la "parola di codice" corretta
    - *n* per i possibili errori di un solo bit
  - Dato che il numero totale di combinazioni di bit è 2^n deve valere
    - $(n+1)2^m \leq 2^n$ da cui deriva $m+r+1 \leq 2^r$
- Il codice di Hamming riesce a correggere tutti i possibili errori su singolo bit usando proprio il numero minimo di bit di controllo *r* che soddisfa la disequazione di cui sopra

Codice di Hamming

- Ad *m* bit si aggiungono *r* bit di controllo, con *r* scelto in modo tale da essere il numero minimo per cui $m+r+1 \leq 2^r$
  - identifichiamo la posizione di un bit nella "parola di codice" usando i numeri binari tra *1* e *m+r*
  - i bit di controllo vengono messi in posizioni identificate da numeri binari con esattamente una sola cifra uguale a 1 (1,2,4,8,… cioè le potenze di due)
  - i bit di controllo sono bit di parità per sottogruppi di bit
- il bit identificato dal numero binario con un 1 in posizione *i*, si raggruppa con quelli in posizioni identificate da un numero la cui cifra *i*-esima è 1
- In caso di errore, il bit errato sarà nella posizione data dalla somma delle posizioni dei bit di parità errati

Un esempio di applicazione del codice di Hamming

- Esempio con m = 16
  - quindi prendiamo r = 5 (16+5+1≤32 mentre 16+4+1>16)
  - i bit di controllo vanno in posizione 1, 2, 4, 8, 16 rispettivamente bit di parità per i gruppi con posizioni
    - 1,3,5,7,9,11,13,15,17,19,21 / 2,3,6,7,10,11,14,15,18,19 / 4,5,6,7,12,13,14,15,20,21 / 8.9.10.11.12.13.14.15 / 16.17.18.19.20.21



Un esempio di applicazione del codice di Hamming (continua)

- Ecco come calcolare i gruppi a cui appartiene il bit in posizione *j*:
  - la posizione *j* appartiene ai gruppi corrispondenti ai bit=1 nella codifica binaria di *j* (es. tutti i *j* nel gruppo rosso fanno parte del gruppo del bit 1)
  - in caso di errore, basta guardare i gruppi che contengono l'errore controllando i bit di parità
    - la posizione dell'errore è quindi identificata dal numero binario che si ottiene mettendo a 1 solo i bit corrispondenti ai gruppi con l'errore

1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
10	01010
11	01011
12	01100
13	01101
14	01110
15	01111
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101