RAPPRESENTAZIONE DEI NUMERI BINARI

Corso di Fondamenti di Informatica
AA 2010-2011

Prof. Franco Zambonelli

Numeri interi senza segno

Caratteristiche generali

- numeri naturali (1,2,3,...) + lo zero
- rappresentabili con diverse notazioni
 - ◆ non posizionali (esempio: notazione romana: I, II, III, IV, V, IX, X, XI...)
 - ◆ posizionale (notazione araba): 1, 2, .. 10, 11, ... 100, ...
- caratteristiche
 - ◆ le notazioni *non posizionali* hanno regole proprie e rendono molto complessa l'esecuzione dei calcoli
 - ◆ la notazione *posizionale*, invece, consente di rappresentare i numeri in modo compatto, e rende semplice l'effettuazione dei calcoli

Notazione posizionale

- concetto di *base* di rappresentazione, ***B***
- rappresentazione del numero come *sequenza di simboli*, detti *cifre*
 - ◆ appartenenti a un *alfabeto* composto di ***B*** simboli distinti
 - ◆ in cui ogni simbolo rappresenta un valore fra **0** e ***B-1***
- il valore di un numero *v* espresso in questa notazione è ricavabile
 - ◆ a partire dal valore rappresentato da ogni simbolo
 - ◆ *pesato* in base alla *posizione* che occupa nella sequenza

Valore di un numero (espresso in notazione posizionale)

- Formalmente, il *valore* di un numero v espresso in questa notazione è dato dalla formula:

$$v = \sum_{k=0}^{n-1} d_k B^k$$

- ◆ B è la base
- ◆ d_k ($k=0..n-1$) sono le *cifre* (comprese fra 0 e $B-1$)

QUINDI, una sequenza di cifre *non è interpretabile* se non si precisa la base in cui è espressa.

Esempi

Stringa	Base	Alfabeto	Calcolo valore	Valore
12	4	{0,1,2,3}	$4 * 1 + 2$	<i>sei</i>
12	8	{0,1,...,7}	$8 * 1 + 2$	<i>dieci</i>
12	10	{0,1,...,9}	$10 * 1 + 2$	<i>dodici</i>
12	16	{0,...,9, A,.., F}	$16 * 1 + 2$	<i>diciotto</i>

Osservazioni

- ogni *numero* è esprimibile in modo univoco in una *qualunque base*; in particolare:
 - ◆ base $B=2$ → **due sole cifre: 0 e 1**
 - ◆ base $B=8$ → otto cifre: 0, 1, 2, 3, 4, 5, 6, 7
 - ◆ base $B=10$ → dieci cifre: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
 - ◆ base $B=16$ → sedici cifre: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Il problema della Conversione di Rappresentazione

- Ogni numero è espresso, in una base data, da una *ben precisa sequenza di cifre*
- Dalla definizione di notazione posizionale segue che, data una rappresentazione sotto forma di sequenza di cifre e una base, il numero corrispondente si può ricavare applicando la formula già vista:

$$v = \sum_{k=0}^{n-1} d_k B^k$$

- *ma come si ricava la rappresentazione di un dato numero, sotto forma di sequenza di cifre, in una base assegnata?*

Conversione da numero a sequenza di cifre

Notazioni posizionali e non

- in una notazione non posizionale, si sarebbe dovuto procedere per confronti e/o applicando complesse regole (strettamente dipendenti dalla rappresentazione)
 - esempio: convertire 27 in notazione romana
 - 27 è compreso fra 20 e 30 → XX (parte residua non rappresentata: 7)
 - 7 è compreso fra 5 e 10 → V (parte residua non rappresentata: 2)
 - 2 si rappresenta direttamente → II
 conclusione: 27 si rappresenta in romano con la stringa di simboli *XXVII*
- invece, **si può sfruttare la notazione posizionale** per operare in modo più semplice ed efficiente.

Algoritmo di conversione (“Algoritmo delle divisioni successive”)

Osservazione preliminare

- in una notazione posizionale,

$$v = d_0 + B^1 * d_1 + B^2 * d_2 + B^3 * d_3 + \dots$$
- che si può riscrivere come

$$v = d_0 + B * (d_1 + B * (d_2 + B * (d_3 + \dots)))$$

Conseguenze

- d_0 si può ricavare come **resto della divisione intera v / B**
- quindi, il quoziente di tale divisione è

$$q = d_1 + B * (d_2 + B * (d_3 + \dots))$$
- perciò, **le altre cifre** si possono perciò ottenere **iterando il procedimento**
 - le cifre vengono prodotte nell'ordine dalla *meno significativa* (LSB) alla *più significativa* (MSB)

Algoritmo di conversione - Metodo operativo

Per convertire il numero v in una stringa di cifre

che ne rappresentino il valore in base B

- **si divide v per B**
 - *il resto* costituisce la cifra meno significativa (LSB)
 - *il quoziente* serve a iterare il procedimento
- **se tale quoziente è zero, l'algoritmo termina;**
se non lo è, lo si assume come nuovo valore v'
- **si itera il procedimento** con il valore v' .

Esempi

Numero	Base	Calcolo valore	Stringa
<i>quindici</i>	4	$15 / 4 = 3$ con resto 2 $3 / 4 = 0$ con resto 3	32
<i>undici</i>	2	$11 / 2 = 5$ con resto 1 $5 / 2 = 2$ con resto 1 $2 / 2 = 1$ con resto 0 $1 / 2 = 0$ con resto 1	1011
<i>sessantatre</i>	10	$63 / 10 = 6$ con resto 3 $6 / 10 = 0$ con resto 6	63
<i>sessantatre</i>	16	$63 / 16 = 3$ con resto 15 $3 / 16 = 0$ con resto 3	3F

Esempi di rappresentazioni in diverse basi

Numero	Rappr. Base 2	Rappr. Base 8	Rappr. Base 16
<i>uno</i>	1	1	1
<i>due</i>	10	2	2
<i>tre</i>	11	3	3
<i>quattro</i>	100	4	4
<i>cinque</i>	101	5	5
<i>otto</i>	1000	10	8
<i>dieci</i>	1010	12	A
<i>quindici</i>	1111	17	F
<i>sedici</i>	10000	20	10
<i>trentuno</i>	11111	37	1F
<i>trentadue</i>	100000	40	20
<i>cento</i>	1100100	144	64
<i>duecentocinquantacinque</i>	11111111	377	FF

Osservazione

- in generale, la rappresentazione di un numero in due basi B1 e B2 è completamente diversa (anche se l'alfabeto A1 è contenuto nell'alfabeto A2, o viceversa)
- **ma la situazione cambia** se le due basi sono **una potenza dell'altra**:

le rappresentazioni *di uno stesso numero su basi che sono una potenza dell'altra* sono **strettamente correlate**.

Relazione fra rappresentazioni di un numero in diverse basi

La relazione fondamentale

- se $B1 = B2^n$, **ogni cifra** nella rappresentazione R1 corrisponde a ***n* cifre** nella rappresentazione R2
 - ogni cifra *esadecimale* corrisponde a 4 cifre binarie
 - ogni cifra *ottale* corrisponde a 3 cifre binarie

Conseguenze

- se $B1 = B2^n$, *per passare dalla rappresentazione di un numero in base B1 a quella in base B2* (o viceversa) **non è necessario** applicare l'algoritmo di conversione, **ma si può agire direttamente**
- **sostituendo ordinatamente ogni cifra di R1 con gruppi di *n* cifre di R2.**

Esempi

Rappr. Base 2	Rappr. Base 2	Rappr. Base 8	Rappr. Base 2	Rappr. Base 16
1	1	1	1	1
10	10	2	10	2
11	11	3	11	3
100	100	4	100	4
101	101	5	101	5
1000	1 000	1 0	1000	8
1010	1 010	1 2	1010	A
1111	1 111	1 7	1111	F
10000	10 000	2 0	1 0000	1 0
11111	11 111	3 7	1 1111	1 F
100000	100 000	4 0	10 0000	2 0
1100100	1 100 100	1 4 4	110 0100	6 4
11111111	11 111 111	2 7 7	1111 1111	F F

Operazioni aritmetiche

Quali regole?

- **tutte** le notazioni **posizionali** utilizzano per le operazioni **le stesse regole**, *indipendentemente dalla base* di rappresentazione adottata
- quindi, **le regole già note** per la familiare rappresentazione in base 10 **restano valide**.

Esempi di somme e sottrazioni

15	+	0000	1111	+	0F	+	
21	=	0001	0101	=	15	=	
36		0010	0100		24		
36	-	0010	0100	-	24	-	
21	=	0001	0101	=	15	=	
15		0000	1111		0F		

Operazioni aritmetiche - Moltiplicazioni e divisioni

- **concettualmente**, operando "con carta e matita", le operazioni si possono svolgere con le usuali regole, essendo queste ultime indipendenti dalla base adottata
- **un elaboratore, però**, può non essere in grado di svolgerle direttamente in quel modo (ciò richiede circuiti appositi), nel qual caso diviene necessario *scomporre* tali operazioni in *mosse più semplici* (esempio: moltiplicazione realizzata come serie di somme)

Errori nelle operazioni

- **in matematica**, le operazioni sui numeri interi (senza segno) non danno mai luogo a errori
(la divisione però è una divisione "intera", che può produrre un *resto*)
- ma possono essere *impossibili*:
 - sottrazione con minuendo maggiore del sottraendo (esempio: 3-8)
 - divisione per zero
- **in un elaboratore**, invece, *possono generarsi degli errori*, perché è impossibile rappresentare *tutti gli infiniti numeri*
 - in particolare, con n bit, **esiste un massimo numero rappresentabile, che è 2^n-1**

Esempio

<i>teoricamente</i>		<i>praticamente</i>	
<i>\decimale</i>	<i>binario</i>	<i>decimale</i>	<i>binario</i>
176+	1011 0000+	176+	1011 0000+
84=	0101 0100=	84=	0101 0100=
----	-----	----	-----
260	10000 0100	004	0000 0100

- **Il risultato è completamente errato**, perché è andato perso *il contributo più significativo!!*
- **La causa** è che i registri dell'elaboratore hanno posto per n bit, e quindi *il bit di riporto* che si genera *viene perduto*: è l' **OVERFLOW** (straripamento)
- All'overflow non c'è rimedio: per evitarlo si può solo *usare un maggior numero di bit*
per questo il C definisce vari tipi numerici diversi.

Numeri interi

Numeri interi in un elaboratore: problematiche

- come rappresentare il “segno meno”
- possibilmente, rendere semplice l'esecuzione delle operazioni

Due tipi di rappresentazioni

- rappresentazione *in modulo e segno*
 - ◆ semplice e intuitiva
 - ◆ ma inefficiente e complessa nella gestione delle operazioni → *non molto usata in pratica!*
- **rappresentazione in complemento a due**
 - ◆ meno intuitiva, costruita ad hoc con un trucco matematico
 - ◆ però rende semplice la gestione delle operazioni → *largamente usata!!*

In pratica, nella notazione in complemento a due, un numero negativo ha una rappresentazione che fa in modo che esso, sommato a un numero positivo, dia un risultato che sia a tutti gli effetti la “differenza”)

$X - Y$ diventa uguale a $X + (Y \text{ rappresentato in complemento a due})$

Rappresentazione in modulo e segno

Caratteristiche generali

- usa UN BIT per *rappresentare esplicitamente il segno* (es: 0 = +, 1 = -)
- usa poi gli altri bit disponibili per *rappresentare il valore assoluto* come numero binario puro
- esempio:
 - ◆ 8 bit (MSB = segno, bit 6...bit 0 = valore assoluto)
 - ◆ -2 → 1 0000010
 - ◆ +5 → 0 0000101
- note:
 - ◆ *segno completamente DISGIUNTO* dal valore assoluto
 - ◆ posizione del bit di segno entro la stringa di bit **IRRILEVANTE** (in linea di principio)

Difetti

- il valore 0 ha due distinte rappresentazioni (10000000 = “-0” e 00000000 = “+0”)
- non permette di usare direttamente gli algoritmi già noti per eseguire le operazioni
 - ◆ in particolare, con le usuali regole di calcolo non è vero che $X + (-X) = 0$:

+5	0	0000101	
-5	1	0000101	
---		-----	
0	1	0001010	??? -10 ???

- e quindi:
 - ◆ richiede circuiti specifici (o software più complesso) per la realizzazione dei sommatore
 - ◆ maggior complicazione, maggior costo → praticamente *non molto utilizzata*.

Notazione in Complemento a due

Scopo

- poter utilizzare direttamente gli algoritmi dei numeri naturali per eseguire le operazioni
- in particolare, rendere verificata la proprietà che $X + (-X) = 0$ usando le regole aritmetiche standard
anche a prezzo di una notazione più complessa.

La notazione

- *rappresentazione non (completamente) posizionale*
- il bit più significativo di una stringa di n bit ha peso -2^{n-1} anziché 2^{n-1} (gli altri bit mantengono il peso che è loro proprio, come in binario puro)
- esempio:
 - ◆ utilizzando $n=8$ bit, il bit 7 ha peso **-128** anziché +128
 - ◆ quindi la stringa 11110001 denota il valore: $-128 + 64 + 32 + 16 + 1 = -15$

Il valore denotato

- per definizione, il valore di un intero v espresso in questa notazione è dato dalla formula:

$$v = -d_{n-1}2^{n-1} + \sum_{k=0}^{n-2} d_k 2^k$$

dove i d_k ($k=0..n-1$) sono le cifre binarie della rappresentazione del numero.

- la formula differisce da quella usata per i numeri naturali *solo* per il peso negativo del MSB.

Notazione in Complemento a due - Caratteristiche ed esempi

Conseguenze

- **MSB = 0** → **numero positivo** (stesso valore che si avrebbe in binario puro: il diverso peso del MSB non ha influenza)
- **MSB = 1** → **numero negativo** (il valore rappresentato si ottiene sommando il contributo negativo del MSB con i contributi positivi degli altri bit)

Esempi

- la stringa 11110001 denota il valore $-128 + 64 + 32 + 16 + 1$,
cioè -15
- la stringa 01110001 denota il valore $64 + 32 + 16 + 1$,
cioè 113
- la stringa 10000000 denota il valore $-128 + 0$,
- la stringa 11111111 denota il valore $-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$,
cioè -1
- la stringa 00000000 denota il valore 0 ,
- la stringa 01111111 denota il valore $64 + 32 + 16 + 8 + 4 + 2 + 1$,
cioè 127.

Osservazioni

- valori opposti, come 15 e -15, hanno rappresentazioni completamente diverse
- rappresentazioni identiche a meno del MSB denotano valori interi completamente diversi (non deve stupire: solo la rappresentazione in modulo e segno conserva le “somiglianze”).

Notazione in Complemento a due - Proprietà

Campo di valori rappresentabili

- Nei due casi:
 - ◆ MSB=0 → $n-1$ bit usabili come in binario puro → range da 0 a $2^{n-1}-1$
 - ◆ MSB=1 → stesso intervallo traslato di -2^{n-1} → range da -2^{n-1} a $2^{n-1}-1$
 - ◆ Totale: range da -2^{n-1} a $2^{n-1}-1$
- Esempio:
 - ◆ 8 bit → 256 combinazioni → range -128...127 (anziché, come in binario puro, 0...255)
 - ◆ 256 combinazioni allocate metà ai positivi e metà ai negativi, anziché tutte ai positivi
- Nota:
 - ◆ il massimo intero *positivo* rappresentabile è di uno inferiore al minimo *negativo* rappresentabile perché lo 0 fa intrinsecamente parte dei positivi.

Proprietà

- lo zero ha ora un'unica rappresentazione (efficienza, semplicità)
- *le somme algebriche si possono eseguire con le stesse regole dell'aritmetica binaria*
 - ◆ tali regole rendono **verificata la proprietà $X + (-X) = 0$** (con una piccola convenzione!)
 - ◆ non è necessario nessun circuito particolare per trattare i negativi
 - ◆ semplicità e basso costo.

Notazione in Complemento a due - Un primo esempio

Operazione: -5 +3

- la rappresentazione di +3 è nota (è identica a quella che si avrebbe in binario puro)
- ma come ricavare la rappresentazione di -5?

Per ricavare la rappresentazione di un intero negativo:

- l'unico contributo negativo possibile è il -128 del MSB
- quindi, per ottenere -5 occorre determinare gli altri bit in modo che rappresentino il positivo +123
- così,

$$-5 = -128 + 123 \rightarrow 1\ 1111011$$

- operazione:

$$\begin{array}{r}
 -5 + \quad 11111011 \\
 +3 = \quad 00000011 \\
 \hline
 -2 \quad 11111110
 \end{array}$$

- e in effetti:

$$1\ 1111110 \rightarrow -128 + 126 = -2$$

Funziona!

Notazione in Complemento a due - Perché funziona

La motivazione di fondo

- i valori positivi sono rappresentati come se la notazione fosse binaria pura
- i valori negativi hanno l'MSB che pesa -2^{n-1} (-128) [anziché $+2^{n-1}$ (+128) come nel caso dei numeri naturali]
- quindi, fra le due interpretazioni *della stessa stringa di bit* vi è una differenza di 2^n (256)
- esempio

$10110110 = -128 + 54 = -74$ (se interpretato in notazione complemento a due)

$10110110 = 128 + 54 = 182$ (se interpretato in binario puro)

Perché funzionano somme e sottrazioni

- **Usando le regole dell'aritmetica binaria** (valide per i naturali) **per sommare o sottrarre valori rappresentati in notazione complemento a due:**
 - ◆ per i positivi, nulla cambia;
 - ◆ per i negativi, si introduce un “errore” pari a 2^n (è come operare non sul negativo $-X$, ma sul positivo 2^n-X)
 - ◆ tale “errore” però **non ha influenza** perché **lavorando su n bit di fatto si opera modulo 2^n**
- resta solo un “inestetismo”:
 - ◆ possono generarsi *riporti o prestiti* oltre l'MSB, che potranno (e dovranno) essere *ignorati*.
- **Nota:** nessuno ha mai detto che funzionino anche *le altre operazioni* (moltiplicazione, divisione...)

Notazione in Complemento a due - Rappresentazione dei negativi

Come determinare la rappresentazione di un intero negativo?

- La definizione formale è precisa, ma poco pratica
- Una macchina ha bisogno di un procedimento più semplice e facilmente meccanizzabile
→ poter risalire alla rappresentazione del negativo $-X$ a partire da quella del positivo X .

Osservazione chiave

- La rappresentazione in notazione complemento a due del negativo $-X$ è espressa dalla stessa stringa che rappresenta, in binario puro, il positivo $Z = 2^n - X$
 - ◆ Esempio: la stringa 11110001, che denota il valore -15 in notazione complemento a due, denota il valore $Z = 256 - 15 = 241$ se interpretata come valore binario puro.

→ **Per trovare la rappresentazione dell'intero negativo $-X$ basta calcolare la rappresentazione (in binario puro) del positivo $2^n - X$**

Ma come si fa a eseguire la sottrazione $2^n - X$? Siamo da capo!!!

Perché $2^n - X$ non è un problema

- $2^n - X$ si può riscrivere come $(2^n - 1 - X) + 1$
- l'operazione $(2^n - 1 - X)$ è una sottrazione solo in apparenza, perché $(2^n - 1)$ è una sequenza di n uni
- quindi, $(2^n - 1 - X)$ si esegue facilmente *invertendo tutti i bit della rappresentazione di X*
- dopo di che, per ottenere 2^n basta aggiungere 1.

Notazione in Complemento a due - L'algoritmo pratico di calcolo di -X

- 1. determinare la rappresentazione binaria del positivo +X**
- 2. *invertire tutti i bit di tale rappresentazione***
- 3. *aggiungere 1 al risultato così ottenuto.***

- Esempio: determinazione della rappresentazione di -15 (su $n=8$ bit)
 - 1) $15 \rightarrow 00001111$
 - 2) *inversione* $\rightarrow 11110000$
 - 3) *incremento di 1* $\rightarrow 11110001$ (verifica: $-128 + 113 = -15$)

Note

- *il procedimento funziona anche al contrario*
 - ◆ data la rappresentazione di -X, invertendo i bit e sommando 1 si ottiene la rappresentazione di X
- la *notazione* in complemento a due non va confusa con *l'operazione* di complementazione a due
 - ◆ la notazione, definita formalmente come sopra, detta la regole per la rappresentazione di *tutti* i numeri interi (positivi e negativi)
 - ◆ l'effettuazione del *calcolo* del complemento a due, secondo l'algoritmo ora definito, serve invece a ottenere la rappresentazione del negativo -X a partire da quella del positivo X (e viceversa).

Notazione in Complemento a due - Errori nei calcoli

Cosa può succedere

- esempio 1

$$\begin{array}{r}
 -65 + \quad 10111111 \\
 -65 = \quad 10111111 \\
 --- \\
 -130 \quad (1)01111110 \quad (+126) \quad \text{È sbagliato!!}
 \end{array}$$

Siamo oltre -128 !

- esempio 2

$$\begin{array}{r}
 +65 + \quad 01000001 \\
 +65 = \quad 01000001 \\
 --- \\
 +130 \quad 10000010 \quad (-126) \quad \text{È sbagliato!!}
 \end{array}$$

Siamo oltre +127 !

Perché succede

- Con n bit, il range dei valori rappresentabili è $-2^{n-1} \dots 2^{n-1}-1$
- ma sommando due valori in quell'intervallo *il risultato può uscire da tale range*
- nel qual caso si ha *invasione del bit di segno* → **OVERFLOW**
 - ◆ si può dimostrare che l'overflow accade quando c'è un riporto *oltre MSB* senza che ci sia stato anche un riporto *verso l'MSB*
 - ◆ in altri termini, perché tutto funzioni occorre che o l'MSB non sia coinvolto in nessun riporto (né generandolo né ricevendolo), oppure che lo sia "totalmente" (ossia lo riceva e lo rigeneri).