Interi negativi, numeri reali (rappresentazione macchina)

Liceo G.B. Brocchi – Bassano del Grappa (VI)
Liceo Scientifico – opzione scienze applicate
Giovanni Mazzocchin

Interi con segno

- Finora abbiamo interpretato le stringhe di bit contenute nei registri e nelle celle di memoria soltanto come numeri interi positivi:
  - ad esempio, la sequenza di bit `1100`, finora, ha sempre rappresentato il numero decimale `12`
- Ovviamente, vorremmo scrivere programmi che siano in grado di lavorare anche sugli interi negativi, sui reali, sui caratteri, sulle stringhe di caratteri, etc…
- Dobbiamo studiare dei metodi per rappresentare tutti questi oggetti della realtà diversi dagli interi positivi, ma sempre tramite sequenze di `1` e `0`, perché internamente un calcolatore digitale non conosce niente altro oltre ai bit. Non possiamo scrivere cose come *virgole* e *segni meno* in memoria…

Codifica con segno e modulo (*sign-magnitude*)

- Nella rappresentazione con **segno e modulo,** il bit più a sinistra (più significativo – **MSB** – ***M**ost **S**ignificant **B**it*) rappresenta il segno del numero, in questo modo:
  - **`0` per il +**
  - **`1` per il –**
  - i bit rimanenti rappresentano il modulo (valore assoluto) del numero

- Bisogna stabilire a priori quanti bit si utilizzano per la codifica

- Ipotizziamo di voler rappresentare un range di numeri interi con segno utilizzando 4 bit

Codifica con segno e modulo

abbiamo utilizzato 4 bit per rappresentare un range di interi relativi con segno e modulo

ovviamente, le disposizioni sono sempre 16 (2^4), ma se prima le utilizzavamo per rappresentare soltanto i numeri interi positivi nell'intervallo `[0 – 15]`, ora le usiamo per rappresentare i numeri relativi nell'intervallo `[−7 – +7]`

prima di questa lezione, `1111` significava `15`, mentre con questa rappresentazione significa `−7`

decimal	binary – sign-magnitude
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 0
+1	**0** 0 0 1
+0	**0** 0 0 0
-0	**1** 0 0 0
-1	**1** 0 0 1
-2	**1** 0 1 0
-3	**1** 0 1 1
-4	**1** 1 0 0
-5	**1** 1 0 1
-6	**1** 1 1 0
-7	**1** 1 1 1

Codifica con segno e modulo

NB: lo `0` è rappresentato 2 volte, come `+0` e come `−0`. Non è utile rappresentare 2 volte lo `0`, che non è né negativo, né positivo

questa rappresentazione è molto scomoda per una macchina, che deve valutare molto spesso se il risultato di un'operazione è `0`

decimal	binary – sign-magnitude
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 0
+1	**0** 0 0 1
+0	**0** 0 0 0
-0	**1** 0 0 0
-1	**1** 0 0 1
-2	**1** 0 1 0
-3	**1** 0 1 1
-4	**1** 1 0 0
-5	**1** 1 0 1
-6	**1** 1 1 0
-7	**1** 1 1 1

Codifica con segno e modulo

con n bit, la rappresentazione con segno e modulo degli interi relativi codifica l'intervallo:
$[-2^{n-1} + 1,\ 2^{n-1} - 1]$

in questo esempio: $[-7,\ +7]$

decimal	binary – sign-magnitude
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 0
+1	**0** 0 0 1
+0	**0** 0 0 0
-0	**1** 0 0 0
-1	**1** 0 0 1
-2	**1** 0 1 0
-3	**1** 0 1 1
-4	**1** 1 0 0
-5	**1** 1 0 1
-6	**1** 1 1 0
-7	**1** 1 1 1

Complementazione

- La complementazione è una tecnica utilizzata per codificare intervalli di interi relativi

- Esplicitiamo il concetto in base 10:
  - il complemento a 9 di una cifra decimale x è la cifra y t.c. $x + y = 9$

Complementazione

digit	9's-complement
0	9
1	8
2	7
3	6
4	5
5	4
6	3
7	2
8	1
9	0

Complementazione

- La complementazione a 9 in base 10 è analoga alla complementazione a 1 in base 2

digit	1's-complement
0	1
1	0

Complementazione

- Per calcolare il complemento a 9 di un numero decimale, si complementa a 9 ciascuna delle sue cifre
- Per calcolare il complemento a 1 di un numero binario, si complementa a 1 ciascuna delle sue cifre

- La somma tra un numero decimale e il suo complemento a 9 dà come risultato…
- La somma tra un numero binario e il suo complemento a 1 dà come risultato…

Complemento a 2

- **Complemento a 2**: per calcolare il complemento a 2 di un numero binario, si calcola il suo complemento a 1, e gli si aggiunge 1 (ignorando l'eventuale overflow)
- Esempi:
  - $twos\_complement(101011) = 010100 + 1 = 010101$
  - $twos\_complement(111111) = 000000 + 1 = 000001$

- Il complemento a 2 in binario è analogo al complemento a 10 in decimale

Complemento a 2

- La somma tra un numero decimale e il suo complemento a 10 dà come risultato…
- La somma tra un numero binario e il suo complemento a 2 dà come risultato…

Complemento a 2

- Procedura per calcolare velocemente il complemento a 2:
  - partire dalla cifra più a destra e mantenere le cifre inalterate fino a che non si incontra il primo 1, compreso. Dopodiché, invertire tutte le cifre seguenti (verso sinistra)

- Applicare la procedura su:
  - 10100111
  - 1110001001
  - 00000000

Complemento a 2

- Il complemento a 2 è molto utilizzato nei calcolatori elettronici digitali per rappresentare i numeri negativi

Codifica in complemento a 2:
la rappresentazione degli interi positivi è identica a quella in segno e modulo. Per rappresentare un intero negativo, si complementa a 2 il corrispondente intero positivo

Complemento a 2

- **Esempio**: rappresentare -6_{dec} in complemento a 2 su 4 bit:
  - rappresentare 6_{dec} : $\texttt{0110}_{\texttt{bin}}$
  - il complemento a 2 di `0110` è `1010`
  - quindi, la rappresentazione del numero richiesto in complemento a 2 è `1010`
  - **NB**: come per la rappresentazione *sign-magnitude*, il MSB indica il segno

Complemento a 2

decimal	binary – two's complement
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 0
+1	**0** 0 0 1
0	**0** 0 0 0
-1	**1** 1 1 1
-2	**1** 1 1 0
-3	**1** 1 0 1
-4	**1** 1 0 0
-5	**1** 0 1 1
-6	**1** 0 1 0
-7	**1** 0 0 1
-8	**1** 0 0 0

ora abbiamo solo una rappresentazione dello 0

notare che l'intero più piccolo rappresentabile in complemento a 2, con 4 bit, è -8:
abbiamo quindi utilizzato la disposizione che prima rappresentava -0 per rappresentare un numero negativo in più

Metodo dei complementi

- Il metodo dei complementi è una tecnica utilizzata per semplificare le operazioni aritmetiche nei calcolatori elettronici digitali (e un tempo nelle calcolatrici meccaniche)

Numeri macchina

- Ipotizziamo di dover rappresentare il numero 12367.7891
- Possiamo rappresentarlo in notazione esponenziale in base 10, nel seguente modo:
  - $1.2367789E + 04 = 1.2367789 \cdot 10^4$
- Consideriamo ora il numero 0.00000526. Possiamo rappresentarlo in notazione esponenziale in base 10, nel seguente modo:
  - $5.26E - 06$
- Consideriamo ora il numero 526000000. Possiamo rappresentarlo in notazione esponenziale in base 10, nel seguente modo:
  - $5.26E + 08$
- Questa notazione permette di rappresentare il maggior numero di cifre significative (cifre che forniscono l'informazione rilevante sul numero) in uno spazio di memoria limitato

Numeri macchina

- I numeri reali, all'interno di una macchina elettronica digitale (computer/calcolatrice), sono rappresentati utilizzando una quantità limitata di locazioni di memoria
- L'aritmetica di un computer, quindi, è necessariamente diversa dall'aritmetica insegnata nei corsi di Matematica, questo perché in Matematica i numeri reali sono infiniti e hanno un ordinamento continuo, mentre i numeri di un determinato elaboratore elettronico costituiscono un insieme finito e hanno un ordinamento discreto
- Un computer non potrà mai rappresentare esattamente numeri irrazionali come $\sqrt{2}$ e $\sqrt{3}$. Dovrà perciò approssimarli (per *arrotondamento* o per *troncamento*)

Numeri macchina

- **Rappresentazione in virgola fissa:** questa rappresentazione prevede che la locazione che memorizza il numero sia suddivisa in bit di segno, bit della parte intera, bit della parte frazionaria
- Immaginiamo di avere a disposizione 4 bit per la parte intera e 3 bit per la parte decimale:
  - `+101.101` è rappresentabile. Conversione in decimale:
    - $2^2 + 2^0 + 2^{-1} + 2^{-3} = 5.625$
  - `100.100001` non è rappresentabile

Rappresentazione in virgola mobile

- Rappresentazione in **virgola mobile** (***floating-point***):
  - in un sistema di numerazione in base $\boldsymbol{b}$, qualunque numero $\boldsymbol{n}$ si può esprimere nella forma:

$$n = m \cdot b^e$$

  - dove:
    - m: mantissa
    - e: esponente

- La notazione in cui la parte intera della mantissa è 0, e la cifra più significativa del numero da rappresentare si trova subito a destra della virgola, viene detta forma normalizzata (i.e. notazione scientifica)
- La virgola è mobile perché può essere spostata di un numero arbitrario di posizioni, scalando l'esponente di conseguenza

Reali - rappresentazione in virgola mobile

- **Esempi**:
  - $3.14 = 0.314 \cdot 10$
  - $1941 = 0.1941 \cdot 10^4$
- Da cosa dipende il segno dell'esponente?
- L'intervallo di numeri rappresentabili dipende dal numero di bit riservati all'esponente, mentre la precisione dipende dal numero di bit riservati alla mantissa
- **NB**: i numeri in floating-point non costituiscono un continuo come i numeri reali

Reali - rappresentazione in virgola mobile

- **IEEE 754 single precision**: tipo **`float`** del C (32 bit)
- **IEEE 754 double precision**: tipo **`double`** del C (64 bit)
  - 1 bit per il segno
  - 11 bit per l'esponente (*characteristic*)
  - 52 bit per la mantissa
  - non serve rappresentare la base, dato che è sempre 2
- **IEEE**: ***I****nstitute of* ***E****lectrical and* ***E****lectronics* ***E****ngineers*

Reali - rappresentazione in virgola mobile

- Aprire una shell Python e lanciare i seguenti comandi:
  - `import math`
  - `math.sqrt(3) * math.sqrt(3)`
  - `0.3 - 0.2`
  - `0.3 - 0.2 == 0.1 - 0.0`
  - `math.sqrt(2) * math.sqrt(2)`
  - `2 ** 4 == math.sqrt(2) ** 8`
  - `2 ** 3 == math.sqrt(2) ** 6`

Reali - rappresentazione in virgola mobile

- Eseguire questo programma Python e commentarne il comportamento:

```
x = 1.0
step = 0.2
while x != 2.0:
    x += step
```

IEEE 754 double precision – 64 bit

```
0 10000000011 1011100100010000000000000000000000000000000000000000
```

- `0`: sign bit
- `10000000011`: characteristic's bits
- `1011100100010000000000000000000000000000000000000000`: mantissa bits

- per convertirlo in decimale esiste una formula, che non vedremo