Esercitazione Informatica I
AA 2012-2013

Nicola Paoletti

4 Gigno 2013

Conversioni

Effettuare le seguenti conversioni, tenendo conto del numero di bit con cui si rappresenta il numero da convertire/convertito.

1. $\boxed{\text{12 bit}}$ $(7\text{B9})_H = (?)_2 = (?)_8 = (?)_{10}$ (unsigned/M+S/complemento a 2)
2. $\boxed{\text{11 bit}}$ $(7\text{B9})_H = (?)_{10}$ (segno + modulo) $= (?)_{10}$ (complemento a 2)

Aritmetica binaria

1. Calcolare la somma tra i seguenti interi con segno a 20 bit rappresentati in modulo e segno. **A3B1C + 34567**
2. Calcolare la somma tra i seguenti interi senza segno a 18 bit. **1475A + 298BF + 0BB45**
3. Calcolare la stessa somma nel caso in cui gli operandi siano rappresentati in complemento a 2.

Floating point

1. Si rappresenti in formato IEEE 754 single precision il valore $(56{,}25)_{10}$
2. Dato il seguente valore rappresentato in IEEE 754 single precision, si determini il valore in decimale con notazione normalizzata $x,y \times 10^z$. **97180000**.

Algoritmi e programmazione JavaScript

1. Si implementi una funzione `ricerca` che prende in ingresso gli argomenti `a` (un array) e `el`, e restituisce `true` se `el` è un elemento dell'array `a`, `false` altrimenti. La ricerca deve avvenire scorrendo tutti gli elementi dell'array e verificando che l'elemento corrente sia uguale all'argomento `el`.

Conversioni - Soluzioni

1) Da notare che $(7B9)_H$ su 12 bit rappresenta la stessa quantità nel caso di rappresentazione unsigned, modulo + segno e complemento a 2 (in quanto positivo, ultimo bit a 0).
   - $(7B9)_H = (\boxed{\mathbf{0111}}\boxed{\mathbf{1011}}\boxed{\mathbf{1001}})_2$
   - $(\boxed{011}\boxed{110}\boxed{111}\boxed{001})_2 = (\mathbf{3671})_8$
   - $(011110111001)_2 = (2^0 + 2^3 + 2^4 + 2^5 + 2^7 + 2^8 + 2^9 + 2^{10})_{10} = (1 + 8 + 16 + 32 + 128 + 256 + 512 + 1024)_{10} = (\mathbf{1977})_{10}$. Notate che la stessa conversione potrebbe essere fatta con un numero minore di operazioni, infatti $(11110111001)_2 = (11111111111 - 1000110)_2$. Quindi $(11110111001)_2 = ((2^{11} - 1) - 2^6 - 2^2 - 2^1)_{10} = (2047 - 64 - 4 - 2)_{10} = (1977)_{10}$

2) Su 11 bit, invece $(7B9)_H$ è negativo (ultimo bit a 1, $(7B9)_H = (\mathbf{1}1110111001)_2$), quindi a seconda della rappresentazione (M+S o complemento a 2), la quantità sottintesa cambia.

M+S: $(\mathbf{1}1110111001)_2 = -(01110111001)_2 = -(2^0+2^3+2^4+2^5+2^7+2^8+2^9)_{10} = \mathbf{-953}_{10}$.

Complemento a 2: $(\mathbf{1}1110111001)_2 = -f_2(11110111001)_2$. $f_1(11110111001) = 00001000110$. $f_2(11110111001) = 00001000111$. $-(00001000111)_2 = -(2^0 + 2^1 + 2^2 + 2^6)_{10} = (-\mathbf{71})_{10}$. Il complemento a 2 poteva essere calcolato direttamente a gruppi di cifre esadecimali, $f_1(7B9_H) = 046_H$, notando che per la cifra più significativa abbiamo solo 3 bit (e quindi $(111)_2 - 7$ e non $(1111)_2 - 7$).

Aritmetica binaria - Soluzioni

1) In generale ci sono due modi utili per risolvere una somma tra interi rappresentati modulo + segno:

   – determinare il segno del risultato a priori, ovvero il segno dell'operando con modulo maggiore, eseguendo la somma dei moduli (se hanno segno concorde), o la differenza tra il modulo maggiore e il minore. $A3B1C = -23B1C$, quindi $A3B1C + 34567 = 34567 - 23B1C$ (segno del risultato positivo).

0000	1	0	1		
011	4	5	6	7	-
010	3	B	1	C	=
0001	**0**	**A**	**4**	**B**	

Sottrazione esadecimale tra 34567 e 23B1C. L'ultima cifra esadecimale è in bit per verificare in modo più chiaro eventuali overflow. Nel caso di rappresentazione modulo e segno, il bit di segno va omesso in quanto non viene modificato dall'operazione, ma è determinato a priori. La prima riga mostra il riporto in entrata. In questo caso l'overflow è assente, dato che il riporto fuori l'ultimo bit è 0.

Quindi, $A3B1C + 34567 = \mathbf{10A4B}$

   – oppure sommando gli operandi convertiti in complemento a 2 (naturalmente valgono regole diverse da M+S per determinare l'overflow. Nel nostro caso, $34567 > 0$, quindi la sua rappresentazione C_2 è la stessa. Invece, $A3B1C = -23B1C = f_2(23B1C) = f_1(23B1C)+1 = DC4E3+1 = \mathbf{DC4E4}$. Quindi la somma C_2 $DC4E4 + 34567$ dovrà restituire il risultato precedente.

11111	0	1	0		
1101	C	4	E	4	+
0011	4	5	6	7	=
0001	**0**	**A**	**4**	**B**	

Nel caso di rappresentazione con complemento a 2, bisogna rappresentare il bit di segno in quanto questo viene sommato come ogni altro bit. Non si ha overflow (riporto sul bit di segno = riporto fuori dal bit di segno).

2) $1475A + 298BF + 0BB45$, 18 bit unsigned.

000	1	1	1		
01	4	7	5	A	+
10	9	8	B	F	=
111	0	0	0		
11	**E**	**0**	**1**	**9**	+
00	B	B	4	5	=
00	**9**	**B**	**5**	**E**	

Abbiamo overflow nella seconda addizione (riporto pari a 1 fuori dal MSB).

3) Nel caso della rappresentazione con complemento a 2, l'operazione non cambia. Ma cambia il criterio con cui determiniamo eventuali overflow. Infatti, concludiamo che in questo caso in entrambe le somme, **non occorre overflow** (i riporti sul bit di segno e fuori dal bit di segno coincidono).

Floating point - Soluzioni

Per comodità, ricordo che le potenze negative del 2 si ottengono dividendo per 2 (anzichè moltiplicare), quindi:

1	**0.5**	**0.25**	**0.125**	**0.0625**	**0.03125**	**0.015625**	...
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	...

Quindi per ottenere la rappresentazione binaria della parte non intera potete:

- sottrarre potenze negative finchè non si ottiene 0
- moltiplicare per un'opportuna potenza di 2, in modo tale da ottenere un valore intero

---

1) "Sottrazioni successive": $0.25 = 2^{-2} = (0.01)_2$, oppure
"Moltiplicazioni successive": $0.25 \times 2^2 = 1$, quindi $(0.25)_{10} = (1)_2 \times 2^{-2} = (0.01)_2$. Quindi $(56.25)_{10} = (111000.01)_2 = \mathbf{1.1100001} \times \mathbf{2^5}$.
   - Mantissa: 1100001;
   - esponente (eccesso -127): $5 + 127 = 132_{10} = 10000100_2$;
   - segno: 0

s	**e**								**m**																			
0	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
4				**2**					**6**				**1**				**0**				**0**				**0**			

Il risultato è quindi **42610000**.

2) Facciamo il procedimento inverso al precedente.

9				7				1				8				0				0				0				0			
1	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
s	**e**							**m**																							

- Mantissa: 00110 . . .;
- esponente (eccesso -127): $00101110_2 = (46)_{10} \rightarrow 46 - 127 = -81$;
- segno: 1 (negativo)

Quindi avremmo $-1.0011_2 \times 2^{-81} = -0.10011_2 \times 2^{-80} = -(2^{-1} + 2^{-4} + 2^{-5} = 0.59375)_{10} \times 2^{-80}$.
Possiamo ottenere la notazione desiderata con le approssimazioni conosciute ($2^{80} \approx 10^{24}$):

$$-0.59375 \times 2^{-80} \approx -0.59375 \times 10^{-24} = \mathbf{-5.9375 \times 10^{-25}}$$

Algoritmi e programmazione JavaScript - Soluzioni

```
function ricerca(a, el){
  for(var i=0; i<a.length; i++)
    if(a[i]==el)
      return true;
  return false;
}
```

Ci sono soluzioni equivalenti (ma non ugualmente efficienti), ad esempio ci si può servire di una variabile booleana `trovato`, che è `false` finchè l'elemento non è stato trovato, `true` altrimenti:

```
function ricerca(a, el){
  var trovato=false;
  for(var i=0; i<a.length; i++)
    if(a[i]==el)
      trovato=true;
  return trovato;
}
```

Questo codice può essere reso più efficiente inserendo un `break` nel momento in cui l'elemento viene trovato. L'istruzione `break` causa l'uscita dal ciclo prima che la condizione di uscita (in questo caso `i<a.length` diventi vera. In questo modo, la seguente implementazione è a livello di prestazioni, pari alla prima fornita. Infatti l'array viene visitato per intero solo nel caso peggiore, ovvero quello in cui l'elemento `el` non viene trovato. Non appena lo trova esce dal ciclo e restituisce `true`.

```
function ricerca(a, el){
  var trovato=false;
  for(var i=0; i<a.length; i++)
    if(a[i]==el){
      trovato=true;
```

```
      break;
    }
  return trovato;
}
```

Errore comune: il seguente codice non è corretto, infatti restituisce `false` non appena non trova l'elemento `el`. Ovvero, se non lo trova alla prima posizione, restituisce subito `false` senza controllare le posizioni dell'array successive.

```
function ricerca(a, el){
  var trovato=false;
  for(var i=0; i<a.length; i++)
    if(a[i]==el)
      return true;
    else
      return false;
}
```