Rappresentazione delle informazioni

Sistemi digitali

- Rappresentazione delle informazioni in un elaboratore classico:
  - Digitale – Ogni elemento base può assumere un numero finito di valori
  - Binaria – Il numero di valori che un elemento base può avere è 2, indicati per convenzione con “1” e “0”, oppure “high”, “low”
- Unità base di rappresentazione delle informazione: il *bit*.

- Un computer quantistico usa invece il *qubit*, di cui non ci occupiamo.

Sistemi digitali

- Tutte le informazioni sono rappresentate da sequenze di bit
  - Byte → 8 bit → 01101001
  - Word → 32 bit → 10010110101001011010100101101011

- Ogni sequenza può assumere un numero limitato di valori:
  - 1 byte → 2^8= 256 combinazioni
  - 2 byte → 2^{16} = 65536 combinazioni
  - 1 word = 4 byte → 2^{32} = 4294967296 combinazioni

Rappresentazione dei numeri

- In generale un numero non può essere rappresentato esattamente, ma sempre con una certa approssimazione.
- Questo perché, la dimensione di una variabile, una cella di memoria o un registro è limitata. Esiste quindi un insieme di valori rappresentabili, a seconda della codifica.

  Es:
  - Numeri interi positivi da 0 a 255
  - Numeri interi da -128 a +127
  - Numeri decimali a due cifre da 0.1 a 9.9

Rappresentazione dei numeri

- Effettuando operazioni in un insieme finito, posso avere diverse condizioni di errore:
  - Overflow – risultato maggiore del più grande valore rappresentabile
  - Underflow – risultato minore del più piccolo valore rappresentabile
  - Non appartenenza all’insieme, pur non essendoci né underflow né overflow

Rappresentazione numeri interi

- Un numero ha necessariamente precisione finita, dipendente dalla sua rappresentazione (e dall’architettura usata)
  – Intero positivo a 16 bit → da 0 a 65535
  – Intero positivo a 32 bit → da 0 a 4294967295

- Come passare dalla rappresentazione decimale alla rappresentazione binaria?

Notazione posizionale

- *Rappresentazione in cui ogni cifra ha un peso diverso a seconda della posizione. Ogni cifra può assumere il range di valori [0, … , base – 1].*
- Ad esempio, in base 10 ogni cifra può assumere i valori [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].
- In generale, un numero espresso come sequenza di cifre

$$c_n c_{n-1} \ldots c_1 c_0$$

nella base *b* rappresenta il valore

$$N = b^n c_n + \ldots + b^1 c_1 + b^0 c_0 = \sum_{i=0}^{n} b^i c_i$$

Cifra più significativa
(MSB – Most Significant Bit)

Cifra meno significativa
(LSB – Least Significant Bit)

Notazione posizionale

In base 10:

$$
\begin{aligned}
75 &= 10^1 \cdot 7 + 10^0 \cdot 5 \\
693 &= 10^2 \cdot 6 + 10^1 \cdot 9 + 10^0 \cdot 3 \\
91864973 &= 10^7 \cdot 9 + 10^6 \cdot 1 + 10^5 \cdot 8 + 10^4 \cdot 6 \\
&\quad + 10^3 \cdot 4 + 10^2 \cdot 9 + 10^1 \cdot 7 + 10^0 \cdot 3
\end{aligned}
$$

Notazione posizionale in altre basi

- Nel caso vogliamo rappresentare un numero in base 2, ogni cifra può essere 0 o 1.
- In base 16, oltre alle cifre da 0 a 9, per convenzione si usano le prime 6 lettere dell’alfabeto, quindi le cifre disponibili sono [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F] senza differenza tra maiuscole e minuscole.
- Esempi:

  0xff4586, 0xFF4586, 0x00001a5b, 0xaadef000

Cifra	Valore
A	10
B	11
C	12
D	13
E	14
F	15

Notazione posizionale

- Quando la base è diversa da 10, possiamo specificare la base con la dicitura *numero*$_{(base)}$, es. 144_{16} oppure 144_{sedici}
- Le basi più usate in ambito informatico sono:
  - Base 2 (prefisso *0b)*

    $01010010_{(2)} = 0b01010010$

    $2^7 \cdot 0 + 2^6 \cdot 1 + 2^5 \cdot 0 + 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 0 = 82$
  - Base 16 (prefisso *0x)*

    $AB52_{(16)} = 0xAB52$

    $16^3 \cdot 10 + 16^2 \cdot 11 + 16^1 \cdot 5 + 16^0 \cdot 2 = 43858$

Conversione da base 10 a base n

- Per convertire un numero decimale *N* in base *b* si calcola ogni cifra dividendo *N* per *b* e prendendo il resto, finché il risultato non è 0.
- Esempio:
  - convertire N=23 in base b=2

23 / 2 = 11 resto 1
11 / 2 = 5 resto 1
5 / 2 = 2 resto 1
2 / 2 = 1 resto 0
1 / 2 = 0 resto 1

Verso di lettura

Cifra meno significativa

$23 = 10111_{(2)} = 0b10111$

Cifra più significativa

Conversione da base 10 a base n

- Esempio:
  - convertire 277455 in base 16

Cifra meno significativa

277455 / 16 = 17340 resto 15=0xF
17340 / 16 = 1083 resto 12=0xC
1083 / 16 = 67 resto 11=0xB
67 / 16 = 4 resto 3
4 / 16 = 0 resto 4

$277455_{(10)} = 43BCF_{(16)} = 0x43BCF$

Cifra più significativa

Conversione da base n a base 10

- Per convertire un numero in base *b* in notazione decimale si applica la formula della notazione posizionale.
- Esempio:
  - convertire 0x1F7 in base 10

$$
\begin{aligned}
N = \sum_{i=0}^{n} b^i c_i &= 16^2 \cdot 1_{(16)} + 16^1 \cdot F_{(16)} + 16^0 \cdot 7_{(16)} \\
&= 16^2 \cdot 1 + 16^1 \cdot 15 + 16^0 \cdot 7 \\
&= 256 \cdot 1 + 16 \cdot 15 + 1 \cdot 7 \\
&= 156 + 240 + 7 \\
&= 503
\end{aligned}
$$

Perché queste basi particolari?

- Base 2:
  - Esprime l'informazione come viene elaborata dalla macchina, utile per debug e operazioni binarie
  - Utile per rappresentare in maniera compatta variabili booleane (flag)
  - Utile per interfacciamento con periferiche

- Base 16
  - Dovuta al fatto di usare logica binaria ($16 = 2^4$) e byte composti da 8 bit (1 cifra = 4 bit, 2 cifre = 1 byte)
  - Comoda per rappresentare grandi numeri in maniera compatta (es. indirizzi di memoria)

0x1248

0b 0001 0010 0100 1000

Addizione di numeri interi

- Cambiando la base di rappresentazione le regole di addizione non cambiano:

  0 + 0 = 0

  1 + 0 = 1

  0 + 1 = 1

  1 + 1 = 10 → 0 con riporto di 1

  Il bit di riporto è anche chiamato bit di ***carry***

Addizione di numeri interi

riporto

	11	1
1000 +	1001 +	1010 +
0011 =	0011 =	1000 =
1011	1100	10010

overflow

Tipi di dato intero

- Senza segno (unsigned)
  - Comprende tutti i numeri positivi rappresentabili in ***N*** bit secondo la rappresentazione binaria.
  - Copre intervallo da 0 a 2^{N-1}
  - Es.
    - in C il tipo standard ***uint8_t*** può rappresentare numeri da 0 a 255
    - in C il tipo standard ***uint64_t*** può rappresentare numeri da 0 a 18446744073709551615

  Conviene sempre usare i tipo ***uintXX_t*** quando si vuole essere sicuri della dimensione delle variabili (portabilità)

Tipi di dato intero con segno

- Con segno (signed)
  - Comprende sia numeri interi positivi che negativi, usando la codifica in complemento a 2 per i negativi.
  - Copre l'intervallo da -2^{N-1} a $+2^{N-1}-1$
  - Es.
    - in C il tipo standard ***int16_t*** può rappresentare numeri da -32768 a 32767
    - in C il tipo standard ***int32_t*** può rappresentare numeri da -2147483648 a 2147483647

  Conviene sempre usare i tipo ***intXX_t*** quando si vuole essere sicuri della dimensione delle variabili (portabilità)

Codifica complemento a 2

- Codifica che semplifica le operazioni tra numeri interi
- Un numero negativo ***N***<*0* viene codificato in ***n*** bit come

$$\sim(-N) + 1$$

- Il bit più significativo indica sempre il segno
  - 0 → il numero è positivo
  - 1 → il numero è negativo

Es. n = 8

$87_{10} = 01010111_2$

$-87_{10} = \sim 01010111_2 + 1_2 = 10101000_2 + 1_2 = 11101001_2$

$-1_{10} = \sim 00000001_2 + 1_2 = 11111110_2 + 1_2 = 11111111_2$

Sottrazione di numeri interi

- In generale valgono le regole classiche:

  0 – 0 = 0   1 – 0 = 1

  0 – 1 = 1 → prestito di 1   1 – 1 = 0

- La codifica complemento a 2 permette di eseguire la sottrazione in maniera semplificata, ovvero come somma del complemento

prestito

```
111
1000 -                1000 +         8 -
0011 = → c2 →         1101 =         3 =
______               ______         ___
0101                 10101          5
```

Scarto il bit di carry

Moltiplicazione di numeri interi

- Anche qui il procedimento è analogo alla base 10

```
   1110 x
   0110 =
  ______
   0000 +
  1110  +
 1110   +
0000    =
_________
1010100
```

- Ogni risultato parziale o è zero o è il moltiplicando spostato progressivamente a sinistra.
- La moltiplicazione si riduce ad una serie di somme e spostamenti a sinistra.
- Il numero di bit necessari per il risultato è al massimo uguale alla somma del numero di bit dei due operandi
- Es. per contenere il risultato della moltiplicazione di due numeri a 32 bit sono necessari al massimo 64 bit.

Operazioni bit-a-bit

- Una sequenza di n bit può essere considerata
  - come un'unica entità (es. un numero senza segno)
  - considerando i bit separatamente, come una sequenza di variabili booleane (***flags***)
  - come una sequenza di bit da manipolare in vari modi
- Nel secondo caso si possono definire operazioni logiche che agiscono su tutti i bit in maniera indipendente → operazioni bit-a-bit o ***bitwise***
  - AND
  - OR
  - XOR
  - NOT
- Nel terzo caso le più comuni sono le operazioni di traslazione (***shift***)

Operazioni logiche

- Agiscono sulla *rappresentazione binaria* delle variabili, su ogni singolo bit o a coppie:

```
a = 6  = 0b00000110
b = 98 = 0b01100010
```

Diverse operazioni logiche:

NOT	
b1	~b1
0	1
1	0

AND		
b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

OR		
b1	b2	b1 | b2
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
b1	b2	b1 ^ b2
0	0	0
0	1	1
1	0	1
1	1	0

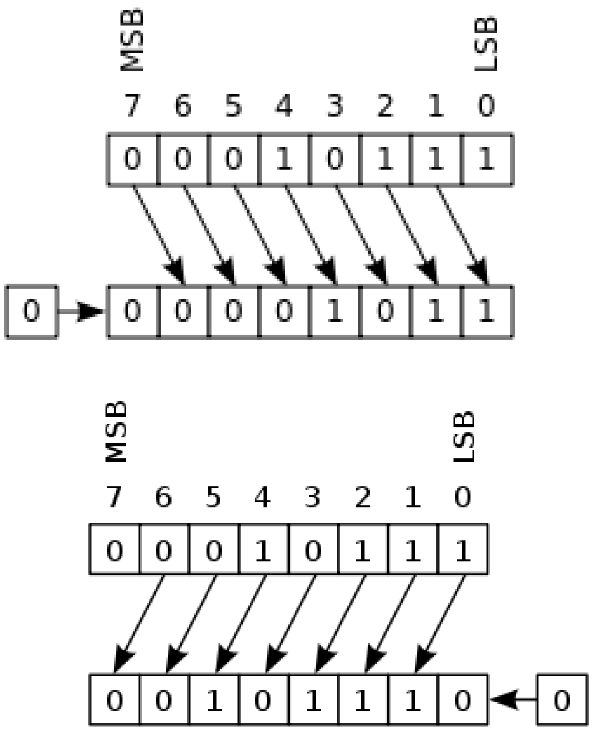
Operazioni bitwise

- Manipolazione flag
  - Mascheramento `0101 & 1100 = 0100`
  - Set/reset flag `0101 | 0010 = 0111`
    `0101 & ~0010 = 0101`
  - Inversione selettiva `0101 ^ 0011 = 0110`

Operazioni di shift

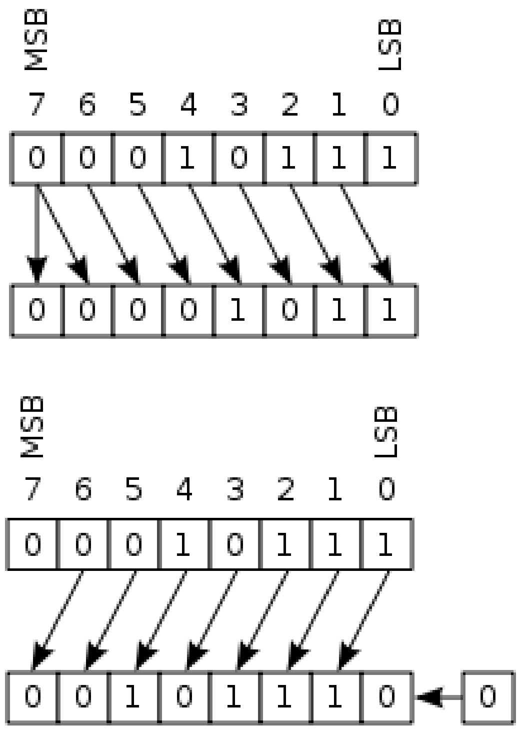
- Le operazioni di shift agiscono complessivamente su tutti i bit di una sequenza, traslandoli a destra o sinistra

- Shift **logico**:
  - L'unico scopo è traslare i bit
  - Per convenzione si inserisce uno 0 sia a sinistra che a destra



Operazioni di shift

- Le operazioni di shift agiscono complessivamente su tutti i bit di una sequenza, traslandoli a destra o sinistra
- Shift **aritmetico**:
  - Pensato per trattare numeri con segno in complemento a 2
  - Lo shift a destra mantiene il bit di segno
  - Lo shift a sinistra è equivalente allo shift logico



Operazioni bitwise

- Shift
  - Una traslazione della rappresentazione binaria corrisponde ad una moltiplicazione o divisione intera per una potenza di 2, in generale **solo per interi positivi**
  - Shift logico → numeri senza segno

`0b00001011 << 3 = 0b01011000`

$$11 \cdot 2^3 = 88$$

  - Shift aritmetico → numeri con segno

`0b10001011 >> 3 = 0b11110001`

$$-116 / 2^3 = -14$$

Esempi bitwise

Resettare i 4 bit più significativi di 0x43

```
0x43 = 0b01000011
0b01000011 & 0b00001111 = 0b00000011
0x43 & 0x0F = 0x03
```

Invertire i 2 bit meno significativi di 0x25

```
0x25 = 0b00100101
0b00100101 ^ 0b00000011 = 0b00100110
0x25 ^ 0x03 = 0x26
```

Esempi bitwise

Eseguire shift logico a sinistra di 3 posizioni di 0x19

```
0x19 = 0b00011001
0b00011001 << 3 = 0b11001000
0x19 << 3 = 0xC8
```

Eseguire shift aritmetico a destra di 3 posizioni di -1

$-1 = 0b11111111$

$0b11111111 >>_{arith} 3 = 0b11111111$

$-1 >>_{arith} 3 = -1$

Risultato inaspettato, ma solo perché la nostra interpretazione è in complemento a 2

Rappresentazione di numeri razionali

Sono numeri esprimibili come rapporto di due numeri interi.

L'insieme dei numeri razionali contiene, oltre ai numeri interi, i numeri decimali (numeri con virgola).

In un sistema posizionale con base *b*, *n* cifre intere ed *m* cifre frazionarie, i numeri razionali sono così rappresentati:

$$\begin{aligned} N &= b^{n-1}c_{n-1} + \ldots + b^1 c_1 + b^0 c_0 + b^{-1}c_{-1} + \ldots + b^{-m}c_{-m} \\ &= \sum_{i=-m}^{n-1} b^i c_i \end{aligned}$$

La notazione con *n* + *m* cifre è detta a virgola fissa (**fixed point**).

Esempio

In base 10:

$$3.141592 = 10^0 \cdot 3 + 10^{-1} \cdot 1 + 10^{-2} \cdot 4 + 10^{-3} \cdot 1$$
$$+ 10^{-4} \cdot 5 + 10^{-5} \cdot 9 + 10^{-6} \cdot 2$$
$$0.000000000063 = 10^{-11} \cdot 6 + 10^{-12} \cdot 3$$

In base 2:

$$10.10_{due} = 2^1 \cdot 1 + 2^0 \cdot 0 + 2^{-1} \cdot 1 + 2^{-2} \cdot 0 = 2.5$$
$$0.0001_{due} = 2^{-4} \cdot 1 = 0.0625$$

Conversione da base 10 a base n

Estendo l'algoritmo visto per i numeri interi, separando la parte intera da quella decimale

Esempio: convertire 23.375 in base 2

23 / 2 = 11 resto 1
11 / 2 = 5 resto 1
5 / 2 = 2 resto 1
2 / 2 = 1 resto 0
1 / 2 = 0 resto 1

$23 = 10111_{(2)}$

$0.375 \cdot 2 = 0.75$ parte intera 0
$0.75 \cdot 2 = 1.5$ parte intera 1
$0.5 \cdot 2 = 1.0$ parte intera 1

$.375 = .011_{(2)}$

Ottengo $23.375_{(10)} = 10111.011_{(2)}$

Conversione da base 10 a base n

E se converto $1/3 = 0.\overline{3}_{(10)}$ in base 3?

$0.\overline{3} \cdot 3 = 1.0$ parte intera 1 $\qquad \Rightarrow \qquad 0.\overline{3}_{(10)} = 0.1_{(3)}$

In generale, per ogni base *b*, ci sono dei numeri razionali che non si possono rappresentare in maniera esatta con un numero finito di cifre, ma hanno una rappresentazione periodica.

Si dimostra che una rappresentazione con un numero finito di cifre nella base *b* di un numero $N > 0$ esiste se e solo se esistono due interi *l, m* tali che

$$N = \frac{l}{b^m}$$

Conversione da base 10 a base n

Converto 3.1 in base 2

3 / 2 = 1 resto 1
1 / 2 = 1 resto 0

$3 = 10_{(2)}$

0.1 · 2 = 0.2 parte intera 0
0.2 · 2 = 0.4 parte intera 0
0.4 · 2 = 0.8 parte intera 0
0.8 · 2 = 1.6 parte intera 1
0.6 · 2 = 1.2 parte intera 1
0.2 · 2 = 0.4 parte intera 0
0.4 · 2 = 0.8 parte intera 0
0.8 · 2 = 1.6 parte intera 1
….

$0.1 = .0001100110011\ldots_{(2)}$

Virgola fissa

- Se si considera base 2 diventa più difficile da maneggiare per i programmatori, spesso si usa virgola fissa in base 10, ovvero si esprimono I valori in decimi, centesimi, millesimi, …
- La notazione in virgola fissa non permette di rappresentare numeri molto grandi o molto piccoli:

  Es. con n=5 e m=3 posso rappresentare i numeri compresi tra -99999.999 e 99999.999, ma non:

    0.000001 perché ho solo 3 cifre dopo la virgola

    1000000 perché ho solo 5 cifre per la parte intera

- Questo può facilmente causare overflow o underflow quando si eseguono dei calcoli

Virgola mobile

- Per ovviare ad alcuni di questi problemi (ottimizzando le risorse a disposizione) si è inventata la cosiddetta notazione in virgola mobile (floating point).
- Si considera la notazione scientifica in base *b*:

$$N = \pm m \cdot b^e$$

  - *b* → base del sistema di numerazione
  - *m* → mantissa del numero
  - *e* → esponente (intero con segno)

Virgola mobile

- Fissata la base, per rappresentare un numero reale è necessario rappresentare segno, mantissa ed esponente.
- La mantissa si suppone in virgola fissa con una sola cifra non nulla a sinistra dell virgola (**notazione scientifica normalizzata**).
- L'intervallo di valori della mantissa è quindi: 1 <= M < b

$$\begin{aligned}
19.375_{10} &= (1.9375 \cdot 10^{1})_{10} \\
0.0031_{10} &= (3.1 \cdot 10^{-3})_{10} \\
4.5_{10} &= (4.5 \cdot 10^{1})_{10} \\
-112.45_{10} &= (-1.1245 \cdot 10^{2})_{10} \\
110011.001_{2} &= (1.0011001 \cdot 10^{101})_{2} \\
&= 51.125 = (5.1125 \cdot 10^{1})_{10}
\end{aligned}$$

Virgola mobile

- Si possono rappresentare numeri con ordini di grandezza molto differenti utilizzando per la rappresentazione un insieme limitato di cifre
- Si possono implementare in maniera molto efficiente e trasparente al programmatore
- Alcuni numeri rimangono rappresentabili solo in maniera approssimata, però si aumenta di molto il range rappresentabile

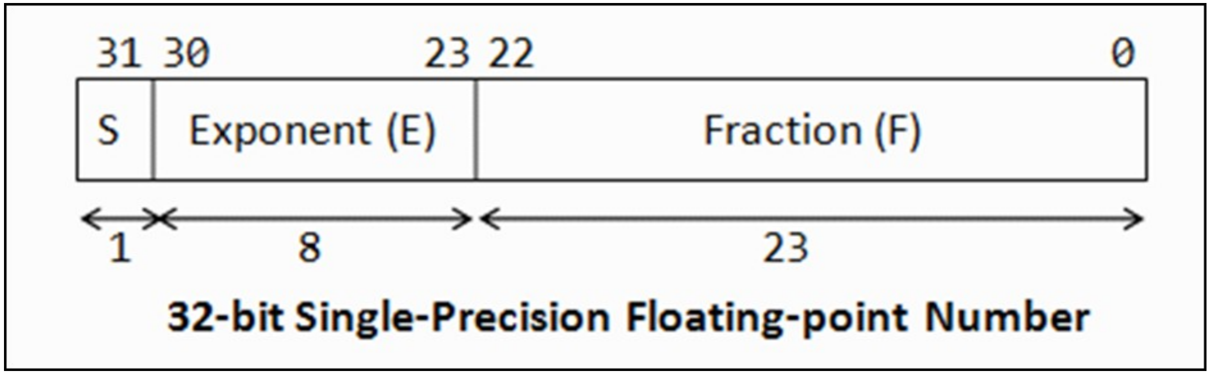
Virgola mobile

- Osservazioni:
  - la mantissa determina la precisione
  - l'esponente determina l'intervallo dei valori rappresentabili
  - moltiplicare un numero per una potenza della base equivale a far scorrere (**shift**) il numero di un numero di posizioni pari all'esponente (destra o sinistra, dipende dal segno):

$$(1.9375 \cdot 10^{2})_{(10)} = (1\underline{93}.75)_{(10)}$$

- si possono rappresentare numeri con ordini di grandezza molto differenti utilizzando per la rappresentazione un insieme limitato di cifre

IEEE 754 singola precisione



- Rappresentazione floating point a 32 bit
  - 1 bit di segno (***S***)
  - 8 bit di esponente (***E***)
  - 23 bit di mantissa (***M***)

$$N = (-1)^S \cdot (1.M) \cdot 2^{E-bias}$$

IEEE 754 singola precisione

Rappresentazione floating point a 32 bit

$$N = (-1)^S \cdot (1.M) \cdot 2^{E-bias}$$

- **S** specifica il segno del numero:
  - S = 0 $\rightarrow$ numero positivo
  - S = 1 $\rightarrow$numero negativo

  (come per i numeri in complemento a 2)

IEEE 754 singola precisione

Rappresentazione floating point a 32 bit

$$N = (-1)^S \cdot (1.M) \cdot 2^{E-bias}$$

- **M** rappresenta la mantissa:
  - nella notazione scientifica normalizzata la mantissa è compresa tra [1, *b*)
  - In base 2 la parte intera sarà sempre 1, si può quindi omettere (bit nascosto)
  - si codifica solo la parte frazionaria in 23 bit, quindi sarà

$$0.0_{(2)} \leq M < 1.0_{(2)}$$

IEEE 754 singola precisione

Rappresentazione floating point a 32 bit

$$N = (-1)^S \cdot (1.M) \cdot 2^{E-bias}$$

- **E** rappresenta l'esponente:
  - Può essere negativo; invece del complemento a 2 si aggiunge un offset (*bias)* di 127
  - i valori di E normalmente vanno da 1 a 254 (0 e 255 sono riservati)
  - I valori dell'esponente senza il bias sono compresi tra -126 (1-bias) e 127 (254-bias)

IEEE 754 singola precisione

Rappresentazione floating point a 32 bit

$$N = (-1)^S \cdot (1.M) \cdot 2^{E-bias}$$

- **Bias** va calcolato in base al range floating point
  - È desiderabile un range simmetrico quindi prendo il bias come metà del range dell'esponente
  - Regola semplificata: prendo come bias il numero che ha solo MSB a 0
  - In single precision ho esponente a 8 bit, bias = 0b01111111 = 127

IEEE 754 singola precisione

Ci sono diversi tipi di numero rappresentabili in floating point:

Categoria	Esponente	Mantissa
Zeri	0	0
Numeri Denormalizzati	0	!= 0
Numeri normalizzati (caso standard)	[1, 254]	qualunque
Infiniti	255	0
NaN (Not a Number)	255	!= 0

IEEE 754 singola precisione

Casi particolari:

- nello standard IEEE-754 sono presenti due tipologie di zeri e di infiniti, a seconda del segno: +∞ , -∞ , +0 , -0
- NaN (not a number) è il risultato di un’operazione non valida, ad esempio 0/0
- i numeri ***denormalizzati*** rappresentano numeri più piccoli di quelli rappresentabili nella forma standard. In questa configurazione il valore del numero è calcolato come:

$$N = (-1)^S \cdot (0.M) \cdot 2^{-126}$$

- come si può notare la mantissa ha la parte intera a 0 e l’esponente è fissato a -126

Confronto tra floating points

- L'ordine dei campi segno, esponente, mantissa esiste per un motivo preciso: **rende molto semplice il confronto tra due floating points**
  - i campi sono in ordine di "importanza"
  - si confrontano i due numeri bit a bit (eccezione fatta per il segno, che segue una logica inversa)
- Es. il primo numero è più piccolo del secondo:

  0.01234 = 0 01111000 10010100010110110110110

  0.32870 = 0 01111101 01010000100101101011110

Altri formati floating point

- IEEE 754 doppia precisione – 64 bit di rappresentazione
  - 1 bit per ***S***
  - 11 bit per ***E***
  - 52 bit per ***M***
- x86 doppia precisione estesa – 80 bit di rappresentazione
  - 1 bit per ***S***
  - 15 bit per ***E***
  - 64 bit per ***M*** (include parte intera)
- IEEE 754 quadrupla precisione – 128 bit di rappresentazione
  - 1 bit per ***S***
  - 15 bit per ***E***
  - 112 bit per ***M***

Altri formati floating point

- Per esercizio possiamo usare un formato ridotto - 8 bit di rappresentazione
  - 1 bit per ***S***
  - 3 bit per ***E***
  - 4 bit per ***M***

Conversione da floating point a decimale (fixed point)

- Si identificano i valori binari di ***S***, ***E*** e ***M***

  0b 0 010 0011 → ***S*** = *0*, ***E*** = $010_{(2)}$, ***M*** = $.0011_{(2)}$

- Si aggiunge la parte intera alla mantissa

  M = $1.0011_{(2)} = 2^{0} + 2^{-3} + 2^{-4} = 1.1875$

- Si sottrae il bias all'esponente

  E = $010_{(2)} - 011_{(2)} = 2 - 3 = -1$

- Si calcola il valore complessivo (in decimale), tenendo conto del bit di segno

$$N = (-1)^0 \cdot 1.1875 \cdot 2^{-1} = 0.59375$$

Conversione da decimale a floating point

- Si converte il numero in fixed-point base 2

  11.75 → 1011.11$_{(2)}$

- Si calcola l'esponente normalizzando la rappresentazione fixed-point

  $1011.11_{(2)} = (1.01111 \cdot 10^{11})_{(2)}$ → *spostare la virgola di 3 posizioni ovvero 011*$_{(2)}$

- Si aggiunge il bias all'esponente

  $\boldsymbol{E} = 011_{(2)} + 011_{(2)} = 110_{(2)}$

- Si calcola la mantissa rimuovendo la parte intera e troncando al numero di bit di mantissa

  $\boldsymbol{M} = .0111_{(2)}$ → *prendo i 4 bit più significativi della parte frazionaria*

- Si aggiunge il bit di segno e si assemblano i vari campi

  S *= 0 → numero positivo*

11.75 → 0 110 0111

Intervallo di valori singola precisione

- Gli esponenti 00000000 e 11111111 sono riservati
- Valore più piccolo (in valore assoluto)
  - E = $00000001_{(due)}$ → 1 - bias = 1 - 127 = -126
  - M = 000...00 → $(1.0)_{(due)}$

  $\pm(1.0)_{(due)} \bullet 2^{-126} \approx \pm 1.175494 \bullet 10^{-38}$ → sotto è underflow (denormalizzazione a parte)

- Valore più grande (in valore assoluto)
  - E = $11111110_{(due)}$ → 254 - bias = 254 - 127 = 127
  - M = 111...11 → $(1.1111...111)_{(due)} \approx (10.0)_{(due)}$

  $\pm(10.0)_{(due)} \bullet 2^{127} \approx \pm 3.402823 \bullet 10^{38}$ → sopra è overflow

- In C solitamente corrisponde al tipo ***float***

Intervallo di valori doppia precisione

- Gli esponenti 00000000000 e 11111111111 sono riservati
- Valore più piccolo (in valore assoluto)
  - $E = 00000000001_{(due)} \rightarrow 1 - bias = 1 - 1023 = -1022$
  - $M = 000...00 \rightarrow (1.0)_{(due)}$

  $\pm(1.0)_{(due)} \bullet 2^{-1022} \approx \pm 2.225073 \bullet 10^{-308}$ → sotto è underflow (denormalizzazione a parte)

- Valore più grande (in valore assoluto)
  - $E = 11111111110_{(due)} \rightarrow 2046 - bias = 2046 - 1023 = 1023$
  - $M = 111...11 \rightarrow (1.1111...111)_{(due)} \approx (10.0)_{(due)}$

  $\pm(10.0)_{(due)} \bullet 2^{1023} \approx \pm 1.797693 \bullet 10^{308}$ → sopra è overflow

- In C solitamente corrisponde al tipo ***double***

Esercizio
decimale → floating point 8 bit

- Codificare π = 3.141592

3 / 2 = 1, → 1
1 / 2 = 0, → 1

0.141592 · 2 = 0.283184 → 0
0.283184 · 2 = 0.566368 → 0
0.566368 · 2 = 1.132736 → 1
0.132736 · 2 = 0.265472 → 0
0.265472 · 2 = 0.530944 → 0
0.530944 · 2 = 1.061888 → 1
0.061888 · 2 = 0.123776 → 0
…

- Normalizzo:

  1.10010010 → 1 shift sx

- Calcolo esponente:

  ***E** = 0b001 + 0b011 = 0b100*

- Calcolo mantissa:

  ***M** = .1001*

- Codifica finale:

  0 100 1001

Esercizio
floating point 8 bit → decimale

Verifico il valore dell'esercizio precedente: 01001001

$\boldsymbol{S} = 0$, $\boldsymbol{E} = 100_{(2)}$, $\boldsymbol{M} = 1001_{(2)}$

Ricostruisco mantissa

$\boldsymbol{M} = 1.1001_{(2)} = 2^{0} + 2^{-1} + 2^{-4} = 1.5625$

Si sottrae il bias all'esponente

$\boldsymbol{E} = 100_{(2)} - 011_{(2)} = 4 - 3 = 1$

Calcolo valore complessivo

$$N = (-1)^0 \cdot 1.5625 \cdot 2^1 = 3.125 \neq 3.141592\tilde{!}!!!$$

Costanti floating point

- In C (e in assembler) si possono definire costanti floating point:

  `#define    E    2.718281`

  che vengono solitamente codificate come **float** o **double**.

  Attenzione al fatto che in realtà il valore utilizzato potrebbe essere leggermente diverso, a causa dell’arrotondamento dato dalla conversione in floating point!

- Esercizio: codificare la costante ***e*** (numero di Nepero) come floating point a 8, 32 e 64 bit
- Qual è l’errore dato dalla conversione?

Costanti floating point

```
#define   E   2.718281828459045235360287471 35
```

- FP 8 bit:

0 100 1011 = 2.6875

errore ~ 3.08 • 10^{-2}

- FP 32 bit:

0 10000000 01011011111100001010100 0 =
2.71828174591064453125

errore ~ 8.25 • 10^{-8}

Costanti floating point

```
#define    E    2.71828182845904523536028747135
```

- FP 64 bit:

  0 10000000000 01011011111100001010100010110001010001010111011010010
  = 2.718281828459045090795598298427648842334747314453125

  errore ~ 1.45 • 10^{-16}

- Notare il (basso) numero di cifre corrette rispetto alle cifre totali in notazione decimale!
- Il numero di cifre decimali corrette, in questo caso, è ridotto a 1 per FP 8 bit, 7 per FP 32 bit e 16 per FP 64 bit

Costanti floating point

Posso stimare l'errore di conversione di una costante?

La rappresentazione floating point più vicina al numero vero può differire al massimo di 1 LSB della mantissa, il cui valore però cambia con l'esponente.

- Nel caso della costante ***e***:

FP 8 bit: errore ~ $3.08 \cdot 10^{-2}$ 1 LSB mantissa = $2^{-4} = 6.25 \cdot 10^{-2}$

FP 32 bit: errore ~ $8.25 \cdot 10^{-8}$ 1 LSB mantissa = $2^{-23} \sim 1.19 \cdot 10^{-7}$

FP 64 bit: errore ~ $1.45 \cdot 10^{-16}$ 1 LSB mantissa = $2^{-52} \sim 2.22 \cdot 10^{-16}$

- Errore di conversione sempre minore del peso dell'LSB della mantissa, fissato l'esponente

Operazioni aritmetiche con floating point

- Le operazioni aritmetiche vengono eseguite in “fixed point” usando come operandi la mantissa dei due numeri
- L’esponente deve essere compatibile, quindi devo denormalizzare un operando
- Posso avere comportamenti indesiderati nel caso opero con numeri di ordini di grandezza (e quindi esponenti) molto diversi

Operazioni aritmetiche con floating point - Addizione

- Step 1: Se necessario, denormalizzo numero con esponente minore
- Per denormalizzare traslo a destra la mantissa e incremento l'esponente di un numero di step pari alla differenza tra gli esponenti
- Devo prima estrarre i valori reali!

Tolgo bias dall'esponente e aggiungo unità intera sulla mantissa

$N_1 = 5.5 = 0\ 101\ 0110 \rightarrow \boldsymbol{S_1} = 0$, $\boldsymbol{E_1} = 010_{(2)}$, $\boldsymbol{M_1} = 1.0110_{(2)}$

$\rightarrow \boldsymbol{E_1} = 011_{(2)}$, $\boldsymbol{M_1} = 0.1011_{(2)}$

Bit "nascosto"

$N_2 = 9.5 = 0\ 110\ 0110 \rightarrow \boldsymbol{S_2} = 0$, $\boldsymbol{E_2} = 011_{(2)}$, $\boldsymbol{M_2} = 1.0011_{(2)}$

Operazioni aritmetiche con floating point - Addizione

- Step 2: Eseguo l'addizione tra le due mantisse, che ora sono relative allo stesso esponente

```
0.1011 +
1.0011 =
--------
1.1110
```

- Step 2a: Se necessario, dopo l'addizione bisogna rinormalizzare il risultato e riaggiustare l'esponente (non in questo esempio)

  Risultato: 0 110 1110 = 15

Arrotondamento

- La de-normalizzazione di un addendo o la rinormalizzazione come descritti prima sono equivalenti ad un arrotondamento per troncamento (metodo semplice)
- In IEEE 754 sono previsti metodi alternativi più sofisticati per l’arrotondamento
- Metodo GRS:
  - 3 bit addizionali (**G**uard, **R**ound, **S**ticky), “estendono” la mantissa
  - Indipendenti dalla precisione del floating point (singola, doppia)

Arrotondamento floating point
Metodo GRS

1.XXXXXXXXXXXXXXXXXXXXXXX 0 0 0

Bit nascosto mantissa Guard Round Sticky

- Quando la mantissa eccede i bit a disposizione, i bit in eccesso vengono traslati nei 3 bit GRS; non appena un 1 raggiunge il bit sticky, questo non cambia più valore
- Il valore finale decide come arrotondare il risultato:
  - 0xx → arrotondamento per difetto (troncamento)
  - 100 → se LSB mantissa = 1 arrotonda per eccesso, altrimenti per difetto
  - 101, 110, 111 → arrotondamento per eccesso (mantissa + 1)

Rappresentazione caratteri - Codice ASCII

- Una delle prime codifiche per i caratteri
- Rappresenta sia caratteri stampabili che caratteri di controllo

- Stampabili: `ABCDE.,&%$/("£)`
- Controllo: `\r \n \t`

- Ogni carattere corrisponde a 7 bit, rappresentabile in un byte
- In C è usato per il tipo di dato ***char***

- Tabella completa: https://upload.wikimedia.org/wikipedia/commons/d/dd/ASCII-Table.svg

Rappresentazione caratteri - Codice ASCII

- Usando codifica ASCII è possibile scambiare dati tra due elaboratori
- I primi 32 caratteri da 0x00 a 0x1F sono caratteri di controllo
- Da 0x20 a 0x7F sono caratteri stampabili:

```
printf(“Hello World!\n”);
```

```
0X48 0x65 0x6C 0x6C 0x6F 0x20 0x57 0x6F 0x72 0x6C 0x64 0x21 0x0A
```

Rappresentazione caratteri - Codice ASCII

- Essendo originariamente progettato per la lingua inglese mancano lettere accentate e altri simboli
- ASCII esteso: aggiungo altri 128 simboli da 0x80 a 0xFF
  - Diverse tabelle caratteri, definite nello standard ISO 8859
  - 8859-1 → EU occidentale (solitamente usata per italiano)
  - 8859-2 → EU centro/est, es. polacco, serbo

Rappresentazione caratteri UNICODE

- Sistema di codifica universale, gestito dall’Unicode Consortium
- Pensato per codificare ogni simbolo in maniera indipendente da lingua e sistemi hardware/software usati
- Codifica non solo caratteri per ogni lingua conosciuta ma anche simboli matematici, ideogrammi, Braille e molto altro
- Varie codifiche, a lunghezza variabile o fissa, in genere retrocompatibili con ASCII nei primi 127 simboli
  - UTF-8 → molto diffusa
  - UTF-16
  - ...