

# Tirocinio su Posit

Luigi Leonardi

## Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Posit . . . . .	2
1.2	Note sui Test . . . . .	2
<b>2</b>	<b>Test</b>	<b>3</b>
2.1	Accuratezza . . . . .	3
2.1.1	Svolgimento . . . . .	3
2.1.2	Conclusioni . . . . .	3
2.1.3	Codice . . . . .	4
<b>3</b>	<b>Codice?</b>	<b>4</b>
3.1	Test Sigmoide . . . . .	4

# 1 Introduzione

## 1.1 Posit

Il Posit è un formato di numero in virgola mobile ideato da John Gustafson, in alternativa allo standard IEEE 754. L'idea di base è fondamentalmente la stessa, anche nei Posit è presente un bit per il segno, dei bit per l'esponente e dei bit per la mantissa, le principali differenze consistono nella presenza di un "super esponente" o regime e nel non avere un numero di bit fissato per quest'ultimo e per la mantissa.

Il vantaggio nell'avere un super esponente, la cui lunghezza non è definita,

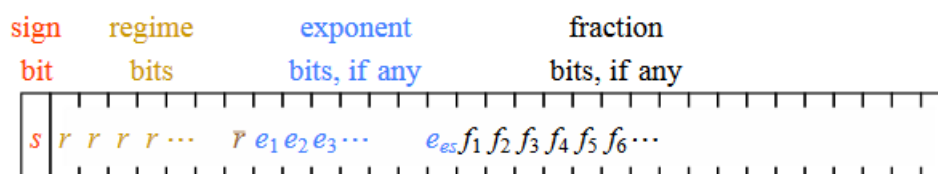


Figura 1: "Formato Posit"

permette di ottenere un range di numeri molto più flessibile, il suo contributo è pari a  $2^{2^{es}}$  con  $es$  il numero di bit dell'esponente<sup>1</sup>, permettendo quindi una riduzione del numero di bit assegnati a quest'ultimo campo oppure un incremento di range.

Questi bit di regime non sono altro che una sequenza di cifre binarie identiche, terminate dal complemento di esse.

Ad esempio: 0001 rappresenta -3, dove 3 è il numero degli 0 ed 1 è il terminatore.<sup>2</sup>[2]

## 1.2 Note sui Test

Tutti i test svolti sui Posit sono stati effettuati sfruttando la libreria in C++ BFP[1] implementando, dove necessario, le funzionalità mancanti.

I Posit scelti sono stati a 32/64 bit, con 0 bit di esponente<sup>3</sup>, dove non specificato diversamente.

<sup>1</sup>L'esponente è l'unico campo ad avere una dimensione fissa

<sup>2</sup>Regimi che iniziano per 0 sono negativi, per 1 invece sono positivi. Lo 0 è rappresentato come 10

<sup>3</sup>Ha senso non utilizzare bit di esponente, in quanto si sfrutta il super esponente

## 2 Test

### 2.1 Accuratezza

Questo test vuole dimostrare che a parità di bit utilizzati, 32 in questo caso, i Posit risultano essere più precisi nel rappresentare i risultati di prodotti, rispetto ad un Float in precisione singola.

In questo particolare caso sono stati impiegati Posit [32,3], ossia 32 bit totali, di cui 3 di esponente.

#### 2.1.1 Svolgimento

Il test consiste nel creare 9 array, 6 per gli operandi e 3 per i risultati, di una dimensione compresa nell'ordine dei milioni:

- 3 di Posit [32,3]
- 3 di Float a 32 bit
- 3 di Double a 64 bit come riferimento

Il passo successivo consiste nel popolare gli array degli operandi, operazione che viene svolta partendo dai double sfruttando una funzione che genera dei numeri in modo pseudo-casuale, per poi provvedere a riempire gli altri semplicemente operando le necessarie conversioni.

Successivamente sono state eseguiti i vari prodotti i cui risultati sono stati collocati nel terzo array di ciascun tipo. Infine, per poter confrontare il tutto, i risultati sono stati riportati in double e ne è stata fatta la differenza, in modulo, rispetto al riferimento.

Definisco:

$$\begin{aligned}\Delta_i(\text{Posit}) &= |\text{PositRes}_i - \text{DoubleRes}_i| \\ \Delta_i(\text{Float}) &= |\text{FloatRes}_i - \text{DoubleRes}_i|\end{aligned}$$

Dove FloatRes e PositRes sono il risultato del prodotto, fra float e posit rispettivamente, convertiti in double.

Se  $\Delta_i(\text{Posit}) < \Delta_i(\text{Float})$  i posit sono più precisi dei float per questo indice.

Se  $\Delta_i(\text{Posit}) > \Delta_i(\text{Float})$  i float sono più precisi dei posit per questo indice.

#### 2.1.2 Conclusioni

Dai risultati dei test svolti, i Posit si sono rivelati essere più precisi dei Float nel  $\approx 56\%$  dei casi. Sono risultati pari merito nel  $\approx 16\%$  dei casi.

I risultati risultano essere in linea con le aspettative, in quanto una maggiore precisione, a parità di dimensione, è imputabile ad un numero maggiore di bit disponibili per la mantissa, rispetto ai Float.

### 2.1.3 Codice

```
1 //Funzione per Numeri Random
2 double generateNumber(){           // numeri fra -30 e 30
3     double tmp = (double)rand()/(double)RAND_MAX;
4
5     return (tmp*60) -30;
6 }
```

## 3 Codice?

### 3.1 Test Sigmoidale

```
1 //Prova.cpp
2 printf("Ciao");
```

## Bibliografia

- [1] Boh. "*Beyond Floating Point*". URL: <https://github.com/libcg/bfp>.
- [2] École polytechnique fédérale de Lausanne. *GSN*. URL: <https://github.com/LSIR/gsn>.
- [3] Mario. "ciao". In: *Giornale* (2017).