Architettura degli elaboratori

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2023/2024

Indice

1 Introduzione

L'informatica è nata per la risoluzione i problemi di calcolo, in particolare quelli di calcolo numerico. Per questo motivo i primi computer erano macchine che eseguivano operazioni aritmetiche. Per risolvere questi problemi si usano degli algoritmi che sono una sequenza di istruzioni semplici che portano poi a risolvere problemi di complessità variabile. Anche gli algoritmi hanno una complessità che deve essere adeguata alla risoluzione del problema.

1.1 Hardware

Un algoritmo deve essere trasformato in un processo di calcolo automatico, quindi deve essere implementato tramite hardware. Ci sono due tipi di hardware:

- **Embedded** che è un hardware dedicato ad un singolo compito. Ad esempio il microonde.
- **General purpose** non si sa l'utilizzo finale, quindi ha funzionalità generali ampliate dal software installato. L'hardware general purpose è programmabile attraverso il software. Un esempio è il PC.

In base al tipo di hardware l'algoritmo viene implementato in diversi modi:

- **Algoritmo** $\rightarrow$ **Software**: Tramite un linguaggio di programmazione
- **Algoritmo** $\rightarrow$ **Hardware embedded**: Tramite linguaggi di basso livello come C, Assembly o il sistema operativo.
- **Algoritmo** $\rightarrow$ **Hardware**: Tramite sintesi logica

1.2 Campionamento dei dati

Ogni cosa nel mondo è rappresentabile da funzioni continue nel tempo $f(t)$, ma con risorse finite è impossibile rappresentare infiniti dati, bisogna quindi campionarli.

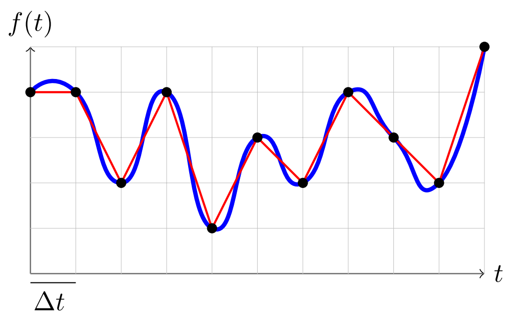

Figura 1: Funzione casuale continua nel tempo

Per campionare la funzione nella figura 1.2 bisogna scegliere un intervallo di tempo Δt e prendere un valore della funzione ogni Δt. In questo caso le linee verticali rappresentano il **campionamento**, mentre quelle orizzontali rappresentano la **discretizzazione o quantizzazione**. La linea rossa è una spezzata approssimata della funzione continua, infatti per il teorema di Shannon:

Teorema 1.1
Deciso il grado di errore da voler compiere, esistono una precisa frequenza di campionamento e un intervallo di discretizzazione che garantiscono quell'errore.

Il sistema di calcolo è ora diventato digitale, cioè elabora i segnali numerici in ingresso per produrre segnali numerici in uscita.

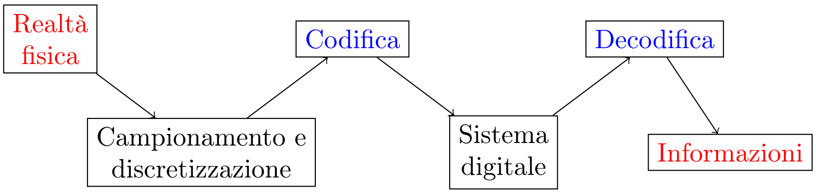

Figura 2: Dalla realtà fisica al sistema digitale

2 Sistemi di codifica

Ogni sistema digitale lavora in base binaria, quindi entrano N bit ed escono M bit. I bit in uscita devono essere codificati per realizzare delle informazioni. Ci sono 2 tipi di informazioni:

- **Informazioni intelleggibili**: sono già chiare agli esseri umani, come un testo scritto.
- **Informazioni non intelleggibili**: hanno bisogno di macchine per essere riprodotte, come le casse per l'audio.

2.1 Codifica di informazioni non numeriche

Ogni informazione deve avere un codice univoco in modo che il sistema digitale non possa sbagliare a decodificarla. Date M informazioni si ricavano $n = log_2(M)$ codici disponibili per rappresentarle.

Esempio 2.1
Con $M = 7$ informazioni:

- $n = log_2(7) \approx 3$ *bit*
- $2^3 = 8$ *codici disponibili*

2.2 Numeri interi assoluti

I numeri interi assoluti rappresentano solo i valori da 0 a $2^n - 1$, dove n è il numero di bit disponibile.

La codifica da base decimale a base binaria prende il nome di **codifica a modulo**

Esempio 2.2
Si deve convertire il numero 57_{10} in base binaria

$$n = log_2(57) = 6 \textit{ bit (minimi)}$$
$$\sum_{i=1}^{n-1} 2^n - 1 = 63 \textit{ (codici massimi)}$$

Si eseguono i seguenti passaggi:

1. *Si sottraggono le potenze di 2 partendo da $n - 1$.*
   - *Se la potenza 2^i è minore o uguale del numero, allora si moltiplica per 1.*
   - *Se la potenza 2^i è maggiore del numero, allora si moltiplica per 0.*
2. *Le sottrazioni continuano fino a quando si giunge a 0.*

$57_{10} - 1 * 2^5 = 25_{10} - 1 * 2^4 = 9_{10} - 1 * 2^3 = 1_{10} - 0 * 2^2 = 1_{10} - 0 * 2^1 = 1_{10} - 1 * 2^0$

2.3 Numeri interi relativi

La codifica più ovvia per i numeri interi relativi è la codifica a **modulo + segno**. Tuttavia rappresenta varie problematiche, per cui si preferisce usare la codifica in **complemento a 2**.

2.3.1 Codifica a modulo + segno

$$\text{Intervallo: } -2^{n-1} \leq N \leq 2^{n-1} - 1$$

Il segno si rappresenta con un bit, 0 per il positivo e 1 per il negativo. Il bit più significativo è il bit del segno, mentre i bit meno significativi rappresentano il modulo.

1 bit: segno ±	7 bit: modulo

Considerando l'esempio 2.2 si hanno le seguenti rappresentazioni:

$$+57_{10} = \mathbf{0}|111001_2$$
$$-57_{10} = \mathbf{1}|111001_2$$

Sorge però un problema quando si vuole rappresentare il valore 0_{10}, che in binario risulterebbe:

$$+0_{10} = \mathbf{0}|000000_2$$
$$-0_{10} = \mathbf{1}|000000_2$$

Inoltre le somme che passano dal positivo al negativo e viceversa risultano errate.

2.3.2 Codifica in complemento a 2

$$\text{Intervallo: } -2^{n-1} \leq N \leq 2^{n-1} - 1$$

La codifica in complemento a 2 rimuove tutti i problemi della codifica in modulo + segno. Questa codifica infatti rende le somme molto più semplici. La somma facile infatti è l'obiettivo di questa codifica e parte dell'idea di trovare la codifica di -1, pertanto si cerca di formulare $-1 + 1 = 0$.

Obiettivo	Risultato
$????_2 +$ $0001_2 =$	$1111_2 +$ $0001_2 =$
$0000_2 =$	0000_2

Se si considera il numero di bit $n = 4$, allora l'intervallo di valori è $-2^3 \leq N \leq 2^3 - 1$:

$0_{10} = 0000_2$	$-1_{10} = 1111_2$
$1_{10} = 0001_2$	$-2_{10} = 1110_2$
$2_{10} = 0010_2$	$-3_{10} = 1101_2$
$3_{10} = 0011_2$	$-4_{10} = 1100_2$
$4_{10} = 0100_2$	$-5_{10} = 1011_2$
$5_{10} = 0101_2$	$-6_{10} = 1010_2$
$6_{10} = 0110_2$	$-7_{10} = 1001_2$
$7_{10} = 0111_2$	$-8_{10} = 1000_2$

I valori nel complemento a 2 ciclano, quindi se si somma 1 a 7 si ottiene -8.

Esempio 2.3
Sottrazione con il complemento a 2: $43 - 17 = 25$

$$n = 7 \text{ bit}$$

1. *Per prima cosa si prende il valore assoluto del numero negativo* 17_{10} *e si converte in binario.*

$$17_{10} = 0010001_2$$

2. *Si inverte il numero trovato.*

$$!(0010001_2) = 1101110_2 = -18_{10}$$

3. *Si somma 1 al numero trovato.*

$$\begin{array}{r} 1101110\ + \\ 0000001\ = \\ \hline 1101111 \end{array}$$
$$1101111_2 = -17_{10}$$

4. *Si somma il numero trovato al numero positivo.*

$$\begin{array}{r} 0010001\ + \\ 1101111\ = \\ \hline \mathit{1}0011010 \end{array}$$

5. *Il risultato ottenuto è:*

$$\mathit{1}0011010$$

Si osserva che c'è un bit in più rispetto a quelli disponibili (quello in grassetto), vuol dire che risulta in overflow[a]*, quindi si scarta il bit più significativo e si ottiene:*

$$0011010_2 = 26_{10}$$

che è il risultato corretto.

[a]Indica il "traboccamento", cioè se viene superato il limite massimo l'overfflow è un errore, non perchè sia sbagliata la somma, ma perchè il risultato non è codificabile con il numero di bit disponibili

Estensione del numero con il complemento a 2

- Se un numero è **positivo** va esteso con gli **0**

$$\begin{array}{r|l} +57_{10} + & 0111001_2\ + \\ +7_{10}\ = & \mathbf{000}1001_2\ = \\ \hline +64_{10} & 1000010_2 \end{array}$$

- Se un numero è **negativo** va esteso con gli **1**

$$\begin{array}{r|l} +57_{10} + & 0111001_2\ + \\ -7_{10}\ = & \mathbf{111}1001_2\ = \\ \hline +50_{10} & 10110010_2 \end{array}$$

3 Numeri razionali

I numeri razionali sono composti da una parte intera e una parte frazionaria. Si possono codificare in 2 modi:

- **Virgola fissa**(fixed point): viene usata maggiormente nei sistemi embedded quando si sa a priori il numero più grande e la precisione che si vuole ottenere
- **Virgola mobile**(floating point): viene usata maggiormente nei sistemi general purpose.

3.1 Codifica in virgola fissa

Esempio 3.1
Si hanno a disposizione 8 bit: 4 per la parte intera e 4 per la parte frazionaria. Vogliamo decodificare il numero 0110.1011_2*:*

$$\underbrace{\overset{2^3}{0}\;\overset{2^2}{1}\;\overset{2^1}{1}\;\overset{2^0}{1}}_{+6}\,.\,\underbrace{\overset{2^{-1}}{1}\;\overset{2^{-2}}{0}\;\overset{2^{-3}}{1}\;\overset{2^{-4}}{1}}_{\frac{1}{2}+\frac{1}{8}+\frac{1}{16}}$$

$$+6+\frac{1}{2}+\frac{1}{8}+\frac{1}{16}=6+\frac{11}{16}=\frac{107}{16}=6.6875$$

Se si vuole codificare un numero da decimale a binario bisogna tenere in considerazione che non è certo che il numero sia razionale anche in base 2, quindi bisogna approssimare per rappresentarlo.

Esempio 3.2
Prendiamo in considerazione $+4+\frac{3}{5}$*, in questo caso bisogna andare "a tentoni" e trovare la rappresentazione binaria che approssima con il minor errore possibile.*

$$4_{10} = 0100_2$$
$$0.1001 = \frac{9}{10}\Delta\frac{3}{80}$$
$$0.0111 = \frac{7}{16}\Delta - \frac{4}{80}$$
$$0.0110 = \frac{3}{8}\Delta\frac{9}{40}$$
$$\underline{0.1010 = \frac{5}{8}\Delta - \frac{1}{40}}$$

Δ *rappresenta l'errore, quindi la rappresentazione più vicina è* 0100.1010_2*.*
Però non è stato rappresentato $\frac{3}{5}$*, ma* $\frac{1}{2}+\frac{1}{16}=\frac{9}{16}$*.*

Questo metodo è pesante perchè bisogna controllare più alternative.

3.1.1 Errore percentuale

Bisogna decidere se calcolarlo rispetto alla parte intera o a quella frazionaria. Nel seguente esempio viene calcolato l'errore percentuale rispetto alla parte frazionaria dell'esempio 3.2.

Esempio 3.3

$$\frac{1}{40} : \frac{3}{5} = \frac{1}{40} * \frac{5}{3} = \frac{1}{24} \approx 0.052\%$$

Il massimo errore che si può fare è l'overflow.

3.2 Codifica in virgola mobile

Gli standard della virgola mobile sono: IEEE 754. Questo standard è stato rivisto molte volte e ora viene usato da tutte le codifiche per i numeri in virgola mobile.
Il numero viene separato in 3 parti:

- **M**: Mantissa
- **B**: Base 2
- **e**: Esponente

La struttura del numero è quindi:

$$N = \pm \cdot B^{\pm e}$$

Questo permette di dividere il numero in modo da poter scegliere quanti bit dedicare alla mantissa e quanti all'esponente. Si riscontrano però i seguenti problemi:

- Bisogna scegliere la base in cui fare la codifica $\rightarrow$ base 2
- Bisogna scegliere la divisione di bit tra *segno*, *mantissa* e *esponente* $\rightarrow$ 1 S, 23 M, 8 e
- La rappresentazione deve essere univoca $\rightarrow 1.\ \ldots_2$
- Bisogna trovare un modo per rappresentare gli errori

Se la mantissa e la base sono in base 2 la moltiplicazione e la divisione sono agevolate tramite l'utilizzo dello *shift*.

- $0110 \cdot 2 = 1100$ è uno shift a sinistra in binario.

$$\begin{array}{c} 0110 \\ \swarrow\swarrow\swarrow \\ 1100 \end{array}$$

- $1010/2 = 0101$ è uno shift a destra in binario.

$$\begin{array}{c} 1010 \\ \searrow\searrow\searrow \\ 0101 \end{array}$$

3.2.1 Divisione di bit tra segno, mantissa ed esponente

Un numero è rappresentabile in 2 modi:

- Singola precisione 32 bit $\rightarrow$ float
- Doppia precisione 64 bit $\rightarrow$ double

Prendiamo in considerazione 32 bit, ora dobbiamo decidere quanti bit dedicare alla mantissa e all'esponente.

$$2^{\pm e}$$

$$|e| = 4bit = 2^{+7}$$
$$5bit = 2^{+15}$$
$$6bit = 2^{+31}$$
$$7bit = 2^{+63}$$
$$8bit = 2^{+127}$$

L'impatto dei bit sull'esponente è doppiamente esponenziale, quindi cresce tantissimo.

- **8 bit** all'esponente, quindi l'esponente può assumere valori da -127 a $+127$.
- **23 bit** alla mantissa, quindi la mantissa può assumere valori da 0 a $2^{23}-1$
- **1 bit** al segno.

1 bit: segno ±	8 bit: esponente	23 bit: mantissa

Per la rappresentazione univoca la mantissa si codifica in virgola fissa. Cioè si parte da una mantissa con un **punto fisso** e dividendo o moltiplicando (shift) si può spostare la virgola per arrivare alla forma **1.00000...** e questa forma è la rappresentazione univoca.

Questa operazioe si chiama **normalizzazione** e visto che la rappresentazione è sempre la stessa l'*1.* non viene rappresentato, quindi viene inserito nella mantissa solo tutto ciò che viene dopo.

$$\begin{array}{c} 11111111\ \pm\infty \\ 11111110\ +127 \\ \dots \\ 00000000\ \pm 0 \\ \dots \\ 00000001\ -126 \\ 00000000\ -127 \end{array}$$

Figura 3: Range dell'esponente

Si è deciso di codificare l'esponente in **Eccesso 127**. Quindi per rappresentare lo zero si usa come esponente il minore numero possibile: $1 \cdot 2^{-127} = 0$. Per codificare i numeri si somma 127 al numero desiderato e visto che i numeri possibili ora vanno da -127 a $+127$ se codifichiamo il risultato in modulo avremo dei numeri da 0 a 256.

Esempio 3.4
Si vuole decodificare il seguente numero:

$$1\ 01110111\ 0110...0$$

$$M = -(1 + \frac{1}{4} + \frac{1}{4}) * 2 = -(\frac{11}{8}) * 2^e$$

$$E = (1 + 2 + 4 + 16 + 32 + 64) - 127 = 119 - 127 = -8$$

$$N = -\frac{11}{8} * 2^{-8}$$

Esempio 3.5
Codifica $+(4 + \frac{1}{2} + \frac{1}{16}) * 2^{+34}$

1. *Sappiamo già che il numero è positivo quindi:*

$$S = 0$$

2. *Calcoliamo la mantissa:*

$$4 + \frac{1}{2} + \frac{1}{16} = \underbrace{100}_{4_{10}} . \underbrace{10010 \ldots 0}_{\frac{1}{2} + \frac{1}{16}}$$

3. *La mantissa va normalizzata moltiplicando per 4:*

$$100.10010 \ldots 0 * 2^{+2} = 1.0010010 \ldots 0$$

$$M = 0010010 \ldots 0$$

4. *Calcoliamo l'esponente:*

- $0\ 00000000\ 0...0 = +0$
- $1\ 00000000\ 0...0 = -0$

Quando l'esponente è tutto 1 e la mantissa tutta 0 allora equivale a $infinito$ + o - in base al primo bit. Se invece la mantissa è diversa da 0 con esponente tutti 1 allora rappresenta un errore NaN.

Somma:

Per un progetto bisogna creare un modello che rappresenti il sistema