Logica e Reti Logiche
(Episodio 10: Numeri e codifiche)

Francesco Pasquale

8 maggio 2023

Con questo episodio iniziamo la seconda parte del corso, in cui vedremo come la logica proposizionale può essere *implementata* con dei circuiti per ottenere funzioni via via più sofisticate.

Tutte le informazioni che devono passare per i circuiti di un computer devono in qualche modo essere codificate in *binario*.

1 Rappresentazione dei numeri naturali

Sono sicuo che molti hanno già visto questa frase scritta da qualche parte:

> *Ci sono soltanto* 10 *tipi di persone al mondo: quelli che conoscono la codifica binaria e quelli che non la conoscono.*

Leggendola la prima reazione potrebbe essere: *non ha senso, dove sono gli altri otto tipi?* Poi un attimo di riflessione e quelli che conoscono il binario capiscono, per gli altri continua a non avere senso.

Un "numero" è un concetto astratto. I modi i cui scriviamo un numero sono moteplici. Per esempio: `sette`, `seven`, `7` e `111` sono sequenze di simboli che esprimono tutte lo stesso concetto astratto, ognuna in una "codifica" diversa (in questo caso, *italiano*, *inglese*, *decimale*, *binario*). Al contrario delle codifiche nelle diverse lingue, le codifiche posizionali, come quella decimale e quella binaria, seguono tutte lo stesso schema: scelta una base $b \in \mathbb{N}$ con $b \geqslant 2$ e un insieme B di b simboli, ognuno dei quali rappresenta un numero compreso fra 0 e $b-1$, una sequenza di simboli $x_{k-1}x_{k-2}\dots x_1x_0$, dove ognuno degli x_i è un simbolo in B, rappresenta il numero

$$x_{k-1}b^k + x_{k-2}b^{k-2} + \cdots x_1 b + x_0 \tag{1}$$

Esercizio 1. Verificare che se prendessimo il numero "tre" come base e i simboli a, b, c per indicare rispettivamente i numeri "zero", "uno" e "due", allora la sequenza di simboli *bacca* rappresenterebbe il numero "centosei".

Scriveremo $(x_{k-1}x_{k-2}\dots x_1x_0)_b$ per indicare che la sequenza di simboli $x_{k-1}x_{k-2}\dots x_1x_0$ rappresenta il numero definito dalla (1).

I componenti elementari di un computer sono progettati per distinguere la differenza di potenziale agli estremi di un conduttore, e il modo più affidabile per farlo è distinguere soltanto fra due valori: *alto* e *basso*. La codifica privilegiata dai computer perciò è quella *binaria*.

Traduzione in binario. Data una sequenza di simboli che esprime un numero in binario, per esempio $(\texttt{10110})_2$, per capire di che numero si tratta ci basta applicare la (1)

$$(\texttt{10110})_2 = 2^4 + 2^2 + 2 = (22)_{10}$$

D'altra parte, anche la rappresentazione binaria di un numero deve venire direttamente dalla definizione. Vediamo come.

Se vogliamo scrivere in binario il numero $(234)_{10}$ dobbiamo trovare la sequenza di bit $x_{k-1} \dots x_0$ tali che

$$234 = x_{k-1}2^{k-1} + x_{k-2}2^{k-2} + \cdots + x_2 2^2 + x_1 2 + x_0 \tag{2}$$

Esercizio 2. Qual è il più piccolo k che ci consente di scrivere $(234)_{10}$ in binario? E per un generico numero naturale $n \in \mathbb{N}$?

Il bit x_0. Nella (2) tutti i bit $x_1, \dots, x_{k-1}$ sono moltiplicati per una qualche potenza di due (quindi, in particolare, la somma $x_{k-1}2^k + x_{k-2}2^{k-2} + \cdots + x_1 2$ è pari) mentre l'ultimo bit x_0 non lo è. Siccome 234 è pari, l'ultimo bit deve essere $x_0 = 0$.

Il bit x_1. Visto che abbiamo stabilito che $x_0 = 0$, se nella (2) dividiamo tutto per 2 abbiamo che

$$\begin{aligned} 117 = \frac{234}{2} &= \frac{x_{k-1} \cdot 2^{k-1} + x_{k-2} \cdot 2^{k-2} + \cdots + x_2 \cdot 2^2 + x_1 \cdot 2}{2} \\ &= x_{k-1} \cdot 2^{k-2} + x_{k-2} \cdot 2^{k-3} + \cdots + x_2 \cdot 2 + x_1 \end{aligned} \tag{3}$$

Tutti i bit $x_2, \dots, x_{k-1}$ sono moltiplicati per una qualche potenza di due, mentre l'ultimo x_1 non lo è. Siccome 117 è dispari, deve essere $x_1 = 1$.

Il bit x_2. Visto che abbiamo stabilito che $x_1 = 1$ se nella (3) sottraiamo 1 a entrambi i termini e dividiamo tutto per 2 abbiamo che

$$\begin{aligned} 58 = \frac{117 - 1}{2} &= \frac{x_{k-1} \cdot 2^{k-2} + x_{k-2} \cdot 2^{k-3} + \cdots + x_2 \cdot 2}{2} \\ &= x_{k-1} \cdot 2^{k-3} + x_{k-2} \cdot 2^{k-4} + \cdots + x_2 \end{aligned} \tag{4}$$

Esercizio 3. Proseguire con le argomentazioni qui sopra fino a scoprire tutti i bit.

Possiamo schematizzare le argomentazioni qui sopra in questo modo:

$$\begin{array}{r|ll} 234 & 0 & \to x_0 \\ 117 & 1 & \to x_1 \\ 58 & 0 & \to x_2 \\ 29 & 1 & \to x_3 \\ 14 & 0 & \to x_4 \\ 7 & 1 & \to x_5 \\ 3 & 1 & \to x_6 \\ 1 & 1 & \to x_7 \end{array} \tag{5}$$

Esercizio 4. Osservare che nello schema precedente, dato un numero n nella collonna di sinistra, il corrispondente numero nella colonna di destra è il resto della divisione di n per 2, mentre il numero successivo nella colonna di sinistra ne è il quoziente. La codifica binaria del numero pertanto si ottiene leggendo i resti della divisione per dur dal basso verso l'altto: $(234)_{10} = (\texttt{11101010})_2$.

Esercizio 5. Sia $b \in \mathbb{N}$ una base qualunque. Descrivere una procedura per scrivere numero $n \in \mathbb{N}$ in base b, e riflettere sul perché la procedura è corretta.

Si osservi che la somma dei numeri in binario si svolge esattamente come in decimale. Per esempio $(3)_{10} = (\texttt{11})$ e $(5)_{10} = (\texttt{101})_2$ e se svolgo la somma nel modo usuale ottengo

$$\begin{array}{rl} 11 & + \\ 101 & = \\ \hline 1000 & \end{array}$$

e $(\texttt{1000})_2 = (8)_{10}$.

Quando lavoriamo con i numeri in binario in informatica spesso abbiamo bisogno di fissare il numero di bit che stiamo dedicando alla codifica dei numeri (tipicamente nei registri dei computer le unità di dati sono sequenze di 32 o 64 bit). Per esempio, se stiamo lavorando a 4 bit, i numeri che possiamo rappresentare sono quelli che vanno da $(0)_{10}$ a $(15)_{10}$. In questo caso, se facciamo la somma fra due numeri il cui risultato è maggiore di 15, diciamo che la somma va in *overflow*. In generale, con k bit in binario possiamo codificare i numeri che vanno da 0 a $2^k - 1$.

2 I parenti stretti del binario

Un numero scritto in binario diventa una sequenza di zeri e uni, che può essere conveniente scrivere anche in altri modi.

Nella codifica *esadecimale* si usano 16 simboli: i simboli $0, \ldots, 9$ per indicare i numeri da zero a nove e le prime lettere dell'alfabeto A, B, C, D, E, F per indicare i numeri da dieci a quindici. Per esempio, la sequenza esadecimale $(A2C)_{16}$ rappresenta il numero duemilaseicentodue. Infatti, dalla (2) abbiamo che

$$\begin{aligned} (A2C)_{16} &= A \cdot 16^2 + 2 \cdot 16 + 12 \\ &= 10 \cdot 256 + 32 + 12 = (2602)_2 \end{aligned}$$

Esercizio 6. Abbiamo detto che $(A)_{16}$ rappresenta il numero *dieci*, quindi in binario si scrive $(1010)_2$. Come si scrive in binario il numero *centosessanta* (cioè *dieci* per *sedici*)? E il numero *duemilacinquecentosessanta* (cioè *dieci* per *sedici al quadrato*)?

Siccome sedici è una potenza di due, per passare da esadecimale a binario, ci basta convertire in binario ogni singolo carattere esadecimale usando quattro bit e poi concatenarli. Per esempio, siccome $(A)_{16} = (1010)_2$, $(2)_{16} = (0010)_2$ e $(C)_{16} = (1100)_2$, il numero $(A2C)_{16}$ in binario si scriverà

$$\texttt{101000101100}$$

Esercizio 7. Scrivere il vostro numero di matricola in binario e in esadecimale.

3 Codifica in complemento a due

Nella sezione precedente abbiamo visto come scrivere in binario i numeri naturali e abbiamo osservato che se abbiamo a disposizione k bit possiamo codificare i numeri che vanno da 0 a $2^k - 1$. Per esempio, con $k = 4$ tutte le sequenze di 4 bit codificano in binario i numeri che vanno da 0 a 15. E se abbiamo bisogno di codificare anche i numeri negativi?

Osservate che se abbiamo a disposizione 4 bit, in tutto possiamo comunque codificare al massimo sedici numeri distinti. La prima idea potrebbe essere: "usiamo un bit per il segno". Quindi, per esempio, con 4 bit scriveremmo il numero $(2)_{10}$ come in binario: `0010` e il numero -2 così `1010`. Questa codifica è perfettamente legittima, ma presenta qualche caratteristica che la rende poco efficace. Per esempio, lo zero ha due codifiche diverse: `0000` e `1000`. Ma soprattutto, la *somma* fra due numeri non si può più fare nel modo usuale. Per esempio,la somma di $(2)_{10}$ e $(-2)_{10}$ deve fare 0, ma se provo a farla in questa codifica ottengo

$$\begin{array}{rl} 0010 & + \\ 1010 & = \\ \hline 1100 & \end{array}$$

che rappresenta il numero $(-4)_{10}$. Una codifica molto conveniente per rappresentare numeri interi positivi e negativi è quella in *complemento a due*.

Definizione 3.1 (Codifica in complemento a due a k bit)**.** Fissato il numero k di bit che abbiamo a disposizione, nella codifica in *complemento a due* la sequenza di bit $x_{k-1}x_{k-2}\dots x_1x_0$ rappresenta il numero

$$-x_{k-1}\cdot 2^{k-1} + x_{k-2}\cdot 2^{k-2} + \cdots + x_1\cdot 2 + x_0$$

Per indicare la codifica in complemento a due useremo $\bar{2}$ come pedice.

Per esempio, il numero $(-2)_{10}$ in complemento a due a quattro bit si scrive `1110`. Infatti

$$(1110)_{\bar{2}} = -2^3 + 2^2 + 2\,.$$

Esercizio 8. Qual è il più piccolo numero rappresentabile in complemento a due a 4 bit? E il più grande? Quali sono i numeri rappresentabili in complemento a due a k bit?

I numeri interi positivi si scrivono in complemento a due esattamente come si scrivono in binario. Come si scrivono i numeri negativi? Per esempio, come faccio a trovare la codifica in complemento a due a quattro bit di $(-5)_{10}$? C'è un metodo molto facile da ricordare: prima scrivo la codifica del numero, poi inverto tutti i bit, infine aggiungo uno. Per esempio, per trovare la codifica di $(-5)_{10}$ in complemento a due a quattro bit, prima scrivo la codifica di $(5)_{10} = (\texttt{0101})_{\bar{2}}$ poi inverto tutti i bit `1010`, infine aggiungo 1 e ottengo `1011`.

Esercizio 9. Ragionando sulla Definizione 3.1 spiegare perché il metodo descritto qui sopra funziona.

Per sommare i numeri in complemento a due si può usare il metodo usuale: per esempio, in complemento a due a quattro bit abbiamo che $(-5)_{10} = (\texttt{1011})_{\bar{2}}$, $(7)_{10} = (\texttt{0011})_{\bar{2}}$ e se faccio la somma ottengo

$$\begin{array}{rl} 1011 & + \\ 0111 & = \\ \hline 0010 & \end{array} \tag{6}$$

e $(\texttt{0010})_{\bar{2}} = (2)_{10}$ che è il risultato corretto.

Notate che nella somma in (6) ho trascurato il bit di riporto che sarebbe stato il "quinto" bit e il risultato è comunque corretto. Infatti in complemento a due a k bit la somma non va in *overflow* quando c'è un riporto non nullo per il quale avrei bisogno di $k+1$ bit, ma quando sommo due numeri positivi la cui somma è maggiore di $2^{k-1}-1$ oppure quando sommo due numeri negativi la cui somma è minore di -2^{k-1}.

I numeri codificati in complemento a due vivono su una circonferenza, dove il successivo del numero più grande rappresentabile è il numero più piccolo rappresentabile. Per esempio, in complemento a due a quattro bit il numero più grande è $(\texttt{0111})_{\bar{2}} = (7)_{10}$ e se sommo 1 a questo numero ottengo $(\texttt{1000})_{\bar{2}} = (-8)_{10}$.

Esercizio 10. Compilate ed eseguite il programma in C qui sotto e vedete che numero scrive a video[1]

```c
#include<stdio.h>
#include<limits.h>

int main(){
    printf("%d\n", INT_MAX);
}
```

Ora considerate quest'altro programma in C qui sotto, che scrive a video tre numeri. Secondo voi che numeri scrive?

```c
#include<stdio.h>
#include<limits.h>

int main(){
    printf("%d\n", INT_MAX);
    int a = INT_MAX - 3;
    printf("%d\n", a);
    int b = a + 5;
    printf("%d\n", b);
}
```

Prima provate a indovinare, poi compilate il programma, eseguitelo e datevi una spiegazione del risultato che ottenete.

[1] `INT_MAX` è una costante che sta nella libreria `limits.h` e indica il più grande numero di tipo `int` rappresentabile (in genere, $2^{31}-1$)

Esercizio 11. Ora provate a indovinare quali sono i due numeri che scrive a video il programma qui sotto.

```c
#include<stdio.h>
#include<limits.h>

int main(){
    int a = INT_MAX;
    a++;
    printf("%d\n", a);
    int b = -1 * a;
    printf("%d\n", b);
}
```

Indovinato?