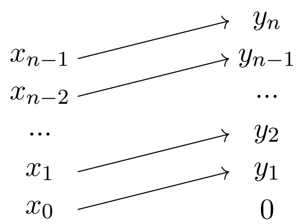
1 Lezione del 22-10-24

1.1 Operazioni a costo nullo

1.1.1 Moltiplicazioni e divisioni per potenze di base

Moltiplicare e dividere per potenze della base β significa semplicemente aggiungere o togliere zeri, ergo si tratta di operazioni a **costo nullo**. Se le operazioni sono a costo nullo, è molto probabile che le reti che le implementano sono **prive di logica**.

- **Moltiplicazione:** effettivamente, la rete che implementa una moltiplicazione per β sposta gli input $x_{n-1}, ..., x_0$ "su", attraverso una mappa:

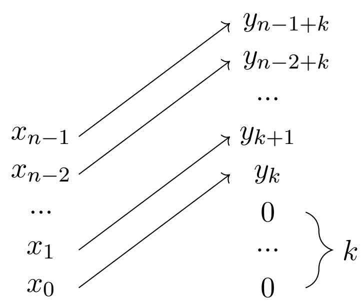
  

  assegnando y_0 ad un generatore di zero.

  Per reti che moltiplicano per multipli $\beta \cdot k$, generalizzeremo la stessa cosa come:

  $$\begin{cases} y_j = x_{j-k}, & k \leq j \leq n-1+k \\ y_j = 0, & 0 \leq j \leq k-1 \end{cases}$$

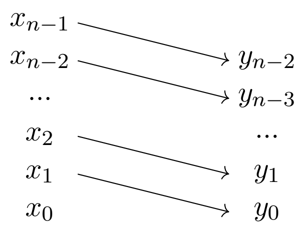
  ottenendo quindi la mappa:

- **Quoziente:** allo stesso modo, si ha che si vogliono spostare gli input "giù", ovvero applicare:

  $$\begin{cases} y_j = x_{j+k}, & k \leq j \leq n-1-k \end{cases}$$

  che per $k = 1$ è:

e per k arbitrari è:

$$
\begin{tikzcd}[row sep=tiny]
& x_{n-1} \arrow[dddr] & \\
& x_{n-2} \arrow[dddr] & \\
& \cdots & \\
& x_{k+1} \arrow[dddr] & y_{n-1-k} \\
& x_k \arrow[dddr] & y_{n-2-k} \\
& x_{k-1} & \cdots \\
k \left\{ \vphantom{\begin{matrix} a \\ a \\ a \end{matrix}} \right. & \cdots & y_1 \\
& x_0 & y_0
\end{tikzcd}
$$

dove i primi k elementi di x vengono trascurati (**troncamento**).

- **Resto:** il resto significa semplicemente "tagliare" tutti gli ingressi prima di x_k, ergo:

$$\begin{cases} y_j = x_j, \quad 0 \leq j \leq k-1 \end{cases}$$

secondo la mappa:

$$
\begin{array}{ccc}
x_{n-1} & & \\
\cdots & & \\
x_{k-1} & \longrightarrow & y_{k-1} \\
x_{k-2} & \longrightarrow & y_{k-2} \\
\cdots & & \cdots \\
x_1 & \longrightarrow & y_1 \\
x_0 & \longrightarrow & y_0
\end{array}
$$

1.1.2 Concatenamento

Concatenare in X due numeri Y e Z a k e $n-k$ cifre significa dire:

$$X = Z \cdot \beta^k + Y$$

Anche questa è un'operazione a complessità nulla, in quanto significa prendere le cifre di Y e Z:

$$\begin{cases} x_j = y_j, \quad 0 \leq j \leq k-1 \\ x_j = z_j, \quad k \leq j \leq n-1 \end{cases}$$

secondo la mappa:

$$
\begin{array}{cccc}
 & z_{n-k-1} & \longrightarrow & y_n \\
z \left\{ \vphantom{\begin{matrix} a \\ a \end{matrix}} \right. & \cdots & & \cdots \\
 & z_0 & \longrightarrow & y_k \\
 & y_{k-1} & & y_{k-1} \\
y \left\{ \vphantom{\begin{matrix} a \\ a \end{matrix}} \right. & \cdots & \longrightarrow & .. \\
 & y_0 & \longrightarrow & y_0
\end{array}
$$

1.1.3 Estensione di campo

L'estensione di campo è l'operazione con cui rappresentiamo un naturale su n cifre su un numero maggiore di cifre. Per i naturali dobbiamo trivialmente aggiungere zero a sinistra della MSD, mentre vedremo che per l'aritmetica intera dovremmo replicare la MSD sulle cifre aggiunte per mantenere il segno corretto.

Abbiamo quindi che per un numero $x = (x_{n-1}, ..., x_0)$ su n cifre vogliamo trovare l'esteso $x' = (x_{n-1+k}, ..., x_0)$ su $n + k$ cifre, cioè il numero tale per cui:

$$\begin{cases} x'_j = x_j, & 0 \leq j \leq n-1 \\ x_j = 0, & n \leq j \leq n-1+k \end{cases}$$

cioè che rispetta la mappa:

$$\begin{array}{ccl} & & \left.\begin{array}{c} 0_{n-1+k} \\ \dots \\ 0_n \end{array}\right\} k \\ x_{n-1} & \longrightarrow & y_{n-1} \\ x_{n-2} & \longrightarrow & y_{n-2} \\ \dots & & \dots \\ x_1 & \longrightarrow & y_1 \\ x_0 & \longrightarrow & y_0 \end{array}$$

1.2 Addizione

La somma, sostanzialmente, consiste nel:

1. Sommare le coppie di cifre di pari posizione, singolarmente, dalla LSD alla MSD e tenendo conto dell'eventuale **riporto entrante**;
2. Se la somma di cifre non è rappresentabile su una singola cifra, usare il **riporto uscente** per la coppia di cifre successive.

Abbiamo che il riporto è sempre $\in \{0, 1\}$, e che per la prima coppia di cifre possiamo assumerlo $= 0$. Ad ogni passaggio, quindi, applichiamo una funzione:

$$(a_i, b_i, c_{in}) \to (s_i, c_{out})$$

Inoltre, si ha che l'algoritmo non dipende dalla base β, ma solamente dalla **notazione posizionale**.

1.2.1 Dimensioni di somme

Avevamo quindi che, dati X, Y in base β su n cifre, cioè $X, Y \in [0, \beta^n - 1]$, con $C_{in} \in [0, 1]$, volevamo calcolare:

$$Z = X + Y + C_{in}$$

ovvero trovare il cosiddetto **full adder**. Possiamo dimostrare che il numero di cifre su cui sta il risultato è:

$$0 \leq X + Y + C_{in} \leq 2\beta^n - 1 \leq \beta^{n+1} - 1$$

dove la cifra $n + 1$ è compresa in $Z_{n+1} \in [0, 1]$, cioè rappresenta il riporto uscente di $X + Y$.

Possiamo quindi affermare con sicurezza che la sommma fra due naturali espressi in base β su n cifre più un'eventuale riporto entrante C_{in} produce un naturale che è sempre rappresentabile su $n+1$ cifre in base β, delle quali la $n+1$-esima cifra è il riporto uscente, e può valere soltanto 0 o 1.

Quello che vogliamo è un circuito sommatore in base β a n cifre che prenda le cifre di due naturali X e Y su n cifre e un riporto entrante C_{in} (un bit), e restituisca un'altro naturale Z, sempre su n cifre e un riporto uscente C_{out} (sempre un bit).

Nel caso uno dei numeri abbia $m > n$ cifre, si estende il numero su n cifre fino a m (aggiungendo $n-m$ zeri in testa), e poi si somma. Se si vuole poi che la somma sia *sempre* rappresentabile, bisogna usare un sommatore ad $n+1$ cifre, ed estenedere gli ingressi su $n+1$ cifre. In questo caso l'ultimo riporto sarà sempre zero.

1.2.2 Ripple carry e full adder

Creare circuiti per $2n+1$ ingressi può essere complicato, quindi si preferisce adottare un approccio **modulare**, dove si scompone ogni somma su una singola coppia di cifre, purchè:

- Le somme vengano eseguite dalla LSD alla MSD;
- Il rapporto si **propaghi** (in inglese *ripple*) da una cifra alla successiva.

Chiamiamo quindi ogni sommatore su due cifre (una di X e una di Y) **full adder**, e il montaggio in cui li disponiamo a **ripple carry** (*propagazione dei resti*).

1.2.3 Full adder in base 2

In base 2, un full adder è un circuito con 3 ingressi (x_i, y_i e c_{in}) e 2 uscite (s_i e c_{out}). Abbiamo che la rete dovrebbe avere tabella di verità:

x_i	y_i	c_{in}	s_i	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Potremmo adesso applicare Karnaugh, ma notiamo che s_i vale 1 quando la somma degli ingressi è dispari, cioè si può dire che è lo XOR in cascata di x_i, y_i e c_{in}. Allo stesso modo, il c_{out} non sarà altro che un circuito SP standard, che prende gli AND di ogni coppia di ingressi e li passa attraverso un OR. Abbiamo che il full adder è una rete a 2 livelli di logica.

1.2.4 Incrementatore

Abbiamo che in assembler potevamo distinguere fra le operazioni `ADD $1, %al` e `INC %al`. Possiamo fare l'assunzione che almeno uno degli ingressi di un full adder sia sempre zero per realizzare un **half adder**: ad esempio, prendiamo $y_i = 0$. In questo caso, c_{out}

potrà essere prodotto con un solo livello di logica, cioè attraverso l'AND fra x_i e c_{in}. Allo stesso modo, potremo ridurre s_i ad un solo XOR fra x_i e c_{in}. Riportiamo una sintesi a tabella di verità e a porte logiche in Verilog:

```
// un half adder in base 2 che calcola @x + @cin, mettendo il
// risultato in @s e il riporto in @cout
module b2_halfadder(x, cin, s, cout);
  input x, cin;
  output s, cout;

  assign {s, cout} = ({x, cin} == 'B00) ? 'B00:
                     ({x, cin} == 'B01) ? 'B10:
                     ({x, cin} == 'B10) ? 'B10:
                   /*({x, cin} == 'B11)?*/'B01;
endmodule

// implementazione a porte logiche
module b2_halfadder(x, cin, s, cout);
  input x, cin;
  output s, cout;

  assign s = x ^ cin;
  assign cout = x & cin;

endmodule
```

1.2.5 Parallelizzazione della somma

Facciamo delle considerazioni sulle prestazioni: se in un full adder ogni input arriva in tempo t, dopo 2 livelli di logica il c_{in} del prossimo full adder arriverà a $t + 2$. Quindi il risultato di quel full adder uscirà a $t+4$ e così via. Si ha che per n full adder concatenati, ergo n cifre, l'$n - 1$-esima cifra viene computata in tempo $t + 2n$. Questo ci dice che la somma è si **scomponibile**, ma non **parallelizzabile**.

In verità, negli anni, sono state sviluppate architetture che implementano il **carry lookahead**, cioè implementano su due livelli di logica, con 5 ingressi, un "precalcolo" del carry a qualche $t + 4$, cioè ogni due full adder (i 5 ingressi sono il primo carry e i 2 + 2 ingressi dei 2 full adder). Questo pressapoco raddoppia la velocità di calcolo delle somme su n cifre, passando quindi da $t + 2n$ a $\approx t + n$.

Riassumendo, quindi, gli incrementatori risultavano più veloci dei full adder per ottimizzazioni di questo tipo e in quanto implementavano tutto su un solo ivello di logica.

1.3 Sottrazione

L'algoritmo di sottrazione consiste nell'applicare un'algoritmo analogo alla somma ma con prestiti al contrario, cioè nel:

1. Sotrarre le coppie di cifre di pari posizione, singolarmente, dalla LSD alla MSD;

2. Se la somma di cifre non è rappresentabile su una singola cifra, generare un **prestito** (*borrow*) per la coppia di cifre successive.

Si ha anche qui che il prestito è sempre $\in [0, 1]$. Inoltre, anche questo algoritmo non dipende dalla base β, ma solo dalla notazione posizionale.

1.3.1 Dimensioni di sottrazioni

Abbiamo quindi due naturali X e Y in base β su n cifre, quindi tali che $X, Y \in [0, \beta^n - 1]$, e un bit B_{in} con $0 \leq b_{in} \leq 1$. Voglio calcolare il naturale:

$$Z = X - Y - b_{in}$$

ammesso che questo naturale *esista!*. Questo perché i naturali non sono chiusi rispetto ala sottrazione, cioè:

$$-\beta^n \leq X - Y - b_{in} \leq \beta^n - 1$$

potrei avere $Z \in \mathbb{Z}$.

1.3.2 Rappresentabilità

Dico quindi che, dal teorema della divisione con resto, posso scrivere Z come quoziente e resto di una divisione per β^n:

$$Z = -b_{out} \cdot \beta^n + D = X - Y - b_{in}$$

definito:

$$-b_{out} = \left\lfloor \frac{X - Y - b_{in}}{\beta^n} \right\rfloor, \quad D = |X - Y - b_{in}|_{\beta^n}$$

dove noto che $b_{out} \in \{0, 1\}$ indipendentemente da β (si comporta come il *carry* della somma).

Posso quindi scrivere Y come il suo complemento, noto che:

$$Y + \overline{Y} = \beta^n - 1, \quad Y = \beta^n - 1 - \overline{Y}$$

da cui sostituendo:

$$(1 - b_{out}) \cdot \beta^n + D = X + \overline{Y} + (1 - b_{in}) \equiv \overline{b_{out}} \cdot \beta^n + D = X + \overline{Y} + \overline{b_{in}}$$

dove si complementano i bit b_{in} e b_{out}. Chiamiamo:

$$\begin{cases} \overline{b_{out}} = c_{out} \\ \overline{b_{in}} = c_{in} \end{cases}$$

Otteniamo che l'equazione finale è sostanzialmente quella di un sommatore:

$$\overline{b_{out}} \cdot \beta^n + D = X + \overline{Y} + \overline{b_{in}} \equiv c_{out} \cdot \beta^n + D = X + \overline{Y} + c_{in}$$

dove la differenza fra X e Y meno un prestito entrante, se naturale, può essere ottenuta se sommo X ad $\overline{Y}$, più un'eventuale riporto entrante ottenuto complementando il prestito entrante. Se a questo punto il riporto uscente di $\overline{b_{out}}$ vale 1, si ha che la differenza è un naturale pari a D, altrimenti non è rappresentabile.

1.3.3 Comparazione di numeri naturali

Dati due **naturali** X e Y, si possono usare i sottrattori per comparare i loro valori, cioè per ottenere $x < Y$. Per fare ciò, si calcola $X - Y$ e si guarda il prestito uscente: se $b_{out} = 1$, allora $X < Y$, altrimenti viceversa.

Per controllare l'uguaglianza, invece, si prende b_{out} e D: se $b_{out} = 1$ (differenza rappresentabile) e $D = 0$, allora $X = Y$, altrimenti viceversa.

1.4 Moltiplicazione

Dati X e C naturali in base β su n cifre, cioè $X, C \in [0, \beta^n - 1]$, e Y naturale in base β su m cifre, cioè $Y \in [0, \beta^m - 1]$, vogliamo calcolare:

$$P = X \cdot Y + C$$

1.4.1 Dimensioni di prodotti

Si ha che, da quanto detto prima:

$$P = X \cdot Y + C \leq (\beta^n - 1) \cdot (\beta^m - 1) + (\beta^n - 1) = \beta^m \cdot (\beta^n - 1) < \beta^{n+m} - 1$$

cioè il risultato sta su $n + m$ cifre.

1.4.2 Algoritmo di moltiplicazione

La moltiplicazione fra naturali si effettua come segue:

1. Si motliplica X per tutte le cifre di Y, iterativamente;
2. Moltiplicando, si generano **risultati parziali**, che vengono disposti a partire dalla cifra per cui stiamo moltiplicando, per quanto ci riguarda si tratta di una moltiplicazione per β^k;
3. I risultati parziali vengono sommati fra di loro con riporto.

Diverse architetture implementano diversi algoritmi di moltiplicazione, ma l'idea fondamentale è quella di creare risultati parziali e sommarli fra di loro. Un modo particolarmente efficiente di fare moltiplicazioni è quello di:

1. Moltiplicare un numero ad n cifre per un numero ad una sola cifra;
2. Sommare gli m addendi, opportunamente traslati, per ottenere il risultato finale.

Possiamo sfruttare il fatto che la somma è **associativa**, e che la cifra i-esima del prodotto, con $0 \leq i \leq n - 1$, è determinata univocamente dai prodotti parziali $j \leq i$, ergo possiamo sommare i risultati parziali mentre si svolgono le moltiplicazioni. Quest'ultima differenza è la più sostanziale dalla classica moltiplicazione *"in colonna"* insegnata a scuola.

Si va quindi a definire una rete detta **moltiplicatore con addizionatore**, che:

1. Moltiplica X per una cifra di Y, sommando un termine C inizialmente nullo, che viene poi impostato alle cifre più significative del risultato parziale trovato. La LSD, invece, viene assegnata direttamente alla posizione corrispondente nel risultato finale.
2. Infine, concatena tutti le cifre ottenute come LSD nel risultato finale.

In questo modo possiamo fare solo moltiplicazioni su $n \times 1$ cifra e somme su due addendi su $n + 1$ cifre.

1.4.3 Moltiplicatore con addizionatore in base 2

Vediamo quindi come realizzare un moltiplicatore con addizionatore $n \times 1$ in base 2, cioè un moltiplicatore con addizionatore ad una cifra.

Vorremmo il risultato, piuttosto triviale in $\beta = 2$:

$$P_i = y_i \cdot X + C = \begin{cases} (0+)\,C, & y_i = 0 \\ X + C, & y_i = 1 \end{cases}$$

Possiamo effettuare la selezione su y_i attraverso quello che è effettivamente un **multiplexer**. Quello che facciamo quindi è collegare un muliplexer fra X e 0 con variabile di controllo y_i a un ingresso di un full adder, e C all'altro ingresso. L'ingresso C_{in} del full adder varrà 0, mentre l'uscita C_{out} verrà concatenata alla somma S.

Abbiamo che in base 2 un multiplexer a due ingressi, con uno di questi negato, è effettivamente una porta AND fra l'ingresso non nullo e la variabile di controllo. Sostituiamo quindi il multiplexer con un AND a n fra X e y_i.

1.4.4 Richiamo all'assembler

Avevamo visto che in assembler la moltiplicazione aveva un solo operando esplicito, mentre l'altro era implicito su AL, AX o EAX. Il risultato veniva poi concatenato in AX, DX_AX o EDX_EAX. Questo rispetta la logica vista finora:partendo da fattori su n e m bit, con $n = m$, si arriva ad un risultato rappresentabile su $n+m = 2n$ bit, cioè $8+8 = 16$ bit (AL $\rightarrow$ AX), $16 + 16 = 32$ bit (AX $\rightarrow$ DX_AX) e $16 + 16 = 32$ bit (EAX $\rightarrow$ EDX_EAX)

1.4.5 Convertitori di base

Vediamo come realizzare un convertitore da 2 cifre, x_1 e x_0 in codifica BCD, alla codifica binaria. Due cifre rappresentano al massimo 99, che in binario sta su 7 bit. Ergo vogliamo un circuito con 8 bit di ingresso (4 bit + 4 bit degli ingressi BCD) e 7 bit di uscita. Abbiamo che, banalmente, la conversione si effettua come:

$$y = 10 \cdot x_1 + x_0$$

Questo si può realizzare con un moltiplicatore con addizionatore con $X = x_1$, $Y = 10$ e $C = x_0$. Abbiamo che il risultato è su 8 bit, di cui sappiamo però possiamo ridurre il campo a 7.

Un circuito più efficiente può essere realizzato usando solo somme e shift, infatti abbiamo che:

$$y = 10 \cdot x_1 + x_0 = 8 \cdot x_1 + 2 \cdot x_1 + x_0$$

che appare migliore dal punto di vista della realizzazione in aritmetica binaria (8 e 2 sono 2^3 e 2^1). Abbiamo quindi che possiamo usare i moltiplicatori per b^k, e ottenere un circuito con lo stesso comportamento.

Per la precisione, prendiamo $8 \cdot x_1$ e troviamo che si estende fino a 7 bit. Prendiamo poi $2 \cdot x_1 + x_0$ e vediamo che la somma si rappresenta su 5 bit. Sommando i 7 bit di $8 \cdot x_1$ ai 5 di $2 \cdot x_1 + x_0$ abbiamo un risultato sempre su 7 bit.