



Aritmetica dei calcolatori

Rappresentazione dei numeri naturali e relativi

Addizione e sommatore: a propagazione di riporto, veloce, con segno

Moltiplicazione e moltiplicatori: senza segno, con segno e algoritmo di Booth

Rappresentazione in virgola mobile e operazioni



La rappresentazione dei numeri

- Rappresentazione dei numeri: **binaria**
- Un numero binario è costituito da un *vettore di bit*

$$B = b_{n-1} \dots b_1 b_0 \quad b_i = \{0, 1\}$$

- Il valore di B e' dato da:

$$V(B) = b_{n-1} \times 2^{n-1} + \dots + b_1 \times 2^1 + b_0 \times 2^0$$

- Un vettore di n bit consente di rappresentare i **numeri naturali** nell'intervallo da 0 a $2^n - 1$.
- Per rappresentare i numeri positivi e negativi si usano diverse codifiche



La rappresentazione dei numeri

- Codifiche per **numeri relativi**

- Modulo e segno
- Complemento a 1
- Complemento a 2

| B $b_2 b_1 b_0$ | V(B) | | |
|--------------------|----------------|-----------------|-----------------|
| | Modulo e segno | Complemento a 1 | Complemento a 2 |
| 000 | +0 | +0 | +0 |
| 001 | +1 | +1 | +1 |
| 010 | +2 | +2 | +2 |
| 011 | +3 | +3 | +3 |
| 100 | -0 | -3 | -4 |
| 101 | -1 | -2 | -3 |
| 110 | -2 | -1 | -2 |
| 111 | -3 | -0 | -1 |



La rappresentazione dei numeri

- **Modulo e segno:**

- rappresentazione con n bit: il bit di segno è 1 per i numeri negativi e 0 per i positivi
- campo rappresentabile $-2^{n-1} \leq N \leq +2^{n-1} - 1$ (due rappresentazioni per lo 0)
- è molto simile alla rappresentazione dei numeri decimali

- **Complemento a 1**

- rappresentazione con n bit: i numeri negativi sono ottenuti invertendo bit a bit il corrispondente numero positivo
- campo rappresentabile $-2^{n-1} \leq N \leq +2^{n-1} - 1$ (due rappresentazioni per lo 0)
- è semplice

- **Complemento a 2**

- rappresentazione con n bit: i numeri negativi sono ottenuti invertendo bit a bit il numero positivo corrispondente, quindi sommando il valore 1
- campo rappresentabile $-2^{n-1} \leq N \leq +2^{n-1} - 1$ (una rappresentazioni per lo 0)
- consente di realizzare circuiti di addizione e sottrazione più semplici
- è quella utilizzata nei dispositivi digitali per rappresentare numeri relativi



ADDIZIONE e ARCHITETTURE DI SOMMATORI

- Addizione e sommatore a propagazione di riporto
- Addizione e sommatore ad anticipazione di riporto
- Addizione di più valori e sommatore Carry Save
- Addizione/sottrazione con segno



Addizione senza segno

- La somma di numeri positivi si esegue sommando coppie di bit parallele, partendo da destra.
- Si ha riporto, diverso da 0, quando si deve eseguire la somma 1+1 (*half adder*).

- Regole per la somma:

| | | | | | |
|-----|-----|-----|-----|-----|-------------------|
| 0 | 0 | 0 | 1 | 1 | Riporto in uscita |
| 0 | 0 | 1 | 1 | 1 | |
| + 0 | + 1 | + 0 | + 1 | + 1 | |
| 0 | 1 | 1 | 0 | 0 | |

- Utilizzando queste regole in modo diretto è possibile
 - Realizzare **sommatori modulari**
 - Composti da blocchi elementari identici
 - Circuiti aritmetici di questo tipo sono detti **bit-slice**



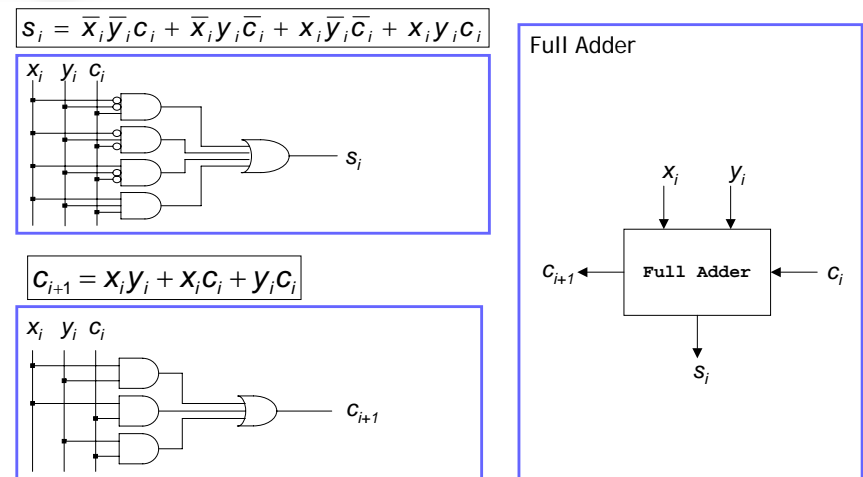
Addizione senza segno *a propagazione di riporto*

- Un sommatore *bit-slice ripple carry* è strutturato in modo che il **modulo in posizione i -esima**:
 - Riceve in ingresso i bit x_i e y_i degli operandi
 - Riceve in ingresso il riporto c_i del modulo precedente
 - Produce la somma $s_i = x_i y_i' c_i + x_i' y_i c_i' + x_i y_i' c_i' + x_i' y_i c_i$
 $\cdot = (x_i \text{ xor } y_i)' c_i + (x_i \text{ xor } y_i) c_i' = x_i \text{ xor } y_i \text{ xor } c_i$
 - Produce il riporto $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$
 $\cdot = x_i y_i + (x_i \text{ xor } y_i) c_i$
- Il modulo in posizione 0 ha il bit di riporto $c_0=0$
- Il riporto c_0 può essere sfruttato per sommare il valore 1
 - Necessario per il calcolo del complemento a 2
- La somma di numero ad n bit richiede un tempo pari ad n volte circa quello richiesto da un modulo di somma

| c_i | x_i | y_i | s_i | c_{i+1} |
|-------|-------|-------|-------|-----------|
| 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 |



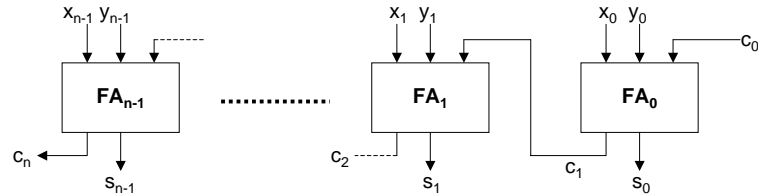
Sommatore per il generico stadio: *Full Adder*





Addizione senza segno - *Ripple-Carry a n bit* Struttura e prestazioni

- Il calcolo esatto del ritardo si effettua basandosi sulla seguente architettura
- Siano T_s e T_r i ritardi per il calcolo della somma e del riporto *i-mi* rispettivamente



- Prestazioni:** il ritardo totale per ottenere tutti i bit della somma è dato dall'espressione:

$$T_{tot} = (n-1)T_r + T_s$$

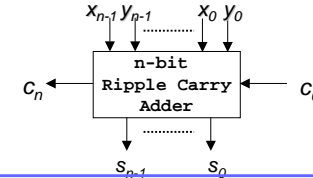
- Il **percorso critico** è quindi quello del **riporto**

- 9 -

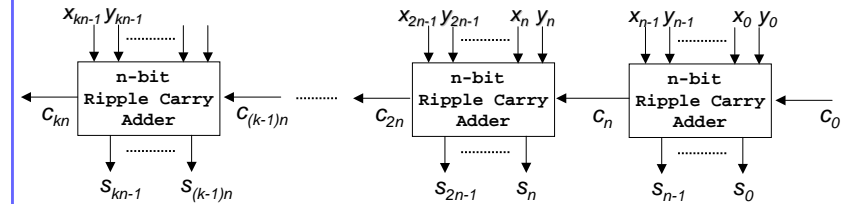


Ripple Carry: architettura a blocchi

Ripple-Carry Adder



Ripple-Carry **Block** Architecture



- 10 -



Addizione veloce (ad anticipazione di riporto) Funzioni di generazione e di propagazione del riporto

Motivazioni: ottenere un **sommatore** con **prestazioni migliori**

Si basa sulle seguenti considerazioni

- Le espressioni di somma e riporto per lo stadio i sono:

$$S_i = X_i Y_i C_i + X_i Y_i C_i' + X_i Y_i' C_i' + X_i Y_i' C_i$$

$$C_{i+1} = X_i Y_i + X_i C_i + Y_i C_i$$
- L'espressione del riporto in uscita può essere riscritta come:

$$C_{i+1} = G_i + P_i C_i \quad \text{con} \quad G_i = X_i Y_i \quad \text{e} \quad P_i = X_i + Y_i \quad (\text{o anche } P_i = X_i \oplus Y_i)$$
- Le funzioni G_i e P_i
 - Sono dette funzioni di **generazione** e **propagazione**
 - G_i : se $x_i=y_i=1$, allora il riporto in uscita **deve** essere generato
 - P_i : se x_i o $y_i=1$ e $c_i=1$, allora il riporto in ingresso **deve** essere propagato in uscita
 - Possono essere calcolate in parallelo, per tutti gli stadi, rispetto alle rispettive somme.

- 11 -



Addizione veloce - *calcolo dei riporti in parallelo*

- L'espressione per il riporto $c_{i+1} = G_i + P_i c_i$ può essere calcolata in modo **iterativo**.
- Sostituendo $c_i = G_{i-1} + P_{i-1} c_{i-1}$ nell'espressione di c_{i+1} si ha:

$$c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}c_{i-1}) = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

- Continuando con l'**espansione** fino a c_0 si ottiene:

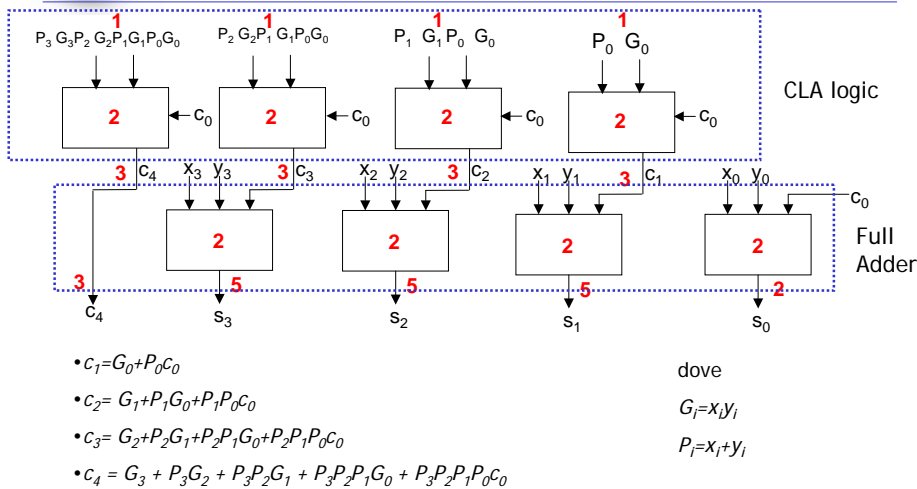
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_1 P_0 c_0$$

- I riporti in uscita di ogni singolo stadio possono essere calcolati tutti in parallelo e con ritardo identico (realizzazione SOP) tramite:
 - le i funzioni di generazione G_i e le i funzioni di propagazione P_i
 - il riporto in ingresso allo stadio 0, c_0
- I sommatore che sfruttano il meccanismo della generazione dei riporti in anticipo sono detti **Carry-Look-Ahead Adders** o **CLA Adders**

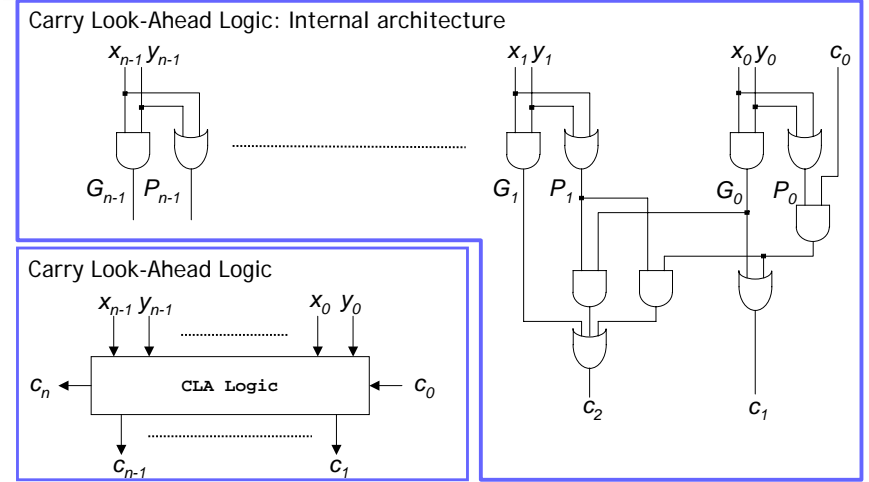
- 12 -



Addizione veloce - *Struttura e prestazioni di un CLA a 4 bit*

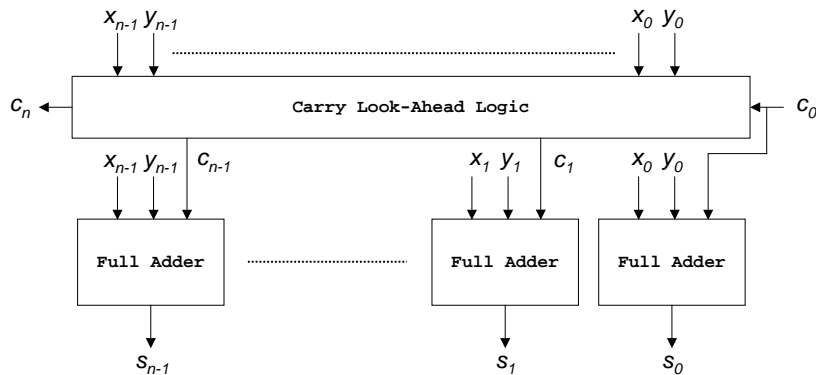


Addizione veloce - *Struttura generale CLA logic*



Sommatori Carry Look-Ahead

Carry Look-Ahead Logic



Addizione veloce: *calcolo delle prestazioni*

- Il ritardo totale per ottenere tutte le somme ed il riporto più a sinistra C_{i+1} è dato dalla somma di:
 - Un ritardo di porta per il calcolo delle funzioni di generazione e di propagazione ($G_i = x_i y_i$ e $P_i = x_i + y_i$)
 - Due ritardi di porta logica per calcolare il riporto i -esimo (SOP)
 - Due ritardi di porta logica per calcolare la somma i -esima (SOP)
- Totale:
 - 5 ritardi di porta logica
- Il ritardo è costante e indipendente dalla lunghezza degli operandi
- Problema:
 - La realizzazione circuitale dei moduli che calcolano i riporti per operandi lunghi (ad esempio 32 bit) fa uso di porte con un *fan-in* molto elevato: non praticabile!!
 - Soluzione: **addizionatore veloce a blocchi**



Addizione veloce a blocchi

- Il sommatore completo a n bit è ottenuto utilizzando un insieme di **blocchi** costituiti da **CLA a m bit** e della **logica CLA**
- Esempio: **blocco** è costituito da un sommatore **CLA a 4 bit** (ragionevole)
- Struttura del blocco di un CLA a 4 bit**

- Il riporto finale di questo sommatore ha la seguente espressione:

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

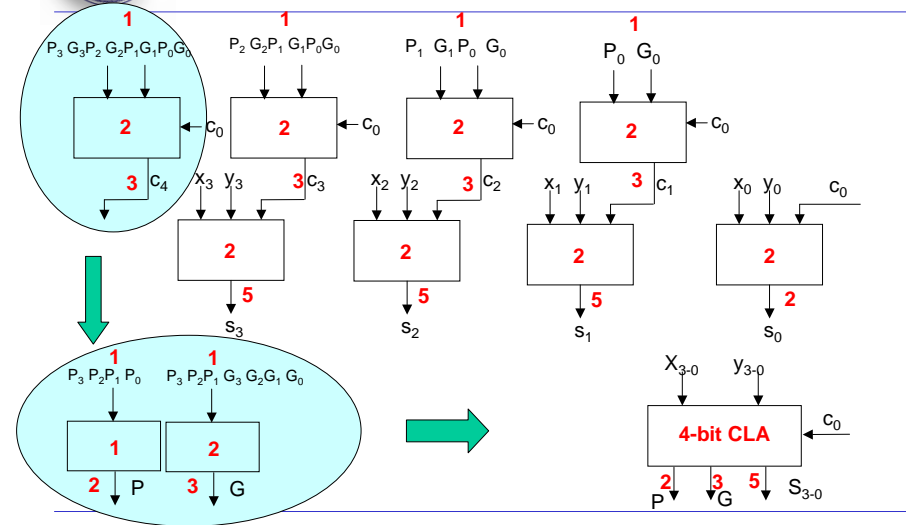
- che può essere riscritta come

$$C_{uscita} = G + PC_0$$

- con il tempo di ritardo per il calcolo di P e G:
 - P = attraversamento di 2 porte logiche (1 per calcolare P_3, P_2, P_1 e P_0 , 1 per calcolare il prodotto)
 - G = attraversamento di 3 porte logiche (calcolo di P_i e G_i , calcolo dei prodotti, calcolo della somma)

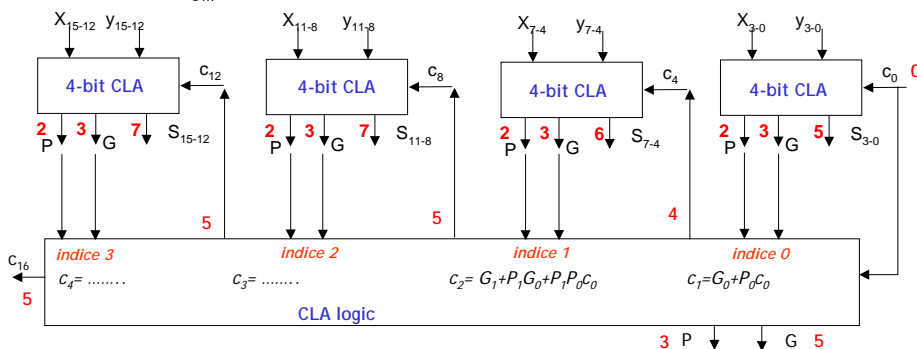


Addizione veloce - blocco CLA a 4 bit

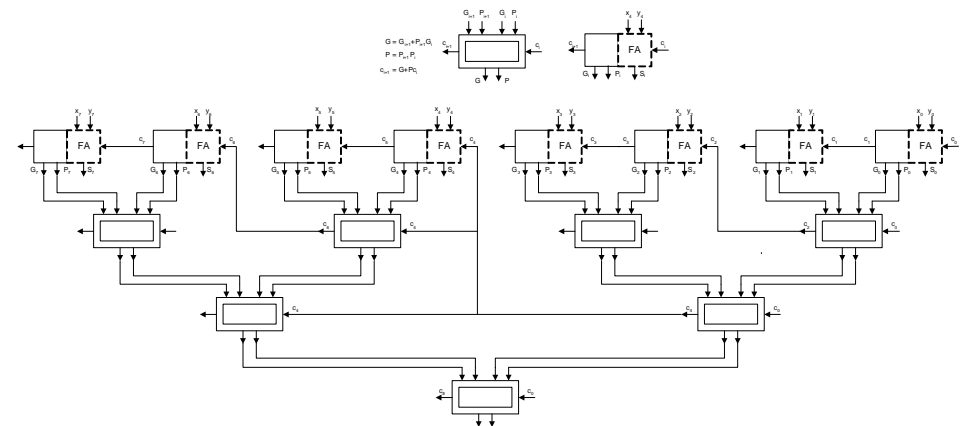


Esempio - sommatore a 16 bit con CLA a 4 bit

- Prestazioni: in questo caso circa $n/2$
- Che cosa succede se devo sommare due numeri da 32 o da 64 bit?
 - Le **prestazioni di un CLA adder a n bit** costituito da blocchi da m bit sono espresse come $\log_m n$, a meno del fattore costante dato dal ritardo di un CLA a m bit.



Esempio - sommatore a 8 bit con CLA a 2 bit





Somma di più valori: sommatore carry save

Calcolo della somma di 3 (o più) valori: $W = X + Y + Z$

Soluzione 1:

- Calcolare una somma intermedia: $T = X + Y$
- E quindi calcolare il risultato finale: $W = T + Z$
- Le somme possono essere realizzate mediante
 - Due **sommatori ripple-carry** (o due sommatore **carry look-ahead**) connessi **in cascata**
 - Ricorda: la somma di N addendi da n bit richiede $n \lceil \log_2 N \rceil$ bit per il risultato

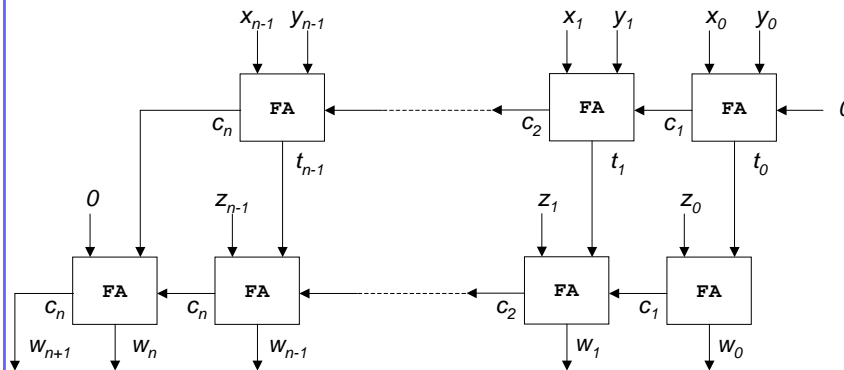
Soluzione 2:

- Modifica dell'algoritmo di somma e uso di **sommatori carry save** per migliorare le prestazioni



Somma di tre addendi - architettura con sommatore ripple-carry

Soluzione con sommatore ripple-carry $W = (X + Y) + Z = T + Z$

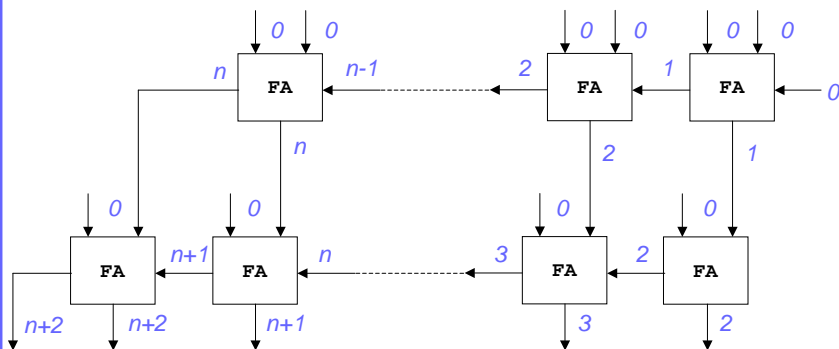


Somma di tre addendi - Prestazioni con sommatore ripple-carry

Prestazioni con sommatore ripple-carry (in blu il ritardo di ogni segnale)

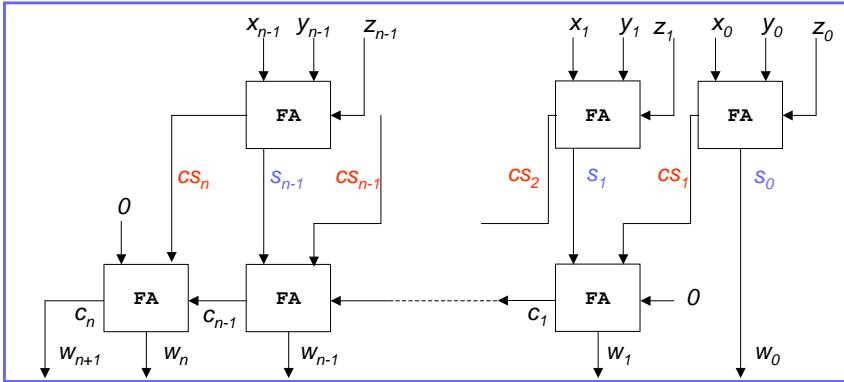
Ritardo $R = (n + 2)\Delta T$ con ΔT ritardo di un Full-Adder

Somma di N addendi da n bit: $N-1$ stadi di somma, risultato su $n \lceil \log_2 N \rceil$ bit, ritardo = $(n \lceil \log_2 N \rceil) \Delta T$



Somma di tre addendi con Sommatore Carry Save

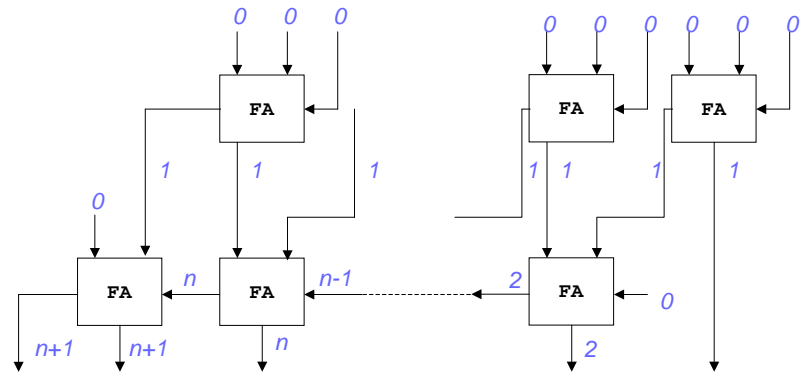
- Il primo stadio calcola le somme S (parziali e senza propagazione di riporto) e i riporti CS (Carry Save Adder)
- Il secondo stadio somma (con propagazione di riporto) i valori provenienti dal primo stadio





Somma di tre addendi - Prestazioni Carry Save

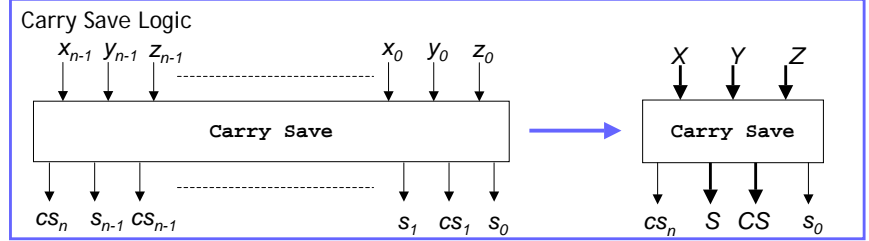
Prestazioni con sommatore ripple-carry per l'ultimo stadio (in blu il ritardo di ogni segnale)
 Ritardo $R = (n + 1)\Delta T$ con ΔT ritardo di un Full-Adder



Sommatore Carry Save come blocco

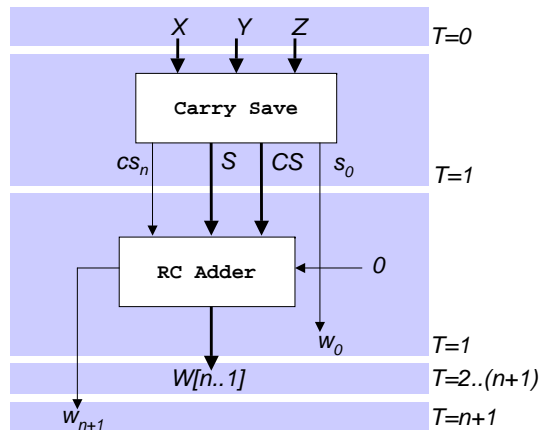
□ Sommatore Carry Save composto da due unità

- Blocco **Carry Save**:
 - Produce i due vettori S e CS
 - Ritardo: $R_{CS} = 1$
- Sommatore **Ripple-Carry**:
 - Produce il risultato finale
 - Ritardo: $R_{RC} = n + 1$

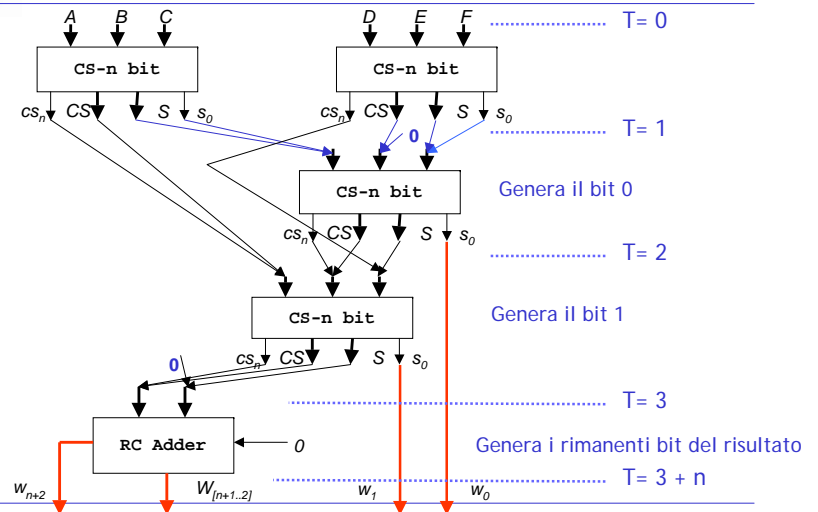


Sommatore Carry Save come blocco

□ Istanti di generazione dei bit di uscita



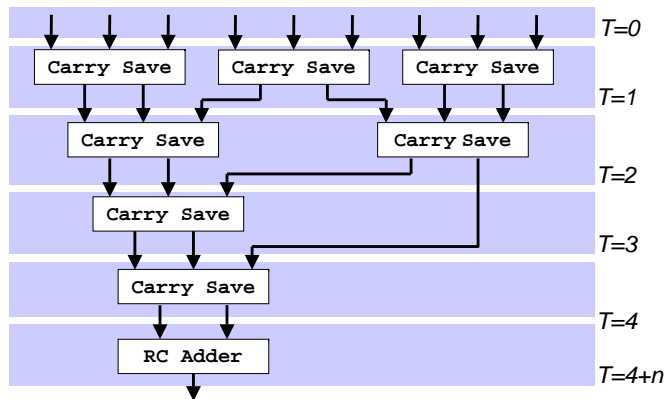
Esempio sommatore a 6 addendi con blocchi Carry Save da 3 addendi





Esempio sommatore a 9 addendi con blocchi Carry Save da 3 addendi

- Vantaggi più evidenti al crescere del numero degli operandi



- 29 -



Addizione e sottrazione per valori rappresentati in complemento a 2

- Regole per la **somma** e **sottrazione** di due numeri **in complemento a 2** su n bit
 - Per calcolare $x+y$
 - Fornire in ingresso ad un sommatore binario naturale le codifiche binarie
 - Ignorare il bit di riporto in uscita
 - Il risultato è in complemento a due
 - Per calcolare $x-y$
 - Ricavare la rappresentazione dell'opposto di y (complemento a due)
 - Sommare i valori così ottenuti come nella regola precedente
 - Il risultato è in complemento a due
- I risultati sono corretti se e solo se, disponendo di un sommatore binario senza segno ad n bit, il risultato sta nell'intervallo:

$$-2^{n-1} \leq x \pm y \leq 2^{n-1}-1$$
- In caso contrario si verifica overflow (o underflow) aritmetico

- 30 -



Addizione e sottrazione per valori rappresentati in complemento a 2

- Condizioni di **overflow** e di **underflow** per somme e sottrazioni in complemento a 2 su n bit

| A+B | | | | |
|-----|-----|-------------|-------|--|
| A | B | Segno somma | Ov/Un | |
| > 0 | > 0 | 0 | Si-Ov | |
| > 0 | < 0 | | no | |
| < 0 | > 0 | | no | |
| < 0 | < 0 | 1 | Si-Un | |

| A-B=A+(-B) | | | | |
|------------|-----|-----------------|-------------|-------|
| A | B | -B = B_{CPL2} | Segno somma | Ov/Un |
| > 0 | > 0 | < 0 | | no |
| > 0 | < 0 | > 0 | 0 | Si-Ov |
| < 0 | > 0 | < 0 | 1 | Si-Un |
| < 0 | < 0 | > 0 | | no |

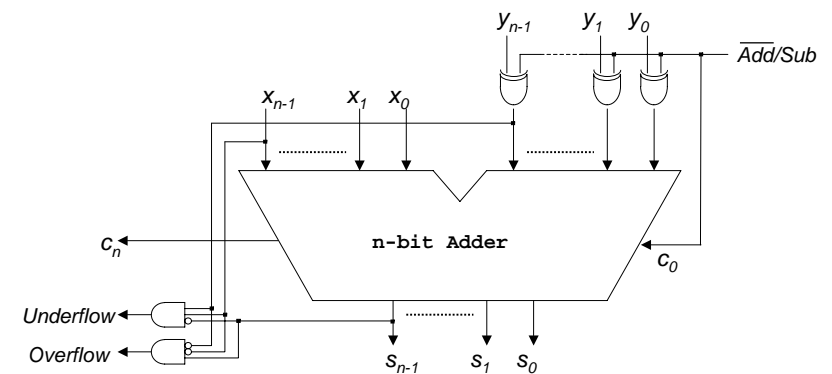
- **overflow** per somma = 0 0 1 (segno addendi e segno somma)
- **underflow** per somma = 1 1 0
- **overflow** per sottrazione = 0 1 1
- **underflow** per sottrazione = 1 0 0

- 31 -



Sommatori Add/Subtract: operazioni in complemento a 2

Add/Subtract Architecture



- 32 -



MOLTIPLICAZIONE e ARCHITETTURE DI MOLTIPLICATORI

- Moltiplicazione senza segno e moltiplicatori per righe e diagonali
- Moltiplicazione con segno con 2 sottomatrici di prodotti parziali
- Moltiplicazione con l'algoritmo di Booth
- Moltiplicazione per colonne e moltiplicatore di Wallace
- Moltiplicatori sequenziali



Moltiplicazione interi senza segno

- La moltiplicazione di numeri **senza segno** si esegue con lo stesso metodo usato per la moltiplicazione decimale
- Il **prodotto** di due numeri binari di n e k bit è un numero binario di $n+k$ bit
- Ogni prodotto parziale deve essere esteso a $n+k$ bit tramite 0
- Ad esempio:

$$\begin{array}{r}
 1101 \times \longrightarrow \text{Moltiplicando } M = 13 \\
 1011 = \longrightarrow \text{Moltiplicatore } Q = 11 \\
 \hline
 00001101 \\
 0001101 \\
 000000 \\
 01101 \\
 \hline
 10001111 \longrightarrow \text{Prodotto } P = 143
 \end{array}$$

} Matrice dei prodotti parziali

- 34 -



Moltiplicatori combinatori

- Prodotto di due numeri positivi di 3 bit (n bit - $2n$ bit prodotto)

| Moltiplicazione bit a bit | | | | | |
|---------------------------|-----------|-----------|-----------|----------|-------|
| | x_2 | x_1 | x_0 | \times | |
| | y_2 | y_1 | y_0 | = | |
| | y_0x_2 | y_0x_1 | y_0x_0 | | |
| y_1x_2 | y_1x_1 | y_1x_0 | | | |
| y_2x_2 | y_2x_1 | y_2x_0 | | | |
| | PP_{02} | PP_{01} | PP_{00} | | |
| | PP_{12} | PP_{11} | PP_{10} | | |
| PP_{22} | PP_{21} | PP_{20} | | | |
| P_5 | P_4 | P_3 | P_2 | P_1 | P_0 |

Matrice di **prodotti parziali** costituita da n righe

- 35 -



Architetture di moltiplicatori

- Le architetture sono definite a seconda del "meccanismo" di somma e propagazione dei riporti delle righe della matrice di prodotti parziali
 - **Somma per righe**: i riporti si propagano per righe
 - **Somma per diagonal**: i riporti si propagano per diagonal
 - **Somma per colonne**: i riporti si propagano per colonne
- Le architetture risultanti sono caratterizzate comunque da strutture regolari

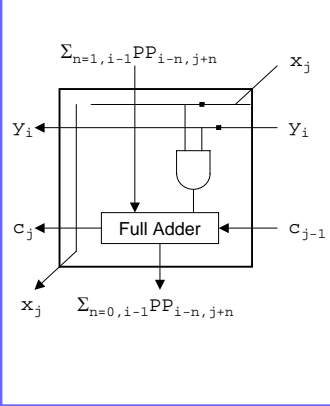
- 36 -



Moltiplicatori combinatori: *somma per righe*

□ Somma per righe

Multiplier Cell



Ogni cella del moltiplicatore calcola

- il **prodotto parziale** corrispondente e
- una **somma parziale**

Il **riporto** delle somme parziali si propaga lungo la **riga**

Le somme si propagano in verticale

Per il calcolo del prodotto parziale, X si propaga in diagonale e Y in verticale

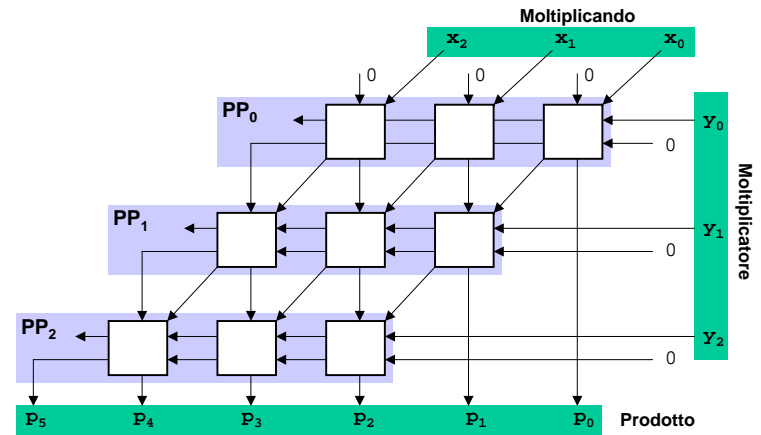
Sono necessari n sommatore a n bit (con eventuale calcolo del prodotto parziale). Il primo non genera riporti

La struttura è regolare

Prestazioni: dipendono dai sommatore, con sommatore non veloci ordine di $2n$



Moltiplicatori combinatori: *somma per righe*



Moltiplicatori combinatori: *somma per diagonali*

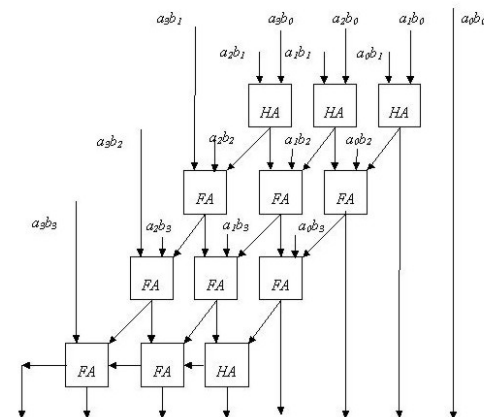
□ Somma per diagonali

- Ogni cella del moltiplicatore (tranne quelle dell'ultima riga) calcola il prodotto parziale corrispondente e una somma parziale
- Il riporto delle somme parziali si propaga lungo le diagonali
- Le somme si propagano in verticale
- Per il calcolo del prodotto parziale, X si propaga in diagonale e Y in verticale
- Sono necessari n sommatore a n bit (di cui il primo non genera riporti)
- La struttura è regolare
- Prestazioni: dipendono dai sommatore, con sommatore non veloci ordine di $2n$



Moltiplicatori combinatori: *somma per diagonali*

Circuito per la somma per diagonali.





Moltiplicazione con segno: *sottomatrici*

Altro modo di definire il complemento a 2: il valore della cifra associata al bit più significativo ha segno negativo.

$$+5 = 0\ 1\ 0\ 1$$

$$-5 = 1\ 0\ 1\ 1$$

$$-5 = -2^3 + 2^1 + 2^0 = -8 + 2 + 1$$

Considerando ad esempio 4 bit, i **valori** di un moltiplicando **M negativo** e di un moltiplicatore **Q negativo** possono essere rappresentati come

$$\cdot \text{Moltiplicando } M = (-m_3)2^3 + m_22^2 + m_12^1 + m_0$$

$$\cdot \text{Moltiplicatore } Q = (-q_3)2^3 + q_22^2 + q_12^1 + q_0$$

· Dove m_3, \dots, m_0 e q_3, \dots, q_0 sono i bit del moltiplicando e del moltiplicatore

Nota: questa rappresentazione viene usata, ed è corretta, sia per valori positivi che negativi.



Moltiplicazione con segno (cont.)

- Costruzione dei prodotti parziali (matrice diagonale)
 - **Cambia** per tener conto del segno dei due fattori (caso M e Q negativi)

$$\begin{array}{cccc}
 & -q_0m_3 & q_0m_2 & q_0m_1 & q_0m_0 \\
 -q_1m_3 & q_1m_2 & q_1m_1 & q_1m_0 & \\
 -q_2m_3 & q_2m_2 & q_2m_1 & q_2m_0 & \\
 +q_3m_3 & -q_3m_2 & -q_3m_1 & -q_3m_0 &
 \end{array}$$



Moltiplicazione con segno (cont.)

- Scomposizione della matrice iniziale in due **sottomatrici**, una con i soli termini negativi, l'altra con solo quelli positivi. Il risultato è dato dalla differenza dei due risultati parziali

Estensione a n+k bit tramite 0

| | | | | | | |
|---------------------------------|----------|----------|----------|----------|----------|---|
| | | 0 | q_0m_2 | q_0m_1 | q_0m_0 | |
| | 0 | q_1m_2 | q_1m_1 | q_1m_0 | | |
| | 0 | q_2m_2 | q_2m_1 | q_2m_0 | | |
| q_3m_3 | 0 | 0 | 0 | | | |
| Somma parziale termini positivi | | | | | | - |
| | | q_0m_3 | 0 | 0 | 0 | |
| | | q_1m_3 | 0 | 0 | | |
| | q_2m_3 | 0 | 0 | | | |
| 0 | q_3m_2 | q_3m_1 | q_3m_0 | | | |
| Somma parziale termini negativi | | | | | | |



Esempio 1

- Si calcoli il prodotto 13×-9

$$\begin{array}{r}
 0\ 1\ 1\ 0\ 1 \times 13 \\
 -1\ 0\ 1\ 1\ 1 = -9 \\
 \hline
 0\ 1\ 1\ 0\ 1 \\
 0\ 1\ 1\ 0\ 1 \\
 0\ 1\ 1\ 0\ 1 \\
 0\ 0\ 0\ 0\ 0 \\
 0\ 0\ 0\ 0\ 0 \\
 \hline
 0\ -1\ -1\ 0\ -1
 \end{array}
 \qquad
 \begin{array}{r}
 0\ 1\ 1\ 0\ 1 \\
 0\ 1\ 1\ 0\ 1 \\
 0\ 1\ 1\ 0\ 1 \\
 0\ 0\ 0\ 0\ 0 \\
 0\ 0\ 0\ 0\ 0 \\
 \hline
 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1 \\
 \qquad\qquad\qquad 91
 \end{array}
 \qquad
 \begin{array}{r}
 0\ 0\ 0\ 0\ 0 \\
 0\ 0\ 0\ 0\ 0 \\
 0\ 0\ 0\ 0\ 0 \\
 0\ 1\ 1\ 0\ 1 \\
 \hline
 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0 \\
 \qquad\qquad\qquad 208
 \end{array}$$

$$\begin{array}{r}
 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1 \\
 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1 \\
 \hline
 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\
 \qquad\qquad\qquad -117
 \end{array}$$

-208 in complemento a 2



Esempio 2

- Si calcoli il prodotto -13×-9

$$\begin{array}{r}
 -1\ 0\ 0\ 1\ 1 \times -13 \\
 -1\ 0\ 1\ 1\ 1 = -9 \\
 \hline
 -1\ 0\ 0\ 1\ 1 \\
 -1\ 0\ 0\ 1\ 1 \\
 -1\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 0\ 0 \\
 1\ 0\ 0\ -1\ -1 \\
 \hline
 0\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 1\ 1 \\
 0\ 0\ 0\ 0\ 0 \\
 1\ 0\ 0\ 0\ 0 \\
 \hline
 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1 \\
 277 \\
 160 \\
 \hline
 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1 \\
 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1 \\
 1 \\
 \hline
 0\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1 \quad 117
 \end{array}$$



Moltiplicazione con segno - Algoritmo di Booth

- Se il moltiplicatore contiene sequenze di 1, l'algoritmo di Booth è più **efficiente** del metodo visto in precedenza (cioè devono essere generati molti meno prodotti parziali)
- Si consideri ad esempio la moltiplicazione per $Q=30$:

$$M \times 30 = M \times (32 - 2) = M \times 32 - M \times 2$$
- In rappresentazione binaria:

$$\begin{aligned}
 M \times 0011110 &= M \times 0100000 - M \times 0000010 \\
 &= M \times 0100000 + M_{(c)} \times 0000010
 \end{aligned}$$
- I moltiplicatori così ottenuti
 - Sono potenze del due
 - Sono sequenze di bit con un solo uno



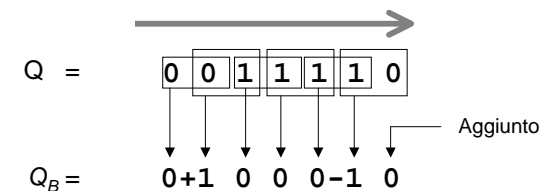
Algoritmo di Booth: codifica del moltiplicatore

- L'algoritmo si basa sulla scomposizione appena vista
- Tale scomposizione è rappresentata come una **codifica del moltiplicatore** basata sulle seguenti regole
- Si consideri un moltiplicatore Q di lunghezza n
 - Si scorre il moltiplicatore da sinistra verso destra
 - Il **moltiplicatore codificato** Q_B si ottiene:
 - Scrivendo il simbolo +1 quando si passa da 0 ad 1
 - Scrivendo il simbolo -1 quando si passa da 1 a 0
 - Scrivendo il simbolo 0 quando due bit successivi sono uguali
 - Se Q termina con 0 aggiungo 0 a Q_B altrimenti aggiungo -1



Algoritmo di Booth: esempio di codifica

- Ad esempio $Q = 30$ è codificato come $Q_B = 0+1000-10$



- Utilizzando tale codifica, i prodotti parziali saranno:
 - 0 con **estensione del segno**, quando $q_{B,i} = 0$
 - $M_{(c)}$ con **estensione del segno**, quando $q_{B,i} = -1$
 - M con **estensione del segno**, quando $q_{B,i} = +1$



Algoritmo di Booth: codifica del moltiplicatore

- Le regole espote per l'algoritmo di Booth possono essere riassunte nella tabella seguente:

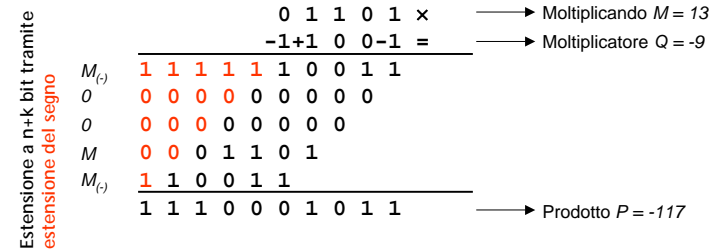
| Moltiplicatore | | Codifica | PP_i |
|----------------|-----------|----------|-------------------------|
| q_i | q_{i-1} | | |
| 0 | 0 | 0 | $0 \times M = 0$ |
| 0 | 1 | +1 | $+1 \times M = M$ |
| 1 | 0 | -1 | $-1 \times M = M_{(c)}$ |
| 1 | 1 | 0 | $0 \times M = 0$ |

- E inoltre, se:
 - $q_0 = 0$, la codifica del bit aggiunto è 0 e quindi il prodotto parziale è 0
 - $q_0 = 1$, la codifica del bit aggiunto è -1 e quindi il prodotto parziale è $M_{(c)}$



Esempio

- Moltiplicare 13×-9 , usando l'algoritmo di Booth su 5 bit
- I valori binari da usare sono:
 - $13 = 01101$ $-13 = 10011$
 - $-9 = 10111$ $-9_B = -1+100-1$
- Il prodotto si esegue quindi nel modo seguente:



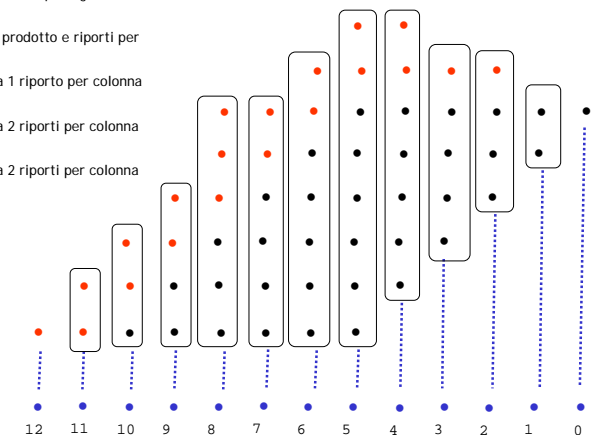
Moltiplicatori combinatori: somma per colonne

- Il metodo è simile a quello utilizzato a mano per effettuare la moltiplicazione
- Si utilizza la **matrice dei prodotti parziali** (matrice di AND) e un insieme di **contatori paralleli**
- Il generico **contatore parallelo** riceve in ingresso **una colonna di prodotti parziali** (e gli eventuali riporti dagli stadi precedenti) e genera il **conteggio degli 1 della colonna**
- Il conteggio generato in ogni stadio produce il **bit del prodotto per lo stadio** considerato e eventuali **riporti per gli stadi successivi**
- Irregolare (contatori diversi)
- Prestazioni: paragonabili a quelle per somma per righe, infatti si ha propagazione di riporti in tutte le colonne



Moltiplicatori combinatori: somma per colonne

- Moltiplicando e moltiplicatore da 6 bit
- In nero la matrice di AND, in rosso i riporti generati dai contatori
- ogni contatore genera 1 bit del prodotto e riporti per le colonne successive
- il contatore di colonna 1 genera 1 riporto per colonna 2
- il contatore di colonna 2 genera 2 riporti per colonna 3 e 4
- il contatore di colonna 3 genera 2 riporti per colonna 4 e 5
- e così via....



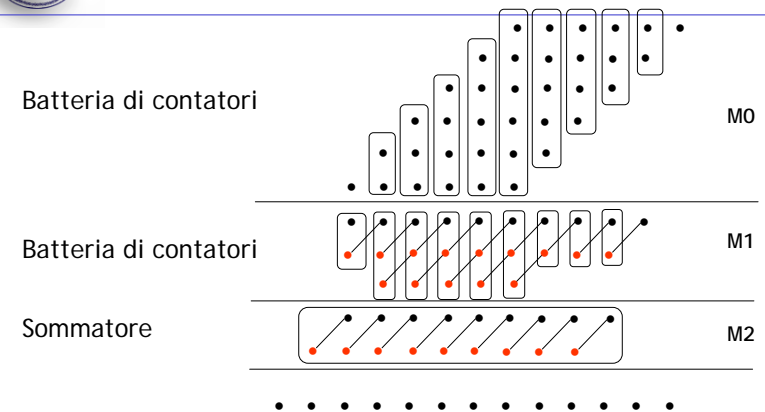


Moltiplicatori combinatori: somma per colonne con riduzione della matrice dei termini prodotto

- Riduzione successiva della matrice dei prodotti parziali
 - La **matrice dei prodotti parziali** M_0 viene ridotta, in termini di righe, tramite contatori paralleli per colonna **che non propagano i riporti**, ma li **usano** (insieme ai bit di somma) **per costruire la matrice ridotta**
 - Il risultato generato dai contatori crea una **matrice successiva M_1** , costituita da un numero inferiore di righe. In questo modo **non c'è propagazione dei riporti all'interno della stessa matrice**
 - Il procedimento viene iterato fino a quando non si ottiene una **matrice di sole due righe**
 - Le due righe costituiscono l'ingresso ad un sommatore
- La riduzione è rapida
- La struttura è irregolare
- Le prestazioni aumentano
 - ipotesi: il tempo di un contatore è identico a quello di un Full-Adder
 - domina il tempo del sommatore finale



Moltiplicatori combinatori: somma per colonne con matrici successive

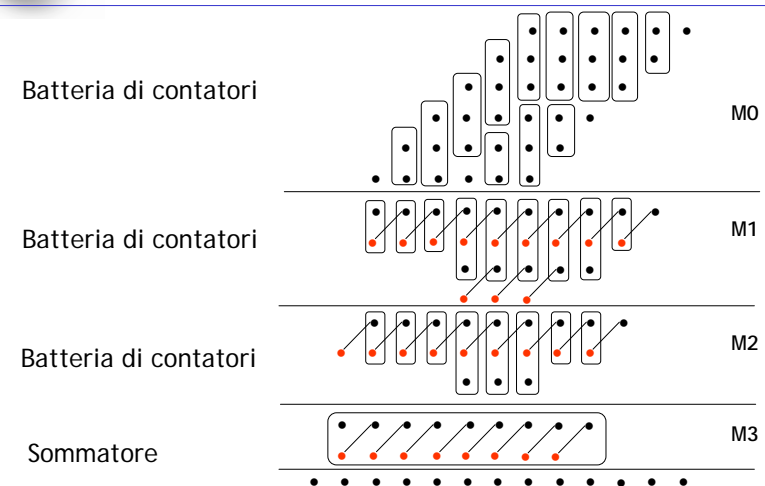


Moltiplicatori combinatori: moltiplicatore di Wallace

- E' basato sulla riduzione successiva della matrice M_0
- Prevede l'utilizzo di soli **contatori a 2 o 3 ingressi**, che sono equivalenti rispettivamente ad un Half-Adder e a un Full-Adder
- Il procedimento di riduzione della matrice a 2 sole righe è più lento rispetto al caso di contatori a ingressi qualsiasi, ma comunque rapido ($\log_{3/2} n$ passi)
 - M_0 di n righe
 - M_1 di $(2/3)n$ righe
 - M_2 di $(2/3)^2 n$ righe
 -
 - M_h di $(2/3)^h n$ righe: se il n° di righe è uguale a 2 la riduzione termina
- La struttura è "regolare"
- Le prestazioni sono dominate dal sommatore finale (veloce)



Moltiplicatori combinatori: moltiplicatore di Wallace





Moltiplicatori sequenziali [1]

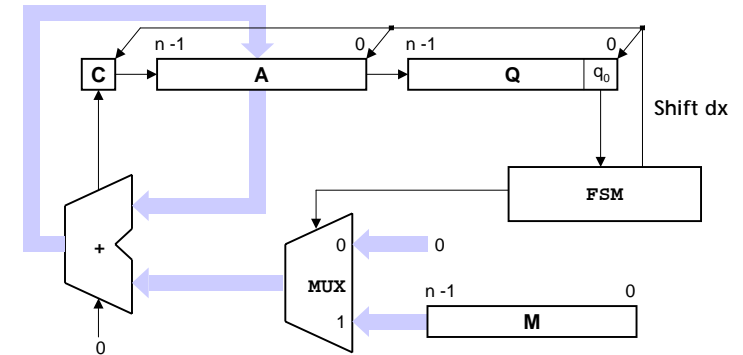
- Moltiplicazione sequenziale tra due numeri di n
- I passi da eseguire sono:
 1. Inizializza a zero un **registro accumulatore A**
 2. Inizializza a zero un **bistabile C per il riporto**
 3. Salva nei **registri Q** ed M moltiplicatore e moltiplicando
 4. Se il bit meno significativo di Q vale 1
 - Somma A ed M
 - Memorizza il risultato in A
 5. Shift a destra del registro [C; A; Q] di una posizione
 6. Ripeti dal punto 4 per n volte
 7. Preleva il risultato della moltiplicazione dai registro [A; Q]

- 57 -



Moltiplicatori sequenziali [2]

Architettura di un moltiplicatore sequenziale



- 58 -



ARITMETICA in VIRGOLA MOBILE

- Confronto con aritmetica in virgola fissa
- Rappresentazione dei valori
- Operazioni
- Struttura di un sommatore/sottrattore



Numeri in virgola fissa

- Fino a questo punto abbiamo assunto che
 - Un vettore di bit rappresentasse sempre un numero intero
 - Eventualmente con segno
- Tutte le considerazioni fatte fino ad ora e tutti i metodi esposti continuano a valere se si attribuisce ai vettori di bit il significato di numeri in **virgola fissa**
- Un sistema di numerazione in virgola fissa è quello in cui:
 - La posizione della virgola decimale è implicita
 - La posizione della virgola decimale uguale in tutti i numeri
- La posizione della virgola equivale alla interpretazione del **valore intero moltiplicato per un fattore di scala**

- 60 -



Numeri in virgola fissa: fattore di scala

- Si consideri ad esempio il vettore di $k+n$ bit (k bit per rappresentare la **parte intera** e n bit per rappresentare la **parte frazionaria**):

$$B = b_{k-1} \dots b_0, b_{-1} \dots b_{-n}$$

- Il suo valore è dato da

$$V(B) = b_{k-1}x2^{k-1} + \dots + b_0x2^0 + \underbrace{b_{-1}x2^{-1} + \dots + b_{-n}x2^{-n}}_{\text{parte frazionaria}}$$

- Il **fattore di scala** che consente di passare dalla rappresentazione intera a quella a virgola fissa è pari a

$$S_n = 2^{-n} = 1 / 2^n$$

- Detti V_I il valore intero e V_{VF} il valore in virgola fissa di B :

$$V_{VF}(B) = V_I(B) \times S_n = V_I(B) \times 2^{-n}$$



Esempio

- Si consideri il vettore binario:

$$B = 010.10110$$

- Il suo valore in virgola fissa è:

$$\begin{aligned} V_{VF}(B) &= 2^1 + 2^{-1} + 2^{-3} + 2^{-4} = 2 + 1/2 + 1/8 + 1/16 \\ &= 43/16 = 2.6875 \end{aligned}$$

- Il fattore di scala da utilizzare per la conversione è:

$$S_5 = 2^{-5} = 1/32 = 0.03125$$

- Il valore di B , considerandolo intero è:

$$V_I(B) = 2^6 + 2^4 + 2^2 + 2^1 = 64 + 16 + 4 + 2 = 86$$

- Da cui, moltiplicando per il fattore di scala, si ha:

$$V_{VF}(B) = V_I(B) \times S_5 = 86 \times 0.03125 = 2.6875$$



Virgola fissa vs. virgola mobile

Intervallo di variazione di un numero binario di **32 bit**

- Codifica intera**

$$0 \leq |V_I(B)| \leq +2^{31} \approx 2.15 \times 10^9$$

- Codifica in virgola fissa**

$$+4.65 \times 10^{-10} \approx +2^{-31} \leq |V_{VF}(B)| \leq +1$$

- Codifica in virgola mobile**

$$\approx +10^{-45} \leq |V_{VM}(B)| \leq \approx +10^{+38}$$

- A **pari numero di bit** disponibili

- con la rappresentazione **intera** o in **virgola fissa**, i valori rappresentati sono distribuiti **uniformemente** nel campo di rappresentabilità
- con la rappresentazione in **virgola mobile**, i valori rappresentati sono distribuiti **non uniformemente** nel campo di rappresentabilità
 - sono "più fitti" vicino allo 0 e "più radi" per valori assoluti grandi

- Nella rappresentazione in **virgola mobile (floating point)** la posizione della virgola è mobile ed è indicata dal valore di un fattore moltiplicativo

(\pm Mantissa \times Base^{Esponente})



Errore di quantizzazione: virgola fissa vs. virgola mobile

- Virgola fissa** (con n bit per la parte frazionaria)

- $E_{Ass} = Val_{Vero} - Val_{Rapp} = \text{costante}$
con $(-1/2)2^{-n} < E_{Ass} < (+1/2)2^{-n}$

- $E_{Rel} = E_{Ass} / Val_{Vero}$
(e cioè $E_{Rel} Val_{Vero} = \text{costante}$)

- tanto più piccolo è il valore vero da rappresentare tanto maggiore è l'errore relativo che si commette nel rappresentarlo
- tanto più grande è il valore vero da rappresentare tanto minore è l'errore relativo che si commette nel rappresentarlo

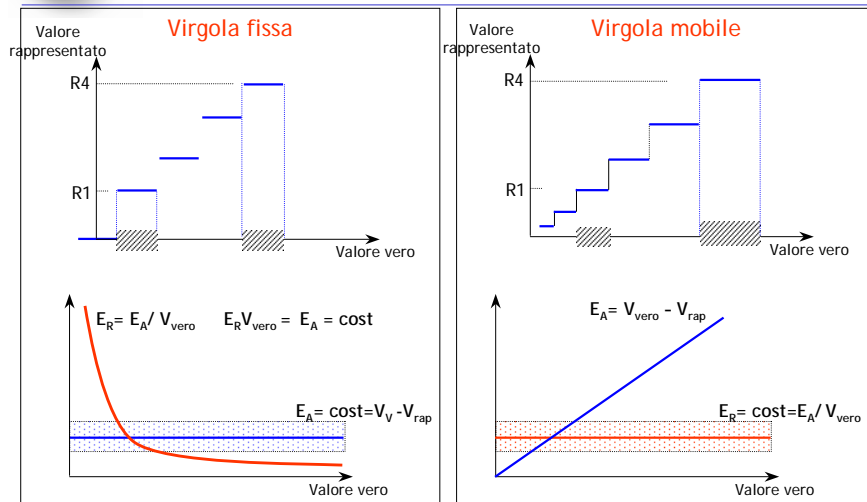
- Virgola mobile**

- $E_{Rel} = \text{costante}$ ($= 2^{-\#bit \text{ della } M}$)

- $E_{Ass} = \text{aumenta all'aumentare del valore vero da rappresentare}$



Errore di quantizzazione: virgola fissa vs. virgola mobile



- 65 -



Esempio

- Numeri in virgola fissa
 - Dato 0.001 ed il suo successivo 0.002
Errore percentuale: $(0.002 - 0.001) / 0.001 * 100 = 100\%$
 - Dato 100.001 ed il suo successivo 100.002
Errore percentuale: $(100.002 - 100.001) / 100.001 * 100 = 0.001\%$
- Numeri in virgola mobile
 - Dato 0.128e-100 ed il suo successivo 0.129e-100
Errore percentuale: $((0.129e-100 - 0.128e-100) / 0.128e-100) * 100 = 0.78125\%$
 - Dato 0.128e+100 ed il suo successivo 0.129e+100
Errore percentuale: $((0.129e+100 - 0.128e+100) / 0.128e+100) * 100 = 0.78125\%$

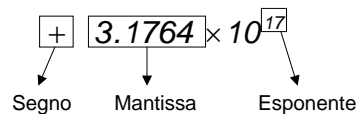
- 66 -



Numeri in virgola mobile

Codifica in virgola mobile per i numeri in base 10

- Si dice **normalizzato** un numero in cui $1 \leq M < 10$



Codifica in virgola mobile per i numeri in base 2

- In un **numero binario in virgola mobile e normalizzato**
 - La prima cifra della mantissa è sempre 1 ($1 \leq M < 2$)
 - Tale cifra non viene rappresentata esplicitamente

- 67 -



Numeri in virgola mobile - Valori rappresentabili

- **IEEE standard:** Numeri floating-point in **singola precisione**

| S | E | M |
|----------------|--------------------|--------------------|
| 1 bit Segno | 8 bit Esponente | 23 bit Mantissa |

- L'**esponente** utilizza la **codifica in eccesso 127**, e cioè il **valore effettivo dell'esponente** è pari a $(E-127)$
 - $E = 0$ e $M = 0$ Rappresenta lo zero (pos/neg)
 - $E = 255$ e $M = 0$ Rappresenta infinito (pos/neg)
 - $E = 255$ e $M \neq 0$ *NaN*
 - $0 < E < 255$ $(-1)^s \times 2^{(E-127)} \times (1, M)$
 $(127 \leq E \leq 254 \text{ esp. positivi}, 126 \leq E \leq 1 \text{ esp. negativi})$
 - $E = 0$ e $M \neq 0$ $(-1)^s \times 2^{-126} \times (0, M)$ non normalizzati
- Standard IEEE 32 bit: intervallo rappresentato $-1.M \times 10^{-38} \leq x \leq +1.M \times 10^{38}$
- La precisione consentita è di circa 7 cifre decimali

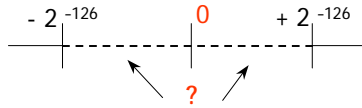
- 68 -



Numeri in virgola mobile - Valori rappresentabili

- Motivazione della rappresentazione **non normalizzata**
 - $E = 0$ e $M \neq 0$ $(-1)^s \times 2^{-126} \times (0, M)$ non normalizzati
- Il **valore** più piccolo rappresentabile **normalizzato** è

$$\pm 2^{1-127} \times 1,00\dots00 = \pm 2^{-126}$$
- che espresso in virgola mobile da $E=1$ e $M = 0$



rappresentazione **non normalizzata** $E=0$ e $M \neq 0$

Interpretata nel modo seguente:

$$\text{Valore numerico} = \pm 2^{-126} \times 0, \dots$$

Il più piccolo valore rappresentabile è

$$\pm 2^{-126} \times 0,00\dots01 = \pm 2^{-126} \times 2^{-23} = \pm 2^{-149}$$



Operazioni in virgola mobile

- Le operazioni che si possono compiere su numeri in virgola mobile sono:
 - Somma
 - Sottrazione
 - Moltiplicazione
 - Divisione
 - Elevamento a potenza
 - Estrazione di radice
- Inoltre sono definite le operazioni di:
 - Normalizzazione
 - Troncamento



Operazioni in virgola mobile

- L'esecuzione di una operazione in virgola mobile può provocare una **eccezione**
- Una **eccezione** è il risultato di una operazione anomala, quale, ad esempio:
 - Divisione per zero
 - Estrazione della radice quadrata di un numero negativo
- Le eccezioni che vengono generate dalle unità aritmetiche in virgola mobile sono:
 - Operazione non valida
 - Divisione per zero
 - Overflow
 - Underflow



Operazioni in virgola mobile: *normalizzazione*

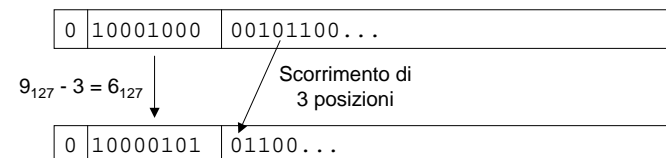
- Tutte le operazioni descritte nel seguito operano su numeri normalizzati (1 implicito prima della virgola)
- Se l'**1 implicito manca**, la **normalizzazione** di un numero con mantissa M ed esponente n , si esegue come segue:
 - Si fa scorrere verso sinistra la mantissa M fino al primo uno, compreso; sia k il numero di posizioni di tale scorrimento
 - Si sottrae k all'esponente n

Ricorda:

Scorrimento a sx equivale a moltiplicazione

Scorrimento a dx equivale a divisione

- Ad esempio:





Operazioni in virgola mobile: *somma e sottrazione*

- La **somma o sottrazione** tra numeri in virgola mobile viene eseguita secondo i seguenti passi:
 - Si sceglie il numero con esponente minore
 - Si fa scorrere la sua mantissa a destra un numero di bit pari alla differenza dei due esponenti
 - Si assegna all'esponente del risultato il maggiore tra gli esponenti degli operandi
 - Si esegue l'operazione di somma (algebraica) tra le mantisse per determinare il valore ed il segno del risultato
 - Si normalizza il risultato così ottenuto
 - Non sempre quest'ultima operazione è necessaria
 - **Attenzione!!!** Il riporto si può propagare anche dopo la posizione della virgola



Operazioni in virgola mobile : *moltiplicazione*

- La **moltiplicazione** tra numeri in virgola mobile viene eseguita secondo i seguenti passi:
 - Si sommano gli esponenti e si sottrae 127
 - Si calcola il risultato della moltiplicazione delle mantisse
 - Si determina il segno del risultato
 - Si normalizza il risultato così ottenuto
 - Non sempre quest'ultima operazione è necessaria
- La sottrazione di 127 dalla somma degli esponenti è necessaria in quanto sono rappresentati in eccesso 127

$$E_{a,127} = E_a + 127$$

$$E_{b,127} = E_b + 127$$

$$E_{axb,127} = E_{axb} + 127 = (E_a + 127) + (E_b + 127) - 127$$



Operazioni in virgola mobile : *divisione*

- La **divisione** tra numeri in virgola mobile viene eseguita secondo i seguenti passi:
 - Si sottraggono gli esponenti e si somma 127
 - Si calcola il risultato della divisione delle mantisse
 - Si determina il segno del risultato
 - Si normalizza il risultato così ottenuto
 - Non sempre quest'ultima operazione è necessaria
- La somma di 127 alla differenza degli esponenti è necessaria in quanto sono rappresentati in eccesso 127

$$E_{a,127} = E_a + 127$$

$$E_{b,127} = E_b + 127$$

$$E_{a/b,127} = E_{a/b} + 127 = (E_a + 127) - (E_b + 127) + 127$$



Operazioni in virgola mobile: *troncamento*

- Spesso accade di rappresentare i risultati intermedi di una operazione con una precisione maggiore di quella degli operandi e del risultato
- Al termine dell'operazione è necessario effettuare una operazione di **troncamento**
- Il troncamento serve a rimuovere un certo numero di bit per ottenere una rappresentazione approssimata del risultato
- Si consideri il valore numerico rappresentato dal vettore:

$$B = 0.b_{-1} \dots b_{-(k-1)} b_{-k} b_{-(k+1)} \dots b_{-n}$$

- Si voglia effettuare **troncamento al bit k-esimo**



Operazioni in virgola mobile: *troncamento*

□ Chopping

- Consiste nell'ignorare i bit dal k -esimo all' n -esimo
- Questo metodo è *polarizzato* o *biased*
- L'errore è sempre positivo e varia nell'intervallo:
 $0 < \varepsilon < +(2^{k+1} - 2^n)$

□ Rounding

- Se il bit k -esimo vale 0, lasciare invariato il bit in posizione $(k-1)$ e ignorare i bit dal k -esimo all' n -esimo
- Se il bit k -esimo vale 1, sommare 1 in posizione $(k-1)$ e ignorare i bit dal k -esimo all' n -esimo
- Questo metodo è *simmetrico* o *unbiased*
- L'errore è centrato sullo zero e vale:
 $-(2^{k+1} - 2^n) < \varepsilon < +(2^{k+1} - 2^n)$



Architetture per aritmetica in virgola mobile: *sommatore*

- I circuiti per la realizzazione delle operazioni in virgola mobile sono molto complessi
- Si consideri l'algoritmo per la **somma** secondo lo standard IEEE Single Precision:
 - Si sceglie il numero con esponente minore e si fa scorrere la sua mantissa a destra un numero di bit pari alla differenza dei due esponenti
 - Si assegna all'esponente del risultato il maggiore tra gli esponenti degli operandi
 - Si esegue l'operazione di somma tra le mantisse per determinare il valore ed il segno del risultato
 - Si normalizza il risultato così ottenuto
 - Non sempre quest'ultima operazione è necessaria
- Nota:
 - se A o $B = \pm\infty$
 - se A o $B = 0$
 - se la differenza tra gli esponenti è maggiore o uguale al numero di bit a disposizione per le mantisse
- è inutile fare la somma



Sommatore in virgola mobile [1]

- Nel seguito viene sviluppato un **sommatore floating point**
- I numeri A e B sono rappresentati
 - Su 32 bit
 - Secondo lo standard IEEE Single Precision
- Gli operandi A e B sono composti come segue:

$$A = \{ S_A, E_A, M_A \}$$

$$B = \{ S_B, E_B, M_B \}$$

□ In cui:

- S_A, S_B Segno, 1 bit
- E_A, E_B Esponente in eccesso 127, 8 bit
- M_A, M_B Mantissa, 23 bit



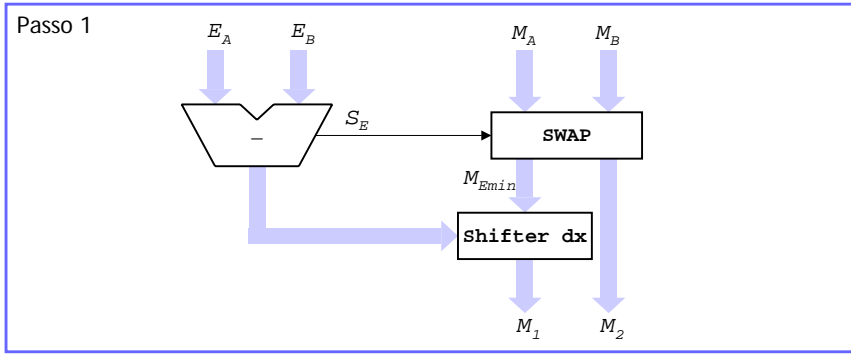
Sommatore in virgola mobile [2]

- Passo 1
 - Si sceglie il numero con esponente minore e si fa scorrere la sua mantissa a destra un numero di bit pari alla differenza dei due esponenti
- Richiede le seguenti operazioni:
 - Individuazione dell'esponente minore E_{min}
 - Calcolo della differenza tra gli esponenti $d = |E_A - E_B|$
 - Selezione della mantissa dell'operando con esponente M_{Emin}
 - Scorrimento della mantissa M_{Emin} di d posizioni a dx (tenendo conto dell'1 implicito)
- Il calcolo della differenza tra gli esponenti consente allo stesso tempo (analizzandone il segno S_d) di individuare l'esponente minore



Sommatore in virgola mobile [3]

- Questa prima sezione del sommatore calcola:
 - M_1 La mantissa del numero con esponente minore, opportunamente shiftata
 - M_2 La mantissa del numero con esponente maggiore

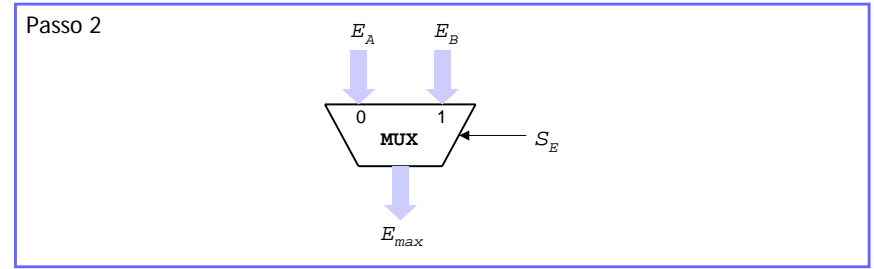


- 81 -



Sommatore in virgola mobile [4]

- Passo 2
 - Si assegna all'esponente del risultato il maggiore tra gli esponenti degli operandi
- Richiede le seguenti operazioni:
 - Selezione dell'esponente minore E_{max} in base a S_E .
 - Si riutilizza il segno S_E della differenza tra gli esponenti

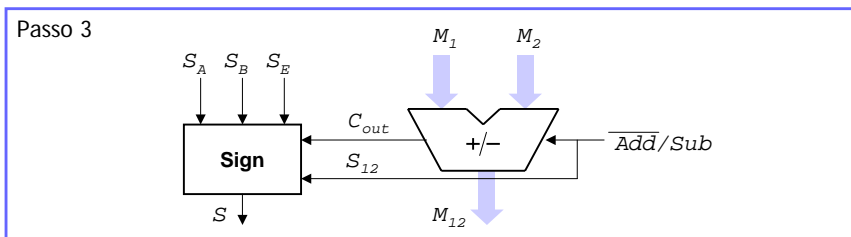


- 82 -



Sommatore in virgola mobile [5]

- Passo 3
 - Si esegue l'operazione di somma tra le mantisse per determinare il valore ed il segno del risultato
- Richiede le seguenti operazioni:
 - Calcolo della somma algebrica M_{12} delle mantisse M_1 ed M_2 ottenute al primo passo, e relativo segno
 - Si utilizza un sommatore/sottrattore su **24 bit con riporto**



- 83 -



Sommatore in virgola mobile [6]

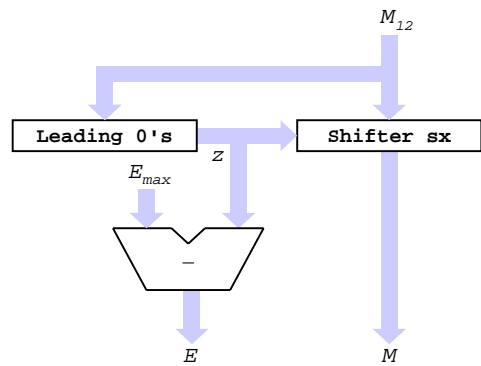
- Se $C_{out} = 0$ e M_{12} normalizzata $E = E_{max}$
- Passo 4
 - Si normalizza il risultato così ottenuto
- Richiede le seguenti operazioni
 - Se $C_{out} = 1$, $Shdx$ M_{12} e $E = E_{max} + 1$, eventuale troncamento
- Altrimenti ($C_{out} = 0$ e M_{12} non normalizzata)
 - Individuazione del numero z degli zeri nei bit più significativi della mantissa M_{12}
 - Shift sx della mantissa M_{12}
 - Calcolo del nuovo esponente $E = E_{max} - z$
 - A tale scopo sono necessari:
 - Un circuito per il calcolo dei *leading zeroes*
 - Un sottrattore su 8 bit
 - Uno *shifter* per l'allineamento della mantissa

- 84 -



Sommatore in virgola mobile [7]

Passo 4



Sommatore in virgola mobile [8]

