POLITECNICO
MILANO 1863

Aritmetica - Parte 2

Federico Reghenzani

federico.reghenzani@polimi.it

Reti Logiche

Scuola di Ingegneria Industriale e dell'Informazione

Figure sommatore CLA completo

Figure sommatore CLA completo

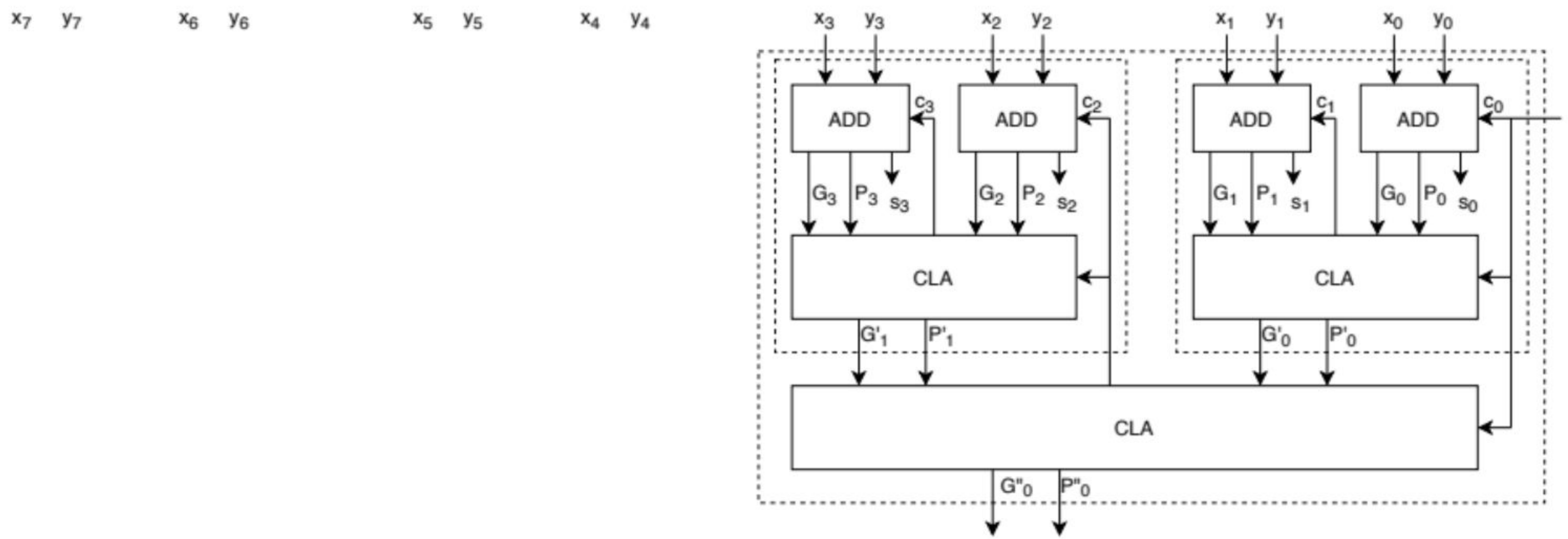
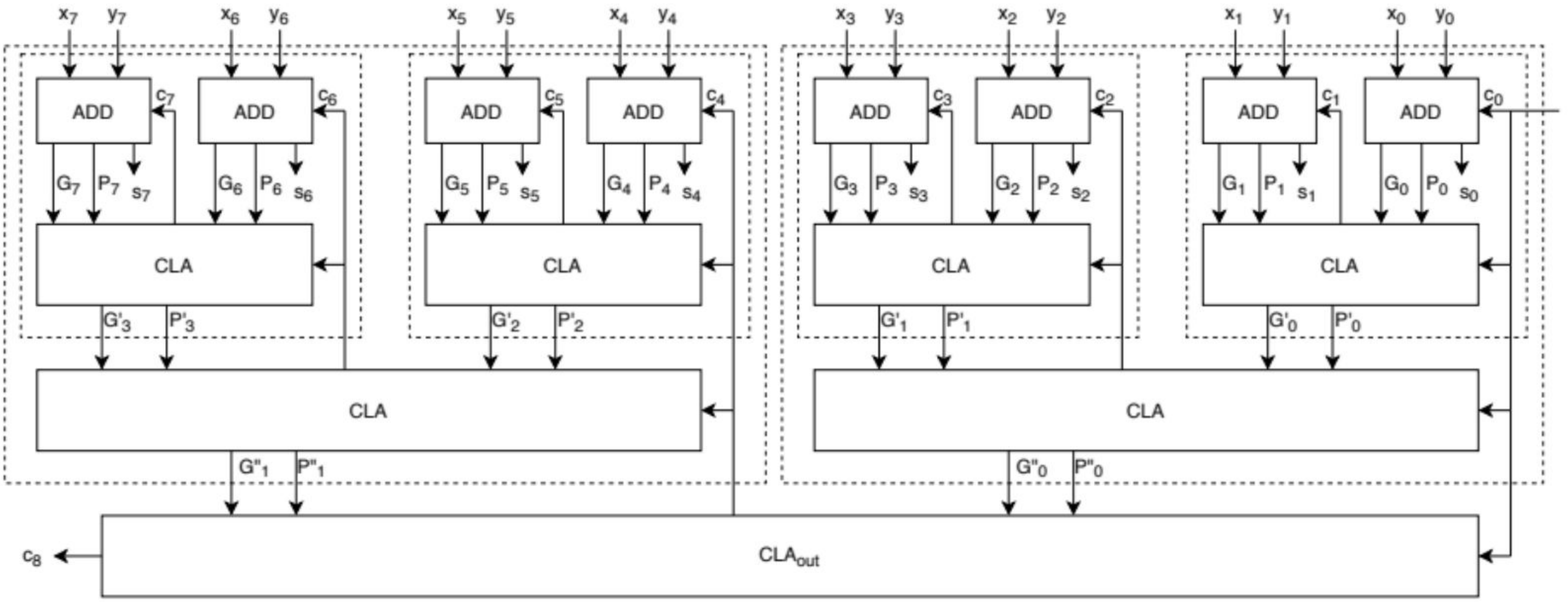


Figure sommatore CLA completo



Sommario

In questa seconda parte vedremo i **moltiplicatori**:

- Moltiplicatore parallelo
- Moltiplicatore di Wallace
- Algoritmo di Booth

E parleremo dei **floating point**:

- Rappresentazione
- Cenni alle operazioni
- Errori

Moltiplicazione binaria

La moltiplicazione binaria con operandi **positivi** si esegue come le normali moltiplicazioni decimali “in colonna”:

```
     1101 *
     1011 =
----------
     1101 +
    1101  +
   0000   +
  1101    =
----------
10001111
```

Il risultato di una moltiplicazione di due numeri binari di **n** bit (moltiplicando) e **m** bit (moltiplicatore) può essere codificato su un numero binario di **n+m** bit

Moltiplicazione binaria - Operandi negativi

La moltiplicazione binaria con operandi **negativi** in complemento a 2 si esegue come precedentemente, ma:

- Se il **moltiplicatore è negativo**, bisogna invertire il segno di moltiplicando e moltiplicatore
  - P=A*B, se B<0, si calcola P=(-A)*(-B)

- Si estende il segno dei prodotti parziali, tanti bit fino a quanto il prodotto risulta grande (n+m visto precedentemente)

Moltiplicazione binaria - Operandi negativi

Esempio con operandi codificati su 5 bit:

```
     10011 *   = -13₁₀
     00101 =   = 5₁₀
-----------
     10011 +
    00000  +
   10011   +
  00000    +
 00000     =
-----------
```

```
     10011 *   = -13₁₀
     00101 =   = 5₁₀
------------
1111110011 +
000000000  +
11110011   +
0000000    +
000000     =
------------
1110111111     = -65₁₀
```

Si ignora qualsiasi bit in posizione > n + m

Moltiplicazione binaria - Operandi negativi

Esempio con operandi codificati su 5 bit e moltiplicatore negativo:

```
     00111 *    =   7_10
     11000 =    = -8_10
----------
     00000 +
    00000  +
   00000   +
  00111    +
 00111     =
----------
```

```
      00111 *    =   7_10
      11000 =    = -8_10
------------
0000000000 +
000000000  +
00000000   +
0000111    +
000111     =
------------
0010101000       = 168_10
```

Moltiplicazione binaria - Operandi negativi

Esempio con operandi codificati su 5 bit e moltiplicatore negativo:

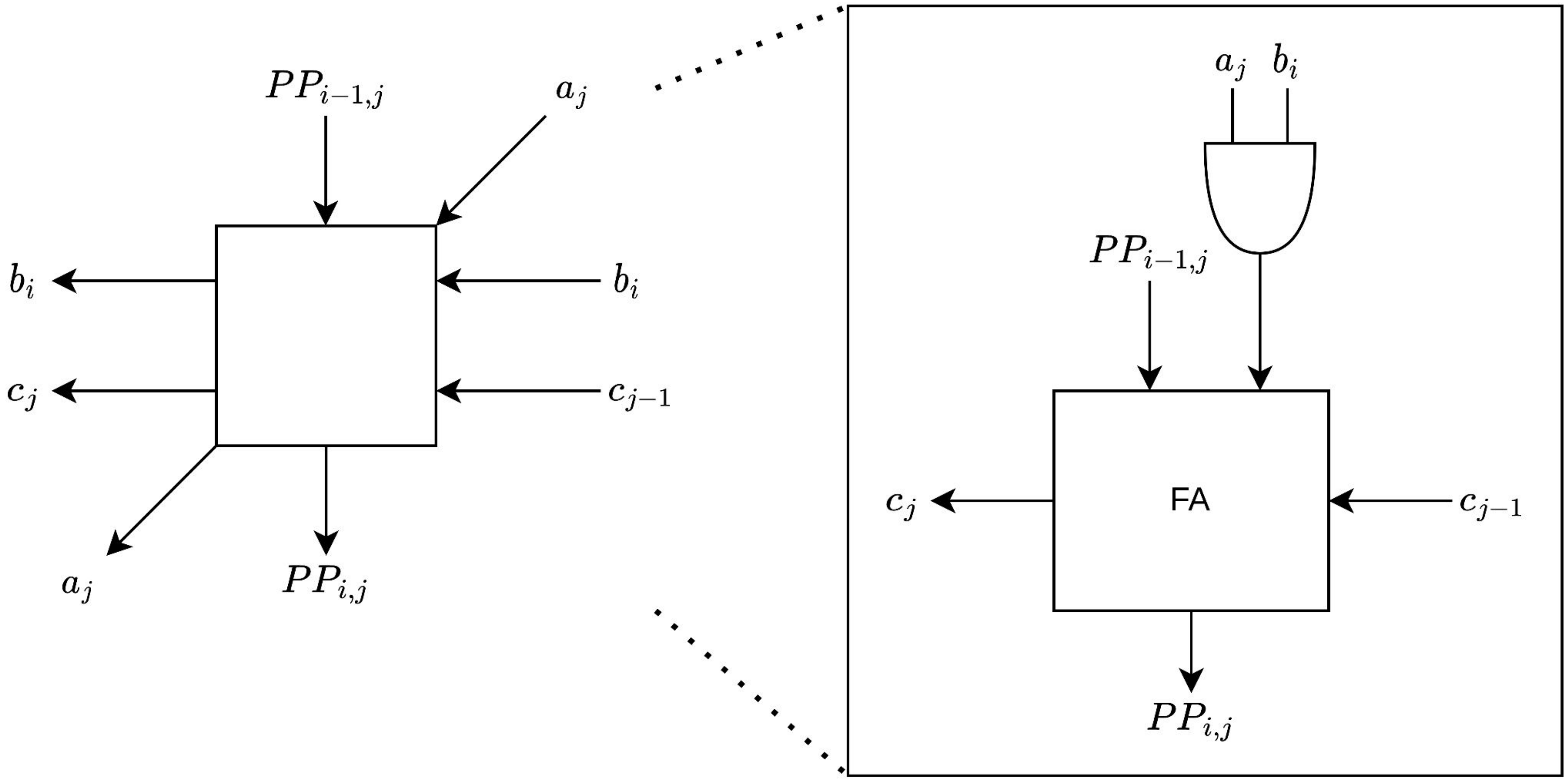
```
00111 *     =  7₁₀
11000 =     = -8₁₀
```

```
   11001 *     = -7₁₀
   01000 =     =  8₁₀
----------
   00000 +
  00000  +
 00000   +
11001    +
00000    =
----------
```

```
   11001 *     = -7₁₀
   01000 =     =  8₁₀
------------
0000000000 +
000000000  +
00000000   +
1111001    +
000000     =
------------
1111001000     = -56₁₀
```

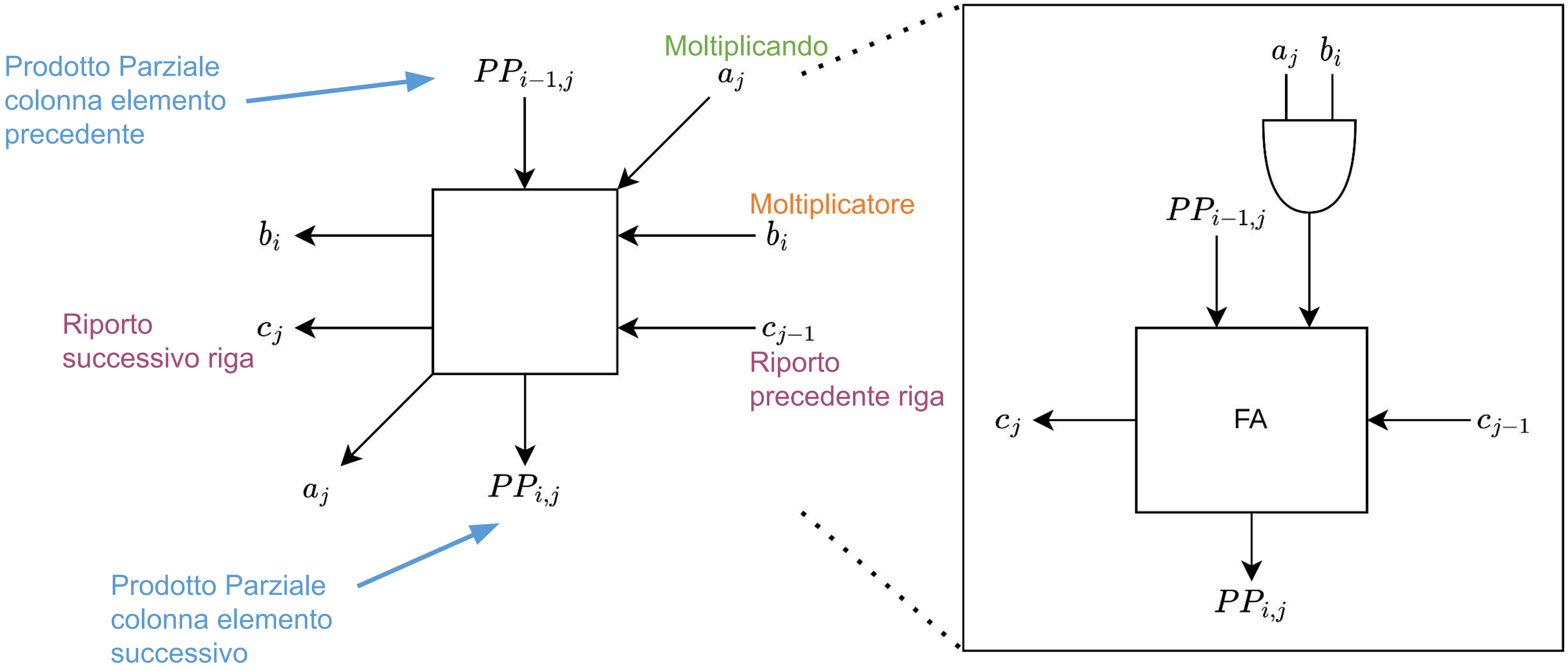
Moltiplicatore parallelo - Cella

Si costruisce un cella moltiplicatrice come segue:

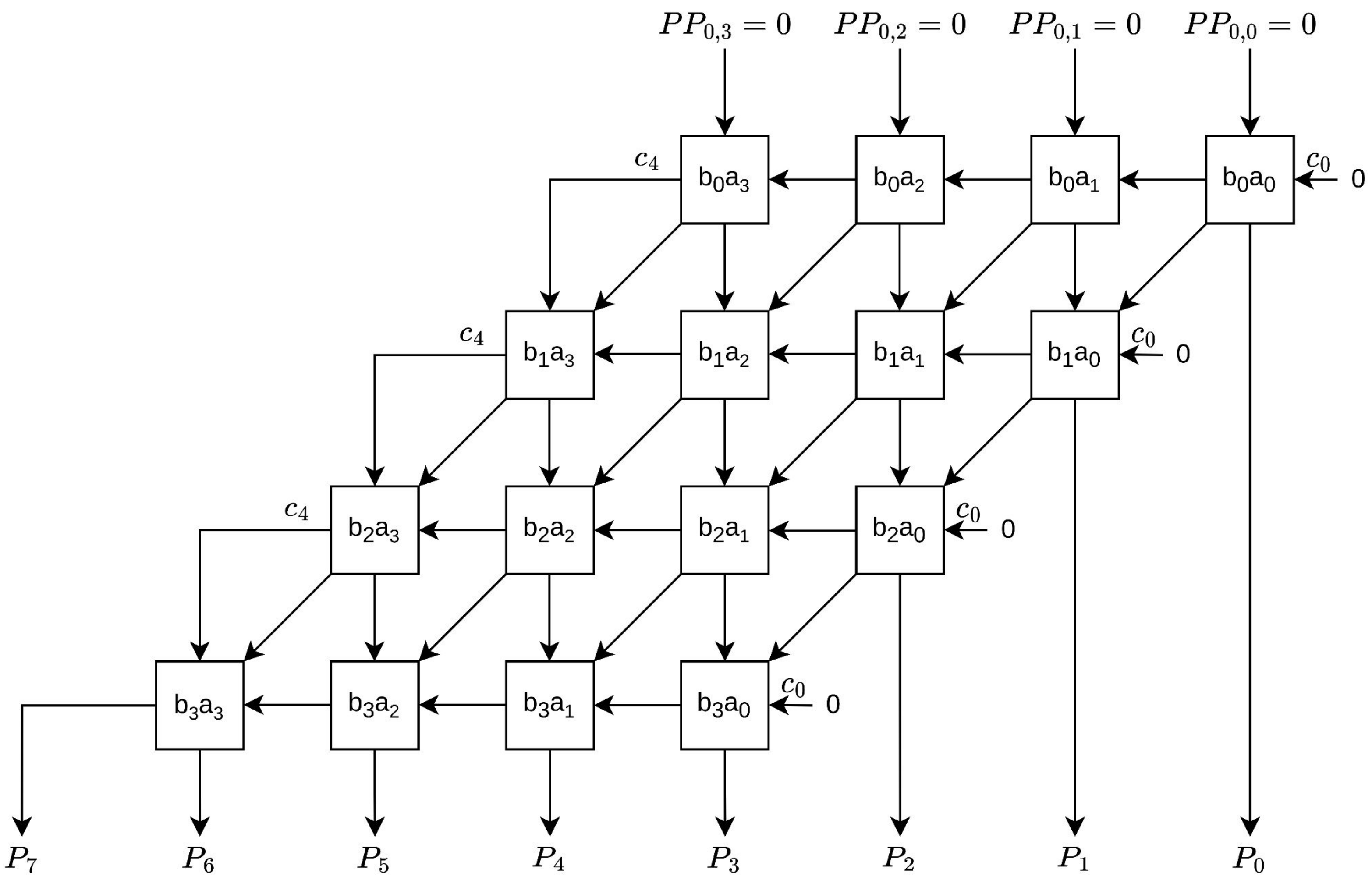


Moltiplicatore parallelo - Cella

Si costruisce un cella moltiplicatrice come segue:

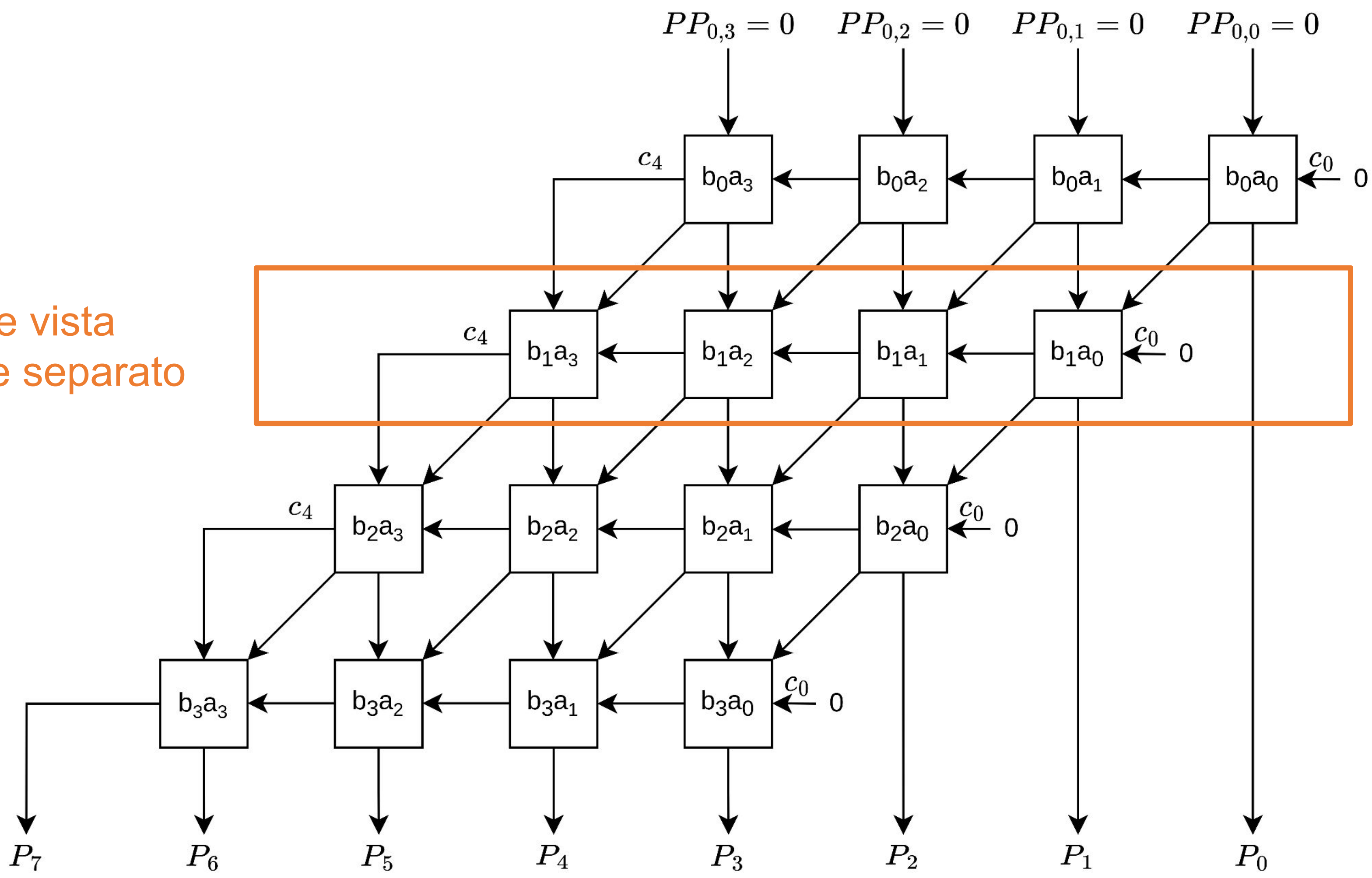


Moltiplicatore parallelo - Circuito completo 4x4



Moltiplicatore parallelo - Circuito completo 4x4

Ogni riga può essere vista
come un sommatore separato



Moltiplicatore parallelo - Costo

Necessitiamo di **m** sommatori da **n** bit

- La prima riga può essere sostituita da **n** porte and, anziché un sommatore vero e proprio, riducendo così a **m-1** il numero di sommatori necessari

Il ritardo dipende principalmente dal tipo di sommatore utilizzato ed è lineare rispetto al numero di livelli (**m**)

Moltiplicatore di Wallace

Il moltiplicatore di **Wallace**:

- Utilizza unicamente HA e FA per ridurre la matrice delle somme di prodotti parziali in due sole righe
- Una volta ridotta, si può utilizzare un solo sommatore finale

Moltiplicatore di Wallace

Fase 1:

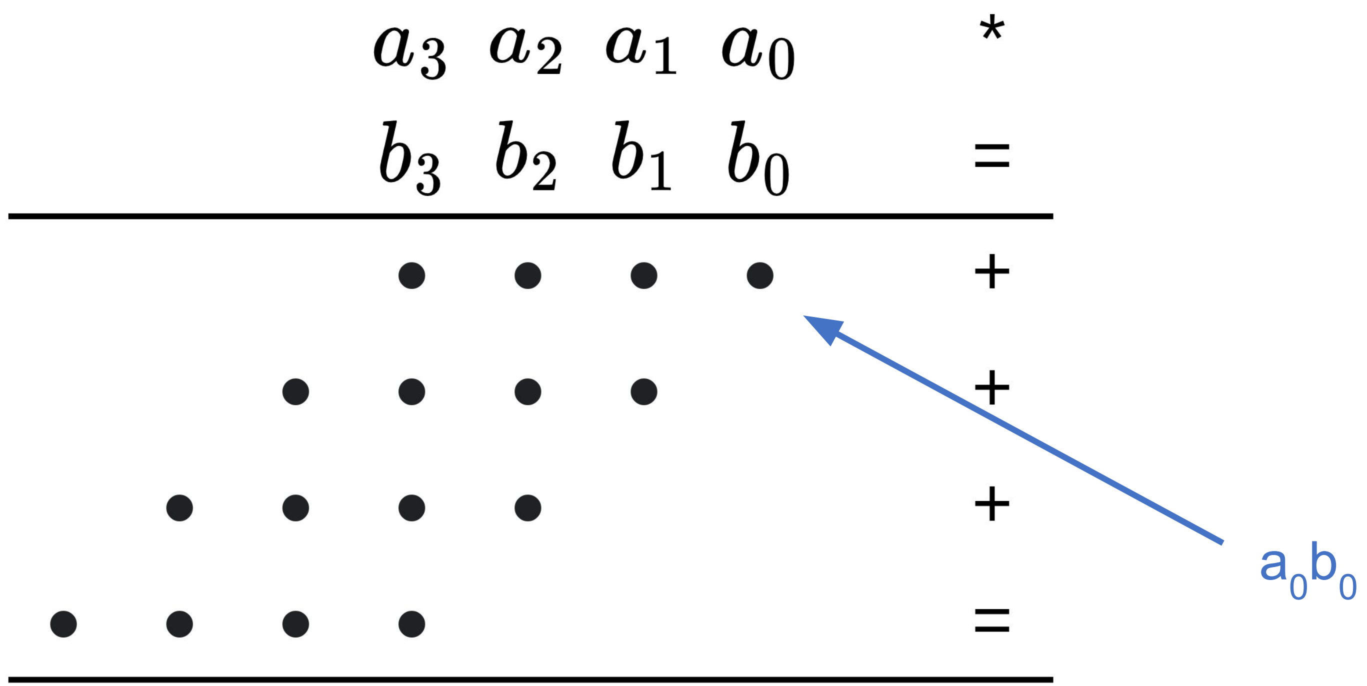
- Si moltiplicano tutte le coppie di bit degli operandi:

a_0b_0, a_0b_1, a_0b_2, …, a_1b_0, a_1b_1, …

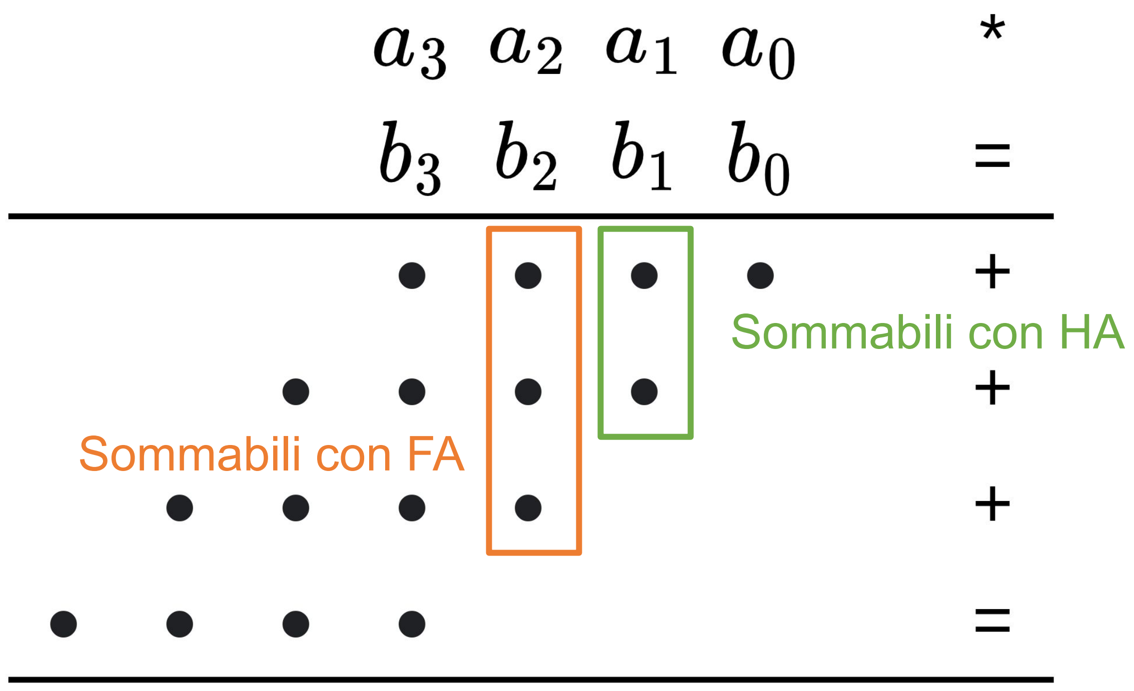
- Necessitiamo di **n** x **m** porte **AND**

Moltiplicatore di Wallace

Fase 2: Riduzione della matrice dei prodotti parziali con HA e FA

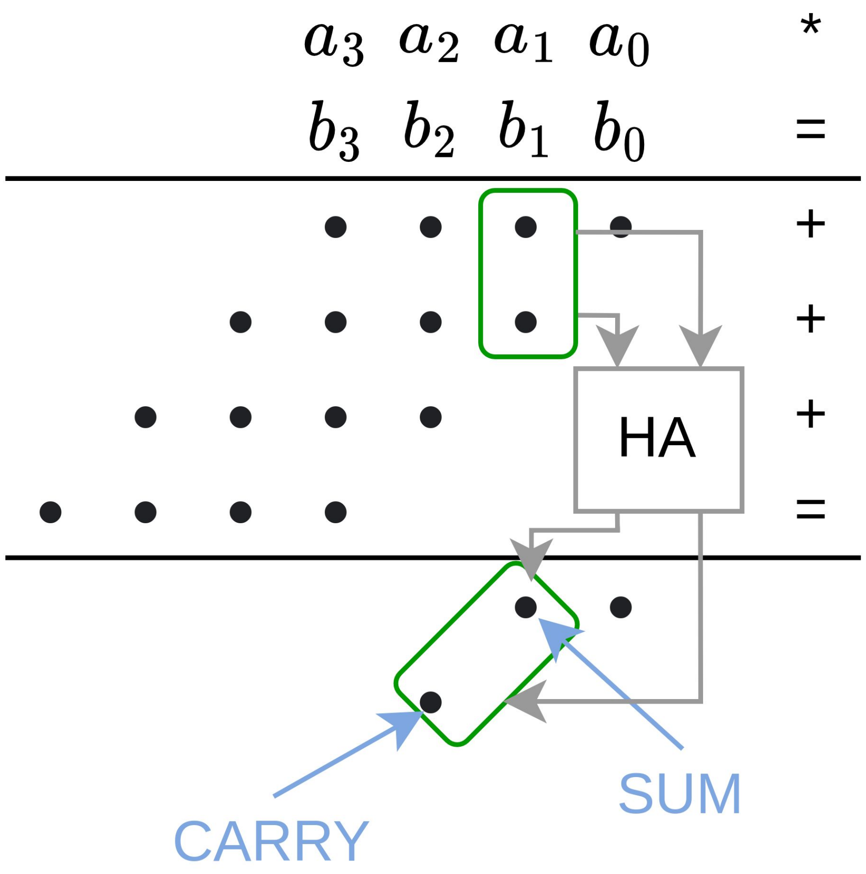


Fase 2: Riduzione della matrice dei prodotti parziali con HA e FA



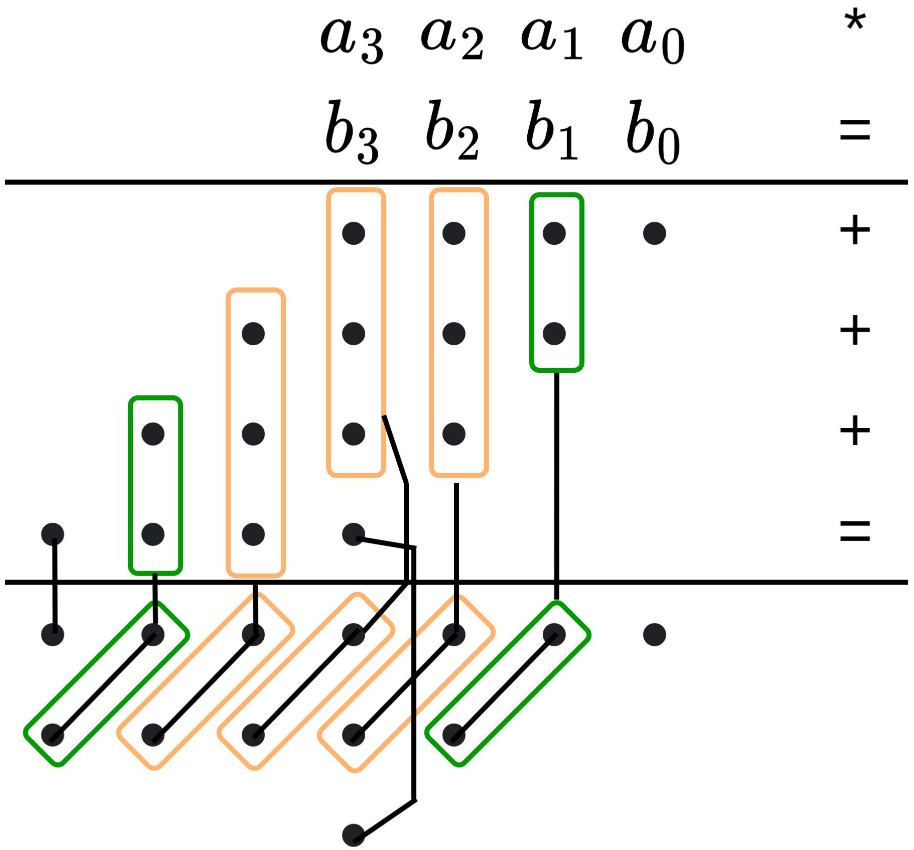
Moltiplicatore di Wallace

Fase 2: Riduzione della matrice dei prodotti parziali con HA e FA



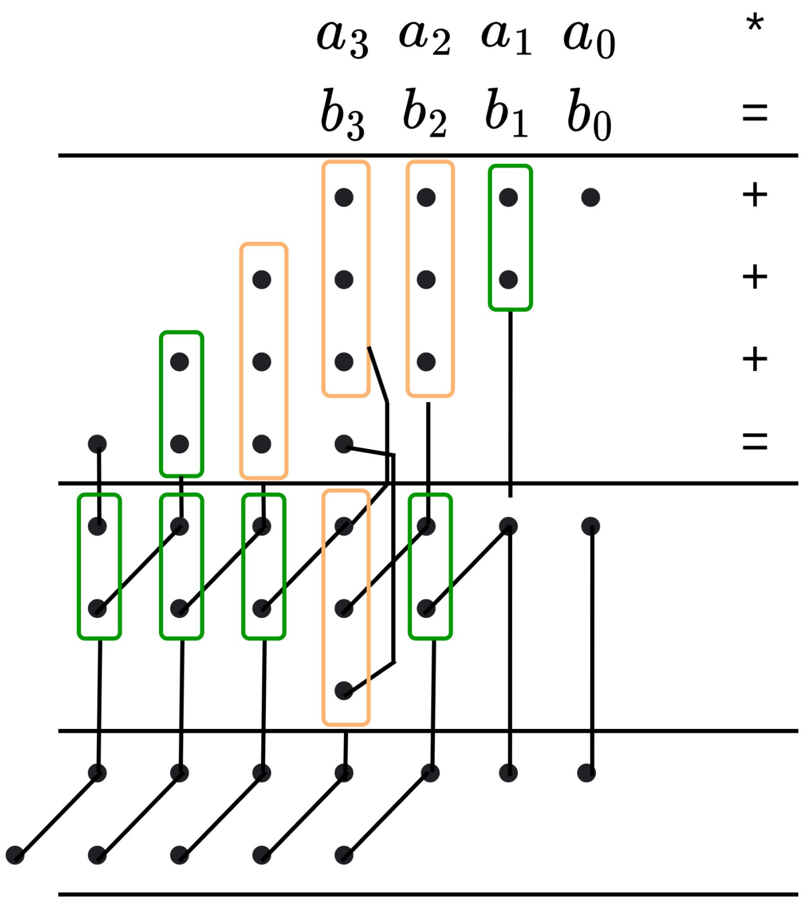
Moltiplicatore di Wallace

Fase 2: Riduzione della matrice dei prodotti parziali con HA e FA



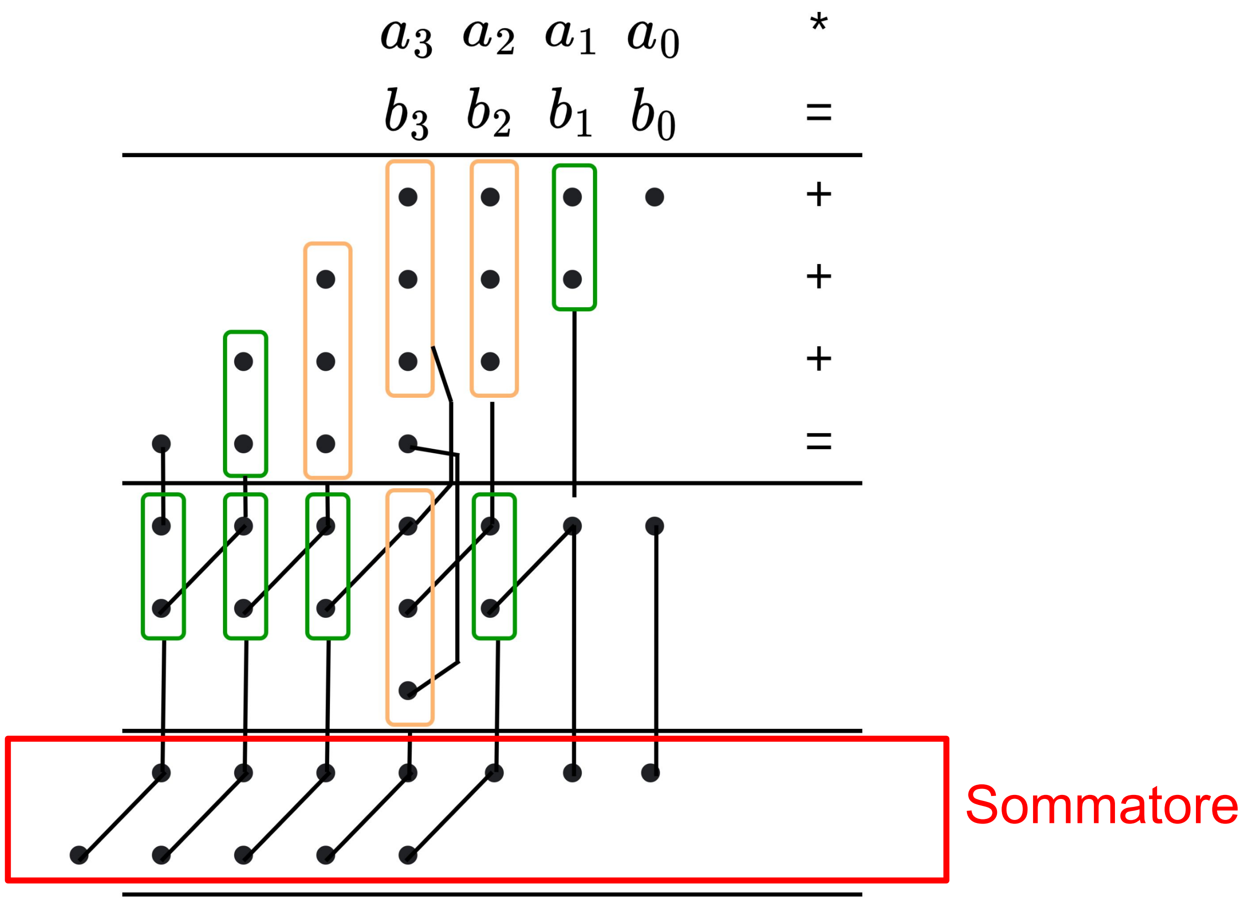
Moltiplicatore di Wallace

Fase 2: Riduzione della matrice dei prodotti parziali con HA e FA



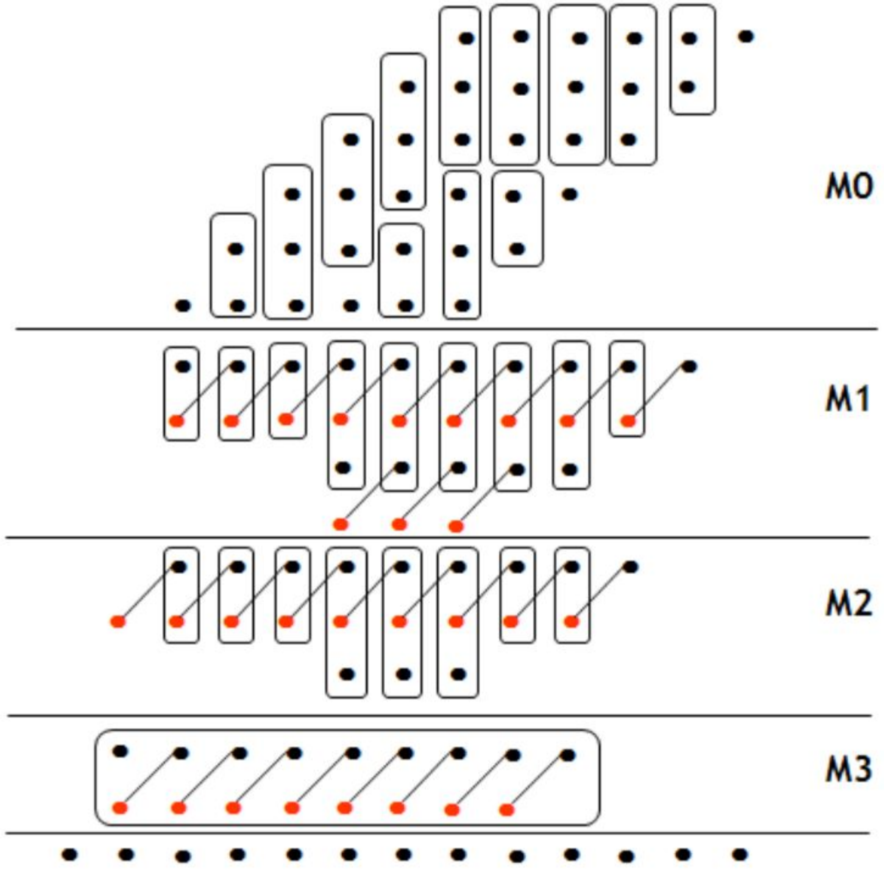
Moltiplicatore di Wallace

Fase 3: Raggiunte 2 righe, utilizziamo un sommatore per produrre il risultato



Moltiplicatore di Wallace

Esempio a 6 bit



Moltiplicatore di Dadda

Non lo vedremo nel corso, ma simile al moltiplicatore di Wallace è il moltiplicatore di Dadda (inventato da Luigi Dadda nel 1965)

- E’ uno dei più comuni moltiplicatori utilizzati nelle moderne CPU

Moltiplicatori in caso di operandi in complemento a 2

In presenza di operandi in complemento a 2 con **segno negativo**, i moltiplicatori si complicano. Una prima possibilità è applicare la regola vista precedentemente, ovvero di invertire i segni degli operandi qualora il moltiplicatore qualora fosse negativo e estendere il segno dei prodotti parziali

- Questa tecnica è molto onerosa in termini di porte logiche e ritardi

Moltiplicatori in caso di operandi in complemento a 2

Una possibile soluzione è la seguente (esempio con codifica a 4 bit):

- Si consideri moltiplicando a_3, a_2, a_1, a_0 e moltiplicatore b_3, b_2, b_1, b_0
- Essi rappresentano i seguenti numeri:
  - $A = (-a_3) * 2^3 + a_2\, 2^2 + a_1\, 2^1 + a_0\, 2^0$
  - $B = (-b_3) * 2^3 + b_2\, 2^2 + b_1\, 2^1 + b_0\, 2^0$

Moltiplicatori in caso di operandi in complemento a 2

I prodotti parziali sono quindi:

$$
\begin{array}{ccccccc|c}
 & & & a_3 & a_2 & a_1 & a_0 & \\
\hline
 & & & -a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & b_0 \\
 & & -a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & b_1 \\
 & -a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & b_2 \\
+a_3b_3 & -a_2b_3 & -a_1b_3 & -a_0b_3 & & & & b_3
\end{array}
$$

Attenzione al segno

Moltiplicatori in caso di operandi in complemento a 2

Si divide la tabella in due tabelle:

			a_3	a_2	a_1	a_0	
			$-a_3b_0$	a_2b_0	a_1b_0	a_0b_0	b_0
		$-a_3b_1$	a_2b_1	a_1b_1	a_0b_1		b_1
	$-a_3b_2$	a_2b_2	a_1b_2	a_0b_2			b_2
$+a_3b_3$	$-a_2b_3$	$-a_1b_3$	$-a_0b_3$				b_3

Moltiplicatori in caso di operandi in complemento a 2

P_+

			a_3	a_2	a_1	a_0	
			0	a_2b_0	a_1b_0	a_0b_0	
		0	a_2b_1	a_1b_1	a_0b_1		
	0	a_2b_2	a_1b_2	a_0b_2			
$+a_3b_3$	0	0	0				

P_-

			a_3	a_2	a_1	a_0	
			$-a_3b_0$	0	0	0	b_0
		$-a_3b_1$	0	0	0		b_1
	$-a_3b_2$	0	0	0			b_2
0	$-a_2b_3$	$-a_1b_3$	$-a_0b_3$				b_3

Moltiplicatori in caso di operandi in complemento a 2

- P+ e P- si sommano separatamente con uno dei metodi visti precedentemente
- Infine si calcola con un sommatore $\mathbf{P = P_+ - P_-}$ e si ottiene il risultato corretto in complemento a 2

>> Esercizio alla lavagna

Algoritmo di Booth

L'**algoritmo di Booth** consente di codificare il moltiplicatore per:

- Consentire l'uso di operandi di segno qualsiasi
- Ridurre il numero di prodotti parziali da sommare attraverso i metodi visti in precedenza

L'algoritmo di Booth è tanto più efficiente quante sono le sequenze di "1" nel moltiplicatori

Vedremo i metodi radix-2 e radix-4

Algoritmo di Booth

Esempio: 10 x -5 = 01010_2 x 11011_2

Fase 1:

- Si aggiunge uno 0 nella posizione meno significativa del moltiplicatore 110110
- Si scompone il moltiplicatore in coppie da 2 sovrapposte a partire da destra, estendendo il segno se necessario:
  11 / 10 / 01 / 11 / 10

Algoritmo di Booth

Fase 2:

- Si applicano le seguenti regole (**radix-2**) e si sostituiscono alle coppie:

Coppia moltiplicatore	**Codifica**
00	0
01	1
10	-1
11	0

11 / 10 / 01 / 11 / 10 = 0 / -1 / 1 / 0 / -1

Algoritmo di Booth

Fase 3:

- Si esegue la moltiplicazione con il moltiplicatore nella nuova codifica:

```
           0 1 0 1 0 *  = 10₁₀
           0-1 1 0-1 =  = -5₁₀ (radix-2)
-----------------------
 1 1 1 1 1 1 0 1 1 0 +
 0 0 0 0 0 0 0 0 0   +
 0 0 0 0 1 0 1 0     +
 1 1 1 0 1 1 0       +
 0 0 0 0 0 0
-----------------------
 1 1 1 1 0 0 1 1 1 0     = -50₁₀
```

Algoritmo di Booth - Radix 4

Booth radix-4 funziona con la stessa logica, ma si prendono gruppi sovrapposti di 3 bit:

001001100 = 001 / 100 / 011 / 100

Algoritmo di Booth - Radix 4

E si applica la seguente codifica:

Tripletta moltiplicatore	Codifica
000	0
001	1
010	1
011	2
100	-2
101	-1
110	-1
111	0

Algoritmo di Booth - Radix 4

Booth radix-4 funziona con la stessa logica, ma si prendono gruppi sovrapposti di 3 bit:

001001100 = 001 / 100 / 011 / 100

= 1 -2 2 -2

Algoritmo di Booth - Radix 4

$$-3_{10} * 38_{10}$$

$$1111101_2 * 0100110_2$$

```
            1 1 1 1 1 0 1 *  = -3₁₀
                1-2 2-2 =  = 38₁₀ (radix-4)
--------------------------
0 0 0 0 0 0 0 0 0 0 1 1 0 +
1 1 1 1 1 1 1 1 0 1 0     +
0 0 0 0 0 0 1 1 0         +
1 1 1 1 1 0 1             +
--------------------------
1 1 1 1 1 1 0 0 0 1 1 1 0    = -114₁₀
```

Algoritmo di Booth - Radix 4

$-3_{10} * 38_{10}$

$1111101_2 * 0100110_2$

Moltiplicare per (-)2 significa shiftare i bit a sinistra di 1 posizione

```
                1 1 1 1 1 0 1 *   = -3₁₀
                      1-2 2-2 =   = 38₁₀ (radix-4)
---------------------------
0 0 0 0 0 0 0 0 0 0 1 1 0 +
1 1 1 1 1 1 1 1 0 1 0     +
0 0 0 0 0 0 1 1 0         +
1 1 1 1 1 0 1             +
---------------------------
1 1 1 1 1 1 0 0 0 1 1 1 0     = -114₁₀
```

Ricordarsi i due spazi in radix-4

Rappresentazione in Virgola Fissa

Spostiamoci ora dai numeri interi dell'insieme Z ai numeri dell'insieme Q

La rappresentazione più immediata è la **virgola fissa**: si dedicano **k** bit per rappresentare la parte intera e **n** bit per rappresentare la parte frazionaria

$$b_{k-1}b_{k-2}\ldots b_0 b_{-1}\ldots b_{-n}$$

$$b_{k-1} * 2^{k-1} + b_{k-2} * 2^{k-2} + \ldots + b_0 * 2^0 + b_{-1} * 2^{-1} + \ldots + b_{-n} * 2^{-n}$$

Rappresentazione in Virgola Fissa

Esempio

Virgola fissa a 3 + 5 bit

$$01101101_2 = 011\mathbf{.}01101_2 = 1 * 2^1 + 1 * 2^0 + 1 * 2^{-2} + 1 * 2^{-3} + 1 * 2^{-5} = 3.40625$$

La posizione della virgola è **fissa** e **implicita**

Rappresentazione in Virgola Mobile

I numeri in **virgola mobile** sono rappresentati da:

- S: Segno
- E: Esponente
- M: Mantissa

con il seguente significato matematico:

$$N = (-1)^S * 2^{(E-127)} * (1 + \boxed{}\ 0.M)$$

127 se E codificato su 8 bit
1023 se E codificato su 11 bit

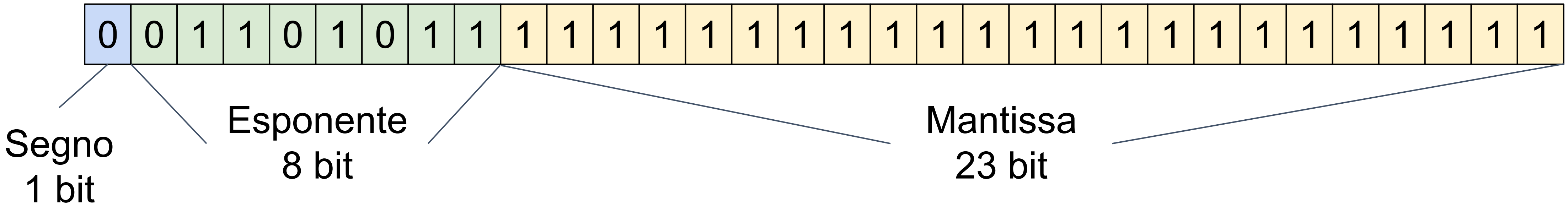
$$2^{-1}m_{n-1} + 2^{-2}m_{n-2} + \ldots$$

n = numero bit mantissa

Rappresentazione in Virgola Mobile

Secondo lo standard **IEEE 754** i floating point sono così rappresentati:

- Singola precisione (32-bit)



- Doppia precisione (64-bit)
    - Segno: 1 bit
    - Esponente: 11 bit
    - Mantissa: 52 bit

Rappresentazione in Virgola Mobile

Esempio di conversione in singola precisione: 314.15

- Convertiamo la parte intera in binario: 100111010_2
- Convertiamo la parte decimale in binario: .00100110011…
- Scriviamo il numero in forma normale:

  100111010.00100110011… diventa 1.00111010001001100…

  abbiamo spostato di 8 posizioni

0	1	0	0	0	0	1	1	1	0	0	1	1	1	0	1	0	0	0	1	0	0	1	1	0	0	1	1	0	0	1	1

Segno 1 bit | Esponente 8 bit | Mantissa 23 bit

Operazione di somma/sottrazione in floating point

Per eseguire la somma (o sottrazione), si procede con lo stesso metodo della somma nella notazione scientifica di un numero:

- Esempio: $1.2 * 10^{13} - 5 * 10^{12}$
  - Non posso sottrarre direttamente i coefficienti ma devo prima **uniformare gli esponenti**: $5 * 10^{12} >> 0.5 * 10^{13}$
  - A questo punto eseguo la **somma**: $(1.2 - 0.5) * 10^{13} = 0.7 * 10^{13}$
  - Il risultato andrà ora **normalizzato** in $7 * 10^{12}$

Operazione di somma/sottrazione in floating point

Applichiamo lo stesso procedimento ai floating point:

$X = -0.005_{10} - 31.210_{10}$

- Convertiamo i due numeri in floating point a 32 bit ottenendo:

-0.005	1	0	1	1	1	0	1	1	1	0	1	0	0	0	1	1	1	1	0	1	0	1	1	1	0	0	0	0	1	0	1	0
-31.21	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	1	1	0	0	0	0	1	0	1	0	0

Operazione di somma/sottrazione in floating point

Bit nascosto: 1

-0.005	1	0	1	1	1	0	1	1	1	0	1	0	0	0	1	1	1	1	0	1	0	1	1	1	0	0	0	0	1	0	1	0

Bit nascosto: 1

-31.21	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	1	1	0	0	0	0	1	0	1	0	0

Gli esponenti sono diversi, uniformiamoli trasformando il più piccolo nel più grande e shiftando adeguatamente la mantissa a dx

0

-0.005	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	1

Differenza esponenti: $1100_2 = 12_{10}$

Il numero così ottenuto è identico all'originale (a meno della perdita di precisione) ma non normalizzato

Operazione di somma/sottrazione in floating point

									0																							
-0.005	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	1
									1																							
-31.21	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	1	1	0	0	0	0	1	0	1	0	0

Possiamo ora procedere a sommare le mantisse

								1																							
1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	0	0	1	0	1	0	0	0	1

- Non necessitiamo di normalizzazione (la mantissa inizia con 1)
- Il segno rimane negativo (somma di due numeri negativi)
- Convertendo il numero ottenuto, troviamo: -31.2149982452_{10}

Errori

Errore assoluto: $E_a = Val_{real} - Val_{actual}$

- Costante in virgola fissa
- Aumenta all'aumentare di Val_{actual}

Errore relativo: $E_r = E_a / Val_{actual}$

- Variabile in virgola fissa
- Costante in virgola mobile (approx)

Approfondimento: "What Every Computer Scientist Should Know About Floating-Point Arithmetic", David Goldberg, ACM Computing Surveys, Vol. 23, No. 1, 1991. https://doi.org/10.1145/103162.103163

Operazioni in virgola mobile

I circuiti che eseguono le operazioni in virgola mobile sono complessi

- IEEE 754 definisce numerose operazioni possibili con i floating point
- Vedremo solo il sommatore ad alto livello

Eccezioni

IEEE 754 definisce una serie di **eccezioni** che il sistema dovrebbe essere in grado di riconoscere e informare dell’accaduto il software:

- Valore inesatto (rispetto al corrispondente matematico) *
- Divisione per zero
- Operazione non valida (es. sqrt(-1))
- Overflow
- Underflow

* Poco usato, la maggior parte delle operazioni generano valori imprecisi

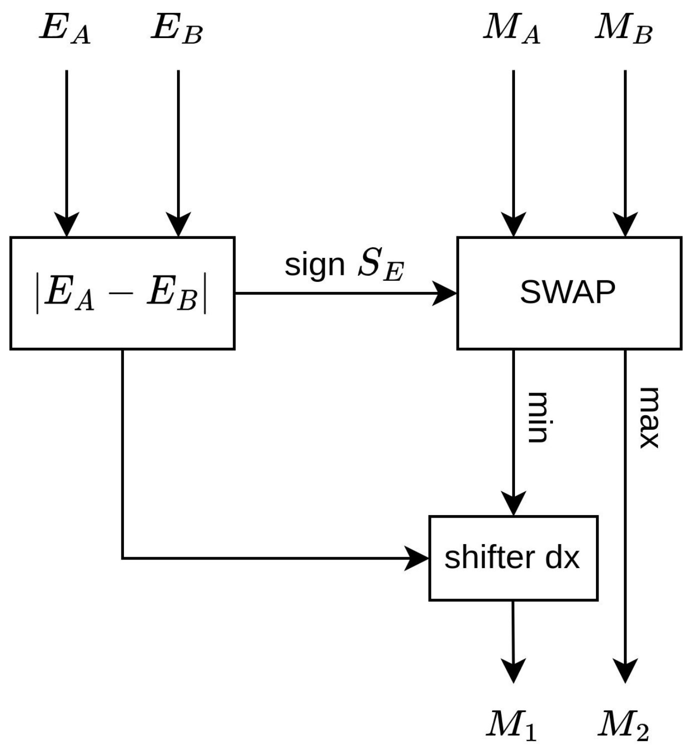
Sommatore in virgola mobile

Dati due numeri A e B codificati in floating point singola precisione (32-bit) secondo IEEE 754:

- S_A: segno A S_B: segno B
- E_A: esponente A E_B: esponente B
- M_A: mantissa A M_B: mantissa B

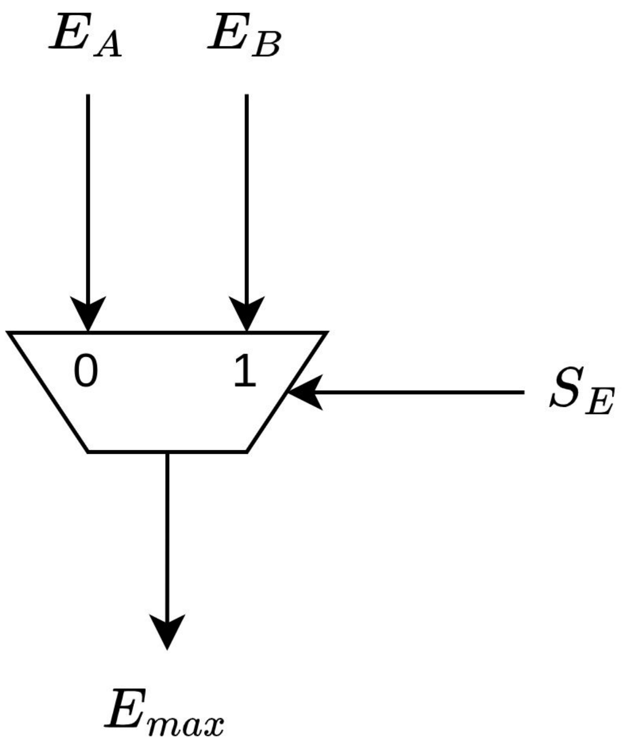
Sommatore in virgola mobile

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



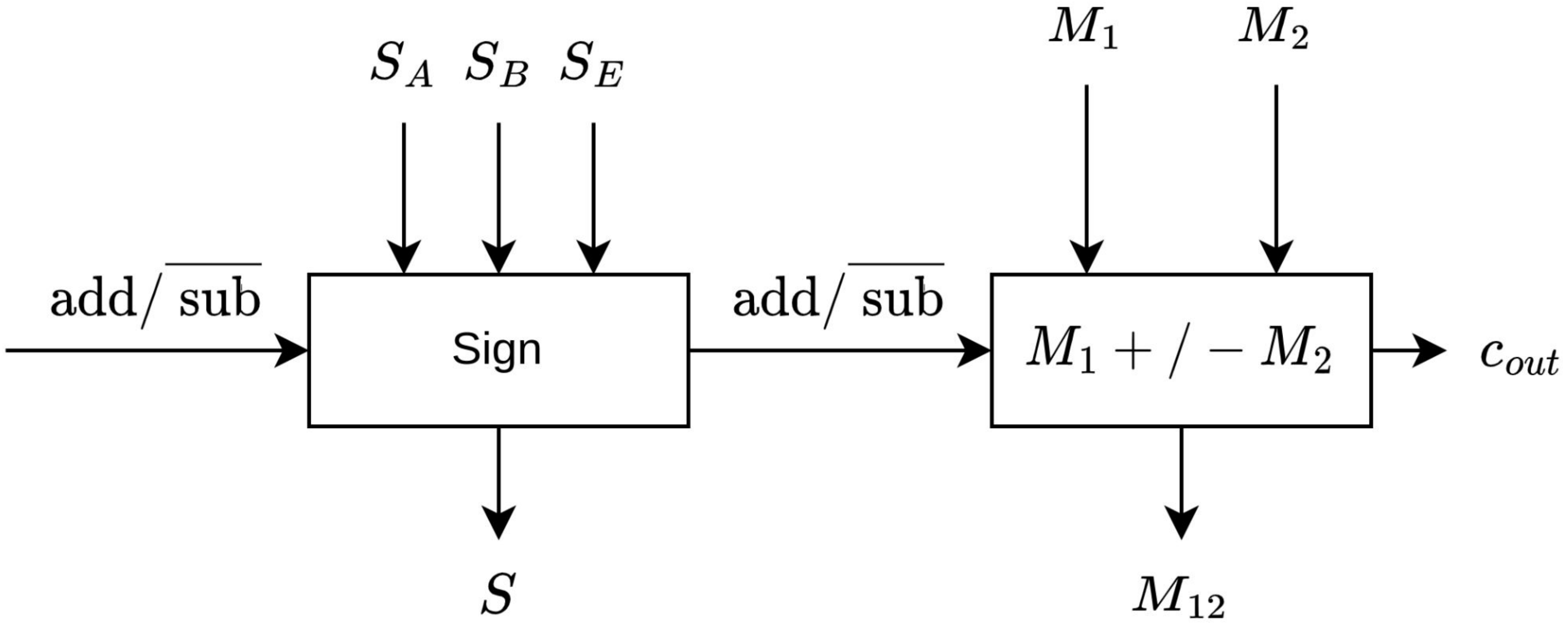
Sommatore in virgola mobile

Passo 2: Si assegna all’esponente del risultato il maggiore tra gli esponenti degli operandi

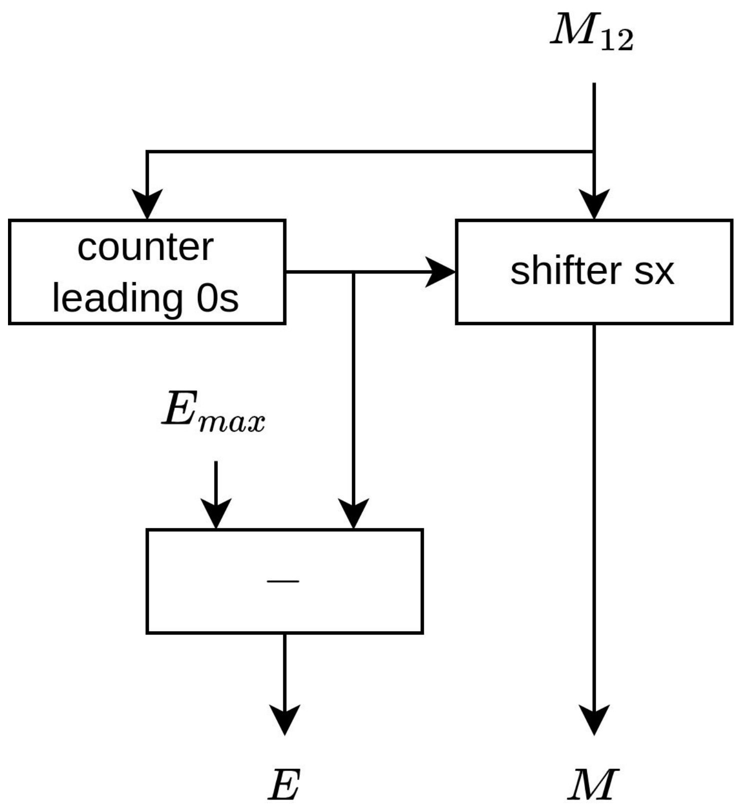


Sommatore in virgola mobile

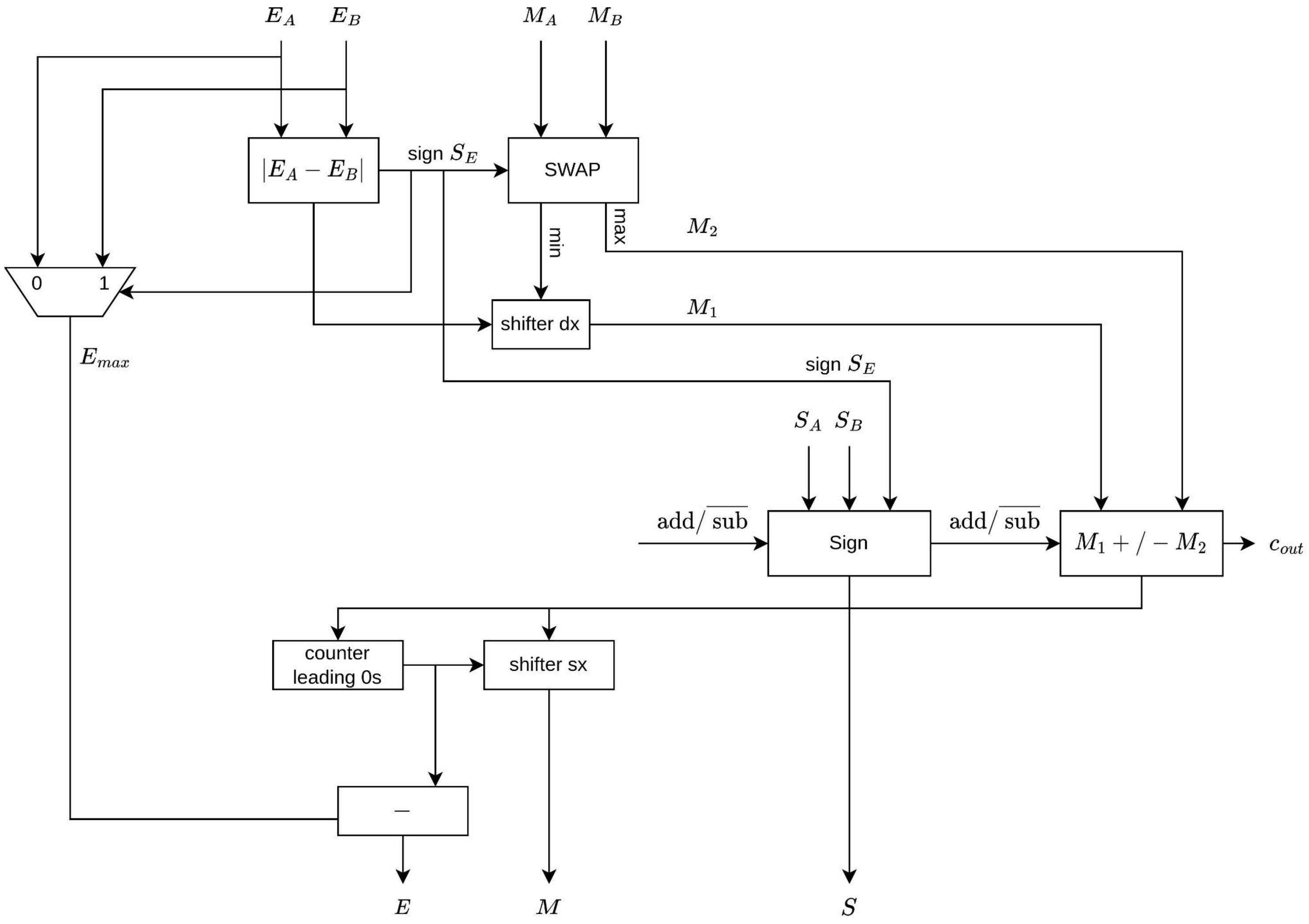
Passo 3: Si esegue l'operazione di somma tra le mantisse per determinare il valore ed il segno del risultato



Passo 4: Normalizzazione



Sommatore completo



Fine

Domande?