



Macchine aritmetiche

prof. Antonino Mazzeo

ing. Alessandra De Benedictis

Testi di riferimento

Franco Fummi, Mariagiovanna Sami, Cristina Silvano - Progettazione digitale - McGraw-Hill
Bolchini, Brandolese, Salice, Sciuto – Reti Logiche – Apogeo

Progettare sistemi embedded

Il progetto di un sistema embedded comporta una serie di scelte che riguardano:

- La selezione della tecnica di rappresentazione dei dati
- La selezione o il progetto di algoritmi per l'elaborazione dei dati
- La selezione delle piattaforme hardware da utilizzare
- Il partizionamento HW-SW

Molte delle attività coinvolte hanno a che fare con lo studio di algoritmi e circuiti aritmetici, specie in presenza dei sistemi che prevedono una grossa quantità di data processing (cifatura, processamento di immagini, firma digitale, biometria)

Cosa realizzare in HW e cosa in SW?

- Tipologie di piattaforme hardware:
 - Processore general purpose (microprocessori, microcontrollori)
 - Processore dedicato (microprocessori evoluti, DSP)
 - Hardware speciale (FPGA, ASIC)
- La scelta della piattaforma HW e il partizionamento HW/SW dipendono da:
 - Dimensione del sistema
 - Prestazioni
 - Costo
 - Consumo energetico
 - Affidabilità..
- Tipicamente si usa un approccio software in presenza di requisiti non stringenti sulle prestazioni, mentre si usano appositi *coprocessori hardware* per le operazioni critiche

Approcci possibili

- Approccio hardware
 - Circuiti distinti per ciascuna operazione aritmetica con differenti architetture
- Approccio firmware
 - Circuiti specifici per fare operazioni semplici
 - Operazioni complesse sintetizzate a partire dall'algoritmo di implementazione, realizzando una unità di controllo che attiva le unità aritmetiche, i registri e i percorsi tra essi.
- Approccio software
 - Soluzione analoga al firmware ma meno efficiente che fa uso di una macchina virtuale più complessa

In taluni casi si ricorre a memorie ROM

- Per operazioni semplici (ad un solo operando) si usano tabelle memorizzate in ROM
- Esempio A^2

Addr	Val
------	-----

0	0
---	---

1	1
---	---

2	4
---	---

3	9
---	---

4	16
---	----

...	...
-----	-----

Presentazione dati ingresso-uscita

- **Presentazione in parallelo (a)**
 - I bit degli operandi sono presentati in ingresso e i risultati sono calcolati *contemporaneamente* (a meno dei ritardi di propagazione)
- **Presentazione seriale pura (b)**
 - Alle linee di ingresso sono presentati *sequenzialmente* nel tempo i bit degli operandi, e analogamente i bit del risultato appaiono in uscita sequenzialmente.
 - Little-endian (primo bit il meno significativo)
 - Big-endian (primo bit il più significativo) ha senso per la divisione ma non per somma e moltiplicazione
- **Presentazione seriale a gruppi di bit**
 - Una voce è divisa in gruppi di bit: i gruppi omologhi sono presentati in parallelo mentre gruppi della stessa voce sono applicati in serie.
- **Presentazione mista (c)**
 - Uno dei due operandi è presentato in parallelo, mentre l'altro con modalità seriale

Tipo di presentazione Vs unità aritmetica

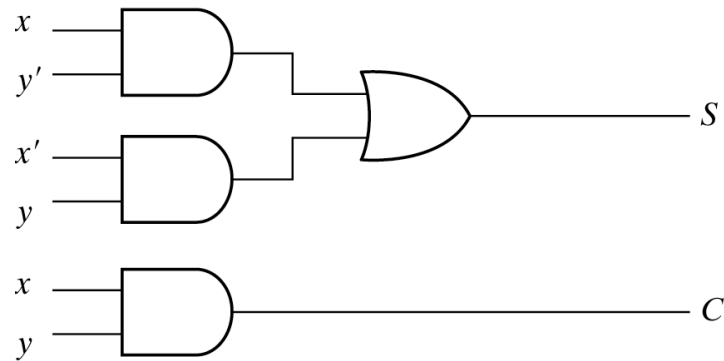
Il tipo di presentazione dei dati influisce su:

- natura dell'unità aritmetica:
 - Rete combinatoria per presentazione parallela
 - Rete sequenziale negli altri casi
- complessità
 - Seriale è più piccola e economica di una parallela
- velocità
 - Seriale richiede molti impulsi di clock (rete sequenziale)
 - Parallela più veloce (solo ritardi rete combinatoria)

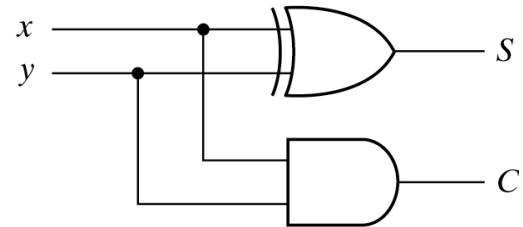
Operazioni aritmetiche su numeri interi

SOMMA

HALF ADDER



(a) $S = xy' + x'y$
 $C = xy$



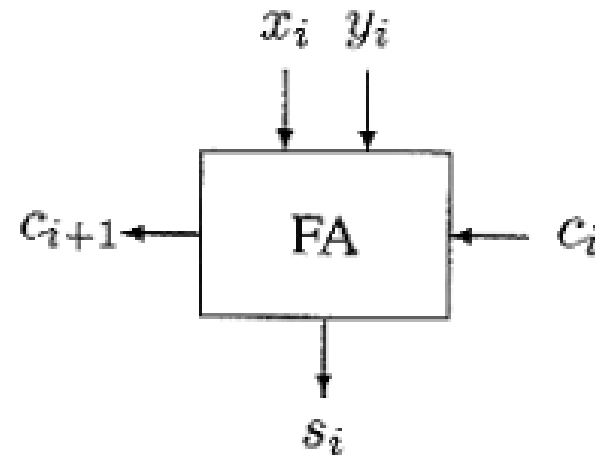
(b) $S = x \oplus y$
 $C = xy$

Fig. 4-5 Implementation of Half-Adder

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

FULL ADDER

x_i	y_i	c_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



$x \backslash y c$	00	01	11	10
0			1	
1		1	1	1

$$C = xy + xc + yc = xy + c(x+y)$$

si può scrivere come:

$$C = xy + c(x \oplus y)$$

$x \backslash y c$	00	01	11	10
0		1		1
1	1		1	

$$S = x'y'c + x'yc' + xy'c' + xyc = x \oplus y \oplus z$$

FULL ADDER: realizzazione mediante half adder

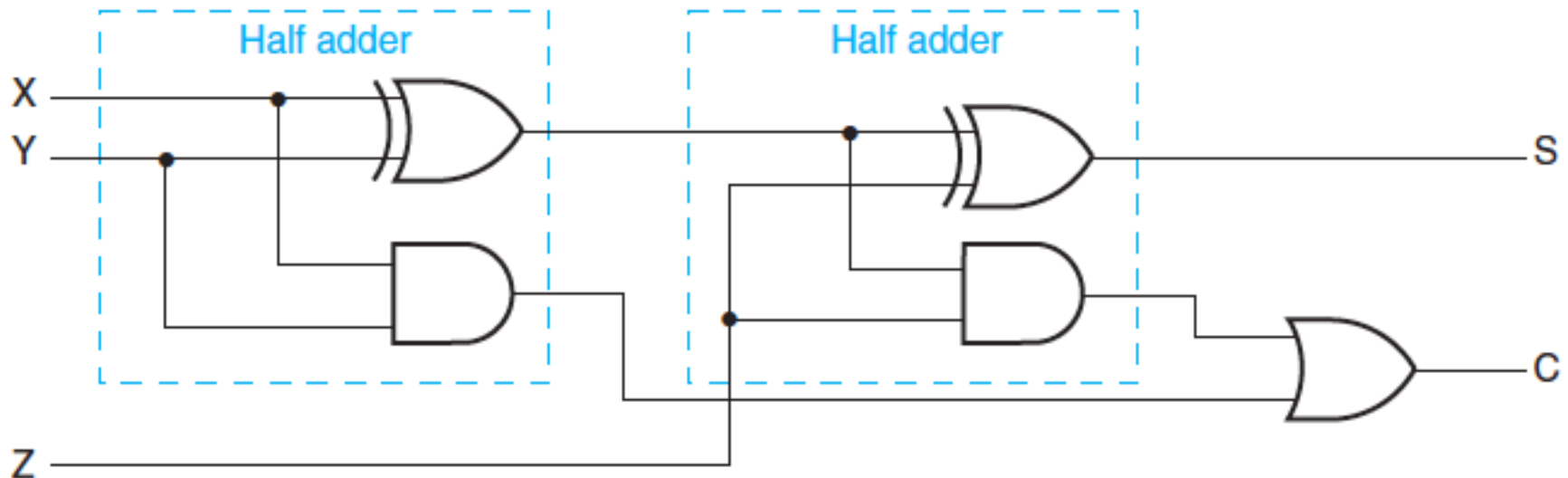
Full adder realizzato mediante due half adder ed una porta or per la composizione del riporto uscente secondo le espressioni precedentemente mostrate

$$C_H = xy$$

$$S_H = x \oplus y$$

$$C_F = xy + z(x \oplus y)$$

$$S_F = x \oplus y \oplus z$$

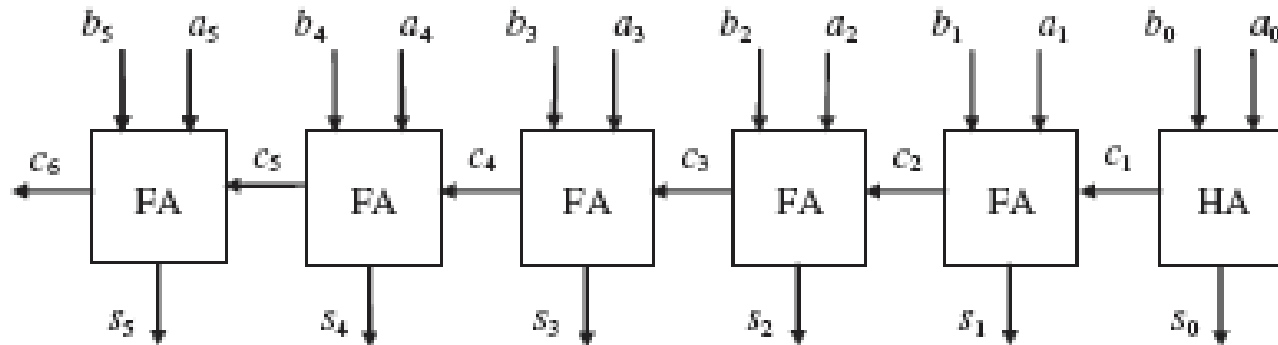


Sommatori a propagazione di riporto

Somma di interi positivi: procedura manuale

riporto	0	1	1	1	1	
1° addendo	0	0	0	1	0	1
2° addendo	0	0	1	0	1	1
risultato	0	1	0	0	0	0

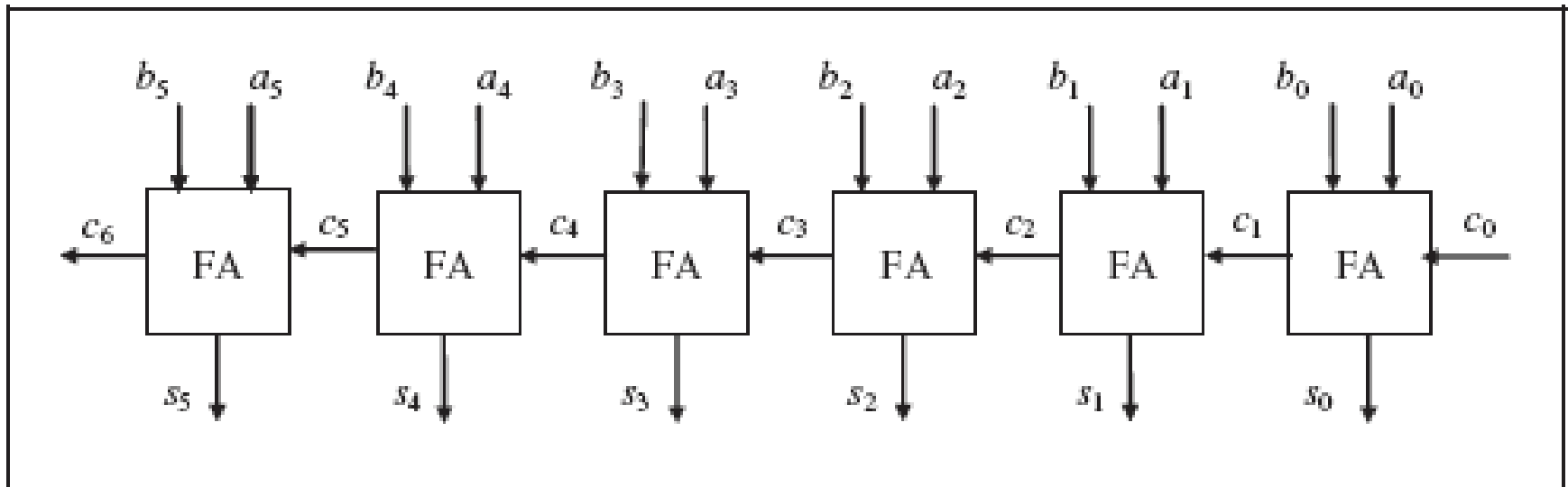
Sommatore a propagazione riporti o *ripple carry adder*



La struttura del sommatore a propagazione dei riporti deriva dall'algoritmo manuale per la somma di due numeri interi positivi di n bit ciascuno:

- la cella a destra riceve i bit meno significativi dei due addendi e produce la somma s_0 ed il riporto c_1 in entrata alla seconda cella;
- la seconda cella da destra riceve i bit a_1 e b_1 dei due addendi ed il riporto c_1 dallo stadio precedente e produce la somma s_1 ed il riporto c_2 allo stadio successivo, e così via.

Ripple carry adder costituito da n Full Adder



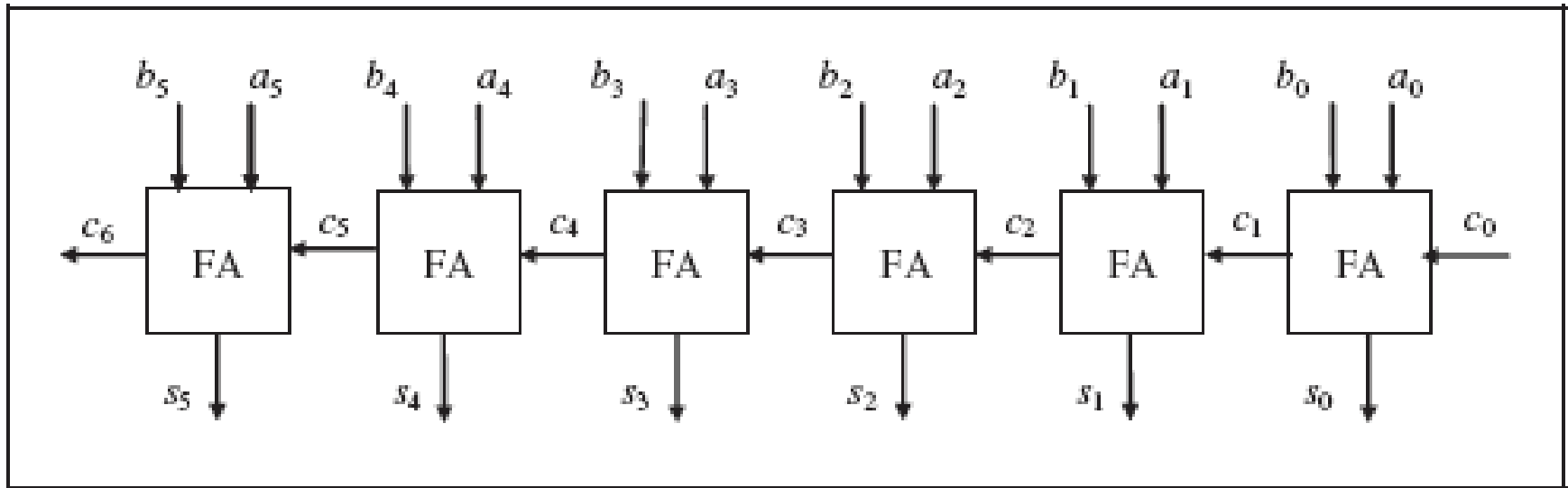
Ponendo $c_0=0$ è possibile ottenere una struttura più regolare costituita da n dispositivi full adder identici connessi in serie.

Il circuito è in grado di eseguire somme algebriche tra numeri rappresentati in complemento a 2 (l'eventuale riporto nella colonna a sinistra di quella più significativa deve essere ignorato)

Sommatore a propagazione di riporti o *ripple carry adder*

- Circuito estremamente regolare
 - Costituito da celle tutte identiche fra loro e interconnesse mediante uno schema che si ripete anch'esso identicamente
 - Rete iterativa monodimensionale
 - Flusso di informazione monodirezionale con celle combinatorie

Ripple carry adder : ritardo e area



$$C = xy + z(x \oplus y)$$
$$S = x \oplus y \oplus z$$

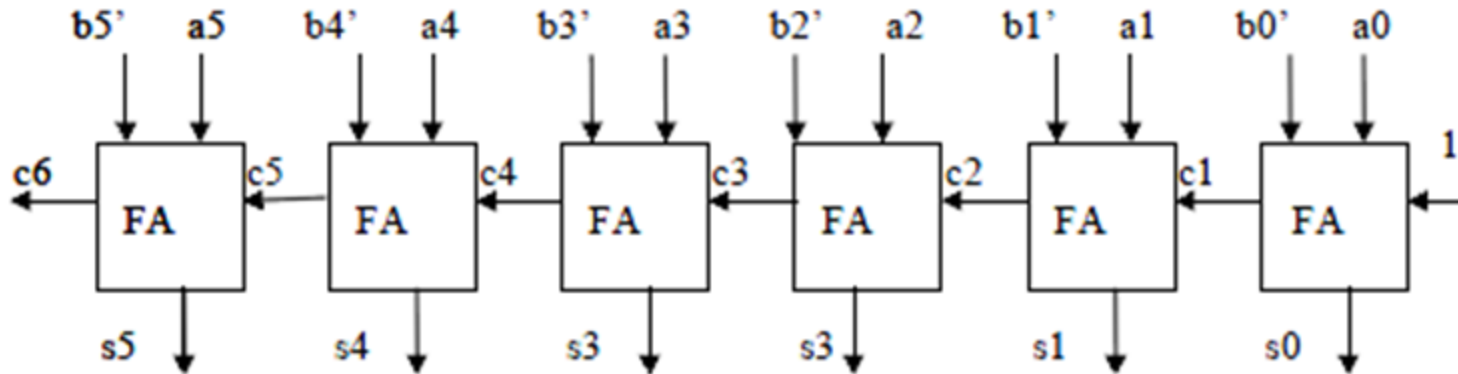
- ❑ il ritardo è pari a $2\Delta n$ per la somma di operandi da n bit, se Δ è il ritardo di porta
- ❑ l'area occupata è di $5n$ porte logiche

Sottrattore su n bit basato su *ripple carry adder*

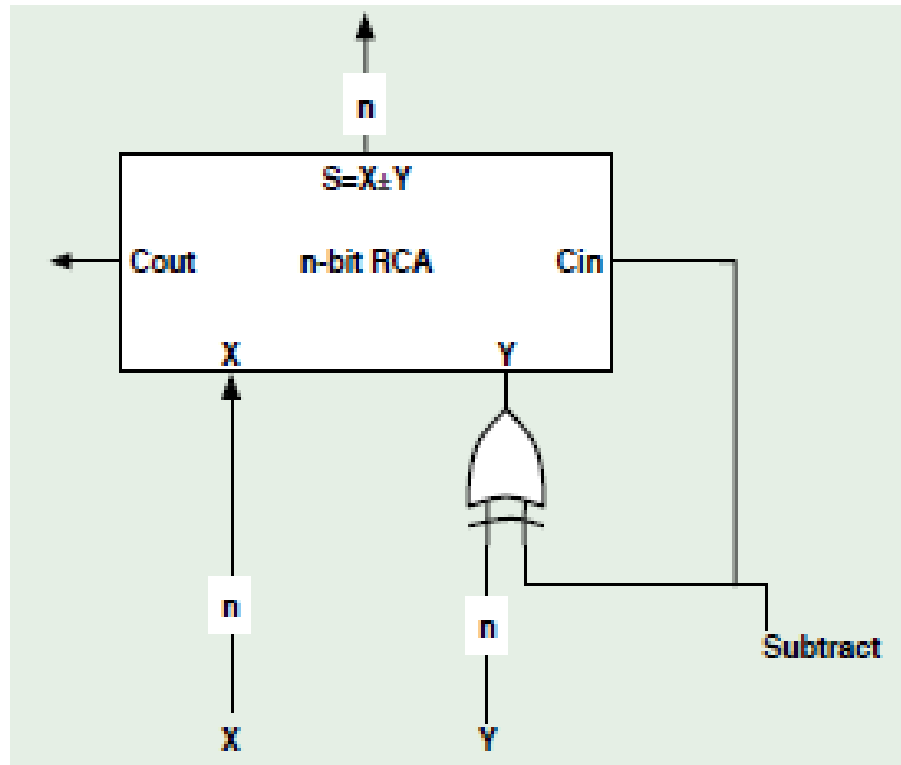
❑ Effettuare l'operazione di sottrazione **A-B** con A e B entrambi addendi su n bit, equivale ad effettuare l'operazione di addizione fra A e il complemento a 2 di B: **A+(-B)**

❑ Il complemento a 2 di B viene ottenuto aggiungendo 1 al complemento diminuito di B:

$$-B = b'_{n-1}b'_{n-2}...b'_0 + 1$$



Circuito adder/subtractor su n bit



$$X = X \text{ xor } 0$$
$$\text{not } X = X \text{ xor } 1$$

$$\begin{aligned} Y - X &= Y + (-X) = \\ &= Y + (\text{not } X + 1) = \\ &= Y + \text{not } X + 1 \end{aligned}$$

Quando il segnale subtract vale 1 l'addizionatore prende in ingresso Y e X negato (dato da $X \text{ xor } 1$); aggiungendo 1 alla somma finale (carry in = 1) si ottiene di fatto l'operazione $Y - X$

Sommatori con valutazione
parallela dei riporti

Limiti dei sommatore RCA

❑ Il sommatore ripple carry presenta prestazioni scarse a causa della propagazione dei riporti (il ritardo è pari a $2\Delta n$ per la somma di operandi da n bit, se Δ è il ritardo di porta)

❑ Nasce la necessità di sviluppare nuove architetture che presentino tempi di propagazione inferiori:

- ✓ Potendo disporre in anticipo di tutti i riporti la somma sui vari bit potrebbe essere calcolata in parallelo

- ✓ Si introducono due funzioni ausiliarie P_i e G_i , dette di propagazione e di generazione rispettivamente, e definite come:

$$P_i = x_i + y_i$$

$$G_i = x_i y_i$$

Condizioni di generazione e propagazione dei riporti

$P_i = x_i + y_i$ *condizione di propagazione*

$G_i = x_i y_i$ *condizione di generazione*

Il riporto in uscita dallo stadio i-esimo assume la forma:

$c_{i+1} = x_i y_i + (x_i + y_i) c_i = G_i + P_i c_i$ dove c_i può essere scritto come:

$c_i = x_{i-1} y_{i-1} + (x_{i-1} + y_{i-1}) c_{i-1} = G_{i-1} + P_{i-1} c_{i-1}$ da cui:

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

Ripetendo lo stesso procedimento è possibile esprimere ciascun riporto in funzione di c_0

$$c_1 = G_0 + P_0 c_0$$

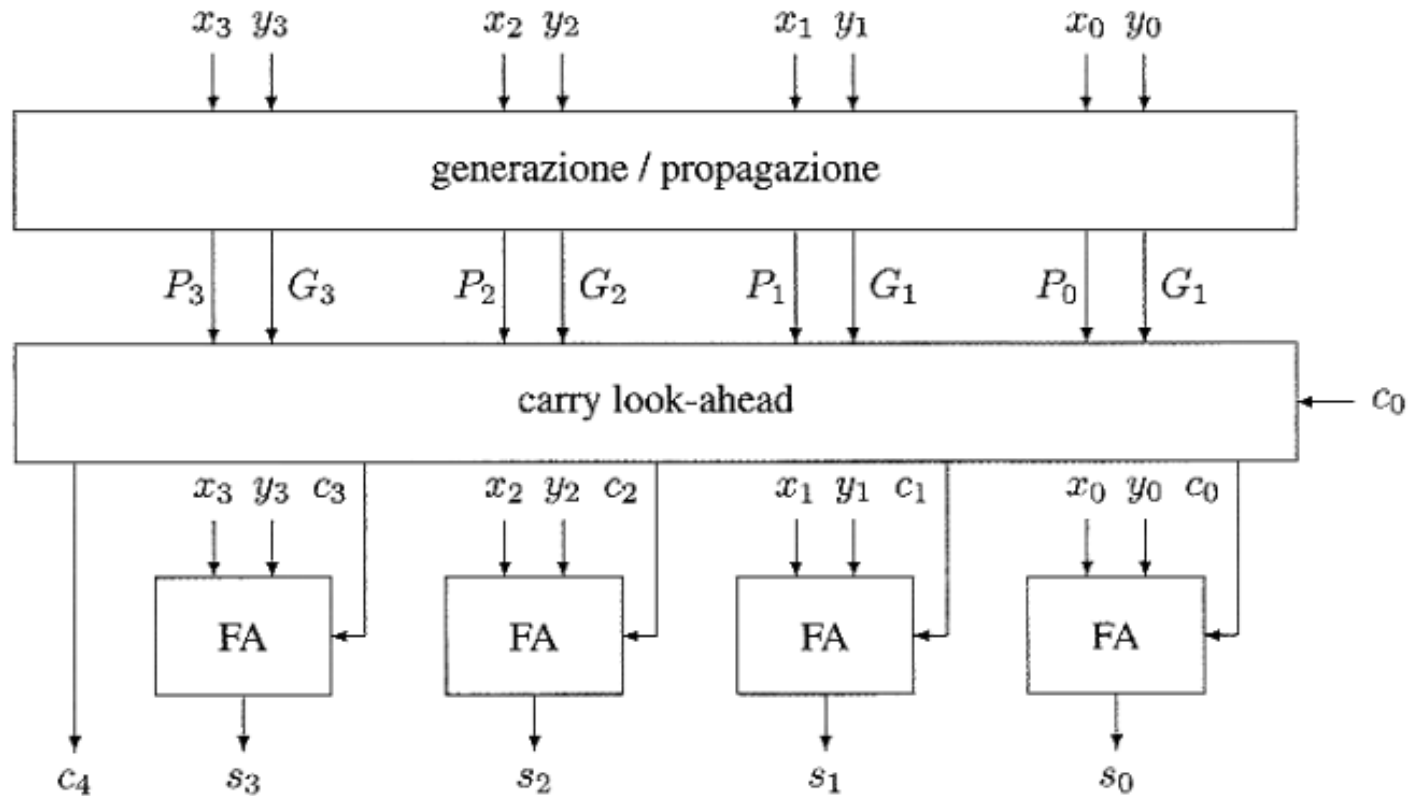
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

.....

Sommatore *carry look-ahead*



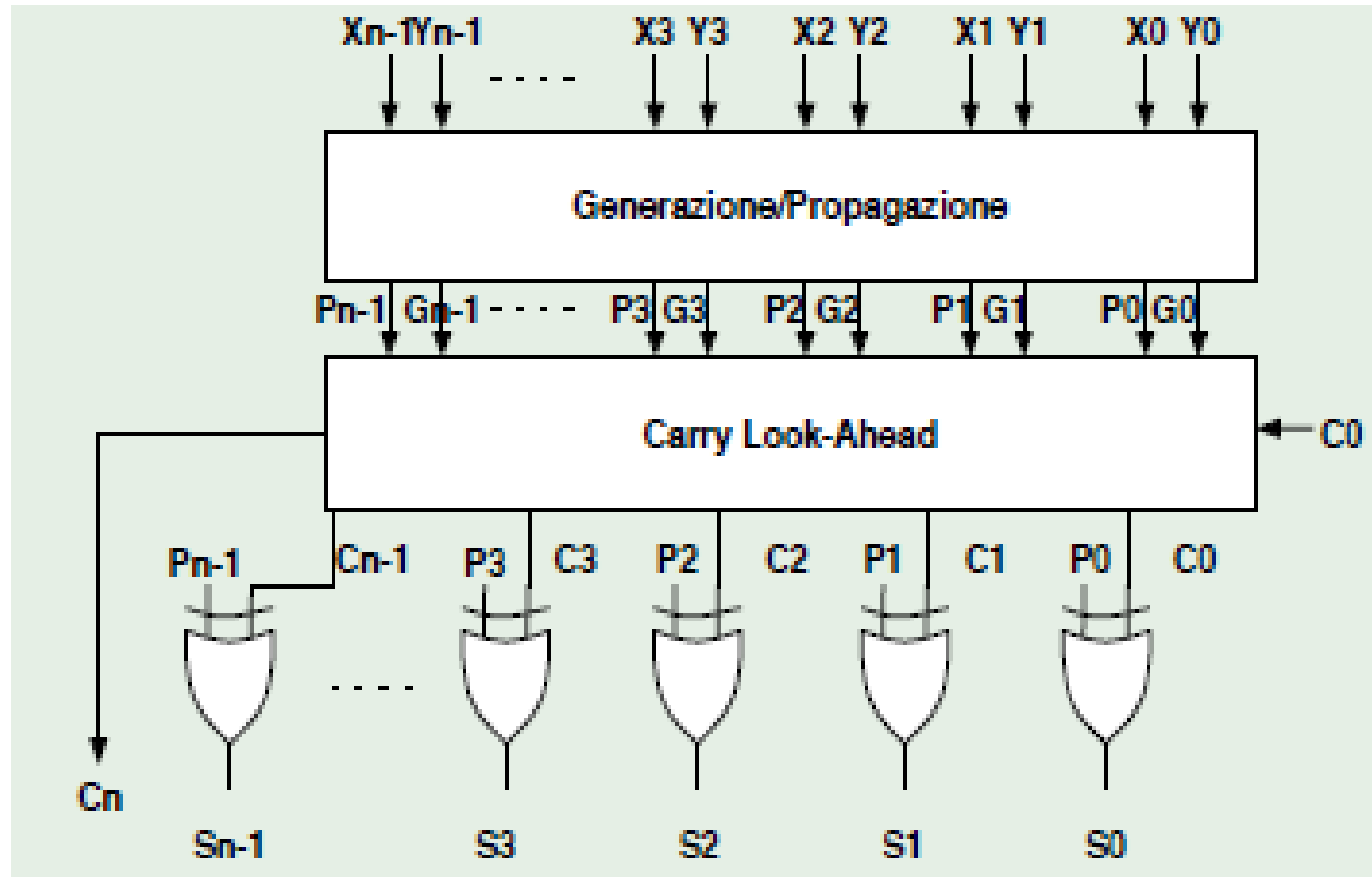
Il ritardo è pari a **5 Δ** poiché:

- ❑ Il calcolo di P e G avviene contemporaneamente ed impiega Δ ;
- ❑ Il calcolo dei riporti impiega 2Δ poiché tutti i riporti sono espressioni SoP;
- ❑ Il FA ha un ritardo di 2Δ ;

Si può dimostrare che l'area occupata è pari a $(n^2 + 9 \cdot n)/2$

Sommatore *carry look-ahead*: *osservazioni*

Ricordiamo che: $S = x \oplus y \oplus C_{in} = P \oplus C_{in}$



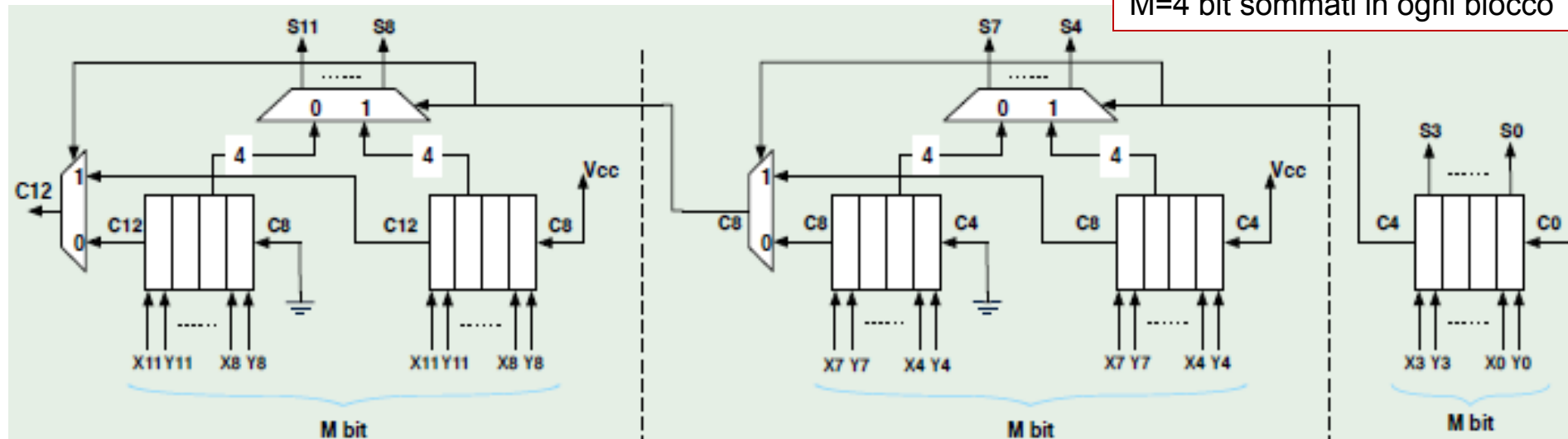
Sommatori veloci

Sommatori carry-Select

Il problema dell'architettura RCA è che la carry chain varia linearmente con il numero di bit. Si può pensare di suddividere tale catena in catene più corte.

- ❑ Supponiamo di dividere un normale RCA in P blocchi, ciascuno dei quali somma M bit:
 - il primo blocco è un RCA di M bit che restituisce in uscita la somma su M bit ed il riporto in uscita (overflow);
 - i blocchi successivi contengono 2 RCA: entrambi effettuano la somma degli stessi M bit ma uno ha $C_{in} = 0$, l'altro $C_{in} = 1$.

$P=3$ blocchi
 $M=4$ bit sommati in ogni blocco

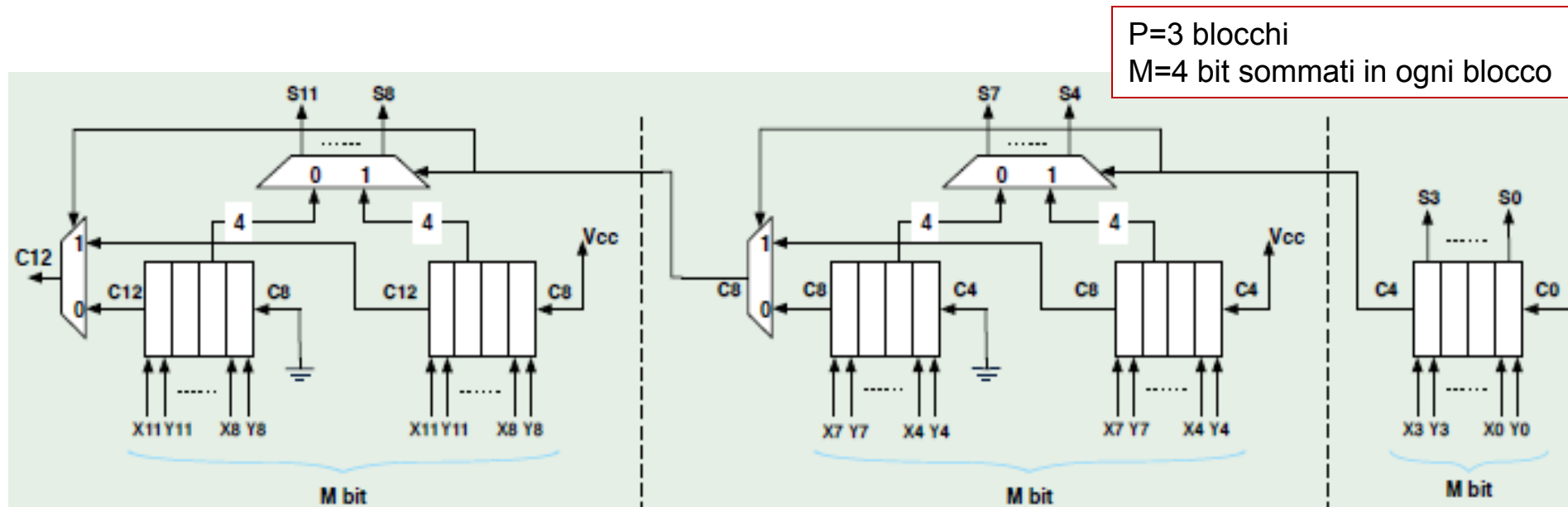


Sommatori carry-Select

Definiamo TFA e TMUX i tempi di propagazione rispettivamente del Full Adder e del Multiplexer;

- ❑ Il CM, ovvero il carry in uscita dal primo blocco, è disponibile al tempo $M \cdot TFA$;
- ❑ Le somme ed il carry all'uscita del secondo blocco saranno disponibili al tempo $M \cdot TFA + TMUX$;
- ❑ Le somme ed il carry finali saranno disponibili al tempo

$$T = M \cdot TFA + (P-1) \cdot TMUX.$$



Sommatori carry-Select

$$T = M \cdot T_{FA} + (P-1) \cdot T_{MUX}.$$

- ❑ Nota la legge che regola il tempo di propagazione dell'architettura è possibile ottimizzarla:

- ***noti T_{FA} e T_{MUX} , quali sono P ed M che minimizzano il tempo?***

- ❑ Il problema di minimo è però ad una sola variabile, poiché $M = N/P$

$$T = N/P \cdot T_{FA} + (P-1) \cdot T_{MUX}$$

- $\frac{dT_p}{dP} = -\frac{n \cdot T_{FA}}{P^2} + T_{MUX} = 0;$

- $\hat{P} = \sqrt{\frac{N \cdot T_{FA}}{T_{MUX}}};$

- $\hat{M} = \frac{N}{\hat{P}} = \sqrt{\frac{n \cdot T_{MUX}}{T_{FA}}};$

- $\hat{T}_p = \hat{M} \cdot T_{FA} + (\hat{P} - 1) \cdot T_{MUX} = 2 \cdot \sqrt{N \cdot T_{FA} \cdot T_{MUX}} - T_{MUX}.$

- ❑ Il miglioramento rispetto al ripple carry è sensibile quando N è elevato
- ❑ L'occupazione d'area è circa il doppio

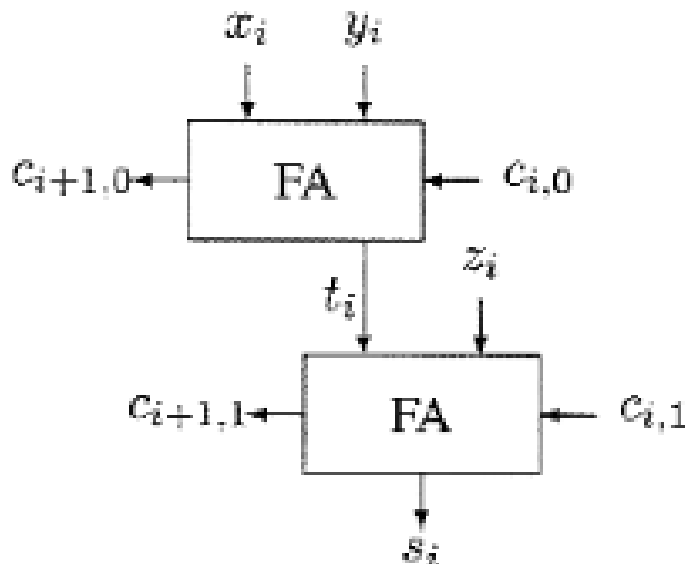
Somma di più operandi:
sommatori carry save

Somma di operandi multipli

□ Si consideri la somma $S = X + Y + Z$

Utilizzando due sommatori in cascata si può realizzare tale operazione come $S = (X + Y) + Z = T + Z$

Considerando un'architettura composta da due sommatori ripple-carry la somma relativa al bit i -esimo può essere effettuata con il seguente schema:



Secondo questo schema si ricava che

$$t_i = x_i + y_i + c_{i,0}$$

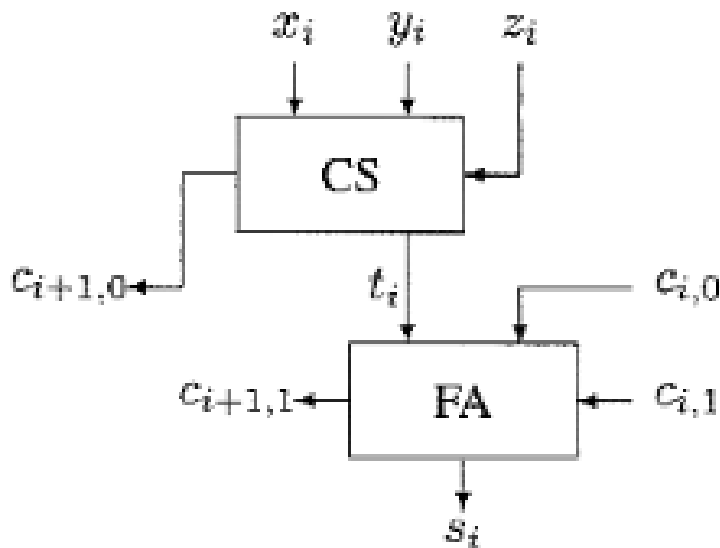
$$s_i = t_i + z_i + c_{i,1}$$

da cui:

$$s_i = (x_i + y_i + z_i) + c_{i,0} + c_{i,1}$$

Sommatori *carry save*

$$s_i = (x_i + y_i + z_i) + c_{i,0} + c_{i,1}$$

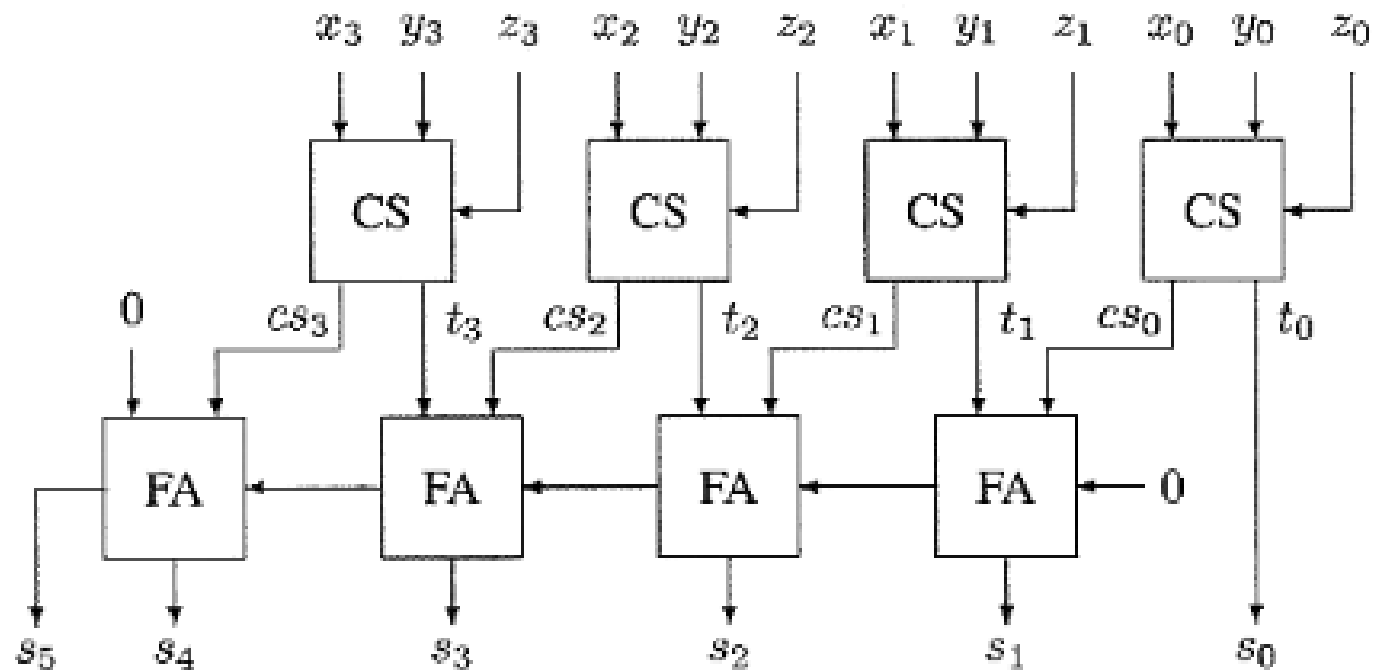


❑ La somma del bit i -esimo dei 3 operandi può essere effettuata da un full adder che realizza la logica di salvataggio del riporto (indicato con il simbolo CS)

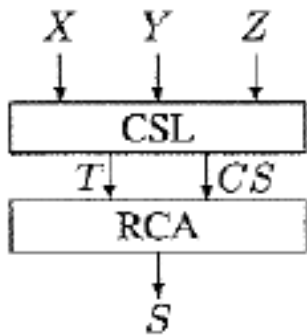
❑ Nel full adder a valle entrano i riporti generati nel livello superiore dalla somma dei bit $(i-1)$ -esimi

Utilizzando il modulo di base mostrato si possono costruire sommatore carry save con operandi di dimensione arbitraria: i blocchi CS operano in parallelo in quanto non sono soggetti alla propagazione del riporto mentre i blocchi FA sono connessi a formare un sommatore ripple-carry

Sommatori *carry save*

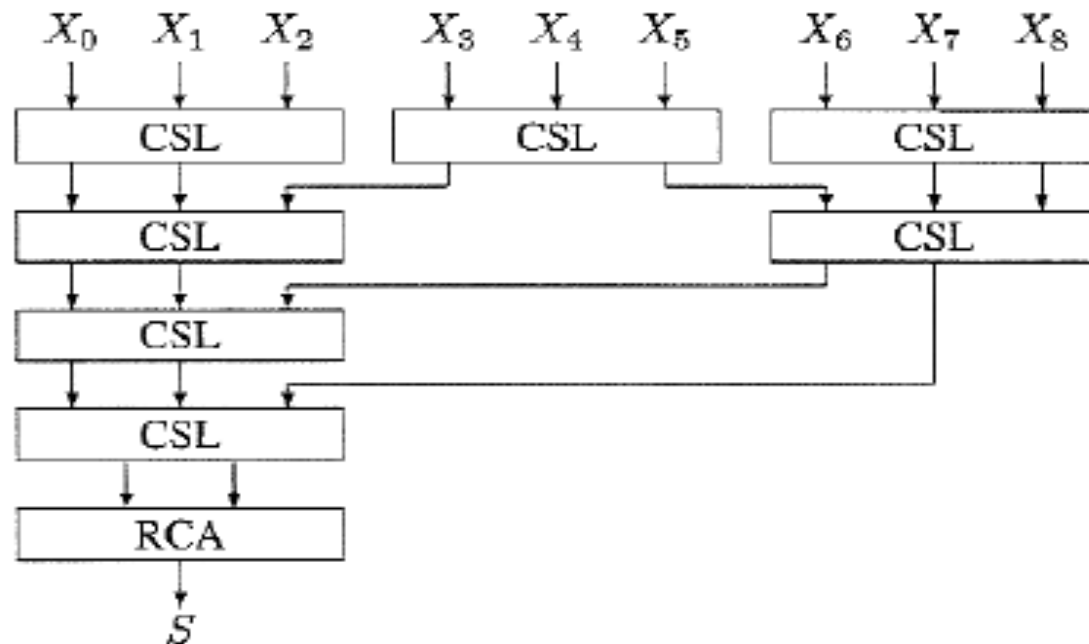


Sommatori *carry save*



Un sommatore carry-save è costituito da un blocco CSL o *carry-save logic* e da un blocco *ripple carry adder*.

Interconnettendo opportunamente blocchi carry-save è possibile realizzare la somma di un numero elevato di operandi riducendo il ritardo complessivo



Moltiplicatori

Parallelismo operandi

- $Z = X \cdot Y$
- X codificato su n bit
- Y codificato su m bit
- Z codificato su $n+m$ bit
 - Se $n=m$, Z è espresso su $2n$ bit

Tipologia di soluzioni

- Moltiplicatori Paralleli
 - prodotto cifre + somma righe
 - multiply and accumulate
- Seriali
 - derivati da procedura manuale (Robertson)
 - basati su codifiche alternative (Booth)

Moltiplicatori paralleli

La procedura manuale di moltiplicazione prevede:

- ❑ **Prima fase:** determinazione della matrice dei prodotti parziali;
 - Viene effettuata semplicemente utilizzando opportune porte AND
- ❑ **Seconda fase:** somma dei prodotti parziali.
 - Può essere effettuata utilizzando varie tecniche, con l'obiettivo di aumentare l'efficienza (velocità) del circuito

Esempio: moltiplicatore binario a 2 bit

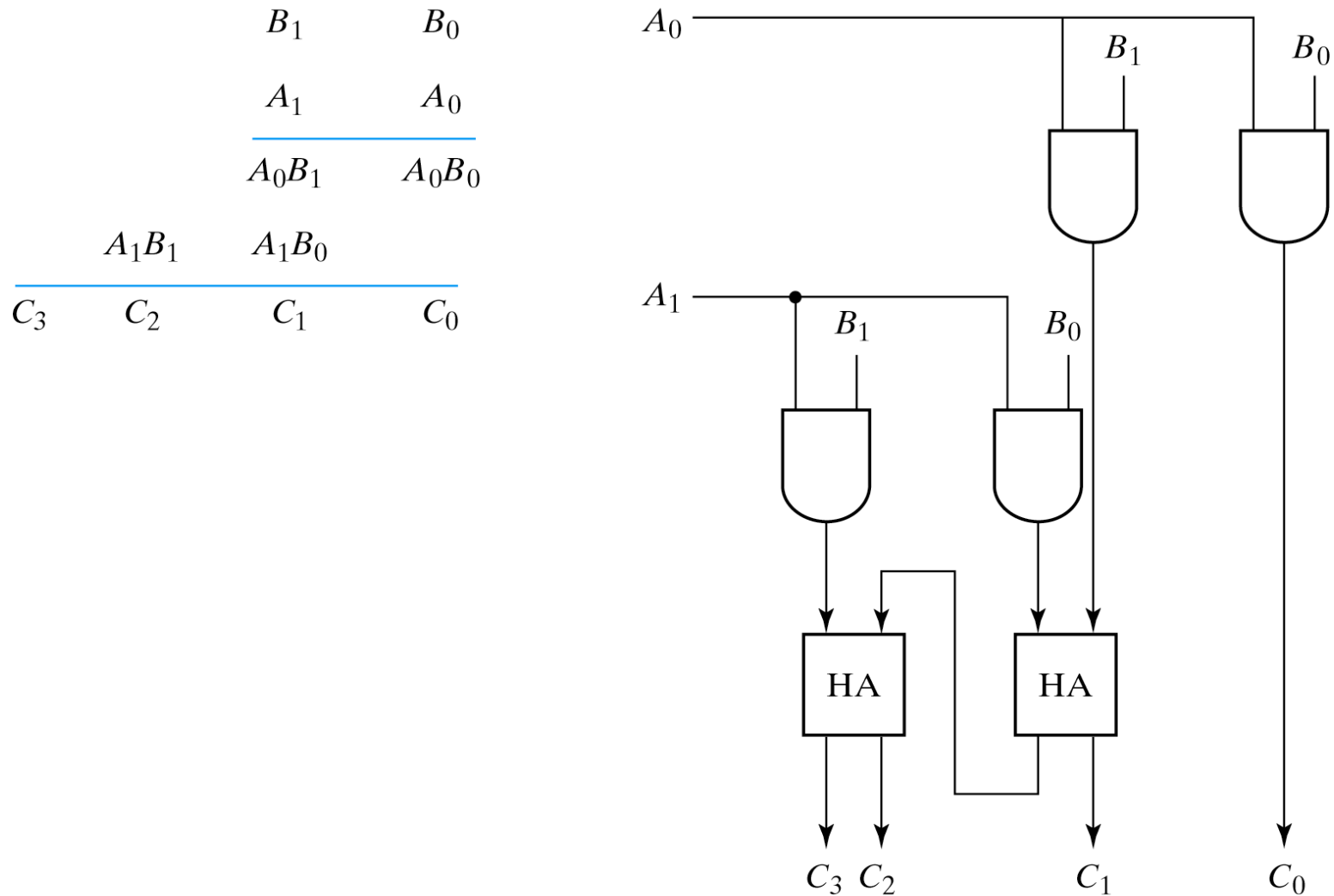
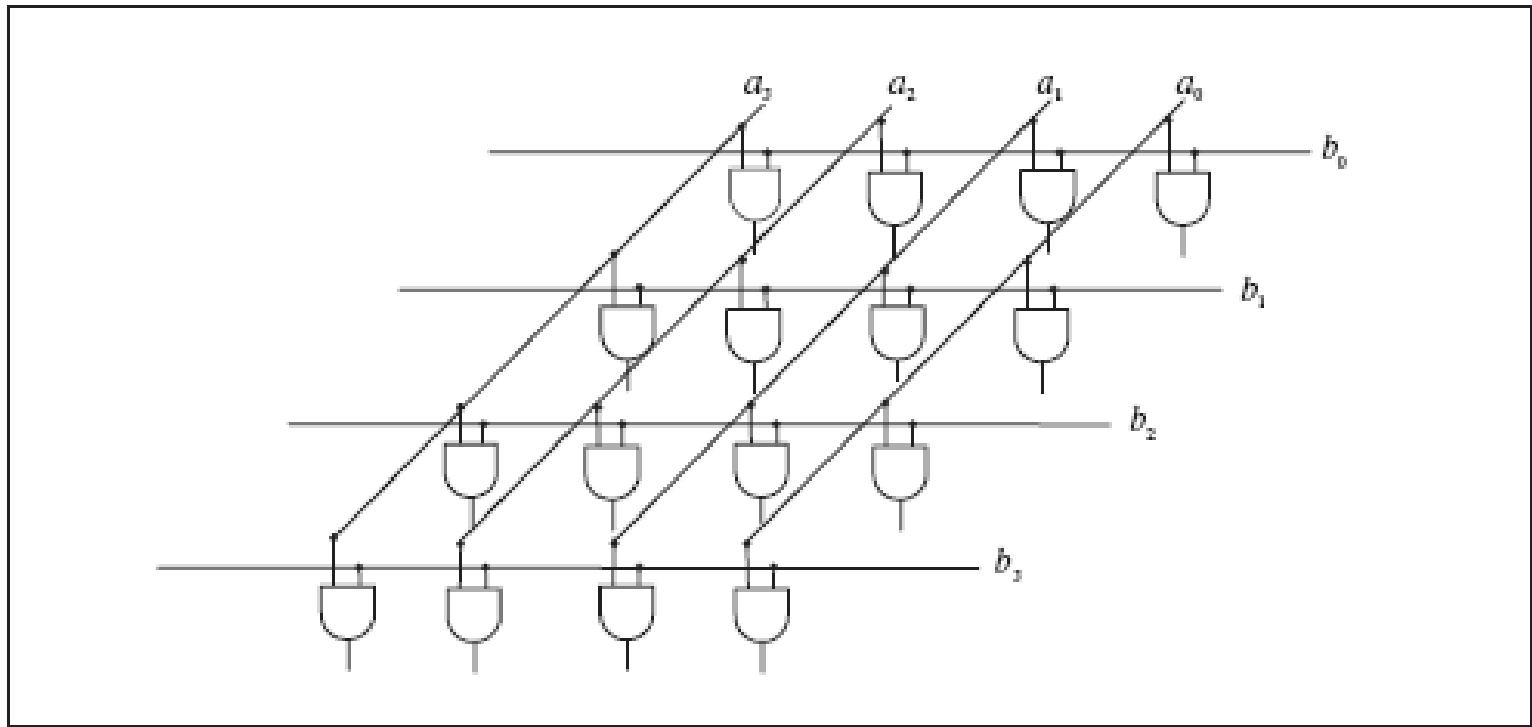


Fig. 4-15 2-Bit by 2-Bit Binary Multiplier

Matrice dei prodotti parziali:
esempio di prodotto fra due fattori
di 4 bit ognuno

			a_3	a_2	a_1	a_0	
			$a_3 b_0$	$a_2 b_0$	$a_1 b_0$	$a_0 b_0$	b_0
		$a_3 b_1$	$a_2 b_1$	$a_1 b_1$	$a_0 b_1$		b_1
	$a_3 b_2$	$a_2 b_2$	$a_1 b_2$	$a_0 b_2$			b_2
$a_3 b_3$	$a_2 b_3$	$a_1 b_3$	$a_0 b_3$				b_3

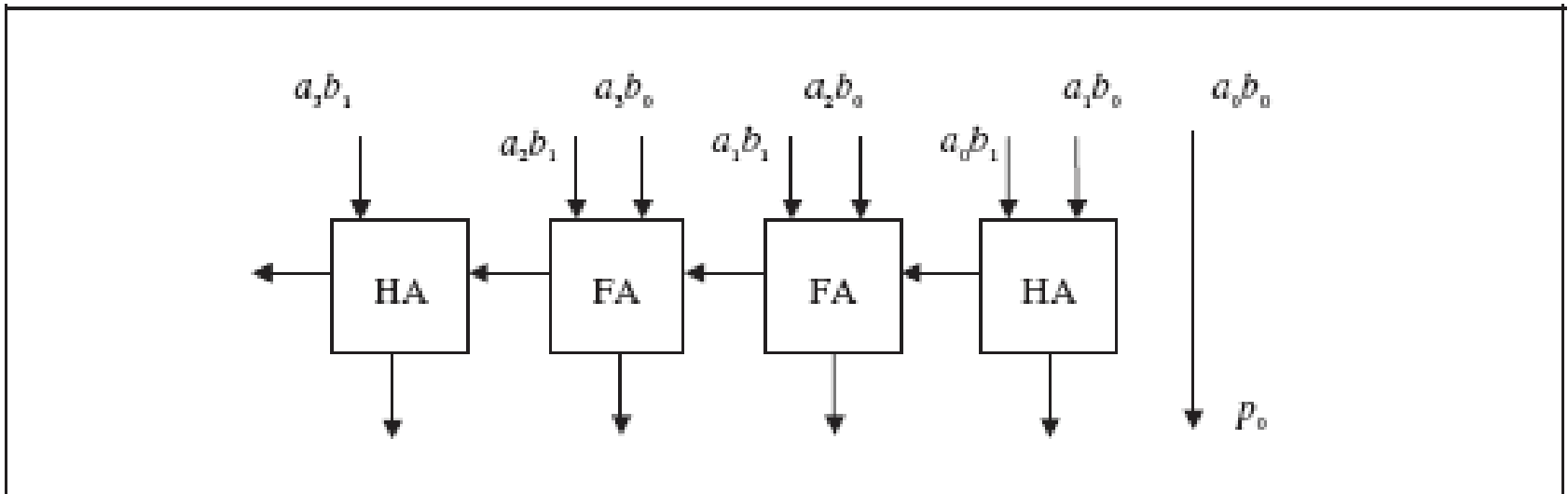
Matrice dei prodotti iniziale: realizzazione mediante porte logiche elementari



Prodotto come somma di righe

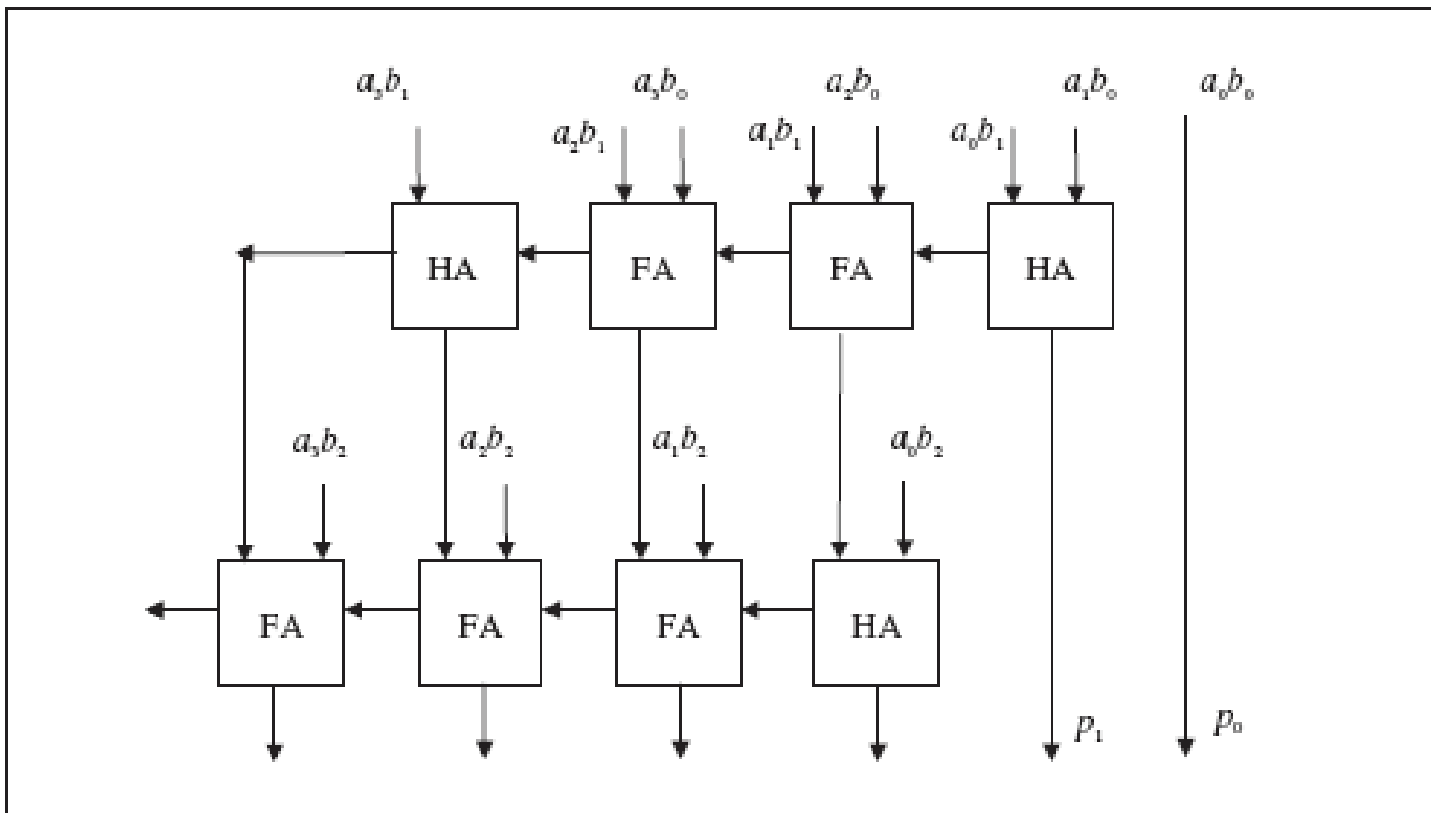
- Le n righe della matrice dei prodotti parziali vengono sommate utilizzando $n-1$ sommatori (i posti vuoti sono degli “zeri”)
 - La struttura ottenuta è molto regolare e, quindi, ben integrabile.

Es. Addizionatore parallelo per le prime due righe della matrice

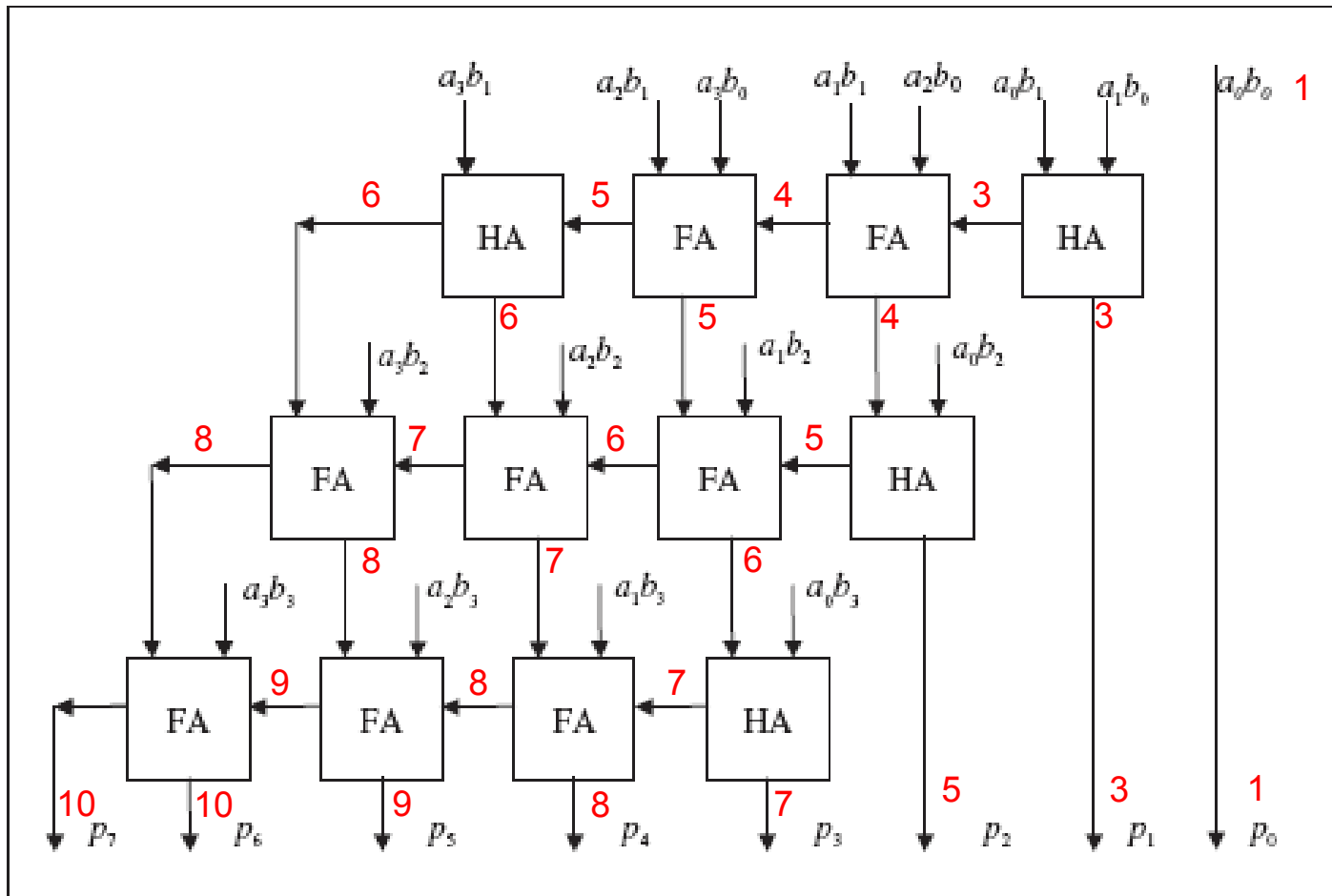


Prodotto come somma di righe

Es. aggiunta della terza riga

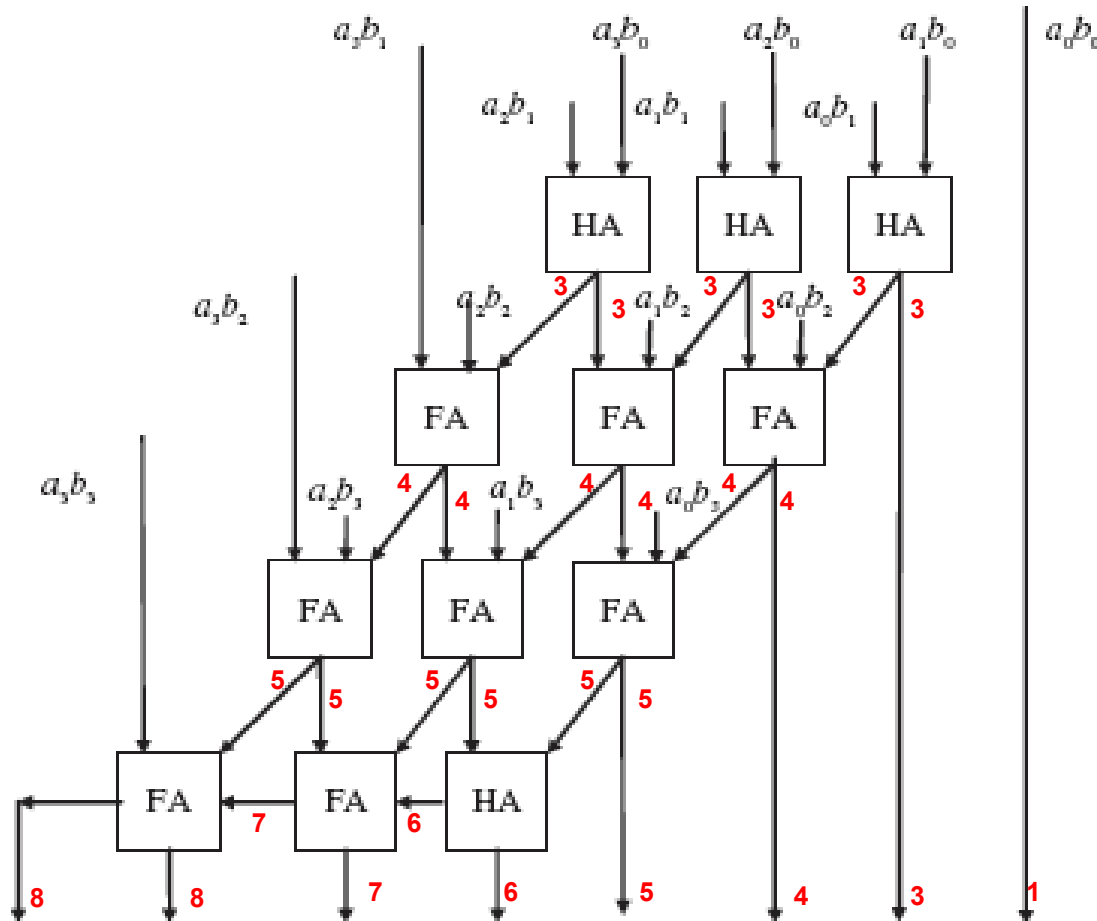


Prodotto come somma di righe (ripple carry multiplication)



HA e FA hanno
tempo di
computazione
2T

Prodotto come somma per diagonali



Ipotesi:
HA e FA hanno
tempo di
computazione
unitario

Prodotto come somma per diagonali: ritardo

Fatte le stesse ipotesi del caso precedente, si vede che il circuito è più veloce di quello ottenuto per somma di righe, poiché il ritardo massimo è di 8 a fronte di 10 ritardi elementari.

- Il costo totale di FA è identico nei due casi ed entrambe le strutture sono facilmente integrabili su silicio poiché fortemente regolari e ripetitive

Prodotto come somma per colonne

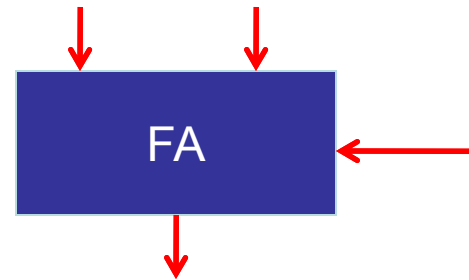
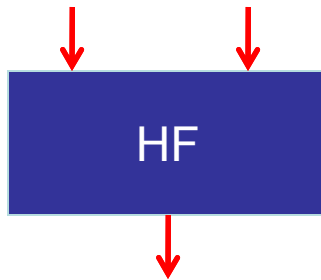
- È il metodo più vicino a quello con cui vengono eseguite manualmente le somme sulla matrice dei prodotti
- Utilizza un contatore del numero di bit 1 presenti in una colonna della matrice dei prodotti parziali
- Per realizzare il conteggio si usa un contatore parallelo avente n ingressi e $m = \log(n+1)$ uscite che forniscono la codifica binaria del numero di ingressi che, in un dato istante, valgono 1

Conteggio 1

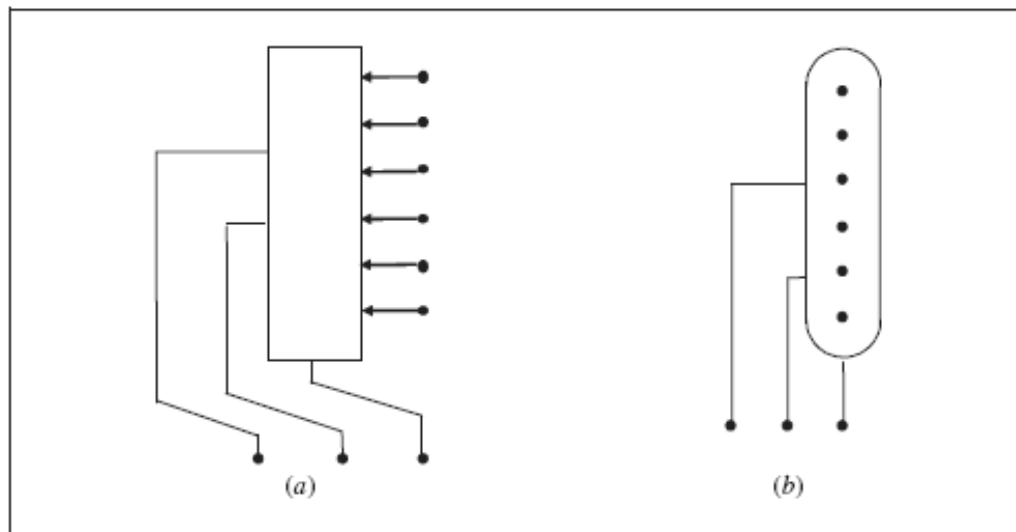
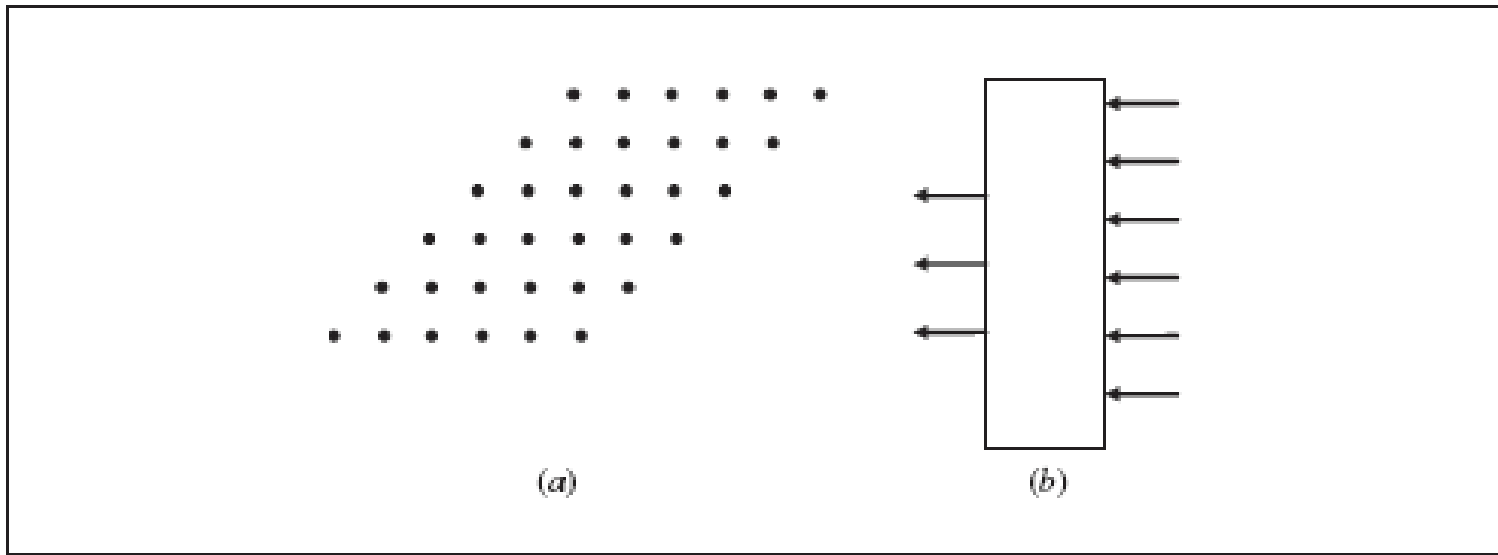
X5	X4	X3	X2	X1		Y2	Y1	y0
0	0	0	0	0		0	0	0
				1		0	0	1
			1	1		0	1	0
		1	1	1		0	1	1
	1	1	1	1		1	0	0
1	1	1	1	1		1	0	1

Realizzazione del contatore

- Il conteggio di un solo bit è il bit stesso
- Il conteggio di due ingressi è un HA
- Il conteggio di tre ingressi è un FA



Schema della matrice dei prodotti elementari (a) e simbolo del contatore parallelo (blocco elementare per la somma per colonne) (b)

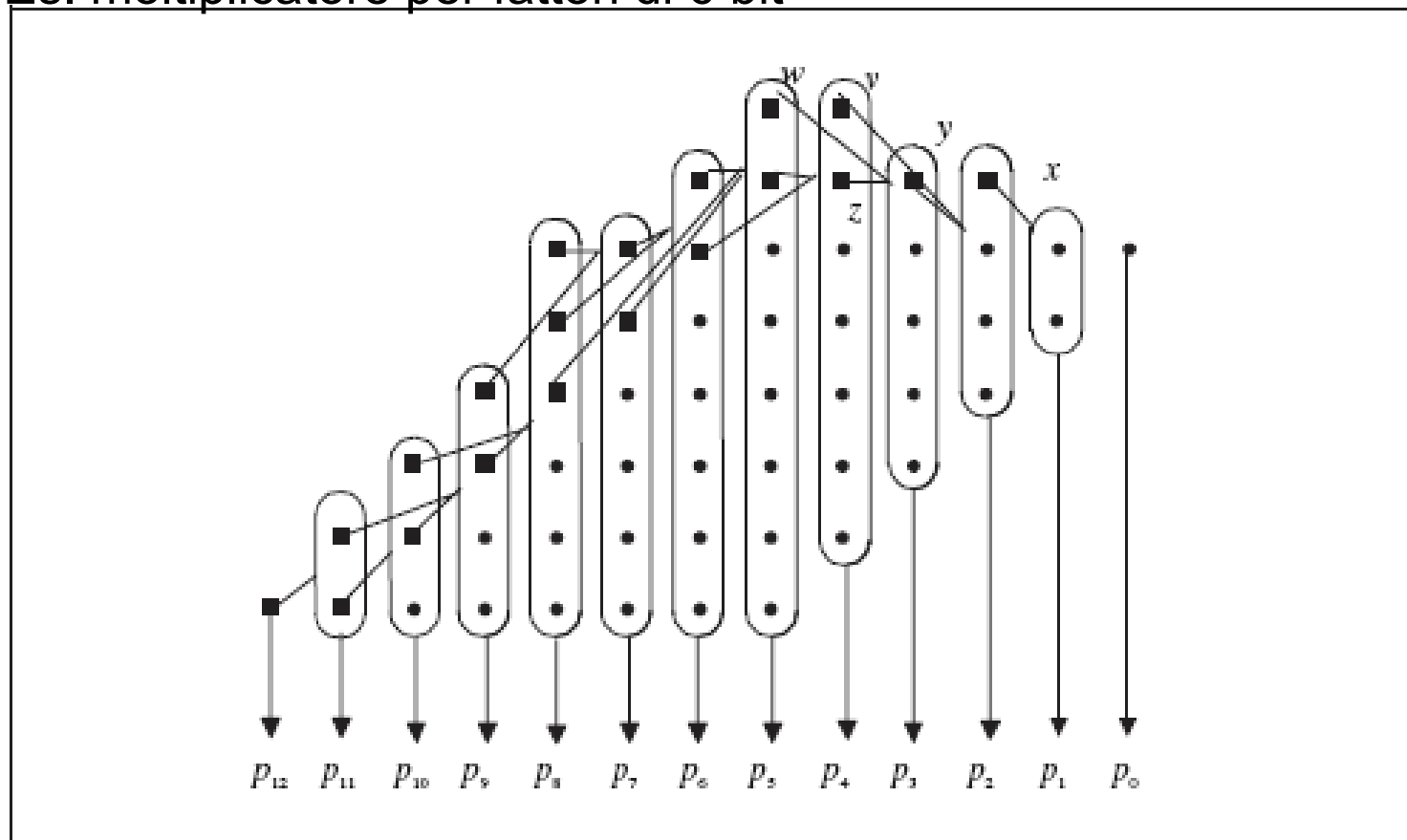


Altri simboli per il
contatore
parallelo...

COME USARE I
CONTATORI??

Schema del circuito di somma per colonne

Es. moltiplicatore per fattori di 6 bit

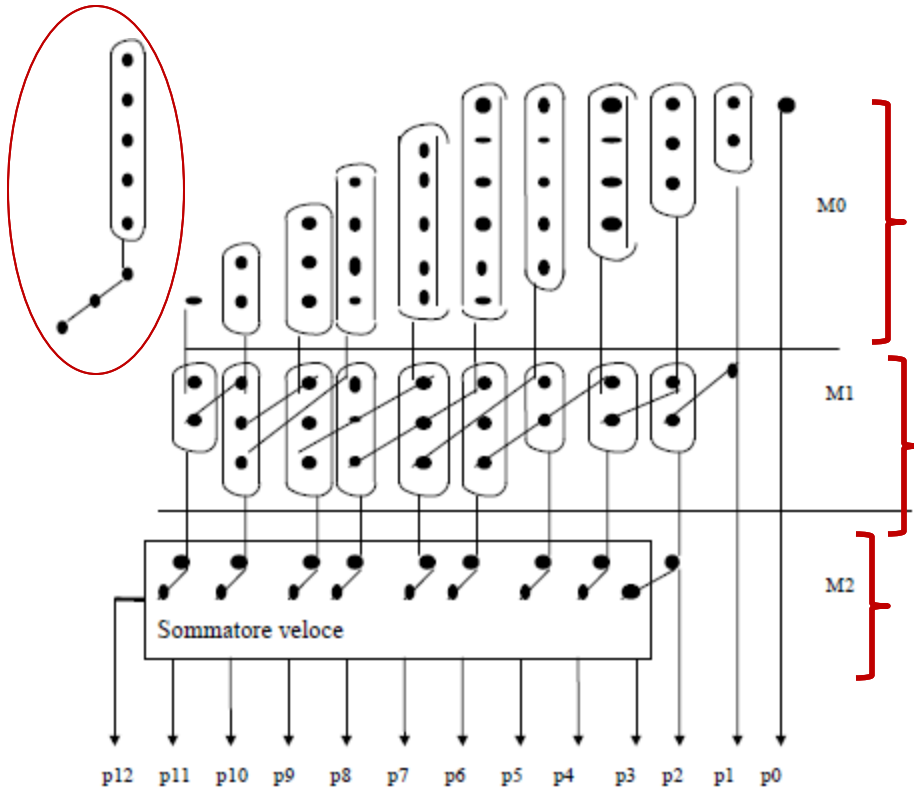


Somma per colonne

- Nell'esempio considerato (moltiplicatore per fattori di 6 bit), la struttura ha 13 uscite, p0-p12; poiché un moltiplicatore con fattori di 6 bit produce 12 uscite, il bit 13 sarà sempre nullo.
- Perdita di regolarità dei circuiti e delle connessioni
- Velocità limitata dalla propagazione dei riporti fra i contatori

Somma per colonne: schema alternativo

altro simbolo per il
contatore:



I bit della matrice iniziale vanno in ingresso a una serie di contatori (matrice M0) le cui uscite costituiscono gli ingressi a una seconda matrice di contatori M1.

Il procedimento viene ripetuto per passare dalla matrice M1 (tre righe) alla matrice M2 (due sole righe). A questo punto la somma può essere eseguita mediante un normale addizionatore

Moltiplicatore basato su somma di colonne

- Il prodotto viene ottenuto in tre fasi successive
 - 1) Determinazione della matrice iniziale M_0 (matrice di porte AND)
 - 2) Riduzione della matrice iniziale tramite applicazione di opportuni contatori parallelo fino ad ottenere una matrice composta da due sole righe.
 - Moltiplicatori con fattori da 8 fino a 127 bit riducono il numero di righe della matrice iniziale a una matrice finale di due righe in non più di 3 passi
 - 3) Somma delle due righe dell'ultima matrice ottenuta nella seconda fase
- Tempo di esecuzione dell'ordine di $\log_2 n$

Moltiplicatore a celle MAC (Multiply-and-Accumulate) (1/4)

0	0	0	x_3y_0	x_2y_0	x_1y_0	x_0y_0
0	0	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0
0	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0
x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0

- ☐ I prodotti parziali sono calcolati con porte AND
- ☐ Tutte le celle sulla stessa riga usano per il prodotto lo stesso bit dell'operando Y
- ☐ Tutte le celle sulla stessa diagonale usano lo stesso bit dell'operando X

Le operazioni che coinvolgono un singolo termine prodotto sono:

- ✓ calcolo del prodotto stesso

$$x_1y_2$$

- ✓ somma parziale con gli altri termini che si trovano sulla stessa colonna

$$s_{2,1} = x_3y_0 + x_2y_1$$

- ✓ somma con il riporto entrante $c_{1,2}$ dallo stadio precedente proveniente da x_0y_2

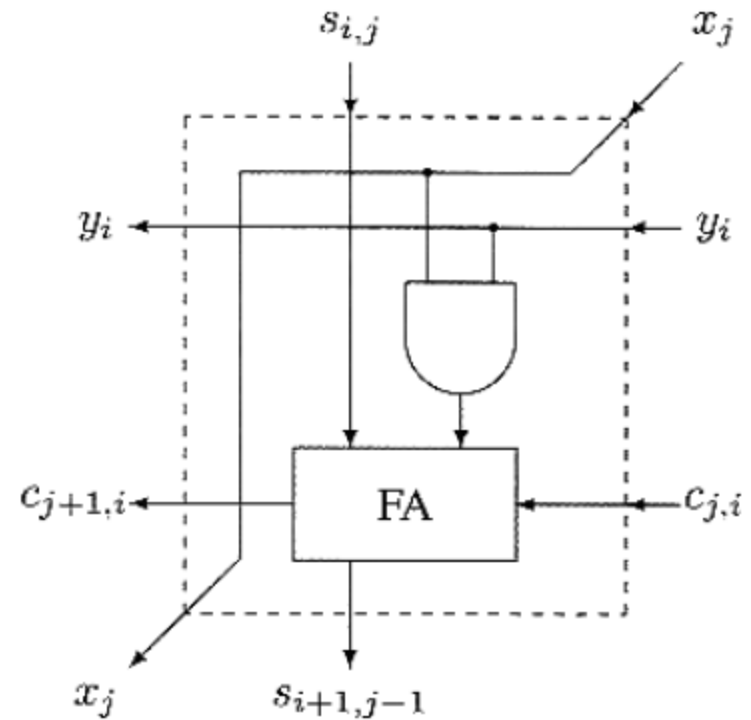
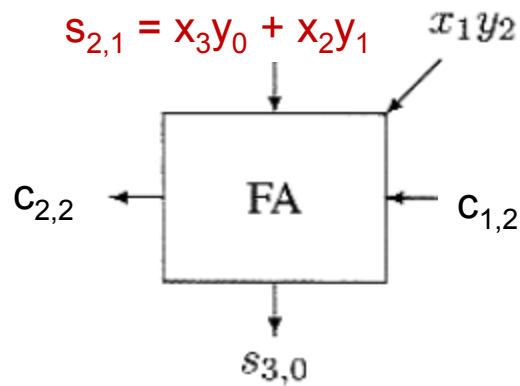
Esse producono in uscita:

- ✓ la nuova somma parziale $s_{3,0} = x_3y_0 + x_2y_1 + x_1y_2$

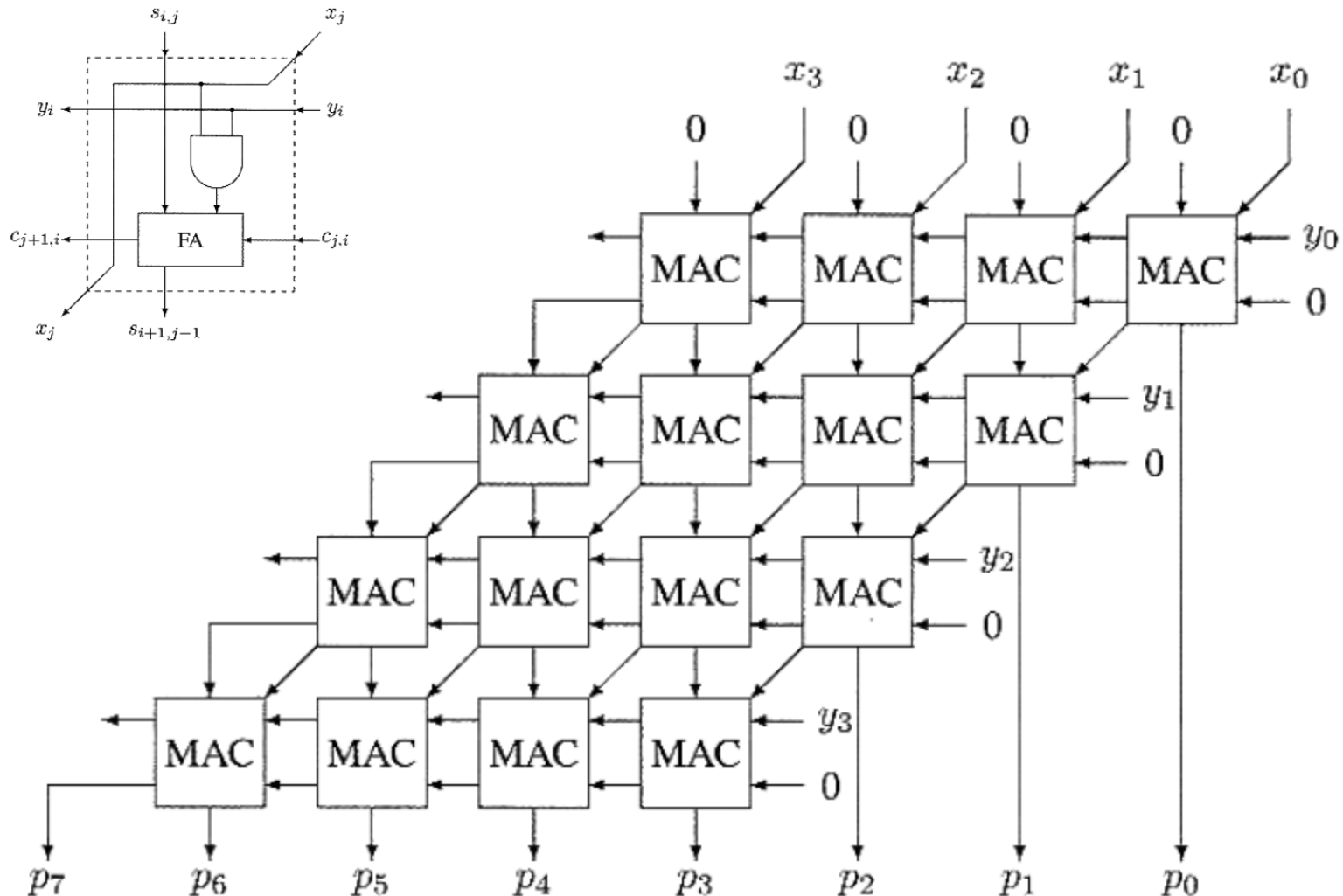
- ✓ Il riporto $c_{2,2}$ da propagare alla analoga somma immediatamente a sinistra

Moltiplicatore a celle MAC: architettura della singola cella (2/4)

0	0	0	x_3y_0	x_2y_0	x_1y_0	x_0y_0
0	0	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0
0	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0
x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0

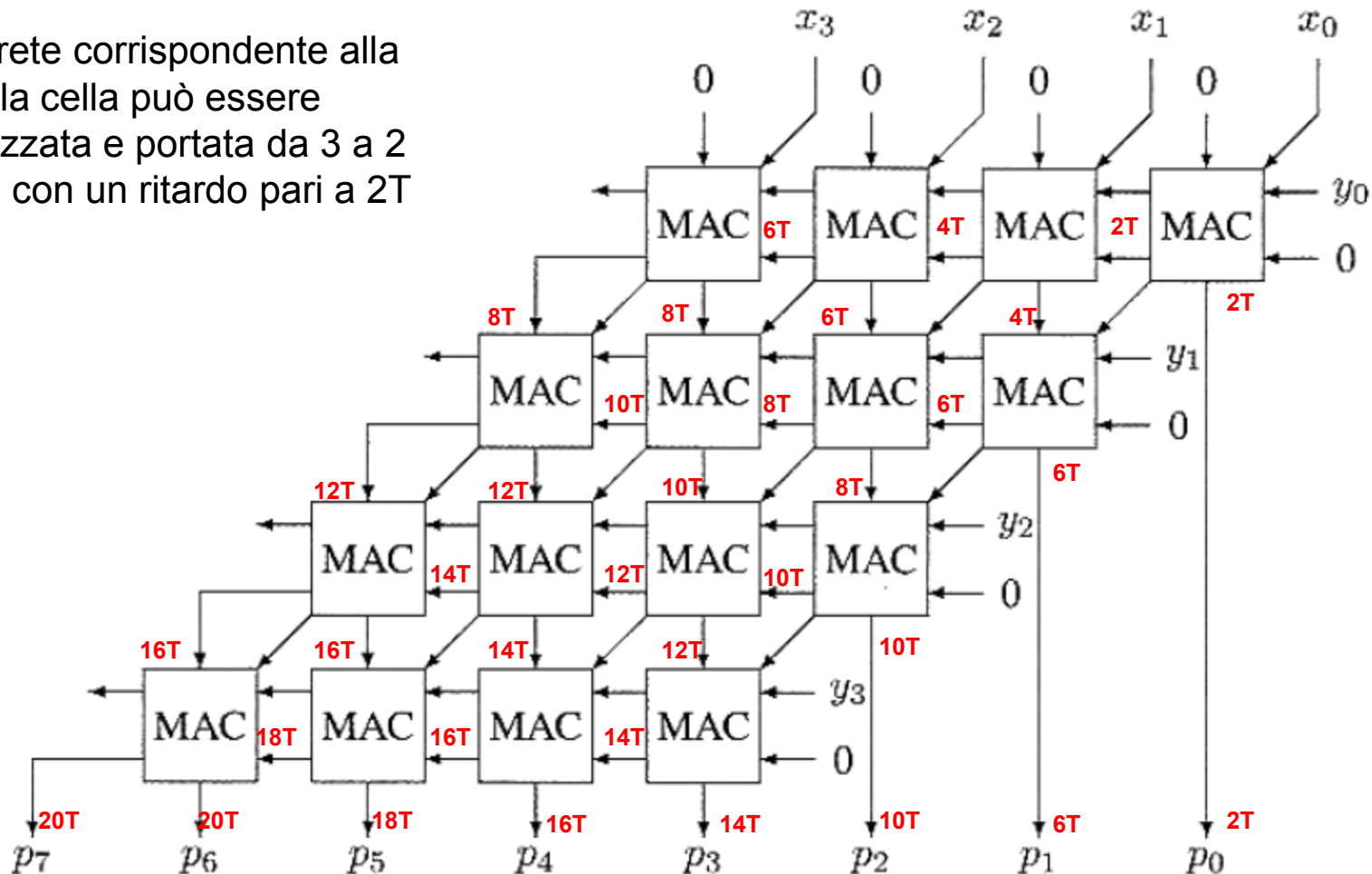


Moltiplicatore a celle MAC: struttura a matrice (3/4)



Moltiplicatore a celle MAC: valutazione dei ritardi (4/4)

□ La rete corrispondente alla singola cella può essere ottimizzata e portata da 3 a 2 livelli, con un ritardo pari a $2T$



Prodotto di numeri interi con segno

Prodotto: considerazioni sull'algoritmo manuale

1010	Multiplicand Y
<u>1101</u>	Multiplier X = $x_3x_2x_1x_0$
1010	x_0Y
0000	x_12Y
1010	x_22^2Y
<u>1010</u>	x_32^3Y
10000010	Product P = $\sum_{j=0}^3 x_j 2^j Y$

$$X = X_{n-1} \dots X_0 \quad Y = Y_{m-1} \dots Y_0 \quad X > 0, Y > 0$$

L'operazione $X \times Y$ prevede il calcolo di una serie di prodotti $x_j Y$ in cui il moltiplicando Y viene moltiplicato per il bit j-esimo del moltiplicatore X, per $j=0..n-1$.

Ciascun prodotto parziale deve essere opportunamente shiftato di j posizioni a sinistra per poter concorrere al calcolo del prodotto finale.

$$P = \sum_{j=0}^{n-1} x_j 2^j Y$$

Prodotto: calcolo della somma parziale dei prodotti generati ad ogni passo

```

  1010
  1101
  ----
00000000
  1010
  ----
00001010
  0000
  ----
00001010
  1010
  ----
00110010
  1010
  ----
10000010
    
```

Multiplicand Y
 Multiplier $X = x_3x_2x_1x_0$
 $P_0 = 0$
 x_0Y
 $P_1 = P_0 + x_0Y$
 x_12Y
 $P_2 = P_1 + x_12Y$
 x_22^2Y
 $P_3 = P_2 + x_22^2Y$
 x_32^3Y
 $P_4 = P_3 + x_32^3Y = P$

Per evitare di dover memorizzare tutti i prodotti parziali per la somma finale, ad ogni passo si può calcolare una somma parziale dei prodotti data da:

$$P_{i+1} = P_i + x_i 2^i Y$$

$$P_0 = 0$$

Ad ogni passo è necessario effettuare un'operazione di shift di i posizioni

Prodotto: algoritmo alternativo

Al passo $(i+1)$ -esimo invece della sequenza di operazioni:

1. moltiplicazione di x_i per Y ,
 2. shift a sinistra di i posizioni del prodotto parziale,
 3. somma del prodotto parziale shiftato con P_i per calcolare P_{i+1} ,
- è possibile considerare una versione alternativa dell'algoritmo che considera la nuova sequenza di operazioni:



1. moltiplicazione di x_i per Y ,
2. somma del prodotto parziale con P_i
3. shift a destra di una posizione del prodotto parziale calcolato al punto 2 per calcolare P_{i+1}

$$P_i := P_i + x_i Y;$$

$$P_{i+1} := 2^{-1} P_i$$

L'algoritmo alternativo è del tutto equivalente a quello derivato dalla procedura manuale, ma ha il vantaggio che ad ogni passo si effettua sempre uno shift di una sola posizione a destra

Prodotto: algoritmo alternativo

Y 1010		
X 1101		
0000 0000		P0
1010		
0000 1010		ADD;shift
000 0101	0	P1
0000		
000 0101	0	ADD;shift
00 0010	10	P2
1010		
00 1100	10	ADD;shift
0 0110	010	P3
1010		
1 0000	010	ADD;shift
1000	0010	
		
A	Q	

CONSIDERAZIONI

- ad ogni passo il contenuto di A viene sommato con $Y \cdot X_i$ e l'intera stringa A.Q viene shiftata a destra di una posizione, inserendo uno 0 in testa
- Q inizialmente è vuoto e viene "riempito" man mano che avvengono gli shift; nell'ottica della realizzazione di questo algoritmo si potrebbe caricare in Q il moltiplicatore X e prendere la cifra corrente da moltiplicare dal bit Q[0]
- l'algoritmo così com'è non va bene per numeri relativi; è necessario tener conto che i fattori possano essere negativi

Prodotto di numeri interi relativi

- ❑ Per il calcolo del prodotto di due interi relativi non è possibile applicare l'algoritmo derivato dalla procedura manuale, che fornisce un risultato errato
- ❑ Se i numeri sono codificati in modulo e segno l'unica possibilità è quella di calcolare separatamente il prodotto dei moduli e dei segni
- ❑ Se i numeri sono codificati in complementi a due una soluzione concettualmente semplice consiste nel negare tutti gli operandi negativi, effettuare un'operazione unsigned sui numeri positivi risultanti e poi negare il risultato se necessario
 - ✓ **Problema:** sono necessari ulteriori cicli di clock per negare X, Y e il risultato a doppia lunghezza P

Numeri relativi in complementi a due: considerazioni

Una soluzione alternativa a quella vista si basa su alcune proprietà della rappresentazione in complementi:

$$\begin{aligned} -X &= x'_{n-1} x'_{n-2} \dots x'_1 x'_0 + 000\dots 1 \pmod{2^n} \\ \text{ma } x'_i &= 1 - x_i \pmod{2} \end{aligned}$$



$$-X = (111\dots 11 - x_{n-1} x_{n-2} \dots x_1 x_0) + 000\dots 01 \pmod{2^n}$$

□ Se $X > 0$ possiamo scrivere $X = \sum_{i=0}^{n-2} 2^i x_i$ poiché $x_{n-1} = 0$

□ Se $X < 0$ la relazione trovata non è valida, mentre si può scrivere

$$\begin{aligned} -X &= 111\dots 11 - (0 x_{n-2} \dots x_1 x_0 + 100\dots 00) + 000\dots 01 = \\ &= (111\dots 11 - 100\dots 00 + 000\dots 01) - x_{n-2} \dots x_1 x_0 = \\ &= 100\dots 00 - x_{n-2} \dots x_1 x_0 = \\ &= 2^{n-1} - x_{n-2} \dots x_1 x_0 \Rightarrow \end{aligned}$$

$$X = -2^{n-1} + x_{n-2} \dots x_1 x_0 = -2^{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

Moltiplicatori in complementi a due

$$\begin{array}{ll} \text{Se } X > 0 & X = \sum_{i=0}^{n-2} 2^i x_i \\ \text{Se } X < 0 & X = -2^{n-1} + \sum_{i=0}^{n-2} 2^i x_i \end{array} \quad \Rightarrow \quad X = -2^{n-1} x^{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

L'equazione trovata implica che **possiamo trattare i bit $x_{n-2}x_{n-3}...x_1x_0$ di un intero in complemento a due come se appartenessero ad un numero unsigned**. Inoltre:

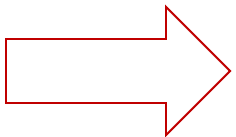
- Per un numero positivo viene assegnato peso 2^{n-1} al bit di segno x_{n-1} : poiché è nullo il suo contributo al numero è 0 sulla cifra più significativa
- Per un numero negativo viene assegnato peso -2^{n-1} al bit di segno x_{n-1} : poiché esso vale 1 il suo contributo al numero è -1 sulla cifra più significativa

Con questo schema è possibile utilizzare una tecnica di moltiplicazione unsigned con un'unica modifica: **quando si moltiplica il bit di segno è necessario effettuare una sottrazione** invece di un'addizione nel passo finale se si incontra un segno negativo.

Modifiche all'algoritmo manuale

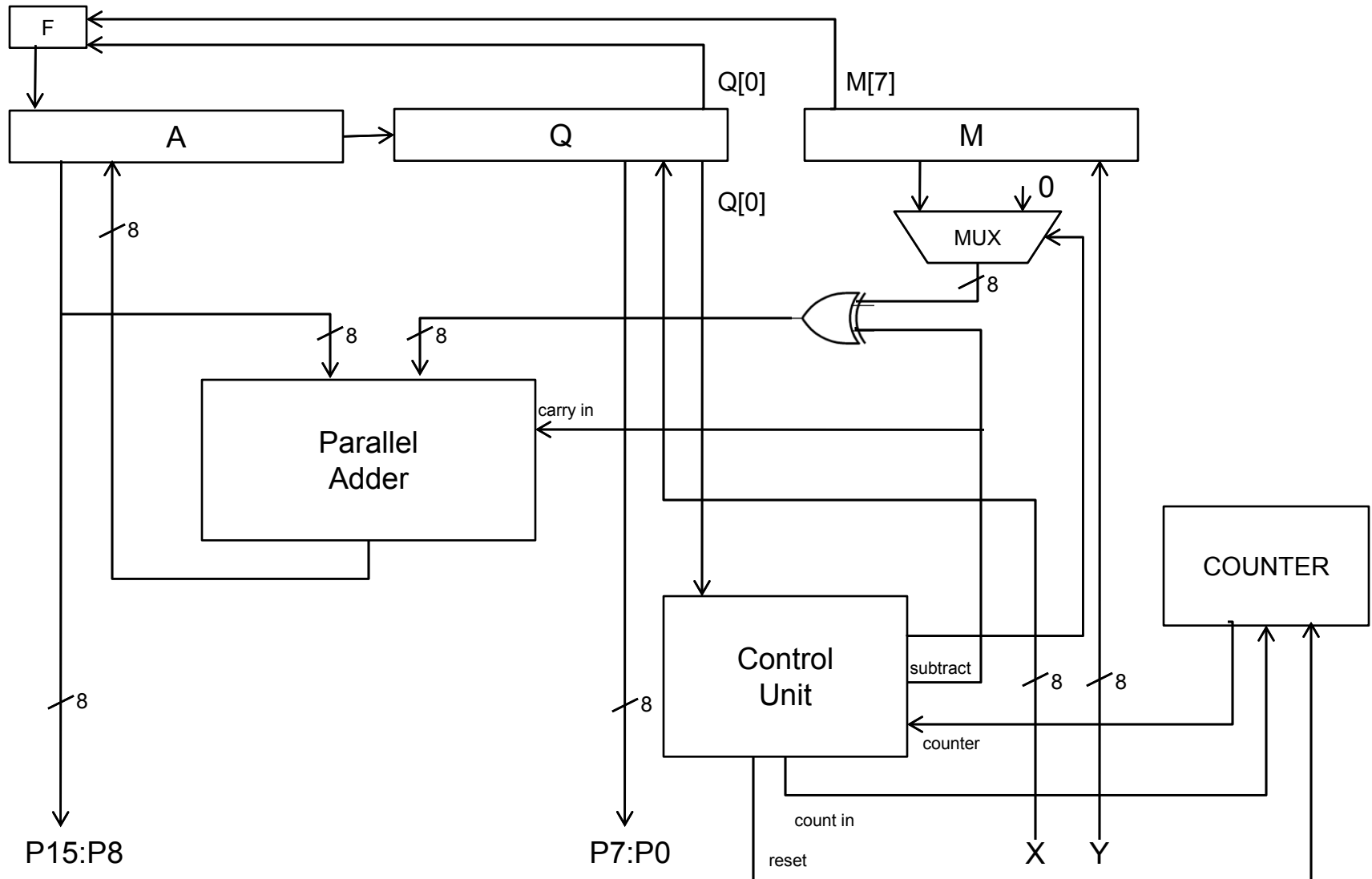
A seconda dei segni dei due fattori X (moltiplicatore) e Y (moltiplicando) si possono avere 4 casi:

1. $X > 0, Y > 0$: moltiplicazione fra unsigned, effettuata con passi add e shift
2. $X > 0, Y < 0$: ogni volta che si moltiplica Y per $x_j \neq 0$ il prodotto parziale è negativo, quindi il bit più significativo di A deve essere 1
3. $X < 0, Y > 0$: per i primi $n-1$ prodotti il prodotto parziale è positivo, mentre solo per l' n -esimo prodotto è necessario effettuare un passo di correzione con la sottrazione $A-M$
4. $X < 0, Y < 0$: la procedura segue il caso 2: il bit più significativo di A è 0 finché $x_j = 0$ e diventa 1 quando moltiplico $x_j = 1$ per Y ; anche in questo caso per l' n -esimo prodotto è necessario effettuare un passo di correzione con la sottrazione $A-M$



- Il caso 2 viene gestito grazie a un latch F che viene messo a 1 quando il moltiplicando è negativo ($M[7]=1$) e la cifra corrente del moltiplicatore ($Q[0]$) è 1
- Il caso 3 viene gestito effettuando la correzione finale

Moltiplicatori in complementi a due: moltiplicatore di Robertson



Algoritmo di Robertson per operandi interi relativi di 8 bit

```
2CMultiplier:    (in:INBUS; OUT:OUTBUS)
                  register A[7:0],M[7:0],Q[7:0],COUNT[2:0],F;
                  bus INBUS[7:0],OUTBUS[7:0];

BEGIN:           A:=0,COUNT:=0,F:=0,
INPUT:           M:=INBUS;Q:=INBUS;

ADD:             A[7:0]:= A[7:0] + M [7:0] x Q[0],
                  F:= (M[7] and Q[0]) or F;
RSHIFT:         A[7]:= F,  A[6:0].Q:= A.Q[7:1],
INCREMENT:      COUNT:=COUNT+1

TEST:           if COUNT<7 then go to ADD;

SUBTRACT:        A[7:0]:=A[7:0]-M[7:0]xQ[0];    {l'ultima op è sempre SUB}
RSHIFT:         A[7]:= A[7],  A[6:0].Q:= A.Q[7:1];

OUTPUT:         OUTBUS:=Q; OUTBUS:=A;
END 2CMultiplier;
```

Esempio di moltiplicazione con operandi di 8 bit

Step	Action	F	Accumulator A	Register Q
0	Initialize registers	0	00000000	<u>1</u> 0110011 = multiplier X
1			11010101	= multiplicand Y = M
	Add M to A	1	11010101	<u>1</u> 0110011
	Right-shift F.A.Q	1	11101010	1 <u>1</u> 011001
2			11010101	
	Add M to A	1	10111111	1 <u>1</u> 011001
	Right-shift F.A.Q	1	11011111	11 <u>1</u> 01100
3			00000000	
	Add zero to A	1	11011111	11 <u>1</u> 01100
	Right-shift F.A.Q	1	11101111	111 <u>1</u> 0110
4			00000000	
	Add zero to A	1	11101111	111 <u>1</u> 0110
	Right-shift F.A.Q	1	11110111	1111 <u>1</u> 011
5			11010101	
	Add M to A	1	11001100	1111 <u>1</u> 011
	Right-shift F.A.Q	1	11100110	01111 <u>1</u> 01
6			11010101	
	Add M to A	1	10111011	01111 <u>1</u> 01
	Right-shift F.A.Q	1	11011101	101111 <u>1</u> 0
7			00000000	
	Add zero to A	1	11011101	101111 <u>1</u> 0
	Right-shift F.A.Q	1	11101110	1101111 <u>1</u>
8			11010101	
	Subtract M from A	1	00011001	1101111 <u>1</u>
	Right-shift A.Q		00001100	11101111 = product P

Codifica di Booth-1 (1/2)

□ Si consideri un intero X la cui rappresentazione in complementi a 2 sia $x_{n-1}x_{n-2}\dots x_0$ e si definisca:

$$y_0 = -x_0,$$

$$y_1 = -x_1 + x_0,$$

$$y_2 = -x_2 + x_1,$$

$$\vdots \quad \vdots$$

$$y_{n-1} = -x_{n-1} + x_{n-2}.$$

□ Moltiplicando la prima equazione per 2^0 , la seconda per 2^1 , la terza per 2^2 e così via, e sommando le n equazioni, si ottiene:

$$y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \dots + y_0 \cdot 2^0 = -x_{n-1} \cdot 2^{n-1} \\ + x_{n-2} \cdot 2^{n-2} + \dots + x_0 \cdot 2^0.$$

Rappresentazione di un
numero in complementi a 2



Codifica di Booth-1 (2/2)

□ Il vettore $y_{n-1}y_{n-2}\dots y_0$, i cui elementi appartengono a $\{-1,0,1\}$, è la **rappresentazione Booth-1** di x e:

$$x = y_{n-1}.2^{n-1} + y_{n-2}.2^{n-2} + \dots + y_0.2^0$$

□ x può essere rappresentato in maniera alternativa utilizzando la sua rappresentazione Booth

□ La rappresentazione Booth-1 di x può essere facilmente ottenuta sostituendo ciascuna coppia di bit adiacenti di x con un valore in $\{-1,0,1\}$, secondo la corrispondenza in tabella (si considera uno zero in coda alla stringa x per valutare la coppia $x_0 x_{-1}$)

$x_j x_{j-1}$	codifica
00/11	0
01	+1
10	-1

1 0 1 1 0 0 1 1 0

-codifico:

-1 +1 0 -1 0 +1 0 -1

Codifica di Booth-2

□ Si consideri un intero X la cui rappresentazione in complementi a 2 sia $x_{n-1}x_{n-2}\dots x_0$ con $n=2m$ bit. Si definisca:

$$y_0 = -2x_1 + x_0,$$

$$y_1 = -2x_3 + x_2 + x_1,$$

$$y_2 = -2x_5 + x_4 + x_3,$$

$$\vdots$$

$$y_{m-1} = -2x_{2m-1} + x_{2m-2} + x_{2m-3}$$

□ Moltiplicando la prima equazione per 4^0 , la seconda per 4^1 , la terza per 4^2 e così via, e sommando le m equazioni, si ottiene:

$$y_{m-1}.4^{m-1} + y_{m-2}.4^{m-2} + \dots + y_0.4^0 = -x_{n-1}.2^{n-1} \\ + x_{n-2}.2^{n-2} + \dots + x_0.2^0.$$

□ Il vettore $y_{m-1}y_{m-2}\dots y_0$, i cui elementi appartengono a $\{-2,-1,0,1,2\}$, è la **rappresentazione Booth-2** di x e:

$$x = y_{m-1}.4^{m-1} + y_{m-2}.4^{m-2} + \dots + y_0.4^0$$

Codifica di Booth: generalizzazione

□ Si consideri un intero X la cui rappresentazione in complementi a 2 sia $x_{n-1}x_{n-2}\dots x_0$ con $n=r.m$ bit. Si definisca:

$$y_0 = -x_{r-1}.2^{r-1} + x_{r-2}.2^{r-2} + \dots + x_1.2 + x_0,$$

$$y_i = -x_{i.r+r-1}.2^{r-1} + x_{i.r+r-2}.2^{r-2} + \dots + x_{i.r+1}.2 + x_{i.r} + x_{i.r-1}, \quad \forall i \in \{1, 2, \dots, m-1\}.$$

□ Il vettore $y_{m-1}y_{m-2}\dots y_0$, i cui componenti appartengono all'intervallo $\{-2^{r-1}, -(2^{r-1}-1), \dots, -2.-1, 0, 1, 2, \dots, 2^{r-1}-1, 2^{r-1}\}$, è la **rappresentazione Booth-r** di X e:

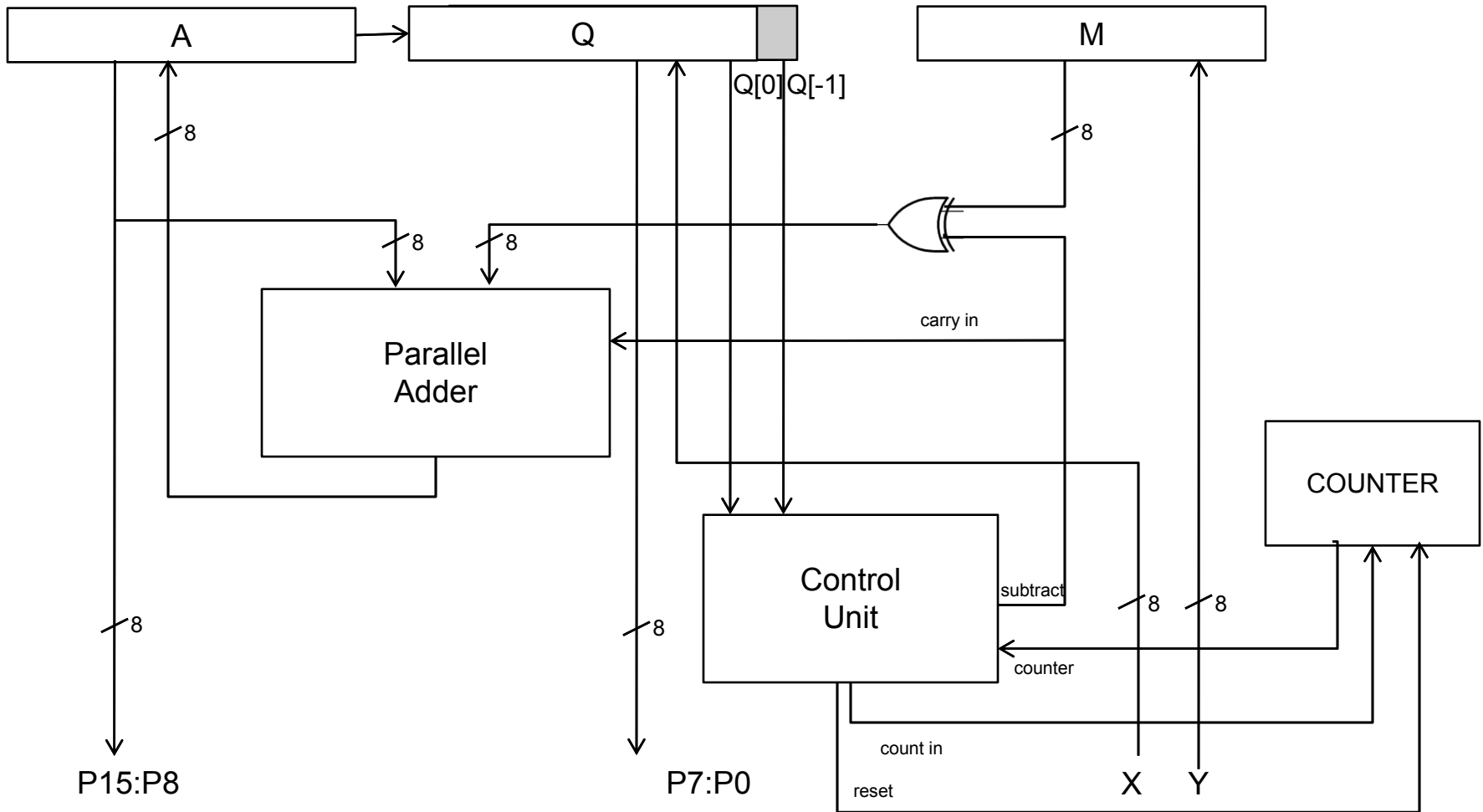
$$x = y_{m-1}.B^{m-1} + y_{m-2}.B^{m-2} + \dots + y_0.B^0, \quad \text{where } B = 2^r.$$

Moltiplicatore di Booth

- ❑ Sfruttando la codifica di Booth è possibile ridurre il numero di moltiplicazioni (e di somme) da effettuare per il prodotto fra numeri relativi.
- ❑ Facendo riferimento alla struttura sequenziale vista per il moltiplicatore di Robertson, poiché ogni cifra della rappresentazione Booth-1 del moltiplicatore appartiene a $\{-1,0,1\}$, a seconda della cifra coinvolta nel prodotto verrà effettuata un'operazione di somma o di differenza (o nessuna delle due) prima dello shift.
- ❑ Mentre nell'algoritmo di Robertson viene esaminato il moltiplicatore $X = x_{n-1} \dots x_j x_{j-1} \dots x_0$ da destra verso sinistra e si considera il bit j -esimo per determinare quale azione effettuare, nell'algoritmo di Booth ad ogni passo si esaminano 2 bit adiacenti, $x_j x_{j-1}$:

$x_j x_{j-1}$	
00/11	Non viene effettuata né la somma né la sottrazione, ma solo lo shift
01	Y viene aggiunto al prodotto parziale corrente
10	Y viene sottratto dal prodotto parziale corrente

Moltiplicatori di Booth



Algoritmo di Booth per operandi interi di 8 bit

BoothMultiplier: (in:INBUS; OUT:OUTBUS)
register A[7:0],M[7:0],Q[7:-1],COUNT[2:0];
bus INBUS[7:0],OUTBUS[7:0];

BEGIN: A:=0,COUNT:=0;
INPUT: M:=INBUS; Q[7:0]:=INBUS;Q[-1]:=0;

SCAN: **if** Q[0]Q[-1] = 01
 then A[7:0]:= A[7:0] + M [7:0];
 else if Q[0]Q[-1] = 10
 then A[7:0]:= A[7:0] - M[7:0];

RSHIFT: A[7]:= A[7], A[6:0].Q:= A.Q[7:0];
INCREMENT: COUNT:=COUNT+1; go to SCAN;

TEST: **if** COUNT<8 **then go to** SCAN;

OUTPUT: OUTBUS:=A;
 OUTBUS:=Q[7:0];

END BoothMultiplier;

Divisori

Divisione fra numeri interi: considerazioni generali

D dividendo $D_{m-1} \dots D_0$
V ≠ 0 divisore $V_{n-1} \dots V_0$

Si vuole determinare un quoziente Q e un eventuale resto R in modo che sia

$$D = Q \times V + R \quad \text{con } 0 \leq R < V$$

□ Se V è espresso su n bit (veri), allora poiché il resto può assumere al massimo il valore $R=V-1$, è anche esso espresso su al più n bit

□ Il massimo quoziente si ha col minimo divisore, ossia con $V=1$: in questo caso Q coincide con D e quindi se D è espresso su m bit anche Q sarà espresso su al più m bit

– in generale Q è espresso su al più **$m-n+1$** bit

NOTA: stiamo considerando i bit veri del numero, cioè assumiamo che il primo degli n/m bit sia non nullo

Divisione fra numeri interi: algoritmo manuale

Nel procedimento manuale di divisione il dividendo viene scandito da sinistra verso destra, e i bit del quoziente vengono determinati uno alla volta a partire da quello più significativo, procedendo con una serie di confronti e sottrazioni.

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

□ Passo iniziale

1. Si confrontano gli n bit più significativi del dividendo (dividendo parziale D_0) con gli n bit del divisore;
2. Si calcola la prima cifra (da sinistra) del quoziente q_0 , che sarà 1 oppure 0 a seconda che il dividendo parziale D_0 contenga o no il divisore;
3. Si effettua la sottrazione fra il dividendo parziale D_0 e il prodotto q_0V , determinando il primo resto parziale R_1

□ Generico passo i

1. Si pone $D_i = R_i$, con R_i resto parziale determinato al passo $i-1$; si confronta D_i con il divisore;
2. Si calcola la i -esima cifra (da sinistra) del quoziente q_i , che sarà 1 oppure 0 a seconda che il dividendo parziale D_i contenga o no il divisore;
3. Si effettua la sottrazione fra il dividendo parziale D_i e il prodotto q_iV shiftato a destra di i posizioni, determinando il nuovo resto parziale R_{i+1}

La procedura termina quando il dividendo è stato scandito completamente: il resto parziale R_i determinato in questo passo costituisce il resto finale R della divisione

Divisione fra numeri interi: passo i-esimo dell'algoritmo manuale

$$D = D_{m-1} \dots D_0$$

$V = V_{n-1} \dots V_0$ si noti che all'inizio dell'algoritmo manuale è necessario allineare il divisore al dividendo parziale shiftandolo a sinistra di $m-n$ posizioni

Al passo i-esimo (con $i=0 \dots m-n$) vengono effettuate le seguenti operazioni:

1. Si confronta $R_i = D_i$ con V : se D_i contiene V , la cifra i-esima (da sinistra) del quoziente q_i sarà 1, altrimenti sarà 0

2. Si calcola il nuovo dividendo parziale

$$R_{i+1} = R_i - q_i 2^{-i} V$$

Ad ogni passo è necessario effettuare un'operazione di shift del divisore di i posizioni a destra e una sottrazione

Divisione: algoritmo manuale

Consideriamo V e Q espressi su n bit e D su $(2n-1)$ bit

D	1 1 1 1 0 1 1	1 0 1 0	V
$R0 = D_7 D_6 D_5 D_4$	1 1 1 1	1 1 0 0	
conf	$q_3 = 1$		
$V^* = 2^3 \cdot V$	1 0 1 0		
$R1 = R0 - V^* q_3$	0 1 0 1 0 1 1		
conf	$q_2 = 1$		
$V^* = 2^2 \cdot V$	1 0 1 0		
$R2 = R1 - V^* q_2$	0 0 0 1 1		
conf	$q_1 = 0$		
$V^* = 2 \cdot V$	1 0 1 0		
$R3 = R2 - V^* q_1$	0 0 1 1		
conf	$q_0 = 0$		
$V^* = 2 \cdot V$	1 0 1 0		
$R4 = R3 - V^* q_0$	0 0 1 1		
			R

$$R_{i+1} = R_i - q_i 2^{-i} V$$

123/10=> Q=12, R=3

Divisione: algoritmo alternativo

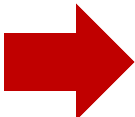
Al passo *i-esimo* invece della sequenza di operazioni:

1. confronto di R_i con V per determinare q_i
2. prodotto di V per q_i e shift di i posizioni a destra
3. sottrazione del prodotto shiftato da R_i per calcolare R_{i+1}

è possibile considerare una versione alternativa dell'algoritmo che effettua la nuova sequenza di operazioni:

1. left shift di una posizione di R_i
2. confronto di $2R_i$ con V per determinare q_i
3. sottrazione del prodotto $q_i V$ da $2R_i$

$$R_i := 2R_i; R_{i+1} := R_i - q_i V;$$


$$R_{i+1} := 2R_i - q_i V;$$

NB: l'algoritmo parte con $R_0 = 2R_0$

L'algoritmo alternativo è del tutto equivalente a quello derivato dalla procedura manuale, ma ha il vantaggio che ad ogni passo si effettua sempre uno shift di una sola posizione a sinistra

Divisione: algoritmo alternativo – es1

Consideriamo V e Q espressi su n bit e D su $(2n-1)$ bit



CONSIDERAZIONI

D=R0	1 1 1 1 0 1 1	1 0 1 0
2R0	1 1 1 1 0 1 1 -	1 1 0 0
conf	1 0 1 0	
2R0-V*=R1	0 1 0 1 0 1 1 -	Q
2R1	1 0 1 0 1 1 - -	
conf	1 0 1 0	
2R1-V*=R2	0 0 0 0 1 1 - -	
2R2	0 0 0 1 1 - - -	
conf	0 0 0 0	
2R2-V*=R3	0 0 0 1 1 - - -	
2R3	0 0 1 1 - - - -	
conf	0 0 0 0	
2R3-V*=R4=R	0 0 1 1 - - - -	
	A=R	Q

- V** - ad ogni passo il dividendo parziale R_i viene shiftato a sinistra, e i suoi primi n bit vengono «confrontati» con V; se risultano maggiori, la cifra corrente di q viene posta a 1 e viene effettuata la sottrazione $R_i - V$
- Il dividendo può essere caricato in una coppia di registri **A.Q**: Q viene “svuotato” man mano che avvengono gli shift e può essere usato per memorizzare le cifre del quoziente calcolate a ogni passo. A conterrà il resto alla fine del processo.
 - l'utilizzo della coppia A.Q come descritto in precedenza implica che $Q[0]$ sia “vuoto” al momento della valutazione; già dal primo passo quindi, il dividendo deve essere shiftato a sinistra di una posizione: ciò implica che abbiamo a disposizione 1 bit in meno per codificare il dividendo

123/10=> Q=12, R=3

Divisione: algoritmo alternativo – es2

D=R0		1 0 0 0 0 1 1	1 0 0 1	V
2R0		1 0 0 0	0 1 1 -	0 1 1 1
conf		0 0 0 0		
2R0-V*=R1		1 0 0 0	0 1 1 -	
2R1	1	0 0 0 0	1 1 - -	
conf		1 0 0 1		
2R1-V*=R2		0 1 1 1	1 1 - -	
2R2		1 1 1 1	1 - - -	
conf		1 0 0 1		
2R1-V*=R3		0 1 1 0	1 - - -	
2R3		1 1 0 1	- - - -	
conf		1 0 0 1		
2R1-V*=R4=R		0 1 0 0	- - - -	
<hr/>				
				
		A=R	Q	

Nota: l'algoritmo mostrato nell'esempio riprende la procedura manuale in cui idealmente si fa uso di comparatori e si abbassano le cifre necessarie ad avere una differenza non negativa

67/9 => Q=7,R=4

Divisione e comparatori

Come visto per la procedura manuale, l'operazione di divisione richiede una successione di sottrazioni e di confronti.

Idealmente il confronto potrebbe essere effettuato da un circuito **comparatore**, che riceve in ingresso due numeri interi A e B e produce due uscite α e β tali che:

$$\alpha = 1 \Rightarrow A > B$$

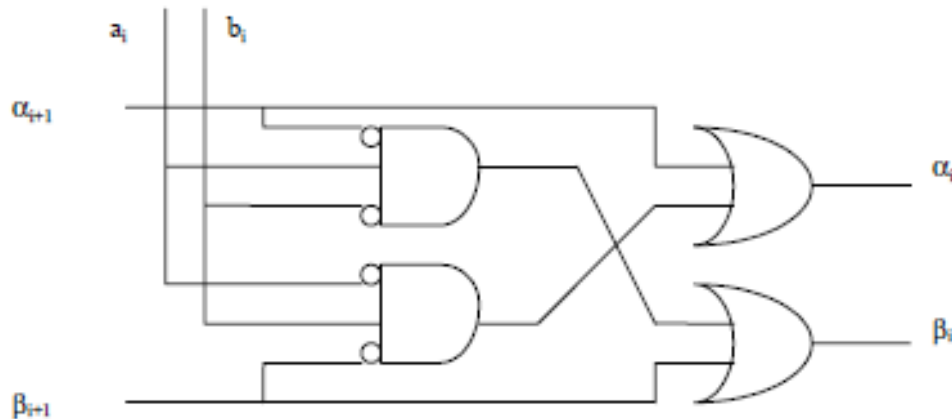
$$\beta = 1 \Rightarrow A < B$$

$$\alpha = \beta = 0 \Rightarrow A = B$$

Per realizzare il comparatore su n bit è possibile collegare opportunamente dei comparatori elementari che confrontano cifre di un solo bit.

Se ambedue i numeri A e B sono interi senza segno si verifica facilmente che confrontando i bit di A e quelli di B a partire da sinistra verso destra (iniziando quindi dai bit più significativi) il risultato del confronto viene determinato non appena uno dei due numeri ha in posizione i-esima un bit di valore 0 e l'altro lo ha di valore 1: il numero col bit a 1 è certamente il maggiore, indipendentemente dai valori dei bit meno significativi.

Comparatore elementare



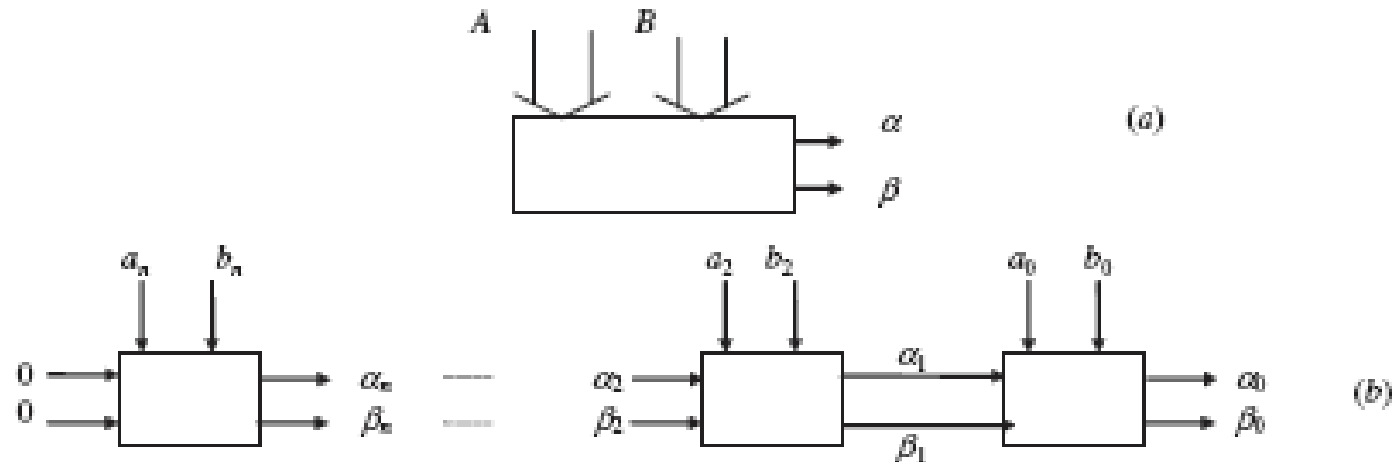
Il comparatore elementare prende in ingresso la cifra i -esima dei numeri A e B e i segnali uscenti dallo stadio precedente e fornisce in uscita i segnali α e β .

I segnali di controllo entranti nel primo stadio sono nulli e quelli uscenti dall'ultimo stadio costituiscono i risultato del confronto.

✓ Non appena uno dei segnali α_i e β_i diventa alto, ciascuno stadio a valle di quello i -esimo dovrà semplicemente propagare tali valori allo stadio finale (α_0 e β_0)

α_{i+1}	β_{i+1}	a_i	b_i	α_i	β_i
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	x	x
1	1	0	1	x	x
1	1	1	0	x	x
1	1	1	1	x	x

Comparatore di interi positivi



Confronto di interi relativi in complemento a due

Se si confrontano due numeri ambedue negativi e rappresentati in complementi a due col circuito appena mostrato, il risultato è corretto. Se invece i due numeri hanno segno opposto il risultato sarebbe scorretto poiché si indicherebbe come maggiore il numero negativo.

Il problema può essere risolto applicando al primo stadio a sinistra i bit di segno complementati:

- ❑ se A e B hanno lo stesso segno i due bit più significativi sono uguali e la loro complementazione non modifica il risultato corretto del confronto.
- ❑ se A e B hanno segno diverso il bit più significativo del numero positivo vale ora 1 mentre quello del numero negativo vale ora 0; il primo stadio a sinistra del comparatore determina quindi immediatamente che il numero positivo è maggiore di quello negativo.

Divisione: confronto senza comparatori

Il problema centrale della divisione è il calcolo della cifra del quoziente q_i come confronto fra V e $2R_i$ al passo i -esimo:

➤ Se $V > 2R_i \Rightarrow q_i = 0$

➤ Se $V \leq 2R_i \Rightarrow q_i = 1$

❑ se V è costituito da un numero elevato di cifre l'utilizzo di comparatori potrebbe risultare oneroso in termini di circuiti logici.

❑ **q_i può essere calcolato sottraendo V da $2R_i$ ed esaminando il segno della differenza:** se è *negativo* $q_i = 0$, altrimenti $q_i = 1$

✓ Si noti che la differenza $2R_i - V$ andrebbe comunque calcolata se $q_i = 1$ e in tal caso fornirebbe R_{i+1}

❑ I processi di determinazione di q_i e R_{i+1} possono essere fra loro combinati secondo due principali algoritmi: *restoring* and *non restoring*

Calcolo combinato di q_i e R_{i+1}

L'algoritmo di divisione prevede ad ogni passo i il calcolo di:

$$R_{i+1} := 2R_i - q_i V;$$

Come si è visto, q_i può essere determinato con la differenza

$$\Delta = 2R_i - V$$

Se $\Delta \geq 0$ allora $q_i=1$, e quindi effettivamente la differenza calcolata fornisce il valore di R_{i+1} ;

Se $\Delta < 0$ allora $q_i=0$ e quindi ho calcolato la quantità $2R_i-V$, mentre avrei dovuto calcolare $R_{i+1} = 2R_i$

Per avere il risultato corretto è necessario in tal caso effettuare un'operazione di **restoring** che consiste nel sommare V alla quantità Δ calcolata:

$$\Delta := R_{i+1} := 2R_i - V$$

$$R_{i+1} := R_{i+1} + V$$

Restoring division

La tecnica di **restoring division** esegue sempre, al passo i -esimo dell'algoritmo di divisione, la sottrazione

$$R_{i+1} = 2R_i - V$$

□ se il risultato di tale differenza è negativo, allora $q_i = 0$ e si effettua un'operazione di “restoring” del valore di R_{i+1} sommando a esso il valore V nello stesso passo

$$R_{i+1} := 2R_i - V$$

$$R_{i+1} := R_{i+1} + V$$

□ se il risultato della differenza è positivo, allora $q_i = 1$ e non ci sono ulteriori operazioni da eseguire nello stesso passo.

$$R_{i+1} := 2R_i - V$$

Non-Restoring division(1/2)

La tecnica di **non-restoring division** prende spunto dal fatto che a un'eventuale operazione di restoring (a) effettuata al passo i -esimo nel caso in cui $q_i=0$, segue sempre, al passo $(i+1)$ -esimo, la sottrazione (b).

$$R_i := R_i + V \quad (a)$$

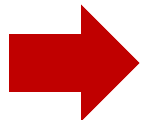
$$R_{i+1} := 2R_i - V \quad (b)$$

Le due operazioni possono essere fuse:

i): $\Delta_i = R_{i+1} = 2R_i - V < 0 \Rightarrow q_i = 0$ effettuo il restoring:

$$R_{i+1} = R_{i+1} + V$$

$i+1$): $\Delta_{i+1} = R_{i+2} = 2R_{i+1} - V = 2(R_i + V) - V = 2R_i + 2V - V =$
 $= 2R_{i+1} + V$

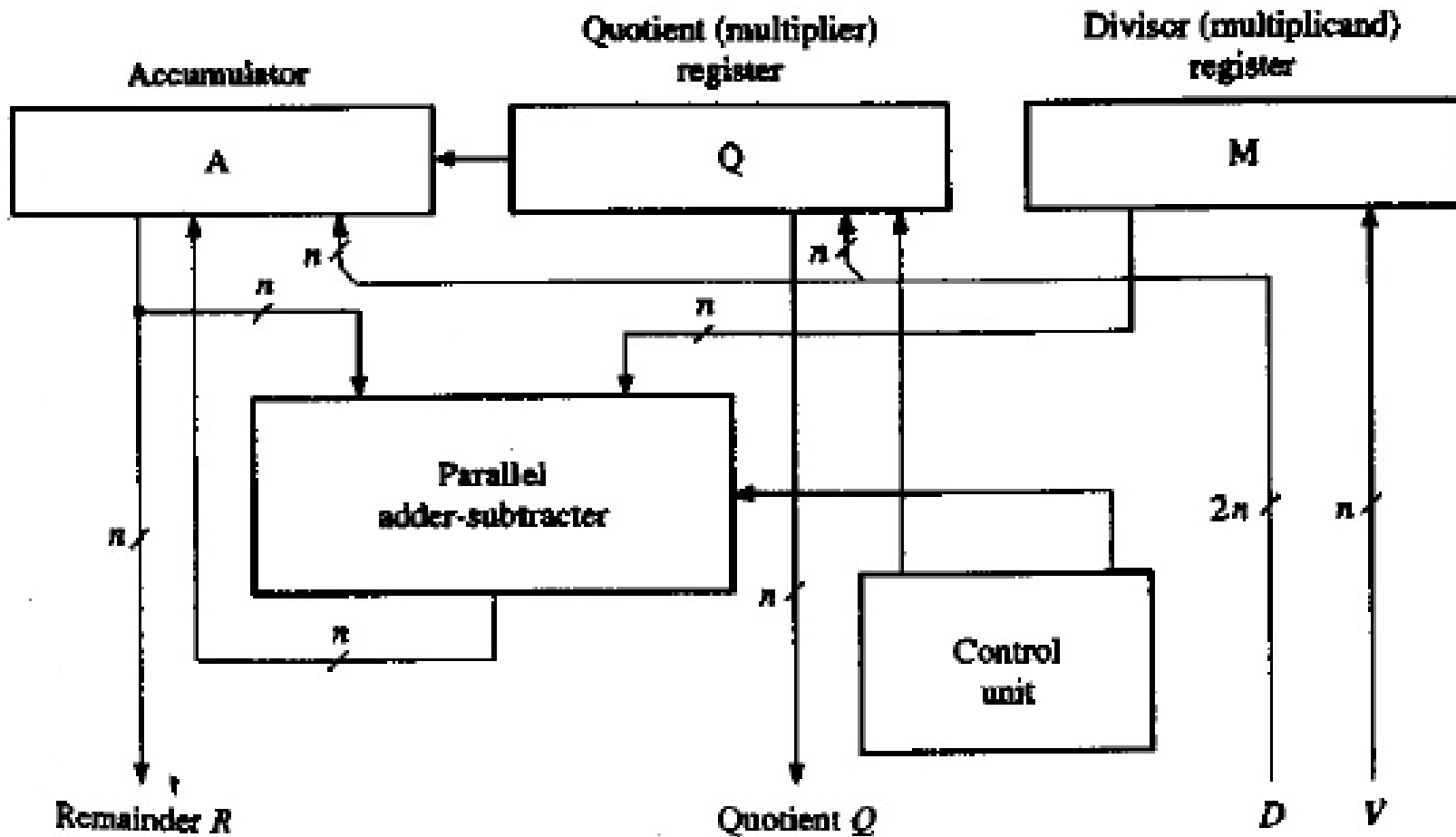


Se al passo i , dopo aver calcolato la differenza $(2R_i - V)$ risulta $q_i = 0$, la prossima operazione eseguita sarà una somma:

$$\Delta = 2R_{i+1} + V$$

NOTA: L'algoritmo deve prevedere esplicitamente la possibilità di avere risultati negativi per la sottrazione e quindi sarà necessario un bit per memorizzare il segno

Schema della divisione



Nota: lo schema mostrato presenta il caso in cui il dividendo contiene al più $2n-1$ bit e Q , M e R hanno n bit ciascuno

Schema della divisione

- ❑ Il divisore V , espresso su n bit, viene caricato in un registro M che rimane costante per tutti i passi del calcolo.
- ❑ Il dividendo D , espresso su $2n-1$ bit, viene memorizzato in $A[n-2:0].Q$ nella fase di inizializzazione. I registri A e Q hanno parallelismo n .
 - ❑ Il primo bit di A viene posto a 0 e la prima operazione è un left shift di A : in questo modo la cifra $Q[0]$ è “libera” e può essere sostituita con la prima cifra calcolata del quoziente. Ad ogni passo i $Q[0]$ conterrà la cifra del quoziente appena calcolata
- ❑ Se si usa la tecnica del **restoring**, ad ogni passo dell'algoritmo viene effettuata la sottrazione del divisore dal dividendo parziale (shiftato) contenuto in A . Il segno della differenza determina la cifra q_i del quoziente e triggera l'eventuale operazione di restoring.
- ❑ Se si usa il metodo del **non restoring**, ad ogni passo dell'algoritmo viene effettuata la somma o la sottrazione del divisore dal dividendo parziale (shiftato) contenuto in A , a seconda che il segno della precedente operazione di somma algebrica sia negativo o positivo.
- ❑ Il **segno** viene memorizzato in un flip-flop S posto in testa al registro A e viene usato per determinare la cifra q_i del quoziente.

Algoritmo di Non-Restoring division

```
NRDivider:      (in:INBUS; OUT:OUTBUS)
                register S,A[n-1:0],M[n-1:0],Q[n-1:0],COUNT[log2n:0];
                bus INBUS[n-1:0], OUTBUS[n-1:0];

BEGIN:          COUNT:=0;S:=0;
INPUT:          A:=INBUS {carico la prima metà del dividendo D (0 in testa)}
                Q:=INBUS {carico la seconda metà del dividendo D}
                M:=INBUS; {divisore V}

LSHIFT:         S.A.Q[n-1:1]=A.Q; {la prima volta S è 0, e dopo lo shift è ancora 0}

SUB:            if S==0 then
                  S.A:=S.A-M;
                else
SUM:            S.A:=S.A+M;
                endif

SETq:          Q[0]:=not S;
                COUNT:=COUNT+1;

COUNT_TEST:   if COUNT< n then goto LSHIFT;
                endif

CORRECTION :    if S==1 then
                  S.A:=S.A+M;
                endif

OUTPUT:        OUTBUS:=Q, OUTBUS:=A;
END NRDivider;
```