Lezione 7
ALU: Moltiplicazione e divisione

F. Pedersini

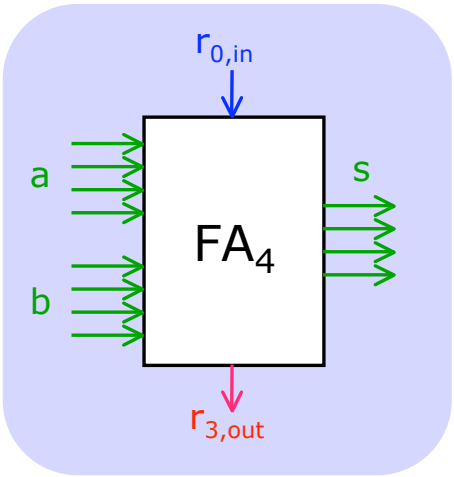
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

Sommario

- Sommatori ad anticipazione di riporto
- Addizionatori modulari
- Moltiplicatori – approccio hardware
- Moltiplicatori – approccio firmware

- Moduli elementari, collegabili in cascata.
  - Complessità del circuito tollerata per piccoli ***n*** **(es. *n*=4)**

- Cammino critico C:
  - M moduli da 4 bit: **C = 6·M**
    N = 16 bit → M = N/4
    → C = 6·N/4 = **24**
  - A propagazione di riporto:
    N = 16 bit
    → C = 3·N = **48**



- Vogliamo 32 bit → 8 sommatori elementari
  - Come collegarli tra loro?

$r_3 = g_3 + p_3 r_2 =$

$= g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 r_0) =$

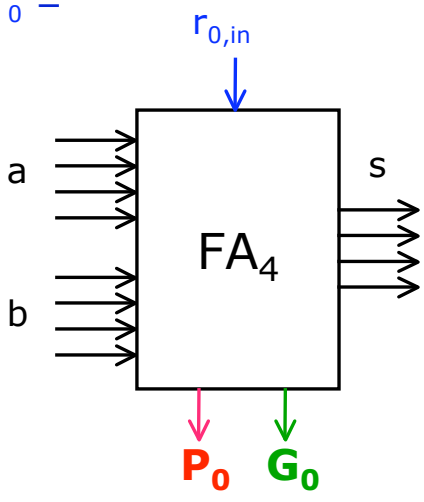
$= (g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0) + p_3 p_2 p_1 p_0 \cdot r_0 =$

$= \mathbf{G_0 + P_0 \cdot r_0}$

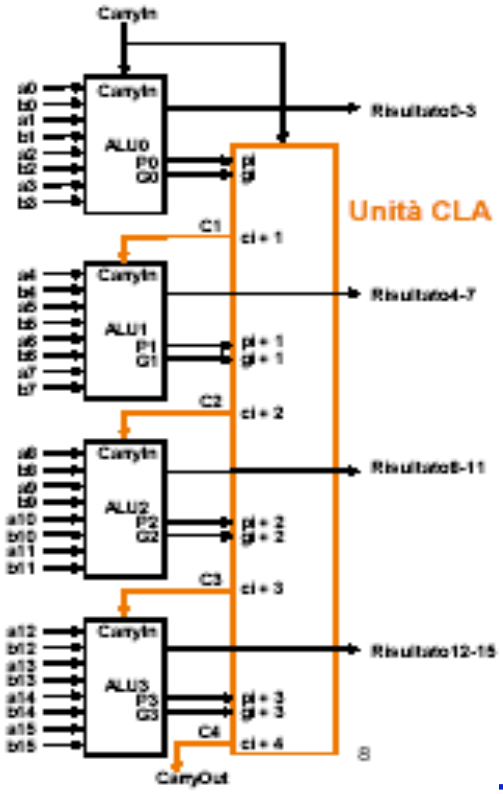
dove:

$\mathbf{P_0} = p_3 p_2 p_1 p_0$

$\mathbf{G_0} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$



Struttura di un sommatore su 16 bit



$C_1 = G_0 + P_0 \cdot r_0$

$C_2 = G_1 + P_1 \cdot C_1 =$
$= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot r_0$

$C_3 = G_2 + P_2 \cdot C_2 =$
$= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 +$
$+ P_2 \cdot P_1 \cdot P_0 \cdot r_0$

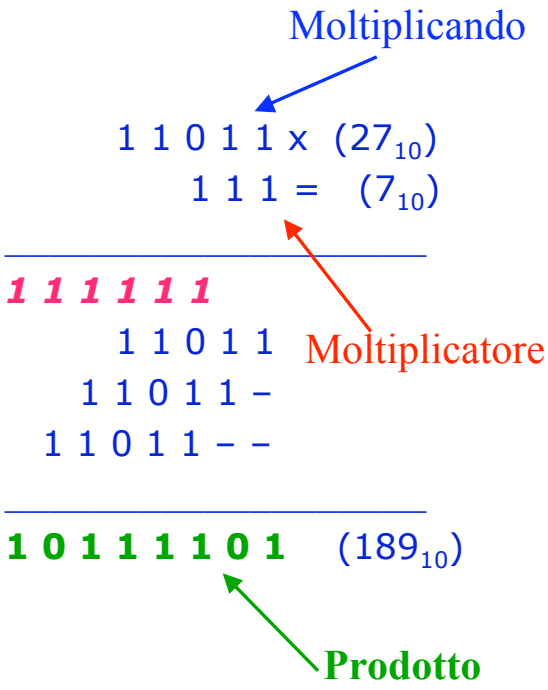
$r_{out} = C_4 = G_3 + P_3 \cdot C_3 =$
$= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 +$
$+ P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot r_0$

Cammino critico = 6+6 = 12
- CLA + prop: 6M = 24
- Prop: 3N = 48

Sommario

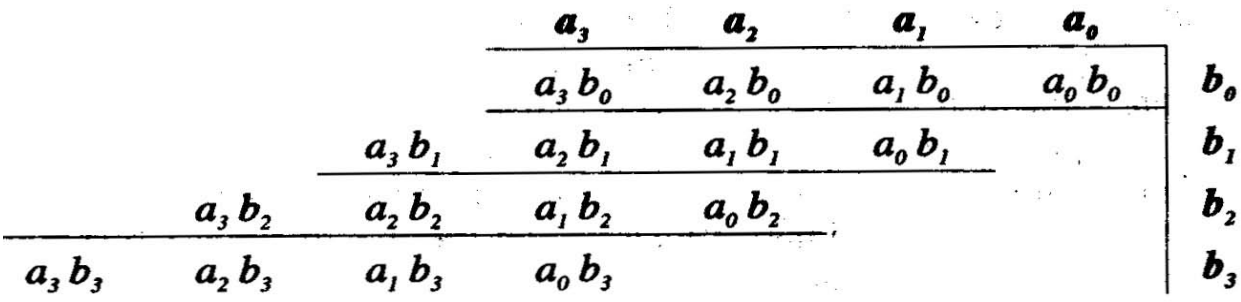
- Sommatori ad anticipazione di riporto
- Addizionatori modulari
- Moltiplicatori – approccio hardware
- Moltiplicatori – approccio firmware

Moltiplicazione binaria



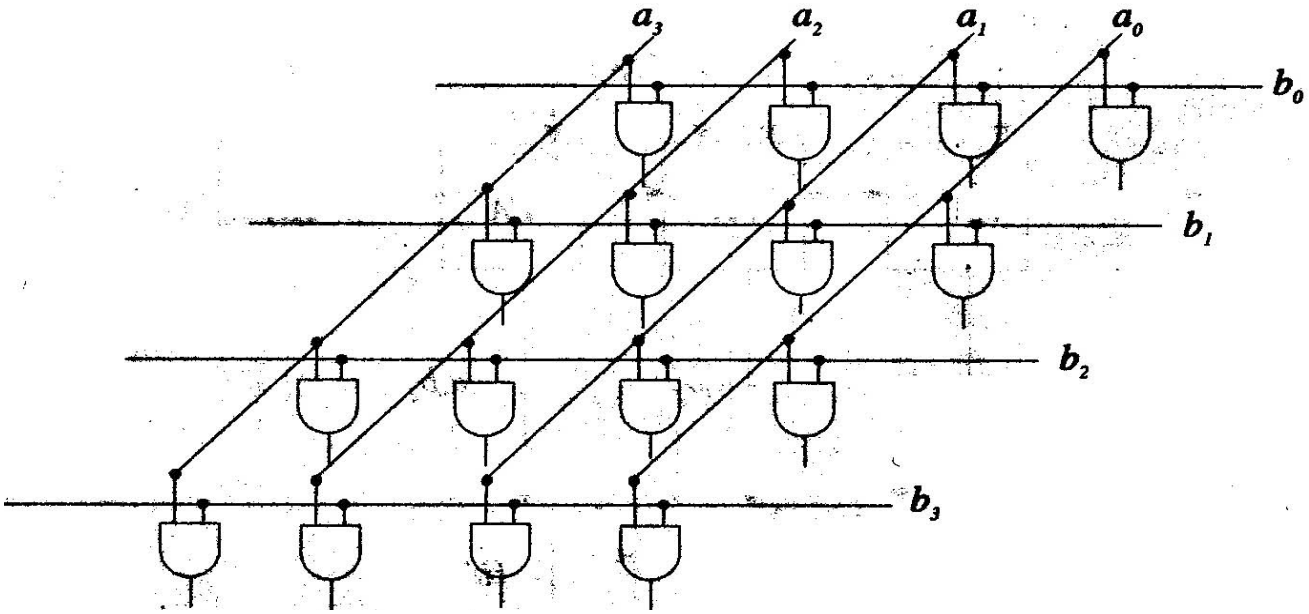
- Come fare il calcolo con circuiti logici ?
  - Possiamo scomporre l'operazione in due stadi:
  - Primo stadio: prodotti parziali
    - si mette in AND ciascun bit del moltiplicatore con i bit corrispondenti del moltiplicando
  - Secondo stadio: somme
    - si effettuano le somme (full adder) dei bit sulle righe contenenti i prodotti parziali

La matrice dei prodotti parziali

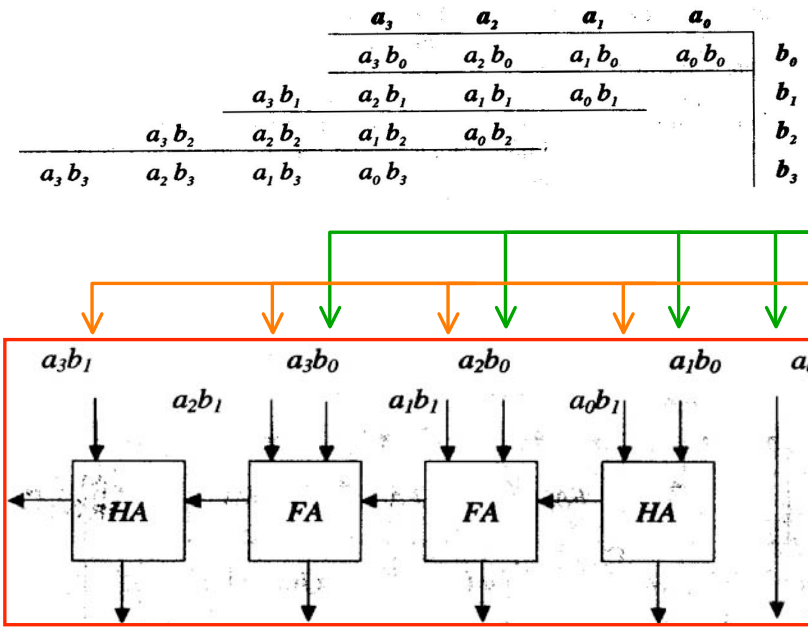


*In binario i prodotti parziali sono degli **AND***

Il circuito che effettua i prodotti

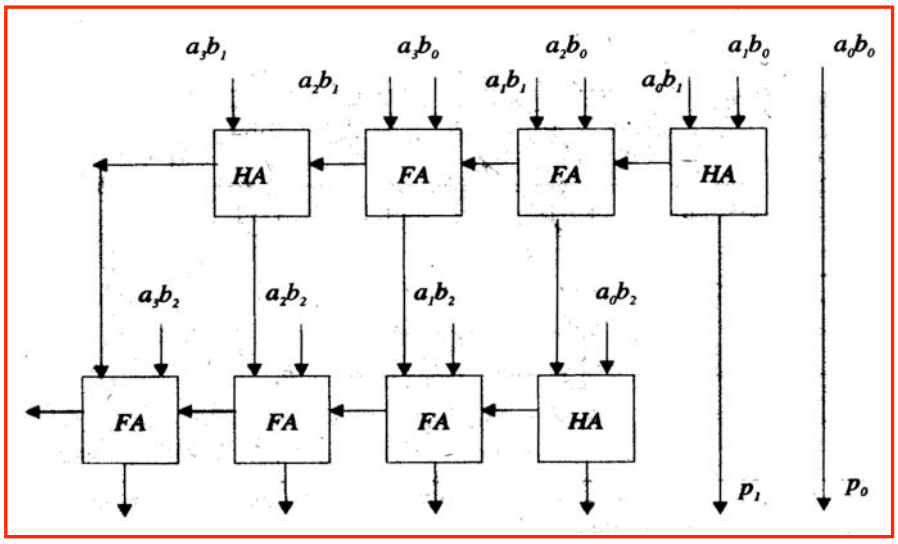


Somma – prime 2 righe dei prodotti parziali




```
      1 0 1 1   x
      0 1 1 1   =
  ---------------
        1 1 1
      1 0 1 1   +
    1 0 1 1 -
  ---------------
      1
    1 0 0 0 0 1 +
    1 0 1 1 - -
  ---------------
  1 1 0 1 1 0 1
```

Somma della terza riga



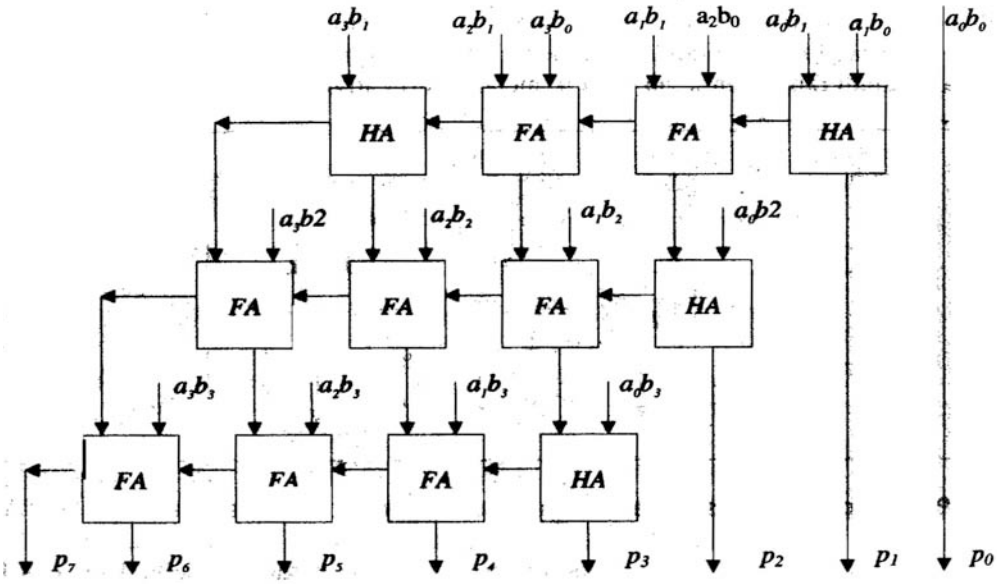

```
      1 1 0 1 1  x
        1 1 1  =

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

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

  ---------------
  1 0 1 1 1 1 0 1
```

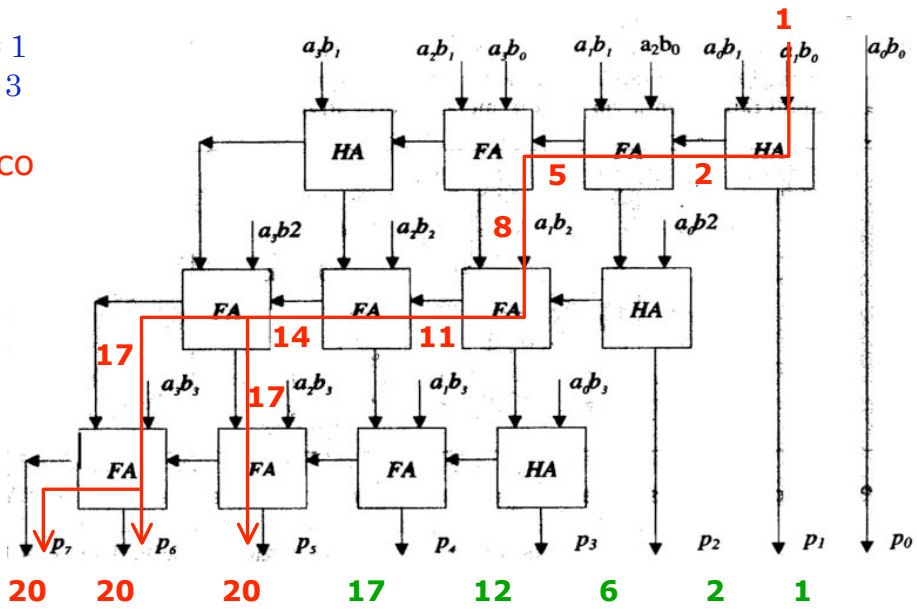

Somma prodotti parziali – circuito completo



Overflow: A e B su: **N bit** → P su: **2N bit**

Valutazione del cammino critico

- **N = 4**
  - Ritardo AND = 1
  - Ritardo HA = 1
  - Ritardo FA = 3
- Cammino critico = 20



Sommario

- Sommatori ad anticipazione di riporto
- Addizionatori modulari
- Moltiplicatori – approccio hardware
- Moltiplicatori – approccio firmware

Approcci tecnologici alla ALU

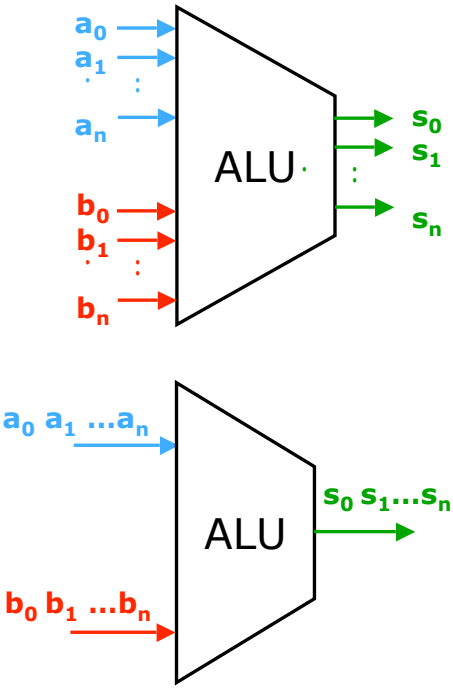
Approcci tecnologici alla costruzione di ALU:

- Hardware
  - Ad ogni operazione corrisponde un circuito combinatorio specifico
- Firmware o microprogrammato
  - Circuiti specifici solamente per alcune operazioni elementari
  - Le operazioni complesse vengono sintetizzate mediante combinazione di operazioni elementari, eseguendo uno specifico algoritmo (implementato mediante microprogrammazione)
  - Approccio *"controllore-datapath"*
    - La ALU contiene una unità di controllo e dei registri

- FIRMWARE vs. HARDWARE:
  - Hardware: più veloce ma più costosa per complessità dei circuiti
    - La soluzione HW conviene per le operazioni frequenti
  - Firmware: meno veloce ma più flessibile. Potenzialmente adatta a modificare o inserire nuove procedure.

Presentazione seriale/parallela

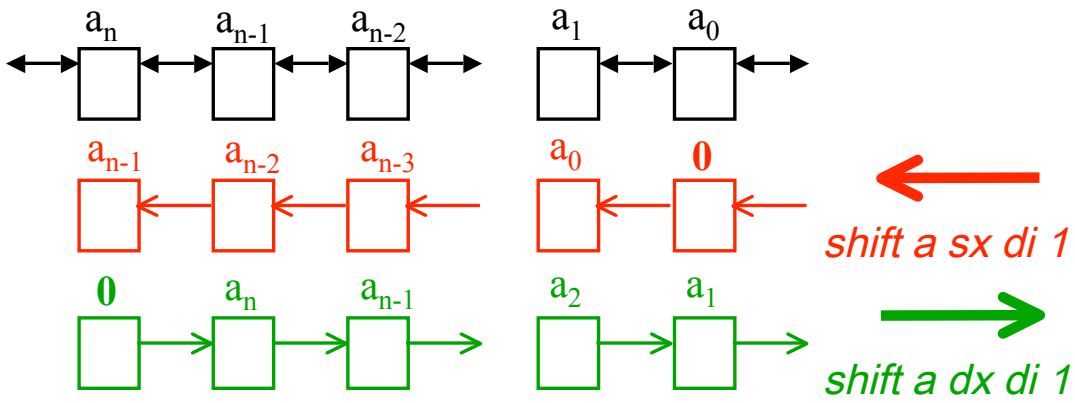
- Presentazione parallela
  - Tutti i bit vengono presentati simultaneamente;
    La ALU genera tutti i bit del risultato
    - Tipico dell'approccio HW
- Presentazione seriale
  - Gli operandi vengono presentati 1 bit alla volta, del quale viene eseguita l'operazione e fornito il risultato
    - Viene utilizzata un'unica copia del circuito aritmetico
    - Il tempo di calcolo cresce linearmente con il numero di bit
    - non si possono utilizzare tecniche di carry look-ahead
- Presentazione seriale a gruppi
  - I bit vengono presentati a gruppi di k bit (es. k=8: bytes)



Operazione di SHIFT (scorrimento)

Dato $A=\{a_n\ a_{n-1} \dots a_1\ a_0\} \rightarrow s_k(A)=A',\ a'_j = a_{j-k}$

- **k**: *shift amount*
  - Tempo comparabile con quello della somma
  - Effettuato al di fuori delle operazioni selezionate dal MUX della ALU, da un circuito denominato **Barrel Shifter**



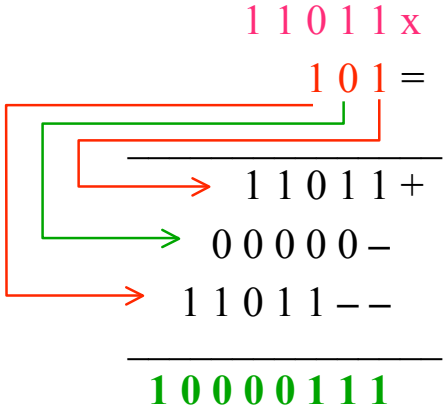
La moltiplicazione firmware

Algoritmo della moltiplicazione – firmware

Si analizzano sequenzialmente i bit del moltiplicatore

se il bit è = 1
→ moltiplicando in posizione opportuna

se il bit è = 0
→ 0 in posizione opportuna

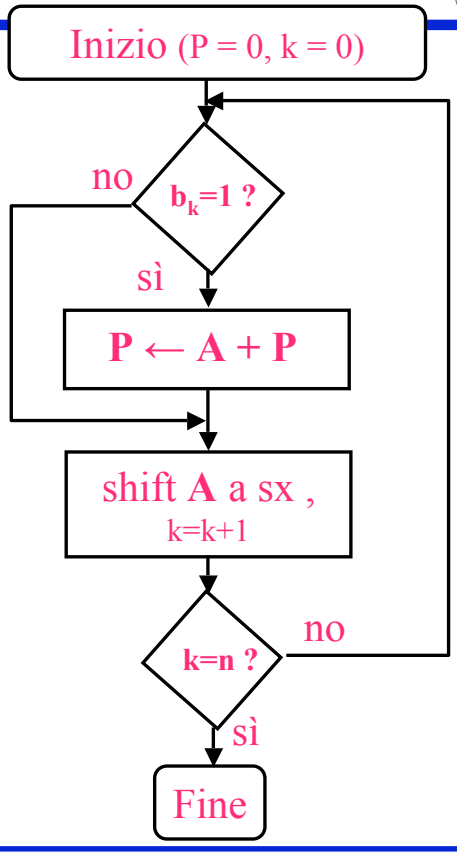


- Utilizzo un registro prodotto da **2n** bit, inizializzato a 0
- Ciclo iterativo: per ogni bit del moltiplicatore
  - Se bit = 1 → sommo il moltiplicando al prodotto
  - shift a SX di 1 bit del moltiplicando

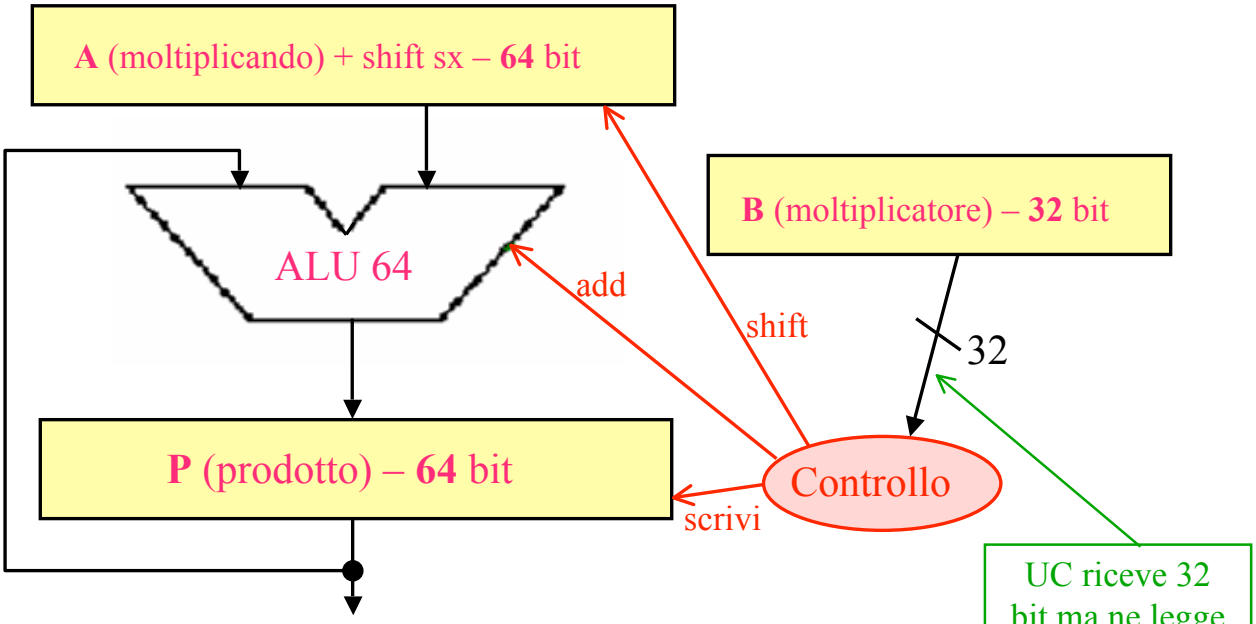
```
      1 1 0 1 1 x
        1 1 1 =
-------------------
    1 1 1 1 1
      1 1 0 1 1 +
    1 1 0 1 1 0
-------------------
  1
    1 0 1 0 0 0 1 +
    1 1 0 1 1 0 0
-------------------
  1 0 1 1 1 1 0 1
```



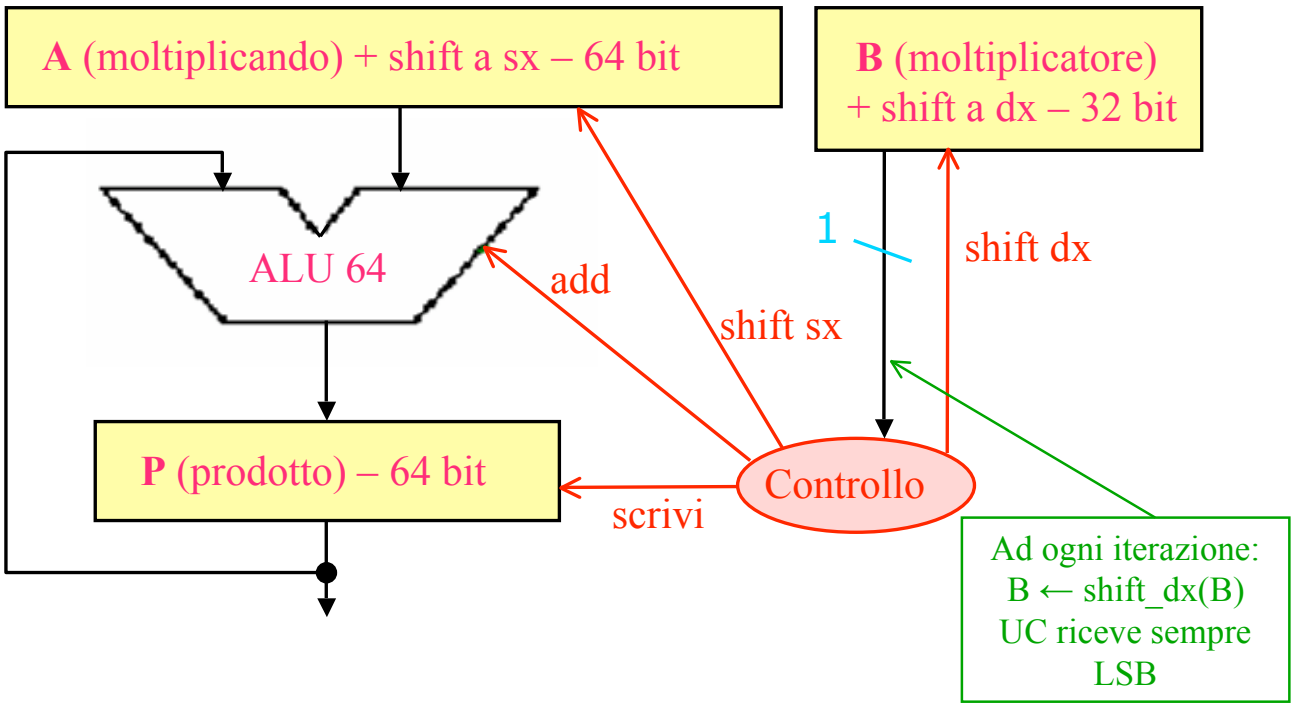
```
A ──→ 1 1 0 1 1 x
B ──→   1 1 1 =
-------------------
    1 1 1 1 1
      1 1 0 1 1 +
    1 1 0 1 1 0
-------------------
  1
    1 0 1 0 0 0 1 +
    1 1 0 1 1 0 0
-------------------
P ──→ 1 0 1 1 1 1 0 1
```



Implementazione circuitale



Implementazione circuitale – modifica



Esempio

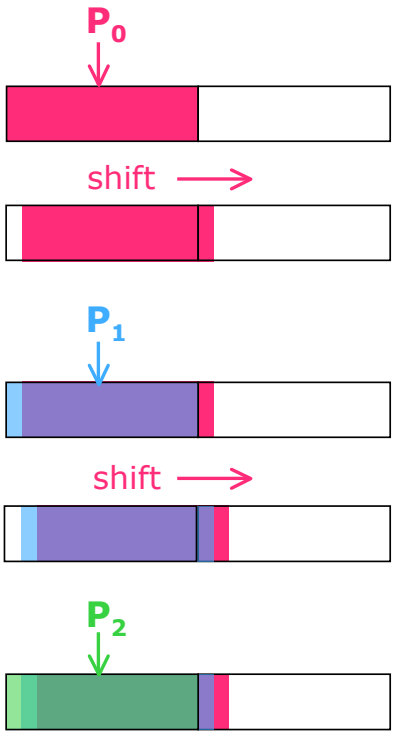
A,B: 4 bit – P: 8 bit

2 x 3 = 6 ↔ 0010 x 0011 = 0110

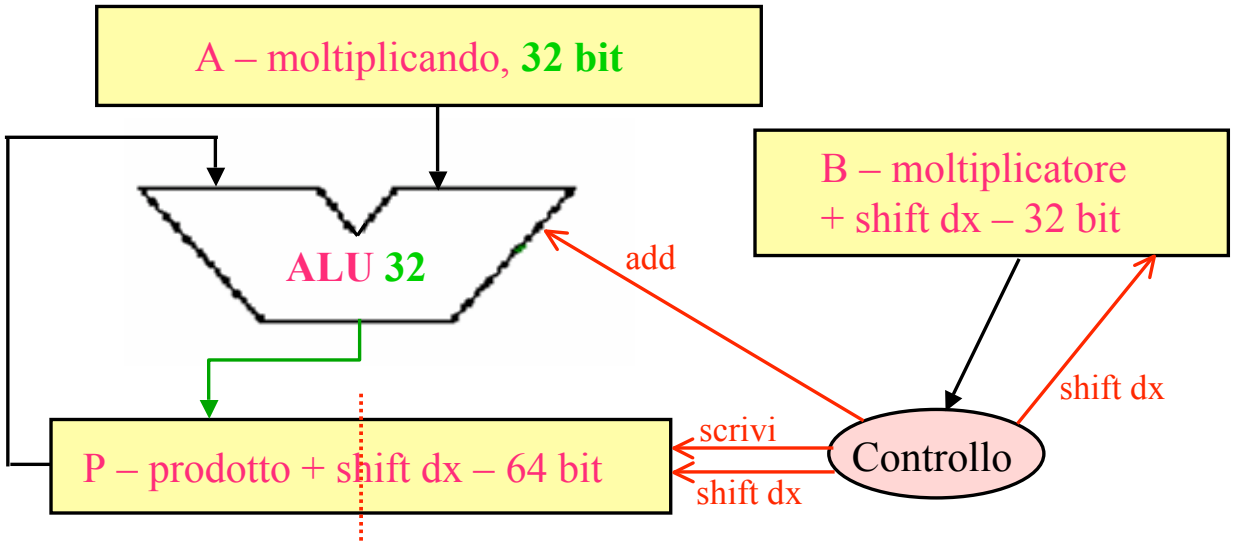
Iterazione	Passo	Moltiplicatore	Moltiplicando	Prodotto
0	Valori iniziali	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcando	0011	0000 0010	0000 0010
	2: Scala a sinistra Moltiplicando	0011	0000 0100	0000 0010
	3: Scala a destra Moltiplicatore	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcando	0001	**0000 0100**	0000 0110
	2: Scala a sinistra Moltiplicando	0001	0000 1000	0000 0110
	3: Scala a destra Moltiplicatore	0000	0000 1000	0000 0110
3	1: 0 ⇒ Nessuna operazione	0000	0000 1000	0000 0110
	2: Scala a sinistra Moltiplicando	0000	0001 0000	0000 0110
	3: Scala a destra Moltiplicatore	0000	0001 0000	0000 0110
4	1: 0 ⇒ Nessuna operazione	0000	0001 0000	0000 0110
	2: Scala a sinistra Moltiplicando	0000	0010 0000	0000 0110
	3: Scala a destra Moltiplicatore	0000	0010 0000	0000 0110

Implementazione alternativa: idea

- Soltanto metà dei bit del registro moltiplicando vengono utilizzati ad ogni iterazione
  - Ad ogni iterazione si aggiunge 1 bit al registro prodotto
- IDEA
  - Si caricano i risultati parziali in **P** nella metà SINISTRA
  - Si sposta la somma dei prodotti parziali (in **P**) verso **destra** ad ogni iterazione



Seconda implementazione



Implementazione ottimizzata

Numero di bit del prodotto corrente

+

Numero di bit da esaminare di B

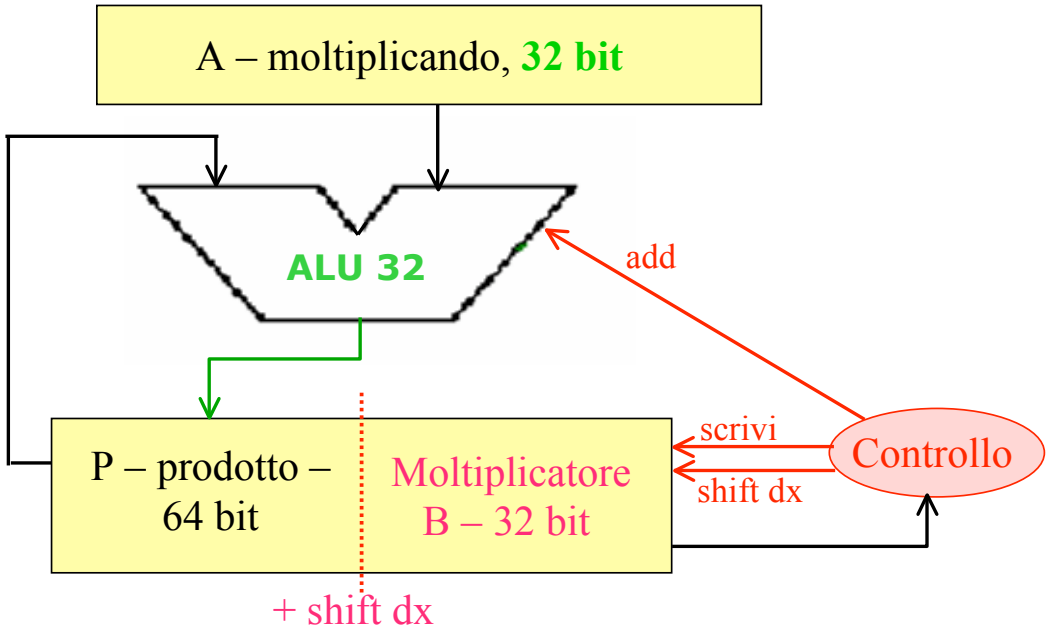
=

64 bit: costante ad ogni iterazione

elimino il registro **moltiplicatore: B**

```
  1 1 0 1 1 x
      1 1 1 =
 ---------------
    1 1 1 1 1
      1 1 0 1 1 +
    1 1 0 1 1 -
 ---------------
  1
    1 0 1 0 0 0 1 +
    1 1 0 1 1 - -
 ---------------
  1 0 1 1 1 1 0 1
```

- Situazione alla prima iterazione

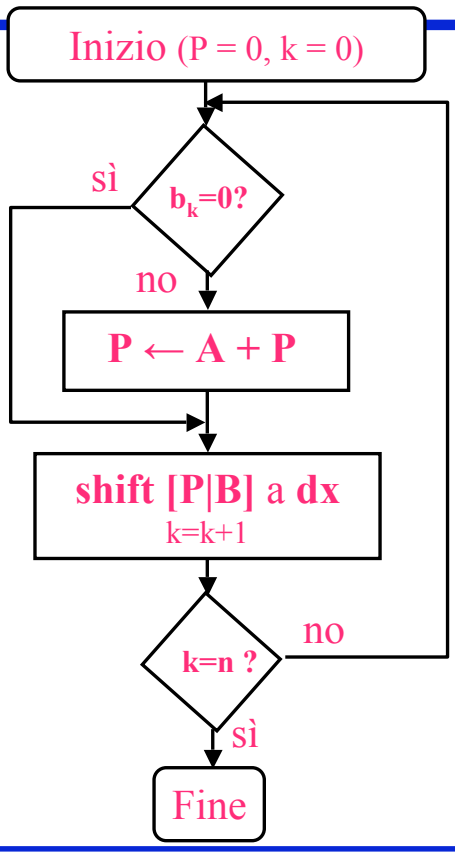


A → 1 1 0 1 1 x
B → 1 1 1 =

1 1 1 1 1
1 1 0 1 1 +
1 1 0 1 1 **0**

1
1 0 1 0 0 0 1 +
1 1 0 1 1 **0 0**

P → 1 0 1 1 1 1 0 1



Esempio di esecuzione alg. ottimizzato

A, B: 4 bit – P: 8 bit

2 x 3 = 6 ↔ 0010 x 0011 = 0000 0110

Iterazione	Passo	Moltiplicando	Prodotto
0	Valori iniziali	0010	0000 0011
1	1a: 1 ⇒ Prod = Prod + Mcando	0010	0010 0011
	2: Scala a destra Prodotto	0010	0001 0001
2	1a: 1 ⇒ Prod = Prod + Mcando	0010	0011 0001
	2: Scala a destra Prodotto	0010	0001 1000
3	1: 0 ⇒ Nessuna operazione	0010	0001 1000
	2: Scala a destra Prodotto	0010	0000 1100
4	1: 0 ⇒ Nessuna operazione	0010	0000 1100
	2: Scala a destra Prodotto	0010	0000 0110

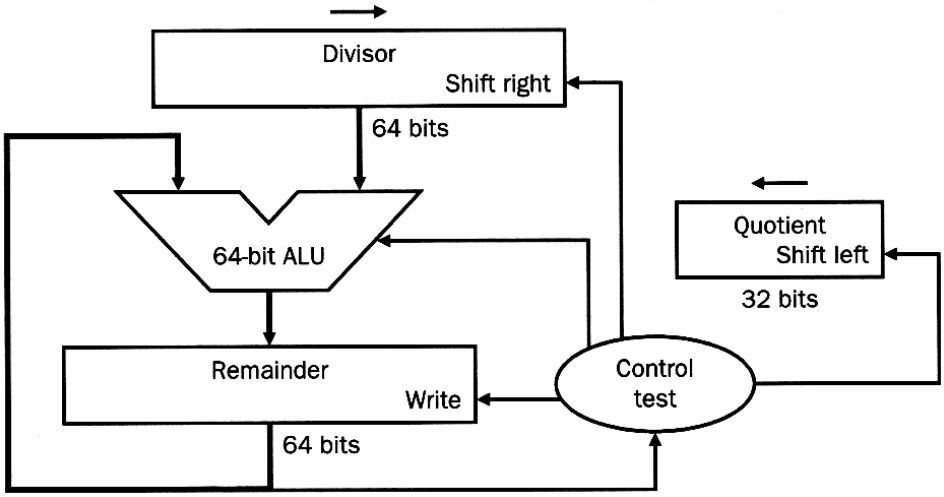
Divisione intera - n bit (unsigned)

- Algoritmo: n+1 iterazioni
  1. *K = n*
  2. *Sottraggo il divisore moltiplicato per 2^K dal dividendo*
  3. *Se differenza > 0 ➔ dividendo = dividendo – divisore x 2^K*
  4. *K = K – 1*
  5. *Se K > 0, torna a 2 altrimenti FINE*

7 : 2 = 3, resto 1 ➔ 0111 : 0010 = 0011, resto 0001

```
00000111 -
00100000 =
bit4 quoz.: 0
```
➔
```
00000111 -
0010000 =
bit3 quoz.: 0
```
➔
```
00000111 -
001000 =
bit2 quoz.: 0
```
➔
```
00000111 -
00100 =
bit1 quoz.: 1
```
➔
```
00000011 -
0010 =
bit0 quoz.: 1
```
➔
```
quoz: 00011
resto: 1
```

- Dividendo, divisore, quoziente, resto: 32 bit
  - Remainder: inizializzato con divisore x 2^{32} ➔ servono 64 bit



- Esempio di funzionamento:

divisore x 2^4

dividendo

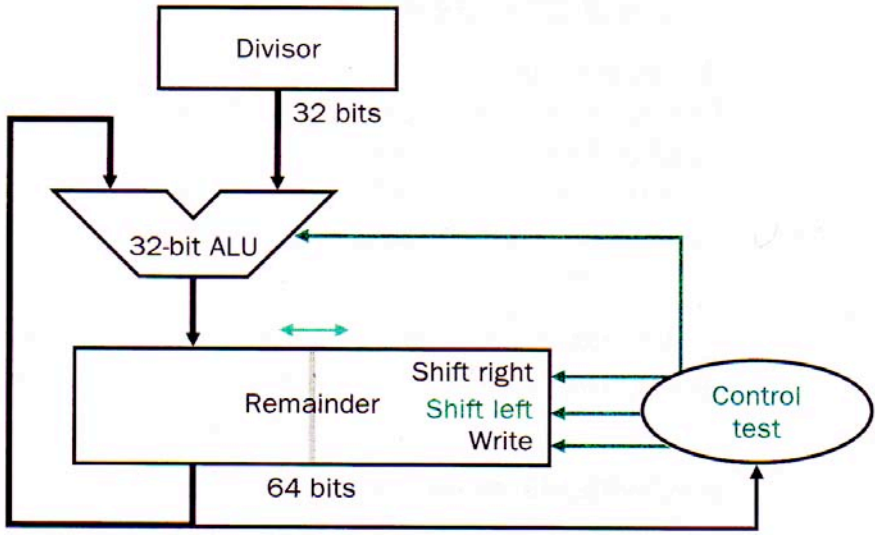
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	(1)110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	(1)111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	(1)111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	(0)000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	(0)000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

resto

quoziente

Divisione intera (unsigned) : n=32 bit

- Progetto ottimizzato:
  - Allineo il dividendo senza shiftare il divisore ➔ ALU a soli 32 bit
  - *Se differenza < 0* ➔ *ripristina di Remainder (+= divisor), SLL remainder, $R_0 = 0$*
  - *altrimenti:* ➔ *SLL remainder, $R_0 = 1$*



Divisione intera (unsigned) : n=32 bit

- Esempio, struttura ottimizzata:

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0 ⟹ + Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0 ⟹ + Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem ≥ 0 ⟹ sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem ≥ 0 ⟹ sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

divisore

dividendo

resto

quoziente