MOLTIPLICATORE BINARIO

Andrea Terenziani – Matr. 128590

Serena Passini - Matr. 136416

L'OPERAZIONE DI MOTIPLICAZIONE

			1	1	0	1	$\times$
				1	0	1	$=$
			1	1	0	1	$+$
		0	0	0	0		$+$
	1	1	0	1			$=$
1	0	0	0	0	0	1	

COSA VUOL DIRE "MOLTIPLICARE"

$$a \times b = \underbrace{b + \cdots + b}_{a\ volte} = \sum_{i=1}^{a} b$$

a : *moltiplicando*

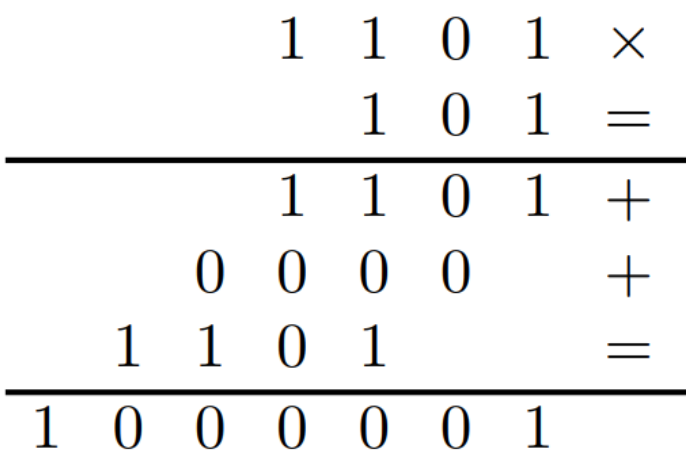
b : *moltiplicatore*

$a \times b$: *prodotto*

Per il resto della presentazione si assumerà che a e b siano valori *binari* da n bit

CALCOLARE IL PRODOTTO

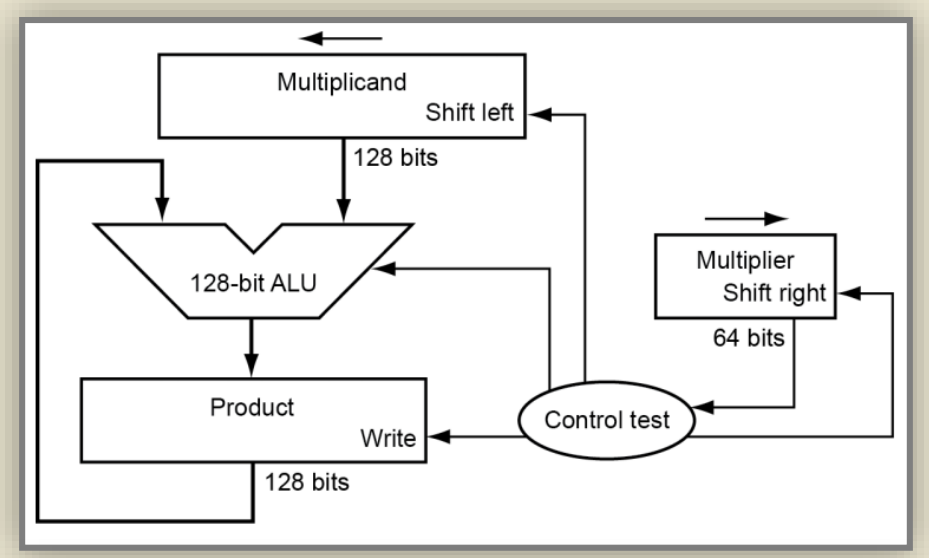
- Il prodotto è calcolato come *somma progressiva di prodotti parziali.*
- Per ottenere il *j*-esimo :
  1. si moltiplica *il moltiplicando* per *la cifra j-esima* del moltiplicatore (partendo da destra)
  2. viene shiftato tale valore 1 posizione più a sinistra del prodotto parziale precedente.
- Il risultato finale è la somma di questi valori

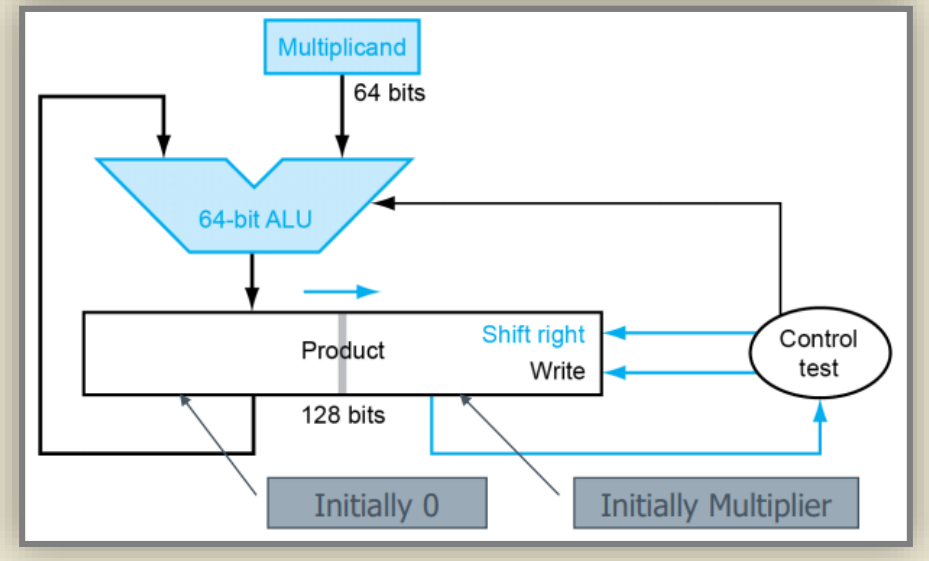


OSSERVAZIONI

$$
\begin{array}{rrrrrrr}
 & & & 1 & 1 & 0 & 1 \times \\
 & & & & 1 & 0 & 1 = \\
\hline
 & & & 1 & 1 & 0 & 1 + (1101 \times 1) \\
 & & 0 & 0 & 0 & 0 & + (1101 \times 0) \\
 & 1 & 1 & 0 & 1 & & = (1101 \times 1) \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

1. Gli operandi sono *binari*, per cui un prodotto parziale (prima dello shifting) può assumere solo due valori :
   - *moltiplicatore* × 1 = *moltiplicatore*
   - *moltiplicatore* × 0 = 0
2. La dimensione (in bit) del prodotto *può essere molto maggiore di quella dei due operandi* → *alto rischio di overflow*

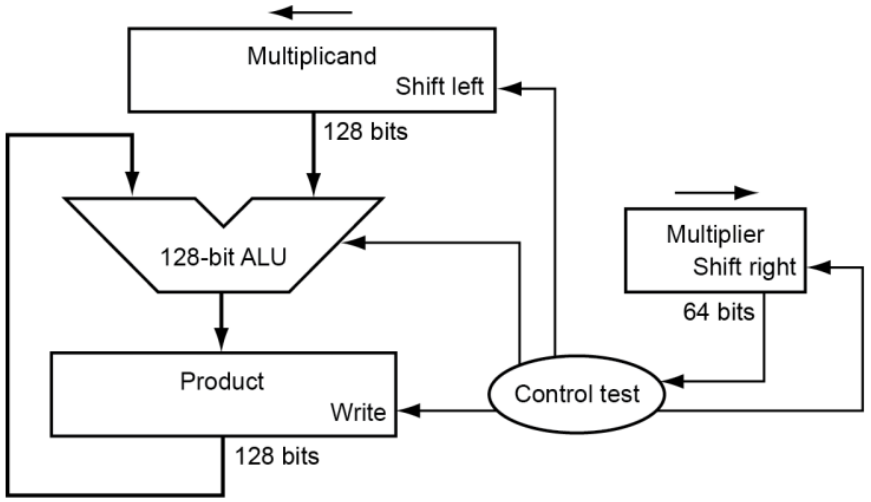


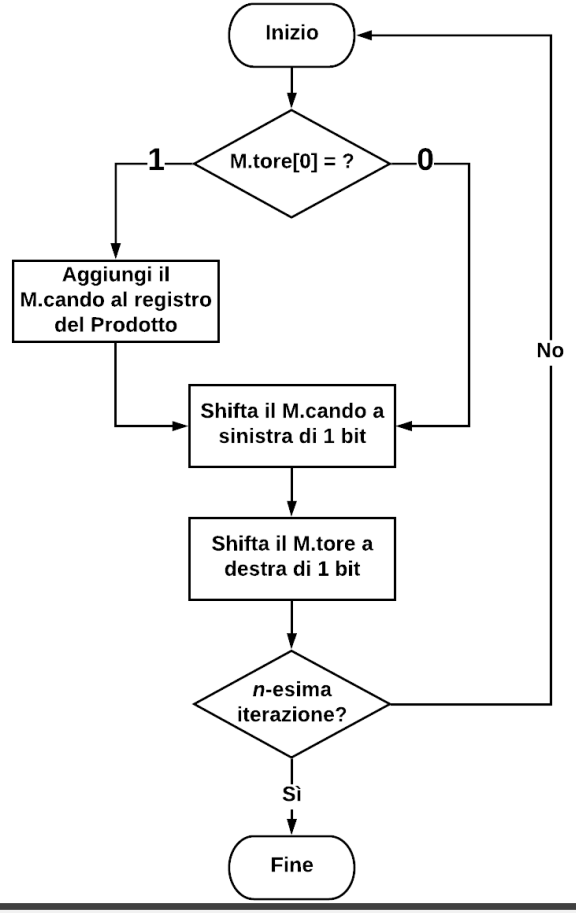


IMPLEMENTARE L'OPERAZIONE

I° IMPLEMENTAZIONE

- **Componenti** :
  - Un registro da *2n* bit per contenere il moltiplicando (inizialmente nei primi *n* bit)
  - Un registro da *2n* bit per contenere la somma dei prodotti parziali inizializzato a 0
  - Un registro da *n* bit per contenere il moltiplicatore
  - Una ALU da *2n* bit per calcolare ogni prodotto parziale e una Control Unit per regolare l'esecuzione



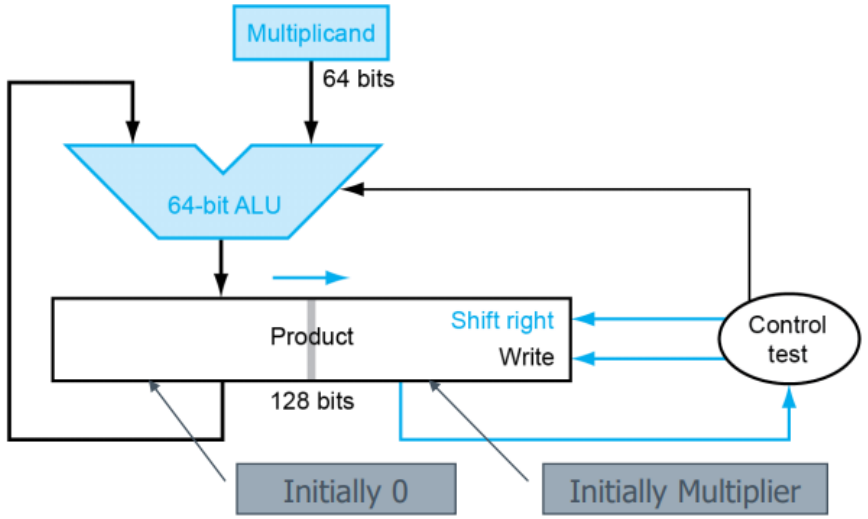


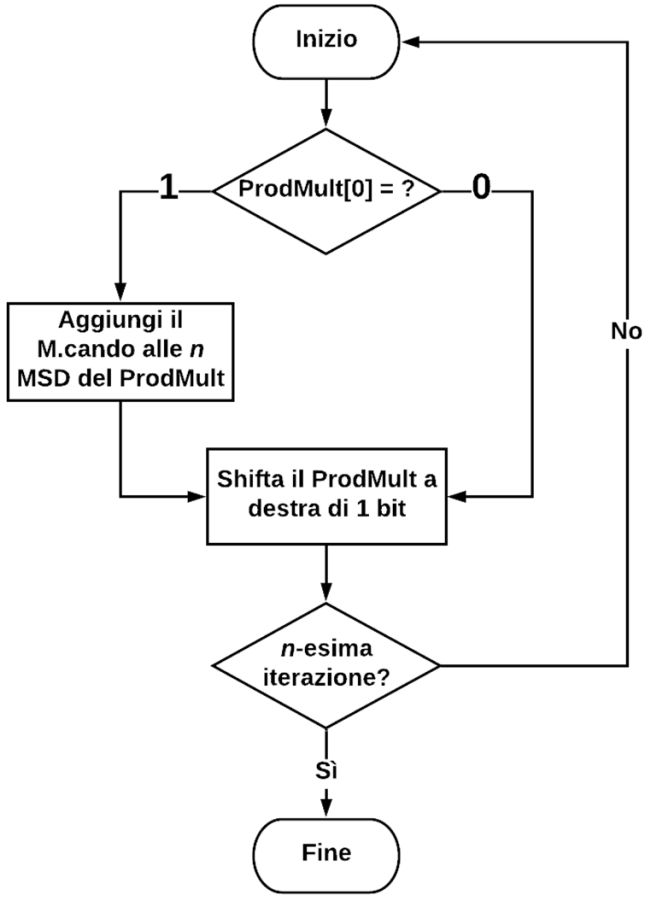
- **Algoritmo** :
  A. Se *l'ultima cifra del Moltiplicatore è 1*, si aggiunge il Moltiplicando al registro del Prodotto
  B. Si shifta il Moltiplicando a sinistra di 1 bit
  C. Si shifta il Moltiplicatore a destra di 1 bit
  D. Se *non siamo alla n-esima iterazione*, ripetiamo dal punto (A)
- Il conteggio delle iterazioni è *responsabilità della Control Unit*

- ***Cosa possiamo migliorare*** :
  - La somma tra Moltiplicando e Prodotto e i due shifting *sono eseguiti ciascuno in un ciclo di clock* su "ordine" della Control Unit → *le n iterazioni impiegano 3n clock cycles*
  - Vengono usati 3 registri, che memorizzano complessivamente *5n* bit di informazione, ma questa non è la minima memoria necessaria

2° IMPLEMENTAZIONE

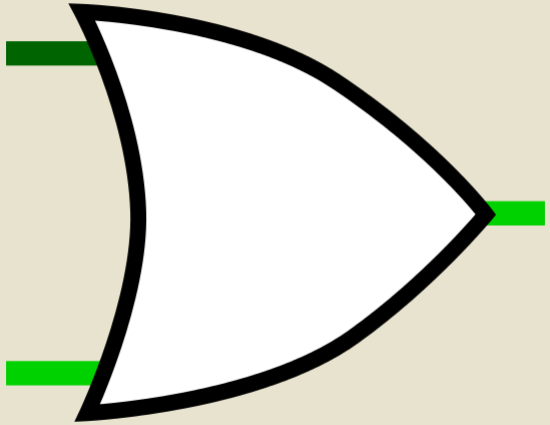
- **Componenti** :
  - Un registro da *n* bit per contenere il moltiplicando
  - Un registro da *2n* bit per contenere *sia la somma dei prodotti parziali* (nei bit *più* significativi) che *il valore del moltiplicatore* (nei bit meno significative), detto "*registro PM*"
  - Una ALU da *n* bit per calcolare ogni prodotto parziale e una Control Unit per regolare l'esecuzione

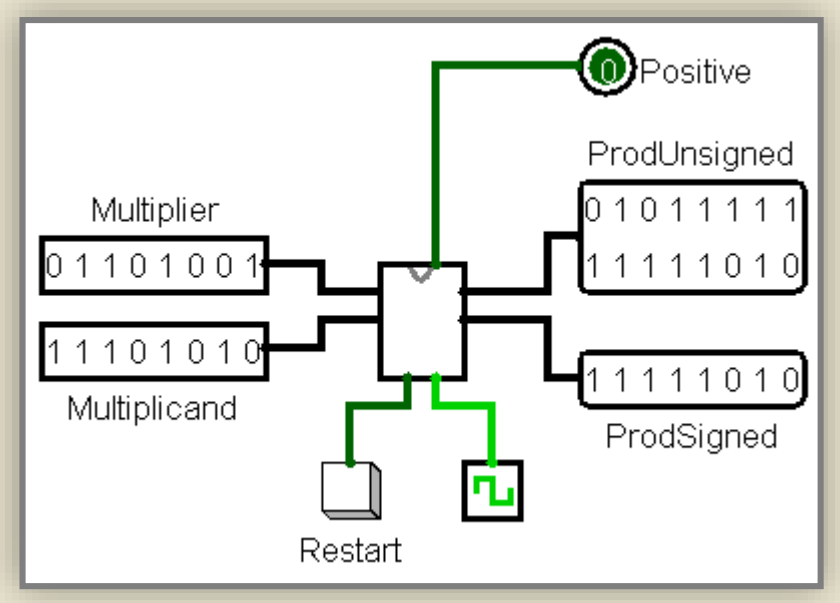




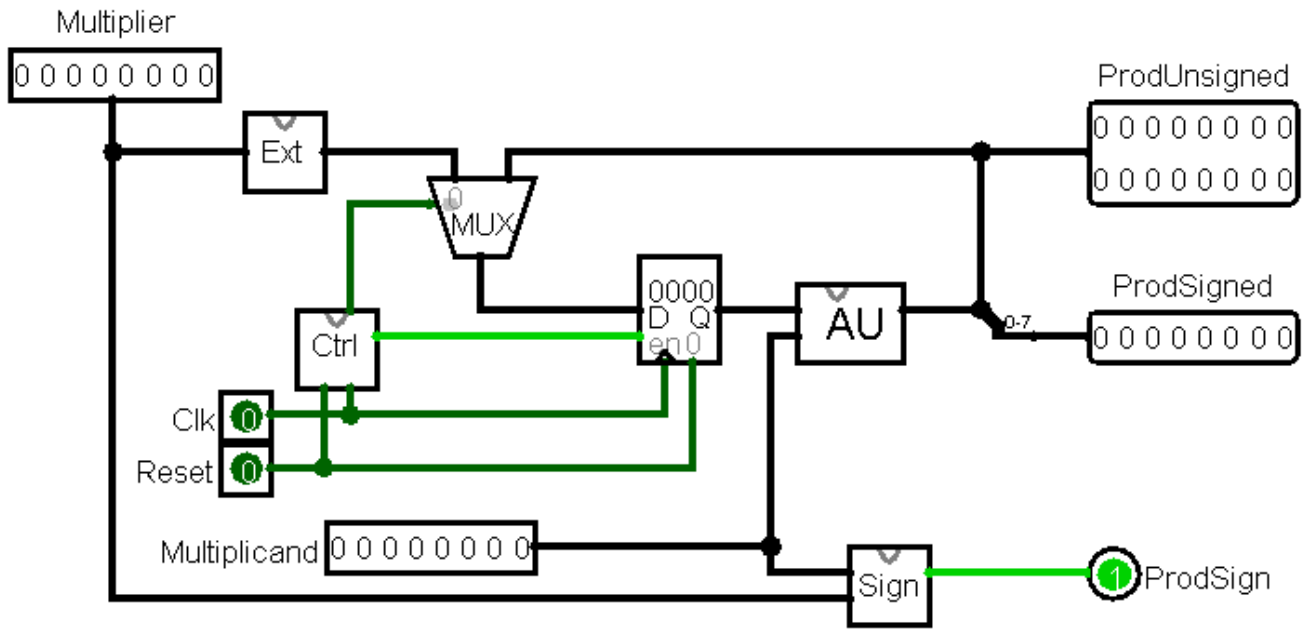
- **Nuovo algoritmo** :
  A. Se *l'ultima cifra del Registro PM* è *1*, si aggiunge il Moltiplicando *alle n cifre più significative del PM*
  B. Si shifta il Registro PM a destra di 1 bit. Questo "*incorpora" sia lo shift a destra del Moltiplicatore che quello a sinistra del Moltiplicando*
  C. Se *non siamo alla n-esima iterazione*, ripetiamo dal punto (A)

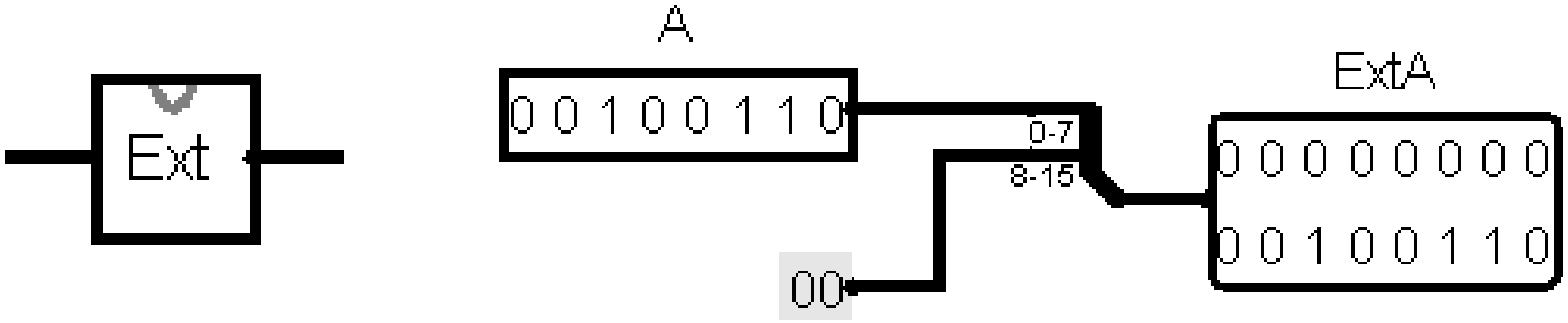
REALIZZAZIONE IN LOGISIM



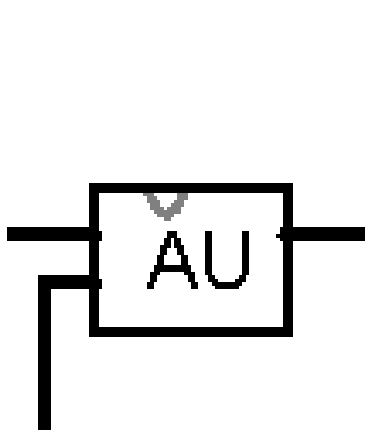


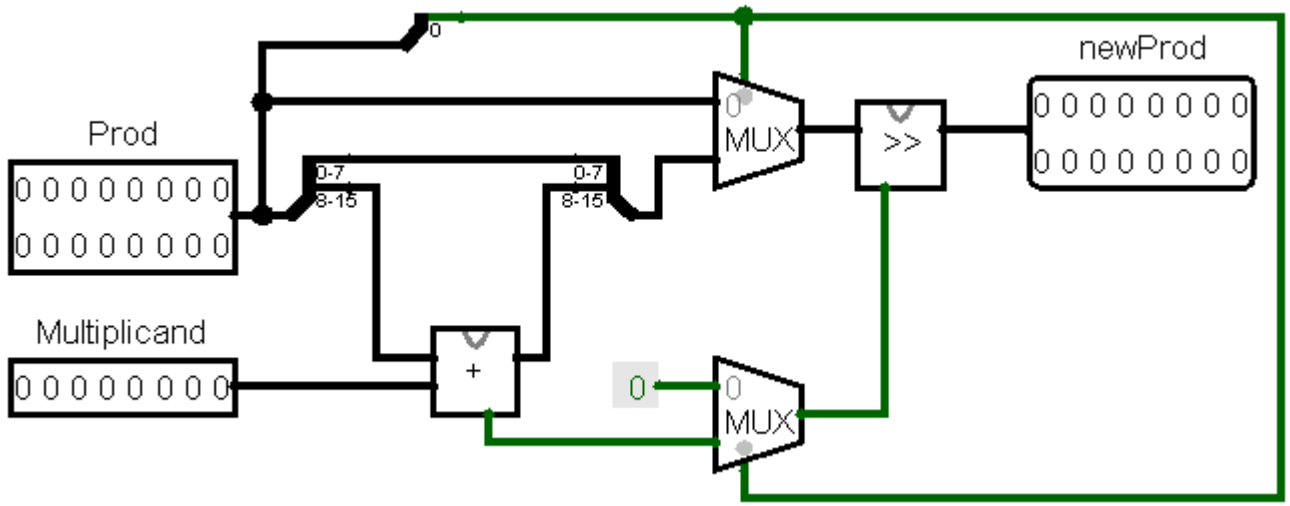
CIRCUITO COMPLETO



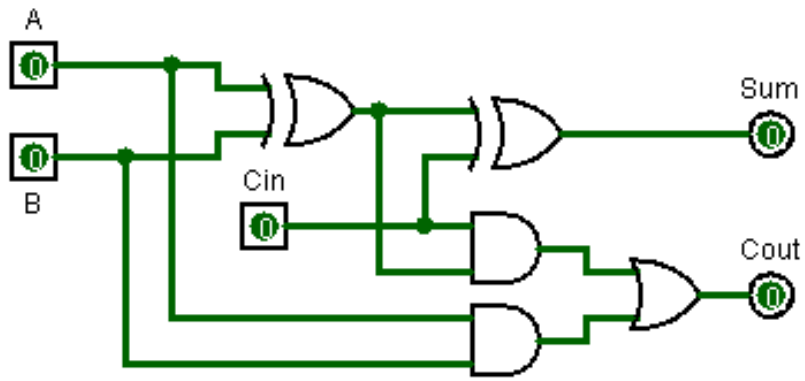


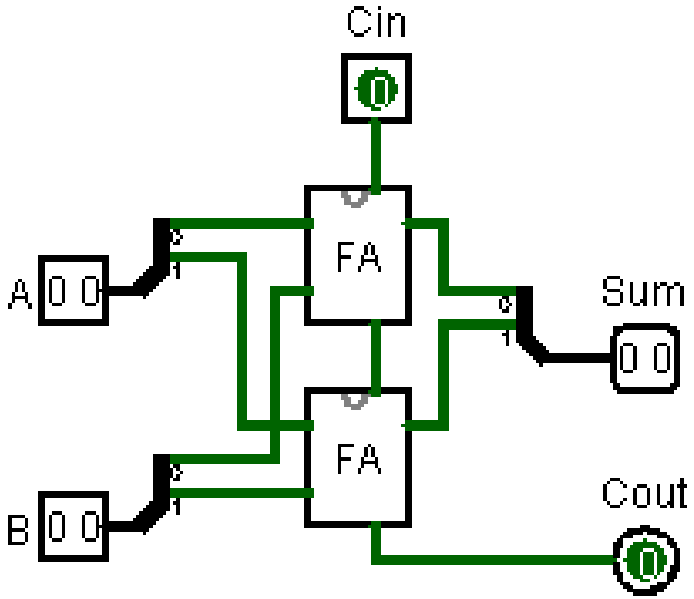
BIT EXTENDER

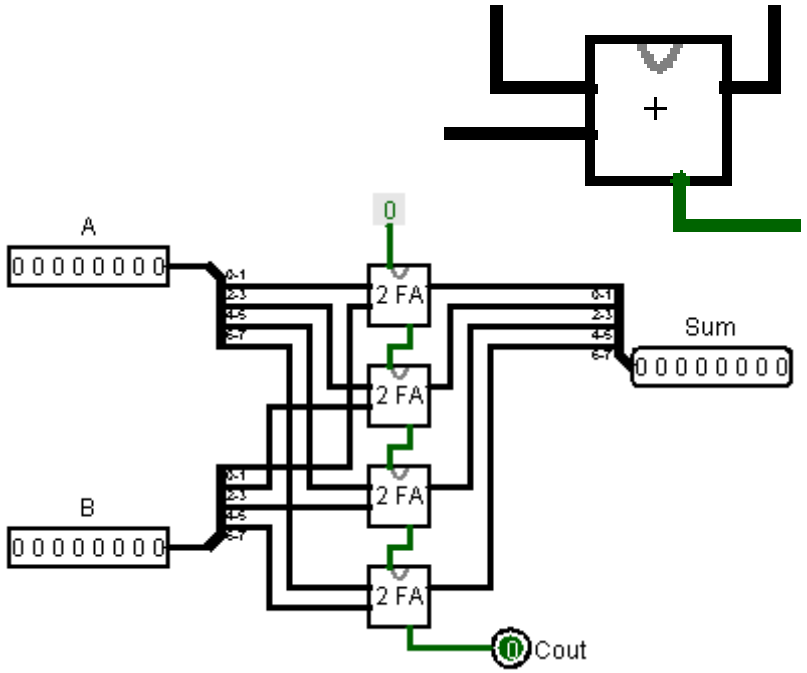




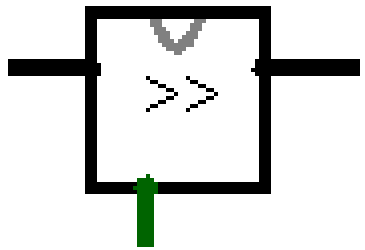
UNITÀ ARITMETICA

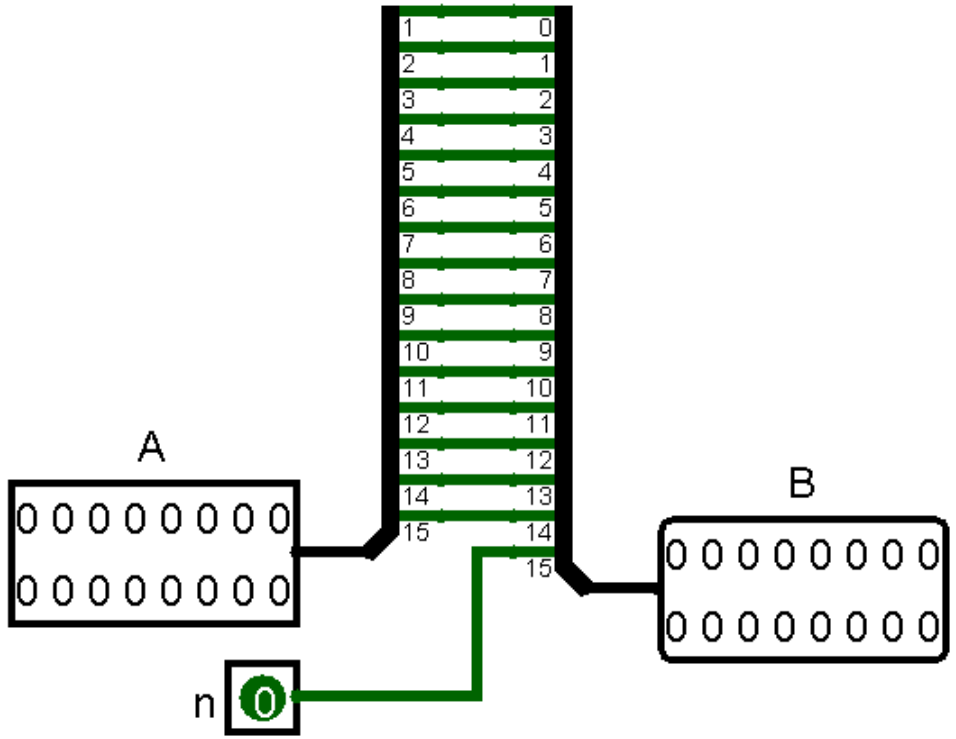

A
B
Cin
Sum
Cout


Cin
A
0 0
FA
FA
Sum
0 0
B
0 0
Cout

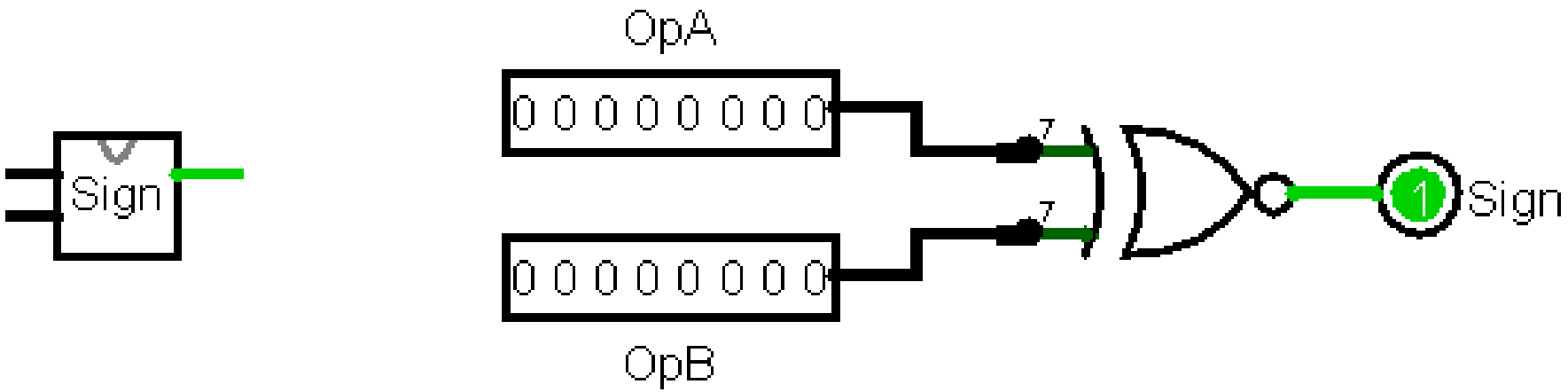

+
A
0 0 0 0 0 0 0 0
0
2 FA
2 FA
2 FA
2 FA
Sum
0 0 0 0 0 0 0 0
B
0 0 0 0 0 0 0 0
Cout

ADDER

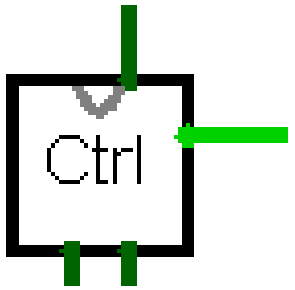

>>

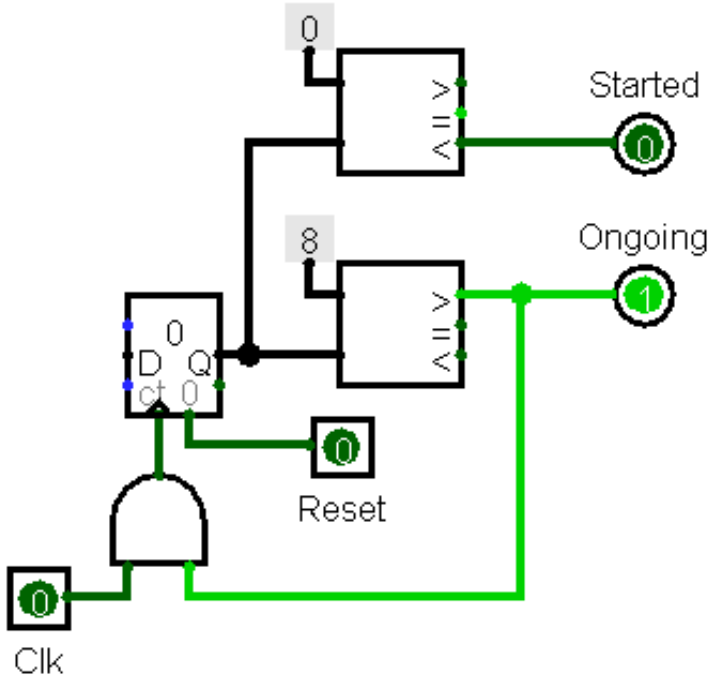

A
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
n
0
B
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 0
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
10 9
11 10
12 11
13 12
14 13
15 14
15

BIT SHIFTER



SEGNO

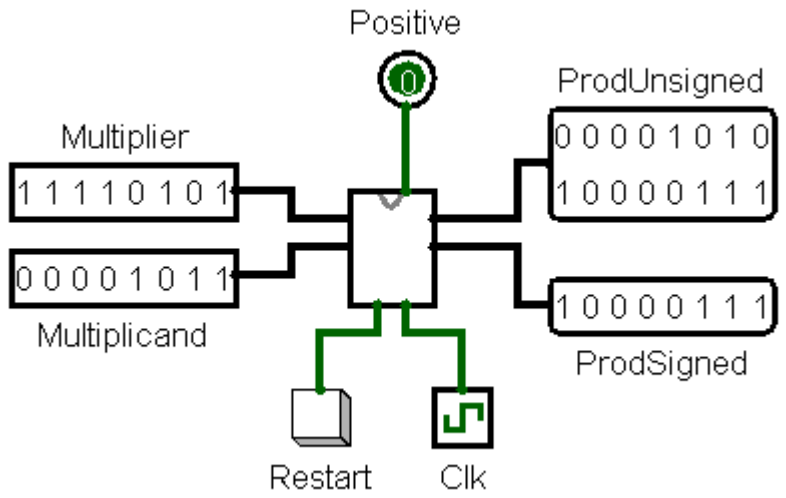




UNITÀ DI CONTROLLO

MODALITÀ D'USO

- ***Input*** :
  - *Multiplier* / *Multiplicand* (gli operandi)
  - *Restart* (segnale di reset – indica se ripetere l'operazione)
  - *Clk* (segnale di clock per scandire l'esecuzione)
- ***Output*** :
  - *ProdUnsigned* (il prodotto *considerando gli input come unsigned*)
  - *ProdSigned* (il prodotto *considerando gli input come signed)*



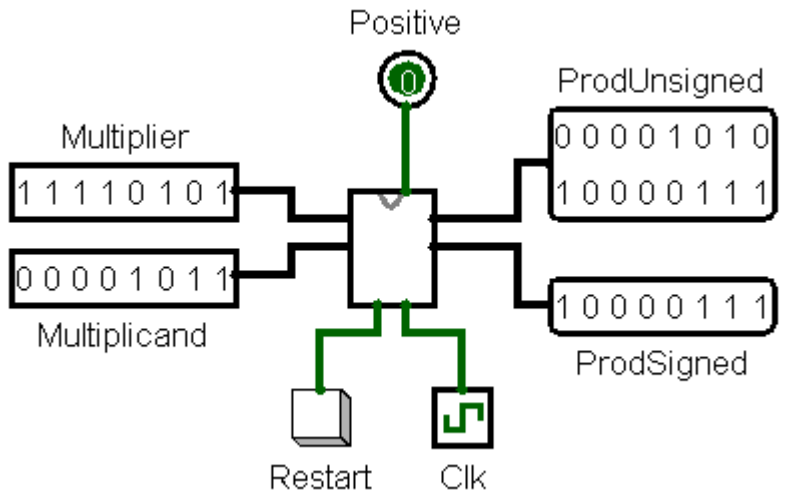
LIMITAZIONI

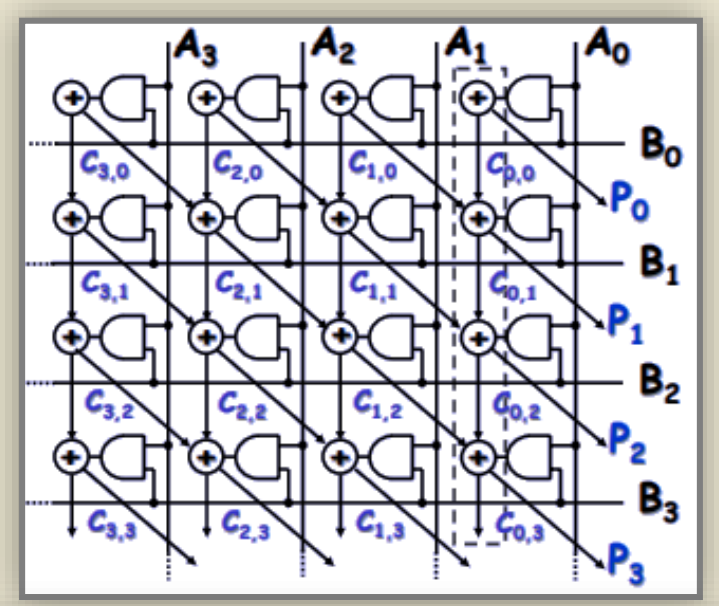
- ***Prodotti con segno***

  Questo moltiplicatore può calcolare un prodotto con *segno*, con valore *limitato tra -127 e 127*. Avendo input da 8 bit, però, *si ha un rischio di overflow*

- ***Prodotti senza segno***

  Se si considerano i due operandi come *unsigned da 8 bit*, allora il loro prodotto sarà *un valore da 16 bit*, contenuto *all'interno del Registro PM*.





$1101_2 \cdot 0110_2 = 01001110_2$

1101	0011	00100111
1101	0100	00110100
1101	0101	01000001
1101	0110	01001110
1101	0111	01011011

ALTRI ESEMPI DI MOLTIPLICATORI

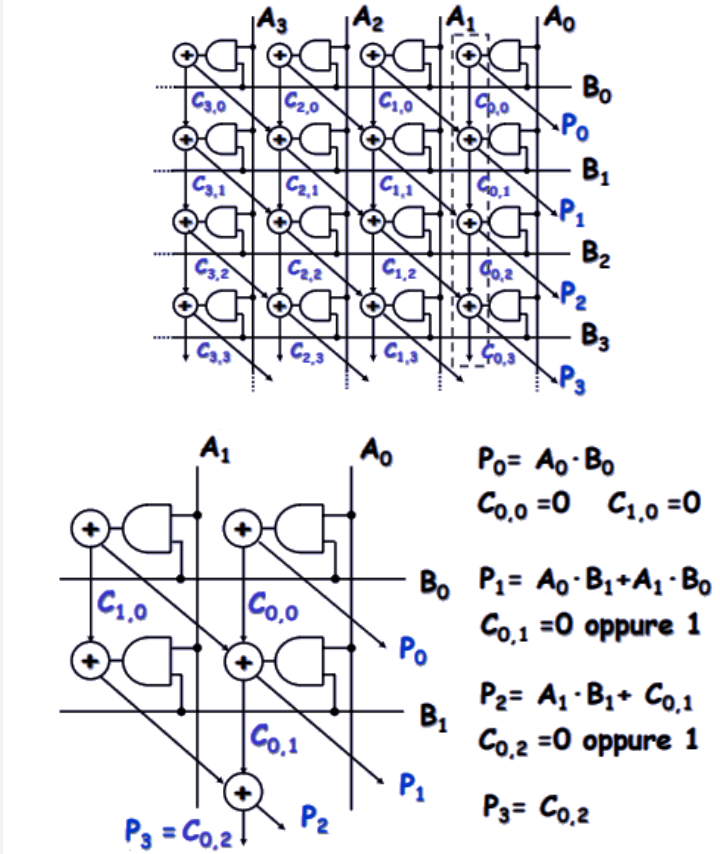
$1101_2 \cdot 0110_2 = 01001110_2$

1101	0011	00100111
1101	0100	00110100
1101	0101	01000001
1101	0110	01001110
1101	0111	01011011

MOLTIPLICATORI LOOK-UP TABLE

- Fanno uso di circuiti di memoria dove sono immagazzinati i *valori pre-calcolati di tutti i possibili prodotti* di due numeri a n bit (unsigned)
- Calcolare il prodotto significa quindi *trovare semplicemente il valore associato ai due operandi*
- La crescita della dimensione della memoria è *esponenziale* con il numero di bit n degli operandi.

MOLTIPLICATORI A MATRICE



- Consiste nella generalizzazione a n bit del circuito combinatorio che per la moltiplicazione di due numeri a 1 bit. (ossia il gate AND).
- Si tratta di una matrice di *n* sommatori a *n* bit. La struttura è regolare quindi semplice da realizzare.
- La performance dipende dai ritardi generati dagli adder, che possono essere mitigate usando adder di tipo CLA (Carry Look-Ahead)

FINE