Left Shift (moltiplicazione per 2, 4, 8 …)

analogo in base 10:
1320 = 132 x 10
13200 = 132 x 100

- *Left-shift* (di *n*): spostare le cifre binarie *n* posti a sinistra
  - le *n* cifre più a sinistra scompaiono, e da destra compaiono *n* **zeri**
- Ricorda:

  uno *shift* a sinistra in base 2 di 1 = raddoppio del numero

  uno *shift* a sinistra in base 2 di *n* cifre = moltiplicazione per 2^n
- Notazione: <<
- Esempio:

  $|00011011|_2 = 27$

  $|00011011|_2 << 1 \;=\; |00110110|_2 = 54 \;\; (= 27 \times 2)$

  $|00011011|_2 << 2 \;=\; |01101100|_2 = 108 \; (= 27 \times 4)$

  $|00011011|_2 << 3 \;=\; |11011000|_2 = 216 \; (= 27 \times 8)$
- <u>**SI! Vale anche in CP2!**</u>

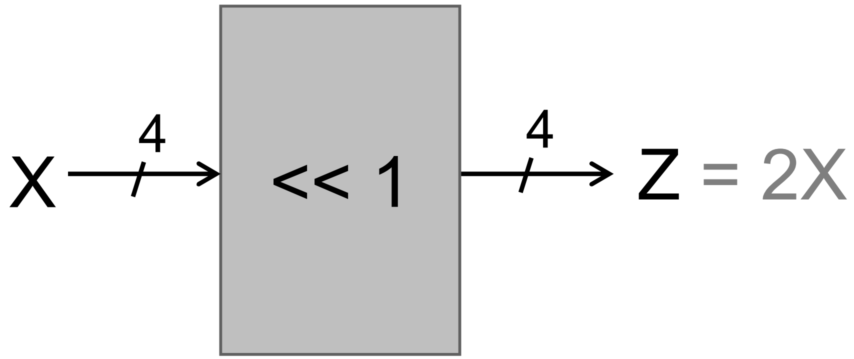
  $|11111011|_2 = -5$

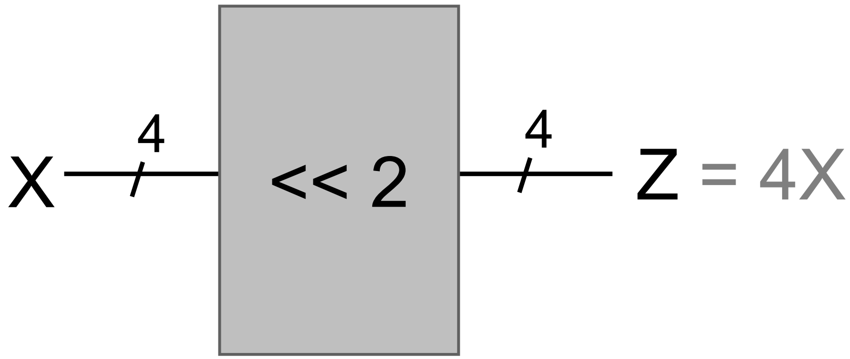
  $|11111011|_2 << 1 \;=\; |11110110|_2 = -10$

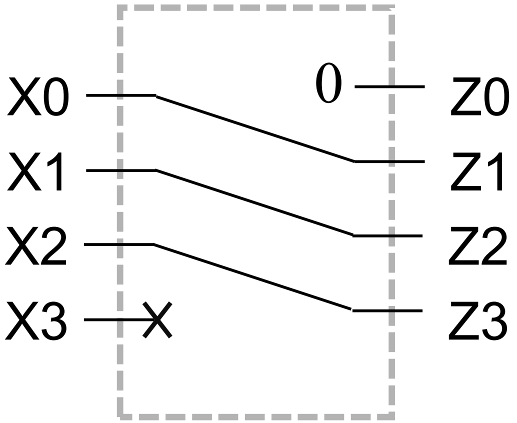
  $|11111011|_2 << 2 \;=\; |11101100|_2 = -20$

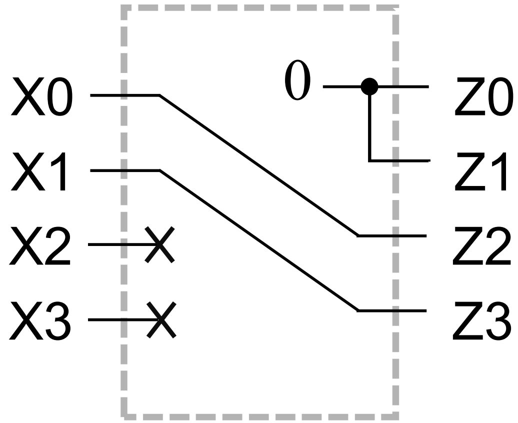
Left Shift
(con secondo parametro costante)

- Blocchi funzionale per shift: (usano…. zero porte!)
- Esempi per 4 bit:







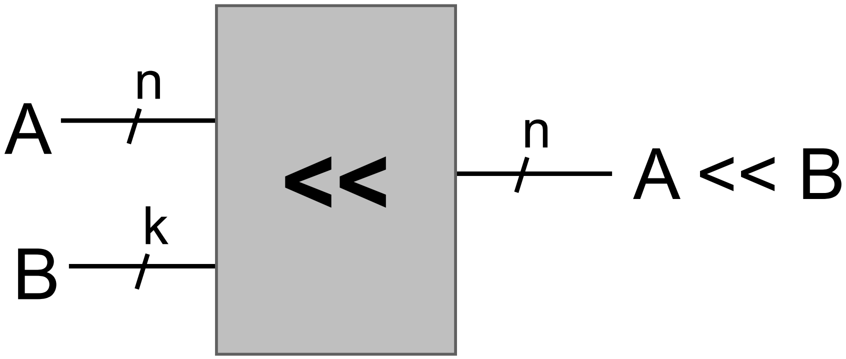


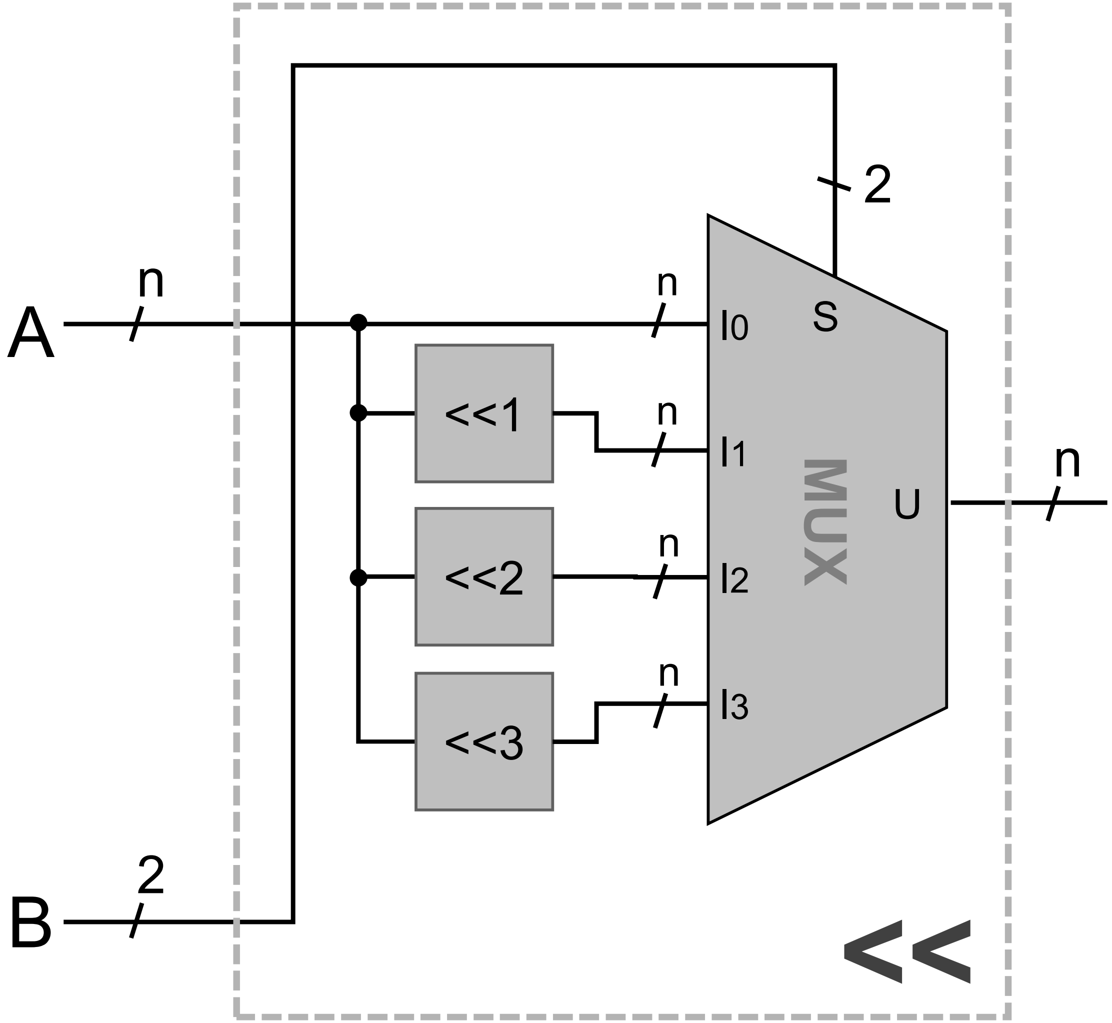
Left-Shift
domande e esercizi

- **Tempo di commutazione**?
  - quali sono i tempi dei blocchi funzionali visti (“<<1”, “<<2”) ?
- **Overflow**?
  considerato come una moltiplicazione per 2^n
  left-shift può generare overflow!
  - come accorgersi se “**<<1**” fa overflow, per **naturali** (no segno) ?
  - come accorgersi se “**<<2**” fa overflow, per **naturali** (no segno) ?
  - come accorgersi se “**<<1**” fa overflow, per **CP2** ?
  - come accorgersi se “**<<2**” fa overflow, per **CP2** ?
  - per ciascun caso, realizza un blocco funzionale
    che prevede un bit ulteriore di uscita “overflow”,
    che vale 1 sse l’operazione ha generato un overflow.
- **Left shift a 2 parametri**?
  - realizza un blocco funzionale che prende
    in input A (*n* bit), e B (2 bit), e dà in output A<<B

← prossimo lucido

Left Shift
(con secondo parametro variabile)





Right Shift
(divisione per 2, 4, 8 …)

analogo in base 10:
132 = 1327 / 10
13 = 1327 / 100

- *Right-shift* (di *n*): spostare le cifre binarie *n* posti a destra
  - *n* le cifre più a destra scompaiono, e da sx compaiono *n* **zeri**
- Ricorda:

  uno *shift* a dx in base 2 di 1 = dimezzamento del numero, *per difetto*

  uno *shift* a dx in base 2 di *n* cifre = divisione per 2^n, *per difetto*
- Notazione: >>
- Esempio: $|00011011|_2 = 27$

  $|00011011|_2 >> 1 \;=\; |00001101|_2 = 13 \quad (= 27 / 2)$

  $|00011011|_2 >> 2 \;=\; |00000110|_2 = 6 \quad (= 27 / 4)$
- Per il **CP2**: l'operazione equivalente è *right shift in segno*
  - da sx compaiono: zeri se MSB = 0, ma uni se MSB = 1
  - l'arrotondamento risultante è comunque *per difetto*

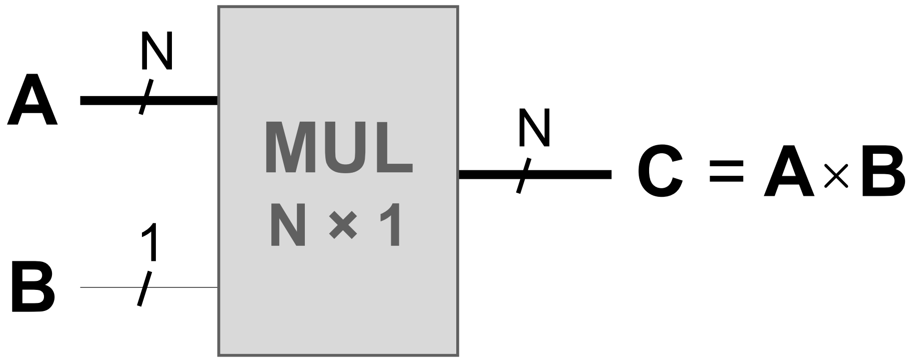
    $|11110011|_2 = -13$

    $|11110011|_2 >> 1 \;=\; |11111001|_2 = -7$

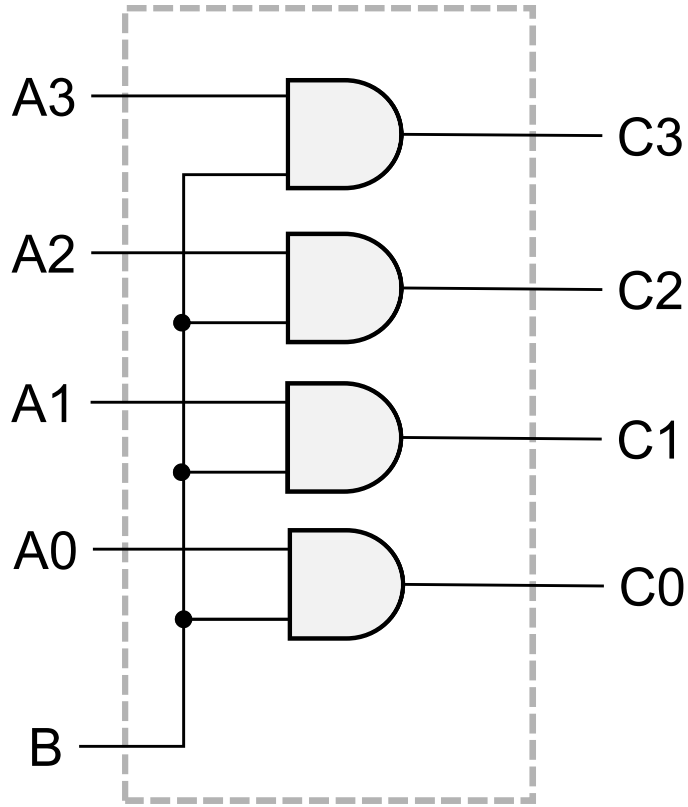
    $|11110011|_2 >> 2 \;=\; |11111100|_2 = -4$

←verificare!

Circuito per moltiplicazione…
…con secondo termine a 1 bit ☺



Realizzazione (banale!):
(qui, per N = 4)



(è un circuito che annulla
A quando B vale 0,
altrimenti lascia A inalterato)

Somme e prodotti : quante cifre richiede il risultato?

Osservazioni:

- **sommando** un numero a *n* cifre con uno a *n* cifre, ottengo al massimo un numero a n+1 cifre.
  (e la (n+1)-sima cifra è 1 oppure 0, in qualsiasi base)
- **moltiplicando** un numero a *n* cifre con uno a *m* cifre, ottengo al massimo un numero a $n+m$ cifre

vale in qualsiasi base!

```
 XXXX +
 YYYY =
 ----
1ZZZZ
```

al max
(è il possibile overflow!)

```
    XXXX ×
    YYYY =
 --------
ZZZZZZZZ
```

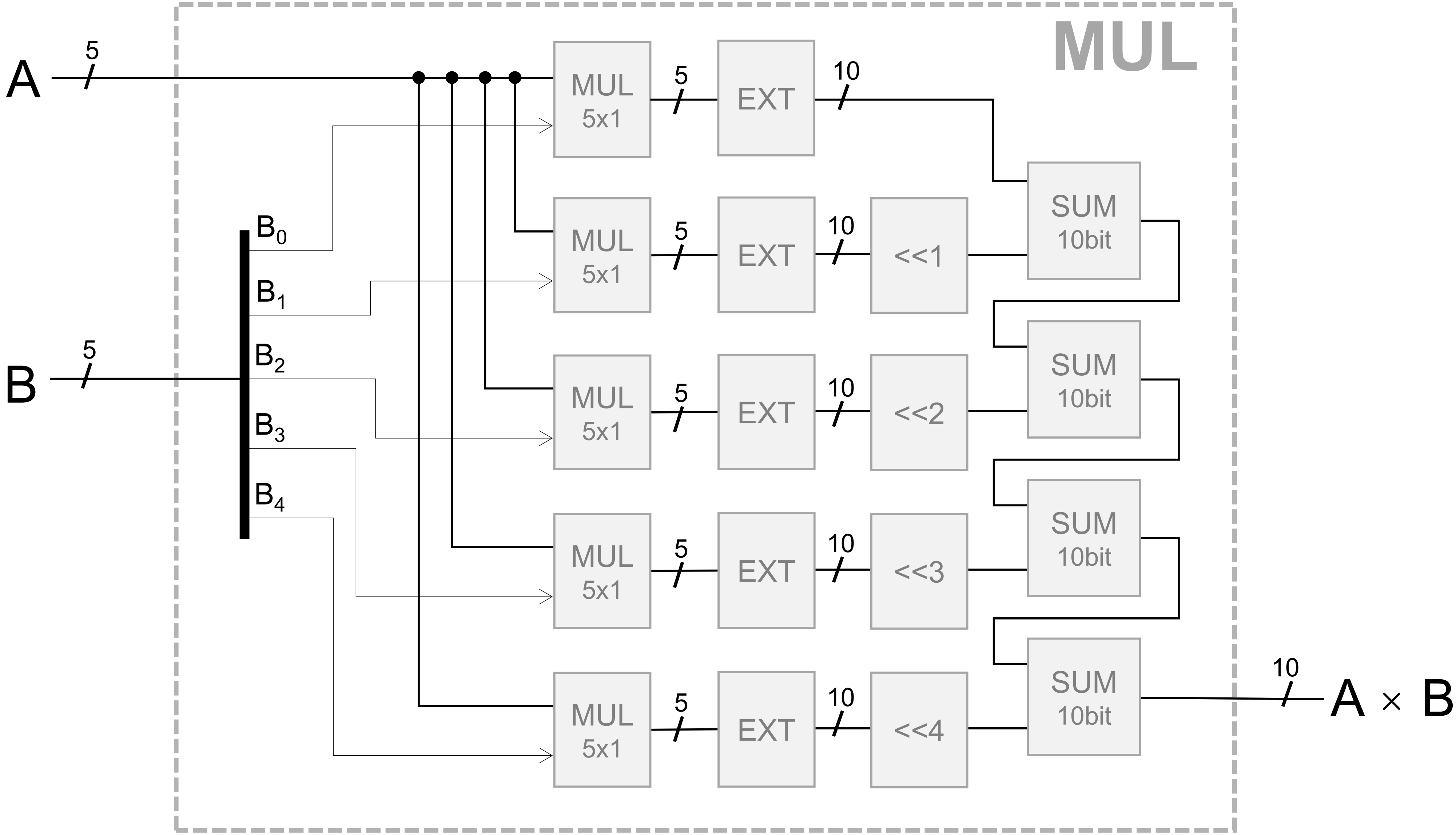
Prodotto: algoritmo delle scuole elementari (ma in base 2)

		11001	×	← A
		10110	=	← B
		1 1 1 1 1		
	~~0×~~	~~00000~~	~~+~~	
	1×	11001	+	
B	1×	11001	+	
	~~0×~~	~~00000~~	+	
	1×	11001	=	
		1000100110		← A × B

Prodotto con left-shift, estensioni, somme. Esempio.

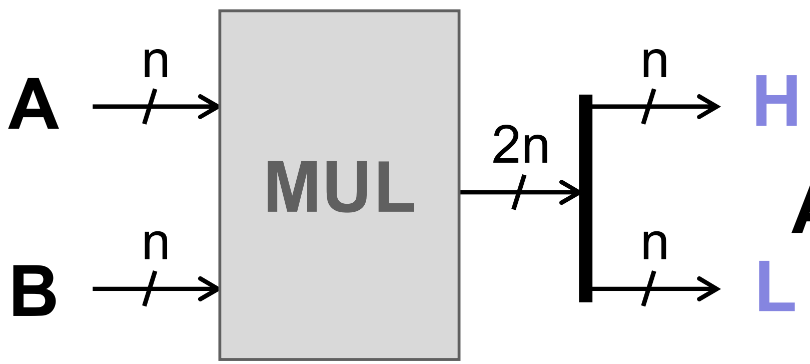
		11001	×	← A		
		10110	=	← B = 16+~~8~~+4+2+~~1~~		
	0×	~~0000011001~~	~~+~~	← ~~A~~	← ~~A × 1~~	+
	1×	0000110010	+	← A<<1	← A × 2	+
B	1×	0001100100	+	← A<<2	← A × 4	+
	0×	~~0011001000~~	~~+~~	← ~~A<<3~~	← ~~A × 8~~	+
	1×	0110010000	=	← A<<4	← A ×16	=
		1000100110		←	$A \times \underbrace{(16+4+2)}_{B}$	

Un circuito per la moltiplicazione a 5 bit



Circuito per Moltiplicazione (fra naturali senza segno) nella pratica

- Moltiplicando 2 numeri a ***n* bit**, ottengo un numero a ***2n bit*** (in generale)
  - almeno, 2*n* cifre binarie bastano sempre: **niente overflow** !
- Scomodo!
  E' poco pratico, per un'architettura, rappresentare il risultato di una operazione matematica con un numero di bit diverso dagli operandi
- Soluzione adottata (a volte):
  considerare il risultato di una moltiplicazione come composto da <u>due numeri</u> da *n* bit ciascuno:
  - LOW: gli *n* bit meno significativi
  - HIGH: gli *n* bit più significativi



$$A \times B = \mathrm{Ext}_{2n}(H) << n + \mathrm{Ext}_{2n}(L)$$