Multiplication: $\boxed{6 \times 13 = 78}$

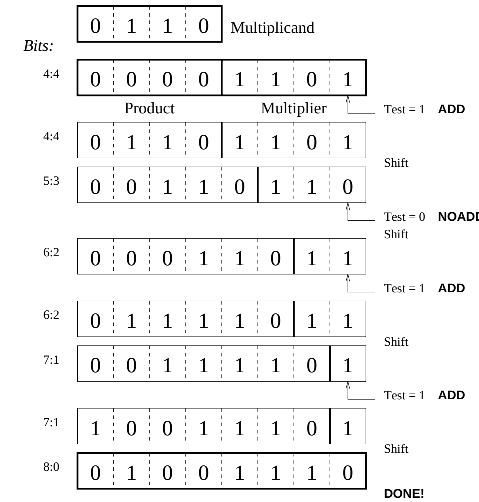
Sequential addition from row to row:

	Sum:
$\;\;\;\;0\,1\,1\,0$	
$\times\;1\,1\,0\,1$	
$\;\;\;\;0\,1\,1\,0$	00000110
$\;\;\;0\,0\,0\,0$	00000110
$\;\;0\,1\,1\,0$	00011110
$+\,0\,1\,1\,0$	01001110
$0\,1\,0\,0\,1\,1\,1\,0$	Product

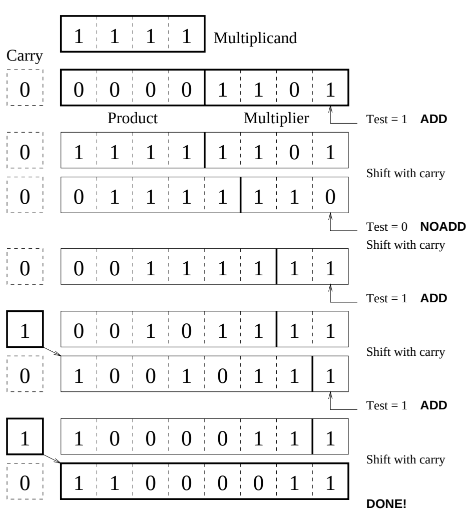
Sequential Shift/Add-Method

- Method to avoid adder arrays
- shift register for partial product and multiplier
- with each cycle,
  1. partial product increases by one digit
  2. multiplier is reduced by one digit
- MSBs of partial product and multiplicand are aligned in each cycle
- not the multiplicand is shifted
  $\Rightarrow$ partial product and multiplier are

Example 1: $\boxed{6 \times 13 = 78}$



Example 2: $\boxed{15 \times 13 = 195}$



Sequential Shift/Add-Method

1. Load multiplier into *lower* half of shift register (the *upper* half is to be zeroed)
2. test LSB of the shift register
3. if LSB is set
   - then add multiplicand to the *upper* half of the shift register
   - else add nothing (make sure carry-bit is cleared!)
4. perform right shift including carry on *full* shift register
5. repeat from 2. as long as multiplier part of shift register is not empty
6. after termination, the shift register (both halves!) contains the product

- Easy to implement in software

Signed Multiplication

Sign and magnitude representation:

- Calculate unsigned product as
  $|p| = |x| \times |y|$
  $\Rightarrow\; p_{0-(2n-2)} = x_{0-(n-2)} \times y_{0-(n-2)}$
- determine sign separately as
  $\mathrm{sgn}(p) = \mathrm{sgn}(x) \times \mathrm{sgn}(y)$
  $\Rightarrow\; p_{2n-1} = x_{n-1} \oplus y_{n-1}$

More difficult if 2's complement is used:

- use 2's complement of negative multiplicand for summing up partial products
- *sign extension* is needed

Example: $\boxed{(-5)\times(+6)=(-30)}$

Multiplicand:

Unsigned	0 1 0 1
1's complement	1 0 1 0
2's complement	1 0 1 1

1 0 1 1	Multiplicand
× 0 1 1 0	Multiplier
0 0 0 0 0 0 0 0	$0000 \times x1011$
1 1 1 1 0 1 1 0	$0010 \times x1011$
1 1 1 0 1 1 0 0	$0100 \times x1011$
0 0 0 0 0 0 0 0	$0000 \times x1011$
1 1 1 1 1 0 0	(carry)
1 1 1 0 0 0 1 0	Product

Disadvantages:

1. Sign-extension for negative *multiplicands* not applicable for negative *multipliers*
2. long sequences of 1's in the multiplier $\Rightarrow$ large number of summands

Solution for problem 1.:

- in case of a negative multiplier, negate both operands by applying the 2's complement

Solution for problem 2.:

- analyze groups of 1's in the multiplier and replace them by a shorter and more efficient representation *(→Booth algorithm)*

Booth Algorithm

Based on the observation that

$$\sum_{i=0}^{n-1} 2^i \;=\; 2^n - 1$$

Generate sequence of 1's in bits j to k by subtraction of two operands with single 1 each:

$$\begin{array}{rl} 1000 & (8) \\ -0001 & (1) \\ \hline 0111 & (7) \end{array} \qquad \begin{array}{rr} 01000000 & (64) \\ -00001000 & (8) \\ \hline 00111000 & (56) \end{array}$$

This method works equally well for both unsigned and 2's complement representations

Booth Recoding

Consider the example of encoding 56:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	1	1	0	0	0
0	+1	0	0	−1	0	0	0

$$\begin{aligned} 2^5+2^4+2^3 &= 32+16+8 \\ &= 56 \\ 2^6-2^3 &= 64-8 \\ &= 56 \end{aligned}$$

Booth Recoding

1. Parse multiplier from left to right $(i = n-1 \cdots 0)$
2. for each change from 0 to 1 or vice versa, encode ± 1:
   - if bit i is 0 and bit $i-1$ is 1 $\Rightarrow$ recode to $+1$
   - if bit i is 1 and bit $i-1$ is 0 $\Rightarrow$ recode to -1
3. for bit 0, assume bit $i = -1$ with value 0

During the multiplication:

- the multiplicand is added for $+1$ digits
- the 2's complement is added for -1 digits

Booth Algorithm and Bit Pairing

The Booth technique has its major advantage if

1. the operands have a large number of bits
2. multiplier contains long sequences of 1's

it has its limitations if

- the multiplier contains only small groups of 1's or even alternating 0–1 pairs

It can be enhanced by *bit-pairing*,

- 50% maximum number of summands
- handling 0–1 pairs efficiently
- additional overhead for multiplicand

Bit-Pairing of Booth Recoding

Apply Booth recoding on the multiplier first, then pair bits —

1. Within a sequence:

$$\begin{array}{cc} 0 & 0 \\ \hline & 0 \end{array}$$

2. Begin of a 1's-sequence:

$$\begin{array}{cc} 0 & +1 \\ \hline & +1 \end{array} \qquad \begin{array}{cc} +1 & -1 \\ \hline & +1 \end{array} \qquad \begin{array}{cc} +1 & 0 \\ \hline & +2 \end{array}$$

3. End of a 1's-sequence:

$$\begin{array}{cc} 0 & -1 \\ \hline & -1 \end{array} \qquad \begin{array}{cc} -1 & +1 \\ \hline & -1 \end{array} \qquad \begin{array}{cc} -1 & 0 \\ \hline & -2 \end{array}$$

Bit-Pair Booth Recoding Examples

Worst-case multiplier:

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
+1	−1	+1	−1	+1	−1	+1	−1	+1	−1	+1	−1	+1	−1	+1	−1
	+1		+1		+1		+1		+1		+1		+1		+1

Ordinary multiplier:

1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
0	−1	0	0	+1	−1	+1	0	−1	+1	0	0	0	−1	0	0
	−1		0		+1		+2		−1		0		−1		0

Good multiplier:

0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
0	0	0	+1	0	0	0	0	−1	0	0	0	+1	0	0	−1
	0		+1		0		0		−2		0		+2		−1

Conclusions for Booth Algorithm

- Booth algorithm can reduce number of non-zero summands considerably
- Worst case: May *double* the number of non-zero summands
- Bit-pairing can reduce the number of summands further
- . . . won't help if multiplier is optimal after conventional Booth recoding
- For the sequential shift/add hardware, bit-pairing reduces the summation effort substantially, with or without Booth recoding