7 Multiplication and Division

- Multiplication is done by doing shifts and additions.
- Multiplying two (unsigned) numbers of $n$ bits each results in a product of $2n$ bits.

Example: $0110 \times 0011$ (6x3)

At start, product = 00000000
looking at each bit of the multiplier 0110, from right to left:
  0: product unchanged: 00000000,
      shift multiplicand left: 00110
  1: add multiplicand to product: 00000000 + 00110 = 00000110,
      shift multiplicand left: 001100
  1: add multiplicand to product: 00000110 + 001100 = 00010010,
      shift multiplicand left: 0011000
  0: product unchanged: 00010010 = 18_{10}
      shift multiplicand left: 00110000
Main difficulty arises when signed numbers are involved.

Naive approach: convert both operands to positive numbers, multiply, then calculate separately the sign of the product and convert if necessary.

A better approach: Booth's Algorithm.

Booth's idea: if during the scan of the multiplier, we observe a sequence of 1s, we can replace it first by subtracting the multiplicand (instead of adding it to the product) and later, add the multiplicand, after seeing the last 1 of the sequence.
Example

0110 x 0011 (6x3)
This can be done by (8-2)x3 as well,
or (1000 - 0010) x 0011 (using 8 bit words)

At start, product = 00000000
looking at each bit of the multiplier 0110, from right to left:
0: product unchanged: 00000000,
    shift multiplicand left: 00110
1: start of a sequence:
    subtract multiplicand from product: 00000000 - 00110 = 11111010,
    shift multiplicand left: 001100
1: middle of sequence, product unchanged: 1111010,
    shift multiplicand left: 0011000
0: end of sequence:
    add multiplicand to product: 1111010 + 0011000 = 00010010 = 18_{10}
    shift multiplicand left: 00110000
Booth's Algorithm

Scan the multiplier from right to left, observing, at each step, both the current bit and the previous bit:

1. Depending on (current, previous) bits:

- **00**: Middle of a string of 0s: do no arithmetic operation.
- **10**: Beginning of a string of 1s: **subtract** multiplicand.
- **11**: Middle of a string of 1s: do no arithmetic operation.
- **01**: End of a string of 1s: **add** multiplicand.

2. Shift multiplicand to the left.
Why does Booth's Algorithm work?

Let \( a \) and \( b \) be signed integer numbers of \( n \) bits each.

Redefine step 1 of Booth's algorithm (see previous page):

\[
\begin{cases}
0 \Rightarrow (a_i, a_{i-1}) = 00 \text{ or } 11 \Rightarrow \text{no arithmetic operation.} \\
1 \Rightarrow (a_i, a_{i-1}) = 01 \Rightarrow \text{add multiplicand.} \\
-1 \Rightarrow (a_i, a_{i-1}) = 10 \Rightarrow \text{subtract multiplicand.}
\end{cases}
\]

Calculate now the product \( a \times b \), according to Booth's algorithm:

\[
a \times b = (a_1 - a_0) \times 2^0 \times b + \leftarrow a_1 = 0
\]

\[
+ (a_0 - a_1) \times 2^1 \times b +
\]

\[
+ (a_1 - a_2) \times 2^2 \times b +
\]

\[
\ldots
\]

\[
+ (a_{n-3} - a_{n-2}) \times 2^{n-2} \times b +
\]

\[
+ (a_{n-2} - a_{n-1}) \times 2^{n-1} \times b = -a_{n-1} \times 2^{n-1} \times b + \sum_{i=0}^{n-2} a_i \times 2^i \times b =
\]

\[
= b \times (-a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i ) = a \times b
\]
Implementing multiplication due to Booth's Algorithm:

At start, the multiplier occupies the right half of the product.
At each step, the control verifies the rightmost bit of the multiplier and decides whether to add the multiplicand to the product, to subtract it from, or just to shift the product right (instead of shifting the multiplicand to the left).
At the end, the multiplier is out of the product register and the product contains the result.
**Division**

- Division is done by doing shifts and subtractions.
- Dividing a number of $2n$ bits by a number of $n$ bits results in a quotient of up to $2n$ bits and a remainder of up to $n$ bits.

**Example:** $01001010 : 1000 (74:8)$

```
          1001  Quotient
Dividend  01001010  1000  Divisor
-  1000
   10
  101
  1010
-  1000
   10  Remainder
```
Division Algorithm

At start, the \( n \) bits divisor is shifted to the left, while \( n \) 0s are added to its right. This way the dividend and the divisor are \( 2n \) bits long.

At each step (repeating the following \( n+1 \) times),
subtract the divisor from the dividend.
if the result is non-negative,
    shift the quotient left and place a 1 in the new place.
else
    shift the quotient left and place a 0 in the new place.
    restore the dividend by adding the divisor to it.
shift the divisor to the right.

Note: the above algorithm assumes a quotient of no more than \( n+1 \) bits long.
Otherwise, at initialization, we should shift the divisor left until its MSB is 1.
Implementing division:

At start, the dividend occupies the right half of the remainder register. Shift remainder left 1 position. At each step, the control subtracts the divisor from the left half of the remainder register, putting there the result. If the remainder is negative, it restores it. Then, instead of shifting the divisor to the right, it shifts the remainder to the left and inserts 0 or 1, according to the sign of the remainder.

At the end, the remainder register contains the quotient in its right half and the remainder in its left half.
Dividing Signed Numbers

The above algorithm and implementation deals only with positive numbers.

If negative numbers are involved, the sign of the quotient is set to MINUS if the divisor and the dividend are of different signs.

The sign of the remainder, though, is more difficult to set.

Follow the rule: **Dividend = Quotient x Divisor + Remainder**

Example:

\[
+7/\pm 2 = +3(\pm 1) \text{ since } +7 = +2\times(+3)+1 \\
+7/-2 = -3(\pm 1) \text{ since } +7 = -2\times(-3)+1 \\
-7/\pm 2 = -3(-1) \text{ since } -7 = +2\times(-3)-1 \\
-7/-2 = +3(-1) \text{ since } -7 = -2\times(+3)-1
\]

Conclusion: the sign of the remainder is set according to the sign of the dividend.