1 Multiplication

Consider two unsigned binary numbers \( X \) and \( Y \). We want to multiply these numbers. The basic algorithm is similar to the one used in multiplying the numbers on pencil and paper. The main operations involved are shift and add.

Recall that the ‘pencil-and-paper’ algorithm is inefficient in that each product term (obtained by multiplying each bit of the multiplier to the multiplicand) has to be saved till all such product terms are obtained. In machine implementations, it is desirable to add all such product terms to form the partial product. Also, instead of shifting the product terms to the left, the partial product is shifted to the right before the addition takes place. In other words, if \( P_i \) is the partial product after \( i \) steps and if \( Y \) is the multiplicand and \( X \) is the multiplier, then

\[
P_{i+1} \leftarrow P_i + x_j \cdot Y
\]

and

\[
P_{i+1} \leftarrow P_i \cdot 2^{-1}
\]

and the process repeats.

Note that the multiplication of signed magnitude numbers requires a straightforward extension of the unsigned case. The magnitude part of the product can be computed just as in the unsigned magnitude case. The sign \( p_0 \) of the product \( P \) is computed from the signs of \( X \) and \( Y \) as

\[
p_0 \leftarrow x_0 \oplus y_0
\]

1.1 Two’s complement Multiplication - Robertson’s Algorithm

Consider the case that we want to multiply two 8 bit numbers \( X = x_0x_1...x_7 \) and \( Y = y_0y_1...y_7 \). Depending on the sign of the two operands \( X \) and \( Y \), there are 4 cases to be considered:

- \( x_0 = y_0 = 0 \), that is, both \( X \) and \( Y \) are positive. Hence, multiplication of these numbers is similar to the multiplication of unsigned numbers. In other words, the product \( P \) is computed in a series of add-and-shift steps of the form

\[
P_{i+1} \leftarrow P_i + x_j \cdot Y
\]

\[
P_{i+1} \leftarrow P_i \cdot 2^{-1}
\]

Note that all the partial products are non-negative. Hence, leading 0s are introduced during right shift of the partial product.
• $x_0 = 1, y_0 = 0$, that is, $X$ is negative and $Y$ is positive. In this case, the partial product is always positive (till the sign bit $x_0$ is used). In the final step, a subtraction is performed. That is,

$$P ← P - Y$$

• $x_0 = 0, y_0 = 1$, that is, $X$ is positive and $Y$ is negative. In this case, the partial product is positive and hence leading 0s are shifted into the partial product until the first 1 in $X$ is encountered. Multiplication of $Y$ by this 1, and addition to the result causes the partial product to be negative, from which point on leading 1s are shifted in (rather than 0s).

• $x_0 = 1, y_0 = 1$, that is, both $X$ and $Y$ are negative. Once again, leading 1s are shifted into the partial product once the first 1 in $X$ is encountered. Also, since $X$ is negative, the correction step (subtraction as the last step) is also performed.

**A Word of Caution:** A difference exists in the correction steps between multiplication of two integers and two fractions. In the case of two integers, the correction step involves subtraction and shift right. In the case of fractions, the correction step involves subtraction and setting $Q(7) ← 0$.

### 1.2 Booth’s Algorithm

Recall that the preceding multiplication algorithms (Robertson’s algorithm) involves scanning the multiplier from right to left and using the current multiplier bit $x_i$ to determine whether the multiplicand $Y$ be added, subtracted or add 0 (do nothing) to the partial product. In Booth’s algorithm, two adjacent bits $x_ix_{i+1}$ are examined in each step. If $x_ix_{i+1} = 01$, then $Y$ is added to the partial product, while if $x_ix_{i+1} = 10$, $Y$ is subtracted from $P_i$ (partial product). If $x_ix_{i+1} = 00$ or 11, then neither addition nor subtraction is performed. Thus, Booth’s algorithm effectively skips over sequences of 1s and 0s in $X$. As a result, the total number of addition/subtraction steps required to multiply two numbers decrease (however, at the cost of extra hardware).

The process of inspecting the multiplier bits required by Booth’s algorithm can be viewed as encoding the multiplier using three digits 0, 1 and $\overline{1}$, where 0 means shift the partial product to the right (that is, no addition or subtraction is performed), while 1 means add multiplicand before shifting and $\overline{1}$ means subtract multiplicand from the partial product before shifting. The number thus produced is called a *signed digit number* and this process of converting a multiplier $X$ into a

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$x_{i+1}$</th>
<th>$x_i^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$\overline{1}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 1: Multiplier Recoding for Basic Booth’s*
signed digit form is called *multiplier recoding*. To generate \( X^* \) from \( X \), append the number \( X \) with a 0 to the right (that is, start with \( X = x_0x_1...x_{n-1}0 \)). Then use Table 1 to generate \( X^* \) from \( X \).

Booth’s algorithm results in reduction in the number of add/subtract steps needed (as compared to Robertson’s algorithm) if the multiplier contains runs (or sequences) of 1s or 0s. The worst case scenario occurs in Booth’s algorithm if \( X = 010101.01 \), where there are \( n/2 \) isolated 1s, which forces \( n/2 \) subtractions and \( n/2 \) additions. This is worse than the standard multiplication algorithm - which contains only \( n/2 \) additions.

The basic Booth’s algorithm can be improved by detecting an isolated 1 in the multiplier and just performing addition at the corresponding point in the multiplication. Similarly, an isolated 0 corresponds to a subtraction. The algorithm thus updated is denoted as *modified Booth’s algorithm*, which always requires fewer addition/subtractions as compared to other multiplication algorithms.

Note that the basic Booth’s algorithm can be implemented by examining two adjacent bits \( x_i x_{i+1} \) of the multiplier. The modified Booth’s algorithm can be implemented by identifying isolated 1s and 0s. This is achieved by using a mode flip-flop \( F \), which is set to 1 when a run of two or more 1s is encountered, and is reset to 0 when the run of 1s end with two or more 0s.

Analogous to the basic Booth’s algorithm recoding technique, the multiplier recoding scheme that takes isolated 0s and 1s into account is called a *canonical signed digit recoding*. The basic steps for canonical recoding are as follows:

- First \( x_{-1} = x_0 \) is appended to the left end and a 0 is appended to the right end of the number \( x_0x_1x_2...x_{n-1} \) to create \( X = x_{-1}x_0x_1...x_{n-1}10 \).
- \( X \) is the scanned from right to left and the pair of bits \( x_{i-1}x_i \) are used to determine bit \( x_i^* \) of the number \( X^* \) using the following algorithm:

  *How to get this table?:* The above conversion table can be easily derived from the basic multiplier recoding table. There are two special cases we need to consider:

  1. \( x_{i-1}x_i = 101 \). This is the situation when an isolated 1 is encountered. Here, we just want to perform a subtraction. Hence, set \( x_i^* = 1 \) and \( f = 1 \).

<table>
<thead>
<tr>
<th>( x_{i-1} )</th>
<th>( x_i )</th>
<th>( f )</th>
<th>( x_i^* )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Canonical Signed Digit Recoding for Modified Booth’s
2. \( x_{i-1}x_i x_{i+1} = 010 \). This is the situation which we want to treat as a sequence of 0s. Hence, we want to perform the addition corresponding to the isolated 1 and set the \( f \) flag to 0. In other words, \( x_i^* = 1 \) and \( f = 0 \).

3. The rest of the entries of the table can be derived by treating \( f \) to be equal to \( x_{i+1} \), \( x_i \) to be equal to \( x_i \) of the basic multiplier recoding table of Table 1 and treating \( x_{i-1} \) as the lookahead. The value of the lookahead (that is, \( x_{i-1} \)) can be used to determine the new value of \( f \).