In More Depth: Booth's Algorithm

A more elegant approach to multiplying signed numbers than above is called *Booth's algorithm*. It starts with the observation that with the ability to both add and subtract there are multiple ways to compute a product. Suppose we want to multiply 2_{ten} by 6_{ten}, or 0010_{two} by 0110_{two}:

```
        0010two
x       0110two
   ------------
+       0000 shift (0 in multiplier)
+      0010  add   (1 in multiplier)
+     0010   add   (1 in multiplier)
+    0000    shift (0 in multiplier)
    ----
     00001100two
```

Booth observed that an ALU that could add or subtract could get the same result in more than one way. For example, since

$$6_{ten} = -2_{ten} + 8_{ten}$$

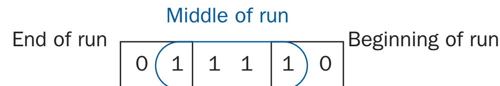
or

$$0110_{two} = -0010_{two} + 1000_{two}$$

we could replace a string of 1s in the multiplier with an initial subtract when we first see a 1 and then later add when we see the bit *after* the last 1. For example,

```
     0010two
x    0110two
   ---------
+    0000   shift (0 in multiplier)
-   0010    sub (first 1 in multiplier)
+ 0000      shift (middle of string of 1s)
+0010       add (prior step had last 1)
 ----
00001100two
```

Booth invented this approach in a quest for speed because in machines of his era shifting was faster than addition. Indeed, for some patterns his algorithm would be faster; it's our good fortune that it handles signed numbers as well, and we'll prove this later. The key to Booth's insight is in his classifying groups of bits into the beginning, the middle, or the end of a run of 1s:



Of course, a string of 0s already avoids arithmetic, so we can leave these alone.

If we are limited to looking at just 2 bits, we can then try to match the situation in the preceding drawing, according to the value of these 2 bits:

If we are limited to looking at just 2 bits, we can then try to match the situation in the preceding drawing, according to the value of these 2 bits:

Current bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1s	00001111000_{two}
1	1	Middle of a run of 1s	00001111000_{two}
0	1	End of a run of 1s	00001111000_{two}
0	0	Middle of a run of 0s	00001111000_{two}

Booth's algorithm changes the first step of the algorithm—looking at 1 bit of the multiplier and then deciding whether to add the multiplicand—to looking at 2 bits of the multiplier. The new first step, then, has four cases, depending on the values of the 2 bits. Let's assume that the pair of bits examined consists of the current bit and the bit to the right—which was the current bit in the previous step. The second step is still to shift the product right. The new algorithm is then the following:

1. Depending on the current and previous bits, do one of the following:

   00: Middle of a string of 0s, so no arithmetic operation.

   01: End of a string of 1s, so add the multiplicand to the left half of the product.

   10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.

   11: Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right 1 bit.

Now we are ready to begin the operation, shown in Figure 3.11.2. It starts with a 0 for the mythical bit to the right of the rightmost bit for the first stage. Figure 3.11.2 compares the two algorithms, with Booth's on the right. Note that Booth's operation is now identified according to the values in the 2 bits. By the fourth step, the two algorithms have the same values in the Product register.

The one other requirement is that shifting the product right must preserve the sign of the intermediate result, since we are dealing with signed numbers. The solution is to extend the sign when the product is shifted to the right. Thus, step 2 of the second iteration turns $1110\ 0011\ 0_{two}$ into $1111\ 0001\ 1_{two}$ instead of $0111\ 0001\ 1_{two}$. This shift is called an *arithmetic right shift* to differentiate it from a logical right shift.

Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⟹ no operation	0000 0110	1a: 00 ⟹ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1 ⟹ Prod = Prod + Mcand	0010 0011	1c: 10 ⟹ Prod = Prod − Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 ⟹ Prod = Prod + Mcand	0011 0001	1d: 11 ⟹ no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 ⟹ no operation	0001 1000	1b: 01 ⟹ Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

FIGURE 3.11.2 Comparing algorithm in Booth's algorithm for positive numbers. The bit(s) examined to determine the next step is circled in color.

EXAMPLE

Booth's Algorithm

Let's try Booth's algorithm with negative numbers: $2_{ten} \times -3_{ten} = -6_{ten}$, or $0010_{two} \times 1101_{two} = 1111\ 1010_{two}$.

ANSWER

Figure 3.11.3 shows the steps.

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: 11 ⇒ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

FIGURE 3.11.3 Booth's algorithm with negative multiplier example. The bits examined to determine the next step are circled in color.

Our example multiplies one bit at a time, but it is possible to generalize Booth's algorithm to generate multiple bits for faster multiplies (see Exercise 3.50)

Now that we have seen Booth's algorithm work, we are ready to see *why* it works for two's complement signed integers. Let a be the multiplier and b be the multiplicand and we'll use a_i to refer to bit i of a. Recasting Booth's algorithm in terms of the bit values of the multiplier yields this table:

a_i	a_{i-1}	Operation
0	0	Do nothing
0	1	Add b
1	0	Subtract b
1	1	Do nothing

Instead of representing Booth's algorithm in tabular form, we can represent it as the expression

$$(a_{i-1} - a_i)$$

where the value of the expression means the following actions:

$$\begin{aligned} 0 &: \text{ do nothing} \\ +1 &: \text{ add } b \\ -1 &: \text{ subtract } b \end{aligned}$$

Since we know that shifting of the multiplicand left with respect to the Product register can be considered multiplying by a power of 2, Booth's algorithm can be written as the sum

$$
\begin{array}{lllll}
 & (a_{-1} & -\ a_0) & \times\ b & \times\ 2^0 \\
+ & (a_0 & -\ a_1) & \times\ b & \times\ 2^1 \\
+ & (a_1 & -\ a_2) & \times\ b & \times\ 2^2 \\
\ldots & \ldots & & & \\
+ & (a_{29} & -\ a_{30}) & \times\ b & \times\ 2^{30} \\
+ & (a_{30} & -\ a_{31}) & \times\ b & \times\ 2^{31}
\end{array}
$$

We can simplify this sum by noting that

$$- a_i \times 2^i + a_i \times 2^{i+1} = (-a_i + 2a_i) \times 2^i = (2a_i - a_i) \times 2^i = a_i \times 2^i$$

recalling that $a_{-1} = 0$ and by factoring out b from each term:

$$b \times ((a_{31} \times -2^{31}) + (a_{30} \times 2^{30}) + (a_{29} \times 2^{29}) + \ldots + (a_1 \times 2^1) + (a_0 \times 2^0))$$

The long formula in parentheses to the right of the first multiply operation is simply the two's complement representation of *a* (see page 163). Thus, the sum is further simplified to

$$b \times a$$

Hence, Booth's algorithm does in fact perform two's complement multiplication of *a* and *b*.

3.23 [30] <§3.6> The original reason for Booth's algorithm was to reduce the number of operations by avoiding operations when there were strings of 0s and 1s. Revise the algorithm on page IMD 3.11-2 to look at 3 bits at a time and compute the product 2 bits at a time. Fill in the following table to determine the 2-bit Booth encoding:

Current bits		Previous bit	Operation	Reason
ai+1	ai	ai–1		
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Assume that you have both the multiplicand and 2 × multiplicand already in registers. Explain the reason for the operation on each line, and show a 6-bit example that runs faster using this algorithm. (Hint: Try dividing to conquer; see what the operations would be in each of the eight cases in the table using a 2-bit Booth algorithm, and then optimize the pair of operations.)