

BOOTH'S ALGORITHM

Introduction

Multiplication was generally implemented via a sequence of addition, subtraction, and shift operations. Multiplication can be considered as a series of repeated additions. The number to be added is the multiplicand, the number of times that it is added is the multiplier, and the result is the product. Each step of addition generates a partial product. In most computers, the operand usually contains the same number of bits. When the operands are interpreted as integers, the product is generally twice the length of operands in order to preserve the information content. This repeated addition method that is suggested by the arithmetic definition is slow that it is almost always replaced by an algorithm that makes use of positional representation. It is possible to decompose multipliers into two parts. The first part is dedicated to the generation of partial products, and the second one collects and adds them. The basic multiplication principle is twofold i.e. evaluation of partial products and accumulation of the shifted partial products. It is performed by the successive additions of the columns of the shifted partial product matrix. The 'multiplier' is successfully shifted and gates the appropriate bit of the 'multiplicand'. The delayed, gated instance of the multiplicand must all be in the same column of the shifted partial product matrix. They are then added to form the product bit for the particular form.

Multiplication is therefore a multi operand operation. To extend the multiplication to both signed and unsigned numbers, a convenient number system would be the representation of numbers in two's complement format. The MAC (Multiplier and Accumulator Unit) is used for image The speed of multiplication operation is of great importance in digital signal processing as well as in processing and digital signal processing (DSP) in a DSP processor. Algorithm of MAC is Booth's algorithm which improves speed and reduces the power.

Booth's algorithm

Booth's algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm was invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is faster than the normal Multiplication Algorithm by using a shifting operation instead of addition operation. Booth's algorithm is a multiplication algorithm which worked for two's complement numbers. It is similar to our paper-pencil method, except that it looks for the current as well as previous bit in order to decided what to do. Booth's algorithm is of interest in the study of computer architecture. It is widely used in the implementations of hardware or software multipliers because its application makes it possible to reduce the number of partial products. It can be used for both sign magnitude numbers as well as 2's complement numbers.

In Booth's algorithm, if the multiplicand and multiplier are n -bit two's complement numbers, the result is considered as $2n$ -bit two's complement value. The overflow bit (outside $2n$ bits) is ignored.

Booth's algorithm is a multiplication algorithm which worked for two's complement numbers. Booth invented this approach in a quest for speed because in machines of his era shifting was faster than addition.

Process

Booth's algorithm changes the first step of the algorithm—looking at LSB bit of the multiplier and booth bit. 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 (x_i) and the bit to the right (x_{i-1})—which was the current bit in the previous step. With this we will have four cases 00, 11, 10 and 01 as shown in the table below. The second step is still to shift the product right. The following are rules.

x_i	x_{i-1}	Comments	Operation
0	0	String of zeros	Shift only
1	1	String of ones	Shift only
1	0	Beginning of a string of ones	Subtract and shift
0	1	End of a string of ones	Add and shift

In Booth's algorithm, if the multiplicand and multiplier are n -bit two's complement numbers, the result is considered as $2n$ -bit two's complement value. The overflow bit (outside $2n$ bits) is ignored.

Booth's algorithm in hardware

The hardware consists of 32-bit register M for the multiplicand, 64-bit product register P , and a 1-bit register C , 32-bit ALU and control. Initially, M contains multiplicand, P contains multiplier (the upper half $P_h = 0$), and C contains bit 0. The algorithm is the following steps.

Repeat 32 times:

1. If (P_0, C) pair is:
 - 10: $P_h = P_h - M$,
 - 01: $P_h = P_h + M$,
 - 00: do nothing,
 - 11: do nothing.
2. Arithmetic shift P right 1 bit. The shift-out bit gets into C .

Logical shift vs. arithmetic shift

The above mentioned shift is arithmetic shift. We have learned the logical shift. For example,

```
shift right logical (srl) 0100 ... 111    -> 00100 ... 11
                          1100 ... 111    -> 01100 ... 11
```

Arithmetic shift preserves the sign of a two's complement number, thus

```
shift right arithmetic (sra) 0100 ... 111    -> 00100 ... 11
                             1100 ... 111    -> 11100 ... 11.
```

Booth's algorithm is illustrated by the following example.

To Calculate 5×-3 using four bit numbers and Booth's algorithm. [1]

First, convert operands into binary

Multiplicand, $5 = 0101$

Multiplier (Q), $-3 = 1101$

Now to find the two's complement of our Multiplicand value so that we can do subtraction of multiplicand from A. We do this by keeping all 0's up until, and including, the first 1 the same. We then flip all the remaining bits. So two's complement of 0101 becomes 1011.

The next step is to set registers, which we name as Multiplicand, A, Q and C. where A is accumulator, Q is multiplier and C is booth bit. A and C are initially set to be zero. A and Q going to be the registers where we get our product result after the working of the problem.

Multiplicand	A	Q	C
0101	0000	1101	0

The next step is to look at the LSB of Q multiplier and the number in the C register which is called booth bit. If the LSB of Q is one, and C is zero, we subtract multiplicand from A. If LSB of Q is zero, C is 1, and then we add multiplicand to A. If both LSB of Q and C are equal, you do nothing and skip to the shifting stage. In our case, the LSB of Q is one, and C is zero, so we subtract multiplicand from A. We then do an arithmetic right shift on A and Q, and also copying the LSB of Q into C. This gives the table:

Multiplicand	A	Q	C
0101	0000	1101	0
	+ <u>1011</u>		
	1011	1101	0
0101	1101	1110	1

We then go through the process again using the same rules. This time we see that the LSB of Q is 0 and C is 1. We must now add multiplicand to A. Once added, we then do an arithmetic right shift on A and Q, with copying the LSB of Q into C. This gives the table:

Multiplicand	A	Q	C
0101	0000	1101	0
	<u>+1011</u>		
	1011	1101	0
0101	1101	1110	1
	<u>+0101</u>		
0101	0010	1110	1
0101	0001	0111	0

Again, we repeat the process again. This time LSB of Q is 1 and C is zero. So like we did on the first pass, we subtract multiplicand from A. We then do an arithmetic right shift on A and Q, and also copy the LSB of Q into C. This gives the table:

Multiplicand	A	Q	C
0101	0000	1101	0
	+ <u>1011</u>		
	1011	1101	0
0101	1101	1110	1
	<u>+0101</u>		
0101	0010	1110	1
0101	0001	0111	0
	<u>+1011</u>		
	1100	0111	0
0101	1110	0011	1

Now on the fourth and final pass, we see that the LSB of Q is 1 and so is C. This makes it easier for us because now we don't have to add the numbers. We only have to do an arithmetic right shift on A and Q, and copy the LSB of Q into C. This gives the table:

Multiplicand	A	Q	C
0101	0000	1101	0
	+ <u>1011</u>		
	1011	1101	0

0101	1101	1110	1
	<u>+0101</u>		
0101	0010	1110	1
0101	0001	0111	0
	<u>+1011</u>		
	1100	0111	0
0101	1110	0011	1
0101	1111	0001	1

And we are finished. The answer is contained in A and Q registers. In this case the answer is 1111 0001 which is -15 in 2's complement notation. This is of course the correct answer.

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 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: [2]

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:

0: do nothing

+1: add b

-1: subtract b

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

$$\begin{aligned}
 & (a_{-1} - a_0) \times b \times 2^0 \\
 & + (a_0 - a_1) \times b \times 2^1 \\
 & + (a_1 - a_2) \times b \times 2^2
 \end{aligned}$$

.....

$$+ (a_{29} - a_{30}) \times b \times 2^{30}$$

$$+ (a_{30} - a_{31}) \times b \times 2^{31}$$

We can simplify the sum by noting that

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

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

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

The long formula in parentheses to the right of the first multiply operation is simply the two's complement representation of a . 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 .

Another example of 4-bit two's complement Booth's algorithm. Compute $2 \times (-3) = -6$ or 0010×1101 .

Iteration	Step	Multiplicand	Product C
0	initial value	0010	0000 1101 0
1	1: Ph = Ph-M 2: arithmetic shift	0010	1110 1101 0 1111 0110 1
2	1: Ph = Ph+M 2: arithmetic shift	0010	0001 0110 1 0000 1011 0
3	1: Ph = Ph-M 2: arithmetic shift	0010	1110 1011 0 1111 0101 1
4	1: do nothing 2: arithmetic shift	0010	1111 0101 1 1111 1010 1

The result 1111 1010 is 8-bit two's complement value of -6.

Example: Multiply 14 times -5 using 5-bit numbers (i.e 10-bit result). [3]

14 in binary: 01110

-5 in binary: 11011

-14 in binary: 10010 (so we can add when we need to subtract the multiplicand)

Iteration	Step	Multiplicand	Product C
0	initial value	01110	00000 11011 0
1	1: Ph = Ph-M 2: arithmetic shift	01110	10010 11011 0 11001 01101 1
2	1: do nothing 2: arithmetic shift	01110	11001 01101 1 11100 10110 1
3	1: Ph = Ph+M 2: arithmetic shift	01110	01010 10110 1 00101 01011 0
4	1: Ph = Ph-M 2: arithmetic shift	01110	10111 01011 0 11011 10101 1
5	1: do nothing 2: arithmetic shift	01110	11011 10101 1 11101 11010 1

The result 11101 11010 is 10-bit two's complement value of -70.

Answer the following questions:

1. Calculate 5×-3 using Booth's multiplication algorithm.

Ans : Refer page no 3

2. Show that booth's algorithm holds good for 2's complement method. How it is different from normal multiplication method?

Ans : Refer page no 5

3. Multiply 14 times -5 using 5-bit numbers using booth's multiplication algorithm.

Ans : Refer page no 6