WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Català
Čeština
Deutsch
Español
فارسی
Français
Italiano
Latviešu
日本語
Português
Русский
Српски / srpski
Українська
中文
Edit links

Booth's multiplication algorithm

From Wikipedia, the free encyclopedia

Booth's multiplication 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 of interest in the study of computer architecture.

Contents [hide]

1 The algorithm
2 A typical implementation
3 Example
4 How it works
5 See also
6 References
7 Further reading
8 External links

The algorithm [edit]

Booth's algorithm examines adjacent pairs of bits of the *N*-bit multiplier *Y* in signed two's complement representation, including an implicit bit below the least significant bit, $y_{-1} = 0$. For each bit y_i, for *i* running from 0 to *N*-1, the bits y_i and y_{i-1} are considered. Where these two bits are equal, the product accumulator *P* is left unchanged. Where $y_i = 0$ and $y_{i-1} = 1$, the multiplicand times 2^i is added to *P*; and where $y_i = 1$ and $y_{i-1} = 0$, the multiplicand times 2^i is subtracted from *P*. The final value of *P* is the signed product.

The multiplicand and product are not specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well. As stated here, the order of the steps is not determined. Typically, it proceeds from LSB to MSB, starting at $i = 0$; the multiplication by 2^i is then typically replaced by incremental shifting of the *P* accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest *N* bits of *P*.[1] There are many variations and optimizations on these details.

The algorithm is often described as converting strings of 1's in the multiplier to a high-order +1 and a low-order –1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

A typical implementation [edit]

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values *A* and *S* to a product *P*, then performing a rightward arithmetic shift on *P*. Let **m** and **r** be the multiplicand and multiplier, respectively; and let *x* and *y* represent the number of bits in **m** and **r**.

1. Determine the values of *A* and *S*, and the initial value of *P*. All of these numbers should have a length equal to $(x + y + 1)$.
   1. A: Fill the most significant (leftmost) bits with the value of **m**. Fill the remaining $(y + 1)$ bits with zeros.
   2. S: Fill the most significant bits with the value of (−**m**) in two's complement notation. Fill the remaining $(y + 1)$ bits with zeros.
   3. P: Fill the most significant *x* bits with zeros. To the right of this, append the value of **r**. Fill the least significant (rightmost) bit with a zero.
2. Determine the two least significant (rightmost) bits of *P*.
   1. If they are 01, find the value of $P + A$. Ignore any overflow.
   2. If they are 10, find the value of $P + S$. Ignore any overflow.
   3. If they are 00, do nothing. Use *P* directly in the next step.

    4. If they are 11, do nothing. Use *P* directly in the next step.
3. Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let *P* now equal this new value.
4. Repeat steps 2 and 3 until they have been done *y* times.
5. Drop the least significant (rightmost) bit from *P*. This is the product of **m** and **r**.

Example [edit]

Find 3 × (−4), with **m** = 3 and **r** = −4, and *x* = 4 and *y* = 4:

- m = 0011, -m = 1101, r = 1100
- A = 0011 0000 0
- S = 1101 0000 0
- P = 0000 1100 0
- Perform the loop four times :
    1. P = 0000 110**0 0**. The last two bits are 00.
        - P = 0000 0110 0. Arithmetic right shift.
    2. P = 0000 011**0 0**. The last two bits are 00.
        - P = 0000 0011 0. Arithmetic right shift.
    3. P = 0000 001**1 0**. The last two bits are 10.
        - P = 1101 0011 0. P = P + S.
        - P = 1110 1001 1. Arithmetic right shift.
    4. P = 1110 100**1 1**. The last two bits are 11.
        - P = 1111 0100 1. Arithmetic right shift.
- The product is 1111 0100, which is −12.

The above mentioned technique is inadequate when the multiplicand is most negative number that can be represented (e.g. if the multiplicand has 4 bits then this value is −8). One possible correction to this problem is to add one more bit to the left of A, S and P. This then follows the implementation described above, with modifications in determining the bits of A and S; e.g., the value of **m**, originally assigned to the first *x* bits of A, will be assigned to the first *x*+1 bits of A. Below, we demonstrate the improved technique by multiplying −8 by 2 using 4 bits for the multiplicand and the multiplier:

- A = 1 1000 0000 0
- S = 0 1000 0000 0
- P = 0 0000 0010 0
- Perform the loop four times :
    1. P = 0 0000 001**0 0**. The last two bits are 00.
        - P = 0 0000 0001 0. Right shift.
    2. P = 0 0000 000**1 0**. The last two bits are 10.
        - P = 0 1000 0001 0. P = P + S.
        - P = 0 0100 0000 1. Right shift.
    3. P = 0 0100 000**0 1**. The last two bits are 01.
        - P = 1 1100 0000 1. P = P + A.
        - P = 1 1110 0000 0. Right shift.
    4. P = 1 1110 000**0 0**. The last two bits are 00.
        - P = 1 1111 0000 0. Right shift.
- The product is 11110000 (after discarding the first and the last bit) which is −16.

How it works [edit]

Consider a positive multiplier consisting of a block of 1s surrounded by 0s. For example, 00111110. The product is given by :

$$M \times {}''0\;0\;1\;1\;1\;1\;1\;0'' = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$$

where M is the multiplicand. The number of operations can be reduced to two by rewriting the same as

$$M \times {}''0\;1\;0\;0\;0\;0\text{-}1\;0'' = M \times (2^6 - 2^1) = M \times 62.$$

In fact, it can be shown that any sequence of 1's in a binary number can be broken into the difference of two binary numbers:

$$(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots)_2 \equiv (\ldots 1\overbrace{0\ldots 0}^{n}0\ldots)_2 - (\ldots 0\overbrace{0\ldots 1}^{n}0\ldots)_2.$$

Hence, we can actually replace the multiplication by the string of ones in the original number by simpler operations, adding the multiplier, shifting the partial product thus formed by appropriate places, and then finally subtracting the multiplier. It is making use of the fact that we do not have to do anything but shift while we are dealing with 0s in a binary multiplier, and is similar to using the mathematical property that 99 = 100 − 1 while multiplying by 99.

This scheme can be extended to any number of blocks of 1s in a multiplier (including the case of a single 1 in a block). Thus,

$$M \times ''0\,0\,1\,1\,1\,0\,1\,0'' = M \times (2^5 + 2^4 + 2^3 + 2^1) = M \times 58$$
$$M \times ''0\,1\,0\,0\text{-}1\,1\text{-}1\,0'' = M \times (2^6 - 2^3 + 2^2 - 2^1) = M \times 58.$$

Booth's algorithm follows this scheme by performing an addition when it encounters the first digit of a block of ones (0 1) and a subtraction when it encounters the end of the block (1 0). This works for a negative multiplier as well. When the ones in a multiplier are grouped into long blocks, Booth's algorithm performs fewer additions and subtractions than the normal multiplication algorithm.

See also [edit]

- Multiplication ALU
- Non-adjacent form
- Redundant binary representation

References [edit]

1. ^ Chi-hau Chen (1988). *Signal processing handbook*. CRC Press. p. 234. ISBN 978-0-8247-7956-6.

Further reading [edit]

1. Andrew D. Booth. A signed binary multiplication technique. The Quarterly Journal of Mechanics and Applied Mathematics, Volume IV, Pt. 2, 1951 [1]
2. Collin, Andrew. Andrew Booth's Computers at Birkbeck College. *Resurrection*, Issue 5, Spring 1993. London: Computer Conservation Society.
3. Patterson, David and John Hennessy. *Computer Organization and Design: The Hardware/Software Interface, Second Edition*. ISBN 1-55860-428-6. San Francisco, California: Morgan Kaufmann Publishers. 1998.
4. Stallings, William. *Computer Organization and Architecture: Designing for performance, Fifth Edition*. ISBN 0-13-081294-3. New Jersey: Prentice-Hall, Inc.. 2000.

External links [edit]

- Radix-4 Booth Encoding
- Radix-8 Booth Encoding in A Formal Theory of RTL and Computer Arithmetic
- Booth's Algorithm JavaScript Simulator
- Implementation in Python

Categories: 1950 introductions | 1950 in London | 1950 in science | Binary arithmetic | Computer arithmetic | Multiplication | Birkbeck, University of London