**Multipliers**

**Booth’s Multiplication Algorithm:**

Booth’s multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two’s complement notation. The algorithm examines adjacent pairs of bits of an -bit multiplier in signed two’s complement notation, including an implicit bit before the least significant bit (LSB), . For each bit , for running from 0 to , the bits and are considered. When these two bits are equal, the product accumulator is left unchanged. When and , the multiplicand times is added to . When and , the multiplicand times is subtracted from . The final value of is the signed product.

Booth’s algorithm can be implemented by repeatedly adding one of two predetermined values and to a product , then performing an arithmetic shift to the right on . Let and be the multiplicand and multiplier, respectively, and let and represent the number of bits in and , respectively.

1. Determine the values of and , and the initial value of . All of these numbers should have a length of .
   1. : Fill the most significant bits (MSBs) with the value of . Fill the remaining bits with zeros.
   2. : Fill the MSBs with the value of in two’s complement notation. Fill the remaining bits with zeros.
   3. : Fill the most significant bits with zeros. To the right of this, append the value of . Fill the LSB with a zero.
2. Determine the two LSBs of .
   1. If they are 01, find the value of . Ignore any overflow.
   2. If they are 10, find the value of . Ignore any overflow.
   3. If they are 00, do nothing. Use directly in the next step.
   4. If they are 11, do nothing. Use directly in the next step.
3. Arithmetically shift the value obtained in the second step by a single place to the right. Let now equal this new value.
4. Repeat steps 2 and 3 until they have been done times.
5. Drop the LSB from . This is the product of and .

Example:

Find the product of , with and .

Perform the loop four times (since ):

1. . The last two bits are 00.
   * . Arithmetic right shift.
2. . The last two bits are 00.
   * . Arithmetic right shift.
3. . The last two bits are 10.
   * . .
   * . Arithmetic right shift.
4. . The last two bits are 11.
   * . Arithmetic right shift.

Therefore, the product is 1111 0100, which is -12.

Booth’s algorithm is inadequate when the multiplicand (i.e. ) is the most negative number that can be represented (e.g. if the multiplicand has 4 bits then this value is -8). One possible correction is to add one more bit to the left of , , and . This then follows the same algorithm, with modifications in determining the bits of , , and . Namely, the value will be assigned to the first bits of , the value of will be assigned to the first bits of , and the MSBs of are filled with zeros for its initial value.

**Wallace Tree:**

A Wallace tree is an efficient hardware implementation of a digital circuit that multiplies two integers. It has three steps:

1. Multiply each bit of one of the arguments, by each bit of the other, yielding results, where is the number of bits of the argument. Depending on the position of the multiplied bits, the wires carry different weights. For example, the weight of a bit carrying the result of is 128 ().
2. Reduce the number of partial products to two by layers of full and half adders.
3. Group the wires in two numbers and add them.

An additional layer is added at the second step as long as there are three or more wires with the same weight. The second step works as follows:

* Take any three wires with the same weights and input them into a full adder. The result will be an output wire of the same weight (i.e., the sum bit) and an output wire with a weight double that of each of the three input wires (i.e., the carry bit).
* If there are two wires of the same weight left, feed them into a half adder.
* If there is one wire left, connect it to the next layer.

The weight of a wire is the radix (in base 2) of the digit that the wire carries. Therefore, the weight of .

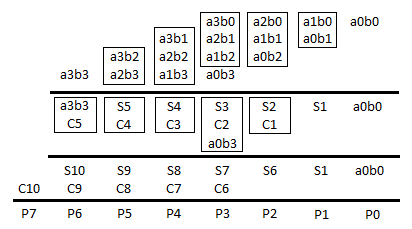
The main advantage of the Wallace tree is that there is only a reduction stage delay of the order O(log n) and each layer has O(1) propagation delay. The partial products have complexity O(1) while the final addition has complexity O(log n).

Example:

Given , find the product of and .

1. Multiply every bit of one argument by every other bit of the other argument:
   * Weight-1:
   * Weight-2: ,
   * Weight-4: , ,
   * Weight-8: , , ,
   * Weight-16: , ,
   * Weight-32: ,
   * Weight-64:
2. Reduction layer 1:
   * Pass , , and through.
   * Add a half adder for and . Outputs: S1 = and C1.
   * Add a full adder for , , and . Outputs: S2 = and C2.
   * Add a full adder for , , and . Outputs: S3 = and C3.
   * Add a full adder for , , and . Outputs: S4 = and C4.
   * Add a half adder for and . Outputs: S5 = and C5.
3. Reduction layer 2:
   * Pass and S1 through.
   * Add a half adder for S2 and C1. Outputs: S6 = S2 + C1 and C6.
   * Add a full adder for , S3, and C2. Outputs: S7 = + S3 + C2 and C7.
   * Add a half adder for S4 and C3. Outputs: S8 = S4 + C3 and C8.
   * Add a half adder for S5 and C4. Outputs: S9 = S5 + C4 and S9.
   * Add a half adder for and C5. Outputs: S10 = + C5 and C6.
4. Now that only sum and carry bits remain, group the wires into a pair of integers and add them. The result will be 8 bits wide.

All calculations listed above are visualized in Figure 1 below.

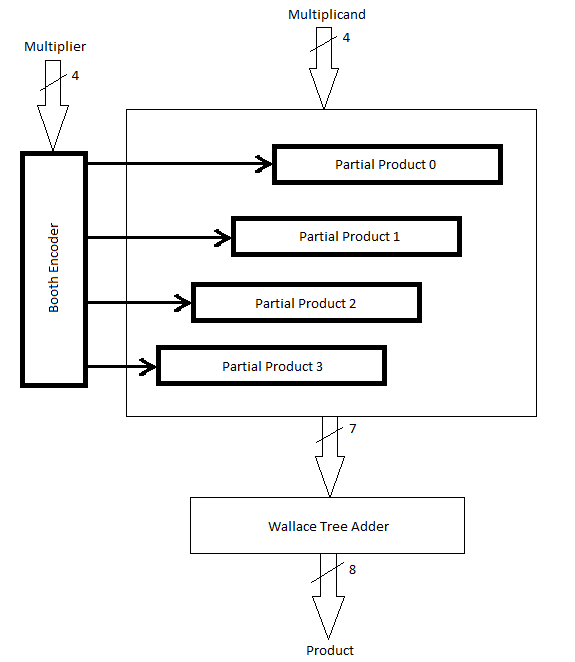


**Figure 1:** A 4x4 Wallace tree.

**Combined Booth-Wallace Multiplier:**

While Booth’s algorithm handles signed integer multiplication very well, it is prone to delays as the argument sizes increase. This can be fixed by combining Booth encoding, the first part of the algorithm, with an efficient multiplier like the Wallace tree. This way, the Wallace tree also has the ability of supporting signed integer multiplication.

The block diagram of a 4x4 combined Booth-Wallace multiplier is shown in Figure 2 below.



**Figure 2: The block diagram of a 4x4 Booth-Wallace multiplier.**

Given a 4-bit multiplicand and a 4-bit multiplier , the partial products are calculated according to the following encoding scheme.

|  |  |  |
| --- | --- | --- |
|  |  | Partial Product |
| 0 | 0 | 0\* |
| 0 | 1 | 1\* |
| 1 | 0 | -1\* |
| 1 | 1 | 0\* |

Note that this is the radix-2 Booth encoding scheme introduced when describing Booth’s multiplication algorithm.

The modified radix-2 Booth encoding scheme is useful for higher-bit multiplication. Its encoding scheme is as follows.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | Partial Product |
| 0 | 0 | 0 | 0\* |
| 0 | 0 | 1 | 1\* |
| 0 | 1 | 0 | 1\* |
| 0 | 1 | 1 | 2\* |
| 1 | 0 | 0 | -2\* |
| 1 | 0 | 1 | -1\* |
| 1 | 1 | 0 | -1\* |
| 1 | 1 | 1 | 0\* |

In order to iterate through the multiplier, the LSB must be padded with a zero to the right, just like in the previous encoding scheme. If the number of bits to be multiplied is even, the MSB should be padded with two zeros to the left. If the number of bits to be multiplied is odd, one zero should be padded to the left of the MSB. Note that this encoding scheme produces only partial products, where is the number of bits of the multiplier, while the normal radix-2 Booth encoder produces partial products.