A Comparative Analysis of Booth's and Modified Booth's Algorithms for Binary Multiplication

Theysen Feldbaumer, Ethan Heckart

Department of Electrical and Computer Engineering, Missouri University of Science and Technology

COMP ENG 3110: Computer Organization and Design

Dr. Alireza Hurson

April 12, 2024

I. Abstract

This paper presents a comprehensive comparative analysis of Booth's algorithm and Modified Booth's algorithm for binary multiplication, implemented in and evaluated using the C++ programming language. In this research, we delve into the number of iterations and quantify the amount of additions, subtractions, and execution times involved in the multiplication process.

II. Introduction Motivation

Binary multiplication is a fundamental operation in computer arithmetic, essential for various applications. Booth's algorithm and Modified Booth's algorithm are two widely used methods for binary multiplication. The motivation behind this research is to understand their efficiency in terms of computational resources and time.

III. Simulator Technical Descriptions

The simulators for Booth's and Modified Booth's algorithms operate on binary numbers stored as strings. They have both been implemented with the intention of mimicking the setup of a physical ALU.

In the Booth's algorithm simulator, the multiplier and multiplicand are passed into the function, which then initializes an accumulator as zeros and an extended bit as a zero. Next, an iteration over the multiplier and extended bit commences. During this iteration, the least significant bit of multiplier and the extended bit value are observed. If both values are the same, the simulator will perform an arithmetic right shift through the accumulator, the multiplier, and the extended bit, then the iteration will repeat. If the least significant bit of the multiplier is zero and the extended bit is one, a binary addition will occur that adds the value of the multiplicand to the accumulator, then an arithmetic right shift through the accumulator, the multiplier, and the extended bit is performed, followed by the iteration repeating. If the least significant bit of the multiplier is a one and the extended bit is a zero, the multiplicand value is subtracted from the accumulator. In the simulator, this is accomplished by finding the two's complement of the multiplicand and then adding it to the accumulator. This iteration will repeat as many times as the number of bits in the multiplicand. Once all the iterations have finished, the result of the multiplication is stored as the combination of the final accumulator and multiplier bits, in that order.

The Modified Booth's algorithm simulator is initialized in the same manner as the Booth's algorithm simulator, with the important addition of the multiplicand being widened by one bit. The reason for this will be explained in an upcoming paragraph. Next, an iteration begins comparing the two least significant bits of the multiplier and the extended bit.

In situations where the second least significant bit is zero, the following logic is employed: if the least significant bit of the multiplier and the extended bit are both zero, two consecutive arithmetic right shifts are performed through the accumulator, multiplier, and extended bit, then the iteration repeats. If the least significant bit of the multiplier and the extended bit are different (zero and one or one and zero), the multiplicand is added to the accumulator using the same logic as the Booth's simulator and two consecutive arithmetic right shifts are performed through the accumulator, multiplier, and extended bit, then the iteration repeats. If the least significant bit of the multiplier and the extended bit are both one, two times the value of the multiplicand is added to the accumulator. There are many ways to achieve such an outcome. In this simulator, the value of two times the multiplicand is generated by performing a logical left shift on the number, with that value then being added to the accumulator. The multiplicand was given an extra bit so that no overflow issues are encountered during the logical left shift. After the addition finishes, two consecutive arithmetic right shifts are performed through the accumulator, multiplier, and extended bit, then the iteration repeats.

In situations where the second least significant bit of the multiplier is one, the following logic is employed: if the least significant bit of the multiplier and the extended bit are both one, two consecutive arithmetic right shifts are performed through the accumulator, multiplier, and extended bit, then the iteration repeats. If the least significant bit of the multiplier and the extended bit have different values (zero and one or one and zero), the multiplicand is subtracted from the accumulator using the same logic as the Booth's simulator and two consecutive arithmetic right shifts are performed through the accumulator, multiplier, and extended bit, then the iteration repeats. If the least significant bit of the multiplier and the extended but are both zero, two times the value of the multiplicand is subtracted from the accumulator using the same logic as above, and two consecutive arithmetic right shifts are performed through the accumulator, multiplier, and extended bit, then the iteration repeats.

This iteration will repeat as many times as half the number of bits in the multiplier. Once all the iterations have finished, the result of the multiplication is stored as the combination of the final accumulator and multiplier bits, in that order. The result from Modified Booth's will sometimes be a wider binary number than the result from Booth's algorithm as a consequence of the widened multiplicand; however, regardless of length, the integer representation of the number is always the same between the two algorithms.

Algorithm 1: Booth's algorithm pseudocode

```
Input: multiplier, multiplicand
Output: Result
accumulator = '0' * multiplier.length();
ext_bit = '0';
for 0 to multiplier.length - 1 do
    if multiplier.lsb() == '1' and ext_bit == '0' then
        accumulator.binarySub(multiplicand);
    else
        if multiplier.lsb() == '0' and ext_bit == '1' then
            accumulator.binaryAdd(multiplicand);
        end
    end
    arithmeticSHR(accumulator, multiplier, ext_bit);
end
Result = accumulator + multiplier;
```

Algorithm 2: Modified Booth's algorithm pseudocode

```
Input: multiplier, multiplicand
Output: Result
multiplicand.addBit();
accumulator = '0' * multiplier.length();
ext_bit = '0';
for 0 to multiplier.length() / 2 do
    if multiplier.secondlsb() == '0' then
        if multiplier.lsb() == '1' then
            if ext_bit == '0' then
                accumulator.binaryAdd(multiplicand);
            end
            else
                accumulator.binaryAdd(multiplicand * 2);
            end
        end
        if multiplier.lsb() == '0' and ext_bit == '1' then
            accumulator.binaryAdd(multiplicand);
        end
    end
    if multiplier.secondlsb() == '1' then
        if multiplier.lsb() == '0' then
            if ext_bit == '1' then
                accumulator.binarySub(multiplicand);
            end
            else
                accumulator.binarySub(multiplicand * 2);
            end
        end
        if multiplier.lsb() == '1' and ext_bit == '0' then
            accumulator.binarySub(multiplicand);
        end
    end
    arithmeticSHR(accumulator, multiplier, ext_bit);
    arithmeticSHR(accumulator, multiplier, ext_bit);
end
Result = accumulator + multiplier;
```

IV. Simulation results

Booth's Algorithm Results

Multiplier	Multiplicand	Multiplication Result (Bin)	Hex	Iterations	Additions	Subtractions
1110	1111	0b00000010	0x02	4	0	1
0101	0000	0b00000000	0x00	4	2	2
111111	111111	0b000000000001	0x001	6	0	1
101110	110111	0b000010100010	0x0A2	6	1	2
111011	100011	0b000010010001	0x091	6	2	1
00011111	01010101	0b0000101001001011	0x0A4B	8	1	1
11010111	01010101	0b1111001001100011	0xF263	8	2	3
01010101	11010111	0b1111001001100011	0xF263	8	4	4
01110111	00110011	0b0001011110110101	0x17B5	8	2	2
00000000	01110111	0b0000000000000000	0x0000	8	0	0
0101010101	0101010101	0b00011100011000111001	0x1C639	10	5	5
1100111011	1001110000	0b00010011001111010000	0x133D0	10	2	3
1001101110	0101111010	0b11011010111001101100	0xDAE6C	10	2	3
010101010101	010101010101	0b000111000110111000111001	0x1C6E39	12	6	6
001111100111	000000000000	0b000000000000000000000000	0x000000	12	2	2
101010101010	101010101010	0b000111000111100011100100	0x1C78E4	12	5	6
111001110000	000011111111	0b111111100111000110010000	0xFE7190	12	1	2

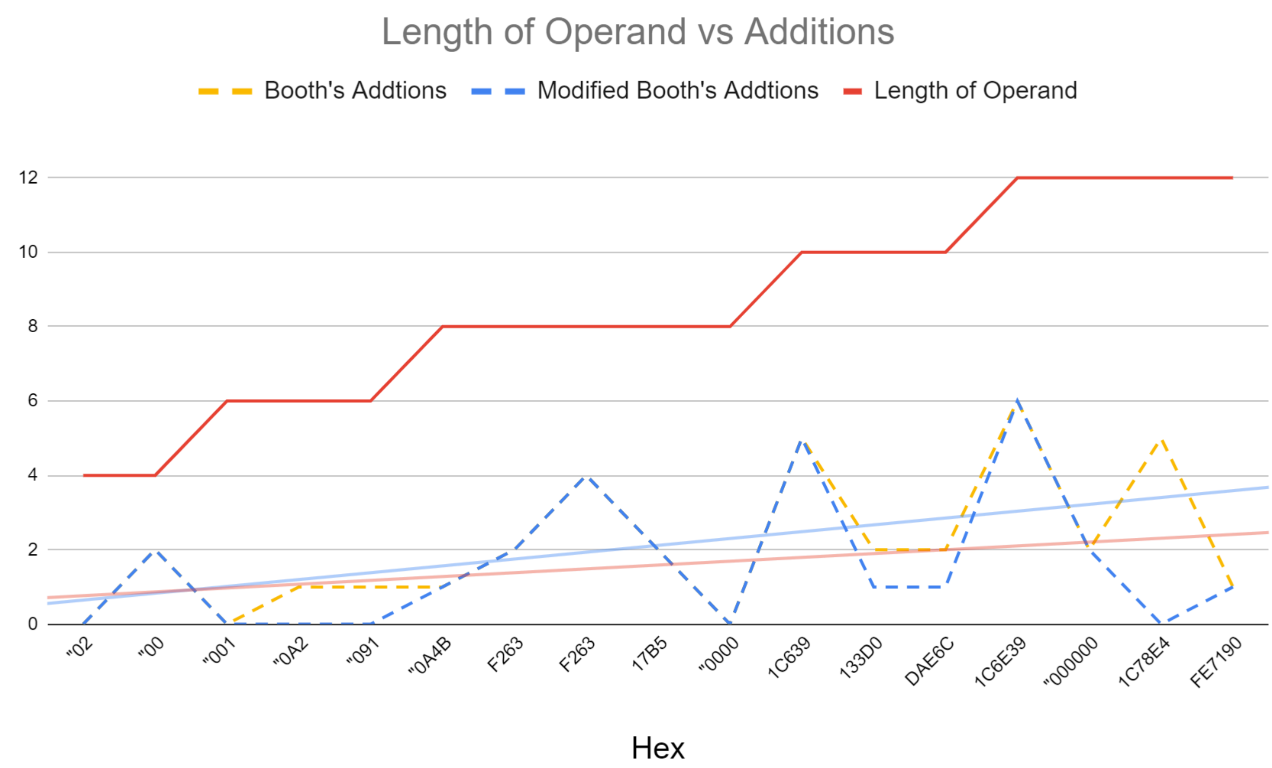
- *This table displays sample multipliers and multiplicands, as used in testing. It shows multiplication results in binary and hex. Finally, the number of iterations, additions, and subtractions that the program performed are displayed. This table is run using Booth's algorithm.*

Modified Booth's Algorithm Results

Multiplier Multiplicand	Multiplication Result (Bin)	Hex	Iterations	Additions	Subtractions
1110 1111	0b00000010	0x02	2	0	1
0101 0000	0b00000000	0x00	2	2	0
111111 111111	0b000000000001	0x001	3	0	1
101110 110111	0b000010100010	0x0A2	3	0	2
111011 100011	0b000010010001	0x091	3	0	2
00011111 01010101	0b0000101001001011	0x0A4B	4	1	1
11010111 01010101	0b1111001001100011	0xF263	4	2	2
01010101 11010111	0b1111001001100011	0xF263	4	4	0
01110111 00110011	0b0001011110110101	0x17B5	4	2	2
00000000 01110111	0b0000000000000000	0x0000	4	0	0
0101010101 0101010101	0b00011100011000111001	0x1C639	5	5	0
1100111011 1001110000	0b00010011001111010000	0x133D0	5	1	3
1001101110 0101111010	0b11011010111001101100	0xDAE6C	5	1	3
010101010101 010101010101	0b000111000110111000111001	0x1C6E39	6	6	0
001111100111 000000000000	0b000000000000000000000000	0x000000	6	2	2
101010101010 101010101010	0b000111000111100011100100	0x1C78E4	6	0	6
111001110000 000011111111	0b111111100111000110010000	0xFE7190	6	1	2

❖ *As stated in the previous table, this table displays sample multipliers and multiplicands, as used in testing. It shows multiplication results in binary and hex. Finally, the number of iterations, additions, and subtractions that the program performed are displayed. This table showcases the results using Modified Booth's algorithm.*

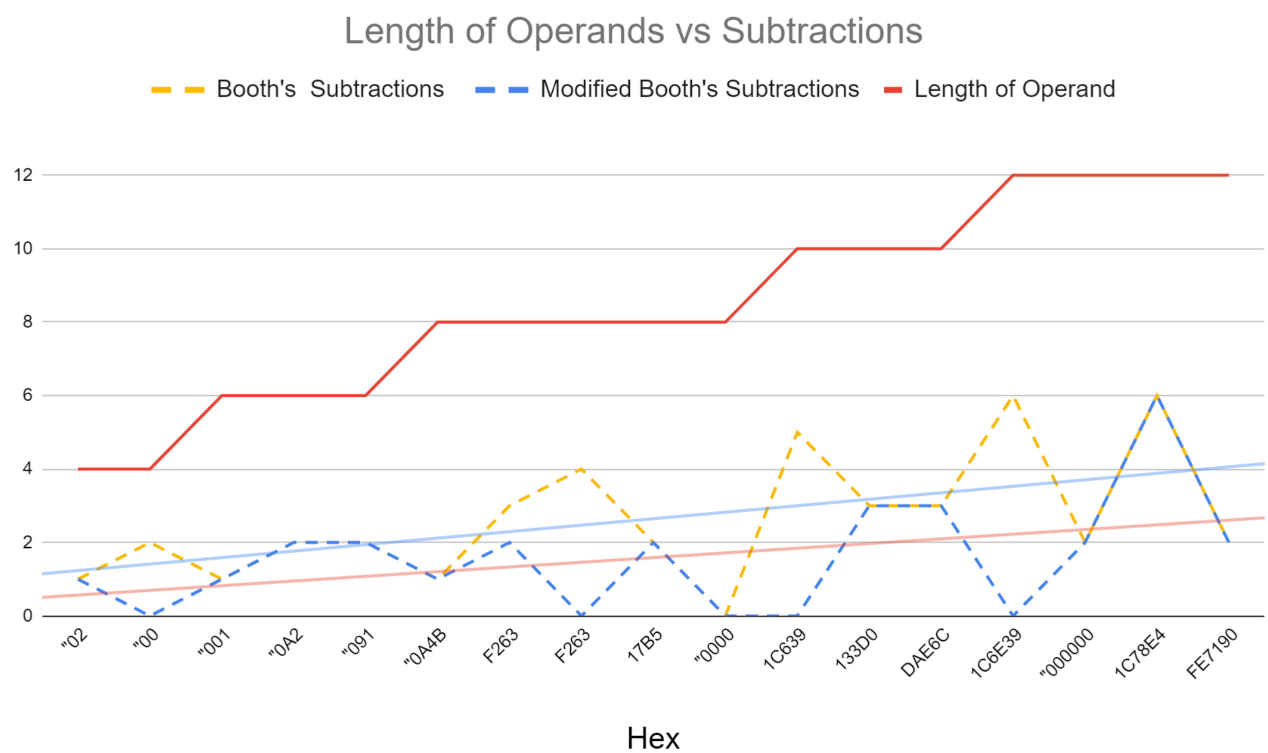
Length of Operand vs Addition



- *In this graph, the red line indicates the operand length for different multiplications. The dashed lines represent Booth's and Modified Booth's algorithm respectively.*

When we compare the length of the operand against the number of additions that are performed, we see a tendency for the number of additions to go up, as can be seen by the trendlines; this is the case for both Booth's and Modified Booth's algorithms. This is because there is more opportunity for additions and subtractions while the product is being generated, as the number of iterations increases.

Length of Operand vs Subtractions

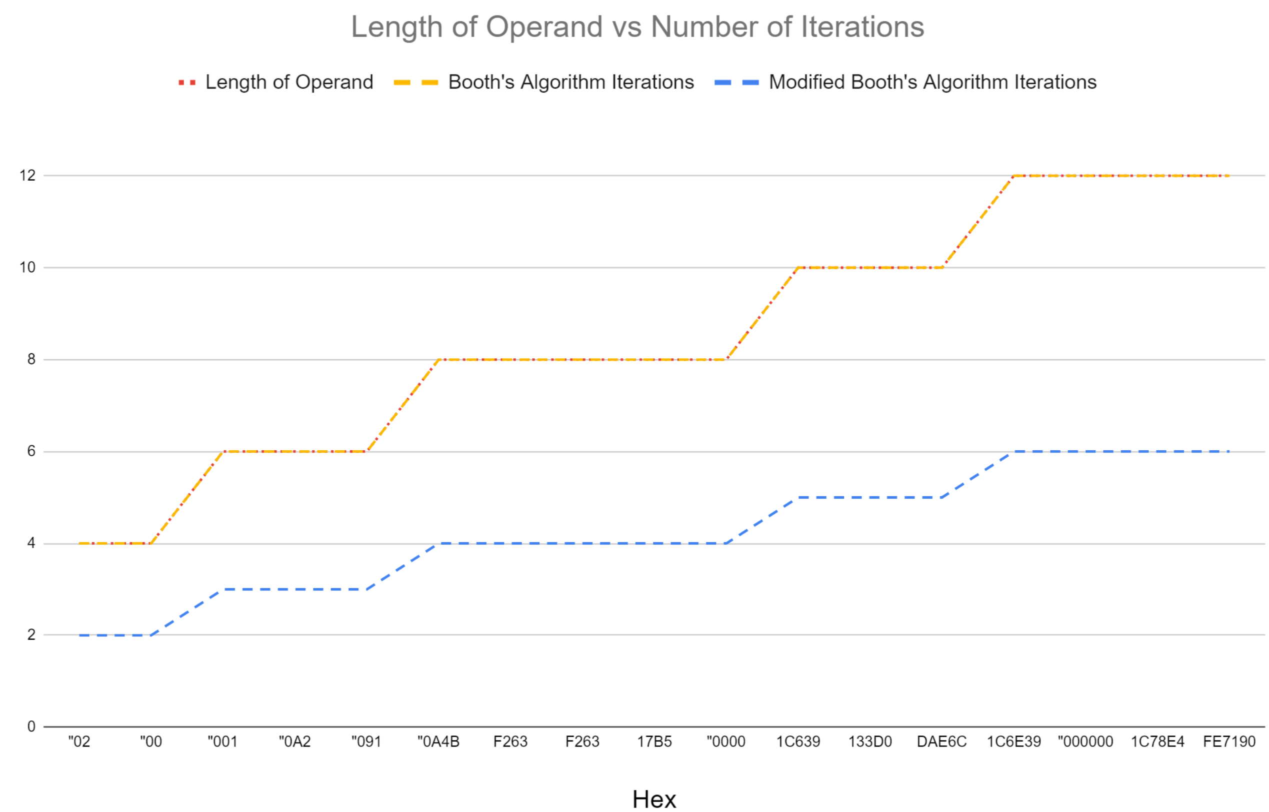


❖ *Once again, in this graph, the red line indicates the operand length for different multiplications. The dashed lines represent Booth's and Modified Booth's algorithm respectively.*

Similar to the number of addition operations, when viewing the length of the operand vs the number of subtractions performed, we see a steady increase in the trendline.

In both graphs, there is one instance where there are not any additions or subtractions. With Booth's algorithm, pairs of zeros (00) and pairs of ones (11) indicate that the program does not need to perform an addition or a subtraction. When the entire multiplier is continuous pairs of zeros or ones, it is a case where there will not be any additions or subtractions altogether. Likewise, with Modified Booth's algorithm pairs of three indicate that there are no additions or subtractions (000) and (111). In cases where there are not any additions or subtractions, the entire multiplier is all the same digit: all zeros or all ones.

Length of Operand vs Number of Iterations



❖ *In this graph the dotted red line represents the length of the operand and the dashed line represents the number of iterations in the two algorithms.*

In Booth's algorithm, these results show that as the length of the operand, for the given multiplicand and multiplier goes up, the number of iterations that the system needs to go through will go up at the same rate; this rate is perfectly matched. This can be seen in the upper yellow, dashed, line, which completely overlaps the operand length line. The number of iterations will always perfectly match the operand length. This is the iterative nature of how Booth's algorithm works. As the algorithm processes, the multiplier is scanned from left to right, one bit at a time; this forces each bit to correspond to its own iteration.

Modified Booth's algorithm differs from Booth's algorithm in this way, however. The encoding scheme is designed to minimize the number of partial products and iterations. The amount of iterations that are needed to be performed is always exactly half of the amount needed in Booth's algorithm; this can be seen in the bottom blue dashed line. In Modified Booth's algorithm, the multiplier is scanned in pairs, rather than individually. This allows most cases to take half the number of iterations and several to cut the amount of time for the algorithm to process in half also. We have furthered our research by showing this in the next section.

Timing of Booth's Algorithm and Modified Booth's Algorithm in Microseconds

Inputs	Booth's (microseconds)		Modified Booth's (microseconds)	
Multiplier / Multiplicand	Mean	Median	Mean	Median
1110 1111	0.516	0.507	0.585	0.563
0101 0000*	1.141	1.096	0.752	0.753
111111 111111	0.752	0.75	0.746	0.732
101110 110111	1.229	1.184	1.152	1.117
111011 100011	1.271	1.271	1.163	1.159
00011111 01010101	1.325	1.318	1.367	1.348
11010111 01010101	2.220	2.126	2.251	2.249
01010101 11010111*	3.271	3.313	1.815	1.81
01110111 00110011	1.852	1.811	2.059	2.048
00000000 01110111	0.584	0.575	0.596	0.58
0101010101 0101010101*	4.305	4.302	2.571	2.538
1100111011 1001110000	2.538	2.509	2.374	2.359
1001101110 0101111010	2.517	2.495	2.586	2.564
010101010101 010101010101*	6.286	6.198	3.419	3.389
001111100111 000000000000	2.593	2.582	2.845	2.812
101010101010 101010101010*	5.761	5.741	4.036	4.065
111001110000 000011111111	2.404	2.423	2.488	2.47

* no instances of two consecutive equal bits in the multiplier

In addition to comparing the number of iterations, additions, and subtractions each algorithm requires to complete the multiplication, the total time in microseconds needed to complete the multiplication was also recorded. For the purposes of this test, the program was compiled using G++ version 12.2.0 with all compiler optimizations turned off. Each test input was run 100,000 times on a Debian 12.5 system using an Intel Core i7-1165G7 processor.

For test inputs which used a multiplier with at least one instance of the same bit occurring back-to-back, Booth's algorithm typically completed faster than Modified Booth's algorithm. It is worth noting that in these cases, both algorithms typically finish within one-tenth of a microsecond of each other on average, regardless of the number of bits used in the test cases.

For test inputs that used a multiplier with no instances of the same bit occurring back-to-back, Modified Booth's algorithm universally performed better than Booth's algorithm. The time difference is significant from the perspective of microseconds, with Modified Booth's algorithm often finishing on the order of 1.5 times faster.

In situations where there are no instances of the same bit occurring back-to-back, both algorithms must perform an addition or subtraction during each iteration. Modified Booth's algorithm is able to benefit significantly from needing half the number of iterations to complete the multiplication (and thus half the number of addition and/or subtraction operations), which explains the execution speed increase.

V. Conclusion

For most multiplications, the performance differences between Booth's and Modified Booth's algorithms are negligible. There are certain situations, namely where there are no instances of consecutively repeating bits, where Modified Booth's algorithm performs notably better. It should also be noted that the logic of Modified Booth's algorithm is significantly more complex than that of Booth's algorithm.

In terms of real-world impact, over the lifetime of an ALU, there is a high chance that an implementation of Modified Booth's algorithm will spend less total time performing multiplications than an implementation of Booth's algorithm due to the significant performance gains in the situation explained above. In a physical ALU that implements a multiplication algorithm using logic gates, the cost (and even physical size) of implementing the two algorithms will likely be different.

A designer's choice between Booth's algorithm and Modified Booth's algorithm should encompass an analysis of the anticipated cost of implementation with the anticipated performance. In applications where execution speed is paramount, it would seem wise to implement Modified Booth's algorithm. In other applications, where it is desirable to balance price and performance, the performance losses of Booth's algorithm may be seen as worthwhile if it involves cost savings.