Analyzing Performance of Booth's Algorithm and Modified Booth's Algorithm

Brett Weiland

April 12, 2024

Abstract

In this paper, the performance of Booth's Algorithm is compared to modified Booth's Algorithm. Each multiplier is simulated in Python, and performance is observed by counting the number of add and subtract operations for various inputs. Results are analyzed and discussed to highlight the potential tradeoffs to consider when deciding how hardware multiplication is implimented.

Introduction

Multiplication is among the most time consuming mathematical operations for processors. In many applications, the time it takes to multiply dramatically influences the speed of the program. Applications of digital signal processing (such as audio modification and image processing) require constant multiply and accumulate operations for functions such as fast fourier transformations and convolutions. Other applications are heavily dependent on multiplying large matrices, such as machine learning, 3D graphics and data analysis. In such scenarios, the speed of multiplication is vital. Consequently, most modern processors implement hardware multiplication. However, not all multiplication circuits are equal; there is often a stark contrast between performance and hardware complexity. To further complicate things, multiplication circuits perform differently depending on what numbers are being multiplied.

Algorithm Description and Simulation

Booth's algorithim computes the product of two signed numbers in two's compliment format. To avoid overflow, the result is placed into a register two times the size of the operands (or two registers the size of a single operand). Additionally, the algorithim must work with a space that is exended one bit more then the result. For the purpose of brevity, the result register and extra bit will be refered to as the workspace, as the algorithim will use this space for its computations. First, the multiplier is placed into the workspace and shifted left by 1. From there, an operation is performed based off the last two bits, as shown by the following table:

Bit 1	Bit 0	Action
0	0	None
0	1	Add
1	0	Subtract
1	1	None

After all iterations are complete, the result is arithmaticlly shifted once to the left, and the process repeats for the number of bits in an operand.

Modified booth's algorithim functions similar to Booth's algorithim, but checks the last *three* bits instead. As such, there are a larger selection of actions for each iteration:

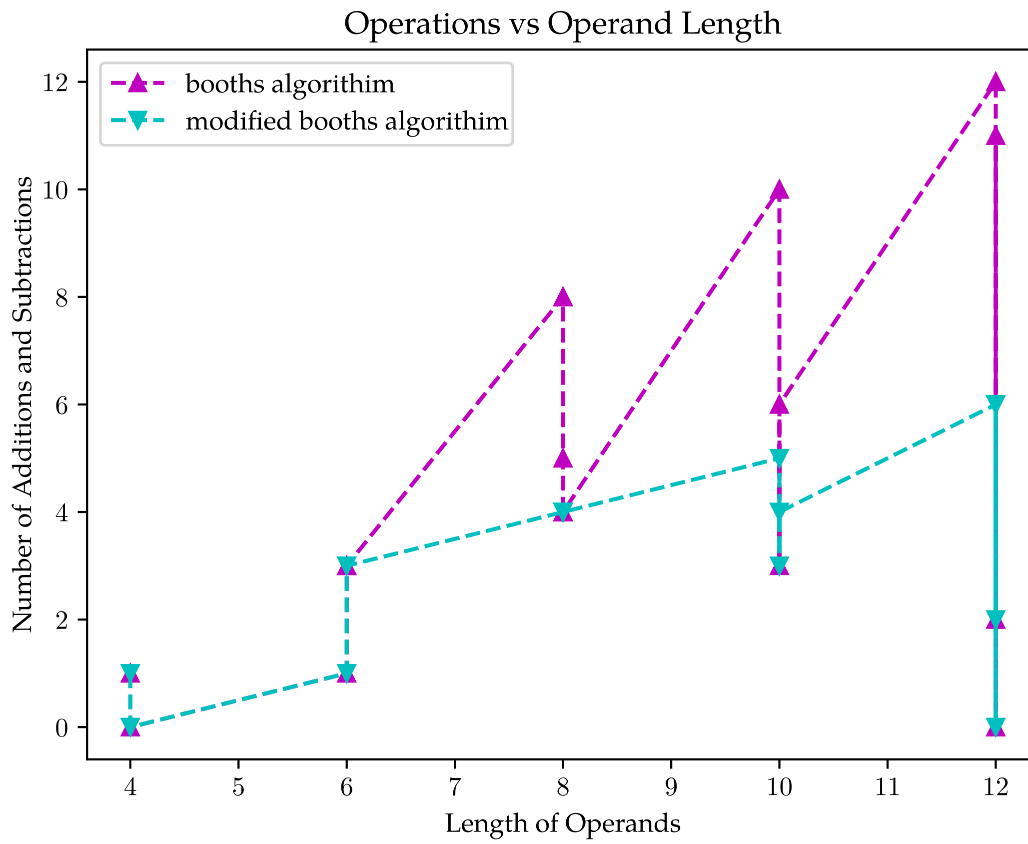
Bit 2	Bit 1	Bit 0	Action
0	0	0	None
0	0	1	Add
0	1	0	Add
0	1	1	Add $\times 2$
1	0	0	Sub $\times 2$
1	0	1	Sub
1	1	0	Sub
1	1	1	None

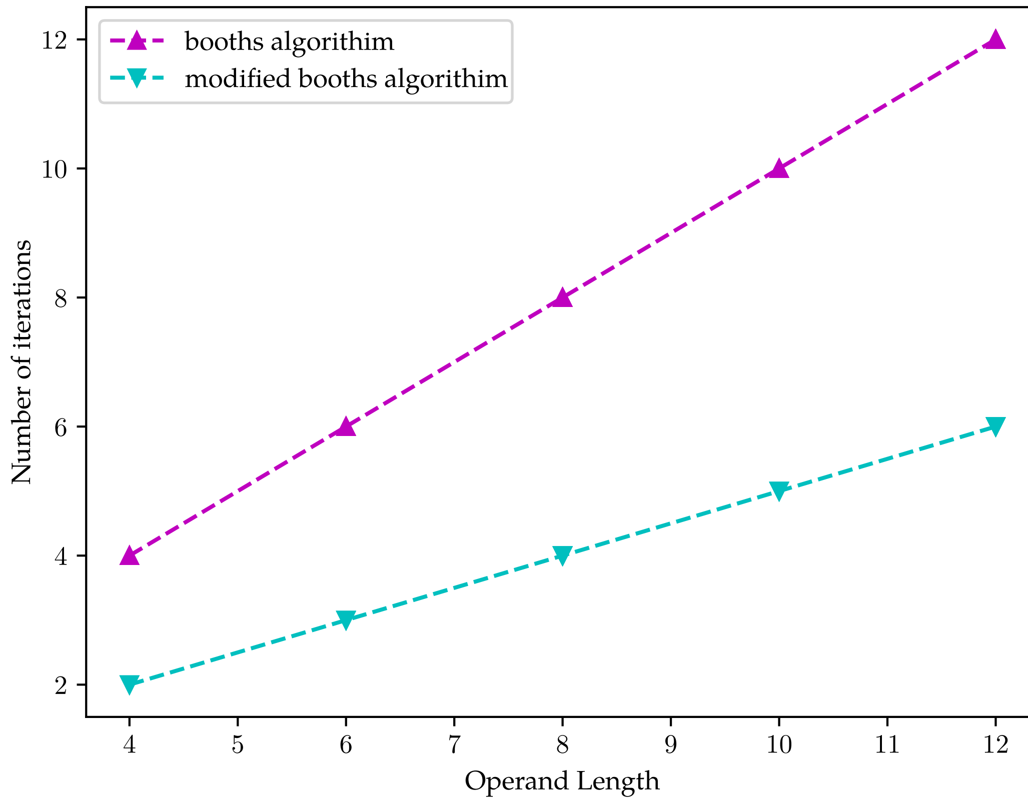
Because some operations require multiplying the multiplicand by 2, an extra bit is added to the most significant side of the workspace to avoid overflow. After each iteration, the result is arithmaticlly shifted right twice. The number of iterations is only half of the length of the operands. After all iterations, the workspace is shifted right once, and the second most significant bit is set to the first most significant bit as the result register does not include the extra bit.
The purpose of this paper is to analyze and compare the peformance of these two algorithms for various operand lengths and values. As such, all arithmatic bitwise operations had to account for the length of operand sizes. Take for example, the arithmatic shift right functions:

put phseudo code here

Additionally, after each iteration, the bits more significant then the workspace length had to be erased (the bitwise functions purposefully do not account for this).

Results





multiplicand	multiplier	length	booth	modified booth
0b1110	0b1111	4	1	1
0b101	0b0	4	0	0
0b111111	0b111111	6	1	1
0b101110	0b110111	6	3	3
0b111011	0b100011	6	3	3
0b11111	0b1010101	8	8	4
0b11010111	0b1010101	8	8	4
0b1010101	0b11010111	8	5	4
0b1110111	0b110011	8	4	4
0b0	0b1110111	8	4	4
0b101010101	0b101010101	10	10	5
0b1100111011	0b1001110000	10	3	3
0b1001101110	0b101111010	10	6	4
0b10101010101	0b10101010101	12	12	6
0b1111100111	0b0	12	0	0
0b101010101010	0b101010101010	12	11	6
0b111001110000	0b11111111	12	2	2

multiplicand	multiplier	result (bin)	result (hex)
0b1110	0b1111	0b10	0x2
0b101	0b0	0b0	0x0
0b111111	0b111111	0b1	0x1
0b101110	0b110111	0b10100010	0xa2
0b111011	0b100011	0b10010001	0x91
0b11111	0b1010101	0b101001001011	0xa4b
0b11010111	0b1010101	0b1111001001100011	0xf263
0b1010101	0b11010111	0b1111001001100011	0xf263
0b1110111	0b110011	0b1011110110101	0x17b5
0b0	0b1110111	0b0	0x0
0b101010101	0b101010101	0b11100011000111001	0x1c639
0b1100111011	0b1001110000	0b10011001111010000	0x133d0
0b1001101110	0b101111010	0b11011010111001101100	0xdae6c
0b10101010101	0b10101010101	0b111000110111000111001	0x1c6e39
0b1111100111	0b0	0b0	0x0
0b101010101010	0b101010101010	0b111000111100011100100	0x1c78e4
0b111001110000	0b11111111	0b111111100111000110010000	0xfe7190

Analysis

Conclusion