Jonah Yates
Professor Hurson
CE 3110 – Comp Org. & Design
11 April 2022

***Abstract* –** This paper will explore Booth's algorithm. Booths Algorithm is a multiplication algorithm designed by Andrew Donald Booth in the 1950s. The algorithm is a method for performing multiplication on two binary numbers within an arithmetic logic unit (ALU). Its relevance is primarily due to the performance advantages it provides compared to the traditional add-and-shift since instead of checking one bit of the multiplier each iteration, the algorithm checks two bits and then uses a coding scheme. This makes Booths faster than add and shift on average since fewer additions are performed on average. The instance where add and shift would be faster than Booth's is when we are dealing with alternating 0s and 1s within the multiplier.

**I. Introduction & Motivation –** In computer systems, the arithmetic logic unit is the component tasked with performing the arithmetic and logic operations in instructions. Since many instructions computers are tasked with performing is math or logic-based, it is important to minimize the time impact the ALU is having on code execution. This leads to the need to have quick algorithms that perform a minimal number of operations to compute the desired data output. In terms of multiplication, there are several algorithms that can be used: repeated addition, add and shift, and Booth's algorithms just to name a few. For improving the speed of each algorithm, the designer implements shortcuts. Like the add-and-shift algorithm, Booth's algorithm works with signed magnitude numbers and the number of iterations is due to the length of the register the numbers are stored in. However, what we will be exploring in this project is the additional shortcut Booth employs: the ability to skip a series of 0s or 1s each iteration. We are curious about how the input impacts the number of additions and subtractions performed.

**II. Code Explanation –** In the project, Booth's algorithm was implemented in C++. The project begins by declaring the empty vectors: *AC*, *q_multiplier*, and *multiplicand*. The user is then prompted for the multiplicand and multiplier. As we receive each number, *length* is computed and stored, the number is verified to be >= 4 bits and <= 12 bits in length, contains only 0s or 1s and finally, the string is parsed, and each bit added to the corresponding integer vector. The Booth's algorithm function is then called and passed *q_multiplier*, *multiplicand*. Within the algorithm function, we will first compute and store the 2s complement of *multiplicand* since subtracting the multiplicand can instead be performed via adding the 2s complement. Furthermore, we will use the fact that Booth's algorithm runs for *length* iterations. From here, in each iteration, we will look at the last 2 bits of *AC Q q0* and use the following coding scheme to decode the bits and execute the corresponding action.

00 – no action
01 – add multiplicand
10 – sub multiplicand (add 2s complement)
11 – no action

Before the end of each iteration, an arithmetic shift right is also performed. This is also the reason why vectors were used; vectors allow indexing at certain positions

and updating values, making shifting easy. Finally, once we exit the loop the result which is in *AC Q* is returned.

**III. Psuedocode**

Below is the pseudocode of the project.

```
main
{
    get multiplier input
    verify multiplier is valid input
    // acceptable length, only 0s or 1s, ect
    get multiplicand input
    verify multiplicand is valid input

    // numAdd and numSub
    // are passed by reference
    boothsAlg(multiplier, multiplicand, &
    numAdd, & numSub)
    output result and num adds and subs

}


boothsAlg()
{
    compute 2s comp of multiplicand
    // copy contents into new vector, flip
    // bits, then add 1

    create AC vector and fill with 0s
    for loop, length times
    {
        decode = "Q[length-1]" + "q0"
        if ("01"), add multiplicand to AC
        if ("10"), add 2s comp of
        multiplicand to AC
        arithmetic right shift AC Q q0
    }
    return string of AC Q

}
```

```
addBinary(AC, B)
{
    carryIn = 0
    for length-1 down to and including 0
    {
        temp = AC[i] XOR B[i] XOR carryIn
        if (AC[i] XOR B[i] XOR carryIn >= 2)
           carry = 1
        else
           carry = 0
        AC[i] = temp

    }

}
```

**IV. Simulation Results & Analysis –** After running the sample test data through the program, the following data was generated. Based on the data in Figure A2, we can see that increasing the bit length of the operands does not guarantee the number of additions and subtractions to increase. In the case of $E70 * $0FF, despite the multiplicand and multiplier being 12 bits each, the number of additions and subtractions combined was only 3. Compared to $55 * $D7, an 8-bit multiplier and multiplicand, yet had 8 total additions and subtractions, we can deduce longer operands do not equate to more additions and subtractions being performed. Despite this, it is important to note though that on average the number of additions and subtractions does increase. This is because the upper bound for the number of additions and subtractions is 2*len. Thus, in the worst case, $555 * $555, when we have sequences of alternating 0s and 1s, we will be adding or subtracting each iteration. In the graph, we can see the number of iterations is the same value as the operand length and the number of additions and subtractions is also a positive linear relationship.

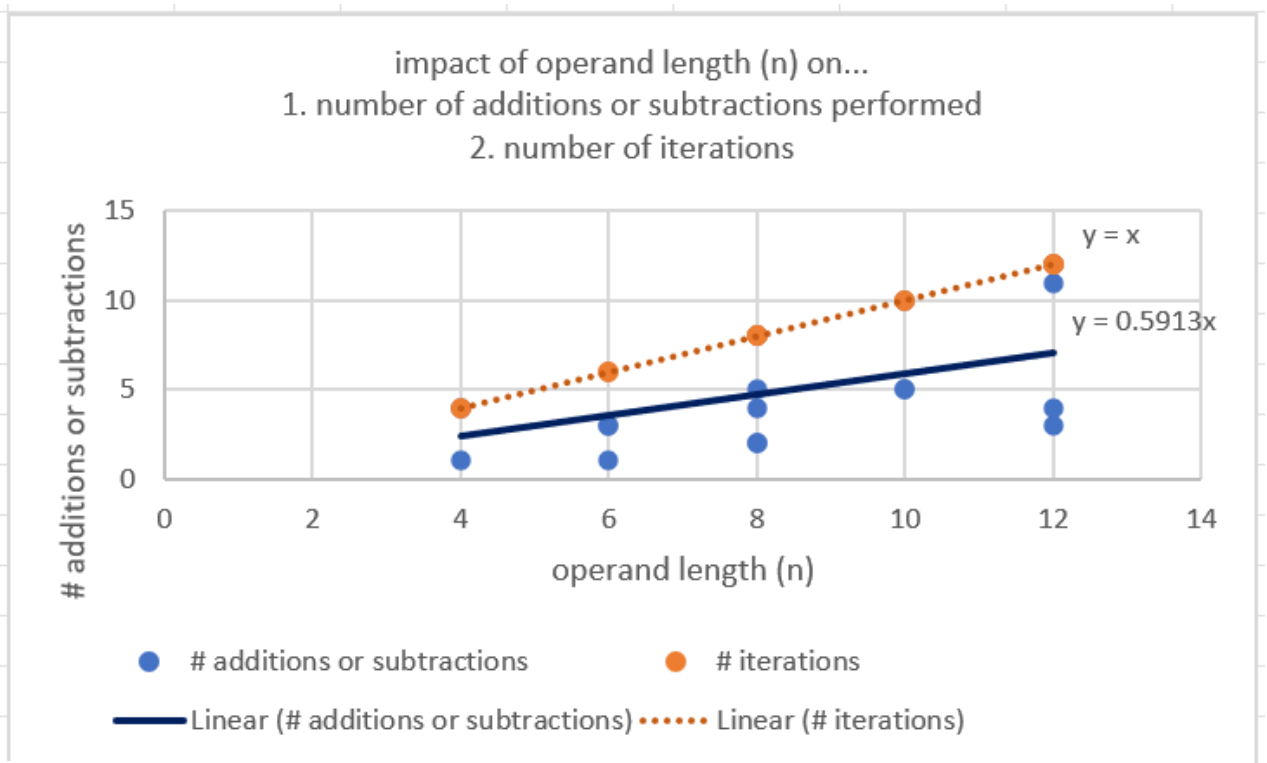


Figure A1. Graph showing the impact operand length (n) has on the number of additions and subtractions performed as well as how many iterations the algorithm needed for computation.

| len (n) | Multiplier | Multiplicand | AC Q (result) | # Adds | # Subs |
|---|---|---|---|---|---|
| 4 | $E = 1110 | $F = 1111 | $02 = 00000010 | 0 | 1 |
| 4 | $5 = 0101 | $5 = 0101 | $19 = 00011001 | 2 | 2 |
| 6 | $3F = 111111 | $3F = 111111 | $001 = 000000000001 | 0 | 1 |
| 6 | $2E = 101110 | $37 = 110111 | $0A2 = 000010100010 | 1 | 2 |
| 6 | $3B = 111011 | $23 = 100011 | $091 = 000010010001 | 1 | 2 |
| 8 | $1F = 00011111 | $55 = 01010101 | $A4B = 0000101001001011 | 1 | 1 |
| 8 | $D7 = 11010111 | $55 = 01010101 | $F263 = 1111001001100011 | 2 | 3 |
| 8 | $55 = 01010101 | $D7 = 11010111 | $F263 = 1111001001100011 | 4 | 4 |
| 8 | $77 = 01110111 | $33 = 00110011 | $17B5 = 0001011110110101 | 2 | 2 |
| 8 | $78 = 01111000 | $77 = 01110111 | $37C8 = 0011011111001000 | 1 | 1 |
| 10 | $155 = 0101010101 | $155 = 0101010101 | $1C639 = 00011100011000111001 | 5 | 5 |
| 10 | $33B = 1100111011 | $270 = 1001110000 | $133D0 = 00010011001111010000 | 2 | 3 |
| 10 | $26E = 1001101110 | $17A = 0101111010 | $DAE6C = 11011010111001101100 | 2 | 3 |
| 12 | $555 = 010101010101 | $555 = 010101010101 | $1C6E39 = 000111000110111000111001 | 6 | 6 |
| 12 | $3E7 = 001111100111 | $0FF = 000011111111 | $03E319 = 000000111110001100011001 | 2 | 2 |
| 12 | $AAA = 101010101010 | $AAA = 101010101010 | $1C78E4 = 000111000111100011100100 | 5 | 6 |
| 12 | $E70 = 111001110000 | $0FF = 000011111111 | $FE7190 = 111111100111000110010000 | 1 | 2 |

Figure A2. Table of the number of additions, subtractions, and iterations based on given test data.

**V. Conclusion –** Throughout the project and implementation of Booth's algorithm, I was able to visually see the details we covered in class regarding its characteristics. Most notably was how Booth runs faster due to it having less add and shifts on average. Through inputting various test data, it became clear that alternating 0s and 1s in the binary number lead to more addition and subtraction operations being performed. Furthermore, the project provided the opportunity to practice writing algorithms

allowing for a deeper understand of their inner workings. The next step in learning

about multiplication algorithms would be to implement the repeated addition and add-and-shift algorithms and then test data on all three simultaneously. Regardless, this project provided a great opportunity in learning more about algorithmic changes that can be made to ALU's to increase computation speed.

**Appendix A –** Figures & Resources

A.1 – graph outlining impact of operand length on the number of additions and subtractions, and iterations.

A.2 – table showing output as well as number of additions and subtractions generated by the program based on certain binary input.

A.3 – simulator code can be found at: https://git-classes.mst.edu/jynqz/booths-algorithm