Report on Booth's Algorithm Simulator

by Khai Thieu
Missouri University of Science and Technology

ABSTRACT

Binary multiplication is one of the basic operations in digital systems. The most simple approach to achieve this is through the add-and-shift method. However, this method lacks efficiency. A more efficient alternative is Booth's algorithm. Compared to add-and-shift, Booth's algorithm performs better than average by skipping over sequences of 1's. Booth's algorithm can be modified to further improve efficiency by allowing it to examine three bits at each iteration. In theory, this modification leads to a reduction of iterations by half. This report focuses on the developed simulator to simulate Booth's and modified Booth's multiplication techniques. By examining these algorithms, we aim to gain a better understanding of the differences in performance between these two algorithms. The results show that the modified Booth's algorithm outperforms the basic Booth's algorithm. On average, the modified Booth's algorithm requires half the number of iterations and the amount of addition/subtractions. This increase in efficiency also comes with some extra overheads since additional codes are required.

INTRODUCTION

Binary multiplication is one of the basic operations in a digital system. One of the simplest methods to achieve this is through a technique called add-and-shift. In add-and-shift, multiplication is performed as a sequence of additions, with the number of iterations equal to the length of the multiplicand. However, this method is not that efficient. A more efficient solution for multiplication is Booth's algorithm. This algorithm works with numbers in 2's complement format and is not compatible with numbers in signed magnitude or 1's complement format. Booth's algorithm enables faster multiplication on average compared to add-and-shift. This efficiency is achieved by allowing the algorithm to skip over sequences of 1's. In Booth's algorithm, each iteration involves four possible bit patterns (00, 01, 10, and 11). Depending on the bit pattern, an additional action, like adding or subtracting, is performed followed by a right shift operation. In terms of hardware implementation, Booth's algorithm shares a similar configuration to add-and-shift, where two single-register operands are multiplied, and the result is generated in a double register. Booth's algorithm can be modified to examine three bits during each iteration instead of two, resulting in Booth's modified algorithm. This modification leads to eight possible bit patterns during each iteration. Depending on the bit pattern, an additional action, like adding or subtracting, may be required before shifting the

combination of (AC Q) registers two positions to the right in each iteration. This modification to the Booth's algorithm enables skipping over sequences of consecutive 1s and reduces the number of iterations by half. This report aims to develop a simulator to mimic the behavior and analyze the performance of both Booth and modified Booth multiplication techniques. By exploring these algorithms, we seek to gain insights into their efficiency and prove that the modified version of Booth's algorithm will lead to a more efficient algorithm.

STRUCTURE OF THE CODE

The simulator is written in Python and there are three files: main.py, get_input.py, and Booth.py.

- The **main.py** file is responsible for calling the appropriate functions to execute Booth's algorithm.
- The **get_input.py** file manages user input retrieval from the keyboard. It receives input and validates it to ensure accurate data entry.
- The **booth.py** file contains functions necessary for executing Booth's algorithm, such as arithmetic shift right and addition.

To run the simulator, there are multiple possible ways depending on how you installed Python and your computer's configuration. Here are some commands you can type into the terminal to run the simulator:

- python3 main.py
- py -3 main.py
- python main.py

PSEUDO CODE

main.py file

- Booth(user_input)
    - Params: user's input in the format of a list of two strings [multiplicand, multiplier]
    - Return: None
    - Functionality:
        1. Create and initialize necessary variables
        2. Use second_com() function to convert B to its 2's complement and store it in its own variable

3. Check the last bit of the multiplier and the extended bit.
    - Following the bit patterns from Booth's algorithm to call appropriate functions:
        - 00 or 11 ⇒ call shift_right() function
        - 01 ⇒ call addition() function then call shift_right() function
        - 10 ⇒ call addition() function but use the second complement of B as one of the params then call shift_right() function
4. Repeat step 3 n times, where n is the length of the multiplicand. While the loop is running, keep track of the number of iterations and the number of additions/subtractions.
5. Print the result.

- Modified_Booth(user_input)
    - Params: user's input in the format of a list of two strings [multiplicand, multiplier]
    - Return: None
    - Functionality:
        1. Create and initialize necessary variables
        2. Use second_comp() function to convert B to its 2's complement and store it in its own variable
        3. Check the last bit of the multiplier and the extended bit.
            - Following the bit patterns from Booth's algorithm to call appropriate functions:
                - 000 or 111 ⇒ call shift_right() function twice
                - 001 or 010 ⇒ call addition() function then call shift_right() function twice
                - 011 ⇒ call addition() function twice then call shift_right() function twice
                - 100 ⇒ call addition() function twice but use the second complement of B as one of the params then call shift_right() function twice
                - 101 or 110 ⇒ call addition() function but use the second complement of B as one of the params then call shift_right() function twice
        4. Repeat step 3 n times, where n is the length of the multiplicand. While the loop is running, keep track of the number of iterations and the number of additions/subtractions.

5. Print the result.

get_input.py file

- contains_other_characters(user_input)
    - Params: a string of user input
    - Return: True if the input string has a character that is not 0 or 1
    - Functionality: It checks each character to ensure that they are either 0 or 1
- validate_input(numbers, user_input)
    - Params:
        - numbers: the list that stores the multiplicand and multiplier
        - user_input: the most recent input
    - Return: True if the input is in the correct format.
    - Functionality: It checks for the correct length, calls the contains_other_characters() function to ensure input is binary, and checks if the multiplicand and multiplier are of the same length.
- get_input()
    - Params: None
    - Return: a list that stores the multiplicand and multiplier
    - Functionality: It asks the user to enter two inputs.

booth.py file

- shift_right(A, Q)
    - Params:
        - A: Accumulator
        - Q: Multiplier
    - Return: a list that stores the accumulator, multiplier, and extended bit
    - Functionality: performs arithmetic shift right
        1. Combines A and Q into one string.
        2. Store the first digit of A and add it to front the string. (arithmetic shift right keeps the sign bit)
        3. Break the string into an array of accumulator, multiplier, and extended bit
- addition(num1, num2)
    - Params: two strings of binary numbers
    - Return: a string of the sum of the binary addition
    - Functionality: add two binary numbers
        1. Initialize an array the size of the multiplicand with zero's. This variable will store the result of the addition
        2. Initialize the carry bit with 0

3. Compare the last bit of both multiplicand and multiplier from right to left
    - if the bits are 0 and 0
        - if carry-in is 0, result bit will be 0 and carry is 0
        - if carry-in is 1, result bit will be 1 and carry is 0
    - if the bits are 1 and 1
        - if carry-in is 0, result bit will be 0 and carry is 1
        - if carry-in is 1, result bit will be 1 and carry is 1
    - if the bits are 0 and 1 or 1 and 0
        - if carry-in is 0, result bit will be 1 and carry is 0
        - if carry-in is 1, result bit will be 0 and carry is 1
4. Combines the result array into a string and returns the result

- second_comp(num)
    - Params: a string of binary number
    - Return: a string of the second complement of that number
    - Functionality: converts the binary number into its second complement form
        1. Loop through the binary number bit-by-bit and convert 0 to 1 and 1 to 0
        2. call the addition(num1, num2) function to add that first complement number with 1 to get the second complement
        3. return the second complement number

TEST RUNS

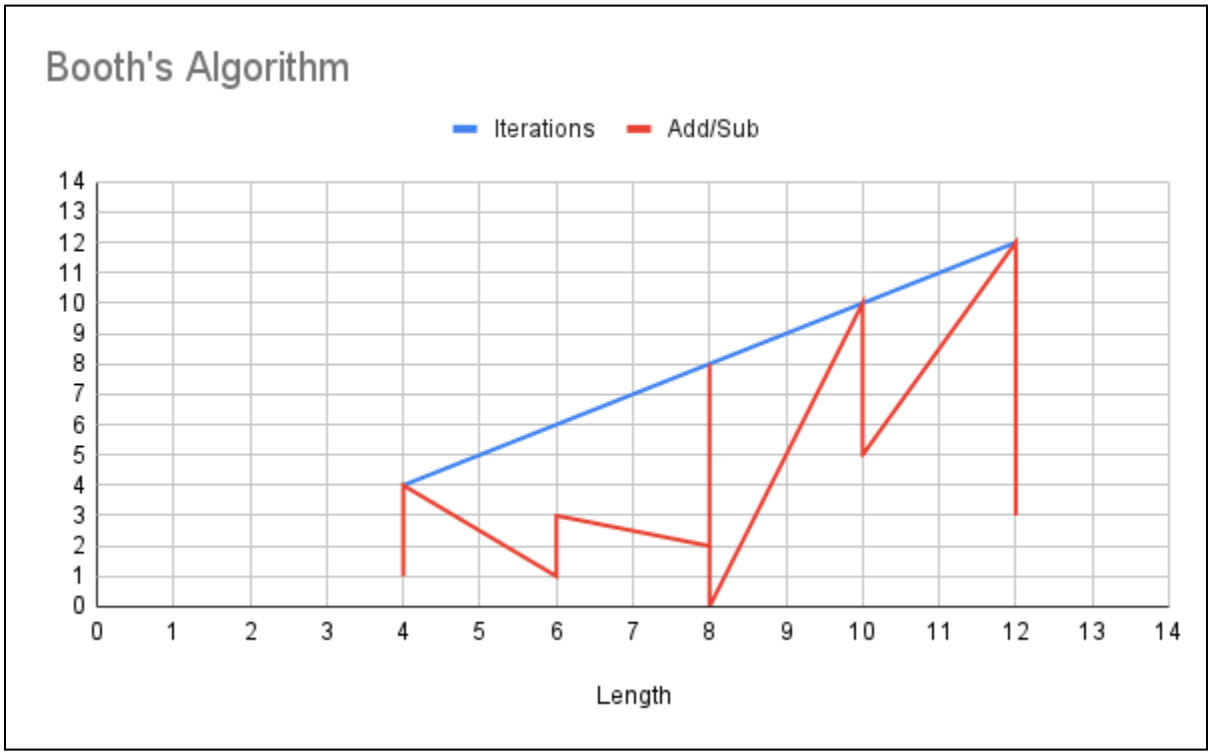
Booth's Algorithm				
Input		**Output**		
Multiplier	Multiplicand	AC	Q	Hex
1110	1111	0000	0010	2
0101	0000	0000	0000	0
111111	111111	000000	000001	1
101110	110111	000010	100010	A2
111011	100011	000010	010001	91
00011111	01010101	00001010	01001011	A4B
11010111	01010101	11110010	01100011	F263

01010101	11010111	11110010	01100011	F263
01110111	00110011	00010111	10110101	17B5
00000000	01110111	00000000	00000000	0
0101010101	0101010101	0001110001	1000111001	1C639
1100111011	1001110000	0001001100	1111010000	133D0
1001101110	0101111010	1101101011	1001101100	DAE6C
010101010101	010101010101	000111000110	111000111001	1C6E39
001111100111	000000000000	000000000000	000000000000	0
101010101010	101010101010	000111000111	100011100100	1C78E4
111001110000	000011111111	111111100111	000110010000	FE7190

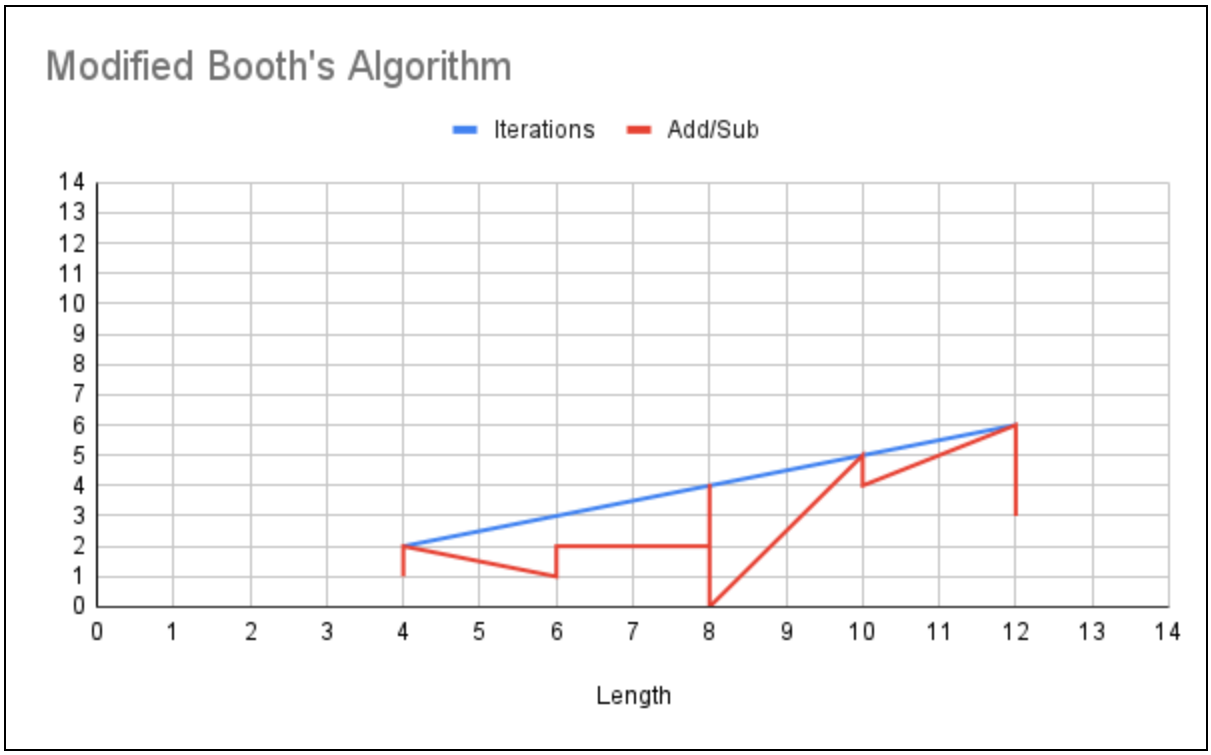
Modified Booth's Algorithm				
Input		**Output**		
Multiplier	Multiplicand	AC	Q	Hex
1110	1111	0000	0010	2
0101	0000	0000	0000	0
111111	111111	000000	000001	1
101110	110111	000010	100010	A2
111011	100011	111110	010001	F91
00011111	01010101	11111010	01001011	FA4B
11010111	01010101	11101110	01100011	EE63
01010101	11010111	11110010	01100011	F263
01110111	00110011	00010111	10110101	17B5
00000000	01110111	00000000	00000000	0

0101010101	0101010101	0001110001	1000111001	1C639
1100111011	1001110000	0001001100	1111010000	133D0
1001101110	0101111010	0000101100	1001101100	B26C
010101010101	010101010101	000111000110	111000111001	1C6E39
001111100111	000000000000	000000000000	000000000000	0
101010101010	101010101010	000111000110	100011100100	1C68E4
111001110000	000011111111	111111100111	000110010000	FE7190

ANALYSIS



Performance of Booth's Algorithm



Performance of Modified Booth's Algorithm

In Booth's algorithm, the number of iterations is equal to the length of the input. On average, the number of additions/subtractions is approximately half of the number of iterations. In the modified Booth's algorithm, the number of iterations is equal to half the length of the input. Similarly to the basic Booth's algorithm, on average, the number of additions/subtractions is approximately half of the number of iterations. Based on those results, it's evident that the modified Booth's algorithm outperforms the basic Booth's algorithm. By modifying the algorithm to examine three bits at a time, it has effectively halved, on average, the number of iterations and the amount of addition/subtractions required.

CONCLUSION

In conclusion, the implementation of the simulator explored the efficiency of binary multiplication techniques of Booth's algorithm and its modified version. In theory, Booth's basic algorithm lacks efficiency compared to its modified version. The modifications made to Booth's algorithm are believed to result in a reduction of iterations and the number of additions/subtractions by half and the results proved it. The results demonstrated that the modified Booth's algorithm outperformed the basic Booth's algorithm on all the test runs. The modified algorithm significantly reduced the number of iterations and the amount of

addition/subtractions required by half, leading to better efficiency. Although the modified algorithm is more efficient in terms of performance, it does come with some additional overheads in terms of mode lines of code. Overall, the findings proved the correctness of the theory and the importance of algorithmic optimization in digital systems, where even minor modifications can result in an improvement in performance.

APPENDIX

Code for the main.py file

```
from get_input import get_input
from booth import *

# performs the basic Booth's algorithm
def Booth(user_input):
  B = user_input[0]
  Q = user_input[1]
  size = len(B)
  AC = "0" * size
  e = "0"
  second_comp_B = second_comp(B)
  add_sub = 0

  for i in range(size):
    if Q[size - 1] == e:
      shift = shift_right(AC, Q)
      AC = shift[0]
      Q = shift[1]
      e = shift[2]
    else:
      if Q[size - 1] == "0" and e == "1":
        AC = addition(B, AC)
      else:
        AC = addition(second_comp_B, AC)

      shift = shift_right(AC, Q)
      AC = shift[0]
      Q = shift[1]
```

```
      e = shift[2]

      add_sub += 1

  print("=== Booth's RESULT ===")
  print(f"AC: {AC}, Q: {Q}")
  print(f"Number Iterations: {size}")
  print(f"Number ADD/SUB: {add_sub}")
  print()

# performs the modified Booth's algorithm
def Modified_Booth(user_input):
  B = user_input[0]
  Q = user_input[1]
  size = len(B)
  AC = "0" * size
  e = "0"
  second_comp_B = second_comp(B)
  shift_num = 0
  add_sub = 0
  iteration = 0

  while shift_num < size:
    if Q[size - 1] == Q[size - 2] and Q[size - 2] == e:
      shift = shift_right(AC, Q)
      shift = shift_right(shift[0], shift[1])
      AC = shift[0]
      Q = shift[1]
      e = shift[2]
    elif Q[size - 2] == "0":
      AC = addition(B, AC)

      if Q[size - 1] == e:
        AC = addition(B, AC)

      shift = shift_right(AC, Q)
      shift = shift_right(shift[0], shift[1])
      AC = shift[0]
      Q = shift[1]
      e = shift[2]
```

```
      add_sub += 1
    else:
      AC = addition(second_comp_B, AC)

      if Q[size - 1] == e:
        AC = addition(second_comp_B, AC)

      shift = shift_right(AC, Q)
      shift = shift_right(shift[0], shift[1])
      AC = shift[0]
      Q = shift[1]
      e = shift[2]

      add_sub += 1

    shift_num += 2
    iteration += 1

  print("=== Modified Booth's RESULT ===")
  print(f"AC: {AC}, Q: {Q}")
  print(f"Number Iterations: {iteration}")
  print(f"Number ADD/SUB: {add_sub}")
  print()

if __name__ == "__main__":
  user_input = get_input()
  Booth(user_input)
  Modified_Booth(user_input)
```

Code for the booth.py file

```
# arithmetic shift right
def shift_right(A, Q):
  result = []
  AQ = A + Q
  left_bit = A[0]
  AQe = left_bit + AQ
```

```
  result.append(AQe[0:len(A)]) # AC
  result.append(AQe[len(A):len(AQe) - 1]) # Q
  result.append(AQe[len(AQe) - 1]) # Extended bit

  return result

# add two binary numbers
def addition(num1, num2):
  sum = [0] * len(num1)
  c = 0

  for i in range(len(num1)):
    index = len(num1) - i - 1

    if num1[index] == num2[index]:
      if num1[index] == "0":
        if c == 1:
          sum[index] = 1
          c = 0
      else:
        if c == 0:
          c = 1
        else:
          sum[index] = 1
          c = 1
    else:
      if c == 0:
        sum[index] = 1
      else:
        c = 1

  for i in range(len(sum)):
    sum[i] = str(sum[i])

  return "".join(sum)

# converts binary number to its second complement
def second_comp(num):
  result = ["0"] * len(num)
  one = ["0"] * (len(num) - 1)
```

```
  one.append("1")

  # converts to 1's complement
  for i in range(len(num)):
    if num[i] == "0":
      result[i] = "1"
    else:
      result[i] = "0"

  return addition("".join(result), "".join(one))
```

Code for the get_input.py file

```
# check if input contain anything other than 0 and 1
def contains_other_characters(user_input):
  for char in user_input:
    if char not in ['0', '1']:
      return True

  return False

# ensure the input binary numbers are valid
def validate_input(numbers, user_input):
  valid = False

  if len(user_input) < 4 or len(user_input) > 12: # check length
    print("\nINVALID INPUT! - Input must have a length of (4 ≤ length ≤
12)\n")
  elif contains_other_characters(user_input): # make sure input is binary
    print("\nINVALID INPUT! - Input can only contain 0 and 1\n")
  elif len(numbers) == 1 and len(numbers[0]) != len(user_input): # make
sure length of Multiplicand and Multiplier are equal
    print("\nINVALID INPUT! - Multiplicand and Multiplier must be the same
length\n")
  else:
    valid = True

  return valid
```

```
# get 2 input binary numbers from user
def get_input():
  numbers = []

  while len(numbers) < 2:
    user_input = input("Enter the first number (Multiplicand): ") if
len(numbers) == 0 else input("Enter the second number (Multiplier): ")

    if validate_input(numbers, user_input):
      numbers.append(user_input)

  print()

  return numbers
```