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 inputs of various lengths. Results are analyzed and discussed to highlight the potential tradeoffs one should consider when deciding what multiplier is to be used.

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 hardware multiplication schemes 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

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 uses this space for its computations. First, the multiplier is placed into the workspace and shifted left by 1. From there, the multiplier is used to either add or subtract from the upper half of the workspace. The specific action is dependent on the last two bits of the workspace.

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 arithmaticly 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 arithmaticly 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.

Simulation Implimentation

Both algorithims were simulated in Python in attempts to utalize its high level nature for rapid development. The table for Booth's algorithim was preformed with a simple if-then loop, while a switch case was used in modified booth's algorithim. Simple integers were used to represent registers.

One objective of this paper is to analyze and compare the peformance of these two algorithms for various operand lengths. As such, the length of operands had to be constantly accounted for. Aritmatic bitwise operations, including finding two's compliment, were all implimented using functions that took length as an input. Further more, extra bits were cleared after each iteration.

To track down issues and test the validity of the multipliers, a debug function was written. To allow Python to natively work with the operands, each value is calculated from its two's compliment format. The converted numbers are then multiplied, and the result is compared to both Booth's Algorithim and Modified Booth's Algorithim. To ensure that the debugging function itself doesn't malfunction, all converted operands and expected results are put into a single large table for checking. The exported version of this table can be seen in table X.

Analysis

Modified Booth's algorithim only requires half the iterations as Booth's algorithim. As such, it can be expected that the benifit of modified Booth's algorithim increases two fold with bit length. This can be shown by comparing the two curves in figure X.

Despite this, the nature of both algorithims dictate that modified booth's algorithim is not explicitly faster. Iteration count translates to the *maxiumum* number of additions and subtractions. Figure X shows the performance of the two algorithims given different input lengths, while table x shows the actual data made to generate the plot. There are some interesting things to note. When operands contain repeating zeros or ones, both operations preform similarly, as only shifting is required. Operands containing entirely ones or zeros result in idential preformance. On the contrary, alternating bits within operands demonstrate where the two algorithims differ, as almost no bits can be skipped over. Operands made entirely of alternating bits result in the maximum performance diffrence, in which modified booth's algorithim is potentially two times faster.

All of this needs to be considered when designing an ALU. Modified booth's algorithim may improve speed, but requires substantially more hardware to impliment. One must consider if die space is to be allocated to optimize multiplication. In many applications, fast multiplication is unnessesary; many early single-chip processors and microcontrollers didn't impliment multiplication, as they were intended for simple embeded applications.

Conclusion

Hardware multipliers can help accellerate applications in which multiplication is frequent. When implimenting hardware multipliers, it's important to consider the advantages and disadvantages of various multiplier schemes. Modified Booth's algorithim gives diminishing returns for smaller operands and requires significantly more logic. In applications that depend heavily on fast multiplication of large numbers, modified booth's algorithim is optimal.

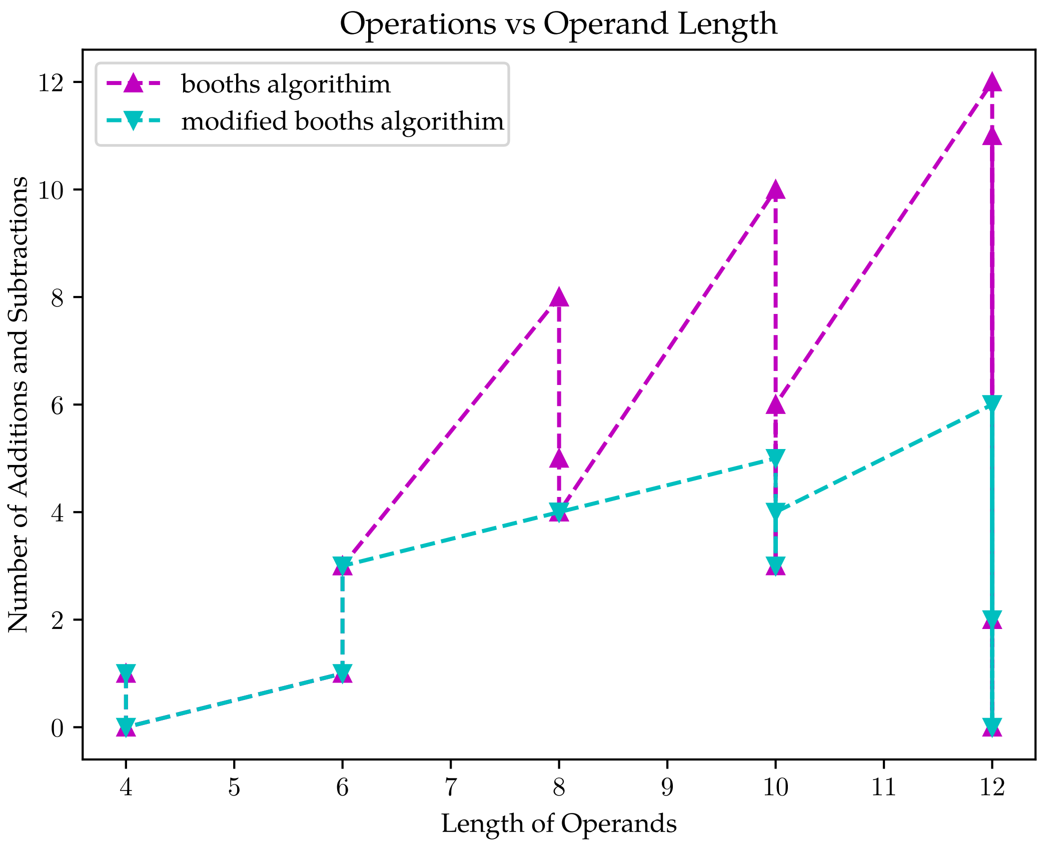
Appendix

```python
#!/usr/bin/env python3
from tabulate import tabulate
import matplotlib
import matplotlib.pyplot as plt

# finds the two's compliment of a given number
def twos_comp(num, length):
  if num == 0:
    return 0
  return abs((num ^ ((1 << length) - 1)) + 1)

# arithmaticlly shifts right; divides by 2
def arithmatic_shiftr(num, length, times):
  for t in range(times):
    num = (num >> 1) | ((1 << length - 1) & num)
  return num

# arithmaticlly shifts left; multiplies by 2
def arithmatic_shiftl(num, length):
  if num & (1 << length - 1):
    return (num << 1) | (1 << length - 1)
  else:
    return (num << 1) & ~(1 << length - 1)

# only used for debugging function to allow python to natively use two's compliment numbers
def twoscomp_to_int(num, length):
  if num & (1 << length - 1):
    return (-1 * twos_comp(num, length))
  return num & (1 << length) - 1

def debug(results):
  headers = ['multiplicand bin', 'multiplier bin', 'multiplicand dec', 'multiplier dec', 'expected bin', 'expected dec', 'booth if correct', 'booth mod if correct']
  table = []
  for [multiplicand_bin, multiplier_bin, result_booth, result_booth_mod, length] in results:
    multiplicand = twoscomp_to_int(multiplicand_bin, length)
    multiplier = twoscomp_to_int(multiplier_bin, length)
    expected = multiplicand * multiplier
    expected_bin = (twos_comp(expected, length * 2), expected) [expected > 0]
    success_b = [bin(result_booth), "PASS"] [result_booth == expected_bin]
    success_bm = [bin(result_booth_mod), "PASS"] [result_booth_mod == expected_bin]

    table.append([bin(multiplicand_bin), bin(multiplier_bin), multiplicand, multiplier, bin(expected_bin), expected, success_b, success_bm])
  print("\nCHECKS: \n", tabulate(table, headers), "\n")
```

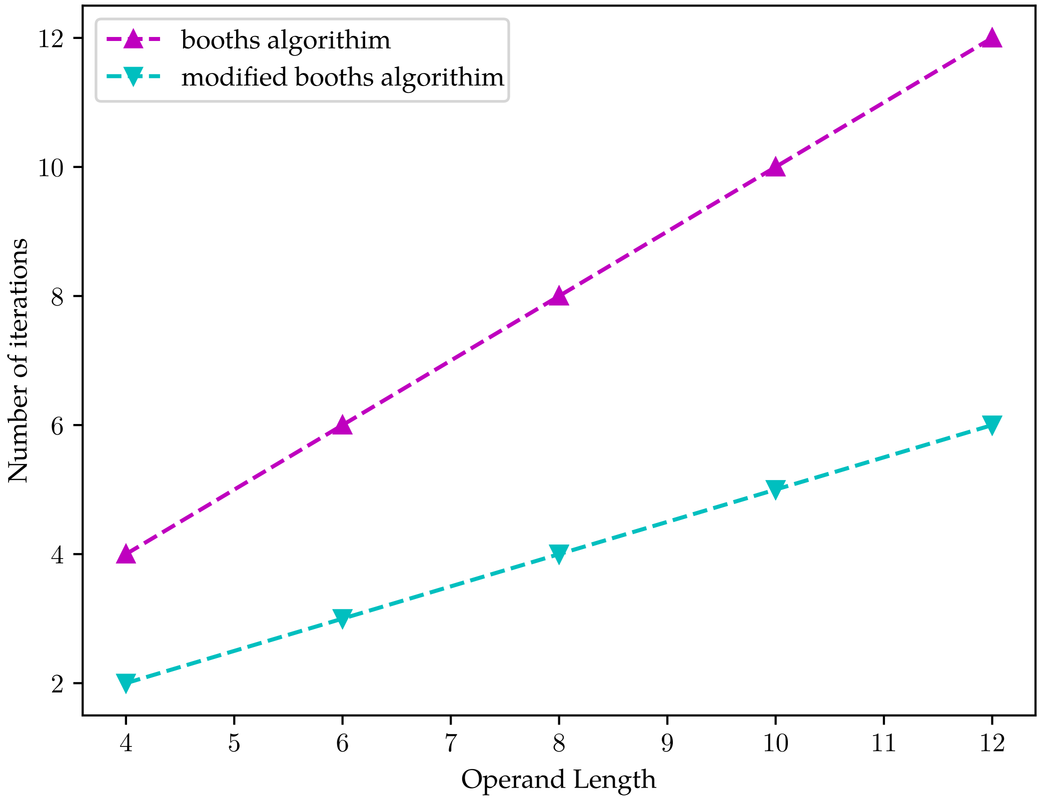
```python
def booth(multiplier, multiplicand, length):
  operations = 0
  multiplicand_twos_comp = twos_comp(multiplicand, length)
  result = multiplier << 1 # extended bit
  for i in range(length):  # iteration count is size of one operand
    op = result & 0b11
    if op == 0b01:
      operations += 1 # add upper half by multiplicand
      result += multiplicand << (length + 1)
    if op == 0b10:
      operations += 1 # subtract upper half by multiplicand
      result += multiplicand_twos_comp << (length + 1)
    result &= (1 << (length * 2) + 1) - 1 # get rid of any overflows
    result = arithmatic_shiftr(result, (length * 2) + 1, 1)
  result = result >> 1
  return (result, operations)

def booth_mod(multiplier, multiplicand, length):
  operations = 0
  # extend workspace by *two* bits, MSB to prevent overflow when mult/sub by 2
  multiplicand |= ((1 << length - 1) & multiplicand) << 1
  multiplicand_twos_comp = twos_comp(multiplicand, length + 1)
  result = multiplier << 1
  for i in range(int((length) / 2)): # number of iterations is half the
    op = result & 0b111
    match op: # take action dependent on last two bits
      case 0b010 | 0b001: # add upper half by multiplicand
        print("add")
        result += multiplicand << (length + 1)
      case 0b011:         # add upper half by 2x multiplicand
        print("add * 2")
        result += arithmatic_shiftl(multiplicand, length + 1) << (length + 1)
      case 0b100:         # subtract upper half by 2x multiplicand
        print("sub * 2")
        result += arithmatic_shiftl(multiplicand_twos_comp, length + 1) << (length + 1)
      case 0b101 | 0b110: # subtract upper half my multiplicand
        print("sub ")
        result += multiplicand_twos_comp << (length + 1)
    if op != 0b111 and op != 0:
      operations += 1
    result &= (1 << ((length * 2) + 2)) - 1 # get rid of any overflows
    result = arithmatic_shiftr(result, (length * 2) + 2, 2)
  # shifts the workspace right by one, while duplicating extra sign bit to second MSB, and clearing the MSB.
  # this ensures the result length is 2x the operands.
  result = ((result | ((1 << ((length * 2) + 2)) >> 1)) & ((1 << ((length * 2) + 1)) - 1)) >> 1
  return (result, operations)

if __name__ == "__main__":
  # set up headers for tables
  result_headers = ['multiplicand', 'multiplier', 'result (bin)', 'result (hex)']
  result_table = []

```

```python
opcount_headers = ['multiplicand', 'multiplier', 'length', 'booth', 'modified booth']
opcount_table = []

lengths = []
ops_booth = []
ops_mod_booth = []

debug_results = []

# Reads operands from file.
# Each line needs to contain two operands in binary two's compliment form seperated by a space.
# Leading zeros should be appended to convey the length of operands.
# Operands must have the same size.
with open('input.txt') as f:
  input_string = f.read().split('\n')

for operation in input_string:
  if operation == '' or operation[0] == '#':
    continue
  length = len(operation.split(" ")[0])
  multiplicand = int(operation.split(" ")[0], 2)
  multiplier = int(operation.split(" ")[1], 2)

  # get result and operation count of both algorithims
  result_booth = booth(multiplier, multiplicand, length)
  result_mod_booth = booth_mod(multiplier, multiplicand, length)

  # gather data for matplotlib
  ops_booth.append(result_booth[1])
  ops_mod_booth.append(result_mod_booth[1])
  lengths.append(length)

  # gather data for report results table
  result_table.append([bin(multiplicand), bin(multiplier), bin(result_booth[0]), hex(result_booth[0])])

  # gather data for test function to check if simulator is working
  debug_results.append([multiplicand, multiplier, result_booth[0], result_mod_booth[0], length])

  # gather data for operation count table
  opcount_table.append([bin(multiplicand), bin(multiplier), length, result_booth[1], result_mod_booth[1]])

# tests validity of results
debug(debug_results)

# generate tables for report
print(tabulate(result_table, result_headers, tablefmt="latex"))
print(tabulate(opcount_table, opcount_headers))

# output
with open("report/result_table.tex", 'w') as f:
  f.write(tabulate(result_table, result_headers, tablefmt="latex_booktabs"))

```

```
with open("report/speed_table.tex", "w") as f:
  f.write(tabulate(opcount_table, opcount_headers, tablefmt="latex_booktabs"))

# set up plotting
matplotlib.use("pgf")
matplotlib.rcParams.update({
    "pgf.texsystem": "pdflatex",
    'font.family': 'serif',
    'text.usetex': True,
    'pgf.rcfonts': False,
})

# generate table for operations vs operand length
plt.title("Operations vs Operand Length")
plt.plot(lengths, ops_booth, '^--m', label='booths algorithim')
plt.plot(lengths, ops_mod_booth, 'v--c', label='modified booths algorithim')
plt.gca().set_xlabel("Length of Operands")
plt.gca().set_ylabel("Number of Additions and Subtractions")
plt.legend(loc='upper left')
plt.savefig('report/performance.pgf')


# generate table of iterations vs operand length
iters_booth = []
iters_mod_booth = []
for length in lengths:
  iters_booth.append(length)
  iters_mod_booth.append(int(length / 2))

plt.figure()
plt.plot(lengths, lengths, '^--m', label='booths algorithim')
plt.plot(lengths, [int(l/2) for l in lengths], 'v--c', label='modified booths algorithim')
plt.gca().set_xlabel("Operand Length")
plt.gca().set_ylabel("Number of iterations")
plt.legend(loc='upper left')
plt.savefig('report/iterations.pgf')
```



Operations vs Operand Length
booths algorithim
modified booths algorithim
Number of Additions and Subtractions
0
2
4
6
8
10
12
Length of Operands
4
5
6
7
8
9
10
11
12



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