Analysis of Restoring and Non-Restoring Division Algorithm

By Tim Kellermann, Sudeep Potluri

Due: 29 November 2022

Abstract

Throughout this article we will verify the complexity of both the Restoring and Non-Restoring division algorithms. Within this analysis we will include useful metrics such as number of additions/subtractions that will quantify the execution time of both algorithms. Along with verification we hope to prove that the non-Restoring algorithm requires on average less additions/subtractions than the Restoring algorithm.

1 Motivation

Our goal is to create a software routine which performs a sequence of subtractions. An algorithm is a well-defined procedure that takes an input and produces some output. This definition is rather broad, many things in everyday life can be written as an algorithm. Brushing your teeth, making your bed, tying your shoes, etc. However, there are many ways to achieve the same solution. For example, I could brush my teeth with a toothpick. This procedure would lead to great inefficiencies. Recognizing these inefficiencies, we have the option to choose what algorithm best fits our needs and minimizes the execution time which then leads to a faster machine. Within a machine the Arithmetic Logic Unit performs integer addition, subtraction, multiplication, and division. Among those operations listed, division is the most taxing on a system. Therefore, throughout this article we hope to provide an algorithm that reduces the number of clock cycles consumed.

2 Introduction

Within a machine, division is executed using a sequence of subtractions. This technique produces many inefficiencies, due to this an efficient algorithm was proposed. The Restoring division algorithm functions by reverting the contents of the partial remainder when the divisor is larger than the partial remainder.

The following equation denotes the Restoring algorithm:

$$(2((A-B)+B)-B)=2A-B$$

Therefore, for the Restoring algorithm it is required to subtract, restore and shift within one iteration. The second iteration will also execute subtraction.

Differing, the Non-Restoring algorithm does not revert the contents of the partial reminder if the divisor is larger than the partial remainder. It continues to the second iteration where the divisor will be added to the partial remainder.

The given equation for the non-Restoring division Algorithm:

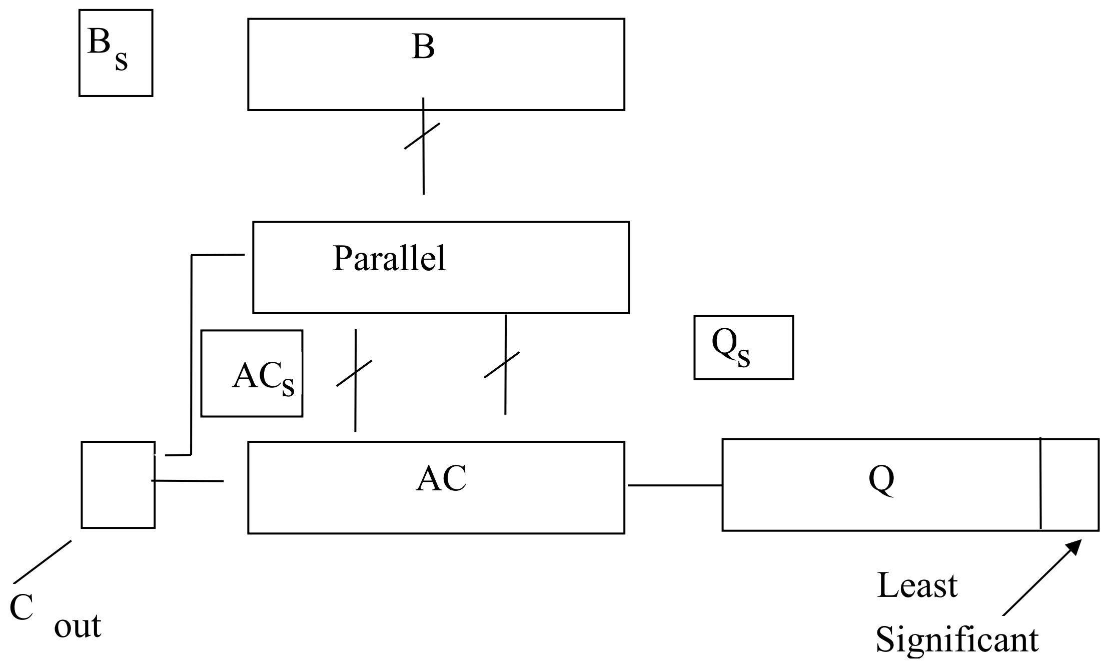
$$2(A - B) + B = 2A - B$$

This illuminates that utilizing non-Restoring techniques the first iteration only requires subtraction and shifting. The following iteration actions will be influenced by the prior subtraction.

Our aspiration is to develop a simulator to mimic the behavior of Restoring and non-Restoring Division techniques. Thus, highlighting the brilliance of the non-Restoring division algorithm.

3 Implementation

Our implementation is done using Python. This was decided due to pythons built in functions and easy manipulation of Strings/Arrays in comparison with C/C++. Most of our testing was completed with our own personal computers but can work on the CS department Linux machine. Due to division and multiplication taking similar forms as add and shift or vice versa, the hardware requirements are identical.



For out implementation, we decided to separate the parts of Restoring and Non-Restoring into different functions. While the solution is constructed in our Restoring and non-Restoring functions the trivial operations such as two's complement, add, shift left, etc are their own functions for sake of consistency. When we made our pseudocode and actual code for our report, we followed a hand-done problem. We used the steps in a normal Restoring and a Non-Restoring Algorithm. After fishing our hand-done problems (and checking our results), we followed the steps we made and constructed the code based on the decisions we made during the handwritten problem. An example of this is after subtracting the B from A in the restoring algorithm you need to check the E bit and change the Q_0 bit and either restore or leave the result. We solved this by using an if statement ties to the E bit after the operation. In addition, we use pretty table which is a library in python that makes our print statements look much nicer. This allows for easier readability of separate sections. We display each step we made in each algorithm, this was to confirm that our code is working as intended. At the end of each algorithm, we display the answers as well as the number additions and subtractions as well as the iterations the operation took.

The following steps correspond to the Restoring Algorithm:

1: If upper half of dividend is greater than or equal to divisor, Divide Overflow occurs Terminate operation

2: Initialize the registers i.e. Q = lower half of dividend, B=divisor, A = upper half of dividend, N= number of bits in the dividend.

3: Shift left contents of A and Q by one position

4: Perform A=A-B.

5: If the result is positive, $Q_0 = 1$. Else, $Q_0 = 0$ and the contents of register A are restored.

6: Decrement the value of N by one.

7: Repeat steps 3 through 6 until N=0.

8: Q = quotient and A = remainder

Algorithm 1 RESTORING (DIVIDEND, DIVISOR)

```
1: procedure Restoring(DIVIDEND, DIVISOR)
2:     if(divideOverflow(DIVIDEND, DIVISOR)) then
3:         return "Divide Overflow"
4:     end if
5:     n = length(Divisor)
```

```
    $A$ = upper half of Dividend
    $Q$ = lower half of Dividend
    $B$ = Divisor
    **while** $n > 0$ **do**
        shiftLeft(A,Q)
        subtract(A,B)
        **if**($A>B$) **then**
            $Q_0 = 1$
        **end**
        **if**($A<0$) **then**
            $Q_0 = 0$
            restore(A)
        **end if**
        $n = n-1$
    **end while**
    **return** $A,Q$
**end procedure**
```

The following steps correspond to the Non-Restoring Algorithm:

1: If upper half of dividend is greater than or equal to divisor, Divide Overflow occurs Terminate operation

2: Initialize the registers i.e. Q = lower half of dividend, B=divisor, A = upper half of dividend, N= number of bits in the dividend.

3: Check the sign of contents in register A.

4: If positive, shift left AQ and perform A=A-B. If negative, shift left AQ and perform A=A+B.

5: Check the sign of contents of register A.

6: If positive, $Q_0 = 1$. If negative, $Q_0 = 0$.

7: Decrement the value of N by one.

8: Repeat steps 3 through 7 until N = 0.

9: Check the sign of contents of A. If negative, perform A=A+B.

10: Q = quotient and A = remainder

Algorithm 2 NONRESTORING (DIVIDEND, DIVISOR)

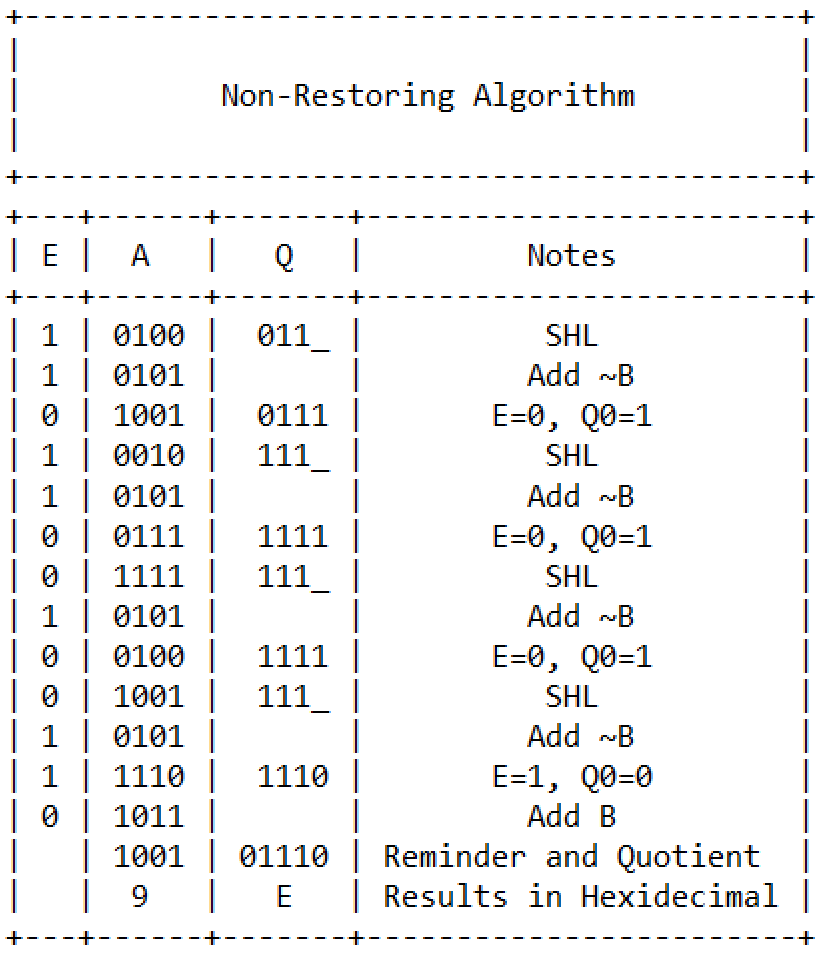
```
**procedure** NonRestoring(DIVIDEND, DIVISOR)
```

```
    if(divideOverflow(DIVIDEND, DIVISOR)) then
        return "Divide Overflow"
    end if
    $n$ = length(Divisor)
    $A$ = upper half of Dividend
    $Q$ = lower half of Dividend
    $B$ = Divisor
    while $n > 0$ do
        if($A[0] = 1$ ) then
            shiftLeft(A,Q)
            add(A,B)
        end if
        if($A[0] = 0$ ) then
            shiftLeft(A,Q)
            sub(A,B)
        end if
        if($A[0] = 0$ ) then
            $Q_0 = 1$
        end if
        if($A[0] = 1$ ) then
            $Q_0 = 0$
        end if
        $n = n-1$
    end while
    if($A[0] = 1$ ) then
        add(A,B)
    end if
    return A,Q
end procedure
```

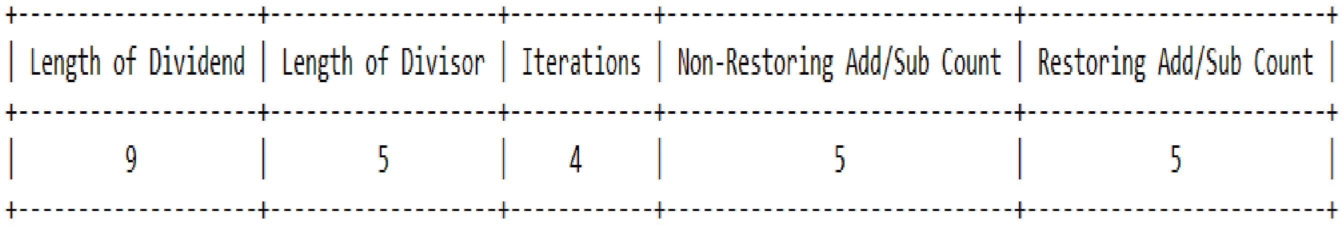
The following output is generated from the simulator. The given inputs are Dividend = 010100011 and Divisor = 01011 in signed magnitude. The following verification has been done alongside an example homework problem.

Restoring Algorithm

E	A	Q	Notes
1	0100	011_	SHL
1	0101		Add ~B
0	1001	0111	E=0, A>B no restore Q0=1
1	0010	111_	SHL
1	0101		Add ~B
0	0111	1111	E=0, A>B no restore Q0=1
0	1111	111_	SHL
1	0101		Add ~B
0	0100	1111	E=0, A>B no restore Q0=1
0	1001	111_	SHL
1	0101		Add ~B
0	1001	1110	E=1, A<0 restore Q0=0
	1001	01110	Reminder and Quotient
	9	E	Results in Hexidecimal



```
+--------------------------------------------+
|                                            |
|           Non-Restoring Algorithm          |
|                                            |
+--------------------------------------------+
+---+------+-------+-------------------------+
| E |  A   |   Q   |          Notes          |
+---+------+-------+-------------------------+
| 1 | 0100 |  011_ |           SHL           |
| 1 | 0101 |       |          Add ~B         |
| 0 | 1001 |  0111 |        E=0, Q0=1        |
| 1 | 0010 |  111_ |           SHL           |
| 1 | 0101 |       |          Add ~B         |
| 0 | 0111 |  1111 |        E=0, Q0=1        |
| 0 | 1111 |  111_ |           SHL           |
| 1 | 0101 |       |          Add ~B         |
| 0 | 0100 |  1111 |        E=0, Q0=1        |
| 0 | 1001 |  111_ |           SHL           |
| 1 | 0101 |       |          Add ~B         |
| 1 | 1110 |  1110 |        E=1, Q0=0        |
| 0 | 1011 |       |          Add B          |
|   | 1001 | 01110 | Reminder and Quotient   |
|   |  9   |   E   | Results in Hexidecimal  |
+---+------+-------+-------------------------+
```



```
+--------------------+------------------+------------+----------------------------+--------------------------+
| Length of Dividend | Length of Divisor | Iterations | Non-Restoring Add/Sub Count | Restoring Add/Sub Count |
+--------------------+------------------+------------+----------------------------+--------------------------+
|         9          |        5         |     4      |             5              |            5             |
+--------------------+------------------+------------+----------------------------+--------------------------+
```

The following data was collected from the test inputs that were provided.

<u>Length of Divisor</u>	<u>Length of the Dividend</u>	<u># of iterations</u>	<u>Non-Restoring Add/Sub Count</u>	<u>Restoring Add/Sub Count</u>
9	5	4	4	4
9	5	4	4	5
9	5	4	4	5

9	5	4	4	7
13	7	6	7	8
13	7	6	6	8
13	7	6	7	8
17	9	8	8	8
17	9	8	8	10
17	9	8	9	12
17	9	8	8	11
25	13	12	12	12
25	13	12	12	18
25	13	12	12	16

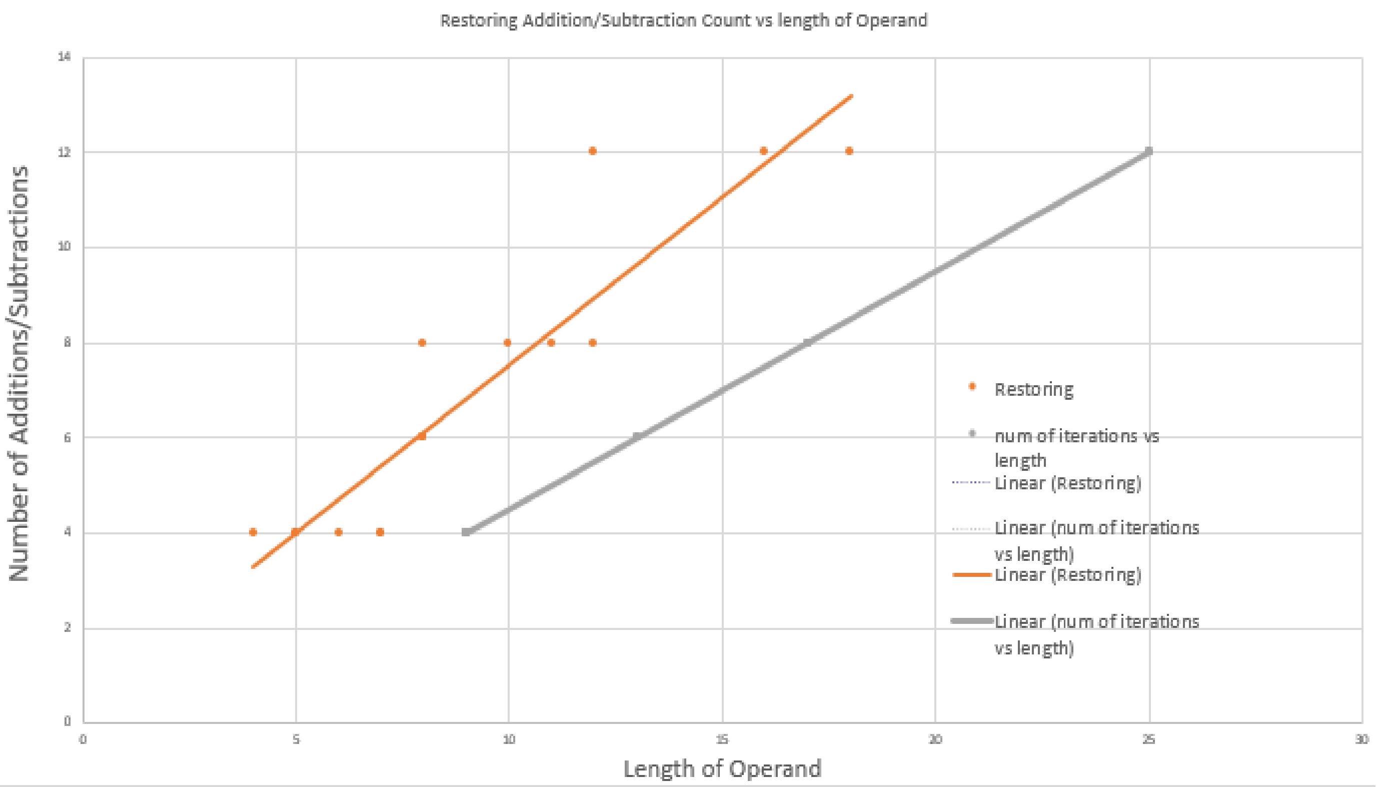


Figure 1: Graph displaying number of Restoring Additions/Subtractions with number of iterations vs length.

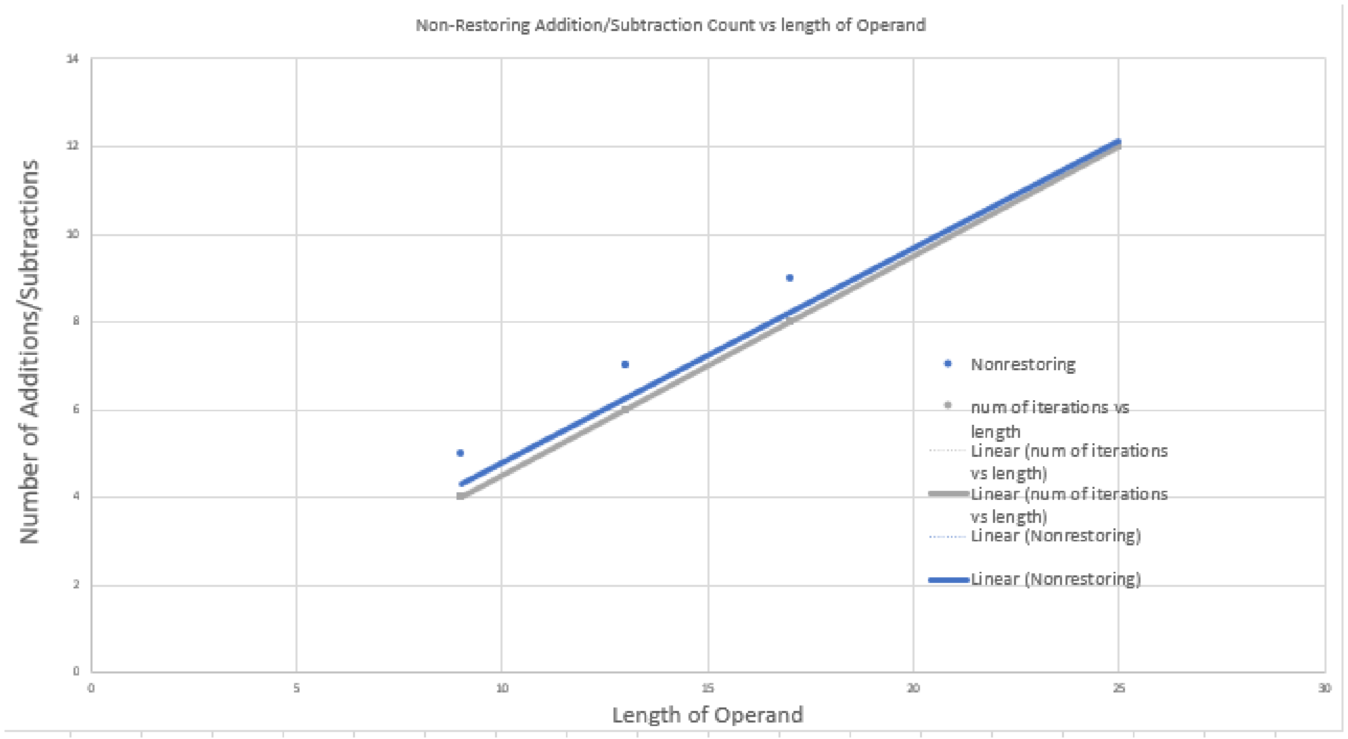


Figure 2: Graph displaying number of non-Restoring Additions/Subtractions with number of iterations vs length.

4 Experimental Analysis

The arithmetic logic unit performs integer division as a sequence of subtractions. However, with each macro-operation there a several micro operations that consume many clock cycles. Therefore, it is crucial to minimize the amount of macro-operations to lessen the load on the machine.

It is now our goal to verify that the non-Restoring division algorithm performs less additions/subtractions than the Restoring algorithm which enables the machine to perform better. The graph shown above illuminates that as the length of the operands grow the number of additions/subtractions will also increase. This phenomenon occurs for both the Restoring and the non-Restoring case; however, number of additions/subtractions for the non-Restoring case is worst case *n+1*. Furthermore, for the Restoring algorithm it is consistently utilizing more additions and subtractions to counteract when the divisor is greater than the partial dividend which worst case can use *2n-1* additions/subtractions which would mean that you would need restore after every iteration thus doubling the number of additions. These equations that have been proposed support our trendlines from the graph above.

5 Conclusions

Within this article we have verified that the Restoring division algorithm is an efficient way to produce a quotient and reminder given two binary numbers. This has been shown through many trials that included varying lengths, signs, and even divide overflow cases. Each case has been verified by hand by both authors for all cases whose operands length is less than ten bits. All the data collected supports the claim that the non-Restoring division algorithm is a much-improved solution requiring on the worst case *n+1* additions/subtractions. This allows the Arithmetic Logic Unit (ALU) to conserve clock cycles from meaningless restores during the restoring division algorithm.

Appendix

Simulation of both the Restoring and non-Restoring division algorithms. With logging of each step with reasoning

PYTHON Source Code: **simulator.py**

```
'''
    Names: Tim Kellermann, Sudeep Potluri
    Project: Restoring vs. Nonrestoring division Algorithm
    Due Date: 11/29/2022
    file name: simulator.py
'''

import math
from prettytable import PrettyTable # pip install prettytable to use library

'''
    +---------------------------------------------------+
    |                                                   |
    |              Global Variables                     |
    |                                                   |
    +---------------------------------------------------+
'''

restoringTable = PrettyTable()
nonrestoringTable = PrettyTable()
metricsTable = PrettyTable()

restoringTable.field_names = ["E", "A" , "Q" , "Notes"]
nonrestoringTable.field_names = ["E", "A" , "Q" , "Notes"]
metricsTable.field_names = ["Length of Dividend", "Length of Divisor" ,"Iterations", "Non-Restoring Add/Sub Count" , "Restoring Add/Sub
Count"]

dividend = "010100011" # has to be EAQ
B = "01011"

'''
    +---------------------------------------------------+
    |                                                   |
    |             Function Definitions                  |
    |                                                   |
    +---------------------------------------------------+

    Purpose: Executes the Restoring division algorithm and logs steps taken.
    Pre-Conditions: input dividend as EAQ, and Divisor
    Post-Conditions: returns the number of additions/subtractions
```

```
'''
def restoring(dividend, divisor):

    restAddSubCount = 0
    sign = int(dividend[0]) ^ int(divisor[0]) #calculates sign bit
    sign = str(sign)
    divisor = "0" + divisor[1::]

    #divide overflow check
    if(divideOverflow(dividend,divisor)):
        restoringTable.add_row([" Divide Overflow has occured"," First half of A>=B","", ""])
        return "Divide Overflow"

    i = len(divisor)-1 # this is equal to the amout of iterations we will  have
    NotB = onesComplement(divisor) #does 1 and twos comp
    while( i > 0 ):
        if(i != len(divisor)-1):
            dividend = A + Q
        E,A,Q= shl(dividend) #Shift left
        restoringTable.add_row([E,A,Q, "SHL"])
        A = E+A
        SavedA = A
        A = add(A,NotB)
        restoringTable.add_row([NotB[0],NotB[1::],"", "Add ~B"]) #this is used to log steps
        restAddSubCount = restAddSubCount + 1 #increments the addition counter
        if(A[0] == '0'):
            Q = Q.replace('_','1')
            restoringTable.add_row([A[0],A[1::],Q, "E=0, A>B no restore Q0=1"]) #this is used to log steps
        else:
            A = SavedA
            Q = Q.replace('_','0')
            restoringTable.add_row([A[0],A[1::],Q, "E=1, A<0 restore Q0=0"]) #this is used to log steps
            restAddSubCount +=1 #increments the addition counter
        i -= 1
    restoringTable.add_row(["",A[1::],(sign+Q), "Reminder and Quotient"]) #Solution in binary
    restoringTable.add_row(["", binToHexa((A[1::])), binToHexa(sign+Q), "Results in Hexidecimal" ]) #Solution in hexadecimal
    return restAddSubCount

'''
   Purpose: Executes the non-Restoring division algorithm and logs steps taken.
   Pre-Conditions: input dividend as EAQ, and Divisor
   Post-Conditions: returns the number of additions/subtractions
'''
def nonrestoring(dividend, divisor):
    nonrestAddSubCount = 0
    sign = int(dividend[0]) ^ int(divisor[0]) #calculates sign of quotient
    sign = str(sign)
    divisor = "0" + divisor[1::]
    dividend = "0" + dividend[1::]

    #divide overflow check
    if(divideOverflow(dividend,divisor)):
        nonrestoringTable.add_row([" Divide Overflow has occured"," First half of A>=B","", ""])
        return "Divide Overflow"

    i = len(divisor)-1 # this is equal to the amout of shifts we will  have
    NotB = onesComplement(divisor)
    E = "0"
    while( i > 0):
        if(i != len(divisor)-1):
            dividend = A + Q

        if(dividend[0] == "1"):
            E,A,Q= shl(dividend)
            nonrestoringTable.add_row([E,A,Q, "SHL"]) #this is used to log steps
            A = E+A
            A = add(A,divisor)
            nonrestAddSubCount +=1 #increments the addition counter
            nonrestoringTable.add_row([divisor[0],divisor[1::],"", "Add B"]) #this is used to log steps
        else:
            E,A,Q= shl(dividend)
            nonrestoringTable.add_row([E,A,Q, "SHL"])
            A = E+A
            A = add(A,NotB)
            nonrestAddSubCount +=1 #increments the addition counter
            nonrestoringTable.add_row([NotB[0],NotB[1::],"", "Add ~B"]) #this is used to log steps
        if(A[0] == '0'):
            Q = Q.replace('_','1')
            nonrestoringTable.add_row([A[0],A[1::],Q, "E=0, Q0=1"]) #this is used to log steps
        else:
            Q = Q.replace('_','0')
            nonrestoringTable.add_row([A[0],A[1::],Q, "E=1, Q0=0"]) #this is used to log steps
        i-=1
    if(A[0] == "1"):
        nonrestoringTable.add_row([divisor[0],divisor[1::],"", "Add B"]) #this is used to log steps
        nonrestAddSubCount +=1 #increments the addition counter
        A = add(A,divisor)

    nonrestoringTable.add_row(["",A[1::],(sign+Q), "Reminder and Quotient"]) #Result in Binary
    nonrestoringTable.add_row(["", binToHexa((A[1::])), binToHexa(sign+Q), "Results in Hexidecimal" ]) #Result in Hexadecimal
    return nonrestAddSubCount

'''
    Purpose: converts binary to decimal
```

```
   Pre-Conditions: input a string that is binary
   Post-Conditions: returns the decimal equivalent
'''
def bin2dec(binary):
    decimal = 0
    for digit in binary:
        decimal = decimal*2 + int(digit)
    return decimal

'''
   Purpose: checks if divide overflow has occured
   Pre-Conditions: input dividend and divisor
   Post-Conditions: returns a bool if true then terminate program
'''
def divideOverflow(dividend, divisor):
    A = dividend[1:(math.ceil((len(dividend)/2)))]
    decDivisor = bin2dec((divisor[1::]))
    decA = bin2dec((A))
    if(decA >= decDivisor): # correlates to the partial Remainder greater than divisor
        return True
    else:
        return False

'''
   Purpose: shifts the binary number left
   Pre-Conditions: input binary string
   Post-Conditions: returns EAQ from the shift
'''
def shl(dividend):
    E = dividend[1]
    A = dividend[2:(math.ceil((len(dividend)/2))+1)] # takes the upper half of bits
    Q = dividend[(math.ceil(len(dividend)/2)+1): len(dividend)] # takes the lower half
    Q += "_" # adds a buffer as Q0

    return E, A , Q

'''
   Purpose: converts binary to hexadecimal
   Pre-Conditions: input binary string
   Post-Conditions: returns the hexadecimal equavalent
'''
def binToHexa(n):
    bnum = int(n)
    temp = 0
    mul = 1

    # counter to check group of 4
```

```
count = 1

# char array to store hexadecimal number
hexaDeciNum = ['0'] * 100

# counter for hexadecimal number array
i = 0
while bnum != 0:
    rem = bnum % 10
    temp = temp + (rem*mul)

    # check if group of 4 completed
    if count % 4 == 0:

        # check if temp < 10
        if temp < 10:
            hexaDeciNum[i] = chr(temp+48)
        else:
            hexaDeciNum[i] = chr(temp+55)
        mul = 1
        temp = 0
        count = 1
        i = i+1

    # group of 4 is not completed
    else:
        mul = mul*2
        count = count+1
    bnum = int(bnum/10)

# check if at end the group of 4 is not
# completed
if count != 1:
    hexaDeciNum[i] = chr(temp+48)

# check at end the group of 4 is completed
if count == 1:
    i = i-1

final = ""
while(i >=0):
    final += hexaDeciNum[i]
    i = i-1

if(final == ""):
    final = "0"
return final
```

```
'''
   Purpose: calculates inverse
   Pre-Conditions: input binary string
   Post-Conditions: returns the inverse as 2's comp
'''
def onesComplement(num):
    oneComp = 0
    oneCompStr = ""
    for i in num:
        oneComp = 1^int(i)
        oneCompStr += str(oneComp)

    oneCompStr = twosComplement(oneCompStr) #adds 0x01 for twos comp
    return oneCompStr

'''
   Purpose: calculates 2's comp
   Pre-Conditions: input binary string
   Post-Conditions: returns the inverse as 2's comp
'''
def twosComplement(num):
    fill = len(num)
    one = "1"
    one = one.zfill(fill) #fills the leading positions with 0's
    sum = (add(num,one))
    return sum

'''
   Purpose: addition function used in both restoring and nonrestoring
   Pre-Conditions: input two binary numbers
   Post-Conditions: returns the sum of both binary numbers
'''
def add(num1,num2):
    i = len(num1) -1

    result = ''
    carry = 0
    while(i >= 0):
        sum = int(num1[i]) + int(num2[i])
        if sum == 2: #1 + 1
            if carry == 1:# 1+ 1 + carry
                carry = 1
                result += '1'
            else: #1 + 1
                carry = 1
                result += '0'
        elif sum == 1: # 1 + 0 or 0 + 1 case
            if carry == 1: # 1 + 0 + carry
                carry = 1
                result += '0'
            else: # 1 + 0
                carry = 0
                result += '1'
        elif sum == 0: # 0 + 0 case
            if carry == 1: # 0 + 0 + carry
                carry = 0
                result += '1'
            else: # 0 + 0
                carry = 0
                result += '0'
        i -= 1
    result = result[::-1]
    return result


'''
   +--------------------------------------------------+
   |                                                  |
   |             Print Steps of Operation             |
   |                                                  |
   +--------------------------------------------------+
'''

restAddSubCount = restoring(dividend, B) #logs all of the Restoring Steps
nonRestAddSubCount = nonrestoring(dividend,B) #logs all of the nonRestoring Steps
metricsTable.add_row([len(dividend), len(B), len(B)-1, nonRestAddSubCount, restAddSubCount]) #logs the counts of each algorithm

print("\n")
print("+---------------------------------------------+")
print("|                                             |")
print("|            Restoring Algorithm              |")
print("|                                             |")
print("+---------------------------------------------+")
print(restoringTable)
print("\n")

print("+-------------------------------------------+")
print("|                                           |")
print("|          Non-Restoring Algorithm          |")
print("|                                           |")
print("+-------------------------------------------+")
print(nonrestoringTable)
print("\n")
print(metricsTable)
```