Analysis of Non-Restoring and Restoring Methods for Binary Division

Lokesh Das
Department of Electrical and computer Engineering
Missouri University of Science and Technology, Rolla, MO 65401

COMP ENG 3110 : Computer Organization and Design
Dr. Alireza Hurson
Apr 11, 2025

Table of Contents

Abstract

This paper presents a comprehensive comparative analysis of the Restoring and Non-Restoring methods for binary division. The algorithms are implemented and evaluated using Python. The simulator assesses the performance of both methods based on the number of iterations and arithmetic operations required. The study also compares their efficiencies across varying operand lengths using graphs and tabulated results. Overall, the paper discusses the implementation details, simulation results, and performance analysis of the simulator.

Introduction and Motivation

Division is the basic arithmetic operation performed by the ALU. It is widely used in applications such as signal processing, cryptography, and numerical computation. Binary division can be implemented in different ways. It can be very inefficient if not implemented properly. The most basic approach to handle the division between two binary numbers is by recursive subtraction. Despite being easy, this approach is time-consuming. Thus, to perform the Division Algorithm while also considering both accuracy and efficiency, we have the Restoring and Non-Restoring methods. These are the two classical methods for performing binary division.

The Restoring method uses restoration of the partial remainder whenever the divisor is larger than the newly generated partial remainder. This implies that, while performing the algorithm mathematically, whenever the extended bit is greater than 1, the next step restores the previous state.

In the Non-Restoring method, if it is detected that the divisor is larger than the partial remainder, instead of restoring, it proceeds with the next step and then adds in the upcoming step instead of subtracting. This step reduces the number of operations required, making the Non-Restoring method computationally more efficient compared to the Restoring method.

The motivation behind this research is to understand their efficiency in terms of computational resources and time. The study also investigates how operand length influences performance, motivating the identification of the most suitable approach for various scenarios.

Technical Aspects of Simulator

Hardware Configurations:

The simulator operates on signed binary numbers represented as strings and is implemented using the high-level dynamic programming language Python. The algorithms are designed to mimic the setup of a physical Arithmetic Logic Unit (ALU). The simulator runs on a standard desktop or laptop, with no requirement for specialized hardware. However, utilizing modern CPUs can enhance execution speed.

Structure of the Simulator:

The simulator is highly modular in nature, and it consists of following components:

1. Main Program (main.py) : This part of the simulator handles the input from the user as integer and converted into string to pass inside the division algorithm. This part is also responsible for validating the user's input. It provides an interactive interface of continuing with Binary Division or exiting the program.
2. Division Algorithm (DivisionAlgo.py) : In this part, I have developed the Restoring and Non-Restoring algorithm as class object. There is the super class called Division which helps to make the code reusable and reduce redundancy by inheriting the common methods in the sub class Restoring and NonRestoring. Sub class Restoring implements the restoring method for binary division. Similarly, Subclass NonRestoring implements the non-restoring method for binary division.
3. Utility Function (utilis.py) : This provides helper function to add binary number, subtract them, do two's complement or convert binary into hexadecimal. Other helper function like overflow detection and sequence counter are also used.

Implementation of Simulator:

The simulators for the Restoring and Non-Restoring algorithms operate exclusively on signed binary numbers. In the main program, the user provides the signed binary inputs as integers. These inputs are validated, and any potential errors are handled to ensure smooth execution. The inputs are interpreted as the dividend and divisor, then passed as strings to instances of the Restoring and NonRestoring subclasses. Passing them as strings preserves leading zeros in the binary representation. The superclass processes the dividend and divisor in a way common to

both algorithms. It extracts the sign bit and stores it separately in memory, ensuring the magnitude portion is treated correctly during computation. The magnitude is then padded with an extra zero bit to act as an extended bit for the arithmetic operations. Other factors like iteration count, operation count is also initiated here. After this preprocessing, the core division logic specific to each algorithm is executed.

Given that the dividend is held in a double register, and it should not be initialized with zeroes, first register in my program is padded with zeros only in the case if the length of dividend is equal to divisor. The algorithm begins by checking for division overflow using the divide overflow helper function from utils.py, which returns a Boolean. If an overflow is detected, the program terminates; otherwise, the dividend is left-shifted, and the iteration count is incremented accordingly. The first part of the dividend is extracted and subtracted from the two's complement of the divisor. Both the original and new values are stored in separate variables. Based on the most significant bit (MSB) of the result:

- If the MSB is 1 (indicating the divisor is greater than the dividend segment), the algorithm restores the previous value and appends 0 to the result.
- If the MSB is 0, 1 is appended, and the loop continues.

This process repeats until the sequence counter indicates completion. The sequence counter, two's complement, and restore are all utility functions defined in utils.py, and their behavior mirrors real-world operations.

The Non-Restoring algorithm follows a similar structure with slight differences in logic:

- After the initial left shift and subtraction using the two's complement of the divisor, the extended bit is checked.
- If the extended bit is 1, the algorithm appends 0 to the result.
- In the next cycle, instead of subtracting, the divisor is added to the shifted dividend.
- If the extended bit becomes 0, then 1 is appended.

Pseudo Code:

Main Function:

Print *welcome message*
Loop *until user chooses to quit:*
Display *menu options*
If *user selects "Binary Division":*
Input dividend and divisor (signed binary)
Initialize Restoring and NonRestoring objects
Run Restoring algorithm and NonRestoring algorithm
Display results (quotient, remainder, iterations, operations)
Else if *user selects "Quit":*
Exit loop
Else:
Prompt for valid choice

Restoring Algorithm:
Initialize *variables (dividend, divisor, accumulator, quotient)*
Check *for overflow:*
Terminate Algorithm if True
While *sequence counter is True:*
Shift left combined accumulator and quotient
Increment the iteration count
Subtract divisor from remainder
Increment the operation count
If *extended bit is 1:*
Restore previous value
Increment operation count
And concatenate 0 at the end
Else*:*
Put 1 at the end of new dividend
Update quotient and accumulator
Check *status of sequence counter*
Display *final results*

NonRestoring Algorithm:
Initialize *variables (dividend, divisor, accumulator, quotient)*
Check *for overflow*
Terminate Algorithm if True
While *sequence counter is True:*
Shift left combined accumulator and quotient

Increment iteration count
Subtract divisor from remainder
Increment operation count
While *extended bit is 1:*
Concatenate 0 at the end
If *iteration count is equal to length divisor # For the case when iteration is completed*
Then perform Addition but adding necessary to correct the result
And Increment Operation count
Recheck or break
Else
Shift left the new dividend
Increment iteration count
Perform addition
And then increment operation count.
Recheck or break
If *extended bit is 0 and operation count is less than the length of divisor:*
Put 1 at the end of new dividend
Update quotient and accumulator
Check *status of sequence counter*
Display *final results*

Note:

- All binary numbers are treated as strings in the code. Terms like "concatenate" or "put" refer to appending 0 or 1 to the least significant bit (LSB) of the binary string.
- The accumulator represents the upper half of the dividend and is not initialized with zeros.

Analysis

The simulator results demonstrate the performance of both the Restoring and Non-Restoring division methods across distinct test cases. For each test case, the simulator computes the quotient and remainder in binary and hexadecimal formats, along with the number of operations (additions/subtractions) and iterations required to complete the division. A key observation is that both methods produce identical quotients and remainders for all valid inputs, confirming their correctness. The notable differences are:

- Operation Count: The Restoring method mostly performs few additional operations compared to the Non-Restoring method when handling negative remainders. This is due to the explicit restoration step in the Restoring algorithm, which involves adding the divisor back to the remainder after a subtraction when the extended bit . In contrast, the Non-Restoring method avoids this step by directly adjusting subsequent operations based on the extended bit, resulting in fewer total operations.

- Iteration Count : Both algorithms exhibit the same number of iterations for each test case, as they follow similar iterative structures. Each iteration corresponds to a shift-left operation combined with either addition or subtraction, depending on the algorithm's logic. This consistency indicates that the primary difference lies in the number of arithmetic operations performed per iteration rather than the overall number of iterations.

- Overflow Handling : The simulator correctly identifies overflow conditions, as seen in Test Case 5 where the dividend " 1111111000001" and divisor " 0000000" result in an overflow detection for both methods. In this case, the first half of the dividend is greater than the divisor. Thus, the division is interrupted mimicking the exact behavior.

- Scalability : As the operand length increases, both methods maintain consistent behavior in terms of operation and iteration counts. However, the Non-Restoring method shows a slight advantage in computational efficiency, particularly for larger operands, due to its reduced reliance on restoration steps. This makes it more suitable for scenarios requiring high-speed division operations.

Test Cases		Restoring Method					
Dividend	Divisor	Quotient	Remainder	Quotient (Hexadecimal)	Remainder (Hexadecimal)	Number of Operation	Number of Iteration
"111100001"	"01111"	11111	10000	0x1F	0x10	4	4
"001001101"	"11011"	10111	"00000"	0x17	0x00	5	4
"000010101"	"10011"	10111	"00000"	0x17	0x00	5	4
"100001101"	"01101"	10001	10000	0x11	0x10	7	4
"1111111000001"	"0000000"	Overflow detected					
"0010110110010"	"1011011"	1110110	"0000000"	0x76	0x00	8	6
"1000010111101"	"1000111"	"0011011"	1000000	0x1B	0x40	8	6
"1100111100010"	"1110111"	"0101110"	"1000000"	0x2E	0x40	8	6
"01111111000000001"	"011111111"	"011111111"	"000000000"	0xFF	0x00	8	8
"10001011110110101"	"000110011"	"101110111"	"100000000"	0x177	0x100	10	8
"00011011111001000"	"001110111"	"001111000"	"000000000"	0x78	0x00	12	8
"00000101001001011"	"101010101"	"100011111"	"000000000"	0x11F	0x000	11	8
"11111111100000000001"	"11111111111"	"01111111111"	"10000000000"	0x3FF	0x400	10	10
"10000011000101101100"	"10000111101"	"00111011100"	"10000000000"	0x1DC	0x400	14	10
"10001110001100011001"	"00101010101"	"10101010101"	"10000000000"	"0x555"	"0x400"	15	10
"00001010110000010000"	"00001111000"	"01011011110"	"00000000000"	0x2DE	0x000	13	10
"011111111111100000000000001"	"0111111111111"	"0111111111111"	"0000000000000"	0xFFF	0x000	12	12
"10001110001101110001100111001"	"1010101010101"	"0010101010101"	"1000000000000"	0x0555	0x1000	18	12
"10000001111100011000111001"	"1000011111111"	"0001111100111"	"1000000000000"	0x03E7	0x1000	16	12
"10000111000010001001000000"	"0000011100000"	Overflow detected					

Table 1: Result of all test case for Restoring method

Test Cases		Non Restoring Method					
Dividend	Divisor	Quotient	Remainder	Quotient (Hexadecimal)	Remainder (Hexadecimal)	Number of Operation	Number of Iteration
"111100001"	"01111"	11111	10000	0x1F	0x10	4	4
"001001101"	"11011"	10111	"00000"	0x17	0x00	4	4
"000010101"	"10011"	10111	"00000"	0x17	0x00	4	4
"100001101"	"01101"	10001	10000	0x11	0x10	4	4
"1111111000001"	"0000000"	Overflow Detected					
"0010110110010"	"1011011"	1110110	"0000000"	0x76	0x00	7	6
"1000010111101"	"1000111"	"0011011"	1000000	0x1B	0x40	6	6
"1100111100010"	"1110111"	"0101110"	"1000000"	0x2E	0x40	7	6
"01111111000000001"	"011111111"	"011111111"	)00000000(	0xFF	0x00	8	8
"10001011110110101"	"000110011"	"101110111"	100000000(	0x177	0x100	8	8
"00011011111001000"	"001110111"	"001111000"	)00000000(	0x78	0x00	9	8
"00000101001001011"	"101010101"	"100011111"	)00000000(	0x11F	0x000	8	8
"11111111100000000001"	"11111111111"	"01111111111"	)00000000(	0x3FF	0x400	10	10
"10000011000101101100"	"10000111101"	"00111011100"	)00000000(	0x1DC	0x400	11	10
"10001110001100011001"	"00101010101"	"10101010101"	)00000000(	"0x555"	"0x400"	10	10
"00001010110000010000"	"00001111000"	"01011011110"	)00000000(	0x2DE	0x000	11	10
"011111111111100000000000001"	"0111111111111"	"0111111111111"	)00000000(	0xFFF	0x000	12	12
"10001110001101110001100111001"	"1010101010101"	"0010101010101"	)00000000(	0x0555	0x1000	12	12
"10000001111100011000111001"	"1000011111111"	"0001111100111"	)00000000(	0x03E7	0x1000	12	12
"10000111000010001001000000"	"0000011100000"	Overflow Detected					

Table 1: Result of all the test cases

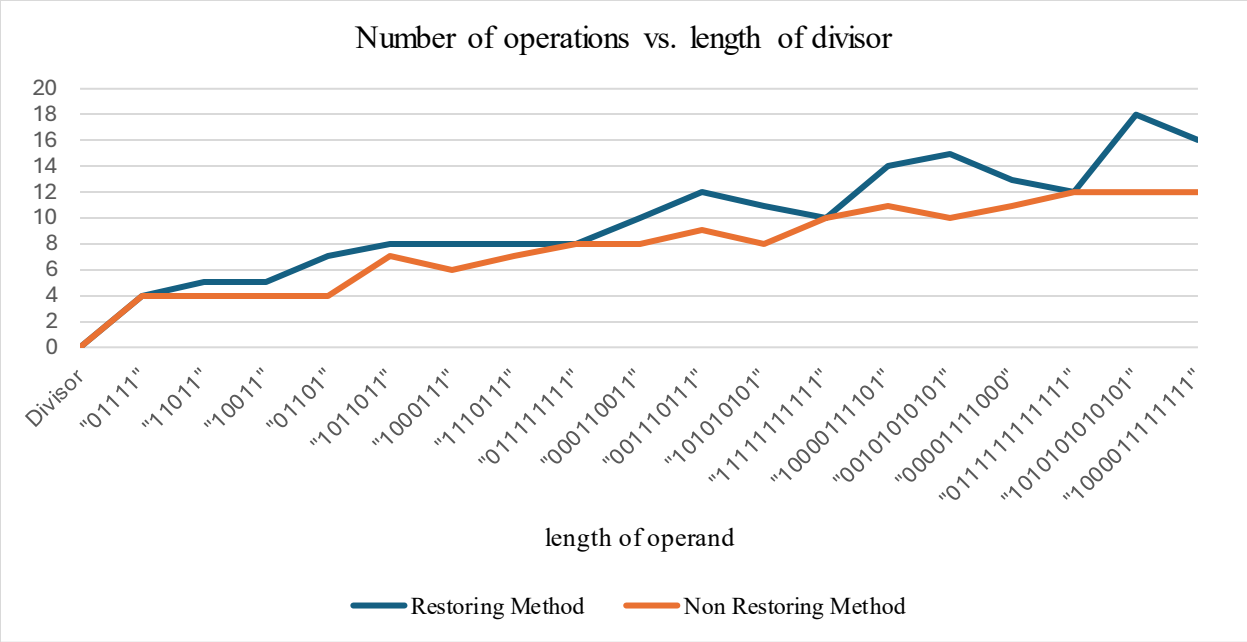

Figure 1: Graph showing number of operations vs. length of divisor

The Graphs 1 and 2 illustrates the relationship between the number of operations (additions/subtractions) and the length of the divisor/dividend for both the Restoring and Non-Restoring division methods. As the length of the operands increases, the number of operations generally rises for both methods, but the Non-Restoring method consistently requires fewer operations compared to the Restoring method, demonstrating its computational efficiency.

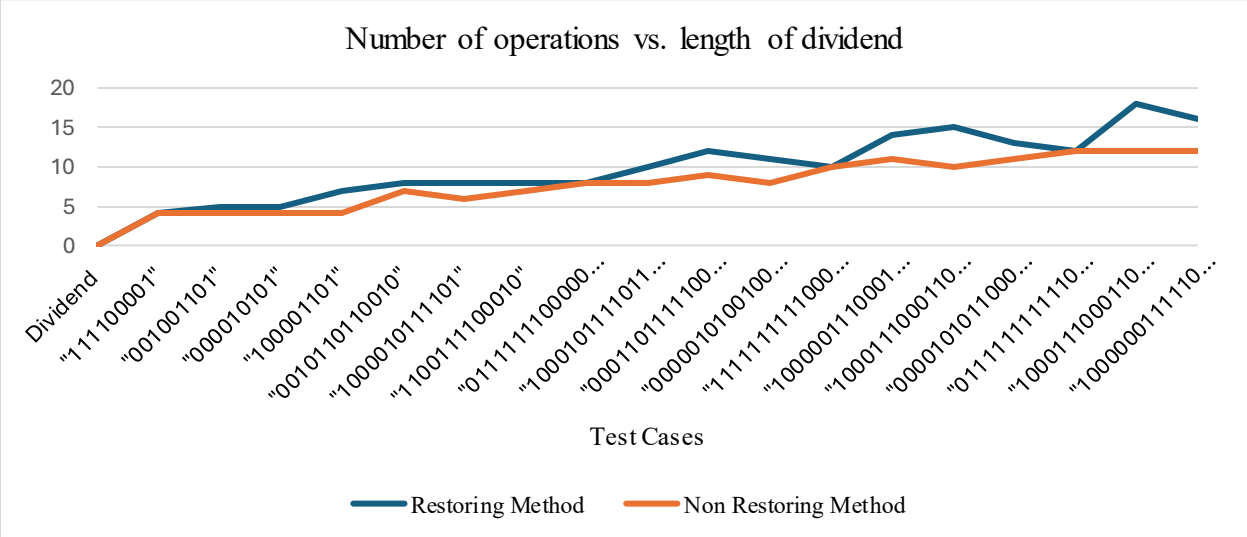

Figure 2: Graph showing number of operations vs. length of dividend

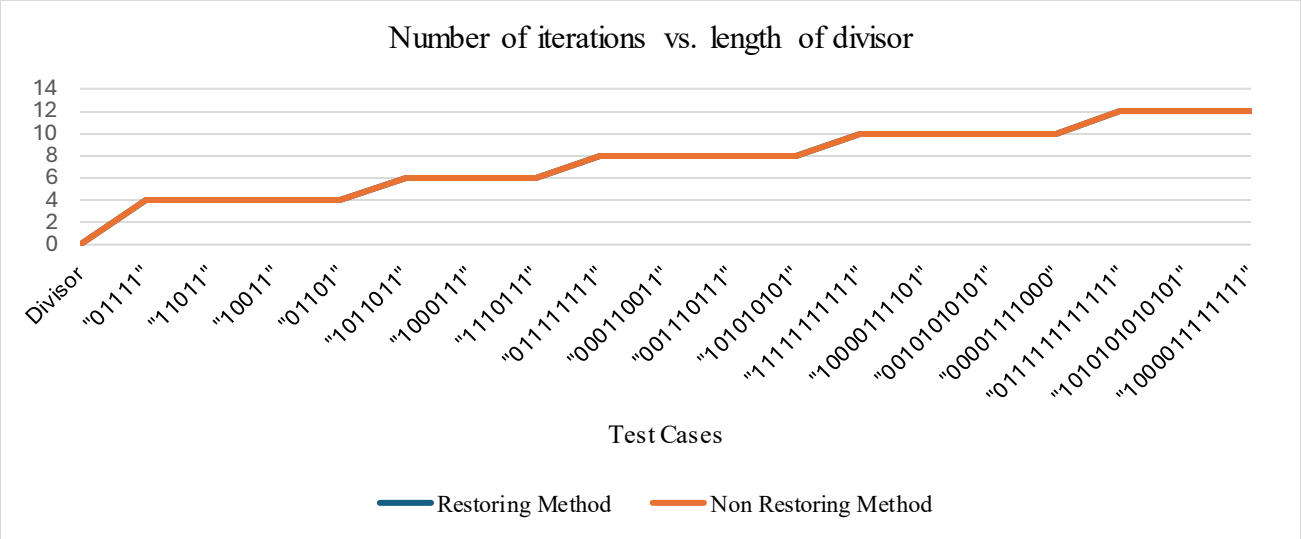

Figure 3: Graph showing number of iterations vs. length of divisor

The graphs 3 and 4 illustrates the relationship between the number of iterations and the length of the divisor for both Restoring and Non-Restoring division algorithms. As expected, the number of iterations increases linearly with the length of the operands, as each iteration processes one bit of the divisor. Both methods exhibit identical iteration counts, confirming that the computational structure of the two algorithms is fundamentally similar, with differences arising only in the number of arithmetic operations performed per iteration.

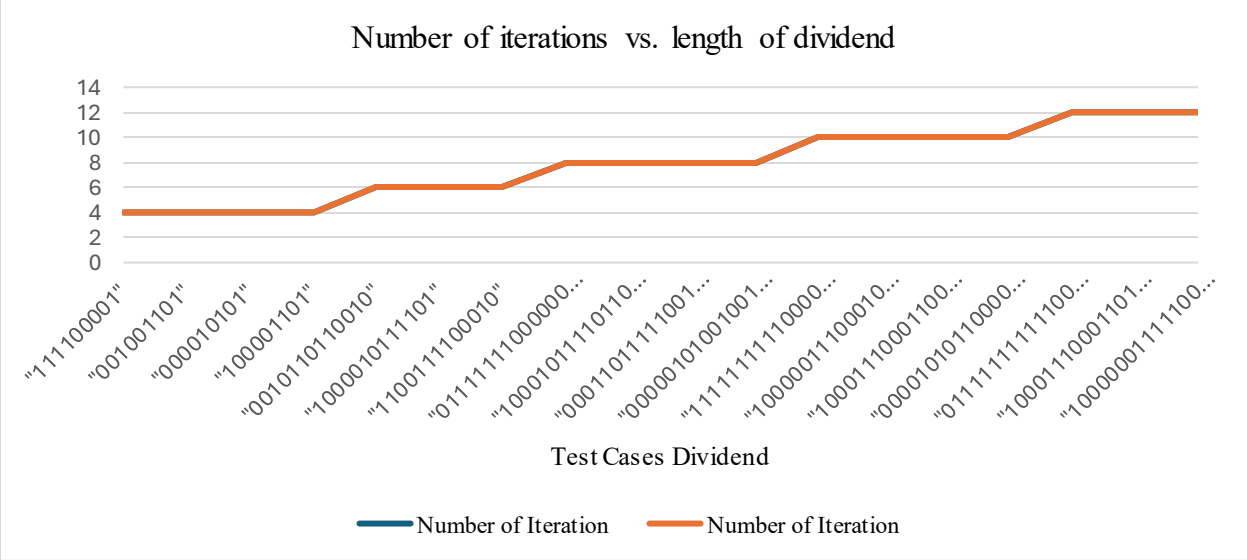

Figure 4: Graph showing number of iterations vs. length of dividend

Conclusion

The comparative analysis of the Restoring and Non-Restoring methods for binary division highlights the computational efficiency and operational differences between the two algorithms. Both methods produce identical quotients and remainders for all valid inputs, confirming their correctness in performing binary division. However, the key distinctions arise in their handling of intermediate results and the number of arithmetic operations required.

The Restoring method as per algorithm restores the partial remainder when a subtraction results in a negative value, which introduces additional operations such as adding the divisor back to the remainder. This restoration step increases the operation count, making the Restoring method less efficient. On the other hand, the Non-Restoring method avoids explicit restoration by directly adjusting subsequent operations based on the extended bit of result produced in new iteration. This approach significantly reduces the number of operations, particularly for larger operands, demonstrating its computational superiority.

Both algorithms exhibit identical iteration counts, as the number of iterations is determined by the length of the divisor and follows a similar iterative structure. However, the Non-Restoring method optimizes this process by avoiding explicit restoration. Instead, it adjusts subsequent operations based on the extended bit (most significant bit) of the intermediate result in each iteration. This approach reduces the operation count, leading to superior performance in terms of speed and computational efficiency, especially beneficial when dividing large binary numbers.

The simulator demonstrates and matches the exact flow of both the division algorithm, ensuring proper handling of inputs. Additionally, the modular design of the simulator, implemented in Python, allows for seamless integration of utility functions such as binary addition, subtraction, two’s complement conversion, hexadecimal representation, sequence counter, overflow check and restore. These features enhance the simulator's usability and provide a comprehensive platform for analyzing binary division algorithms. In addition, this modularity, can make even better user interface by updating some of the features.

In conclusion, while both Restoring and Non-Restoring methods are useful and effective for binary division, the Non-Restoring method emerges as the preferred choice for applications requiring computational efficiency and reduced reliance on restoration steps. This study underscores the importance of selecting the most appropriate algorithm based on operand size, performance requirements, and resource constraints.

Appendix

Main.py :

```
from DivisionAlgo import Restoring, NonRestoring

def main():
    """
    This is the main function of the program. It allows the user to perform binary division
    on signed binary numbers using both the Restoring and Non-Restoring algorithms.
    The program runs only for signed binary numbers and can give wrong results for unsigned
numbers.
    """
    print("\nWelcome to the Binary Division Program for the Signed number\n")

    exit = False
   while(not(exit)):
        print("Binary Division Simulator")
        print(" 1. Binary Division")
        print(" 2. Quit\n")

        choice = input("Please enter your choice: ").strip()

       if choice == '2':
            exit = True
            break
       elif choice != '1': #check for the valid choice
            print("\nPLEASE ENTER A VALID CHOICE\n")
            continue
       else:
            try:
                dividend = input("\nPlease enter the signed Binary Number as dividend: ").strip()
                divisor = input("Enter the signed Binary Number as divisor: ").strip()

                D1 = Restoring(str(dividend), str(divisor))
               D1.displayResult()
                D2 = NonRestoring(str(dividend), str(divisor))
              D2.displayResult()

           except (ValueError, IndexError):
                print("\nNo Reult for the invalid inputs. Please enter only a valid binary number\n")
                continue

if __name__ == "__main__":
    main()
```

DivisionAlgo.py:

```
import time
```

```
from utils import checkOverflow, shiftLeft, subtractBinary, restore, sequenceCounter, addBinary,
Hexadecimal

class Division:
    def __init__(self, dividend, divisor):
        """
        Initializes the Restoring Division instance with the given binary dividend and divisor.

        Arguments:
        dividend (str): The binary dividend for the division.
        divisor (str): The binary divisor for the division.

        Steps:
        1. Pull the sign bit from the dividend and divisor
        2. Remove the sign bit from the dividend and divisor
        3. Add a leading zero to the dividend as an extended bit
        4. Initialize the quotient with the dividend
        5. pad zeros to the accum as the double register if needed
        6. Initialize the iteration and operation counters
        """
        self.signDividend = dividend[0]
        self.signDivisor = divisor[0]
        divisor = divisor[1:]
        dividend = dividend[1:]
        self.dividend = "0" + dividend
        self.divisor = "0" + divisor
        self.quotient = dividend
        self.accum = "0" * len(self.divisor)
        self.iteration_count = 0
        self.operation_count = 0

    def determineSigns(self):
        """
        Determines the final outputs after performing division on the magnitude of the sign numbers
        Logical operator XOR will be performed between Numbers.
        Sign of quotient is XOR of the signs of dividend and divisor and Sign of remainder is same
as the sign of dividend
        """
        self.signQuotient = int(self.signDividend) ^ int(self.signDivisor)
        self.signRemainder = self.signDividend

    def displayResult(self):
        """
        Displays the quotient and remainder of the division in binary format.
        Doesn't process if overflow is detected.

        This prints:
```

```
    - The quotient and remainder of the division in both binary considering sign bit and
hexadecimal.
    - The total number of operations performed.
    - The total number of iterations executed during the division.
    - The execution time of the division process.
    """
    if not checkOverflow(self.dividend, self.divisor):
        self.quotient = str(self.signQuotient) + self.quotient
        self.rem = self.signRemainder + self.rem

        print("Quotient:",self.quotient, "| In Hexadecimal:",Hexadecimal(self.quotient))
        print("Remainder:",self.rem, "| In Hexadecimal:",Hexadecimal(self.rem))
        print("Number of Subtraction/Addition performed: ", self.operation_count)
        print("Number of iteration: ", self.iteration_count)
        print("Execution Time: {:.6f} seconds\n".format(self.end_time - self.start_time))

class Restoring(Division):
    def __init__(self, dividend, divisor):
        super().__init__(dividend, divisor)

    def run(self):
        """
        Executes the Restoring Division Algorithm to divide the binary dividend by the binary
divisor.

        The algorithm involves:
        1. Shifting the combined accumulator and quotient left.
        2. Subtracting the divisor from the accumulator.
        3. Restoring the accumulator if the subtraction results in a negative value.
        4. Don't Restore if the subtaraction results in 0 as extended bit.
        5. Repeating the process until sequence counter flag the end.
        """
        # Perform the overflow check
        self.start_time = time.time()
        print("\nChecking overflow...")
        if checkOverflow(self.dividend, self.divisor):
            print("Overflow Status: Overflow detected. Division cannot be proceed.\n")
            return
        else:
            print("Overflow Status: No overflow detected.")

        print("\nRunning the Restoring Division Method...")

        #No need to pad zeros to the accum as the double register if dividend's length is twice the
divisor's length minus 1
        if len(self.dividend) - 1 == 2 * (len(self.divisor) - 1):
            self.accum = "0"
```

```
        sequence_counter = True
        while sequence_counter:

            combined = self.accum + self.quotient    #Perform shift operation (shift left the combined
accumulator and quotient)
            shifted = shiftLeft(combined)
            self.iteration_count += 1
            operation1 = shifted                    #Store the shifted value for potential restoration

            self.Remainder = shifted[:len(self.divisor)]#Extract the remainder portion for subtraction

            subtracted = subtractBinary(self.Remainder, self.divisor)
            self.operation_count += 1

            operation2 = subtracted + operation1[len(subtracted):]

            #Check the most significant bit (MSB) to determine restoration
            if int(subtracted[0]) == 1:
                operation1 = restore(operation1) #Restore if subtraction resulted in a negative value
(MSB = 1)
                self.operation_count += 1
                self.quotient = operation1[len(self.divisor):]
                self.accum = operation1[:-len(self.quotient)]
            else:
                operation2 += "1"           #No restore needed, update values and set quotient bit to 1
                self.quotient = operation2[len(self.divisor):]
                self.accum = operation2[:-len(self.quotient)]

            sequence_counter = sequenceCounter((len(self.divisor) - 1), self.iteration_count)

        self.rem = self.accum[1:]                                   #Extract remainder
        self.end_time = time.time()

    def displayResult(self):
        self.run()
        super().determineSigns()
        super().displayResult()

class NonRestoring(Division):
    def __init__(self, dividend, divisor):
        super().__init__(dividend, divisor)

    def run(self):
        """
        Executes the Non-Restoring Division Algorithm to divide the binary dividend by the binary
divisor.

        The algorithm involves:
```

```
    1. Shifting the combined accumulator and quotient left.
    2. Subtracting the divisor from the accumulator.
    3. Checking the most significant bit (MSB) of the accumulator:
       - If MSB is 1, set the quotient's LSB bit to 0 and add the divisor
       - If MSB is 0, set the quotient's LSB bit to 1.
    4. Repeating the process until the sequence counter signals the end of the division.
    5. Extracting the final remainder and updating the quotient.
    """
    # Perform the overflow check
    self.start_time = time.time()
    print("Checking overflow...")
    if checkOverflow(self.dividend, self.divisor):
       print("Overflow Status: Overflow detected. Divison cannot be proceed.\n")
       return
    else:
       print("Overflow Status: No overflow detected.")

    print("\nRunning the Non-Restoring Division Method...")

    #No need to pad zeros to the accum as the double register if dividend's length is twice the
divisor's length minus 1
    if len(self.dividend) - 1 == 2 * (len(self.divisor) - 1):
       self.accum = "0"

    sequence_counter = True
    while sequence_counter:

       combined = self.accum + self.quotient
       shifted = shiftLeft(combined)
       self.iteration_count += 1                    #Update the number of iterations
       self.Remainder = shifted[:len(self.divisor)]#Extract the remainder portion for subtraction

       new_iteration = subtractBinary(self.Remainder, self.divisor)
       self.operation_count += 1

       iteration1 = new_iteration + shifted[len(new_iteration):]

       #Check the most significant bit (MSB) to determine if addition is needed
       while int(iteration1[0]) == 1:
          iteration1 += "0"                                    #Add zeros to iteration1
          if self.iteration_count == (len(self.divisor) - 1):
             new_iteration = addBinary(iteration1[:len(self.divisor)], self.divisor)
             self.operation_count += 1
             iteration1 = new_iteration + iteration1[len(new_iteration):]
             break
          else:
             shifted = shiftLeft(iteration1)
             self.iteration_count += 1
```

```
            new_iteration = addBinary(shifted[:len(self.divisor)], self.divisor)
            self.operation_count += 1
            iteration1 = new_iteration + shifted[len(new_iteration):]

        if int(iteration1[0]) == 0 and self.operation_count <= (len(self.divisor) - 1):      #If MSB
is 0 and iteration count is within the limit
            iteration1 += "1"                                                    #Update the iteration with '1'

        self.quotient = iteration1[len(self.divisor):]              #Update quotient and accumulator
        self.accum = iteration1[:-len(self.quotient)]

        sequence_counter = sequenceCounter((len(self.divisor) - 1), self.iteration_count)

    self.rem = self.accum[1:]                                #Extrac remainder (ignoring the sign bit)
    self.end_time = time.time()

  def displayResult(self):
    self.run()
    super().determineSigns()
    super().displayResult()
```

Utilis.py

```
def addBinary(BinNum1, BinNum2):
  """
  Takes two binary numbers as strings (BinNum1 and BinNum2) and performs binary addition.

  Steps:
  1. Convert both binary strings to decimal using base 2.
  2. Perform the addition.
  3. Convert the result back to a binary string.
  4. Trim any overflow bit if the result exceeds the original input length.
  5. return the final binary sum.
  """
  BinarySum = bin(int(BinNum1, 2) + int(BinNum2, 2))

  BinarySum = BinarySum[2:]

  max_len = max(len(BinNum1), len(BinNum2))

  if len(BinarySum) > max_len:
    x = len(BinarySum) - max_len
    return BinarySum[x:]

  BinarySum = BinarySum.zfill(max_len)
  return BinarySum

def FindTwosCom(BinNum):
```

```
    """
    Takes a binary number (as a string) and returns its two's complement.

    Steps:
    1. Compute the one's complement by flipping each bit (0 to 1 and 1 to 0).
    2. Add 1 to the one's complement using the addBinary() function to obtain the two's
complement.
    3. return the result.
    """
    one_s_comp = ""

    for i in BinNum:
        if i == "1":
            i = "0"
            one_s_comp += i
        else:
            i = "1"
            one_s_comp += i

    TwoS_com = addBinary(one_s_comp, "1")

    return TwoS_com

def subtractBinary(BinNum1, BinNum2):
    """
    Takes two binary numbers as strings (BinNum1 and BinNum2) and performs binary
subtraction.

    Steps:
    1. Find the two's complement of BinNum2 using the FindTwosCom() function to negate it.
    2. Add BinNum1 and the two's complement of BinNum2 using the addBinary() function.
    3. Return the result as the binary subtraction output.
    """
    twoS_BinNum = FindTwosCom(BinNum2)
    SubBin = addBinary(BinNum1, twoS_BinNum)
    return SubBin


def shiftLeft(BinNum):
    """
    Performs a left shift operation on a binary number represented as a string.

    Steps:
    1. Remove the first bit (leftmost bit) from the binary string.
    2. Return the remaining binary string.
    """
    BinNum = BinNum[1:]
    return BinNum
```

```
def sequenceCounter(num1, num2):
    """
    Compares two numbers to determine if the maximum number of iterations is reached.

    Steps:
    1. Compare the integers and return the True of False based on that.
    Returns:
    - bool: True if iterations should continue, False if the limit is reached.
    """
    if num1 == num2 or num1 < num2:
        return False
    return True

def restore(BinNum):
    """
    Restores the binary number by adding a '0' to the right, simulating a shift.

    Steps:
    1. Append '0' to the end of the binary string to simulate a restore step.
    2. Return the updated binary number.
    """
    BinNum = BinNum + "0"
    return BinNum

def checkOverflow(BinNum1, BinNum2):
    """
    Checks for overflow during binary division using two's complement arithmetic.

    Steps:
    1. Extract the leftmost bits of BinNum1 (equal to the length of the divisor) to check for
overflow.
    2. Find the two's complement of BinNum2 using the FindTwosCom function.
    3. Perform binary addition using the addBinary function.
    4. Check the most significant bit (leftmost bit) of the result.
        - If it is '1', overflow has occurred (indicating a negative result in two's complement).
        - Otherwise, no overflow.
    """
    num1 = BinNum1[:-(len(BinNum2)-1)]
    num2 = FindTwosCom(BinNum2)

    output = addBinary(num1, num2)

    if output[0] == "1":
        return False
    else:
        return True
```

```
def Hexadecimal(BinNum):
    """
    converts a binary number to hexadecimal.

    steps:
    1. convert it to decimal using int() function
    2. convert the decimal number to hex using hex() function
    4. return the final hex number
    """

    HexNum = hex(int(BinNum, 2))
    HexNum = HexNum[2:].upper()
    return "0x" + HexNum
```