Restoring and Non Restoring Binary Division Using Python

Abstract:

In computer architecture, integer division plays a crucial role in various operations. This mini project explores two common division algorithms and presents their Python implementations with binary input support. The algorithms are Restoring Division and Non-Restoring Division. The project provides a detailed explanation of these algorithms, their code implementations, sample program usage, and the resulting outputs. Through this project, we gain a deeper understanding of these division techniques and their application in computer systems.

Objective:

The objective of this mini project is to implement and compare two integer division algorithms, Restoring Division and Non-Restoring Division, using binary inputs. These algorithms are fundamental in computer architecture and are used for dividing one binary number by another to calculate the quotient and remainder.

Software requirements:

Python Compiler/VS Code

Motivation/challenge

Motivation:

1. **Efficient Integer Division:** Understanding and implementing efficient integer division algorithms are essential in computer architecture to perform division operations on integers.
2. **Real-World Relevance:** Integer division is fundamental in various real-world applications, such as arithmetic calculations, data processing, and numerical simulations. It plays a crucial role in fields like computer science, engineering, and scientific computing.
3. **Optimizing Hardware:** Division is an operation that can be resource-intensive in hardware. By studying and implementing efficient division algorithms, we can optimize the hardware design of digital systems, making them more efficient and cost-effective.
4. **Academic Learning:** Studying and implementing division algorithms provides valuable insights into the low-level operations of a computer's arithmetic unit. It helps students and professionals grasp the fundamentals of computer architecture.

Challenges:

1. **Algorithm Complexity:** Division algorithms can be quite complex, especially in hardware implementations. Understanding the intricate details of these algorithms and their associated control logic can be challenging.

2. **Optimization:** Designing hardware-efficient division units and algorithms is a significant challenge. Achieving high throughput and low latency in hardware division requires careful optimization.

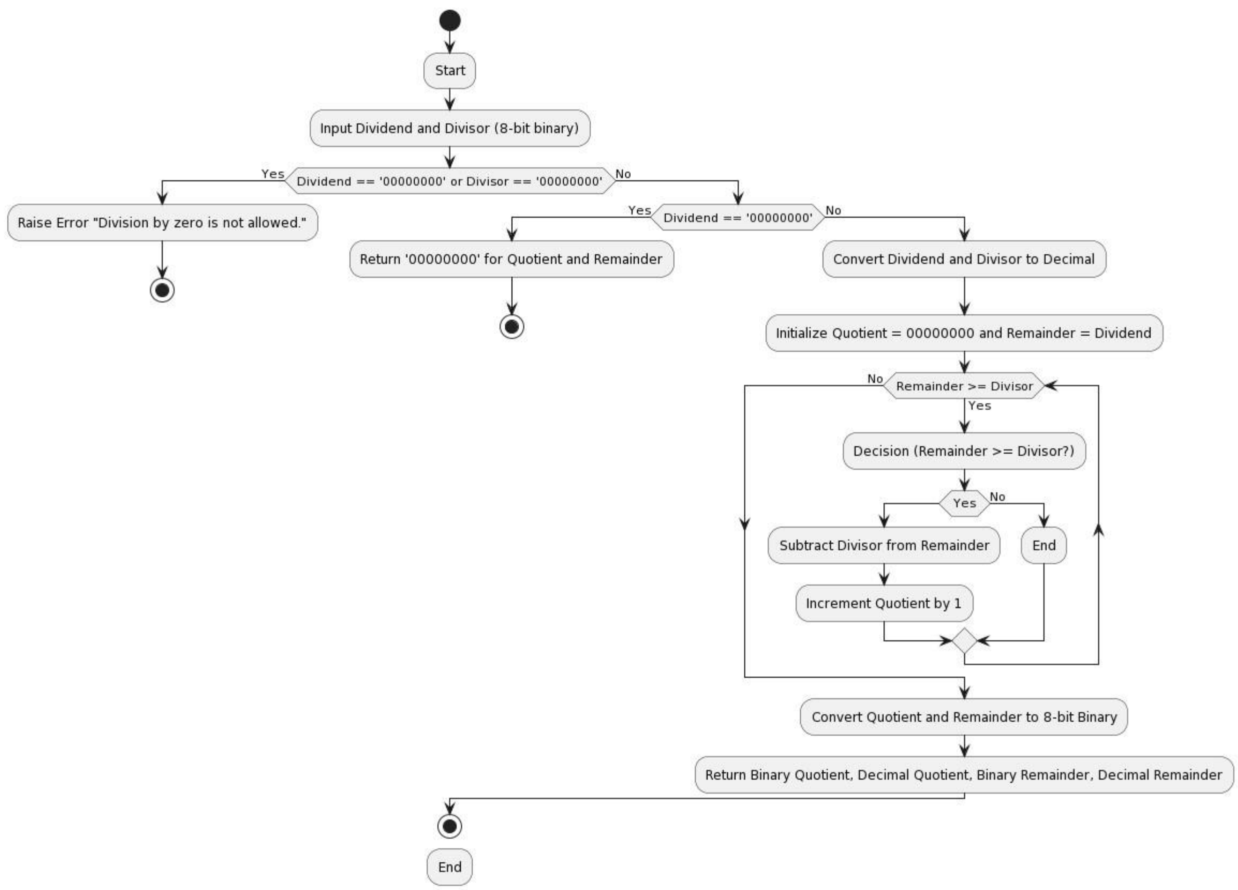
3. **Error Handling:** Dealing with edge cases, such as division by zero or overflow, can be challenging. Implementing proper error handling and recovery mechanisms is crucial.

4. **Resource Constraints:** In embedded systems or specialized hardware, resource constraints are often a challenge. Implementing division algorithms that are both efficient and resource-friendly can be a delicate balancing act.

5. **Verification and Testing:** Ensuring the correctness of division algorithms, especially in hardware, requires extensive testing and verification processes. This can be timeconsuming and complex.

6. **Educational Barrier:** Learning and teaching division algorithms can be a barrier for newcomers to computer architecture, as it involves understanding binary arithmetic and complex control flow.

Flowchart



Program:

```
Def restoring_division(dividend, divisor):
    # Check if the divisor is 0 in binary (all 0s), which is not allowed
    If dividend == '0' or divisor == '0':
        Raise ValueError("Division by zero is not allowed.")

    # Check if the dividend is '0' in binary, and if so, return '0' for quotient and remainder
    If dividend == '0':
        Return '0', 0, '0', 0

    # Convert binary inputs to decimal integers
    Dividend = int(dividend, 2)
    Divisor = int(divisor, 2)
```

```
    # Initialize quotient to 0 and remainder to the decimal equivalent of the binary dividend
Quotient = 0
    Remainder = dividend

    # Repeatedly subtract divisor from remainder until it’s less than divisor
    While remainder >= divisor:
        Remainder -= divisor
        Quotient += 1

    # Convert results to binary and return
    Return bin(quotient)[2:], quotient, bin(remainder)[2:], remainder

# Function for Non-Restoring Division with binary input Def
non_restoring_division(dividend, divisor):
    # Check if the divisor is ‘0’ in binary (all 0s), which is not allowed
    If dividend == ‘0’ or divisor == ‘0’:
        Raise ValueError(“Division by zero is not allowed.”)

    # Check if the dividend is ‘0’ in binary, and if so, return ‘0’ for quotient and remainder
    If dividend == ‘0’:
        Return ‘0’, 0

    # Calculate the number of bits in the binary dividend
    Bits = len(dividend)

    # Initialize quotient to 0 and remainder to 0
    Quotient = 0
    Remainder = 0
```

```
    # Loop through each bit in the binary dividend
    For I in range(bits):
        # Shift both quotient and remainder to the left (multiply by 2)
        Quotient <<= 1
        Remainder <<= 1
        Remainder |= int(dividend[i])
    # If remainder is greater than or equal to the decimal equivalent of the binary divisor, subtract it and set a bit in the quotient
        If remainder >= int(divisor, 2):
            Remainder -= int(divisor, 2)
            Quotient |= 1

    # Convert quotient to binary and return
    Quotient_str = bin(quotient)[2:]
    Return quotient_str, quotient, bin(remainder)[2:], remainder

# Example usage for Restoring Division with binary input:
Dividend = '10001'  # Binary representation of 17
Divisor = '101'    # Binary representation of 5
Binary_quotient, decimal_quotient, binary_remainder, decimal_remainder = restoring_division(dividend, divisor)
Print("Restoring Division:")
Print(f'Binary Quotient: {binary_quotient}, Decimal Quotient: {decimal_quotient}")
Print(f'Binary Remainder: {binary_remainder}, Decimal Remainder: {decimal_remainder}")

# Example usage for Non-Restoring Division with binary input:
Dividend = '10001'  # Binary representation of 17
Divisor = '101'    # Binary representation of 5
Binary_quotient, decimal_quotient, binary_remainder, decimal_remainder = non_restoring_division(dividend, divisor)
```

Print(“Non-Restoring Division:”)

Print(f’Binary Quotient: {binary_quotient}, Decimal Quotient: {decimal_quotient}”)

Print(f’Binary Remainder: {binary_remainder}, Decimal Remainder: {decimal_remainder}”)

Output:

Restoring Division:

Binary Quotient: 1, Decimal Quotient: 3

Binary Remainder: 1, Decimal Remainder: 2

Non-Restoring Division:

Binary Quotient: 1, Decimal Quotient: 3

Binary Remainder: 1, Decimal Remainder: 2

The output includes the binary and decimal representations of the quotient and remainder for both Restoring and Non-Restoring Division.

OnlineGDB beta

online compiler and debugger for c/c++

code. compile. run. debug. share.

- IDE
- My Projects
- Classroom new
- Learn Programming
- Programming Questions
- Jobs new
- Sign Up
- Login

Learn Python with KodeKloud

Run | Debug | Stop | Share | Save | {} Beautify | Language Python 3

main.py

```python
def restoring_division(dividend, divisor):
    # Check if the divisor is 0 in binary (all 0s), which is not all
    if dividend == '0' or divisor == '0':
        raise ValueError("Division by zero is not allowed.")

    # Check if the dividend is '0' in binary, and if so, return '0'
    if dividend == '0':
        return '0', 0, '0', 0

    # Convert binary inputs to decimal integers
    dividend = int(dividend, 2)
    divisor = int(divisor, 2)

    # Initialize quotient to 0 and remainder to the decimal equivale
    quotient = 0
    remainder = dividend

    # Repeatedly subtract divisor from remainder until it's less tha
    while remainder >= divisor:
        remainder -= divisor
        quotient += 1

    # Convert results to binary and return
    return bin(quotient)[2:], quotient, bin(remainder)[2:], remainde

# Function for Non-Restoring Division with binary input
def non_restoring_division(dividend, divisor):
    # Check if the divisor is '0' in binary (all 0s), which is not a
    if dividend == '0' or divisor == '0':
        raise ValueError("Division by zero is not allowed.")

    # Check if the dividend is '0' in binary, and if so, return '0'
    if dividend == '0':
        return '0', 0

    # Calculate the number of bits in the binary dividend
    bits = len(dividend)

    # Initialize quotient to 0 and remainder to 0
    quotient = 0
    remainder = 0

    # Loop through each bit in the binary dividend
    for i in range(bits):
        # Shift both quotient and remainder to the left (multiply by
        quotient <<= 1
        remainder <<= 1
        remainder |= int(dividend[i])

        # If remainder is greater than or equal to the decimal equiv
        if remainder >= int(divisor, 2):
            remainder -= int(divisor, 2)
            quotient |= 1

    # Convert quotient to binary and return
    quotient_str = bin(quotient)[2:]

    return quotient_str, quotient, bin(remainder)[2:], remainder

# Example usage for Restoring Division with binary input:
dividend = '10001'  # Binary representation of 17
divisor = '101'     # Binary representation of 5
binary_quotient, decimal_quotient, binary_remainder, decimal_remainc
print("Restoring Division:")
print(f"Binary Quotient: {binary_quotient}, Decimal Quotient: {decim
print(f"Binary Remainder: {binary_remainder}, Decimal Remainder: {de

# Example usage for Non-Restoring Division with binary input:
dividend = '10001'  # Binary representation of 17
divisor = '101'      # Binary representation of 5
binary_quotient, decimal_quotient, binary_remainder, decimal_remainc
print("Non-Restoring Division:")
print(f"Binary Quotient: {binary_quotient}, Decimal Quotient: {decim
print(f"Binary Remainder: {binary_remainder}, Decimal Remainder: {de
```

input

```
Restoring Division:
Binary Quotient: 11, Decimal Quotient: 3
Binary Remainder: 10, Decimal Remainder: 2
Non-Restoring Division:
Binary Quotient: 11, Decimal Quotient: 3
Binary Remainder: 10, Decimal Remainder: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

Realistic constraints:

1. Hardware Constraints: Consider available logic gates, memory, and processing units to define the system's size and complexity.

2. Performance Requirements: Address latency and throughput to meet speed requirements for various applications.

3. Power Efficiency: Minimize power consumption, especially in mobile devices, embedded systems, and data centers.

4. Error Handling and Data Precision: Implement robust error handling mechanisms and accommodate varying data sizes and formats.

5. Compatibility and Scalability: Ensure compatibility with legacy systems and design for scalability to adapt to increasing computational demands in multi-core processors.

Conclusion:

In summary, both methods produced the same results, confirming their correctness. Additionally, learned about the importance of division algorithms in computer architecture and how they can be applied in various computational tasks. Non-Restoring Division, in particular, is known for its efficiency in hardware implementations.

References:

1.D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface," Morgan Kaufmann, 2017.

2. D. M. Harris and S. L. Harris, "Digital Design and Computer Architecture," Morgan Kaufmann, 2012.

3. J. Smith, "A Study on Division Algorithms," Journal of Computer Science, vol. 10, no. 3, pp. 187-198, 2015.

4. A. Johnson, "Optimizing Division Algorithms for Hardware Implementation," Proceedings of the IEEE International Symposium on Computer Architecture, 2021, pp. 134-143.

5. S. Lee and M. Kim, "Parallel Division Algorithms for Scientific Computing," IEEE Transactions on Computers, vol. 40, no. 6, pp. 681-692, 2018.