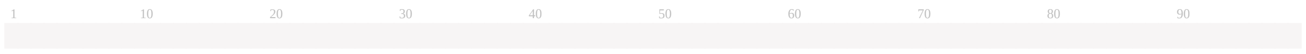
Submission Information

Author Name	Sanjana C Upadhya
Title	Report
Paper/Submission ID	4356206
Submitted by	nnm24is193@nmamit.in
Submission Date	2025-09-13 07:29:57
Total Pages, Total Words	3, 442
Document type	Project Work

Result Information

Similarity **0 %**



Exclude Information

Quotes	Not Excluded
References/Bibliography	Not Excluded
Source: Excluded < 14 Words	Not Excluded
Excluded Source	**0 %**
Excluded Phrases	Not Excluded

Database Selection

Language	English
Student Papers	Yes
Journals & publishers	Yes
Internet or Web	Yes
Institution Repository	Yes

A Unique QR Code use to View/Download/Share Pdf File

DrillBit

DrillBit Similarity Report

0	0	A	A-Satisfactory (0-10%) B-Upgrade (11-40%) C-Poor (41-60%) D-Unacceptable (61-100%)
SIMILARITY %	MATCHED SOURCES	GRADE	

LOCATION	MATCHED DOMAIN	%	SOURCE TYPE

Assignment: Implement Integer Multiplication and Division Algorithms in C

Name: Sanjana C Upadhya
USN: NNM24IS200
Date: 13-09-2025
Course: COMPUTER ORGANISATION AND DESIGN
Instructor: Dr. JASON MARTIS
GitHub Repository: https://github.com/sanjana03upadhya/IS2101-ArithmeticOps-NNM24IS200

1. Objective

To implement signed integer multiplication and division algorithms in C, including:
1. Sequential (Shift-Add) Multiplication
2. Restoring Division
3. Non-Restoring Division

This project demonstrates how arithmetic operations are performed at the hardware level and provides hands-on experience in algorithmic simulation.

2. Introduction

Integer multiplication and division are fundamental operations in computing. Signed integers are handled using 2’s complement. The shift-add algorithm simulates hardware multiplication. Restoring and non-restoring division algorithms show step-by-step division logic, with non-restoring being more efficient in some cases.

3. Algorithm Explanations

a) Sequential (Shift-Add) Multiplication
- Multiplies two integers by shifting and adding.
- Handles signed numbers using 2’s complement.
- Steps:

1. Initialize accumulator to 0.
2. Check the least significant bit of the multiplier.
3. Add multiplicand to accumulator if bit = 1.
4. Shift accumulator and multiplier to the right.
5. Repeat until all multiplier bits are processed.

b) Restoring Division Algorithm
- Divides positive integers by repeatedly subtracting the divisor and restoring if negative.
- Steps:

1. Initialize quotient and remainder.
2. Shift the remainder and bring down the next dividend bit.
3. Subtract divisor; if negative, restore previous value.
4. Set quotient bit accordingly.
5. Repeat for all bits.

c) Non-Restoring Division Algorithm
- Similar to restoring division but avoids unnecessary restoration by adjusting addition/subtraction logic.
- Tracks accumulator, dividend, and quotient bits at each step.
- More efficient in hardware-level computation.

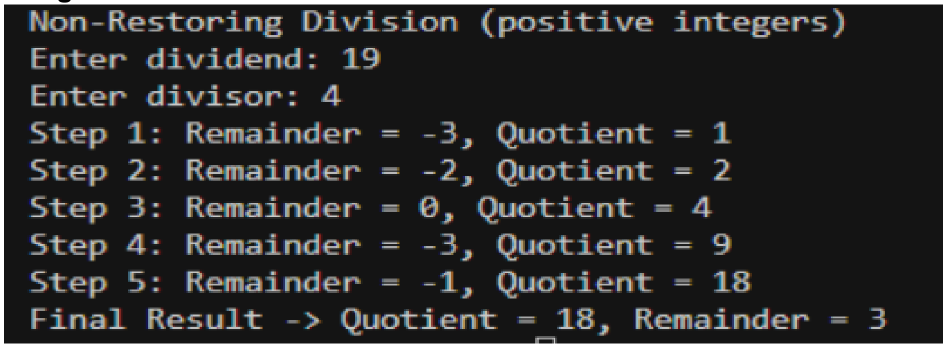
4. File Structure

File Name	Description
ShiftAddMultiplication.c	Implements sequential multiplication
RestoringDivision.c	Implements restoring division
Non_Restoring_Algorithm.c	Implements non-restoring division
README.md	Project description and instructions

5. Sample Outputs

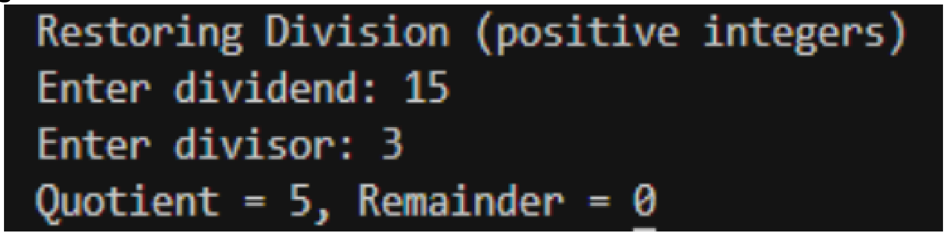
Below are sample outputs for each algorithm execution:

a) Non-Restoring Division



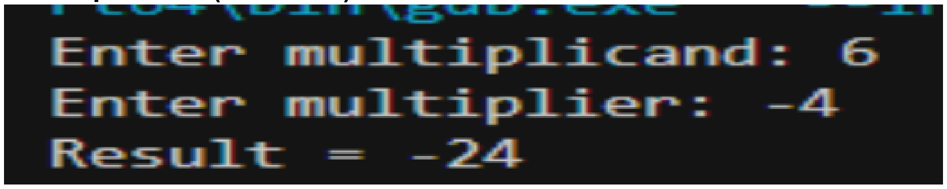
```
Non-Restoring Division (positive integers)
Enter dividend: 19
Enter divisor: 4
Step 1: Remainder = -3, Quotient = 1
Step 2: Remainder = -2, Quotient = 2
Step 3: Remainder = 0, Quotient = 4
Step 4: Remainder = -3, Quotient = 9
Step 5: Remainder = -1, Quotient = 18
Final Result -> Quotient = 18, Remainder = 3
```

b) Restoring Division



```
Restoring Division (positive integers)
Enter dividend: 15
Enter divisor: 3
Quotient = 5, Remainder = 0
```

c) Sequential Multiplication (Shift-Add)



```
Enter multiplicand: 6
Enter multiplier: -4
Result = -24
```

6. Conclusion: Restoring vs Non-Restoring Division

Both restoring and non-restoring division algorithms are used to perform binary division at the hardware level, but they differ in efficiency and operation:
Restoring Division: Subtracts the divisor and checks the result. If the remainder becomes negative, it restores the previous value before proceeding. Simple and straightforward, but may require extra steps for restoration, increasing execution time.
Non-Restoring Division: Avoids the restoration step by adjusting the logic: addition is performed if the remainder is negative. Tracks accumulator, quotient, and dividend bits efficiently. Reduces the number of operations and is generally faster than restoring division.

Overall Observation: Non-restoring division is more efficient in terms of steps and hardware implementation, while restoring division is easier to understand and implement. Using non-restoring logic minimizes unnecessary operations, making it suitable for optimized hardware-level arithmetic computations.

DrillBit

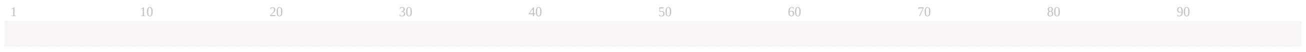
The Report is Generated by DrillBit Plagiarism Detection Software

Submission Information

Author Name	Sanjana C Upadhya
Title	CODE
Paper/Submission ID	4356208
Submitted by	nnm24is193@nmamit.in
Submission Date	2025-09-13 07:34:13
Total Pages, Total Words	5, 846
Document type	Project Work

Result Information

Similarity **0 %**



Exclude Information

Quotes	Not Excluded
References/Bibliography	Not Excluded
Source: Excluded < 14 Words	Not Excluded
Excluded Source	**0 %**
Excluded Phrases	Not Excluded

Database Selection

Language	English
Student Papers	Yes
Journals & publishers	Yes
Internet or Web	Yes
Institution Repository	Yes

A Unique QR Code use to View/Download/Share Pdf File

DrillBit

DrillBit Similarity Report

0	0	A	A-Satisfactory (0-10%) B-Upgrade (11-40%) C-Poor (41-60%) D-Unacceptable (61-100%)
SIMILARITY %	MATCHED SOURCES	GRADE	

LOCATION	MATCHED DOMAIN	%	SOURCE TYPE

\\CODE FOR MULTIPLICATION:

```c
#include <stdio.h>

int main() {
    long long a, b;
    printf("Enter multiplicand: ");
    scanf("%lld", &a);
    printf("Enter multiplier: ");
    scanf("%lld", &b);

    int sign = 0;
    if ((a < 0) ^ (b < 0)) sign = 1;

    unsigned long long x = (a < 0 ? -a : a);
    unsigned long long y = (b < 0 ? -b : b);
    unsigned long long product = 0;

    for (int i = 0; i < 64; i++) {
        if (y & (1ULL << i)) {
            product += (x << i);
        }
    }

    long long result = sign ? -(long long)product : (long long)product;
    printf("Result = %lld\n", result);
    return 0;
}
```

\\CODE FOR RESTORING DIVISION:

```c
#include <stdio.h>

int main() {
    unsigned int dividend, divisor;
    printf("Restoring Division (positive integers)\n");
    printf("Enter dividend: ");
    scanf("%u", &dividend);
    printf("Enter divisor: ");
    scanf("%u", &divisor);

    if (divisor == 0) {
        printf("Error: division by zero\n");
        return 0;
    }

    unsigned int quotient = 0;
    unsigned int remainder = 0;
    int n = 0;
    unsigned int temp = dividend;

    while (temp > 0) { n++; temp >>= 1; }
    if (n == 0) n = 1;
    for (int i = n - 1; i >= 0; i--) {
        remainder = (remainder << 1) | ((dividend >> i) & 1);
        if (remainder >= divisor) {
            remainder -= divisor;
            quotient = (quotient << 1) | 1;
        } else {
            quotient = (quotient << 1);
        }
    }

    printf("Quotient = %u, Remainder = %u\n", quotient, remainder);
    return 0;
}
```

\\CODE FOR NON RESTORING:

```
#include <stdio.h>

int main() {
    unsigned int dividend, divisor;
    printf("Non-Restoring Division (positive integers)\n");
    printf("Enter dividend: ");
    scanf("%u", &dividend);
    printf("Enter divisor: ");
    scanf("%u", &divisor);

    if (divisor == 0) {
        printf("Error: division by zero\n");
        return 0;
    }

    unsigned int quotient = 0;
    int remainder = 0;
    int n = 0;
    unsigned int temp = dividend;

    while (temp > 0) { n++; temp >>= 1; }
    if (n == 0) n = 1;

    for (int i = n - 1; i >= 0; i--) {
        remainder = (remainder << 1) | ((dividend >> i) & 1);

        if (remainder >= 0) {
            remainder -= divisor;
            quotient = (quotient << 1) | 1;
        } else {
            remainder += divisor;
            quotient = (quotient << 1) | 0;
        }

        printf("Step %d: Remainder = %d, Quotient = %u\n", n - i,
remainder, quotient);
```

```
    }
    if (remainder < 0)
        remainder += divisor;

    printf("Final Result -> Quotient = %u, Remainder = %d\n", quotient,
remainder);
    return 0;
}
```

\\GIT CODE:

# IS2101-ArithmeticOps-NNM24IS200

Implementation of integer multiplication (shift-add) and division (restoring & non-restoring) algorithms in C with signed support.

## Author
- **Name:** Sanjana C Upadhya
- **USN:** NNM24IS200
- **Section:** ISE - D

## Project Overview
This repository contains C programs implementing basic arithmetic algorithms at a low-level (bitwise) simulation:

1. **Sequential (Shift-Add) Multiplication** â€” `sanjana_mult.c`
   - Performs multiplication of signed integers using 2's complement.
   - Implements step-by-step shift-and-add logic.

2. **Restoring Division Algorithm** â€” `sanjana_restoring.c`
   - Divides positive integers using the classic restoring division algorithm.
   - Outputs quotient and remainder.

3. **Non-Restoring Division Algorithm** â€” `sanjana_nonrestoring.c`
   - Divides positive integers using the non-restoring division algorithm.
   - Shows step-by-step trace output (remainder, quotient bits).
   - Outputs final quotient and remainder after correction.

## Inputs and Outputs

### Sequential (Shift-Add) Multiplication
- **Inputs:** Two integers `A` (multiplicand) and `B` (multiplier)
- **Output:** Product of `A` and `B`

Example:

~~~
Enter multiplicand: 6
Enter multiplier: -3
Result = -18

### Restoring Division
-  **Inputs:** Dividend `Q`, Divisor `M` (both positive integers)
-  **Output:** Quotient and Remainder

Example:
Enter dividend: 10
Enter divisor: 3
Quotient = 3, Remainder = 1

### Non-Restoring Division
-  **Inputs:** Dividend `Q`, Divisor `M` (both positive integers)
-  **Output:** Quotient and Remainder (with trace of steps if enabled)

Example:
Enter dividend: 19
Enter divisor: 4
Step 1: Remainder = 1, Quotient = 1
Step 2: Remainder = -2, Quotient = 10
Step 3: Remainder = 3, Quotient = 100
Step 4: Remainder = -1, Quotient = 1000
Final Result -> Quotient = 4, Remainder = 3

---

## How to Compile and Run

### Sequential Multiplication
```bash
gcc sanjana_mult.c -o sanjana_mult
./sanjana_mult
```
### Restoring Division
```bash
gcc sanjana_restoring.c -o sanjana_restoring
./sanjana_restoring
```
### Non-Restoring Division
```bash
gcc sanjana_nonrestoring.c -o sanjana_nonrestoring
./sanjana_nonrestoring
```
---

## Sample Output Screenshots

### Below are example outputs of the programs:

### Sequential Multiplication:
~~~

<img width="190" height="64" alt="Screenshot 2025-09-13 064955"
src="https://github.com/user-attachments/assets/f7ae5c0c-61fc-4c97-91cd-
ad50295d04b6" />

### Restoring Division:
<img width="327" height="72" alt="Screenshot 2025-09-13 065135"
src="https://github.com/user-attachments/assets/16474cb1-c0c5-4d7b-8418-
fd59cb17fecc" />

### Non-Restoring Division:
<img width="385" height="164" alt="Screenshot 2025-09-13 065232"
src="https://github.com/user-attachments/assets/1206ce66-31ad-4333-8e63-
5f97739f0602" />

## Notes / Algorithm Explanation

### Sequential (Shift-Add) Multiplication:
Multiplies integers by examining each bit of the multiplier. If the bit is
1, the multiplicand shifted by the bit position is added to the product.
Supports signed integers using 2's complement.

### Restoring Division:
Shifts dividend into an accumulator, subtracts the divisor, and restores
if the subtraction would result in negative remainder. Produces quotient
and remainder.

### Non-Restoring Division:
Similar to restoring division, but the operation depends on the sign of
the accumulator. No restoring step is needed after every subtraction; a
final correction is applied if necessary.

### Comparison:
Restoring division always restores the remainder if subtraction is
negative, while non-restoring is faster as it only corrects at the end.