National University of Computer and Emerging Sciences

Fast School of Computing Fall 2024

CS1002-Programming Fundamentals (CS-A,B,C,D,E,F,G)
Assignment 03

<u>Instructions for submission:</u>

Dear students we will be using auto-grading tools, so failure to submit according to the below format would result in zero marks in the relevant evaluation instrument.

i. For each question in your assignment, make a separate cpp file e.g. for question 1, make ROLL-NUM_SECTION_Q#.cpp (23i-0001_A_Q1.cpp) and so on. Each file that you submit must contain your name, student-id, and assignment # on top of the file in comments.
ii. Combine all your work in one folder. The folder must contain only .cpp files (no binaries, no exe files etc.).
iii. Run and test your program on a lab machine before submission.
iv. Rename the folder as ROLL-NUM_SECTION (e.g. 23i-0001_A) and compress the folder as a zip file. (e.g. 23i-0001_A.zip). do not submit .rar file.
v. Submit the .zip file on Google Classroom within the deadline.
vi. Submission other than Google classroom (e.g. email etc.) will not be accepted.
vii. The student is solely responsible to check the final zip files for issues like corrupt file, virus in the file, mistakenly exe sent. If we cannot download the file from Google classroom due to any reason it will lead to zero marks in the assignment.
viii. Displayed output should be well mannered and well presented. Use appropriate comment and indentation in your source code.
ix. Total Marks: 150.
x. If there is a syntax error in code, zero marks will be awarded in that part of assignment.
xi. Your code must be generic.
xii. Solve the assignment by using the concepts of bitwise operators and iterative structure (while loop only) as well as the concepts we have studied before that.
xiii. You cannot use advance constructs like arrays/value returning functions for this assignment
xiv. Try to submit your assignment 3 hours before the deadline to avoid any problem(e.g; Internet issue etc)

<u>Deadline:</u>

Deadline to submit assignment is 23rd October, 2024 11:59 PM. You are supposed to submit your assignment on GOOGLE CLASSROOM (CLASSROOM TAB not lab). Only ".ZIP" files are acceptable. Other formats should be directly given ZERO. Correct and timely submission of the assignment is the responsibility of every student, hence no relaxation will be given to anyone. **<u>Late Submission policy will be applied as described in course outline</u>**.

Tip: For timely completion of the assignment, start as early as possible.

Plagiarism: Plagiarism is not allowed. If found plagiarized, you will be awarded zero marks in the assignment (copying from the internet is the easiest way to get caught).

<u>Note: Follow the given instruction to the letter, failing to do so will result in a zero.</u>

General Instructions for the assignment:

1. **Variable and Function Naming:** Use variables and function names that reflect the context of the problem. Avoid generic names like \`x\`, \`y\`, or \`z\`.

2. **Logical Thinking:** In your code comments, explain why you chose specific variable names and why a particular operation (like \`+\` or \`%\`) is necessary for the problem's solution. These comments will be checked for correctness.

3. **Code Structure:** Your program should contain a clear \`main()\` function that primarily calls other functions where appropriate. Break down your program into smaller functions where necessary.

4. **Comments and Documentation:** Add a comment at the top of your code that includes your name, roll number, and a brief description of the program. Each function should have a comment explaining its purpose and parameters. Use comments to explain any non-obvious parts of your code.

5. **Input/Output Handling:** Provide clear instructions when taking input from the user. Format your output clearly, ensuring it's easy to understand and follows the requirements of the scenario in the assignment.

Evaluation Criteria

1. Your assignment will be evaluated based on:
2. Correctness: Does the program produce the correct results for all inputs?
3. Complexity: Are multiple conditions and adjustments applied using appropriate decision structures (e.g. switch, nested if-else statements and ternary operators etc)?
4. Efficiency: Is the code clean, efficient, and well-commented?
5. Comprehensive Output: Does the program handle all scenarios with clear and concise output?
6. Error Handling: Ensure the program manages invalid inputs gracefully

Question 1: [30 marks] **“Heroes of the Cosmos” SuperHero Management**

Background

In a new and exciting game called "**Heroes of the Cosmos**," players control powerful superheroes who battle evil forces in a futuristic world. Each superhero has a unique set of powers and stats that influence gameplay. To efficiently manage superheroes, the development team has decided to use bit manipulation techniques, with each superhero represented by a 64-bit integer. As the lead developer, your task is to design and implement a system that manages these superheroes' attributes and powers.

Player Management Overview

Each superhero is represented as a 64-bit integer, where specific ranges of bits encode various stats. The bits are mapped as follows:

- **Bits 0-7:** Level (8-bit integer representing the superhero's experience level)
- **Bits 8-15:** Money (8-bit integer representing the superhero's cash level)
- **Bits 16-23:** Powers (8-bit integer representing the superhero's active powers)
- **Bits 24-31:** Wisdom (8-bit integer representing the superhero's wisdom attribute)
- **Bits 32-39:** Strength (8-bit integer representing the superhero's strength attribute)
- **Bits 40-47:** Endurance (8-bit integer representing the superhero's endurance attribute)
- **Bits 48-55:** Agility (8-bit integer representing the superhero's agility attribute)
- **Bits 56-63:** Speed (8-bit integer representing the superhero's speed attribute)

Superhero Powers

Each superhero can have up to 8 distinct powers, and each power influences the superhero's stats. These powers are stored in bits 16-23 (8 bits total), with each bit representing a power being either active (1) or inactive (0).

When a power is activated, it modifies the superhero’s stats as follows:

- **Sky Soar**: Increases Wisdom and Speed by 2.
- **Blade of Eternity**: Increases Strength by 3.
- **Shadow Cloak**: Increases Endurance by 1.
- **Inferno Burst**: Increases Strength and Endurance by 3.
- **Titan Strength**: Doubles Strength.
- **Photon Dash**: Doubles Speed.
- **Arcane Arsenal**: Increases Agility by 4.
- **Chrono Freeze**: Increases Agility and Endurance by 3.

Program Requirements

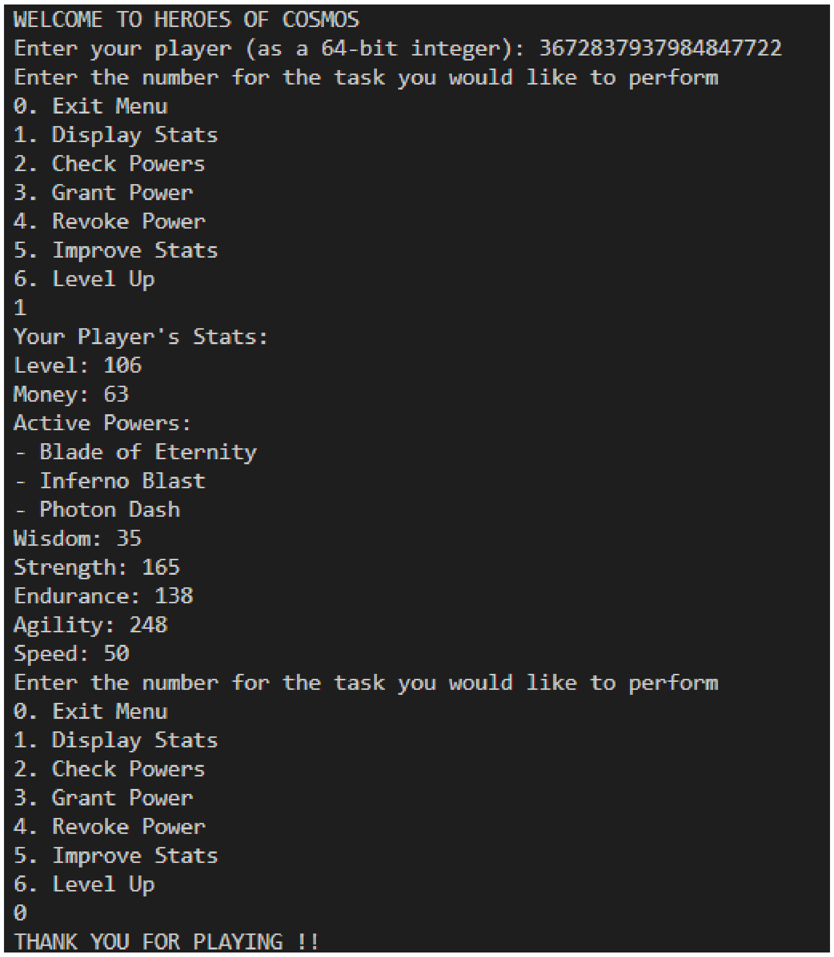
You are tasked with creating a menu-driven interface for the superhero management system, where users can perform actions such as:

- **Display Stats:** Displays the superhero's current level, money, powers, wisdom, strength, endurance, agility, and speed.
- **Check Powers:** Allows the user to check if a specific power is active or inactive.
- **Grant a Power:** Allows the user to activate a new power and adjust stats accordingly.
- **Revoke a Power:** Disables a currently active power and reverts the stat changes associated with it.
- **Improve Stats:** Allows the user to spend 5 units of money to increase any one stat by 1 point.
- **Level Up:** Increases the superhero's level by 1.
    - When leveling up, the superhero automatically gains the next available power in the sequence if it hasn't already been activated.
    - For example, if the superhero's powers are currently 00101011 (Sky Soar, Blade of Eternity, Inferno Burst, Photon Dash), when they level up, they should gain the Shadow Cloak, resulting in a power set of 00101111.

Constraints

- The program must be implemented in C++.
- Ensure the input is a 64-bit integer.
- Provide appropriate error messages where needed.
- All operations should be encapsulated in functions, and these functions should not return any value. You can use global variables.
- The use of loops is prohibited in this program except for the menu display.
- You can use addition and subtraction operators but the use of multiplication, division, and modulus operators is prohibited.
- Use bitwise operators to solve the question.
- You cannot use any external libraries in this question.

Example Output



```
WELCOME TO HEROES OF COSMOS
Enter your player (as a 64-bit integer): 3672837937984847722
Enter the number for the task you would like to perform
0. Exit Menu
1. Display Stats
2. Check Powers
3. Grant Power
4. Revoke Power
5. Improve Stats
6. Level Up
1
Your Player's Stats:
Level: 106
Money: 63
Active Powers:
- Blade of Eternity
- Inferno Blast
- Photon Dash
Wisdom: 35
Strength: 165
Endurance: 138
Agility: 248
Speed: 50
Enter the number for the task you would like to perform
0. Exit Menu
1. Display Stats
2. Check Powers
3. Grant Power
4. Revoke Power
5. Improve Stats
6. Level Up
0
THANK YOU FOR PLAYING !!
```

This is just an example the better the menu the more more you will score.

Question 2:[20] – **Meezan Bank Error Detection System**

Meezan Bank is responsible for ensuring the seamless transfer of sensitive financial transaction data across multiple branches and the central banking system. The nature of the transmitted information, including account details and transaction amounts, demands a robust mechanism for ensuring data integrity and security. Any errors during transmission must be swiftly detected and resolved to avoid data corruption and prevent financial discrepancies.

To maintain data integrity, the bank requires an efficient error detection system that operates at the binary level of each transmitted packet. Every data packet is 64 bits long, with the actual transaction data occupying the first 48 bits and the remaining 16 bits reserved for error detection techniques. These include a parity bit check, a checksum validation, and a secret unique number check. The goal is to provide a high level of assurance that the data transmitted between the bank's systems is free from transmission errors.

System Overview:

The transmission of financial data is prone to errors caused by network noise, signal interference, or hardware malfunctions. To mitigate these risks, the bank uses a combination of error detection techniques, implemented within each 64-bit data packet. The packet is structured as follows:

- **Bits 0-47**: Contains the actual transaction data (e.g., account number, transaction amount).
- **Bits 48-55**: An 8-bit checksum for error detection.
- **Bits 56-62**: A 7-bit secret number used for proprietary error detection.
- **Bit 63**: The parity bit used to detect single-bit errors.

Error Detection Methods:

1. **Parity Bit Check**
   The Most Significant Bit (MSB) of the 64-bit packet (bit 63) is the parity bit. This bit ensures that the total number of 1s in the 48-bit transaction data is even, providing a simple yet effective method to detect single-bit errors during transmission.
   - Calculation Rule:
     - If the total number of 1s in the 48-bit transaction data is odd, the parity bit is set to 0.
     - If the total number of 1s in the 48-bit transaction data is even, the parity bit is set to 1.
   - The parity bit helps detect whether any single bit in the transaction data has been flipped during transmission. While it doesn't provide enough detail to pinpoint the location of the error, it is an immediate indication of data corruption.

2. **Checksum Validation**
   The checksum is an 8-bit value stored in bits 48-55. It is generated based on the transaction data and is used to detect errors such as multiple bit flips.
   - Calculation Rules
     - The transaction data (48 bits) is split into two 24-bit groups: one from the bits at even positions and another from the bits at odd positions.
     - These two 24-bit numbers are then combined in such a way that bits are removed if corresponding bits are 1 in both numbers.
     - The result is divided by two and the final 8 bits (right to left) of this result is the checksum, which is transmitted with the packet.
   - The checksum method helps detect and correct small transmission errors by comparing the recalculated checksum with the transmitted one.

3. **Secret unique Numer**
   The secret unique number is a proprietary 7-bit value stored in bits 56-62. It is calculated from the transaction data as follows:
   - Caluclation Rules
     - Start by toggling all bits of the transaction data.
     - Traverse the modified transaction data from right to left (bit 0 to bit 47).
     - If two alternate bits are the same (either both 1s or both 0s), increment the secret number and multiply it by 2.
     - After processing all the bits, take the last(right to left) 7 bits and toggle the final 7-bit secret number (flip the bits) and transmit it along with the data.
   - The secret number is a sophisticated error detection technique unique to Meezan Bank. It ensures data integrity by detecting unusual bit patterns that may have been introduced during transmission.

4. **Prime Number Check**
   - The transaction data (bits 0-47) must represent a Meezan Prime Number, which adheres to specific divisibility rules. A number is considered a Meezan Prime if it is not divisible by 2, 4, 8, 16, or 32. However, conventional division and modulo arithmetic are not permitted.

Tasks:

Task 1: Input

The system should accept a 64-bit integer from the user.

Task 2: Extract and Verify Data

Implement void functions that accepts a 64-bit integer data. The function should:

- Extract the 48-bit transaction data , the 1-bit parity, the 8 bit checksum, and the 7-bit secret number.

- Recalculate the parity, checksum, and secret number using bitwise operations to verify whether the transmission was error-free.
- Verify whether the transaction data represents a Meezan Prime using bitwise operations.

Task 3: Error Detection Reporting

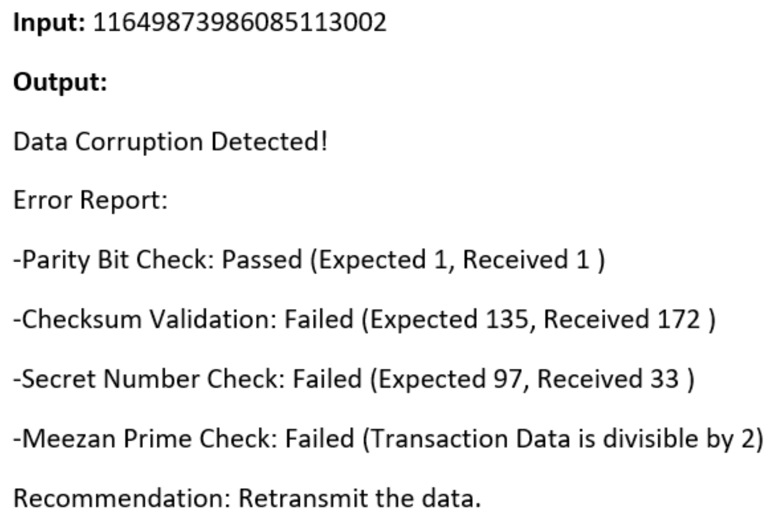
The system must implement a void function that checks whether the transmission was error-free. If an error is detected, the function should notify the user that the data may be corrupted and recommend retransmission. The error detection method (parity bit, checksum, secret number, or Meezan Prime check) used to identify the problem should be clearly communicated to the user.

Restrictions:

The following restrictions must be adhered to when implementing the system:

- No user-defined functions are permitted. The entire error detection logic must be contained within a single block of code.
- No multiplication (*), division (/), or modulo (%) operations allowed.
- No standard library imports are allowed, such as <string>, <cmath>, or <bitset>. All functionality must be implemented using basic C++ constructs and bitwise operations.

Sample Output:



Input: 11649873986085113002

Output:

```
Data Corruption Detected!

Error Report:

-Parity Bit Check: Passed (Expected 1, Received 1 )

-Checksum Validation: Failed (Expected 135, Received 172 )

-Secret Number Check: Failed (Expected 97, Received 33 )

-Meezan Prime Check: Failed (Transaction Data is divisible by 2)

Recommendation: Retransmit the data.
```

Question 3:[40] – **Futuristic Vault System with Toggle Mechanisms**

In the not-too-distant future, advanced vault systems protect high-value assets across various locations. These vaults are secured using binary-state mechanisms, where each vault can either be unlocked (1) or locked (0). The vault system is protected by a series of security agents, each responsible for toggling vaults based on a specific pattern. Additionally, there is a special agent with a unique toggle pattern that targets vaults with prime indices.

The mission is to determine the final locked or unlocked state of a series of vaults after all agents have completed their rounds of toggling. The challenge is to compute the state of the vaults efficiently using bitwise operations, without the use of arrays, external libraries, or the bitset library.

System Overview:

1. **Vault System Setup**:
    - You are in charge of managing n vaults. Each vault is initially unlocked.
    - Each vault has a binary state:
        - 1 represent unlocked.
        - 0 represnets locked.
    - You have n security agents, each with their own toggle pattern for locking/unlocking vaults.
    - There is one special agent with a unique toggle pattern.
2. **Unlocking Pattern:**
    - The special agent toggles every vault that has a prime-numbered index (the first vault is indexed at 1).
    - Each security agent k toggles every k-th vault.
    - For example, the 3rd agent toggles every 3rd vault (vaults 3, 6, 9, etc.).
3. **Mission Requirement**:
    - After all agents have finished toggling the vaults, the system must reflect a specific unlock pattern that determines the final state of all the vaults.
    - The final state of the vaults (locked or unlocked) must be calculated based on the toggle patterns described above.
    - The result must be displayed in binary format as the final state of the vaults.

Input:

The first line of the input indicates number of testcases T .

In every Kth testcase (1< K <= T) , the user enters the number of lamps n ,where (0 < n <= 64)

Output:

For each test case, your task is to display the final state of n lamps in binary format

Sample Input and Output:

```
2
5
0 0 0 0 0
9
0 0 0 0 0 1 0 1 0
```

Constraints:

- You are not allowed to use any kind of array. Zero marks will be awarded for the use of array.
- You are allowed to use loops only in this question
- You are only allowed to use bitwise operators
- You are not allowed to use "bitset" or any other external libraries

Bonus Marks:

Hackerrank is an online platform where you can practice and refine your coding skills. The platform features automatic grading based on predefined test cases. You are required to submit your solution for the contest we have designed specifically for this question. Students who successfully pass all the test cases will receive bonus marks.

Note: The code you submit must be your final solution. If any discrepancies are found, you will lose all marks for this question.

Contest Link : https://www.hackerrank.com/talha-shafi

Question 4:[20] **The Digital Nexus: Full Adders, Half Adders, and the Sacred Art of Bitwise Arithmetic**

In the vast expanse of the *Digital Nexus*, where the ethereal hum of processors reigns supreme, there lies a structure at the very heart of computation: the *Full Adder*. This circuit is no mere collection of wires and gates; it is the embodiment of binary arithmetic, the silent worker behind every addition operation in the ALU. But even the Full Adder itself is not a monolith. No, its inner workings are composed of simpler components—*Half Adders*. These Half Adders are the primal building blocks, the primordial entities from which all addition flows.

Each *Half Adder* is a marvel of digital simplicity. It operates using two key gates: the AND (&) gate and the XOR (^) gate. The AND gate finds the commonality between two bits, the hidden shared energy that must be passed on as a carry, while the XOR gate, ever mystical, blends the bits together in a dance of uniqueness, identifying where they differ to produce the initial sum.

Together, two Half Adders fuse into the Full Adder, a more complex entity capable of not only summing individual bits but also handling the crucial *carry-in*—the ghostly presence left behind by previous bits. This ripple of influence passes through the circuit, each stage managing the carry until the entire 16-bit number is summed.

Now, as the chosen one, you must channel this intricate balance of Half Adders and Full Adders to simulate their behavior in code. Your task is to summon the raw logic of these circuits using only bitwise operations, bringing the ancient knowledge of XOR and AND to life

The Digital Quest of Full Adder

1. **Input**

   The user must enter two unsigned 16-bit integers, the treasures of the tribes. These numbers are stored within the sanctum of binary logic, each occupying 16 bits in the processor's memory.

2. **Full Adder Simulation**

   The ALU, with its Full Adders, will guide your way. Begin by breaking the problem down into Half Adders, where XOR (^) generates the preliminary sum and AND (&) isolates the carry. Once the Half Adders have completed their task, the Full Adder must carry the remaining influence forward, mimicking the intricate cascade of bitwise logic, repeating until no carry remains.

3. **Overflow and Its Mystical Wrath**

   If the sum exceeds the sacred 16-bit limit (65535), chaos could descend upon the binary realm. To prevent this, you must trim the result to fit within the 16-bit boundary, using the masking powers of bitwise operations. Furthermore, you must detect and report the presence of overflow, for it signifies a disturbance in the natural order.

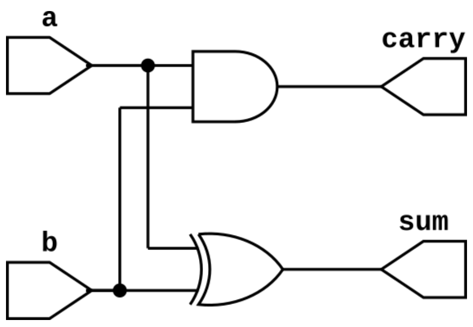
4. **Output**

   Display the final treasure value as it exists within the 16-bit boundary. Alongside it, issue a proclamation indicating whether the sum has overflowed beyond the confines of the 16-bit world.

The Inner Mechanics – Half Adders and Full Adders

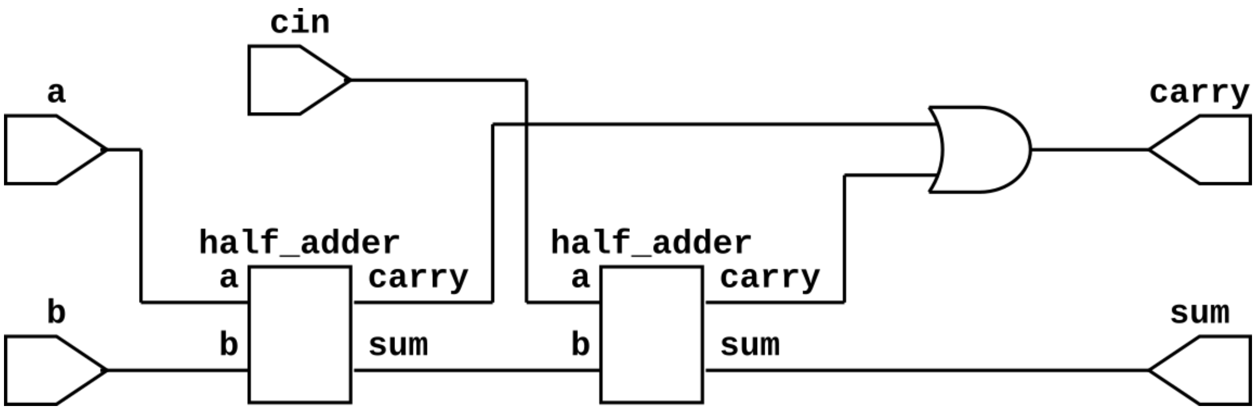
- **Half Adder**

The Half Adder, using the XOR gate, computes the preliminary sum of two bits. The AND gate detects whether these bits share a common 1, sending this carry forward to the next higher bit.



- **Full Adder**

The Full Adder, a combination of two Half Adders, extends this logic. It handles not only the bits being summed but also the carry-in from lower-order bits, ensuring that the ripple of influence from past additions is accounted for.



The Sacred Computation Examples

Case **1**:
Input:

- a = 60000
- b = 60000

The Full Adder, composed of its Half Adder counterparts, will now perform its duty:

- XOR blends the bits to form a temporary sum.

- AND, recognizing shared bits, passes the carry onward.
- The Full Adder’s recursive logic ensures that every carry is managed, and the result fits within 16 bits.

Output:

- Result: 54464
- Overflow: Yes

Case **2**:
Input:

- a = 1000
- b = 100

The simple yet profound logic of the Half Adder handles each bit:

- XOR performs the preliminary addition, while AND captures any carry to pass along.
- The result is calculated without overflow, safely confined to 16 bits.

Output:

- Result: 1100
- Overflow: No

The Journey of Bitwise Operations

You must now take these insights, distilled from the Full Adder's architecture, and implement them in C++. Your code shall weave the powers of XOR and AND into a tapestry of bitwise operations, simulating the addition of two treasures. Let the processor guide you as you explore the depths of binary addition, and beware of the overflow, for it will signal your mastery—or failure—over the digital domain.

Question 5: [40] **The Pyramid Code**

In 2024, a team of archaeologists uncovers a hidden chamber inside the Great Pyramid of Giza. Within the chamber lies an ancient manuscript revealing an encryption system named **"The Pyramid Code"**, believed to have safeguarded the pharaoh's most valuable secrets. This code relies on mathematical operations and a secret key to encode messages.

The scholars now need computer science students to decode the ancient logic and implement it using C++. This encryption method is both unique and strict: it requires a 16-digit key, split into four segments, and the use of specific mathematical operations.

Students must design the program to:

1. Encrypt a message based on the secret key.
2. Decrypt the encrypted message back into the original message using the same key.

However, the ancient scribes left challenging conditions for anyone trying to use the code—making it a task fit for the finest programmers!

The Pyramid Code System

The code uses a 16-digit secret key, which is divided into four equal parts of 4 digits each:

- **Part A**: Digits 1-4
- **Part B**: Digits 5-8
- **Part C**: Digits 9-12
- **Part D**: Digits 13-16

Each part plays a distinct role in the encryption process.

Encryption Process

1. Convert each character of the message into its ASCII code value.
    - For example, 'H' = 72, 'E' = 69, and so on.
2. Process each character sequentially:
    - For the first character, multiply its ASCII code value by Part A of the key.
    - For the second character, multipliy its ASCII code value by Part B of the key.
3. Apply an XOR operation with Part C of the key on the results.
4. Divide the results by Part D of the key and round to the nearest integer.

5. Repeat steps 2-4 for every pair of adjacent characters until the message is encrypted. Forexample for “HELLO” process “H and E” and then “L and L” and so on
6. Special Rule: If the message has an odd number of characters, the last character is multiplied only by Part A and XORed with Part C.

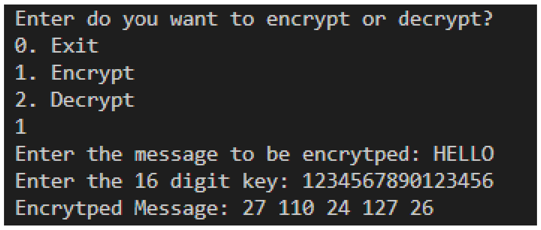
Decryption Process

1. For each encrypted value, multiply by Part D and reverse the XOR operation with Part C.
2. Extract the original ASCII values by reversing the multiplication steps using Part A and Part B.
3. Handle odd-length messages accordingly during decryption.
4. The input for decryption would be a string in the form "27 110 24 127 26" where each character is encoded in a number, that is separated by spaces. As arrays are not allowed in this question the input for decryption would be a maximum of 5 characters as shown above.

Constraints

1. Arrays are strictly forbidden.
2. Students must only use while loops for this question.
3. No functions can return values—all operations must occur within void functions.
4. Precision issues during division must be carefully handled, and integer overflow/underflow must be considered.
5. Error Handling: Make sure the program gracefully handles non-alphabetic characters.
6. Edge Cases: Handle odd-length messages correctly.
7. Testing: Students should verify encryption and decryption with multiple inputs to ensure accuracy.
8. You can use the string and cmath library for this question only.

Example Data



```
Enter do you want to encrypt or decrypt?
0. Exit
1. Encrypt
2. Decrypt
1
Enter the message to be encrytped: HELLO
Enter the 16 digit key: 1234567890123456
Encrytped Message: 27 110 24 127 26
```

*****___*** HAPPY CODING!   ***___*****