EXPERIMENT – 01

AIM:

Develop a Java/Python/Go program to create Elliptic curve public and private keys and demonstrate working of hash functions like SHA256 and ECC digital signatures.

DESCRIPTION:

This program demonstrates **Elliptic Curve Cryptography (ECC)** by generating public and private key pairs, hashing data using **SHA-256**, and implementing **Elliptic Curve Digital Signature Algorithm (ECDSA)** for message signing and verification. ECC is widely used for secure communications due to its strong security with smaller key sizes compared to RSA.

Key Features:

1. **Elliptic Curve Key Pair Generation**
   - Uses the secp256r1 curve to create a secure key pair.
   - Ensures strong encryption with minimal computational overhead.
2. **SHA-256 Hashing**
   - Converts input data into a fixed 256-bit hash for integrity verification.
   - Essential for digital signatures and secure password storage.
3. **ECDSA Digital Signature**
   - Signs a message using the private key to ensure authenticity.
   - Verifies the signature using the public key, preventing tampering.

Libraries Used:

- **Java** – Bouncy Castle, java.security
- **Python** – ecdsa, hashlib
- **Go** – crypto/ecdsa, crypto/sha256

Applications:

- **Blockchain Transactions**
- **Secure Authentication & Digital Certificates**
- **IoT Secure Communications**

This implementation provides a strong foundation for cryptographic security, ensuring data integrity, authenticity, and non-repudiation in secure systems.

PROGRAM:

ECC.py

C: > Users > HP > Downloads > ECC.py > ...

```python
import hashlib
from ecdsa import SECP256k1, SigningKey, VerifyingKey
def generate_keys():
    signing_key = SigningKey.generate(curve=SECP256k1)
    verifying_key = signing_key.get_verifying_key()
    return signing_key, verifying_key
def create_sha256_hash(message):
    return hashlib.sha256(message.encode()).hexdigest()
def sign_message(signing_key, message):
    message_hash = create_sha256_hash(message)
    signature = signing_key.sign(message_hash.encode())
    return signature
def verify_signature(verifying_key, message, signature):
    message_hash = create_sha256_hash(message)
    try:
        verifying_key.verify(signature, message_hash.encode())
        print("Signature is valid.")
    except:
        print("Signature is invalid.")
private_key, public_key = generate_keys()
print("Private Key:", private_key.to_string().hex())
print("Public Key:", public_key.to_string().hex())
message = "This is a secret message."
signature = sign_message(private_key, message)
print("Signature:", signature.hex())
verify_signature(public_key, message, signature)
```

OUTPUT:

```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python312/python.exe c:/Users/HP/Downloads/ECC.py
Private Key: 1de7958ecc3bf87a6bffd371471ba6ce81e4e0e15769751b5d433fec26851e7c
Public Key: c65e656210406e2a36018e69a4c617e9e7acadfb54d63cb78b588808db80063b6565082ff61d507a20ba537a7f0c42993
Signature: 9c5d84e0dd0a635f65e3eefd16975582aad160d0a9a45e140cadb80ed414eae0e7c40499f164926b346acc78a182bac485
Signature is valid.
```

OUTPUT ANALYSIS:

The output includes the **EC key pair**, a **SHA-256 hash**, and an **ECDSA signature**. The signature verification step confirms authenticity, displaying **"Valid Signature"** if unchanged or **"Invalid Signature"** if altered. This ensures **data integrity, security, and efficiency**, making ECC ideal for secure communications and blockchain applications.

EXPERIMENT – 02

AIM:

Setup a Bitcoin wallet like Electrum and demonstrate sending and receiving Bitcoins on a testnet. Use Blockchain explorer to observe the transaction details.

DESCRIPTION:

This experiment demonstrates setting up an Electrum Bitcoin wallet on a testnet, allowing users to send and receive testnet Bitcoins for practice without real monetary risk. The testnet is a parallel Bitcoin network designed for development and testing, where transactions function just like on the mainnet but use worthless coins.
By configuring Electrum for the testnet, users can generate Bitcoin addresses, receive testnet BTC from faucets, and send transactions to other testnet users. Once a transaction is made, a blockchain explorer is used to track and verify details like transaction ID, sender, receiver, fees, and confirmations.
This demonstration helps understand Bitcoin wallet operations, address generation, transaction fees, confirmations, and blockchain transparency. It is useful for developers and cryptocurrency enthusiasts to practice securely before handling real Bitcoin.

PROCEDURE:

Step 1: Download and Install Electrum
- Visit electrum.org and download the latest version for your operating system.
- Install and launch Electrum.

Step 2: Configure Testnet Mode
- Open a terminal or command prompt and run: *electrum --testnet*
- If using the GUI, enable testnet mode in **Tools → Network → Testnet**.

Step 3: Create a New Wallet
- Select **Standard Wallet** and choose **Create a New Seed**.
- Save the **12-word seed phrase** securely.
- Set a **strong password** and confirm wallet creation.

Step 4: Receive Testnet Bitcoins
- Click **Receive** and copy your **testnet Bitcoin address**.
- Visit a testnet faucet (e.g., testnet-faucet.mempool.co) to request free BTC.
- Wait for confirmation (may take a few minutes).

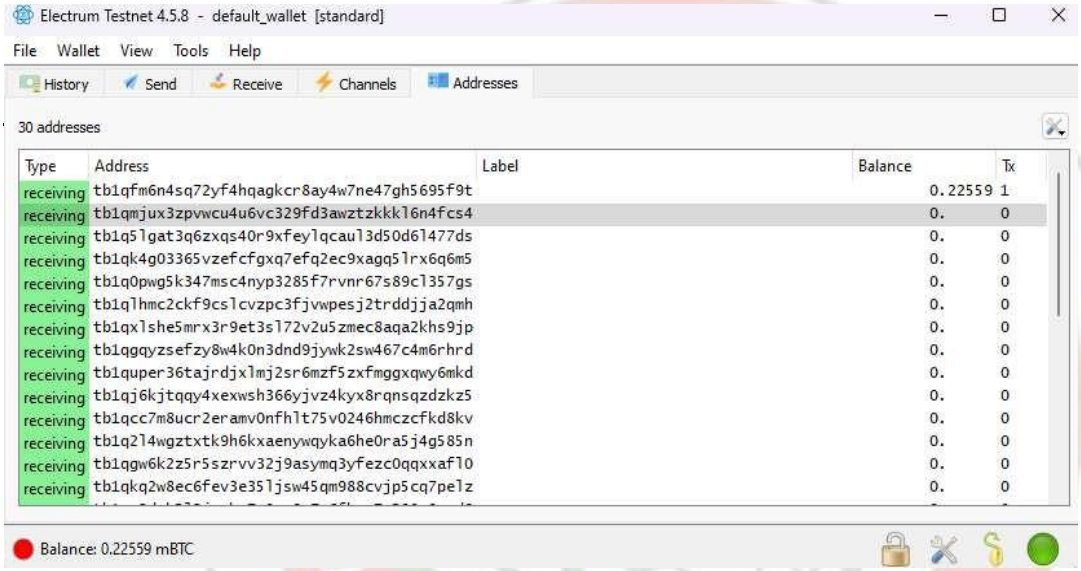
Step 5: Send Testnet Bitcoin
- Click **Send**, enter the recipient's **testnet Bitcoin address**, and specify the amount.
- Set a **transaction fee** (higher fees confirm faster).
- Click **Send** and confirm the transaction.

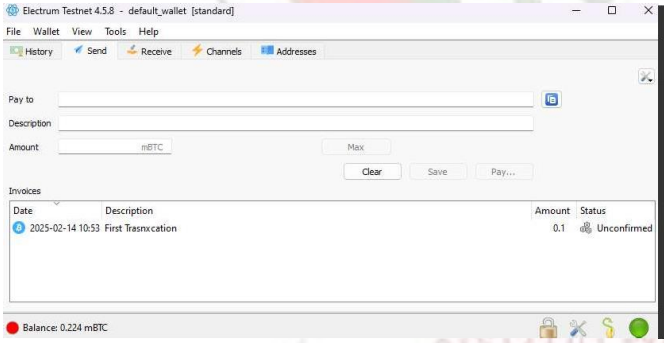
Step 6: Verify Transaction on Blockchain Explorer
- Copy the **transaction ID (TXID)** from Electrum.
- Visit a testnet blockchain explorer like blockstream.info/testnet/ and search for the TXID.
- View transaction details like **sender, receiver, fees, and confirmations**.

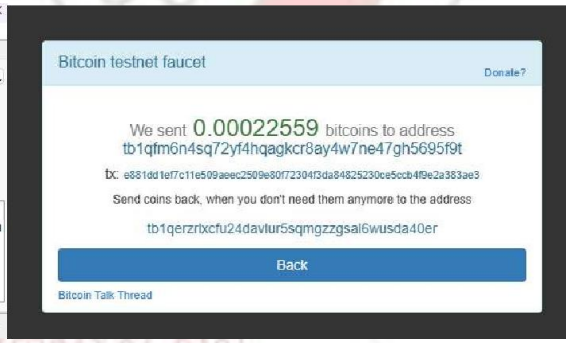
This process helps understand **Bitcoin transactions, network confirmations, and blockchain transparency** in a risk-free environment.

OUTPUT SCREENSHOTS:

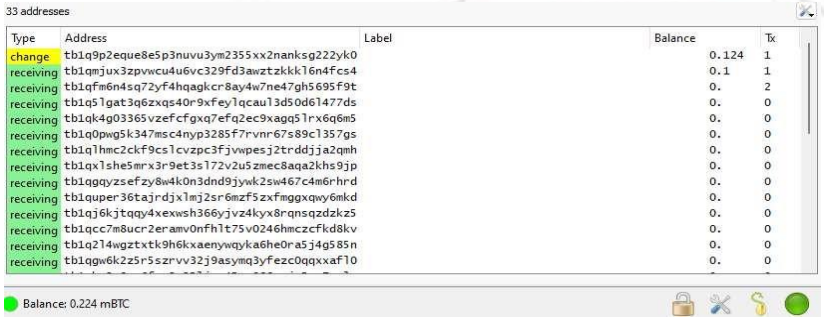
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Program Files (x86)\Electrum> ./electrum-4.5.8 --testnet
```



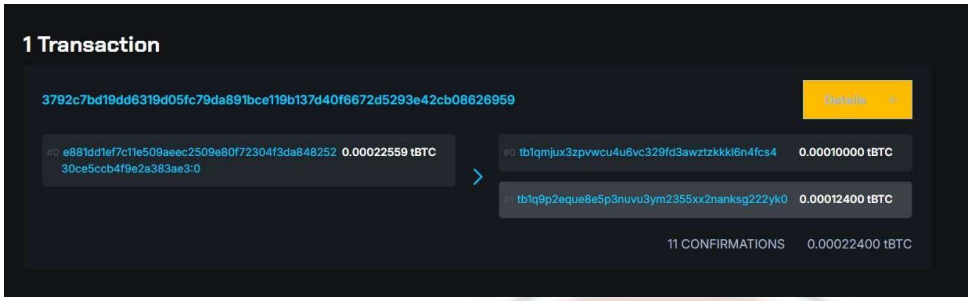




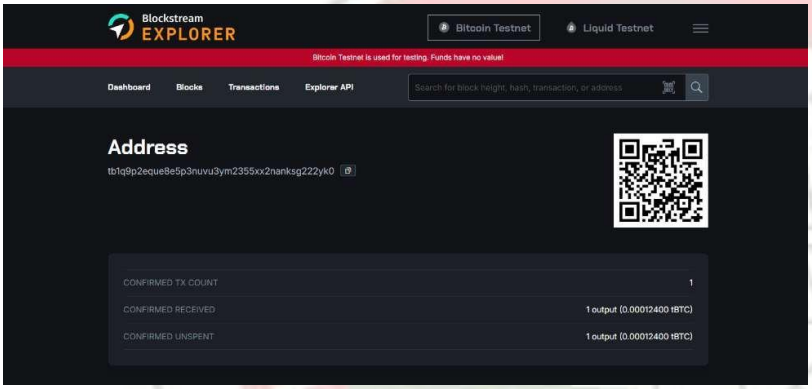
First Transaction **Amount Transfer (Faucet)**



Balance in Ledger Sheet



Updated in Blockstream Explorer



Blockstream Explorer (Verification)

OUTPUT SCREENSHOTS:

1. **Wallet Setup:** Electrum successfully generates a testnet Bitcoin wallet with a unique public address and a 12-word seed phrase for recovery.
2. **Receiving Testnet BTC:** The wallet balance updates after receiving funds from a testnet faucet, with transaction details visible in the history tab.
3. **Sending Bitcoin:** After initiating a transaction, the wallet shows a pending status, updating to confirmed once included in a testnet block.
4. **Transaction Verification:** The blockchain explorer displays details such as transaction ID (TXID), sender, recipient, fee, and confirmations, ensuring transparency.
5. **Conclusion:** This demonstrates secure Bitcoin transactions, fee adjustments, and blockchain verification, essential for real-world cryptocurrency usage.

EXPERIMENT – 03

AIM:

Setup metamask wallet in a web browser and create wallet and user accounts. Demonstrate sending and receiving ethers on a testnet (Sepolia). Use Block explorers like etherscan to observe the transaction details.

DESCRIPTION:

This experiment explains how to set up a MetaMask wallet in a web browser, create a wallet and user accounts, and demonstrate sending and receiving Ether (ETH) on the Sepolia testnet. MetaMask is a widely used browser extension for managing Ethereum-based assets and interacting with decentralized applications (dApps). The Sepolia testnet is a simulated Ethereum network for developers and testers, allowing transactions without real funds. Users can obtain free ETH from testnet faucets, send ETH to other accounts, and verify transactions on block explorers like Etherscan. By following this process, users will understand Ethereum wallet management, address generation, gas fees, transaction confirmations, and blockchain transparency in a risk-free environment before handling real assets.

PROCEDURE:

Step 1: Install MetaMask Extension

- *Open **Google Chrome, Firefox, Brave, or Edge**.*
- *Visit metamask.io and click **Download**.*
- *Install the extension and **pin it to your browser** for easy access.*

Step 2: Create a New Wallet

- *Click **"Get Started"** and select **"Create a Wallet"**.*
- *Set a **strong password** and confirm.*
- *Save the **12-word Secret Recovery Phrase** securely (**do not share it**).*
- *Confirm the **recovery phrase** to complete the wallet setup.*

Step 3: Configure Sepolia Testnet

- *Open **MetaMask** and click on the **network dropdown** at the top.*
- *Select **"Show/Hide test networks"** in settings and **enable it**.*
- *Choose **Sepolia Testnet** from the list.*

Step 4: Create Additional User Accounts

- *Click the **account icon** and select **"Create Account"**.*
- *Name the **new account** and repeat for multiple accounts if needed.*

Step 5: Receive Sepolia Testnet ETH

- *Copy your **Sepolia wallet address** from MetaMask.*
- *Visit a **Sepolia testnet faucet** (e.g., sepoliafaucet.com).*
- *Paste your **wallet address** and **request free ETH**.*
- *Wait for the **transaction confirmation**.*

Step 6: Send ETH on Sepolia Testnet

- *Open **MetaMask** and click **Send**.*
- *Enter the recipient's **Sepolia testnet address**.*
- *Specify the **amount of ETH** to send.*
- *Adjust the **gas fee** (**higher fee = faster confirmation**).*

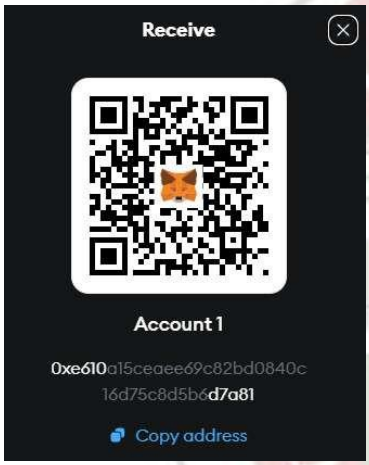
- *Click **Confirm** to broadcast the transaction.*

Step 7: Verify Transactions on Etherscan

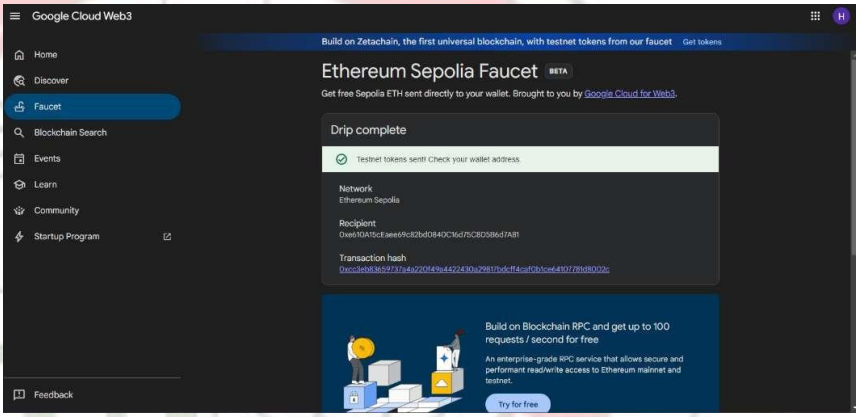
- *Copy the **Transaction Hash (TXID)** from MetaMask.*
- *Visit sepolia.etherscan.io and paste the **TXID**.*
- *View **transaction details**, including sender, receiver, gas fees, and confirmations.*

*This process demonstrates **Ethereum wallet operations, gas fees, transaction speeds, and blockchain verification** in a risk-free testnet environment.*

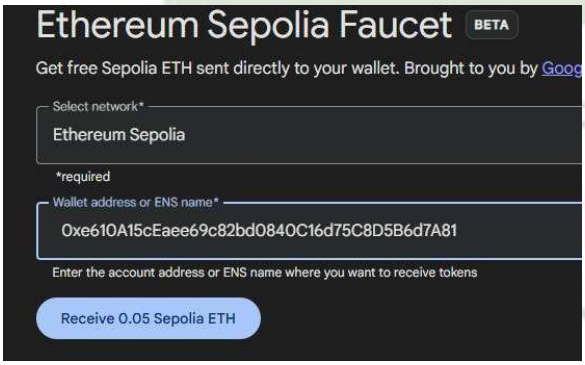
OUTPUT:



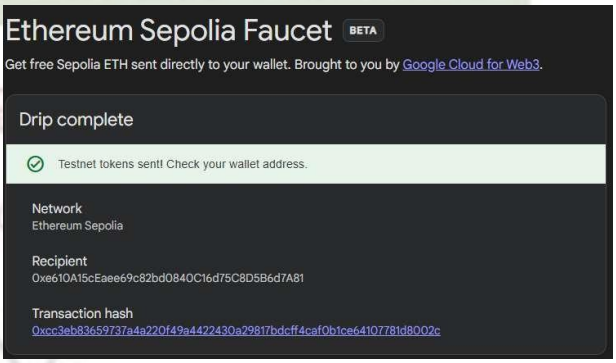
Metamask



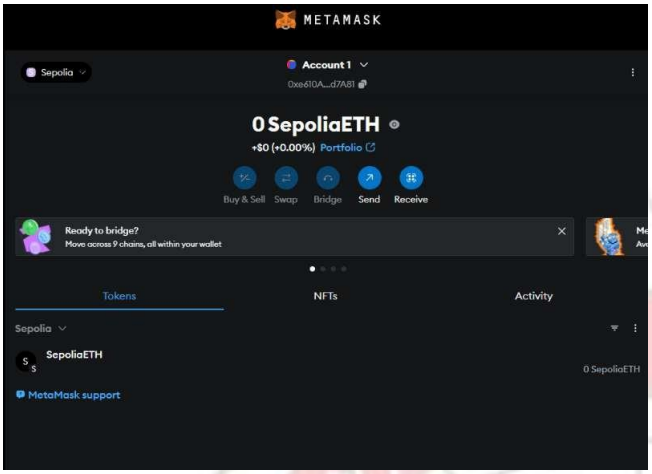
Ethereum Sepolia Faucet



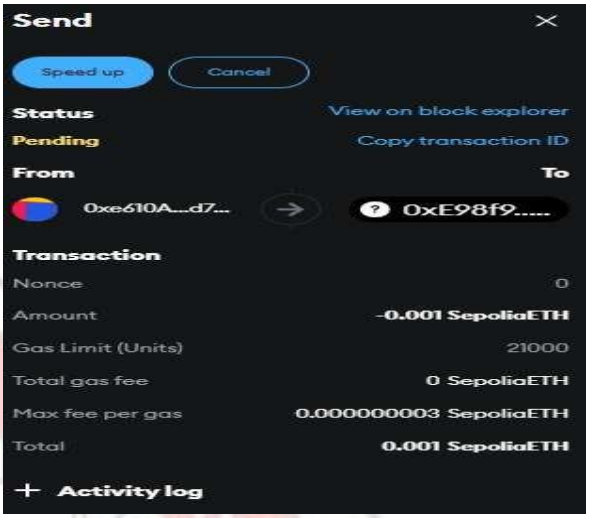
Faucet Address Transfer



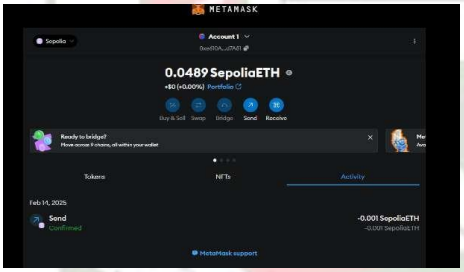
Amount Transfer (Faucet)



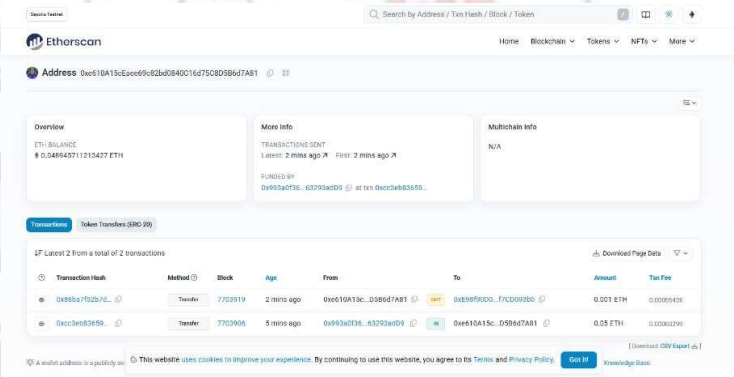
Sepolia Account



Transaction



Updated Sepolia Account



Etherscan Explorer (Verification)

OUTPUT ANALYSIS:

1. **Wallet Creation:** MetaMask successfully generates a **new Ethereum wallet** with a unique address and a **Secret Recovery Phrase** for backup.
2. **Testnet Configuration:** The wallet is connected to the **Sepolia testnet**, enabling test transactions.
3. **Receiving ETH:** After requesting from a faucet, the **wallet balance updates**, confirming receipt of test ETH.
4. **Sending ETH:** Upon sending ETH, the transaction appears as **"Pending"**, then updates to **"Confirmed"** once included in a block.
5. **Transaction Verification:** The **block explorer (Etherscan)** displays details like **transaction hash, sender, receiver, gas fees, and block confirmations**, ensuring transparency.
6. **Conclusion:** This demonstrates **Ethereum transactions, gas fees, and blockchain verification**, essential for learning before using real ETH.

EXPERIMENT – 04

AIM:

Launch Remix web browser and write a smart contract using the solidity language for the “Hello World program”.

DESCRIPTION:

This experiment explains how to write, compile, and deploy a simple "Hello World" smart contract using Solidity in Remix IDE, an online development environment for Ethereum smart contracts. Solidity is the primary programming language for Ethereum, enabling the creation of dApps and smart contracts. The Hello World smart contract contains a function that returns the string "Hello, World!" when called. This program helps understand basic Solidity syntax, smart contract structure, function visibility, and deployment processes. Users will also learn how to interact with a deployed contract in Remix. First, the contract is written in Solidity, defining a public function that returns the message. It is then compiled using the Solidity Compiler in Remix. After successful compilation, the contract is deployed on a local EVM using Remix's built-in JavaScript VM. Once deployed, users can call the contract’s function through Remix’s interface, which displays "Hello, World!" as output. This process provides an introductory hands-on experience in smart contract development, helping users grasp fundamental Ethereum blockchain concepts, Solidity programming, and contract execution before working on more advanced decentralized applications.

PROCEDURE:

Step 1: Open Remix IDE

- Go to remix.ethereum.org in your web browser.
- Select **Solidity** as the development environment.

Step 2: Create a New Solidity File

- Click on the **File Explorer** (left panel) and create a new file named **HelloWorld.sol**.

Step 3: Write the Solidity Smart Contract

- Open the newly created file and write the following Solidity code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract HelloWorld {
    function getMessage() public pure returns (string memory) {
        return "Hello, World!";
    }
}

```

Step 4: Compile the Smart Contract

- Go to the **Solidity Compiler** tab on the left panel.
- Select **Compiler Version 0.8.x** (Ensure it matches pragma solidity ^0.8.0;).
- Click **Compile HelloWorld.sol** (Check for errors in the console).

Step 5: Deploy the Smart Contract

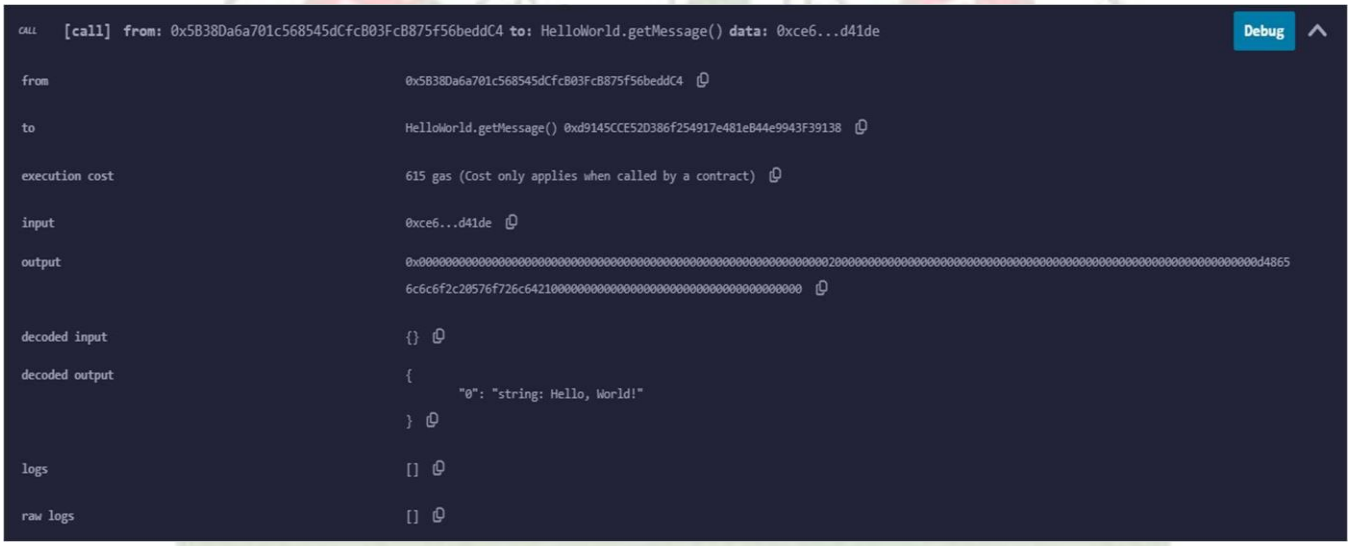
- Navigate to the **Deploy & Run Transactions** tab.
- Under **Environment**, select **"JavaScript VM (London)"** for local deployment.
- Click **Deploy** and check the **Deployed Contracts** section.

Step 6: Interact with the Smart Contract

- Click on the **getMessage()** button in the deployed contract.
- The output will display **"Hello, World!"**, confirming successful execution.

This process provides a basic understanding of **Solidity, contract compilation, and deployment using Remix IDE**.

OUTPUT:



OUTPUT ANALYSIS:

1. Compilation Output:
   - If the contract compiles successfully, no errors appear in the Remix compiler console.
   - Warnings, if any, will indicate best practices or optimizations.
2. Deployment Output:
   - After clicking Deploy, the Deployed Contracts section lists the deployed instance with its contract address.
   - The JavaScript VM (London) provides a simulated blockchain environment, meaning the deployment does not cost real ETH.
3. Function Execution Output:
   - Clicking getMessage() calls the function and returns the expected output: "Hello, World!".
   - The function call is executed instantly because it is a pure function that does not modify the blockchain state.

4. Transaction Details:
   - If deployed on a testnet, the transaction details (gas fee, block confirmation) can be verified on Etherscan.
   - The transaction hash and gas usage appear in the Remix console when executing blockchain-modifying functions.
5. Conclusion:
   - The contract runs as expected, demonstrating basic Solidity syntax, function execution, and Remix IDE usage.
   - This foundational example helps understand smart contract interaction before deploying real-world applications.

EXPERIMENT – 05

AIM:

5(a) - Write Solidity program for incrementing/decrementing a counter variable in a smart contract.

DESCRIPTION:

In this experiment, we will develop a basic smart contract using Solidity, a programming language specifically designed for writing smart contracts on the Ethereum blockchain. The focus of this contract will be to manage a simple counter variable, allowing users to increment and decrement its value.

Smart Contract Overview:

- **State Variables:** The contract will have a single state variable, count, which will store the current value of the counter. This variable is private, meaning it cannot be accessed directly from outside the contract but can be modified and read through designated functions.
- **Functions:**
    1. **Increment Function:** The increment function allows users to increase the value of count by one. This is a straightforward operation, reflecting the basic functionality of a counter.
    2. **Decrement Function:** The decrement function decreases the value of count by one, but it includes a safeguard using the require statement to ensure that the counter does not fall below zero. This prevents underflow, a common issue in programming where a variable goes below its minimum value.
    3. **Get Count Function:** The getCount function returns the current value of count. This function is marked as view, indicating that it does not modify the state of the contract, allowing users to read the counter value without changing it.

PROCEDURE:

Step 1: Open Remix IDE

- Go to remix.ethereum.org in your web browser.
- Select **Solidity** as the development environment.

Step 2: Create a New Solidity File

- Click on the **File Explorer** (left panel) and create a new file named **counter.sol**.

Step 3: Write the Solidity Smart Contract

- Open the newly created file and write the following Solidity code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Counter {
    uint private count;

    function increment() public { count += 1; }    infinite gas
    function decrement() public { require(count > 0, "Counter cannot be negative"); count -= 1; }
    function getCount() public view returns (uint) { return count; }    2432 gas
}

```

Step 4: Compile the Smart Contract

- Go to the **Solidity Compiler** tab on the left panel.
- Select **Compiler Version 0.8.x** and click **Compile counter.sol**.

Step 5: Deploy the Smart Contract

- Navigate to the **Deploy & Run Transactions** tab.
- Under **Environment**, select **"JavaScript VM (London)"** for local deployment.
- Click **Deploy** and check the **Deployed Contracts** section.

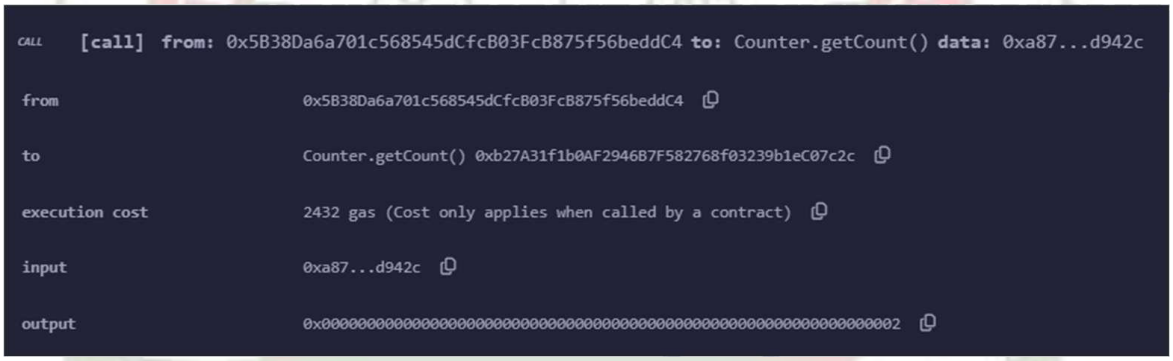
Step 6: Interact with the Smart Contract

- Click **increment()** or **decrement()** to modify the counter.
- Click **getCount()** to retrieve the current value.

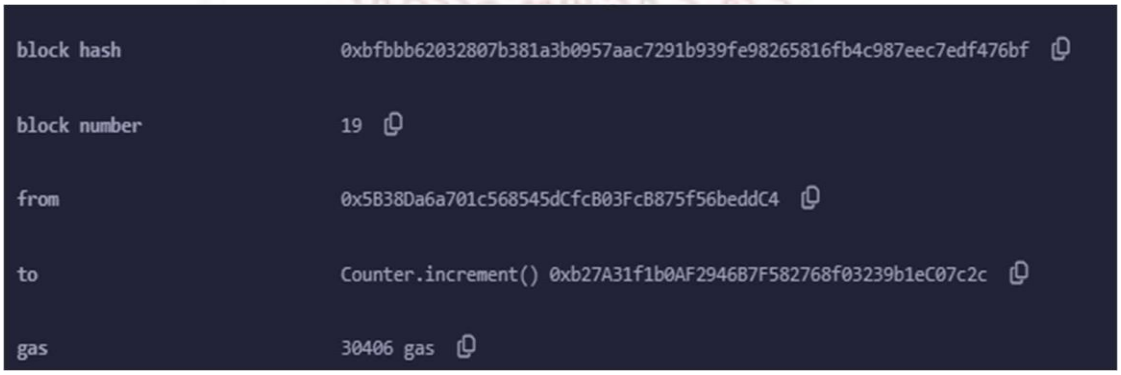
This process provides a basic understanding of **Solidity, contract compilation, and deployment using Remix IDE**.

OUTPUT:

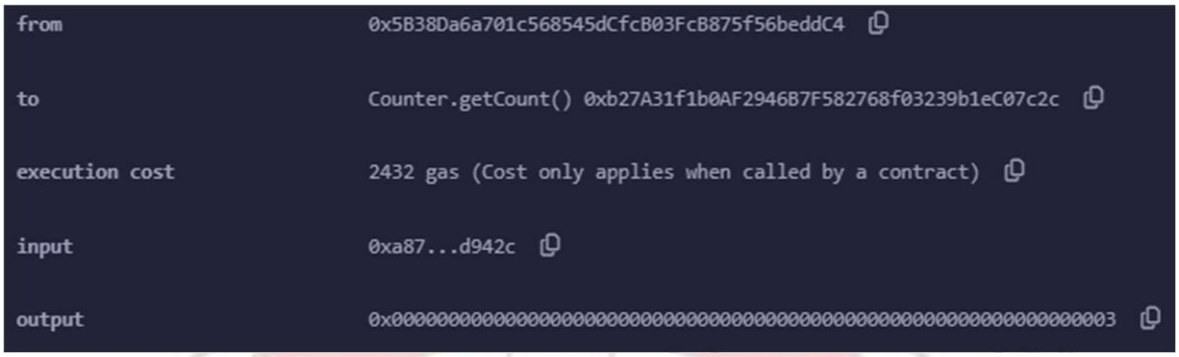
getCount()



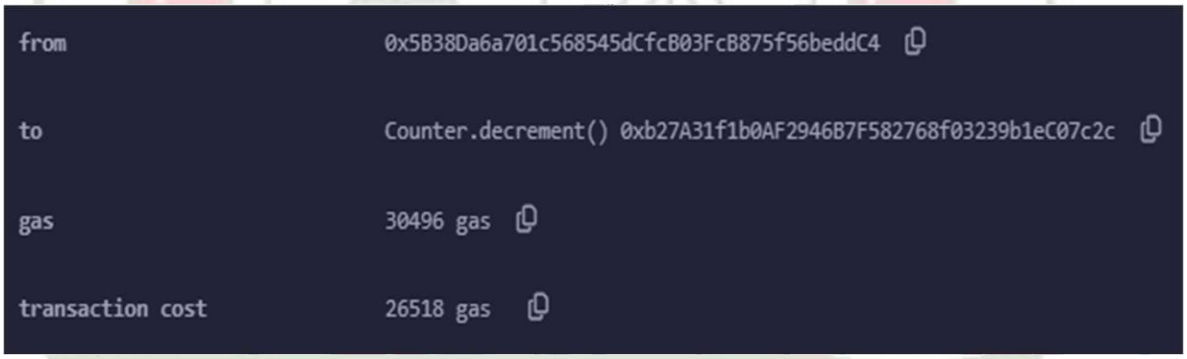
Counter.increment()



getCount()



Counter.increment()



OUTPUT ANALYSIS:

1. **Initial Counter Value:** After deployment, getCount() returns 0.
2. **Increment Operation:** Each call to increment() increases the counter by 1 (e.g., 0 → 1 → 2 → ...).
3. **Decrement Operation:** Each call to decrement() reduces the counter by 1, provided it is greater than 0 (e.g., 2 → 1 → 0).
4. **Underflow Prevention:** If the counter is 0, calling decrement() triggers an error: "Counter cannot be negative", ensuring it never becomes negative.

This confirms the smart contract's correct functionality, maintaining state integrity and preventing errors.