Steps to execute

To conduct any experiment download the code files and upload them onto Remix IDE.

Experiment 1

PoS

- Deploy the contract CarbonCreditTrading_PoS.sol
    - Inputs: initialCredits(Credits for the owner), minStake(Minimum stake for any validator to participate)
- To update the balance of the owner of the contract(To transfer the credits to a buyer at later stage) use *mintCarbonCredits()*
- To update the stake values of individual accounts use *updateBalance(address toAddr)*
    - Updates the balance with 20
- To participate in the validation use stakeCryptocurrency(address fromAddr, uint stake) public
    - Inputs: from Address and the stake value(should be greater than the minimum value of stake)
    - Updates the stake balance of an user, total stake in the contract, and the time taken for this transfer
- To buy carbon credits from the owner use the function *buyCarbonCredits(address payable toAddr, uint amount) public payable*
    - Inputs: to address(address of the user that should receive the credits), amount(amount of carbon credits to be obtained), msg.value(ethers to transfer)
    - Updates the carbon credits balance of buyer, owner, and the buy and sell transaction counts in the contract, and transfers ethers from buyer to owner
- To sell carbon credits to another user use the function *sellCarbonCredits(address fromAddr, address toAddr, uint amount) public payable*
    - Inputs: from address(address of the sender), to address (address of the receiver), amount(amount of carbon credits to be transferred), msg.value(ethers to transfer)
    - Updates the carbon credits balance of buyer, seller, and the buy and sell transaction counts in the contract, and transfers ethers from buyer to seller

PoW

- Deploy the contract PoW.sol
    - Inputs: The amount of carbon credits available and the price per credit
- To buy buy carbon credits use *buyCarbonCredits(uint amount)*
    - Inputs: Amount
    - Checks for the value greater than the required amount and the price per credit
- sellCarbonCredits(uint amount)
    - Inputs: The number of credits you want to sell
- submitProofOfWork(bytes32 hash, uint difficulty)
    - Input: Provide a hash and set a difficulty for the puzzle you want the miners to solve. The carbon credit trading will happen only after the puzzle is solved.

Experiment 2

- Deploy the contract *CarbonCreditTrading.sol*
    - Inputs: initialCredits(Credits for the owner), minStake(Minimum stake for any validator to participate)
    - Collect the address of the side chain contract from *addr* variable
- Deploy the contract *MainChain.sol*
    - Input the above collected address of side chain
- Order of execution is similar to above process of PoS from experiment 1, only difference is that the methods are triggered from the main contract *MainChain.sol*

Experiment 3 (Integer Overflow)

- Deploy the contract
- To add balance in the contract use *deposit* function
    - Provide an integer as an input
- To withdraw balance from the contract use *withdraw* function
    - Provide an integer as an input

Experiment 4

This includes the experiment with the implementation of a bridge, side chain and main chain.

- Deploy the contract CarbonCreditTrading_PoS.sol
    - Inputs: initialCredits(Credits for the owner), minStake(Minimum stake for any validator to participate)
- Deploy the contract *MainCCTrading.sol*
- Deploy the contract *BridgeCarbonCreditTrading.sol*
    - Inputs: Addresses of the side chain and the main chain
- Order of execution is the same as PoS from experiment 1 and the methods are triggered from the contract *BridgeCarbonCreditTrading.sol*
- The functionalities minting the carbon credits,balance updates, stake updates of participants, transfer of carbon credits, buy and sell of credits are executed in side chain and the currency transfer is handled in main chain

Similar implementation with an addition of private key splitting and distribution among various validators is attempted and can be found in the SharedKey folder of the experiments 4 folder. It included the multi party key signatures to initiate smart contracts, locking assets on one chain and unlocking assets on another chain.

For a shared private key, the inputs are: the number of validators, the parts required to split the private key and the encrypted private key. Deploy the smart contract with these values.
The lock asset function requires a lockId for the asset and the amount that you want to tag to that asset.

The unlock asset function requires a lockId for the asset, all parts of the private key, and the validators.
The buy and sell functionality is similar to the above functionalities.

Graphs

For the graphs code to execute, download the csv files into the local machine and provide the absolute file path in the pd1.read_csv(“<path>”) placeholder and execute.