DEX Assignment Report

Theory Questions and Analysis

Implementation Explanation

This section explains the components of the decentralized exchange (DEX) system implemented in this assignment. The system consists of four Solidity smart contracts and a JavaScript simulation script:

- **Token.sol**
  This contract is an ERC-20 compliant token used to deploy `TokenA` and `TokenB`. The constructor accepts the token name, symbol, and initial supply. Each token mints the full supply to the deployer account:

  ```
  constructor(string memory name, string memory symbol, uint256 initialSupply)
  ```

  These tokens are used for all liquidity and trading operations in the DEX.

- **LPToken.sol**
  This contract defines the LP tokens representing liquidity provider shares. Only the DEX contract (as the deployer) is allowed to mint or burn LP tokens:

  ```
  modifier onlyDEX() {
      require(msg.sender == dex, "Only DEX can call this function");
      ...
  }
  ```

  LP tokens are minted on `addLiquidity()` and burned on `removeLiquidity()`.

- **DEX.sol**
  This is the main DEX contract. It supports:

  - **Liquidity addition and removal:** Users provide both tokens in the correct ratio (or get refunded for excess). Initial LP tokens are minted using geometric mean; subsequent LP token minting is proportional to reserves.

    - **Swapping:** The `swap()` function implements the constant product AMM model $x \cdot y = k$ with a 0.3% fee. The output amount is calculated to preserve the invariant after fee deduction.
    - **Metrics tracking:** `getReserves()` and `getSpotPrice()` expose reserve and price info to external contracts and scripts.

- **Arbitrage.sol**
  This contract automates arbitrage between two DEX instances. It:
    - Compares spot prices from both DEXes.
    - Calculates potential profits for both directions: A→B→A and B→A→B.
    - Executes the more profitable path if the estimated gain exceeds the user-defined threshold (`minProfit`).
    - Uses constant product swap estimates with 0.3% fee and returns the final profit to the caller.

- **simulate_DEX.js (Remix JavaScript)**
  This script simulates trading activity on the DEX. It:
    - Deploys the tokens and DEX contract, distributes tokens to users.
    - Runs 50 randomized operations (add liquidity, remove liquidity, or swap).
    - Limits trades to a small fraction of pool reserves to minimize slippage.
    - Tracks and logs metrics like TVL, reserve ratios, spot prices, and swap events.

  The script demonstrates the dynamics of the AMM and the flow of tokens in the system.

- **simulate_arbitrage.js (Remix JavaScript)**
  This script simulates arbitrage opportunities between two DEX instances using the `Arbitrageur` contract. It:
    - Deploys two DEX contracts with different initial token reserves to create a price difference.
    - Invokes the `performArbitrage()` function with a chosen input amount.
    - Demonstrates both a profitable arbitrage (profit $\geq$ `minProfit`) and a failed one (profit $<$ `minProfit`).
    - Logs the calculated profit from both arbitrage directions (A→B→A and B→A→B).
    - Emits events such as `ProfitCalculated` and `ArbitrageExecuted` for transparency and debugging.

  This script validates that arbitrage logic is correctly implemented and functions only when a true opportunity exists.

Together, these components form a functioning DEX that supports trading, liquidity provision, LP token management, price-based arbitrage, and metrics tracking — all within a fully decentralized, permissionless, and transparent framework.

Security and Validations

Sanity checks and validation mechanisms have been implemented in the smart contracts to ensure safe behavior, prevent incorrect state transitions, and mitigate common vulnerabilities. The following measures were taken:

- **Zero-value input checks:**
  - `addLiquidity()`, `removeLiquidity()`, and `swap()` all reject zero-value inputs using `require(... > 0)` conditions.
  - Example:

    ```
    require(amountA > 0 && amountB > 0, "Amounts must be greater than zero");
    require(inputAmount > 0, "Input amount must be greater than zero");
    ```

- **Ratio validation during liquidity addition:**
  - To prevent imbalance in reserves, liquidity is only accepted if it maintains the existing reserve ratio:

    ```
    require((reserveA == 0 && reserveB == 0) ||
            (amountA * reserveB == amountB * reserveA), "Invalid ratio");
    ```

  - This avoids manipulation through incorrect liquidity injection.
- **Adjusted deposit calculation with refund:**
  - Liquidity addition uses an internal function `_calculateAdjustedDeposits()` to maintain reserve ratios.
  - Any excess token (if the ratio isn't exact) is refunded:

    ```
    if (adjustedA < amountA) {
        tokenA.transfer(msg.sender, amountA - adjustedA);
    }
    ```

  - This protects users from losing funds due to ratio mismatch.
- **Access control:**
  - The LP token can only be minted or burned by the DEX contract:

    ```
    modifier onlyDEX() {
        require(msg.sender == dex, "Only DEX can call this function");
    }
    ```

  - Prevents unauthorized manipulation of LP token supply.

- **Swap output validation:**
  - The `swap()` logic ensures reserves are correctly updated, and output token is transferred only after verifying inputs and computing output safely using the constant product formula.
- **Arbitrage profit threshold:**
  - The `Arbitrageur` contract only executes trades if expected profit exceeds a threshold:

    ```
    require(profitAtoBtoA >= minProfit || profitBtoAtoB >= minProfit,
            "No profitable opportunity");
    ```

  - This protects users from unintentional loss due to front-running or gas misestimation.

Theory Questions

1. Which address(es) should be allowed to mint/burn the LP tokens?

Only the DEX smart contract should have the authority to mint and burn LP tokens. This restriction ensures that:

- LP tokens are issued only when liquidity is added via `addLiquidity()`.
- LP tokens are burned only during `removeLiquidity()`.
- Malicious or unauthorized actors cannot inflate their ownership.

In our implementation: The LP token is instantiated inside the DEX contract, and the functions `lpToken.mint()` and `lpToken.burn()` are called exclusively from within DEX.

```
// LPToken.sol
modifier onlyDEX() {
    require(msg.sender == dex, "Only DEX can call this function");
}

function mint(address to, uint256 amount) external onlyDEX {
    _mint(to, amount);
}
```

2. In what way do DEXs level the playing field between powerful traders and retail investors?

Decentralized exchanges (DEXs) enforce rules via smart contracts which:

- Apply equal pricing through a public AMM ($x \cdot y = k$) formula.
- Allow permissionless participation in liquidity provision and trading.
- Ensure there are no intermediaries, brokers, or privileged roles.

In our implementation: All traders and LPs call the same DEX functions (`swap`, `addLiquidity`, `removeLiquidity`), and the `Arbitrageur` contract is open to any user.

```
// DEX.sol
function swap(address inputToken, address outputToken, uint256 inputAmount) external
function addLiquidity(uint256 amountA, uint256 amountB) external
function removeLiquidity(uint256 lpTokens) external
```

3. Suppose there are many transaction requests to the DEX sitting in a miner's mempool. How can the miner take undue advantage of this information? Is it possible to make the DEX robust against it?

Miners or bots can front-run transactions in the mempool by:

- Observing large swaps or arbitrage in the mempool.
- Inserting their own transaction with a higher gas fee.
- Exploiting the price shift before the original transaction is mined.

Mitigations:

- **Commit-reveal schemes**: to hide transaction details.
- **Slippage tolerance checks**: revert if minimum received amount isn't met.
- **Batch auctions**: to prevent order manipulation.

In our implementation: The arbitrage contract avoids executing if profit is below a threshold (`minProfit`), which indirectly mitigates front-running.

```
// Arbitrageur.sol
require(
    profitAtoBtoA >= minProfit || profitBtoAtoB >= minProfit,
    "No profitable opportunity"
);
```

4. How does gas fees influence economic viability of the DEX and arbitrage?

- Gas fees reduce net profits, especially in arbitrage where two swaps are required.
- If gas cost exceeds arbitrage profit, the trade results in loss.
- Frequent liquidity interactions become costly with high gas.

In our implementation: The arbitrage contract uses `minProfit` as a threshold. This ensures that only profitable arbitrage (net of gas assumptions) is executed.

```
// Arbitrageur.sol
constructor(..., uint256 _minProfit) {
    ...
    minProfit = _minProfit;
}
...
function performArbitrage(uint256 amountIn) external {
    ...
    require(profitAtoBtoA >= minProfit || profitBtoAtoB >= minProfit, ...);
}
```

5. Could gas fees lead to undue advantages to some transactors over others? How?

Yes. Ethereum's fee model allows users to:

- Set higher gas prices to prioritize their transaction in the block.
- Front-run profitable trades by beating lower gas fee transactions.

In our simulation: All users use default gas fees. In real-world deployments, bots can exploit this via MEV strategies unless mitigated by transaction ordering protections.

```
// simulate_DEX.js (Remix)
// All users perform transactions with default gas prices
await web3.eth.sendTransaction({ from, to, gas, data });
```

6. What are the various ways to minimize slippage in a swap?

- Trade small fractions of the reserve.
- Increase pool liquidity.
- Add slippage tolerance checks in `swap()` to avoid unfavorable trades.
- Use DEX aggregators to split large trades across pools.

In our code:

- Traders in the simulation trade at most 10% of reserves.
- Slippage protection logic is not implemented yet, but could be added using `minOut` parameters.

```
// simulate_DEX.js
const maxSwap = inputReserve.div(web3.utils.toBN(10));
```

7. How does slippage vary with the trade lot fraction?

Definitions:

- Trade lot fraction $= \frac{\text{Input amount}}{\text{Reserve}}$
- Expected price $= \frac{\text{Reserve}_Y}{\text{Reserve}_X}$
- Actual price $= \frac{\text{Output}}{\text{Input}}$
- Slippage (%) $= \left(\frac{\text{Expected}-\text{Actual}}{\text{Expected}}\right) \times 100$

Analysis: Slippage increases non-linearly with trade size due to the constant product formula. Small trades have negligible slippage, while large trades drastically impact price.

Implementation: In our simulation, we track reserves before and after each swap, which allows computing slippage and plotting it against the trade lot fraction.

```
// simulate_DEX.js (to be inserted)
const expected = reserveOut / reserveIn;
const actual = outputAmount / inputAmount;
const slippage = ((expected - actual) / expected) * 100;
metrics.slippage.push(slippage);
```