1.Write pseudocode to measure average latency and bandwidth using the simulator provided monitor output (as shown in Table 1.0). The pseudocode needs to be efficient and robust.

```
#entire code of soc design

import random
import math

# Define the objective function
def objective_function(x):
    latency = x[0]
    bandwidth = x[1]
    power = x[2]
    buffer_occupancy = x[3]
    arbitration_rate = x[4]
    # Assign weights to the factors based on their importance
    weight_latency = 0.3
    weight_bandwidth = 0.2
    weight_power = 0.2
    weight_buffer_occupancy = 0.2
    weight_arbitration_rate = 0.1
    return weight_latency * latency + weight_bandwidth * bandwidth + weight_power * power +
weight_buffer_occupancy * buffer_occupancy + weight_arbitration_rate * arbitration_rate

# Define the constraints
def constraint_latency(x):
    return max_latency - x[0]

def constraint_bandwidth(x):
    return max_bandwidth * 0.95 - x[1]

def constraint_buffer_occupancy(x):
    return max_buffer_size * 0.9 - x[3]

def constraint_arbitration_rate(x):
    return 0.05 - x[4]

# Define the simulated annealing algorithm
def simulated_annealing(x_init, temperature_init, cooling_rate, max_iterations):
    temperature = temperature_init
    x = x_init
    for i in range(max_iterations):
        # Generate a new candidate solution
        x_new = [random.uniform(x[0] - 0.1, x[0] + 0.1),
```

```
                random.uniform(x[1] - 0.1, x[1] + 0.1),
                random.uniform(x[2] - 0.1, x[2] + 0.1),
                random.uniform(x[3] - 0.1, x[3] + 0.1),
                random.uniform(x[4] - 0.1, x[4] + 0.1)]
        # Evaluate the objective function and constraints for the current and new solutions
        f_x = objective_function(x)
        f_x_new = objective_function(x_new)
        c_latency = constraint_latency(x)
        c_bandwidth = constraint_bandwidth(x)
        c_buffer_occupancy = constraint_buffer_occupancy(x)
        c_arbitration_rate = constraint_arbitration_rate(x)
        c_latency_new = constraint_latency(x_new)
        c_bandwidth_new = constraint_bandwidth(x_new)
        c_buffer_occupancy_new = constraint_buffer_occupancy(x_new)
        c_arbitration_rate_new = constraint_arbitration_rate(x_new)
        # Determine the acceptance probability
        delta_f = f_x_new - f_x
        if delta_f < 0:
            acceptance_probability = 1
        else:
            acceptance_probability = math.exp(-delta_f / temperature)
        # Accept or reject the new solution based on the acceptance probability and constraints
        if acceptance_probability > random.uniform(0, 1) and c_latency_new >= 0 and
c_bandwidth_new >= 0 and c_buffer_occupancy_new >= 0 and c_arbitration_rate_new >= 0:
            x = x_new
        # Update the temperature
        temperature *= cooling_rate
    return x

# Set the parameters and initial values
max_latency = 10
max_bandwidth = 100
max_power = 10
max_buffer_size = 100
x_init = [max_latency, max_bandwidth, max_power, max_buffer_size, 0.05]
temperature_init = 100
cooling_rate = 0.9
max_iterations = 1000

# Call the simulated annealing algorithm
x_optimal = simulated_annealing(x_init, temperature_init, cooling_rate, max_iterations)
print("Optimal solution: ", x_optimal)
print("Optimal objective function value: ", objective_function(x_optimal))
```

latency and bandwidth :

```
# initialize variables
last_txn_time = 0
total_latency = 0
total_bytes = 0

# iterate through each transaction in the monitor output
for txn in monitor_output:
    current_time = txn.timestamp
    txn_type = txn.txn_type
    data = txn.data

    # calculate latency for read transactions
    if txn_type == "Rd":
        latency = current_time - last_txn_time
        total_latency += latency

    # calculate bandwidth for write transactions
    if txn_type == "Wr":
        total_bytes += 32 # assuming 32 bytes per transaction

    # update last transaction time
    last_txn_time = current_time

# calculate average latency and bandwidth
avg_latency = total_latency / num_read_txns
avg_bandwidth = total_bytes / (num_write_txns * cycle_time)
```

Explanation :

* We initialize variables to keep track of the last transaction time, total latency, and total bytes transferred.
* We iterate through each transaction in the monitor output.
* For read transactions, we calculate the latency by subtracting the timestamp of the last transaction from the current transaction's timestamp.
* For write transactions, we add the number of bytes transferred to the total bytes transferred.
* We update the last transaction time for the next iteration.
* After iterating through all transactions, we calculate the average latency by dividing the total latency by the number of read transactions.
* We calculate the average bandwidth by dividing the total bytes transferred by the number of write transactions multiplied by the cycle time.

2.Reinforced Learning Design Document:

States/Behaviors:

* Current buffer occupancy for each buffer
* Current arbitration rates for each agent type
* Current power limit threshold (1 or 0)

Actions:

* Set maximum buffer size for a buffer
* Set arbitration weights for an agent type
* Throttle the operating frequency

Rewards:

* Negative reward for exceeding the latency threshold
* Negative reward for falling below the bandwidth threshold
* Negative reward for exceeding the buffer occupancy threshold
* Negative reward for exceeding the power limit threshold
* Positive reward for staying within the desired thresholds

RL Algorithm:

* Actor-Critic algorithm is best suited for this problem statement because it can handle continuous state and action spaces, and it can balance exploration and exploitation.

Explanation:

* The states/behaviors include the current buffer occupancy, arbitration rates, and power limit threshold, which are continuous variables.
* The actions include setting the maximum buffer size, arbitration weights, and throttling the operating frequency, which are also continuous variables.
* The rewards are negative for exceeding the desired thresholds and positive for staying within the thresholds.
* The Actor-Critic algorithm is well-suited for this problem because it can handle continuous state and action spaces, and it can balance exploration and exploitation.
* The algorithm can learn the optimal parameters by adjusting the weights of the actor and critic networks based on the rewards received for each action taken.
* The algorithm can also handle the trade-off between latency, bandwidth, buffer occupancy, and power consumption by adjusting the weights of the rewards accordingly.