Practical 1

Aim: SDES Implementation

Description:

This practical involves implementing the Simplified Data Encryption Standard (SDES) algorithm in Java. SDES is a symmetric-key block cipher that operates on 8-bit blocks and uses a 10-bit key. It involves multiple steps, including key generation, initial permutation, expansion, S-box substitutions, and permutation. The process includes two rounds of encryption to transform the plaintext into ciphertext. The key used for the encryption is derived from an initial 10-bit key through permutations and shifts.

The steps involved in the SDES algorithm include:

1. Key permutation and generation (K1, K2).
2. Initial permutation (IP) and final permutation (IP-1).
3. Function fK, which includes expansion, permutation, and substitution through S-boxes.
4. Two rounds of encryption, involving function fK and a switch operation.

Code:

```
import java.util.BitSet;


public class SDESEncryption {


   // Helper function to print BitSet as binary string
   public static void printBinary(BitSet bits) {
      StringBuilder sb = new StringBuilder();
      for (int i = 0; i < 8; i++) {
         sb.append(bits.get(i) ? "1" : "0");
      }
      System.out.println(sb.toString());
   }


   // 1. P10 Permutation for the 10-bit key
   // Reorders the bits of the key as per P10 permutation table
```

```
public static BitSet P10(BitSet key) {
    BitSet permutedKey = new BitSet(10);
    permutedKey.set(0, key.get(2));
    permutedKey.set(1, key.get(4));
    permutedKey.set(2, key.get(1));
    permutedKey.set(3, key.get(6));
    permutedKey.set(4, key.get(3));
    permutedKey.set(5, key.get(9));
    permutedKey.set(6, key.get(0));
    permutedKey.set(7, key.get(8));
    permutedKey.set(8, key.get(7));
    permutedKey.set(9, key.get(5));
    return permutedKey;
}

// 2. Circular Left Shift (LS-1) for 5 bits
// This function shifts the 5-bit half of the key left by one position
public static BitSet leftShift(BitSet halfKey) {
    BitSet shifted = new BitSet(5);
    shifted.set(0, halfKey.get(1));
    shifted.set(1, halfKey.get(2));
    shifted.set(2, halfKey.get(3));
    shifted.set(3, halfKey.get(4));
    shifted.set(4, halfKey.get(0));
    return shifted;
}

// 3. P8 Permutation for key subkey generation
// Takes a 10-bit key and reduces it to an 8-bit subkey using the P8 permutation
public static BitSet P8(BitSet key) {
    BitSet subKey = new BitSet(8);
    subKey.set(0, key.get(5));
```

```
    subKey.set(1, key.get(2));
    subKey.set(2, key.get(6));
    subKey.set(3, key.get(3));
    subKey.set(4, key.get(7));
    subKey.set(5, key.get(4));
    subKey.set(6, key.get(9));
    subKey.set(7, key.get(8));
    return subKey;
}

// 4. Key Generation: K1 and K2
// Generates two 8-bit subkeys K1 and K2 using the key permutation (P10, left shift, P8)
public static BitSet[] generateKeys(BitSet key) {
    // Apply P10 permutation to the key
    BitSet permutedKey = P10(key);

    // Split the permuted key into two 5-bit halves
    BitSet left = new BitSet(5);
    BitSet right = new BitSet(5);
    for (int i = 0; i < 5; i++) {
        left.set(i, permutedKey.get(i));
        right.set(i, permutedKey.get(i + 5));
    }

    // Apply the left shift and generate K1
    left = leftShift(left);
    right = leftShift(right);
    BitSet combinedKey1 = new BitSet(10);
    for (int i = 0; i < 5; i++) {
        combinedKey1.set(i, left.get(i));
        combinedKey1.set(i + 5, right.get(i));
    }
```

```
    BitSet K1 = P8(combinedKey1);

    // Apply another left shift and generate K2
    left = leftShift(left);
    right = leftShift(right);
    BitSet combinedKey2 = new BitSet(10);
    for (int i = 0; i < 5; i++) {
        combinedKey2.set(i, left.get(i));
        combinedKey2.set(i + 5, right.get(i));
    }
    BitSet K2 = P8(combinedKey2);

    return new BitSet[] {K1, K2};
}

// 5. Initial Permutation (IP) of the plaintext
public static BitSet IP(BitSet data) {
    BitSet permutedData = new BitSet(8);
    permutedData.set(0, data.get(1));
    permutedData.set(1, data.get(5));
    permutedData.set(2, data.get(2));
    permutedData.set(3, data.get(0));
    permutedData.set(4, data.get(3));
    permutedData.set(5, data.get(7));
    permutedData.set(6, data.get(4));
    permutedData.set(7, data.get(6));
    return permutedData;
}

// 6. Inverse Initial Permutation (IP-1)
public static BitSet IP1(BitSet data) {
    BitSet permutedData = new BitSet(8);
```

```
    permutedData.set(0, data.get(3));
    permutedData.set(1, data.get(0));
    permutedData.set(2, data.get(2));
    permutedData.set(3, data.get(4));
    permutedData.set(4, data.get(6));
    permutedData.set(5, data.get(1));
    permutedData.set(6, data.get(7));
    permutedData.set(7, data.get(5));
    return permutedData;
}

// 7. Expansion and permutation (E/P) for fK function
public static BitSet EP(BitSet halfBlock) {
    BitSet expanded = new BitSet(8);
    expanded.set(0, halfBlock.get(3));
    expanded.set(1, halfBlock.get(0));
    expanded.set(2, halfBlock.get(1));
    expanded.set(3, halfBlock.get(2));
    expanded.set(4, halfBlock.get(1));
    expanded.set(5, halfBlock.get(2));
    expanded.set(6, halfBlock.get(3));
    expanded.set(7, halfBlock.get(0));
    return expanded;
}

// S-Boxes (S0 and S1) for substitution
public static BitSet S0(BitSet input) {
    int row = (input.get(0) ? 1 : 0) << 1 | (input.get(3) ? 1 : 0);
    int col = (input.get(1) ? 1 : 0) << 1 | (input.get(2) ? 1 : 0);
    int[][] S0 = {{1, 0, 3, 2}, {3, 2, 1, 0}, {0, 2, 1, 3}, {3, 1, 3, 2}};
    int result = S0[row][col];
    BitSet output = new BitSet(2);
```

```
    output.set(0, (result & 0b10) != 0);
    output.set(1, (result & 0b01) != 0);
    return output;
}

public static BitSet S1(BitSet input) {
    int row = (input.get(0) ? 1 : 0) << 1 | (input.get(3) ? 1 : 0);
    int col = (input.get(1) ? 1 : 0) << 1 | (input.get(2) ? 1 : 0);
    int[][] S1 = {{0, 1, 2, 3}, {2, 0, 1, 3}, {3, 0, 1, 0}, {2, 1, 0, 3}};
    int result = S1[row][col];
    BitSet output = new BitSet(2);
    output.set(0, (result & 0b10) != 0);
    output.set(1, (result & 0b01) != 0);
    return output;
}

// 8. Function fK: applies S-Boxes, permutation, and XOR operations to transform data
public static BitSet fK(BitSet data, BitSet subkey) {
    // Split data into left (L) and right (R) 4-bit halves
    BitSet L = new BitSet(4);
    BitSet R = new BitSet(4);
    for (int i = 0; i < 4; i++) {
        L.set(i, data.get(i));
        R.set(i, data.get(i + 4));
    }

    // Expansion and permutation (E/P) on right half (R)
    BitSet expandedR = EP(R);

    // XOR expanded R with subkey
    expandedR.xor(subkey);
```

```
// Split into two parts for S-boxes
BitSet left = new BitSet(4);
BitSet right = new BitSet(4);
for (int i = 0; i < 4; i++) {
    left.set(i, expandedR.get(i));
    right.set(i, expandedR.get(i + 4));
}

// Apply S-boxes to both halves
BitSet S0Out = S0(left);
BitSet S1Out = S1(right);

// Combine the outputs of S-boxes and apply P4 permutation
BitSet combined = new BitSet(4);
combined.set(0, S0Out.get(0));
combined.set(1, S1Out.get(0));
combined.set(2, S0Out.get(1));
combined.set(3, S1Out.get(1));

// Apply P4 permutation
BitSet P4Out = new BitSet(4);
P4Out.set(0, combined.get(1));
P4Out.set(1, combined.get(3));
P4Out.set(2, combined.get(0));
P4Out.set(3, combined.get(2));

// XOR with left half (L)
BitSet newL = (BitSet) L.clone();
newL.xor(P4Out);

// Combine new L with right half (R) to get the result
BitSet result = new BitSet(8);
```

```
        for (int i = 0; i < 4; i++) {
            result.set(i, newL.get(i));
            result.set(i + 4, R.get(i));
        }
        return result;
    }

    // 9. Switch function (SW): swaps the left and right halves of the data
    public static BitSet SW(BitSet data) {
        BitSet L = new BitSet(4);
        BitSet R = new BitSet(4);
        for (int i = 0; i < 4; i++) {
            L.set(i, data.get(i));
            R.set(i, data.get(i + 4));
        }
        BitSet result = new BitSet(8);
        for (int i = 0; i < 4; i++) {
            result.set(i, R.get(i));
            result.set(i + 4, L.get(i));
        }
        return result;
    }

    // 10. S-DES Encryption: applies initial permutation, fK rounds, switching, and final permutation
    public static BitSet encrypt(BitSet plaintext, BitSet key) {
        BitSet[] keys = generateKeys(key);
        BitSet K1 = keys[0];
        BitSet K2 = keys[1];

        // Apply Initial Permutation (IP) to plaintext
        BitSet permutedData = IP(plaintext);
```

```
    // First round of fK and switch
    BitSet fK1Result = fK(permutedData, K1);
    BitSet switchedData = SW(fK1Result);

    // Second round of fK and final result
    BitSet fK2Result = fK(switchedData, K2);
    return IP1(fK2Result);
}

// 11. S-DES Decryption: the inverse of the encryption process
public static BitSet decrypt(BitSet ciphertext, BitSet key) {
    BitSet[] keys = generateKeys(key);
    BitSet K1 = keys[0];
    BitSet K2 = keys[1];

    // Apply Initial Permutation (IP) to ciphertext
    BitSet permutedData = IP(ciphertext);

    // First round of fK using K2 and switch
    BitSet fK2Result = fK(permutedData, K2);
    BitSet switchedData = SW(fK2Result);

    // Second round of fK using K1 and final result
    BitSet fK1Result = fK(switchedData, K1);
    return IP1(fK1Result);
}

public static void main(String[] args) {
    // Example 10-bit key and 8-bit plaintext
    BitSet key = new BitSet(10);
    for (int i = 0; i < 10; i++) {
```

```
        key.set(i, "1010000010".charAt(i) == '1');
    }

    BitSet plaintext = new BitSet(8);
    for (int i = 0; i < 8; i++) {
        plaintext.set(i, "10111101".charAt(i) == '1');
    }

    // Print original plaintext
    System.out.print("Original plaintext: ");
    printBinary(plaintext);

    // Encryption
    BitSet ciphertext = encrypt(plaintext, key);
    System.out.print("Ciphertext: ");
    printBinary(ciphertext);

    // Decryption
    BitSet decryptedText = decrypt(ciphertext, key);
    System.out.print("Decrypted plaintext: ");
    printBinary(decryptedText);
  }
}
```

Output:

```
C:\Users\jimit\Documents>javac SDESEncryption.java && java SDESEncryption
Original plaintext: 10111101
Ciphertext: 01010001
Decrypted plaintext: 10111101
```

Code explanation:

1. **Key Generation**:

- **P10**: The key is first permuted using a P10 permutation, which rearranges the 10 bits based on a fixed pattern.
- The permuted key is split into two halves, and a circular left shift (LS-1) is applied to both halves.
- **P8**: A final 8-bit subkey is generated from the permuted key after shifting.
- This process is repeated to generate two subkeys, K1 and K2, used in two rounds of encryption.

2. **Initial Permutation (IP)**:

- The 8-bit plaintext is rearranged using a fixed permutation pattern defined by IP. This is the first step of the encryption process.

3. **fK Function**:

- The fK function is the core transformation applied during encryption. It includes:
    - Expansion and permutation (E/P) of the right half of the data.
    - XORing the expanded data with a subkey.
    - Substitution using two 2x4 S-boxes (S0 and S1).
    - A final permutation (P4) of the combined S-box output.
    - XORing the result with the left half of the data.
- The left half (L) and right half (R) of the data are swapped in each round using the SW function.

4. **Rounds of Encryption**:

- The data undergoes two rounds of transformation using the subkeys K1 and K2. The rounds are as follows:
    1. First round: The IP-permuted data undergoes the first application of the fK function, and the halves are switched.
    2. Second round: The switched data undergoes the second application of the fK function, and the result is permuted again using the IP-1 function to generate the final ciphertext.

5. **Inverse Permutation (IP-1)**:

- After the two rounds, an inverse permutation (IP-1) is applied to the data to produce the final ciphertext. The IP-1 rearranges the bits back to their original positions in a fixed pattern.

Complexity analysis:

- **Time Complexity**:

- **Key Generation**: The key generation process involves the following operations:
  - P10 permutation: This takes constant time, O(1).
  - Left shifts: These involve constant-time bit operations, O(1).
  - P8 permutation: Also takes constant time, O(1).
  - Overall, key generation runs in constant time, O(1).
- **Encryption Process**:
  - The IP and IP-1 permutations each take constant time, O(1).
  - In each round of encryption, the operations (expansion, S-box substitutions, and permutations) are all constant-time operations, O(1).
  - Thus, each round takes constant time, O(1), and since there are two rounds, the overall encryption process also takes constant time, O(1).
- **Overall Time Complexity**: The overall time complexity of SDES encryption is **O(1)**, since all the operations involved (permutations, substitutions, and shifts) are constant-time operations, independent of the size of the input data.

- **Space Complexity**:

- The space complexity is determined by the storage required for key generation, input data, intermediate values, and the output data.
  - The 10-bit key requires constant space, O(1).
  - The 8-bit plaintext and the resulting ciphertext require constant space, O(1).
  - Intermediate values like subkeys (K1, K2), permuted data, expanded data, and S-box outputs all require constant space, O(1).
- **Overall Space Complexity**: The overall space complexity is **O(1)**, as the space used does not depend on the size of the input, and the amount of space required is constant.