TA39 Tanmay Mane

S-DES implementation code :

```
# Initial permutation
IP = [2, 6, 3, 1, 4, 8, 5, 7]

# Expansion permutation
EP = [4, 1, 2, 3, 2, 3, 4, 1]

# Permutation P4
P4 = [2, 4, 3, 1]

# Permutation P8
P8 = [6, 3, 7, 4, 8, 5, 10, 9]

# Permutation P10
P10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]

# Inverse permutation
IP_inv = [4, 1, 3, 5, 7, 2, 8, 6]

# S-Boxes
S0 = [
    [1, 0, 3, 2],
    [3, 2, 1, 0],
    [0, 2, 1, 3],
    [3, 1, 3, 2]
]

S1 = [
    [0, 1, 2, 3],
    [2, 0, 1, 3],
    [3, 0, 1, 0],
    [2, 1, 0, 3]
]

# Circular left shift
def left_shift(key, n):
    return key[n:] + key[:n]

# Initial permutation (IP)
def initial_permutation(plaintext):
    return [plaintext[i - 1] for i in IP]

# Expansion permutation (EP)
def expansion_permutation(R):
    return [R[i - 1] for i in EP]
```

```
# Permutation (P4)
def permutation_P4(input):
    return [input[i - 1] for i in P4]


# Permutation (P8)
def permutation_P8(input):
    return [input[i - 1] for i in P8]


# Permutation (P10)
def permutation_P10(input):
    return [input[i - 1] for i in P10]


# Inverse permutation (IP_inv)
def inverse_permutation(input):
    return [input[i - 1] for i in IP_inv]


# XOR operation
def xor(bits1, bits2):
    return [b1 ^ b2 for b1, b2 in zip(bits1, bits2)]


# S-Box substitution
def sbox_substitution(bits, sbox):
    row = bits[0] * 2 + bits[3]
    col = bits[1] * 2 + bits[2]
    return [int(b) for b in format(sbox[row][col], '02b')]

# F function
def f_function(R, K):
    expanded_R = expansion_permutation(R)
    xor_result = xor(expanded_R, K)
        # Split into two parts
    left_half = xor_result[:4]
    right_half = xor_result[4:]
        # S-Box substitution
    sbox0_output = sbox_substitution(left_half, S0)
    sbox1_output = sbox_substitution(right_half, S1)
        # Permutation (P4)
        p4_input = sbox0_output + sbox1_output
    return permutation_P4(p4_input)

# Generate subkeys
def generate_subkeys(key):
        # Permutation (P10)
    key = permutation_P10(key)
        # Split into two parts
    left_half = key[:5]
    right_half = key[5:]
        # Circular left shifts
    left_half_shifted = left_shift(left_half, 1)
    right_half_shifted = left_shift(right_half, 1)
        # Concatenate and permute (P8)
    round_key1 = permutation_P8(left_half_shifted + right_half_shifted)
        # Another shift
    left_half_shifted = left_shift(left_half_shifted, 2)
    right_half_shifted = left_shift(right_half_shifted, 2)
        # Concatenate and permute (P8)
    round_key2 = permutation_P8(left_half_shifted + right_half_shifted)
    return round_key1, round_key2

# Encrypt a plaintext using S-DES
def encrypt(plaintext, key):
    round_key1, round_key2 = generate_subkeys(key)
        # Initial permutation
    plaintext = initial_permutation(plaintext)
        # Initial permutation
        L = plaintext[:4]
        R = plaintext[4:]
        # Round 1
    f_result = f_function(R, round_key1)
    new_R = xor(L, f_result)
        # Round 2
        L = R
        R = new_R
    f_result = f_function(R, round_key2)
    new_R = xor(L, f_result)
        # Inverse permutation
    ciphertext = inverse_permutation(new_R + R)
    return ciphertext


# Decrypt a ciphertext using S-DES
def decrypt(ciphertext, key):
    round_key1, round_key2 = generate_subkeys(key)
        # Initial permutation
    ciphertext = initial_permutation(ciphertext)
        # Initial permutation
        L = ciphertext[:4]
        R = ciphertext[4:]
        # Round 1
    f_result = f_function(R, round_key2)
    new_R = xor(L, f_result)
        # Round 2
        L = R
        R = new_R
    f_result = f_function(R, round_key1)
    new_R = xor(L, f_result)
        # Inverse permutation
    plaintext = inverse_permutation(new_R + R)
    return plaintext

# Function to get binary input
def get_binary_input(msg):
    binary = input(msg)
    binary = binary.replace(" ", "")
    binary = [int(b) for b in binary]
    return binary

# Main function
def main():
            # Get input from user
    plaintext = get_binary_input("Enter 8-bit plaintext (e.g., 10101010): ")
    key = get_binary_input("Enter 10-bit key (e.g., 1010101010): ")

            # Check input lengths
    if len(plaintext) != 8 or len(key) != 10:
        print("Invalid input lengths!")
        return

            # Encrypt
    ciphertext = encrypt(plaintext, key)
    print("Encrypted ciphertext:", ''.join(map(str, ciphertext)))
```

```
        # Decrypt
    decrypted = decrypt(ciphertext, key)
    print("Decrypted plaintext:", ''.join(map(str, decrypted)))


if __name__ == "__main__":
    main()
```

—---

Output:

```
Enter 8-bit plaintext (e.g., 10101010): 01011111

Enter 10-bit key (e.g., 1010101010): 0000011111

Encrypted ciphertext: 00001111

Decrypted plaintext: 01011111
```