**Assignment no 5: Implementation of DES**

**2020BTECS00085**

**MANJIRI CHANDURE**

**BATCH : B5**

# Introduction

DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits.

DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition.

* In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
* The initial permutation is performed on plain text.
* Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
* Now each LPT and RPT go through 16 rounds of the encryption process.
* In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
* The result of this process produces 64-bit ciphertext.

# Implementation Initial Permutation:

The Initial Permutation (IP) is a one-time operation that occurs before the first round of DES encryption.

IP involves rearranging the bits of the original plaintext block according to a predefined rule.

Each bit in the rearranged block is determined by swapping it with a specific bit from the original plaintext block, as specified by the IP permutation table.

IP is essentially a bit-position juggling operation that establishes the initial data arrangement for subsequent DES rounds.

# Step-1: Key transformation:

We have noted initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key. Thus, for each a 56-bit key is available. From this 56-bit key, a different 48-bit Sub Key is generated during each round using a process called key transformation. For this, the

56-bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round.

# Step-2: Expansion Permutation:

Recall that after the initial permutation, we had two 32-bit plain text areas called Left Plain Text(LPT) and Right Plain Text(RPT). During the expansion permutation, the RPT is expanded from 32 bits to 48 bits. Bits are permuted as well hence called expansion permutation. This happens as the 32-bit RPT is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4-bit block of the previous step is then expanded to a corresponding 6-bit block, i.e., per 4-bit block, 2 more bits are added.

This process results in expansion as well as a permutation of the input bit while creating output. The key transformation process compresses the 56-bit key to 48 bits. Then the expansion permutation process expands the 32-bit RPT to 48-bits. Now the 48-bit key is XOR with 48-bit RPT and the resulting output is given to the next step, which is the S-Box substitution.

Now apply the XOR function to the output and the left plain text and store it in the right plain text. Store the initial right plain text in the left plain text. Both the LPT and RPT halves are forwarded to the next rounds for further operations. At the end of the last round, swap the data in the LPT and RPT. In the last step, apply the inverse permutation step to get the cipher text.

**Code in Cpp:**

// Including dependancies

#include <bits/stdc++.h>

using namespace std;

// Array to hold 16 keys

string round\_keys[16];

// String to hold the plain text

string pt;

// Function to convert a number in decimal to binary

string textToBinary(const string &text)

{

    string binary = "";

    for (char c : text)

    {

        binary += bitset<8>(c).to\_string();//

    }

    return binary;

}

string binaryToText(const string &binary)

{

    string text = "";

    for (size\_t i = 0; i < binary.length(); i += 8)

    {

        string byte = binary.substr(i, 8);

        char c = static\_cast<char>(bitset<8>(byte).to\_ulong());

        text += c;

    }

    return text;

}

string convertDecimalToBinary(int decimal)

{

    string binary;

    while (decimal != 0)

    {

        binary = (decimal % 2 == 0 ? "0" : "1") + binary;

        decimal = decimal / 2;

    }

    while (binary.length() < 4)

    {

        binary = "0" + binary;

    }

    return binary;

}

// Function to convert a number in binary to decimal

int convertBinaryToDecimal(string binary)

{

    int decimal = 0;

    int counter = 0;

    int size = binary.length();

    for (int i = size - 1; i >= 0; i--)

    {

        if (binary[i] == '1')

        {

            decimal += pow(2, counter);

        }

        counter++;

    }

    return decimal;

}

// Function to do a circular left shift by 1

string shift\_left\_once(string key\_chunk)

{

    string shifted = "";

    for (int i = 1; i < 28; i++)

    {

        shifted += key\_chunk[i];

    }

    shifted += key\_chunk[0];

    return shifted;

}

// Function to do a circular left shift by 2

string shift\_left\_twice(string key\_chunk)

{

    string shifted = "";

    for (int i = 0; i < 2; i++)

    {

        for (int j = 1; j < 28; j++)

        {

            shifted += key\_chunk[j];

        }

        shifted += key\_chunk[0];

        key\_chunk = shifted;

        shifted = "";

    }

    return key\_chunk;

}

// Function to compute xor between two strings

string Xor(string a, string b)

{

    string result = "";

    int size = b.size();

    for (int i = 0; i < size; i++)

    {

        if (a[i] != b[i])

        {

            result += "1";

        }

        else

        {

            result += "0";

        }

    }

    return result;

}

// Function to generate the 16 keys.

void generate\_keys(string key)

{

    // The PC1 table

    int pc1[56] = {

        57, 49, 41, 33, 25, 17, 9,

        1, 58, 50, 42, 34, 26, 18,

        10, 2, 59, 51, 43, 35, 27,

        19, 11, 3, 60, 52, 44, 36,

        63, 55, 47, 39, 31, 23, 15,

        7, 62, 54, 46, 38, 30, 22,

        14, 6, 61, 53, 45, 37, 29,

        21, 13, 5, 28, 20, 12, 4};

    // The PC2 table

    int pc2[48] = {

        14, 17, 11, 24, 1, 5,

        3, 28, 15, 6, 21, 10,

        23, 19, 12, 4, 26, 8,

        16, 7, 27, 20, 13, 2,

        41, 52, 31, 37, 47, 55,

        30, 40, 51, 45, 33, 48,

        44, 49, 39, 56, 34, 53,

        46, 42, 50, 36, 29, 32};

    // 1. Compressing the key using the PC1 table

    string perm\_key = "";

    for (int i = 0; i < 56; i++)

    {

        perm\_key += key[pc1[i] - 1];

    }

    // 2. Dividing the key into two equal halves

    string left = perm\_key.substr(0, 28);

    string right = perm\_key.substr(28, 28);

    for (int i = 0; i < 16; i++)

    {

        // 3.1. For rounds 1, 2, 9, 16 the key\_chunks

        // are shifted by one.

        if (i == 0 || i == 1 || i == 8 || i == 15)

        {

            left = shift\_left\_once(left);

            right = shift\_left\_once(right);

        }

        // 3.2. For other rounds, the key\_chunks

        // are shifted by two

        else

        {

            left = shift\_left\_twice(left);

            right = shift\_left\_twice(right);

        }

        // Combining the two chunks

        string combined\_key = left + right;

        string round\_key = "";

        // Finally, using the PC2 table to transpose the key bits

        for (int i = 0; i < 48; i++)

        {

            round\_key += combined\_key[pc2[i] - 1];

        }

        round\_keys[i] = round\_key;

    }

}

// Implementing the algorithm

string DES()

{

    // The initial permutation table

    int initial\_permutation[64] = {

        58, 50, 42, 34, 26, 18, 10, 2,

        60, 52, 44, 36, 28, 20, 12, 4,

        62, 54, 46, 38, 30, 22, 14, 6,

        64, 56, 48, 40, 32, 24, 16, 8,

        57, 49, 41, 33, 25, 17, 9, 1,

        59, 51, 43, 35, 27, 19, 11, 3,

        61, 53, 45, 37, 29, 21, 13, 5,

        63, 55, 47, 39, 31, 23, 15, 7};

    // The expansion table

    int expansion\_table[48] = {

        32, 1, 2, 3, 4, 5, 4, 5,

        6, 7, 8, 9, 8, 9, 10, 11,

        12, 13, 12, 13, 14, 15, 16, 17,

        16, 17, 18, 19, 20, 21, 20, 21,

        22, 23, 24, 25, 24, 25, 26, 27,

        28, 29, 28, 29, 30, 31, 32, 1};

    // The substitution boxes. The should contain values

    // from 0 to 15 in any order.

    int substition\_boxes[8][4][16] =

        {{14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,

          0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,

          4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,

          15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13},

         {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,

          3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,

          0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,

          13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9},

         {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,

          13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,

          13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,

          1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12},

         {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,

          13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,

          10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,

          3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14},

         {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,

          14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,

          4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,

          11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3},

         {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,

          10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,

          9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,

          4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13},

         {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,

          13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,

          1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,

          6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12},

         {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,

          1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,

          7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,

          2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}};

    // The permutation table

    int permutation\_tab[32] = {

        16, 7, 20, 21, 29, 12, 28, 17,

        1, 15, 23, 26, 5, 18, 31, 10,

        2, 8, 24, 14, 32, 27, 3, 9,

        19, 13, 30, 6, 22, 11, 4, 25};

    // The inverse permutation table

    int inverse\_permutation[64] = {

        40, 8, 48, 16, 56, 24, 64, 32,

        39, 7, 47, 15, 55, 23, 63, 31,

        38, 6, 46, 14, 54, 22, 62, 30,

        37, 5, 45, 13, 53, 21, 61, 29,

        36, 4, 44, 12, 52, 20, 60, 28,

        35, 3, 43, 11, 51, 19, 59, 27,

        34, 2, 42, 10, 50, 18, 58, 26,

        33, 1, 41, 9, 49, 17, 57, 25};

    // 1. Applying the initial permutation

    string perm = "";

    for (int i = 0; i < 64; i++)

    {

        perm += pt[initial\_permutation[i] - 1];

    }

    // 2. Dividing the result into two equal halves

    string left = perm.substr(0, 32);

    string right = perm.substr(32, 32);

    // The plain text is encrypted 16 times

    for (int i = 0; i < 16; i++)

    {

        string right\_expanded = "";

        // 3.1. The right half of the plain text is expanded

        for (int i = 0; i < 48; i++)

        {

            right\_expanded += right[expansion\_table[i] - 1];

        }; // 3.3. The result is xored with a key

        string xored = Xor(round\_keys[i], right\_expanded);

        string res = "";

        // 3.4. The result is divided into 8 equal parts and passed

        // through 8 substitution boxes. After passing through a

        // substituion box, each box is reduces from 6 to 4 bits.

        for (int i = 0; i < 8; i++)

        {

            // Finding row and column indices to lookup the

            // substituition box

            string row1 = xored.substr(i \* 6, 1) + xored.substr(i \* 6 + 5, 1);

            int row = convertBinaryToDecimal(row1);

            string col1 = xored.substr(i \* 6 + 1, 1) + xored.substr(i \* 6 + 2, 1) + xored.substr(i \* 6 + 3, 1) + xored.substr(i \* 6 + 4, 1);

            ;

            int col = convertBinaryToDecimal(col1);

            int val = substition\_boxes[i][row][col];

            res += convertDecimalToBinary(val);

        }

        // 3.5. Another permutation is applied

        string perm2 = "";

        for (int i = 0; i < 32; i++)

        {

            perm2 += res[permutation\_tab[i] - 1];

        }

        // 3.6. The result is xored with the left half

        xored = Xor(perm2, left);

        // 3.7. The left and the right parts of the plain text are swapped

        left = xored;

        if (i < 15)

        {

            string temp = right;

            right = xored;

            left = temp;

        }

    }

    // 4. The halves of the plain text are applied

    string combined\_text = left + right;

    string ciphertext = "";

    // The inverse of the initial permuttaion is applied

    for (int i = 0; i < 64; i++)

    {

        ciphertext += combined\_text[inverse\_permutation[i] - 1];

    }

    // And we finally get the cipher text

    return ciphertext;

}

int main()

{

    // A 64 bit key

    string key;

    cout << "Enter a 64 bit (8 letter) key: ";

    cin >> key;

    // A block of plain text of 64 bits

    cout << "Enter a 64 bit (8 letter) plain text: ";

    cin >> pt;

    key = textToBinary(key);

    pt = textToBinary(pt);

    string apt = pt;

    // Calling the function to generate 16 keys

    generate\_keys(key);

    cout << "Plain text: " << pt << endl;

    // Applying the algo

    string ct = DES();

    cout << "Ciphertext: " << ct << endl;

    // Reversing the round\_keys array for decryption

    int i = 15;

    int j = 0;

    while (i > j)

    {

        string temp = round\_keys[i];

        round\_keys[i] = round\_keys[j];

        round\_keys[j] = temp;

        i--;

        j++;

    }

    pt = ct;

    string decrypted = DES();

    cout << "Decrypted text: " << binaryToText(decrypted) << endl;

    // Comapring the initial plain text with the decrypted text

    if (decrypted == apt)

    {

        cout << "Plain text encrypted and decrypted successfully." << endl;

    }

}

**OutPut:**

