# Homework 5: Chapter 3 Questions

Mudit Vats
mpvats@syr.edu
5/20/2020

# Table of Contents

# Overview

This lab report presents observations and explanations for the exercises described in Professor Weissman's HW5 assignment.

# Problem 3.1

As stated in Sect. 3.5.2, one important property which makes DES secure is that the S-boxes are nonlinear. In this problem we verify this property by computing the output of S1 for several pairs of inputs.

Show that S1(x1) ⊕ S1(x2) != S1( x1 ⊕ x2 ), where "⊕" denotes bitwise XOR, for:

## x1 = 000000, x2 = 000001

Left side:

- S1(000000) XOR S1(000001)
- S1(Row=00, Col=0000) XOR S1(Row=01, Col=0000)
- 14 XOR 00
- 1110 XOR 0000
- 1110 => 14

Right side:

- S1(000000 XOR 000001)
- S1(000001)
- S1(Row=01, Col=0000)
- 00

Left side != Right side

- 14 != 0

## x1 = 111111, x2 = 100000

Left side:

- S1(111111) XOR S1(100000)
- S1(Row=11, Col=1111) XOR S1(Row=10, Col=0000)
- 13 XOR 04
- 1101 XOR 0100
- 1001 => 9

Right side:

- S1(111111 XOR 100000)
- S1(011111)
- S1(Row=01, Col=1111)
- 08

Left side != Right side

- 9 != 8

### x1 = 101010, x2 = 010101

Left side:

- S1(101010) XOR S1(010101)
- S1(Row=10, Col=0101) XOR S1(Row=01, Col=1010)
- 06 XOR 12
- 0110 XOR 1100
- 1010 => 10

Right side:

- S1(101010 XOR 010101)
- S1(111111)
- S1(Row=11, Col=1111)
- 13

Left side != Right side

- 10 != 13

## Problem 3.2

We want to verify that IP(·) and IP^−1(·) are truly inverse operations. We consider a vector x = (x1,x2, . . . ,x64) of 64 bit. Show that IP^−1(IP(x)) = x for the first five bits of x, i.e. for xi, i = 1,2,3,4, 5.

### For i=1

- IP^-1(IP(1))
- IP^-1(58)
- 1
- 1 = 1

### For i=2

- IP^-1(IP(2))
- IP^-1(50)
- 2
- 2 = 2

### For i=3

- IP^-1(IP(3))
- IP^-1(42)
- 3
- 3 = 3

### For i=4

- IP^-1(IP(4))

- IP^-1(34)
- 4
- 4 = 4

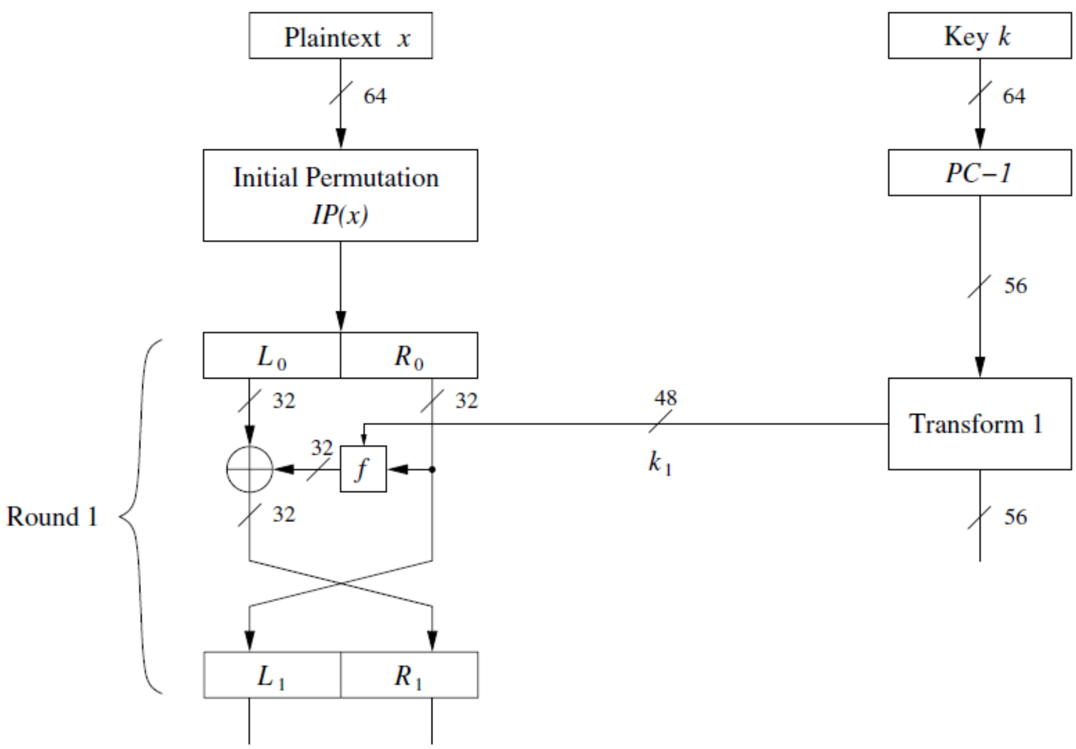
### For i=5

- IP^-1(IP(5))
- IP^-1(26)
- 5
- 5 = 5

## Problem 3.3

What is the output of the first round of the DES algorithm when the plaintext and the key are both all zeros?

Using the diagram below:




- Key k = 0 (size 56 bits)
- Plaintext x = 0 (size 64 bits)

### Calculate ki Subkey

- At Key k, parity bits are added to the 56 bit key.
- Permuted Choice 1: PC-1
  - At PC-1, the parity bits are removed. The key is still k=0.

    - At PC-1, the initial key permutation occurs using Table 3.11 (in the book). For a k=0, this results in k=0, since no matter how you move 0's around, the resultant bits are zero.
- C0 and D0
    - We now split C0 and D0 into two 28 bit halves:
        - C0 = 0000000000000000000000000000
        - D0 = 0000000000000000000000000000
    - We left rotate to get:
        - C1 = 0000000000000000000000000000
        - D1 = 0000000000000000000000000000
- Permuted Choice 2: PC-2
    - At PC-2, the permutation we use is in Table 3.12 (in the book). For C1 and D1 that are all zero's, after PC-2, we still have a key of zero, so k1 = 0.
- k1 = 0 (48-bits).

## Initial Permutation

- Plaintext x = 0 (size 64 bits)
- Using IP Fig 3.8 (in the book), we end up with IP(0) = 0.
- IP(0) = 0.

## Round 1

- L0 = 0 (32 bits), R0 = 0 (32-bits)
- The f-Function
    - R0 = 0 (32-bits) gets expanded to 48 bits. The result, since all zeros, will be E(R0) = 0 (48-bits).
    - E(R0) XOR k1, where E(R0)=0 and k1=0 => 0 (48-bits).
    - Applying S1 – S8 to 0 (48 bits) as input, will yield:
        - Input Bits: 000000 000000 000000 000000 000000 000000 000000 000000
        - S1(000000) = 14
        - S2(000000) = 15
        - S3(000000) = 10
        - S4(000000) = 07

        - S5(000000) = 02
        - S6(000000) = 12
        - S7(000000) = 04
        - S8(000000) = 13
        - Output Bits: 1110 1111 1010 0111 0010 1100 0100 1101
    - Applying the P permutation (using tool I created below)
        - Input Bits: 1110 1111 1010 0111 0010 1100 0100 1101
        - Output Bits: 1101 1000 1101 1000 1101 1011 1011 1100

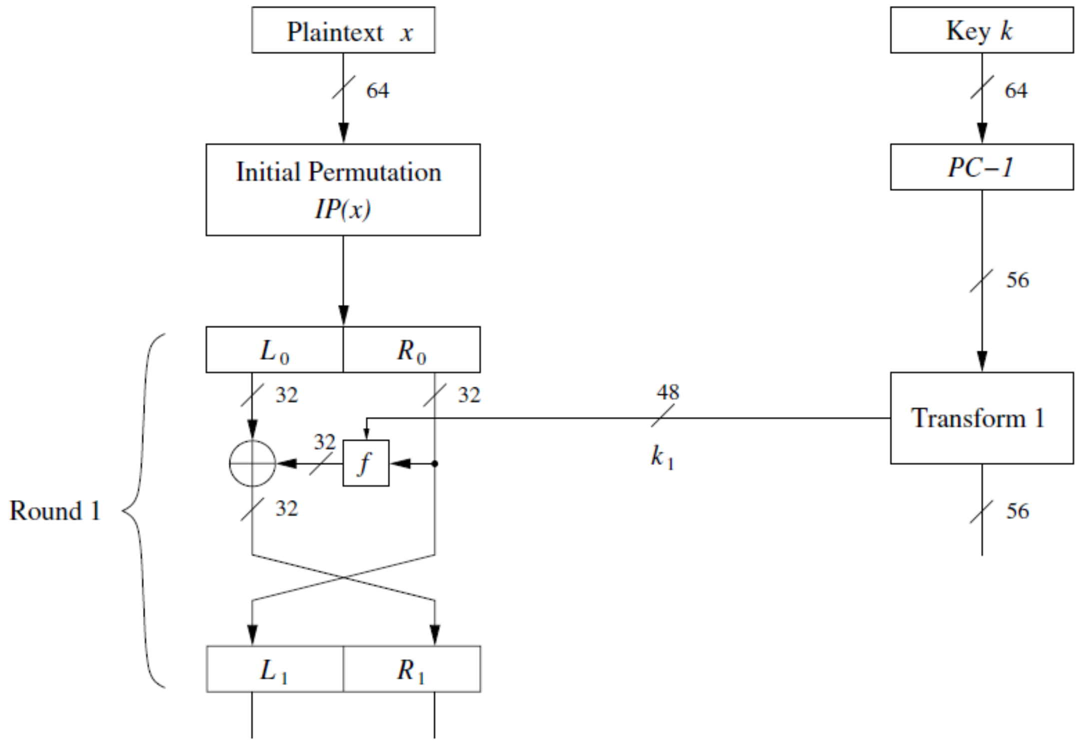
```
C:\Develop\cse628\desp\Debug>desp 11101111101001110010110001001101
Hello DES P Mapper!
 input: 11101111101001110010110001001101
output: 11011000110110001101101110111100
```

    - L0 XOR Output Bits
        - L0 = 0
        - f-Function Bits: 1101 1000 1101 1000 1101 1011 1011 1100
        - L0 XOR f-Function Bits = 1101 1000 1101 1000 1101 1011 1011 1100
        - This becomes R1 = 1101 1000 1101 1000 1101 1011 1011 1100 (0xd8d8dbbc)
        - L1 becomes 0 (32-bits)

# Problem 3.4

What is the output of the first round of the DES algorithm when the plaintext and the key are both all ones?

Using the diagram below:




- Key k = 0xffffffffffffff (size 56 bits)
- Plaintext x = 0xffffffffffffffff (size 64 bits)

## Calculate ki Subkey

- At Key k, parity bits are added to the 56-bit key.
- Permuted Choice 1: PC-1
    - At PC-1, the parity bits are removed. The key is still k=0xffffffffffffff.
    - At PC-1, the initial key permutation occurs using Table 3.11 (in the book). For a k=0xffffffffffffff, this results in k=0xffffffffffffff, since no matter how you move 1's around, the resultant bits are 1's.
- C0 and D0
    - We now split C0 and D0 into two 28 bit halves:
        - C0 = 1111111111111111111111111111
        - D0 = 1111111111111111111111111111
    - We left rotate to get:
        - C1 = 1111111111111111111111111111
        - D1 = 1111111111111111111111111111
- Permuted Choice 2: PC-2
    - At PC-2, the permutation we use is in Table 3.12 (in the book). For C1 and D1 that are all 1's, after PC-2, we still have a key of 1's, so k1 = 0xffffffffffff.
- k1 = 0xffffffffffff (48-bits).

## Initial Permutation

- Plaintext x = 0xffffffffffffffff (size 64 bits)
- Using IP Fig 3.8 (in the book), we end up with IP(0xffffffffffffffff) = 0xffffffffffffffff.
- IP(0xffffffffffffffff) = 0xffffffffffffffff.

## Round 1

- L0 = 0xffffffff (32 bits), R0 = 0xffffffff (32-bits)
- The f-Function
    - R0 = 0xffffffff (32-bits) gets expanded to 48 bits. The result, since all 1's, will be E(R0) = 0xffffffffffff (48-bits).
    - E(R0) XOR k1, where E(R0)= 0xffffffffffff and k1=0xffffffffffff => 0 (48-bits).
    - Applying S1 – S8 to 0 (48 bits) as input, will yield:
        - Input Bits: 000000 000000 000000 000000 000000 000000 000000 000000
        - S1(000000) = 14
        - S2(000000) = 15
        - S3(000000) = 10
        - S4(000000) = 07
        - S5(000000) = 02
        - S6(000000) = 12
        - S7(000000) = 04
        - S8(000000) = 13
        - Output Bits: 1110 1111 1010 0111 0010 1100 0100 1101
    - Applying the P permutation (using tool I created below)
        - Input Bits: 1110 1111 1010 0111 0010 1100 0100 1101
        - Output Bits: 1101 1000 1101 1000 1101 1011 1011 1100

```
C:\Develop\cse628\desp\Debug>desp 11101111101001110010110001001101
Hello DES P Mapper!
 input: 11101111101001110010110001001101
output: 11011000110110001101101110111100
```

- 
    - L0 XOR Output Bits
        - L0 = 0xffffffff =  1111 1111 1111 1111 1111 1111 1111 1111
        - f-Function Bits:   1101 1000 1101 1000 1101 1011 1011 1100
        - L0 XOR f-Function Bits = 0010 0111 0010 0111 0010 0100 0100 0011
        - This becomes R1 = 0010 0111 0010 0111 0010 0100 0100 0011 (0x27272443)
        - L1 becomes 0xffffffff (32-bits)

# Problem 3.5

Remember that it is desirable for good block ciphers that a change in one input bit affects many output bits, a property that is called diffusion or the avalanche effect. We try now to get a feeling for the avalanche property of DES. We apply an input word that has a "1" at bit position 57 and all other bits as well as the key are zero. (Note that the input word has to run through the initial permutation.)

Key = 0x0 (56-bits).

Input bits = 0x0100000000000000 (64-bits).

## 1. How many S-boxes get different inputs compared to the case when an all-zero plaintext is provided?

Initial Permutation

- For IP(0x0100000000000000) => 0x0000000080000000, this yields R0=0x80000000 (32-bits) and L0=0x00000000.
- For IP(0x0000000000000000) => 0x0000000000000000, this yields R0=0x00000000 (32-bits) and L0=0x00000000.

f-Function

- Expansion
  - E(R0=0x80000000) => 0x400000000001 (48-bits)
  - E(R0=0x00000000) => 0x000000000000 (48-bits)
- E(R0) XOR with key
  - E(R0=0x80000000) = 0x400000000001 XOR 0x0 (48-bits) => 0x400000000001 (48-bits)
  - E(R0=0x00000000) = 0x000000000000 XOR 0x0 (48-bits) => 0x000000000000 (48-bits)

For 0x400000000001 XOR 0x0, we can see that two bits are set. When we look at the bit positions, those positions are bit 2 and bit 48. This will allow S1 and S8 boxes to receive different inputs compared to E(R0=0x00000000), since all of those S boxes will receive 0's as inputs.

## 2. What is the minimum number of output bits of the S-boxes that will change according to the S-box design criteria?

Per the textbook (page 65, #4) "If two inputs to an S-box differ in exactly one bit, their outputs must differ in at least two bits.". So, if we change one input bit, two output bits of the S-box will change.

Therefore, in our example above, if two bits are different (0x400000000001 vs 0x000000000000), we would have two sets of changes, which would result in four changed bits. Two bits in S1 and two bits in S8.

## 3. What is the output after the first round?

For 0x400000000001, we have 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001. Broken up into 6 bit chunks, we have:

- 010000 000000 000000 000000 000000 000000 000000 000001

The S-box's are:

- S1(010000) = 03
- S2(000000) = 15
- S3(000000) = 10
- S4(000000) = 07
- S5(000000) = 02
- S6(000000) = 12
- S7(000000) = 04
- S8(000001) = 01

Applying the P permutation (using tool I created below)

- Input Bits: 0011 1111 1010 0111 0010 1100 0100 0001
- Output Bits: 1101 0000 0101 1000 0101 1011 1001 1110

```
C:\Develop\cse628\desp\Debug>desp.exe 00111111101001110010110001000001

Hello DES P Mapper!

 input: 00111111101001110010110001000001

output: 11010000010110000101101110011110
```

- L0 XOR Output Bits
  - L0 = 0x00000000 = 0000 0000 0000 0000 0000 0000 0000 0000
  - f-Function Bits: 1101 0000 0101 1000 0101 1011 1001 1110
  - L0 XOR f-Function Bits = 1101 0000 0101 1000 0101 1011 1001 1110
  - This becomes R1 = 1101 0000 0101 1000 0101 1011 1001 1110 (0xd0585b9e)
  - L1 becomes 0x80000000 (32-bits)

## 4. How many output bit after the first round have actually changed compared to the case when the plaintext is all zero? (Observe that we only consider a single round here. There will be more and more output differences after every new round. Hence the term avalanche effect.)

For the case where the plaintext and key are 0's. Using problem 3.3 answer above, we have:

- This becomes R1 = 1101 1000 1101 1000 1101 1011 1011 1100 (0xd8d8dbbc)

The answer from #3 above is:

- This becomes R1 = 1101 0000 0101 1000 0101 1011 1001 1110 (0xd0585b9e)

The difference are highlighted above. There are six different bits.

# Problem 3.6

An avalanche effect is also desirable for the key: A one-bit change in a key should result in a dramatically different ciphertext if the plaintext is unchanged.

## 1. Assume an encryption with a given key. Now assume the key bit at position 1 (prior to PC–1) is being flipped. Which S-boxes in which rounds are affected by the bit flip during DES encryption?

Key = 0x80000000000000 (56-bits).

Cipher = 0x00000000

Shift 1: Round 1, 2, 9, 16

Shift 2: All other rounds

Note: For all of the PC-2 calculations below, I created a program to handle the permutations. See DES More Advanced Implementation at the end of this homework.

PC-1

- PC-1(0x80000000000000) = 0x01000000000000
- C0 = 0x0100000, D0 = 0x0000000

Transform 1, shift 1

- C1 = LS1(C0) = 0x0200000, D1 = LS1(D0) = 0x0000000
- K1 = PC-2(C1 D1) => PC-2(0x02000000000000) = 0x000010000000
- Bit 20, S4 affected

Transform 2, shift 1

- C2 = LS2(C1) = 0x0400000, D2 = LS2(D1) = 0x0000000
- K2 = PC-2(C2 D2) => PC-2(0x04000000000000) = 0x004000000000
- Bit 10, S2 affected

Transform 3, shift 2

- C3 = LS3(C2) = 0x1000000, D3 = LS2(D2) = 0x0000000
- K3 = PC-2(C3 D3) => PC-2(0x10000000000000) = 0x000100000000

- Bit 16, S3 affected

Transform 4, shift 2

- C4 = LS2(C3) = 0x4000000, D4 = LS2(D3) = 0x0000000
- K4 = PC-2(C4 D4) => PC-2(0x40000000000000) = 0x000001000000
- Bit 24, S4 affected

Transform 5, shift 2

- C5 = LS2(C4) = 0x0000001, D5 = LS2(D4) = 0x0000000
- K5 = PC-2(C5 D5) => PC-2(0x00000010000000) = 0x010000000000
- Bit 8, S2 affected

Transform 6, shift 2

- C6 = LS2(C5) = 0x0000004, D6 = LS2(D5) = 0x0000000
- K6 = PC-2(C6 D6) => PC-2(0x00000040000000) = 0x000080000000
- Bit 17, S3 affected

Transform 7, shift 2

- C7 = LS2(C6) = 0x0000010, D7 = LS2(D6) = 0x0000000
- K7 = PC-2(C7 D7) => PC-2(0x00000100000000) = 0x100000000000
- Bit 4, S1 affected

Transform 8, shift 2

- C8 = LS2(C7) = 0x0000040, D8 = LS2(D7) = 0x0000000
- K8 = PC-2(C8 D8) => PC-2(0x00000400000000) = 0x000000000000
- Bit not set, no S-boxes affected

Transform 9, shift 1

- C9 = LS2(C8) = 0x0000080, D9 = LS2(D8) = 0x0000000
- K9 = PC-2(C9 D9) => PC-2(0x00000800000000) = 0x002000000000
- Bit 11, S2 affected

Transform 10, shift 2

- C10 = LS2(C9) = 0x0000200, D10 = LS2(D9) = 0x0000000
- K10 = PC-2(C10 D10) => PC-2(0x00002000000000) = 0x000400000000
- Bit 14, S3 affected

Transform 11, shift 2

- C11 = LS2(C10) = 0x0000800, D11 = LS2(D10) = 0x0000000
- K11 = PC-2(C11 D11) => PC-2(0x00008000000000) = 0x400000000000
- Bit 2, S1 affected

Transform 12, shift 2

- C12 = LS2(C10) = 0x0002000, D2 = LS2(D10) = 0x0000000
- K12 = PC-2(C12 D12) => PC-2(0x00020000000000) = 0x008000000000
- Bit 9, S2 affected

Transform 13, shift 2

- C13 = LS2(C11) = 0x0008000, D13 = LS2(D11) = 0x0000000
- K13 = PC-2(C13 D13) => PC-2(0x00080000000000) = 0x000002000000
- Bit 23, S4 affected

Transform 14, shift 2

- C14 = LS2(C12) = 0x0020000, D14 = LS2(D12) = 0x0000000
- K14 = PC-2(C14 D14) => PC-2(0x00200000000000) = 0x200000000000
- Bit 3, S1 affected

Transform 15, shift 2

- C15 = LS2(C14) = 0x0080000, D15 = LS2(D14) = 0x0000000
- K15 = PC-2(C15 D15) => PC-2(0x00800000000000) = 0x000000000000
- Bit not set, no S-boxes affected

Transform 16, shift 1

- C16 = LS2(C15) = 0x0100000, D16 = LS2(D15) = 0x0000000
- K16 = PC-2(C16 D16) => PC-2(0x01000000000000) = 0x000040000000
- Bit 18, S3 affected

## 2. Which S-boxes in which DES rounds are affected by this bit flip during DES decryption?

Same S-boxes are affected. From the book "This leads to the interesting property that C0 = C16 and D0 = D16. This is very useful for the decryption key schedule where the subkeys have to be generated in reversed order, as we will see in Sect. 3.4."

So, if C16 and D16 equal C0 and D0, respectively, at the start of the decryption, the same key schedule is followed (aside for round 1 not rotating), yielding the same results and affected S-box bits.

## Problem 3.9

Assume we perform a known-plaintext attack against DES with one pair of plaintext and ciphertext. How many keys do we have to test in a worst-case scenario if we apply an exhaustive key search in a straightforward way? How many on average?

In the book, definition 3.5.1 DES Exhaustive key search, we see the formula:

- $DES_{ki}^{-1}(y) = x$, $i=0\ldots2^{56} - 1$

Worst case would be a search over the full $2^{56}$ key space. Average case would be $2^{56} / 2$ if, on average, we get the correct key with 50% probability.

## Source Code

### DES Permutation (P)

```
// desp.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>

// table 3.10 f-function final permutation
int p[] = {
    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
};

int main(int argc, char *argv[])
{
    char input[32];
    char output[32];
    memset(output, 0, 32);

    std::cout << "Hello DES P Mapper!\n";

    if (argc < 2)
    {
        std::cout << "provide the input bits" << std::endl;
    }

    // copy the argv
    // MSB=0... LSB=31
    for (int i = 0; i <32; i++)
    {
        input[i] = argv[1][i];
    }


    // do the mapping
    // input bit of the table is mapped to output bit of table index
```

```
    // output[table index] = input[ table[table index] ]
    for (int i = 0; i < 32; i++)
    {
        int pIndex = p[i] - 1;
        output[i] = input[pIndex];
    }

    // print the input MSB...LSB
    printf(" input: ");
    for (int i = 0; i < 32; i++)
    {
        printf("%c", input[i]);
    }
    printf("\n");

    // print the output MSB...LSB
    printf("output: ");
    for (int i = 0; i < 32; i++)
    {
        printf("%c", output[i]);
    }
    printf("\n");
}
```

## DES More Advanced Implementation

This code is a modified version of the previous code. I added PC-1 and PC-2 to the implementation. Plus, added some error checking, hex/binary conversion/input and better use of C++.

On a similar note, probably much easier in Python or other higher-level languages (C# or JAVA). Note to self is to spend more time in these languages so quick-n-dirty tools can be more easily implemented…. Working on the code above/below took way longer then necessary. ☹

```
// desp.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>
#include <vector>

#define SIZE_MAX_BUFFER                                    80
#define SIZE_FFUNC_PERMUTATAION_INPUT                      32
#define SIZE_FFUNC_PERMUTATAION_OUTPUT                     32
#define SIZE_KEYSCHED_PERMUTED_CHOICE_1_INPUT              56
#define SIZE_KEYSCHED_PERMUTED_CHOICE_1_OUTPUT             56
#define SIZE_KEYSCHED_PERMUTED_CHOICE_2_INPUT              56
#define SIZE_KEYSCHED_PERMUTED_CHOICE_2_OUTPUT             48

#define MAX_ARGS          4
#define ARG_PERMUTATION   1
#define ARG_INPUT_TYPE    2
#define ARG_INPUT_VALUE   3
```

```
// table 3.10 f-function final permutation
// 32 bit to 32 bit
int p[] = {
    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
    };

// table 3.11 Key Schedule PC-1
// 56 to 56 bit
int pc1[] = {
    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
    };

// table 3.12 Key Schedule PC-2,
// 56 to 48 bit
int pc2[] = {
    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
    };

void dumpArray(std::vector<char> charArray);
std::vector<char> asciiHexToasciiBinary(std::vector<char> charArray);
std::vector<char> asciiBinToasciiHex(std::vector<char> charArray);

int main(int argc, char* argv[])
{
    int* pTable;
    int pTableSize;
    int inputSize;
    bool isBinaryInput = true;
    std::vector<char> input;
    std::vector<char> output;

    std::cout << "Hello DES P, PC-1, PC-2 Table Mapper!" << std::endl;

    if (argc < MAX_ARGS)
    {
        std::cout << "not enough arguments" << std::endl;
        return 1;
    }
```

```
    if (strcmp(argv[ARG_PERMUTATION], "p") == 0)
    {
        pTable = p;
        pTableSize = SIZE_FFUNC_PERMUTATAION_OUTPUT;
        inputSize = SIZE_FFUNC_PERMUTATAION_INPUT;
        std::cout << "using f function final P" << std::endl;
    }
    else if (strcmp(argv[ARG_PERMUTATION], "pc1") == 0)
    {
        pTable = pc1;
        pTableSize = SIZE_KEYSCHED_PERMUTED_CHOICE_1_OUTPUT;
        inputSize = SIZE_KEYSCHED_PERMUTED_CHOICE_1_INPUT;
        std::cout << "using key schedule PC-1" << std::endl;
    }
    else if (strcmp(argv[ARG_PERMUTATION], "pc2") == 0)
    {
        pTable = pc2;
        pTableSize = SIZE_KEYSCHED_PERMUTED_CHOICE_2_OUTPUT;
        inputSize = SIZE_KEYSCHED_PERMUTED_CHOICE_2_INPUT;
        std::cout << "using key schedule PC-2" << std::endl;
    }
    else
    {
        std::cout << "unrecognized key type" << std::endl;
        return 1;
    }

    if (strcmp(argv[ARG_INPUT_TYPE], "bin") == 0)
    {
        isBinaryInput = true;
    }
    else if (strcmp(argv[ARG_INPUT_TYPE], "hex") == 0)
    {
        isBinaryInput = false;
    }
    else
    {
        std::cout << "unrecognized input type" << std::endl;
        return 1;
    }

    // validate input size
    if (isBinaryInput)
    {
        if (strlen(argv[ARG_INPUT_VALUE]) != inputSize)
        {
            std::cout << "bin input length mismatch" << std::endl;
            return 1;
        }
    }
    else
    {
        if (strlen(argv[ARG_INPUT_VALUE]) != (inputSize/4))
        {
```

```
            std::cout << "hex input length mismatch" << std::endl;
            return 1;
        }
    }

    // vectorize the argv, NOTE: MSB=0... LSB=31
    for (int i = 0; i < (int)strlen(argv[ARG_INPUT_VALUE]); i++)
    {
        input.push_back(argv[ARG_INPUT_VALUE][i]);
    }

    if (!isBinaryInput)
    {
        // overwrite input
        input = asciiHexToasciiBinary(input);
    }

    // add parity bits if pc1
    // this step takes the 56 bit key and adds parity bits at
    // ever 7th bit; 8 bits total, so the final size is 64 bits
    std::vector<char> expansion;
    if (pTable == pc1)
    {
        for (int i = 0; i < (int)input.size(); i++)
        {
            expansion.push_back(input[i]);

            if ( ((i + 1) % 7) == 0 )
            {
                // add parity bits
                expansion.push_back('0');
            }
        }

        // pre-expansion
        std::cout << "input " << input.size() << "-bits:" << std::endl;
        dumpArray(input);
        std::cout << " (0x";
        dumpArray(asciiBinToasciiHex(input));
        std::cout << ")";
        std::cout << std::endl;

        std::cout << "expanded key, parity bits added to input" << std::endl;

        // use expansion now
        input = expansion;
    }

    // hex or binary?

    // do the mapping
    // input bit of the table is mapped to output bit of table index
    // output[table index] = input[ table[table index] ]
    for (int i = 0; i < pTableSize; i++)
```

```
    {
        int pIndex = pTable[i] - 1;
        output.push_back(input.at(pIndex));
    }

    // print the input MSB...LSB
    std::cout << "input " << input.size() << "-bits:" << std::endl;
    dumpArray(input);
    std::cout << " (0x";
    dumpArray(asciiBinToasciiHex(input));
    std::cout << ")";
    std::cout << std::endl;

    // print the output MSB...LSB
    std::cout << "output " << output.size() << "-bits:" << std::endl;
    dumpArray(output);
    std::cout << " (0x";
    dumpArray(asciiBinToasciiHex(output));
    std::cout << ")";
    std::cout << std::endl;
}


void dumpArray(std::vector<char> charArray)
{
    for (char c : charArray)
    {
        std::cout << c;
    }
}

std::vector<char> asciiBinToasciiHex(std::vector<char> charArray)
{
    int a, b, c, d = 0;
    std::vector<char> hexArray;

    for (int i = 0; i < (int)charArray.size(); i+=4)
    {
        if (charArray[i + 0] == '1') a = 1; else a = 0;
        if (charArray[i + 1] == '1') b = 1; else b = 0;
        if (charArray[i + 2] == '1') c = 1; else c = 0;
        if (charArray[i + 3] == '1') d = 1; else d = 0;

        int abcd = (a << 3) | (b << 2) | (c << 1) | (d << 0);

        switch (abcd)
        {
        case 0:
            hexArray.push_back('0');
            break;
        case 1:
            hexArray.push_back('1');
            break;
        case 2:
```

```
            hexArray.push_back('2');
            break;
        case 3:
            hexArray.push_back('3');
            break;
        case 4:
            hexArray.push_back('4');
            break;
        case 5:
            hexArray.push_back('5');
            break;
        case 6:
            hexArray.push_back('6');
            break;
        case 7:
            hexArray.push_back('7');
            break;
        case 8:
            hexArray.push_back('8');
            break;
        case 9:
            hexArray.push_back('9');
            break;
        case 10:
            hexArray.push_back('a');
            break;
        case 11:
            hexArray.push_back('b');
            break;
        case 12:
            hexArray.push_back('c');
            break;
        case 13:
            hexArray.push_back('d');
            break;
        case 14:
            hexArray.push_back('e');
            break;
        case 15:
            hexArray.push_back('f');
            break;
        default:
            break;
        }

    }


    return hexArray;
}

std::vector<char> asciiHexToasciiBinary(std::vector<char> charArray)
{
    std::vector<char> binaryArray;
```

```
for (char c : charArray)
{
    switch (c)
    {
    case '0':
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        break;
    case '1':
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        break;
    case '2':
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        binaryArray.push_back('0');
        break;
    case '3':
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        binaryArray.push_back('1');
        break;
    case '4':
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        binaryArray.push_back('0');
        binaryArray.push_back('0');
        break;
    case '5':
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        break;
    case '6':
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        binaryArray.push_back('1');
        binaryArray.push_back('0');
        break;
    case '7':
        binaryArray.push_back('0');
        binaryArray.push_back('1');
        binaryArray.push_back('1');
        binaryArray.push_back('1');
        break;
    case '8':
```

```
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            binaryArray.push_back('0');
            binaryArray.push_back('0');
            break;
        case '9':
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            binaryArray.push_back('0');
            binaryArray.push_back('1');
            break;
        case 'a':
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            break;
        case 'b':
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            break;
        case 'c':
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            binaryArray.push_back('0');
            break;
        case 'd':
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            binaryArray.push_back('1');
            break;
        case 'e':
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            binaryArray.push_back('0');
            break;
        case 'f':
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            binaryArray.push_back('1');
            break;
        default:
            break;
        }
    }

    return binaryArray;
}
```