Lab 10
Implementing Data Encryption Standard (DES)

Name: Syed Asghar Abbas Zaidi	**Student ID: 07201**

10.1 Data Encryption Standard

10.1 Objective

The Objectives of this lab are:

- Perform the initial permutation and final permutation.
- Constructing the DES function
- Designing the Key Generator function
- Implementing the Cipher algorithm
- Encrypt data using the Data Encryption Standard (DES) algorithm.
- Validate the DES implementation with various inputs.

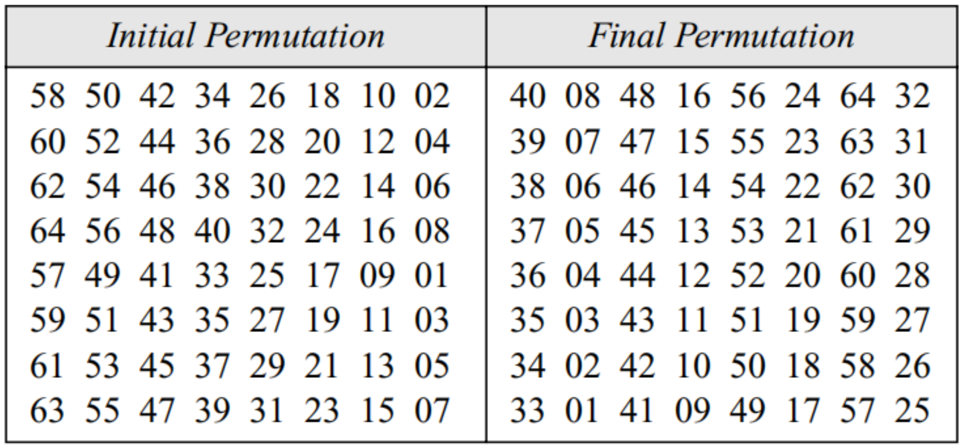
10.2 Background/Scenario

The Data Encryption Standard (DES) is a historically significant symmetric-key block cipher encryption algorithm that played a pivotal role in the early days of computer cryptography. Developed by IBM in the 1970s, DES was adopted as a federal standard in the United States and became one of the most widely used encryption schemes in the world for several decades. In this lab, you will implement Data Encryption Standard (DES) using Verilog HDL, and subsequently employ it to encrypt your data. or the submission, ensure that you provide all your modules and results where applicable under the relevant task headings.

Format

[Design Code Block]
[Simulation Code Block]
[Simulation Pictures Block]

Task 1 – Initial and Final Permutation Tables



Initial Permutation								*Final Permutation*							
58	50	42	34	26	18	10	02	40	08	48	16	56	24	64	32
60	52	44	36	28	20	12	04	39	07	47	15	55	23	63	31
62	54	46	38	30	22	14	06	38	06	46	14	54	22	62	30
64	56	48	40	32	24	16	08	37	05	45	13	53	21	61	29
57	49	41	33	25	17	09	01	36	04	44	12	52	20	60	28
59	51	43	35	27	19	11	03	35	03	43	11	51	19	59	27
61	53	45	37	29	21	13	05	34	02	42	10	50	18	58	26
63	55	47	39	31	23	15	07	33	01	41	09	49	17	57	25

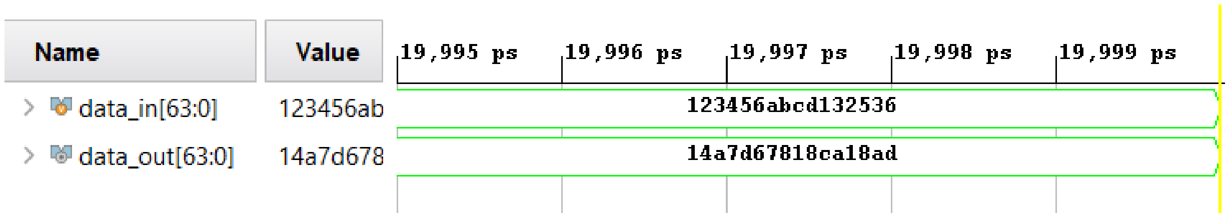
Figure 1: Initial and Final Permutation tables

1. Create a module that returns as output the initial permutation table consisting of 64 bits. Refer to Figure 1 for the change in position of each bit.

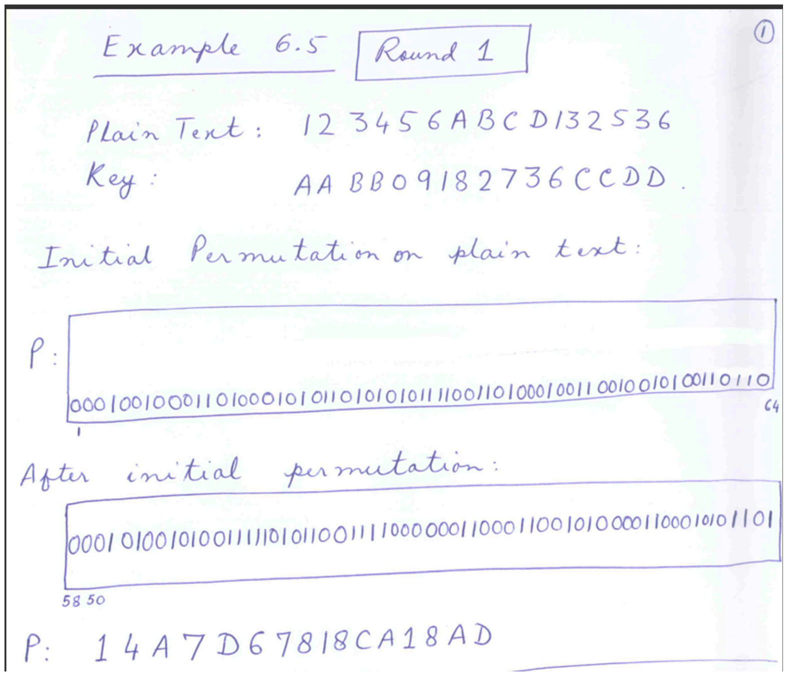
```
module initial_permutation(
        input [0:63] data_in,  // 64-bit input data
        output [0:63] data_out // 64-bit output data after permutation
);

// Perform the bit permutation according to the correct order (starting from data_out[0])
 //58 50 42 34 26 18 10 02
assign data_out[0] = data_in[57];   // Bit 58 to output 0
assign data_out[1] = data_in[49];   // Bit 50 to output 1
assign data_out[2] = data_in[41];   // Bit 42 to output 2
assign data_out[3] = data_in[33];   // Bit 34 to output 3
assign data_out[4] = data_in[25];   // Bit 26 to output 4
assign data_out[5] = data_in[17];   // Bit 18 to output 5
assign data_out[6] = data_in[9]; // Bit 10 to output 6
assign data_out[7] = data_in[1]; // Bit 02 to output 7

// 60 52 44 36 28 20 12 04
assign data_out[8] = data_in[59];   // Bit 60 to output 8
assign data_out[9] = data_in[51];   // Bit 52 to output 9
assign data_out[10] = data_in[43];  // Bit 44 to output 10
assign data_out[11] = data_in[35];  // Bit 36 to output 11
assign data_out[12] = data_in[27];  // Bit 28 to output 12
assign data_out[13] = data_in[19];  // Bit 20 to output 13
assign data_out[14] = data_in[11];  // Bit 12 to output 14
assign data_out[15] = data_in[3];   // Bit 04 to output 15

//  62 54 46 38 30 22 14 06
assign data_out[16] = data_in[61];  // Bit 62 to output 16
assign data_out[17] = data_in[53];  // Bit 54 to output 17
assign data_out[18] = data_in[45];  // Bit 46 to output 18
assign data_out[19] = data_in[37];  // Bit 38 to output 19
assign data_out[20] = data_in[29];  // Bit 30 to output 20
assign data_out[21] = data_in[21];  // Bit 22 to output 21
assign data_out[22] = data_in[13];  // Bit 14 to output 22
assign data_out[23] = data_in[5];   // Bit 06 to output 23

// 64 56 48 40 32 24 16 08
assign data_out[24] = data_in[63];  // Bit 64 to output 24
assign data_out[25] = data_in[55];  // Bit 56 to output 25
assign data_out[26] = data_in[47];  // Bit 48 to output 26
assign data_out[27] = data_in[39];  // Bit 40 to output 27
assign data_out[28] = data_in[31];  // Bit 32 to output 28
assign data_out[29] = data_in[23];  // Bit 24 to output 29
assign data_out[30] = data_in[15];  // Bit 16 to output 30
assign data_out[31] = data_in[7];   // Bit 08 to output 31
```

```
// 57 49 41 33 25 17 09 01
assign data_out[32] = data_in[56];  // Bit 57 to output 32
assign data_out[33] = data_in[48];  // Bit 49 to output 33
assign data_out[34] = data_in[40];  // Bit 41 to output 34
assign data_out[35] = data_in[32];  // Bit 33 to output 35
assign data_out[36] = data_in[24];  // Bit 25 to output 36
assign data_out[37] = data_in[16];  // Bit 17 to output 37
assign data_out[38] = data_in[8];   // Bit 09 to output 38
assign data_out[39] = data_in[0];   // Bit 01 to output 39

// 59 51 43 35 27 19 11 03
assign data_out[40] = data_in[58];  // Bit 59 to output 40
assign data_out[41] = data_in[50];  // Bit 51 to output 41
assign data_out[42] = data_in[42];  // Bit 43 to output 42
assign data_out[43] = data_in[34];  // Bit 35 to output 43
assign data_out[44] = data_in[26];  // Bit 27 to output 44
assign data_out[45] = data_in[18];  // Bit 19 to output 45
assign data_out[46] = data_in[10];  // Bit 11 to output 46
assign data_out[47] = data_in[2];   // Bit 03 to output 47

//  61 53 45 37 29 21 13 05
assign data_out[48] = data_in[60];  // Bit 61 to output 48
assign data_out[49] = data_in[52];  // Bit 53 to output 49
assign data_out[50] = data_in[44];  // Bit 45 to output 50
assign data_out[51] = data_in[36];  // Bit 37 to output 51
assign data_out[52] = data_in[28];  // Bit 29 to output 52
assign data_out[53] = data_in[20];  // Bit 21 to output 53
assign data_out[54] = data_in[12];  // Bit 13 to output 54
assign data_out[55] = data_in[4];   // Bit 05 to output 55

//63 55 47 39 31 23 15 07
assign data_out[56] = data_in[62];  // Bit 63 to output 56
assign data_out[57] = data_in[54];  // Bit 55 to output 57
assign data_out[58] = data_in[46];  // Bit 47 to output 58
assign data_out[59] = data_in[38];  // Bit 39 to output 59
assign data_out[60] = data_in[30];  // Bit 31 to output 60
assign data_out[61] = data_in[22];  // Bit 23 to output 61
assign data_out[62] = data_in[14];  // Bit 15 to output 62
assign data_out[63] = data_in[6];   // Bit 07 to output 63

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();

reg [63:0] data_in;
wire [63:0] data_out;

        // Instantiate the initial_permutation module
initial_permutation uut (
        .data_in(data_in),
        .data_out(data_out)
);

// Initial block for simulation setup
initial begin
// Initialize input with hex value "123456ABCD132536"
 data_in = 64'h123456ABCD132536;
 #10 //to simulate propagation

 // Display the input and output
 $display("Input Data  (Hex): %h", data_in);
  $display("Output Data (Hex): %h", data_out);

  // End the simulation
  #10;
  $finish;
end

endmodule
```



Name	Value	19,995 ps	19,996 ps	19,997 ps	19,998 ps	19,999 ps
data_in[63:0]	123456ab	123456abcd132536				
data_out[63:0]	14a7d678	14a7d67818ca18ad				



Example 6.5 Round 1

Plain Text : 123456ABCD132536
Key : AABB09182736CCDD.

Initial Permutation on plain text:

P: 0001001000110100010101101010101111001101000100110010010100110110

After initial permutation:

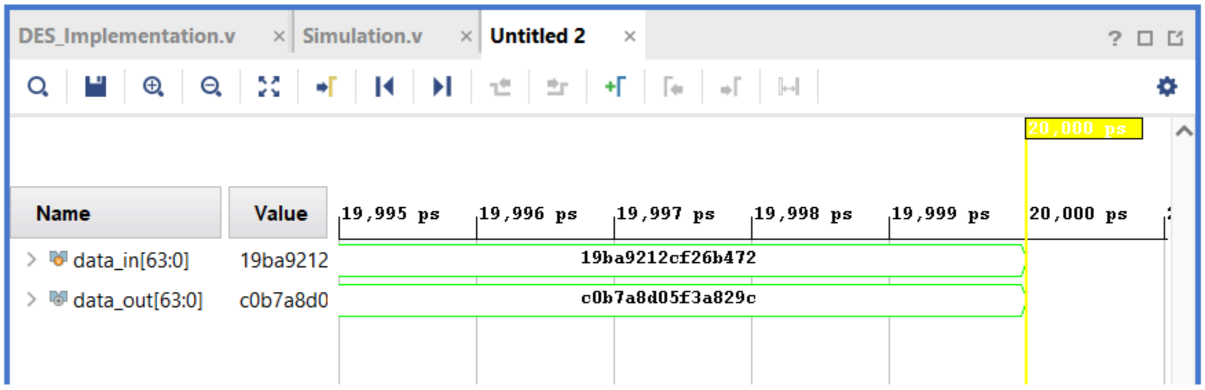
0001010010100111110101100111100000011000110010100001100010101101

P: 14A7D67818CA18AD

2. Create a module that returns as output the final permutation table consisting of 64 bits. Refer to Figure 1 for the change in position of each bit.

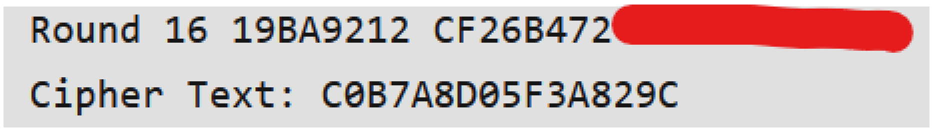
```
module final_permutation(
        input [0:63] data_in,  // 64-bit input data
        output [0:63] data_out // 64-bit output data after final permutation
);

// Perform the bit permutation according to the final permutation (FP) table
//40 08 48 16 56 24 64 32
assign data_out[0] = data_in[39];   // Bit 40 to output 0
assign data_out[1] = data_in[7]; // Bit 08 to output 1
assign data_out[2] = data_in[47];   // Bit 48 to output 2
assign data_out[3] = data_in[15];   // Bit 16 to output 3
assign data_out[4] = data_in[55];   // Bit 56 to output 4
assign data_out[5] = data_in[23];   // Bit 24 to output 5
assign data_out[6] = data_in[63];   // Bit 64 to output 6
assign data_out[7] = data_in[31];   // Bit 32 to output 7

//39 07 47 15 55 23 63 31
assign data_out[8] = data_in[38];   // Bit 39 to output 8
assign data_out[9] = data_in[6]; // Bit 07 to output 9
assign data_out[10] = data_in[46];  // Bit 47 to output 10
assign data_out[11] = data_in[14];  // Bit 15 to output 11
assign data_out[12] = data_in[54];  // Bit 55 to output 12
assign data_out[13] = data_in[22];  // Bit 23 to output 13
assign data_out[14] = data_in[62];  // Bit 63 to output 14
assign data_out[15] = data_in[30];  // Bit 31 to output 15

// 38 06 46 14 54 22 62 30
assign data_out[16] = data_in[37];  // Bit 38 to output 16
assign data_out[17] = data_in[5];   // Bit 06 to output 17
assign data_out[18] = data_in[45];  // Bit 46 to output 18
assign data_out[19] = data_in[13];  // Bit 14 to output 19
assign data_out[20] = data_in[53];  // Bit 54 to output 20
assign data_out[21] = data_in[21];  // Bit 22 to output 21
assign data_out[22] = data_in[61];  // Bit 62 to output 22
assign data_out[23] = data_in[29];  // Bit 30 to output 23

// 37 05 45 13 53 21 61 29
assign data_out[24] = data_in[36];  // Bit 37 to output 24
assign data_out[25] = data_in[4];   // Bit 05 to output 25
assign data_out[26] = data_in[44];  // Bit 45 to output 26
assign data_out[27] = data_in[12];  // Bit 13 to output 27
assign data_out[28] = data_in[52];  // Bit 53 to output 28
assign data_out[29] = data_in[20];  // Bit 21 to output 29
assign data_out[30] = data_in[60];  // Bit 61 to output 30
assign data_out[31] = data_in[28];  // Bit 29 to output 31

// 36 04 44 12 52 20 60 28
assign data_out[32] = data_in[35];  // Bit 36 to output 32
assign data_out[33] = data_in[3];   // Bit 04 to output 33
assign data_out[34] = data_in[43];  // Bit 44 to output 34
assign data_out[35] = data_in[11];  // Bit 12 to output 35
assign data_out[36] = data_in[51];  // Bit 52 to output 36
assign data_out[37] = data_in[19];  // Bit 20 to output 37
assign data_out[38] = data_in[59];  // Bit 60 to output 38
assign data_out[39] = data_in[27];  // Bit 28 to output 39

// 35 03 43 11 51 19 59 27
assign data_out[40] = data_in[34];  // Bit 35 to output 40
assign data_out[41] = data_in[2];   // Bit 03 to output 41
assign data_out[42] = data_in[42];  // Bit 43 to output 42
assign data_out[43] = data_in[10];  // Bit 11 to output 43
assign data_out[44] = data_in[50];  // Bit 51 to output 44
assign data_out[45] = data_in[18];  // Bit 19 to output 45
assign data_out[46] = data_in[58];  // Bit 59 to output 46
assign data_out[47] = data_in[26];  // Bit 27 to output 47

// 34 02 42 10 50 18 58 26
assign data_out[48] = data_in[33];  // Bit 34 to output 48
assign data_out[49] = data_in[1];   // Bit 01 to output 49
assign data_out[50] = data_in[41];  // Bit 41 to output 50
assign data_out[51] = data_in[9];   // Bit 09 to output 51
assign data_out[52] = data_in[49];  // Bit 49 to output 52
assign data_out[53] = data_in[17];  // Bit 17 to output 53
assign data_out[54] = data_in[57];  // Bit 57 to output 54
assign data_out[55] = data_in[25];  // Bit 25 to output 55

// 33 01 41 09 49 17 57 25
assign data_out[56] = data_in[32];  // Bit 33 to output 56
assign data_out[57] = data_in[0];   // Bit 01 to output 57
assign data_out[58] = data_in[40];  // Bit 41 to output 58
assign data_out[59] = data_in[8];   // Bit 09 to output 59
assign data_out[60] = data_in[48];  // Bit 49 to output 60
assign data_out[61] = data_in[16];  // Bit 17 to output 61
assign data_out[62] = data_in[56];  // Bit 57 to output 62
assign data_out[63] = data_in[24];  // Bit 25 to output 63

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();

reg [0:63] data_in;
wire [0:63] data_out;

        // Instantiate the final_permutation module
final_permutation uut (
            .data_in(data_in),
            .data_out(data_out)
);

// Initial block for simulation setup
initial begin
// Initialize input with hex value "123456ABCD132536"
 data_in = 64'h19BA9212CF26B472;
 #10 //to simulate propagation

 // Display the input and output
 $display("Input Data  (Hex): %h", data_in);
  $display("Output Data (Hex): %h", data_out);

  // End the simulation
  #10;
  $finish;
end

endmodule
```



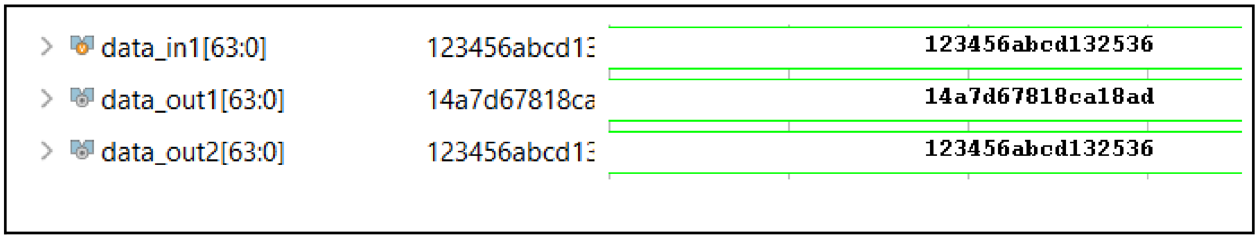
```
DES_Implementation.v   Simulation.v   Untitled 2

Name             Value       19,995 ps  19,996 ps  19,997 ps  19,998 ps  19,999 ps  20,000 ps
> data_in[63:0]  19ba9212               19ba9212cf26b472
> data_out[63:0] c0b7a8d0               c0b7a8d05f3a829c
```

https://www.geeksforgeeks.org/data-encryption-standard-des-set-1/



```
Round 16 19BA9212 CF26B472[redacted]

Cipher Text: C0B7A8D05F3A829C
```

Doing initial as well as final permutation:



```
data_in1[63:0]    123456abcd13    123456abcd132536
data_out1[63:0]   14a7d67818ca    14a7d67818ca18ad
data_out2[63:0]   123456abcd13    123456abcd132536
```

Task 2 – Constructing the DES function

The DES function applies a 48-bit key to the rightmost 32 bits (RI−1) to produce a 32-bit output. This function is made up of four sections: an expansion P-box, a whitener (that adds key), a group of S-boxes, and a straight P-box

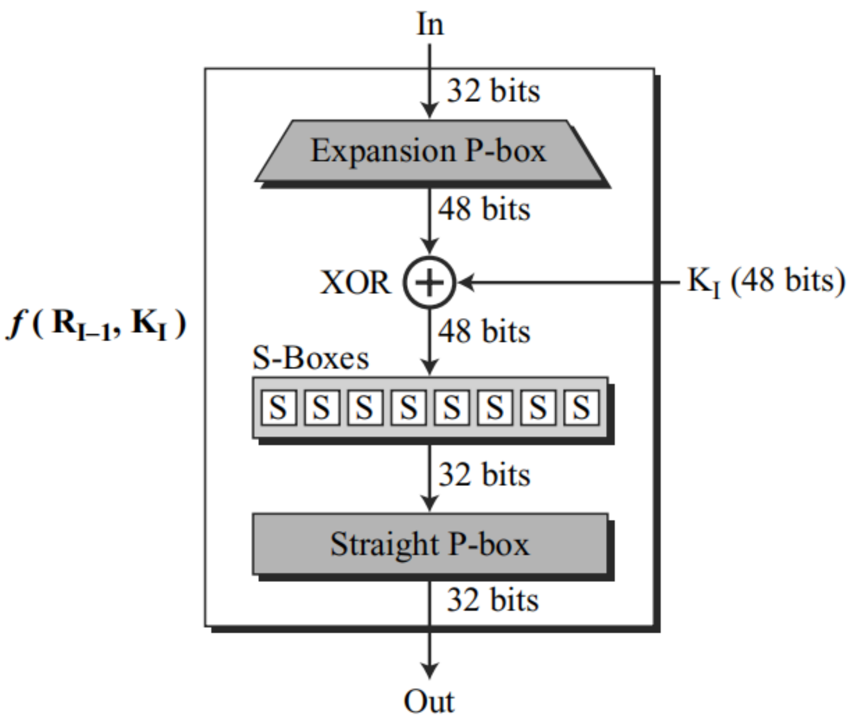


Figure 2: DES function

Task 2.1 - Creating the Expansion P-Box Module

Use Figure 3 to create a module that takes as input 32-bit and returns as output a 48-bit text.

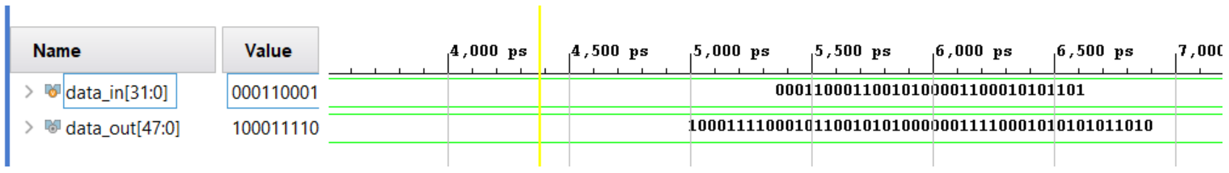
32	01	02	03	04	05
04	05	06	07	08	09
08	09	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	01

Figure 3: Expansion P-box Table

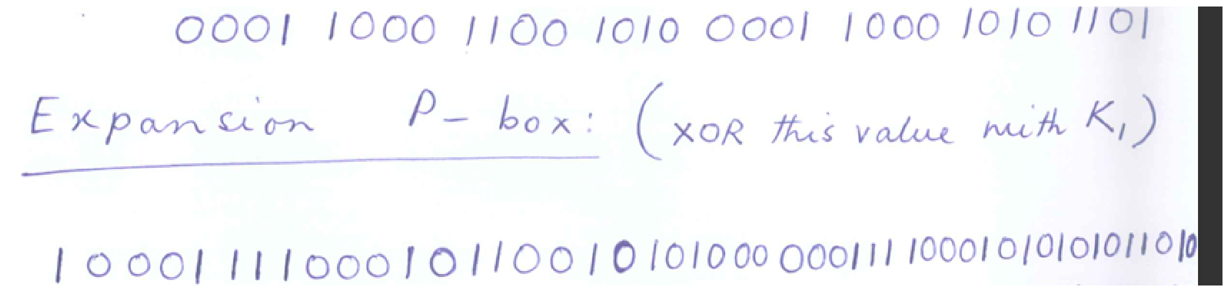
```
module expansion_pbox(
        input [31:0] data_in,   // 32-bit input
        output [47:0] data_out  // 48-bit expanded output
);

// Perform the bit expansion according to the expansion P-box table
assign data_out[0] = data_in[31];  // Bit 32 to output 0
assign data_out[1] = data_in[0];   // Bit 01 to output 1
assign data_out[2] = data_in[1];   // Bit 02 to output 2
assign data_out[3] = data_in[2];   // Bit 03 to output 3
assign data_out[4] = data_in[3];   // Bit 04 to output 4
assign data_out[5] = data_in[4];   // Bit 05 to output 5

assign data_out[6] = data_in[3];   // Bit 04 to output 6
assign data_out[7] = data_in[4];   // Bit 05 to output 7
assign data_out[8] = data_in[5];   // Bit 06 to output 8
assign data_out[9] = data_in[6];   // Bit 07 to output 9
assign data_out[10] = data_in[7];  // Bit 08 to output 10
assign data_out[11] = data_in[8];  // Bit 09 to output 11

assign data_out[12] = data_in[7];  // Bit 08 to output 12
assign data_out[13] = data_in[8];  // Bit 09 to output 13
assign data_out[14] = data_in[9];  // Bit 10 to output 14
assign data_out[15] = data_in[10]; // Bit 11 to output 15
assign data_out[16] = data_in[11]; // Bit 12 to output 16
assign data_out[17] = data_in[12]; // Bit 13 to output 17
assign data_out[18] = data_in[11]; // Bit 12 to output 18
assign data_out[19] = data_in[12]; // Bit 13 to output 19
assign data_out[20] = data_in[13]; // Bit 14 to output 20
assign data_out[21] = data_in[14]; // Bit 15 to output 21
assign data_out[22] = data_in[15]; // Bit 16 to output 22
assign data_out[23] = data_in[16]; // Bit 17 to output 23

assign data_out[24] = data_in[15]; // Bit 16 to output 24
assign data_out[25] = data_in[16]; // Bit 17 to output 25
assign data_out[26] = data_in[17]; // Bit 18 to output 26
assign data_out[27] = data_in[18]; // Bit 19 to output 27
assign data_out[28] = data_in[19]; // Bit 20 to output 28
assign data_out[29] = data_in[20]; // Bit 21 to output 29

assign data_out[30] = data_in[19]; // Bit 20 to output 30
assign data_out[31] = data_in[20]; // Bit 21 to output 31
assign data_out[32] = data_in[21]; // Bit 22 to output 32
assign data_out[33] = data_in[22]; // Bit 23 to output 33
assign data_out[34] = data_in[23]; // Bit 24 to output 34
assign data_out[35] = data_in[24]; // Bit 25 to output 35

assign data_out[36] = data_in[23]; // Bit 24 to output 36
assign data_out[37] = data_in[24]; // Bit 25 to output 37
assign data_out[38] = data_in[25]; // Bit 26 to output 38
assign data_out[39] = data_in[26]; // Bit 27 to output 39
assign data_out[40] = data_in[27]; // Bit 28 to output 40
assign data_out[41] = data_in[28]; // Bit 29 to output 41

assign data_out[42] = data_in[27]; // Bit 28 to output 42
assign data_out[43] = data_in[28]; // Bit 29 to output 43
assign data_out[44] = data_in[29]; // Bit 30 to output 44
assign data_out[45] = data_in[30]; // Bit 31 to output 45
assign data_out[46] = data_in[31]; // Bit 32 to output 46
assign data_out[47] = data_in[0];  // Bit 01 to output 47

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();

reg [31:0] data_in;
wire [47:0] data_out;

        // Instantiate the final_permutation module
```

```
expansion_pbox uut (
            .data_in(data_in),
            .data_out(data_out)
);

// Initial block for simulation setup
initial begin
// Initialize input with hex value "123456ABCD132536"
 data_in = 32'h18ca18ad;
 #10 //to simulate propagation

 // Display the input and output
 $display("Input Data  (Hex): %h", data_in);
  $display("Output Data (Hex): %h", data_out);

  // End the simulation
  #10;
  $finish;
end

endmodule
```



```
data_in[31:0]    000110001    00011000110010100001100010101101
data_out[47:0]   100011110    100011110001011001010100000011110001010101011010
```



0001 1000 1100 1010 0001 1000 1010 1101

Expansion P- box: (XOR this value with K_1)

1 0 001 111000101100 10 101000 000111 100010101010110 10

Matches with Sir Solution when he used Expansion P-box

Task 2.2 - Whitener (XOR)

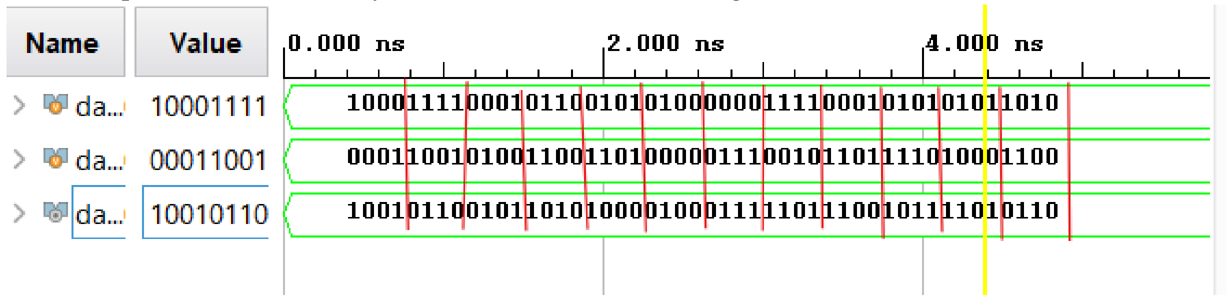
After the expansion permutation, DES uses the XOR operation on the expanded right section and the round key. Note that both the right section and the keys are 48-bits in length. Also note that the round key is used only in this operation.

Create a module for XOR that implements this functionality.

```
module xor_48bit(
            input [47:0] num1,  // First 48-bit input
            input [47:0] num2,  // Second 48-bit input
            output [47:0] result  // 48-bit XOR result
);

// XOR operation between the two 48-bit numbers
assign result = num1 ^ num2;

endmoduleS
```

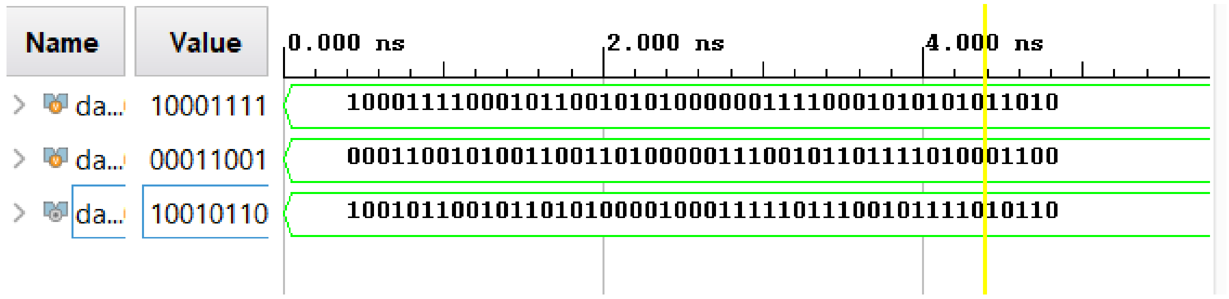
```
 // Set the timescale for simulation
`timescale 1ns/1ps
module simulation();

reg [47:0] data_in1;
reg [47:0] data_in2;
wire [47:0] data_out;

            // Instantiate the final_permutation module
xor_48bit uut (data_in1, data_in2, data_out);

// Initial block for simulation setup
initial begin
// Initialize input with hex value "123456ABCD132536"
 data_in1 = 48'b1000_1111_0001_0110_0101_0100_0000_1111_0001_0101_0101_1010;
 data_in2 = 48'b0001_1001_0100_1100_1101_0000_0111_0010_1101_1110_1000_1100;
 #10 //to simulate propagation

 // Display the input and output
// $display("Input Data  (Hex): %h", data_in);
//  $display("Output Data (Hex): %h", data_out);

  // End the simulation
  #10;
  $finish;
end

endmodule
```

Red-lines put for better visibility that XOR function is working as intended



```
Name    Value
da...   10001111   100011110001011001010100000011110001010101011010
da...   00011001   000110010100110011010000011100101101111010001100
da...   10010110   100101100101101010000100011111011100101111010110
```

(Without red-lines)



```
Name    Value
da...   10001111   100011110001011001010100000011110001010101011010
da...   00011001   000110010100110011010000011100101101111010001100
da...   10010110   100101100101101010000100011111011100101111010110
```

Sir's working// Set the timescale for simulation

```
`timescale 1ns/1ps
module simulation();

reg [47:0] data_in1;
reg [47:0] data_in2;
wire [47:0] data_out;

            // Instantiate the final_permutation module
xor_48bit uut (data_in1, data_in2, data_out);

// Initial block for simulation setup
initial begin
// Initialize input with hex value "123456ABCD132536"
 data_in1 = 48'b1000_1111_0001_0110_0101_0100_0000_1111_0001_0101_0101_1010;
 data_in2 = 48'b0001_1001_0100_1100_1101_0000_0111_0010_1101_1110_1000_1100;
 #10 //to simulate propagation

 // Display the input and output
// $display("Input Data  (Hex): %h", data_in);
//  $display("Output Data (Hex): %h", data_out);

  // End the simulation
  #10;
  $finish;
end
```

```
endmodule
```

Task 2.3 - Construction of S-boxes.

DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output. The 48-bit data from the second operation is divided into eight 6-bit chunks, and each chunk is fed into a box. The result of each box is a 4-bit chunk; when these are combined the result is a 32-bit text.

1. Construct the table for S-box 1 using the table below.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
0	14	04	13	01	02	15	11	08	03	10	06	12	05	09	00	07
1	00	15	07	04	14	02	13	10	03	06	12	11	09	05	03	08
2	04	01	14	08	13	06	02	11	15	12	09	07	03	10	05	00
3	15	12	08	02	04	09	01	07	05	11	03	14	10	00	06	13

Table 1: S-box 1

2. Construct the table for S-box 2 using the table below.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
0	15	01	08	14	06	11	03	04	09	07	02	13	12	00	05	10
1	03	13	04	07	15	02	08	14	12	00	01	10	06	09	11	05
2	00	14	07	11	10	04	13	01	05	08	12	06	09	03	02	15
3	13	08	10	01	03	15	04	02	11	06	07	12	00	05	14	09

Table 2: S-box 2

3. Construct the table for S-box 3 using the table below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	00	09	14	06	03	15	05	01	13	12	07	11	04	02	08
1	13	07	00	09	03	04	06	10	02	08	05	14	12	11	15	01
2	13	06	04	09	08	15	03	00	11	01	02	12	05	10	14	07
3	01	10	13	00	06	09	08	07	04	15	14	03	11	05	02	12

Table 3: S-box 4

4. Construct the table for S-box 4 using the table below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	07	13	14	03	00	6	09	10	1	02	08	05	11	12	04	15
1	13	08	11	05	06	15	00	03	04	07	02	12	01	10	14	09
2	10	06	09	00	12	11	07	13	15	01	03	14	05	02	08	04
3	03	15	00	06	10	01	13	08	09	04	05	11	12	07	02	14

Table 4: S-box 4

5. Construct the table for S-box 5 using the table below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	02	12	04	01	07	10	11	06	08	05	03	15	13	00	14	09
1	14	11	02	12	04	07	13	01	05	00	15	10	03	09	08	06
2	04	02	01	11	10	13	07	08	15	09	12	05	06	03	00	14
3	11	08	12	07	01	14	02	13	06	15	00	09	10	04	05	03

Table 5: S-box 5

6. Construct the table for S-box 6 using the table below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	01	10	15	09	02	06	08	00	13	03	04	14	07	05	11
1	10	15	04	02	07	12	09	05	06	01	13	14	00	11	03	08
2	09	14	15	05	02	08	12	03	07	00	04	10	01	13	11	06
3	04	03	02	12	09	05	15	10	11	14	01	07	10	00	08	13

Table 6: S-box 6

7. Construct the table for S-box 7 using the table below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	11	2	14	15	00	08	13	03	12	09	07	05	10	06	01
1	13	00	11	07	04	09	01	10	14	03	05	12	02	15	08	06
2	01	04	11	13	12	03	07	14	10	15	06	08	00	05	09	02
3	06	11	13	08	01	04	10	07	09	05	00	15	14	02	03	12

Table 7: S-box 7

8. Construct the table for S-box 8 using the table below.

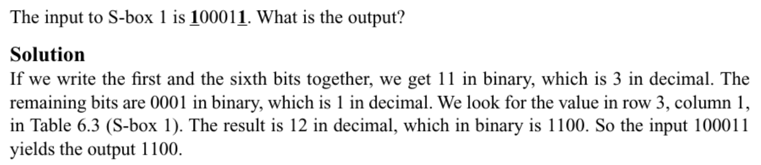
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	02	08	04	06	15	11	01	10	09	03	14	05	00	12	07
1	01	15	13	08	10	03	07	04	12	05	06	11	10	14	09	02
2	07	11	04	01	09	12	14	02	00	06	10	10	15	03	05	08
3	02	01	14	07	04	10	8	13	15	12	09	09	03	05	06	11

Table 8: S-box 8

So basically, here I am attaching the proof for my logic.
I break my 48-bit block into 8 6-bits blocks.
There is a module named S_box_pre_calculation which maps the following

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

This is important cause regardless of what s-box I am calculating for, I will need to do these types of calculation, attached below is a book example



The input to S-box 1 is **1**0001**1**. What is the output?

Solution

If we write the first and the sixth bits together, we get 11 in binary, which is 3 in decimal. The remaining bits are 0001 in binary, which is 1 in decimal. We look for the value in row 3, column 1, in Table 6.3 (S-box 1). The result is 12 in decimal, which in binary is 1100. So the input 100011 yields the output 1100.

AFTER WHICH, I just need to look up the index for that relevant s-box.

```
module S_box_pre_calculation(
        input [5:0] in,
        output reg [5:0] out
        );

  always @* begin
        out = 6'b000000; // Initialize to 0
        case (in)
        6'b0_0000_0: out = 0;
        6'b0_0001_0: out = 1;
        6'b0_0010_0: out = 2;
        6'b0_0011_0: out = 3;
        6'b0_0100_0: out = 4;
        6'b0_0101_0: out = 5;
        6'b0_0110_0: out = 6;
        6'b0_0111_0: out = 7;
        6'b0_1000_0: out = 8;
        6'b0_1001_0: out = 9;
        6'b0_1010_0: out = 10;
        6'b0_1011_0: out = 11;
        6'b0_1100_0: out = 12;
        6'b0_1101_0: out = 13;
        6'b0_1110_0: out = 14;
        6'b0_1111_0: out = 15;

        6'b0_0000_1: out = 16;
        6'b0_0001_1: out = 17;
        6'b0_0010_1: out = 18;
        6'b0_0011_1: out = 19;
        6'b0_0100_1: out = 20;
        6'b0_0101_1: out = 21;
        6'b0_0110_1: out = 22;
        6'b0_0111_1: out = 23;
        6'b0_1000_1: out = 24;
        6'b0_1001_1: out = 25;
        6'b0_1010_1: out = 26;
        6'b0_1011_1: out = 27;
        6'b0_1100_1: out = 28;
        6'b0_1101_1: out = 29;
        6'b0_1110_1: out = 30;
        6'b0_1111_1: out = 31;

        6'b1_0000_0: out = 32;
        6'b1_0001_0: out = 33;
        6'b1_0010_0: out = 34;
        6'b1_0011_0: out = 35;
        6'b1_0100_0: out = 36;
        6'b1_0101_0: out = 37;
        6'b1_0110_0: out = 38;
        6'b1_0111_0: out = 39;
        6'b1_1000_0: out = 40;
        6'b1_1001_0: out = 41;
        6'b1_1010_0: out = 42;
        6'b1_1011_0: out = 43;
        6'b1_1100_0: out = 44;
        6'b1_1101_0: out = 45;
        6'b1_1110_0: out = 46;
        6'b1_1111_0: out = 47;

        6'b1_0000_1: out = 48;
        6'b1_0001_1: out = 49;
        6'b1_0010_1: out = 50;
        6'b1_0011_1: out = 51;
        6'b1_0100_1: out = 52;
        6'b1_0101_1: out = 53;
        6'b1_0110_1: out = 54;
        6'b1_0111_1: out = 55;
        6'b1_1000_1: out = 56;
        6'b1_1001_1: out = 57;
        6'b1_1010_1: out = 58;
        6'b1_1011_1: out = 59;
        6'b1_1100_1: out = 60;
        6'b1_1101_1: out = 61;
        6'b1_1110_1: out = 62;
        6'b1_1111_1: out = 63;
        default: out = 0;
        endcase
        end
endmodule
```

```
module S1(
	input [5:0] in,
	output reg [3:0] out
	);

  initial begin
	out = 4'b0000;  // Initialize out to 0
 end

 always @* case (in)
	0 : out = 14;
	1 : out = 4;
	2 : out = 13;
	3 : out = 01;
	4 : out = 02;
	5 : out = 15;
	6 : out = 11;
	7 : out = 08;
	8 : out = 03;
	9 : out = 10;
	10 : out = 06;
	11 : out = 12;
	12 : out = 05;
	13 : out = 09;
	14 : out = 00;
	15 : out = 07;

	16 : out = 0;
	17 : out = 15;
	18 : out = 07;
	19 : out = 04;
	20 : out = 14;
	21 : out = 02;
	22 : out = 13;
	23 : out = 10;
	24 : out = 03;
	25 : out = 06;
	26 : out = 12;
	27 : out = 11;
	28 : out = 09;
	29 : out = 05;
	30 : out = 03;
	31 : out = 08;

	32 : out = 04;
```

```
        33 : out = 01;
        34 : out = 14;
        35 : out = 08;
        36 : out = 13;
        37 : out = 06;
        38 : out = 02;
        39 : out = 11;
        40 : out = 15;
        41 : out = 12;
        42 : out = 09;
        43 : out = 07;
        44 : out = 03;
        45 : out = 10;
        46 : out = 05;
        47 : out = 00;

        48 : out = 15;
        49 : out = 12;
        50 : out = 08;
        51 : out = 02;
        52 : out = 04;
        53 : out = 09;
        54 : out = 01;
        55 : out = 07;
        56 : out = 05;
        57 : out = 11;
        58 : out = 03;
        59 : out = 14;
        60 : out = 10;
        61 : out = 00;
        62 : out = 06;
        63 : out = 13;
  endcase
endmodule

module S2(
        input [5:0] in,
        output reg [3:0] out
        );
        initial begin
        out = 4'b0000;  // Initialize out to 0
  end

  always @* case (in)
  //15 01 08 14 06 11 03 04 09 07 02 13 12 00 05 10
```

```
0 : out = 15;
1 : out = 01;
2 : out = 08;
3 : out = 14;
4 : out = 06;
5 : out = 11;
6 : out = 03;
7 : out = 04;
8 : out = 09;
9 : out = 07;
10 : out = 02;
11 : out = 13;
12 : out = 12;
13 : out = 00;
14 : out = 05;
15 : out = 10;

//03 13 04 07 15 02 08 14 12 00 01 10 06 09 11 05
16 : out = 03;
17 : out = 13;
18 : out = 04;
19 : out = 07;
20 : out = 15;
21 : out = 02;
22 : out = 08;
23 : out = 14;
24 : out = 12;
25 : out = 00;
26 : out = 01;
27 : out = 10;
28 : out = 06;
29 : out = 09;
30 : out = 11;
31 : out = 05;

32 : out = 00;
33 : out = 14;
34 : out = 07;
35 : out = 11;
36 : out = 10;
37 : out = 04;
38 : out = 13;
39 : out = 01;
40 : out = 05;
41 : out = 08;
```

```
		42 : out = 12;
		43 : out = 06;
		44 : out = 09;
		45 : out = 03;
		46 : out = 02;
		47 : out = 15;

		48 : out = 13;
		49 : out = 08;
		50 : out = 10;
		51 : out = 01;
		52 : out = 03;
		53 : out = 15;
		54 : out = 04;
		55 : out = 02;
		56 : out = 11;
		57 : out = 06;
		58 : out = 07;
		59 : out = 12;
		60 : out = 00;
		61 : out = 05;
		62 : out = 14;
		63 : out = 09;
  endcase
endmodule

module S3(input [5:0] in, output reg [3:0] out);

  initial begin
		out = 4'b0000;  // Initialize out to 0
  end

  always @* case (in)
		0 : out = 10;
		1 : out = 00;
		2 : out = 09;
		3 : out = 14;
		4 : out = 06;
		5 : out = 03;
		6 : out = 15;
		7 : out = 05;
		8 : out = 01;
		9 : out = 13;
		10 : out = 12;
		11 : out = 07;
```

```
12 : out = 11;
13 : out = 04;
14 : out = 02;
15 : out = 08;

16 : out = 13;
17 : out = 07;
18 : out = 00;
19 : out = 09;
20 : out = 03;
21 : out = 04;
22 : out = 06;
23 : out = 10;
24 : out = 02;
25 : out = 08;
26 : out = 05;
27 : out = 14;
28 : out = 12;
29 : out = 11;
30 : out = 15;
31 : out = 01;

32 : out = 13;
33 : out = 06;
34 : out = 04;
35 : out = 09;
36 : out = 08;
37 : out = 15;
38 : out = 03;
39 : out = 00;
40 : out = 11;
41 : out = 01;
42 : out = 02;
43 : out = 12;
44 : out = 05;
45 : out = 10;
46 : out = 14;
47 : out = 07;

48 : out = 01;
49 : out = 10;
50 : out = 13;
51 : out = 00;
52 : out = 06;
53 : out = 09;
```

```
          54 : out = 08;
          55 : out = 07;
          56 : out = 04;
          57 : out = 15;
          58 : out = 14;
          59 : out = 03;
          60 : out = 11;
          61 : out = 05;
          62 : out = 02;
          63 : out = 12;
  endcase
endmodule

module S4(input [5:0] in, output reg [3:0] out);

  initial begin
          out = 4'b0000;  // Initialize out to 0
  end
  always @* case (in)
          0 : out = 07;
          1 : out = 13;
          2 : out = 14;
          3 : out = 03;
          4 : out = 00;
          5 : out = 06;
          6 : out = 09;
          7 : out = 10;
          8 : out = 01;
          9 : out = 02;
          10 : out = 08;
          11 : out = 05;
          12 : out = 11;
          13 : out = 12;
          14 : out = 04;
          15 : out = 15;

          16 : out = 13;
          17 : out = 08;
          18 : out = 11;
          19 : out = 05;
          20 : out = 06;
          21 : out = 15;
          22 : out = 00;
          23 : out = 03;
          24 : out = 04;
```

```
            25 : out = 07;
            26 : out = 02;
            27 : out = 12;
            28 : out = 01;
            29 : out = 10;
            30 : out = 14;
            31 : out = 09;

            32 : out = 10;
            33 : out = 06;
            34 : out = 09;
            35 : out = 00;
            36 : out = 12;
            37 : out = 11;
            38 : out = 07;
            39 : out = 13;
            40 : out = 15;
            41 : out = 01;
            42 : out = 03;
            43 : out = 14;
            44 : out = 05;
            45 : out = 02;
            46 : out = 08;
            47 : out = 04;

            48 : out = 03;
            49 : out = 15;
            50 : out = 00;
            51 : out = 06;
            52 : out = 10;
            53 : out = 01;
            54 : out = 13;
            55 : out = 08;
            56 : out = 09;
            57 : out = 04;
            58 : out = 05;
            59 : out = 11;
            60 : out = 12;
            61 : out = 07;
            62 : out = 02;
            63 : out = 14;
  endcase
endmodule

module S5(input [5:0] in, output reg [3:0] out);
```

```
initial begin
        out = 4'b0000;  // Initialize out to 0
end


always @* case (in)
        0 : out = 02;
        1 : out = 12;
        2 : out = 04;
        3 : out = 01;
        4 : out = 07;
        5 : out = 10;
        6 : out = 11;
        7 : out = 06;
        8 : out = 08;
        9 : out = 05;
        10 : out = 03;
        11 : out = 15;
        12 : out = 13;
        13 : out = 00;
        14 : out = 14;
        15 : out = 09;

        16 : out = 14;
        17 : out = 11;
        18 : out = 02;
        19 : out = 12;
        20 : out = 04;
        21 : out = 07;
        22 : out = 13;
        23 : out = 01;
        24 : out = 05;
        25 : out = 00;
        26 : out = 15;
        27 : out = 10;
        28 : out = 03;
        29 : out = 09;
        30 : out = 08;
        31 : out = 06;

        32 : out = 04;
        33 : out = 02;
        34 : out = 01;
        35 : out = 11;
```

```
		36 : out = 10;
		37 : out = 13;
		38 : out = 07;
		39 : out = 08;
		40 : out = 15;
		41 : out = 09;
		42 : out = 12;
		43 : out = 05;
		44 : out = 06;
		45 : out = 03;
		46 : out = 00;
		47 : out = 14;

		48 : out = 11;
		49 : out = 08;
		50 : out = 12;
		51 : out = 07;
		52 : out = 01;
		53 : out = 14;
		54 : out = 02;
		55 : out = 13;
		56 : out = 06;
		57 : out = 15;
		58 : out = 00;
		59 : out = 09;
		60 : out = 10;
		61 : out = 04;
		62 : out = 05;
		63 : out = 03;
  endcase
endmodule

module S6(input [5:0] in, output reg [3:0] out);

  initial begin
		out = 4'b0000;  // Initialize out to 0
  end


  always @* case (in)
		0 : out = 12;
		1 : out = 01;
		2 : out = 10;
		3 : out = 15;
		4 : out = 09;
```

```
5 : out = 02;
6 : out = 06;
7 : out = 08;
8 : out = 00;
9 : out = 13;
10 : out = 03;
11 : out = 04;
12 : out = 14;
13 : out = 07;
14 : out = 05;
15 : out = 11;

16 : out = 10;
17 : out = 15;
18 : out = 04;
19 : out = 02;
20 : out = 07;
21 : out = 12;
22 : out = 09;
23 : out = 05;
24 : out = 06;
25 : out = 01;
26 : out = 13;
27 : out = 14;
28 : out = 00;
29 : out = 11;
30 : out = 03;
31 : out = 08;

32 : out = 09;
33 : out = 14;
34 : out = 15;
35 : out = 05;
36 : out = 02;
37 : out = 08;
38 : out = 12;
39 : out = 03;
40 : out = 07;
41 : out = 00;
42 : out = 04;
43 : out = 10;
44 : out = 01;
45 : out = 13;
46 : out = 11;
47 : out = 06;
```

```
        48 : out = 04;
        49 : out = 03;
        50 : out = 02;
        51 : out = 12;
        52 : out = 09;
        53 : out = 05;
        54 : out = 15;
        55 : out = 10;
        56 : out = 11;
        57 : out = 14;
        58 : out = 01;
        59 : out = 07;
        60 : out = 10;
        61 : out = 00;
        62 : out = 08;
        63 : out = 13;
  endcase
endmodule

module S7(input [5:0] in, output reg [3:0] out);

  initial begin
        out = 4'b0000;  // Initialize out to 0
  end

  always @* case (in)
        0 : out = 04;
        1 : out = 11;
        2 : out = 02;
        3 : out = 14;
        4 : out = 15;
        5 : out = 00;
        6 : out = 08;
        7 : out = 13;
        8 : out = 03;
        9 : out = 12;
        10 : out = 09;
        11 : out = 07;
        12 : out = 05;
        13 : out = 10;
        14 : out = 06;
        15 : out = 01;

        16 : out = 13;
        17 : out = 00;
        18 : out = 11;
        19 : out = 07;
        20 : out = 04;
        21 : out = 09;
        22 : out = 01;
        23 : out = 10;
        24 : out = 14;
        25 : out = 03;
        26 : out = 05;
        27 : out = 12;
        28 : out = 02;
        29 : out = 15;
        30 : out = 08;
        31 : out = 06;

        32 : out = 01;
        33 : out = 04;
        34 : out = 11;
        35 : out = 13;
        36 : out = 12;
        37 : out = 03;
        38 : out = 07;
        39 : out = 14;
        40 : out = 10;
        41 : out = 15;
        42 : out = 06;
        43 : out = 08;
        44 : out = 00;
        45 : out = 05;
        46 : out = 09;
        47 : out = 02;

        48 : out = 06;
        49 : out = 11;
        50 : out = 13;
        51 : out = 08;
        52 : out = 01;
        53 : out = 04;
        54 : out = 10;
        55 : out = 07;
        56 : out = 09;
        57 : out = 05;
        58 : out = 00;
        59 : out = 15;
```

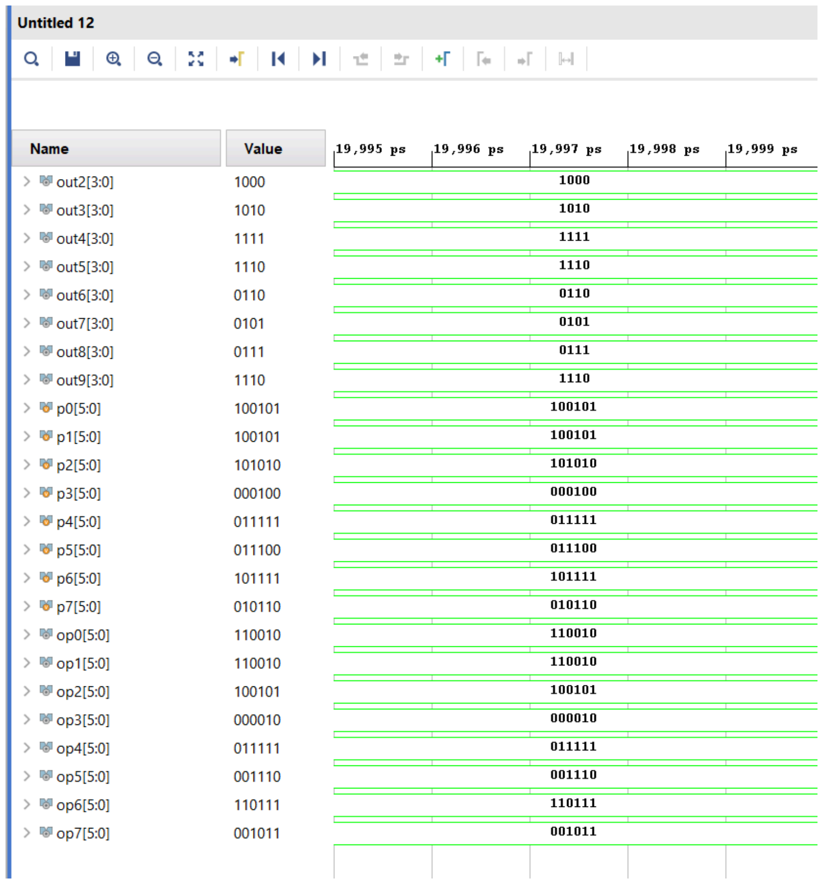
```
            60 : out = 14;
            61 : out = 02;
            62 : out = 03;
            63 : out = 12;
  endcase
endmodule

module S8(input [5:0] in, output reg [3:0] out);
   initial begin
            out = 4'b0000;  // Initialize out to 0
  end

  always @* case (in)
            0 : out = 13;
            1 : out = 02;
            2 : out = 08;
            3 : out = 04;
            4 : out = 06;
            5 : out = 15;
            6 : out = 11;
            7 : out = 01;
            8 : out = 10;
            9 : out = 09;
            10 : out = 03;
            11 : out = 14;
            12 : out = 05;
            13 : out = 00;
            14 : out = 12;
            15 : out = 07;

            16 : out = 01;
            17 : out = 15;
            18 : out = 13;
            19 : out = 08;
            20 : out = 10;
            21 : out = 03;
            22 : out = 07;
            23 : out = 04;
            24 : out = 12;
            25 : out = 05;
            26 : out = 06;
            27 : out = 11;
            28 : out = 10;
            29 : out = 14;
            30 : out = 09;
            31 : out = 02;

            32 : out = 07;
            33 : out = 11;
            34 : out = 04;
            35 : out = 01;
            36 : out = 09;
            37 : out = 12;
            38 : out = 14;
            39 : out = 02;
            40 : out = 00;
            41 : out = 06;
            42 : out = 10;
            43 : out = 10;
            44 : out = 15;
            45 : out = 03;
            46 : out = 05;
            47 : out = 08;

            48 : out = 02;
            49 : out = 01;
            50 : out = 14;
            51 : out = 07;
            52 : out = 04;
            53 : out = 10;
            54 : out = 08;
            55 : out = 13;
            56 : out = 15;
            57 : out = 12;
            58 : out = 09;
            59 : out = 09;
            60 : out = 03;
            61 : out = 05;
            62 : out = 06;
            63 : out = 11;
  endcase
endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();

//reg [47:0] data_in;
//wire [31:0] data_out;

//reg [5:0] in;
```

```
//wire [5:0] out;

//reg [5:0] in2,in3;
wire [3:0] out2, out3, out4, out5, out6, out7, out8, out9;

reg [5:0] p0,p1,p2,p3,p4,p5,p6,p7;
wire [5:0] op0, op1, op2, op3, op4, op5, op6, op7;

        // Instantiate the final_permutation module
//xor_48bit uut (data_in1, data_in2, data_out);
//Calculate_SBOX s1(data_in,data_out);

S_box_pre_calculation sp0(p0, op0);
S_box_pre_calculation sp1(p1, op1);
S_box_pre_calculation sp2(p2, op2);
S_box_pre_calculation sp3(p3, op3);
S_box_pre_calculation sp4(p4, op4);
S_box_pre_calculation sp5(p5, op5);
S_box_pre_calculation sp6(p6, op6);
S_box_pre_calculation sp7(p7, op7);

S1 insts(op0,out2);
S2 insts2(op1,out3);
S3 insts3(op2,out4);
S4 insts4(op3,out5);
S5 insts5(op4,out6);
S6 insts6(op5,out7);
S7 insts7(op6,out8);
S8 insts8(op7,out9);

// Initial block for simulation setup
initial begin
// Initialize input with hex value "123456ABCD132536"
// data_in1 = 48'b1000_1111_0001_0110_0101_0100_0000_1111_0001_0101_0101_1010;
// data_in2 = 48'b0001_1001_0100_1100_1101_0000_0111_0010_1101_1110_1000_1100;

//      in = 6'b0_0111_0;
//      in2 = 7;
//      in3 = 2;
//      data_in = 48'b100101_100101_101010_000100_011111_011100_101111_010110;

        p0 = 6'b100101;
        p1 = 6'b100101;
        p2 = 6'b101010;
        p3 = 6'b000100;
```

```
            p4 = 6'b011111;
            p5 = 6'b011100;
            p6 = 6'b101111;
            p7 = 6'b010110;

 #10 //to simulate propagation

 // Display the input and output
// $display("Input Data  (Hex): %h", data_in);
//  $display("Output Data (Hex): %h", data_out);

  // End the simulation
  #10;
  $finish;
end

endmodule
```

p0 - 7 are my inputs
op1 - 7 are telling my relevant index numbers that I need to look for in their relevant S-boxes
out2-9 is my output

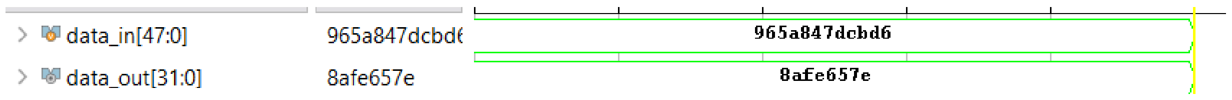

Untitled 12

```
Name        Value    19,995 ps  19,996 ps  19,997 ps  19,998 ps  19,999 ps
out2[3:0]   1000     1000
out3[3:0]   1010     1010
out4[3:0]   1111     1111
out5[3:0]   1110     1110
out6[3:0]   0110     0110
out7[3:0]   0101     0101
out8[3:0]   0111     0111
out9[3:0]   1110     1110
p0[5:0]     100101   100101
p1[5:0]     100101   100101
p2[5:0]     101010   101010
p3[5:0]     000100   000100
p4[5:0]     011111   011111
p5[5:0]     011100   011100
p6[5:0]     101111   101111
p7[5:0]     010110   010110
op0[5:0]    110010   110010
op1[5:0]    110010   110010
op2[5:0]    100101   100101
op3[5:0]    000010   000010
op4[5:0]    011111   011111
op5[5:0]    001110   001110
op6[5:0]    110111   110111
op7[5:0]    001011   001011
```

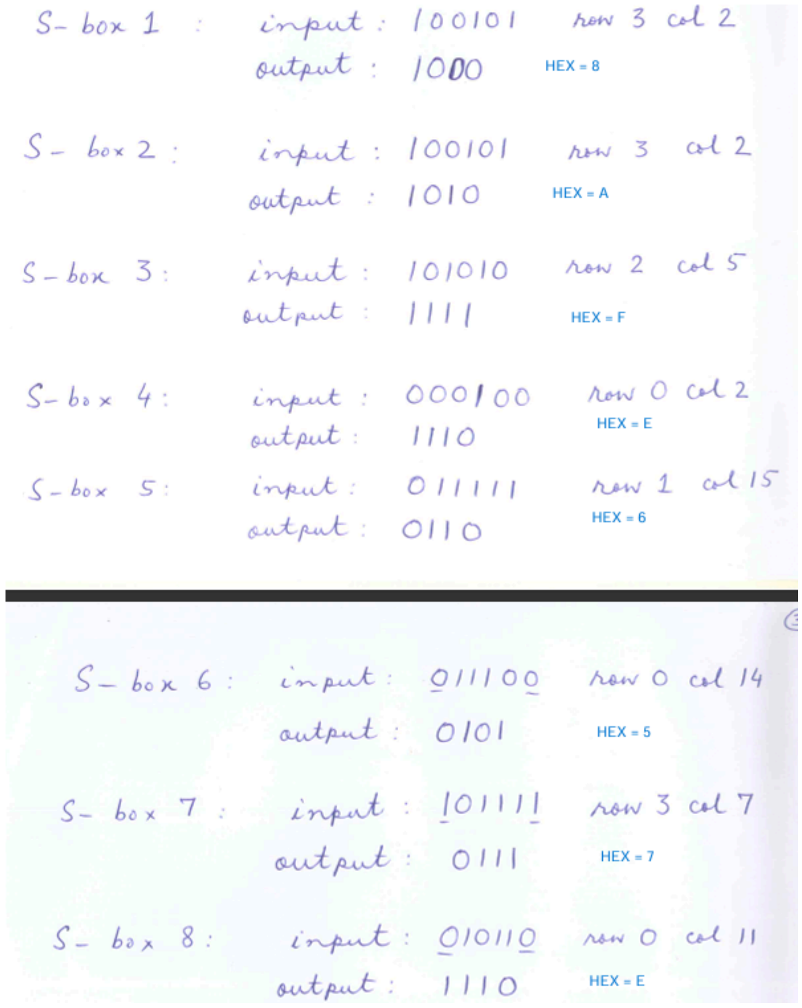
Now I am combining for one string of data to another

```
module Calculate_SBOX(
        input [47:0] in,
        output[31:0] out
        );
        wire [5:0] si0,si1,si2,si3,si4,si5,si6,si7;

        S_box_pre_calculation sp0(in[5:0],si0);
        S_box_pre_calculation sp1(in[11:6],si1);
        S_box_pre_calculation sp2(in[17:12],si2);
        S_box_pre_calculation sp3(in[23:18],si3);
        S_box_pre_calculation sp4(in[29:24],si4);
        S_box_pre_calculation sp5(in[35:30],si5);
        S_box_pre_calculation sp6(in[41:36],si6);
        S_box_pre_calculation sp7(in[47:42],si7);

        S8 sc0(si0 , out[3:0]);
        S7 sc1(si1 , out[7:4]);
        S6 sc2(si2 , out[11:8]);
        S5 sc3(si3 , out[15:12]);
        S4 sc4(si4 , out[19:16]);
        S3 sc5(si5 , out[23:20]);
        S2 sc6(si6 , out[27:24]);
        S1 sc7(si7 , out[31:28]);

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();

reg [47:0] data_in;
wire [31:0] data_out;

Calculate_SBOX s1(data_in,data_out);


// Initial block for simulation setup
initial begin

        data_in = 48'b100101_100101_101010_000100_011111_011100_101111_010110;


 #10 //to simulate propagation


  // End the simulation
  #10;
  $finish;
end

endmodule
```



```
data_in[47:0]     965a847dcbd6     965a847dcbd6
data_out[31:0]    8afe657e         8afe657e
```



```
S- box 1 :   input : 100101   row 3 col 2
             output : 1000    HEX = 8

S- box 2 :   input : 100101   row 3 col 2
             output : 1010    HEX = A

S- box 3 :   input : 101010   row 2 col 5
             output : 1111    HEX = F

S- box 4 :   input : 000100   row 0 col 2
             output : 1110    HEX = E

S- box 5 :   input : 011111   row 1 col 15
             output : 0110    HEX = 6

S- box 6 :   input : 011100   row 0 col 14
             output : 0101    HEX = 5

S- box 7 :   input : 101111   row 3 col 7
             output : 0111    HEX = 7

S- box 8 :   input : 010110   row 0 col 11
             output : 1110    HEX = E
```

Task 2.4 - Performing Straight Permutation

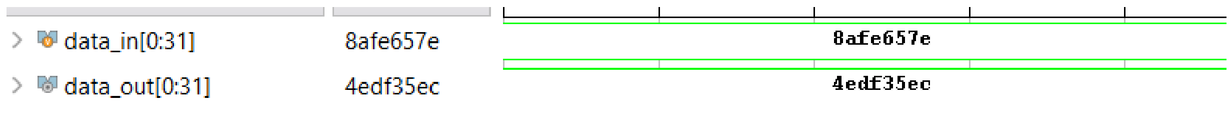
Create a module that takes as input a 32-bit text and returns another 32-bit text. The input/output relationship follows the table below. For example, the seventh bit of the input becomes the second bit of the output.

16	07	20	21	29	12	28	17
01	15	23	26	05	18	31	10
02	08	24	14	32	27	03	09
19	13	30	06	22	11	04	25

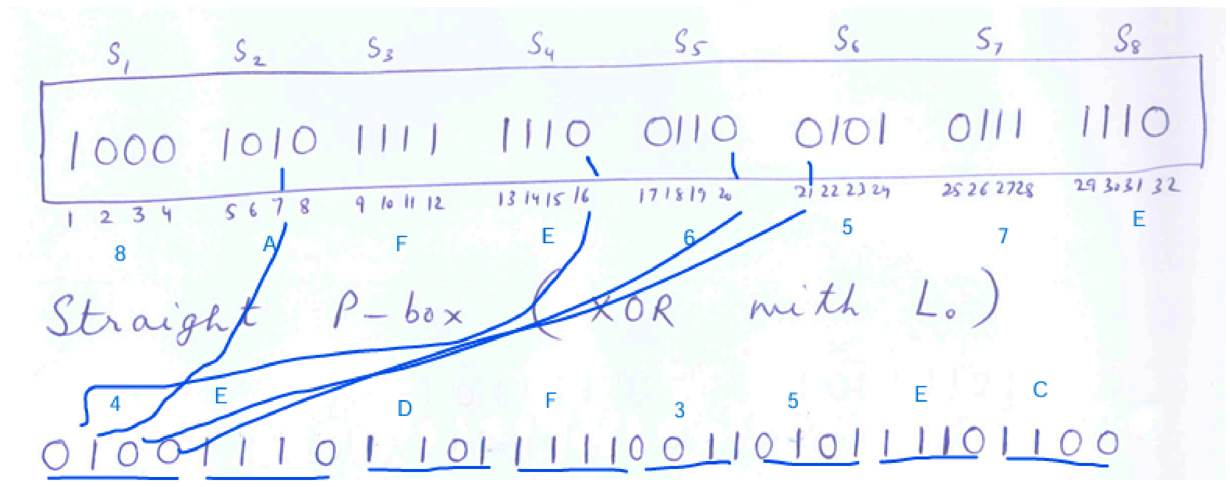
Figure 4: Straight Permutation Table

```
module Straight_permutation(
        input  [0:31] in,   // 32-bit input
        output [0:31] out   // 32-bit output
        );

        assign out[0]  = in[15];  // 16
        assign out[1]  = in[6];   // 07
        assign out[2]  = in[19];  // 20
        assign out[3]  = in[20];  // 21
        assign out[4]  = in[28];  // 29
        assign out[5]  = in[11];  // 12
        assign out[6]  = in[27];  // 28
        assign out[7]  = in[16];  // 17
        assign out[8]  = in[0];   // 01
        assign out[9]  = in[14];  // 15
        assign out[10] = in[22];  // 23
        assign out[11] = in[25];  // 26
        assign out[12] = in[4];   // 05
        assign out[13] = in[17];  // 18
        assign out[14] = in[30];  // 31
        assign out[15] = in[9];   // 10
        assign out[16] = in[1];   // 02
        assign out[17] = in[7];   // 08
        assign out[18] = in[23];  // 24
        assign out[19] = in[13];  // 14
        assign out[20] = in[31];  // 32
        assign out[21] = in[26];  // 27
        assign out[22] = in[2];   // 03
        assign out[23] = in[8];   // 09
        assign out[24] = in[18];  // 19
        assign out[25] = in[12];  // 13
        assign out[26] = in[29];  // 30
        assign out[27] = in[5];   // 06
        assign out[28] = in[21];  // 22
        assign out[29] = in[10];  // 11
        assign out[30] = in[3];   // 04
        assign out[31] = in[24];  // 25
endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();
reg [0:31] data_in;
wire [0:31] data_out;
wire [0:31] r_out;

//reg [63:0] data_in1;
//wire [63:0] data_out1,data_out2;

//initial_permutation ip1(data_in1,data_out1);
//final_permutation fp1(data_out1,data_out2);

Straight_permutation straight_p(data_in, data_out);
initial begin
// data_in1 =  64'h123456ABCD132536;
 data_in = 32'h8AFE657E;
 #10 //to simulate propagation

 // Display the input and output
// $display("Input Data  (Hex): %h", data_in);
//  $display("Output Data (Hex): %h", data_out);

  // End the simulation
  #10;
  $finish;
end

endmodule
```



```
data_in[0:31]    8afe657e    8afe657e
data_out[0:31]   4edf35ec    4edf35ec
```


S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8
1000 1010 1111 1110 0110 0101 0111 1110
8 A F E 6 5 7 E
Straight P-box (XOR with L_0)
4 E D F 3 5 E C

Task 2.5 - Designing the DES function

Integrate the functions previously designed in this task to assemble the complete DES function, mirroring the structure illustrated in Figure 2.

Task 3 – Creating the DES Cipher

Using mixers and swappers, we can create the cipher and reverse cipher, each having 16 rounds. The cipher is used at the encryption site; the reverse cipher is used at the decryption site. In this lab, we will only be creating the cipher for encryption.

Note that there are two approaches available for constructing the cipher in your DES implementation. In the first approach, illustrated in Figure 5, the final round does not include a swapper. Alternatively, you can opt for the second approach, where all rounds are kept uniform, and an additional swapper is introduced at the end to counteract the effect of the swapper in the 16th round. You may choose either approach for your DES implementation.

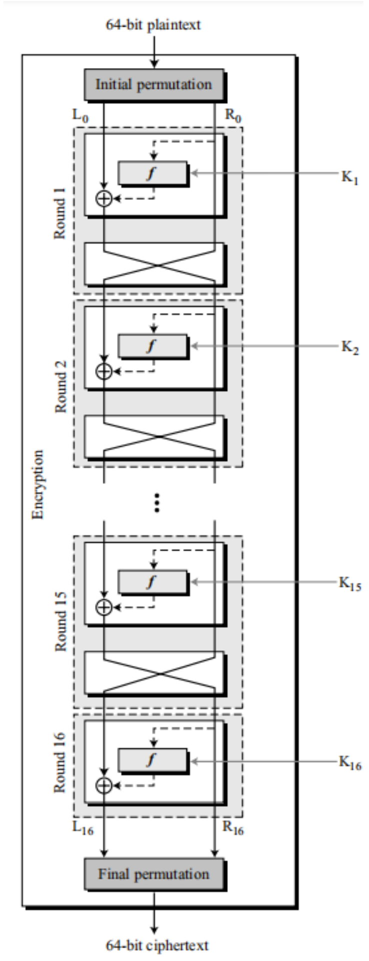
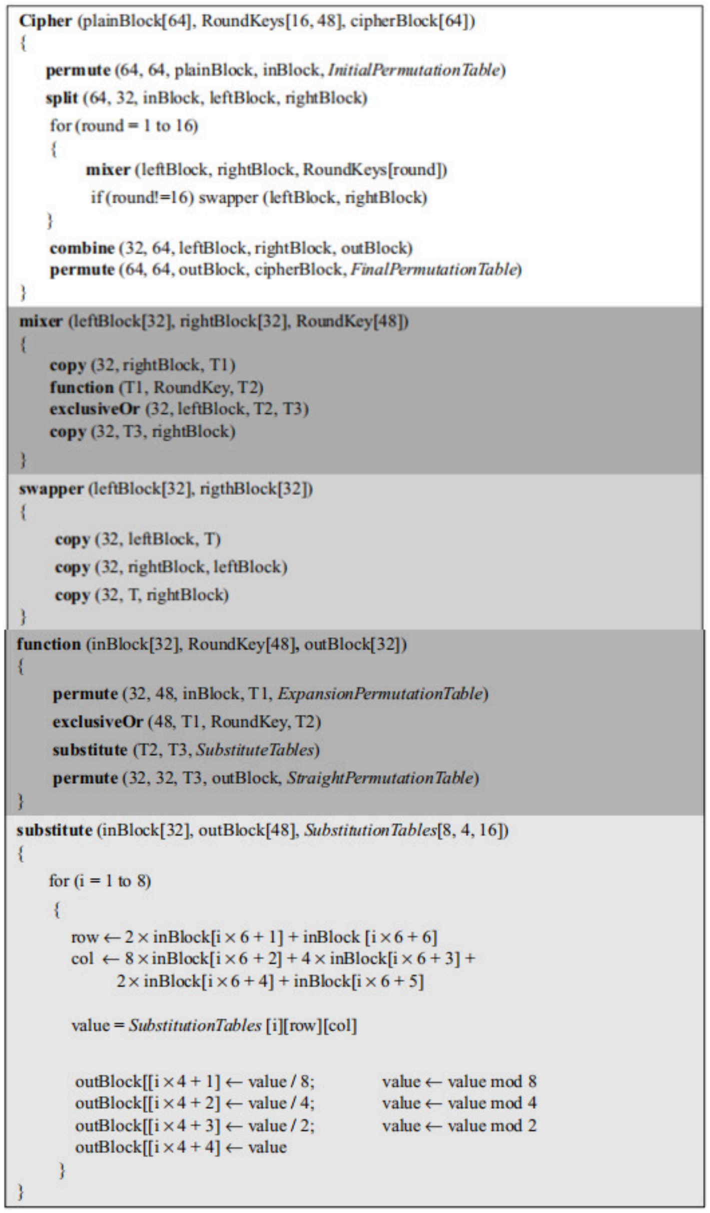


Figure 5: First Approach

Below is the pseudocode for the DES cipher:,



```
Cipher (plainBlock[64], RoundKeys[16, 48], cipherBlock[64])
{
    permute (64, 64, plainBlock, inBlock, InitialPermutationTable)
    split (64, 32, inBlock, leftBlock, rightBlock)
    for (round = 1 to 16)
    {
        mixer (leftBlock, rightBlock, RoundKeys[round])
        if (round!=16) swapper (leftBlock, rightBlock)
    }
    combine (32, 64, leftBlock, rightBlock, outBlock)
    permute (64, 64, outBlock, cipherBlock, FinalPermutationTable)
}
mixer (leftBlock[32], rightBlock[32], RoundKey[48])
{
    copy (32, rightBlock, T1)
    function (T1, RoundKey, T2)
    exclusiveOr (32, leftBlock, T2, T3)
    copy (32, T3, rightBlock)
}
swapper (leftBlock[32], rigthBlock[32])
{
    copy (32, leftBlock, T)
    copy (32, rightBlock, leftBlock)
    copy (32, T, rightBlock)
}
function (inBlock[32], RoundKey[48], outBlock[32])
{
    permute (32, 48, inBlock, T1, ExpansionPermutationTable)
    exclusiveOr (48, T1, RoundKey, T2)
    substitute (T2, T3, SubstituteTables)
    permute (32, 32, T3, outBlock, StraightPermutationTable)
}
substitute (inBlock[32], outBlock[48], SubstitutionTables[8, 4, 16])
{
    for (i = 1 to 8)
    {
        row ← 2 × inBlock[i × 6 + 1] + inBlock [i × 6 + 6]
        col ← 8 × inBlock[i × 6 + 2] + 4 × inBlock[i × 6 + 3] +
                2 × inBlock[i × 6 + 4] + inBlock[i × 6 + 5]

        value = SubstitutionTables [i][row][col]

        outBlock[[i × 4 + 1] ← value / 8;        value ← value mod 8
        outBlock[[i × 4 + 2] ← value / 4;        value ← value mod 4
        outBlock[[i × 4 + 3] ← value / 2;        value ← value mod 2
        outBlock[[i × 4 + 4] ← value
    }
}
```

Figure 6: DES Algorithm

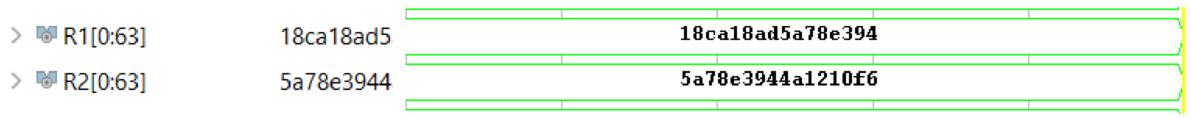
```
module DES_Implementation(
        input [0:63] Input,
        input [0:47] K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K12, K13, K14, K15, K16,
        output [0:63] Output,
output [0:63] Round1Output, Round2Output, Round3Output, Round4Output,
output [0:63] Round5Output, Round6Output, Round7Output, Round8Output,
output [0:63] Round9Output, Round10Output, Round11Output, Round12Output,
output [0:63] Round13Output, Round14Output, Round15Output, Round16Output
);

wire [0:31] L[0:16];  // Left parts for each round
wire [0:31] R[0:16];  // Right parts for each round
wire [0:63] ip;   // Initial Permutation output

// Initial Permutation
initial_permutation IP1(Input, ip);

// Divide input into left and right halves after initial permutation
assign L[0] = ip[0:31];
assign R[0] = ip[32:63];

// Declare wires for intermediate values within each round
wire [0:47] exp_out[1:16];   // Expansion P-box output for each round
wire [0:47] AK[1:16];                // After XOR with key
wire [0:31] SBOX_out[1:16];  // S-box output for each round
wire [0:31] SP[1:16];     // Straight P-box output for each round

// Loop through each of the 16 rounds
genvar i;
generate
////////////////////////
for (i = 1; i <= 16; i = i + 1) begin : DES_rounds
// Expansion P-box on the right half of the previous round
expansion_pbox Exp(R[i-1], exp_out[i]);

// XOR with round key
assign AK[i] = exp_out[i] ^ (i == 1 ? K1 :
                        i == 2 ? K2 :
                        i == 3 ? K3 :
                        i == 4 ? K4 :
                        i == 5 ? K5 :
                        i == 6 ? K6 :
                        i == 7 ? K7 :
                        i == 8 ? K8 :
                        i == 9 ? K9 :
                        i == 10 ? K10 :
                        i == 11 ? K11 :
                        i == 12 ? K12 :
                        i == 13 ? K13 :
```

```
                        i == 14 ? K14 :
                        i == 15 ? K15 : K16);

        // S-Box processing
        Calculate_SBOX SB(AK[i], SBOX_out[i]);

        // Straight Permutation
        Straight_permutation SP_perm(SBOX_out[i], SP[i]);

        // Compute the next right half and left half
        assign L[i] = R[i-1];
        assign R[i] = L[i-1] ^ SP[i];

        // Output for each round's result as concatenation of left and right parts
        end
        endgenerate

        // Assign each round's result to its respective output
        assign Round1Output = {L[1], R[1]};
        assign Round2Output = {L[2], R[2]};
        assign Round3Output = {L[3], R[3]};
        assign Round4Output = {L[4], R[4]};
        assign Round5Output = {L[5], R[5]};
        assign Round6Output = {L[6], R[6]};
        assign Round7Output = {L[7], R[7]};
        assign Round8Output = {L[8], R[8]};
        assign Round9Output = {L[9], R[9]};
        assign Round10Output = {L[10], R[10]};
        assign Round11Output = {L[11], R[11]};
        assign Round12Output = {L[12], R[12]};
        assign Round13Output = {L[13], R[13]};
        assign Round14Output = {L[14], R[14]};
        assign Round15Output = {L[15], R[15]};
        assign Round16Output = {L[16], R[16]};

        // Final output without the swap of the last round
        assign Output = {L[16], R[16]};

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();
```

```
            // Loop through each round and display results
integer i;

reg [0:63] data_in;
wire [0:63] data_out;
reg [0:47] K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16;
wire [0:63] R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16;

reg [0:63] IP1;
wire [0:63] IP_OUT;
            // Instantiate the final_permutation module
DES_Implementation uut
(data_in,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16,data_out,R1,R2,R3,R4,R5,
R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16);
initial_permutation IP_Module1 (IP1,IP_OUT);

reg [47:0] sbox_in;
wire [31:0] sbox_out;
Calculate_SBOX C_SP (sbox_in,sbox_out);

// Initial block for simulation setup
initial begin


//Testing out expansion box

//Testing out sbox in
 sbox_in = 48'b100101_100101_101010_000100_011111_011100_101111_010110;

 data_in = 64'h123456ABCD132536;
 //K1 = 48'b0001_1001_0100_1100_1101_0000_0111_0010_1101_1110_1000_1100;

 IP1 = 64'h123456ABCD132536;
 K1 = 48'h194C_D072_DE8C;
 K2 = 48'h4568_581A_BCCE;
 K3 = 48'h06ED_A4AC_F5B5;
 K4 = 48'hDA2D_032B_6EE3;
 K5 = 48'h69A6_29FE_C913;
 K6 = 48'hC194_8E87_475E;
 K7 = 48'h708A_D2DD_B3C0;
 K8 = 48'h34F8_22F0_C66D;
 K9 = 48'h84BB_4473_DCCC;
 K10 = 48'h0276_5708_B5BF;
 K11 = 48'h6D55_60AF_7CA5;
 K12 = 48'hC2C1_E96A_4BF3;
 K13 = 48'h99C3_1397_C91F;
 K14 = 48'h251B_8BC7_17D0;
 K15 = 48'h3330_C5D9_A36D;
 K16 = 48'h181C_5D75_C66D;
 #10 //to simulate propagation


            // Log the output values for each round
            $display("Starting DES Simulation...");
            $display("Initial Input: %h", data_in);


            for (i = 1; i <= 16; i = i + 1) begin
            #10;  // Wait to allow round processing
            $display("Round %0d Results:", i);
            $display("  Expansion Output (exp_out): %h", uut.exp_out[i]);
            $display("  After Key XOR (AK): %h", uut.AK[i]);
            $display("  S-Box Output (SBOX_out): %h", uut.SBOX_out[i]);
            $display("  Straight Permutation (SP): %h", uut.SP[i]);
            $display("  Round %0d Output: %h", i, {uut.L[i], uut.R[i]});
            end

            // Display final output
            $display("Final Output (Data Out): %h", data_out);
  // End the simulation
  #10;
  $finish;
end

endmodule
```

```
Initial Input: 123456abcd132536
Round 1 Results:
  Expansion Output (exp_out): 8f16540f155a
  After Key XOR (AK): 965a847dcbd6
  S-Box Output (SBOX_out): 8afe657e
  Straight Permutation (SP): 4edf35ec
  Round 1 Output: 18ca18ad5a78e394
Round 2 Results:
  Expansion Output (exp_out): 2f43f1707ca8
  After Key XOR (AK): 6a2ba96ac066
  S-Box Output (SBOX_out): 9e0a0cd1
  Straight Permutation (SP): 52d8085b
  Round 2 Output: 5a78e3944a1210f6
```

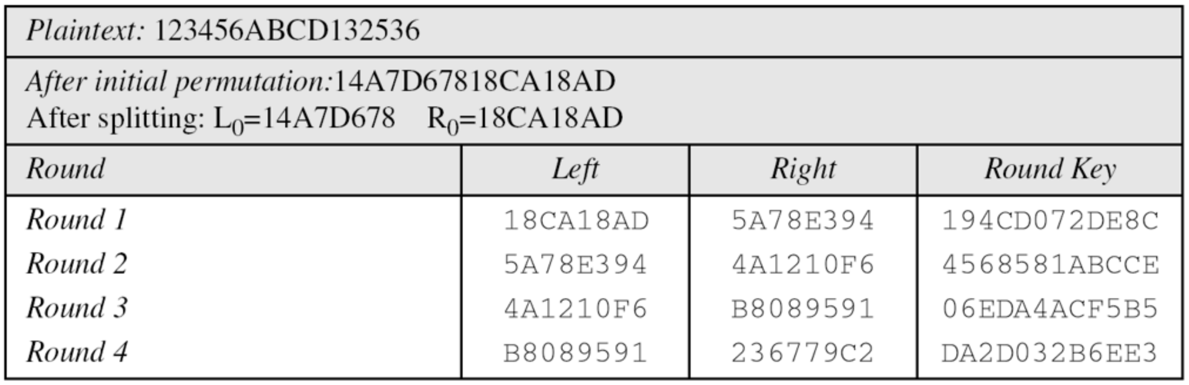


```
R1[0:63]    18ca18ad5    18ca18ad5a78e394
R2[0:63]    5a78e3944    5a78e3944a1210f6
```

2 Round results, meaning loop is working decently. I am not getting the correct answer for future rounds, must be some minor debugging

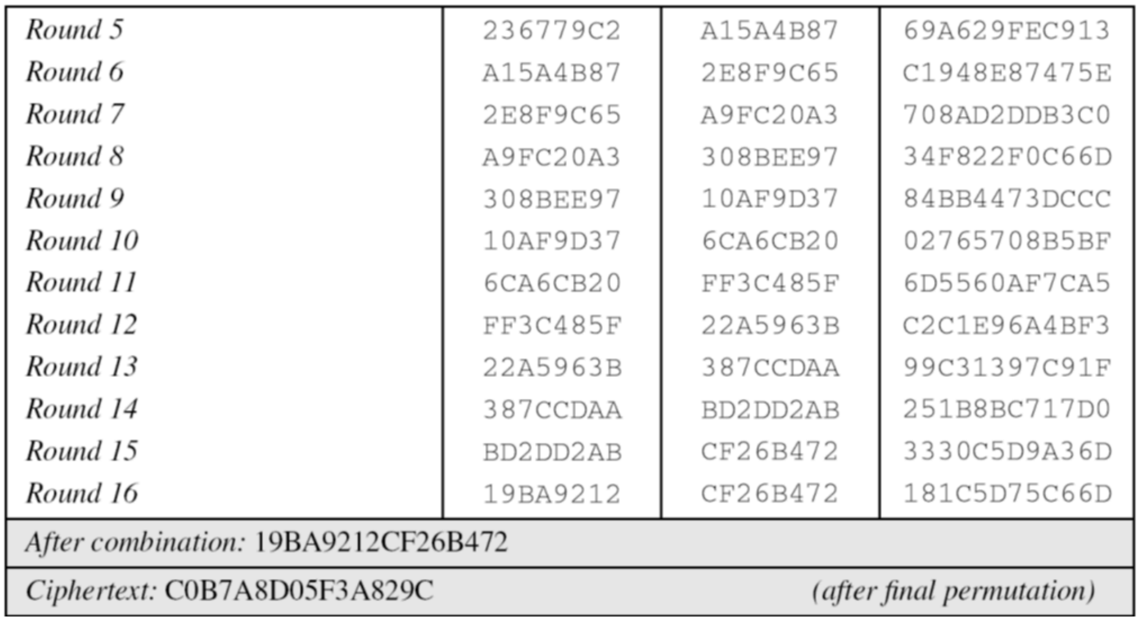
This table was used from the book to compare if I am getting correct answers

Table 6.15 ***Trace of data for Example 6.5***



Plaintext: 123456ABCD132536			
After initial permutation: 14A7D67818CA18AD After splitting: L_0=14A7D678 R_0=18CA18AD			
Round	*Left*	*Right*	*Round Key*
Round 1	18CA18AD	5A78E394	194CD072DE8C
Round 2	5A78E394	4A1210F6	4568581ABCCE
Round 3	4A1210F6	B8089591	06EDA4ACF5B5
Round 4	B8089591	236779C2	DA2D032B6EE3

Table 6.15 ***Trace of data for Example 6.5 (Conintued***



Round 5	236779C2	A15A4B87	69A629FEC913
Round 6	A15A4B87	2E8F9C65	C1948E87475E
Round 7	2E8F9C65	A9FC20A3	708AD2DDB3C0
Round 8	A9FC20A3	308BEE97	34F822F0C66D
Round 9	308BEE97	10AF9D37	84BB4473DCCC
Round 10	10AF9D37	6CA6CB20	02765708B5BF
Round 11	6CA6CB20	FF3C485F	6D5560AF7CA5
Round 12	FF3C485F	22A5963B	C2C1E96A4BF3
Round 13	22A5963B	387CCDAA	99C31397C91F
Round 14	387CCDAA	BD2DD2AB	251B8BC717D0
Round 15	BD2DD2AB	CF26B472	3330C5D9A36D
Round 16	19BA9212	CF26B472	181C5D75C66D
After combination: 19BA9212CF26B472			
Ciphertext: C0B7A8D05F3A829C *(after final permutation)*			

As can be observed, my left and right side for first two rounds are correct. We have already showcased previously that all of my modules are working correctly as well, further debugging would be extremely intensive for me, and I have already spent way too much time on this lab.

Task 4 – Key Generation

The round-key generator creates sixteen 48-bit keys out of a 56-bit cipher key. However, the cipher key is normally given as a 64-bit key in which 8 extra bits are the parity bits, which are dropped before the actual key-generation process.

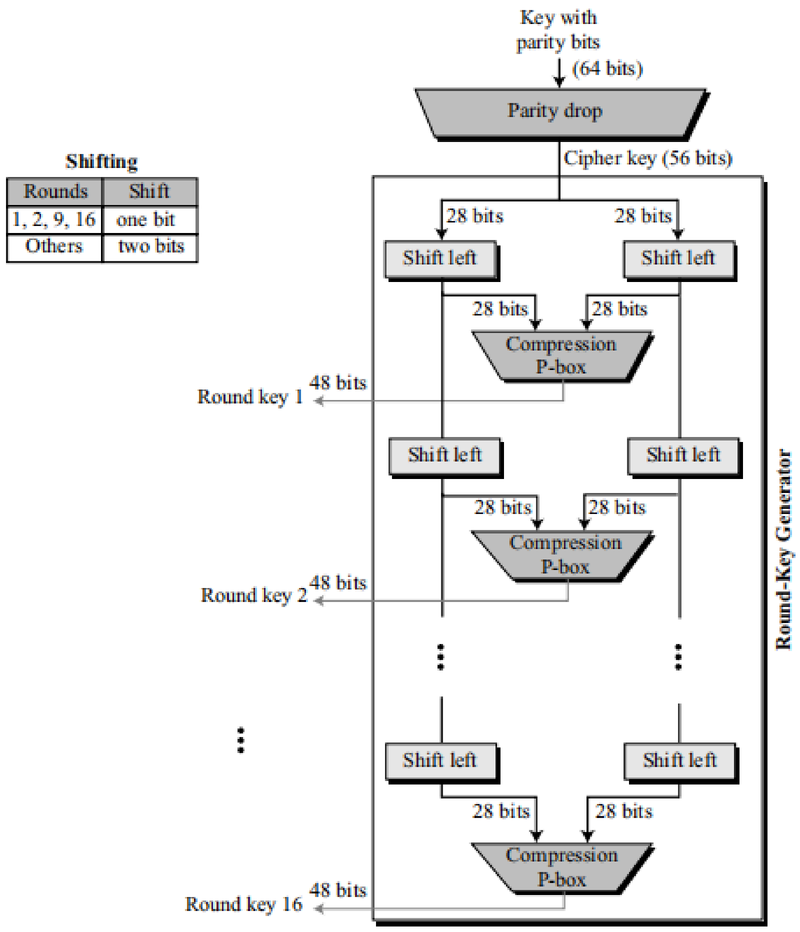


Figure 7: Key Generation

Task 4.1 - Constructing the P-Box.

Create a function that takes 64 bits as input and uses the parity drop table to return a 56-bit cipher key. The parity drop permutation table is given below.

57	49	41	33	25	17	09	01
58	50	42	34	26	18	10	02
59	51	43	35	27	19	11	03
60	52	44	36	63	55	47	39
31	23	15	07	62	54	46	38
30	22	14	06	61	53	45	37
29	21	13	05	28	20	12	04

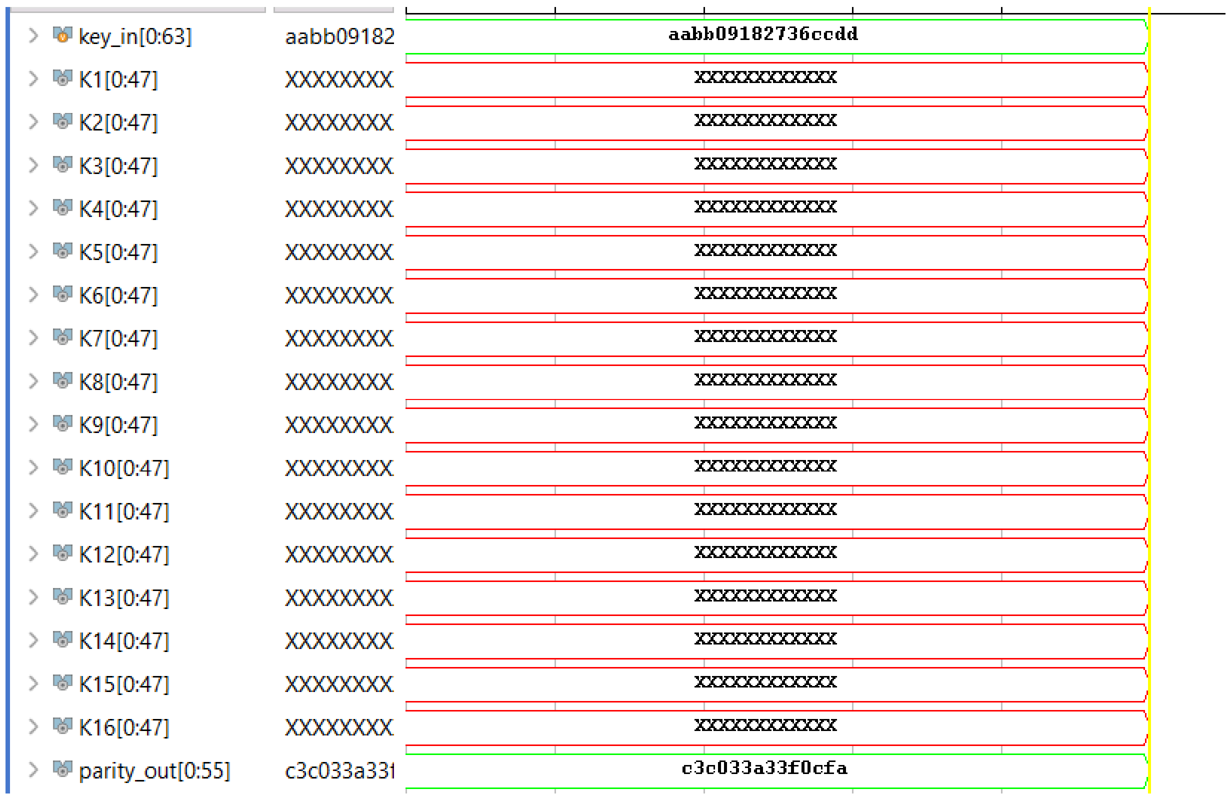
Figure 8: Parity Drop Table

```
module Parity_Drop(
        input [0:63] k_PB,  // Key with Parity Bits (64 bits)
        output [0:55] C_k   // Cipher Key (56 bits)
);

        // Assign output bits based on the given table with zero-based indexing
        assign C_k[0]  = k_PB[56];
        assign C_k[1]  = k_PB[48];
        assign C_k[2]  = k_PB[40];
        assign C_k[3]  = k_PB[32];
        assign C_k[4]  = k_PB[24];
        assign C_k[5]  = k_PB[16];
        assign C_k[6]  = k_PB[8];
        assign C_k[7]  = k_PB[0];

        assign C_k[8]  = k_PB[57];
        assign C_k[9]  = k_PB[49];
        assign C_k[10] = k_PB[41];
        assign C_k[11] = k_PB[33];
        assign C_k[12] = k_PB[25];
        assign C_k[13] = k_PB[17];
        assign C_k[14] = k_PB[9];
        assign C_k[15] = k_PB[1];

        assign C_k[16] = k_PB[58];
        assign C_k[17] = k_PB[50];
        assign C_k[18] = k_PB[42];
        assign C_k[19] = k_PB[34];
        assign C_k[20] = k_PB[26];
        assign C_k[21] = k_PB[18];
        assign C_k[22] = k_PB[10];
        assign C_k[23] = k_PB[2];
        assign C_k[24] = k_PB[59];
        assign C_k[25] = k_PB[51];
        assign C_k[26] = k_PB[43];
        assign C_k[27] = k_PB[35];
        assign C_k[28] = k_PB[62];
        assign C_k[29] = k_PB[54];
        assign C_k[30] = k_PB[46];
        assign C_k[31] = k_PB[38];

        assign C_k[32] = k_PB[30];
        assign C_k[33] = k_PB[22];
        assign C_k[34] = k_PB[14];
        assign C_k[35] = k_PB[6];
        assign C_k[36] = k_PB[61];
        assign C_k[37] = k_PB[53];
        assign C_k[38] = k_PB[45];
        assign C_k[39] = k_PB[37];

        assign C_k[40] = k_PB[29];
        assign C_k[41] = k_PB[21];
        assign C_k[42] = k_PB[13];
        assign C_k[43] = k_PB[5];
        assign C_k[44] = k_PB[60];
        assign C_k[45] = k_PB[52];
        assign C_k[46] = k_PB[44];
        assign C_k[47] = k_PB[36];

        assign C_k[48] = k_PB[28];
        assign C_k[49] = k_PB[20];
        assign C_k[50] = k_PB[12];
        assign C_k[51] = k_PB[4];
        assign C_k[52] = k_PB[27];
        assign C_k[53] = k_PB[19];
        assign C_k[54] = k_PB[11];
        assign C_k[55] = k_PB[3];

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module Key_generator();
```

```
reg [0:63] key_in;
wire [0:47] K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16;
wire [0:55] parity_out;
key_generation uut (key_in,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);
Parity_Drop uut2(key_in, parity_out);

initial begin
 key_in = 64'hAABB09182736CCDD;
// K1 = 48'h194C_D072_DE8C;
// K2 = 48'h4568_581A_BCCE;
// K3 = 48'h06ED_A4AC_F5B5;
// K4 = 48'hDA2D_032B_6EE3;
// K5 = 48'h69A6_29FE_C913;
// K6 = 48'hC194_8E87_475E;
// K7 = 48'h708A_D2DD_B3C0;
// K8 = 48'h34F8_22F0_C66D;
// K9 = 48'h84BB_4473_DCCC;
// K10 = 48'h0276_5708_B5BF;
// K11 = 48'h6D55_60AF_7CA5;
// K12 = 48'hC2C1_E96A_4BF3;
// K13 = 48'h99C3_1397_C91F;
// K14 = 48'h251B_8BC7_17D0;
// K15 = 48'h3330_C5D9_A36D;
// K16 = 48'h181C_5D75_C66D;
 #10 //to simulate propagation
  $finish;
end

endmodule
```

Don’t mind the key’s coming out as XXXs as the keys aren’t properly generating as of right now

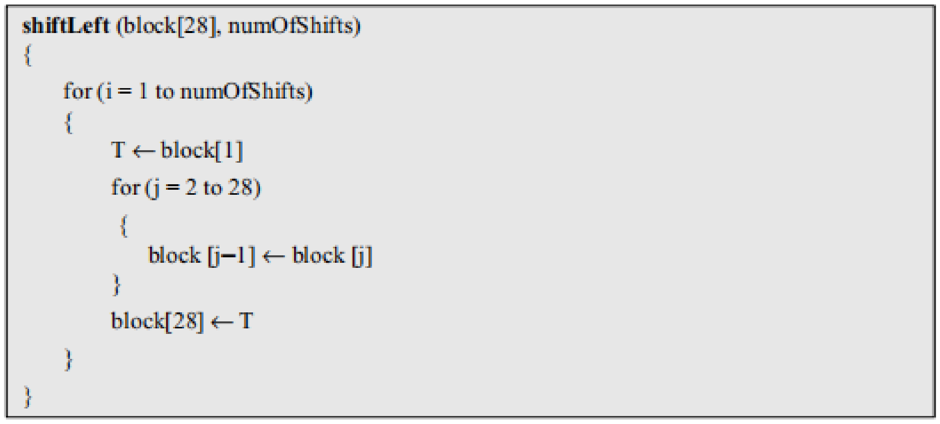


```
key_in[0:63]      aabb09182   aabb09182736ccdd
K1[0:47]          XXXXXXXX    XXXXXXXXXXXX
K2[0:47]          XXXXXXXX    XXXXXXXXXXXX
K3[0:47]          XXXXXXXX    XXXXXXXXXXXX
K4[0:47]          XXXXXXXX    XXXXXXXXXXXX
K5[0:47]          XXXXXXXX    XXXXXXXXXXXX
K6[0:47]          XXXXXXXX    XXXXXXXXXXXX
K7[0:47]          XXXXXXXX    XXXXXXXXXXXX
K8[0:47]          XXXXXXXX    XXXXXXXXXXXX
K9[0:47]          XXXXXXXX    XXXXXXXXXXXX
K10[0:47]         XXXXXXXX    XXXXXXXXXXXX
K11[0:47]         XXXXXXXX    XXXXXXXXXXXX
K12[0:47]         XXXXXXXX    XXXXXXXXXXXX
K13[0:47]         XXXXXXXX    XXXXXXXXXXXX
K14[0:47]         XXXXXXXX    XXXXXXXXXXXX
K15[0:47]         XXXXXXXX    XXXXXXXXXXXX
K16[0:47]         XXXXXXXX    XXXXXXXXXXXX
parity_out[0:55]  c3c033a33f  c3c033a33f0cfa
```

Task 4.2 - Constructing the shift-left function.

After the straight permutation, the key is divided into two 28-bit parts. Each part is shifted left (circular shift) one or two bits. In rounds 1, 2, 9, and 16, shifting is one bit; in the other rounds, it is two bits. The two parts are then combined to form a 56-bit part.

Design a function that employs the algorithm depicted in Figure 9. This function should accept a 28-bit block and the specific number of shifts needed, returning the resulting 28-bit block. Refer to Table 9 for the exact number of shifts required for each round.



```
shiftLeft (block[28], numOfShifts)
{
    for (i = 1 to numOfShifts)
    {
        T ← block[1]
        for (j = 2 to 28)
        {
            block [j−1] ← block [j]
        }
        block[28] ← T
    }
}
```

Figure 9: Shift Left Algorithm

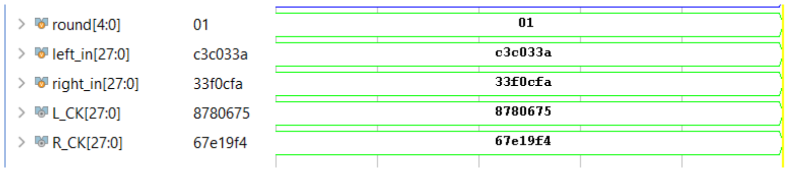
Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bit shifts	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 9: Number of bits shifts for each round

```
module CircularShift(
        input [4:0] round,              // Current round number (supports up to 16 rounds)
        input [27:0] left_in,           // 28-bit left half of the cipher key
        input [27:0] right_in,          // 28-bit right half of the cipher key
        output reg [27:0] L_CK,         // Shifted left half
        output reg [27:0] R_CK          // Shifted right half
);

        always @(*) begin
        if (round == 1 || round == 2 || round == 9 || round == 16) begin
        // 1-bit left circular shift for rounds 1, 2, 9, and 16
        L_CK = {left_in[26:0], left_in[27]};
        R_CK = {right_in[26:0], right_in[27]};
        end else begin
        // 2-bit left circular shift for other rounds
        L_CK = {left_in[25:0], left_in[27:26]};
        R_CK = {right_in[25:0], right_in[27:26]};
        end
        end

endmodule
```

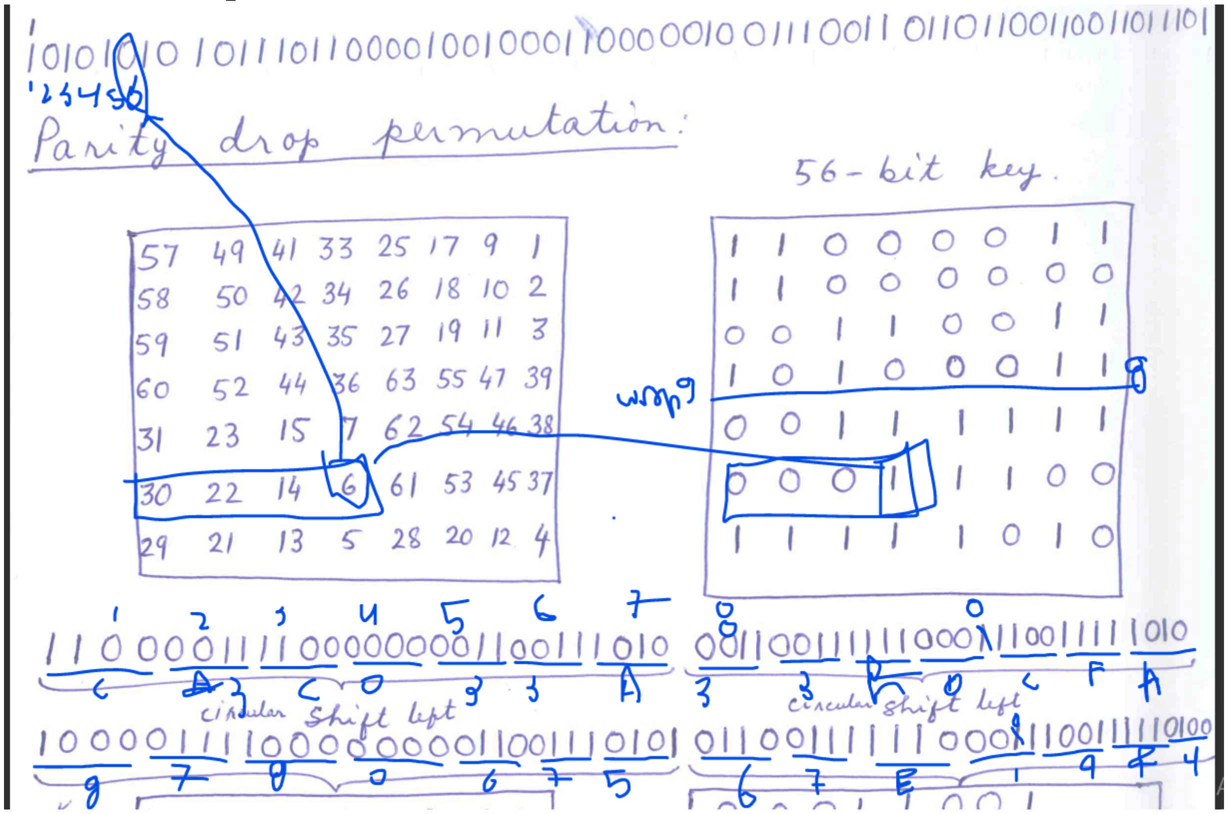
```
// Set the timescale for simulation
`timescale 1ns/1ps
module Key_generator();
```

```
reg [0:63] key_in;
wire [0:47] K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16;
wire [0:55] parity_out;

reg [0:27] key_in_cp1;
reg [0:27] key_in_cp2;
wire [0:47] cp_out;

        reg [4:0] round;            // Current round number (supports up to 16 rounds)
        reg [27:0] left_in;         // 28-bit left half of the cipher key
        reg [27:0] right_in;        // 28-bit right half of the cipher key
        wire [27:0] L_CK;           // Shifted left half
        wire [27:0] R_CK;

key_generation uut (key_in,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);
Parity_Drop uut2(key_in, parity_out);
CircularShift cs(round, left_in, right_in, L_CK, R_CK);
//Compression_PBOX_Key uut3(key_in_cp1, key_in_cp2,cp_out);


initial begin
 key_in = 64'hAABB09182736CCDD;
 round = 5'b00001;
 left_in = 28'hC3C033A;
 right_in = 28'h33F0CFA;
// key_in_cp1 = ;
// key_in_cp2 = ;
// K1 = 48'h194C_D072_DE8C;
// K2 = 48'h4568_581A_BCCE;
// K3 = 48'h06ED_A4AC_F5B5;
// K4 = 48'hDA2D_032B_6EE3;
// K5 = 48'h69A6_29FE_C913;
// K6 = 48'hC194_8E87_475E;
// K7 = 48'h708A_D2DD_B3C0;
// K8 = 48'h34F8_22F0_C66D;
// K9 = 48'h84BB_4473_DCCC;
// K10 = 48'h0276_5708_B5BF;
// K11 = 48'h6D55_60AF_7CA5;
// K12 = 48'hC2C1_E96A_4BF3;
// K13 = 48'h99C3_1397_C91F;
// K14 = 48'h251B_8BC7_17D0;
// K15 = 48'h3330_C5D9_A36D;
// K16 = 48'h181C_5D75_C66D;
 #10 //to simulate propagation
  $finish;
end

endmodule
```



```
round[4:0]       01        01
left_in[27:0]    c3c033a   c3c033a
right_in[27:0]   33f0cfa   33f0cfa
L_CK[27:0]       8780675   8780675
R_CK[27:0]       67e19f4   67e19f4
```

Sir's Solution proof:


Parity drop permutation:
56-bit key.
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
1 1 0 0 0 0 1 1
1 1 0 0 0 0 0 0
0 0 1 1 0 0 1 1
1 0 1 0 0 0 1 1
0 0 1 1 1 1 1 1
0 0 0 1 1 1 0 0
1 1 1 1 1 0 1 0
wrap
circular shift left
circular shift left

Task 4.3 - Key Compression Table.

The compression permutation (P-box) changes the 58 bits to 48 bits, which are used as a key for a round.

Create a module that accepts a 58-bit input and based on the provided permutation table, generates a 48-bit output.

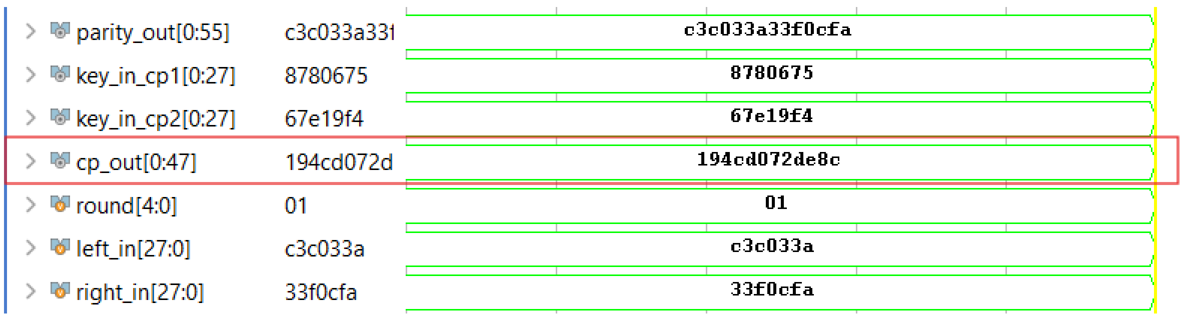
14	17	11	24	01	05	03	28
15	06	21	10	23	19	12	04
26	08	16	07	27	20	13	02
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

Figure 10: Key Compression Table

```
module Compression_PBOX_Key(
        input [0:27] L_CK,  // Left-shifted C_K (28 bits)
        input [0:27] R_CK,  // Right-shifted C_K (28 bits)
        output [0:47] C_k   // Cipher Key (48 bits)
);

        // Concatenate L_CK and R_CK to form the 56-bit key CK
        wire [0:55] CK = {L_CK, R_CK};

        // Assign output bits based on the Compression P-box mapping
        assign C_k[0]  = CK[13];
        assign C_k[1]  = CK[16];
        assign C_k[2]  = CK[10];
        assign C_k[3]  = CK[23];
        assign C_k[4]  = CK[0];
        assign C_k[5]  = CK[4];
        assign C_k[6]  = CK[2];
        assign C_k[7]  = CK[27];

        assign C_k[8]  = CK[14];
        assign C_k[9]  = CK[5];
        assign C_k[10] = CK[20];
        assign C_k[11] = CK[9];
        assign C_k[12] = CK[22];
        assign C_k[13] = CK[18];
        assign C_k[14] = CK[11];
        assign C_k[15] = CK[3];

        assign C_k[16] = CK[25];
        assign C_k[17] = CK[7];
        assign C_k[18] = CK[15];
        assign C_k[19] = CK[6];
        assign C_k[20] = CK[26];
        assign C_k[21] = CK[19];
        assign C_k[22] = CK[12];
        assign C_k[23] = CK[1];

        assign C_k[24] = CK[40];
        assign C_k[25] = CK[51];
        assign C_k[26] = CK[30];
        assign C_k[27] = CK[36];
        assign C_k[28] = CK[46];
        assign C_k[29] = CK[54];
        assign C_k[30] = CK[29];
        assign C_k[31] = CK[39];

        assign C_k[32] = CK[50];
        assign C_k[33] = CK[44];
        assign C_k[34] = CK[32];
        assign C_k[35] = CK[47];
        assign C_k[36] = CK[43];
        assign C_k[37] = CK[48];
        assign C_k[38] = CK[38];
        assign C_k[39] = CK[55];

        assign C_k[40] = CK[33];
        assign C_k[41] = CK[52];
        assign C_k[42] = CK[45];
        assign C_k[43] = CK[41];
        assign C_k[44] = CK[49];
        assign C_k[45] = CK[35];
        assign C_k[46] = CK[28];
        assign C_k[47] = CK[31];

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module Key_generator();



reg [0:63] key_in;
wire [0:47] K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16;
wire [0:55] parity_out;

wire [0:27] key_in_cp1;
wire [0:27] key_in_cp2;
wire [0:47] cp_out;
```

```
        reg [4:0] round;            // Current round number (supports up to 16 rounds)
        reg [27:0] left_in;         // 28-bit left half of the cipher key
        reg [27:0] right_in;        // 28-bit right half of the cipher key
        wire [27:0] L_CK;           // Shifted left half
        wire [27:0] R_CK;

key_generation uut (key_in,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);
Parity_Drop uut2(key_in, parity_out);
CircularShift cs(round, left_in, right_in, key_in_cp1, key_in_cp2);
Compression_PBOX_Key uut3(key_in_cp1, key_in_cp2,cp_out);


initial begin
 key_in = 64'hAABB09182736CCDD;
 round = 5'b00001;
 left_in = 28'hC3C033A;
 right_in = 28'h33F0CFA;
// key_in_cp1 = ;
// key_in_cp2 = ;
// K1 = 48'h194C_D072_DE8C;
// K2 = 48'h4568_581A_BCCE;
// K3 = 48'h06ED_A4AC_F5B5;
// K4 = 48'hDA2D_032B_6EE3;
// K5 = 48'h69A6_29FE_C913;
// K6 = 48'hC194_8E87_475E;
// K7 = 48'h708A_D2DD_B3C0;
// K8 = 48'h34F8_22F0_C66D;
// K9 = 48'h84BB_4473_DCCC;
// K10 = 48'h0276_5708_B5BF;
// K11 = 48'h6D55_60AF_7CA5;
// K12 = 48'hC2C1_E96A_4BF3;
// K13 = 48'h99C3_1397_C91F;
// K14 = 48'h251B_8BC7_17D0;
// K15 = 48'h3330_C5D9_A36D;
// K16 = 48'h181C_5D75_C66D;
 #10 //to simulate propagation
  $finish;
end

endmodule
```



```
parity_out[0:55]   c3c033a33f   c3c033a33f0cfa
key_in_cp1[0:27]   8780675      8780675
key_in_cp2[0:27]   67e19f4      67e19f4
cp_out[0:47]       194cd072d    194cd072de8c
round[4:0]         01           01
left_in[27:0]      c3c033a      c3c033a
right_in[27:0]     33f0cfa      33f0cfa
```

PROOF THAT IT'S CORRECT

K_1 = 194CD072DE8C ~~48-bit key~~

Task 4.4 - Designing the Key Generator.

Create a function that generates the keys required for all 16 rounds following the algorithm presented in Figure 11. To better grasp the operation of this function, you can also refer to Figure 7.

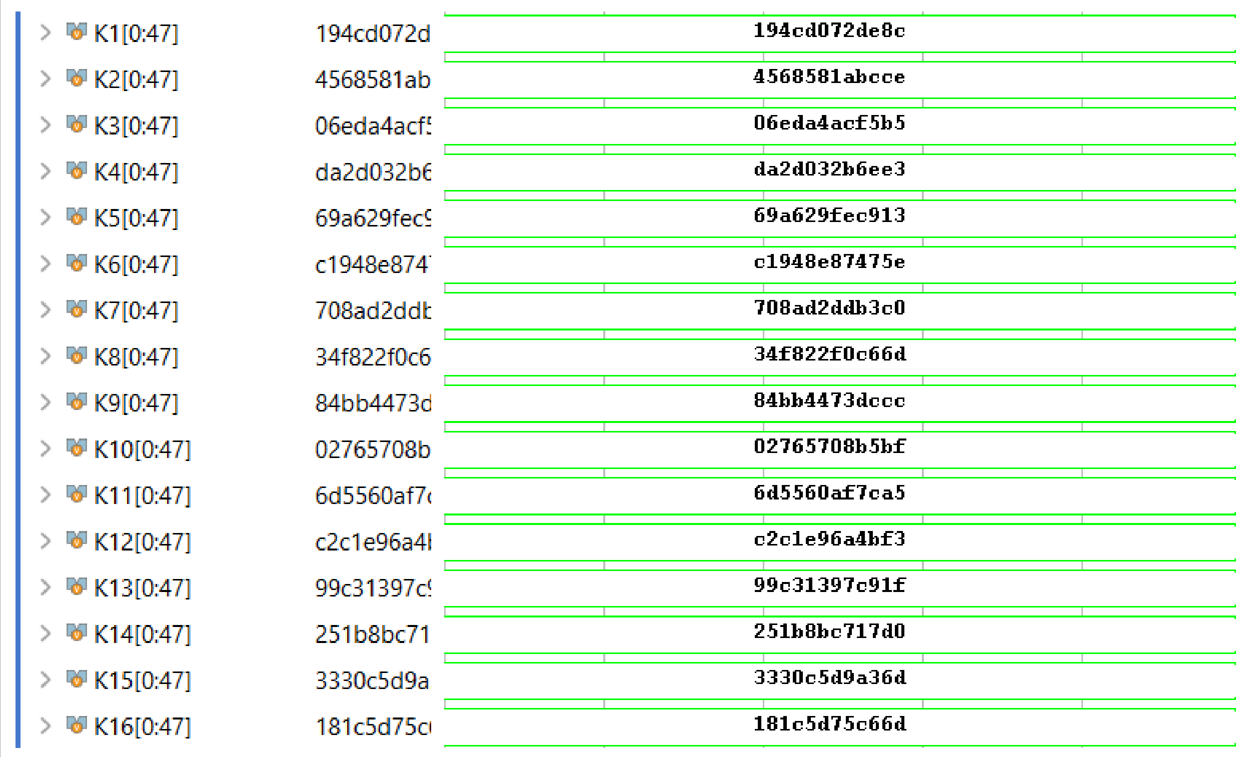
```
Key_Generator (keyWithParities[64], RoundKeys[16, 48], ShiftTable[16])
{
    permute (64, 56, keyWithParities, cipherKey, ParityDropTable)
    split (56, 28, cipherKey, leftKey, rightKey)
    for (round = 1 to 16)
    {
        shiftLeft (leftKey, ShiftTable[round])
        shiftLeft (rightKey, ShiftTable[round])
        combine (28, 56, leftKey, rightKey, preRoundKey)
        permute (56, 48, preRoundKey, RoundKeys[round], KeyCompressionTable)
    }
}
```

Figure 11: Round Key Generator Algorithm

```
module key_generation(
        input [63:0] in,            // Initial 64-bit key with parity bits
        output reg [47:0] k1, k2, k3, k4, k5, k6, k7, k8, k9, k10, k11, k12, k13, k14, k15, k16 // 48-bit
keys for each round
);

        wire [55:0] cipher_key;             // 56-bit cipher key after parity drop
        wire [27:0] left, right;    // 28-bit left and right halves of the cipher key for each round
        wire [27:0] L_CK, R_CK;         // Shifted left and right halves for the current round
        wire [47:0] round_key [1:16];  // Array to hold 48-bit keys for each round

        // Instantiate the Parity Drop module to generate the 56-bit key from the 64-bit input
        Parity_Drop parity_drop(in, cipher_key);
        assign left = cipher_key[55:28];
        assign right = cipher_key[27:0];

        // Loop through each of the 16 rounds
        genvar j;
        generate
        ////////////////////////
        for (j = 1; j <= 16; j = j + 1) begin : Key_generation
        Compression_PBOX_Key cpbox(left,right,round_key[j]);
        CircularShift CS(j, left,right,L_CK, R_CK);
        // Update left and right halves after shifting for the next round
        assign left = L_CK;
        assign right = R_CK;

        always @(*) begin
        case (j)
        1:  assign k1 = round_key[j];
        2:  assign k2 = round_key[j];
        3:  assign k3 = round_key[j];
        4:  assign k4 = round_key[j];
        5:  assign k5 = round_key[j];
        6:  assign k6 = round_key[j];
        7:  assign k7 = round_key[j];
        8:  assign k8 = round_key[j];
        9:  assign k9 = round_key[j];
        10: assign k10 = round_key[j];
        11: assign k11 = round_key[j];
        12: assign k12 = round_key[j];
        13: assign k13 = round_key[j];
        14: assign k14 = round_key[j];
        15: assign k15 = round_key[j];
        16: assign k16 = round_key[j];
        endcase
        end
        end
```

```
            endgenerate
endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module Key_generator();

reg [0:63] key_in;
wire [0:47] K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16;
wire [0:55] parity_out;

wire [0:27] key_in_cp1;
wire [0:27] key_in_cp2;
wire [0:47] cp_out;

            reg [4:0] round;            // Current round number (supports up to 16 rounds)
            reg [27:0] left_in;         // 28-bit left half of the cipher key
            reg [27:0] right_in;        // 28-bit right half of the cipher key
            wire [27:0] L_CK;           // Shifted left half
            wire [27:0] R_CK;

key_generation uut (key_in,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);
Parity_Drop uut2(key_in, parity_out);
CircularShift cs(round, left_in, right_in, key_in_cp1, key_in_cp2);
Compression_PBOX_Key uut3(key_in_cp1, key_in_cp2,cp_out);


initial begin
 key_in = 64'hAABB09182736CCDD;
 round = 5'b00001;
 left_in = 28'hC3C033A;
 right_in = 28'h33F0CFA;
// key_in_cp1 = ;
// K2 = 48'h4568_581A_BCCE;
// K3 = 48'h06ED_A4AC_F5B5;
// K4 = 48'hDA2D_032B_6EE3;
// K5 = 48'h69A6_29FE_C913;
// K6 = 48'hC194_8E87_475E;
// K7 = 48'h708A_D2DD_B3C0;
// K8 = 48'h34F8_22F0_C66D;
// K9 = 48'h84BB_4473_DCCC;
// K10 = 48'h0276_5708_B5BF;
// K11 = 48'h6D55_60AF_7CA5;
// K12 = 48'hC2C1_E96A_4BF3;
// K13 = 48'h99C3_1397_C91F;
// K14 = 48'h251B_8BC7_17D0;
// K15 = 48'h3330_C5D9_A36D;
// K16 = 48'h181C_5D75_C66D;
 #10 //to simulate propagation
  $finish;
end

endmodule
```



```
K1[0:47]    194cd072d    194cd072de8c
K2[0:47]    4568581ab    4568581abcce
K3[0:47]    06eda4acf5   06eda4acf5b5
K4[0:47]    da2d032b6    da2d032b6ee3
K5[0:47]    69a629fec9   69a629fec913
K6[0:47]    c1948e874    c1948e87475e
K7[0:47]    708ad2ddb    708ad2ddb3c0
K8[0:47]    34f822f0c6   34f822f0c66d
K9[0:47]    84bb4473d    84bb4473dccc
K10[0:47]   02765708b    02765708b5bf
K11[0:47]   6d5560af7    6d5560af7ca5
K12[0:47]   c2c1e96a4    c2c1e96a4bf3
K13[0:47]   99c31397c9   99c31397c91f
K14[0:47]   251b8bc71    251b8bc717d0
K15[0:47]   3330c5d9a    3330c5d9a36d
K16[0:47]   181c5d75c    181c5d75c66d
```

Task 5 – Validating the DES cipher.

1. Run the key generator function for "AABB09182736CCDD" to obtain the round keys for all 16 rounds.
2. Run the DES cipher for "123456ABCD132536".
3. Verify your output. Your results should match the values shown in Figure 12.

Note: For step 1 and 2, you will have to convert the plaintext into binary first.

```
module DES_Implementation(
        input [0:63] Input,
//      input [0:47] K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K12, K13, K14, K15, K16,
        input [0:63] key,
        output [0:63] Output,
        output [0:63] Round1Output, Round2Output, Round3Output, Round4Output,
        output [0:63] Round5Output, Round6Output, Round7Output, Round8Output,
        output [0:63] Round9Output, Round10Output, Round11Output, Round12Output,
        output [0:63] Round13Output, Round14Output, Round15Output, Round16Output
        );

        wire [0:31] L[0:16];  // Left parts for each round
        wire [0:31] R[0:16];  // Right parts for each round
        wire [0:63] ip;   // Initial Permutation output

        // Initial Permutation
        initial_permutation IP1(Input, ip);

        //key_generation
        wire [0:47] K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K12, K13, K14, K15, K16;
        key_generation kg(key,K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K12, K13, K14, K15,
K16);

        // Divide input into left and right halves after initial permutation
        assign L[0] = ip[0:31];
        assign R[0] = ip[32:63];

        // Declare wires for intermediate values within each round
        wire [0:47] exp_out[1:16];   // Expansion P-box output for each round
        wire [0:47] AK[1:16];               // After XOR with key
        wire [0:31] SBOX_out[1:16];  // S-box output for each round
        wire [0:31] SP[1:16];     // Straight P-box output for each round

        // Loop through each of the 16 rounds
        genvar i;
        generate
        ///////////////////////
        for (i = 1; i <= 16; i = i + 1) begin : DES_rounds
        // Expansion P-box on the right half of the previous round
        expansion_pbox Exp(R[i-1], exp_out[i]);

        // XOR with round key
        assign AK[i] = exp_out[i] ^ (i == 1 ? K1 :
```

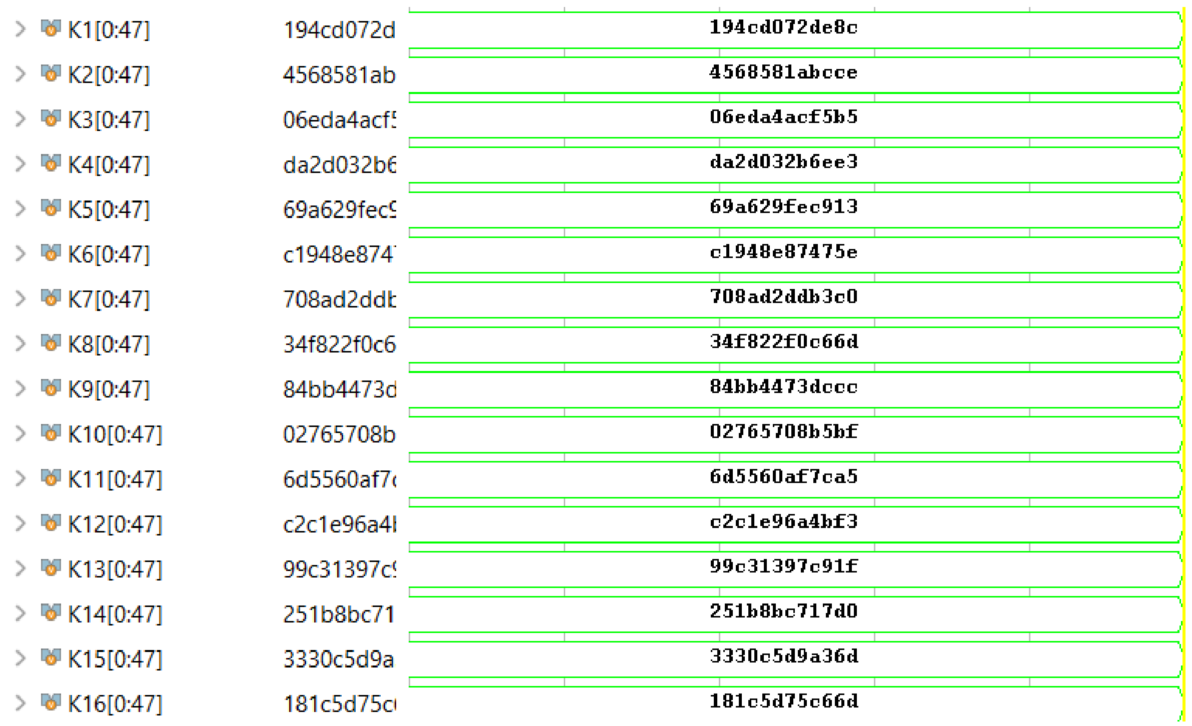
```
                    i == 2 ? K2 :
                    i == 3 ? K3 :
                    i == 4 ? K4 :
                    i == 5 ? K5 :
                    i == 6 ? K6 :
                    i == 7 ? K7 :
                    i == 8 ? K8 :
                    i == 9 ? K9 :
                    i == 10 ? K10 :
                    i == 11 ? K11 :
                    i == 12 ? K12 :
                    i == 13 ? K13 :
                    i == 14 ? K14 :
                    i == 15 ? K15 : K16);

// S-Box processing
Calculate_SBOX SB(AK[i], SBOX_out[i]);

// Straight Permutation
Straight_permutation SP_perm(SBOX_out[i], SP[i]);

// Compute the next right half and left half
assign L[i] = R[i-1];
assign R[i] = L[i-1] ^ SP[i];

// Output for each round's result as concatenation of left and right parts
end
endgenerate

// Assign each round's result to its respective output
assign Round1Output = {L[1], R[1]};
assign Round2Output = {L[2], R[2]};
assign Round3Output = {L[3], R[3]};
assign Round4Output = {L[4], R[4]};
assign Round5Output = {L[5], R[5]};
assign Round6Output = {L[6], R[6]};
assign Round7Output = {L[7], R[7]};
assign Round8Output = {L[8], R[8]};
assign Round9Output = {L[9], R[9]};
assign Round10Output = {L[10], R[10]};
assign Round11Output = {L[11], R[11]};
assign Round12Output = {L[12], R[12]};
assign Round13Output = {L[13], R[13]};
assign Round14Output = {L[14], R[14]};
assign Round15Output = {L[15], R[15]};
```

```
        assign Round16Output = {L[16], R[16]};

        // Final output without the swap of the last round
        assign Output = {L[16], R[16]};

endmodule
```

```
// Set the timescale for simulation
`timescale 1ns/1ps
module simulation();

        // Loop through each round and display results
integer i;

reg [0:63] data_in;
wire [0:63] data_out;
wire [0:63] R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16;

reg [0:63] IP1;
wire [0:63] IP_OUT;
        // Instantiate the final_permutation module
DES_Implementation uut
(data_in,data_out,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16);
initial_permutation IP_Module1 (IP1,IP_OUT);

reg [47:0] sbox_in;
wire [31:0] sbox_out;
Calculate_SBOX C_SP (sbox_in,sbox_out);

// Initial block for simulation setup
initial begin


//Testing out expansion box

//Testing out sbox in
 sbox_in = 48'b100101_100101_101010_000100_011111_011100_101111_010110;

 data_in = 64'h123456ABCD132536;
 //K1 = 48'b0001_1001_0100_1100_1101_0000_0111_0010_1101_1110_1000_1100;

 IP1 = 64'h123456ABCD132536;
 #10 //to simulate propagation
```

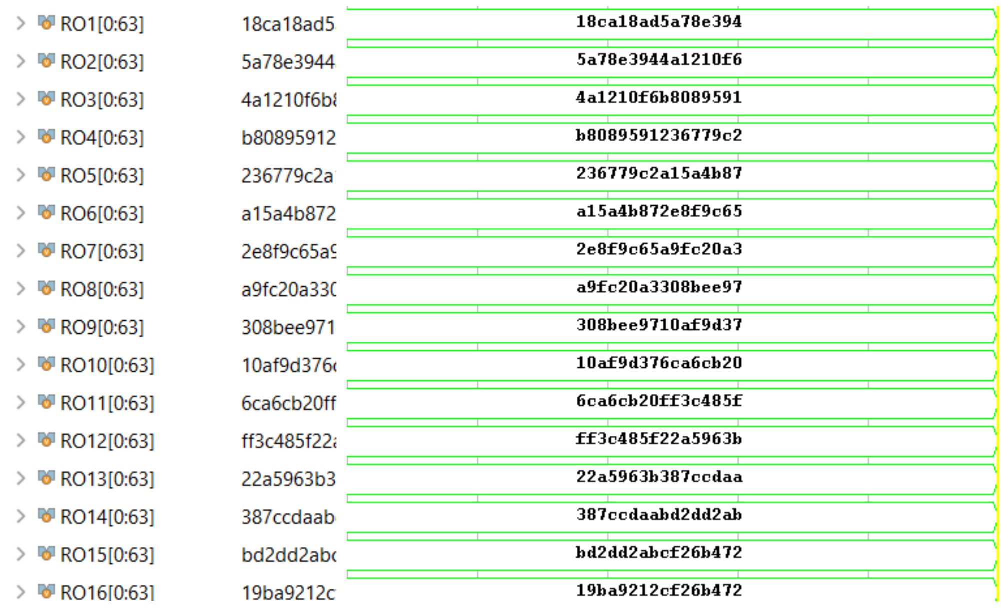
```
        // Log the output values for each round
        $display("Starting DES Simulation...");
        $display("Initial Input: %h", data_in);


        for (i = 1; i <= 16; i = i + 1) begin
        #10;  // Wait to allow round processing
        $display("Round %0d Results:", i);
        $display("  Expansion Output (exp_out): %h", uut.exp_out[i]);
        $display("  After Key XOR (AK): %h", uut.AK[i]);
        $display("  S-Box Output (SBOX_out): %h", uut.SBOX_out[i]);
        $display("  Straight Permutation (SP): %h", uut.SP[i]);
        $display("  Round %0d Output: %h", i, {uut.L[i], uut.R[i]});
        end

        // Display final output
        $display("Final Output (Data Out): %h", data_out);
  // End the simulation
  #10;
  $finish;
end

endmodule
```

PROOF THAT EVERYTHING IS WORKING (after I was done debugging the first question)
KEYS

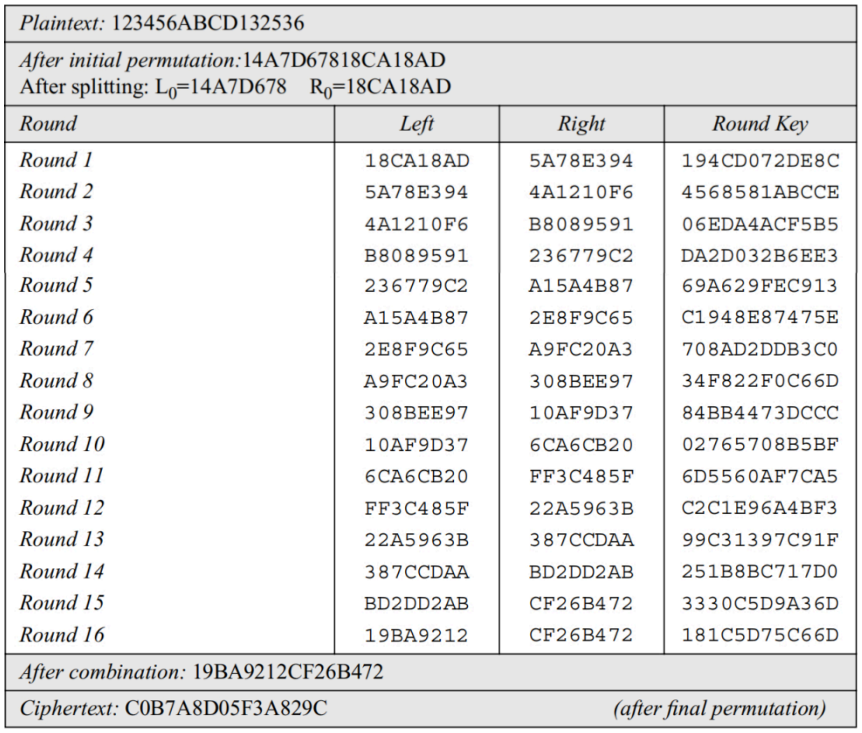


```
K1[0:47]     194cd072d    194cd072de8c
K2[0:47]     4568581ab    4568581abcce
K3[0:47]     06eda4acf    06eda4acf5b5
K4[0:47]     da2d032b6    da2d032b6ee3
K5[0:47]     69a629fec    69a629fec913
K6[0:47]     c1948e874    c1948e87475e
K7[0:47]     708ad2dd     708ad2ddb3c0
K8[0:47]     34f822f0c6   34f822f0c66d
K9[0:47]     84bb4473d    84bb4473dccc
K10[0:47]    02765708b    02765708b5bf
K11[0:47]    6d5560af7    6d5560af7ca5
K12[0:47]    c2c1e96a4    c2c1e96a4bf3
K13[0:47]    99c31397c    99c31397c91f
K14[0:47]    251b8bc71    251b8bc717d0
K15[0:47]    3330c5d9a    3330c5d9a36d
K16[0:47]    181c5d75c    181c5d75c66d
```

OUTPUT



```
RO1[0:63]    18ca18ad5    18ca18ad5a78e394
RO2[0:63]    5a78e3944    5a78e3944a1210f6
RO3[0:63]    4a1210f6b    4a1210f6b8089591
RO4[0:63]    b80895912    b8089591236779c2
RO5[0:63]    236779c2a    236779c2a15a4b87
RO6[0:63]    a15a4b872    a15a4b872e8f9c65
RO7[0:63]    2e8f9c65a    2e8f9c65a9fc20a3
RO8[0:63]    a9fc20a33    a9fc20a3308bee97
RO9[0:63]    308bee971    308bee9710af9d37
RO10[0:63]   10af9d376    10af9d376ca6cb20
RO11[0:63]   6ca6cb20ff   6ca6cb20ff3c485f
RO12[0:63]   ff3c485f22   ff3c485f22a5963b
RO13[0:63]   22a5963b3    22a5963b387ccdaa
RO14[0:63]   387ccdaab    387ccdaabd2dd2ab
RO15[0:63]   bd2dd2ab     bd2dd2abcf26b472
RO16[0:63]   19ba9212c    19ba9212cf26b472
```



Plaintext: 123456ABCD132536

After initial permutation: 14A7D67818CA18AD
After splitting: L_0=14A7D678 R_0=18CA18AD

Round	*Left*	*Right*	*Round Key*
Round 1	18CA18AD	5A78E394	194CD072DE8C
Round 2	5A78E394	4A1210F6	4568581ABCCE
Round 3	4A1210F6	B8089591	06EDA4ACF5B5
Round 4	B8089591	236779C2	DA2D032B6EE3
Round 5	236779C2	A15A4B87	69A629FEC913
Round 6	A15A4B87	2E8F9C65	C1948E87475E
Round 7	2E8F9C65	A9FC20A3	708AD2DDB3C0
Round 8	A9FC20A3	308BEE97	34F822F0C66D
Round 9	308BEE97	10AF9D37	84BB4473DCCC
Round 10	10AF9D37	6CA6CB20	02765708B5BF
Round 11	6CA6CB20	FF3C485F	6D5560AF7CA5
Round 12	FF3C485F	22A5963B	C2C1E96A4BF3
Round 13	22A5963B	387CCDAA	99C31397C91F
Round 14	387CCDAA	BD2DD2AB	251B8BC717D0
Round 15	BD2DD2AB	CF26B472	3330C5D9A36D
Round 16	19BA9212	CF26B472	181C5D75C66D

After combination: 19BA9212CF26B472

Ciphertext: C0B7A8D05F3A829C *(after final permutation)*

Figure 12: Trace of Data

Assessment Rubric
Lab 10
Data Encryption Standard (DES)

Name: Syed Asghar Abbas Zaidi	**Student ID:** 07201

Points Distribution

Task No.	LR 2 Code	LR5 Results	LR9 Report
Task 1	5		
Task 2.1	5		
Task 2.2	5		
Task 2.3	5		
Task 2.4	5		
Task 2.5	5		
Task 3	15		
Task 4.1	5		
Task 4.2	5		
Task 4.3	5		
Task 4.4	10		
Task 5	5	10	
Total	/75	/10	/5
CLO Mapped	CLO 2	CLO 2	CLO2

Affective Domain Rubric		Points	CLO Mapped
AR 7	Report Submission	/10	CLO 2

CLO	Total Points	Points Obtained
2	90	
2	10	
Total	**100**	

For description of different levels of the mapped rubrics, please refer the provided Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics.

Lab Evaluation Assessment Rubric

#	**Assessment Elements**	**Level 1: Unsatisfactory Points 0-1**	**Level 2: Developing Points 2**	**Level 3:Good Points 3**	**Level 4:Exemplary Points 4**
LR2	**Program/Code / Simulation Model/ Network Model**	Program/code/simulation model/network model does not implement the required functionality and has several errors. The student is not able to utilize even the basic tools of the software.	Program/code/simulation model/network model has some errors and does not produce completely accurate results. Student has limited command on the basic tools of the software.	Program/code/simulation model/network model gives correct output but not efficiently implemented or implemented by computationally complex routine.	Program/code/simulation /network model is efficiently implemented and gives correct output. Student has full command on the basic tools of the software.
LR5	**Results & Plots**	Figures/ graphs / tables are not developed or are poorly constructed with erroneous results. Titles, captions, units are not mentioned. Data is presented in an obscure manner.	Figures, graphs and tables are drawn but contain errors. Titles, captions, units are not accurate. Data presentation is not too clear.	All figures, graphs, tables are correctly drawn but contain minor errors or some of the details are missing.	Figures / graphs / tables are correctly drawn and appropriate titles/captions and proper units are mentioned. Data presentation is systematic.
LR9	**Report**	All the in-lab tasks are not included in report and / or the report is submitted too late.	Most of the tasks are included in report but are not well explained. All the necessary figures / plots are not included. Report is submitted after due date.	Good summary of most the in-lab tasks is included in report. The work is supported by figures and plots with explanations. The report is submitted timely.	Detailed summary of the in-lab tasks is provided. All tasks are included and explained well. Data is presented clearly including all the necessary figures, plots and tables.