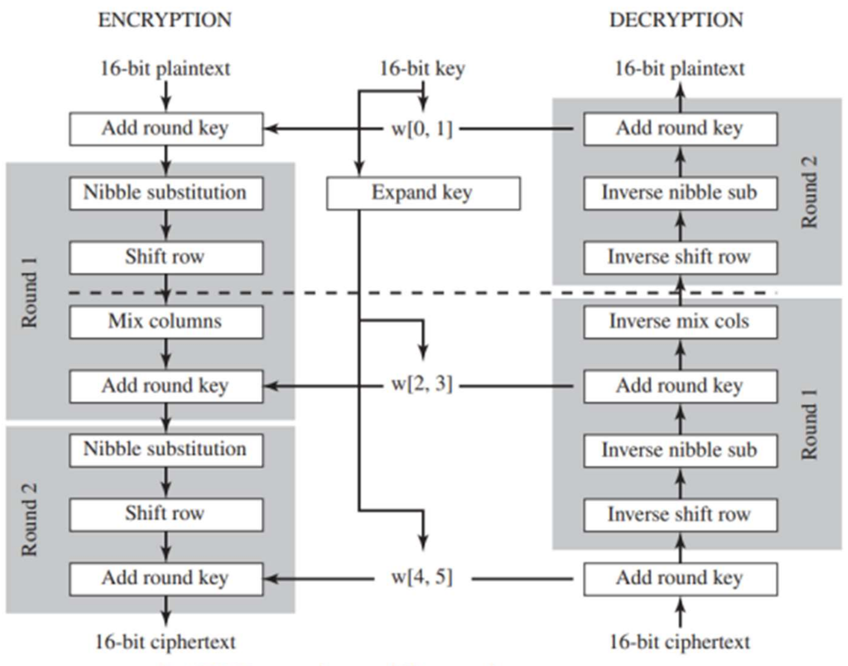
Simplified AES implementation in Java

Rimjhim Singh
1910110317

*NOTE: Explanation of code and output is included in this file. Executable code is submitted as a file named **seas.java***

Simplified AES encryption and decryption:

The difference between AES and S-AES is in key size (16 bit) , block size (16 bit) and number of rounds (2).



S-AES Encryption and Decryption

Data Structures used in S-AES:

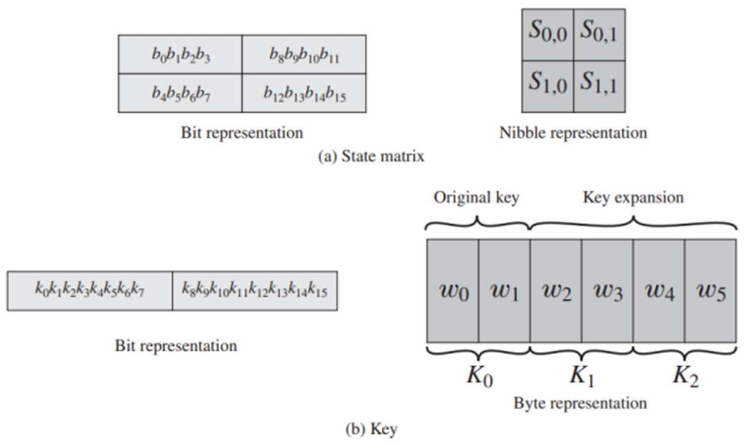


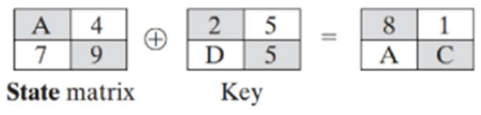
Figure 5.12 S-AES Data Structures

CODE:

```
import javax.sound.midi.Soundbank;
import java.util.*;
import java.io.*;


public class seas {

    static HashMap<String, String> sbox_map = new HashMap<>();
    static String[] W = new String[6];
    //S-Box for nibble substitution
    static String[] sbox = {"1001", "0100", "1010", "1011", "1101", "0001", "1000", "0101", "0110", "0010", "0000", "0011",
"1100", "1110","1111", "0111"};

    //Inverse S-Box for inverse nibble substitution
    static String[] sbox_inverse = {"1010", "0101", "1001", "1011", "0001", "0111", "1000", "1111", "0110", "0000", "0010",
"0011", "1100","0100", "1101", "1110"};


    public static void main(String[] args) throws IOException {

        Scanner s = new Scanner(System.in);

        BufferedReader ob = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter 1 for encryption");
        System.out.println("Enter 2 for decryption");
        int ch = s.nextInt();
        if (ch == 1)
        {
            System.out.println("Enter 16 bit plaintext");
            String plaintext = ob.readLine();
            System.out.println("Enter 16 bit key");
            String key = ob.readLine();
            keyGeneration(key);
            System.out.println("The corresponding ciphertext:");
            encrypt(plaintext); //encrypt function is called
        }
        else if (ch == 2)
        {
            System.out.println("Enter 16 bit Ciphertext");
            String ciphertext = ob.readLine();
            System.out.println("Enter 16 bit key");
            String key = ob.readLine();
            keyGeneration(key);
            System.out.println("The corresponding plaintext:");
            decrypt(ciphertext);//decrypt function is called

        }
        else
        {
            System.out.println("You have entered an invalid number, please run the program again");
        }
    }
```

Add key function:

The add key function consists of the bitwise XOR of the 16-bit State matrix and the 16-bit round key. This is used in encryption and decryption. The inverse of the add key function is identical to the add key function, because the XOR operation is its own inverse.
Example:

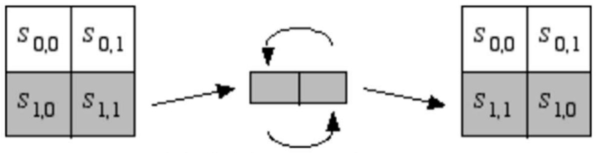


Both these concepts are implemented below, in the encrypt and decrypt functions.

```
    public static void encrypt(String plaintext) //encrypt function
    {
        // Adding Round 0 key
        String temp_value = xor(plaintext, W[0] + W[1]);

        //nibble substitution using s boxes
        String s1 = nibbleSubstitution(temp_value.substring(0, 8));
        String s2 = nibbleSubstitution(temp_value.substring(8, 16));
        temp_value = s1 + s2;
        // shift row function, swapping 2nd and 4th row
        temp_value = shiftRow(temp_value);

        //calling the mix column function
        temp_value = mixColumn(temp_value);
        //Adding round 1 key
        temp_value = xor(temp_value, W[2] + W[3]);

        //Final round, round 2 key
        //nibble substitution
        temp_value = nibbleSubstitution(temp_value.substring(0, 8)) +
                nibbleSubstitution(temp_value.substring(8, 16));


        //performing shift row again
        temp_value = shiftRow(temp_value);

        //ciphertext obtained after adding round 2 key
        String cipher_text = xor(temp_value, W[4] + W[5]);
        System.out.println(cipher_text);
    }

    public static void decrypt(String ciphertext) {
        //Adding round 2 key
        String temp_value = xor(ciphertext, W[4] + W[5]);

        //performing inverse shift row
        temp_value = shiftRow(temp_value);

        //inverse nibble substitution
        temp_value = inverseNibbleSubsitution(temp_value.substring(0, 8))     +inverseNibbleSubsitution(temp_value.substring(8, 
16));

        //adding round 1 key
        temp_value = xor(temp_value, W[2] + W[3]);
        //Nibbles in matrix form
        String rot = rotateNibble(temp_value.substring(4, 12));
        temp_value = temp_value.substring(0, 4) + rot + temp_value.substring(12, 16);

        //performing inverse mix column
        temp_value = inverseMixColumn(temp_value);

        //inverse shift row
        temp_value = shiftRow(temp_value);

        //inverse nibble substitution
        temp_value = inverseNibbleSubsitution(temp_value.substring(0, 8)) +inverseNibbleSubsitution(temp_value.substring(8, 
16));

        //Adding Round 0 key to get the plaintext
        String plaintext = xor(temp_value, W[0] + W[1]);
        System.out.println(plaintext);
    }
```

Shift row function:

The shift row function performs a one-nibble circular shift of the second row of State the first row is not altered.



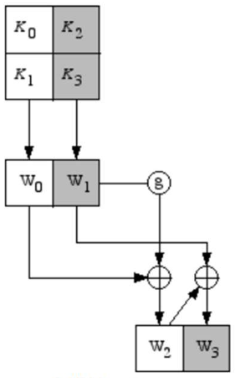
S-AES shift row transformation

The inverse shift row function is identical to the shift row function, because it shifts the second row back to its original position.

```
   public static String shiftRow(String str)
{

      String s1 = str.substring(0, 4);
      String s2 = str.substring(4, 8);
      String s3 = str.substring(8, 12);
      String s4 = str.substring(12, 16);
      return s1 + s4 + s3 + s2;
   }
```

Key expansion:

For key expansion, the 16 bits of the initial key are grouped into a row of two 8-bit words.
The g function is very similar to AES, first rotating the nibbles and then putting them through the S-boxes. The main difference is that the round constant is produced using x^ j+2, where j is the number of the round of expansion. That is, the first time you expand the key you use a round constant of x^3 = 1000 for the first nibble and 0000 for the second nibble. The second time you use x^4 = 0011 for the first nibble and 0000 for the second nibble.



S-AES key expansion

The algorithm is:

$$\begin{aligned}
w_2 &= w_0 \oplus g(w_1) = w_0 \oplus \text{Rcon}(1) \oplus \text{SubNib}(\text{RotNib}(w_1)) \\
w_3 &= w_2 \oplus w_1 \\
w_4 &= w_2 \oplus g(w_3) = w_2 \oplus \text{Rcon}(2) \oplus \text{SubNib}(\text{RotNib}(w_3)) \\
w_5 &= w_4 \oplus w_3
\end{aligned}$$

This concept is illustrated in the keygeneration function below.

```
   public static void keyGeneration(String key) {
      // Key 0 is W[0]+W[1]
      // Key 1 is W[2]+W[3]
      // Key 2 is W[4]+W[5]

      W[0] = key.substring(0, 8);
      W[1] = key.substring(8, 16);
      W[2] = xor(xor(W[0], "10000000"), nibbleSubstitution(rotateNibble(W[1])));
      W[3] = xor(W[2], W[1]);
      W[4] = xor(xor(W[2], "00110000"), nibbleSubstitution(rotateNibble(W[3])));
      W[5] = xor(W[4], W[3]);
   }

   public static String rotateNibble(String key)
   {
      String s1 = key.substring(0, 4);
```

```
        String s2 = key.substring(4, 8);
        return s2 + s1;
    }
```

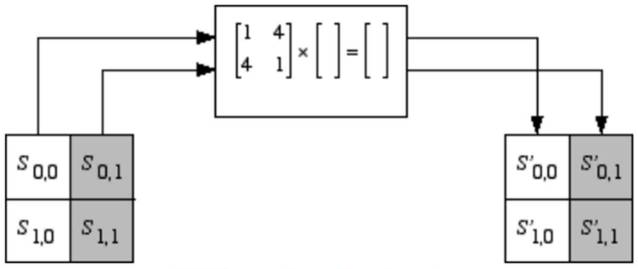
Nibble Substitution:
The nibble substitution function is a simple table lookup. AES defines a 4 * 4 matrix of nibble values, called an S-box that contains a permutation of all possible 4-bit values. Each individual nibble of State is mapped into a new nibble in the following way:

The leftmost 2 bits of the nibble are used as a row value, and the rightmost 2 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 4-bit output value.
The inverse nibble substitution function makes use of the inverse S-box
The functions below executes this.

```
    public static String nibbleSubstitution(String key)
    {
        // using s box to substitute nibbles
        String s1 = sbox[Integer.parseInt(key.substring(0, 4), 2)];
        String s2 = sbox[Integer.parseInt(key.substring(4, 8), 2)];
        return s1 + s2;
    }

    public static String inverseNibbleSubsitution(String key) {
        // using inverse s box to substitute nibbles
        String s1 = sbox_inverse[Integer.parseInt(key.substring(0, 4), 2)];
        String s2 = sbox_inverse[Integer.parseInt(key.substring(4, 8), 2)];
        return s1 + s2;
    }
```

Mix Column function:

The mix column function operates on each column individually. Each nibble of a column is mapped into a new value that is a function of both nibbles in that column. The transformation can be defined by the following matrix multiplication on State.



S-AES mix column transformation

$$\begin{bmatrix}1 & 4\\4 & 1\end{bmatrix}\begin{bmatrix}s_{0,0} & s_{0,1}\\s_{1,0} & s_{1,1}\end{bmatrix} = \begin{bmatrix}s'_{0,0} & s'_{0,1}\\s'_{1,0} & s'_{1,1}\end{bmatrix}$$

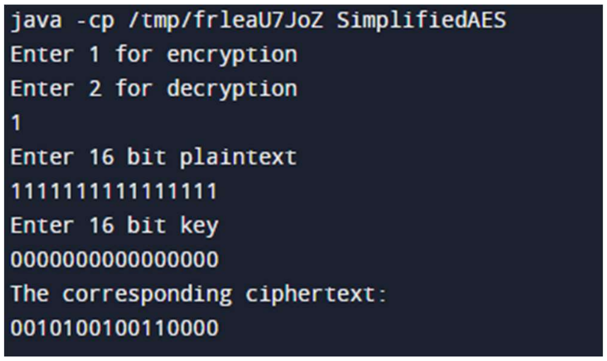
```
    public static String mixColumn(String str) {

        String s00 = xor(str.substring(0, 4), multiplication("0100", str.substring(8, 12)));
        String s10 = xor(str.substring(8, 12), multiplication("0100", str.substring(0, 4)));
        String s01 = xor(str.substring(4, 8), multiplication("0100", str.substring(12, 16)));
        String s11 = xor(str.substring(12, 16), multiplication("0100", str.substring(4, 8)));

        return s00 + s10 + s01 + s11;
    }

    public static String inverseMixColumn(String str) {
        String s00 = xor(multiplication("1001", str.substring(0, 4)),
                multiplication("0010", str.substring(8, 12)));
        String s10 = xor(multiplication("1001", str.substring(8, 12)),
                multiplication("0010", str.substring(0, 4)));
        String s01 = xor(multiplication("1001", str.substring(4, 8)),
                multiplication("0010", str.substring(12, 16)));
        String s11 = xor(multiplication("1001", str.substring(12, 16)),
                multiplication("0010", str.substring(4, 8)));
        return s00 + s10 + s01 + s11;
    }

    public static String multiplication(String s1, String s2) {

        int t1 = Integer.parseInt(s1, 2);
        int t2 = Integer.parseInt(s2, 2);
        int p = 0;
        while (t2 > 0) {
            if ((t2 & 0b1) != 0)
            {
                p ^= t1;
            }
            t1 <<= 1;
            if (((t1 & 0b10000) != 0))
                t1 ^= 0b11;
            t2 >>= 1;
        }
        int val = p & 0b1111;
        // adding zeroes to make strings even
        String ans = "";
        if (Integer.toBinaryString(val).length() < 4)
        {
            int temp = 4 - Integer.toBinaryString(val).length();
            for (int i = 0; i < temp; i++)
            {
                ans = ans + "0";
            }
            ans = ans + Integer.toBinaryString(val);
            return ans;

        }
        else
            return Integer.toBinaryString(val);
    }
```

<u>XOR function:</u>
This function simply XORs the two strings sent as an input

```
     public static String xor(String a, String b)
     {
        String xor = "";
        for (int i = 0; i < a.length(); i++) {
            if (a.charAt(i) == b.charAt(i))
                xor = xor + "0";
            else
                xor = xor + "1";
        }
        return xor;
    }
}
```

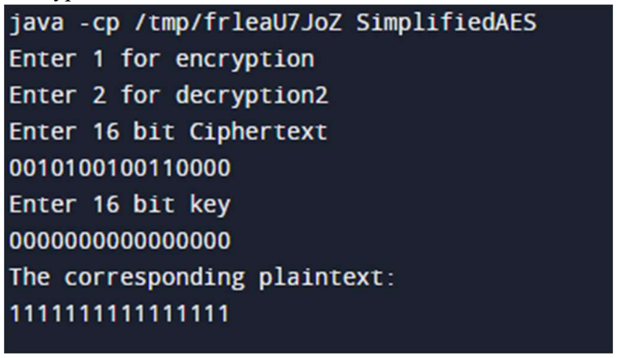
Output:

Example 1:
Encryption



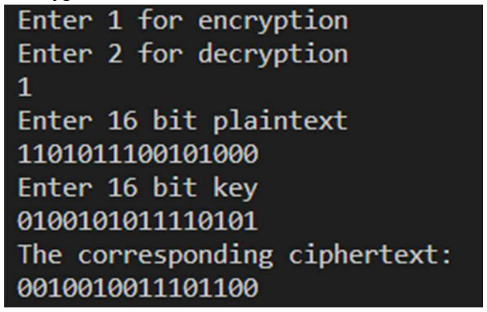
```
java -cp /tmp/frleaU7JoZ SimplifiedAES
Enter 1 for encryption
Enter 2 for decryption
1
Enter 16 bit plaintext
1111111111111111
Enter 16 bit key
0000000000000000
The corresponding ciphertext:
0010100100110000
```

Decryption



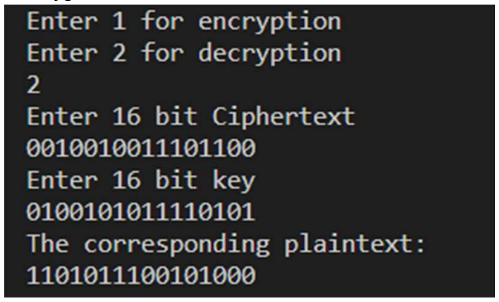
```
java -cp /tmp/frleaU7JoZ SimplifiedAES
Enter 1 for encryption
Enter 2 for decryption2
Enter 16 bit Ciphertext
0010100100110000
Enter 16 bit key
0000000000000000
The corresponding plaintext:
1111111111111111
```

Example 2:
Encryption



```
Enter 1 for encryption
Enter 2 for decryption
1
Enter 16 bit plaintext
1101011100101000
Enter 16 bit key
0100101011110101
The corresponding ciphertext:
0010010011101100
```

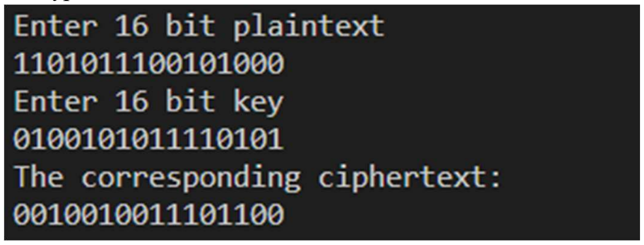
Decryption



```
Enter 1 for encryption
Enter 2 for decryption
2
Enter 16 bit Ciphertext
0010010011101100
Enter 16 bit key
0100101011110101
The corresponding plaintext:
1101011100101000
```

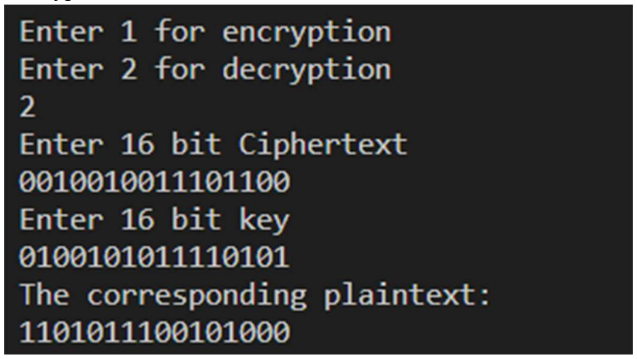
Example 3:

Encryption



```
Enter 16 bit plaintext
1101011100101000
Enter 16 bit key
0100101011110101
The corresponding ciphertext:
0010010011101100
```

Decryption



```
Enter 1 for encryption
Enter 2 for decryption
2
Enter 16 bit Ciphertext
0010010011101100
Enter 16 bit key
0100101011110101
The corresponding plaintext:
1101011100101000
```