ASSIGNMENT – 6

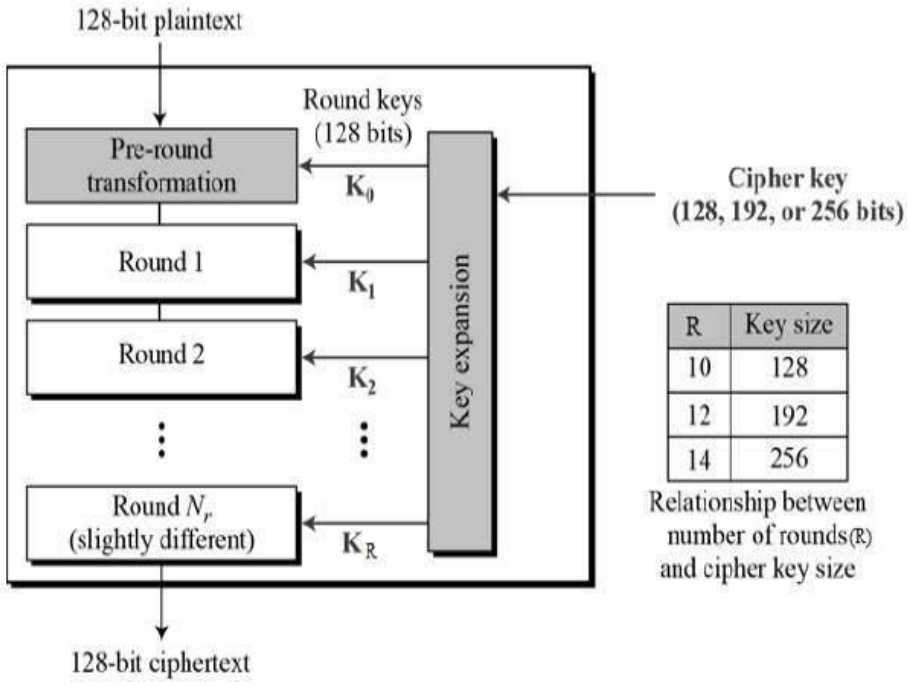
AES (Advanced Encryption Standard)

Theory :

AES is an iterative rather than Feistel cipher. It is based on 'substitution–permutation network'. It comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).

Interestingly, AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged in four columns and four rows for processing as a matrix.

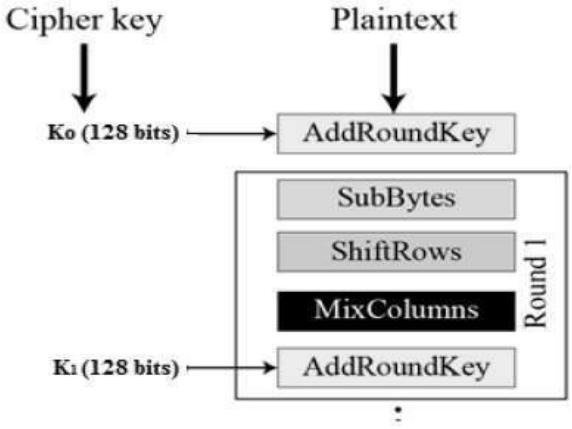
The schematic of AES structure is given in the following illustration –



R	Key size
10	128
12	192
14	256

Relationship between number of rounds(R) and cipher key size

Encryption :



Sub Byte : 16 Input bytes are substituted using a lookup table.

Shift Rows : Each of the four rows of the matrix is shifted to the left. First row is not shifted, second row is shifted by one position, third row by two position and fourth row by three positions.

Mix columns : We multiply the resultant matrix to another given fixed matrix.

Add Round Key : The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key.

Code :

```
#include <bits/stdc++.h>
using namespace std;
typedef bitset<8> bytes;
typedef bitset<32> word;

const int Nr = 10;
const int Nk = 4;

bytes S_Box[16][16] = {
      {0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B,
0xFE, 0xD7, 0xAB, 0x76},
      {0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF,
0x9C, 0xA4, 0x72, 0xC0},
      {0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1,
0x71, 0xD8, 0x31, 0x15},
      {0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2,
0xEB, 0x27, 0xB2, 0x75},
```

```
	{0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3,
0x29, 0xE3, 0x2F, 0x84},
	{0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39,
0x4A, 0x4C, 0x58, 0xCF},
	{0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F,
0x50, 0x3C, 0x9F, 0xA8},
	{0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21,
0x10, 0xFF, 0xF3, 0xD2},
	{0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D,
0x64, 0x5D, 0x19, 0x73},
	{0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14,
0xDE, 0x5E, 0x0B, 0xDB},
	{0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62,
0x91, 0x95, 0xE4, 0x79},
	{0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA,
0x65, 0x7A, 0xAE, 0x08},
	{0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F,
0x4B, 0xBD, 0x8B, 0x8A},
	{0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9,
0x86, 0xC1, 0x1D, 0x9E},
	{0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9,
0xCE, 0x55, 0x28, 0xDF},
	{0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F,
0xB0, 0x54, 0xBB, 0x16}
};

bytes Inv_S_Box[16][16] = {
	{0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E,
0x81, 0xF3, 0xD7, 0xFB},
	{0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44,
0xC4, 0xDE, 0xE9, 0xCB},
	{0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B,
0x42, 0xFA, 0xC3, 0x4E},
	{0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49,
0x6D, 0x8B, 0xD1, 0x25},
	{0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC,
0x5D, 0x65, 0xB6, 0x92},
	{0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57,
0xA7, 0x8D, 0x9D, 0x84},
	{0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05,
0xB8, 0xB3, 0x45, 0x06},
	{0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03,
0x01, 0x13, 0x8A, 0x6B},
	{0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE,
0xF0, 0xB4, 0xE6, 0x73},
	{0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8,
0x1C, 0x75, 0xDF, 0x6E},
```

```
	{0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E,
0xAA, 0x18, 0xBE, 0x1B},
	{0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE,
0x78, 0xCD, 0x5A, 0xF4},
	{0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59,
0x27, 0x80, 0xEC, 0x5F},
	{0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F,
0x93, 0xC9, 0x9C, 0xEF},
	{0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C,
0x83, 0x53, 0x99, 0x61},
	{0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63,
0x55, 0x21, 0x0C, 0x7D}
};


word Rcon[10] = {0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000,
		0x20000000, 0x40000000, 0x80000000, 0x1b000000, 0x36000000};


void SubBytes(bytes mtx[4*4])
{
	for(int i=0; i<16; ++i)
	{
	int row = mtx[i][7]*8 + mtx[i][6]*4 + mtx[i][5]*2 + mtx[i][4];
	int col = mtx[i][3]*8 + mtx[i][2]*4 + mtx[i][1]*2 + mtx[i][0];
	mtx[i] = S_Box[row][col];
	}
}


void ShiftRows(bytes mtx[4*4])
{
	//The second line circle moves one bit to the left
	bytes temp = mtx[4];
	for(int i=0; i<3; ++i)
	mtx[i+4] = mtx[i+5];
	mtx[7] = temp;
	//The third line circle moves two places to the left
	for(int i=0; i<2; ++i)
	{
	temp = mtx[i+8];
	mtx[i+8] = mtx[i+10];
	mtx[i+10] = temp;
	}
	//The fourth line moves three left circles
	temp = mtx[15];
	for(int i=3; i>0; --i)
	mtx[i+12] = mtx[i+11];
```

```
	mtx[12] = temp;
}

bytes GFMul(bytes a, bytes b) {
	bytes p = 0;
	bytes hi_bit_set;
	for (int counter = 0; counter < 8; counter++) {
	if ((b & bytes(1)) != 0) {
		p ^= a;
	}
	hi_bit_set = (bytes) (a & bytes(0x80));
	a <<= 1;
	if (hi_bit_set != 0) {
		a ^= 0x1b; /* x^8 + x^4 + x^3 + x + 1 */
	}
	b >>= 1;
	}
	return p;
}

void MixColumns(bytes mtx[4*4])
{
	bytes arr[4];
	for(int i=0; i<4; ++i)
	{
	for(int j=0; j<4; ++j)
		arr[j] = mtx[i+j*4];

	mtx[i] = GFMul(0x02, arr[0]) ^ GFMul(0x03, arr[1]) ^ arr[2] ^ arr[3];
	mtx[i+4] = arr[0] ^ GFMul(0x02, arr[1]) ^ GFMul(0x03, arr[2]) ^ arr[3];
	mtx[i+8] = arr[0] ^ arr[1] ^ GFMul(0x02, arr[2]) ^ GFMul(0x03, arr[3]);
	mtx[i+12] = GFMul(0x03, arr[0]) ^ arr[1] ^ arr[2] ^ GFMul(0x02, arr[3]);
	}
}


void AddRoundKey(bytes mtx[4*4], word k[4])
{
	for(int i=0; i<4; ++i)
	{
	word k1 = k[i] >> 24;
	word k2 = (k[i] << 8) >> 24;
	word k3 = (k[i] << 16) >> 24;
	word k4 = (k[i] << 24) >> 24;

	mtx[i] = mtx[i] ^ bytes(k1.to_ulong());
	mtx[i+4] = mtx[i+4] ^ bytes(k2.to_ulong());
	mtx[i+8] = mtx[i+8] ^ bytes(k3.to_ulong());
```

```
	mtx[i+12] = mtx[i+12] ^ bytes(k4.to_ulong());
	}
}


void InvSubBytes(bytes mtx[4*4])
{
	for(int i=0; i<16; ++i)
	{
	int row = mtx[i][7]*8 + mtx[i][6]*4 + mtx[i][5]*2 + mtx[i][4];
	int col = mtx[i][3]*8 + mtx[i][2]*4 + mtx[i][1]*2 + mtx[i][0];
	mtx[i] = Inv_S_Box[row][col];
	}
}


void InvShiftRows(bytes mtx[4*4])
{
	//The second line circle moves one bit to the right
	bytes temp = mtx[7];
	for(int i=3; i>0; --i)
	mtx[i+4] = mtx[i+3];
	mtx[4] = temp;
	//The third line circle moves two to the right
	for(int i=0; i<2; ++i)
	{
	temp = mtx[i+8];
	mtx[i+8] = mtx[i+10];
	mtx[i+10] = temp;
	}
	//Fourth line circle moves three to the right
	temp = mtx[12];
	for(int i=0; i<3; ++i)
	mtx[i+12] = mtx[i+13];
	mtx[15] = temp;
}

void InvMixColumns(bytes mtx[4*4])
{
	bytes arr[4];
	for(int i=0; i<4; ++i)
	{
	for(int j=0; j<4; ++j)
		arr[j] = mtx[i+j*4];

	mtx[i] = GFMul(0x0e, arr[0]) ^ GFMul(0x0b, arr[1]) ^ GFMul(0x0d, arr[2])
^ GFMul(0x09, arr[3]);
```

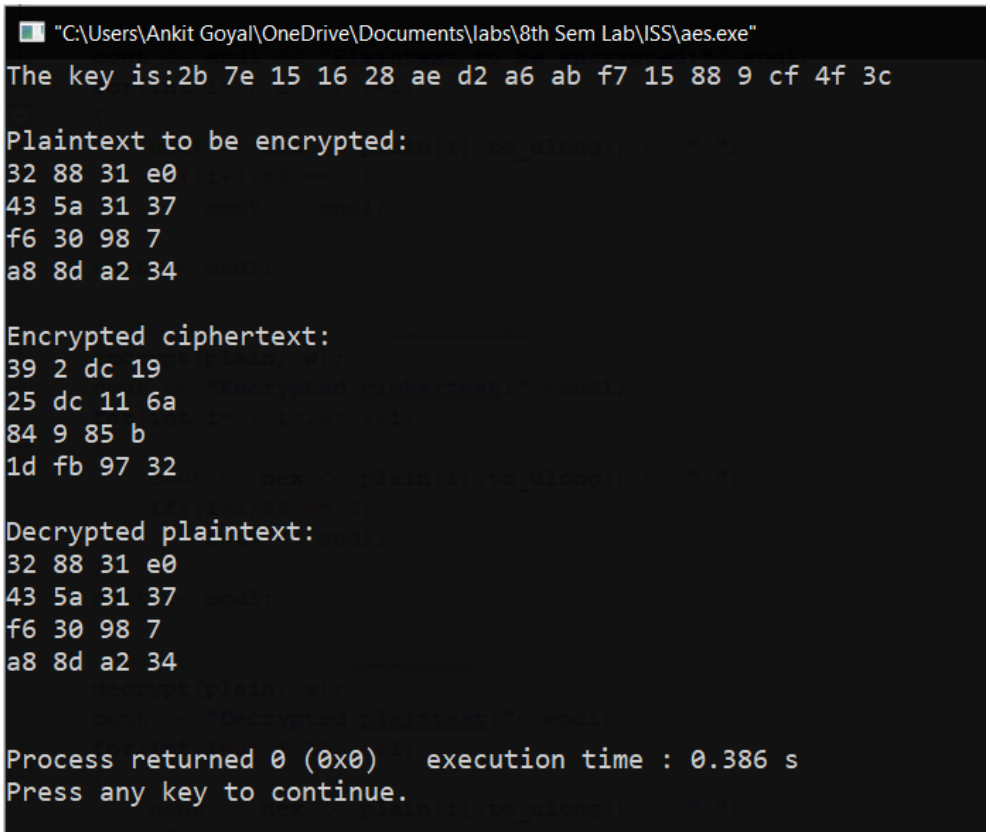
```
		mtx[i+4] = GFMul(0x09, arr[0]) ^ GFMul(0x0e, arr[1]) ^ GFMul(0x0b,
	arr[2]) ^ GFMul(0x0d, arr[3]);
		mtx[i+8] = GFMul(0x0d, arr[0]) ^ GFMul(0x09, arr[1]) ^ GFMul(0x0e,
	arr[2]) ^ GFMul(0x0b, arr[3]);
		mtx[i+12] = GFMul(0x0b, arr[0]) ^ GFMul(0x0d, arr[1]) ^ GFMul(0x09,
	arr[2]) ^ GFMul(0x0e, arr[3]);
		}
	}

	word Word(bytes& k1, bytes& k2, bytes& k3, bytes& k4)
	{
		word result(0x00000000);
		word temp;
		temp = k1.to_ulong();  // K1
		temp <<= 24;
		result |= temp;
		temp = k2.to_ulong();  // K2
		temp <<= 16;
		result |= temp;
		temp = k3.to_ulong();  // K3
		temp <<= 8;
		result |= temp;
		temp = k4.to_ulong();  // K4
		result |= temp;
		return result;
	}


	word RotWord(word& rw)
	{
		word high = rw << 8;
		word low = rw >> 24;
		return high | low;
	}


	word SubWord(word& sw)
	{
		word temp;
		for(int i=0; i<32; i+=8)
		{
		int row = sw[i+7]*8 + sw[i+6]*4 + sw[i+5]*2 + sw[i+4];
		int col = sw[i+3]*8 + sw[i+2]*4 + sw[i+1]*2 + sw[i];
		bytes val = S_Box[row][col];
		for(int j=0; j<8; ++j)
			temp[i+j] = val[j];
		}
		return temp;
```

```
}
void KeyExpansion(bytes key[4*Nk], word w[4*(Nr+1)])
{
	word temp;
	int i = 0;
	while(i < Nk)
	{
	w[i] = Word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]);
	++i;
	}

	i = Nk;

	while(i < 4*(Nr+1))
	{
	temp = w[i-1];
	if(i % Nk == 0)  {
		word r=RotWord(temp);
		w[i] = w[i-Nk] ^ SubWord(r) ^ Rcon[i/Nk-1];
	}
	else
		w[i] = w[i-Nk] ^ temp;
	++i;
	}
}

void encrypt(bytes in[4*4], word w[4*(Nr+1)])
{
	word key[4];
	for(int i=0; i<4; ++i)
	key[i] = w[i];
	AddRoundKey(in, key);

	for(int round=1; round<Nr; ++round)
	{
	SubBytes(in);
	ShiftRows(in);
	MixColumns(in);
	for(int i=0; i<4; ++i)
		key[i] = w[4*round+i];
	AddRoundKey(in, key);
	}

	SubBytes(in);
	ShiftRows(in);
	for(int i=0; i<4; ++i)
	key[i] = w[4*Nr+i];
	AddRoundKey(in, key);
```

```
}

void decrypt(bytes in[4*4], word w[4*(Nr+1)])
{
	word key[4];
	for(int i=0; i<4; ++i)
	key[i] = w[4*Nr+i];
	AddRoundKey(in, key);

	for(int round=Nr-1; round>0; --round)
	{
	InvShiftRows(in);
	InvSubBytes(in);
	for(int i=0; i<4; ++i)
		key[i] = w[4*round+i];
	AddRoundKey(in, key);
	InvMixColumns(in);
	}

	InvShiftRows(in);
	InvSubBytes(in);
	for(int i=0; i<4; ++i)
	key[i] = w[i];
	AddRoundKey(in, key);
}

int main()
{
	bytes key[16] = {0x2b, 0x7e, 0x15, 0x16,
			0x28, 0xae, 0xd2, 0xa6,
			0xab, 0xf7, 0x15, 0x88,
			0x09, 0xcf, 0x4f, 0x3c};

	bytes plain[16] = {0x32, 0x88, 0x31, 0xe0,
			0x43, 0x5a, 0x31, 0x37,
			0xf6, 0x30, 0x98, 0x07,
			0xa8, 0x8d, 0xa2, 0x34};
	//Output key
	cout << "The key is:";
	for(int i=0; i<16; ++i)
	cout << hex << key[i].to_ulong() << " ";
	cout << endl;

	word w[4*(Nr+1)];
	KeyExpansion(key, w);

	//Output plaintext to be encrypted
	cout << endl << "Plaintext to be encrypted:"<<endl;
```

```
	for(int i=0; i<16; ++i)
	{
	cout << hex << plain[i].to_ulong() << " ";
	if((i+1)%4 == 0)
		cout << endl;
	}
	cout << endl;

	//Encryption, output ciphertext
	encrypt(plain, w);
	cout << "Encrypted ciphertext:"<<endl;
	for(int i=0; i<16; ++i)
	{
	cout << hex << plain[i].to_ulong() << " ";
	if((i+1)%4 == 0)
		cout << endl;
	}
	cout << endl;

	//Decrypt, output plaintext
	decrypt(plain, w);
	cout << "Decrypted plaintext:"<<endl;
	for(int i=0; i<16; ++i)
	{
	cout << hex << plain[i].to_ulong() << " ";
	if((i+1)%4 == 0)
		cout << endl;
	}
	cout << endl;
	return 0;
}
```

Output:


"C:\Users\Ankit Goyal\OneDrive\Documents\labs\8th Sem Lab\ISS\aes.exe"
The key is:2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
Plaintext to be encrypted:
32 88 31 e0
43 5a 31 37
f6 30 98 7
a8 8d a2 34
Encrypted ciphertext:
39 2 dc 19
25 dc 11 6a
84 9 85 b
1d fb 97 32
Decrypted plaintext:
32 88 31 e0
43 5a 31 37
f6 30 98 7
a8 8d a2 34
Process returned 0 (0x0) execution time : 0.386 s
Press any key to continue.