**Rijandel algorithm**

#include <iostream>

#include <array>

using namespace std;

void printBlock(const array<array<int,4>,4>& block){

    //Function to print a block to terminal

    for (size\_t i = 0; i < 4; ++i){

        printf("%.2x, %.2x, %.2x, %.2x\n", block[i][0], block[i][1], block[i][2], block[i][3]);

    }

}

void printRow(const array<int,4>& row){

    //Function to print a row to terminal

    printf("%.2x, %.2x, %.2x, %.2x\n", row[0], row[1], row[2], row[3]);

}

void print\_1d\_State(const array<int,16>& e){

    //Function to print a 1d version of the state to terminal

    printf("%.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x, %.2x\n",

    e[0], e[1], e[2], e[3], e[4], e[5], e[6], e[7], e[8], e[9], e[10], e[11], e[12], e[13], e[14], e[15]);

}

void printKeySchedule(const array<array<int,16>,11>& keySchedual){

    //Function to print keys for debugging

    for (size\_t i=0; i<11; ++i){

        print\_1d\_State(keySchedual[i]);

    }

}

array<array<int,4>,4> inputToState(const array<int,16>& input){

    //Function turns a 128bit/ 16 element array and minipulates into a 4x4 matrix

    array<array<int,4>,4> state = {};

    //Loop the input and assign to state

    for(size\_t i=0; i<4; ++i){

        for(size\_t j=0; j<4; ++j){

            state[i][j] = input[i+4\*j];

        }

    }

    return state;

}

array<int,16> stateToOutput(const array<array<int,4>,4>& state){

    //Function takes a 128bit 4x4 block/ matrix and minipulates into a 1d array of words

    array<int,16> output = {};

    //Loop the state and assign to output

    for(size\_t i=0; i<4; ++i){

        for(size\_t j=0; j<4; ++j){

            output[i+4\*j] = state[i][j];

        }

    }

    return output;

}

array<array<int,4>,4> keyToWords(const array<int,16>& roundKey){

    //Function will take a rounds key and minipulate it into a 4x4 array

    //With each row being a word

    array<array<int,4>,4> wordBlock = {};

    for (size\_t i= 0; i < 4; ++i){

        //Get words from key

        wordBlock[i] = {roundKey[4\*i], roundKey[1+i\*4], roundKey[2+i\*4], roundKey[3+i\*4]};

    }

    return wordBlock;

}

array<array<int,4>,4> transposeState(const array<array<int,4>,4>& state){

    //Function will transpose the state into a 4x4 matrix of words

    array<array<int,4>,4> tState = {};

    for(size\_t i = 0; i < 4; ++i){

        for(size\_t j = 0; j < 4; ++j){

            tState[i][j] = state[j][i];

        }

    }

    return tState;

}

int multiply\_by\_2(const int v){

/\*

Function to impliment multiplication by 2 of Galois Field GF(2^8).

Shift bits left by 1, if high bit is 0 return the value.

If the high bit is one XOR the value with 0x1B.

(0x1B comes from the field representation)

\*/

    int s = v << 1; //Shift bits left by 1

    //If high bit = 1 (0x80 = 10000000bin)

    if (v & 0x80){

        s &= 0xff; //Bitwise AND (ff = 11111111bin) This effectivly takes first 8 bits

        s = s ^ 0x1b;

    }

    return s;

}

int multiply\_by\_3(int v){

/\*

Function to impliment multiplication by 3 of Galois Field GF(2^8).

This is simply the XOR of multiply by 2, with the original value.

\*/

    return multiply\_by\_2(v) ^ v;

}

int lookupByte(int &byte){

    //This method takes a byte, performs a lookup in the AES SBOX and returns the corresponding value

    int x = byte >> 4; //Shifts 4 bits right i.e. takes first 4 bits and discards the rest

    int y = byte & 0x0f; // 0x0f = 15 = 00001111(bin). Effectivly takes last 4 bits and dicards the rest

    const int sbox[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}};

     return sbox[x][y];

}

int invLookupByte(int &byte){

    //This method takes a byte, performs a lookup in the inverse AES SBOX and returns the corresponding value

    int x = byte >> 4; //Shifts 4 bits right i.e. takes first 4 bits and discards the rest

    int y = byte & 0x0f; // 0x0f = 15 = 00001111(bin). Effectivly takes last 4 bits and dicards the rest

    const int sboxinv[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}};

    return sboxinv[x][y];

}

auto subBytes(const array<array<int,4>,4>& state){

    array<array<int,4>,4> result = {};

    int byte;

    for (size\_t i=0; i<4; ++i){

        for (size\_t j=0; j<4; ++j){

            byte = state[i][j];

            result[i][j] = lookupByte(byte);

        }

    }

    return result;

}

auto unSubBytes(const array<array<int,4>,4>& state){

    array<array<int,4>,4> result = {};

    int byte;

    for (size\_t i=0; i<4; ++i){

        for (size\_t j=0; j<4; ++j){

            byte = state[i][j];

            result[i][j] = invLookupByte(byte);

        }

    }

    return result;

}

array<int,4> shiftRow(const array<int,4>& row, const int shift){

    array<int,4> result = {};

    result = row;

    if(shift){

        //Shift by 1

        int temp = result[0];

        for (size\_t i=0; i<3; ++i){

            result[i] = result[i+1];

        }

        result[3] = temp;

        //reduce shift and perform again

        result = shiftRow(result, shift -1);

    }

    else{

        return result;

    }

}

auto shiftRows(const array<array<int,4>,4>& state){

    array<array<int,4>,4> result = {};

    for (size\_t i = 0; i < 4; i++){

        result[i] = shiftRow(state[i], i);

    }

    return result;

}

array<int,4> unShiftRow(const array<int,4>& row, const int shift){

    array<int,4> result = {};

    result = row;

    if(shift){

        //Shift by 1

        int temp = result[3];

        for (int i=3; i>0; --i){

            result[i] = result[i-1];

        }

        result[0] = temp;

        //reduce shift and perform again

        result = unShiftRow(result, shift -1);

    }

    else{

        return result;

    }

}

auto unShiftRows(const array<array<int,4>,4>& state){

    //Function to shift rows of the state right by the value of their index

    array<array<int,4>,4> result = {};

    for (size\_t i = 0; i < 4; i++){

        result[i] = unShiftRow(state[i], i);

    }

    return result;

}

auto mixColumn(const array<int,4>& stateColumn){

    array<int,4> result = {};

    result[0] = multiply\_by\_2(stateColumn[0]) ^ multiply\_by\_3(stateColumn[1]) ^ stateColumn[2] ^ stateColumn[3];

    result[1] = multiply\_by\_2(stateColumn[1]) ^ multiply\_by\_3(stateColumn[2]) ^ stateColumn[3] ^ stateColumn[0];

    result[2] = multiply\_by\_2(stateColumn[2]) ^ multiply\_by\_3(stateColumn[3]) ^ stateColumn[0] ^ stateColumn[1];

    result[3] = multiply\_by\_2(stateColumn[3]) ^ multiply\_by\_3(stateColumn[0]) ^ stateColumn[1] ^ stateColumn[2];

    return result;

}

auto mixColumns(const array<array<int,4>,4>& state){

    array<array<int,4>,4> result = {};

    array<int,4> column = {};

    for(size\_t i = 0; i < 4; ++i){

        for(size\_t j = 0; j < 4; ++j){

            column[j] = state[j][i];

        }

        column = mixColumn(column);

        //Transpose the columns back into rows

        for(size\_t j = 0; j < 4; ++j){

            result[j][i] = column[j];

        }

    }

    return result;

}

auto unMixColumns(const array<array<int,4>,4>& state){

    array<array<int,4>,4> result = {};

    result = mixColumns(state);

    result = mixColumns(result);

    result = mixColumns(result);

    return result;

}

array<int,4> xorWords(const array<int,4>& wordA, const array<int,4>& wordB){

    array<int,4> result = {};

    for (size\_t i = 0; i < 4; i++){

        result[i] = wordA[i] ^ wordB[i];

    }

    return result;

}

array<int,4> subWord(const array<int,4>& word){

    //Function takes the a word, performs an sbox lookup on each element and returns the result

    array<int,4> result = {};

    int byte;

    for (size\_t i=0; i<4; ++i){

        byte = word[i];

        result[i] = lookupByte(byte);

    }

    return result;

}

array<int,4> rotWord(const array<int,4>& word){

    //Function takes the a word, performs a rightward cyclic shift and returns the result

    array<int,4> result = {};

    result = shiftRow(word, 1);

    return result;

}

auto addRoundKey(const array<array<int,4>,4>& state, const array<int,16>& roundKey){

    //Function takes 2 4x4 matricies, xors the corresponding columns and returns the result

    array<array<int,4>,4> result = {};

    array<array<int,4>,4> stateWords = {};

    array<array<int,4>,4> keyWords = {};

    //Convert state and key into rows of words

    stateWords = transposeState(state);

    keyWords = keyToWords(roundKey);

    //Loop columns of the state and xor with the corresponding key column

    for (size\_t i = 0; i < 4; ++i){

        result[i] = xorWords(stateWords[i], keyWords[i]);

    }

    //Transpose words back into columns when returning

    return transposeState(result);

}

array<array<int,16>,11> generateKeys(const array<int,16>& key){

    array<array<int,16>,11> keySchedule = {};

    array<array<int,4>,4> roundKey = {};

    array<array<int,4>,4> pRoundKey = {};

    array<int,16> temp = {};

    const array<array<int,4>,10> RCON = {{{0x01, 0x00, 0x00, 0x00},

                                        {0x02, 0x00, 0x00, 0x00},

                                        {0x04, 0x00, 0x00, 0x00},

                                        {0x08, 0x00, 0x00, 0x00},

                                        {0x10, 0x00, 0x00, 0x00},

                                        {0x20, 0x00, 0x00, 0x00},

                                        {0x40, 0x00, 0x00, 0x00},

                                        {0x80, 0x00, 0x00, 0x00},

                                        {0x1b, 0x00, 0x00, 0x00},

                                        {0x36, 0x00, 0x00, 0x00}}};

    keySchedule[0] = key;

    //Extract the words from the key and arrange into a block

    for (size\_t i=0; i<4; ++i){

        pRoundKey[i] = {key[4\*i], key[4\*i+1], key[4\*i+2], key[4\*i+3]};

    }

    //Loop rounds

    for (size\_t i=0; i<10; ++i){

        //Rotate last word of PRK

        roundKey[0] = rotWord(pRoundKey[3]);

        //SubBytes of last word in PRK

        roundKey[0] = subWord(roundKey[0]);

        //XOR : first word PRK, current RK state, rcon(round)

        roundKey[0] = xorWords(xorWords(pRoundKey[0], roundKey[0]), RCON[i]);

        //XOR the other words in sequential order

        for (size\_t j=1; j<4; ++j){

            roundKey[j] = xorWords(roundKey[j-1], pRoundKey[j]);

        }

        //Arrange block back into 1d key

        for (size\_t r=0; r<4; ++r){

            for (size\_t c=0; c<4; ++c){

                temp[4\*r+c] = roundKey[r][c];

            }

        }

        //Add new round key to key schedule

        keySchedule[i+1] = temp;

        //Set new PRK

        pRoundKey = roundKey;

    }

    return keySchedule;

}

array<array<int,4>,4> encryptBlock(const array<array<int,4>,4>& block, const array<array<int,16>,11> keySchedual){

    array<array<int,4>,4> state = block;

    state = addRoundKey(state, keySchedual[0]);

    for (size\_t i=1; i<10; ++i){

        //SubBytes

        state = subBytes(state);

        //ShiftRows

        state = shiftRows(state);

        //MixColumns

        state = mixColumns(state);

        //AddRoundKey

        state = addRoundKey(state, keySchedual[i]);

    }

    state = subBytes(state);

    state = shiftRows(state);

    state = addRoundKey(state, keySchedual[10]);

    return state;

}

array<array<int,4>,4> decryptBlock(const array<array<int,4>,4>& block, const array<array<int,16>,11> keySchedual){

    array<array<int,4>,4> state = {};

    state = block;

    state = addRoundKey(state, keySchedual[10])

    state = unShiftRows(state);

    state = unSubBytes(state);

    //Undo 9 main rounds

    for (size\_t i=9; i>0; --i){

        //Undo addRoundKey

        state = addRoundKey(state, keySchedual[i]);

        //Undo mixColumns

        state = unMixColumns(state);

        //Undo shiftRows

        state = unShiftRows(state);

        //Undo SubBytes

        state = unSubBytes(state);

    }

    //undo first round

    //Undo AddRoundKey

    state = addRoundKey(state, keySchedual[0]);

    return state;

}

int main(){

    /\*

    Example values given in FIPS 197

    Input = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

    Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

    \*/

    array<int,16> input = {0x32,0x43,0xf6,0xa8,0x88,0x5a,0x30,0x8d,

                        0x31,0x31,0x98,0xa2,0xe0,0x37,0x07,0x34};;

    array<int,16> key = {0x2b,0x7e,0x15,0x16,0x28,0xae,0xd2,0xa6,

                        0xab,0xf7,0x15,0x88, 0x09,0xcf,0x4f,0x3c};

    array<array<int,16>,11> keySchedual = {};

    array<array<int,4>,4> pTextBlock = {};

    array<array<int,4>,4> cTextBlock = {};

    array<int,16> output = {};

    //Generate keys

    keySchedual = generateKeys(key);

    //Perform encryption

    cout << "INITIAL INPUT:\n";

    print\_1d\_State(input);

    pTextBlock = inputToState(input);

    cTextBlock = encryptBlock(pTextBlock, keySchedual);

    cout << "CIPHER TEXT:\n";

    output = stateToOutput(cTextBlock);

    print\_1d\_State(output);

    //Perform decryption

    pTextBlock = decryptBlock(cTextBlock, keySchedual);

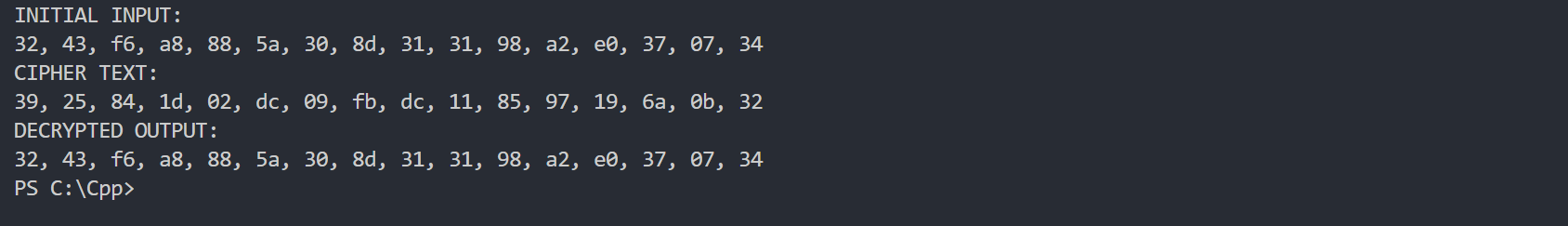
    cout << "DECRYPTED OUTPUT:\n";

    output = stateToOutput(pTextBlock);

    print\_1d\_State(output);

}

**Output**

****