EEN1059 Assignment

Name: Vamsi Kiran Mekala
ID: A00048280
Email Address: vamsikiran.mekala2@mail.dcu.ie
Programme: MEng Electronic and Computer Engineering

EEN1059 Assignment ---- 1
Assignment Implementation (AES): ---- 1
Key Expansion: ---- 1
Word Generation: ---- 1
Code Snippet: ---- 1
Transformation Logic ---- 2
Code Snippet: ---- 2
Copying: ---- 3
Code Snippet ---- 3
Substitution Layer (SubBytes): ---- 3
Code Snippet: ---- 4
ShiftRows: ---- 4
Code Snippet: ---- 4
Xtime: ---- 5
Code Snippet: ---- 5
MixColoumns: ---- 5
Code Snippet: ---- 6
Key Addition (AddRoundKey): ---- 6
Code Snippet: ---- 6
Encryption Workflow: ---- 7
Code Snipping: ---- 7
The Main Function: ---- 8
Code Snippet: ---- 8

Assignment Implementation (PRESENT):---------- 9
Key Expansion:---------- 9
Code Snippet: ---------- 10
Substitution Layer (sBoxLayer):---------- 11
Code Snippet: ---------- 11
Permutation Layer (pLayer): ---------- 12
Code Snippet: ---------- 13
Encryption Workflow:---------- 13
Code Snippet: ---------- 13
Main Function: ---------- 14
Code Snippet: ---------- 14
Validate the implementation using the Known Answer Tests (KATs): ---------- 15
AES:---------- 15
TEST1:---------- 15
Code Snippet: ---------- 15
Output:---------- 15
TEST2:---------- 16
Code Snippet: ---------- 16
Output:---------- 16
TEST3:---------- 17
Code Snippet: ---------- 17
Output:---------- 17
PRESENT:---------- 18
TEST1:---------- 18
Code Snippet: ---------- 18
Output:---------- 18
TEST2:---------- 19
Code Snippet: ---------- 19
Output:---------- 19
TEST3:---------- 19
Code Snippet: ---------- 20

Output:---------- 20

Assignment Implementation (AES):

Key Expansion:

Word Generation:

- AES-128 requires a 128-bit key (16 bytes) and expands it into 11 separate round keys (totaling 176 bytes).
- **Initial Step:** The first 4 words (W0 –W3) are copied directly from the 16-byte Master Key (Encryption Key).
- `(uint32_t)Encryption_Key[4*i]<<24)` what this line does is basically shifts the encryption key to the most significant byte of the 32bit word.Same with the rest of the keys created below like Shifts left by 16 bits which places it in the 2nd most significant byte position.

Code Snippet:

```
unsigned char KeyExpansion[176];
//As for in Key_Expansion function we have 176 bit array we can think of it as
44 words(176/4=44)
//First four words as W0,W1,W2,W3 for the encryption key
void Key_Expansion(unsigned char*Encryption_Key,unsigned char*RoundConst) {
    //AES-128 requires 11 sets of 16-byte round keys (11×16=176).

    //Temporary array of 44 words where each 4byte is a word
    //First 4 words just copying from the Encryption Key
    uint32_t word[44];
    for (int i=0; i<4;i++) {
        word[i] = ((uint32_t)Encryption_Key[4*i]<<24) |
                ((uint32_t)Encryption_Key[4*i + 1]<<16) |
                ((uint32_t)Encryption_Key[4*i + 2]<<8) |
                ((uint32_t)Encryption_Key[4*i + 3]);
    }
```

Transformation Logic

- Every 4th word (i%4==0) undergoes three following important steps
    - **Rotword**: Performs a circular left shift to scramble byte positions (`wordN << 8)` shifts everything left 1 byte and `(wordN >> 24) extracts the top byte and then we combine them.`
    - **SubWord:** Uses the S-Box lookup table to substitute each byte, providing non-linearity. `(sbox[(wordN >> 24) & 0xff] << 24)` what this do is it shifts desired byte to LSB position mask with 0xff (11111111 in binary) to extract 8 bits then use that value as an index into sbox. This happens for all 4 bytes.
    - **Round constant XOR:** XOR's the word with a unique round constant to prevent symmetry between rounds. (`wordN ^= (uint32_t)RoundConst[i/4 - 1] << 24 )` what this line does is it XORS's the most significant byte with a round constant which ensures each round key is unique.
    - Recursive Generation of words: All other words are generated by XORing the word from four positions ago with the previous word. (`word[i] = word[i-4] ^ wordN;)` This happens for every iteration not just multiplies of 4.

Code Snippet:

```
//Generating remaining 40 words using loop
    //AES Key Schedule logic
    //Each new word is created by XORing previos words to the word from 4
positions ago
    //word[i] = word[i-1] ^ word[i-4]
    for (int i = 4; i<44; i++) {
        uint32_t wordN = word[i-1];
        //Every 4th word(i%4==0) goes a special transformation
        if(i%4==0) {
            /*Next three steps is Rotword,Subword and Rcon
            Rotword-Rotate it to scramble positions
            Subword-Substitute to hide its mathematical relationsships
            XOR with Rcon-Gives specific round its own identity
            */
            //Rot word - Performs a circular shift on the 4 bytes(which is a
word)
            wordN = (wordN << 8) | (wordN >> 24);
            //Sub word looking for each byte in the sbox
            //substitute each byte using sbox look up table to provide non
linearity
            wordN = (sbox[(wordN >> 24) & 0xff] << 24) |
                    (sbox[(wordN >> 16) & 0xff] << 16) |
                    (sbox[(wordN >> 8) & 0xff] << 8) |
```

```
                    (sbox[(wordN & 0xff)]);
            //Round constant XOR
            //XORs the leftmost byte with a specific value from RoundConst
to ensure each round key is unique.
            //i/4-1 because i starts at 4 for the 1st constant
            wordN ^= (uint32_t)RoundConst[i/4 - 1] << 24;
        }
        word[i] = word[i-4]^wordN;
    }
```

Copying:

- Now copying that temporary array back into KeyExpansion.
- `KeyExpansion[4*i] = (word[i] >> 24) & 0xFF;.` what this line does basically moves top 8 bits in to the lowest positions and mask them with 0xFF to keep the 8 bits and stores in the first byte slot same for the remaining as well.

Code Snippet

```
for (int i = 0; i < 44; i++) {           // 44 words
        KeyExpansion[4*i]     = (word[i] >> 24) & 0xFF;
        KeyExpansion[4*i + 1] = (word[i] >> 16) & 0xFF;
        KeyExpansion[4*i + 2] = (word[i] >> 8)  & 0xFF;
        KeyExpansion[4*i + 3] =  word[i] & 0xFF;
    }
```

Substitution Layer (SubBytes):

- The SubBytes function is the primary source of confusion in the cipher.It performs non-linear byte substitution on the 4×4 state matrix.
- Every byte in the 4×4 State matrix is replaced by a value from the S-Box.This make relationship between input and output non-linear.

Code Snippet:

```
//Added a new function for subsytes as it can make too cluttter in the main
function
/*
Each byte is swapped for another using the S-Box.
This makes the relationship between input and output non-linear
*/
void SubBytes(unsigned char State[4][4]) {
```

```
    for(int i=0; i<4; i++) {
        for(int j=0; j<4; j++) {
            State[i][j] = sbox[State[i][j]];
        }
    }
}
```

ShiftRows:

- ShiftRow provides **diffusion** across the state horizontally.
- **Cyclic Shifting:** Row 0 is unchanged. Row 1 is shifted left by 1 byte, Row 2 by 2 bytes, and Row 3 by 3 bytes. Basically ShiftRows shifts each row by its row index.
- After Copying Row to Temporary Array then we perform the circular shift
- This ensures that bytes that were previously in the same column are spread into different columns for the next mathematical transformation.
- SubBytes gives non-linearity. Shiftrows spreads nonlinearity across columns.

Code Snippet:

```
//Shifting rows
/*
Bytes are shifted horizontally.
This ensures that bytes in the same column get moved to different columns for
the next step.
*/
void ShiftRow(unsigned char State[4][4]) {
    for(int i=0;i<4;i++) {
        unsigned char temp[4];
        for(int j=0; j<4; j++) {
            temp[j] = State[i][j];
        }
        for (int j=0; j<4; j++) {       // shift row left by i
            State[i][j] = temp[(j + i) % 4];
        }
    }
}
```

Xtime:

- This function implements multiplication by {02} in the Galois Field GF(28). It performs a left shift; if the original byte had its high bit set (overflow), it XORs the result with the polynomial constant 0x1b. It is heavily used inside MixColumns.

Code Snippet:

```
//xtime function
//The xtime function performs a left shift and, if the result overflows, it
performs an XOR with the value 0x1b.
unsigned char xtime(unsigned char b) {
//single byte consists of 8 bits which is 0x80 inn hexa decimal
//checking is the high bit 0x80 is set or not(0x80 in binary is 10000000)
//shifting it left causes that 1 to overflow(MSB)
    if(b & 0x80) {
        return ((b<<1) ^0x1b);
    }else {
        return (b<<1);
    }
}
```

MixColoumns:

- Each column of the State is treated as a polynomial and multiplied by a fixed matrix. A change in just one byte of the column will, after this step, affect all four bytes of that column.
- It mathematically combines four bytes in each column. Because of this, a change in one byte of the plaintext will eventually affect every single byte of the ciphertext.
- It works Column by column. Each column (4 bytes) is transformed into a new column.The transformation is a matrix multiplication in GF(2^8).
- `State[0][j] = xtime(temp[0]) ^ (xtime(temp[1]) ^ temp[1]) ^ temp[2] ^ temp[3];` what this line is doing is :
    - xtime(x) = 2 × x
    - (xtime(x) ^ x) = (2 × x) ^ x = 3 × x
    - Multiplication by 1 = just x
    - Addition in GF(2^8) = XOR (^) (Same for the rest)

Code Snippet:

```
//MixColumns
/*
It mathematically combines the four bytes in each column.
Because of this, a change in one byte of the plaintext will eventually affect
```

```
every single byte of the ciphertext.
*/
/*
AES uses Galois Field GF(28) arithmetic. In this field:
Addition is just the XOR (^) operation.
Multiplication is specialized. Multiplying by 1 does nothing,
but multiplying by 2 (binary 10) requires a special function often called
xtime(created above to handle multiplication by 2)
*/
void MixColoumn(unsigned char State[4][4]) {
    for(int j=0;j<4;j++) {
        unsigned char temp[4];
        for(int i=0;i<4;i++) {
            temp[i] = State[i][j];
        }
        State[0][j] = xtime(temp[0]) ^ (xtime(temp[1]) ^ temp[1]) ^ temp[2] ^
temp[3];
        State[1][j] = temp[0] ^ xtime(temp[1]) ^ (xtime(temp[2]) ^ temp[2]) ^
temp[3];
        State[2][j] = temp[0] ^ temp[1] ^ xtime(temp[2]) ^ (xtime(temp[3]) ^
temp[3]);
        State[3][j] = (xtime(temp[0]) ^ temp[0]) ^ temp[1] ^ temp[2] ^
xtime(temp[3]);
    }
}
```

Key Addition (AddRoundKey):

- This is the only stage where the secret key is actually introduced into the data.
- **Bitwise XOR:** The current 16-byte State is XORed with the corresponding 16-byte segment of the KeyExpansion array for that specific round.
- There is **no special math**, no finite field multiplication , just XOR.But this is the step that actually injects the secret key into the cipher.
- All other AES operations are fixed mathematical operations they do no depend on secret key. Only AddRoundKey injects key material Without it, AES becomes a fixed permutation completely insecure.

Code Snippet:

```
//Function for the Addroundkey
/*
At the end of every round, we XOR the result with a brand-new 16-byte key
generated by the Key_Expansion.
*/
```

```
void AddRoundKey(unsigned char State[4][4], int round) {
    int temp = round * 16;
    for (int j = 0; j < 4; j++) {
        for (int i = 0; i < 4; i++) {
            State[i][j] ^= KeyExpansion[temp + (j * 4 + i)];
        }
    }
}
```

Encryption Workflow:

- This function coordinates the previous sections in the order specified by the AES standard.
- **Initial Round:** Begins with AddRoundKey (Round 0).
- **Main Rounds (1–9):** Executes a full sequence of SubBytes, ShiftRows, MixColumns, and AddRoundKey.
- **Final Round (Round 10):** Executes SubBytes, ShiftRows, and AddRoundKey but ignore Mixcolumns. This is done to make the algorithm more efficient without compromising security.

Code Snipping:

```
//Created a new function encrypt to test both plaintexts at once
void Encrypt(unsigned char State[4][4]) {
        //Initial Round(0)
    AddRoundKey(State, 0);

    //Round (1 to 9)
    for(int round =1 ; round < 10; round++) {
        SubBytes(State);
        ShiftRow(State);
        MixColoumn(State);
        AddRoundKey(State, round);
    }

    //Final Round(10)
    SubBytes(State);
    ShiftRow(State);
    AddRoundKey(State, 10);
}
```

The Main Function:

- AES specifies that the 16 bytes of data are filled into the 4×4 matrix column-by-column. Code correctly implements this mapping: State[i][j] = Plaintext[j * 4 + i].
- The main function demonstrates the **Avalanche Effect** by running two plaintexts that differ by only a single bit.

Code Snippet:

```
int main() {
    Key_Expansion(Encryption_Key, RoundConst);
    //STATE MATRIX
    //The 16 bytes are filled into the matrix column by column.
    //Byte 0 goes to row 0, col 0 ;Byte 1 goes to row 1, col 0.
    //the formula to find the 1D index from 2D coordinates (i,j) is Index =
(Coloum *4) + row
    unsigned char State[4][4];
//-----------------TEST Plaintext 1------------
    Key_Expansion(Encryption_Key, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = Plaintext_1[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);
//-----------------TEST Plaintext 2-------------
    Key_Expansion(Encryption_Key, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = Plaintext_2[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);

    return 0;

}
```

Assignment Implementation (PRESENT):

Key Expansion:

- The Key Expansion process generates 32 unique 64-bit round keys from a single 80-bit master key. In your implementation, this is stored in the keyExpansion array (256 bytes total).
    - PRESENT-80 is a **lightweight block cipher,**Block size: 64 bits,Key size: 80 bits,Number of rounds: 31,Round keys: 32 × 64 bits → 256 bytes total
- **Initialization:** The first round key consists of the leftmost 64 bits of the original 80-bit key.
    - `memcpy(keyExpansion, key, 8)-` copying first round key after a temporary array 'key' is created.
    - `for(int round = 1; round < 32; round++)-` Generates **round keys 1 to 31** (32 total). Each round key = 8 bytes.
    - Algorithm uses: 61-bit **circular left rotation** (done via 19-bit right shift here), **S-box substitution** on top 4 bits, **Round counter XOR** to break symmetry.
- **Rotation (61-bit Circular Shift):** The entire 80-bit key register is rotated. Your code efficiently achieves a 61-bit left shift by performing a 19-bit right shift using a temporary array.
    - `(key[(i + 8) % 10] >> 3)` This line takes bytes from two positions back and move its bits to 3 positions right.this fills bottom 5 bits on the byte.
    - `(key[(i + 7) % 10] << 5)` This line takes three bits that are falling off the right side of the byte three positions back and puts them all the way to the left side of the current byte.
    - Combine them to a new byte. This ensures all 80 bits are circularly rotated.
- **S-Box Substitution:** The leftmost 4 bits of the key (bits 79-76) are passed through the S-box. This ensures the round keys are generated non-linearly.
    - `memcpy(key, temp, 10);` Once the key is updated here new key contains a rotated 80bit value.
    - Once the top 4 bits are extracted it is passes through sbox(Non linear substitution). `top_bits = sbox[top_bits];`
    - `key[0] = (key[0] & 0x0F) | (top_bits << 4);` Keeps the bottom four bits and puts substituted top bits back.This adds non linearity to each round key.
- **Round Counter XOR:** To break symmetry and ensure each round is distinct, bits 19-15 of the key are XORed with the current round index.

- `memcpy(keyExpansion + (round * 8), key, 8);` Copying Roundkey into Key expansion.
- Each round key = **8 bytes,** Round n = bytes n*8 to (n*8 + 7) in keyExpansion.
- key[0 to 7] -round key and key[8 to 9] - used only for next round calculations.

Code Snippet:

```
/*
    Present_80 encryption process has 31 rounds plus one final round, each
round it uses 64bit round key
    IT has total of 32 rounds so total size will be 32*8=256
*/
unsigned char keyExpansion[256];
void Key_Expansion(unsigned char *EncryptionKey) {
    //For circular shift instead of doing left shift of 61 bits .
    //doing the right shift of 19 bits.
    uint8_t key[10];
    memcpy(key,EncryptionKey,10);
    uint8_t temp[10];

    memcpy(keyExpansion,key,8); //first key is the leftmost 64 bits of the
orginal key
    //SBOX with the top bits and XOR with a round counter
    //since each roundkey is 8 bytes long below offset is taken.
    for(int round=1; round<32; round++) {

        /*
        (key[(i + 8) % 10] >> 3) takes bytes from two positions back and move
its bits to 3 positions right.this fills bottom 5 bits on the byte.
        (key[(i + 7) % 10] << 5) takes three bits that are falling off the
right side of the byte three positions back and puts them all the
        way to the left side of the current byte.
        */
        for(int i=0; i<10; i++) {
            temp[i] = (key[(i + 8) % 10] >> 3) |
                    (key[(i + 7) % 10] << 5);
        }
        memcpy(key,temp,10);
        uint8_t top_bits = key[0] >>4;
        //apply sbox
        top_bits = sbox[top_bits];
        //clear top bits in key and putting the new top bits
```

```
        //using 0x0f because in binary it is 00001111 so it keeps bottom bits
and clear top bits
        key[0] = (key[0] & 0x0F) | (top_bits << 4);
        //we need to xor(as algorithm spficies) the bits in key[0] - 19 to 15
to give a unique stamp each round
        //It is used to break symmetry between rounds
        //key[7] - 23-16, key[8] - 15-8
        /*round >>1 - what is does is
        round = abcde after the logic operation it becomes 0abcd now we have
the upper four bits
        round = abcde after logic operation round & 1 → e and then after e<<7
e0000000
        */
        key[7] ^= (round >>1);
        key[8] ^= (round & 0x01) << 7;
        //copying the key back to the keyExpansion
        memcpy(keyExpansion + (round*8),key,8);
    }

}
```

Substitution Layer (sBoxLayer):

- The sBoxLayer provides the primary "confusion" by replacing state values with non-linear substitutes.
    - Present uses a 4-bit S-box, the code splits each 8-bit byte into two (top and bottom 4 bits).
        - `uint8_t top = State[i] >> 4;` Shifts bits 7–4 into position 3–0 and Extracts the upper 4 bits
        - `uint8_t bottom = State[i] & 0x0F;` Masks with 00001111 and Extracts the lower 4 bits.
    - `top = sbox[top]; and bottom = sbox[bottom];` Both of them substituted using a sbox. Which introduces non linearity.
        - `top = top << 4;` moves top back to bits 7–4
        - `State[i] = top | bottom` Combines top and bottom into one byte.

Code Snippet:

```
//Sbox Layer
// 4-bit S-Box substitution
void sboxLayer(uint8_t State[8]) {
    for(int i=0; i<8; i++) {
```

```
        //get top four bits and get bottom four bits and combine them
        uint8_t top = State[i] >> 4;//shifts 7-4 to 3-0
        uint8_t bottom = State[i] & 0x0F; //mask out 7-4 bits
        top = sbox[top];
        bottom = sbox[bottom];
        top = top << 4;
        State[i] = top | bottom;
    }
}
```

Permutation Layer (pLayer):

- The **pLayer** is a bit-level permutation that provides "diffusion" by moving bits to entirely new positions.
- **Mathematical Formula:** Each bit at position i is moved to a new position defined by the formula $P(i)=(16\times i)(\mathrm{mod}63)$ (with bit 63 remaining at 63).
- Implementation:
    - Once a temporary array created to hold the permutated state.
    - Loop through all 64 bits
    - `int bit_pos = (16*i) % 63;`
      `if(i == 63) bit_pos = 63;` bit 63 maps to 63 itself . What it does is Spreads bits of the state across bytes
    - `uint8_t bit_val = (State[7 - (i / 8)] >> (i % 8)) & 0x01;`
        - Bits are numbered **0–63**, with **bit 0 = LSB of State[7]**
        - Byte index: 7 - (i / 8) → counts bytes from most significant (State[0]) to least (State[7])
        - Bit within byte: i % 8
        - >> (i % 8) & 0x01 → extracts the value of that bit (0 or 1)
    - `if(bit_val) {`
      `temp[7 - (bit_pos / 8)] |= (1 << (bit_pos % 8));`
      `}`
        - If the bit is 1, place it at the new location bit_pos
        - Byte index in temp: 7 - (bit_pos / 8)
        - Bit within byte: bit_pos % 8
        - "|= "to set the bit without affecting other bits
    - `memcpy(State, temp, 8);` After all bits are rearranged, copying temp back into original state.

Code Snippet:

```
        //pLayer
        // Bit permutation
```

```
void pLayer(uint8_t State[8]) {
    uint8_t temp[8] = {0, 0, 0, 0, 0, 0, 0, 0}; //temporary state
    for (int i=0; i<64; i++) {
        //Finding bit position
        //The permutation formula: P(i) = (16 * i) % 63 (except
for bit 63)
        int bit_pos = (16*i)%63;
        if(i==63) {
            bit_pos = 63;
        }
        //Finding the bit value
        //Bits are numbered 0-63 (0 is least significant bit of
State[7])
        //To get bit i: byte index = 7 - (i / 8), bit position =
i % 8
        uint8_t bit_val = (State[7 - (i / 8)] >> (i % 8)) & 0x01;
        //2. If the bit is 1, set the bit in the temp state at
'bit_pos'
        if (bit_val) {
            temp[7 - (bit_pos / 8)] |= (1 << (bit_pos % 8));
        }
    }
    // Copying the permuted state back to the original State
array
    memcpy(State, temp, 8);
}
```

Encryption Workflow:

- **31 Main Rounds:** For rounds 0 through 30, the state undergoes a full cycle of addRoundkey, sboxLayer, and pLayer
- **Final Round:** After the 31 iterative rounds, a final addRoundkey is performed using the 32nd round key. This final XOR ensures that the last permutation and substitution cannot be easily reversed without the key.

Code Snippet:

```
//For Encryption
void Encrypt(uint8_t State[8]) {
    for(int round = 0; round < 31; round++) {
        addRoundkey(State, round);
        sboxLayer(State);
        pLayer(State);
```

```
    }
    //Final Round key
    addRoundkey(State, 31);
}
```

Main Function:

- Tests happen in the main function according to the given plain text and Encryption keys.

Code Snippet:

```
    //-------Plaintext 1--------
    Key_Expansion(EncryptionKey_1);
    Encrypt(Plaintext_1);
    print(Plaintext_1);

    //-------Plaintext 2--------
    Key_Expansion(EncryptionKey_2); // Must re-expand whenever the key
changes
    Encrypt(Plaintext_2);
    print(Plaintext_2);

    //-------Plaintext 3--------
    Key_Expansion(EncryptionKey_3);
    Encrypt(Plaintext_3);
    print(Plaintext_3);

    //-------Plaintext 4--------
    Key_Expansion(EncryptionKey_4);
    Encrypt(Plaintext_4);
    print(Plaintext_4);

    return 0;
}
```

Validate the implementation using the Known Answer Tests (KATs):

AES:

TEST1:

- Encryption key : 00000000000000000000000000000000
- Plaintext : f34481ec3cc627bacd5dc3fb08f273e6
- Cpihertext : 0336763e966d92595a567cc9ce537f5e
- Source : AESAVS Appendix B.1

Code Snippet:

```
/*
------------------AES-128 Validation (ECB)
-----Test1
Test Result 1: All Zeros (GFSbox Test)
Expected Cyphertext - 0336763e966d92595a567cc9ce537f5e
AESAVS Appendix B.1(location)
*/
unsigned char ECB_EncryptionKey_1[16] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};
unsigned char ECB_Plaintext_1[16] =
{0xf3,0x44,0x81,0xec,0x3c,0xc6,0x27,0xba,0xcd,0x5d,0xc3,0xfb,0x08,0xf2,0x73,0
xe6};

/*
------------------AES-128 Validation (ECB) Test1*/
    Key_Expansion(ECB_EncryptionKey_1, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = ECB_Plaintext_1[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);

/*
```

Output:

Line 3 is the output for the above plain text

```
FF 0B 84 4A 08 53 BF 7C 69 34 AB 43 64 14 8F B9
61 2B 89 39 8D 06 00 CD E1 16 22 7C E7 24 33 F0
03 36 76 3E 96 6D 92 59 5A 56 7C C9 CE 53 7F 5E
3A D7 8E 72 6C 1E C0 2B 7E BF E9 2B 23 D9 EC 34
F0 31 D4 D7 4F 5D CB F3 9D AA F8 CA 3A F6 E5 27
```

TEST2:

- Encryption key : 00000000000000000000000000000000
- Plaintext : 80000000000000000000000000000000
- Cpihertext : 3ad78e726c1ec02b7ebfe92b23d9ec34
- Source : AESAVS Appendix D.1

Code Snippet:

```
/*-----Test2
Test Result 2: Variable Text Test
Expected Cyphertext - 3ad78e726c1ec02b7ebfe92b23d9ec34
AESAVS Appendix D.1(location)
*/
unsigned char ECB_EncryptionKey_2[16] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};
unsigned char ECB_Plaintext_2[16] =
{0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};

/*
-------------------AES-128 Validation (ECB) Test2*/
    Key_Expansion(ECB_EncryptionKey_2, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = ECB_Plaintext_2[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);

/*
```

Output:

Line 4 is the output for the above plain text

```
FF 0B 84 4A 08 53 BF 7C 69 34 AB 43 64 14 8F B9
61 2B 89 39 8D 06 00 CD E1 16 22 7C E7 24 33 F0
03 36 76 3E 96 6D 92 59 5A 56 7C C9 CE 53 7F 5E
3A D7 8E 72 6C 1E C0 2B 7E BF E9 2B 23 D9 EC 34
F0 31 D4 D7 4F 5D CB F3 9D AA F8 CA 3A F6 E5 27
```

TEST3:

- Encryption key : 00000000000000000000000000000000
- Plaintext : e00000000000000000000000000000000
- Cpihertext : f031d4d74f5dcbf39daaf8ca3af6e527
- Source : AESAVS Appendix D.1 on Page 20

Code Snippet:

```
/*-----Test3
Test Result 3: Avalanche Effect Validation
Expected Cyphertext - f031d4d74f5dcbf39daaf8ca3af6e527
Appendix D.1 on Page 20.(location)
*/
unsigned char ECB_EncryptionKey_3[16] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};
unsigned char ECB_Plaintext_3[16] =
{0xe0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};

/*
-------------------AES-128 Validation (ECB) Test3*/
    Key_Expansion(ECB_EncryptionKey_3 ECB_EncryptionKey_1, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = ECB_Plaintext_3[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);
```

Output:

Line 5 is the output for the above plain text


```
FF 0B 84 4A 08 53 BF 7C 69 34 AB 43 64 14 8F B9
61 2B 89 39 8D 06 00 CD E1 16 22 7C E7 24 33 F0
03 36 76 3E 96 6D 92 59 5A 56 7C C9 CE 53 7F 5E
3A D7 8E 72 6C 1E C0 2B 7E BF E9 2B 23 D9 EC 34
F0 31 D4 D7 4F 5D CB F3 9D AA F8 CA 3A F6 E5 27
```

PRESENT:

TEST1:

- Encryption key : 01000000000000000000
- Plaintext : 0000000000000000
- Cpihertext : FF1DF4571108C38B
- Source : Set 1, vector# 7 (GITHUB repo From assignment description)

Code Snippet:

```
/*
-------------------PRESENT-80 Validation (ECB)
-----Test1
Test Result 1
Expected Cyphertext - FF1DF4571108C38B
Github repo from assignment(location)- Set 1, vector#  7:
*/
unsigned char ECB_EncryptionKey_1[10] =
{0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
unsigned char ECB_Plaintext_1[8] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

/*
-------------------PRESENT-80 Validation (ECB) Test1*/
    Key_Expansion(ECB_EncryptionKey_1);
    Encrypt(ECB_Plaintext_1);
    print(ECB_Plaintext_1);
```

Output:

Line 4 is the output above plaintext

```
Ciphertext: E72C46C0F5945049
Ciphertext: A112FFC72F68417B
Ciphertext: 3333DCD3213210D2
Ciphertext: FF1DF4571108C38B
Ciphertext: 0FC6284AABCD6AFA
Ciphertext: EA024714AD5C4D84
```

TEST2:

- Encryption key : EAEAEAEAEAEAEAEAEAEA
- Plaintext : EAEAEAEAEAEAEAEA

- Cpihertext : 0FC6284AABCD6AFA
- Source : Set 3, vector#234 (GITHUB repo From assignment description)

Code Snippet:

```
/*-----Test2
Test Result 2
Expected Cyphertext - 0FC6284AABCD6AFA
Github repo from assignment(location)- Set 3, vector#234:
*/
unsigned char ECB_EncryptionKey_2[10] =
{0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA};
unsigned char ECB_Plaintext_2[8] = {0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA};

/*
------------------PRESENT-80 Validation (ECB) Test2*/
    Key_Expansion(ECB_EncryptionKey_2);
    Encrypt(ECB_Plaintext_2);
    print(ECB_Plaintext_2);
/*
```

Output:

Line 5 is the output above plaintext

```
Ciphertext: E72C46C0F5945049
Ciphertext: A112FFC72F68417B
Ciphertext: 3333DCD3213210D2
Ciphertext: FF1DF4571108C38B
Ciphertext: 0FC6284AABCD6AFA
Ciphertext: EA024714AD5C4D84
```

TEST3:

- Encryption key : 2BD6459F82C5B300952C
- Plaintext : 0CEDD50400427E41
- Cpihertext : EA024714AD5C4D84
- Source : Set 8, vector#1 (GITHUB repo From assignment description)

Code Snippet:

```
/*-----Test3
Test Result 3
Expected Cyphertext - EA024714AD5C4D84
Github repo from assignment(location)- Set 8, vector#  1:
*/
unsigned char ECB_EncryptionKey_3[16] =
```

```
{0x2B,0xD6,0x45,0x9F,0x82,0xC5,0xB3,0x00,0x95,0x2C};
unsigned char ECB_Plaintext_3[16] =
{0x0C,0xED,0xD5,0x04,0x00,0x42,0x7E,0x41};

/*
------------------PRESENT-80 Validation (ECB) Test3*/
    Key_Expansion(ECB_EncryptionKey_3);
    Encrypt(ECB_Plaintext_3);
    print(ECB_Plaintext_3);
```

Output:

Line 6 is the output above plaintext

```
Ciphertext: E72C46C0F5945049
Ciphertext: A112FFC72F68417B
Ciphertext: 3333DCD3213210D2
Ciphertext: FF1DF4571108C38B
Ciphertext: 0FC6284AABCD6AFA
Ciphertext: EA024714AD5C4D84
```

COMPARISION (AESvsPRESENT):

Feature	AES-128	PRESENT-80
Block Size	128 bits	64 bits
Key Size	128 bits	80 bits
Binary Size	**Larger.** Due to the 256-byte S-Box and 176-byte expanded key array.	**Smaller.** Due to the 16-byte S-Box and smaller state array.
Algorithm Structure	Substitution-Permutation Network (SPN)	Substitution-Permutation Network (SPN)
Number of Rounds	10 Rounds	31 Rounds
Code Size (Software)	**Large.** Requires 256-byte S-Box and complex matrix multiplication (MixColumns).	**Small.** Uses a tiny 16-nibble S-Box and bitwise permutations (pLayer).
Execution	**Fast (Software).** Optimized for 32/64-bit processors using word-based operations.	**Slow (Software).** Bit-level permutations are computationally expensive in software.
Execution Time	**Faster.** AES is optimized for 32-bit software and will likely process more bits per second.	**Slower.** PRESENT's bit-level permutations (pLayer) are mathematically simple but slow to simulate bit-by-bit in C.

Hardware Efficiency	**Moderate.** Requires significant gate area for MixColumns and S-Boxes.	**Ultra-Lightweight.** Designed specifically for minimal gate count in hardware.
Security Strenght	**High.** High resistance to all known attacks; standard for government/bank data.	**Normal.** Sufficient for low-value data but has a smaller security margin than AES.
Avalanche Effect	**Very Strong.** A 1-bit change affects ~58/128 bits in one execution.	**Strong.** Achieves full diffusion after several rounds, though slower than AES.
Application Scenarios	Web traffic (HTTPS), Disk encryption, WiFi (WPA2), Mobile apps.	RFID tags, IoT sensor nodes, Smart cards, medical implants.

APPENDIX:

AES.H:

```
#ifndef AES_H
#define AES_H
#include <stdint.h>
#include <string.h>
#include <time.h>


unsigned char Encryption_Key[16] =
{0x0f,0x15,0x71,0xc9,0x47,0xd9,0xe8,0x59,0x0c,0xb7,0xad,0xd6,0xaf,0x7f,0x67,0
x98};
unsigned char Plaintext_1[16] =
{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0
x10};
unsigned char Plaintext_2[16] =
{0x00,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0
x10};

/*
------------------AES-128 Validation (ECB)
-----Test1
Test Result 1: All Zeros (GFSbox Test)
```

```
Expected Cyphertext - 0336763e966d92595a567cc9ce537f5e
AESAVS Appendix B.1(location)
*/
unsigned char ECB_EncryptionKey_1[16] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};
unsigned char ECB_Plaintext_1[16] =
{0xf3,0x44,0x81,0xec,0x3c,0xc6,0x27,0xba,0xcd,0x5d,0xc3,0xfb,0x08,0xf2,0x73,0
xe6};

/*-----Test2
Test Result 2: Variable Text Test
Expected Cyphertext - 3ad78e726c1ec02b7ebfe92b23d9ec34
AESAVS Appendix D.1(location)
*/
unsigned char ECB_EncryptionKey_2[16] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};
unsigned char ECB_Plaintext_2[16] =
{0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};

/*-----Test3
Test Result 3: Avalanche Effect Validation
Expected Cyphertext - f031d4d74f5dcbf39daaf8ca3af6e527
Appendix D.1 on Page 20.(location)
*/
unsigned char ECB_EncryptionKey_3[16] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};
unsigned char ECB_Plaintext_3[16] =
{0xe0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0
x00};

/*The constants for rounds 1-10 are:
01,02,04,08,10,20,40,80,1b,36 (in hexadecimal).*/
unsigned char RoundConst[10] =  {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
0x80, 0x1b, 0x36};

static const uint8_t sbox[256] = {
    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
};


//Functions Declarations
void Key_Expansion(unsigned char*Encryption_Key,unsigned char*RoundConst);
void SubBytes(unsigned char State[4][4]);
void ShiftRow(unsigned char State[4][4]);
unsigned char xtime(unsigned char b);
void MixColoumn(unsigned char State[4][4]);
void AddRoundKey(unsigned char State[4][4], int round);
void Encrypt(unsigned char State[4][4]);
void print(unsigned char State[4][4]) ;

#endif
```

AES.C:

```
#include <stdio.h>
#include "AES.h"

/* AES-128 ENCRYPTION WORKFLOW
  -------------------------
  1. KEY EXPANSION:
      - Expand 16-byte Master Key into 11 Round Keys (176 bytes total).
       - Uses: RotWord, SubWord (S-Box), and Rcon XOR.
  2. INITIAL ROUND:
       - AddRoundKey(Round 0): XOR Plaintext with the original Master Key.
  3. MAIN ROUNDS (1 to 9):
       - SubBytes: Non-linear byte substitution using S-Box.
       - ShiftRows: Cyclic shift of rows (Row 0: 0, Row 1: 1, Row 2: 2, Row
3: 3).
       - MixColumns: Matrix multiplication in Galois Field GF(2^8).
       - AddRoundKey: XOR State with the current Round Key.
  4. FINAL ROUND (Round 10):
       - SubBytes -> ShiftRows -> (No MixColumns) -> AddRoundKey.
 */

unsigned char KeyExpansion[176];
//As for in Key_Expansion function we have 176 bit array we can think of it
as 44 words(176/4=44)
//First four words as W0,W1,W2,W3 for the encryption key
void Key_Expansion(unsigned char*Encryption_Key,unsigned char*RoundConst) {
    //AES-128 requires 11 sets of 16-byte round keys (11×16=176).

    //Temporary array of 44 words where each 4byte is a word
    //First 4 words just copying from the Encryption Key
    uint32_t word[44];
    for (int i =0; i<4;i++) {
        word[i] = ((uint32_t)Encryption_Key[4*i]<<24) |
                  ((uint32_t)Encryption_Key[4*i + 1]<<16) |
                  ((uint32_t)Encryption_Key[4*i + 2]<<8) |
                  ((uint32_t)Encryption_Key[4*i + 3]);
    }

    //Generating remaining 40 words using loop
    //AES Key Schedule logic
```

```
    //Each new word is created by XORing previos words to the word from 4
positions ago
    //word[i] = word[i-1] ^ word[i-4]
    for (int i = 4; i<44; i++) {
        uint32_t wordN = word[i-1];
        //Every 4th word(i%4==0) goes a special transformation
        if(i%4==0) {
            /*Next three steps is Rotword,Subword and Rcon
            Rotword-Rotate it to scramble positions
            Subword-Substitute to hide its mathematical relationssships
            XOR with Rcon-Gives specific round its own identity
            */
            //Rot word - Performs a circular shift on the 4 bytes(which is a
word)
            wordN = (wordN << 8) | (wordN >> 24);
            //Sub word looking for each byte in the sbox
            //substitute each byte using sbox look up table to provide non
linearity
            wordN = (sbox[(wordN >> 24) & 0xff] << 24) |
                    (sbox[(wordN >> 16) & 0xff] << 16) |
                    (sbox[(wordN >> 8) & 0xff] << 8) |
                    (sbox[(wordN & 0xff)]);
            //Round constant XOR
            //XORs the leftmost byte with a specific value from RoundConst to
ensure each round key is unique.
            //i/4-1 because i starts at 4 for the 1st constant
            wordN ^= (uint32_t)RoundConst[i/4 - 1] << 24;
        }
        word[i] = word[i-4]^wordN;
    }
    //copying in to KeyExpansion using memcpy from the temporary array word
    //memcpy(KeyExpansion, word, 176);
    for (int i = 0; i < 44; i++) {           // 44 words
        KeyExpansion[4*i]     = (word[i] >> 24) & 0xFF;
        KeyExpansion[4*i + 1] = (word[i] >> 16) & 0xFF;
        KeyExpansion[4*i + 2] = (word[i] >> 8)  & 0xFF;
        KeyExpansion[4*i + 3] =  word[i] & 0xFF;
    }
}


//Added a new function for subsytes as it can make too cluttter in the main
function
```

```
/*
Each byte is swapped for another using the S-Box.
This makes the relationship between input and output non-linear
*/
void SubBytes(unsigned char State[4][4]) {
    for(int i=0; i<4; i++) {
        for(int j=0; j<4; j++) {
            State[i][j] = sbox[State[i][j]];
        }
    }
}
//Shifting rows
/*
Bytes are shifted horizontally.
This ensures that bytes in the same column get moved to different columns for
the next step.
*/
void ShiftRow(unsigned char State[4][4]) {
    for(int i=0;i<4;i++) {
        unsigned char temp[4];
        for(int j=0; j<4; j++) {
            temp[j] = State[i][j];
        }
        for (int j=0; j<4; j++) {       // shift row left by i
            State[i][j] = temp[(j + i) % 4];
        }
    }
}
//xtime function
//The xtime function performs a left shift and, if the result overflows, it
performs an XOR with the value 0x1b.
unsigned char xtime(unsigned char b) {
//single byte consists of 8 bits which is 0x80 inn hexa decimal
//checking is the high bit 0x80 is set or not(0x80 in binary is 10000000)
//shifting it left causes that 1 to overflow(MSB)
    if(b & 0x80) {
        return ((b<<1) ^0x1b);
    }else {
        return (b<<1);
    }
}
```

```
//MixColumns
/*
It mathematically combines the four bytes in each column.
Because of this, a change in one byte of the plaintext will eventually affect
every single byte of the ciphertext.
*/
/*
AES uses Galois Field GF(28) arithmetic. In this field:
Addition is just the XOR (^) operation.
Multiplication is specialized. Multiplying by 1 does nothing,
but multiplying by 2 (binary 10) requires a special function often called
xtime(created above to handle multiplication by 2)
*/
void MixColoumn(unsigned char State[4][4]) {
    for(int j=0;j<4;j++) {
        unsigned char temp[4];
        for(int i=0;i<4;i++) {
            temp[i] = State[i][j];
        }
        State[0][j] = xtime(temp[0]) ^ (xtime(temp[1]) ^ temp[1]) ^ temp[2] ^
temp[3];
        State[1][j] = temp[0] ^ xtime(temp[1]) ^ (xtime(temp[2]) ^ temp[2]) ^
temp[3];
        State[2][j] = temp[0] ^ temp[1] ^ xtime(temp[2]) ^ (xtime(temp[3]) ^
temp[3]);
        State[3][j] = (xtime(temp[0]) ^ temp[0]) ^ temp[1] ^ temp[2] ^
xtime(temp[3]);
    }
}


//Function for the Addroundkey
/*
At the end of every round, we XOR the result with a brand-new 16-byte key
generated by the Key_Expansion.
*/
void AddRoundKey(unsigned char State[4][4], int round) {
    int temp = round * 16;
    for (int j = 0; j < 4; j++) {
        for (int i = 0; i < 4; i++) {
            State[i][j] ^= KeyExpansion[temp + (j * 4 + i)];
        }
    }
```

```
}
//Created a new function encrypt to test both plaintexts at once
void Encrypt(unsigned char State[4][4]) {
        //Initial Round(0)
    AddRoundKey(State, 0);

    //Round (1 to 9)
    for(int round =1 ; round < 10; round++) {
        SubBytes(State);
        ShiftRow(State);
        MixColoumn(State);
        AddRoundKey(State, round);
    }

    //Final Round(10)
    SubBytes(State);
    ShiftRow(State);
    AddRoundKey(State, 10);
}

//For printing both ciphertexts
void print(unsigned char State[4][4]) {

printf("Ciphertext: ");
    for (int j = 0; j < 4; j++) {      // Iterate through columns
        for (int i = 0; i < 4; i++) {  // Iterate through rows
            printf("%02X ", State[i][j]);
        }
    }
    printf("\n");
}


int main() {
    Key_Expansion(Encryption_Key, RoundConst);
    //STATE MATRIX
    //The 16 bytes are filled into the matrix column by column.
    //Byte 0 goes to row 0, col 0 ;Byte 1 goes to row 1, col 0.
    //the formula to find the 1D index from 2D coordinates (i,j) is Index =
(Coloum *4) + row
    unsigned char State[4][4];
//-----------------TEST Plaintext 1-------------
    Key_Expansion(Encryption_Key, RoundConst);
```

```
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = Plaintext_1[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);
//-----------------TEST Plaintext 2-------------
    Key_Expansion(Encryption_Key, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = Plaintext_2[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);

/*
-------------------AES-128 Validation (ECB) Test1*/
    Key_Expansion(ECB_EncryptionKey_1, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = ECB_Plaintext_1[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);

/*
-------------------AES-128 Validation (ECB) Test2*/
    Key_Expansion(ECB_EncryptionKey_2, RoundConst);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            State[i][j] = ECB_Plaintext_2[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);

/*
-------------------AES-128 Validation (ECB) Test3*/
    Key_Expansion(ECB_EncryptionKey_3, RoundConst);
    for(int i = 0; i < 4; i++) {
```

```
        for(int j = 0; j < 4; j++) {
            State[i][j] = ECB_Plaintext_3[j * 4 + i];
        }
    }
    Encrypt(State);
    print(State);

    return 0;

}
```

PRESENT.H:

```
#ifndef PRESENT_80_H
#define PRESENT_80_H
#include <stdint.h>
#include <string.h>

//----Encryption key 1 and Plaintext 1
uint8_t EncryptionKey_1[10] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
uint8_t Plaintext_1[8]   = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
//----Encryption key 2 and Plaintext 2
uint8_t EncryptionKey_2[10] =
{0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
uint8_t Plaintext_2[8]   = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
//----Encryption key 3 and Plaintext 3
uint8_t EncryptionKey_3[10] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
uint8_t Plaintext_3[8]   = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
//----Encryption key 4 and Plaintext 4
uint8_t EncryptionKey_4[10] =
{0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
uint8_t Plaintext_4[8]   = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
const uint8_t sbox[16] = {
    0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD,
    0x3, 0xE, 0xF, 0x8, 0x4, 0x7, 0x1, 0x2
};

/*
------------------PRESENT-80 Validation (ECB)
-----Test1
Test Result 1
```

```
Expected Cyphertext - FF1DF4571108C38B
Github repo from assignment(location)- Set 1, vector#  7:
*/
unsigned char ECB_EncryptionKey_1[10] =
{0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
unsigned char ECB_Plaintext_1[8] =
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

/*-----Test2
Test Result 2
Expected Cyphertext - 0FC6284AABCD6AFA
Github repo from assignment(location)- Set 3, vector#234:
*/
unsigned char ECB_EncryptionKey_2[10] =
{0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA};
unsigned char ECB_Plaintext_2[8] = {0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA,0xEA};

/*-----Test3
Test Result 3
Expected Cyphertext - EA024714AD5C4D84
Github repo from assignment(location)- Set 8, vector#  1:
*/
unsigned char ECB_EncryptionKey_3[16] =
{0x2B,0xD6,0x45,0x9F,0x82,0xC5,0xB3,0x00,0x95,0x2C};
unsigned char ECB_Plaintext_3[16] =
{0x0C,0xED,0xD5,0x04,0x00,0x42,0x7E,0x41};



//Functions Declarations
void Key_Expansion(unsigned char *EncryptionKey);
void addRoundkey(uint8_t State[8],int round);
void sboxLayer(uint8_t State[8]);
void pLayer(uint8_t State[8]);
void Encrypt(uint8_t State[8]);
void print(unsigned char State[8]);
#endif
```

PRESENT.C:

```
#include <stdio.h>
#include "PRESENT_80.h"
/*
```

```
    PRESENT-80 ENCRYPTION WORKFLOW
    1. KEY EXPANSION:
        -Expand the 80-bit Master Key into 32 Round Keys (64 bits each).
        -Uses: 61-bit Left Circular Rotation, S-Box substitution on the top 4
bits, and a Round Counter XOR to break symmetry.
    2. INITIAL STEP:
        -The 64-bit Plaintext is loaded into the State.
    3. MAIN ROUNDS (1 to 31):
        -addRoundKey: XOR the current 64-bit State with the 64-bit Round Key.
        -sBoxLayer: Pass the State through sixteen 4-bit S-Boxes at the same
time.
        -pLayer: A bit-level permutation that moves each bit to a new
position.
    4. FINAL STEP:
        -After 31 rounds, a final addRoundKey is performed using the 32nd
Round Key to produce the Ciphertext.
*/



/*
    Present_80 encryption process has 31 rounds plus one final round, each
round it uses 64bit round key
    IT has total of 32 rounds so total size will be 32*8=256
*/
unsigned char keyExpansion[256];
void Key_Expansion(unsigned char *EncryptionKey) {
    //For circular shift instead of doing left shift of 61 bits .
    //doing the right shift of 19 bits.
    uint8_t key[10];
    memcpy(key,EncryptionKey,10);
    uint8_t temp[10];

    memcpy(keyExpansion,key,8); //first key is the leftmost 64 bits of the
orginal key
    //SBOX with the top bits and XOR with a round counter
    //since each roundkey is 8 bytes long below offset is taken.
    for(int round=1; round<32; round++) {

        /*
        (key[(i + 8) % 10] >> 3) takes bytes from two positions back and move
its bits to 3 positions right.this fills bottom 5 bits on the byte.
        (key[(i + 7) % 10] << 5) takes three bits that are falling off the
right side of the byte three positions back and puts them all the
```

```
        way to the left side of the current byte.
        */
        for(int i=0; i<10; i++) {
            temp[i] = (key[(i + 8) % 10] >> 3) |
                    (key[(i + 7) % 10] << 5);
        }
        memcpy(key,temp,10);
        uint8_t top_bits = key[0] >>4;
        //apply sbox
        top_bits = sbox[top_bits];
        //clear top bits in key and putting the new top bits
        //using 0x0f because in binary it is 00001111 so it keeps bottom bits
and clear top bits
        key[0] = (key[0] & 0x0F) | (top_bits << 4);
        //we need to xor(as algorithm spficies) the bits in key[0] - 19 to 15
to give a unique stamp each round
        //It is used to break symmetry between rounds
        //key[7] - 23-16, key[8] - 15-8
        /*round >>1 - what is does is
        round = abcde after the logic operation it becomes 0abcd now we have
the upper four bits
        round = abcde after logic operation round & 1 → e and then after e<<7
e0000000
        */
        key[7] ^= (round >>1);
        key[8] ^= (round & 0x01) << 7;
        //copying the key back to the keyExpansion
        memcpy(keyExpansion + (round*8),key,8);
    }

}
//Add round key
// XOR with Round Key
void addRoundkey(uint8_t State[8],int round) {
    for(int i=0; i<8; i++) {
        //We use the round multiplier to jump to the right 8-byte block,
        //then add 'i' to get the specific byte.
        State[i] ^= keyExpansion [(round*8)+i];
    }
}

//Sbox Layer
// 4-bit S-Box substitution
```

```c
void sboxLayer(uint8_t State[8]) {
    for(int i=0; i<8; i++) {
        //get top four bits and get bottom four bits and combine them
        uint8_t top = State[i] >> 4;//shifts 7-4 to 3-0
        uint8_t bottom = State[i] & 0x0F; //mask out 7-4 bits
        top = sbox[top];
        bottom = sbox[bottom];
        top = top << 4;
        State[i] = top | bottom;
    }
}

//pLayer
// Bit permutation
void pLayer(uint8_t State[8]) {
    uint8_t temp[8] = {0, 0, 0, 0, 0, 0, 0, 0}; //temporary state
    for (int i=0; i<64; i++) {
        //Finding bit position
        //The permutation formula: P(i) = (16 * i) % 63 (except for bit 63)
        int bit_pos = (16*i)%63;
        if(i==63) {
            bit_pos = 63;
        }
        //Finding the bit value
        //Bits are numbered 0-63 (0 is least significant bit of State[7])
        //To get bit i: byte index = 7 - (i / 8), bit position = i % 8
        uint8_t bit_val = (State[7 - (i / 8)] >> (i % 8)) & 0x01;
        //2. If the bit is 1, set the bit in the temp state at 'bit_pos'
        if (bit_val) {
            temp[7 - (bit_pos / 8)] |= (1 << (bit_pos % 8));
        }
    }
    // Copying the permuted state back to the original State array
    memcpy(State, temp, 8);
}

//For Encryption
void Encrypt(uint8_t State[8]) {
    for(int round = 0; round < 31; round++) {
        addRoundkey(State, round);
        sboxLayer(State);
        pLayer(State);
    }
```

```
    //Final Round key
    addRoundkey(State, 31);
}
//For printing cypher text
void print(unsigned char State[8]) {
    printf("Ciphertext: ");
    for (int i = 0; i < 8; i++) {
        printf("%02X", State[i]);
    }
    printf("\n");
}

int main() {

    //-------Plaintext 1--------
    Key_Expansion(EncryptionKey_1);
    Encrypt(Plaintext_1);
    print(Plaintext_1);

    //-------Plaintext 2--------
    Key_Expansion(EncryptionKey_2); // Must re-expand whenever the key
changes
    Encrypt(Plaintext_2);
    print(Plaintext_2);

    //-------Plaintext 3--------
    Key_Expansion(EncryptionKey_3);
    Encrypt(Plaintext_3);
    print(Plaintext_3);

    //-------Plaintext 4--------
    Key_Expansion(EncryptionKey_4);
    Encrypt(Plaintext_4);
    print(Plaintext_4);


/*
-------------------PRESENT-80 Validation (ECB) Test1*/
    Key_Expansion(ECB_EncryptionKey_1);
    Encrypt(ECB_Plaintext_1);
    print(ECB_Plaintext_1);
```

```
/*
------------------PRESENT-80 Validation (ECB) Test2*/
    Key_Expansion(ECB_EncryptionKey_2);
    Encrypt(ECB_Plaintext_2);
    print(ECB_Plaintext_2);
/*
------------------PRESENT-80 Validation (ECB) Test3*/
    Key_Expansion(ECB_EncryptionKey_3);
    Encrypt(ECB_Plaintext_3);
    print(ECB_Plaintext_3);


    return 0;
}
```