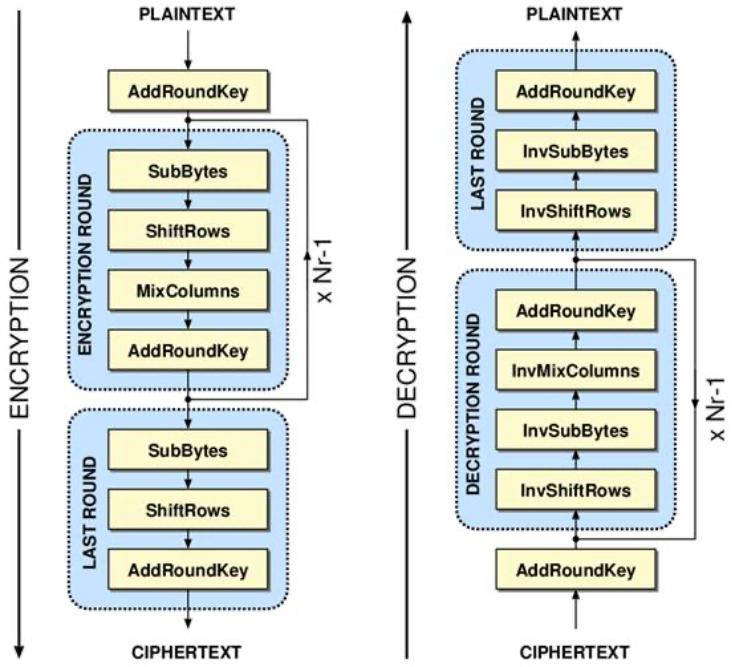
Q1:



Decryption is the conversion of encrypted data into its original form, and that is what we do here, by reversing the encryption process. At the beginning and the end of the AES algorithm we add a key, called "AddRoundKey". In the first step, we get the initial state of the key and then the algorithm generates several round keys depending on how many rounds the encryption process goes.

After this is done, we enter the decryption round which we can see from the image above. Here we have the 4 processes conducted in reverse order. Depending on whether we have AES-128, AES-192 or AES-256 they will continue iterating minimum 9 times within this decryption round. Each round in AES consists of the four processes conducted in the reverse order:

- **Inverse shift rows –** Shift rows to the left until we obtain a new matrix containing the same message but in a shifted manner. Speaks to the last assignment regarding diffusion.
- **Inverse sub bytes –** Simply the s-box, each input byte is substituted according to the lookup-table, speaks to confusion.
- **Inverse mix columns –** Each column of four bytes is transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes.
- **Add round key -** Here we use the round keys from the start of the encryption, where the result from the previous layer gets added together with the round key. If this is the last round then we have performed the decryption algorithm and the output is the desired plaintext, otherwise the output of this round is taken back and we begin another round.

Q2

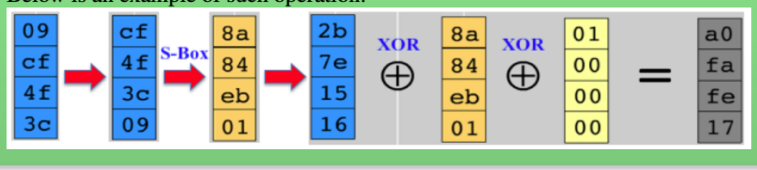
AES Key scheduling

(a) Take w_{i-1} column and rotate 1 byte up

(b) Apply S-box to each of the 4 bytes

(c) XOR w_{i-4} to the result and XOR $RCON(i/4)$ to the result, where $RCON$ stands for Round Constant.

Below is an example of such operation:



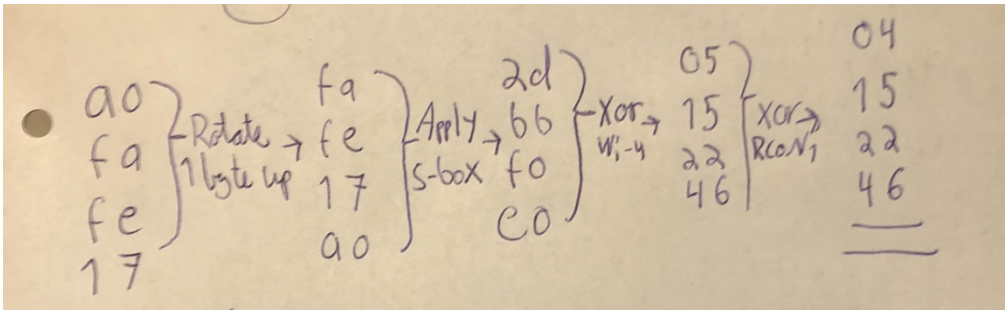
As shown above, the AES key scheduling operation consists of:

- Rotating 1 byte up
- Applying s-box to each of the 4 bytes
- Performing XOR-operation twice

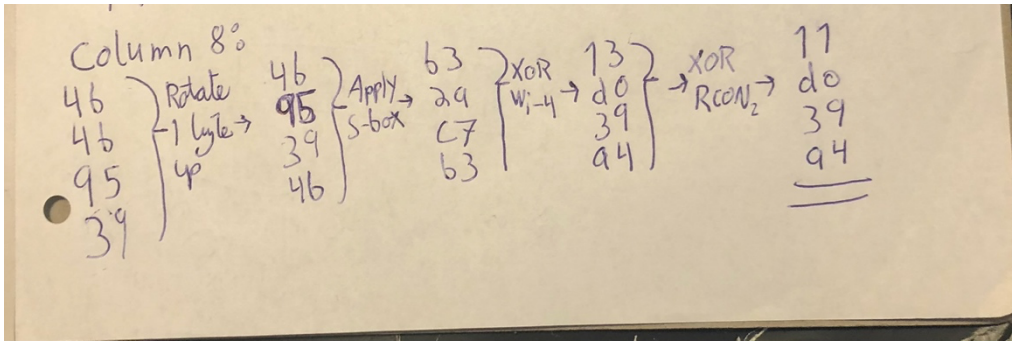
By doing this operation, I got the result as shown in the table below:

2b	**28**	**ab**	**09**	**a0**	**04**	**f3**	**4b**	**11**
7e	**ae**	**f7**	**cf**	**fa**	**15**	**64**	**4b**	**d0**
15	**D2**	**15**	**4f**	**fe**	**22**	**4f**	**95**	**39**
16	**A6**	**88**	**3c**	**17**	**46**	**7a**	**39**	**a4**

Calculations: Firstly, here is the calculation for column 4, I simply follow the steps as explained above, rotating 1 byte up, applying s-box, performing an XOR-operation with column 4 and lastly performing an XOR-operation again. The same steps are applied in the calculation for the last column as shown below



Calculation for column 4

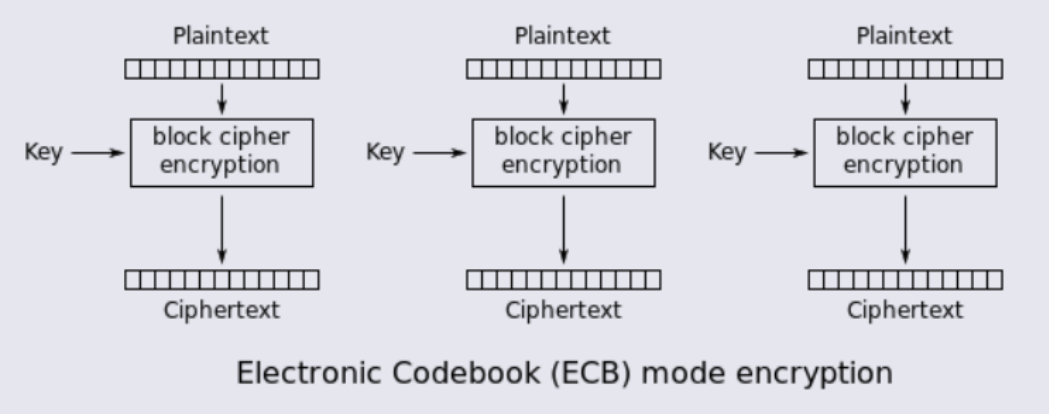


Calculation for last column (8)

Consider a message string "DCS-3101", the initialization vector "NO" and the key "EU". For this exercise, consider one block consists of 16 bits and encryption is simply XOR operation. Please refer to lecture notes on Block ciphers and Stream ciphers to solve this problem.

Q3.1: Illustrate the encryption using ECB on the whole message string. Encrypt a block of 16 bits using ECB. The message string is "DCS-3101", the IV is "NO" and the key is "EU".

In cryptography, a **block cipher mode of operation** is an algorithm that uses a block cipher to provide information security such as confidentiality or authenticity. The simplest of the encryption modes is the **electronic codebook** (ECB). Here, each block is encrypted independently, identical plaintexts is encrypted similarly with no chaining and no error propagation. Encryption formula to obtain ciphertext: C1, . . . , Ct as $Ci = E_k (M_i)$, i = 1 . . .t

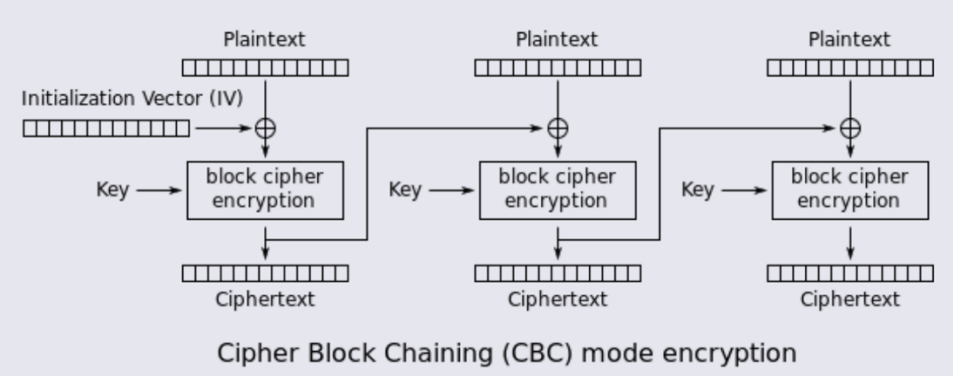


ECB doesn't include an initialization vector, so we work with the plaintext and the key. We want to match the key length "EU" with the plaintext: "DCS-3103", so we extend the key to "EUEUEUEU". Thereafter, I divide into blocks of 16 bits and perform xor-operation with the plaintext and the key to achieve the ciphertext below:

DCS-3103(Plaintext)	**EU(Key):**	**Ciphertext:**
Block 1:		
01000100	01000101	00000001
01000011	01010101	00010110
Block 2:		
01010011	01000101	00010110
00101101	01010101	01111000
Block 3:		
00110011	01000101	01110110
00110001	01010101	01100100
Block 4:		
00110011	01000101	01110101
00110000	01010101	01100100

Q3.2 Illustrate the encryption using CBC:

Cipher block chaining (CBC) solves the problem of ECB regarding that identical plaintext blocks are encrypted to identical ciphertext blocks gives an unwanted structure to encrypted data that reveals information about the plaintext. CBC does this by "chaining" blocks together by taking the output of one encryption and mixing it into the input for the next block.



Cipher Block Chaining (CBC) mode encryption

We start by converting the plaintext into binary using the ASCII table. Then we match the length of the initialization vector from "NO" to "NONONONO" to match with the plaintext. Then we split this into 4 blocks:

Block 1:

Plaintext: "DC"	**NO(IV)**	**Intermediate value**
0100 0100	0100 1110	0000 1010
0100 0011	0100 1111	0000 1100

We then perform another xor-operation with the key "EUEUEUEU".

Intermediate value	**Key**	**Ciphertext**
0000 1010	0100 0101	0100 1111
0000 1100	0101 0101	0101 1001

We now take the ciphertext from block 1 and use it as the IV for block 2:

Block 2:

Plaintext: "S-"	**IV from B1**	**Intermediate value**
0101 0011	0100 1111	0001 1100
0010 1101	0101 1001	0111 0100

Intermediate value	**Key**	**Ciphertext**
0001 1100	0100 0101	01011001
0111 0100	0101 0101	00100001

We now take the ciphertext from block 2 and use it as the IV for block 3:

Block 3:

Plaintext: "31"	**IV from B2**	**Intermediate value**
0011 0011	0101 1001	0110 1010
0011 0001	0010 0001	0001 0000

Intermediate value	Key	Ciphertext
0110 1010	01000101	0010 1111
0001 0000	01010101	0100 0101

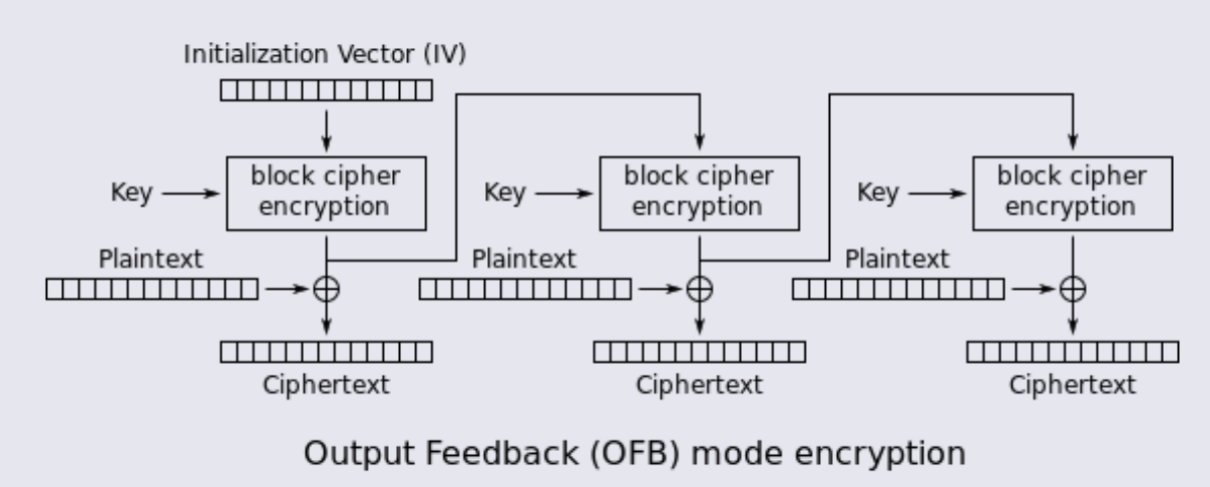
We now take the ciphertext from block 3 and use it as the IV for block 4:

Block 4:

Plaintext: “01”	IV from B3	Intermediate value
0011 0000	0010 1111	0001 1111
0011 0001	0100 0101	0111 0100
Intermediate value	**Key**	**Ciphertext**
0001 1111	0100 0101	0101 1010
0111 0100	0101 0101	0010 0001

Q3.3 Illustrate the encryption using OFB:

The output feedback (OFB) mode makes a block cipher into a synchronous stream cipher. It generates keystream blocks, which are then XORed with the plaintext blocks to get the ciphertext.



We start by encrypting the IV with the key by performing an XOR-operation, then we use the encrypted IV and Xor it with the plaintext to achieve the ciphertext.

Block 1:

IV:”NONONONO”	Key: “EUEUEUEU”	Encrypted IV
0100 1110	0100 0101	0000 1011
0100 1111	0101 0101	0001 1010

Then we perform an xor-operation with the encrypted IV and the plaintext

Encrypted IV	Plaintext: “DC”	Ciphertext
0000 1011	0100 0100	0100 1111
0001 1010	0100 0011	0101 1001

We repeat the same process for block 2, except here we use the encrypted IV from block 1 as the input for block 2,

Block 2:

IV	Key	Encrypted IV
0000 1011	0100 0101	0100 1110
0001 1010	0101 0101	0100 1111

Encrypted IV	Plaintext: "S-"	Ciphertext
0100 1110	0101 0011	0001 1101
0100 1111	0010 1101	0110 0010

Again, the encrypted IV is used as the input for the block 3

Block 3:

IV	Key	Encrypted IV
0100 1110	0100 0101	0000 1011
0100 1111	0101 0101	0001 1010

Encrypted IV	Plaintext: "31"	Ciphertext
0000 1011	0011 0011	0011 1000
0001 1010	0011 0001	0010 1011

Repeat the same process as above.

Block 4:

IV	Key	Encrypted IV
0000 1011	0100 0101	0100 1110
0001 1010	0101 0101	0100 1111

Encrypted IV	Plaintext: "01"	Ciphertext
0100 1110	0011 0000	0111 1110
0100 1111	0011 0001	0111 1110

Q3.4

Output from block 1 is the ciphertext: **0100 1111 0101 1001**
By flipping the last bit, we get: **0100 1111 0101 1000**. If we use this to illustrate the impact on the ouput, i.e. decryption we get the following:

IV	Key	Encrypted IV
0100 1110	0100 0101	0000 1011
0100 1111	0101 0101	0001 1010
Encrypted IV	**Ciphertext**	**Plaintext**
0000 1011	0100 1111	0100 0100
0001 1010	0101 1000	0100 0010

A bit-flipping attack is an attack on a cryptographic cipher in which the attacker can change the ciphertext in such a way as to result in a predictable change of the plaintext. By us comparing this bit-flipping to the original result (in hexa: 4442 and 4443) we see that they are very similar in the ofb mode. Although the attacker is not able to learn the plaintext itself, this

can for example become a Denial of service attack against all messages on a particular channel using that cipher.

References:

https://www.tutorialspoint.com/cryptography/advanced_encryption_standard.htm
https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
https://www.rapidtables.com/convert/number/binary-to-ascii.html
https://resources.infosecinstitute.com/cbc-byte-flipping-attack-101-approach/
https://en.wikipedia.org/wiki/Bit-flipping_attack