

Forum:[Hack Forums](#)

Thread: [How much do you know about AES E/D operations?](#)

putnik1313 - May 26, 2025, 9:35 am

You will see how AES works, it's internal operations, block processing, key scheduling, etc...

Note that i will use some additional optional attributes, and lastly, the code can actually be simplified and more performant, i will be using explicit operations because that's the purpose here.

first let's begin by defining all the required constants and tables, there aren't too many, but is good practice to know them:

```
Code #define AES128KS 0x80          // this is the key size for AES-128
#define AES192KS 0xC0              // key for AES-192
#define AES256KS 0x100            // and key for AES-256
#define AES128_ROUNDS 10          // for AES-128, there are a total of 10 rounds to perform
#define AES192_ROUNDS 12          // for AES-192 = 12 rounds
#define AES256_ROUNDS 14          // for AES-256 = 14 rounds
#define AES_BLOCK_SIZE 16         // the block size for AES is 16, meaning that AES processes
blocks of 16 bytes(except for some modes of operation)
```

```
int Nk;                // number of subkeys to generate
int Nr,                // number of rounds for the specific AES key size
int Nb = 4;            // this is a constant and is always set to 4 for AES
byte round_keys[240]; // supports up to AES-256
byte sbox[256];         // a 256 byte table used in the E function
byte inv_sbox[256];     // this one is used in the D function
byte rcon[256];         // round constant, used in the key expansion process, mainly for xor
operations
```

Important AES operations:

AES Encryption Main Round:SubBytes

ShiftRows

MixColumns

AddRoundKey

AES Encryption Last Round:SubBytes

ShiftRows

AddRoundKey

AES Decryption Main Round:InvShiftRows

InvSubBytes

AddRoundKey

InvMixColumns

AES Decryption Last Round:InvShiftRows

InvSubBytes

AddRoundKey

Now, let's create the functions used to generate/initialize the lookup tables like sbox and invSbox, i could stick the actual table value instead of using these functions to create them, but then you won't fully understand the concept.

```
Code /* ===== Galois Field & S-Box ===== */
```

```
__attr_nodiscard static inline byte gf_mul(byte a, byte b) __noexcept
```

```
{
    byte p = 0;
    for (int i = 0; i < 8; ++i)
    {
        if (likely(b & 1)) p ^= a;
        byte hiBitSet = a & 0x80;
        a <<= 1;
        if (hiBitSet) a ^= 0x1B;
        b >>= 1;
    }
    return p;
}
```

```
__attr_nodiscard static inline byte gf_inv(byte x) __noexcept
```

```
{
    byte y = x;
    for (int i = 0; i < 4; ++i)
    {
        y = gf_mul(y, y);
        y = gf_mul(y, x);
    }
    return y;
}
```

```
__attr_nodiscard static inline byte affine_transform(byte x) __noexcept
```

```
{
    byte result = 0x63;
    for (int i = 0; i < 8; ++i) if ((x >> i) & 1) result ^= (0xF1 >> (7 - i)) & 0xFF;
    return result;
}
```

```

}

__attr_nodiscard static inline byte sbbox_entry(byte x) __noexcept
{
    return affine_transform(gf_inv(x));
}

static inline void create_sbbox(byte sbbox[256]) __noexcept
{
    for (int i = 0; i < 256; ++i) sbbox[i] = sbbox_entry((byte)i);
}

static inline void create_inv_sbbox(const byte sbbox[256], byte inv_sbbox[256]) __noexcept
{
    for (int i = 0; i < 256; ++i) inv_sbbox[sbbox[i]] = (byte)i;
}

static inline void create_rcon(byte rcon[256]) __noexcept
{
    byte c = 1;
    for (int i = 0; i < 256; ++i)
    {
        rcon[i] = c;
        c = gf_mul(c, 0x02);
    }
}

```

we will actually use a structure to encapsulate lookup tables, AES constants, and subkeys into a structure called `aes_ctx`(aes context):

```

Code typedef struct
{
    int Nk, Nr, Nb;
    byte round_keys[240]; // supports up to AES-256
    byte sbbox[256];
    byte inv_sbbox[256];
    byte rcon[256];
} aes_ctx;

```

Good, now i will continue by defining the actual function for initializing `aes_ctx`:

```

Code static inline void aes_init(aes_ctx *ctx, int key_size) __noexcept
{
    ctx->Nb = 4; // this is always 4 for AES
    if (key_size == AES128KS){
        ctx->Nk = 4;
        ctx->Nr = 10;
    } else if (key_size == AES192KS){
        ctx->Nk = 6;
        ctx->Nr = 12;
    } else if (key_size == AES256KS){
        ctx->Nk = 8;
        ctx->Nr = 14;
    }
}

```

```

}else{
    ctx->Nk = 4;
    ctx->Nr = 10;
}
create_sbox(ctx->sbox);                // populate the sbox table
create_inv_sbox(ctx->sbox, ctx->inv_sbox); // populate the invSbox table
create_rcon(ctx->rcon);                // populate the round constant(rcon) table
}

```

The next important function to define is the KeyExpansion, which will be in the next post because apparently posts can't be larger than 8k bytes:

now let's define the key Scheduling function, this function is responsible for generating subkeys for aes encryption/decryption rounds, which can be either 10, 12 or 14, these subkeys are generated using the main key:

```

Code // AES Key Expansion (Key Schedule)
// This function takes the original key and expands it into all the round keys needed for AES
encryption/decryption.
// ctx: structure with S-box, round constant (rcon), and space for round keys
// key: the original secret key
static inline void aes_key_expansion(aes_ctx *ctx, const byte *key) __noexcept
{
    int i = 0, Nk = ctx->Nk, Nb = ctx->Nb, Nr = ctx->Nr;
    byte temp[4];
    byte *w = ctx->round_keys;

    // 1. Copy the original key into the first Nk words of the round key array.
    // Each word is 4 bytes. For AES-128, Nk=4; for AES-192, Nk=6; for AES-256, Nk=8.
    for (; i < Nk; ++i)
    {
        w[4 * i + 0] = key[4 * i + 0];
        w[4 * i + 1] = key[4 * i + 1];
        w[4 * i + 2] = key[4 * i + 2];
        w[4 * i + 3] = key[4 * i + 3];
    }

    // 2. Expand the key to fill the round key array for all AES rounds.
    // Total number of words needed = Nb * (Nr + 1), where Nb = block size in words (4 for AES), Nr
    = number of rounds.
    for (; i < Nb * (Nr + 1); ++i)
    {
        // Copy previous word into temp
        temp[0] = w[4 * (i - 1) + 0];
        temp[1] = w[4 * (i - 1) + 1];
        temp[2] = w[4 * (i - 1) + 2];
        temp[3] = w[4 * (i - 1) + 3];

        // Every Nk words, apply the "key schedule core":
        // - Rotate left (RotWord)
        // - Substitute each byte with S-box (SubWord)
        // - XOR the first byte with a round constant (Rcon)
        if (i % Nk == 0)

```

```

{
    byte t = temp[0];
    temp[0] = ctx->sbox[temp[1]] ^ ctx->rcon[i / Nk]; // S-box + Rcon on rotated first byte
    temp[1] = ctx->sbox[temp[2]];
    temp[2] = ctx->sbox[temp[3]];
    temp[3] = ctx->sbox[t];
}
// For AES-256 (Nk > 6), every 4th word also gets S-boxed (no rotate or Rcon)
else if (Nk > 6 && (i % Nk == 4))
{
    temp[0] = ctx->sbox[temp[0]];
    temp[1] = ctx->sbox[temp[1]];
    temp[2] = ctx->sbox[temp[2]];
    temp[3] = ctx->sbox[temp[3]];
}

// XOR temp with the word Nk positions earlier to get the new word
// This links the expanded key material back to the original key.
w[4 * i + 0] = w[4 * (i - Nk) + 0] ^ temp[0];
w[4 * i + 1] = w[4 * (i - Nk) + 1] ^ temp[1];
w[4 * i + 2] = w[4 * (i - Nk) + 2] ^ temp[2];
w[4 * i + 3] = w[4 * (i - Nk) + 3] ^ temp[3];
}
}

```

The key scheduling operation is ready, now we need to implement the block routine operations:

Operation: AddRoundKey

This operation performs a bitwise XOR between each byte of the state matrix and the corresponding byte of the round key. Specifically, for every column c and row r in the state array, $state[r][c]$ is updated as $state[r][c] \oplus rk[4 * c + r]$, effectively combining the state with the round key for this round

```

Code // For each column (c) and row (r), the state byte is XOR'ed with the corresponding key byte:
// state[r][c] ^= rk[4 * c + r];
static inline void aes_add_round_key(byte state[4][4], const byte *rk) __noexcept
{
    // Loop over columns
    for (int c = 0; c < 4; ++c)
        // Loop over rows
        for (int r = 0; r < 4; ++r)
            // XOR each state byte with the corresponding round key byte
            state[r][c] ^= rk[4 * c + r];
}

```

Operation: SubBytes

This operation substitutes each byte in the state matrix with a corresponding value from the S-box (substitution box), in this case, from `aes_ctx.sbox`. Specifically, for every row r and column c in the state array, $state[r][c]$ is updated as $state[r][c] = sbox[state[r][c]]$, providing non-linearity and confusion in the AES cipher.

Confusion:

Confusion refers to the property of an encryption algorithm that obscures the relationship between the ciphertext and the encryption key. In AES, confusion is primarily achieved through the SubBytes operation, which substitutes each byte of the state with a value from a nonlinear S-box, making it difficult to deduce the key from the ciphertext.

Diffusion:

Diffusion is the property that ensures the influence of one plaintext symbol spreads over many ciphertext symbols, making patterns in the plaintext less apparent in the ciphertext. In AES, diffusion is mainly provided by the ShiftRows and MixColumns operations, which rearrange and mix the bytes in the state so that a change in a single byte of the plaintext affects many bytes in the ciphertext after several rounds.

Code // For each row (r) and column (c), the state byte is replaced with the corresponding S-box value:

```
// state[r][c] = sbox[state[r][c]];
static inline void aes_sub_bytes(byte state[4][4], const byte sbox[256]) __noexcept
{
    // Loop over rows
    for (int r = 0; r < 4; ++r)
        // Loop over columns
        for (int c = 0; c < 4; ++c)
            // Substitute each state byte using the S-box
            state[r][c] = sbox[state[r][c]];
}
```

Operation: InvSubBytes

This operation substitutes each byte in the state matrix with a corresponding value from the inverse S-box (invsbox). Specifically, for every row r and column c in the state array, state[r][c] is updated as state[r][c] = invsbox[state[r][c]], reversing the non-linear substitution applied during encryption.

Code // For each row (r) and column (c), the state byte is replaced with the corresponding inverse S-box value:

```
// state[r][c] = invsbox[state[r][c]];
static inline void aes_inv_sub_bytes(byte state[4][4], const byte invsbox[256]) __noexcept
{
    // Loop over rows
    for (int r = 0; r < 4; ++r)
        // Loop over columns
        for (int c = 0; c < 4; ++c)
            // Substitute each state byte using the inverse S-box
            state[r][c] = invsbox[state[r][c]];
}
```

Operation: ShiftRows

This operation shifts the rows of the state matrix to the left by different offsets to provide diffusion. Specifically, the first row is left unchanged, the second row is shifted left by 1 position, the third row by 2 positions, and the fourth row by 3 positions. This helps to spread the influence of each input byte over multiple columns in the state.

Code // For each row in the state (except the first), shift the row to the left by an offset equal to its

row index:

```
// Row 0: no shift
// Row 1: shift left by 1
// Row 2: shift left by 2
// Row 3: shift left by 3 (or right by 1)
static inline void aes_shift_rows(byte state[4][4]) __noexcept
{
    byte t;
    // Shift row 1 left by 1
    t = state[1][0];
    state[1][0] = state[1][1];
    state[1][1] = state[1][2];
    state[1][2] = state[1][3];
    state[1][3] = t;

    // Shift row 2 left by 2
    t = state[2][0];
    byte t2 = state[2][1];
    state[2][0] = state[2][2];
    state[2][1] = state[2][3];
    state[2][2] = t;
    state[2][3] = t2;

    // Shift row 3 left by 3 (equivalent to right by 1)
    t = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = state[3][0];
    state[3][0] = t;
}
```

Operation: InvShiftRows

This operation reverses the row shifts applied during the ShiftRows step in AES encryption. Each row of the state matrix is cyclically shifted to the right by an offset equal to its row index: the first row remains unchanged, the second row is shifted right by 1, the third row by 2, and the fourth row by 3 positions. This restores the original byte positions before the ShiftRows step, helping to reverse diffusion during decryption.

Code // For each row in the state (except the first), shift the row to the right by an offset equal to its row index:

```
// Row 0: no shift
// Row 1: shift right by 1
// Row 2: shift right by 2
// Row 3: shift right by 3 (or left by 1)
static inline void aes_inv_shift_rows(byte state[4][4]) __noexcept
{
    byte t;
    // Shift row 1 right by 1
    t = state[1][3];
    state[1][3] = state[1][2];
    state[1][2] = state[1][1];
    state[1][1] = state[1][0];
```

```

state[1][0] = t;

// Shift row 2 right by 2
t = state[2][0];
byte t2 = state[2][1];
state[2][0] = state[2][2];
state[2][1] = state[2][3];
state[2][2] = t;
state[2][3] = t2;

// Shift row 3 right by 3 (equivalent to left by 1)
t = state[3][0];
state[3][0] = state[3][1];
state[3][1] = state[3][2];
state[3][2] = state[3][3];
state[3][3] = t;
}

```

Operation: MixColumns

This operation mixes each column of the state matrix by treating each column as a four-term polynomial and multiplying it by a fixed polynomial modulo $(x^4 + 1)$ in the Galois Field $GF(2^8)$. Specifically, for each column, the bytes are combined using Galois Field multiplications and additions, spreading the influence of each input byte to all four bytes in the column, thus providing diffusion.

```

Code // For each column in the state, mix the bytes using Galois Field arithmetic:
// Each byte in a column is replaced with a linear combination of all four bytes in that column:
// state[0][c] = (2 * a0) ^ (3 * a1) ^ a2 ^ a3
// state[1][c] = a0 ^ (2 * a1) ^ (3 * a2) ^ a3
// state[2][c] = a0 ^ a1 ^ (2 * a2) ^ (3 * a3)
// state[3][c] = (3 * a0) ^ a1 ^ a2 ^ (2 * a3)
static inline void aes_mix_columns(byte state[4][4]) __noexcept
{
    for (int c = 0; c < 4; ++c)
    {
        // Extract original column values
        byte a0 = state[0][c], a1 = state[1][c], a2 = state[2][c], a3 = state[3][c];
        // Apply the MixColumns transformation using GF(2^8) multiplication
        state[0][c] = gf_mul(a0, 2) ^ gf_mul(a1, 3) ^ a2 ^ a3;
        state[1][c] = a0 ^ gf_mul(a1, 2) ^ gf_mul(a2, 3) ^ a3;
        state[2][c] = a0 ^ a1 ^ gf_mul(a2, 2) ^ gf_mul(a3, 3);
        state[3][c] = gf_mul(a0, 3) ^ a1 ^ a2 ^ gf_mul(a3, 2);
    }
}

```

Operation: InvMixColumns

This operation reverses the mixing transformation applied by MixColumns during AES encryption. For each column in the state matrix, the bytes are combined using multiplication by fixed values (0x0e, 0x0b, 0x0d, 0x09) in the Galois Field $GF(2^8)$. This restores the original column values before MixColumns, undoing the diffusion in the decryption process.

Code // For each column in the state, reverse the MixColumns operation using Galois Field arithmetic:

// Each byte in a column is replaced with a linear combination of all four bytes in that column, using specific multipliers (0x0e, 0x0b, 0x0d, 0x09) as defined by the AES standard:

// state[0][c] = (0x0e * a0) ^ (0x0b * a1) ^ (0x0d * a2) ^ (0x09 * a3)

// state[1][c] = (0x09 * a0) ^ (0x0e * a1) ^ (0x0b * a2) ^ (0x0d * a3)

// state[2][c] = (0x0d * a0) ^ (0x09 * a1) ^ (0x0e * a2) ^ (0x0b * a3)

// state[3][c] = (0x0b * a0) ^ (0x0d * a1) ^ (0x09 * a2) ^ (0x0e * a3)

static inline void aes_inv_mix_columns(byte state[4][4]) __noexcept

{

for (int c = 0; c < 4; ++c)

{

// Extract original column values

byte a0 = state[0][c], a1 = state[1][c], a2 = state[2][c], a3 = state[3][c];

// Apply the InvMixColumns transformation using GF(2^8) multiplication

state[0][c] = gf_mul(a0, 0x0e) ^ gf_mul(a1, 0x0b) ^ gf_mul(a2, 0x0d) ^ gf_mul(a3, 0x09);

state[1][c] = gf_mul(a0, 0x09) ^ gf_mul(a1, 0x0e) ^ gf_mul(a2, 0x0b) ^ gf_mul(a3, 0x0d);

state[2][c] = gf_mul(a0, 0x0d) ^ gf_mul(a1, 0x09) ^ gf_mul(a2, 0x0e) ^ gf_mul(a3, 0x0b);

state[3][c] = gf_mul(a0, 0x0b) ^ gf_mul(a1, 0x0d) ^ gf_mul(a2, 0x09) ^ gf_mul(a3, 0x0e);

}

}

Operation: AES Encrypt Block

This function encrypts a single 16-byte block using the AES algorithm. It initializes the state matrix from the input, applies the initial AddRoundKey, then performs a series of rounds (each consisting of SubBytes, ShiftRows, MixColumns, and AddRoundKey). The final round omits MixColumns. The result is written to the output array.

Code // Encrypt a single 16-byte block using AES.

// 1. Initialize the state matrix from the input.

// 2. Add the initial round key.

// 3. For each round except the last, perform:

// - SubBytes: substitute bytes using the S-box

// - ShiftRows: shift rows for diffusion

// - MixColumns: mix columns for diffusion

// - AddRoundKey: combine with round key

// 4. Final round (without MixColumns):

// - SubBytes

// - ShiftRows

// - AddRoundKey

// 5. Write the resulting state to output.

__attr_hot static inline void aes_encrypt_block(aes_ctx *ctx, const byte in[16], byte out[16])

__noexcept

{

byte state[4][4];

// Initialize state from input (column-major order)

for (int r = 0; r < 4; ++r)

for (int c = 0; c < 4; ++c)

state[r][c] = in[r + 4 * c];

// Initial AddRoundKey

aes_add_round_key(state, ctx->round_keys);

// Main rounds

```

for (int round = 1; round < ctx->Nr; ++round)
{
    aes_sub_bytes(state, ctx->sbox);          // Substitute bytes
    aes_shift_rows(state);                    // Shift rows
    aes_mix_columns(state);                   // Mix columns
    aes_add_round_key(state, ctx->round_keys + 16 * round); // Add round key
}
// Final round (no MixColumns)
aes_sub_bytes(state, ctx->sbox);
aes_shift_rows(state);
aes_add_round_key(state, ctx->round_keys + 16 * ctx->Nr);
// Write state to output (column-major order)
for (int r = 0; r < 4; ++r)
    for (int c = 0; c < 4; ++c)
        out[r + 4 * c] = state[r][c];
}

```

Operation: AES Decrypt Block

This function decrypts a single 16-byte block using the AES algorithm. It initializes the state matrix from the input, applies the initial AddRoundKey with the last round key, then performs a series of rounds (each consisting of InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns). The final round omits InvMixColumns. The result is written to the output array.

Code // Decrypt a single 16-byte block using AES.

```

// 1. Initialize the state matrix from the input.
// 2. Add the round key for the last round.
// 3. For each round in reverse (except the first), perform:
//    - InvShiftRows: reverse row shifting
//    - InvSubBytes: reverse byte substitution using the inverse S-box
//    - AddRoundKey: combine with the corresponding round key
//    - InvMixColumns: reverse column mixing
// 4. Final round (no InvMixColumns):
//    - InvShiftRows
//    - InvSubBytes
//    - AddRoundKey (first round key)
// 5. Write the resulting state to output.
__attr_hot static inline void aes_decrypt_block(aes_ctx *ctx, const byte in[16], byte out[16])
__noexcept
{
    byte state[4][4];
    // Initialize state from input (column-major order)
    for (int r = 0; r < 4; ++r)
        for (int c = 0; c < 4; ++c)
            state[r][c] = in[r + 4 * c];
    // Initial AddRoundKey with the last round key
    aes_add_round_key(state, ctx->round_keys + 16 * ctx->Nr);
    // Main rounds (in reverse)
    for (int round = ctx->Nr - 1; round >= 1; --round)
    {
        aes_inv_shift_rows(state);            // Reverse row shifting
        aes_inv_sub_bytes(state, ctx->inv_sbox); // Reverse byte substitution
    }
}

```

```

    aes_add_round_key(state, ctx->round_keys + 16 * round); // Add round key
    aes_inv_mix_columns(state);                             // Reverse column mixing
}
// Final round (no InvMixColumns)
aes_inv_shift_rows(state);
aes_inv_sub_bytes(state, ctx->inv_sbox);
aes_add_round_key(state, ctx->round_keys);
// Write state to output (column-major order)
for (int r = 0; r < 4; ++r)
    for (int c = 0; c < 4; ++c)
        out[r + 4 * c] = state[r][c];
}

```

In this AES tutorial, i will also show you how to implement padding, which many AES modes of operation require, but not all tutorials online(at least that i saw), they fail to explain how to implement padding ,which is actually a crucial operation, and honestly, i don't get the reason they omit padding explanation, is very easy to implement, the most common padding scheme is PKCS#7.

Padding ensures that data is a multiple of the block size, this is very important in the context of AES and many other block cipher algorithms, you will see why...

Operation: PKCS#7 Padding

This function applies PKCS#7 padding to a buffer so that its length becomes a multiple of the block size. It calculates how many padding bytes are needed, fills the end of the buffer with that value, and returns the new padded length. Each padding byte is set to the value of the number of padding bytes added.

Code // Pad the buffer using PKCS#7 so its length is a multiple of block_size.

// 1. Calculate the number of padding bytes needed.

// 2. Fill the end of the buffer with the value of the padding length (as required by PKCS#7).

// 3. Return the new length after padding.

```

__attr_nodiscard static inline size_t pkcs7_pad(byte *buf, size_t len, size_t block_size) __noexcept
{
    size_t pad = block_size - (len % block_size); // Calculate required padding
    for (size_t i = 0; i < pad; ++i)
        buf[len + i] = (byte)pad;                // Fill with padding value
    return len + pad;                             // Return new padded length
}

```

Operation: PKCS#7 Unpadding

This function removes PKCS#7 padding from a buffer after decryption. It reads the value of the last byte to determine how many padding bytes were added and subtracts that amount from the total length. If the padding value is invalid, it returns the original length.

Code // Remove PKCS#7 padding from the buffer.

// 1. If the buffer length is zero, return 0 (empty).

// 2. Read the last byte to determine the padding length.

// 3. If the padding value is invalid (greater than AES block size), return the original length.

// 4. Otherwise, return the length minus the padding.

```

__attr_nodiscard static inline size_t pkcs7_unpad(byte *buf, size_t len) __noexcept

```

```

{
    if (!len)
        return 0; // No data to unpad
    byte pad = buf[len - 1]; // Last byte indicates padding length
    if (unlikely(pad > AES_BLOCK_SIZE))
        return len; // Invalid padding, return original length
    return len - pad; // Remove padding
}

```

Good, we are almost done, what's missing? Mode of operation implementation!

Let's implement the logic for different modes, so you get the idea. A mode of operation basically defines how the cipher processes blocks of data, some modes may require additional data like an IV or nonce, others like GCM mode, also require a tag and aad, but the most basic mode like ECB mode, requires no additional data, this is why is the default mode, actually, when you are not using or implemented any mode of operation, you are just using ECB mode.

Operation: AES ECB Encrypt

This function performs AES encryption in ECB (Electronic Codebook) mode. The input data is divided into 16-byte blocks, and each block is encrypted independently using the provided AES context. In ECB mode, identical plaintext blocks result in identical ciphertext blocks.

ECB MODE

Code // Encrypt data in ECB (Electronic Codebook) mode.

// 1. Divide the input into 16-byte blocks.

// 2. Encrypt each block independently using `aes_encrypt_block`.

```

__attr_hot static inline void aes_ecb_encrypt(aes_ctx *ctx, const byte *in, byte *out, size_t len)
__noexcept
{
    for (size_t i = 0; i < len; i += AES_BLOCK_SIZE)
        aes_encrypt_block(ctx, in + i, out + i); // Encrypt one block at a time
}

```

Operation: AES ECB Decrypt

This function decrypts data using AES in ECB (Electronic Codebook) mode. The input is divided into 16-byte blocks, and each block is decrypted independently using the AES context.

Code // Decrypt data in ECB (Electronic Codebook) mode.

// 1. Divide the input into 16-byte blocks.

// 2. Decrypt each block independently using `aes_decrypt_block`.

```

__attr_hot static inline void aes_ecb_decrypt(aes_ctx *ctx, const byte *in, byte *out, size_t len)
__noexcept
{
    for (size_t i = 0; i < len; i += AES_BLOCK_SIZE)
        aes_decrypt_block(ctx, in + i, out + i); // Decrypt one block at a time
}

```

CBC MODE

Operation: AES CBC Encrypt

This function implements AES encryption in CBC (Cipher Block Chaining) mode. In CBC mode, each plaintext block is XORed with the previous ciphertext block (or the IV for the first block) before being encrypted. This chaining ensures that identical plaintext blocks will encrypt to different ciphertext blocks if their positions or previous blocks differ.

```
Code // Encrypt using AES in CBC (Cipher Block Chaining) mode.
// 1. Initialize 'prev' with the IV (Initialization Vector).
// 2. For each 16-byte block in the input:
//    a) XOR the current plaintext block with 'prev' (previous ciphertext or IV).
//    b) Encrypt the result and write to output.
//    c) Update 'prev' to the newly created ciphertext block.
__attr_hot static inline void aes_cbc_encrypt(aes_ctx *ctx, const byte *in, byte *out, size_t len, byte
iv[16]) __noexcept
{
    byte prev[16];
    memcpy(prev, iv, 16); // Start with IV
    for (size_t i = 0; i < len; i += AES_BLOCK_SIZE)
    {
        byte block[16];
        // XOR input block with the previous ciphertext (or IV for first block)
        for (int j = 0; j < 16; ++j)
            block[j] = in[i + j] ^ prev[j];
        aes_encrypt_block(ctx, block, out + i); // Encrypt the XORed block
        memcpy(prev, out + i, 16);             // Update prev for next block
    }
}
```

Operation: AES CBC Decrypt

This function implements AES decryption in CBC (Cipher Block Chaining) mode. In CBC mode, each ciphertext block is first decrypted, then XORed with the previous ciphertext block (or the IV for the first block) to recover the original plaintext.

```
Code // Decrypt using AES in CBC (Cipher Block Chaining) mode.
// 1. Initialize 'prev' with the IV (Initialization Vector).
// 2. For each 16-byte block in the input:
//    a) Decrypt the current ciphertext block into 'block'.
//    b) XOR the decrypted block with 'prev' (previous ciphertext or IV) to get plaintext.
//    c) Write the plaintext to output.
//    d) Update 'prev' to the current ciphertext block for the next round.
__attr_hot static inline void aes_cbc_decrypt(aes_ctx *ctx, const byte *in, byte *out, size_t len, byte
iv[16]) __noexcept
{
    byte prev[16];
    memcpy(prev, iv, 16); // Start with IV
    for (size_t i = 0; i < len; i += AES_BLOCK_SIZE)
    {
        byte block[16];
        aes_decrypt_block(ctx, in + i, block); // Decrypt ciphertext block
        for (int j = 0; j < 16; ++j)
```

```

        out[i + j] = block[j] ^ prev[j];    // XOR with previous ciphertext (or IV)
        memcpy(prev, in + i, 16);          // Update prev for next block
    }
}

```

CTR MODE

For CTR mode, we will first define a function for incrementing the counter value:

Operation: Increment Counter (CTR Mode)

This function increments a 16-byte counter, typically used in AES-CTR (Counter) mode. It treats the counter as a big-endian integer and increments it by one, handling overflow (carry) from lower to higher bytes.

```

Code // Increment a 16-byte counter (big-endian).
// 1. Start from the least significant byte (the last byte).
// 2. Increment the byte; if it becomes non-zero, stop (no further carry needed).
// 3. If the byte overflows to zero, continue to the next higher byte.
static inline void increment_counter(byte counter[16]) __noexcept
{
    for (int i = 15; i >= 0; --i)
    {
        if (++counter[i])
            break; // Stop if no carry is needed
    }
}

```

Operation: AES CTR Encrypt (Counter Mode)

This function implements AES encryption in CTR (Counter) mode. In CTR mode, a counter (starting from a nonce) is encrypted to produce a keystream block, which is then XORed with the plaintext to get the ciphertext. The counter is incremented for each block.

```

Code // Encrypt (or decrypt) data using AES in CTR (Counter) mode.
// 1. Initialize 'counter' with the given nonce.
// 2. For each 16-byte block of input:
//    a) Encrypt the current counter value to produce a 16-byte keystream.
//    b) XOR the input bytes with the keystream bytes and write to output.
//    c) Increment the counter for the next block.
__attr_hot static inline void aes_ctr_encrypt(aes_ctx *ctx, const byte *in, byte *out, size_t len, byte
nonce[16]) __noexcept
{
    byte counter[16];
    memcpy(counter, nonce, 16); // Start counter with nonce
    for (size_t i = 0; i < len; i += AES_BLOCK_SIZE)
    {
        byte keystream[16];
        aes_encrypt_block(ctx, counter, keystream); // Generate keystream
        for (size_t j = 0; j < AES_BLOCK_SIZE && i + j < len; ++j)
            out[i + j] = in[i + j] ^ keystream[j]; // XOR with input
    }
}

```

```
        increment_counter(counter);           // Increment counter
    }
}
```

In CTR mode, encryption and decryption use the same function (XOR with keystream), so no need to create another function for decryption.

That's it, we are done, that's how AES basically works, of course this does not implement the best security practices, but is enough to understand how AES works.

Let me know if you liked this post, and if you want me to write more like this, maybe for other algorithms like ChaCha20, SERPENT, RSA, etc... i have plenty of time this month! ?

full code here: [AES full code!](#)