AES-256 RTL

Design and Microarchitecture Reference

Author:

Aleksandar Lilic

Table of Contents

Table of Figures

1. AES overview

Advanced Encryption Algorithm (AES) is a specification for the cryptographic algorithm that can be used to protect electronic data. As it operates on a block of data, it's categorized as block cipher. It specifies both encryption and decryption operations. Input data is called plaintext while the output of the encryption process is called ciphertext. AES specifies three possible key lengths: 128, 192, and 256 bits and a single block size of 128 bits.

This document details only the implemented version, encryption in AES-256. AES-256 algorithm is shown on the Figure 1.

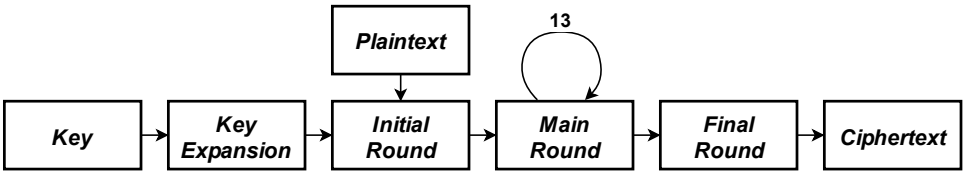


Figure 1 – AES-256 algorithm

AES encryption uses the notion of state matrix when operating on data. For the input A_0, A_1, … A_{15}, where each element is 8-bit wide, state is shown on the Figure 2.

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

Figure 2 – AES state

1.1. Encryption

AES defines four different transformations during encryption process, organized in rounds as shown in the Figure 3:

1. `AddRoundKey` – XOR of the current round key and the current state
2. `SubBytes` – non-linear transformation of the current state where each byte is substituted for another via S-Box (substitution box with known good non-linear properties)
3. `ShiftRows` – Shifts each row of the state matrix to the left (as shift wraps around, this type of operation is often referred to as rotation)
4. `MixColumns` – Each column of the current state matrix is multiplied by the 4x4 matrix with constant values

Encryption can start only after the key has been first expanded. With one key, one or more plaintexts can be encrypted.

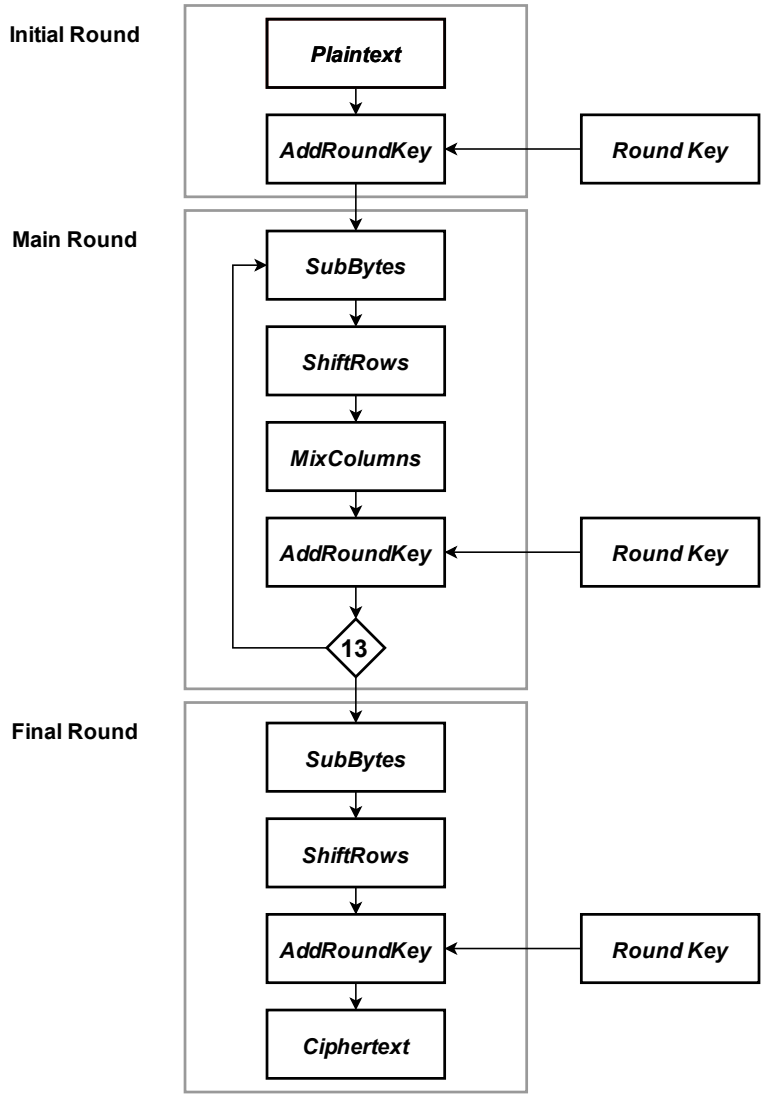


Figure 3 – AES-256 Transformations

```
1   encryption(plaintext[16], key)
2   {
3       Nr = 14; // number of rounds
4       state [16]; // algorithm internal state (4x4 matrix)
5       round_keys [15];
6       key_schedule(key, round_keys); // key expansion
7       // encryption
8       state [16] = plaintext;
9       AddRoundKey(state, round_keys[0]); // initial round
10      for(i = 0; i < Nr-1; i++){ // main rounds loop
11          SubBytes(state);
12          ShiftRows(state);
13          MixColumns(State);
14          AddRoundKey(state, round_keys[i+1]);
15      }
16      // final round
17      SubBytes(state);
18      ShiftRows(state);
19      AddRoundKey(state, round_keys[15]);
20      // state contains CT
21      return state;
22  }
```

Listing 1 – Encryption pseudo code

1.2. Key Expansion

Encryption requires fifteen 128-bit round keys, one for each round, while input key is 256-bit wide. In order to generate keys for each round, key expansion uses that initial 256-bit key and expands it further, using similar transformations like the encryption, though now operating on 32-bit words instead of single 8-bit byte. Sub-rounds with transformations order are shown on the Figure 4 while the rounds order and the overall algorithm are shown on the Figure 5. Key expansion is required to be done only once for each new key, before the encryption process can start.

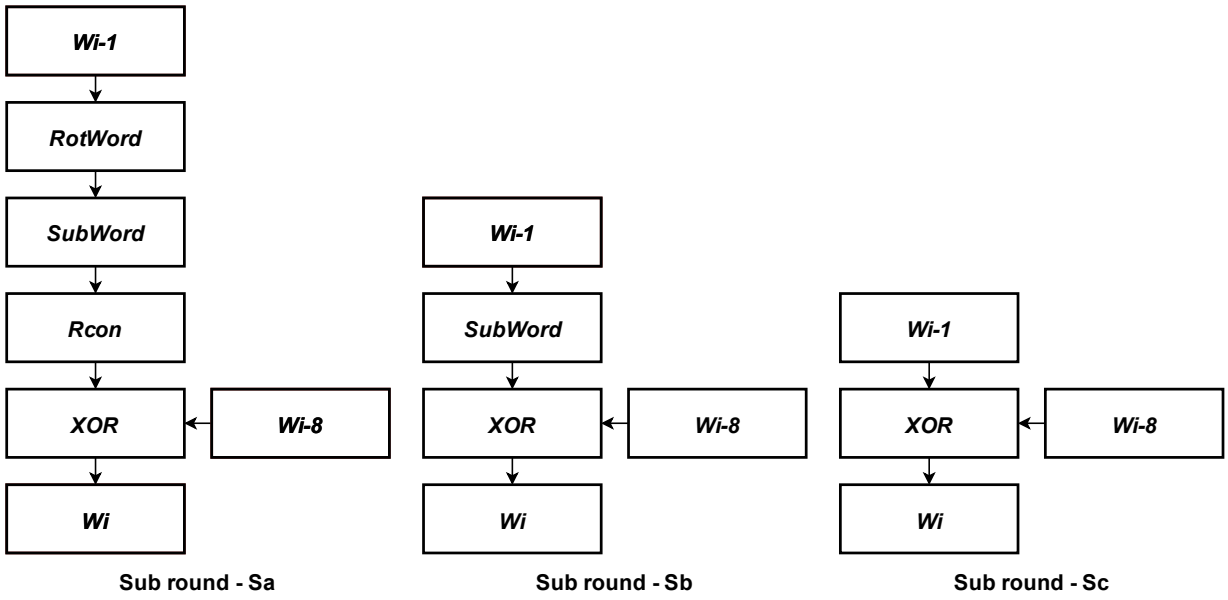


Figure 4 – Sub-rounds and transformations

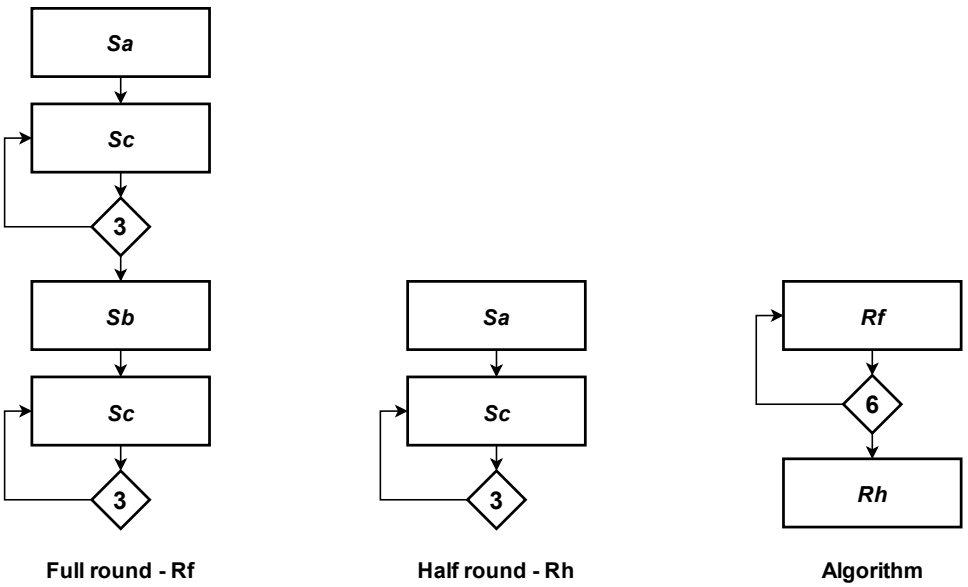


Figure 5 – Rounds – order and number of sub-rounds, and the overall algorithm

Each sub-round produces one word, while the next sub-round uses that word as its input. The very first round of key expansion uses the last word of the input key as its input. One full round produces eight words, while half round (half round is just first 4 transformations of the full round) produces four. The algorithm loops over the full round six times, producing 48 words, and once over half round to get the final four words, for total of 52 words during expansion. Together with 256-bit input key (eight words), this makes 60 words total, which are then split into fifteen 128-bit round keys for the encryption algorithm.

```
key_schedule(key)
{
    x = 60; // number of 32-bit words in all round keys
    Nk = 8; // number of 32-bit words in AES-256 key
    Nr = 15; // number of 128-bit round keys
    Rcon [7]; // round constants
    input_word;
    word [x]; // words array
    round_keys [Nr]; // round keys array
    for(i = 0; i < Nk; i++){ // AES-256 key parsing, used for first 8 words
        word[i] = key >> 32*(8-i);
    }
    for(i = Nk; i <= x; i++){ // expansion continues from 9th word
        input_word = word[i-1];
        if (i % 8 == 0) {
            RotWord(input_word);
            SubWord(input_word);
            RconXOR(input_word, Rcon);
        } else if (i % 4 == 0) {
            SubWord(input_word);
        }
        word[i] = input_word ^ word[i-Nk];
    }
    // concatenation of 4 words into one round key
    for (i = 0; i < Nr; i++) {
        round_keys = word[4*i] & word[4*i+1] &
                     word[4*i+2] & word[4*i+3];
    }
    return round_keys;
}
```

Listing 2 – Key Expansion pseudo code

2. Top level design

Hardware (RTL) implementation closely resembles the program flow of the algorithm as described in the NIST specification. Design is also expanded with data loading block, which takes the 128-bit ciphertext and sends it over the 8-bit data line in 16 packets over 16 clock cycles. Top level design implemented in the VHDL is shown on the Figure 6.

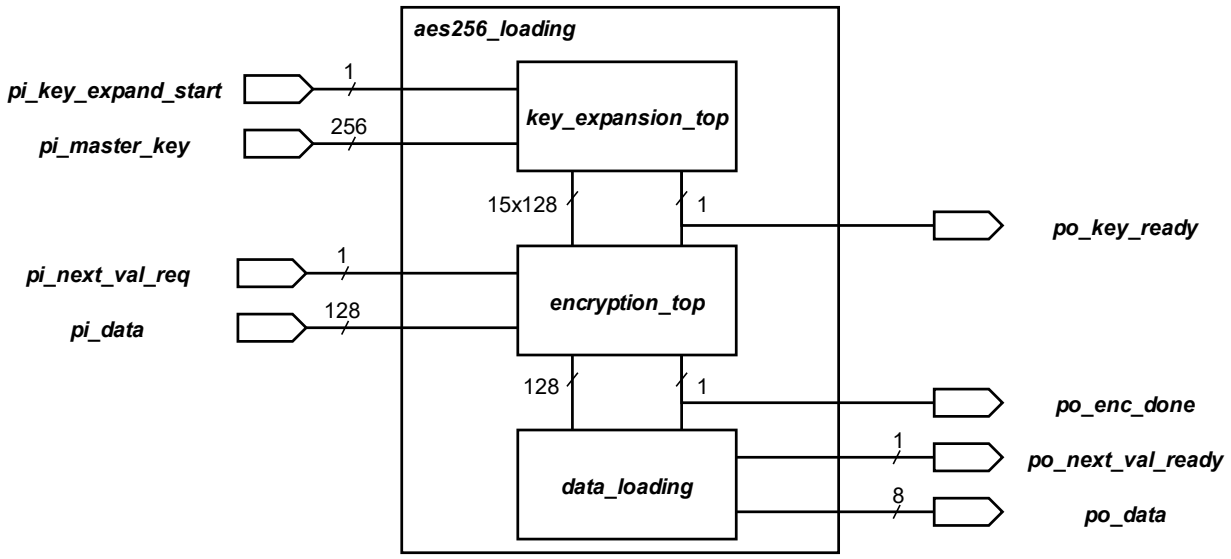
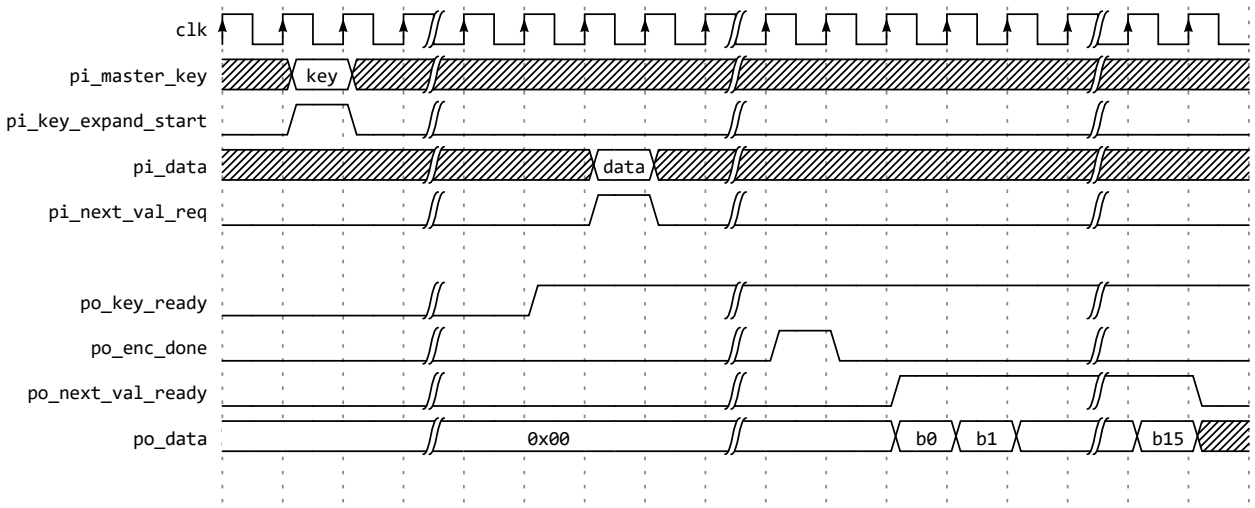


Figure 6 – Top level design

2.1. Operation

Common operation case is as follows:

1. AES-256 key is set and the expansion is requested. Upon completing the key expansion, block is ready to take requests for encryption
2. AES-256 plaintext is set and the encryption is requested. Upon completing the encryption, block is ready to take new requests for encryption
3. Loading module sends sixteen 8-bit packets over the next sixteen consecutive cycles after encryption has finished



For other scenarios, it's best to consider three main blocks (expansion, encryption and loading) as three mostly independent entities that can take new requests at any point, as explained in the next three paragraphs.

Key Expansion

Requests for new key expansion are essentially a soft reset. When new request comes in, previously expanded key is considered invalid. Key ready is required to go low in the next cycle. Any encryption and loading processes still ongoing are aborted and their respective ready signals are required to go low in the next cycle. Requests for new key expansion and new encryption at the same time are invalid.

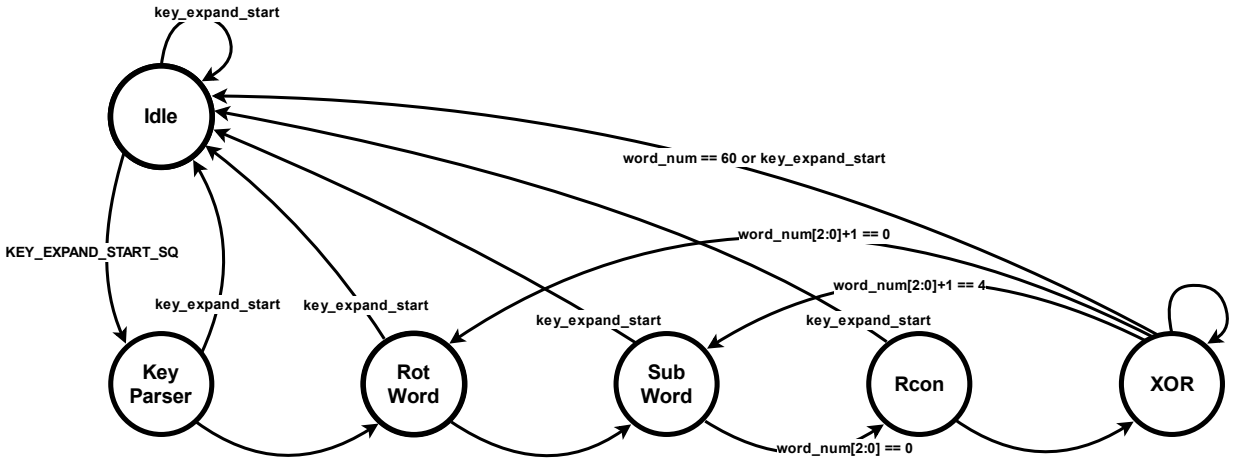


Figure 7 – Key Expansion FSM

Encryption

Requests for new encryption while key is not ready are invalid.

Requests for new encryption while previous plaintext is being encrypted results in aborting the ongoing encryption and the start of the new one.

Requests for new encryption while loading of the previous ciphertext is ongoing have no impact on the data loading block. This operation is commonly used to overlap encryption of the current plaintext with the sending of the previous ciphertext. Also known as pipelining, it reduces the effective time it takes to produce new ciphertext during continuous operation.

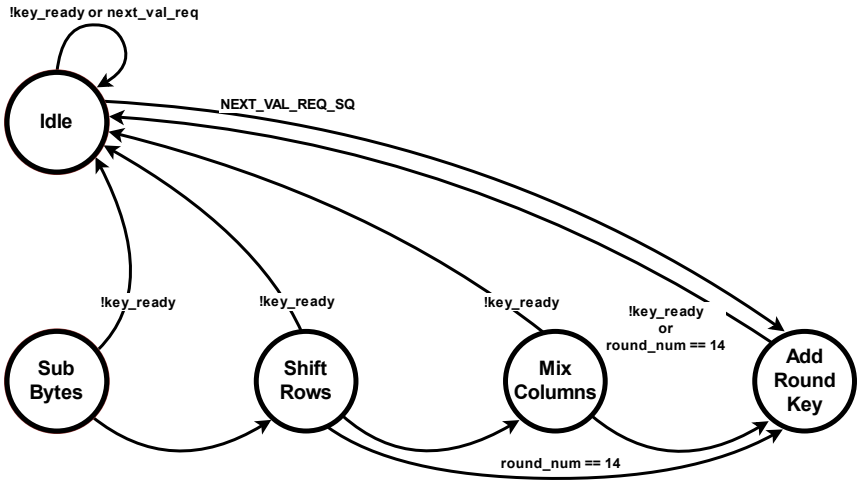


Figure 8 – Encryption FSM

Loading

Loading block is directly controlled only by other blocks internal to the design.

Loading starts upon receiving ready signal from encryption block. Data is sampled internally in the Loading block on the falling edge of the ready signal.

Purpose of the loading block is to prepare data to be sent over communication protocols that use 8-bit bus like UART, I2C, SPI, etc.

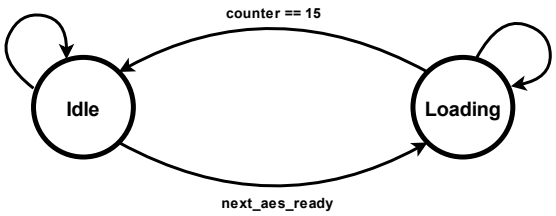


Figure 9 – Data loading FSM

2.2. Ports

Name	Direction	Width	Description
clk	**in**	**1**	System clock
rst	**in**	**1**	System reset
pi_key_expand_start	**in**	**1**	Request pulse for the start of the key expansion Requires sequence 0->1->0 to start Low/High states can be any duration
pi_master_key	**in**	**256**	AES-256 key Sampled at the 1->0 transition of the **pi_key_expand_start**
po_key_ready	**out**	**1**	Expansion of the round keys has finished Block is ready to take in requests for new ciphertexts
pi_next_val_req	**in**	**1**	Request pulse for the start of the encryption Requires sequence 0->1->0 to start Low/High states can be any duration
pi_data	**in**	**128**	AES-256 plaintext sampled at the 1->0 transition of the **pi_next_val_req**
po_enc_done	**out**	**1**	AES-256 ciphertext is produced Block is ready to take in requests for new ciphertexts
po_next_val_ready	**out**	**1**	Packets of 8-bit data are available on the **po_data** High for 16 consecutive cycles if not interrupted
po_data	**out**	**8**	8-bit slice of the AES-256 ciphertext, starting from MSB