|  |  |
| --- | --- |
| AES-256-CTR mode encryption/ decryption Core  VHDL Implementation | Abstract  Open-source pipelined VHDL Implementation of AES 256 CTR mode encryption and decryption.  Author: Stian K. Endresen |

Table of Contents

[2 CTR-mode Encryption Theory 2](#_Toc209026908)

[2.1 Overview 2](#_Toc209026909)

[2.2 Implementation 2](#_Toc209026910)

[3 VHDL Implementation 3](#_Toc209026911)

[3.1 Overview 3](#_Toc209026912)

[3.1.1 Config Register and Control Register 3](#_Toc209026913)

[3.1.2 Description of Operation 4](#_Toc209026914)

[3.2 Performance 4](#_Toc209026915)

[3.2.1 Resource Usage 4](#_Toc209026916)

[3.2.2 Throughput and Latency 5](#_Toc209026917)

[4 Acknowledgements 6](#_Toc209026918)

[5 Footnotes 7](#_Toc209026919)

# CTR-mode Encryption Theory

## Overview

CTR is one of the most common modes of AES encryption. It requires a key (256 bits for AES-256) and an Initialization Vector (IV). The width of the IV matches the block cipher width (128 bits with AES). The IV is divided into nonce and a counter. A common choice is 96 bits for the nonce and 32 bits for the counter. The main two things to keep in mind when using CTR is:

* The same key/nonce pair should never be reused for more than one plaintext stream.
* With a given key/nonce pair, you should never encrypt more than AES-blocks, where CL is the Counter Length (bits). Since each AES-block is 128 bits, this equates to 68 GB with a 32-bit counter.

Failure to adhere to any of these requirements will result in a critical security violation. Furthermore, keep in mind that while CTR provides excellent data confidentiality, it lacks any form of authentication, so we cannot detect if the ciphertext is tampered with.

## Implementation

CTR-mode encryption works by generating a *keystream* from the key and IV. The ciphertext is the keystream XOR plaintext. The keystream is generated in blocks of 128 bits/16 bytes. To generate the first block, the (once||counter) is sent into the AES Core. To generate the second keystream block, the (nonce||counter+1) is sent as inputs to the AES Core, and (nonce||counter+2) for the third block, etc. An illustration of this is provided below. A common choice is to set the initial counter value to 0 at the start of the encryption, but other initial values for the counter are also allowed.

To decrypt the ciphertext, the exact same procedure is used as when encrypting. The plaintext is now keystream XOR ciphertext. In mathematical terms, one can write this as:

This symmetry between encryption and decryption means that the same hardware instance can be used for both encryption and decryption without modification, and even without switching operation mode.

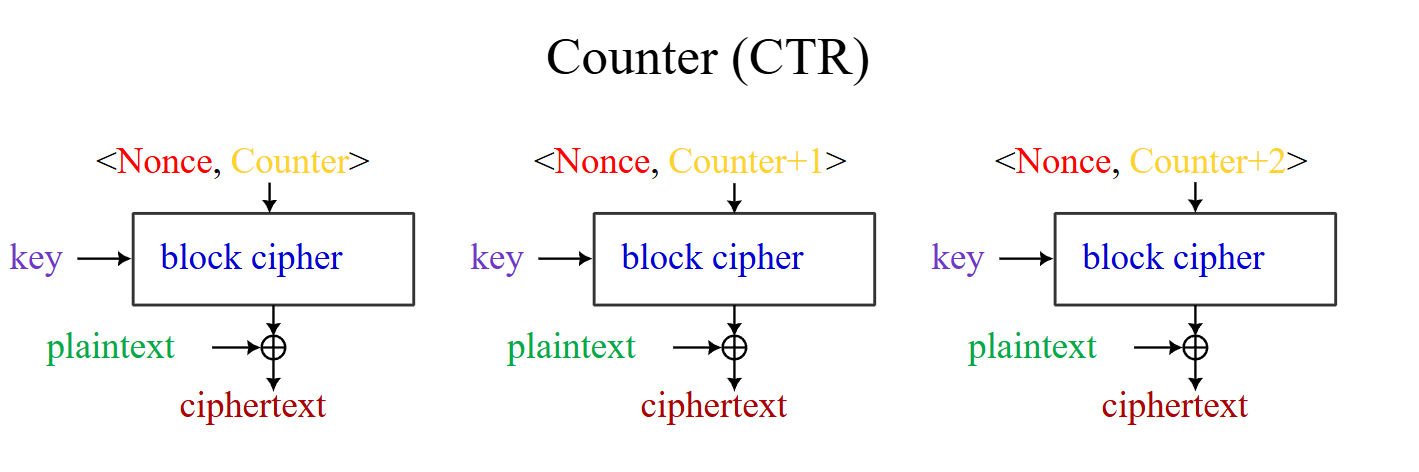


Figure 1: The workings of CTR-mode encryption illustrated [Image by A.M. Rowsell and Epachamo[[1]](#endnote-1)].

# VHDL Implementation

## Overview

The high-level interface of the implemented core is as illustrated in Figure 2. A documentation of the purpose and operation of the different signals follows.

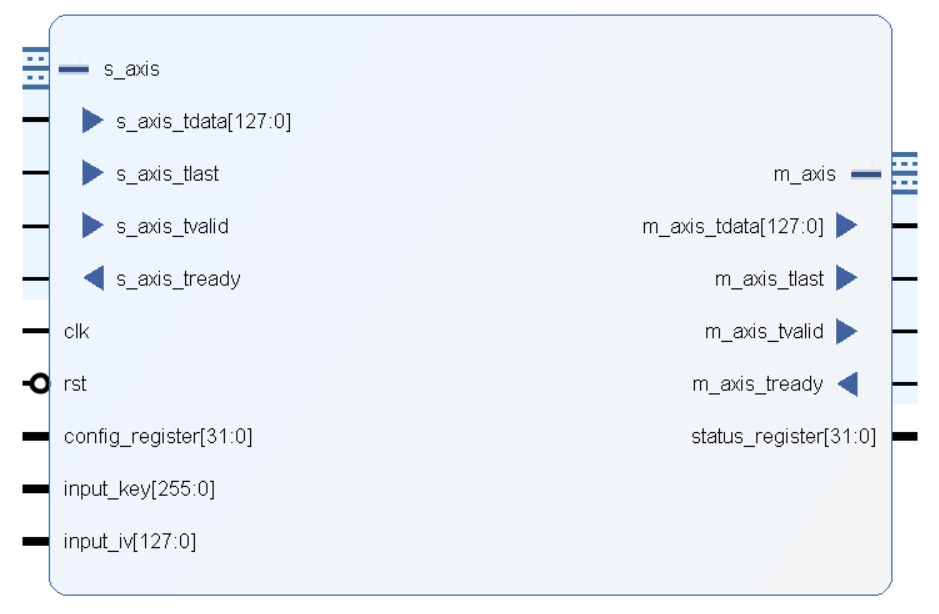


Figure 2: Illustration of the ports and interfaces for the AES-256-CTR core.

### Config Register and Control Register

Config Register

|  |  |  |
| --- | --- | --- |
| Bit | Default value | Signal name |
| 0 | 0 | load\_key\_and\_iv |
| 1 | 0 | tx\_raw\_keystream |
| 2-31 | 0 | Not in Use |

Status Register

|  |  |  |
| --- | --- | --- |
| Bit | Default value | Signal name |
| 0 | 0 | key\_ready |
| 1 | 0 | tx\_raw\_keystream |
| 2-31 | 0 | Not in Use |

### Description of Operation

#### Key expansion

Before encryption or decryption can occur, a key/IV pair must be loaded. This is done by setting the load\_key\_and\_iv bit in the control register to 1 for at least one cycle while the input\_key and input\_iv inputs are set to the desired key and IV. The input\_iv input sets both the nonce and the initial counter value. When load\_key\_and\_iv in the control register is asserted, key\_ready in the status register is immediately set to 0. When load\_key\_and\_iv is de-asserted, the key and IV pair are loaded and key expansion starts. When the key expansion completes (after 84 clock cycles), the key\_ready signal in the status register is set to 1, and the AES core is ready for encrypting/decrypting. The input\_key and input\_iv inputs can now be set to zero, as they are unused until load\_key\_and\_iv is re-asserted.

If load\_key\_and\_iv is set to 1 during key expansion or normal operation, the AES Core immediately aborts its current operation in preparation for key expansion with the new key.

#### Normal operation

Following key expansion, it will take another 60 cycles to generate the first keystream blocks. When a keystream block is ready, the s\_axis interface will accept incoming plaintext data and xor with the keystream to encrypt. The resulting chiphertext is available on m\_axis on the next cycle. For decryption, the same procedure and data flow as for encryption is used.

#### Key and IV management

The counter in the IV is auto-incremented whenever a block is encrypted. Readback of the active counter value is not currently supported. To change the key or IV, one must repeat the steps in section 3.1.2.1.

#### Transmit keystream mode

The CTR Core supports a special operation mode, designed for situations where the limiting factor on throughput is the speed at which data can be moved between the PL and PS. If the tx\_raw\_keystream bit in the control register is set to 1, the core will ignore the s\_axis interface, and start transmitting the raw keystream directly on the m\_axis interface. The PL must perform XOR between the plaintext and keystream to generate the ciphertext (or vice verca), but the amount of data that must move between the PL and PS is halved. The tx\_raw\_keystream bit in the status register is a readback of the value of tx\_raw\_keystream in the control register.

## Performance and Utilization

The tradeoff between resource utilization and throughput is highly configurable. By increasing the generic NUM\_AES\_CORES to more than one, several AES Cores will be used in the design. Using N cores will increase the throughput N-fold, but will also increase resource usage according to the numbers in the row “AES\_cores[0].INST” in Figure 3. The maximum limit for NUM\_AES\_CORES is 15, which results in one AES block being encrypted per clock cycle.

### Resource Utilization

The implemented design is synthesized for a zynq7000 device to get an estimate of the resource usage. The result is given below, in the row aes256\_ctr\_mode:

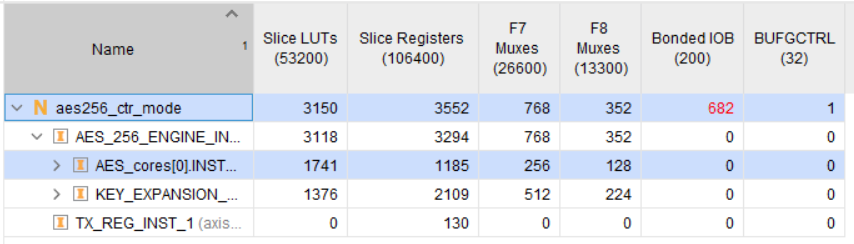


Figure 3: Resource utilization when synthesized for a zynq7000

### Throughput and Latency

It takes a total of 60 clock cycles to generate a keystream block, but the AES Core is pipelined to generate four keystream blocks at once. This results in an average throughput of one 128-bit AES block per 15 clock cycles. At e.g. 200 MHz, this equates to Gbps.

The listed throughput is with a single AES-core. With more cores, the throughput increases accordingly.

# Acknowledgements

This design builds on the open-source VHDL implementation of the AES256-Core by Aleksandar Lilic, containing both key generation and AES-encryption. Compared to his design, this design features:

* The AES Core is pipelined, giving 4x throughput with only a minor utilization increase.
* Implemented option to use more than one AES-Core in the design, increasing both throughput and resource utilization.
* Implemented AXI4-Stream interfaces for the plaintext and ciphertext.
  + Subsequently, the old *data\_loading* and *aes256\_loading* modules are no longer used.
* Implemented CTR-mode encryption on top of the AES Core.
* The key generation entity is identical.

# Footnotes

1. Image by A.M. Rowsell and Epachamo, taken from: <https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:BlockCipherModesofOperation.svg>, license: [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0) [↑](#endnote-ref-1)