|  |  |
| --- | --- |
| 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 Introduction 2](#_Toc209884482)

[2.1 Features 2](#_Toc209884483)

[3 CTR-mode Encryption Theory 3](#_Toc209884484)

[3.1 Overview 3](#_Toc209884485)

[3.2 CTR mode details 3](#_Toc209884486)

[4 Documentation of VHDL Implementation 4](#_Toc209884487)

[4.1 Ports and Interfaces 4](#_Toc209884488)

[4.2 Register Space 5](#_Toc209884489)

[4.2.1 Config Register and Control Register 5](#_Toc209884490)

[4.3 Description of Operation 6](#_Toc209884491)

[4.3.1 Key expansion 6](#_Toc209884492)

[4.3.2 Normal operation 6](#_Toc209884493)

[4.3.3 Key and IV management 6](#_Toc209884494)

[4.3.4 Transmit keystream mode 6](#_Toc209884495)

[4.4 Performance and Utilization 7](#_Toc209884496)

[4.4.1 Throughput considerations 7](#_Toc209884497)

[5 Acknowledgements 8](#_Toc209884498)

[6 Footnotes 9](#_Toc209884499)

# Introduction

The AES-256-CTR Encryption Core offers a ready-made open-source VHDL implementation of CTR encryption and decryption with AES256. The design exposes generics for tailoring the design to the users needs.

## Features

* A ready-made FPGA CTR encryption/decryption solution.
* A compile-time customizable CTR Counter Width.
* 128-bit AXI4-Stream interfaces for the plaintext and ciphertext.
* AXI4-Lite interface that allows for reconfiguration of the key and IV in use.
* A user-configurable number of AES Cores, giving a high degree of customization over the tradeoff between resource usage and throughput.
* A keystream buffer for avoiding throughput loss imposed by wait-cycles when using AXI4-Stream Data Width Convertors on he input or output.

# 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.

## CTR mode details

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.

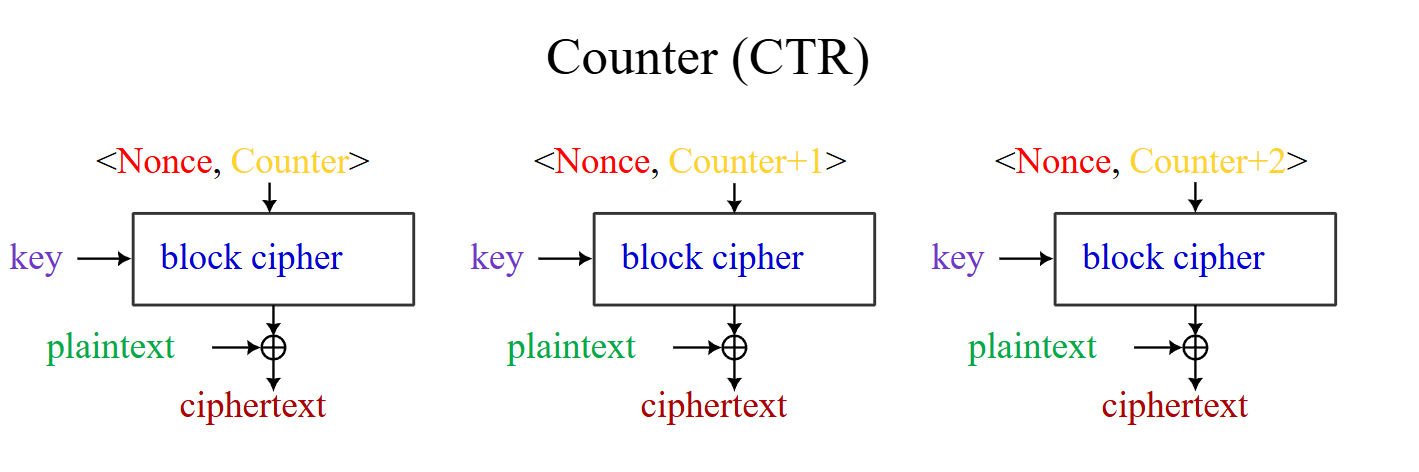


Figure 1: The workings of CTR-mode encryption illustrated [Image credit: A.M. Rowsell and Epachamo[[1]](#endnote-1)].

# Documentation of VHDL Implementation

## Ports and Interfaces

The high-level interfaces of the implemented design are illustrated in Figure 2. A documentation of the purpose and operation of the inputs and outputs is given below.

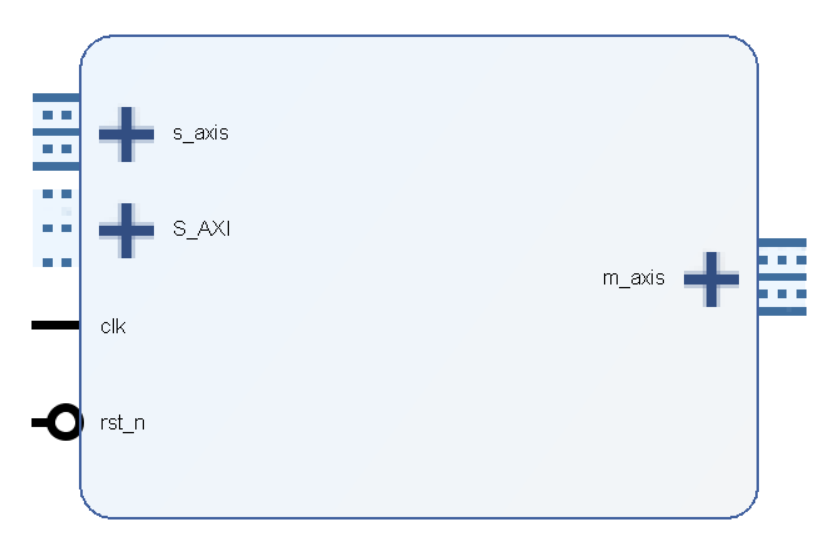


Figure 2: Illustration of the ports and interfaces for the AES-256-CTR core.

As shown by S\_AXI in the block design in Figure 2, the encryption core exposes an AXI4 Lite interface for setting the key, IV and control signals for the encryption core, as well as a readback of status signals. See chapter 4.2 and 4.3 for a detailed description of this.

The s\_axis and m\_axis interfaces are both 128-bit interfaces used for the plaintext and ciphertext. When encrypting data, plaintext is sent to the s\_axis interface, and ciphertext is being generated on m\_axis. When decrypting data, ciphertext is inputted on s\_axis while the decrypted plaintext is generated on m\_axis.

The reset signal rst\_n is active low, and is used by both the AXI4 Lite interface, the AXI4-Stream interface and the internal AES256 encryption process.

## Register Space

The following table describes the configuration registers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Register Offset (hex) | Register Name | Reset Value (hex) | Access Type | Description |
| 0x00 | Control Register | 0x0000\_0000 | R/W | See section 4.2.1 and 4.3 for a description of the control register. |
| 0x04 | Status Register | 0x0000\_0000 | R | See section 4.2.1 and 4.3 for a description of the status register. |
| 0x08 | Key Part 0 | 0x0000\_0000 | R/W | Bits 0 to 31 of the 256-bit encryption key. |
| 0x0C | Key Part 1 | 0x0000\_0000 | R/W | Bits 32 to 63 of the 256-bit encryption key. |
| 0x10 | Key Part 2 | 0x0000\_0000 | R/W | Bits 64 to 95 of the 256-bit encryption key. |
| 0x14 | Key Part 3 | 0x0000\_0000 | R/W | Bits 96 to 127 of the 256-bit encryption key. |
| 0x18 | Key Part 4 | 0x0000\_0000 | R/W | Bits 128 to 159 of the 256-bit encryption key. |
| 0x1C | Key Part 5 | 0x0000\_0000 | R/W | Bits 160 to 191 of the 256-bit encryption key. |
| 0x20 | Key Part 6 | 0x0000\_0000 | R/W | Bits 192 to 223 of the 256-bit encryption key. |
| 0x24 | Key Part 7 | 0x0000\_0000 | R/W | Bits 224 to 255 of the 256-bit encryption key. |
| 0x28 | IV Part 0 | 0x0000\_0000 | R/W | Bits 0 to 31 of the 128-bit IV. This will contain the counter-part of the IV, though parts 1-3 may also contain the counter if the counter width exceeds 32. |
| 0x2C | IV Part 1 | 0x0000\_0000 | R/W | Bits 32 to 63 of the 128-bit IV. |
| 0x30 | IV Part 2 | 0x0000\_0000 | R/W | Bits 64 to 95 of the 128-bit IV. |
| 0x3C | IV Part 3 | 0x0000\_0000 | R/W | Bits 96 to 127 of the 128-bit IV. |

### Config Register and Control Register

Control 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 desired key and IV are loaded in their respective. Both the nonce and the initial counter value are set from the provided IV. 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 into the device 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 AXI registers containing the key can now be set to zero, as they are unused until load\_key\_and\_iv is re-asserted. The AXI registers containing the nonce-part of the IV, however, must not be changed during encryption or decryption.

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. A new key and IV pair can be loaded as many times as desired. The recommended way of doing this is by first setting the load\_key\_and\_iv bit in the control register high, and then write new values to the key and IV registers before finally de-asserting the load\_key\_and\_iv bit in the control register.

### Normal operation

Following completion of key expansion, it will take 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 supported. To change the key or IV, one must repeat the steps in section 4.3.1.

### 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 transmit the keystream directly on the m\_axis interface. The PL must perform XOR between the plaintext and keystream to generate the ciphertext (or vice verca for decryption), but the amount of data that must move between the PL and PS is cut in half. The tx\_raw\_keystream bit in the status register is a readback of the value of tx\_raw\_keystream in the control register.

Note that if this approach is used, care must be taken when changing key/IV pair, that every keystream block from the old encryption session is flushed out from FIFO buffers in the FPGA etc. to avoid mixing the keystreams.

## Performance and Utilization

The tradeoff between throughput and resource utilization is highly configurable. By increasing the generic NUM\_AES\_CORES, several AES Cores will be used, increasing throughput and resource usage. Allowed values for NUM\_AES\_CORES are {1-5, 8, 15}. At 15 cores, a full 128-bit block is encrypted every clock cycle.

The implemented design is synthesized for a zynq7000 to estimate the resource usage The result is shown below, as well as an estimated throughput for each configuration.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of cores | LUTs | Slice Registers | F7 Muxes | F8 Muxes | Throughput multiplier | Throughput at 250 MHz |
| 1 | 3092 | 2911 | 768 | 352 | 1 | 2.133 Gbps |
| 2 | 3977 | 3446 | 1152 | 480 | 1.875 | 4.0 Gbps |
| 3 | 4866 | 3981 | 1280 | 608 | 3 | 6.4 Gbps |
| 4 | 6008 | 4516 | 1536 | 736 | 3.75 | 8.0 Gbps |
| 5 | 7156 | 5051 | 1792 | 864 | 5 | 10.667 Gbps |
| 8 | 8800 | 6244 | 2304 | 1120 | 7.5 | 16.0 Gbps |
| 15 | 15522 | 9768 | 2560 | 1248 | 15 | 32.0 Gbps |

### Throughput considerations

CTR encryption essentially works by generating a stream of random data (keystream), and XORing it with the plaintext blocks. The implemented design pipelines four 128-bit keystream blocks at once, meaning that 4 blocks of data can be encrypted every 60/32/20/16/12/8/4 cycles with 1/2/3/4/5/8/15 cores. NB: The keystream generator can only hold four blocks, and will halt keystream generation when a keystream block is ready, but s\_axis is empty or m\_axis is blocking.

This becomes important if data width convertors are used on the input or output, as with e.g. 32-to-128-bit convertors on the input, plaintext blocks are only available to the CTR encryptor every 4 cycles. Since all four keystream blocks must be used before new keystream blocks are generated, this imposes 9 idle cycles, decreasing throughput. When data width convertors are used on the output, the same behavior is observed. This complication can be solved by adding a 3-block wide FIFO buffer for the keystream blocks, which is achieved by setting the ADD\_KEYSTREAM\_BUFFER generic to True. The FIFO buffer uses an additional 150 LUTs and 400 slice registers. 3 blocks buffer suffice, regardless of the number of cores or the data width conversion ratio.

When using all 15 cores, the FIFO is never needed since one block is encrypted each clock cycle anyway.

# 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 implementation, this design features:

* A pipelined AES Core, giving 4x throughput with no meaningful increase in resource usage.
* A customizable number of AES-Cores, exposing a user-configurable tradeoff between throughput and resource usage.
* Standard AXI4-Stream interfaces for the plaintext and ciphertext.
  + Subsequently, the old *data\_loading* and *aes256\_loading* modules are no longer used.
* Implemented AXI-Lite slave interface for writing key/iv/config and reading status
* Implemented CTR-mode encryption on top of the AES Core.

The AXI-Lite slave interface is based on the design by mzeghers (<https://github.com/mzeghers/hdl-axi-regs>), though some modifications are performed. Notably, bresp and rresp now return SLVERR when incorrect addresses are given, and bvalid is only driven high when applicable.

# 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)