Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API version 2.0

Ekawat Homsirikamol, Panasayya Yalla, Farnoud Farahmand, William Diehl, Ahmed Ferozpuri, Jens-Peter Kaps, and Kris Gaj

Electrical and Computer Engineering Department,
George Mason University
Fairfax, Virginia 22030
{ehomsiri, pyalla, ffarahma, wdiehl, aferozpu, jkaps, kgaj}@gmu.edu

December 5, 2017

Contents

Preface and Acknowledgement

This document accompanies the Development Package for Hardware Implementations Compliant with the CAESAR Hardware API v.2.0 [1]. It replaces the previous version entitled Implementer's Guide to the CAESAR Hardware API, v.1.1, which accompanied the Development Package for the CAESAR Hardware API, v1.0, last revised on June 10, 2016. The previous versions of the Implementer's Guide and the Development Package were successfully used in hardware implementation and benchmarking of the majority of Round 2 and Round 3 CAESAR candidates.

In Round 2, 14 groups from all over the world contributed a total of 43 hardware design packages, covering 28 candidate families and 75 variant-architecture pairs [2–5]. In Round 3, 10 groups developed 27 hardware design packages, covering all 15 candidate families [4–6]. The majority of these groups used the Development Package v1.0. Some of the groups modified the design files belonging to the Package (e.g., the PreProcessor and/or PostProcessor) to better suite the needs of the respective candidates. Others, reported their problems to the GMU Benchmarking Team, which led to a few customized versions of the design files, not released to the general public.

The current release is intended to incorporate all lessons learned to date, and pave the way for effective and time-efficient hardware benchmarking of CAESAR candidates in the final rounds of the competition.

Apart from the feedback received from the submitters of the VHDL/Verilog code, very important contributions were made by the group attempting the experimental validation of the CAESAR cores, using general-purpose All Programmable System on Chip boards, such as Pynq. We would like to express our special gratitude to our colleagues from Technische Universität München (TUM), Germany, for reporting multiple problems related to practical experimental testing of CAESAR cores [7, 8]. As a result of these

changes, any cores based on the Development Package v.2.0 are likely to be much better suited for experimental testing using arbitrary general-purpose and specialized FPGA and SoC boards.

Additionally, the current release introduces also a much requested extension of our framework to support lightweight cipher cores. This support is particularly desired for the CAESAR Use Case 1 candidates, targeting lightweight applications and constrained environments. In the current version of the Development Package, the supporting files are provided for both high-speed designs (assumed in versions 1.0), as well as for the lightweight designs (supported starting in v.2.0). Both types of designs share the same interface (up to the data port widths), testbench, and test-vector generator. They differ only in terms of the internal CipherCore interface, and the detailed designs for the PreProcessor and PostProcessor cores. For both types of designs, we provide separate examples of the hardware implementations of dummy authenticated ciphers. We would like to express our appreciation to Fabrizio De Santis and Michael Tempelmeier from TUM, who have helped us to shape the control logic for our framework, as well as to validate our example cores.

Major changes introduced in the new Development Package v2.0 are described in the next section. Additionally, for designers who have already developed implementations of Round 2 or Round 3 CAESAR candidates based on the Development Package v1.0, we provide a clear migration path.

Major Changes in v2.0

This major release provides additional support for the development of lightweight implementations of authenticated ciphers. We have also resolved various bugs that manifested themselves when the high-speed implementations based on the previous Development Package v1.0 were experimentally tested using FPGA and SoC boards.

The most important changes include:

- Introducing enhanced handshaking mechanism for the transfer of the message authentication result from the CipherCore to the PostProcessor during authenticated decryption
- fixing a bug that caused an incorrect behavior when the input data bus was idle in a certain state
- fixing a stall bug that manifested itself when DBLK_SIZE = W
- fixing a stall bug that manifested itself when ABLK_SIZE < DBLK_SIZE.

The change in the handshaking mechanism has necessitated a small change in our internal CipherCore API. In particular, *msg_auth** signals are now synchronized via AXI-like handshaking signals.

Migration from v1.0 to v2.0 for high-speed designs

The following ports of the CipherCore and the PostProceesor should be modified as follows:

- $msg\_auth\_valid \rightarrow msg\_auth$
- $msg\_auth\_done \rightarrow msg\_auth\_valid$

Then, when the authentication result is transferred to the PostProcessor, msg_auth and msg_auth_valid should be held constant until the msg_auth_ready is active. The high value of the msg_auth_ready port indicates that the PostProcessor is ready to receive the authentication result using the msg_auth port.

1 Introduction

The CAESAR Hardware API [9, 10] is intended to meet the requirements of all algorithms submitted to the CAESAR competition, as well as many earlier developed authenticated ciphers, such as AES-GCM, AES-CCM, etc. The major parts of its specification [9] include the minimum compliance criteria, interface, communication protocol, and timing characteristics supported by the core. All of these parts have been defined with the goals of guaranteeing (a) compatibility among implementations of the same algorithm developed by different designers, and (b) fair benchmarking of authenticated ciphers in hardware.

The CAESAR API is suitable for both high-speed and lightweight implementations of authenticated ciphers. The only difference at the API level is the width of the Public Data Input (PDI) and Data Output (DO) ports, which is defined as follows:

$$\text{Lightweight implementations: } w = 8, 16, 32$$
$$\text{High-speed implementations: } 32 \leq w \leq 256.$$

From the implementer's point of view, this difference is important, as small values of w (used in lightweight implementations) imply that any preprocessing (such as padding) and any postprocessing (such as zeroization of unused bytes) are significantly easier to implement compared to the case of large values of w (used in high-speed implementations).

The designers of both types of implementations are provided with the following support aimed at speeding-up and simplifying the development process:

- universal top-level block diagram
- universal VHDL code for the PreProcessing unit
- universal VHDL code for the PostProcessing unit

- hardware API for the heart of the design, called the CipherCore
- recommended design procedure for the CipherCore
- examples of VHDL code for the CipherCore of several dummy authenticated ciphers, fully compliant with the CipherCore API
- universal testbench
- universal test vector generator, based on the reference C implementations of the respective authenticated ciphers.

Below we describe all these supporting materials one by one. It should be stressed that the *implementations of authenticated ciphers compliant with the CAESAR Hardware API can be also developed without using any resources described in this document, by just following directly the specification of the CAESAR Hardware API [9, 10].*

2 Top-level Block Diagrams

2.1 High-Speed Implementations

The proposed top-level block diagram of a high-speed, non-pipelined implementation of a *single-pass* authenticated cipher compliant with the CAESAR Hardware API is shown in Fig. 2.1. The corresponding block diagram for a *two-pass* authenticated cipher is shown in Fig. 2.2. The only differences are in the ports used for communication with an external Two-Pass FIFO, which is used to store an output from the first pass of an implemented algorithm.

In each case, the top-level unit is divided into four lower-level units, called the PreProcessor, PostProcessor, Command (CMD) FIFO, and CipherCore. The universal VHDL code of the first three units are designed to be suitable for all authenticated ciphers to be implemented as a part of the CAESAR benchmarking project. This code is provided as a part of the supporting Development Package [1]. Due to the availability of this package, as well as the well-defined hardware API of the CipherCore itself (described in Chapter 5), the implementers of any specific authenticated cipher do not need to be concerned with the internal details of the PreProcessor, the PostProcessor, and the CMD FIFO. Instead they can focus exclusively on the development of the CipherCore unit, which can be further separated into its own datapath and controller, if desired.

Below is a high-level description of major functions of these units.

2.1.1 PreProcessor

The PreProcessor is responsible for the execution of the following tasks common for the majority of CAESAR candidates:

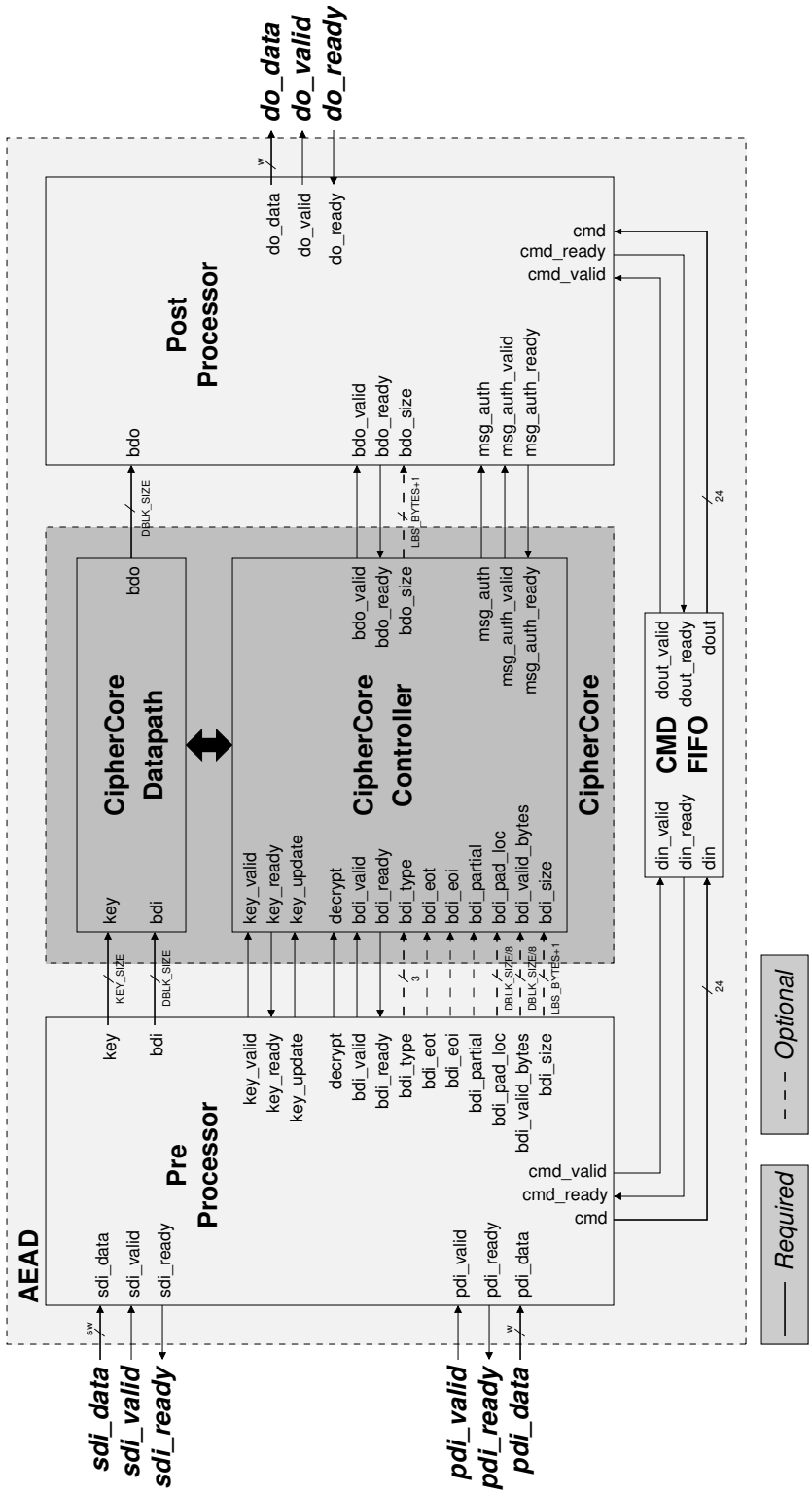


Figure 2.1: Top-level block diagram of a high-speed architecture of a *single-pass* authenticated cipher core, AEAD

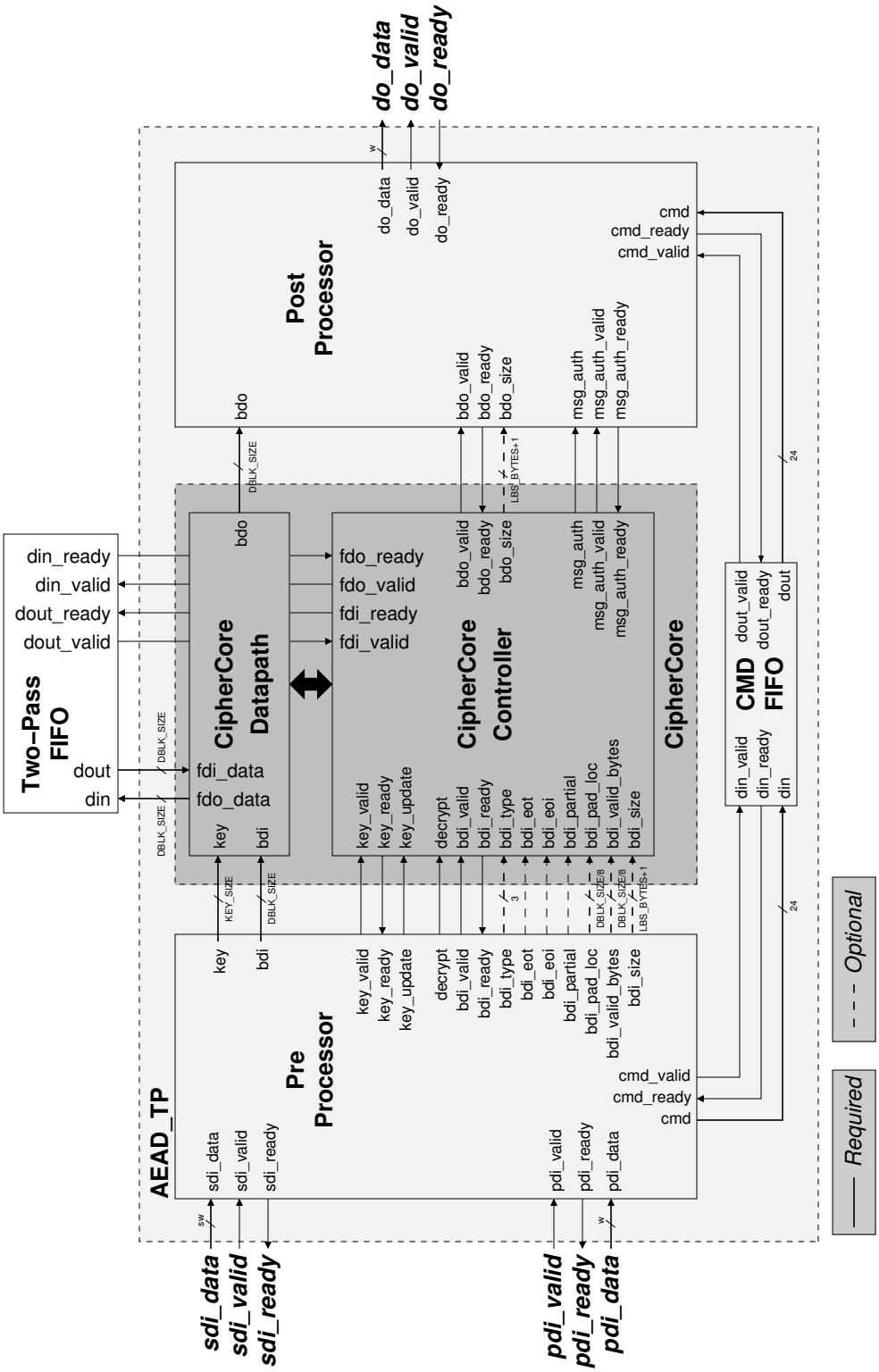
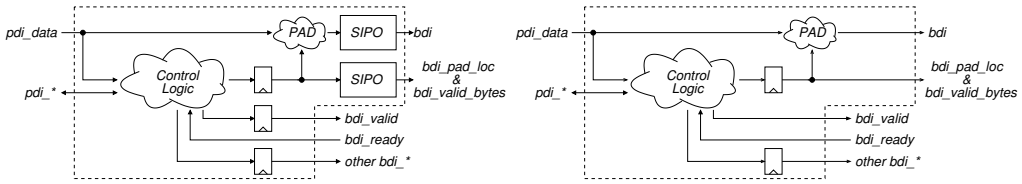


Figure 2.2: Top-level block diagram of a high-speed architecture of a *two-pass* authenticated cipher core, AEAD_TP



(a) registered (b) non-registered

Figure 2.3: The PreProcessor Design. SIPO = Serial-In Parallel-Out unit. pdi_* and bdi_* stand for all PreProcessor ports, shown in Fig. 2.1, with the names starting from the respective strings.

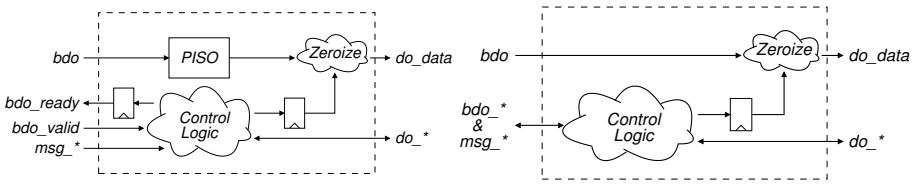
- parsing segment headers
- loading and activating keys
- Serial-In-Parallel-Out loading of input blocks
- padding input blocks, and
- keeping track of the number of data bytes left to process.

An overview of the PreProcessor design is shown Fig. 2.3. This unit can be configured to operate in two modes, registered and non-registered. The choice between these modes is made based on the width of public data input, pdi_data, (denoted as w in Fig. 2.1) and the size of an input block (denoted as $DBLK\_SIZE$ in Fig. 2.1).

In a typical scenario, where the size of an input block is larger than the width of pdi_data, w, the PreProcessor operates in the *registered* mode. If the width of pdi_data is the same as the size of an input block, the *non-registered* mode should be used. The non-registered mode ensures a high-throughput operation for algorithms that require a new block of data every clock cycle. It must be noted that operating the design in non-registered mode may affect the overall maximum clock frequency of the design due to additional critical path associated with the padding logic (if used).

2.1.2 PostProcessor

The PostProcessor is responsible for the following tasks:



(a) registered (b) non-registered

Figure 2.4: The PostProcessor Design. PISO = Parallel-In Serial-Out unit. msg_*, bdo_*, and do_* stand for all the PostProcessor ports, shown in Fig. 2.1, with the names starting from the respective strings.

- clearing any portions of output blocks not belonging to the ciphertext or plaintext
- Parallel-In-Serial-Out conversion of output blocks into words
- formatting output words into segments
- generating the status block with the result of authentication.

An overview of the PostProcessor design is shown Fig. 2.4. This unit can be configured to operate in either registered or non-registerd mode. The choice is made based on the dependence between the size of an output block (equal to the size of an input block, $DBLK\_SIZE$) and the width of the do_data port (equal to width of the public data input, pdi_data). Namely, when an output block size is larger than the width of do_data, the registered mode is preferable. Otherwise, the non-registered mode should be used. Similarly to the PreProcessor design, when the unit operates in the non-registered mode, the maximum clock frequency maybe be affected.

The PreProcessor and PostProcessor units are highly configurable using generics of AEAD. These generics can be used, for example, to determine:

- the widths of the pdi, sdi, and do ports
- the size of the associated data block, message/ciphertext block, key, and tag
- padding for the associated data and the message.

They have been designed to assure:

- Ease of use
- No influence on the maximum clock frequency of AEAD (up to 300 MHz in Virtex 7)
- Limited area overhead.

2.1.3 CMD FIFO

The Command (CMD) FIFO is a small 4x24 First-Word-Fall-Through (FWFT) FIFO that temporarily stores all significant bits of instructions and segment headers that need to be passed to the output. This module allows the Pre-Processor to operate with the maximum efficiency. This FIFO's width is selected based on the fact that the instructions defined in [9], Fig. 7, contain only 4 significant bits, and segment headers, defined in [9], Fig. 8, contain only 24 significant bits.

2.2 Lightweight Implementations

Fig. 2.5 shows the proposed non-pipelined lightweight architecture of a *single-pass* authenticated cipher compliant with the CAESAR Hardware API.

For lightweight implementations, the top-level unit is made of four lower-level units called the *PreProcessor*, *CipherCore*, *Header/Tag FIFO*, and *PostProcessor*.

2.2.1 PreProcessor

The *PreProcessor* is responsible for the following tasks

- parsing segment headers
- loading keys
- passing input blocks to the *CipherCore*, along with information required for padding
- keeping track of the number of data bytes left to process.

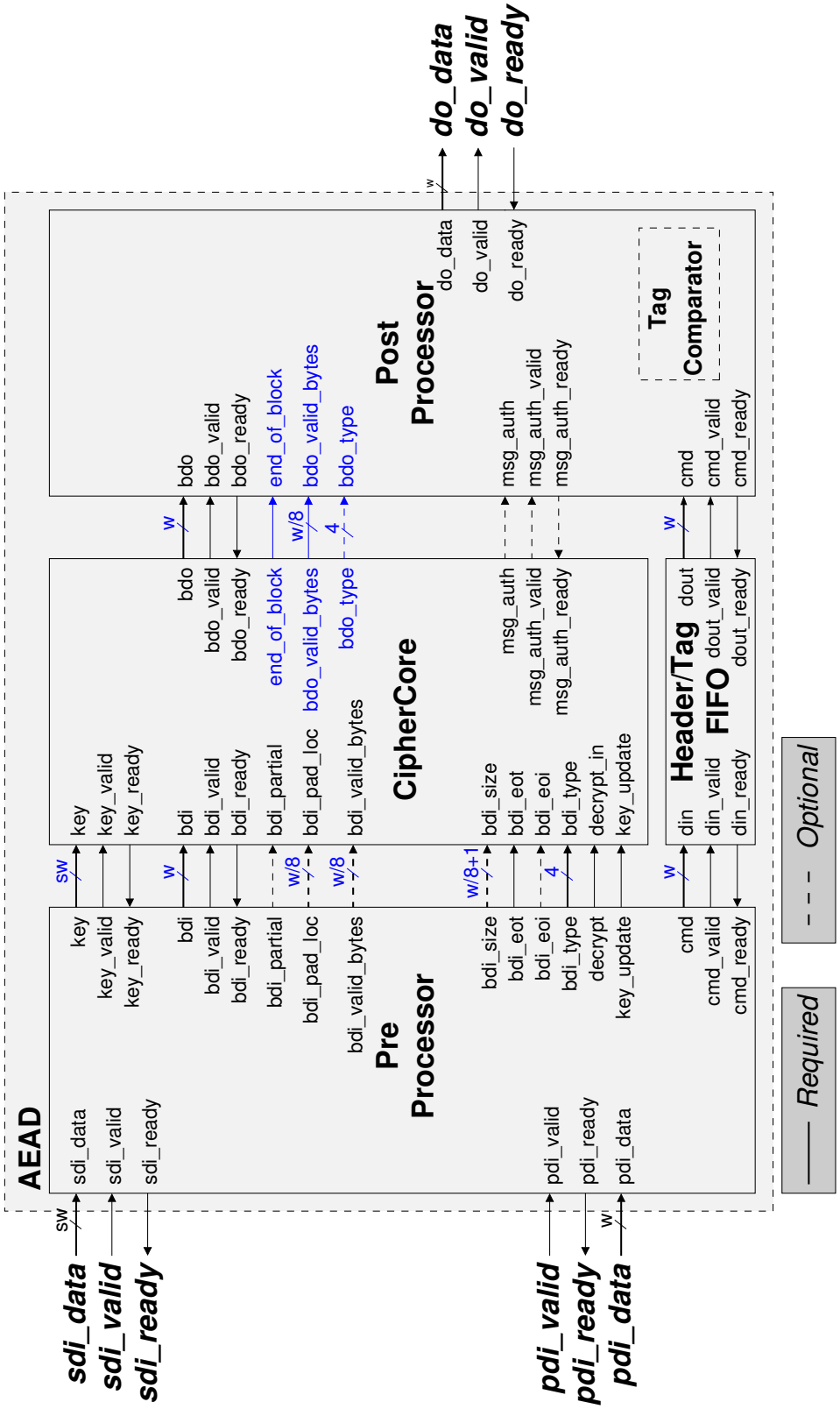


Figure 2.5: Top-level block diagram of a lightweight architecture of a single-pass authenticated cipher core, AEAD

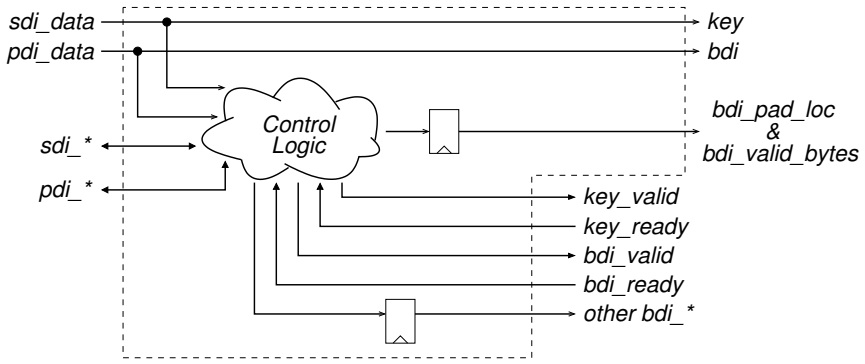


Figure 2.6: The *PreProcessor* Design. pdi_* and bdi_* stand for all the PreProcessor ports, shown in Figs. 2.5, with the names starting from the respective strings.

This unit only supports non-registered mode and does not include padding unit. It is assumed that padding is performed within the *CipherCore*. The *PreProcessor* provides all the necessary information required for padding to the *CipherCore*. The *bdi_type* signal specifies the type of data on the *bdi_data* bus. Table 6.2 lists the encoding for different data types. An overview of the *PreProcessor* design is shown in Fig. 2.6.

2.2.2 PostProcessor

The *PostProcessor* is responsible for the following tasks:

- clearing any portions of output words not belonging to the ciphertext or plaintext
- generating the header for the output data blocks
- Tag comparison[1]
- generating the status block with the result of authentication.

An overview of the *PostProcessor* design is shown in Fig. 2.7. The lightweight version only supports non-registered mode. The *PostProcessor* performs the tag comparison if G_TAG_INTERNAL=False. In this case, the tag is not provided as an input to the *CipherCore*. Instead it bypasses

[1]only when G_TAG_INTERNAL=*False*

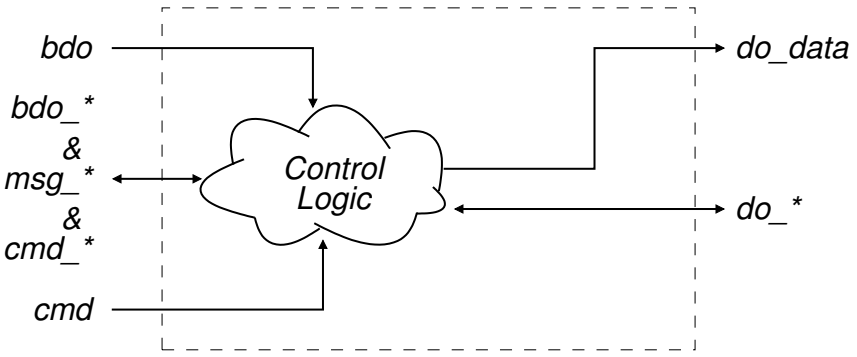


Figure 2.7: The *PostProcessor* Design. msg_*, cmd_*, bdo_*, and do_* stand for all the PostProcessor ports, shown in Figs. 2.5 with the names starting from the respective strings.

the *CipherCore* using the *Header/Tag FIFO* module. Consequently, the CipherCore should not have ports msg_auth_* and the ports msg_auth_* of the PostProcessor should be left unconnected.

2.2.3 Header/Tag FIFO

The *Header/Tag FIFO* is a small 4xW First-Word-Fall-Through (FWFT) FIFO that temporarily stores all segment headers that need to be passed to the output. As stated in Section 2.2.2, it also provides a bypass path for the tag when the tag comparator within the *PostProcessor* is used.

3 CipherCore Development and Benchmarking

The development and benchmarking of a **high-speed** implementation of a selected authenticated cipher can be performed using the following major steps, described in the subsequent chapters of this guide:

1. Configure the provided AEAD or AEAD_TP entity declaration for high-speed implementations (Chapter 4.1)
2. Develop the CipherCore (Chapter 5)
3. Generate test vectors (Chapter 7.1.3)
4. Verify the AEAD design (including the CipherCore design) using functional simulation (Chapter 7.2)
5. Generate optimized results for AEAD or AEAD_TP using FPGA tools (Chapter 8).

The development and benchmarking of a **lightweight** implementation of a selected authenticated cipher can be performed using the following major steps, described in the subsequent chapters of this guide:

1. Configure the provided AEAD entity declaration for lightweight implementations (Chapter 4.1)
2. Develop the CipherCore (Chapter 6)
3. Generate test vectors (Chapter 7.1.3)
4. Verify the AEAD design using functional simulation (Chapter 7.2)

5. Generate optimized results for AEAD using FPGA tools (Chapter 8).

As can be seen from the above description, only the first two steps are different. All remaining steps are universal and apply to both high-speed and lightweight implementations.

4 The AEAD Configuration

4.1 High-Speed Implementations

The entity declarations of AEAD and AEAD_TP for high-speed implementations are available as a part of the supporting Development Package in the files

`$ROOT/hardware/AEAD/src_rtl_hs/AEAD.vhd`
`$ROOT/hardware/AEAD/src_rtl_hs/AEAD_TP.vhd`

These entity declarations contain multiple generics defined in Table 4.1. Additional generics, used to determine the desired padding scheme are defined in Tables 4.2 and 4.3. The names of all generics, listed in the aforementioned tables, are supplemented in the VHDL code with the prefix G_.

The following restrictions must be considered when configuring the AEAD and AEAD_TP entities for high-speed implementations:

4.1.1 I/O Port Widths

Consistently with the specification of the CAESAR Hardware API [9], the allowed values of the port widths for high-speed implementations are as follows:

$$32 \leq w \leq 256,$$
$$32 \leq sw \leq 64.$$

These widths are set in the AEAD and AEAD_TP entity declarations using generics W and SW.

4.1.2 Block sizes

Values of the generics ABLK_SIZE and DBLK_SIZE, describing the sizes of input blocks for associated data and message/ciphertext, respectively,

must be multiples of the generic W. Similarly, the generic KEY_SIZE must be a multiple of the generic SW. Additionally, ABLK_SIZE is assumed to be smaller than or equal to DBLK_SIZE.

4.1.3 The Preprocessor and PostProcessor Maximum Input/Output Rates

The maximum rate at which the PreProcessor can provide a block of data and the PostProcessor can accept a block of data is dependent on the size of the message/ciphertext block (DBLK_SIZE) and the I/O port width (W). In the registered mode of operation, a new block of input data can be provided by the PreProcessor and accepted by the PostProcessor every $DBLK\_SIZE/W + 1$ clock cycles. In the non-registered mode, a new block of input data can be provided by the PreProcessor and accepted by the PostProcessor every clock cycle.

4.1.4 Limitations

The current implementation of the Pre- and PostProcessor do not support the following features:

- Ciphertext||Tag segment
- Intermediate tags
- multiple segments of the same type separated by segments of another type, e.g. header and trailer, treated as two segments of the type AD, separated by message segments.
- data blocks are never split across two segments as shown in Figures 4.1, and 4.2.

4.2 Lightweight Implementations

The entity declaration of AEAD for lightweight implementations is available as a part of the Development Package in the file

`$ROOT/hardware/AEAD/src_rtl_lw/AEAD.vhd`

This entity declaration contains multiple generics defined in Table 4.4. The names of all generics, listed in the aforementioned table, are supplemented in the VHDL code with the prefix G_. All the generic used for lightweight implementation can be configured in the package file

`$ROOT/hardware/AEAD/src_rtl_lw/design_pkg.vhd`

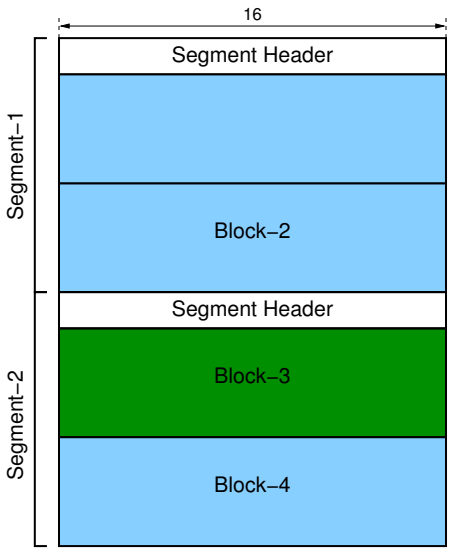


Figure 4.1: Correct way of splitting blocks

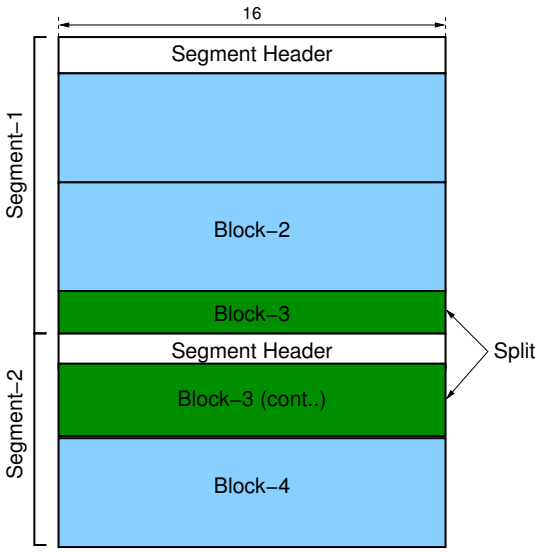


Figure 4.2: Incorrect way of splitting blocks

4.2.1 I/O Port Widths

The generics W and SW are used to determine the I/O port widths, w and sw, respectively. Consistently with the specification of the CAESAR Hardware API [9], the allowed values of these port widths are as follows:

$$w = 8, 16, 32,$$
$$sw = 8, 16, 32.$$

In order to limit the use of additional logic, widths of both ports are assumed to be equal, and thus both generics, W and SW, have to be set to the same value.

4.2.2 Block Sizes

Values of generics ABLK_SIZE, DBLK_SIZE, KEY_SIZE, and TAG_SIZE describe the sizes of associated data blocks, message/ciphertext blocks, key, and tag respectively.

4.2.3 Tag Comparison

The tag comparison can be performed either within the CipherCore or in the PostProcessor. If the generic TAG_INTERNAL is set to *True*, the comparison is performed within the CipherCore. Hence, the expected tag is passed to the CipherCore. If the tag comparison needs to be performed in the PostProcessor, the generic must be set to *False*. In this case, the tag words have to be passed to PostProcessor by a FIFO, bypassing the CipherCore.

4.2.4 Limitations

On top of the limitations described in Section 4.1.4, which are the same for High-Speed and Lightweight implementations, the PreProcessor does not apply any padding to the data blocks.

Table 4.1: The AEAD and AEAD_TP Generics for High-Speed

Generic	Type	Default Value	Definition
I/O Widths in Bits			
W	Integer	32	Public data input and data output width
SW	Integer	32	Secret data input width
FW	Integer	32	Two-pass FIFO data input and data output width
Reset Behavior			
ASYNC_RSTN	Boolean	False	Reset behavior. True=Asynchronous active low, False= Synchronous active high.
Special Features			
ENABLE_PAD	Boolean	False	Enable padding (See additional settings in Tables 4.2 and 4.3)
CIPH_EXP	Boolean	False	Ciphertext expansion mode. This option should be used when the ciphertext size is not the same as the plaintext size, i.e., the ciphertext is expanded. It should also be used when Ciphertext=Ciphertext||Tag.
REVERSE_CIPH	Boolean	False	Reverse ciphertext mode. Used, for example, by PRIMATEs-APE. Currently not supported.
MERGE_TAG	Boolean	False	No tag segment. This parameter should be set to True when the CipherCore does not separate Tag from Ciphertext, i.e., Ciphertext=Ciphertext||Tag. Currently not supported.
Block Size Parameters in Bits			
ABLK_SIZE	integer	128	Size of Associated Data block to be transferred through the bdi bus. This value should be smaller than or equal to DBLK_SIZE.
DBLK_SIZE	integer	128	The bdi and bdo data bus size (may be a divisor of the Data block size).
KEY_SIZE	integer	128	The key bus size (may be a divisor of the Key size).
TAG_SIZE	integer	128	Tag size. Note: This value is not used when MERGE_TAG is True.
Padding Parameters			
PAD_STYLE	integer	0	Padding style. See Table 4.2.
PAD_AD	integer	1	Padding behavior for associated data. See Table 4.3.
PAD_D	integer	1	Padding behavior for message. See Table 4.3.
Maximum supported AD/message/ciphertext length			
MAX_LEN	integer	See the Definition	Maximum supported AD/message/ciphertext length = $2^{MAX\_LEN} - 1$, where two default values of MAX_LEN are SINGLE_PASS_MAX=32 for AEAD, and TWO_PASS_MAX=11 for AEAD_TP.

Table 4.2: Supported padding rules, determined by the allowed values of the generic PAD_STYLE.

Value	Description
0	No padding
1	10* padding rule
2	ICEPOLE padding rule
3	0000_0001_0* padding rule

Table 4.3: Supported values of the generics PAD_AD and PAD_D, and their respective meaning. A = Pad enable. B = Extra block is added when AD/D is empty. C = Extra block is added when AD/D is a non-zero multiple of a block size.

Value	Feature		
	A	**B**	**C**
0			
1	x		
2	x	x	
3	x		x
4	x	x	x

Table 4.4: AEAD Generics for Lightweight

Generic	Type	Default Value	Definition
I/O Widths in Bits			
W	Integer	32	Public data input and data output width. Only 8, 16, and 32 are the allowed widths.
SW	Integer	32	Secret data input width. Only 8, 16, and 32 are the allowed widths.
Block Size Parameters in Bits			
ABLK_SIZE	integer	128	Associated data block size.
DBLK_SIZE	integer	128	Message/ciphertext block size.
KEY_SIZE	integer	128	Key size.
TAG_SIZE	integer	128	Tag size.
Padding Parameters			
TAG_INTERNAL	Boolean	False	False = Verification done by the *PostProcessor*, True = Verification done by the *CipherCore*

5 CipherCore Development for High-Speed Implementations

5.1 Interface

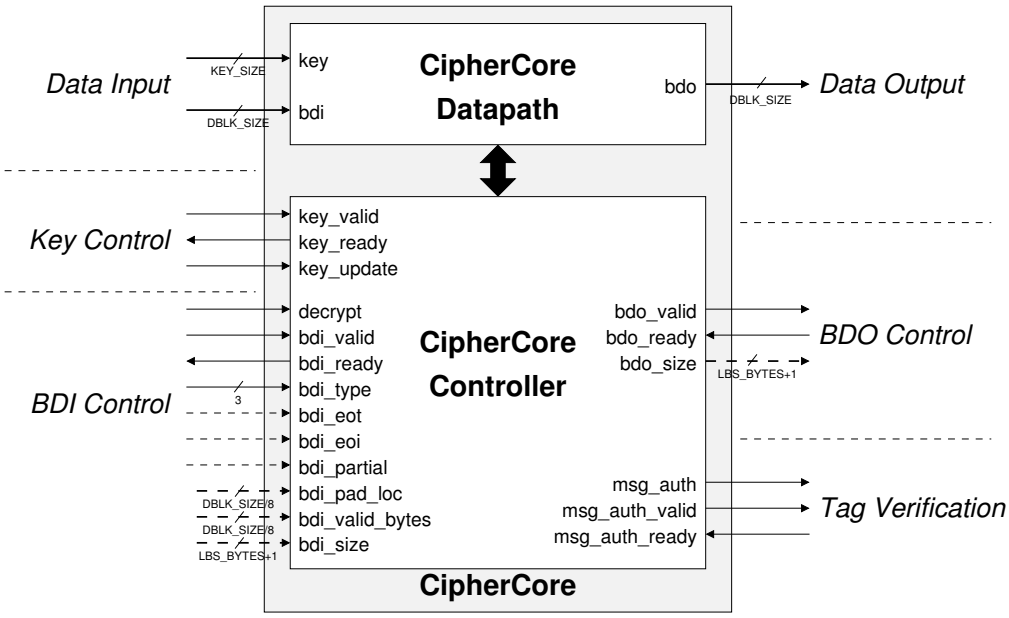


Figure 5.1: Interface of the CipherCore for High-Speed Implementations

The interface of the CipherCore for High-Speed Implementations is shown in Figure 5.1. Ports marked using dashed arrows are optional and used only if required. This approach allows the synthesis tool to trim the unused ports and the associated logic from the design, resulting in a better resource utilization.

Data input ports are limited to *key* and *bdi* (block data input). The key port is controlled using the handshake signals `key_valid` and `key_ready`. *key_update* is used to notify the CipherCore that it should update the internal key prior to processing the next message.

The *decrypt* signal informs the core whether the current operation is encryption or decryption. The *bdi* port is controlled using the *bdi_valid* and *bdi_ready* handshake signals. The *bdi_type* input indicates the type of input data, with the encoding shown in Table 5.1.

Table 5.1: *bdi_type* Encoding. – represents don't care.

Encoding	**Type**
00–	Associated Data/Associated Data||Npub/ Npub||Associated Data
01–	Message/Ciphertext/Ciphertext||Tag
100	Tag
101	Length
110	Public message number
111	Secret message number

The signal `bdi_eot` indicates that the current BDI block is the last block of its type. This signal is used only when the type is either AD, Message, or Ciphertext. The signal `bdi_eoi` indicates that the current BDI block is the last block of input other than a block of the Length segment, a block of the Tag segment, or a block of padding. The input `bdi_partial` indicates the case of partial block.

The correct values of *bdi_valid_bytes*, *bdi_pad_loc* and *bdi_size* for various numbers of valid bytes within a 4-byte data block are shown in Table 5.2, where:

- Case A: Either not the last block or the last block with all 4 bytes valid.
- Case B: The last block with 3 bytes valid.
- Case C: The last block with 1 byte valid.
- Case D: The last block with no valid bytes. Assuming the 10* padding, this block consists of a single 1 followed by 31 zeros.

Table 5.2: Values of the special control signals *bdi_valid_bytes*, *bdi_pad_loc*, and *bdi_size* for the *bdi* bus with the width $DBLK\_SIZE = 32$. *Byte Validity* represents the byte locations in *bdi* that were the part of input (e.g., AD or message) before padding.

Byte/Bit Position	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
	Case A				**Case B**			
Byte Validity								
`bdi_valid_bytes`	1	1	1	1	1	1	1	0
`bdi_pad_loc`	0	0	0	0	0	0	0	1
`bdi_size`		1	0	0		0	1	1
	Case C				**Case D**			
Byte Validity								
`bdi_valid_bytes`	1	0	0	0	0	0	0	0
`bdi_pad_loc`	0	1	0	0	1	0	0	0
`bdi_size`		0	0	1		0	0	0

It must be noted that all ports of the BDI Control group and *bdi* are synchronized with the *bdi_valid* input. Their values should be read only when the *bdi_valid* signal is high. The same scenario also applies to the BDO Control group and *bdo*, which are synchronized with the value of the *bdo_valid* output.

The *bdo* port is controlled using the *bdo_valid* and *bdo_ready* handshake signals. *bdo_size* is not used unless the $CIPH\_EXP$ generic of AEAD is set to True. When this is the case, each active value of *bdo_valid* must be accompanied by providing the size of an output block, in bytes, using the *bdo_size* port.

The Tag Verification ports (*msg_auth∗*) are only used during the authenticated decryption operation, when the core must provide output signals indicating whether the authentication is done and the result is (or is not) valid. Similar to *bdo* port, *msg_auth* is synchronized via AXI compatible handshake signals.

The description of all *CipherCore* ports are provided in Table 5.3. Ports related to the `bdi` control are categorized according to the following criteria:

COMM A handshake signal.

INPUT INFO An auxiliary signal that remains valid until a given input is fully processed. Deactivation is typically done at the end of input.

SEGMENT INFO An auxiliary signal that remains valid for the current segment. Its value changes when a new segment is received via the PDI data bus.

BLOCK INFO An auxiliary signal that is valid for the current input block. Its value changes when a new block is read.

For the CipherCore that supports Two-Pass algorithms, additional ports have been added to accommodate the communication with the external FIFO, as shown in Figure 5.2.

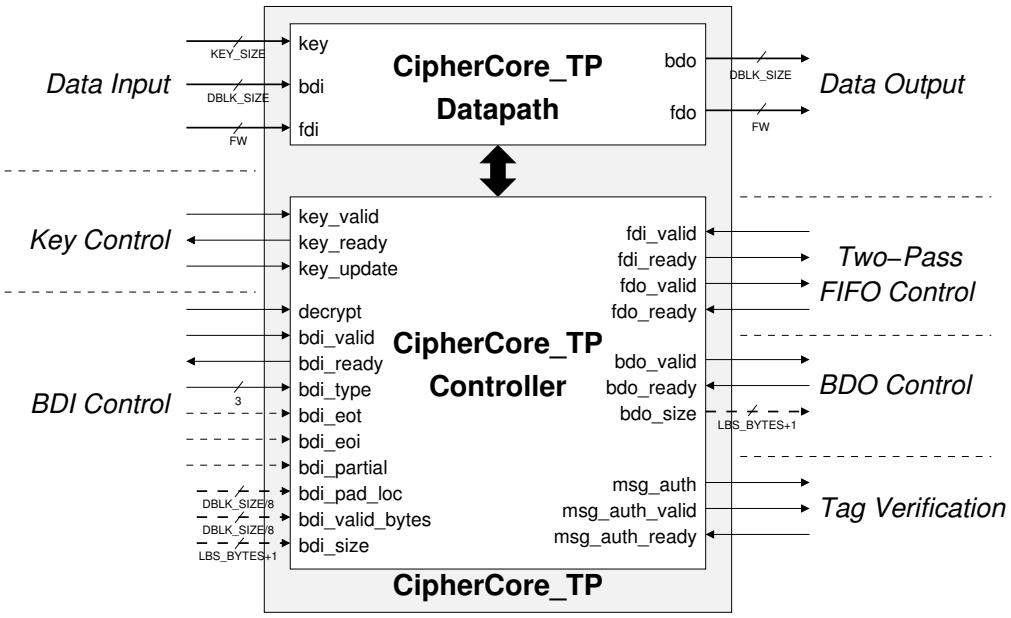


Figure 5.2: Interface of Two-Pass CipherCore

The additional port descriptions required for a CipherCore that supports Two-Pass algorithms are provided in Table 5.4. It must be noted that all the ports listed in Table 5.3 are also present in the interface of the Two-Pass core.

Table 5.3: CipherCore Port Descriptions for High-Speed Implementations. LBS_BYTES = $\log_2(DBLK\_SIZE/8)$

Name	Direction	Size	Description
Data Input & Output			
key	in	KEY_SIZE	Key data
bdi	in	DBLK_SIZE	Block data input
bdo	out	DBLK_SIZE	Block data output
Key Control			
key_valid	in	1	Key data is valid
key_ready	out	1	CipherCore is ready to receive a new key
key_update	in	1	Key must be updated prior to processing a new input
BDI Control			
decrypt	in	1	[INPUT INFO] 0=Encryption, 1=Decryption
bdi_valid	in	1	[COMM] BDI data is valid
bdi_ready	out	1	[COMM] CipherCore is ready to receive data
bdi_type	in	3	[BLOCK INFO] Type of BDI data. See Table 5.1.
bdi_eot	in	1	[BLOCK INFO] The current BDI block is the last block of its type. Note: Only applies when the type is either AD, Message, or Ciphertext.
bdi_eoi	in	1	[BLOCK INFO] The current BDI block is the last block of input other than a block of the Length segment, a block of the Tag segment, or a block of padding.
bdi_partial	in	1	[SEGMENT INFO] The current block is either a partial block of AD or Message, or the result of encryption of a partial message block. Note: This optional signal is used only in the implementations of the ciphertext expansion algorithms. We are aware of its necessity only for the implementation of the Round 2 AES-COPA.
bdi_pad_loc	in	DBLK_SIZE/8	[BLOCK INFO] Encoding of the byte location where padding begins. See Table **??**.
bdi_valid_bytes	in	DBLK_SIZE/8	[BLOCK INFO] Encoding of the byte locations that are valid. See Table **??**.
bdi_size	in	LBS_BYTES+1	[BLOCK INFO] Number of valid bytes in bdi.
BDO Control			
bdo_valid	out	1	BDO data is valid
bdo_ready	in	1	PostProcessor is ready to receive data.
bdo_size	out	LBS_BYTES+1	Number of valid bytes in bdo. This port **must** be used when $CIPH\_EXP$ is active.
Tag Verification			
msg_auth	out	1	1=Authentication success, 0=Authentication failure
msg_auth_valid	out	1	Authentication output is valid
msg_auth_ready	in	1	PostProcessor is ready to accept authentication result

Table 5.4: Additional Port Descriptions for a Two-Pass CipherCore.

Name	Direction	Size	Description
Data Input & Output			
fdi_data	in	FW	Input data from the two-pass FIFO
fdo_data	out	FW	Output data to the two-pass FIFO
Control			
fdi_valid	in	1	fdi data is valid
fdi_ready	out	1	CipherCore is ready to receive a new two-pass data
fdo_valid	out	1	CipherCore is ready to send a new two-pass data
fdo_ready	in	1	two-pass FIFO is ready to receive a new data

5.2 Handshakes

This section presents examples of handshakes. All ports in the figures of this section are represented by a blue and red color, for input and output ports, respectively.

Fig. 5.3 provides an example of a handshake used for loading a block of data using the (*bdi*) port. Data and its auxiliary signals are synchronized with the *bdi_valid* signal. Similarly for *key*, data is synchronized with the *key_valid* signal, as shown in Figure 5.4.

Fig. 5.5 provides an example of a handshake used to write output to the PostProcessor. Fig. 5.5a presents an example for the standard mode of operation of an authenticated cipher. Fig. 5.5b presents an example for the case of an algorithm operating in the ciphertext expansion mode. An additional output port (*bdo_size*) is now required to inform the PostProcessor about the size of the current message block after decryption. This information is used by the PostProcessor to update the header with correct value of the last segment size.

Finally, an example of signal timing for message authentication is shown in Fig. 5.6. For every decryption operation, CipherCore must provide *msg_auth* signal to indicate authentication result to PostProcessor via its corresponding handshake signals (*msg_auth_valid* and *msg_auth_ready*).

5.3 Design Procedure

It is recommended that you start the development of the CipherCore, specific to a given authenticated cipher, by using the code provided in the

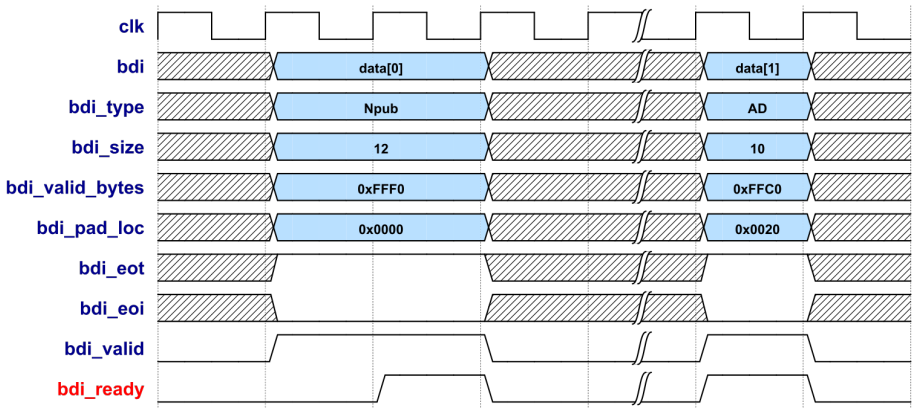


Figure 5.3: An example of a handshake used for loading data using the input *bdi*

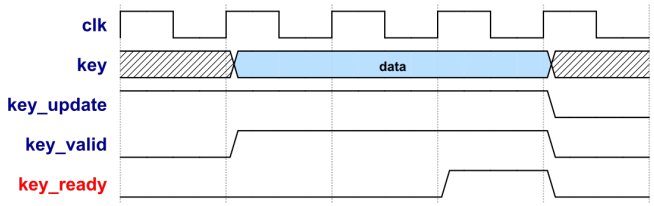
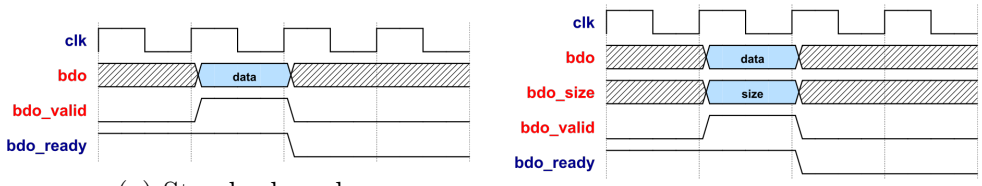


Figure 5.4: An example of a handshake used for loading a key



(a) Standard mode

(b) Ciphertext expansion mode

Figure 5.5: An example of a handshake used for writing data in the a) Standard mode, b) Ciphertext expansion mode

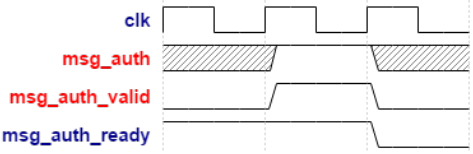


Figure 5.6: An example of a handshake used to perform message authentication

Development Package, in the folder

`$ROOT/hardware/AEAD/src_rtl_hs`

In particular, the appropriate connections among the CipherCore, the PreProcessor, the PostProcessor, and the CMD FIFO modules are already specified in this code. A designer needs to modify generics in the AEAD module, and then develop the CipherCore Datapath and the CipherCore Controller.

The development of the CipherCore is left to individual designers and can be performed using their own preferred design methodology. Typically, when using a traditional RTL (Register Transfer Level) methodology, the CipherCore Datapath is first modeled using a block diagram, and then translated to a hardware description language (VHDL or Verilog HDL). The CipherCore Controller is then described using an algorithmic state machine (ASM) chart or a state diagram, further translated to HDL.

An ASM chart of the CipherCore Controller typically contains the following states:

1. Idle
2. Activate Key
3. Load Npub
4. Load Data
5. Process AD
6. Process AD Last
7. Process Data
8. Process Data Last
9. Generate/verify Tag (GenVer Tag).

An example ASM chart for the CipherCore Controller is shown Fig. 5.7. After a new instruction or after reset, the control should wait for the first block of data in the *Idle* state. The CipherCore should monitor the `bdi_valid` for the first block of data, which is typically Npub. When this signal is active, the circuit should check whether the current key requires an update by inspecting the `key_update` signal. If it does, the controller changes its state to *Activate Key*. In this state either a new key is stored internally within the CipherCore or the corresponding round keys are precomputed. Once this task is completed, `key_ready` should be activated to acknowledge the key activation.

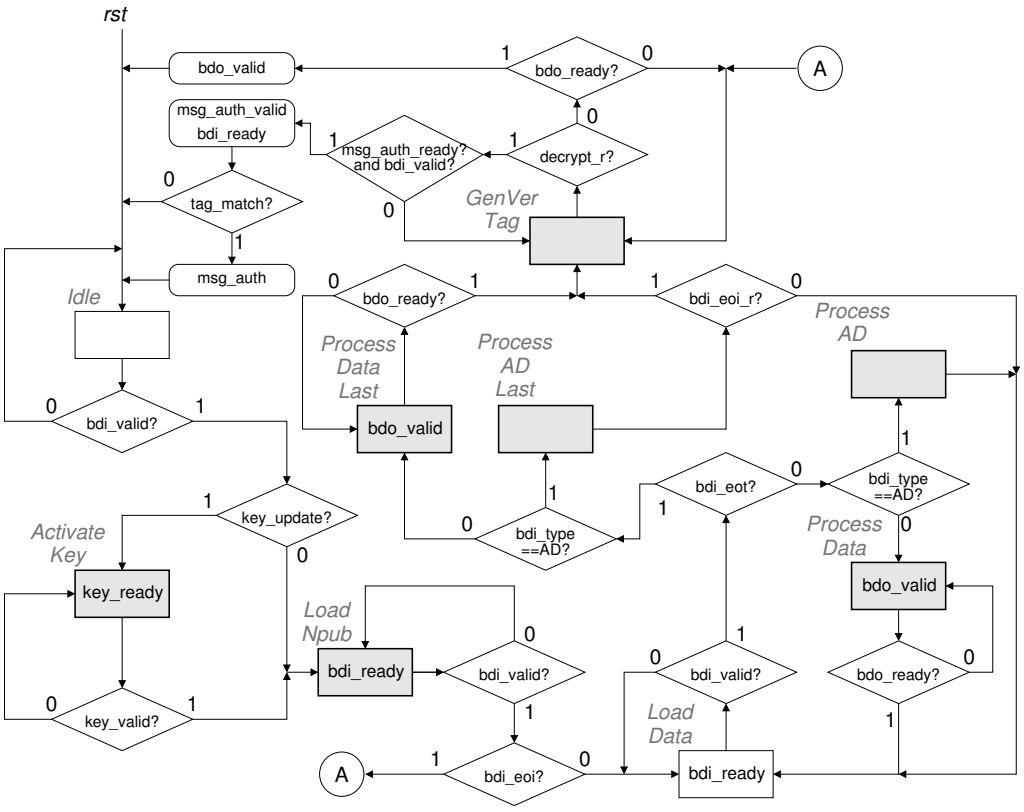


Figure 5.7: A typical Algorithmic State Machine (ASM) chart of the CipherCore Controller. Each shaded state in this diagram may need to be replaced by a sequence of states in the actual implementation of a complex authenticated cipher. $*\_r$ are status registers storing values of the respective inputs read during the last bdi handshake.

Once a new key is activated or no new key is required (`key_update`=0), the circuit is ready to process the first block of data (Npub) in the *Load Npub* state. At the same time, that the Npub block is loaded into the CipherCore, the circuit needs to acknowledge its receipt by setting the `bdi_ready` output to high. The controller then moves to the next processing state, *Load Data*. In the case that Npub is the last block of data (AD size = Message/Ciphertext size = 0), which can be determined using the `bdi_eoi` input, the controller state can change directly to *Generate/verify tag*.

In the *Load Data* state, the circuit waits until the next input block is valid (`bdi_valid`=1), and then processes data based on the incoming input type (`bdi_type`). Depending on the algorithm, additional processing may be required for the last block of data. This block can be determined using the end-of-type input (`bdi_eot`). At the same time, the end-of-input signal (`bdi_eoi`) may be stored in a register within the CipherCore to keep track of the last input state. This status register is useful to determine when no additional data block is expected after processing of the last AD block, so that the controller can progress to the last state (*Generate/verify tag*) directly.

In the last state, *Generate/verify tag*, during the authenticated encryption operation, the core should generate a new tag and pass it to the PostProcessor via the `bdo` bus. During the authenticated decryption operation, `msg_auth` should be provided. At the same time, the `msg_auth_valid` signal should remain active until the PostProcessor is ready to receive `msg_auth`, which is indicated by an active value of the `msg_auth_ready` signal.

5.4 Dummy Authenticated Ciphers

Five example designs of the CipherCore and AEAD, corresponding to five Dummy Authenticated Ciphers, are provided as a part of our distribution. The first three Dummy Authenticated Ciphers are specified using the following equations:

$$AD = AD_1, AD_2, ..., AD_{n-1}, AD_n \tag{5.1}$$

$$PT = PT_1, PT_2, ..., PT_{m-1}, PT_m \tag{5.2}$$

$$CT = CT_1, CT_2, ..., CT_{m-1}, CT_m \tag{5.3}$$

$$CT_i = PT_i \oplus i \oplus Key \oplus Npub \tag{5.4}$$
$$\text{for } i = 1..m-1.$$

$$CT_m = Trunc(PT_m \oplus i \oplus Key \oplus Npub, PT_m) \tag{5.5}$$
$$\text{when } \textbf{CIPH\_EXP}=\text{False}.$$

$$CT_m = Pad(PT_m) \oplus m \oplus Key \oplus Npub \tag{5.6}$$
$$\text{when } \textbf{CIPH\_EXP}=\text{True}.$$

$$Tag = Key \oplus Npub \oplus Len \oplus \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \tag{5.7}$$

where,

- PT_i and CT_i are the plaintext (message) and ciphertext blocks, respectively,
- AD_i = associated data block,
- $Pad(\cdot)$ represents a padding operation applied to the last AD and/or the last plaintext block,
- $Trunc(X, Y)$ truncates X to the size of Y,
- i = 128-bit block number,
- Key = 128-bit key,
- $Npub$ = Public message number,
- Len = 64-bit associated data length (in bits) || 64-bit plaintext length (in bits).

For an XOR operation with inputs of different sizes, the smaller operands are appended with zeros to have the same length as the longest operand. The result has the length of the longest operand. All examples are based on a 128-bit data block, unless specified otherwise. The differences between

each Dummy Authenticated Cipher are primarily based on the definition of padding and values of parameters described below. Please note that a typical padding behavior is either appending all zeros (0^*) or one followed by zeros (10^*).

The design of the controllers used in our dummy cores is based on the ASM chart discussed in the previous section. The features of all five dummy cores are summarized in Table 5.5.

Table 5.5: Summary of features/parameters of five dummy high-speed authenticated ciphers

	CIPH EXP?	**Npub Size**	**AD**		**PT**		**Tag Size**	**Off-line?**	**Pre-Processor**	
			Block size	**Pad?**	**Block size**	**Pad?**			**Data buffer?**	**Key buffer?**
dummy1	False	96	128	True	128	True	128	False	True	True
dummy2	False	128	96	False	128	True	128	True	True	True
dummy3	True	128	128	True	128	True	128	False	True	True
dummy4	False	128	32	True	32	True	64	False	False	True
dummy5	False	128	32	True	32	True	128	False	False	False

5.4.1 dummy1

This example is aimed at presenting the behavior of the Pre- and Post-processors for a typical CipherCore. The following parameters are used:

- $AD_{block\_size} = PT_{block\_size} = 128$ bits
- $Npub_{size} = 96$ bits
- $Pad(AD_n) = AD_n$ **if** $len(AD_n) = block\_size$ **else** $AD_n||10^*$
- $Pad(PT_m) = PT_m$ **if** $len(PT_m) = block\_size$ **else** $PT_m||10^*$
- **CIPH_EXP**=False

5.4.2 dummy2

This example aims at presenting the behavior of the PreProcessor when $AD_{block\_size} \neq PT_{block\_size}$, and zero padding is applied to AD. The following parameters are used:

- $AD_{block\_size} = 96$ bits

- $PT_{block\_size} = Npub_{size} = 128$ bits
- $Pad(AD_n) = AD_n$ **if** $len(AD_n) = block\_size$ **else** $AD_n||0^*$
- $Pad(PT_m) = PT_m$ **if** $len(PT_m) = block\_size$ **else** $PT_m||10^*$
- **CIPH_EXP**=False

5.4.3 dummy3

This example aims at presenting an example implementation for algorithms that have ciphertext expansion. The following parameters are used:

- $AD_{block\_size} = PT_{block\_size} = Npub_{size} = 128$ bits
- $Pad(AD_n) = AD_n$ **if** $len(AD_n) = block\_size$ **else** $AD_n||10^*$
- $Pad(PT_m) = PT_m$ **if** $len(PT_m) = block\_size$ **else** $PT_m||10^*$
- **CIPH_EXP**=True

Additionally, the *Len* segment is removed from the tag generation for this dummy core, so the new equation for Tag is

$$Tag = Key \oplus Npub \oplus \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \quad (5.8)$$

5.4.4 dummy4

This example aims at presenting the behavior of the Pre- and Post-processor for the following cases:

- External public bus size is equal to the internal data bus size, i.e., W = DBLK_SIZE. This allows the PreProcessor to operate in the non-registered mode for the `bdi` input.
- Tag size is larger than the data bus size, i.e., TAG_SIZE > DBLK_SIZE.
- Npub size is larger than the data bus size.

For this example, the same padding rules as those used in *dummy1* are applied, together with the following values of parameters:

- $AD_{block\_size} = PT_{block\_size} = 32$ bits
- $Npub_{size} = 128$ bits
- $Key = 128$ bits
- $Tag = 64$ bits.

Additionally, the ciphertext and the tag are described as followed:

$$CT_i = PT_i \oplus i \oplus KN \quad \text{for } i = 1..m-1. \tag{5.9}$$

$$CT_m = Trunc(PT_m \oplus m \oplus KN, PT_m) \tag{5.10}$$

$$Tag_{63..32} = KN \oplus \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \tag{5.11}$$

$$Tag_{31..0} = \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \tag{5.12}$$

where,

$$KN = Key_{127..96} \oplus Key_{95..64} \oplus Key_{63..32} \oplus Key_{31..0} \oplus Npub_{127..96} \oplus Npub_{95..64} \oplus Npub_{63..32} \oplus Npub_{31..0}$$

5.4.5 dummy5

This example uses the same algorithm as *dummy4* except that the hardware implementation relies on the different PreProcessor settings. In particular, the `key` bus size (KEY_SIZE) is set to the same width as `sdi` bus size (SW). As a result, the PreProcessor operates in a non-registered mode for the `key` as well as the `bdi` input. This mode reduces the AEAD overall resource utilization as the key is not buffered inside the PreProcessor.

5.5 AES and Keccak Permutation F

Additional support is provided for designers of cipher cores of CAESAR candidates based on AES and Keccak. Fully verified VHDL code, block diagrams, and ASM charts of AES and Keccak Permutation F have been developed and made available at [8]. Our AES core implements a basic iterative architecture of a block cipher, with the SubBytes operation realized using memory. Either distributed memory (implemented using multipurpose LUTs) or block memory is inferred depending on the specific options of FPGA tools.

6 CipherCore Development for Lightweight Implementation

6.1 Interface

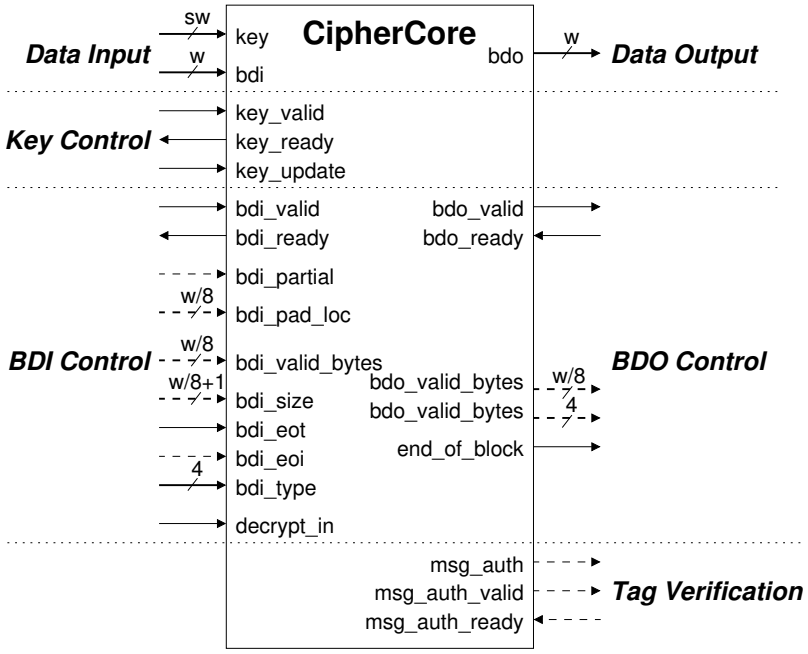


Figure 6.1: Interface of CipherCore for Lightweight Implementations.

The interface of CipherCore for Lightweight Implementations is shown in Figure 6.1. Ports marked using dashed arrows are optional and used only if required. This approach allows the synthesis tool to trim the unused

ports and the associated logic from the design, resulting in a better resource utilization.

Data input ports are limited to `key` and `bdi` (block data input). The key port is controlled using the handshake signals `key_valid` and `key_ready`. *key_update* is used to notify the CipherCore that it should update the internal key prior to processing the next message.

The *bdi* port is controlled using the *bdi_valid* and *bdi_ready* handshake signals. The input `bdi_partial` indicates the case of partial block.

The correct values of *bdi_valid_bytes*, *bdi_pad_loc* and *bdi_size* for various numbers of valid bytes within a 4-byte data block are shown in Table 6.1, where:

- Case A: Either not the last block or the last block with all 4 bytes valid.
- Case B: The last block with 3 bytes valid.
- Case C: The last block with 1 byte valid.
- Case D: The last block with no valid bytes. Assuming the 10* padding, this block consists of a single 1 followed by 31 zeros.

Table 6.1: Values of the special control signals `bdi_valid_bytes`, `bdi_pad_loc`, and `bdi_size` for the `bdi` bus with the width $DBLK\_SIZE = 32$. *Byte Validity* represents the byte locations in `bdi` that were the part of input (e.g., AD or message) before padding.

Byte/Bit Position	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
	Case A				**Case B**			
Byte Validity								
`bdi_valid_bytes`	1	1	1	1	1	1	1	0
`bdi_pad_loc`	0	0	0	0	0	0	0	1
`bdi_size`		1	0	0		0	1	1
	Case C				**Case D**			
Byte Validity								
`bdi_valid_bytes`	1	0	0	0	0	0	0	0
`bdi_pad_loc`	0	1	0	0	1	0	0	0
`bdi_size`		0	0	1		0	0	0

The signal `bdi_eot` indicates that the current BDI block is the last block of its type. This signal is used only when the type is either AD, Message, or Ciphertext. The signal `bdi_eoi` indicates that the current BDI block is the last block of input other than a block of the Length segment, a block of the Tag segment, or a block of padding.

The input and output data types are indicated by `bdi_type` and `bdo_type` using the encoding shown in Table. 6.2.

The input *decrypt_in* signal informs the core whether the current operation is encryption or decryption. The output *decrypt_out* passes this information to the PostProcessor.

It must be noted that all ports of the BDI Control group and *bdi* are synchronized with the *bdi_valid* input. Their values should be read only when the *bdi_valid* signal is high. The same scenario also applies to the BDO Control group and *bdo*, which are synchronized with the value of the *bdo_valid* output.

The *bdo* port is controlled using the *bdo_valid* and *bdo_ready* handshake signals. `bdo_valid_bytes` is the encoding of the byte locations in `bdo` that are valid, using the same convention as that concerning `bdi_valid_bytes`, illustrated in Table 6.1. The `end_of_block` signal indicates the last word of an output block.

The Tag Verification ports (*msg_auth_*$*$) are only used when the generic $TAG\_INTERNAL$ is set to True, meaning that the verification is done by CipherCore. If the generic is set to True, the ports are used only during an authenticated decryption operation. The CipherCore must provide `msg_auth` to indicate its result and set `msg_auth_valid` to high until the PostProcessor is ready (`msg_auth_ready` is active).

The description of all *CipherCore* ports are provided in Table 6.3. Ports related to the `bdi` control are categorized according to the following criteria:

COMM A handshake signal.

INPUT INFO An auxiliary signal that remains valid until a given input is fully processed. Deactivation is typically done at the end of input.

SEGMENT INFO An auxiliary signal that remains valid for the current segment. Its value changes when a new segment is received via the PDI data bus.

BLOCK INFO An auxiliary signal that is valid for the current input block. Its value changes when a new block is read.

Table 6.2: *bdi_type* and *bdo_type* Encoding

Encoding	Generic	Type
0001	HDR_AD	Associated Data
0100	HDR_MSG	Message
0101	HDR_CT	Ciphertext
1000	HDR_TAG	Tag
1100	HDR_KEY	Key
1101	HDR_NPUB	Npub
1110	HDR_NSEC	Nsec
1111	HDR_ENSEC	Enc Nsec

The description of all ports of the *Tag/Header FIFO* are provided in Table 6.4.

6.2 Handshakes

This section presents examples of handshakes. All ports in the figures of this section are represented by a blue and red color, for input and output ports, respectively.

The data on the buses is controlled using the handshake signals. The `*_valid` signals are set to high if the data on the corresponding bus is valid. If the module is ready to receive the data, the corresponding `*_ready` signals are set to high. These two handshaking signals operate independently.

Fig. 6.2 shows an example of loading a 128-bit key, for $sw = 32$. The `key_update` signal indicates the update of the key by asserting high for one clock cycle. The transfer of the key follows. An example of loading a 128-bit Npub is shown in Fig. 6.3.

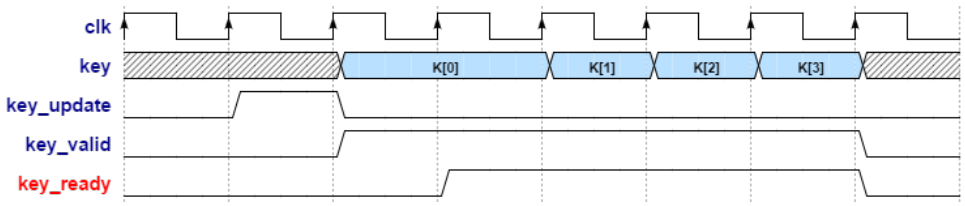


Figure 6.2: Handshake example of loading a key, for sw=32

Table 6.3: CipherCore Port Descriptions.

Name	Direction	Size	Description
Data Input & Output			
key	in	sw	Key data
bdi_data	in	w	Block data input
bdo_data	out	w	Block data output
Key Control			
key_valid	in	1	Key data is valid
key_ready	out	1	CipherCore is ready to receive a new key
key_update	in	1	Key must be updated prior to processing a new input
BDI Control			
bdi_valid	in	1	[COMM] BDI data is valid
bdi_ready	out	1	[COMM] CipherCore is ready to receive data
bdi_partial	in	1	[SEGMENT INFO] The current block is either a partial block of AD or Message, or the result of encryption of a partial message block.
bdi_pad_loc	in	w/8	[BLOCK INFO] Encoding of the byte location where padding begins.
bdi_valid_bytes	in	w/8	[BLOCK INFO] Encoding of the byte locations that are valid.
bdi_size	in	w/8+1	[BLOCK INFO] Number of valid bytes in bdi.
bdi_eot	in	1	[BLOCK INFO] The current BDI block is the last block of its type. Note: Only applies when the type is either AD, Message, or Ciphertext.
bdi_eoi	in	1	[BLOCK INFO] The current BDI block is the last block of input other than a block of the Length segment, a block of the Tag segment, or a block of padding.
bdi_type	in	4	[BLOCK INFO] Type of BDI data. See Table 6.2.
decrypt_in	in	1	[INPUT INFO] 0=Encryption, 1=Decryption
BDO Control			
bdo_valid	out	1	BDO data is valid
bdo_ready	in	1	PostProcessor is ready to receive data.
bdo_valid_bytes	in	w/8	[BLOCK INFO] Encoding of the byte locations that are valid.
end_of_block	out	1	[BLOCK INFO] The current BDO block is the last block of its type.
bdo_type	out	4	[BLOCK INFO] Type of BDO data. See Table 6.2.
decrypt_out	out	1	[INPUT INFO] 0=Encryption, 1=Decryption
TAG Verification			
msg_auth	out	1	1=Authentication success, 0=Authentication failure
msg_auth_valid	out	1	Authentication output is valid
msg_auth_ready	in	1	PostProcessor is ready to accept authentication result

Table 6.4: Header/Tag FIFO Port Descriptions.

Name	Direction	Size	Description
PreProcesor & FIFO			
`din`	in	w	Header info or Tag input
`din_valid`	in	1	data is valid
`din_ready`	out	1	FIFO ready to receive data
PostProcesor & FIFO			
`dout`	out	w	Header info or Tag out
`dout_valid`	out	1	data is valid
`dout_ready`	in	1	PostProcessor ready to receive data

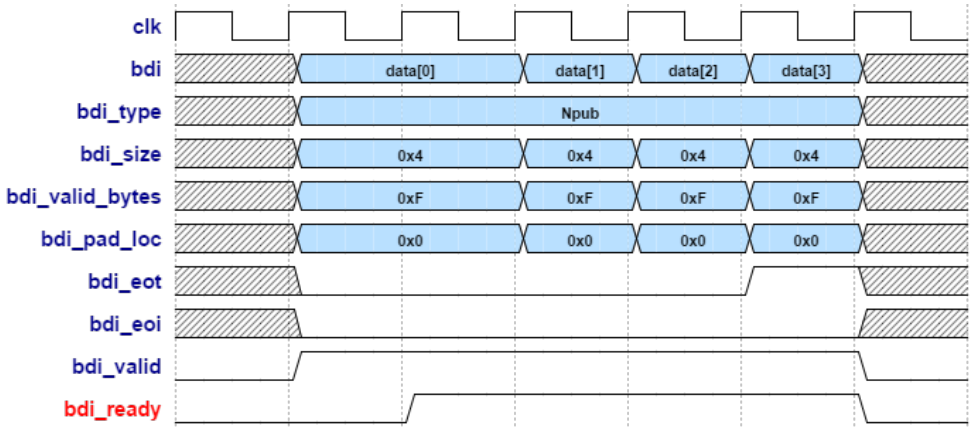


Figure 6.3: Handshake example of loading Npub, for w=32

Figures 6.4 and 6.5 illustrate examples of loading 120-bit AD and 104-bit message respectively.

Finally, an example of a handshake for authentication is shown in Fig. 6.6. For every decryption operation, the PostProcessor will set the *msg_ auth_ ready* signal to indicate its readyness to accept verification result. The result should be provided by CipherCore via `msg_auth` and indicated that it's valid by `msg_auth_valid`.

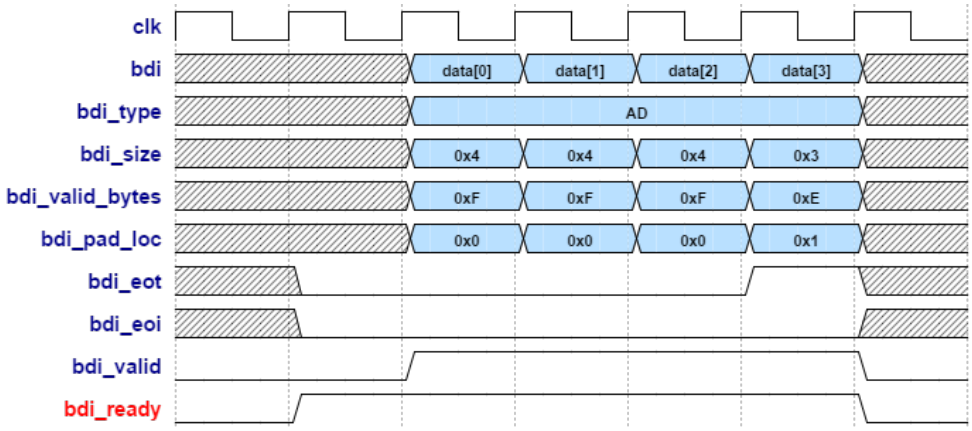


Figure 6.4: Handshake example of loading AD, for w=32, with data[3] containing the last 3 bytes of AD

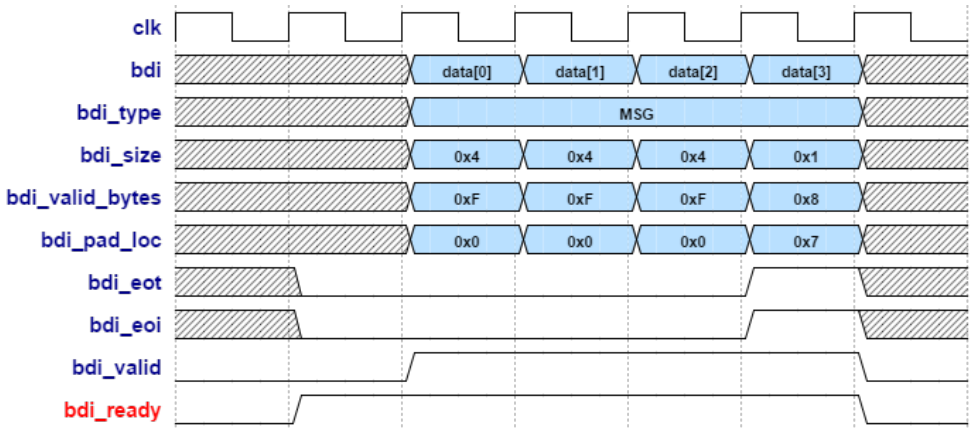


Figure 6.5: Handshake example of loading a message, for w=32, with data[3] containing the last 1 byte for encryption mode

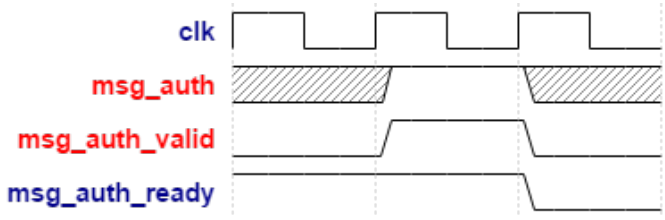


Figure 6.6: Handshake example for message authentication

6.3 Design Procedure

It is recommended that you start the development of the CipherCore, specific to a given authenticated cipher, by using the code provided in the Development Package, in the folder

`$ROOT/hardware/AEAD/src_rtl_lw`

In particular, the appropriate connections among the CipherCore, the PreProcessor, the PostProcessor, and the Header/Tag FIFO modules are already specified in this code. A designer needs to modify generics in the AEAD module, and then develop the CipherCore Datapath and the CipherCore Controller. The development of the CipherCore is left to individual designers and can be performed using their own preferred design methodology. Typically, when using a traditional RTL (Register Transfer Level) methodology, the CipherCore Datapath is first modeled using a block diagram, and then translated to a hardware description language (VHDL or Verilog HDL). The CipherCore Controller is then described using an algorithmic state machine (ASM) chart or a state diagram, further translated to HDL. An ASM chart of the CipherCore Controller typically contains the following states:

1. Idle
2. Wait Key
3. Load Key
4. Wait Npub
5. Load Npub
6. Wait AD
7. Load AD
8. Wait Data
9. Load Data
10. Process Data
11. Output Data
12. Process Tag
13. Output/Verify Tag (GenVer Tag).

An example ASM chart for the CipherCore Controller is shown Fig. 6.7. After a new instruction or after reset, the Controller should wait for the first block of data in the *Idle* state. The CipherCore should monitor the

bdi_valid for the first block of data, which is typically Npub. When this signal is active, the circuit should check whether the current key requires an update by inspecting the key_update signal. If it does, the controller changes its state to *Wait_ Key*. In this state, the Controller waits for the active value of the input key_valid, and then changes the state to *Load_ Key*. The key_ready should be activated in this state until all words of the key are read, and the new key is stored internally within the CipherCore or the corresponding round keys are pre-computed. A counter is needed to count the number of words at each load or process state. The counter will be incremented after each key_valid or bdi_valid signal and the value of counter is compared to the corresponding number of words generics (NumKwords, NumNwords, NumADwords, NumDwords and NumTwords).

Once a new key is activated or no new key is required (key_update=0), the circuit is ready to process the first block of data (Npub). It waits for the bdi_valid signal in *Wait_ Npub* state and loads the Npub in the *Load_ Npub* state. The bdi_ready signal must remain high until all words of Npub are loaded. The controller then moves to the next processing states, *Wait_ AD* and *Load_ AD* to load the Associated Data. In the case that Npub is the last block of data (AD size = Message/Ciphertext size = 0), which can be determined using the bdi_eoi input, the controller state can change directly to *Process_ Tag*. In the *Wait_ Data* state, the circuit waits until the next input block is valid (bdi_valid=1), and then loads and processes data in *Load_ Data* and *Process_ Data* states respectively.

Depending on the algorithm, additional processing may be required for the last block of data. This block can be determined using the end-of-type input (bdi_eot). At the same time, the end-of-input signal (bdi_eoi) may be stored in a register within the CipherCore to keep track of the last input state. This status register is useful to determine when no additional data block is expected after processing of the last AD block, so that the controller can progress to the *Process_ Tag* state directly.

In the *Process_ Tag* state, we finalize tag generation process and based on the decrypt_in signal, one of the *Output_ Tag* or *Verify_ Tag* states will be selected. During the authenticated encryption operation, the core should generate a new tag and pass it to the PostProcessor via the bdo bus. During the authenticated decryption operation, msg_auth_valid should be activated, and the msg_auth signal should be used to provide the result of authentication until msg_auth_ready is active.

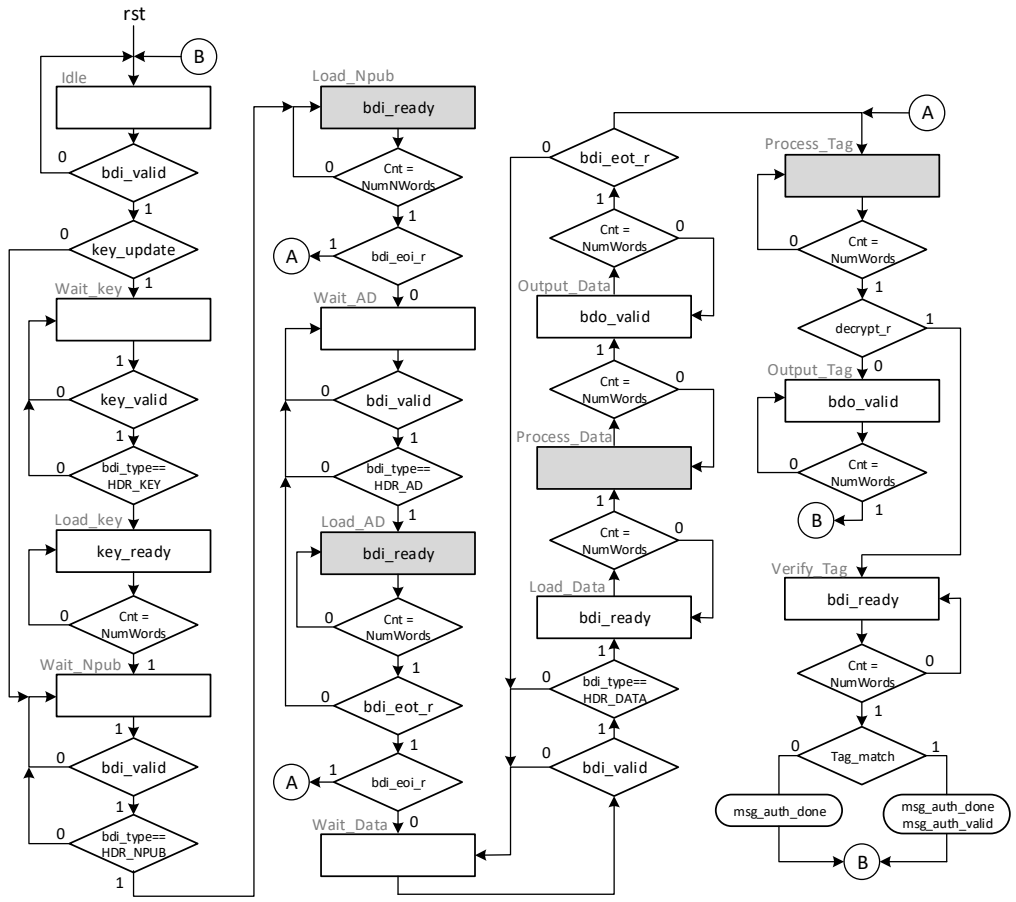


Figure 6.7: A typical Algorithmic State Machine (ASM) chart of the CipherCore Controller. Each shaded state in this diagram may need to be replaced by a sequence of states in the actual implementation of a complex authenticated cipher. $*\_r$ are status registers storing values of the respective inputs read during the last bdi handshake. HDR_KEY, HDR_NPUB, HDR_AD and HDR_DATA are defined in LWPI_pkg.vhd based on Table 6.2. NumDwords = DBLK_SIZE/W, NumKwords = KEY_SIZE/W, NumNwords = NPUB_SIZE/W, NumTwords = TAG_SIZE/W, and cnt is a counter value.

6.4 Dummy Authenticated Cipher

An example design of the ligtweight CipherCore and AEAD, corresponding to a dummy authenticated cipher, dummy1, is provided as a part of our distribution.

The following parameters are used:

- $AD_{block\_size} = PT_{block\_size} = Npub_{size} = 128$ bits
- $Pad(AD_n) = AD_n$ **if** $len(AD_n) = block\_size$ **else** $AD_n||10^*$
- $Pad(PT_m) = PT_m$ **if** $len(PT_m) = block\_size$ **else** $PT_m||10^*$

$$CT_i = PT_i \oplus i \oplus Key \oplus Npub \quad \text{for } i = 1..m-1. \tag{6.1}$$

$$CT_m = Trunc(PT_m \oplus i \oplus Key \oplus Npub, PT_m) \tag{6.2}$$

$$Tag = Key \oplus Npub \oplus Len \oplus \bigoplus_{i=1}^{n-1} AD_i \oplus Pad(AD_n) \oplus \bigoplus_{i=1}^{m-1} PT_i \oplus Pad(PT_m) \tag{6.3}$$

The code of the CipherCore is developed to work correctly with W=8, 16, and 32.

7 Verification

7.1 Test vector generation (*aeadtvgen*)

The Python script called *aeadtvgen* and accompanying examples provide a framework to generate test vectors for any authenticated cipher based on the user's specified parameters. The script is located in the folder

`$ROOT/software/aeadtvgen/aeadtvgen`

and the examples of calling it with parameters specific to multiple authenticated ciphers in the folder

`$ROOT/software/aeadtvgen/examples`

The framework relies on the reference implementations of authenticated ciphers (including, but not limited to CAESAR candidates) placed in the folder

`$ROOT/software/CAESAR`

Multiple reference implementations by themselves rely on the OpenSSL library.

7.1.1 Setup

In order to run *aeadtvgen*, you need to have installed in your system:

- gcc
- OpenSSL
- Python v3.5+

The below instructions describe how to install and configure these packages from scratch.

Windows

Download and install the latest 64-bit version of MSYS2 from http://www.msys2.org. The 64-bit version is the package with x86-64 identifier, e.g., `msys2-x86_64_20161025.exe`. The package will install MSYS2 and MinGW in `C:/msys64` by default. Navigate to this folder and open `mingw64.exe`. We will use this terminal for all subsequent operations.

```
pacman -Syu             # Update
# Close the terminal and open a new one once completed
pacman -Syu             # Second update

### Install tools to create shared libraries
pacman -S make          # Install make tool
# Warning! The following instructions use _pacboy_ command
pacboy -S gcc           # Install GCC
pacboy -S openssl       # Install openssl

### Install python3 and dependencies
pacboy -S python3       # install python3
pacboy -S python3-pip   # Install python3 package manager
pacboy -S python3-cffi  # Install python3-cffi and its dependencies

### Compile a distribution wheel (Optional)
# A distribution wheel (*.whl) will be created in the subfolder /dist
cd $ROOT/software/aeadtvgen
pip3 install wheel
python3 setup.py bdist_wheel

### Install test vector generation script
cd $ROOT/software/aeadtvgen/dist                # Go to the distribution folder
pip3 install aeadtvgen-<package_version>-py3-none-any.whl   # Install test vector
    generation tool

### Test that the program has been installed
### by calling help
python3 -m aeadtvgen -h
```

Linux

The following instructions assume the use of Ubuntu v16.04 or above.

```
### Install required tools
sudo apt-get install libssl-dev
sudo apt-get install python3-pip
sudo apt-get install python3-cffi

### Compile a distribution wheel (Optional)
# A distribution wheel (*.whl) will be created in the subfolder /dist
cd $ROOT/software/aeadtvgen
python3 -m pip install --upgrade pip
```

```
python3 -m pip install --upgrade setuptools
python3 -m pip install cffi==1.3.1
python3 -m pip install wheel
python3 setup.py bdist_wheel

### Install wheel
cd dist
python -m pip install aeadtvgen-{<package_version>}-py3-none-any.whl

### Test that the program has been installed
### by calling help
python3 -m aeadtvgen -h
```

7.1.2 Compiling shared libraries

```
### Compile example shared libraries
cd $ROOT/software/CAESAR

# Clean previously generated libraries
make clean

# Edit the PRIMITIVES variable inside of Makefile.txt to include libraries for the
    selected authenticated ciphers
# (make sure there is no trailing character after "/")

# Generate shared libraries
make
```

7.1.3 Adding a new library

A new software library, corresponding to a new authenticated cipher, can be added to our framework by first placing a CAESAR Software API compliant C/C++ source code in

`$ROOT/software/CAESAR/$new_algorithm/ref`

and performing the following modifications:

1. include the following line in the top-level file encrypt.c

   ```
   #include "../../dll.h"
   ```

2. prepend the keyword *EXPORT* to the definitions of the CAESAR API functions crypto_aead_encrypt() and crypto_aead_decrypt() located in the same file.

```
EXPORT int crypto\_aead\_encrypt(..) { ... }
EXPORT int crypto\_aead\_decrypt(..) { ... }
```

3. Edit the PRIMITIVES variable inside `Makefile.txt` to include the new algorithm.

7.1.4 Generating test vectors

It is recommended that the user understands the arguments of *aeadtvgen*, in order to properly create test vectors for the design under verification. The arguments to be used are the function of

- algorithm
- parameters of the algorithm (e.g., key size, block size)
- parameters of the implementation (e.g., public data input width, secret data input width, output width)
- phase of verification.

As a result, basic knowledge of the target design, including the parameters of the algorithm and implementation, are required. While it is possible to generate test vectors using pure shell command syntax, this process is likely to be error prone due to the large number of available options. Instead, we recommend that the user creates a Python script that utilizes *aeadtvgen* as a third party library in Python and then calls it using aeadtvgen(args).

Various examples of such Python scripts be found in

`$ROOT/software/aeadtvgen/examples`

An example of generating a set of test vectors for dummy1 (high-speed version) is shown below:

```
### Generate test vectors for dummy1
cd $ROOT/software/aeadtvgen/examples

# Create test vectors for high speed core dummy1
# Output files will be automatically copied to
# $ROOT/hardware/hs_examples/dummy1/scripts
python3 hs_dummy1.py
```

The user is encouraged to use the file
`$ROOT/software/aeadtvgen/examples/hs_dummy_1.py`
as a template and a starting point to create the customized script for the targeted design.

The provided template contains a list of possible options for the majority of use cases. It must be noted, however, that the user must take into account the specific characteristics of the algorithm and design when generating these test vectors. Providing as much coverage as possible ensures that the design can withstand a real-world usage.

In particular, a typical process of verifying the functionality of an authenticated cipher module includes the following phases, devoted to the verification of:

1. Single AD and Message/Ciphertext Block
2. Random Inputs with Custom Selected Sizes
3. Empty Message, Empty AD, Basic Message/ID Sizes
4. Randomly Generated Test Vectors with Varying AD, Message, and Ciphertext Lengths.

Test vectors for these phases can be generated using the *aeadtvgen* options:

1. *--gen_single*
2. *--gen_custom*
3. *--gen_test_routine*, and
4. *--gen_random*,

respectively, as illustrated in `hs_dummy_1.py`.

The choice of one of these phases can be accomplished simply by uncommenting the respective line of the script, e.g.,

```
## PHASE 3:
    args = basic_args + gen_test_routine
```

Please note that only for the --gen_single option, the knowledge of the key, Npub, Nsec, AD, and Data sizes is required to generate test vectors. For all other cases, these sizes are inferred from the values of basic arguments

(basic_args), such as --io, --key_size, --block_size, etc., which need to be specified only once.

After the analysis using these most commonly used sets of option, the designer has the flexibility of generating his own verification strategy, based on the detailed knowledge and understanding of options of *aeadtvgen*. This additional verification may be necessary to cover the full functionality offered by the specific algorithm, especially in case of encrypting and decrypting multiple inputs of various sizes and internal compositions.

7.2 Hardware Simulation

Once test vectors are generated, copy them into your simulation folder and ensure that the PWIDTH and SWIDTH generics of the testbench are set to PW and SW, respectively. Note that PW and SW are the widths, for the *pdi/do* and *sdi* buses, respectively. These parameters should be located in a package file called *design_pkg.vhd*.

Simulation is performed until the end-of-file is reached or a mismatch between expected output and actual output occurs. A clock signal is deactivated when either of these two conditions is met. In the case that user wants to ignore the simulation mismatch, one can set the STOP_AT_FAULT generic to $False$ and the testbench will ignore the verification error.

In the case that the target implementation is ASIC, the design can be simulated by setting ASYNC_RSTN to *True*.

Finally, in the practical experimental testing of any module, there is no guarantee that the input source will be ready with the new input whenever the module attempts to read it. Similarly, the destination circuit may not be always ready to receive the new output. These conditions must be comprehensively verified using simulation, before the experimental testing is attempted.

In our testbench, these conditions can be accomplished using the features of stalling input and stalling output. The rate at which the data is stalled can be configured using TEST_IPSTALL (public input stall), TEST_IPSTALL (secret input stall) and TEST_OSTALL (output stall), expressed in clock cycles. These settings will only become active if TEST_MODE is set to the value shown in Table 7.1.

Finally, it must be stressed that the aforementioned verification is paramount to ensuring that the design can withstand a real-world usage, where the in-

Table 7.1: Test modes

Value	Description
0	No stall
1	Input & Output stall test
2	Input only stall test
3	Output only stall test

termittent data transmission is very common. At the very least, the user should ensure that the design under verification is successfully validated when *TEST_MODE* is set to 1.

8 Generation and Publication of Results

Generation of results is possible for AEAD and CipherCore. We strongly recommend generating results primarily for AEAD. This recommendation is based on the fact that CipherCore has an incomplete functionality and a full-block-width interface.

In terms of optimization of tool options, for Virtex 7 and Zynq, we recommend generating results using Minerva [11, 12]. For Virtex 6 and below, we recommend using Xilinx ISE and ATHENa [13]. For Altera FPGAs, we suggest using Altera Quartus II and ATHENa.

In the case that the number of ports exceed the number of ports available in the target part, we recommend using a simple wrapper, with five ports: clk, rst, sin, sout, piso_mux_sel, provided as a part of the Development Package [1].

Finally, our database of results for authenticated ciphers is available at [14].

A The Development Package Description

The contents of our Development Package is shown in Table A.1.

Table A.1: Directory structure of the development package

Folder		Files	Description
software	aeadtvgen	*	The test vector generator program and its related setup
		examples/*	Examples of usage
	CAESAR/{algorithm}/ref		Directory that contains reference software implementation of a specific algorithm of a CAESAR candidate from the SUPERCOP distribution
hardware	AEAD/src_rtl_hs	AEAD.vhd	Entity only file of a Single-Pass AEAD
		AEAD_Arch.vhd	Architecture only file of a Single-Pass AEAD
		AEAD_pkg.vhd	Package file required for high-speed architectures
		AEAD_TP.vhd	Entity only file of a Two-Pass AEAD
		AEAD_TP_Arch.vhd	Architecture only file of a Two-Pass AEAD
		CipherCore.vhd	Single-Pass CipherCore template
		CipherCore_TP.vhd	Two-Pass CipherCore template
		fwft_fifo.vhd	First-Word-Fall-Through FIFO
		PostProcessor.vhd	PostProcessor
		PreProcessor.vhd	PreProcessor
	AEAD/src_rtl_lw	AEAD.vhd	Top-level module for lightweight architectures
		CAESAR_LWAPI_pkg.vhd	CAESAR Lightweight API package
		CipherCore.vhd	CipherCore template for lightweight architectures
		fwft_fifo.vhd	First-Word-Fall-Through FIFO
		GeneralComponents_pkg.vhd	Package containing general components
		PostProcessor.vhd	PostProcessor file
		PreProcessor.vhd	PreProcessor file
		Reg*.vhd	Various register files
		StepDownCountLd.vhd	Step-down counter
	AEAD/src_tb	AEAD_TB.vhd	Universal testbench
		std_logic_1164_additions.vhd	Additional simulation package
	AEAD/src_rtl /wrappers	AEAD_Wrapper.vhd	Wrapper file for the implementation of a Single-Pass AEAD
		CipherCore_Wrapper.vhd	Wrapper file for the implementation of a Single-Pass CipherCore
		AEAD_TP_Wrapper.vhd	Wrapper file for the implementation of a Two-Pass AEAD
		CipherCore_TP_Wrapper.vhd	Wrapper file for the implementation of a Two-Pass CipherCore
	_examples/dummy	KAT	Known-Answer-Tests for dummy designs
		src_rtl	Reference dummy related code
		scripts	ModelSim script folder to perform a quick simulation

B aeadtvgen help

```
usage: aeadtvgen [--gen_random N] [--gen_custom_mode MODE]
                 [--gen_custom Array] [--gen_test_routine BEGIN END MODE]
                 [--gen_single DECRYPT KEY NPUB NSEC AD PT] [-h] [--dbg]
                 [--verify_lib] [-V] [-v]
                 [--io PUBLIC_PORTS_WIDTH SECRET_PORT_WIDTH] [--key_size BITS]
                 [--npub_size BITS] [--nsec_size BITS] [--tag_size BITS]
                 [--block_size BITS] [--block_size_ad BITS] [--ciph_exp]
                 [--ciph_exp_noext] [--add_partial]
                 [--reverse_ciph REVERSE_CIPH]
                 [--msg_format SEGMENT_TYPE [SEGMENT_TYPE ...]] [--offline]
                 [--min_ad BYTES] [--max_ad BYTES] [--min_d BYTES]
                 [--max_d BYTES] [--max_block_per_sgmt COUNT]
                 [--max_io_per_line COUNT] [--pdi_file FILENAME]
                 [--sdi_file FILENAME] [--do_file FILENAME]
                 [--dest PATH_TO_DEST] [--human_readable] [--cc_hls]
                 [--cc_pad_enable] [--cc_pad_ad PAD_AD_MODE]
                 [--cc_pad_d PAD_D_MODE] [--cc_pad_style PAD_STYLE]
                 lib_path lib_name

Test vectors generator for CAESAR (Competition for
Authenticated Encryption: Security, Applicability, and Robustness)
candidates. The script REQUIREs that the C library for the
intended algorithm is compiled first.

:::::Required Parameters::::
  Library specifier::

  lib_path              Path to CAESAR shared library, i.e.
                            c:/GMU_HW_API_v2/software/lib.
  lib_name              Shared library's name, i.e. aes128gcmv1
                        Note: The library should be generated prior to the start
                        of the program.

:::::Test Generation Parameters::::
  Test vectors generation modes (use at least one from the list below)::
  Common notation and convetions:
  AD - Associated Data
  DATA - Plaintext/Message or Ciphertext
  PT - Plaintext/Message
  CT - Ciphertext
  (*)_LEN - Length of data (*) type, i.e. AD_LEN.
```

```
  Operation - 0: encryption, 1: decryption
  H* - a string composed of multiple repetitions of the hexadecimal
       digit H (the number of repetitions is determined by the size
       of a given argument)
       All lengths are expressed in bytes.

  For Boolean arguments, 0 can be used instead of False,
  and 1 can be used instead of True.

  --gen_random N        Randomly generates multiple test vectors with
                        varying AD_LEN, PT_LEN, and operation (default: 0)
  --gen_custom_mode MODE
                        The mode of test vector generation used by the --gen_custom
    option.

                        Meaning of MODE values:
                            0 = All random data
                            1 = Fixed test values.
                                Key=0xFF*, Npub=0x55*, Nsec=0xDD*,
                                AD=0xA0*, PT=0xC0*
                            2 = Same as option 1, except an input is now a running
    value
                                (each subsequent byte is a previous byte incremented
     by 1).
                         (default: 0)
  --gen_custom Array    Randomly generate multiple test vectors, with each test
    vector
                        specified using the following fields:
                           NEW_KEY (Boolean), DECRYPT (Boolean), AD_LEN, PT_LEN
                           ":" is used as a separator between two consecutive test
    vectors.

                        Example:
                        --gen_custom True,False,0,20:0,1,100,500

                        Generates 2 test vectors. The first vector will
                        create a new key and perform an encryption with a dataset
    that has
                        AD_LEN and PT_LEN of 0 and 20 bytes, respectively.
                        The second vector does _not_ generate a new key and perform
                        decryption with a dataset that has AD_LEN and PT_LEN of 100
                        and 500 bytes, respectively. (default: None)
  --gen_test_routine BEGIN END MODE
                        This mode generates test vectors for the common sizes of AD
    and PT
                        that the hardware designer should, at a minimum, verify.
                        The test vectors are specified using the following array:
                           [NEW_KEY (boolean),
                            DECRYPT (boolean),
                            AD_LEN,
                            PT_LEN]:
                        The following parameters are used:
                           [[True ,     False,      0,         0    ],
                            [False,     True,       0,         0    ],
```

```
                            [True ,     False,      1,          0    ],
                            [False,     True,       1,          0    ],
                            [True ,     False,      0,          1    ],
                            [False,     True,       0,          1    ],
                            [True ,     False,      1,          1    ],
                            [False,     True,       1,          1    ],
                            [True ,     False,      bsa,        bsd  ],
                            [False,     True,       bsa,        bsd  ],
                            [True ,     False,      bsa-1,      bsd-1],
                            [False,     True,       bsa-1,      bsd-1],
                            [True ,     False,      bsa+1,      bsd+1],
                            [False,     True,       bsa+1,      bsd+1],
                            [True ,     False,      bsa*2,      bsd*2],
                            [False,     True,       bsa*2,      bsd*2],
                            [True ,     False,      bsa*3,      bsd*3],
                            [False,     True,       bsa*3,      bsd*3],
                            [True ,     False,      bsa*4,      bsd*4],
                            [False,     True,       bsa*4,      bsd*4]]


                        where,
                            bsa is the associated data block size (block_size_ad),
    and
                            bsd is the data block size (block_size).

                        BEGIN (min=1,max=20) determines the starting test number.
                        END (min=1,max=20) determines the ending test numer.
                        MODE determines the test vector generation mode, where
                            0 = All random data
                            1 = Fixed test values.
                                Key=0xF*, Npub=0x5*, Nsec=0xD*,
                                Ad=0xA0*, PT=0xC0*
                            2 = Same as option 1, except each input is now a running
     value
                                (each subsequent byte is a previous byte incremented
     by 1).


                        Example:

                        --gen_test_routine 1 20 0

                        Generates tests 1 to 20 with MODE=0.


                        --gen_test_routine 5 5 1

                        Generates test 5 with MODE=1.
                         (default: None)
  --gen_single DECRYPT KEY NPUB NSEC AD PT
                        Generate a single test vector based on the provided values
    of all
                        inputs expressed in the hexadecimal notation.
```

```
                        Example:
                        --gen_single 0 5555 0123456 789ABCD 010204 08090A

                        Note:
                        KEY, NPUB and NSEC must have its size equal to the expected
    value.
                        Exception: NSEC is ignored --nsec_size is set to 0.
                         (default: None)

:::::Optional Parameters:::::
  Debugging options::

  -h, --help            Show this help message and exit.
  --dbg                 Run the C code with the DBG preprocessor flag. (default:
    False)
  --verify_lib          This operation will verify the generated test vectors
                        via the decryption operation.

                        Note: This option provides an additional check against
    possible
                              mismatch of results between encryption and decryption
                              in the reference software.
                         (default: False)
  -V, --version         show program's version number and exit
  -v, --verbose         Verbose for script debugging purposes. (default: False)

:
  Algorithm and implementation specific options::

  --io PUBLIC_PORTS_WIDTH SECRET_PORT_WIDTH
                        Size of PDI/DO and SDI port in bits. (default: (32, 32))
  --key_size BITS       Size of key in bits (default: 128)
  --npub_size BITS      Size of public message number in bits (default: 128)
  --nsec_size BITS      Size of secret message number in bits (default: 0)
  --tag_size BITS       Size of authentication tag in bits (default: 128)
  --block_size BITS     Algorithm's data block size (default: 128)
  --block_size_ad BITS  Algorithm's associated data block size.
                                This parameter is assumed to be equal to block_size
    if unspecified. (default: None)
  --ciph_exp            Ciphertext expansion algorithm. When this option is set, the
     last
                        block will have its own segment. This is required for a
    correct
                        operation of the accompanied PostProcessor.

                        Currently, we assume that PAD_AD and PAD_D are both set to 4
                        when this mode is used.
                         (default: False)
  --ciph_exp_noext      [requires --ciph_exp]

                        Additional option for the ciphertext expansion mode. This
    option
                        indicates that the algorithm does not expand the ciphertext
    (i.e.,
```

```
                        does not make the ciphertext size greater than the message
    size)
                        if the message size is a multiple of a block size.
                         (default: False)
  --add_partial         [requires --ciph_exp]

                        For use with --ciph_exp flag. When this option is set, a
    PARTIAL
                        bit will be set to 1 in the header of a data segment
                        if the size of this segment is not a multiple of a block
    size.

                        Note: This option is required for algorithms such as
    AES_COPA
                         (default: False)
  --reverse_ciph REVERSE_CIPH
                        Note: Not yet supported. Coming soon ~~~

                        [requires --ciph_exp]

                        Reversed ciphertext. When this option is set, the input
    ciphertext
                        is provided in a reversed order (including the possible
    length
                        segment).
                        Note: Only used by PRIMATEs-APE.')
                         (default: False)

:
  Formatting options::

  --msg_format SEGMENT_TYPE [SEGMENT_TYPE ...]
                        Specify the order of segment types in the input to
    encryption and
                        decryption. Tag is always omitted in the input to encryption
    , and
                        included in the input to decryption. In the expected output
    from
                        encryption tag is always added last. In the expected output
    from
                        decryption only nsec and data are used (if specified).
                        Len is always automatically added as a first segment in the
                        input for encryption and decryption for the offline
    algorithms.
                        Len is not allowed as an input to encryption or decryption
    for
                        the online algorithms.

                        Example 1:
                        --msg_format npub tag data ad

                        The above example generates
                        for an input to encryption: npub, data (plaintext), ad
```

```
                        for an expected output from encryption: data (ciphertext),
   tag
                        for an input to decryption: tag, data (ciphertext), ad
                        for an expected output from decryption: data (plaintext)

                        Example 2:
                        --msg_format npub_ad data_tag

                        The above example generates
                        for an input to encryption:  npub_ad, data (plaintext)
                        for an expected output from encryption: data_tag (
   ciphertext_tag)
                        for an input to decryption: npub_ad, data_tag (
   ciphertext_tag)
                        for an expected output from decryption: data (plaintext)

                        Valid Segment types (case-insensitive):
                            npub    -> public message number
                            nsec    -> secret message number
                            ad      -> associated data
                            ad_npub -> associated data || npub
                            npub_ad -> npub || associated data
                            data    -> data (pt/ct)
                            data_tag -> data (pt/ct) || tag
                            tag     -> authentication tag

                        Note: no support for multiple segments of the same type,
                        separated by segments of another type e.g., header and
   trailer,
                        treated as two segments of the type AD, separated by the
   message segments

                         (default: ('npub', 'ad', 'data', 'tag'))
  --offline             Indicate that the cipher is offline, i.e., the length of AD
   and
                        DATA must be known before the encryption/decryption starts.
   If this
                        option is used, the length segment will be automatically
   added as
                        a first segment in the input to encryption and decryption.
                        Otherwise, the length segment will not be generated for
   either
                        encryption or decryption.
                         (default: False)
  --min_ad BYTES        Minimum randomly generated AD length (default: 0)
  --max_ad BYTES        Maximum randomly generated AD length (default: 1000)
  --min_d BYTES         Minimum randomly generated data length (default: 0)
  --max_d BYTES         Maximum randomly generated data length (default: 1000)
  --max_block_per_sgmt COUNT
                        Maximum data block per segment (based on --block_size)
   parameter (default: 9999)
  --max_io_per_line COUNT
                        Maximum data length in multiples of I/O width in a data line
    of test
```

```
                        file. This option helps readability when a test vector is
    large.

                        Example:
                        If a user wants to limit a vector representation of data in
    a file
                        to a block size where a block size is 64-bit and I/O = 32-
    bit,
                        the value should be set to 2 (32*2 = 64 bits).

                        --io 32 --block_size 64
                        DAT = 000102030405060708090A0B0C0D0E0F

                        --io 32 --block_size 64 --max_io_per_line 2

                        DAT = 0001020304050607
                        DAT = 08090A0B0C0D0E0F
                         (default: 9999)
  --pdi_file FILENAME   Public data input filename (default: pdi.txt)
  --sdi_file FILENAME   Secret data input filename (default: sdi.txt)
  --do_file FILENAME    Data output filename (default: do.txt)
  --dest PATH_TO_DEST   Destination folder where the files should be written to. (
    default: .)
  --human_readable      Create a human readable file (tests_vectors.txt) for each
                        test vector in the format similar to NIST test vectors
                        used in SHA-3, i.e.:

                        # Message 1
                        Key     = HEXSTR
                        Npub    = HEXSTR
                        Nsec_PT = HEXSTR    # if --nsec_size > 0
                        AD      = HEXSTR
                        PT      = HEXSTR
                        Nsec_CT = HEXSTR    # if --nsec_size > 0
                        CT      = HEXSTR
                        TAG     = HEXSTR
                         (default: False)

:
  [Experimental] CipherCore options::

  --cc_hls              Generates test vectors for CipherCore in C (used by HLS)
                         (default: False)
  --cc_pad_enable       Enable padding operation (default: False)
  --cc_pad_ad PAD_AD_MODE
                        Associated data padding mode (default: 0)
  --cc_pad_d PAD_D_MODE
                        Data input padding mode (default: 0)
  --cc_pad_style PAD_STYLE
                        Padding style (default: 1)
```

Bibliography

[1] Cryptographic Engineering Research Group (CERG) at GMU. (2017, December) Development Package for Hardware Implementations Compliant with the CAESAR Hardware API. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=CAESAR

[2] E. Homsirikamol, P. Yalla, A. Ferozpuri, W. Diehl, F. Farahmand, M. X. Lyons, and K. Gaj. (2017, August) Benchmarking of Round 2 CAESAR Candidates in Hardware: Methodology, Designs and Results. [Online]. Available: https://cryptography.gmu.edu/athena/presentations/CAESAR_R2_HW_Benchmarking_v1.1.pdf

[3] ——, "Toward Fair and Comprehensive Benchmarking of CAESAR Candidates in Hardware: Standard API, High-Speed Implementations in VHDL/Verilog, and Benchmarking Using FPGAs," in *Directions in Authenticated Ciphers, DIAC 2016, Nagoya, Japan, Sep. 2016*.

[4] CERG. (2017, December) VHDL/Verilog Code of CAESAR Candidates: Summary I. [Online]. Available: https://cryptography.gmu.edu/athena/CAESAR_HW_Summary_1.html

[5] ——. (2017, December) VHDL/Verilog Code of CAESAR Candidates: Summary II. [Online]. Available: https://cryptography.gmu.edu/athena/CAESAR_HW_Summary_2.html

[6] E. Homsirikamol, F. Farahmand, W. Diehl, and K. Gaj. (2017, December) Benchmarking of Round 3 CAESAR Candidates in Hardware: Methodology, Designs and Results. [Online]. Available: https://cryptography.gmu.edu/athena/presentations/CAESAR_R3_HW_Benchmarking.pdf

[7] M. Tempelmeier, F. DeSantis, J.-P. Kaps, and G. Sigl, "to be published. currently under peer review," 2018.

[8] CERG. (2017, August) GMU Source Code of Round 3 and Round 2 CAESAR Candidates, AES-GCM, AES, AES-HLS, and Keccak Permutation F. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=CAESAR_source_codes

[9] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, P. Yalla, J.-P. Kaps, and K. Gaj, "CAESAR Hardware API," Cryptology ePrint Archive, Report 2016/626, June 2016, http://eprint.iacr.org/.

[10] ——. (2016, June) Addendum to the CAESAR Hardware API v1.0. [Online]. Available: https://cryptography.gmu.edu/athena/CAESAR_HW_API/CAESAR_HW_API_v1.0_Addendum.pdf

[11] F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated Hardware Optimization Tool," in *2017 International Conference on Reconfigurable Computing and FPGAs - ReConFig 2017, Cancun, Mexico*, Dec 2017.

[12] CERG. (2017, December) Minerva: Automated Hardware Optimization Tool. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=Minerva

[13] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa – automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs," in *20th International Conference on Field Programmable Logic and Applications - FPL 2010*. IEEE, 2010, pp. 414–421.

[14] Cryptographic Engineering Research Group (CERG) at GMU. (2017, December) GMU ATHENa Database of Results. [Online]. Available: https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view