Implementer's Guide to the CAESAR Hardware API

Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri,
Farnoud Farahmand, and Kris Gaj

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

May 12, 2016

Contents

1 Introduction

The CAESAR Hardware API [1] 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 [1] 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.

Our proposed API is suitable for both high-speed and lightweight implementations of authenticated ciphers. The only difference at the API level is the width of 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).

As a result, we leave the internal structure of any lightweight implementation entirely to the designers of such implementations. The only support we provide to the designers of lightweight implementations is in the areas of test vector generation (Chapter 6), simulation (Chapter 7), as well as result generation and publication (Chapter 8).

On the other hand, for the designers of high-speed implementations, we provide the following support:

- universal top-level block diagram (see Fig. 2.1)

- universal VHDL code for the PreProcessing unit
- universal VHDL code for the PostProcessing unit
- hardware API for the heart of the design, called CipherCore
- implementer's guide to designing any specific CipherCore
- VHDL code for the three dummy CipherCores following the CipherCore API

Below we describe all these supporting materials one by one. It should be stressed that the high-speed 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 API [1].

2 Top-level Block Diagram of a High-Speed Implementation

The proposed top-level block diagram of a high-speed, non-pipelined implementation of any authenticated cipher compliant with the CAESAR hardware API is shown in Fig. 2.1.

The top-level unit, called AEAD, is divided into four lower-level units, called PreProcessor, PostProcessor, Command (CMD) FIFO, and CipherCore. The universal VHDL codes 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. These codes are provided as a part of the Development Package [2]. 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, PostProcessor, and CMD FIFO.

Because of the availability of the open source code for the PreProcessor, PostProcessor, and CMD FIFO, the designers of high-speed implementations of authenticated ciphers 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 PreProcessor

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

- parsing segment headers

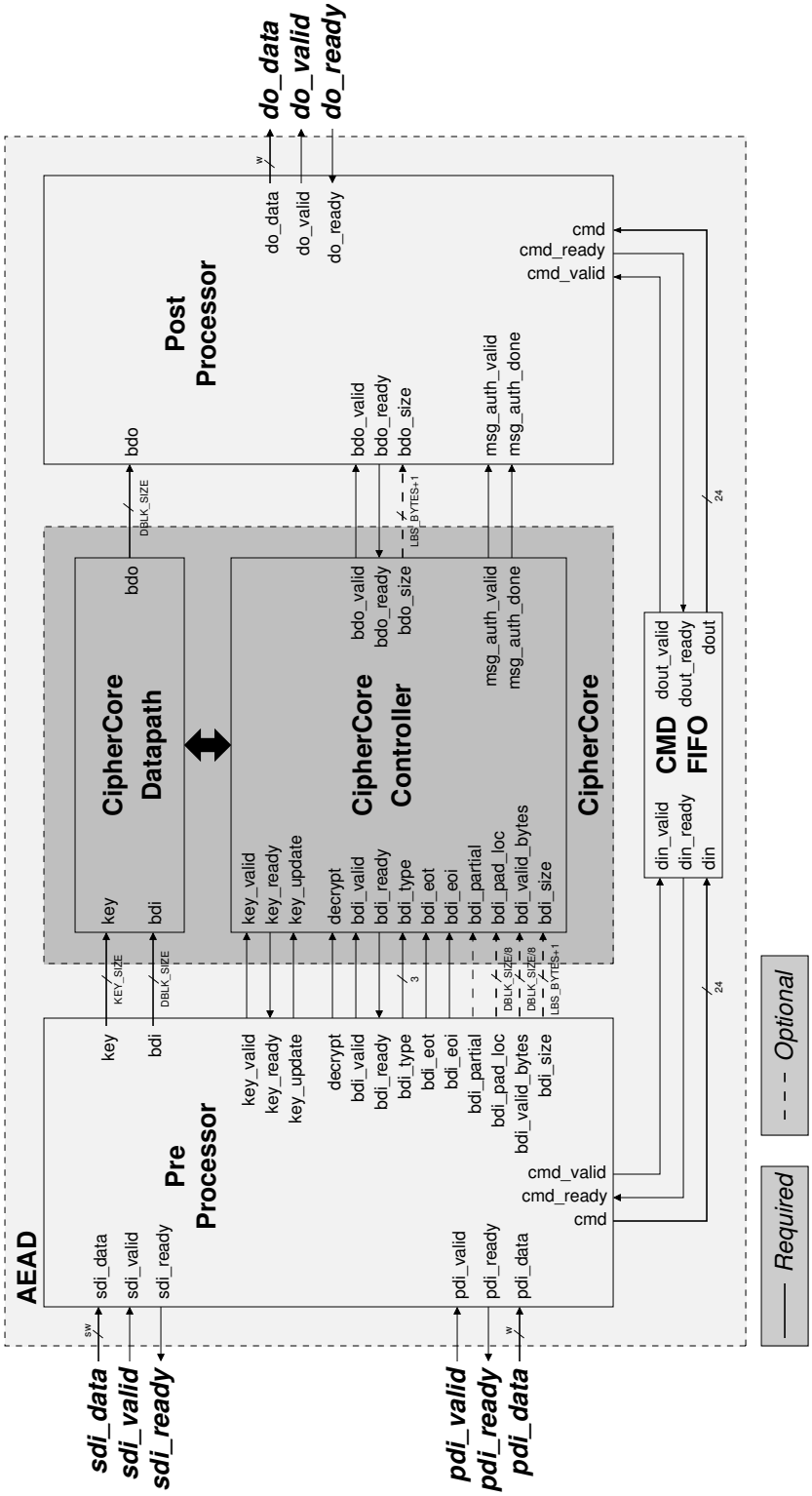
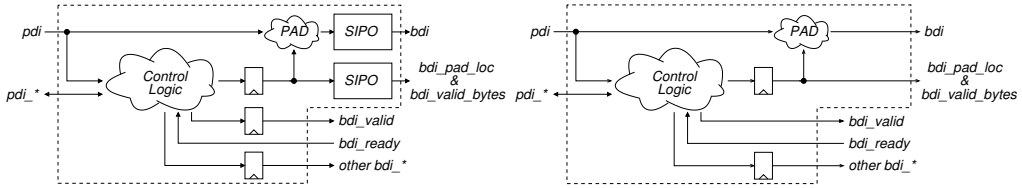


Figure 2.1: Top-level block diagram of a high-speed architecture of an authenticated cipher core, AEAD



(a) registered (b) non-registered

Figure 2.2: 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.

- 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.

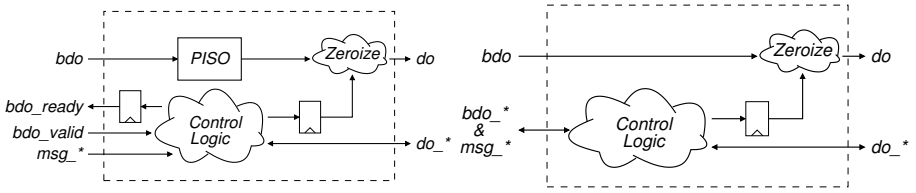
An overview of the PreProcessor design is shown Fig. 2.2. This unit can be configured to operate in two modes, registered and non-registered. The choice between these modes is made based on the width of public data input, PDI, (denoted as w in Fig. 2.1) and the size of an input block (denoted as $DBLK\_SIZE$ in Fig. 2.1).

In a typical scenario, where the size of an input block is larger than the width of PDI, w, the PreProcessor operates in the *registered* mode. If the width of PDI 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).

2.2 PostProcessor

The PostProcessor is responsible for the following tasks:

- clearing any portions of output blocks not belonging to the ciphertext or plaintext



(a) registered (b) non-registered

Figure 2.3: The PostProcessor Design. PISO = Parallel-In Serial-Out unit. msg_*, bdo_*, and do_* stand for all PostProcessor ports, shown in Fig. 2.1, with the names starting from the respective strings.

- 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. 2.3. 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) and the width of the data out, DO port (equal to width of public data input, PDI). Namely, when an output block size is larger than the width of DO, 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.

2.3 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 [1], Fig. 7, contain only 4 significant bits, and segment headers, defined in [1], Fig. 8, contain only 24 significant bits.

3 The Development and Benchmarking of High-Speed and Lightweight Implementations

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 entity declaration for high-speed implementations (Chapter 4)
2. Develop CipherCore (Chapter 5)
3. Generate test vectors (Chapter 6)
4. Verify the AEAD design (including the CipherCore design) using functional simulation (Chapter 7)
5. Generate optimized results for AEAD 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)
2. Develop the entire AEAD core from scratch, based on the CAESAR Hardware API specification [1]
3. Generate test vectors (Chapter 6)

4. Verify the AEAD design using functional simulation (Chapter 7)

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 declaration of AEAD for high-speed implementations is available as a part of the Development Package in the file
`$ROOT/hardware/AEAD/src_rtl_hs/AEAD.vhd`
This entity declaration contains 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 entity for high-speed implementations:

4.1.1 I/O Port Widths

Consistently with the specification of the CAESAR hardware API, 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 described in the AEAD entity declaration using generics W and SW.

4.1.2 Block sizes

Values of 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.

4.2 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 only values of generics G_W and G_SW, used to determine the I/O port widths, w and sw, respectively. Consistently with the specification of the CAESAR hardware API, the allowed values of these port widths are as follows:

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

Table 4.1: AEAD Generics

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
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.
Block Size Parameters in Bits			
ABLK_SIZE	integer	128	Associated data block size. This value should be smaller than or equal to DBLK_SIZE.
DBLK_SIZE	integer	128	Data (message/ciphertext) block size
KEY_SIZE	integer	128	Key size
TAG_SIZE	integer	128	Tag size. Note: This value is not used when MERGE_TAG is True.
Padding Parameters			
PAD_STYLE	integer	1	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.

Table 4.2: Extended description of PAD_STYLE.

Value	Description
0	No padding
1	10* padding rule

Table 4.3: Parameters of PAD_AD and PAD_D. 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

5 CipherCore Development for High-Speed Implementations

5.1 Interface

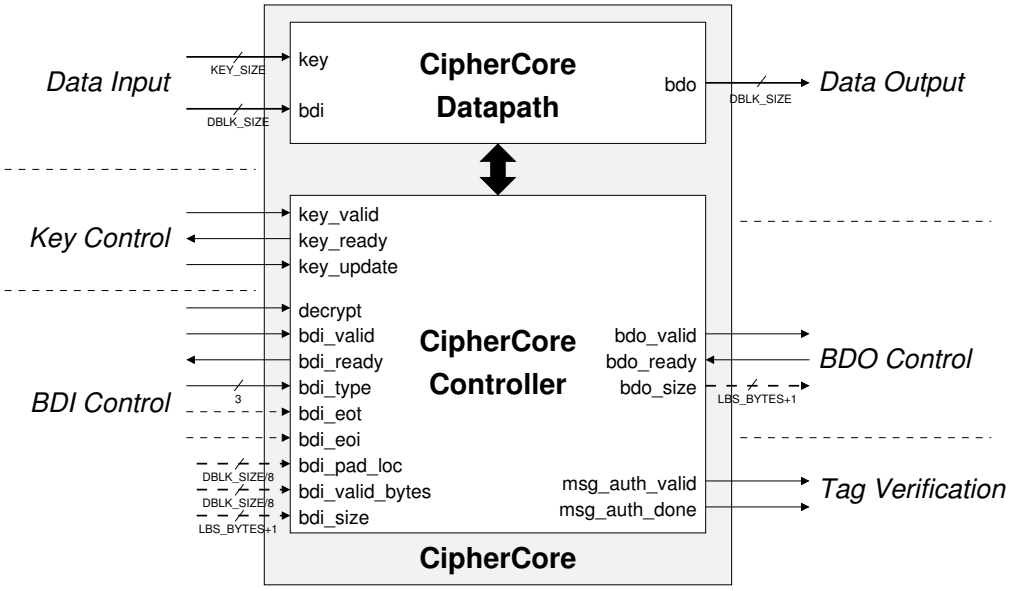


Figure 5.1: CipherCore

The interface of CipherCore is shown in Figure 5.1. Ports marked using dashed lines 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 *key_valid* and *key_ready* handshake signals.

key_update is used to notify the CipherCore that it should update the internal key prior to processing the next message.

Similarly to the *key* port, the *bdi* port is controlled using the *bdi_valid* and *bdi_ready* handshake signals. The *decrypt* signal informs the core whether the current operation is encryption or decryption. The *bdi_type* input indicates the type of input data, with the encoding shown in Table 5.1. It must noted that all ports of the BDI communication group and *bdi* are synchronized with the *bdi_valid* input. Their values should be read only when the *bdi_valid* signal is high.

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

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

The same scenario also applies to the block data output port (*bdo*) and its associated control signals, which are synchronized with the value of the *bdo_valid* output. *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 message authentication 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. Note that *msg_auth_valid* signal is synchronized with *msg_auth_done* signal.

Port descriptions are provided in Table 5.3. Ports related to `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.

Additionally, 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 w=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

Table 5.3: CipherCore Port Descriptions. 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 5.2.
bdi_valid_bytes	in	DBLK_SIZE/8	[BLOCK INFO] Encoding of the byte locations that are valid. See Table 5.2.
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_valid	in	1	1=Authentication success, 0=Authentication failure
msg_auth_done	in	1	Authentication done

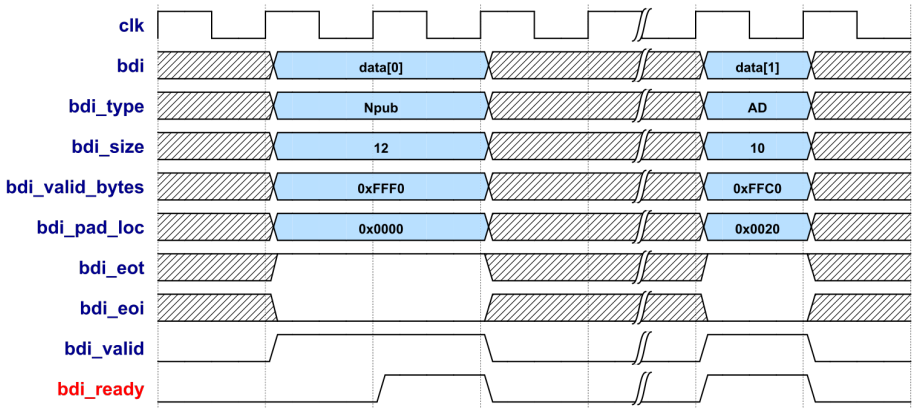


Figure 5.2: An example of a handshake used for loading data using the input *bdi*

5.2 Handshakes

This section presents examples of handshakes. All ports in the figures of this section are represented by the blue and the red color, for input and output ports, respectively. Fig. 5.2 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 5.3.

Fig. 5.4 provides an example of a handshake used to write output to the PostProcessor. Fig. 5.4a presents an example for the standard mode of operation of an authentiocated cipher. Figure 5.4b 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 update 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 a handshake for authentication is shown in Fig. 5.5. For every decryption operation, PostProcessor should issue the *msg_auth_done* signal to indicate the completion of the authentication check. At the same time, *msg_auth_valid* is captured by the PostProcessor to determine the result of authentication. These two signals should only be activated once for every decryption. Subsequent values of of the *msg_auth_done* signal during the same decryption operation are ignored.

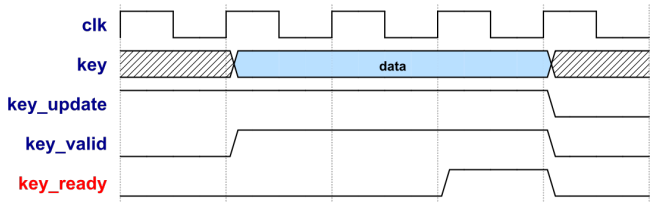
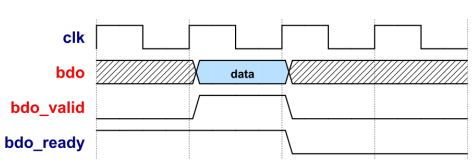
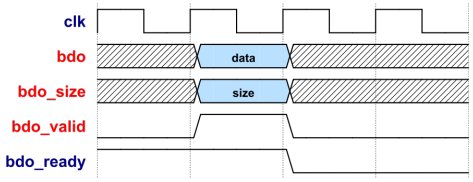


Figure 5.3: An example of a handshake used for loading a key



(a) Standard mode



(b) Ciphertext expansion mode

Figure 5.4: An example of a handshake used for writing data in the a) Standard mode, b) Ciphertext expansion mode

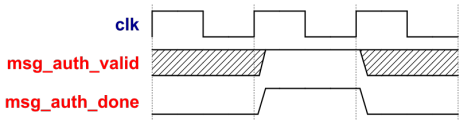


Figure 5.5: An example of a handshake used to perform message authentication

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 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. 5.6. 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.

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

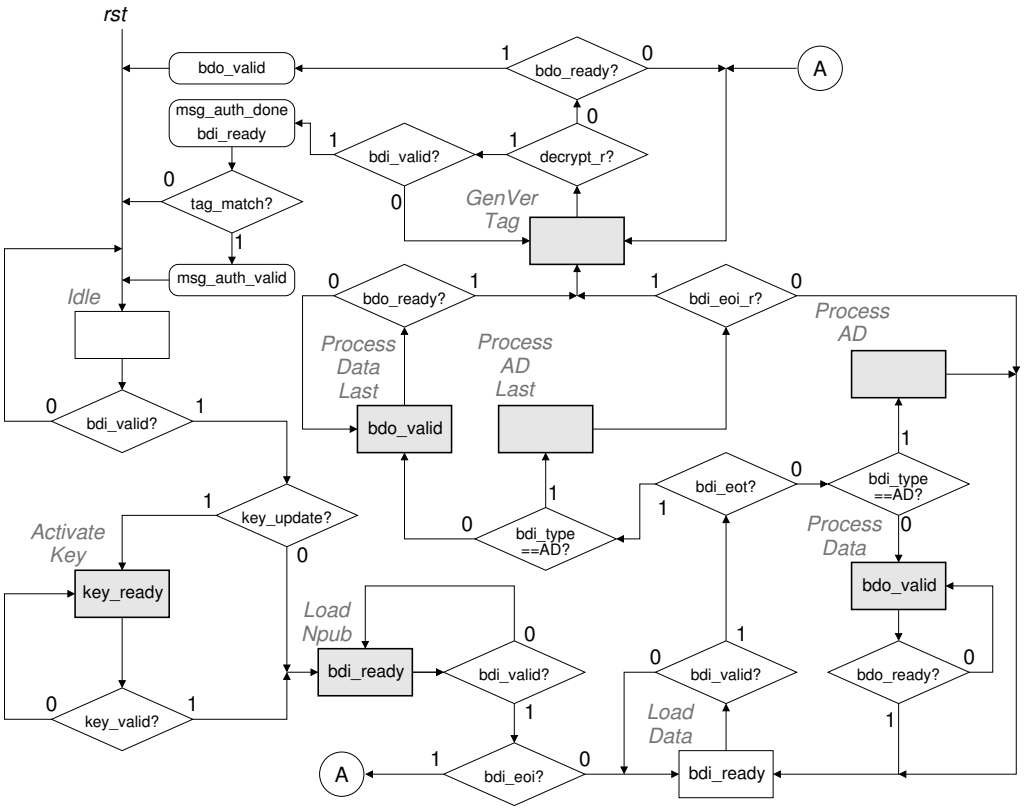


Figure 5.6: A typical Algorithmic State Machine (ASM) chart of the CipherCore Controller. Each shaded state in this diagram may need to be replaced by a sequence of states in the actual implementation of a complex authenticated cipher. $*\_r$ are status registers storing values of the respective inputs read during the last bdi handshake.

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 Post-Processor via the `bdo` bus. During the authenticated decryption operation, `msg_auth_done` should be activated, and the `msg_auth_valid` signal should be used to provide the result of authentication.

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 is specified using the following equations:

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

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

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

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

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

$$CT_m = Pad(PT_m) \oplus m \oplus Key \oplus Npub \tag{5.6}$$

when **CIPH_EXP**=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 bytes) || 64-bit plaintext length (in bytes).

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 5.4.

Table 5.4: Summary of features/parameters of five dummy authenticated ciphers and their high-speed implementions

	CIPH EXP?	Npub Size	AD		PT		Tag Size	Len-gth?	Pre-Processor	
			Block size	Pad?	Block size	Pad?			Data buffer?	Key buffer?
dummy1	False	96	128	True	128	True	128	True	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 allow 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 a 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 codes, block diagrams, and ASM charts of AES and Keccak Permutation F have been developed and made available at [1]. 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 Test Vector Generation

Test vectors for the targeted algorithm can be generated using our test vector generator (*aeadtvgen*), available in the *software* folder of our Development Package. The program relies on a reference software implementation of the given algorithm, located in the folder
`$ROOT/software/CAESAR/$algorithm/ref`,
where `$algorithm` is the name of the cipher or the ciphers's variant, such as `acorn128`, `deoxyseq128128v13`, etc. A limited set of reference C implementations of Round 2 CAESAR candidates is provided as part of our Development Package. A common feature of all these implementations is that they follow the CAESAR software API [3].

In the case when the software implementation of the targeted algorithm is not available as part of our package, the user can add a new implementation at the location noted above. Then, a slight modification has to be made to the file containing the definitions of functions crypto_aead_encrypt() and crypto_aead_decrypt(), typically called *encrypt.c*. In particular,

1. an additional header file (`dll.h`), located in the folder `$REPO/software/CAESAR`, needs to be included at the beginning of the aforementioned file

2. the EXPORT keyword needs to be placed in front of the declarations of crypto_aead_encrypt() and crypto_aead_decrypt().

For example, the beginning of the modified file may need to look as follows:

```
#include "../../dll.h"
EXPORT int crypto_aead_encrypt(..) { ... }
EXPORT int crypto_aead_decrypt(..) { ... }
```

In case of either preinstalled or new implementations, there may exist a dependence on a specific software library, such as Open SSL. The instruc-

tions on the installation of this very popular cryptographic library can be found in Appendix B.2.

In Appendix B, we also provide instructions on the installation of additional C and Python tools and libraries, recommended to be used together with our test vector generation application.

A standard procedure for generating test vectors can be executed as follows:

1. Create shared CAESAR libraries (*.dll in Windows and *.so in Linux)

   a) In your console, navigate to the CAESAR folder: `$root/software/CAESAR`.
   In particular, in Windows, please perform this step using the *msys* console.

   b) Modify *Makefile* to include only targeted primitive(s).

   c) (Situational) An algorithm may require the OpenSSL library in order to compile. If it does, one needs to provide an appropriate compilation flag inside of the following clause:

   ```
   ifeq ($(OS),Windows_NT)
   ...
   else
   ...
   endif
   ```

   Note: The flags are available but uncommented by default.

   d) type

   ```
   make
   ```

2. Generate the script using the aeadtvgen python program. The user can directly call the program from the command line, or create a script similar to examples shown in the folder: `$root/software/aeadtvgen/examples`.
   The full description of the program can be found by typing

   ```
   python -m aeadtvgen -h
   ```

3. Copy the three generated test vectors files (pdi.txt, sdi.txt, and do.txt) to the simulation folder.

7 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 the same numerical values as the generics W and SW of `AEAD.vhd`.

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

In the case when the target implementation is ASIC, one may need to by set the generic ASYNC_RSTN to *True*.

Finally, four test modes, summarized in Table 7.1, are provided to simulate the conditions when the availability of the source and/or destination communication modules is intermittent. The number of clock cycles between the two consecutive one-clock-cycle periods of the availability of these modules can be configured using the generics TEST_ISTALL and TEST_OSTALL, for input and output, respectively.

Table 7.1: Test modes

Value	**Description**
0	Always available
1	Input & Output intermittent test
2	Input intermittent test
3	Output intermittent test

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 case AEAD, for Virtex 7 and Zynq, we recommend generating results using Xilinx Vivado [4], operating in the Out-of-Context (OOC) mode [5]. In this mode, no pin limit applies. For Virtex 6 and below, since Xilinx ISE must be used, and the OOC mode is not supported by this tool, 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].

In case of CipherCore, because of a large number of port bits and limited effectiveness of the OOC mode, we recommend using the aforementioned five-port wrapper for all FPGA families.

In terms of optimization of tool options, for Virtex 7 and Zynq, we recommend the use of 25 default optimization strategies available in Xilinx Vivado. The corresponding scripts, used to run Xilinx Vivado in batch mode, are included in our Development Package [1]. For Virtex 6 and below, we recommend using Xilinx ISE and ATHENa [6]. For Altera FPGAs, we suggest using Altera Quartus II and ATHENa.

Our database of results for authenticated ciphers is available at [7]. After receiving an account in the database, the designers can enter results by themselves.

Bibliography

[1] Cryptographic Engineering Research Group (CERG) at GMU. (2016, May) CAESAR Hardware API. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=download

[2] ——. (2016, May) CAESAR Development Package. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=download

[3] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. (2016, May) Cryptographic competitions. [Online]. Available: http://competitions.cr.yp.to/index.html

[4] Xilinx. Vivado Design Suite. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[5] ——, *Vivado Design Suite User Guide: Hierarchical Design*, April 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug905-vivado-hierarchical-design.pdf

[6] 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.

[7] Cryptographic Engineering Research Group (CERG) at GMU. (2016, May) GMU ATHENa Database of Results. [Online]. Available: https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/rankings_view

A Description of the Development Package

The contents of our development package is shown in Table A.1.

Table A.1: Directory structure of the development package

Folder	Files	Description
scripts	VivadoBatch	Folder that contains a set of scripts for result generation using Vivado
software	aeadtvgen/aeadtvgen	Test vector generation program
	aeadtvgen/dist	Pre-compiled Python wheel binary folder
	aeadtvgen/examples	Example usage folder
	CAESAR/{algorithm}/ref	Folder that contains C implementation of a specific algorithm
hardware	AEAD/src_rtl_hs	AEAD.vhd	Entity only file of AEAD
		AEAD_Arch.vhd	Architecture only file of AEAD
		fwft_fifo.vhd	First-Word-Fall-Through FIFO
		CipherCore.vhd	CipherCore template
		PostProcessor.vhd	PostProcessor file
		PreProcessor.vhd	PreProcessor file
	AEAD/src_rtl_lw	AEAD.vhd	Top-level template file for lightweight hardware design.
	AEAD/src_tb	AEAD_TB.vhd	Universal testbench
		std_logic_1164_additions.vhd	Additional simulation package
	AEAD/src_rtl_hs/wrappers	AEAD_Wrapper.vhd	Wrapper file for implementation of AEAD
		CipherCore_Wrapper.vhd	Wrapper file for implementation of CipherCore
	dummy*/KAT		Known-Answer-Test folder for dummy design
	dummy*/src_rtl		Reference dummy related code
	dummy*/scripts	modelsim.tcl	ModelSim script to perform a quick simulation

B Installation of Libraries and Tools

B.1 Interpreter and compiler

B.1.1 Windows

- **MinGW with MSYS as a compiler**
  Download and install the latest version from http://www.mingw.org. MSYS should be included in the installation package.
  Note: MSYS is the console for MinGW in Windows.

  Below is an example on how to compile the program using MinGW with MSYS console (MinGW shell).

```
cd /c/Downloads/GMU_API_v20/software/CAESAR
make
```

- **Python v3.5+**
  Download and install the latest Python distribution package from https://www.python.org.
  Note: Please make sure that all installations are done with administrative privileges and the path to the python folder is correctly set using the environmental variable.

B.1.2 Linux

- **Python v3.5+**

B.2 OpenSSL Installation

1. Download and uncompress the latest version of OpenSSL
2. Navigate to download folder and uncompress files
   a) Open terminal (MSYS console for Windows)
   b) Navigate to the download folder
      - **Windows**
        ```
        cd /c/Users/$USER/Downloads/openssl-1.0.2e
        ```
      - **Linux** Open terminal
        ```
        cd /home/$USER/Downloads
        ```
   c) Uncompress downloaded file, e.g.
      ```
      tar -zxvf openssl-1.0.2e.tar.gz
      ```
   d) Change working directory
      ```
      cd openssl-1.0.2e
      ```
   e) Configure OpenSSL
      - **Windows**
        ```
        ./Configure mingw --prefix=/usr/local shared
        ```
        Note: Possible error
        "gcc command not found".
        This error is caused by a problem during the installation, when your MinGW */bin* folder is not pointed to by the environmental variable. You can either try to re-install or add the variable manually. To do it manually
        i. Access environmental variables in the Windows system, *right-click @ My Computer* (or *This PC* on some systems) -> Properties -> Advanced System Settings -> Advanced tab -> Environmental Variables
        ii. Prepend *C:/MinGW/bin;* to *PATH* variable by editing *PATH* variable in either *User variable for $USER* or *System variable*.

iii. OK -> Apply

- **Linux**

```
./Configure --prefix=/usr/local shared
```

f) Compile and install

```
make && make install
```

B.3 Python module (aeadtvgen)

The distribution package for *aeadtvgen* can be found as a wheel (*.whl) package under `$root/software/aeadtvgen/dist` folder. To install, type

```
python -m pip install _PACKAGED_MODULE_.whl
```