AES core for FPGA on SEcube™

Project Documentation

Release: October 2021


HW
SW
Open
Source

Proprietary Notice

The following document offers information, which is subject to the terms and conditions described hereafter.
While care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinions expressed are in good faith and while every care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinion expressed are in good faith and while every care has been taken in preparing this document.

Authors

Nicole DAI PRA' s274501@studenti.polito.it
Lorenzo GROTTESI s268978@studenti.polito.it
Leonardo IZZI s278564@studenti.polito.it
Paolo PRINETTO *(President, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *(PhD Candidate, Politecnico di Torino)* gianluca.roascio@polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Trademarks

Disclaimer

Contents

1 Work presentation

The purpose of this project is to demonstrate the potential of the **SE*cube*™** platform in being used as a cryptographic accelerator also from the point of view of its FPGA component. In fact, the presence on board of the reconfigurable Lattice MachX02-7000 hardware[1] represents an important opportunity for the fast execution of algorithms concerning data encryption and signature, which can be inserted within the context already created by Software Development Kit[2], and used efficiently by mature services such as **SE*key*™** [3] and **SE*file*™** [4].

The core is inserted within the environment referring to the IP Manager project[5], so the resources available on the device, as well as the operating frequency, are determined by the presence of other functional blocks.

This document is intended to present all information on the designed core. After providing the background on the algorithm, the core architecture will be illustrated, followed by a detailed guide on how to insert the core into the **SE*cube*™** environment and how to interface with it. Details will also be provided about a sample test program that has been included in the project source files in order to perform the functional check of the core.

[1] http://www.latticesemi.com/view_document?document_id=38834
[2] https://github.com/SEcube-Project/SEcube-SDK
[3] https://github.com/SEcube-Project/SEkey
[4] https://github.com/SEcube-Project/SEfile
[5] https://github.com/SEcube-Project/IP-core-Manager-for-FPGA-based-design

2 Overview

2.1 Why AES

AES is a symmetric algorithm developed in the late 1990s. It is a NIST specification developed to supersede the DES and 3DES algorithms[6]. AES is a block cipher that works with three different key sizes: 128, 192 and 256 bits. Being a symmetric algorithm, it requires less bits for the key and less computational power with respect to asymmetric algorithms such as RSA.

Due to its structure, which is based on simple logical operations, it is suitable for hardware implementations.

2.2 The AES Algorithm

AES works on blocks of 128 bits, iterating over the data multiple times to ensure a strong encryption. The 128-bit block can be pictured as a 4x4 8-bit matrix. The key, which can be 128, 192 or 256 bits wide, is first expanded to generate multiple *round keys*, all of 128 bits. The key size also determines the number of iterations, that is rounds, used to process the data block. In each round, the following operations are executed:

- *SubBytes*, which performs a non-linear byte substitution
- *ShiftRows*, which rotates the matrix's rows by different offsets
- *MixColumns*, which is a mixing operation
- *AddRoundKey*, which computes the XOR between the data block and a round key

The decryption process follows a similar approach, where the operations are the inverse of the encryption ones and the ordering is slightly changed.

2.3 AES Modes

Block cyphers are always associated with a mode of operation. The mode is used to describe how the algorithm must be applied to the data.

The modes are:

- **ECB**: Each block is independently processed;
- **CBC**: The output of each processed block is applied to the input of the next one;
- **CTR**: It turns the block cipher in a stream one. Each block is independently encrypted using a counter;
- **CFB**: It turns the block cipher in a stream one. It uses the output of the block i as input of the block $i + 1$;
- **OFB**: It turns the block cipher in a stream one. It works as CFB, with the difference that the input of the $i + 1$ block is the keystream of the previous one.

[6] United States National Institute of Standards and Technology (NIST). Announcing the ADVANCED ENCRYPTION STANDARD (AES). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

3 System Architecture and Behavior

This Section is intended to present the architecture of the developed core. This core is a modular solution which provides a high degree of flexibility. However, given the system and libraries constraints of the current state of the platform, the key generation could not fit in the FPGA and it has been moved to software. This choice led to a non-negligible overhead, which will be characterized later on.

The standard provides one way to encrypt data, but two for decrypting it. These two methods are called *inverse cipher* and *equivalent inverse cipher*. The original AES software library by Blu5 present in the firmware, which is the one used by this prohect to expand the keys, uses the latter implementation. For this reason, the decryption datapath follows it too. The nice property of the equivalent inverse cipher is that the operation order is the same as the encryption one, meaning that a single control unit can be used to manage both of them.

Given the same ordering in encryption and decryption, it is, in theory, possible to use a single datapath for both the encryption and decryption process. With a multiplexer in front of each stage, it would be possible to reuse existing components to their full extent. Unfortunately, given the fact that the datapath works on 128-bit data, the multiplexer size requires a lot more area than 2 dedicated datapath. The difference is so much that the multiplexed solution would not fit in the FPGA.

Having multiple, independent datapath turns out to be also a future-proof solution. In the event that the system will migrate on a multi-core architecture, with a bigger FPGA, it is possible to duplicate the datapath to parallelize even more the workload.

The encryption and decryption units are pipelined to improve the core performance. It would be possible, then, to work in parallel on multiple processes. The same could be done by adding more datapath and by sharing the same keys between them. Yet, such approach would be applicable only for two modes of operations: ECB and CTR, which are the only parallelizable ones. In contrast, the target of this project is to support every possible mode, hence parallel encryption/decryption is not supported.

3.1 CPU-FPGA Communication

The communication between the CPU and the FPGA is supported by the IP Manager, whose project resources can be found at link
https://github.com/SEcube-Project/IP-core-Manager-for-FPGA-based-design. It provides a 64-entry 16-bit shared memory where the CPU and the cores can communicate. The first memory block, that is the one at address `0x00`, is reserved for the control word. This leaves 63 halfwords usable for data exchange. As explained in Section 2, the core can work only on 128 bits at the time, since only one encryption or decryption block can be processed at any time. The shared memory locations chosen for transferring the data have been selected in the range `0x08`–`0x0F`. These particular values are used for the easiness of address generation in hardware. `0x08` corresponds to `001000` in binary, while `0x0F` corresponds to `001111`. With 8 16-bit locations is possible to transfer exactly 128 bits, so these addresses can be generated with a simple counter on 3 bits concatenated to `001`. Only using the counter is not feasible, because it would use the reserved word located at `0x00`.

The IP manager provides 2 possible communication modes: *polling mode* and *interrupt mode*. On the current software, using interrupt is of no use, because the CPU would have to wait nevertheless the processed data before continuing its work. In other words, an encryption operation can be considered as atomic and blocking for the application flow. Therefore, polling has been selected. The polling status is located in a different memory location than the ones used for data transfer. Its address is `0x01`. The CPU writes `0xFFFF` after the AES block processing is started.

Once the core has finished to transfer the data in the Data Buffer, it writes `0x0000` to unlock the CPU.

For further details on the IP Manager, please refer to its documentation at the link reported above.

3.2 High-Level Design

At a high level, the AES core is a standard IP core to be handled by the IP Manager unit. The main architecture is shown in Figure 1.

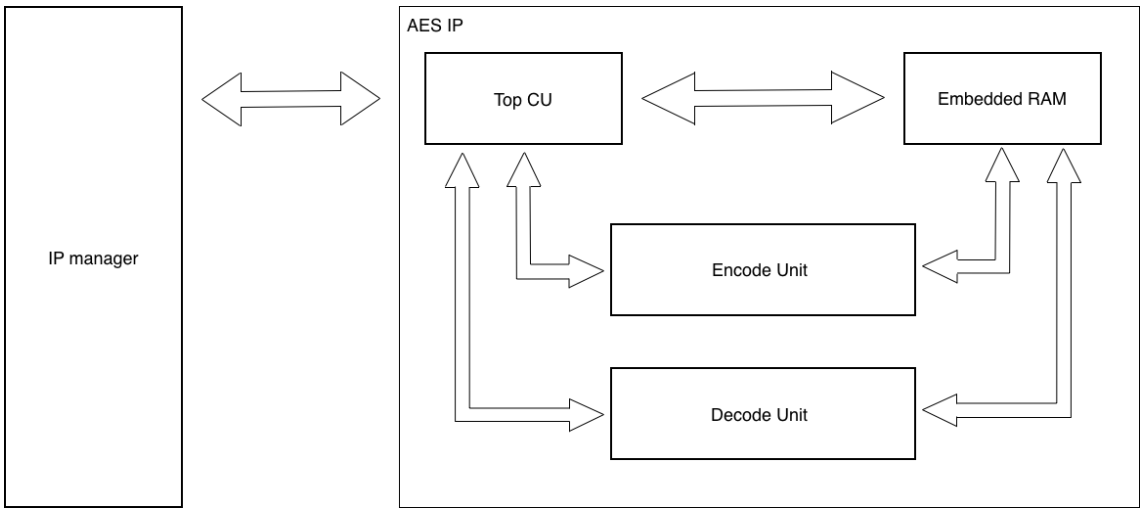


Figure 1: AES core high-level view.

To ensure a correct timing in the communications and the exchange of data, there is a component called Top CU which implements the IP manager interface. Its main task is to regulate the data traffic and to activate the right AES operation as requested by the user. Due to area constraints it was not feasible to perform the key expansion on FPGA, hence the Top CU also receives the round keys from the CPU and transfer them on a dedicated embedded RAM. The RAM has a 128-bits data bus and a 4-bits address bus, which is enough to contain all the round keys required by AES-256.

Once the keys and data to process are received, the Top CU starts the actual computation on one of the two processing units. In particular, one unit is in charge of the encryption process while the other of the decryption one. These units are designed in such a way to allow the co-existance of multiple replicas, if there are enough FPGA LUTs available. Each unit has its own dedicated CU and datapath, hence it is trivial in the future to add support for parallel processing calls by simply adjusting the Top CU unit.

The units are designed to signal the end of their computation, so, once the Top CU receives such signal, it sends the processed data to the data buffer and communicates to the CPU that it is allowed to read such data.

3.2.1 Byte Ordering

As the keys are generated in software and then used in hardware, it is important to keep the same byte ordering. The hardware expects everything to be in little-endian. While this holds for the data, it does not for the keys. In fact, the original firmware library of AES by Blu5 reverses the

bytes of every 32-bit word. To not place the burden of the swap software-side, it is implemented on FPGA.

3.3 Low-Level Design

In this Subsection, all the main components of the AES core will be detailed.

3.3.1 Top Control Unit

The Top CU is the middle layer between the IP Manager and the processing units. It is the hardware component which, based on CPU directives, orchestrate the work on FPGA. Its main duties are:

1. to reconstruct the keys and data coming from the IP Manager;
2. start the encryption or decryption process with the right modality (i.e., AES 128, 192 or 256);
3. send back the data to the IP Manager.

The first and last operations are performed in multiple steps because, while the AES standard and hence the core work on 128-bits at the time, the CPU-FPGA connection allows to send only 16-bits at the time.
The Top CU is a standard HLSM, with the following states:

- IDLE
- ALG_SEL
- KEY_TRANSFER
- KEY_WRITE
- DATA_RX
- ENCRYPTION
- DECRYPTION
- ENC_DATA_TX
- DEC_DATA_TX
- CORE_DONE
- WAIT_TR_CLOSE

In the following, for each state a detailed explanation of what the Top CU does will be given, along with its opcode. For debugging purposes, the opcode is also forwarded on the LEDs present on the **SE*cube*™** DevKit. Since there are 8 LEDs, but the opcode is encoded on 6 bits, the two MSBs are fixed to 0, which means that the corresponding LEDs are always turned on. A visual explanation is available in Figure 2.

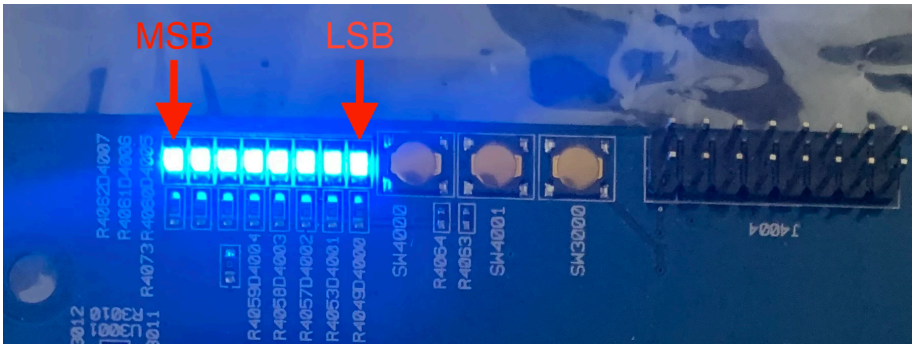


Figure 2: **SE*cube*™** LEDs

IDLE Opcode: `0x00`

In `IDLE`, the core waits to be enabled. As the core works only in polling mode, it also checks if the right communication mode has been selected. If that is not the case, it remains in this state, otherwise it jumps to the opcode passed by the IP Manager. While in `IDLE`, the internal counters and shift registers used to access and store correctly are reset.

ALG_SEL Opcode: `0x02`

In `ALG_SEL`, the data buffer is enabled in reading mode. When a CPU write is detected, the value stored by the CPU at the address `0x08` is sampled in an internal register. Such value represents the number of keys that must be sampled by the CU. Implicitly, this value also specifies if the algorithm to be executed is AES 128, AES 192 or AES 256.

Due to the current implementation of the Data Buffer inside the IP Manager architecture, the signal `cpu_write_completed` is on the critical path, which is already close to the target frequency of 60 MHz. Hence, checking here whether the number of keys is a valid number is not feasible, so the assumption here is that the user always passes the correct value.

The CU then automatically goes to the `WAIT_TR_CLOSE` state.

KEY_TRANSFER Opcode: `0x04`

In `KEY_TRANSFER`, the `Top` CU reads the keys from the data buffer. Due to the key size and the data buffer structure, they are read 16 bits at the time. Each key slice is sampled in a dedicated 16-bit register, which are sequentially activated each time a CPU write is detected. Once all the slice registers are filled, the CU moves to the `KEY_WRITE` state. Instead, if the number of stored keys in the RAM matches the value passed in `ALG_SEL`, the CU goes to `WAIT_TR_CLOSE`.

KEY_WRITE Opcode: `0x05`

In `KEY_WRITE`, the 128 bits key built in `KEY_TRANSFER` is written into the embedded RAM. The CU automatically goes back to `KEY_TRANSFER`.

DATA_RX Opcode: `0x08`

In `DATA_RX`, the CU performs a similar job as done in `KEY_TRANSFER`. It builds a standard 128

bits data from 16-bit slices read obtained from the data buffer. Once all the slices are received, the CU moves to the WAIT_TR_CLOSE state.

ENCRYPTION Opcode: 0x10

In ENCRYPTION, the AES encryption unit is started. It executes a different number of rounds depending on the selected algorithm. If an invalid value has been passed during the ALG_SEL operation, an error is raised. The RAM is now piloted by the encryption unit. When the computation is over, the CU switches to the ENC_DATA_TX state.

DECRYPTION Opcode: 0x11

The work done in this state is the same as the one done in ENCRYPTION, with the only difference being that the unit started is the decryption one.

ENC_DATA_TX Opcode: 0x18

The data is enabled in write mode. In each clock cycle, a 16-bit slice of encrypted data is written in the data buffer. While the last slice is being written in the buffer, the CU goes in the CORE_DONE state.

DEC_DATA_TX Opcode: 0x19

It does the same work as in ENC_DATA_TX, with the only difference being that the data transferred is the decrypted one.

CORE_DONE Opcode: 0x20

It unlocks the CPU, giving it the possibility to read the processed data. The CU waits in this state until the core is disabled by the IP Manager, then it goes back to the IDLE state.

WAIT_TR_CLOSE Opcode: 0x01

This state is entered each time it is necessary to wait the deactivation of the core, i.e., a different transaction is needed to continue the work.

3.4 Datapath

The datapath closely follows the AES specification. As already explained in Section 2, the encryption and decryption datapath follow the same operation ordering. For this reason, the rest of the explanation will focus on the encryption datapath, highlighting the differences with the decryption one only when it is relevant.

The main structure is shown in Figure 3. Such implementation aims to reduce the area as much as possible without sacrificing performance. In fact, besides the components implementing the functions, only three 128-bit registers and a single 128 bits multoplexer are required. The registers are used to pipeline the design to not kill the frequency. The multiplexer, instead, is used to implement the first and last AES rounds. They differ from the standard ones, as in the former only an AddRoundKey operation is required, while in the latter the MixColumns must be bypassed. Going deeper in the architecture, in the following the blocks implementation will be discussed.

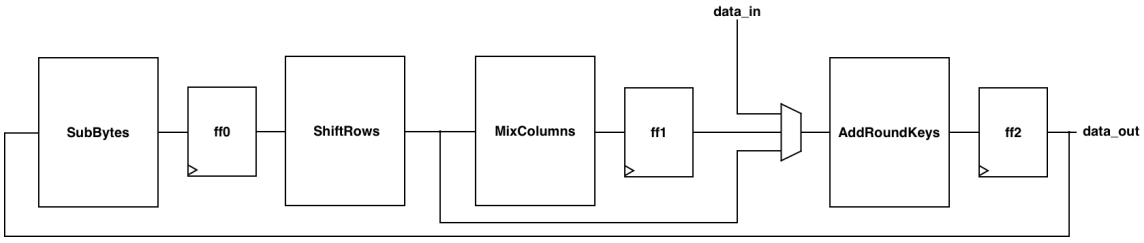


Figure 3: Encryption datapath.

The **SubBytes** is an operation where each byte goes through a non-linear substitution. It is easily implemented in hardware by look-up tables. However, since a table for each byte is needed, the required area grows rapidly. They are implemented on LUTs and not on the embedded RAMs for two reasons: the number of RAMs was not sufficient to compensate the additional clock cycle required to sample the input address, and it has been decided to leave some free RAMs to other cores in the event that another IP can be integrated. To not use an excessive amount of LUTs, the SubBytes works in two clock cycles, processing 64 bits at the time. For the decryption, the only change is in the content of the look-up tables.

The **ShiftRows** rotates the data of each row by a different offset. This operation is free on hardware, as it can be reduced to a simple rewiring. Hence, it executes in the same clock cycle of the MixColumns. For the decryption, the direction of the rotation is inverted.

The **MixColumns** is a matrix multiplication between the data (status) block and the following matrix:

$$\begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix}$$

for the encryption, and the following matrix:

$$\begin{bmatrix} 0x0e & 0x0b & 0x0d & 0x09 \\ 0x09 & 0x0e & 0x0b & 0x0d \\ 0x0d & 0x09 & 0x0e & 0x0b \\ 0x0b & 0x0d & 0x09 & 0x0e \end{bmatrix}$$

for the decryption. In hardware, this operation is splitted in 4 vector multiplications run in parallel. Since these two matrices are constants, the multiplication can be replaced by a series of additions. For example, $x \cdot 0x0e$ can be rewritten as $x \cdot 2^3 + x \cdot 2^2 + x \cdot 2^1$. As the addition can be implemented with XORs and a multiplication by 2^n can be translated as a left shift by n, the hardware required to implement the multiplication is greatly reduced. The AES standard specifies that such operations are performed in GF(2^8), so that the result always fits 8 bits due to the modulo operation. Computing the modulo in hardware is even more complex than a multiplication, but luckily the standard suggests the following solution: whenever the MSB of the byte to be shifted is '1', the multiplication must be compensated by XORing the result with `0x1b`. Such operation is called `xtime`, and it must be repeated every time a shift occurs.

This algorithm is the basis of the MixColumns unit.

The **AddRoundKey** is the byte-by-byte XOR between the data block and a round key.

3.5 Control Unit

The control unit implementation is shared between the encryption and decryption units, as the algorithm and the blocks layout is the same. However, each unit has its own instantiation. The main purposes of this CU are to count the number of rounds executed to know when the computation is over and to select the right expanded key, as well as to drive the datapath's mux.

During the first round, a dedicated signal, called `first_round`, is raised by the CU to pass to the AddRoundKey stage the input data. Then, for a variable amount of rounds, it switches between 3 states that represent the SubBytes, the ShiftRows + MixColumns, and AddRoundKey. In the first of these, it waits for the SubBytes to communicate the end of its operations. Such behavior allows to easily change the SubBytes, so that it can use more or less clock cycles in case a reduction in area or a speed-up is required.

Eventually, it has two states representing the final SubBytes and ShiftRows + AddRoundKey. In these states, a dedicated signal called `last_round` is raised, to bypass the MixColumns output.

4 Application Program Interface

This Section is intended to describe the driver developed for interfacing with the AES hardware implemented on FPGA. The developer can find the implementation of the AES via FPGA in the files "AES_FPGA.h" and "AES_FPGA.c". Since AES is a block cipher, the stream of data to encrypt/decrypt must be organized in blocks of constant length. This lead the algorithm to work in two main steps: the AES *block processing* and the *operating mode*. The block processing stage is done inside the FPGA, while the operating mode is performed via software. The AES_FPGA API supports all the key lengths: 128, 192 and 256 bits, in both encryption and decryption. The supported operating modes are: **ECB, CBC, CTR, OFB, CFB**. Furthermore, the library supports both the standard-AES mode and the CMAC-AES mode.

It is important to highlight that the AES_FPGA files must be correctly integrated with the SDK, as explained later in Section 5. So, this Section will assume:

- the correct integration of the IP Manager infrastructure with the SDK;
- the correct integration of the "AES_FPGA.h" and "AES_FPGA.c" in the SDK;
- the correct programming of the FPGA.

4.1 How to Use this Driver

The AES_FPGA driver is thought to work in the workflow shown in Figure 4, for both encryption and decryption.

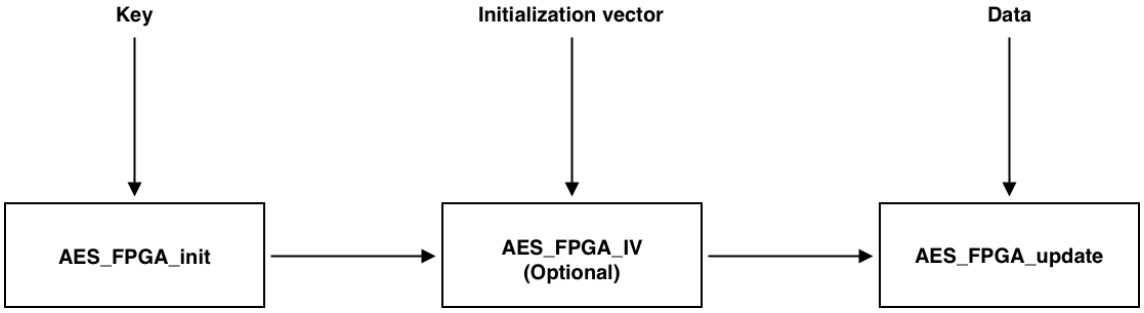


Figure 4: API workflow.

Whenever the operation changes from encryption to decryption or vice-versa, the structure must be re-initialized with the new modality. The initialization function is used to setup the context according to the input key and its length. In this phase the key is also used to generate all the AES keys via the expansion algorithm. Lastly, the method passes all the generated keys to the FPGA. In order to setup the context, the initialization function requires to know the operating mode that is going to be used. Some operating modes, such as CBC, CFB or OFB, require an initialization vector. Depending on the chosen operating mode, the IV function may be necessary to setup the Initialization Vector. Once the context and the IV (if necessary) are initialized correctly the algorithm can start. The data must be packed in group of 128 bit to respect the block size of AES. Then, the update function must be called specifying the input data and the number of data blocks. Whenever the data length is not compliant with a multiple of the block size, the update method will return an error. In this case the developer must perform padding of the data to fit the block size.
The same workflow can be applied considering the CMAC-AES version using the CMAC methods. In this case, the padding is automatically performed by the algorithm.

4.2 API Synopsis

```
AES_FPGA_RETURN_CODE AES_FPGA_init(B5_t_AesCtx *ctx, const
   uint8_t *Key, int16_t keySize, aes_mode_t aes_mode);
```

Inputs are:

- `ctx`, Blue5 context structure pointer. If the function success the context will be initialized correctly.
- `key`, pointer to the input key.
- `keySize`, size of the input key in bytes. The only available values are: 16, 24 and 32 for AES 128, 192 and 256 respectively.
- `mode`, AES mode selector. Supported values are: `AES_OFB`, `AES_ECB_ENC`, `AES_ECB_DEC`, `AES_CBC_ENC`, `AES_CBC_DEC`, `AES_CFB_ENC`, `AES_CFB_DEC`, `AES_CTR`.

The function will return:

- `AES_KEY_INIT_ERROR`, when the initialization of the context and the keys fails.
- `FPGA_ERROR`, when the sharing of the key with the FPGA fails.
- `AES_FPGA_RES_OK`, when all the operations are successful.

```
int32_t AES_FPGA_SetIV(B5_tAesCtx *ctx, const uint8_t *IV);
```

Inputs are:

- `ctx`, already initialized pointer to the Blu5 context structure.
- `IV`, pointer to the Initialization Vector.

The function will return:

- `B5_AES256_RES_INVALID_CONTEXT`, when the `ctx` is not initialized.
- `B5_AES256_RES_INVALID_ARGUMENT`, when invalid `IV`.
- `B5_AES256_RES_OK`, when `IV` is correctly added in the context.

```
AES_FPGA_RETURN_CODE AES_FPGA_update(B5_tAesCtx *ctx, uint8_t *
   encData, uint8_t *clrData, uint16_t nBlk, uint32_t data_len);
```

Inputs are:

- `ctx`, already initialized pointer to the Blu5 context structure.
- `encData`, pointer to ciphertext.
- `clrData`, pointer to plaintext[7].
- `nBlk`, number of data blocks.
- `data_len`, size of the data.

[7] NOTE: Depending on the operating mode in the context, the input variable may vary. For instance, supposing to have an encryption operation in `ctx`, the input variable will be `clrData` and the output will be `encData`. When decrypting, the input will be `encData` and output will be `clrData`. The variables `nBlk` and `data_len` refer to the operational input.

The function will return:

- FPGA_ERROR, when the communication with FPGA fails.
- AES_FPGA_RES_OK, when all the operations are successful.

```
int32_t AES_FPGA_Cmac_Init(B5_tCmacAesCtx *ctx, const uint8_t *
    Key, int16_t keySize);
```

Inputs are:

- ctx, pointer to the Blu5 CMAC context structure. If the function succeeds, the context will be initialized correctly.
- key, pointer to the input key.
- keySize, size of the input key in bytes. The only available values are: 16, 24 and 32 for AES 128, 192 and 256 respectively.

The function will return:

- B5_CMAC_AES256_RES_INVALID_CONTEXT, when invalid ctx.
- B5_CMAC_AES256_RES_INVALID_ARGUMENT, when invalid key.
- B5_CMAC_AES256_RES_OK.

```
int32_t AES_FPGA_Cmac_Update(B5_tCmacAesCtx *ctx, const uint8_t
    *data, int32_t dataLen);
```

Inputs are:

- ctx, already initialized pointer to the Blu5 CMAC context structure.
- data, data pointer.
- dataLen, size of the data.

The function will return:

- B5_CMAC_AES256_RES_INVALID_CONTEXT, when invalid ctx.
- B5_CMAC_AES256_RES_INVALID_ARGUMENT, when invalid data.
- B5_CMAC_AES256_RES_OK.

```
int32_t AES_FPGA_Cmac_Finit(B5_tCmacAesCtx *ctx, uint8_t *
    rSignature);
```

Inputs are:

- ctx, already initialized pointer to the Blu5 CMAC context structure.
- rSignature, pointer to a blank memory area that can store the computed output signature.

The function will return:

- B5_CMAC_AES256_RES_INVALID_CONTEXT, when invalid ctx.

- `B5_CMAC_AES256_RES_INVALID_ARGUMENT`, when invalid `data`.
- `B5_CMAC_AES256_RES_OK`.

```
int32_t AES_FPGA_Cmac_reset(B5_tCmacAesCtx *ctx);
```

Inputs are:

- `ctx`, already initialized pointer to the Blu5 CMAC context structure.

The function will return:

- `B5_CMAC_AES256_RES_INVALID_CONTEXT`, when invalid `ctx`.
- `B5_CMAC_AES256_RES_OK`.

```
int32_t AES_FPGA_Cmac_Sign(const uint8_t *data, int32_t dataLen,
    const uint8_t *Key, int16_t keySize, uint8_t *rSignature);
```

Inputs are:

- `data`, pointer to the input data.
- `dataLen`, input data length (in bytes).
- `Key`, pointer to the key that must be used.
- `keySize`, key size.
- `rSignature`, pointer to a blank memory area that can store the computed output signature.

The function will return:

- `B5_CMAC_AES256_RES_OK`.

5 User Manual and Validation Test

The following Section is intended to explain how to use the core, starting from its insertion in the development environment.

In order to validate the FPGA design coupled with the API, a specific test suite has been written. Given the fact that the FPGA AES API follows the same structure and workflow of the Blue5 AES API, the validation test is designed as an equality check using the Blue5 algorithm as golden model. A software random number generator generates the keys and data, which are then fed to both algorithms (Blue5 and FPGA). They are configured with the same operating mode and the same modality (encryption or decryption). To observe the outcome of the test, a UART communication is used to launch the test and visualize the results. The overall structure of the test is shown in Figure 5.

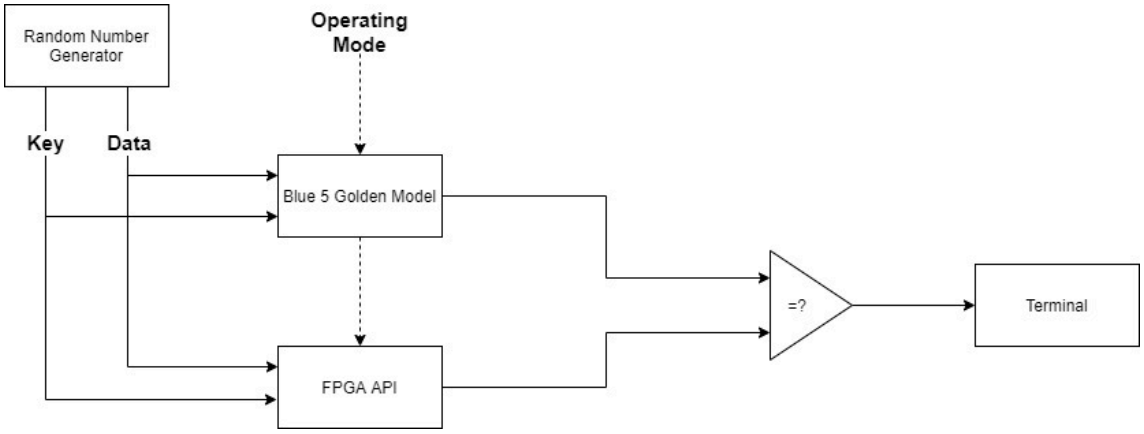


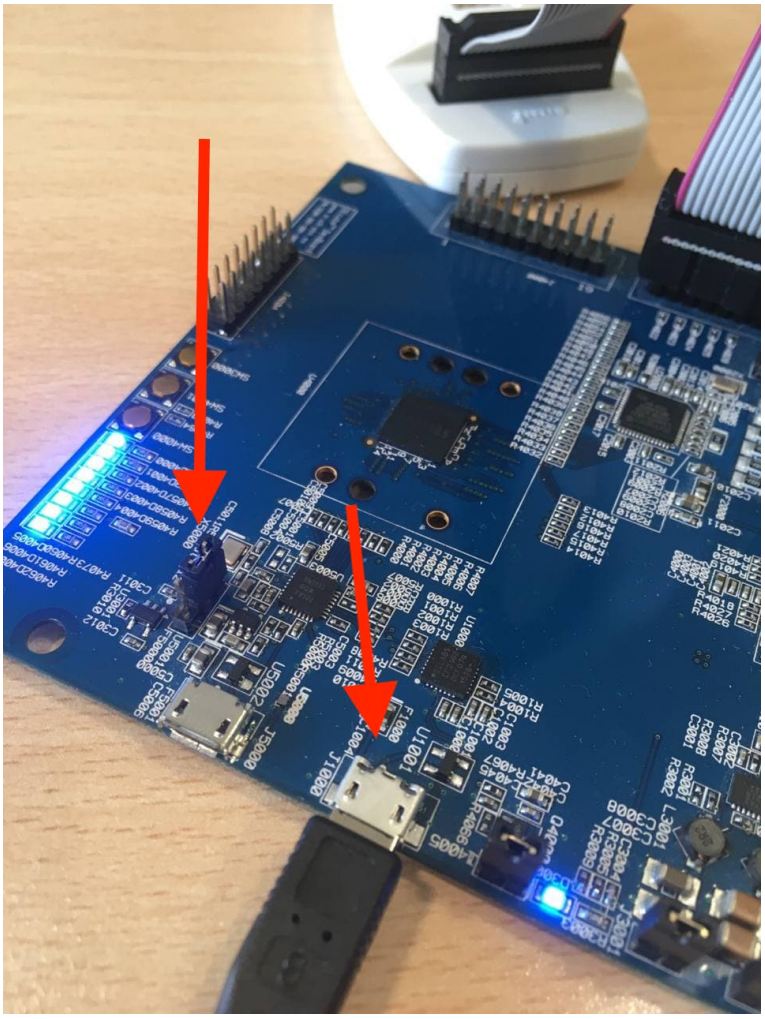
Figure 5: Validation test structure

To start with, you need to add the core to the working environment. Note that the project is working inside the architecture containing the IP Manager and the Data Buffer. Hence, you need a custom Device-Side **SE*cube*™** project opened in Eclipse, ready to work with the FPGA available on the **SE*cube*™** . To do so:

1. Get a copy of the 1.5.1 release of the **SE*cube*™** SDK[8]
2. Import the USBStick project in your IDE as explained in the general project wiki
3. Create a new Device-Side project by following the steps that are listed in the related documentation
4. To import the necessary files in your Device-Side project, select “File » Import…”, then “Filesystem” and press “Next”
5. Browse to the directory where the API libraries and the code for the AES core are located, which is called “code”
6. From this folder, import all the code contained in subfolders (src/, inc/, /syn and /test and relative subfolders) in the location you prefer. This includes the example “main.c” and all the files to work both with the IP Manager and the AES core

[8] https://github.com/SEcube-Project/SEcube-SDK, on the right side, in Section “Releases”, by downloading “SECube SDK 1.5.1”.

7. Configure both “Debug” and “Release” configurations from «Project » Properties » C/C++ Build » MCU GCC Compiler » Includes» and add the “Destination folder”
8. Save the changes to all files and build the project
9. Connect the DevKit to the PC using the J1000 UART connector (as in Figure below)
10. Move the jumper J3002 as specified in the **SE*cube*™** SDK wiki to power the board from the UART USB (as in Figure below)



11. Program the DevKit by right-clicking on it in the Project Explorer and selecting the Release binary under «Target » Program Chip»
12. If required, install the Bridge driver for the development board. On Windows, a COM device should appear under the device list
13. Detach all the cables from the development board (ST-Link or J-link programmer included), and reconnect only the USB cable to the J1000 USB connector. The LEDs should turn on. If not, change the position of the jumper as specified above
14. Open a UART connection with the board, setting the baud rate to 115200. Note that the device can also have a different name in your system. Please make sure to connect to the right UART receiver

15. A void screen should open now. Press the SW3000 button on the DevKit (Reset) to see program output

At this point, you can follow the instructions on the screen. Press 's' to start the programming of the FPGA. Next, the test suite will start. For each operating mode, the program will report the key length and the modality (enc/dec). Whenever the encryption of a modality is validated the terminal will show "Encryption successful" or "Decryption successful", depending on the test. At the first mismatch between the golden model and the Algorithm Under Test the program will stop showing the difference. There may be warnings telling that the creation of the initialization vector have failed, but they are expected and do not influence the test execution.

6 Synthesis Details

This Section is intended to offer to the reader the direct experience that the authors had in synthesizing the AES IP core for the **SE*cube*™** FPGA, with the hope they may be of any help for the users and developers that may want to enhance the project.

The synthesis of this design integrated within the IP Manager is not a trivial task for the Lattice Diamond synthesizer. By using the default options, the design requires around 75% of the LUTs. Filling too much the FPGA leads to a high wiring count, which results in a congested network. A high LUT count also means that the Place & Route tool optimization possibilities are rather limited. For these reasons, the critical path was mainly composed by routing, and not logic, resources. Since the synthesis process uses heuristic algorithms, the critical path was sometimes too close to the 60 MHz frequency target, or was even greater.

To solve this issues, the synthesis parameters have been fine-tuned to favour performance over area. In particular, for the synthesizer, the following options have been changed:

- the max fanout limit moved from 1000 to 250, to reduce the fanout and hence the cell congestion;
- the optimization goal changed from “Balanced” to “Timing”;
- the resource sharing was disabled, to reduce the overall congestion;
- the duplicated register removal feature is disabled, again to reduce the overall congestion;
- the adders do not use carry chains.

For the Place & Route tool, instead, the modified parameters are:

- the clock skew minimization is enabled;
- the congestion-driven placement algorithm is enabled;
- the congestion-driven routing algorithm is enabled;
- the iterations of the delay reduction pass have been set from 0 to 4;
- the number of routing resource optimization is changed from 0 to 6;
- the maximum number of routing passes in changed from 6 to 8;
- the number of placement iterations in modified from 1 to 4.

With these parameters the Place & Route phase is repeated multiple times, using different starting seeds every time, to have greater chances of obtaining valid results.

In the end, the amount of area required raised to about 85% with a critical path of $\sim$ 15 ns, a valid value with enough slack to amortize uncertainties.

All the critical paths lies either in the IP Manager or in the Data Buffer, suggesting optimization opportunities in its design.