Hardware and Embedded System Security - AES

1. The AES Algorithm

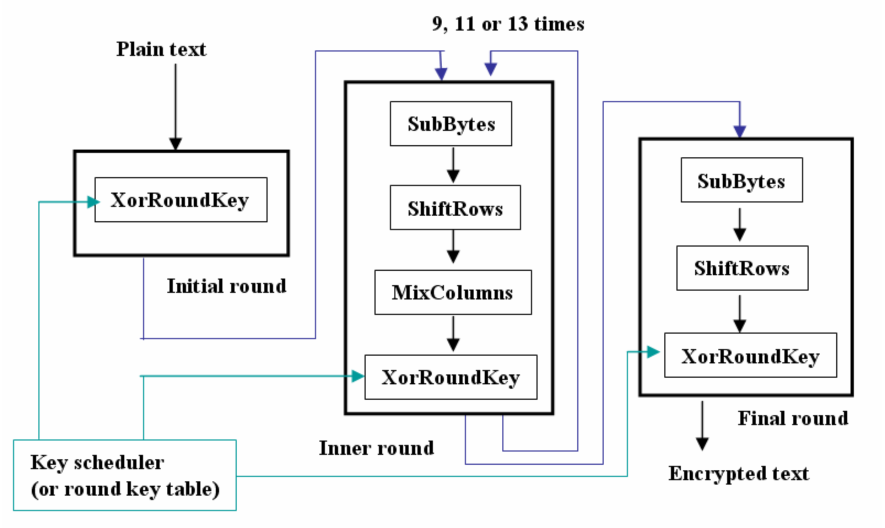
The AES algorithm is a symmetric block cipher based on an iterative structure. Its main characteristics are summarized in the list below:

Input block size:	128 bits
Output block size:	128 bits
Key sizes:	128, 192 or 256 bits
Number of iterations:	10, 12 or 14 (depending on key length)

AES has an iterative structure, consisting of a repetition of a round which is applied to the data block to be encrypted, for a fixed number of times. During the computation, the data is stored into a square matrix of 16 byte called state. The number of rounds is determined by the key size. For the three key sizes of 128, 196 and 256 bits, a number of 10, 12 and 14 rounds is required, respectively, plus an initial special round (called round 0). A round consists of a fixed sequence of transformations. Except for the first round (round 0, which consists only of the key addition) and the last round, the other rounds (internal rounds) are identical and consist of four transformations each. The first and last rounds are incomplete. The four round transformations are called *SubBytes, ShiftRows, MixColumns* and *AddRoundKey*:

- *SubBytes*: it is a non-linear byte substitution.
- *ShiftRows*: the rows of state matrix are rotated and the amount of rotation depends on the row index.
- *MixColumns*: it is a linear transformation of each column in the field $GF((2^8)^4)$ by a fixed multiplication; this operation is not performed in the last round.
- *AddRoundKey*: the round key is added to the state matrix modulo-2.

The *Key Schedule* produces the round keys from the initial secret key by using the same basic constituents of the round. An iteration of the Key Schedule is a combination of the following operations: a right rotation, a *SubBytes* operation, an addition of a byte constant, and a linear mixing phase. Further details about the algorithm can be found in the AES specifications.



2. The original architecture

The original design was developed by the Graz University and published in Transaction on Computers, vol. 52(4). It is here briefly explained as it is the version which has been used as the basis for the DDR design. We quote here the original description of the architecture.

The data unit is the main module of the architecture. The types of rounds that are executed are always the same. Consequently, the data unit is independent of the key size. The data unit has a highly regular structure. It consists of 16 instances of a so-called data cell and a certain number of S-Boxes. The more S-Boxes are used, the higher is the performance of the AES module. The standard version of the data unit has four S-Boxes. […]

To compute a normal AES round, the registers are rotated vertically to perform the Inv-/SubBytes and the Inv-/ShiftRows transformation row by row. In the first clock cycle, the Inv-/SubBytes transformation starts for row three. *Due to the fact that the implementation of the S-Boxes is pipelined the result of this Inv-/SubBytes transformation is stored in row* zero *two clock cycles later. Using the pipelined S-Boxes and the Barrel shifter the Inv-/SubBytes and the Inv-/ShiftRows transformations can be applied to all 16 bytes of the state within five clock cycles. In the sixth clock cycle of a normal AES round, the Inv-/MixColumns and the AddRoundKey transformations are performed by all data cells in parallel. Since the S-Boxes are not used by the data unit during the sixth clock cycle, they can be utilized by the key unit to perform the key expansion for the next round key. In order to compute the final round of an encryption or decryption, the Inv-/MixColumns transformation is omitted by the data cells in this clock cycle.*

Using the standard data unit, the minimal number of clock cycles that are required to perform an AES-128 encryption or decryption is 60, plus the cycles needed for the I/O operations, which depend on the interface width.

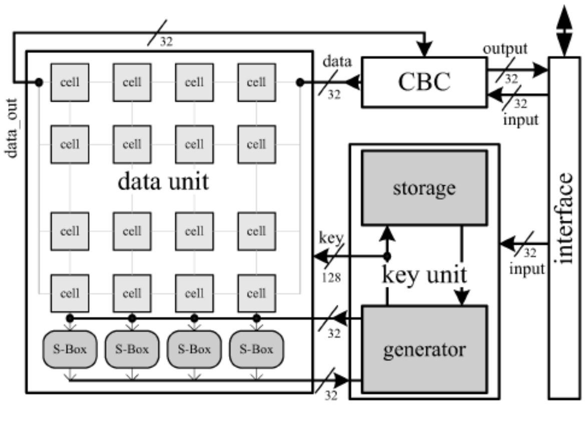


Figure 1. General structure of the original design.

3. The Double Data Rate design template

An effective protection scheme must be considered against fault attacks. Based on the fact that dual rail already provides some sort of hardware redundancy by itself, and the fact that duplicating some components may increase the sensitivity to DPA attacks, temporal redundancy looks the preferable solution.

However, temporal redundancy obtained by mere repetition of the process is trivial and can be implemented more easily at higher level (e.g., operating system or driver level);

moreover, the performance overhead is not negligible, although performance can be sacrificed in favor of security. Further observations reveal that the architecture can be highly performing and that the critical path is very short even without optimizations which might be expensive in terms of silicon area. As a result, the target clock frequency might be very high, but on the other hand this may not hold for the complete device, which may run at slower speed. This means that at each clock cycle, this AES architecture has all the values ready much earlier than the end of the clock cycle.

We can thus modify the architecture in order to perform two operations per clock cycle, instead of a single one. For instance, the S-Box might process two bytes per clock, exploiting both high and low duty cycles of the clock. This can be accomplished if the critical path is less than half the clock period and if the clock duty cycle is 0.5, which is quite reasonable.

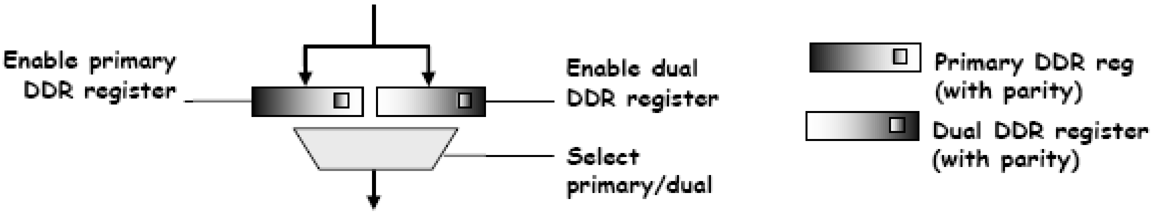
Hence, we implemented the same architecture using a double data rate (DDR) approach, in order to process two bytes of the same column at each clock cycle. This is achieved by coupling two different regular registers into a single DDR register: the clock signal is used to select the internal register that must be active (read from or written to).

This allows halving the number of clock cycles required by each round: 3 cycles only are required with the DDR approach, thus reaching the same performance of a faster and larger design. Please note that this technique allows saving the time that would be wasted while waiting, but it does not improve the efficiency of the architecture: on the contrary, if the device would run at its maximum frequency, using the DDR mechanism would make the performance slightly worse, due to the additional logic required to manage the different clock fronts.

With this approach, the same process can be recomputed a second time and the result verified against computation errors, with no significant performance hit with respect to the original design. In a few words, the DDR approach is used in this design to improve the usage of the pipeline stages over time and, as a consequence, the throughput.

However, the final objective is processing the same data twice, in order to provide error detection by comparing two copies of the result. This can be done by actually executing the process twice, or by performing the inverse process on the result. The implemented solution is the former, since it allows using the existing pipelining and does not require the additional decryption logic, which may not be included in the device. Both solution, however, require an additional register to store either a backup copy of the original data, or the result for the comparison after the inverse operation is complete. Hence, each DDR register in the state will have a twin, acting as backup. This, however, does not hold for the substitution layer, where the data is not stored, but only computed and moved immediately for further processing.

The final dual DDR register can store two copies of two bytes at each time and it is shown in figure below. It must be observed that the additional parity bit it is shown just as an example of a register augmented with an error parity code; obviously, it is required only if the parity code is used as an additional error detection countermeasure.



High-level Structure

The main core is made of the following components:

- DataUnit: the main encryption logic. It has a structure similar to Figure 1, but each column is transformed by using the DDR template described below and shown in Figure 4 and Figure 5. Please note that the four S-Boxes are shared with the key scheduler, in order to save area.
- Key Schedule: the logic responsible to generate the round keys, which are used at each iteration and computed from the secret key. The currently supported key size is only 128 bits. The key scheduler uses the S-boxes of the DataUnit at the last cycle of each round in order to compute the next round key.
- Control Unit: the logic responsible for all the control signals within the core IP.

IO interface

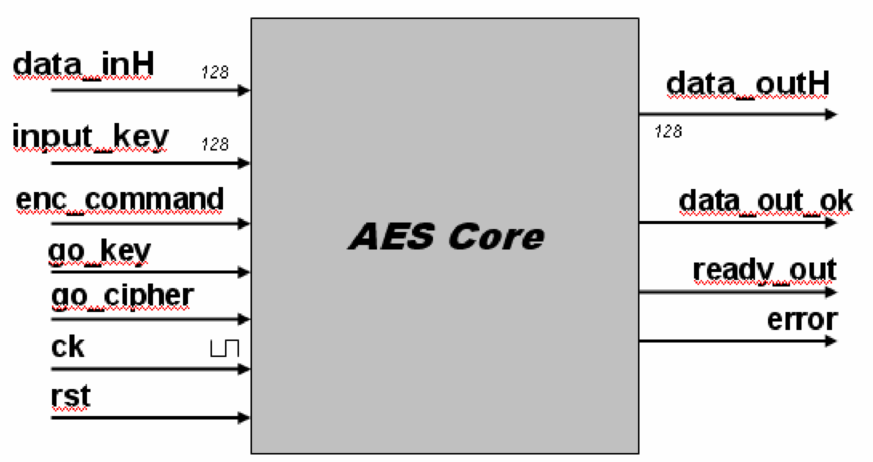


Figure 2: AES core port description.

Port	**In/Out**	**Width**	**Description**
Data_inH	in	128	Plain text to encrypt (Ciphered text to decrypt).
Input_key	in	128	Key for encryption/decryption
Enc_command	in	1	Allows choosing between encryption (**low**) or decryption (**high**)
Go_key	in	1	Initializes the secret key; active **high**
Go_cipher	in	1	Start the encryption (decryption); active **high**
Ck	in	1	Clock signal
Rst	in	1	Asynchronous reset for the whole core; active **low** (can be configured in global parameter file).
Data_outH	out	128	Encrypted (decrypted) result
Data_out_ok	out	1	Signals valid data at the output (active **high** for 1 clock cycles)
Ready_out	out	1	Signal availability of the IP core (active **high**; low for busy)
Error	out	1	Signal showing if an error occurred; active **high**.

Command Timing

Device reset: Reset signal is activated for one clock cycle to reset internal state and other registers.

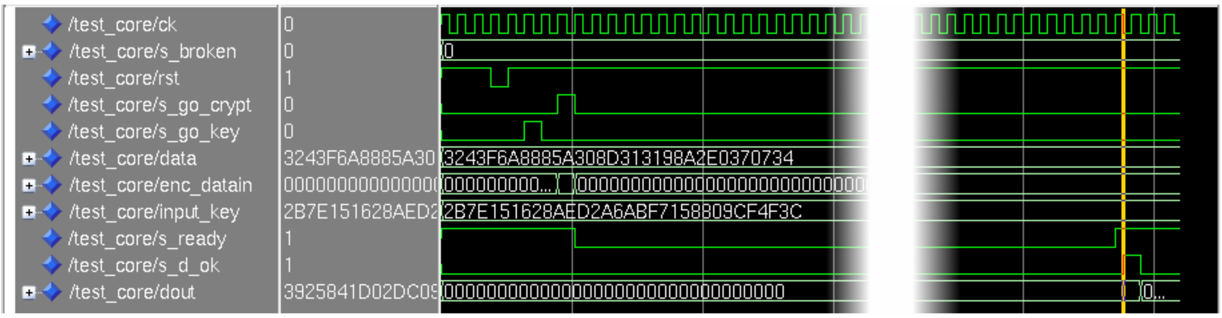
Key setup: Secret key is sent at the *Input_key* input port, the *enc_command* signal is set accordingly, and the *go_key* signal is set active for **one** clock cycle. If the unit has decryption capabilities and the decryption command is set, computation starts in order to compute the last round key: the unit goes into *busy* state and it is not available for user input. If encryption is chosen, then the device is immediately available for computation.

Data loading: Data is set at the *data_inH* port in **one** clock cycle. During the loading phase, the **following signal** are set or activated: *enc_command* (consistently with the provided secret key), and *go_cipher* command; the device enters the *busy* state.

Computation: The device starts the encryption (or the decryption). The whole process requires 60 clock cycles for the regular architecture (i.e., 6 cycles per round). The DDR implementation needs only 3 clock cycles per round; however, each round is computed twice for error detection purposes, thus a complete round is again computed in 6 clock cycles.

Data validation: At the end of each round, intermediate values are implicitly validated: this is done concurrently with the normal computation and it is transparent to the external interface. When an inconsistency is detected, the error signal is activated and the computation continues nonetheless.

Data output: Data is sent at the *data_outH* port, requiring only **one** clock cycle to give the whole result; at the same time, *data_out_ok* output signal is set active to identify that data is significant.



```
/test_core/ck            0
/test_core/s_broken      0
/test_core/rst           1
/test_core/s_go_crypt    0
/test_core/s_go_key      0
/test_core/data          3243F6A8885A30
/test_core/enc_datain    000000000000000
/test_core/input_key     2B7E151628AED2
/test_core/s_ready       1
/test_core/s_d_ok        1
/test_core/dout          3925841D02DC09
```

Figure 3: Signal timings for the DDR design.

The DataUnit

The ability to process two data per clock cycle reflects in some small but significant changes to the architecture. The original column design is schematically depicted in Figure 4. It is easy to identify the architecture stages that make the pipeline: the linear (permutation) layer, one for each byte, and the substitution (non-linear) layer, one for each column to save as much area as possible.

The first change in the design is merging two regular registers into a DDR register; the area increases slightly, due to some logic needed to drive signals during different clock states. Now there are two parallel data path during each clock cycle; hence, some combinatorial logic might be shared between two DDR registers. This might be applied to the substitution layer, where the computation is independent of the specific row, but not to the permutation layer: in fact, sharing the *MixColumns* logic would require some logic to set the coefficients properly. Due to the fact that the linear layer is relatively small when compared to the substitution stage, the benefits would be very limited: hence, no sharing was allowed within a column. In Figure 5 a detailed representation of the design of a single column in DDR mode is shown. It is easy to identify: the first permutation layer, relative to the first two rows, and the dual DDR register; the second permutation layer and the second dual DDR register bank, relative to rows three and four; the *SubBytes* implementation. The barrel shifter for row rotation and the final DDR register bank are not shown.

By comparing directly the two figures, it can be seen that the major changes are

- the regular registers merged into a DDR register,
- the dual register bank acting as backup (not in the SBox),
- the logic required to manage DDR and dual registers.

On the other hand, the computational part of the architecture is unchanged. Thus, each linear functional block has still a vertical input V, a horizontal input H, the key input, and miscellaneous input control signals. Moreover, for each linear block that was implemented in the basic design, an almost identical block is implemented in the DDR design. The only difference is an additional latch that is used as a synchronization layer.

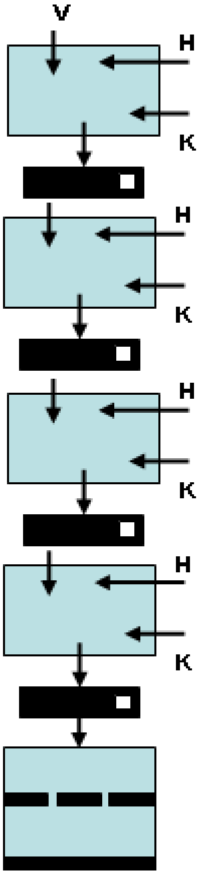


Figure 4: Column design in Single Data Rate mode

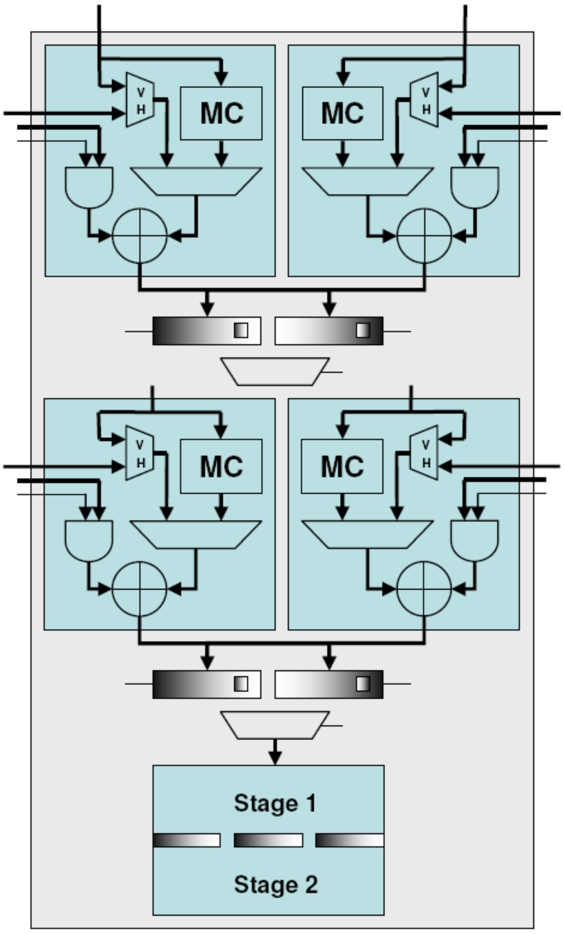


Figure 5: Column design in Double Data Rate mode

To resume, if we analyze the DDR data path depicted in Figure 5, we find from top to bottom:

- two linear blocks (light blue squares), implementing the MixColumns, the key addition, and the input selection; they correspond to the linear layer of the two first rows;
- the main and the backup DDR registers, which are alternately active during the main and the secondary round, respectively; the only exception is during the loading phase, since they are both active in order to store the initial input. Each DDR register stores

the data of the first two rows and it may optionally include the flip flops dedicated to parity storage;

- the multiplexer controlling which value must be propagated (the main or the backup);
- two linear blocks, corresponding to the linear blocks of rows 3 and 4;
- another pair of DDR registers, devoted to the storage of the last two rows of the state, and the corresponding selection multiplexer;
- the pipelined S-Box, where the internal pipeline registers also follow the DDR template; there is no backup register in this case, since there is no need to save the temporary computed values.

The control unit

In the DDR design, a regular round is complete in half the time; thus, two rounds can be computed in the time that was originally required by a single round. This applies to the generic and final round, while the initial phase (loading and initial key addition) are performed only once and the result used both for primary and backup computation. The main FSM is shown in Figure 6: the number of states is significant, but they can be grouped intuitively:

- Idle and/or loading (state IDLE): the device is waiting for the plain text and the start command; when the command is issued, the data at the input is added with the key currently saved and stored into the internal state of the architecture. The input interface is 128-bit wide, hence the loading phase is completed in a single clock cycle.
- Key computation (K*n*): the last round key is directly computed to allow fast decryption; these states are not required when decryption is not implemented and are used only for key unrolling.
- Primary main round (R*n*A): the regular round is computed for the primary data registers; R1A and R2A are used to compute the SBox substitution, while R3A is used for the linear permutation layer (*MixColumns* and *AddRoundKey*).
- Backup main round (R*n*B): as the primary main round, but it operates on the backup registers. The validation of the computation is done at this stage, when both the primary and the backup state have stable values. If an error is detected, then the error signal is raised.
- Primary final round (F*n*A): as the main round, but it lacks the *MixColumns* operation, i.e., in the last transition only the key addition is computed.
- Backup final round (F*n*B): as the primary final round, but it operates on the backup data registers.
- Output result (O1): the data is sent at the output and the proper signal is raised to identify valid data.

An additional finite state machine is also defined: it is started by the main FSM and allows switching some control signals at the falling clock edge. This is required since a few signals must change at each front to support the DDR mechanism. However, this has no effect on the way the commands are issued to the cryptographic IP: the synchronization is on the raising edge. On the other hand, the output ports update their value at the falling clock edge, to stabilize the values coming from the DDR registers.

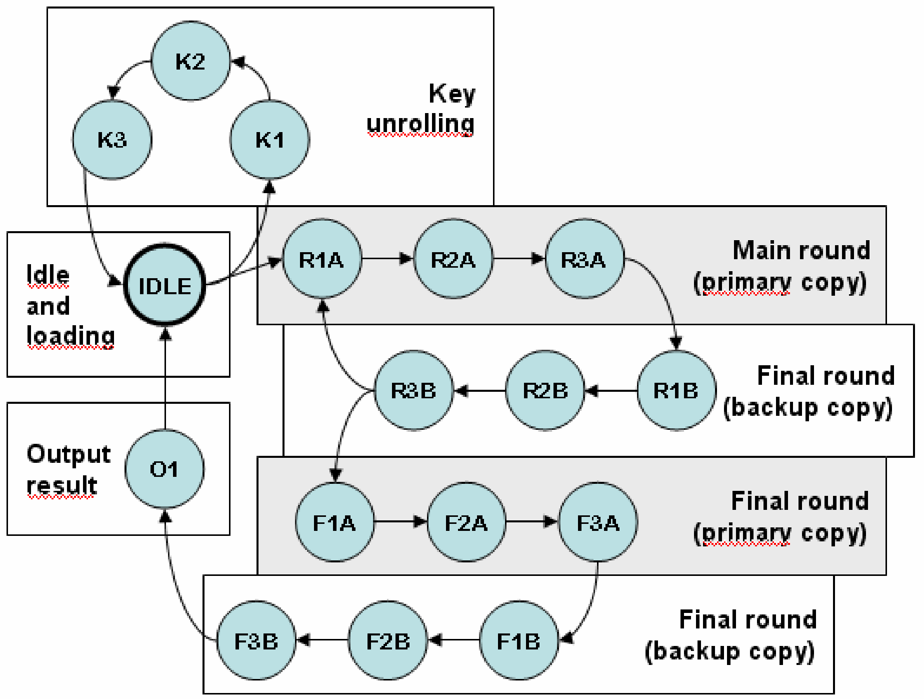

K2
K3
K1
Key unrolling
Idle and loading
IDLE
R1A
R2A
R3A
Main round (primary copy)
R3B
R2B
R1B
Final round (backup copy)
Output result
O1
F1A
F2A
F3A
Final round (primary copy)
F3B
F2B
F1B
Final round (backup copy)

Figure 6: Main FS control machine

4. Attack model

The goal of this work is to validate the countermeasure based on DDR against fault attacks. In order to do so, you have to identify the weaknesses of the design and propose possible solutions.

The fault model that has to be considered depends on the fact the attack must replicate the capabilities and the objective of a real attacker. This means that it is **not allowed** to introduce stuck-at faults in the main encryption data path: these faults are destructive and it is often hard to abstract the error in order to analyze the result. Similarly, you **cannot** introduce any fault into the key scheduler, which is considered to be shielded and protected with dedicated schemes. You **can**:

- Inject bit-flip faults into the encryption data path (i.e., the *data unit*): you are allowed only a single “shot”, but you can make it last for several cycles; the error should span over several bits, up to a whole byte (i.e., no single-bit faults!)
- Inject **bit-flip** or **stuck-at** faults in the controller; you are allowed *preparatory* faults, i.e. faults that do not directly reveal the key but are needed in order to make the following injection successful; the granularity allowed in the controller is much finer, which means that you can alter single signals.
- Alter the key signals outside the key scheduler, with the same rules as used when targeting the encryption data path (i.e., multiple bit-flips).

Once you have proved the feasibility of the attack, you should propose some countermeasures addressing the security flaw.

5. Tasks

1. Validate the behavior of the architecture in order to understand how it works, in particular the timings at the interface. Do this step before and after the synthesis process.
2. Clearly identify and describe the protections against fault attacks that are implemented in the architecture. Keep in mind how the DDR template works and that it is applied only to the encryption data path; the key related logic and the controller are not directly affected by the DDR mechanism.
3. Highlight the weaknesses and describe how they might be exploited: use a fault model as much realistic as possible. For instance, propose attacks that do not rely on the injection of a particular value (but if you find them, they are still worth being mentioned!)
4. Implement the attack by modifying the VHDL code in order to emulate the fault; validate the attack by using simulations before and after synthesis with Xilinx ISE.
5. Embed the cryptographic core in a design developed with the Xilinx EDK tool. Write a simple program that uses your new IP and verify that everything works as expected.
6. Choose one or a few attacks and implement them: replace the architecture embedded in EDK with the faulted one in order to emulate the attack and validate with simulations.
7. Implement the system on the programmable board!