AES-256 RTL

Verification document

Author:

Aleksandar Lilic

Table of Contents

1. Test environment

Testbench is composed of four distinct components:

1. UVM testbench
2. SystemVerilog wrapper for the VHDL design
3. Interface and concurrent assertions
4. Coverage collector

Next four chapters cover the organization of each of these components, their role in the environment, and some relevant implementation details. Figure 1 shows the structure of the testbench. Dotted lines and blocks (subscriber and reference vectors) are not used in all tests.

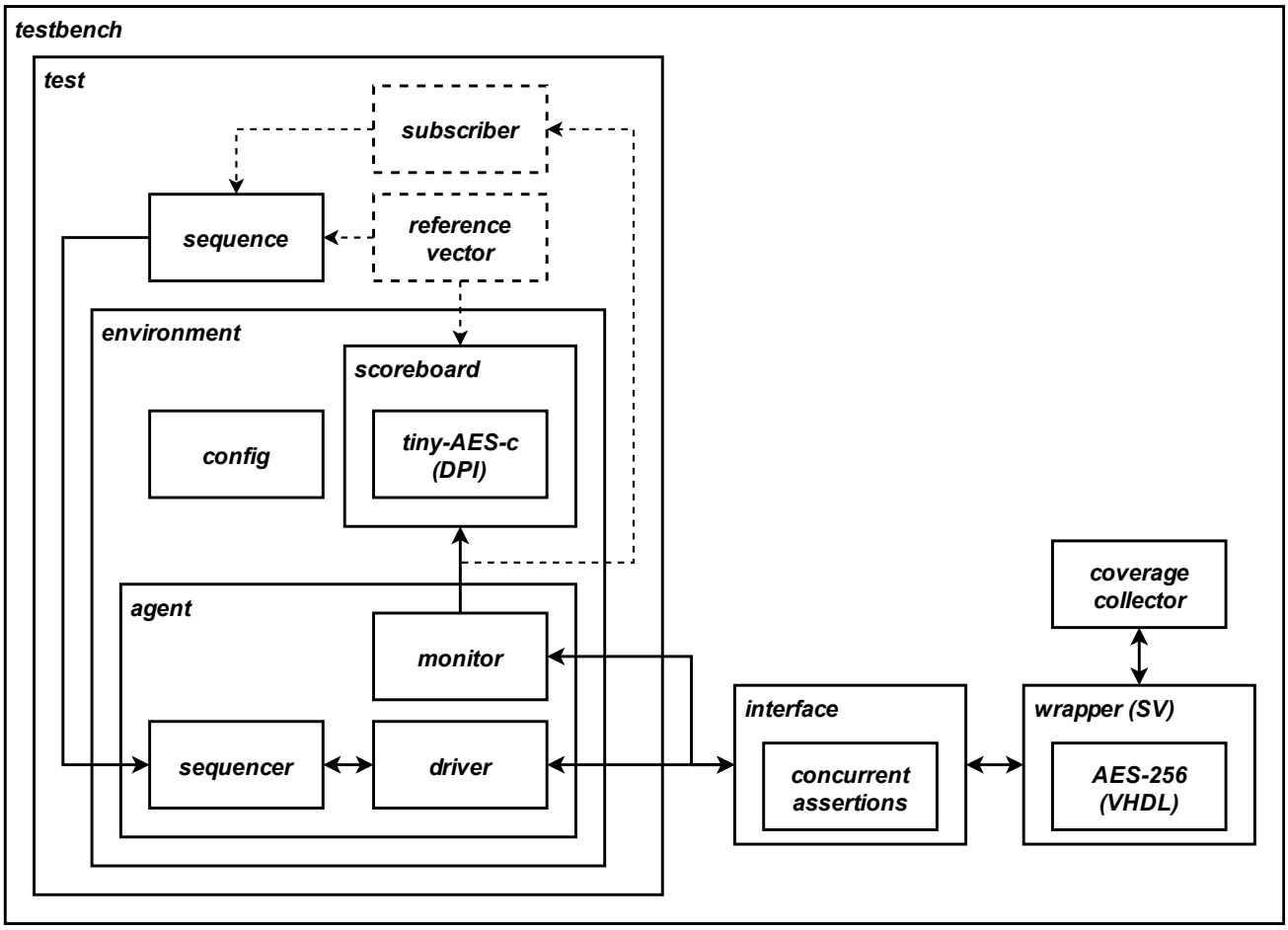

Figure 1 – Test environment

1.1. UVM Testbench

Testbench uses UVM methodology for the functional verification of the AES-256 VHDL design (referred to as DUT in the rest of this document). UVM testbench, and implemented UVM objects and components are briefly described in the following chapters.

Sequence Item

```
class aes256_seq_item extends uvm_sequence_item;
```

The UVM sequence item **`aes256_seq_item`** is used to define all transaction details needed by the test environment. Sequence class members, shown in Listing 1, can broadly be divided in three categories:

1. Design inputs
2. Design outputs
3. Timing relationships (between different requests and waiting periods)

Listing 1 – Sequence item class members

```
// design inputs
bit key_expand_start = 0;
rand bit [`MATRIX_KEY_WIDTH-1:0] master_key = 0;
bit next_val_req = 0;
rand bit [`MATRIX_DATA_WIDTH-1:0] data_in = 0;
// design outputs
bit next_val_ready;
bit [15:0] [7:0] data_out;
`ifdef HIER_ACCESS
bit [0:`N_ROUNDS-1] [`MATRIX_ROUND_KEY_WIDTH-1:0] key_exp_round_keys;
`endif
`ifdef VIVADO_RND_WORKAROUND
sweep_type_t sweep_type;
`endif
// timing relationships
rand byte unsigned key_expand_start_pulse;
rand byte unsigned key_expand_start_delay;
rand byte unsigned next_val_req_pulse;
rand byte unsigned next_val_req_delay;
bool_t wait_for_key_ready = TRUE;
bool_t wait_for_enc_done = TRUE;
bool_t key_exp_wait_for_loading_end = TRUE;
```

Desing inputs and outputs are self-explanatory in their purpose. Expanded round keys are only defined if hierarchical access to the modules is allowed during the compile stage. It's also the only class member that's not registered in the factory due to its length and also generally not being required when tracking the results during long simulations. Since it's not registered in the factory, **`do_copy()`** method is inherited from parent class and implements copying of the round keys.

Timing relationship themselves can be divided into two groups: wait flags and requests. Wait flags are defined as custom bool type, and specify if the testbench should wait on DUT's response before continuing. Two **`*_pulse`** members specify how many cycles the request should stay high while two **`*_delay`** members specify how much should the request be delayed by the driver. These timing relationships serve two main purposes: to test that DUT can recover from interrupts at any point of the expansion, encryption and loading, and that pipelined requests (loading while encrypting) can work independently no matter when they are requested. Random members are constrained within the expected ranges, with two of the constraints being **`soft`** to allow for overrides when the sequence item is instantiated, as show in the Listing 2.

Listing 2 – Sequence item constraints

```
constraint c_key_expand_start_delay { key_expand_start_delay inside { [0:32] }; }
constraint c_key_expand_start_pulse { soft key_expand_start_pulse inside { [1:8] }; }
constraint c_next_val_req_delay { next_val_req_delay inside { [0:2*LOADING_CYCLES] }; }
constraint c_next_val_req_pulse { soft next_val_req_pulse inside { [1:8] }; }
```

Special mention is also required for the **`VIVADO_RND_WORKAROUND`** define. Vivado version used in the project, 2022.3, has a bug in the random number generator, where the most significant nibble (4-bits) of the 32-bit value can never be randomized to have values in the range from 4 to 7. Furthermore, it appears that random number generator is internally generating 32-bit numbers and then concatenates them before returning the requested length, since this nibble issue occurs at each 32-bit boundary of any value longer than 32 bits. For example, 128-bit plaintext has four 32-bit values concatenated and therefore has four nibbles that have the randomization issue. To deal with this, sequence item implements function **`post_randomize()`** where it generates new random values needed to fix the original randomization. As the randomization fails only for three least significant bits of the nibble, only those three are "re-generated". The **`sweep_type`** class member is also defined only for the workaround case, the purpose of which is described in the Sweep sequence chapter. In short, sweep sequence doesn't randomize inputs, so sequence item shouldn't override them. As this is a Vivado specific workaround, it can safely be disabled if the testbench is run using different simulation tool.

Sequences

Testbench implements three separate sequences, governed by the type of test which uses them. The verification of the DUT is critically dependent on the sequence(s), and more specifically sequence's **`body()`** task as it's a central mechanism for stimulating the DUT. Next three chapters go into more detail about the three used sequences.

Standard sequence

```
class aes256_sequence extends uvm_sequence#(aes256_seq_item);
```

This sequence is the cornerstone of the test environment and it's used in five out of eight tests. It follows a simple two-part approach of key expansion followed by encryption. First, a sequence item containing key expansion information is started with appropriate settings with aspect to wait periods and the DUT's feedback. Second, after successful key expansion, encryption is initiated, with its own rules for the available settings. The sequence has two class members that specify how many keys and how many plaintexts for each key it should send (Listing 3), implemented as two nested loops (Listing 4). At the end of the last encryption of the last key, sequence implements a short idle period.

Listing 3 – Sequence class members

```
rand int unsigned number_of_keys = 1;
rand int unsigned number_of_plaintexts = 2;
bool_t wait_for_key_ready = TRUE;
bool_t wait_for_enc_done = TRUE;
bool_t key_exp_wait_for_loading_end = TRUE;
rand byte unsigned wait_period_at_the_end = 10;
rand exp_delay_mode_t exp_delay_mode = EXP_RANDOM;
rand enc_delay_mode_t enc_delay_mode = ENC_RANDOM;
```

Listing 4 – Sequence class two nested loops

```
for (key_cnt = 0; key_cnt < number_of_keys; key_cnt++) begin
    // ...
    for (pt_cnt = 0; pt_cnt < number_of_plaintexts; pt_cnt++) begin
        // ...
    end
end
```

By varying three Boolean variables and constraining the rest within specific ranges, every intended scenario can be covered. Five tests that are using this sequence are going into more details on how these class members are used.

Sweep sequence

```
class aes256_sequence_sweep extends uvm_sequence#(aes256_seq_item);
```

Sweep sequence doesn't rely on randomizing DUT inputs but rather it aims to cover specific cases that might cause DUT to cause overflow or underflow. Values the sequence generates 'sweep' over the entire value range by toggling one bit at a time. Value that is under sweep starts from 0 and then for each iteration toggles next index bit to 1. After all bits are toggled to 1, it starts again from index zero, now toggling values back to 0, as show in Example 1 for the 3-bit value. Sequence implements two sweep types: key sweep and plaintext sweep. For the key sweep, each time a key is expanded, one plaintext with value `'h0` is also sent to the DUT to finish the encryption process. For the plaintext sweep, a single master key with value `'h0` is used and the plaintext sweep is done only for that key. The total number of iterations, and therefore items sent, is twice the bit length of the value under sweep.

Example 1 – 3-bit value sweep

```
001 -> 011 -> 111 -> 110 -> 100 -> 000
```

Reference vectors sequence

```
class aes256_sequence_ref_vectors extends uvm_sequence#(aes256_seq_item);
```

Main purpose of the reference vectors is validation against the NIST supplied test vectors. Additionally, it allows for specifically targeting some value(s) of interest, like with manual entries in the vector file or with vector/trace exports from other tools. Another advantage is that the reference vectors don't require DUT and test recompile when vectors are added, changed, or expanded. Checker method used for reference vectors is described in the Scoreboard chapter.

Instead of relying on the randomization of the sequence item, this sequence simply reads the key and the plaintext from the vectors file and then sends assigns these values to the sequence item instance to be sent to the driver. All timing-related parameters are fixed to the lowest possible values. Additionally, if the master key doesn't change between two subsequent entries in the vector file, key is not expanded again and only the encryption is started.

Special case vectors are MCT vectors provided by NIST. Format of these vectors is the same as for other vectors, however the mode of operation is different as one MCT entry provides master key and the only the ***first*** plaintext. Upon producing the first ciphertext, that same ciphertext is fed in as the next plaintext, in essence chaining the operations. This is often referred to as the reactive sequence as it produces next item based on the output of the DUT. To facilitate collecting the DUT outputs, sequence instantiates one subscriber which is in turn subscribed to monitor's analysis port. This communication method is further discussed in the Subscriber chapter. Sequence then waits on the event trigger form the subscriber to get the current ciphertext. The ciphertext specified in the MCT entry is the output of 1000th chained operation. MCT mode is disabled by default and can be enabled with `$test$plusargs("MCT_VECTORS")` during runtime.

Driver

```
class aes256_driver extends uvm_driver #(aes256_seq_item);
```

Driver simply abstracts away the details of sending the data defined in the sequence item over DUT ports. When UVM run phase starts, driver first resets the DUT and then waits for the next item from the sequencer. Upon receiving the item, it can either send new key expansion or encryption request, or an inactive item (i.e. passively waiting for the DUT). Driver also implements waiting mechanism for the DUT's response that the previously requested action has finished.

Monitor

```
class aes256_monitor extends uvm_monitor;
```

Monitor's run phase consists of five concurrent processes, observing and collecting the data from the DUT. Five processes, started with a **`fork ... join_none`** block, are:

1. Key expansion: collects DUT inputs when the expansion starts and collects the expanded keys when the expansion ends.
2. Key expansion timeout: Watchdog thread that ensures the key expansion doesn't take longer that expected. Cognizant of the fact that new key expansion might be requested even when the current expansion didn't finish
3. Encryption: collects DUT inputs when the encryption starts
4. Encryption timeout: Watchdog thread that ensures the encryption doesn't take longer than expected. Cognizant of the fact that new encryption or key expansion might be requested even when the current encryption didn't finish
5. Loading: Collects DUT outputs on the 8-bit bus. Copies current sequence item when encryption has finished as that original sequence item will be used for the next encryption process. Timeout check is integrated in the data collection, as monitor has to receive exactly 16 bytes in the next 16 cycles. Loading can be interrupted by the new key expansion request, be it while waiting for loading to start or during loading process. Loading can't be interrupted by the next encryption request as this is used for pipelining the two operations and is a valid (and desired from the performance standpoint) scenario. Upon receiving 16 bytes, this process sends the received item over the analysis port to all subscribers.

Agent

```
class aes256_scoreboard extends uvm_scoreboard;
```

Agent's role in the testbench comes down to the build and connect phase. During build phase, agent instantiates monitor and fetches configuration class from the factory. If agent is configured as active, it also instantiates driver and sequencer during build phase and later connects the two during connect phase. Sequencer is the only component that's used unchanged from the UVM version as there was no need to modify it. All four class members are show in the Listing 5. Since this is the only agent in the testbench, making it inactive wouldn't make much sense from the functional point of view as there is nothing else driving the DUT. On the other side, if the DUT is integrated as a part of a larger design, it would make sense to make the it passive and reuse the monitor for the purpose of collecting the inputs and outputs.

Listing 5 – Agent class members

```
aes256_driver driver_1;
aes256_monitor monitor_1;
uvm_sequencer#(aes256_seq_item) sequencer_1;
aes256_cfg cfg;
```

Scoreboard

```
class aes256_scoreboard extends uvm_scoreboard;
```

In order to ensure functional correctness of the generated ciphertexts by the DUT, a known good ciphertext is needed to compare against. Testbench, through scoreboard, uses two approaches:

1. C model integrated over DPI – default, supported by every test
2. Reference vectors – optional, supported only by the reference vectors test

C model is based on the open source *tiny-AES-c* implementation, set up to work in the AES-256 ECB mode. Model is then expanded with a single DPI function that exposes AES functionality to the SystemVerilog testbench. DPI function requires four arguments, all four in the form of pointers. Two inputs are AES-256 key and AES-256 plaintext, while two outputs are 15 AES-256 round keys and AES-256 ciphertext, as shown in the Listing 6.

Listing 6 – DPI AES-256 function

```
DPI_DLLESPEC
void aes_dpi(uint8_t* key, uint8_t* plaintext, uint8_t* round_keys, uint8_t* ciphertext)
```

Second checker method is via reference vectors as described in the Reference vectors sequence chapter. Reference vectors can best be viewed as an additional checker layer, as they and the C model are independent from each other. DUT's output value is first compared with the C model, and then, if provided, with the reference vectors. This also allows for special case vectors where ciphertext is specified as "NONE" in the vectors file, so the comparison is not actually performed with the vectors, but only with the C model. While this deviates from the original reference vector definition since it doesn't actually provide reference output value, it's useful when trying to cover specific cases while relying on the C model for output comparison. This mode is disabled by default and can be enabled with `$test$plusargs("ALLOW_VECTOR_CHECKER_NONE")` function during runtime. For the MCT vectors, first 999 ciphertexts are compared only with the C model, while 1000th ciphertext goes through usual compare with both model and the entry in the vector file.

Environment

```
class aes256_env extends uvm_env;
```

Environment ties together agent and the scoreboard and, crucially, connects the monitor's analysis port with the scoreboard's analysis implementation in order to facilitate sending collected DUT inputs and outputs to scoreboard for checking.

Configuration

```
class aes256_cfg extends uvm_object;
```

Configuration class is mostly barebones as the DUT doesn't have any modes of operation or other settings that might be required to set up before running the test. The only thing it sets up is agent's mode between active or passive. Even this is more futureproofing of the testbench rather than a requirement since setting agent in passive mode would mean that there are no drivers connected to the DUT.

Subscriber

```
class aes256_subscriber extends uvm_subscriber #(aes256_seq_item);
```

Subscriber is used for the sole purpose of creating a reactive sequence for the NIST MCT vectors. It has analysis implementation and a UVM event which it uses to trigger the sequence item. During the test connect phase (discussed in Reference vectors test chapter), subscriber's analysis implementation is connected to the monitor's analysis port, the same one used by the scoreboard. Upon receiving new item from monitor, subscriber gets the handle to the sequence item and triggers an event, enabling the sequence to continue.

Tests

There are a total of nine test classes: one base class and eight child test classes expanded from the aforementioned base class. Next nine chapters go into relevant details of each test. Testbench top chapter deals with making all tests available while Appendix A: Running tests: Make-based flow details test selection for the simulation.

Base test

```
class aes256_test_base extends uvm_test;
```

Base test is foundation for all other tests, it therefore implements functionality needed by each of the child tests. During build phase, base test creates one configuration and one environment class instance. Additionally, it provides two utility functions that handle number of items which are used for sequences and scoreboard later in the inherited tests.

Listing 7 – Base test class members

```
aes256_cfg cfg;
aes256_env env;
int unsigned num_keys = 0;
int unsigned num_pts = 0;
```

Smoke test

```
class aes256_test_smoke extends aes256_test_base;
```

Smoke test, consistent with its name, provides the simplest possible scenario for the DUT: it first expands one key, then encrypts one word, waits for the loading to send all 16 packets, adds short idle period at the end and finishes the test. Before the test start, it also prints out the topology of the entire UVM testbench. Since other tests don't change the underlying components, topology is the same for all other tests. Entire output waveform is show in the Appendix B: .

It's worth discussing the overall structure of the smoke test, as the same pattern is mostly followed for all other tests. After initially waiting for 10 time units (defined as 1ns in the testbench), test creates one sequence item instance, sets the number of keys and plaintexts for the sequence and then randomizes sequence with a set of constraints. Number of keys and plaintexts, and sequence constraints are particular to the test class since each class targets a different scenario for the DUT. After ensuring that randomization was successful, test sets the number of expected items for the scoreboard and finally sends the sequence via sequencer. When sequence sends all specified sequence items, test ends by dropping the objection for the run phase.

Max throughput test

```
class aes256_test_max_throughput extends aes256_test_base;
```

Max throughput test sends new request with minimum possible delay between operations. Key expansion is done, followed by encryption request in the next cycle. Encryption is that repeated 100 times. Each encryption is fully pipelined with the loading process, upon finishing the current plaintext encryption, DUT is requested to start loading of the current ciphertext and encryption of the next plaintext. In the normal operation, this is expected to be the most commonly used scenario. Test requests 50 keys total, and the aforementioned 100 plaintexts for each, totaling 5000 ciphertexts.

Key generation test

```
class aes256_test_key_gen extends aes256_test_base;
```

Key generation is similar to the max throughput except that it targets specifically key expansion logic. Test sends 4000 key requests and one plaintext for each, totaling 4000 ciphertexts.

Delays test

```
class aes256_test_delays extends aes256_test_base;
```

Delay test is structured as two nested loops that go over all available delay modes as shown in the Listing 8. The goal of the test is to ensure the DUT is responding as expected in various points after previous operation has finished, which is especially important for encryption and loading processes as they can be pipelined. Test doesn't interrupt any of the operations, but it rather waits for the completion and then makes new request.

Listing 8 – Delays test nested loops structure

```
for (int i = 0; i < exp_mode.num(); i++) begin
    // ...
    for (int j = 0; j < enc_mode.num(); j++) begin
        // ...
    end
end
```

Interrupts test

```
class aes256_test_interrupts extends aes256_test_base;
```

Interrupts target each of the four possible scenarios where request of the higher priority interrupts the ongoing process. As each of these requests act like a soft reset for the (part of) the design, test ensures that DUT is able to appropriately respond regardless of when and which interrupts comes in. Four test cases are:

1. New key expansion request while the current key is still being expanded
2. New key expansion request while the encryption is running
3. New encryption request while the current encryption is still running
4. New key expansion request while the loading is still sending data

Each of the four test cases first send a sequence containing the particular interrupt, followed by one 'common case' sequence to ensure DUT is still responding as expected. This process of interrupt followed by common case is repeated 100 times for each of the test cases.

Sweep key test

```
class aes256_test_sweep_key extends aes256_test_base;
```

Before starting the single sequence, test first sets the sweep type to 'key'. Secondly, it sets the number of expected items to 512, twice the length of the AES-256 key bit width, since the sequence sweeps in both directions, as described in the Sweep sequence chapter.

Sweep plaintext test

```
class aes256_test_sweep_pt extends aes256_test_base;
```

Similarly to the key sweep, before starting the single sequence, this test sets the sweep type to 'plaintext' and the number of expected items to 256, twice the length of the AES-256 plaintext bit width.

Reference vectors test

```
class aes256_test_ref_vectors extends aes256_test_base;
```

Test uses two file descriptors as it opens the same vector file twice, and sends the one file pointer each to the sequence and the scoreboard. Additionally, test defines one subscriber needed for the reactive sequence for MCT vectors. MCT vectors and 'NONE' option are incompatible, and the test aborts simulation if those two options are defined together. Scoreboard infers number of expected items from the file entries.

Each of the available CSV files are run as a separate test instance:

1. vectors_base – simple three-entry file to show the example usage
2. vectors_edge_cases – used to cover corner case scenarios like min and max values of the inputs and outputs of modules' and LUTs
3. vectors_NIST_examples256 – test vectors collected from various official NIST documents
4. vectors_NIST_ECBGFSbox256 – NIST provided test vector for encryption Sbox
5. vectors_NIST_ECBKeySbox256 – NIST provided test vector for key expansion Sbox
6. vectors_NIST_ECBMCT256_p1 – NIST provided MCT vector, part 1
7. vectors_NIST_ECBMCT256_p2 – NIST provided MCT vector, part 2
8. vectors_NIST_ECBVarKey256 – NIST provided test vector that varies AES-256 key
9. vectors_NIST_ECBVarTxt256 – NIST provided test vector that varies AES-256 plaintext

Reference vectors format

Use cases and checkers for the reference vectors are described in detail in their respective chapters about Reference vectors sequence, Scoreboard and Reference vectors test. This chapter adds the exact format that the parsing functions are expecting.

Reference vectors are formatted as a three-column CSV file. Although the header is not used in the testbench, its presence is required and the order of columns must not change. Example of the general case vector is shown in Listing 9, while special case with NONE value is shown in Listing 10. Vectors with and without NONE can be used in the same file, as long as the NONE is allowed for the entire test.

Listing 9 – General case vector file entry

```
master_key,plaintext,ciphertext
<user_provided_key>,<user_provided_plaintext>,<user_provided_ciphertext>
```

Listing 10 – Special case vector file entry without specified ciphertext

```
master_key,plaintext,ciphertext
<user_provided_key>,<user_provided_plaintext>,NONE
```

1.2. Design wrapper

```
module aes256_loading_wrap(
    aes256_if aes256_if_conn
);
```

In order to make interfacing with VHDL DUT simpler, testbench uses wrapper module written in SystemVerilog. Wrapper accomplishes to things. First, it instantiates VHDL design and connects corresponding interface ports to it and then exposes that interface as its own port definition. Second, if hierarchical access is allowed, it exposes AES-256 round keys array and also exposes all DUT internal signals required for collecting coverage.

1.3. Interface and concurrent assertions

```
interface aes256_if;
```

In addition to providing testbench with interface to the DUT, interface also implements concurrent assertions to ensure both DUT and testbench follow the defined rules with respect to request and ready signals.

1.4. Coverage

Functional coverage is implemented as a set of covergroups which are first defined for each of the data types and then instantiated for each relevant signal. Each covergroup takes pointer to the signal and it samples those signals at the positive edge of the clock signal, as defined during covergroup definition. List of the defined covergroups is shown in the Listing 11.

Listing 11 – Covergroup definitions

```
covergroup cg_256bit_data
covergroup cg_128bit_data
covergroup cg_32bit_data
covergroup cg_8bit_LUT
covergroup cg_8bit_data
covergroup cg_rcon_LUT_in
covergroup cg_rcon_LUT_out
covergroup cg_key_exp
covergroup cg_enc
covergroup cg_loading
```

Data covergroups have four coverpoints which split an entire range, each with equal number of bits and number of auto bins appropriately set based on the data width. Additionally, min and max values are explicitly covered with one coverpoint. The 8-bit data group, used only for loading output, is outlier as it requires hits in all 256 possible bins.

Look up table (LUT) covergroups also require hit in all 256 bins as this ensures all LUT entries are correctly defined. RCON LUTs are a special case, where only seven bins are ever possible, and these two covergroups are defined appropriately.

Key expansion and encryption covergroups each cover their respective FSMs and counters. FSM coverage consists of four groups of bins. Two groups of legal bins to cover expected states and expected transitions between those states. The other two groups are illegal bins which cover the opposite: unexpected states and unexpected transitions. Counter covergroup is similar in that it covers expected min, max and the value range bins, while defining bins outside of expected range as illegal. Similarly, loading covergroup ensures its counter goes through all expected values: min, max and the range between.

By default, functional coverage is disabled in the testbench, and the covergroup instances are only created if `$test$plusargs("FUNC_COVERAGE")` is supplied at runtime.

In most cases, input and output ports of all modules are chosen for coverage. One exception is made in case the output of one module is tied to the input of another, when only one of those ports is used for coverage. Second exception is made for important internal signals like FSM states. Each LUT module which is instantiated more than once, is only covered for its first instance.

VHDL code coverage is not supported by Vivado 2023.2 version, so detailed code coverage of the design is not possible. However, as SystemVerilog wrapper exposes all important DUT signals, toggle coverage of each of those signals is possible.

1.5. Testbench top

Testbench top module is used to set up the underlying environment before passing on the execution to the UVM test. It instantiates DUT interface, the DUT itself, and connects the two. Clock is generated in one **`initial`** block, while in another the virtual interface is registered to the UVM factory and all available tests are listed. If UVM test is not specified during runtime, first listed test (smoke test in this case) is run. Test is specified, and can therefore be changed, with plusarg **`UVM_TESTNAME`**.

2. Results

This chapter summarizes achieved results when running all of the available tests.

Results can be broken down into four categories:

1. Test suite UVM tests
2. NIST validation (MCT) vectors
3. Functional coverage
4. Code (toggle) coverage

Test Suite

Test suite consists of all UVM tests except the vector-based MCT vector test: seven UVM tests and another seven vector-based tests. Achieved pass rate is 100%.

NIST validation vectors

NIST validation vectors are using UVM vector-based test in the special MCT mode for a total of 100,000 ciphertexts. Achieved pass rate is 100%. This implies, but doesn't guarantee, that design is fully compliant with the AES-256 algorithm.

Functional coverage

Functional coverage is collected while running all of the above tests and then merged in a single database. The achieved functional coverage is 100% as shown on the Figure 2.

Groups Coverage Summary

Score	Inst Score
100	100

Total groups in report: 10

Name	Score	Num Instances	Avg Instances Score	Weight	Goal	Merge Instances	Get Inst Coverage	Per Instance	Auto Bin Max	Comment
top::cg_256bit_data	100	1	100	1	100	0	0	1	64	
top::cg_128bit_data	100	8	100	1	100	0	0	1	64	
top::cg_32bit_data	100	9	100	1	100	0	0	1	64	
top::cg_8bit_LUT	100	6	100	1	100	0	0	1	64	
top::cg_8bit_data	100	1	100	1	100	0	0	1	64	
top::cg_rcon_LUT_in	100	1	100	1	100	0	0	1	64	
top::cg_rcon_LUT_out	100	1	100	1	100	0	0	1	64	
top::cg_key_exp	100	1	100	1	100	0	0	1	64	
top::cg_enc	100	1	100	1	100	0	0	1	64	
top::cg_loading	100	1	100	1	100	0	0	1	64	

Figure 2 – Functional coverage results

Code (toggle) coverage

Likewise, code coverage is collected while running all of the available test. It's limited to the toggle coverage of the VHDL signals exposed in the SystemVerilog wrapper, as noted in the Coverage chapter. Achieved toggle coverage score is 98.4% as shown on the Figure 3.

Missing toggle bits are expected:

1. **`lut_rcon_in[7:6]`** – never used as the LUT input
2. **`lut_rcon_out[31]`** and **`[23:0]`** – never used for the LUT output

Total Toggles Variables	Covered Variables	Uncovered Variables	Score
44	42	2	95.45

	Total	Covered	Toggle Coverage Score
Total Bits	3392	3338	98.4
Bits 0->1	1696	1669	98.4
Bits 1->0	1696	1669	98.4

Figure 3 – Toggle code coverage

Complete code coverage with the tool that supports it is needed to get the remaining three code coverage metrics: statement, branch and condition.

Appendix A: Running tests

1.1. Environment and tools

System requirements:

1. Linux-based OS (tested on Ubuntu 22.04)
2. AMD Vivado 2023.2
3. GNU Make (tested with GNU Make 4.3)
4. Python 3.6+ (tested on Python 3.10)

1.2. Make-based flow

Make provides recipes used to build and simulate the testbench, as well as for generating coverage reports. By running **`make sim`**, makefile goes through four steps:

1. Compile of the DUT and the testbench
2. Compile of the C model into a shared library
3. Elaborate of the previously compiled DUT and testbench, and linking the model shared library
4. Simulation for the specified test name

By default, **`make sim`** runs smoke test, though this can be changed by overriding the make variable **`UVM_TESTNAME`** to any of the supported tests. UVM verbosity is also exposed through **`UVM_VERBOSITY`** variable. Alternatively, running **`make sim_vec`** would use the reference vectors test, by default supplying the base vectors, specified through make variable **`REF_VECTORS`**. Make recipe **`coverage`** is provided to generate the report when only one functional and one code coverage databases exist. If multiple tests are run, and therefore multiple code and multiple functional coverage databases are generated, **`code_cov_merge`** and **`func_cov_merge`** should be used. These two recipes merge all specified databases (each one accepts only its respective database type) and generate a final coverage report for each coverage type.

1.3. Test suite

Test suite, implemented as a python script, provides a straightforward method for dealing with multiple tests at once. By using the make-base flow, test suite handles:

1. Building a design snapshot
2. Running each of the specified tests in parallel (up to the max number of allowed workers) while reusing one common design snapshot
3. Checking of pass/fail status of each test
4. Generating coverage reports from all tests

The only required argument is which test(s) should be run. Test selection can be done in multiple ways:

1. single test: **`-t TEST, --test TEST`**
2. test(s) via JSON test list: **`--testlist TESTLIST`**
3. single vector test: **`-v REF_VECTORS, --ref-vectors REF_VECTORS`**
4. vector test(s) via JSON test list: **`--ref-vectors-list REF_VECTORS_LIST`**

JSON test lists have priority over a single test arguments. Regular tests and vector tests can be specified at the same time. All arguments and their descriptions are available with **`-h`** or **`–help`**. Script uses two internal JSON files (one for regular tests, one for reference vectors) to check if the specified test is supported by the testbench.

Example commands

Run single test

```
python3 run_test.py -t aes256_test_smoke
```

Run single test with run directory

```
python3 run_test.py -t aes256_test_smoke --rundir aestest
```

Run single test with run directory and keep build (if already built with previous script call, it will not build again)

```
python3 run_test.py -t aes256_test_smoke --rundir aestest --keep-build
```

Use JSON testlist instead of a single test

```
python3 run_test.py --testlist testlist.json --rundir aestest
```

Use JSON testlist and generate coverage

```
python3 run_test.py --testlist testlist.json --rundir aestest --coverage
```

Use testlist for regular tests, use separate testlist for reference vectors, and generate coverage reports

```
python3 run_test.py --testlist testlist.json --ref-vectors-testlist
ref_vectors/ref_vectors_testlist.json --rundir aestest --coverage
```

Run only MCT reference vectors with previously built snapshot and run with 1 job to avoid running out of memory

```
python3 run_test.py --ref-vectors-testlist ref_vectors/ref_vectors_MCT.json --rundir aestest
--keep-build --jobs 1
```

Generate coverage reports only for specified tests in the testlists

```
python3 run_test.py --testlist testlist.json --ref-vectors-testlist
ref_vectors/ref_vectors.json --rundir aestest --coverage-only
```

Appendix B: Smoke test example waveform

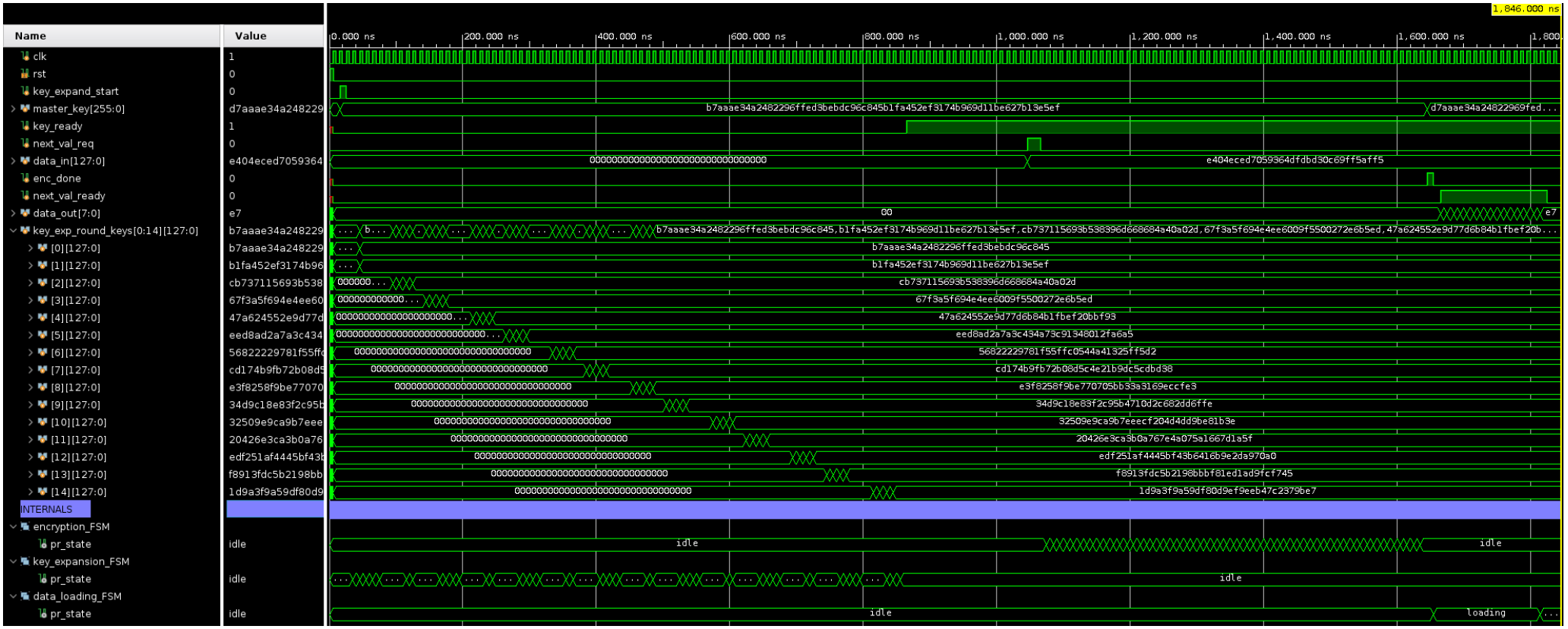

Figure B1 – Smoke test waveform