Converting SystemVerilog Testbench for Decryptor to UVM Testbench

Ruosi Feng, Hanyang Xu

In the project, we convert a complete SystemVerilog testbench for a decryptor to a testbench under UVM testbench. The UVM testbench is functionally equivalent to the SystemVerilog version but has additional capabilities in generating random stimulus given by UVM structure.

Introduction to the SystemVerilog Testbench:

The SystemVerilog version of the decryptor functions as follow:

1) It hardwares the two variables `pre_length, pat_sel` used to generate the encrypted message which could be randomized in UVM.
2) It generates an encrypted message through a for loop and saves the encrypted message to the variable `msg_crypto2`.
3) It writes addresses and writes data into the decryptor DUT.
4) It waits for the output of DUT and record the result in variable `msg_decryp2`
5) It converted to the output to string and display the string

Outline of SystemVerilog Testbench and UVM Modules Correspondence

Base on what we learned from UVM structure, the Systemverilog version of the test code can be transformed into UVM test logically by the following correspondence:

1) The encrypted message generation part of Systemverilog testcode builds the test case and provides it to the decryptor DUT. In UVM, these functionalities are provided by:
    - The UVM sequence item encapsulates the data required to pass between UVM objects and DUT. In this case, it contains one test case: an input address, an input value and an output value.
    - The UVM sequencer forms the test cases which is a queue of sequence items. It creates test cases in a controlled randomized way and builds sequence items to contain these generated test cases.
    - The UVM driver reads sequence items from the sequencer continuously, and put the test cases data on the interface to feed to DUT.
    - The UVM interface is the shared bus between UVM objects and DUT. It provides an abstraction of connection so that all the objects can transfer data without knowing what other classes are.
2) After the message is decrypted by DUT, the Systemverilog testcode records the data, transforms it into string and prints it to the console. In UVM, these functionalities are provided by:
    - The UVM Monitor records and stores the output value. It also calls the scoreboard and passes the recorded value to the scoreboard.

- The UVM Scoreboard evaluates the result by comparing it with the ideal cases. It also presents the results and overall progress of the test in some ways to users.

In general, our UVM design conducts the test by

1) The Sequencer randomly generated one encrypted message, put the data into 64 sequence items and notified the driver.
2) The Driver opened these sequence items one by one, put it on to the interface which passed it to DUT.
3) After DUT finishes all writing operations, the Monitor reads all the output data, stores them in one sequence item and passes that sequence item to the Scoreboard.
4) The Scoreboard gets the test result from the sequence item and compares the result with the original message character by character and reports the result.

UVM Modules Function & Classes

- `Top`:

The top-level file of the UVM testbench. It contains all the included class and header files.

- `Test`: extends `uvm_test`:

This class is the top level representation of the entire UVM test. It instantiates the test environment.

- `Env`: extends `uvm_env`:

The environment class represents the test environment. It instantiates the agent which conducts the test and scoreboard which evaluates the test results.

- `Agent` extends `uvm_agent`:

The agent class represents the top level definition of the process of conducting tests. It instantiates a sequencer which produces a test sequence, driver which drives the test case to DUT and monitor which records the test result.

- `Generator`:

This class implements the process of random generation of one encrypted message. It first randomly generated the three variables, pre_length, pat_sel, and LSFR_init that are required for encryption with constants specified in the SystemVerilog testbench. It then uses the same encryptor part of the original testbench to generate the 64 bytes encrypted message to be sent to the DUT as the input.

- `Reg_Item`: extends `uvm_sequence_item`:

This class defines the transaction object in the test environment. Each item can contain the one-byte write address, 64 bytes decrypted string read from the DUT and one-byte data to be written to the DUT.

- Sequencer: `Gen_item_seq` extends `uvm_sequence`:

This class implements the `reg_item` generation pipeline. In this case, it calls the `Generator` class to generate one test case, creates 64 instances of `Reg_Item` and puts the test case data into these `Reg_Item`.

- `Driver`: extends `uvm_driver`:

This class takes care of the actual transactions of objects between `gen_item_seq`, and `interface` of the DUT. In the `run_phase` task, it first sets the `wr_en` and `wr_init` signals high in order for DUT to start writing, then it transfers the write address and write data of one `Reg_Item` from the `gen_item_seq` each clock cycle to the `interface` and thus DUT for 64 cycles. Finally, it sets the `wr_en` and `wr_init` signals low to finish the writing process.

- `Monitor`: extends `uvm_monitor`:

This class monitors the `done` flag and output of the DUT. In the task `run_phase`, it waits for the DUT writing process to finish, and then reads out the entire decrypted message from the DUT and stores it into one `reg_item`. After storing the entire string, it passes that `reg_item` to the scoreboard through a `uvm_analysis_port`.

- `ScoreBoard`: extends `uvm_scoreboard`:

This class receives the `Reg_Item` from an implementation port, `uvm_analysis_imp` which contains the decrypted result from the `Monitor` and compares it character by character with the original message, then it also outputs the comparison result each bit.

- Interface `reg_if`:

This class is the shared bus between UVM objects and DUT. In this testbench, it contains: `init`, `wr_en`, `raddr`, `waddr`, `rdata`, `wdata` and `done`, all of which are signals required by DUT.

Screenshot of Results

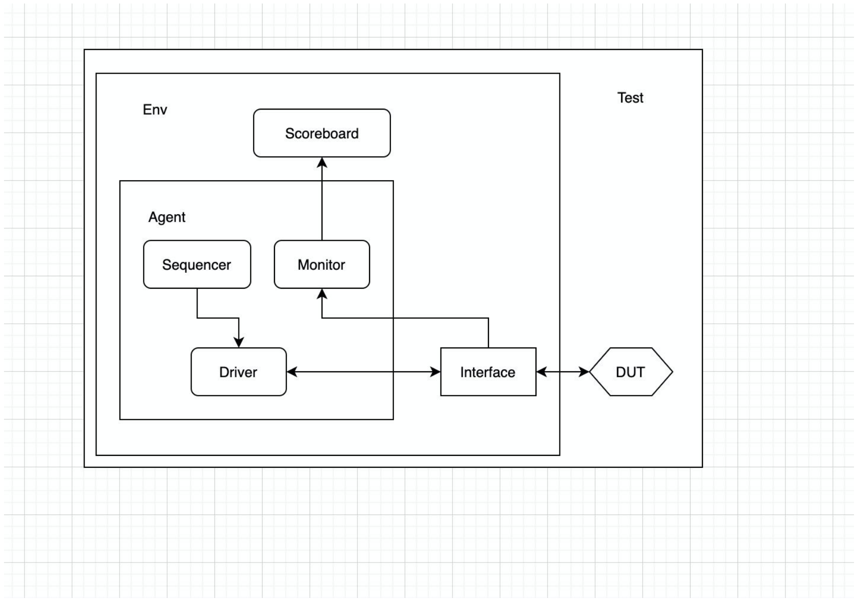
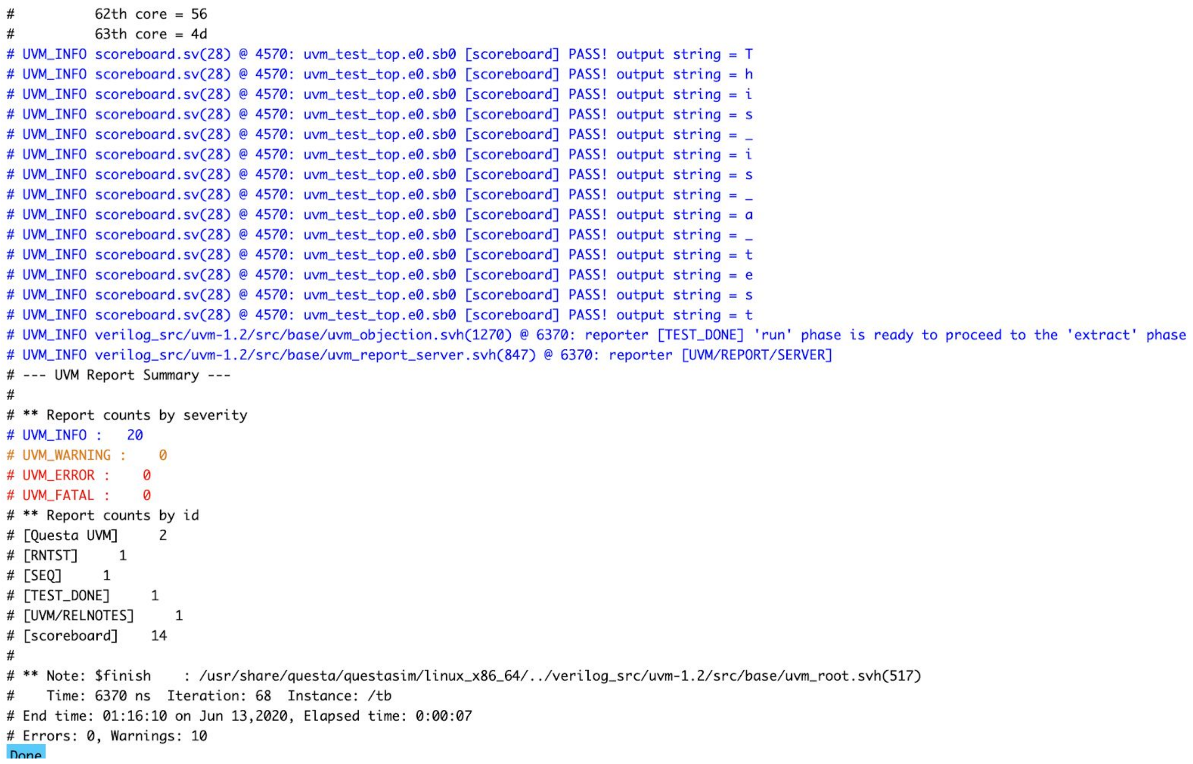


Figure 1. Block diagram of our UVM system



```
#          62th core = 56
#          63th core = 4d
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = T
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = h
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = i
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = s
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = _
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = i
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = s
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = _
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = a
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = _
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = t
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = e
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = s
# UVM_INFO scoreboard.sv(28) @ 4570: uvm_test_top.e0.sb0 [scoreboard] PASS! output string = t
# UVM_INFO verilog_src/uvm-1.2/src/base/uvm_objection.svh(1270) @ 6370: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO verilog_src/uvm-1.2/src/base/uvm_report_server.svh(847) @ 6370: reporter [UVM/REPORT/SERVER]
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   20
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [SEQ]     1
# [TEST_DONE]     1
# [UVM/RELNOTES]     1
# [scoreboard]    14
#
# ** Note: $finish    : /usr/share/questa/questasim/linux_x86_64/../verilog_src/uvm-1.2/src/base/uvm_root.svh(517)
#    Time: 6370 ns  Iteration: 68  Instance: /tb
# End time: 01:16:10 on Jun 13,2020, Elapsed time: 0:00:07
# Errors: 0, Warnings: 10
Done
```

Figure 2. The input string is "This_is_a_test". The scoreboard reports that the outputs match the expected result.

Conclusion and Future Work:

This version of UVM test on decryptor successfully produced a random encrypted message of string "This_is_a_test", passed it to DUT, recorded the data and checked the result as demonstrated in the previous section.

During the development, we found out that the DUT will somehow fail to write its memory from address 16 to address 39 which caused our longer test string to miss some character. We checked both the input and the output of the UVM/DUT interface on both ends and they appear to function correctly. Therefore, we strongly believe that it is not our system's bug.

If we could still improve our system, we could first let our system generate one than one randomized encryption each test. In fact, we already have most of the structure in the code for it to work. For example, in the sequencer class, the outer loop at line 16 can be used to generate more then one randomized encryption if the `num` variable is not set to 1 for now. Second, we could further expand the concept of isolating test generation by introducing coverage, which could cover more scenarios than our current test generation system.

EDA Playground Link to our UVM design: https://www.edaplayground.com/x/5_fh.