How easier to built Basic Verification Testbench using UVM compared to SystemVerilog

Nimesh Prajapati

Department of Electronics & Communication, L.D.College of Engg.

Gujarat Technological University, Ahmedabad

Abstract

ASIC verification is done to get the maximum confidence in the correctness of DUT. Overall, more than 70% of the time is spent on verification. So there is a need for constructing a reusable and robust verification environment. Universal Verification Methodology was introduced to fulfil that goal. This article describes that how easier to built the basic verification testbench using UVM compared to SystemVerilog.

Keywords

UVM (Universal Verification Methodology), SV (SystemVerilog), DUT (Design Under Test)

IJERT

1. Introduction

The purpose of a Testbench is to determine the actual correctness of the DUT and the goal is to ensure full conformance with specification. The testbench creates constrained random stimulus, and gathers functional coverage. The testbench includes the following steps

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness
- Measure progress against the overall verification goals

.

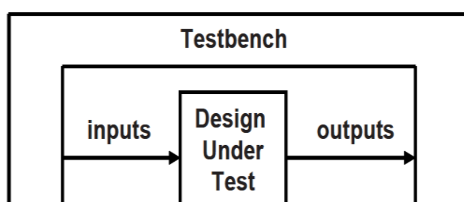

Figure 1. Basic verification testbench

Figure 1 shows basic verification testbench environment of DUT. A testbench that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. A test bench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output, and conditional testing. Testbench mimic the environment in which the design will reside. It checks whether the RTL Implementation meets the design spec or not. This Environment creates invalid and unexpected as well as valid and expected conditions to test the design.

2. Basic Testbench Using SV

SystemVerilog has become a primary language for the design and verification of digital hardware designs. SystemVerilog was first introduced in 2002 as an Accellera standard that specified a large number of extensions to the Verilog-2001 Hardware Description Language.

SystemVerilog provides a very powerful mechanism to generate random stimulus. It is based on class-object randomization, which means random variables of a class-object are automatically randomized by a call to the predefined randomize method associated with the object. Constraints further argument the randomization feature. Constraints are properties that define the boundaries within which the randomization feature works. Figure 2 shows verification testbench environment using SystemVerilog.

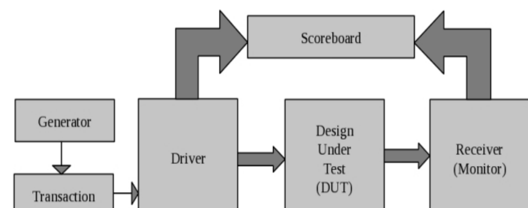

Figure 2. Verification testbench using SV

2.1 Generator

In generator class packet elements are declared i.e. preamble, start frame, address, data, end frame etc. Constraints are also written in the same class. Figure 3 shows an example of generator class.

```
class sv_generator;
      ...
      // packet elements
            bit[7:0] preamble;
      ...
      rand bit[7:0]     data;
      ...
      // constraints
      constraint data_size {data.size inside {[0:1000]}; }
      ...
      endclass
```

Figure 3. Generator class in SV

2.2 Transaction

In transaction class the elements are randomized, then make a packet and forward it to Driver class with the help of Mailbox. Figure 4 shows an example of transaction class.

```
class sv_transaction;
      ...
      //declaration of mailbox
      mailbox transaction_2_driver =new();
      ...
      //implement your logic for randomize
      //the elements and make a packet
      ...
endclass
```

Figure 4. Transaction class in SV

2.3 Driver

The driver class translates the operations produced by the generator into the actual inputs for the DUT. Driver and DUT are connected through interface. Driver also send same packet to Scoreboard for comparison process. This is done with the help for Mailbox. Figure 5 shows an example of driver class.

```
class sv_driver;
      ...
      //declaration of mailbox
      mailbox driver_2_scoreboard =new();
      ...
      //make the handler of Interface
      ...
      //make constructor
      ...
      //implement your logic
      ...
endclass
```

Figure 5. Driver class in SV

2.4 Receiver (Monitor)

Receiver reports the protocol violation and identifies all the transactions. There are two types of receivers, (i) Passive and (ii) Active. Passive Receivers do not drive any signals. Active Receivers can drive the DUT signals. Sometimes this is also referred as Monitor. Receiver and DUT are connected through interface. Receiver receives all transactions form the DUT and combines them to form the packet. Then it sends it to Scoreboard to compare it with actual packet. Figure 6 shows an example of receiver.

```
class sv_receiver;
      ...
      //declaration of mailbox
      mailbox receiver_2_scoreboard =new();
      ...
      //make the handler of Interface
      ...
      //make constructor
      ...
      //implement your logic
      ...
      ...
endclass
```

Figure 6. Receiver class in SV

2.5 Scoreboard

Scoreboard receives packet form Driver and Receiver and then compares them. Scoreboard has two mailboxes. One is used to for getting the packets from the driver and other from the receiver. Then the packets are compared and if they don't match, then error is asserted. Figure 7 shows an example of scoreboard.

```
class sv_scoreboard;
      ...
      //implement your logic for comparison
      //of packet received from
      //Driver and Receiver
      ...
      ...
endclass
```

Figure 7. Scoreboard class in SV

IJERT

3. Basic Testbench Using UVM

UVM represents the latest advancements in verification technology and is designed to enable creation of robust, reusable, interoperable verification IP and testbench components. It uses system Verilog as its language. UVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments.

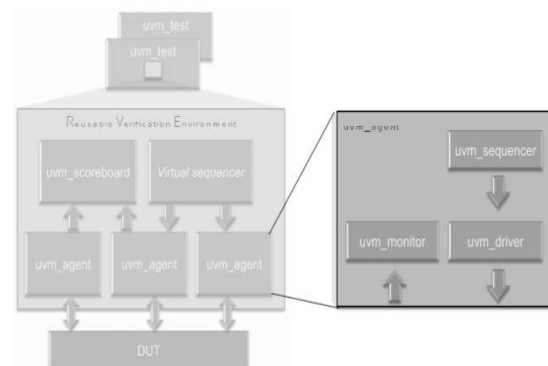

Figure 8. Verification testbench using UVM

In UVM, the class is used as a container to represent components, transactions, sequences, tests, and

configurations. Because it is a class, a UVM component can be extended after-the-fact in arbitrary ways. An extension can add new features or can modify existing features.

In particular, we require this extension capability so that a test can extend a transaction or a sequence in order to add constraints, and then use the factory mechanism to override the generation of those transactions or sequences. Figure 8 shows verification testbench environment using UVM.

3.1 Transaction

Transactions are the basic data objects that are passed between components. Data item are basically the input to the DUT. All the transfer done between different verification components in UVM is done through transaction object.

Networking packets, instructions for processor are some examples of transactions. From the top level test many data items are generated and applied to the DUT so by intelligently randomizing the data items object we can check corner cases and maximize the coverage on the device under test. Pack, unpack, print and compare methods are overwrite in this class. Figure 9 shows an example of transaction class.

```
class transaction extends uvm_sequence_item;
        `uvm_object_utils(transaction)
        function new (string name = "");
                super.new(name);
        endfunction
        ...
endclass
```

.Figure 9. Transaction class in UVM

3.2 Sequence

Sequences are assembled from transactions and are used to build realistic sets of stimuli. A sequence could generate a specific pre-determined set of transactions, a set of randomized transactions, or anything in between. Figure 10 shows an example of sequence class.

```
class my_seq extends uvm_sequence #(transaction);
        `uvm_object_utils(my_seq)
        function new (string name = "");
                super.new(name);
        endfunction
        task body;
        ...
        endtask
        ...
endclass
```

Figure 10. Sequence class in UVM

It is similar to a transaction in outline, but the base class uvm_sequence is parameterized with the type of the transaction of which the sequence is composed. Also every sequence contains a body task, which when it executes generates those transactions or runs other sequences. Transactions and sequences together represent the domain of dynamic data within the verification environment.

3.3 Sequencer

Sequencer is the component on which the sequences will run. The DUT needs to be applied a sequence of transaction to test its behavior. So sequence of transaction is generated and it is applied to driver whenever it demands by the sequencer. Figure 11 shows an example of sequencer class.

```
class sequencer extends uvm_sequencer#(transaction);
        `uvm_sequencer_utils(sequencer)

        function new(string name,uvm_component parent);
                super.new(name,parent);
        endfunction
        ...
endclass
```

Figure 11. Sequencer class in UVM

3.4 Driver

A driver is an active entity that emulates logic that drives the DUT. The driver pulls transactions from its sequencer and controls the signal-level interface to the DUT.

A typical driver repeatedly receives a data item and drives it to the DUT by sampling and driving the DUT signals. For example it can generate read or write signal, write address and data to be transferred. It is the active part of the verification logic.Figure 12 shows an example of driver class.

```
class driver extends uvm_driver #(transaction);
        `uvm_component_utils(driver)
        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction

        function void build;
        ...
        endfunction

        task run;
        ...
        endtask
        ...
endclass
```

Figure 12. Driver class in UVM

3.5 Monitor

A monitor is the passive element of the verification environment. It just samples the DUT signal from the interface but does not drive them.

It collects the pin information, package it in form of a packet and then transfer it to scoreboard or other components for coverage information.

Figure 13 shows an example of monitor class.

IJERT

```
class monitor extends uvm_monitor #(transaction);
      `uvm_component_utils(monitor)
      function new(string name, uvm_component parent);
            super.new(name, parent);
      endfunction
      function void build;
      ...
      endfunction
      task run;
      ...
      endtask
      ...
endclass
```

Figure 13. Monitor class in UVM

3.6 Agent

Sequencers, drivers, and monitors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities. To reduce the amount of work and knowledge required by the test writer, UVM recommends that environment developers create a more abstract container called an agent. An environment may contain one or more agent. Figure 14 shows an example of agent class.

```
class agent extends uvm_agent;
      `uvm_component_utils(agent)
      monitor mon;
      sequencer sqncr;
      driver drv;
      function new(string name, uvm_component parent);
            super.new(name, parent);
      endfunction
      function void build();
      ...
      endfunction
      function void connect();
      ...
      endfunction
endclass
```

Figure 14. Agent class in UVM

3.7 Scoreboard

A critical component of self checking test-benches is the scoreboard that is responsible for checking data integrity from input to output. A scoreboard checks that the DUT is behaving correctly. It keeps track of how many times the response matched with the expected response and how many time it failed.

```
class scoreboard extends uvm_scoreboard;
      `uvm_component_utils(scoreboard)

      function new(string name, uvm_component parent);
            super.new(name, parent);
      endfunction
      function void build();
      ...
      endfunction
      function void connect();
      ...
      endfunction
      task run();
      ...
      endtask
      function void report();
      ...
      endfunction
endclass
```

Figure 15. Scoreboard class in UVM

Figure 15 shows an example of scoreboard class.

3.8 Environment

Environment is at the top of the test bench architecture, it will contain one or more agents depend on design. If more than one agent is there then it will be connected in this component. Agents are also connected to other components like scoreboard in this component. Figure 16 shows an example of environment class

```
class environment extends uvm_env;
      `uvm_component_utils(environment)
      agent1 ag1;
      agent2 ag2;
      scoreboard sb;
      function new(string name, uvm_component parent);
            super.new(name, parent);
      endfunction
      function void build();
      ...
      endfunction
      function void connect();
      ...
      endfunction
endclass
```

Figure 16. Environment class in UVM

3.9 Test

The test class enables configuration of the testbench and verification components, as well as utilities for command-line test selection. Tests in UVM are classes that are derived from an uvm_test class. Figure 17 shows an example of test class.

```
class test extends uvm_test;
      `uvm_component_utils(test)
      environment env;
      function new(string name, uvm_component parent);
            super.new(name, parent);
      endfunction
      function void build();
      ...
      endfunction
      task run;
      ...
      endtask
endclass
```

Figure 17. Test class in UVM

4. Comparison of SV and UVM

- In UVM, there are predefined functions are already available for copy, compare, pack, unpack, print etc. If we want to use them, then directly call them from library. While in SV it is not available. We have to write our own logic code for copy, compare pack, unpack, print etc.

- In UVM, the communication between the modules is done through Ports and Exports. While in SV, It is done through Mailboxes.

IJERT

- Predefined macros are available in UVM i.e. `uvm_info, `uvm_error etc. These types of macros are not available in SV.

- Making the testbench using UVM takes less time compared to making the testbench using SV.

Conclusion

It can be concluded that using UVM, it is easier to built verification testbench compared to SV. Moreover it takes less time compared to SV. Using UVM, we can develop testbench more reusable and perfect compared to SV.

References

1. Universal Verification Methodology 1.1, Accellera, May, 2011
2. SystemVerilog 3.1a Language Reference Manual Accellera's Extensions to Verilog
3. http://www.testbench.co.in
4. http://www.doulos.com/knowhow/sysverilog/uvm/
5. Mark Glasser, "UVM: The Next Generation in Verification Methodology", Verification Horizons, Feb-2011
6. John Aynsley, "Easier UVM for Functional Verification by Mainstream Users", DVcon, Mar-2011
7. Martin Keaveney, Anthony McMahon, Niall O'Keeffe,Kevin Keane, James O'Reilly, "THE DEVELOPMENT OF ADVANCED VERIFICATION ENVIRONMENTS USING SYSTEM VERILOG", ISSC(International System Safety Conference),June-2008
8. Rudra Mukherjee, Sachin Kakkar, "Towards an Object-Oriented Design Methodology Using SystemVerilog", Mentor Graphics

IJERT