Coverage Driven Vip for Avalon Memory Mapped Interface Using UVM

Neetu Goel¹, Neeraj Kr. Shukla² and Shilpi Birla³

¹²VLSI Group, Department of EECE, The Northcap University, Gurgaon 122017, Haryana, India; neetugl06@gmail.com, neerajkshukla@ncuindia.edu
³ECE Department, Manipal University, Jaipur 303007, Rajasthan, India.

Abstract

Objectives: Implementation of Verification IP with randomized stimulus and coverage for Avalon Memory Mapped interface using Universal Verification Methodology (UVM). Methods/Statistical Analysis: The VIP is implemented using UVM. The VIP is implemented using UVM. The unique feature of UVM is its reusability. All the UVM components are implemented to create the VIP for Avalon Memory Mapped (AVMM) interface. The environment uses random stimuli to create different scenarios and improves coverage during regression. The coverage collection measures how much verification goal is achieved. Findings: The VIP created is unique and novel as it is flexible and scalable and allows user to modify according to the need. The factory pattern and phased test flow makes it versatile and reusable to plug and play in any SOC. The cover groups and coverage bins gives the report on what features has been covered and what needs to be exercised. The functional coverage achieved with implemented VIP is 91.6% Application/Improvements: AVMM interface is used to provide communicate between various components. The AVMM VIP can be used for verification in any SOC that uses AVMM interface. The random environment can be constrained to fill the coverage holes.

Keywords: Functional Coverage, Regression, SOC, Universal Verification Methodology, Verification IP

1. Introduction

The fundamental goal of a verification engineer is to ensure that the Device Under Test (DUT) meets the specification as per requirement. As the designs of SOC grow larger and more complex day by day to achieve this goal is difficult. There is a need for a verification environment that minimizes wasted effort. The Universal Verification Methodology (UVM) is a standard methodology in the semiconductor industry for the verification of complex designs. UVM test benches are complete verification environments composed of reusable verification components, and used as part of an overarching methodology of constrained random, coverage-driven verification.

Coverage-Driven Verification (CDV) describes a methodology built around coverage metrics as the primary way to manage verification. The usefulness of coverage depends on how well the chosen coverage model reflects the aims of the testing program. Constrained random stimulus, directed tests are written to fill any coverage holes. The wide range of coverage information helps verification teams assess progress and determine what to do next.

Avalon interface bus is ALTERA’s standard which is used as SOPC Builder. All Avalon compatible cores communicate between various components easily and also specify the communication timing. The port connections are specified between master and slave components. It is used in applications of memory mapping and high speed streaming.

Literature survey has been performed to review the methodologies and work done in past. A comparison between the System Verilog and UVM verification environment is done using the I2C Master Core as the DUT in paper. With the help of experimental results it has been observed that UVM based environment is better performance, reusability, synchronization, and debugging and run flow.

The paper is organized as follows. A small description
of Avalon Memory Mapped Interface is covered in Section 2. The implementation of VIP and the test bench component is explained in Section 3. Experimental results and conclusions are provided in Section 4 and 5, respectively.

2. Avalon Memory Mapped Interface

The Avalon-MM Interface Verification IP (VIP) provides a solution for verification of the designs that uses Avalon-Memory-Mapped interface. The protocol connects Avalon memory-mapped master address-based read/write interface to control a number of Avalon memory-mapped slave peripherals. Typically, Avalon-MM master is a microprocessor; and memories, DMA, UARTs and timers are used as slaves. The Avalon-MM interface defines: set of signal types, their behavior and the types of transfers supported by these signals.

Some of the features of the Avalon-MM interface are:
- Avalon MM peripherals do not need to decode address and data cycles as it uses separate Address, Data and Control lines.
- It supports up to 1024-bit Data Width.
- Avalon-MM interface provides an interface for synchronous operation and facilitates high speed peripheral integration.
- It handles data transfer between peripherals with dynamic bus size.
- It is simple and easy to understand interface protocol.
- It utilizes low on-chip logic resources.
- It provides high performance up to one-transfer-per-clock.
- It supports bus idle insertion between data frames
- It supports burst read and write and misaligned transfers.

In AVMM two types of transfer are discussed.
- **Transfer**: is a read or write operation of a word or one or more symbol of data. Here master initiates the transfer and the slave responds to it. It takes one or more clock cycles to complete.
- **Master-slave pair**: this transfer involves master and slave interface. During a transfer, the master interface passes control and data signals through the interconnect fabric to interact with the slave interface.

3. Implementation of UVM Components

UVM is a standard methodology under Accellera for functional verification using SystemVerilog, complete with a supporting library of SystemVerilog code. It helps in designing and implementing modular test bench components and stimulus. Moreover, UVM provides many useful verification features, reusability across various projects and easier migration from simulation to emulation.

3.1 UVM Verification Components

The main parts of our UVM environment are:
- Top module
- Test
- Configuration class
- Testbench class
- Environment class

Figure 1 shows the various UVM verification components that are created to verify AVMM design.

![Figure 1. The block diagram of UVM testbench components of AVMM.](image)

1. **DUT**: The `avalon_mm_slave` is the design-under-test (DUT) module.

2. **Interface**: The interface `mm_if` consists of all the signal...
ports. It specifies the directional information using modport and timing information in clocking block.

3. **Top**: The `mm_top` instantiates the `avalom_mm_slave` (DUT) and `mm_if` (interface). The top module generates the necessary clock signal and calls the predefined UVM task `run_test()` inside an initial block, which is used to activate UVM testbench.

   ```
   initial begin
   run_test();
   end
   ```

   Tests can be run in a UVM environment by any of the two methods:
   (i) The testname is specified as an argument to `run_test()`. Example: `run_test(mm_test)`
   (ii) As a command-line argument using `+UVM_TESTNAME=[test_name]`. Example: `<SIMULATION_COMMANDS> +UVM_TESTNAME=mm_test`

   This is an entry point to UVM to start and configure each component and run a simulation.

4. **Test**: Test defines the development of events to be tested in the testbench. The testcase `mm_test` is extended from `uvm_test`. During `build_phase()`, the objects of `env` and `seq` are created. A sequence/sequences are created and started in the `run_phase` of test.

   ```
   class mm_test extends uvm_test;
   ....
   virtual function void build_phase(uvm_phase phase);
   env = mm_env::type_id::create("env", this);
   seq = mm_sequence::type_id::create("seq");
   endfunction : build_phase
   task run_phase(uvm_phase phase);
   seq.start(env.agent.sequencer);
   endtask : run_phase
   endclass : mm_test
   ```

   ![Figure 2](connection-dut-testbench-top-module)

   ![Figure 3](execution-flow-run-test)

   Figure 2. Connection of DUT and testbench components through top module.

   Figure 3. Execution flow for `run_test()`.
5. **Sequencer, Sequence, Sequence Item**: Sequence will start execution upon calling the start of sequence from the test.

```verilog
define sequencer.start(sequencer_name);
```

A uvm_sequence is derived from an uvm_sequence_item and it is parameterised with the type of mm_seq_item.

```verilog
class mm_sequence extends uvm_sequence #(<mm_seq_item>);
```

Sequence contains handle `req` and `rsp` of `mm_seq_item`. The execution of sequence is defined in body method of sequence. `m_sequencer` handle contains the reference to the sequencer.

The `mm_sequencer` will create sequence items and send them to the `mm_driver`. The sequencer control the flow of request and response sequence items between sequences and the driver. Sequencer and driver use TLM Interface to communicate transactions.

**uvm_sequencer** and **uvm_driver** base classes has `seq_item_export` and `seq_item_port` defined respectively. User need to connect them using TLM connect method in the `connect_phase()` of the agent.

Example:

```verilog
driver.seq_item_port.connect(sequencer.seq_item_export)
```

Figure 4. Communication process between Sequencer and Driver.

6. **Driver**: The `mm_driver` is written by extending uvm_driver. The uvm_driver is a parameterised class with the type of request and response `mm_seq_item`.

```verilog
class mm_driver extends uvm_driver #(mm_seq_item);
```

The communication between sequencer and driver occurs as shown in Figure 4. The sequencer creates the “transaction item” with the item_handle and calls `start_item(item_handle)`. The Sequencer is blocked by this call till the Sequence and transaction access to the Driver is granted. It randomizes the transaction and then the Driver is ready to use it.

Similarly on the Driver side “transaction item” with a handle is generated. The processing is blocked by the call to method `get_next_item(req)` until the sequencer request FIFO has the `req` transaction object and the `req` object is returned as a pointer to `get_next_item()`. Then working with the virtual interface driver completes its protocol transfer. The call `item_done()` method completes the driver-sequencer handshake. As a result of this the `rsp` object handle updates the Sequencer response FIFO.

On the sequencer side, the blocking call `finish_item(<item_handle>)` waits till the transaction data related to protocol is transferred by the Driver.

7. **Monitor**: The `mm_monitor` will sample the DUT signals through virtual interface. This information will be passed down to the functional coverage model, which records and totals the read and write transactions.

```verilog
class mm_monitor extends uvm_monitor;
```

8. **Agent**: An agent typically contains a driver, sequencer, and monitor. A `mm_agent` is written by extending uvm_agent. The instances of driver, monitor and sequencer are declared. Agent components in build phase, driver and sequencer will be created for active agent only.

```verilog
class mm_agent extends uvm_agent;
```

forever begin

```verilog
seq_item_port.get_next_item(req);
```

... driving logic ...

```verilog
seq_item_port.item_done();
```

end

datatype : run_phase
endclass : mm_driver
mm_sequencer sequencer;
mm_monitor monitor;

........
// creation of components in build_phase
function void build_phase(uvm_phase phase);
if(get_is_active() == UVM_ACTIVE) begin
  driver = mm_driver::type_id::create("driver", this);
  sequencer = mem_sequencer::type_id::create("sequencer", this);
end
monitor = mm_monitor::type_id::create("monitor", this);
endfunction : build_phase
// connect_phase
function void connect_phase(uvm_phase phase);
if(get_is_active() == UVM_ACTIVE) begin
  driver.seq_item_port.connect(sequencer.seq_item_export);
end
endfunction : connect_phase

4. Simulation Results

The compilation and simulation of code was done using QuestaSim. The verification was carried on both read and write cycles. The simulation results carried out on DUT by creating verification environment is shown in Figure 5.

The driver is implemented to drive the stimulus to the input of the DUT. The sequence items are randomized to get the random stimulus generation. Below is the snapshot from the logfile showing all the testbench components and phases are working properly.

The environment is successfully implemented. The covergroup is written to collect coverage on address, read and write. Coverpoints have been implemented. Figure 7 shows the snapshot of the coverage which is shown to be achieved as 91.6%.

![Simulation results](image)

**Figure 5.** Simulation results.
Coverage Driven Vip for Avalon Memory Mapped Interface Using UVM

Figure 6. Snapshot showing the execution of simulation log

Figure 7. Coverage report.
5. Conclusion and Future Scope

In this project Avalon Memory Mapped Interface VIP is implemented using UVN constructors and base classes. Sequences are generated using randomized constraints which cover all the necessary cases along with some corner cases. The simulation result ensures the correct functionality of design. Functional coverage reports how much behavior of the design has been tested. The coverage collected on mm_cvgrp is 91.6%. The environment used random stimulus which helps in regression to create different scenarios. The test environment is interoperable and can be implemented in real time systems. The work can be enhanced by implementing scoreboard and response checkers. More regressions can be run to enhance the coverage report to 100%.

6. References