Lessons in developing and deploying OVM Compliant VIP

Mike Bartley, Test and Verification Solutions
Andy Bond, Lead Verification Engineer, Icera

Using external VIP (Verification IP) brings several advantages including availability, independence in both checkers and coverage, robustness from use in several environments. However, the VIP must be developed so that it is easy for the user to incorporate the VIP into their environment. In this paper, we look at practical lessons learned in both the development and deployment of VIP for use in complex OVM (Open Verification Methodology) SoC (System-on-Chip) verification environments.

The VIP in question was deployed by Icera, a fabless semiconductor company that develops chipsets for high-performance mobile broadband applications. Icera had a need for two new pieces of verification IP. The first VIP they required was to be used in the verification of a new SDCard interface they were adding to their chipset. They had bought in the design IP from a supplier and wanted to source the SDCard VIP from a different supplier for use in their chip level test environment. The purpose of the verification environment was partly to validate the design IP, but mostly to verify the integration of the IP into the SoC and develop the lower levels of a SDC driver. The VIP could mimic the number of SDCard standards (SDC, SDHC, SDXC) and was also highly configurable thus enabling Icera to verify the chip and their software in a wide range of configurations and settings. The protocol checkers delivered with the VIP gave Icera confidence that the design IP met the relevant specification. And the functional coverage built into the VIP allowed Icera to ensure that they had covered all the necessary scenarios in their verification.

Despite the lack of real-time constraints, simulation speeds prevented Icera from actually running a full device driver. The level of SW threading/context switching etc. is too complex, resulting in excessive runs times for even the simplest of tests. However, basic routines such as card identification or individual SDC commands upon which the driver is constructed may all be tested. In addition, the nature of an SDC driver is advantageous for this style of top level testing. The device driver must first identify available/connected SD cards before establishing their capabilities. This allows the verification engineer to construct a test program (or series of test programs) that by their very nature should be agnostic to the SD card it detects.

As such the test bench was constructed to instance specific variants of SD cards, like physically inserting different cards on an evaluation board. These instances were configured using “ovm_test” extensions, each with an individual set of factory overrides defining the VIP’s capability. All other card characteristics, such as response times, error responses etc. are left to be randomized using the standard OVM sequencing seeded from the supplied SV seed. Using this approach allows the same test program to be run varying only the “ovm_test” configurations and random seed. Each simulation run is equally valid, but each alters the execution flow through the SW and HW. For example, under the SDC protocol, by default the card sends the response “R1” for CMD11. However, the protocol allows for the situation where the card does not respond. In order to program this scenario into the SDCard VIP, the user sets the following command in the test

- set_config_int("sve.tvs_sdc0.slaves[0].slave_engine", "hold_cmd11_resp", user_val);
  where user_val is either 0 (response enabled), and 1(response disabled)
This simple setting allows Icera to test that the design IP and software correctly handles this corner case.

Icera already used OVM in their verification environment and so this made it possible to get the VIP developed externally and then easily integrated into their environment. The VIP was developed by TVS (Test and verification Solutions) who have many years of experience in both developing and using VIP.

The VIP followed the OVM development guidelines but there were also number of lessons that were learned regarding the delivery and integration of the VIP. For example, when delivering VIP the supplier has to respect the fact that they don’t own the compile line. Each switch the supplier requires on a command line is a switch that may conflict with one that the customer is already using. Thus, the supplier should avoid any command line switches. In addition, the directory structure / file-naming is important for several reasons.

- There must be an easy distinction between “deliverable” code (code which makes up the VIP which will be used in other test-benches) and example usage code provided to prove the VIP.
  - If the supplier does not take care the typedef’s, constant declarations and define’s can find themselves in test-bench code that won’t be part of a higher-level test-bench, but are required by the actual VIP.
- In order to easily integrate the VIP into a higher-level build structure, the “deliverable” directories must be easily isolated, as the customer may only want to copy the VIP into their source repository.
- Some simulators make distinctions between SystemVerilog and Verilog code by file extensions, this can cause issues when built into a higher-level compile.
- Code that must be compiled should be easily identifiable from code that must be included.
- Files that contain package declarations must be compiled before code that uses that package. As such these files, should be identifiable by name.

Companies also often have their own mechanisms for such things as messaging and objections and the supplier cannot assume that the OVM mechanisms (“ovm_report”, “ovm_objection”, etc) are ubiquitous. This is very common in companies with lots of C, C++, specman, etc. components in their test bench. A simple solution to this is to use macros for each message / objection, defined in a separate file. This allows the customer the ability to easily override these macro definitions for simpler integration.

The second piece of VIP Icera that Icera required was OVM VIP for several serial protocols (e.g. SPI, I2C, I2S). The Icera Livanto digital baseband chip is a soft modem in regards to digital processing. However, by supporting large number of off-chip interface standards Icera can provide their customers with more flexibility in their board level design. For example, by providing support for a large variety of serial protocols for communication with PAs, PMICs, audio chips etc. Icera customers can use the best, most easily available and most cost-efficient components for their product. However, this chip level flexibility is at the cost of the verification effort required in proving each individual protocol.

Unfortunately, there aren’t any shortcuts with protocol verification. There can be an element of re-use for related standards, but fundamentally you should treat each with the same level of attention. That said, all serial interfaces just basically move
data from one point to another, the packet format may change or even be wrapped in a higher-level packet, but the fundamental movement of data through a channel is common. This opened the prospect of creating a generic UVC which can distinguish between the data-pump and the protocol so that most of the testbench design can be re-used across standards. The monitor/driver must be extended to understand the finer points as do the protocol checkers and coverage, but the data scoreboarding can be completely re-used. By providing the test-bench integrator with a common (pre-verified) score-boarding interface for connectivity/integration testing not only is the total effort in test-bench development reduced, but the verification engineers confidence in the VIP is also increased.

Icera thus decided to take a novel approach to serial protocol verification and asked TVS to develop a generic piece of OVM VIP that could then be adapted to the various protocols. Figure 1 below shows the typical structure of a block level test bench where the generic serial IO VIP would be used. Of course, however, the generic VIP would first need to be converted into a specific serial protocol – I2C in this example. The object-oriented features and factory mechanism built into SystemVerilog and OVM enabled such an approach as explained below. Below we discuss how to design the VIP to allow for this specialisation.

**Figure 1: Generic Serial IO Slave VIP**
The object-oriented features allow us to create generic classes that will be specialised for the various protocols. For example, generic sequence items are generated to be used by the sequencer for sending to the driver in the generic test bench. Figure 2 below shows how these are then specialised for the SPI protocol.

```markdown
class tvs_sio_spi_slave_sequence_item extends tvs_sio_generic_slave_sequence_item;
    // Random Variable selecting which mode of CPOL_CPHA Operation
    rand spi_cpol_cpha_mode_e   spi_cpol_cpha_mode;

    // Random Variable selecting mode of data transmission: MSB or LSB
    rand spi_lsbfe_e            spi_lsbfe;

    // Random Variable selecting mode of spi: NORMAL or BI-DIRECTIONAL
    rand spi_opr_e              spi_opr;
    // etc
```

**Figure 2: Code demonstrating specialising sequence item for SIO**

Also, the slave monitor has an analysis port that allows it to connect to the scoreboard. Figure 3 below shows how the analysis port sends a generic sequence item to the scoreboard. This means that the user’s scoreboard only should deal with the generic sequence items rather than sequence items for each protocol.

```markdown
class tvs_sio_generic_slave_monitor extends ovm_monitor;
    // Instantiation of the Generic Virtual Interface
    virtual tvs_sio_generic_intf generic_s_if;

    // Instantiation of the Generic Slave Sequence Item
    tvs_sio_generic_slave_sequence_item   generic_xaction;

    //PORT DECLARATION
    ovm_analysis_port #(tvs_sio_generic_slave_sequence_item) slave_monitor_port;
```

**Figure 3: Code demonstrating generic slave monitor**

Figure 4 below shows how the VIP user can connect multiple slaves to the main test bench scoreboard.
The DUT and the VIP are connected using a System Verilog interface. The generic interface is adapted to specific protocols using the "modport" mechanism. Now that TVS has completed the generic slave code, the only changes required to the generic code to add an additional protocol is to add the required signals and define a modport specific for the protocol. The generic code then should be extended for the specific protocol.

The user of the VIP has a few very simple steps to go through.

1. Instantiate the tvs_sio_generic_intf instance in the test bench top.
2. Use the ovm function set_config_object to replace the generic interface present globally with top level instance as show below.

```hls
// Initiating the Test Process
initial
begin
  set_config_object("*", "generic_intf_wrapper", generic_intf_wrapper, 0);
  run_test();
end
```

3. Instantiate the tvs_sio_env in the top level environment.

```hls
// Instantiation of the tvs_sio environment
tvs_sio_env  sio_env;
```

4. Create the object of the tvs_sio_env in the top-level environment and pass the number of generic agents you want to create as show below.

```hls
function void tvs_sio_top_env::build();
  super.build();
  sio_env = tvs_sio_env::type_id::create("sio_env", this);
  sio_env.no_generic_slaves = this.no_generic_slaves;
endfunction
```

5. Create the necessary connections for the tvs_sio_env:
   a. Connect the user top level virtual sequencer with the generic sequencer used inside the tvs_sio_env.
   b. Connect the generic monitor output ports to the user scoreboard.

6. The user can make the necessary configurations in their virtual sequence.
7. The necessary configurations to be done by the user at the test case are as show below.
   a. Setting the number of generic slave to be used.
   b. Setting the generic slave agent to be in the active mode.
   c. Setting the generic agent sequencer count to be zero, as the sequence operations are going to be controlled from the virtual sequencer.
   d. Use the set type override instance by type function to factory override the generic slave components using the protocol specific components.
   e. Variable setting necessary for the protocol agent monitor.
This paper has considered the issues beyond those covered in the OVM manual to ease integrating OVM VIP into an OVM test bench. It has demonstrated how a well-constructed VIP can allow the user to easily adjust the configuration settings to test both their hardware and software in various corner cases. It has also demonstrated how the object-oriented features and factory mechanism built into SystemVerilog and OVM has allowed TVS to build a generic serial interface VIP that is easily adapted for different protocols.

```verilog
class tvs_sio_spi_basic_test extends tvs_sio_test;

virtual function void build();
//Setting the number of APB Master Agent and the Generic Slave Agent
set_config_int("sio_top_env", "no_generic_slaves", 1);
....

//Using the spi slave driver insted of the generic slave driver
set_inst_override_by_type("sio_top_env.sio_env.generic_slave_agent[0].driver", tvs_sio_generic_slave_driver::get_type(),
tvs_sio_spi_slave_driver::get_type());

//Using the spi slave monitor instead of the generic slave monitor
set_inst_override_by_type("sio_top_env.sio_env.generic_slave_agent[0].monitor", tvs_sio_generic_slave_monitor::get_type(),
tvs_sio_spi_slave_monitor::get_type());

//Using the sphi bus monitor instead of the generic bus monitor
set_inst_override_by_type("sio_top_env.sio_env.generic_bus_monitor", tvs_sio_generic_bus_monitor::get_type(),
tvs_sio_sphi_bus_monitor::get_type());

//Variable settings for the generic slave monitor
set_config_int("sio_top_env.sio_env.generic_slave_agent[0].monitor", "spi_sphi", SPI_NORMAL_MODE);
set_config_int("sio_top_env.sio_env.generic_slave_agent[0].monitor", "spi_sphi_sphi_mode", SPI_CPOL_CPHA_00);
set_config_int("sio_top_env.sio_env.generic_slave_agent[0].monitor", "spi_sphi_iodev", SPI_IODATA);
set_config_int("sio_top_env.sio_env.generic_slave_agent[0].monitor", "spi_sphi_biddir_mode", SPI_BIDIR_MODE_OFF);
set_config_int("sio_top_env.sio_env.generic_slave_agent[0].monitor", "spi_sphi_biddir_enable", SPI_BIDIR_ENABLE_OFF);

//Set the default virtual sequence
set_config_string("sio_top_env_top_virtual_sequence", "default_sequence", "tvs_sio_spi_sphi_bfm_normal_tx_rx_00_lsb_vs");

//Create the vip
super.build();
endfunction: build

endclass: tvs_sio_spi_basic_test
```