A Reusable Verification Testbench Architecture Supporting C and UVM Mixed Tests

Richard Tseng

Qualcomm

Boulder CO, USA

www.qualcomm.com

ABSTRACT

A SOC design in a company usually would have been evolved for many product generations. There are many software tests recognized as "golden" regression suites have been used for signing off the tape out. While the design verification community moving toward the UVM, engineers have been facing a big challenge to reuse or import those golden C tests in a UVM testbench. In this paper, a generic UVM testbench architecture that overcomes this challenge will be demonstrated. The testbench allows engineers to reuse high-level verification components, and C and UVM test sequences for future design generations. It also allows users to run multiple C and UVM mixed test sequences concurrently. Test writers can use those existing API tasks and UVM reporting macros, such as `uvm_info and `uvm_error in C and UVM interchangeably. The integration of the UVM register layer will be addressed as well.

Table of Contents

1 Introduction 5
2 Motivations and Goals 6
3 Problems and Challenges 6
  3.1 Driving Main Test Flow with C and UVM Sequences 6
  3.2 Creating UVM Sequences with API Tasks Instead of UVM “do” Macros 7
  3.3 Mapping Generic Transactions to Bus Interface Specific API Tasks 9
  3.4 Supporting Reuse of High-Level C and UVM Tests 10
  3.5 Handling Multi-Threaded Test Sequences 11
  3.6 Building C and SV Transaction Interface 11
  3.7 Using UVM Reporting Services in C 12
  3.8 Integrating UVM Register Layer 12
4 C and UVM Test Sequence Examples 13
5 Layered Testbench Architecture 15
  5.1 DUT (Design Under Test) 17
  5.2 UVM Agent Layer 17
  5.3 Generic Transaction Translation Layer (GTTL) 17
  5.4 Project Virtual Sequencer and Project Base Sequence 24
  5.5 C to SV Translation Sequences 25
  5.6 Top-Level Interface 28
  5.7 C and SV Test Sequences 31
    5.7.1 Project Base Sequence 31
    5.7.2 Integrating UVM Reporting Macros in C 37
    5.7.3 Launch C Test from UVM Sequence 38
    5.7.4 Launch Multiple C and SV Mixed Test Sequences 40
    5.7.5 UVM Register Layer 42
  5.8 Testbench Environment 48
  5.9 Test Flow Reviews 50
    5.9.1 UVM Test Sequence Flow 50
    5.9.2 C Test Sequence Flow 50

6 Conclusions 51
7 References 52
8 Acknowledgements 52

Table of Figures

Figure 1 - Bus Interface Specific Test Sequence 9
Figure 2 - Generic Test Sequence 10
Figure 3 - C Test Sequence 13
Figure 4 - UVM Test Sequence 14
Figure 5 - Testbench Architecture 16
Figure 6 - gpb_item class 19
Figure 7 - gpb_pkg 19
Figure 8 - gpb2axi_mst_xl_vseq 19
Figure 9 - gpb2axi_mst 20
Figure 10 - axi_master_base_seq class 22
Figure 11 - Project Virtual Sequencer 24
Figure 12 - GPB C to SV Translation Sequence (gpb_c2sv_xl_seq) 25
Figure 13 - gpb_sequencer 26
Figure 14 - GPB C Interface (gpb_c_if) 27
Figure 15 - c_main_test 27
Figure 16 - Top-Level Interface 28
Figure 17 - gpb_dpi.sv 29
Figure 18 - gpb_base_vseq 31
Figure 19 - Project Base Sequence 33
Figure 20 - uvm_rpt.sv 37
Figure 21 - uvm_rpt.h 38
Figure 22 - Start the C test sequence from UVM SV test 38
Figure 23 - Single C process function 40
Figure 24 - Example of Launching Multiple C/SV Test Sequences 41
Figure 25 - UVM Register Layer Integration Diagram 43
Figure 26 - reg2gpb_adapter 45
Figure 27 - gpb_monitor 46
Figure 28 - tb_env 48

1 Introduction

The UVM (Universal Verification Methodology) is a SystemVerilog based verification methodology. It has just emerged a few years ago. It has been recognized to be the ultimate methodology used by the industry for integrated circuit verifications.

In many companies, the existing designs could have been evolved for many product generations. There were thousands lines of legacy test code written in C. All these C tests were coded with API tasks. They have been proven as “golden” tests for signing off the tape out. In addition, there are hundreds of scripts to auto generate C tests and to support the C test flow.

All major EDA tool vendors are now supporting UVM, and more and more verification IP vendors are delivering UVM based simulation models. However it is a big challenge to reuse C legacy test code in latest UVM technology. Even to convert all the “golden” C tests to UVM test sequences is not a trivial task.

So our goal is to create new test benches and new tests with UVM, and to reuse the existing C tests in the same UVM environment.

This paper presents a testbench architecture that supports the C and UVM SV mixed simulations, so that we can reuse all the “golden” C tests in the UVM testbench. With the testbench architecture, test writers have choices to create new tests either with C or UVM, or even mixed languages, within the same simulation environment.

The testbench architecture would allow us:

- To create UVM test sequences with API tasks instead of using UVM macros, i.e. `uvm_do_with(), `uvm_do_on_with().
- To integrate multiple interface API tasks in one testbench.
- To reuse the legacy, proven C test sequences in the UVM SV environment.
- To be able to launch multiple C and SV test sequences simultaneously.
- To create UVM test sequences in exactly the same way as the C tests, because the API tasks are identical in C and SV. This provides an easy transition path for migrating C tests to UVM test sequences.
- To use UVM message reporting macros in C in exact the same way as `uvm_info(), `uvm_warning(), `uvm_error(), and `uvm_fatal().
- To easily integrate the standard UVM register layer in the same environment (previously also known as RAL in VMM or RVM, which was originally developed by Synopsys).
- To reuse all the UVM verification components, C and SV test sequences, and register tests in future design generations by only swapping out lower level UVM agents.

2 Motivations and Goals

In our design and verification team, engineers have different skill levels in C and UVM SV. Some engineers are more familiar with C, and would like to create C tests, but some engineers are more comfortable to write UVM SV tests. Our goal is to eventually use the latest and the greatest verification methodology, which is UVM for all the verification work, but also, in the meantime, we need a simulation environment to support the both C and UVM test sequences.

The legacy C test sequences have been evolved through many product generations. They have been improved and tested along the road for a long period of time. They are proven to be golden reference test code and will not likely to be changed dramatically in new design generations. The core design itself has been evolved for many product generations as well. Each design generation has progressive changes than the previous version. However the core's bus interfaces have been changed from generation to generation in order to meet the performance requirement, or to fit in a different FPGA platform.

For example, in our design projects, the same design has been dropped or reused in many products and emulation platforms. The primary bus interface of the design has been replaced with a variety of bus interfaces, such as AHB, AXI, and a few other propriety SOC buses, in order to meet the requirements of different targets. The design core itself does not have too many changes between the current and the previous product generations. We would like to reuse most of the high level test code from the previous generation in the current verification environment.

So the motivation was to create a generic testbench which allows us to reuse the higher-level test code on different design targets. At the lower layer of the testbench, we have the freedom to use any UVM bus agents that the DUT bus interface requires.

The ultimate goal is to be able to run the same set of golden test suite either coded in C or UVM for all different design generations and simulation platforms.

3 Problems and Challenges

There were many problems and challenges to be overcome to reach this goal. They are addressed in the following sections.

3.1 Driving Main Test Flow with C and UVM Sequences

In our legacy verification simulation flow, we used Vera to build the testbench and linked the pre-compiled C code into simulator at run time thru Vera Direct-C interface. The tests were started from the Vera TB. We grouped many big C functions and imported them into Vera TB, and let Vera to drive the main test flow by calling those imported C functions. This seemed to be a good way to drive the tests, but there were many problems. First, there were still many sub test flows and logic built in legacy C code. Second, many intermediate arrays had to be created at run time in order to exchange information between C and Vera domains. This made the test flow very complicated. The entire simulation environment turned out to be very confusing, and it was very difficult to debug a test failure.

Also in the old test environment, many scripts have been created to auto generate C code and to support the C test sequences in the test flow. There is simply no easy way to write a script to convert all the C test code to Vera or any other verification languages.

So for the design with the aggressive tape-out schedule at that time, we decided to reuse the legacy C code to drive the main test flow in a UVM testbench. This will be in sync with the regular software testing flow, and it is easy for engineers to debug problems. We chose to build the UVM testbench for the long term compatibility issue. This is because all the simulation bus models we needed are already available in UVM, and all major EDA and VIP vendors support UVM.

As described in the previous section, we would like to gradually import/translate all the C test code to UVM SV test sequences as a long term plan. One reason is that the pure native UVM tests run much faster than a C driven test. For all the legacy C tests, the main test flow will still be driven by C, but for the new created UVM tests, the test flow will be controlled by UVM sequences. So a big challenge is to architect the testbench to support both UVM and C driven test flows.

3.2 Creating UVM Sequences with API Tasks Instead of UVM “do” Macros

In our legacy simulation environment, the C test sequences were consisted of many register read/write tasks calls. In order to reuse those test sequences in UVM testbench, we had to seek a way to implement those C API tasks in UVM sequences. UVM provides many utility macros, such as `uvm_do(), `uvm_do_with(), and `uvm_do_on_with(), to help test writer to create tests. There are no application notes addressed in UVM user’s guides to show how to create/use API tasks within UVM sequences. It is a big challenge to translate the higher-level API tasks to lower-level UVM sequence items.

Basically, what those UVM macros do is to creates a sequence object, randomize it with some constraints, and send it down to the bus driver (also known as Bus Functional Model, BFM) to wiggle the signals on the interface. The interface is connected between the Design Under Test (DUT) and the testbench (TB). The usage of these macros makes it difficult to write tests. Also, macros are very difficult to debug. Test writers usually will spend a huge amount of time on understanding the bus protocol constraints and on how to use these macros that have been integrated in Verification IPs (VIPs) before they can start focusing on the real functional problems. Here are the typical problems using UVM macros to create a test:

1. Test writers will need to know exactly the appropriate, legal constraint combinations that the interface would allow and what the testbench can support before using these macros.

2. With the modern SOC designs, the bus protocols are usually very complicated. The verification models or VIPs usually were developed and delivered by third party vendors. Test writers usually don't have much experience and/or knowledge about their verification models. Sometimes it is very difficult for test writers to specify the proper constraints for the UVM macros in order to create a meaningful test. This is especially a problem if the design only uses a sub-set of a complex protocol.

3. The modern SOC designs usually have more than one interface in the design. To integrate multiple bus interface transactions with UVM macros and to use those macros in the tests are big challenges. In addition, the VIPs could be received from many different vendors. This makes the challenges more difficult to overcome.

4. Since the interface constraints could be very complicated, the test writers should let the simulator's constraint solver calculate the legal constraints and then pass down to the TB, instead of specifying the constraints in the UVM macros. The UVM macros are better suited for verifying the data path oriented designs, such as Ethernet packet router or switches. In other words, the UVM macros do not fit in verification tasks for control oriented design very well. This is because the testing of a control oriented design is usually done by setting many registers at the beginning of the test, and setting off a 'go' command to start a complex hardware operation. There are not many constraints need to be specified for a register access besides the read/write command, address, and data.

5. The VIPs delivered from vendors can be reused across multiple projects. However, many modifications are required to be made in the test environment for each project. Currently there is not an industry standard for consistently reusing and integrating UVM based VIPs into an existing environment. Such a standard would be useful as it would facilitate consistent integration whether the VIP was from internal or external sources.

With all above factors, it is difficult for test writer to create a test in a fairly reasonable amount of time. With the fast paced chip design industry today, we don't have so much leisure time to let the test writers or designers to understand all the details of the third party verification IPs. They should spend more time focusing on debugging real functional problems, instead of being bogged down by the complex bus protocols and the usages of the VIPs.

3.3 Mapping Generic Transactions to Bus Interface Specific API Tasks

Usually a design has more than one bus interface. The bus interfaces could be changed among different design generations. In order to reuse the existing the test sequences, tests should be composed with generic API tasks. Testbench builder needs to map the generic transactions to the bus interface specific API tasks when building the testbench for the next generation of design. One of the challenges is to architect a reusable testbench structure to map the generic transactions to the bus specific API tasks.

For ease of demonstration, two generic API tasks, **GPB** and **GST,** are used throughout the paper.

GPB stands for **Generic Primary Bus**. In our examples, the GPB API tasks are mapped to AXI master API tasks.

GST stands for **Generic Slave Transactions**. The GST tasks are mapped to AXI slave back door API tasks.

For instance, an UVM test sequence that includes AXI master front door transactions and slave back door transactions can be written as Figure 1.

Figure 1 - Bus Interface Specific Test Sequence

```
// write reg1 thru axi master interface
axi_master_write(reg1, data);
// write reg2 thru axi master interface
axi_master_write(reg2, data);

// perform a read modify write to mem_addr1 location
axi_slave_bkdr_read(mem_addr1, read_data);
read_data = read_data | 32'hFFFF_0000;
axi_slave_bkdr_write(mem_addr1, read_data);

// Read the reg3 thru axi master interface, and
// check if the result is returned as expected.
// An error will be reported if there is a mismatch.
axi_master_read_check(reg3, 32'h5A5A_5A5A);
```

However, the test sequence would be reusable in the future design generations if the API tasks are replaced with generic API tasks as shown in Figure 2.

Figure 2 - Generic Test Sequence

```
// write reg1 thru GPB interface
gpb_write(reg1, data);
// write reg2 thru GPB interface
gpb_write(reg2, data);

// perform a read modify write to mem_addr1 location
gst_bkdr_read(mem_addr1, read_data);
read_data = read_data | 32'hFFFF_0000;
gst_bkdr_write(mem_addr1, read_data);

// Read the reg3 thru GPB interface, and
// check if the result is returned as expected.
// An error will be reported if there is a mismatch.
gpb_read_check(reg3, 32'h5A5A_5A5A);
```

Again, the GPB and GST are generic terms used in this paper for ease of demonstration. Depending on the design verification requirements, a different design may have multiple bus interfaces to access the DUT's register spaces and may have multiple slave devices in the testbench. User may have GPB1 transactions mapped to Ethernet packets, and GPB2 transactions translated to the USB transactions. It is also possible the GST1 back door accesses are mapped to the external DDR memory back door transactions, and GST2 transactions are translated to the on-chip AHB slave device's backdoor transactions. The point is the GPB and GST transactions are not limited to be mapped to AXI on chip master and slave transactions. We use AXI bus as examples in this paper since they are commonly used in SOC designs. However the goal is that we should be able to reuse the high level tests in a new design verification environment if they were coded with the generic API tasks, no matter what the lower level bus protocols or UVM agents are used. The testbench needs to support the translation of generic API tasks to bus interface specific transactions.

3.4 Supporting Reuse of High-Level C and UVM Tests

As we discussed earlier, one of our goals is to keep the higher level test sequence intact. To run the same set of tests for a new design, we can simply replace the lower-level UVM bus models of the old design with new models due to the bus interface changes. The new design's regression results should be similar to the old designs. We also would like to use the same format of C API tasks in UVM test sequences. One benefit of this method is it provides an easy transition path to migrate the C tests to UVM sequences in future. Another benefit is both C and UVM tests look consistent across different projects. Users do not need to learn a new set of API tasks when switching to a different design verification platform. However, there are two challenges related to this reuse methodology:

- **Inserting an abstraction layer to separate the high level testbench components from the low level UVM agents** – We should keep in mind that the high level test code needs to be carried over to the next project. To reuse the high level test code and testbench components in a new design, we should use the generic API tasks in all of the test sequences as stated in the previous section. Again, in this paper, the "gpb_write(addr, data)" task will be used to issue the **Generic Primary Bus** write transactions to DUT through front door (on actual bus interface), and the "gpb_bkdr_read(addr, data)" task will be used to perform read accesses on the same bus interface. The **Generic Slave Transaction** back door tasks, "gst_bkdr_write(addr,data)" and "gst_bkdr_read(addr,data)" will be used to issue back door write and read transactions to the slave model. To support these generic read/write transactions, an abstraction layer, **Generic Transaction Translation Layer**, must be built in testbench, so that the high level test sequences can be translated to the transactions that low level UVM bus agents can recognize.

- **Integrating C API tasks to UVM** - The higher-level test sequences could be coded in C or UVM SV, so the API task interfaces need to be kept the same for easy C-to-UVM transitions. For example, a C test composed with GPB and GST API tasks should look identical to UVM test sequences, and both C and UVM tests can be able to run in the same testbench.

3.5 Handling Multi-Threaded Test Sequences

Ideally the testbench would provide the multi-threaded capabilities to mimic the real scenarios. One of the applications is there could be multiple treads of software code running simultaneously. SystemVerilog provides the fork-join process that can be used to launch multiple SV sequences concurrently. However, to support multiple C/SV mixed processes running in the same testbench is another challenge.

3.6 Building C and SV Transaction Interface

SystemVerilog provides the Directed Programming Interface (DPI) to handle the C and SV communications. However there are a couple caveats involved to just use the DPI functions directly to pass the C and SV transactions due to the reliability and compatibility issues across different simulators. The UVM testbench components, bus transactions, and test sequences are all coded with classes. One caveat is that SystemVerilog class member functions cannot be exported [4]. The other caveat is that the testbench needs to support multi-threaded C and/or SV tests running simultaneously. It is a challenge to make the testbench and test sequences reliably reusable for the long term. A C-SV transaction interface may exist in the testbench to handle the translations of C and SV sequence items because of this.

In addition, the transactions issued by C API tasks may target on different bus interfaces. The C-SV interface needs to route transactions to appropriate UVM components for the specified bus interfaces. To integrate the multiple C-SV translation interfaces and to make it consistent and reusable through multiple projects is another challenge.

3.7 Using UVM Reporting Services in C

UVM provides many reporting classes functions to let users report messages in a consistent way throughout the simulation. These functions were encapsulated in the utility macros, such as ‘uvm_info(), ‘uvm_warning (), ‘uvm_error(), and ‘uvm_fatal(). It is easy for users to report messages with these macros. Users can also specify the message’s verbosity level when using these macros, such as UVM_LOW, UVM_MEDIUM, UVM_HIGH, etc. These macros and their features work nicely in a pure UVM SVTB. However, in the C and UVM mixed mode simulation, a challenge is to build the testbench to support the usage of the UVM macros in C tests. Users should be able to use the UVM macros in C code exactly the same way as in UVM tests. The simulation log file should have a consistent message format no matter the messages are reported from a C or UVM test.

3.8 Integrating UVM Register Layer

Register testing is one of the common requirements for sign off the tape out. In one of our ASIC designs, we had over 16,000 registers in the design. It is impossible to perform the register testing without an automated flow. To perform register tests, the register models must be generated and integrated in the verification environment. Some tools or scripts are usually setup to read the register specifications and auto-generate the register models. To integrate the tool flow in the verification environment is definitely a big challenge. Since every company has a different format of the resister specification, and there are already many solutions defined by the EDA tool vendors to facilitate the generation flow, this challenge will not be covered in this paper.

The UVM library provides many test sequences, components, and register model base classes to facilitate the register testing task. However, the UVM register layer cannot be used as is. A few verification components in the testbench need to be integrated to support the predicting or scoreboarding mechanism provided by UVM register layer. This is one challenge.

In the testbench, the Generic Transaction Translation Layer (GTTL) was set to translate the Generic Primary Bus (GPB) transactions to bus interface specific accesses. The UVM register layer is built on top of the GTTL. The register read/write transactions are issued from the register layer, passed down to the GTTL, and translated to the bus interface specific transactions for accessing the registers in DUT (see Figure 5 for reference). To map the UVM register transactions to the GPB transactions, a couple translation tasks in the register layer must be defined. This is another challenge.

4 C and UVM Test Sequence Examples

The examples shown in Figure 3 and Figure 4 demonstrate the expected test sequences for C and UVM SV respectively.

Figure 3 - C Test Sequence

```
void c_main_seq(void) {

  unsigned int read_data; // 32 bit unsigned integer

  // print test start message;
  // the __func__ argument identifies the current
  // function name
  uvm_info(__func__, "Test starts", UVM_MEDIUM)

  // write 0x3204 to control register thru Generic Primary
  // Bus (GPB) interface - front door access
  gpb_write(control_reg, 0x00003204);

  // read status register thru GPB interface, return
  // value stored in read_data
  gpb_read(status_reg, &read_data);

  // back door write to memory address 0x400 with data 0xA5
  gst_bkdr_write(0x400, 32'hA5);

  // back door read from memory address 0x400, read data
  //stored in read_data
  gst_bkdr_read(0x400, &read_data);

  // determine whether the test is pass or fail, and print
  // message
  if (read_data == 0xA5) {
    uvm_info(__func__, "Test Passed!", UVM_MEDIUM)
  } else {
    uvm_error(__func__, "Test Failed!!")
  }
}
```

Figure 4 - UVM Test Sequence

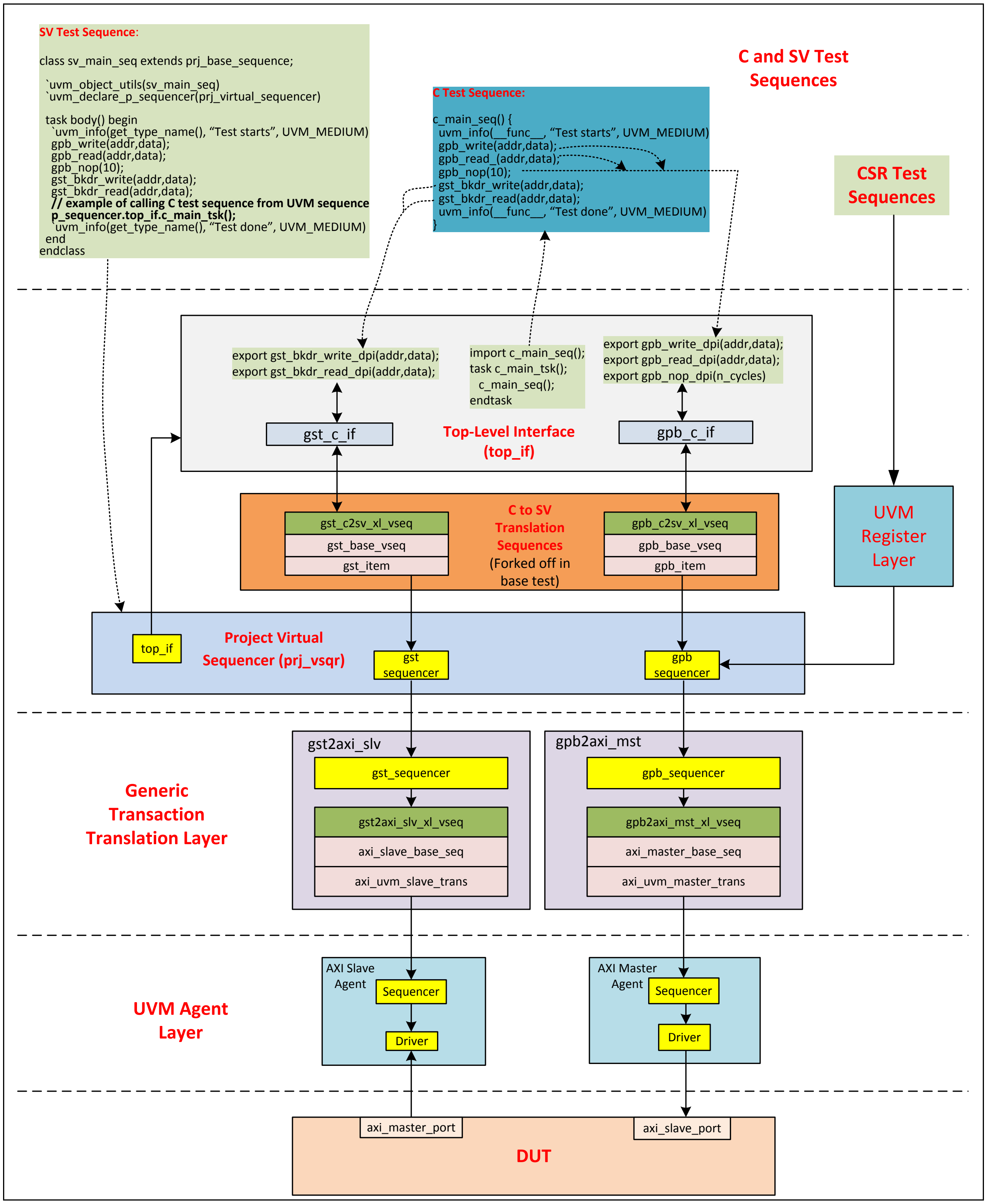
```
class sv_main_seq extends prj_base_sequence;
  …
  task body();

    bit[31:0] read_data;

    // print test start message
    `uvm_info(get_type_name(), "Test starts", UVM_MEDIUM)

    // write 0x3204 to control register thru Generic
    // Primary Bus (GPB) interface - front door access
    gpb_write(control_reg, 32'h0000_3204);

    // read status register thru GPB interface, return
    // value stored in read_data
    gpb_read(status_reg, read_data);

    // back door write to memory address 0x400 with data 0xA5
    gst_bkdr_write('h400, 32'hA5);

    // back door read from memory address 0x400, read data
    // stored in read_data
    gst_bkdr_read('h400, read_data);

    // determine whether the test is pass or fail, and
    // print message
    if (read_data == 'hA5) begin
      `uvm_info(get_type_name(), "Test Passed!", UVM_MEDIUM)
    end else begin
      `uvm_error(get_type_name(), "Test Failed!!")
    end

  endtask: body

endclass: sv_main_seq
```

Note that these two test sequences are almost identical. The test sequence coded in C should be coded in a similar way in UVM, with minimum differences. Users can easily write a script to translate the C test to UVM sequence if it's need, and vice versa.

5 Layered Testbench Architecture

Figure 5 shows the new TB architecture that provides solutions for all the challenges addressed in the above sections.

Figure 5 - Testbench Architecture



SV Test Sequence:

```
class sv_main_seq extends prj_base_sequence;

 `uvm_object_utils(sv_main_seq)
 `uvm_declare_p_sequencer(prj_virtual_sequencer)

 task body() begin
  `uvm_info(get_type_name(), "Test starts", UVM_MEDIUM)
  gpb_write(addr,data);
  gpb_read(addr,data);
  gpb_nop(10);
  gst_bkdr_write(addr,data);
  gst_bkdr_read(addr,data);
  // example of calling C test sequence from UVM sequence
  p_sequencer.top_if.c_main_tsk();
  `uvm_info(get_type_name(), "Test done", UVM_MEDIUM)
 end
endclass
```

C Test Sequence:

```
c_main_seq() {
 uvm_info(__func__, "Test starts", UVM_MEDIUM)
 gpb_write(addr,data);
 gpb_read_(addr,data);
 gpb_nop(10);
 gst_bkdr_write(addr,data);
 gst_bkdr_read(addr,data);
 uvm_info(__func__, "Test done", UVM_MEDIUM)
}
```

```
export gst_bkdr_write_dpi(addr,data);
export gst_bkdr_read_dpi(addr,data);
```

```
import c_main_seq();
task c_main_tsk();
  c_main_seq();
endtask
```

```
export gpb_write_dpi(addr,data);
export gpb_read_dpi(addr,data);
export gpb_nop_dpi(n_cycles)
```

The entire testbench architecture is organized in multiple abstraction layers/blocks for easy demonstration. Each layer/block could be reused in different testbench platforms. As shown in Figure 5, from bottom up, we have DUT, UVM Agent Layer, Generic Transaction Translation Layer (GTTL), Project Virtual Sequencer, C to SV Translation Sequences, Top-Level Interface, and the C and SV test sequences. The following sections describe how each layer/block works in details.

5.1 DUT (Design Under Test)

This is the design that we are trying to verify. For the ease of demonstration purpose, only the AXI master and AXI slave interfaces are illustrated in Figure 5. The AXI slave port of the DUT is connected to the AXI master port of the UVM bus model. The AXI master port of the DUT is connected to the AXI slave port of the UVM bus model.

5.2 UVM Agent Layer

This layer includes the lower level UVM compliant bus models. The UVM agent usually includes a bus driver, a sequencer, and a monitor. To simplify the illustration, the monitors are not shown in the diagram. The UVM agents that involve complicated bus protocols could be developed by the third party VIP vendors, such as the AXI master and slave models in our example. The bus model receives a UVM sequence item (transaction object) from the upper layer, and drives the bus signals to the DUT according to the interface protocol.

Since there are already many UVM bus models available through the VIP vendors and they were all created and verified by dedicated engineers, it is easier for verification engineers to integrate these models in the testbench instead of creating their own. In addition, due to the strong momentum of UVM adoption, there will be more and more UVM VIPs available. That is why we choose the UVM as the primary verification methodology.

5.3 Generic Transaction Translation Layer (GTTL)

The purpose of this layer existing in the TB is for the reusability. Let’s say for the next project, we will use a different bus interface to communicate with DUT, such as AHB interface instead of AXI. With this layer existing in the TB, we can simply replace the AXI agents with AHB agents, and modify the translation sequences in this layer. Then we can quickly rebuild a testbench and start to run all the existing tests without changing the test sequences and upper layer verification components.

To make it generic, we name two major bus transactions, GPB and GST. They both can be issued from C or UVM test sequences.

The **Generic Primary Bus (**GPB) verification component behaves as the master model in the testbench. It receives transactions from test sequences and issues the front door transactions to the DUT.

The **Generic Slave Transaction (**GST) verification component acts as a slave device in the testbench. It usually has a dynamic array to mimic the memory in the device. The slave component is connected to DUT's master port and interacts with the design through the bus interface protocol. It also receives the back door transactions from the test sequences to access the slave memory. The name **GST** is used throughout this paper for these generic back door transactions.

Essentially we use GPB to issue all the primary bus transactions to the DUT, such as register read/write accesses. In our examples, the GPB transactions will be translated to AXI master transactions and passed down to the AXI master agent. GST is primarily used to issue the backdoor transactions directly to the memory resides in AXI slave model to perform preload, dump, and back door read/write operations. These GST transactions will not go thru the actual signal interface to access the memory, but will perform the back door accesses directly to the memory inside the model. The GST transactions consume zero simulation time, and can be used to improve the simulation performance, especially for a test trying to access a big chunk of the slave memory. On the other side of the slave model, DUT can access the memory thru the actual AXI bus interface.

There is a translation sequence existing in the gpb2axi_mst component and gst2axi_slv component. They both are extended from uvm_component. Their job is to translate the high-level, generic UVM transactions to the bus specific transactions.

As shown in Figure 5, the translation sequence existing in the gpb2axi_mst is called gpb2axi_mst_xl_vseq. It retrieves the GPB sequence item (gpb_item) from the gpb_sequencer, translates it to the axi_uvm_master_trans object, and passes it down to the AXI master agent. The axi_uvm_master_trans is the AXI master transaction base class provided by the VIP vendor. It is extended from the uvm_sequence_item.

The same mechanism applies to the translation sequence in the gst2axi_slv component (gst2axi_slv_xl_vseq). It retrieves the GST sequence item (gst_item) from the gst_sequencer, translates it to the axi_uvm_slave_trans object, and passes it down to the AXI slave agent. Similarly, the axi_uvm_slave_trans is the AXI slave transaction base class provided by VIP vendor. It is also extended from uvm_sequence_item.

In this paper, we will focus on demonstrating how to construct the GPB transaction and how to use and translate it to AXI front door operation in the verification environment. The definition and application of GST transaction are similar to those of GPB. The details will not be repeated.

The generic API tasks issue the generic front door or generic back door transactions. Figure 6 shows how the transaction object gpb_item (also known as GPB sequence item) is constructed. Figure 7 shows the gpb_pkg that includes all the GPB relates SV files.

Figure 6 - gpb_item class

```
class gpb_item extends uvm_sequence_item;

  // properties
  rand gpb_op_type op;

  rand bit[GPB_ADDR_MSB:GPB_ADDR_LSB] addr;
  rand bit[GPB_DATA_MSB:GPB_DATA_LSB] data;
  rand int unsigned nop_cycle;
  …
endclass : gpb_item
```

Figure 7 - gpb_pkg

```
package gpb_pkg;

  import uvm_pkg::*;
  …

  // define operation type for the gpb_item transaction
  typedef enum int {WRITE=0, READ, NOP} gpb_op_type;

  `include "gpb_item.sv"
  `include "gpb_sequencer.sv"
  `include "gpb_reg2gpb_adapter.sv"
  `include "gpb_base_vseq.sv"
  `include "gpb_c2sv_xl_vseq.sv"
  …
endpackage
```

Figure 8 shows the code snippet of the translation sequence (gpb2axi_mst_xl_vseq), and Figure 9 shows how it is started in the gpb2axi_mst component. The GST sequence item, the GST translation sequence, and the gst2axi_slv component are similar to those of GPB.

Figure 8 - gpb2axi_mst_xl_vseq

```
import gpb_pkg::*;
```

```
class gpb2axi_mst_xl_vseq extends axi_master_base_seq;

  `uvm_object_utils(gpb2qbsb_xl_vseq)

  // refers to the sequencer in AXI master agent
  `uvm_declare_p_sequencer(axi_master_sequencer)// ref. [2]

  gpb_sequencer gpb_sqr;   // upper layer sequencer
  gpb_item      gpb_item0; // upper layer sequence item

  virtual task body();
    forever begin
      gpb_sqr.get_next_item(gpb_item0); // ref. [1]

      case (gpb_item0.op)
        gpb_pkg::WRITE: begin
          // map gpb_write() to axi_mst_32bit_write()
          axi_mst_32bit_write(gpb_item0.addr, gpb_item0.data);
        end
        gpb_pkg::READ: begin
          // map gpb_read() to axi_mst_32bit_read()
          axi_mst_32bit_read(gpb_item0.addr, gpb_item0.data);
        end
        gpb_pkg::NOP: begin
          // map gpb_nop() to axi_mst_nop()
          axi_mst_nop(gpb_item0.nop_cycle);
        end
      endcase

      gpb_sqr.item_done();
    end
  endtask: body
endclass: gpb2axi_mst_xl_vseq
```

Figure 9 - gpb2axi_mst

```
class gpb2axi_mst extends uvm_component;
  gpb_sequencer                  gpb_sqr;
  gpb_monitor                    gpb_mon;
  gpb2axi_mst_xl_vseq            xl_seq;
  axi_uvm_master_agent           axi_mst_agent;
  …
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    gpb_sqr = gpb_sequencer::type_id::create("gpb_sqr",…);
    gpb_mon = gpb_monitor::type_id::create("gpb_mon”,…);
    analysis_export = new("analysis_export", this);
    ap = new("ap", this); // analysis port
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    // analysis_export is in uvm_subscriber
    analysis_export.connect(gpb_mon.analysis_export);
    gpb_mon.ap.connect(ap);
  endfunction

  task run_phase(uvm_phase phase);
    xl_seq = gpb2axi_mst_xl_vseq::type_id::create("xl_seq");
    xl_seq.gpb_sqr = gpb_sqr;
    // start the translation sequence
    xl_seq.start(axi_mst_agent.sequencer);
  endtask: run_phase

endclass: gpb2axi_mst
```

Note that the gpb_sqr is built in the gpb2axi_mst component. The translation sequence is started on the sequencer in AXI master agent. The axi_mst_agent is instantiated in the top-level testbench environment. The gpb2axi_mst holds a handle of that. The connection between the gpb2axi_mst component and the axi_mst_agent is also done in the top-level testbench environment as shown in Figure 28.

In the example, the task axi_mst_32bit_write() is called when the operation property of the gpb_item is gpb_pkg::WRITE, which is defined as an enumeration type in gpb_pkg. The address and the data arguments are passed into the task. The similar translation mechanism is used to map gpb_pkg::READ and gpb_pkg::NOP operations to axi_mst_32bit_read() and axi_mst_nop() tasks respectively.

The axi_mst_*() tasks are defined in the axi_master_base_seq class. The do_mst_cmd() is a generic task defined in this class, and is actually called to issue the AXI master transactions to the UVM agent, as shown in Figure 10.

Verification engineers should evaluate what types of transaction need to be supported in the project and create this generic task. Then other tasks in this class could call this generic task with different address, data, and other arguments. The task would generate corresponding transactions to AXI bus agent. For example, as shown in Figure 10, this class defines axi_mst_*() tasks that support 32 bit and 64 bit single beat bus transaction, 64-bit burst transaction, and the bus delay transaction (A.K.A no operation, or NOP). These types of transaction are required to be supported in the design. All these tasks call the generic task do_mst_cmd(). The arguments passed to the do_mst_cmd() task would tell this is a 32 bit or 64 bit transaction, a single or burst transaction, or a NOP transaction. The example in Figure 10 shows how the axi_mst_32bit_write() is defined and how it is mapped to the do_mst_cmd() task.

Figure 10 - axi_master_base_seq class

```
import axi_uvm_pkg::*;

class axi_master_base_seq extends
  uvm_sequence #(axi_uvm_master_trans #(AXI_DATA_WIDTH,
                                        AXI_ADDR_WIDTH));

  `uvm_object_utils(axi_master_base_seq)
  `uvm_declare_p_sequencer(axi_uvm_master_sequencer #
                                        (AXI_DATA_WIDTH,
                                         AXI_ADDR_WIDTH))

  // generic task to support 32 bit, 64 bit, burst,
  // and NOP transactions
  extern virtual task do_mst_cmd(
    input axi_command                    cmd_arg,
    input bit    [AXI_ADDR_WIDTH-1:0]    addr_arg,
    inout bit    [AXI_DATA_WIDTH-1:0]    data_arg[],
    input bit    [AXI_DATA_WIDTH/8-1:0]  wstrb_arg[],
    input axi_length                     burst_length_arg,
    input axi_size                       burst_size_arg
  );

  extern virtual task axi_mst_32bit_write(input bit [31:0] addr,
                                          input bit [31:0] data);

  extern virtual task axi_mst_32bit_read(input  bit [31:0] addr,
                                         output bit [31:0] data);

  extern virtual task axi_mst_64bit_write(input  bit [31:0] addr,
                                          output bit [63:0] data);

  extern virtual task axi_mst_64bit_read(input  bit [31:0] addr,
                                         output bit [63:0] data);

  extern virtual task axi_mst_64bit_write_burst(
    input bit [31:0] addr,
    input bit [31:0] data[]
  );
  extern virtual task axi_mst_64bit_read_burst(
    input  bit [31:0] addr,
    output bit [63:0] data[]
  );

  extern virtual task axi_mst_nop(input int unsigned cycles);
  …
endclass: axi_master_base_seq

task axi_master_base_seq::axi_mst_32bit_write(
  input bit [31:0] addr,
  input bit [31:0] data
);
  bit [63:0] m_data[]  = new[1];
  bit [7:0]  m_wstrb[] = new[1];
    …
  // call the AXI master generic task
  do_mst_cmd(AXI_WR_INC,
             addr,
             m_data,
             m_wstrb,
             AXI_LEN_1,
             AXI_BURST_SIZE_64BIT);

  m_data.delete();
  m_wstrb.delete();

endtask: axi_mst_32bit_write

task axi_master_base_seq::do_mst_cmd (
  input axi_command                cmd_arg,
  input bit   [AXI_ADDR_WIDTH-1:0]  addr_arg,
  inout bit   [AXI_DATA_WIDTH-1:0]  data_arg[],
  input bit   [AXI_DATA_WIDTH/8-1:0]wstrb_arg[],
  input axi_length                 burst_length_arg,
  input axi_size                   burst_size_arg
);
  // generate AXI master transaction. randomize it, and send to
  // UVM agent
  `uvm_create(req)

   assert (req.randomize() with {
        req.cmd_type     == cmd_arg;
        req.address      == addr_arg;
        req.burst_length == burst_length_arg;
        req.burst_size   == burst_size_arg;
```

```
      }) else begin
        `uvm_fatal(get_type_name(), "Randomization failed!")
      end
  …
  `uvm_send(req)
endtask: do_mst_cmd
```

Note that in this example we see how we use the task to generate the transaction and pass down to the sequencer (axi_uvm_master_sequencer in this case). Also note that the randomization constraints are assigned in the generic task thru the arguments. For a control oriented design, there are not many constraints need to be assigned. Most API tasks in our example only require address and data. While an engineer is developing the generic task, he or she is actually simplifying the legal constraint combinations, which is one of the biggest challenges for the test writers to figure out. By developing the generic task, the verification engineer is filtering out those unnecessary constraints that the design does not support.

In this section, the code examples demonstrated how a specific bus interface API tasks are created, and how they are mapped to the higher-level, generic transactions.

5.4 Project Virtual Sequencer and Project Base Sequence

The Project Virtual Sequencer (prj_vsqr) holds the handles of all sequencers that were instantiated in testbench. It also holds the handle of top-level interface (top_if). The top_if is instantiated in the top-level testbench. All other sequencers are instantiated in the top level UVM environment.

Including all of the sequencer handles in the prj_vsqr, we can create a project-wide base sequence (prj_base_sequence), which defines all the bus interface API tasks that would be used in the entire project. In other words, any test sequence that is extended from prj_base_sequence can use all the API tasks defined in the base sequence. Therefore test writers can quickly create a test and access any bus interfaces that are supported in the testbench by calling these API tasks, as we demonstrated in Figure 4. The details of the project base sequence will be discussed in Section 5.7.1.

Figure 11 shows the project virtual sequencer includes all the sequencer handles and the top-level interface.

Figure 11 - Project Virtual Sequencer

```
class prj_virtual_sequencer extends uvm_sequencer
                                    #(uvm_sequence_item);
  `uvm_component_utils(prj_virtual_sequencer)
  virtual top_if top_if;

  axi_uvm_master_sequencer axi_mst_sqr;
  axi_uvm_slave_sequencer  axi_slv_sqr;

  gpb_sequencer gpb_sqr;
  gst_sequencer gst_sqr;
  …
endclass : prj_virtual_sequencer
```

5.5 C to SV Translation Sequences

In the section above, we briefly discussed how to create the project base sequence that includes all the API tasks needed to create a UVM SV test. In this section, we are going to demonstrate how to map a C API task to a UVM SV API task. This means we can reuse those UVM API tasks in the C test sequences.

The C to SV translation sequences map the transactions received from the top-level interface to the Generic Transaction Translation Layer. For the GPB example, the sequence item, gpb_item, will be generated when the GPB task is called. Figure 12 shows an example of how the C to SV translation sequences mapped to GPB transactions:

Figure 12 - GPB C to SV Translation Sequence (gpb_c2sv_xl_seq)

```
`define sig p_sequencer.gpb_c_if0

class gpb_c2sv_xl_vseq extends gpb_base_vseq;

  `uvm_object_utils(gpb_c2sv_xl_vseq)
  `uvm_declare_p_sequencer(gpb_sequencer)
  …
  task body();
    forever begin
      @(`sig.cmd_rdy);

       case (`sig.cmd)
          WRITE: begin
            gpb_write(`sig.addr, `sig.data);
          end
          READ: begin
            gpb_read(`sig.addr, `sig.data);
          end
          NOP: begin
            gpb_nop(`sig.nop_cycle);
          end
        endcase

        -> `sig.cmd_done;
      end
  endtask: body

endclass: gpb_c2sv_xl_vseq

`undef sig
```

Notes that the gpb_c2sv_xl_vseq declares the p_sequecner as gpb_sequencer type. The gpb_sequencer is listed in Figure 13. The gpb_c_if is the interface defined to pass and synchronize the transactions in between the C and SV domains.

Figure 13 - gpb_sequencer

```
class gpb_sequencer extends uvm_sequencer #(gpb_item);
  `uvm_component_utils(gpb_sequencer)

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if (!uvm_config_db#(virtual gpb_c_if)::get(this, "",
                        "gpb_c_if0", gpb_c_if0)) begin
      `uvm_info(get_type_name(), "gpb_c_if is not set!",
                UVM_LOW)
    end
  endfunction: connect_phase

endclass: gpb_sequence
```

In the connect_phase() task of the gpb_sequencer, the GPB C interface (gpb_c_if) is obtained thru the uvm_config_db::get() static method. The translation sequence declares the p_sequencer which points to the gpb_sequencer. The p_sequencer holds the handle of the gpb_c_if. Figure 14 shows a quick view of the gpb_c_if. We will explain how this sub-interface is integrated into the top-level interface in the next section.

Figure 14 - GPB C Interface (gpb_c_if)

```
interface gpb_c_if();

  // parameter package defines the GPB constants
  import gpb_params_pkg::*;
  import gpb_pkg::*;

  gpb_op_type cmd;
  bit [GPB_ADDR_MSB:GPB_ADDR_LSB] addr;
  bit [GPB_DATA_MSB:GPB_DATA_LSB] data;
  int unsigned                    nop_cycle;

  // synchronization events and semaphore
  event      cmd_rdy;
  event      cmd_done;
  semaphore  tr_sm;

endinterface: gpb_c_if
```

In Figure 15, between the fork-join_none block, it shows how the C to SV translation sequences are launched from the run_phase task of the test. The c_main_test class is extended from prj_base_test, which is the base test for the project. The orange colored code shows how a C test is launched from SVTB. More details will be discussed in the Section 5.7.3.

Figure 15 - c_main_test

```
class c_main_test extends prj_base_test;
  `uvm_component_utils(c_main_test)

  virtual task run_phase(uvm_phase phase);
    fork
      begin: start_gpb_c2sv_xl_vseq
        gpb_c2sv_xl_vseq vseq0;
        vseq0 = gpb_c2sv_xl_vseq::type_id::create(…);
        vseq0.start(vsqr.gpb_sqr);
      end: start_gpb_c2sv_xl_vseq

      begin: start_gst_c2sv_xl_vseq
        gst_c2sv_xl_vseq vseq0;
        vseq0 = gst_c2sv_xl_vseq::type_id::create(…);
        vseq0.start(vsqr.gst_sqr);
      end: start_gst_c2sv_xl_vseq
      …
    join_none

    // Start the c_main_seq_wrapper on the project virtual
    // sequencer (prj_vsqr), which is instantiated
    // in prj_base_test
    begin
      c_main_seq_wrapper seq0;
      seq0 = c_main_seq_wrapper::type_id::create("seq0", …);
      seq0.start(prj_vsqr);
    end

  endtask

endclass: c_main_test
```

5.6 Top-Level Interface

In this section we are going to demonstrate how to integrate communication interfaces between C and SV.

As shown in Figure 16, the top-level interface includes multiple interface DPI files, such as gpb_dpi.sv and gst_dpi.sv. The uvm_rpt.sv is the DPI task files for exporting the UVM report functions to C. This file will be discussed in Section 5.7.2.

Figure 16 - Top-Level Interface

```
interface top_if(…);
  // include all exported dpi tasks and C-SV interfaces
  `include "gpb_dpi.sv"
  `include "gst_dpi.sv"
  `include "uvm_rpt.sv"
  // import C functions to SV domain, and map the C function
  // names to SV task names
  import "DPI-C" context task c_main_seq = c_main_seq_dpi();
  import "DPI-C" context task c_function_2 = sv_seq_2();
  …
endinterface: top_if
```

As shown in Figure 17 for the GPB DPI example, each DPI file includes the followings:

- Import statements that map the C function names to SV tasks names.
- Export DPI tasks that will be called from C domain. For the GPB interface, the C to SV communication is thru the sub-interface gpb_c_if. Basically, when C calls the gpb_write() function, it will actually calls the gpb_write_dpi() in SV domain. In the gpb_c_if, the command will be assigned to gpb_pkg::WRITE, and the address and data will be passed from C domain. Then a semaphore key is retrieved. This is to ensure no race condition would occur when multiple C functions issue transactions to the same interface. Then the cmd_rdy event is triggered. Once it's triggered, it will wait until the lower level translation sequence finishing the write transaction. In this case, the gpb_c2sv_xl_vseq triggers the cmd_done event. Before the control exits out of the gpb_write_dpi() task, the semaphore key is dropped back so that the other threads can issue the next transaction to this interface. The handshaking mechanism for the cmd_rdy and cmd_done events can be found in Figure 12 and Figure 17.
- Instantiation of the C interfaces. In Figure 17, we instantiate the GPB C interface (gpb_c_if0) and it is included in the top-level interface (top_if). This structure is called **"interface of interface"**.
- Setting of the interface into UVM configure database. This is done with the uvm_config_db::set() static method defined in the UVM. Once the interface is set, other UVM components can get the virtual interface with uvm_config_db::get() method. An example has been shown in Figure 13, gpb_sequencer.

Figure 17 - gpb_dpi.sv

```
// One important restriction exists. Class member functions
// cannot be exported, but all other SystemVerilog
// functions can be exported. So we define all the
// export tasks/functions in this file, which can be
// included in the top-level interface file, top_if.sv,
// so that they can be globally visible from C code.

// map exported task names between C and SV
export "DPI-C" gpb_write = task gpb_write_dpi;
export "DPI-C" gpb_read  = task gpb_read_dpi;
export "DPI-C" gpb_nop   = task gpb_nop_dpi;

// instantiate and set gpb_c_if
gpb_c_if gpb_c_if0();

// gpb_write_dpi() task
task automatic gpb_write_dpi (
  input int unsigned addr,
  input int unsigned data
);

  gpb_c_if0.tr_sm.get(1); // blocks until get one key

  gpb_c_if0.cmd     = gpb_pkg::WRITE;
  gpb_c_if0.addr    = addr;
  gpb_c_if0.wr_data = data;

  ->gpb_c_if0.cmd_rdy;
  @gpb_c_if0.cmd_done;

  gpb_c_if0.tr_sm.put(1); // put a key back

endtask: gpb_write_dpi

// gpb_read_dpi() task
task automatic gpb_read_dpi (
  input  int unsigned addr,
  output int unsigned data
);

  gpb_c_if0.tr_sm.get(1); // blocks until get one key

  gpb_c_if0.cmd  = gpb_pkg::READ;
  gpb_c_if0.addr = addr;

  ->gpb_c_if0.cmd_rdy;
  @gpb_c_if0.cmd_done;

  data = gpb_c_if0.rd_data;

  gpb_c_if0.tr_sm.put(1); // put a key back

endtask: gpb_read_dpi

// gpb_nop_dpi() task
task automatic gpb_nop_dpi (input int unsigned cycles=0);

  gpb_c_if0.tr_sm.get(1); // blocks until get one key

  gpb_c_if0.cmd       = gpb_pkg::NOP;
  gpb_c_if0.nop_cycle = cycles;

  ->gpb_c_if0.cmd_rdy;
  @gpb_c_if0.cmd_done;
  gpb_c_if0.tr_sm.put(1); // put a key back

endtask: gpb_nop_dpi

// set interface and allocate semaphore
initial begin
  uvm_config_db#(virtual gpb_c_if)::
    set(uvm_root::get(), "*", "gpb_c_if0", gpb_c_if0);

  // allocate 1 semaphore key
  gpb_c_if0.tr_sm = new(1);
end
```

The code snippet illustrated in this section provides the solutions for the challenge mentioned in Section 3.6 - Building C and SV Transaction Interface.

5.7 C and SV Test Sequences

5.7.1 Project Base Sequence

Before a test sequence can be coded as in Figure 3 and Figure 4, the project base sequence needs to be created. It provides all the API tasks need to be supported in the verification environment. Essentially it instantiates all the bus interface base sequences and associate each interface base sequence to the corresponding sequencer in the project virtual sequencer.

Figure 18 shows the GPB base sequence. It provides the generic write, read, and nop API tasks. The GST base sequence class would be similar to this, but it provides the back door transactions to the slave model instead.

Figure 18 - gpb_base_vseq

```
class gpb_base_vseq extends uvm_sequence #(gpb_item);
  ...
  extern virtual task gpb_write(
    input logic[GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
    input logic[GPB_DATA_MSB:GPB_DATA_LSB] data
  );

  extern virtual task gpb_read(
    input  logic[GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
    output logic[GPB_DATA_MSB:GPB_DATA_LSB] data
  );

extern virtual task gpb_nop(int unsigned cycles = 1);
```

```
endclass: gpb_base_vseq

//------------------------------------------------------------
task gpb_base_vseq::gpb_write(
  input logic[GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
  input logic[GPB_DATA_MSB:GPB_DATA_LSB] data
);

  logic [GPB_ADDR_MSB:GPB_ADDR_LSB] m_addr = addr;
  logic [GPB_DATA_MSB:GPB_DATA_LSB] m_data = data;

  start_item(m_req);

  assert(
    m_req.randomize() with {
      m_req.op   == WRITE;
      m_req.addr == m_addr;
      m_req.data == m_data;
    }
  ) else begin
    `uvm_fatal(get_full_name(), "Randomize failed");
  end

  finish_item(m_req);

endtask: gpb_write

//------------------------------------------------------------
task gpb_base_vseq::gpb_read(
  input  logic[GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
  output logic[GPB_DATA_MSB:GPB_DATA_LSB] data
);

  logic [GPB_ADDR_MSB:GPB_ADDR_LSB] m_addr = addr;
  gpb_item m_req = gpb_item::type_id::
    create("m_req", , get_full_name());

  start_item(m_req);

  assert(
    m_req.randomize() with {
      m_req.op   == READ;
      m_req.addr == m_addr;
    }
  ) else begin
    `uvm_fatal(get_full_name(), "Randomize failed");
  end

  finish_item(m_req);

  data = m_req.data;

endtask: gpb_read

//----------------------------------------------------------------
task gpb_base_vseq::gpb_nop(int unsigned cycles = 1);
  bit ok;
  int unsigned m_nop_cycle = cycles;
  gpb_item m_req = gpb_item::type_id::
    create("m_req", , get_full_name());

  start_item(m_req);

  ok = m_req.randomize() with {
    m_req.op == NOP;
    m_req.addr == 0;
    m_req.data == 0;
    m_req.nop_cycle == m_nop_cycle;
  };

  if (!ok) `uvm_fatal(get_type_name(), "Randomization failed")

  finish_item(m_req);

endtask: gpb_nop
```

Figure 19 shows an example how the GPB, GST, and AXI master specific API tasks were defined and instantiated in the project base sequence (prj_base_sequence). The way how other API tasks are defined is similar to those of GPB.

Figure 19 - Project Base Sequence

```
import uvm_pkg::*;
import …

class prj_base_sequence extends uvm_sequence;
  `uvm_object_utils(prj_base_sequence)
  `uvm_declare_p_sequencer(prj_virtual_sequencer)
  gpb_base_vseq   gpb_seq0;
  gst_base_vseq   gst_seq0;

  // AXI base sequences for issuing master/slave transactions
  axi_master_base_seq   mst_seq0;
  axi_slave_base_seq    slv_seq0;

  // pre_body() - it instantiates all base sequences and
  // connects to corresponding sequencers
  extern virtual task pre_body();

  // GPB tasks
  extern virtual task gpb_write(
    input [GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
    input [GPB_DATA_MSB:GPB_DATA_LSB] data
  );

  extern virtual task gpb_read(
    input  [GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
    output [GPB_DATA_MSB:GPB_DATA_LSB] data
  );

  extern virtual task gpb_nop(input int unsigned cycles=10);

  // GST tasks
  extern virtual task gst_bkdr_write(
    input [GST_ADDR_MSB:GST_ADDR_LSB] addr,
    input [GST_DATA_MSB:GST_DATA_LSB] data
  );

  extern virtual task gst_bkdr_read(
    input  [GST_ADDR_MSB:GST_ADDR_LSB] addr,
    output [GST_DATA_MSB:GST_DATA_LSB] data
  );

  // AXI master specific tasks
  extern virtual task axi_mst_64bit_write(
    input bit [31:0] addr,
    input bit [63:0] data,
    input bit [7:0]  wstrb = 8'hFF
  );

  extern virtual task axi_mst_64bit_read(
    input  bit [31:0] addr,
    output bit [63:0] data
  );
  ...
```

```
endclass: prj_base_sequence

//-----------------------------------------------------------
// pre_body() task
//-----------------------------------------------------------
task prj_base_sequence::pre_body();

  // create base sequences
  mst_seq0 = axi_master_base_seq::type_id::create("mst_seq0"…);
  slv_seq0 = axi_slave_base_seq::type_id::create("slv_seq0",…;)
  gpb_seq0 = gpb_base_vseq::type_id::create("gpb_seq0",…);
  gst_seq0 = gst_base_vseq::type_id::create("gpb_seq0",…);
   …
  // hook up the base sequences to the sequencers
  mst_seq0.set_sequencer(p_sequencer.axi_mst_sqr);
  slv_seq0.set_sequencer(p_sequencer.axi_slv_sqr);
  gpb_seq0.set_sequencer(p_sequencer.gpb_sqr);
  gst_seq0.set_sequencer(p_sequencer.gst_sqr);
  …
endtask: pre_body

//-----------------------------------------------------------
// Top-level API tasks used in the test sequences.
// Each task instantiates its base sequence, which is
// imported from its library package
//-----------------------------------------------------------
task prj_base_sequence::gpb_write(
  input [GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
  input [GPB_DATA_MSB:GPB_DATA_LSB] data
);
  gpb_seq0.gpb_write(addr, data);
endtask: gpb_write

task prj_base_sequence::gpb_read(
  input  [GPB_ADDR_MSB:GPB_ADDR_LSB] addr,
  output [GPB_DATA_MSB:GPB_DATA_LSB] data
);
  gpb_seq0.gpb_read(addr, data);
endtask: gpb_read

task prj_base_sequence::gpb_nop(input int unsigned cycles=10);
  gpb_seq0.gpb_nop(cycles);
endtask: gpb_nop

task prj_base_sequence::gst_bkdr_write(
  input [GST_ADDR_MSB:BST_ADDR_LSB] addr,
  input [GST_DATA_MSB:BST_ADDR_LSB] data
);
  gst_seq0.gst_bkdr_write(addr, data);
endtask: gst_bkdr_write

task prj_base_sequence::gst_bkdr_read(
  input  [GST_ADDR_MSB:BST_ADDR_LSB] addr,
  output [GST_DATA_MSB:BST_DATA_LSB] data
);
  gst_seq0.gst_bkdr_read(addr, data);
endtask: gst_bkdr_read

task prj_base_sequence::axi_mst_64bit_write(
  input bit [31:0] addr,
  input bit [63:0] data,
  input bit [7:0]  wstrb = 8'hFF
);
  mst_seq0.axi_mst_64bit_write(addr, data, wstrb);
endtask: axi_mst_64bit_write

task prj_base_sequence::axi_mst_64bit_read(
  input  bit [31:0] addr,
  output bit [63:0] data
);
  mst_seq0.axi_mst_64bit_read(addr, data);
endtask: axi_mst_64bit_read
…
```

In the pre_body() task, we create the base sequence for each bus interface. Each base sequence includes all the API tasks required for that interface. Then we associate each base sequence with the corresponding sequencer with the set_sequencer() task. This task is inherited from the uvm_sequence.

In the example, the gpb_write() task calls gpb_seq0.gpb_write(). The gpb_seq0 has the type of the gpb_base_vseq which was created in the pre_body(). When the gpb_write() task is called, it generates the gpb_item which has the operation type defined as GPB_PKG::WRITE. The gpb_write() task will be translated/mapped to axi_mst_32bit_write() as shown in Figure 8.

The purpose of defining the prj_base_sequence is so that any test sequences that extend from this class inherit all the API tasks that are already defined for this project. Test writers can quickly use the pre-defined API tasks to create new tests.

The examples demonstrated in the section provide the solutions for the challenge mentioned in Chapter 3.3- Mapping Generic Transactions to Bus Interface Specific API Tasks.

5.7.2 Integrating UVM Reporting Macros in C

As the section 3.7 addressed, one of the challenges is to use the UVM reporting macros in the same way in C tests. Under the hood, those UVM macros actually call the reporting functions. So if those reporting functions are exported to C domain, and a set of macros are defined with the same uvm_*() names, we are virtually using those UVM macros in C.

In Figure 20, the uvm_rpt.sv defines the wrapper functions which call out the UVM message macros, such as uvm_info(), uvm_error(), and uvm_fatal(). These wrapper functions are exported to C domain. Then a set of C macros are defined with the same UVM macro names, and to make each C macro calls out the exported SV function. With this method, we can use the UVM message macros in the same way in C code. Figure 21 shows how the C macros call the exported SV functions.

Figure 20 - uvm_rpt.sv

```
// This file can be included in the top-level interface to
// TB module. The functions listed below provide interface
// to C so that we can use
// the uvm_info()/uvm_error()/uvm_error() macros in C code.
// User will need to include the uvm_rpt.h
// in the C compile in order to use the UVM macros
// in exact the same way.

`include "uvm_macros.svh"

//---------------------------------------------------------
// uvm_rpt_info()
//---------------------------------------------------------
export "DPI-C" function uvm_rpt_info;

function uvm_rpt_info(string id,
                      string message,
                      int verbosity = UVM_MEDIUM);
  `uvm_info(id, message, verbosity)
endfunction: uvm_rpt_info

//---------------------------------------------------------
// uvm_rpt_error()
//---------------------------------------------------------
export "DPI-C" function uvm_rpt_error;

function uvm_rpt_error(string id, string message);
  `uvm_error(id, message)
endfunction: uvm_rpt_error
```

```
//-------------------------------------------------------
// uvm_rpt_fatal()
//-------------------------------------------------------
export "DPI-C" function uvm_rpt_fatal;

function uvm_rpt_fatal(string id, string message);
  `uvm_fatal(id, message)
endfunction: uvm_rpt_fatal
```

Figure 21 - uvm_rpt.h

```
//----------------------------------------------
// C version of uvm message services
//----------------------------------------------
#define UVM_NONE   0
#define UVM_LOW    100
#define UVM_MEDIUM 200
#define UVM_HIGH   300
#define UVM_FULL   400
#define UVM_DEBUG  500

#define uvm_info(id, message, verbosity)
  uvm_rpt_info(id, message, verbosity);
#define uvm_error(id, message) uvm_rpt_error(id, message);
#define uvm_fatal(id, message) uvm_rpt_fatal(id, message);
```

5.7.3 Launch C Test from UVM Sequence

Both C and UVM sequences need to be launched from the SV simulation environment. The C test function, c_main_seq(), in Figure 3, is actually started from the SVTB. Figure 22 demonstrates how a SV sequence wrapper is created to start the C function from an UVM test.

Figure 22 - Start the C test sequence from UVM SV test

```
// SV sequence wrapper which calls the C imported task
// from top level interface
class c_main_seq_wrapper extends prj_base_sequence;
  task body();
```

```
    // kick off c_main_seq() function in C domain.
    // Note that p_sequencer is project virtual sequencer
    // (prj_vsqr), which holds a handle of top_if.
    // The c_main_seq_dpi() task is imported in top_if.
    p_sequencer.top_if0.c_main_seq_dpi();
  endtask: body
endclass: c_main_seq_wrapper

// top level interface
interface top_if;
  import "DPI-C" context task c_main_seq()=c_main_seq_dpi();
  …
endinterface

// Start the c_main_seq_wrapper on the project virtual
// sequencer (prj_vsqr), which is already instantiated
// in prj_base_test
class c_main_test extends prj_base_test;
  …
  virtual task run_phase(uvm_phase phase);
    c_main_seq_wrapper seq0;
    …
    // fire off C-SV translation sequences as
    // described in Figure 15
    fork
      …
    join_none

    // start the C test sequence
    seq0 = c_main_seq_wrapper:: type_id::create("seq0", …);
    seq0.start(prj_vsqr);
  endtask

endclass: c_main_test
```

With the testbench infrastructure introduced here, users are able to launch the C and SV test sequences as illustrated in Figure 3 and Figure 4.

The main test flow are driven by C or UVM sequences, and the API task are used in the tests instead of UVM do macros. This chapter covers the challenges mentioned in Section 3.1- Driving Main Test Flow with C, and Section 3.2- Creating UVM Sequences with API Tasks Instead of UVM “do” Macros.

Also the existing C tests and new developed UVM sequences can be reused in this testbench. This provides a solution for the challenge mentioned in 3.4 - Supporting Reuse of High-Level C and UVM Tests.

5.7.4 Launch Multiple C and SV Mixed Test Sequences

As we mentioned earlier in Section 3.6- Building C and SV Transaction Interface, one of the requirements is to launch multiple threads of code simultaneously. With the testbench architecture, users are able to fire multiple C and/or SV test sequences with SV fork-join blocks.

Figure 23 is a single C process function that consists of a few GPB and GST transactions. This C test function will be duplicated three times. They are launched along with another UVM sequence simultaneously in Figure 24.

Figure 23 - Single C process function

```
int single_proc(int proc_id) {

  uint32 addr, wr_data, rd_data;

  //-------------------------------------------
  // GPB transaction tests
  //-------------------------------------------
  switch (proc_id) {
    case 1:
      addr = ADDR1;
      break;

    case 2:
      addr = ADDR2;
      break;

    case 3:
      addr = ADDR3;
      break;

    default:
     sprintf(msg, "Invalid proc_id=%0d\n", proc_id);
     uvm_fatal(__func__, msg)
     break;
  }

  wr_data = proc_id;
  gpb_write(addr, wr_data);
  gpb_read(addr, &rd_data);

  if (wr_data != rd_data) {
    sprintf(msg, "\nProc %0d - GPB read data mis-compared,
              addr=%x, rd_data=%x, proc_id, expt_data=%x\n",
    addr, rd_data, wr_data);
    uvm_error(__func__, msg)
  }

  gpb_nop(proc_id);

  //-----------------------------------------
  // GST transaction tests
  //-----------------------------------------
  addr = proc_id;
  wr_data = proc_id;

  gst_bkdr_write(addr, wr_data);
  gst_bkdr_read(addr, &rd_data);

  if (wr_data != rd_data) {
    sprintf(msg,
      "Proc %0d - GST backdoor read data mis-compared,
      addr=%x, rd_data=%x, expt_data=%x", proc_id, addr,
      rd_data, wr_data);
    uvm_error(__func__, msg)
  }
}
```

Figure 24 - Example of Launching Multiple C/SV Test Sequences

```
interface top_if(…);
  import "DPI-C" context single_proc =
    task single_proc_dpi(int proc_id);
  …
endinterface

class multi_procs_vseq extends prj_base_sequence;

  `uvm_object_utils(multi_procs_vseq)
  `uvm_declare_p_sequencer(prj_virtual_sequencer)

  virtual task body();
    // fork off multiple processes, proc_1-3 are C
    // driven sequences
    // proc_4 is SV test
    fork

      begin: proc_1
        p_sequencer.top_if0.single_proc_dpi(1);
      end

      begin: proc_2
        p_sequencer.top_if0.single_proc_dpi(2);
      end

      begin: proc_3
        p_sequencer.top_if0.single_proc_dpi(3);
      end

      begin: proc_4
        gpb_write(FING_MODE_CNTL_ARY4, 4);
        gpb_read_chk(FING_MODE_CNTL_ARY4, 4);
        gpb_nop(4);
        gst_bkdr_write(4,4);
        gst_bkdr_read_chk(4,4);
      end
    join
  endtask: body
endclass: multi_procs_vseq
```

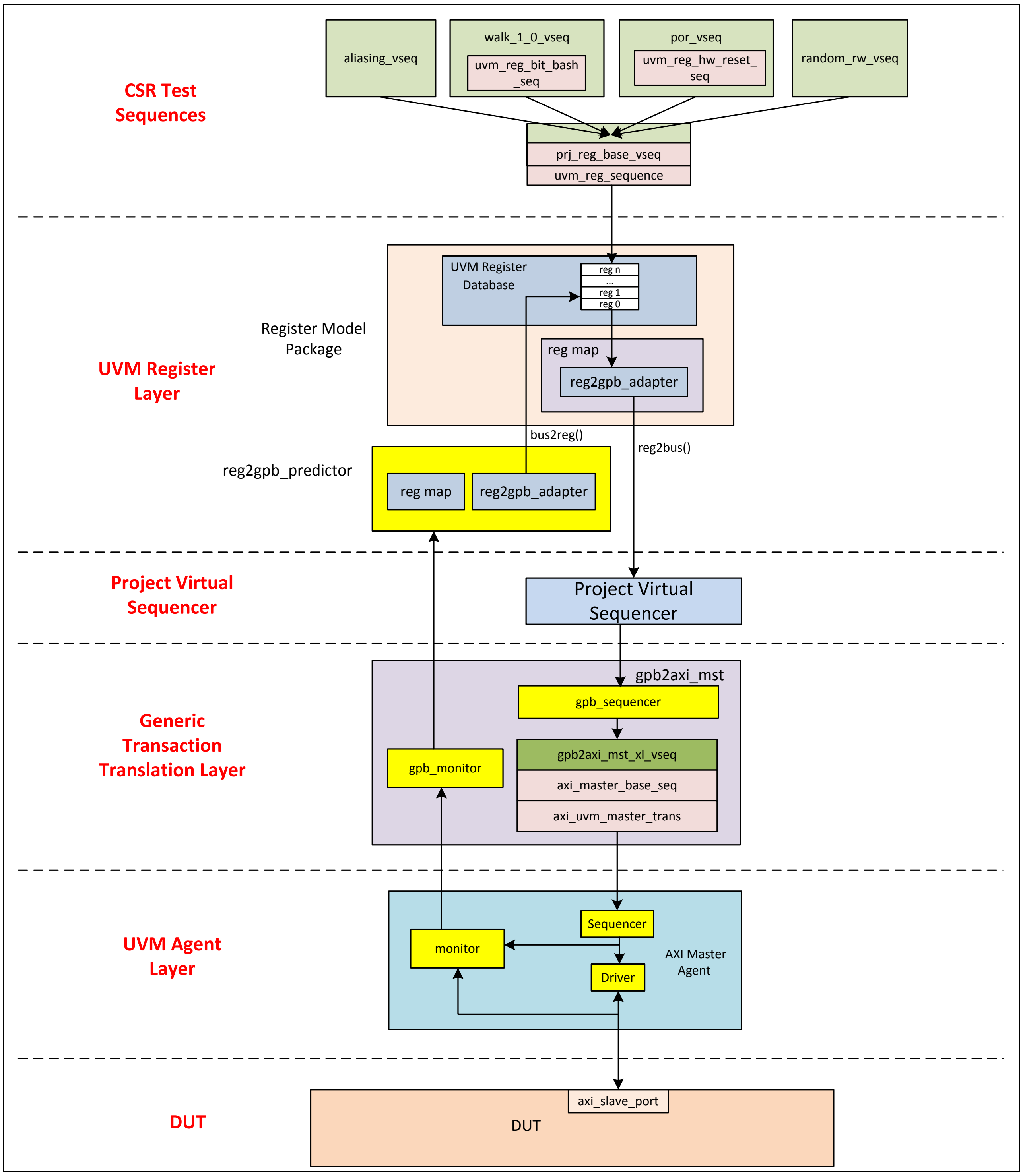
Note that the C function, single_proc(), is imported in the top-level interface (top_if), and is mapped to single_proc_dpi() SV task name, so that it can be referenced in the multi_procs_vseq class through the p_sequencer.

5.7.5 UVM Register Layer

The UVM library provides many test sequences, components, and register model base classes to facilitate the register testing task. As one of the challenges addressed in Section 3.8- Integrating UVM Register Layer, the UVM register layer cannot be used as is. It requires some modifications in the testbench in order to use the register layer.

With the same testbench introduced so far, the verification engineer can easily add the UVM register layer on top of the Generic Transaction Translation Layer (GTTL) to integrate the register model package. The register layer receives the UVM register read/write transactions from the higher level test sequences, translates them to GPB sequence items, and then passes them down to the UVM Agent Layer. Figure 25 shows the details of how the UVM register layer is integrated into the testbench.

Figure 25 - UVM Register Layer Integration Diagram



Note that in our testbench example, the register layer receives transactions from UVM sequences, not from C. The register layer can be placed in parallel with the Project Virtual Sequencer Layer as shown in Figure 5.

Due the huge number of registers existing in a common design, the register model package is usually auto-generated by a tool or script. The UVM library provides all the base classes for the registers, fields, and address map that are used to define the register models. The tool flow of generating the register package is beyond the scope of the discussion.

Some register test sequences are available from UVM library. Users can use them directly, or to override them to customize their needs.

In UVM register package, there are three structural bus agent integration approaches for keeping the register model's mirror values in sync with the DUT: implicit prediction, explicit prediction, and passive. [1]

The implicit prediction is also called **auto prediction** by the industry. It only requires of integration the register model with one or more bus sequencers. The register database mirror is updated whenever a register transaction issued from a register sequence/register model. The integration is simple and quick, but it does not update the register database if a register operation is issued from another bus agent that is not integrated with the register model.

The passive prediction only requires the integration of the bus monitor that is connected to the register model. Because the register model is not integrated with a bus sequencer, the back door transactions, such as set, get and randomize commands, will not be captured, but only the bus transactions (front door transactions) captured on the interface will update the register model.

By default, the register model uses a process called "explicit prediction" to update the register database each time a read or write transaction that the model has generated completes. Explicit prediction requires the use of an uvm_reg_predictor component [3]. It will observe all bus operations, whether they originated from the register model or a third-party bus agent, and thus appropriately update the corresponding mirror values [1]. It is the recommended register model integration method.

The bus monitor in the UVM agent layer captures transactions from the bus interface and passes them to Generic Transaction Translation Layer. The GPB sequence items then are built by the gpb_monitor and passed to the predictor (reg2gpb_predictor). The predictor updates the register database to reflect the current register values in the design.

The register predictor reg2gpb_predictor in Figure 25 is setup to update the register model database whenever there is a read/write access. It has two major components, register map and adapter. The register map is created in the register model package. The adapter is created in the environment. The reg2gpb_predictor holds the handles of both.

There are functions, reg2bus() and bus2reg(), in the adapter need to be defined. The reg2bus() function translates the register transaction (uvm_reg_bus_op) to the GPB sequence item. The function is called when a test sequence issues a register transaction to DUT. The bus2reg()

function translates the GPB transaction to register transaction when a transaction is captured by the bus monitor and passed to gpb_monitor. Figure 26 shows the implementation of the adapter.

Figure 26 - reg2gpb_adapter

```
class reg2gpb_adapter extends uvm_reg_adapter;

  `uvm_object_utils(reg2gpb_adapter)

  extern function new(string name = "reg2gpb_adapter");

  extern virtual function uvm_sequence_item reg2bus(
    const ref uvm_reg_bus_op rw
  );

  extern virtual function void bus2reg(
    uvm_sequence_item bus_item,
    ref uvm_reg_bus_op rw
  );

endclass: reg2gpb_adapter

//-------------------------------------------------------------
function reg2gpb_adapter::new(string name = "reg2gpb_adapter");
  super.new(name);
endfunction

//--------------------------------------------------------------
function uvm_sequence_item reg2gpb_adapter::reg2bus(
  const ref uvm_reg_bus_op rw
);
  gpb_item gpb_item0 = gpb_item::type_id::create("gpb_item0");
  gpb_item0.op = (rw.kind == UVM_READ)? gpb_pkg::READ
                                    : gpb_pkg::WRITE;
  gpb_item0.addr = rw.addr;
  gpb_item0.data = rw.data;

  // display the item translated
  `uvm_info(...)

  return gpb_item0;
endfunction: reg2bus

//--------------------------------------------------------------
function void reg2gpb_adapter::bus2reg(
  uvm_sequence_item bus_item,
  ref uvm_reg_bus_op rw
);
  gpb_item gpb_item0;
  string msg = "";

  if (!$cast(gpb_item0, bus_item)) begin
    `uvm_fatal(…)
    return;
  end

  if (gpb_item0.op == gpb_pkg::WRITE) begin
    rw.kind = UVM_WRITE;
  end else if(gpb_item0.op == gpb_pkg::READ) begin
    rw.kind = UVM_READ;
  end else begin
    `uvm_fatal(...)
  end

  rw.addr = gpb_item0.addr;
  rw.data = gpb_item0.data;
  rw.status = UVM_IS_OK;

  // display the item translated
  `uvm_info(...)

endfunction: bus2reg
```

The instantiation of the predictor and the register layer connection is done in the testbench environment, which will be shown in the next section.

A monitor must exist in the testbench to support the explicit prediction. Since the register layer is built on top of the Generic Transaction Translation Layer, the monitor named gpb_monitor needs to be created to translate the captured bus transactions to GPB transactions as shown in Figure 27.

Figure 27 - gpb_monitor

```
import gpb_pkg::*;

// uvm_subscriber has analysis_export
class gpb_monitor extends uvm_subscriber
  #(axi_uvm_master_trans #(AXI_DATA_WIDTH, AXI_ADDR_WIDTH));

  // Provide implementations of virtual methods such as
  // get_type_name and create
  `uvm_component_utils(axi2gpb_monitor)

  uvm_analysis_port #(gpb_item) ap; //analysis port

  // Constructor
  extern function new(string name, uvm_component parent);
  extern virtual function void write(
    axi_uvm_master_trans #(AXI_DATA_WIDTH, AXI_ADDR_WIDTH) t
  );

endclass: gpb_monitor

//-------------------------------------------------------------
function gpb_monitor::new(string name, uvm_component parent);
  super.new(name, parent);
  ap = new("ap", this);
endfunction: new

//-------------------------------------------------------------
function void gpb_monitor::write(
  axi_uvm_master_trans#(AXI_DATA_WIDTH, AXI_ADDR_WIDTH) t
);

  gpb_item gpb_mon_item0;
  gpb_mon_item0 = gpb_item::type_id::create("gpb_mon_item");

  // reconstruct the gpb_item received from analysis_export
  case (t.cmd_type)
    AXI_WR_INC: gpb_mon_item0.op = gpb_pkg::WRITE;
    AXI_RD_INC: gpb_mon_item0.op = gpb_pkg::READ;
    default   : `uvm_fatal(…)
  endcase

  gpb_mon_item0.addr = t.address;

  // extract 32bit data
  if (t.address[2]) begin
    gpb_mon_item0.data = t.data[0][63:32];
  end else begin
    gpb_mon_item0.data = t.data[0][31:0];
  end

  // pass the gpb_item to ap (analysis port)
  ap.write(gpb_mon_item0);

endfunction: write
```

The write method in the monitor needs to be defined to pass the captured transactions on the actual bus (AXI bus in this case) to the predictor for updating the register database. The GPB monitor is instantiated in the AXI master agent (gpb2axi_mst) as shown in Figure 25 and Figure 9.

5.8 Testbench Environment

We have described how each verification component is built in the earlier sections. In this section, we will demonstrate the build phase and connect phase of the testbench environment.

As shown in Figure 28, the verification components are instantiated with the factory create method in the build phase and the connections among the components are done in the connect phase.

Figure 28 - tb_env

```
class tb_env extends uvm_env;
  …
//------------------------------------------
// build_phase() -
//------------------------------------------
function void build_phase(uvm_phase phase);
  super.build_phase(phase);

  // build Generic Transaction Translation Layer
  gpb2axi_mst = gpb2axi_mst::type_id::create(…);
  gst2axi_slv = gst2axi_slv::type_id::create(…);

  // build register layer and components
  // predictor is a parameterized UVM base class
  gpb2reg_predictor = uvm_reg_predictor#(gpb_item)::type_id::create(…);
  reg2gpb_adapter = reg2gpb_adapter::type_id::create(…);

  // build project virtual sequencer
  prj_vsqr = prj_virtual_sequencer::type_id::create(…);

endfunction: build_phase

//------------------------------------------
// connect_phase() -
//------------------------------------------
function void connect_phase(uvm_phase phase);

  super.connect_phase(phase);
  // connect GPB and AXI master sequencers and components
  prj_vsqr.axi_mst_sqr = axi_mst_agent0.sequencer;
  prj_vsqr.gpb_sqr = gpb2axi_mst.gpb_sqr;
  gpb2axi_mst.axi_mst_agent0 = axi_mst_agent0;

  // connect AXI monitor in UVM agent to gpb2axi_mst component
  axi_mst_agent0.monitor.post_resp_port.connect(
    gpb2axi_mst.analysis_export
  );

  // connect GST and AXI slave sequencers and components
  prj_vsqr.axi_slv_sqr = axi_slv_agent0.sequencer;
  prj_vsqr.gst_sqr     = gst2axi_slv.gst_sqr;
  gst2axi_slv.axi_slv_agent0 = axi_slv_agent0;

  // connect register model and related components
  // register model (rm) is instantiated in system
  // configuration object (sys_cfg).
  // associate the GPB sequencer and adapter to the register map
  sys_cfg.rm.PRJ_REG_MAP.set_sequencer(gpb2axi_mst.gpb_sqr,
                                       reg2gpb_adapter);

  // Register prediction component:
  // Replacing implicit register model prediction
  // with explicit prediction
  // based on GPB bus activity observed by the GPB agent monitor
  // Set the predictor map:
  gpb2reg_predictor.map = sys_cfg.rm.PRJ_REG_MAP;

  // Set the predictor adapter:
  gpb2reg_predictor.adapter = reg2gpb_adapter;

  // Disable the register models auto-prediction
  sys_cfg.rm.PRJ_REG_MAP.set_auto_predict(0);

  // Connect the predictor to the bus agent monitor analysis port
  gpb2axi_mst.ap.connect(gpb2reg_predictor.bus_in);

endfunction: connect_phase

endclass: tb_env
```

The register models exist in the system configuration object. It is not listed in the example since it is a common way to pass other configuration parameters from a test to the testbench components.

Since the explicit prediction mechanism is used in the register layer, the set_auto_predict(0) task is called to disable the register model's auto prediction mechanism.

The connections for the register predictor (reg2gpb_predictor), the GPB monitor (gpb_monitor), and the AXI bus monitor are shown in Figure 28 with green colored code.

5.9 Test Flow Reviews

The gpb_write() command will be used in C and UVM SV sequences as examples to trace what will occur in the testbench for the entire test flow. Please refer to Figure 5 for the following two sections.

5.9.1 UVM Test Sequence Flow

When the gpb_write() command is issued from the UVM sequence, the transaction object gpb_item will be created and sent to the gpb_sequencer that resides in project virtual sequencer (prj_vsqr). The prj_vsqr holds the handle of the gpb_sequencer, which points to the gpb_sequencer in the gpb2axi_mst block. The gpb_item will then be translated to axi_uvm_master_trans transaction object. The translation process is implemented thru the gpb2axi_mst_xl_vseq sequence as shown in Figure 8. The axi_uvm_master_trans will be recognized by the sequencer in the AXI master agent, and it will be passed down to the bus driver to wiggle the signals and to communicate with DUT.

5.9.2 C Test Sequence Flow

The C test sequence still needs to be launched from the UVM sequence. As shown in the example of Figure 5, the C test sequence is launched by calling the "p_sequencer.top_if.c_main_seq_dpi()" statement from the UVM SV test sequence. This is just an example for demonstrating how the C function can be called from a SV interface.

In the sv_main_seq class, the UVM p_sequencer is declared to prj_virtual_sequencer type. The prj_virtual_sequencer holds the handle of the top-level interface (top_if), which defines the task c_main_seq. Note that the c_main_seq_dpi() defined in top_if actually calls the imported C function c_main_seq(). The gpb_write() command is then issued from there. The gpb_write() is exported from SV domain, so indeed, C is now calling the gpb_write_dpi() task defined in SV top_if. The gpb_write_dpi() task passes the address and data values to the gpb_c_if, and also triggers the cmd_rdy event as shown in Figure 17. The translation sequence shown in Figure 12 was waiting for the cmd_rdy event being triggered. Once it sees the event turning on, it looks at the command, (in this case, it is gpb_pkg::WRITE), and calls the gpb_write() SV task. The rest test flow occurred at lower-level components is identical to the procedures described in the previous section, UVM Test Sequence Flow. When the gpb_write() is finished in SV domain, the translation sequence gpb_c2sv_xl_vseq then triggers the cmd_done event to notify the gpb_c_if that the task is finished. This would complete the handshaking process, as the code listed in Figure 12. Then the gpb_write_dpi() task is finished, and C gets the control back and is ready for the next task command.

6 Conclusions

In this paper, a testbench architecture that supports C and UVM SV test sequences has been introduced. It allows us to reuse the legacy, task driven C test sequences in the UVM testbench, and it provides an environment to let users to develop new tests with the latest UVM technology.

The entire testbench architecture was designed to create tests with API tasks. The paper demonstrated how a verification engineer could create and integrate multiple bus interface API tasks into one testbench environment.

Test writers can use the do macros, such as `uvm_do_with() and `uvm_do_on_with() to create tests, but those macros are not user friendly. Using these UVM macros to create tests, test writers have to understand all the legal constraint combinations. However, if the testbench is built with the approach demonstrated in this paper, test writers can use the integrated API tasks to quickly create new tests either in C or UVM. They do not need to worry about how to specify those legal constraints for the third party VIPs. The API tasks used in UVM sequences are exported or mapped to C domain, and they can be reused across different project platforms because they are generic high-level tasks. In addition, the test sequences will look much more concise and clean, and much easier to debug, as compare to UVM macro created tests.

The testbench does not limit users to only use API tasks to create tests. For a data path oriented design with complicated bus interface protocol, test writers can still have the freedom to use the standard UVM do macros to specify the complex constraints when creating new SV tests.

UVM reporting macros provide many good features for the message reporting mechanism, such as `uvm_info(), and `uvm_error(). By exporting the wrapper functions from SV to C domain, test writers are able to use theses reporting macros in C code with a similar way as in UVM.

The integration of the UVM register layer into the testbench has been demonstrated. The GPB monitor (gpb_monitor) and the predictor (reg2gpb_predictor) need to be built to update the register database and to reflect the current register values in the register model.

Reusability is the biggest challenge for the fast paced ASIC design verification industry. The testbench architecture presented in this paper extends the reusability beyond the scope of the UVM technology, and across the C and SV language boundary.

7 References

[1] Accellera UVM 1.1 User's Guide

[2] Accellera UVM Reference Manual

[3] Mentor Graphics UVM/OVM Documentation – Verification Methodology Online Cookbook

[4] IEEE P1800/D8 Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language

8 Acknowledgements

I especially would like to thank my project lead and hardware development director Curt Musfeldt for his advice and guidance throughout the idea evolutions and the testbench development. I need to thank him for giving me the opportunities to implement the ideas on real design projects. I also need to thank my manager Tom Funk who has spent time on reviewing my paper, and my department senior director Brian Harms, who created the working environment to let me to extend my professional expertise at Qualcomm.

I also need to thank SNUG paper review committee member Dinesh Tyagi, technical reviewer Jean Fong, and our local Synopsys support Kevin Geiger, who reviewed some very rough drafts and inspired many improvements.

I would like to thank my wife Jennifer, my daughters Ashley and Rachel for their patience and tolerance while I could not spend more time with them during weekends and holidays.

Finally I do need to appreciate God who has provided everything I needed for the paper preparation, and everything else in my life!