2024
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Verification for Everyone

Linking C++ and SystemVerilog

Noel McCarthy, Mixed Signal Verification Engineer, Limerick, Ireland, (noel.mccarthy@analog.com)

Paul Wright, Mixed Signal Verification Engineer, Limerick, Ireland, (paul.wright@analog.com)

***Abstract*—Verification testbenches are powerful tools when developing IC designs but are not always the most "welcoming" for non-verification engineers. When analog designers want to run certain blocks as transistors in a chip-level simulation, or test and evaluation teams want to gain vital insights to ensure their test setups are ready for day 1 of silicon on the bench, making the verification environment user friendly allows others access to this invaluable resource. For our teams, all that is required is knowledge of how to code in C/C++. For numerous projects, we have used a C++ based frontend to the UVM DV environment by utilizing the DPI feature of SystemVerilog.**

In this paper, a description of DPI will be given, with an explanation of how the C++ frontend was implemented and the benefits it brought to our projects.

***Keywords*— C++, SystemVerilog, DPI**

I. Introduction

The continuous drive for innovation and the complexity of the technologies that are being delivered today means we can no longer work in silos. Large project teams must ensure that collaboration between its members is imperative to drive "Right First Time Silicon". Mistakes incur the penalties of time and reputation. Cross-disciplinary engagement becomes more important as project size, complexity, and importance increase.

Mixed Signal Design Verification (MSDV) is one such discipline that straddles the boundaries. Verification engineers spend time pre-tapeout communicating with their analog and digital design colleagues to ensure that no bug "slips through" and that all expected functionality is understood. Likewise, post-tapeout, test and evaluation engineers often request test sequences to better understand the device-under-test (DUT) functionality.

Modern verification testbenches are packed full of powerful features to exercise all the functionality of our designs. They allow us to push the DUT through simulation. This is to give us the confidence that when silicon appears on the bench, it will be highly functional and ready to sample to our customers. But it takes time and expertise to become proficient with these testbenches. SystemVerilog is not a language that is taught universally. What if we could unlock this power and make it easier for our colleagues to utilize this wonderful resource at our fingertips. To provide a platform from which design, evaluation and test engineers, amongst others, can develop testcases that bring the "Wisdom of the Crowd" to the project. Where analog designers can see how their blocks perform at transistor-level when incorporated within a chip-level testbench. Where test and evaluation engineers can perform pre-silicon validation routines so that they can prove their tests will work when silicon arrives.

For these reasons and more, reducing the effort required to effectively use the verification environment makes technical and business sense. The C Programming language is one that is taught in nearly every under-graduate electronic engineering course and as such should be familiar to most engineers working in an IC development role. Universal Verification Methodology (UVM) environments are based on SystemVerilog; a language that is not as prevalent as C. However, [1] defines a Direct Programming Interface (DPI) between SystemVerilog and a foreign language. In the standard, the only foreign language referred to is C. It is therefore possible for both languages to "work together" to provide a verification environment for everyone.

II. Project Description

Non-Verification engineers often find the learning curve of SystemVerilog prohibitive. When given the option to use the verification environment, often they resort to building their own testbenches (losing the functionality of the chip-level testbench) or the verification engineer spends significant time assisting in testcase development. Neither of these options are appealing.

The DPI feature of the SystemVerilog language was found to be a means of opening the verification environment to non-verification engineers by using a familiar language. Any infrastructural maintenance required is handled by the verification team, thus providing a platform where team members can contribute productively without too much effort.

DPI is described in [1] as having two layers – the SystemVerilog layer and that of a foreign language, in this case, C. Both are compiled separately and are independent of each other, with DPI providing the interface between the two. A key feature of DPI is that it provides the ability to Import and Export tasks and functions between the two layers. Functions in C can be imported into SystemVerilog while tasks and functions in SystemVerilog can be exported to C. This allows operations such as writing/reading to/from registers, updating/sensing DUT testbench signals etc., all of which are available to the SystemVerilog testcase writer, to be accessible to the C test writer. Additionally, pre-written tasks / functions in SystemVerilog for more complex operations can also be exported for use within a C test. The C test itself is imported into the SystemVerilog verification environment.

Figure 1 shows an outline of the simulation environment. The testbench is predominantly created in SystemVerilog including all the relevant agents, scoreboard, and the Register Abstract Layer (RAL) model. Connections to/from the DUT are via a Testbench Harness. This standard setup allows for SystemVerilog testcases and sequences to be written to exercise the DUT and gather the relevant metrics as part of the verification effort.

The environment is further extended by using the DPI. Here, testcases written in C, which can use tasks and functions exported from SystemVerilog, can imported into the SystemVerilog environment. The goal is that with an understanding of the DUT, basic knowledge of the testbench and some C coding, an engineer will be quickly writing testcases.

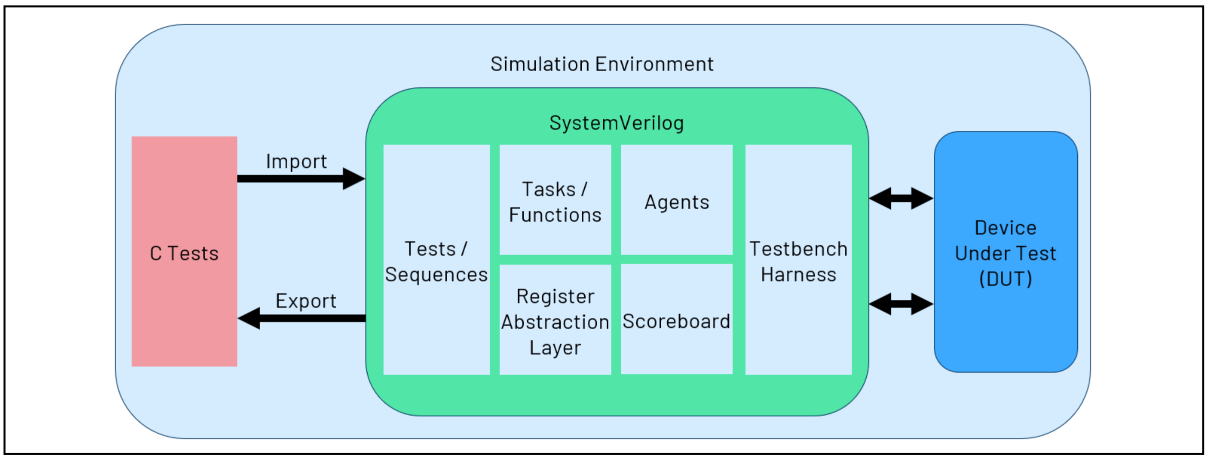

Figure 1 Verification Simulation Environment

In the following sections, it will be shown how the individual parts of the system work relative to C testcases. As an outline

- SystemVerilog Tasks and Functions are written for standard operations and are exported.
- A static C testcase wrapper is created to encapsulate all that is needed from a C point of view.
- A user C testcase is written and included in the static C testcase wrapper.
- A SystemVerilog Base Test for is run, that calls the static C testcase.

This layered approach has provided an efficient means for getting people engaged and productive quickly, by putting the structures in place to handle all of the SystemVerilog and C environment related steps and allow the testcase writer to focus solely on the testcase.

Figure 2 shows several tasks and functions created in SystemVerilog. They are representative of basic operations used in verification and include writing / reading to/from registers, creating a time delay, printing to the logfile and writing / reading real values to / from the testbench. These tasks and functions can be exported for DPI using the export declaration. This will allow them to be used or called by a C testcase. Additionally, C functions, such as a testcase, can be imported.

```
// Register Write Operation
task automatic c_reg_write(input int addr, input int reg_data);
    seq.regWrite(addr, reg_data);
    `uvm_info("C_STIM", $psprintf("WRITE %h-<%h", addr, reg_data), UVM_LOW)
endtask

// Register Read Operation
task automatic c_reg_read(input int addr, output int data);
    seq.regRead(addr, data);
    `uvm_info("C_STIM", $psprintf("READ %h->%h", addr, data), UVM_LOW)
endtask

// Time Delay Operation
task automatic delay_1us(input real number);
    `uvm_info("C_STIM", $psprintf("TB_INFO_COL::DELAY::%fus::TB_INFO_END\n", number), UVM_LOW)
    #(number * 1us);
endtask

// Print Message
function c_tb_print(string message);
   `uvm_info("TB_PRINT", message, UVM_LOW)
endfunction: c_tb_print

// Testbench Write of real value
function write_real(string sig_name, real wdata);
   case (sig_name)
         "rload_a_rout" : tb_if.rload_a_rout    = wdata;
         "rload_b_rout" : tb_if.rload_b_rout   = wdata;
         "v_ref_rout"     : tb_if.v_ref_rout       = wdata;
         default: $display("Error: write_real %s not handled", sig_name);
   endcase
endfunction

// Testbench Read of Real Value
function read_real(string signal_name, output real wdata);
   case (signal_name)
         "rload_a_rin" : wdata = tb_if.rload_a_rin;
         "rload_b_rin" : wdata = tb_if.rload_b_rin;
         "v_ref_rin"     : wdata = tb_if.v_ref_rin;
         default: $display("Error: read_real %s not handled", signal_name);
   endcase
endfunction

// DPI-C Export Tasks and Functions for use in C Code
export "DPI-C" task c_reg_write;
export "DPI-C" task c_reg_read;
export "DPI-C" task delay_1us;
export "DPI-C" function c_tb_print;
export "DPI-C" function write_real;
export "DPI-C" function read_real;

// DPI-C Import testcase from C Code
import "DPI-C" context task main_c_func();
```

Figure 2 SystemVerilog Code Example

A Static C code testcase is used as a wrapper for the user testcase and is shown in Figure 3 . The main constitutes of this file are:

- The header file “svdpi.h” is included. This comes from Annex I of [1]. It “contains the constant definitions, structure definitions, and routine declarations used by SystemVerilog DPI.”
- The user C code file is included. This contains the C code developed by the testcase writer. The main function from that file, “main_c_testcase()” is called from this static C code testcase.
- The functions and tasks exported from SystemVerilog are declared. Note the ‘extern “C”’ in the declaration to ensure that the names have C linkage and are not mangled.
- The standard IO and Library files are included as well as the register map header file for register accesses.

```
// Include additional header files
#include "stdio.h"
#include "stdlib.h"
#include "svdpi.h
#include "reg_map.h"

// Include the user test file
#include "user_main.cpp"

// Typedefs used
typedef unsigned int u32;

// DPI-C exported Tasks and Functions from SystemVerilog
extern "C" void c_reg_write (u32 addr, u32 wdata);
extern "C" void c_reg_read  (u32 addr, u32 *rdata);
extern "C" void write_real (const char* signal_name, double wdata);
extern "C" void read_real  (const char* signal_name, double *rdata);
extern "C" void delay_1us(double number);

int main_c_func()
{
  // Call the C test
  main_c_testcase();
  return 0;
}
```

Figure 3 Static C Code Example

An example user testcase is shown in Figure 4. There are several processes shown here to illustrate the DPI in action.

Firstly, writing a real value from the testbench harness – in this case, v_ref_rout. The “write_real” function is exported from SystemVerilog as seen in Figure 2 so that it will be available for use within the C environment. The function is declared and compiled with the testcase in C in Figure 3. Note at this point, this SystemVerilog function is now available for use in both SystemVerilog and C testcases. In this example, a real value of 2.5 is applied to “v_ref_rout” of our testbench harness. Likewise, there is a function that reads a real value from the harness – in this case “v_ref_rin”. In similar fashion, it is exported, declared, and utilized in the C testcase. Once the value is read, it is available for processing such as checking if it is within limits or in the case shown, using another function to print the value to a logfile.

Another important feature in any complex mixed-signal design is the ability access internal registers. It is imperative that the correct functionality is exercised based on what is written to the part and that the data read is consistent with what is expected. Hence, as shown in the example, writing / reading to / from registers, in this case a SCRATCH or spare register, must be available. It can be noted that delays are inserted within the example testcase for demonstrating purposes.

```
// Example of code from test file in user_main.cpp
int main_c_testcase()
{
  // Variable Declaration
  double vprobe_double;
  u32 my_data;

  // Write / Read real value to the testbench
  write_real("v_ref_rout", 2.5);
  delay_1us(10);
  read_real("v_ref_rin", &vprobe_double);
  c_tb_print("Voltage Probed : %lf \n", vprobe_double);

  // Register Write / Read
  c_reg_write (SCRATCH0, 0x5566);
  delay_1us(10);
  c_reg_read  (SCRATCH0, &my_data);
  delay_1us(10);
  c_tb_print("Data read from Scratch Register : 0x%x \n", my_data);

  return 0;
}
```

Figure 4 User C Code Example

At simulation time, a SystemVerilog base test is run that extends from our standard base test. The key point here is that this base test is static, and the C test writer is essentially oblivious to its existence. Its main function is to execute the static C function that was shown in Figure 3 which includes the user C code from Figure 4. Once the function is executed, a message is printed to indicate that the C testcase has completed. The objection is then dropped so that the SystemVerilog testcase can end.

```
// Example of code from test file in user_main.cpp
class c_base_test extends base_test;
  host_ral_base_seq x_host_ral_base_seq;
  `uvm_component_utils_begin(c_base_test);
  `uvm_component_utils_end;

  function new(string name = "c_base_test", uvm_component parent);
    super.new(name,parent);
  endfunction : new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    x_host_ral_base_seq = host_ral_base_seq::type_id::create("x_host_ral_base_seq");
  endfunction

  virtual task main_phase(uvm_phase phase);
    super.main_phase(phase);
    set_c_stimulus_seq(x_host_ral_base_seq);
    x_host_ral_base_seq.start(env.host_ral_seqr_i);
    main_c_func();
    `uvm_info("Testcase", "Test Complete", UVM_INFO);
    phase.drop_objection(this);
  endtask
endclass
```

Figure 5 SystemVerilog Base Testcase

III. Results

The success of implementing the DPI testcase environment can be seen from several different aspects, some of which may not be overly apparent at first. To share some of the benefits, a view from the disciplines will highlight the attraction of the setup.

Experienced analog designers have highlighted the fact that the learning curve of UVM / SystemVerilog is a barrier to using the verification environment and that simplifying the process allows them to contribute to the verification effort more meaningfully pre-tapeout. When it comes to knowledge of the analog domain, it is the individual block designers that are most familiar with it. As the design progresses, analog designers want to move up the hierarchy to confirm that their blocks are behaving as expected with both the digital and other analog circuitry. The best way of doing so is through the verification environment. This is especially true for designers of analog-heavy mixed-signal blocks who need access to relevant DSP engines in the design. Additionally, our setup allows for blocks to be simulated in behavioral or transistor level. Providing this flexibility has been invaluable in finding unexpected behaviors in the design. Finally, sharing a common verification infrastructure with the analog designers has brought benefits in the de-bugging/fixing/sharing of issues uncovered by the verification team helping to speed up the IC development.

Prior to implementing the DPI architecture, most, if not all, of our measurement teams were reluctant to dig down into our designs from schematic point of view. For one, our designs are developed in a Linux environment and Test and Evaluation work is done on Windows platforms. Therefore, there is some work required to do the initial setup – this is not too onerous but is still required. Once the initial setup is done, there are a few options available. Use the setup for just viewing the design. This lacks the interactivity that truly aids the learning and development process. Start using SystemVerilog. While this approach is perfectly reasonable it comes with the "price tag" of learning how to write testcases in that language. And finally, start writing in C. It may not have all the "bells and whistles" that SystemVerilog has but it provides an extremely quick ramp up in getting started. Quickly the engineer is focusing on the design and the sequence of the instructions, rather than the constructs of the language. Working with the design interactively, accelerates the engineer's knowledge of the part long before the physical design arrives.

Tester time is costly, project time is even more expensive. Having our test engineers simulating their test methods prior to tapeout has shown to reduce the tester debug time. In many cases, the order of writes within test methods are critical. Therefore, checking the sequence prior to silicon arrival has saved hours, if not days, of potential costly debug. This can be done quickly and efficiently by test engineers through simulation, thus allowing fast progress of silicon bring-up and samples production.

From a verification point of view, we have benefited from having members from other disciplines join the team as part of the verification effort. This has allowed us to meet tight tapeout deadlines without a significant amount of additional effort. In fact, it was commented that the expectation was that the members from other disciplines would rely heavily on the verification engineers to maintain momentum in developing testcases but in reality, this was not the case. Occasionally, additions, modifications, clarifications were required within the environment but overall, each member of the team was able to work seamlessly and productively on the projects. It also allowed for the verification engineers to seek feedback from colleagues more familiar with working with physical devices rather than just in the virtual world.

So far, it has been the pre-tapeout effort that has been discussed but post-tapeout, the DPI environment can also prove its worth. When issues occur in first silicon, inevitably the debug work begins. The first stage is to try and recreate the issue in simulation and the top-level testbench is best placed for this. Being able to quickly put a testcase together to identify the failing condition allows the team to put corrective measures in place either through software workarounds or proving out a silicon edit that is required.

Finally, it is the collective wisdom of the team, with varying ranges of experiences, that have been brought together pre-tapeout to ensure that functional first-time silicon is available. We have had several design, test, and evaluation engineers, both experienced and new college graduates, work as part of the mixed signal design verification team. This allowed us to deliver high quality samples to our customers in a timely manner thus allowing them to prove out their systems.

IV. CONCLUSION

The main goal of any IC development is “Right First Time Silicon”. To achieve this on complex SOC designs requires a huge amount of effort from all the team. Collaboration is key and therefore sharing the same verification infrastructure allows for, and drove, meaningful cooperation between the different disciplines. With not too much effort, the power of the verification chip-level testbench was made available for all to use. This has been hugely beneficial, allowing us to draw on different experiences and points of view during verification of our designs. The C++ frontend provides a win-win situation whereby verification benefit from additional team members being made available, analog designers get to “see the bigger picture” of their blocks in the chip level hierarchy and measurement gain invaluable exposure to the design at an early stage to save time when packaged parts arrive. All of this leads to providing our customers with quality silicon.

ACKNOWLEDGMENT

The development of the DPI setup that we use has had several contributors over the years including former colleagues Andy Sha and David Harty. The authors would like to acknowledge and thank them for their efforts. Additionally, the authors would like to thank the design, evaluation and test engineers that have used the setup. Their feedback has helped guide its development and has promoted it to the wider community.

REFERENCES

[1] "IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1-1315, 2018.