Taking UVM to wider user base – the open-source way

Nagasundaram Thillaivasagam,ASIC DV Engineer,VerifWorks,Bangalore, India.(naga@cvcblr.com)

Santhosh Kumar, ASIC DV Engineer, VerifWorks,Bangalore, India.(santhosh@cvcblr.com)

Gurubasappa Kinagi, ASIC DV Engineer, VerifWorks,Bangalore, India.(guruk@cvcblr.com)

***Abstract*—Universal Verification Methodology (UVM) has taken the Design Verification (DV) field by storm! While UVM has been very successful with DV teams, the traditional design and small time verification teams have looked at UVM and a bit taken back by the complexity of UVM. Also there are significant challenges in creating full-fledged UVM environment for non-OOP users. In this paper we present our experience in working with customers coming from different backgrounds and being able to deploy UVM to them via a convenience layer around the standard UVM named *Go2UVM*. To make it clear *Go2UVM* is NOT a script to generate a set of files that users can fill-in, rather it is an OOP layer around standard UVM hiding all the glory details of UVM and providing the first-time UVM users an easy to use procedural interface to UVM. *Go2UVM* is open-sourced, sits on top of standard UVM and hence is 100% in-line with UVM philosophy of test creation.**

***Keywords*— Verilog HDL, System Verilog, UVM, Go2UVM**

I. INTRODUCTION

Universal Verification Methodology (UVM), as defined by the Accellera standard and soon becoming IEEE 1800.2, is getting adopted widely across ASIC design teams. UVM is also getting its flavor in SystemC so that the Electronic System Level (ESL) community can move to UVM-based approach for verifying models. Given the standard way of information flow, well defined test sequencing (in terms of phasing for instance), UVM is attractive to almost all electronics design teams.

However, the full-fledged UVM can be perceived as overkill for some specific tasks and for certain classes of designs. Consider small design teams, with hardware background. Many of these small teams may not receive a structured training on advanced, software centric approach being promoted by UVM. Without a solid introduction, it is hard for hardware engineers to adopt UVM in one-go! The other group of electronic designs includes FPGA designs with majority of them using simpler, linear and procedural testbench styles with access to less powerful EDA tools. Also FPGA teams often look for template creation tools as add-ons to their EDA tools to speed up their testbench creation process. There are also cases in ASIC design flow, where-in the test inputs come as stream of structured data (such as DFT patterns). Classical example is ATPG test patterns intended to test for stuck-at faults. ATPG patterns are usually self-checking and interact with design at PIO (Primary Input/Output) level. Such DFT verification teams are finding it difficult to leverage on well-defined UVM approach. Last but not the least, various university students across the globe are looking for a quick start into industry standard UVM.They do not have access to all the sophisticated training and hand-holding needed to get started with UVM. Given that the future DV engineers emerge from these universities, it is imperative for the industry to get the students started UVM as early as possible.

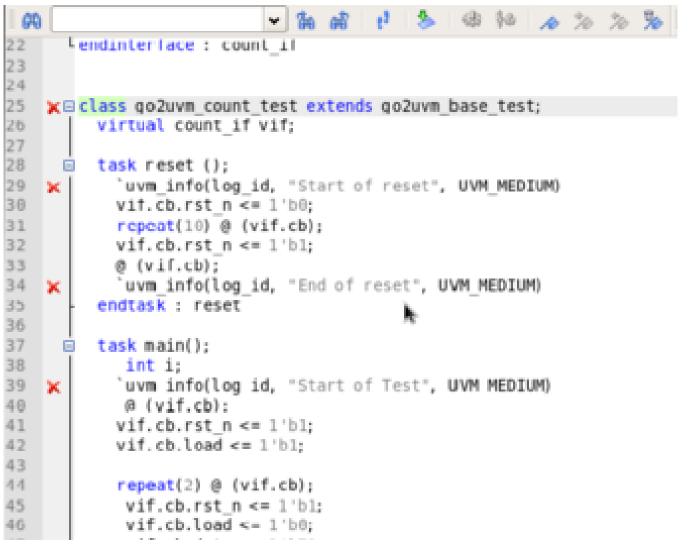
Given the technical commonalities across all these various design verification tasks, one could imagine that some portions of UVM are still applicable to all of these tasks, while some advanced UVM features may not be so appealing to them. In this paper we present our experience in working with customers coming from many of the previously mentioned backgrounds and being able to deploy UVM to them via a convenience layer around the standard UVM named *Go2UVM*. *Go2UVM* is not a script to generate a set of files that users can fill-in, rather it is an OOP layer around standard UVM hiding all the glory details of UVM and providing the first-time UVM users an easy to use procedural interface to UVM. *Go2UVM* is open-sourced, sits on top of standard UVM and hence is 100% in-line with UVM philosophy of test creation. There are also several “apps” being developed to speed up the process of creating templates for *Go2UVM* to make the industry move to UVM the fastest way!

II. Go2UVM

A. *What's Go2UVM?*

In simple words, *Go2UVM* is an open-source, SystemVerilog package around Accellera UVM base class library. It provides a test layer around standard UVM to hide the common complexities such as:

- Phasing (reset_phase, main_phase etc.)
- Objection mechanism (A must in UVM to get even a simple stimulus through to the DUT)
- Multiple layers of components, that at times, smaller designs may not require
- Hierarchical component hook-ups via UVM's preferred uvm_component::new (string name, uvm_component parent) pattern
- UVM macros that may not add value to a given task at hand



```
endinterface : count_if


class go2uvm_count_test extends go2uvm_base_test;
  virtual count_if vif;

  task reset ();
    `uvm_info(log_id, "Start of reset", UVM_MEDIUM)
    vif.cb.rst_n <= 1'b0;
    repeat(10) @ (vif.cb);
    vif.cb.rst_n <= 1'b1;
    @ (vif.cb);
    `uvm_info(log_id, "End of reset", UVM_MEDIUM)
  endtask : reset

  task main();
     int i;
    `uvm info(log id, "Start of Test", UVM MEDIUM)
     @ (vif.cb);
    vif.cb.rst_n <= 1'b1;
    vif.cb.load <= 1'b1;

    repeat(2) @ (vif.cb);
     vif.cb.rst_n <= 1'b1;
     vif.cb.load <= 1'b0;
```

Figure-1 *Go2UVM* test

B. *Motivation behind Go2UVM*

Taking a step back, let's see how the mobile phone evolution has been over many decades. Martin Cooper, inventor of mobile phones, who was the lead engineer of the Motorola team that developed the first mobile phone recently said:

Well, we knew that someday everybody would have a [cell] phone. Phones have gotten so complicated, so hard to use,that you wonder if this is designed for real people or for engineers [alone].

Some customers (from the category of design teams as mentioned earlier) felt UVM is on a similar route. To summarize their feeling on UVM:

While the "U" in UVM reads as "Universal", we wonder if it is only for software savvy, OOP fanatic engineers or can it be used by many others who are more of hardware design engineers trying to get simple verification done.

This is what got us started on the simple, yet powerful open-source package *Go2UVM* with the sole intention of making UVM easy to use for the first-timers.

C. *Inside Go2UVM*

As mentioned earlier, *Go2UVM* is a SystemVerilog package around standard UVM. It extends the uvm_test base class as shown in UML diagram below:

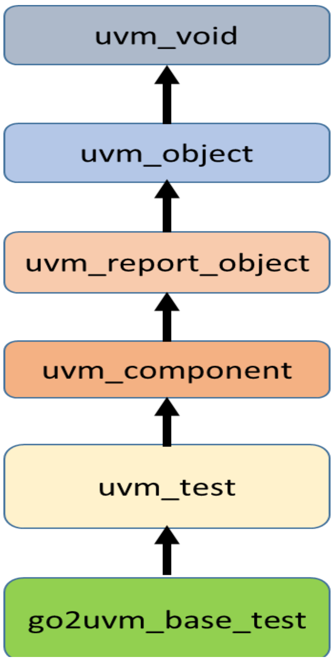
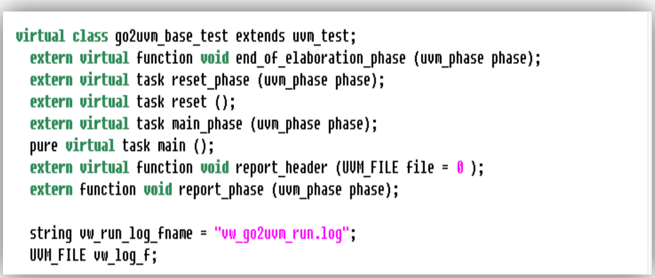


Figure-2 Go2UVM UML diagram

Following the general idea of standard object oriented paradigm Go2UVM base test contains common code that needs to be repeated by every test/user. It is declared as *virtual class* indicating that it is an abstract class and shall be extended by the user with a concrete class. In UVM even to add a simple trace (a series of input signal wiggling is generally called a "trace"), the following needs to be done:

- Use *uvm_test* (and typically more UVM components, for now focus on test)
- Hierarchically connect to the UVM framework via *parent* argument of the constructor
- Create a task that performs reset of the given DUT (consider that all hardware designs typically require reset and the reset is a time consuming task).
- Create a task that performs the intended DUT signal wiggling (more sophisticated transaction based approach is next step in UVM)
- Invoke the above 2 tasks to perform *reset* and the *main* functionality inside standard UVM phasing (So that UVM BCL can invoke these tasks at appropriate time in simulation)
- While performing any time consuming action within UVM framework, make sure to raise and drop objections.

The above steps are done using a base class named *go2uvm_base_test*. The prototypes of relevant methods are shown in the code snippet below:



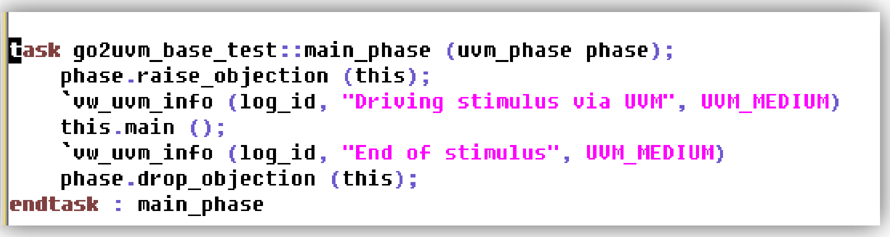
```
virtual class go2uvm_base_test extends uvm_test;
  extern virtual function void end_of_elaboration_phase (uvm_phase phase);
  extern virtual task reset_phase (uvm_phase phase);
  extern virtual task reset ();
  extern virtual task main_phase (uvm_phase phase);
  pure virtual task main ();
  extern virtual function void report_header (UVM_FILE file = 0 );
  extern function void report_phase (uvm_phase phase);

  string vw_run_log_fname = "vw_go2uvm_run.log";
  UVM_FILE vw_log_f;
```

Figure-3Go2UVM base test code snippet

From an end user perspective (read it as non-UVM aware engineer), this class adds 2 methods: *reset()* and *main()* – both of them are user extendable methods. The *go2uvm_base_test* invokes these methods inside standard UVM phasing with proper objection raising and dropping under-the-hood.

This is one of the very common mistakes a first-time UVM user gets into (Ref: 3) and Go2UVM takes care of this. The other common mistake is to declare a task and not calling it. While UVM handles this cleverly with standard phasing, the code to declare any phase is little more than what a first-timer would like it to be. It is augmented by the fact that all UVM phases take *uvm_phase phase* as argument and majority of them don't use this argument – this is confusing to many users indeed. In *Go2UVM* we simplify this by hiding the phases from user code and instead requiring them to fill-in two simple tasks *reset & main* and the base class invokes them inside the correct phase as shown below:



```
task go2uvm_base_test::main_phase (uvm_phase phase);
    phase.raise_objection (this);
    `vw_uvm_info (log_id, "Driving stimulus via UVM", UVM_MEDIUM)
    this.main ();
    `vw_uvm_info (log_id, "End of stimulus", UVM_MEDIUM)
    phase.drop_objection (this);
endtask : main_phase
```

Figure-4 Go2UVM test main phase

What if a user forgets to fill-in ***task main()*** in derived class? Well, this is why *Go2UVM* package declares this method as *pure virtual.*As per the standard definition from Wiki (Ref – 5):

*A **pure virtual function** or **pure virtual method** is a virtual function that is required to be implemented by a concrete, derived class*

Hence with *Go2UVM* if one forgets to implement *task main()* in derived class, a compiler would flag this almost instantaneously:

```
../unit_test_src/vw_ahb_lite_cip_go2uvm_test.svi : (22, 27): Cannot declare class
vw_ahb_lite_cip_test as non abstract class due to not implemented pure virtual methods:main()
```

It takes care of the standard UVM requirement of component hierarchy hook-up via *name*&*parent* under the hood so that first time users do not need to bother about it. More details with the full source code can be found at: http://www.go2uvm.org/download/VW_Go2UVM_Pkg_2016.05.tar.gz.

The idea is to provide the fastest way for an engineer to get started with UVM. Since it is 100% IEEE 1800 (SystemVerilog) and IEEEP1800.2 (UVM) compatible, users can easily start with *Go2UVM* and move to a full-fledged UVM environment, as they get mature with the technology. *Go2UVM* is well tested on all major simulators from popular EDA vendors.

III. RESULTS/APPLICATIONS

A. *DFT patterns in simulation*

Design For Testability (DFT) is a critical step in standard ASIC design flow. As part of verification of the DFT structures engineers often run a set of scan patterns on the design annotated with scan cells and ensure nothing is broken. Given the exorbitant cost of failed silicon, it is a standard practice to run a sub-set of the scan patterns in simulation itself during the Pre-Silicon stage of the ASIC design. Except that the source of the stimulus is from DFT pattern generator, this task is quite similar to the functional verification. Since standard verification is all done with UVM, it is very lucrative for the DFT teams to run their design patterns in a UVM framework and be able to report failures, analyze coverage etc. just like how the functional verification teams do. *Go2UVM* has been successfully deployed in such situations to provide a convenience layer around the UVM and feeding the DFT patterns to DUT without having to go through layers of standard UVM components.

A typical fault injection process for a given DUT looks as shown in figure below (Ref-7)

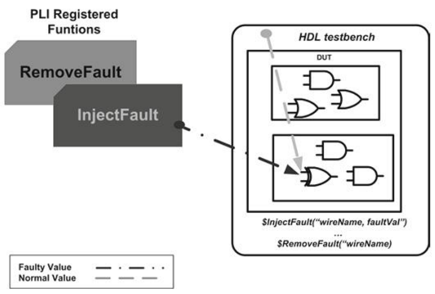


Figure-5 Typical fault injection and removal flow

A test-fixture (as it is commonly referred to in DFT domain) for a simple full-adder looks like below. A detailed explanation of the PLI calls and sophisticated test-fixture is available in reference 7 & 8.

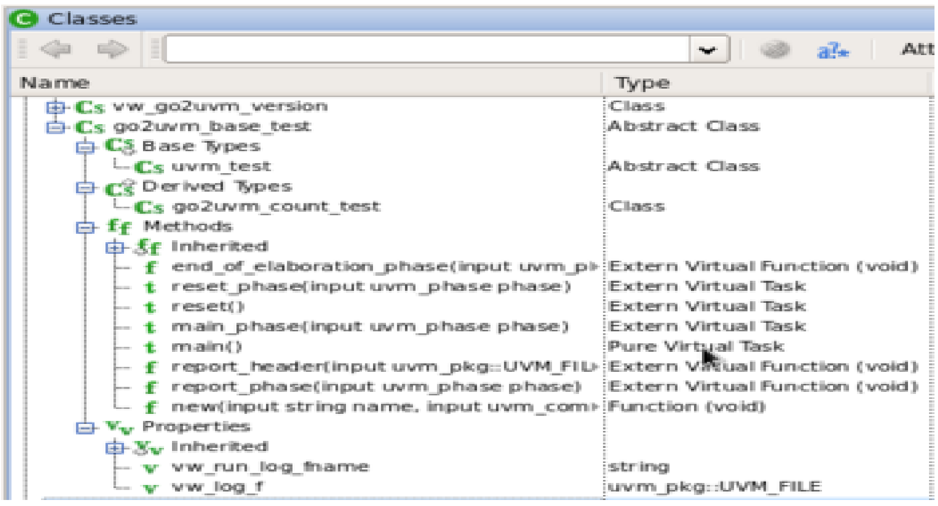
```
module testbench();
reg a, b, cin;
wire sum_f, sum_g;
wire co_f, co_g;

   FA FA_ golden (a, b, cin, sum_g, co_g);
   FA FA_faultable (a, b, cin, sum_f, co_f);

   initial begin
      #20;
      $InjectFault("testbench.FA_faultable.s", 1'b0);
      repeat(10) begin
         #150;
         {a,b,cin} = $random();
      end
      $RemoveFault("testbench.FA_faultable.s");
      repeat(10) begin
         #150;
         {a,b,cin} = $random();
      end
      $stop;
    end
endmodule
```

Figure-6Simple test-fixture for fault testing

Now to bring UVM into the picture, it is quite a task should we use all standard UVM components such as driver, sequencer, monitor, agent, scoreboard etc. Also the value of all those components will be very little in a typical DFT verification flow. However as DV teams use UVM, it is useful for the DFT verification team also to produce regressions results in the same UVM format. *Go2UVM* is a natural fit to this as it directly uses a *virtual interface* inside a test and drives/samples pins. From the simple test-fixture shown in figure-6 above, the *initial* block should become part of *dft_test::main()* and the rest shall be auto-generated using the free apps available (Refer to next chapter).



Name	Type
vw_go2uvm_version	Class
go2uvm_base_test	Abstract Class
Base Types	
uvm_test	Abstract Class
Derived Types	
go2uvm_count_test	Class
Methods	
Inherited	
end_of_elaboration_phase(input uvm_ph	Extern Virtual Function (void)
reset_phase(input uvm_phase phase)	Extern Virtual Task
reset()	Extern Virtual Task
main_phase(input uvm_phase phase)	Extern Virtual Task
main()	Pure Virtual Task
report_header(input uvm_pkg::UVM_FIL	Extern Virtual Function (void)
report_phase(input uvm_phase phase)	Extern Virtual Function (void)
new(input string name, input uvm_com	Function (void)
Properties	
Inherited	
vw_run_log_fname	string
vw_log_f	uvm_pkg::UVM_FILE

Figure-7 Go2UVM base testflow

B. DO-254 &FPGA designs

One of our observations of FPGA design industry (and the associated DO-254 kind of designs) is that engineers doing verification of these designs have a great affinity to use Graphical User Interface (GUI) very early – as soon as the design is ready. This is quite contrary to typical ASIC design verification teams who bring up the GUI once a test (say in UVM) is ran, shows a failure and a debug database (such as DUMP file) is created. Even teams using interactive debug in ASIC design verification flow tend to use the GUI for"debug" than "test authoring".The goal is to enable RTL designers to compile their designs, bring up the GUI and with a click-of-a-button generate a UVM test with necessary hooks in place.

Figure-8below shows a typical FPGA engineer's desktop with Mentor Graphics's Questa simulator. Once the RTL design is loaded to the familiar GUI, a new pull-down menu can generate necessary Go2UVM files.

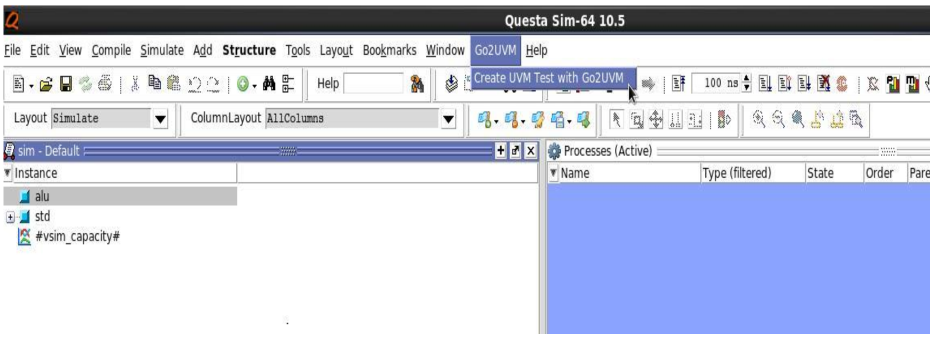


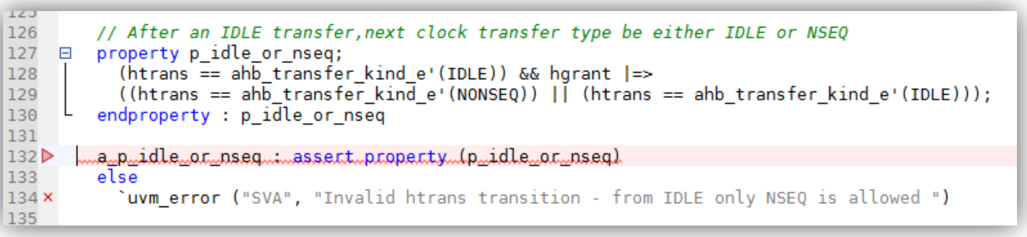
Figure-8 Go2UVM app as Pull-Down menu in Mentor Graphics's Questa simulator.

The main take-away for the FPGA designers is that moving to UVM is not a big task, especially with *Go2UVM* package and some of the emerging "apps" that make such move smooth (as the apps are integrated into their favorite GUI-s).

With FPGAs becoming complex in capacity and speed, many native FPGA teams are moving to full-fledged UVM. Also the ASIC teams are increasingly using FPGA prototyping prior to committing mask costs. Hence an easy entry to the UVM based verification is very useful for the FPGA teams. We believe this *Go2UVM* package and its accompanying apps will fuel that move to UVM for FPGA teams across the globe.

C. Writing traces for assertions

SystemVerilog Assertions (SVA)(Ref-6) are great to capture standard protocol behaviors as independent checkers and reuse them across designs. However, given the complex nature of designs being verified and the language nuances in SVA, often designers write simple traces to verify these assertions in stand-alone manner. For example, consider a typical AHB ([9])protocol requirement on *htrans* that says after IDLE, only NONSEQEUENTIAL type of transfer is allowed. Following assertion captures this requirement:

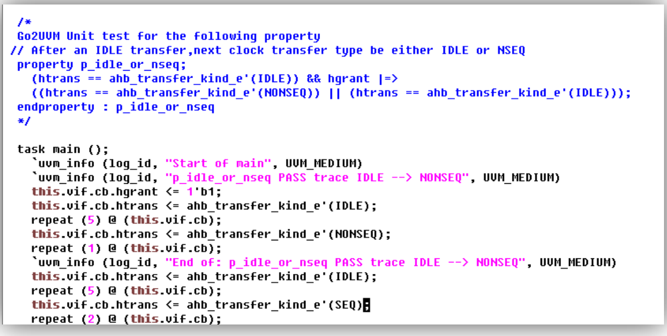


```
// After an IDLE transfer,next clock transfer type be either IDLE or NSEQ
property p_idle_or_nseq;
  (htrans == ahb_transfer_kind_e'(IDLE)) && hgrant |=>
  ((htrans == ahb_transfer_kind_e'(NONSEQ)) || (htrans == ahb_transfer_kind_e'(IDLE)));
endproperty : p_idle_or_nseq

a_p_idle_or_nseq : assert property (p_idle_or_nseq)
else
  `uvm_error ("SVA", "Invalid htrans transition - from IDLE only NSEQ is allowed ")
```

Figure-10 Simple assertion to capture AHB *htrans* transition requirement

A stand-alone, unit-test to verify the above assertion would be referred to as trace – a series of signal changes over several clocks to satisfy and falsify the above assertion. Since these assertions are reusable across designs, they need to be thoroughly verified. A full-fledged UVM will again be overkill for this verification requirement. A *Go2UVM* test to capture a trace for this assertion is shown below:



```
 /*
 Go2UVM Unit test for the following property
// After an IDLE transfer,next clock transfer type be either IDLE or NSEQ
 property p_idle_or_nseq;
   (htrans == ahb_transfer_kind_e'(IDLE)) && hgrant |=>
   ((htrans == ahb_transfer_kind_e'(NONSEQ)) || (htrans == ahb_transfer_kind_e'(IDLE)));
 endproperty : p_idle_or_nseq
 */

 task main ();
   `uvm_info (log_id, "Start of main", UVM_MEDIUM)
   `uvm_info (log_id, "p_idle_or_nseq PASS trace IDLE --> NONSEQ", UVM_MEDIUM)
   this.vif.cb.hgrant <= 1'b1;
   this.vif.cb.htrans <= ahb_transfer_kind_e'(IDLE);
   repeat (5) @ (this.vif.cb);
   this.vif.cb.htrans <= ahb_transfer_kind_e'(NONSEQ);
   repeat (1) @ (this.vif.cb);
   `uvm_info (log_id, "End of: p_idle_or_nseq PASS trace IDLE --> NONSEQ", UVM_MEDIUM)
   this.vif.cb.htrans <= ahb_transfer_kind_e'(IDLE);
   repeat (5) @ (this.vif.cb);
   this.vif.cb.htrans <= ahb_transfer_kind_e'(SEQ);
   repeat (2) @ (this.vif.cb);
```

Figure-11 Sample trace with *Go2UVM* to verify the assertion onAHB *htrans*

IV. SMART "APPS" AROUND*GO2UVM*

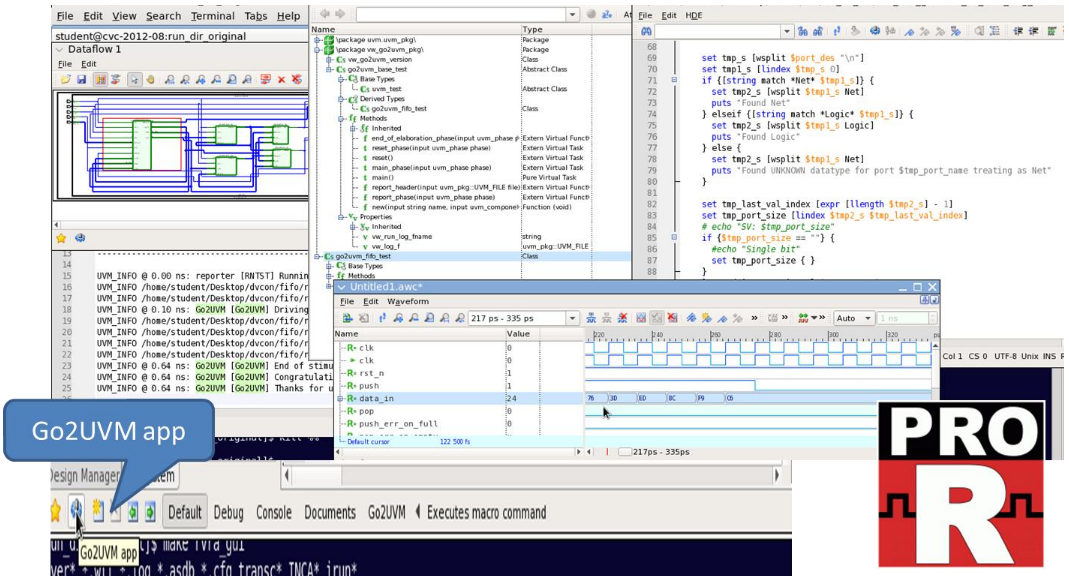
One of the good things about open-source approach is that it lets more innovation to emerge from across the community. In case of *Go2UVM* while we believe the approach is simple and straight forward for practicing engineers, the student community found it difficult without a basic introductory training. While there is some effort in providing free training around *Go2UVM* for the academia (Ref: 3), we quickly realized it is very difficult to train every budding VLSI engineer on this. However, given the fixed structure of *UVM* and *Go2UVM* we have developed some "apps" (applications) to run on existing EDA tools to create the skeleton code needed to start with UVM.

Consider the case of many FPGA designers and engineering students with a simple RTL module that they intend to verify. Broadly speaking there are four significant steps to get started with UVM to verify this simple RTL.

1. The first and foremost step in creating a UVM bench is to create a SystemVerilog interface file with clocking blocks and modports. While the concept is simple, there are number of lines to be coded to get this right. Given that EDA tools do provide some form of API to query design ports, this is a task that can be automated.
2. Once a well-defined SystemVerilog interface is made available, a skeleton test case that extends *go2uvm_base_test* can be created with a handle to this interface declared as a *virtual interface*.
3. The third part of this process is to hook-up the DUT pins with the test and call the standard UVM *run_test ().*
4. Last but not the least, pass relevant arguments to your favorite EDA tool to compile UVM, *Go2UVM* package, set of RTL files and any custom options (such as *+define, `incdir* etc.).

Given the open-source nature of *Go2UVM*, these "apps" are being developed by several university students for every major EDA simulator that support UVM. These apps will soon be available at www.go2uvm.org for free download (including, for commercial usage).

Figure-8 below shows Go2UVM app integrated into Aldec's Riviera-PRO simulator. The icon at the bottom-left corner of the main R-PRO windows invokes Go2UVM app and generates the necessary files.



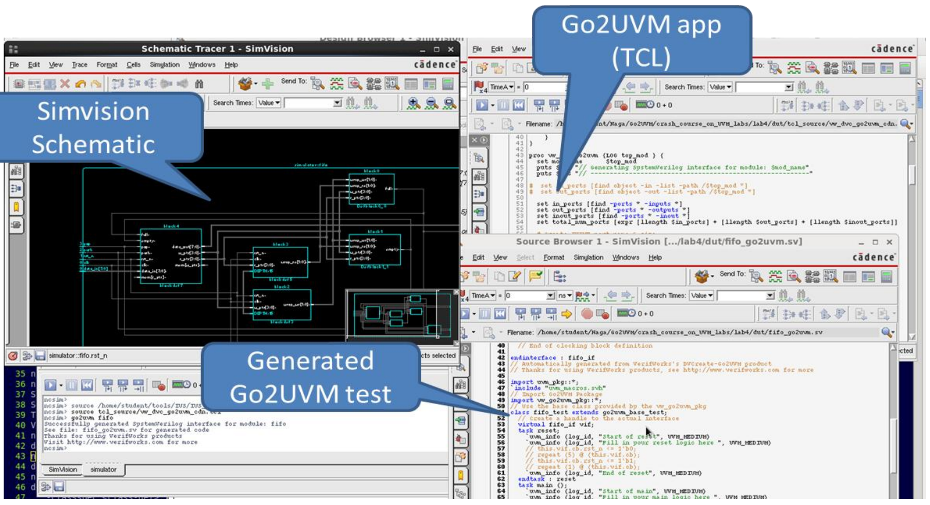
```
UVM_INFO @ 0.00 ns: reporter [RNTST] Runnin
UVM_INFO /home/student/Desktop/dvcon/fifo/r
UVM_INFO /home/student/Desktop/dvcon/fifo/r
UVM_INFO @ 0.10 ns: Go2UVM [Go2UVM] Driving
UVM_INFO /home/student/Desktop/dvcon/fifo/r
UVM_INFO /home/student/Desktop/dvcon/fifo/r
UVM_INFO /home/student/Desktop/dvcon/fifo/r
UVM_INFO /home/student/Desktop/dvcon/fifo/r
UVM_INFO @ 0.64 ns: Go2UVM [Go2UVM] End of stimu
UVM_INFO @ 0.64 ns: Go2UVM [Go2UVM] Congratulati
UVM_INFO @ 0.64 ns: Go2UVM [Go2UVM] Thanks for u
```

```
set tmp_s [wsplit $port_des "\n"]
set tmp1_s [lindex $tmp_s 0]
if {[string match *Net* $tmp1_s]} {
  set tmp2_s [wsplit $tmp1_s Net]
  puts "Found Net"
} elseif {[string match *Logic* $tmp1_s]} {
  set tmp2_s [wsplit $tmp1_s Logic]
  puts "Found Logic"
} else {
  set tmp2_s [wsplit $tmp1_s Net]
  puts "Found UNKNOWN datatype for port $tmp_port_name treating as Net"
}

set tmp_last_val_index [expr [llength $tmp2_s] - 1]
set tmp_port_size [lindex $tmp2_s $tmp_last_val_index]
# echo "SV: $tmp_port_size"
if {$tmp_port_size == ""} {
  #echo "Single bit"
  set tmp_port_size { }
}
```

Figure-8 Go2UVM app integrated into Aldec’s Riviera-PRO simulator.

Go2UVM app is ported to work with Cadence’s IUS simulator through its TCL interface. A typical use of this app in IUS flow looks as shown in Figure-9 below:


Schematic Tracer 1 - SimVision
Go2UVM app (TCL)
Simvision Schematic
Generated Go2UVM test
Source Browser 1 - SimVision [.../lab4/dut/fifo_go2uvm.sv]

```
ncsim>
ncsim> source /home/student/tools/IUS/IUS
ncsim> source tcl_source/vw_dvc_go2uvm_cdn.
ncsim> go2uvm fifo
Successfully generated SystemVerilog interface for module: fifo
See file: fifo_go2uvm.sv for generated code
Thanks for using VerifWorks products
Visit http://www.verifworks.com for more
ncsim>
```

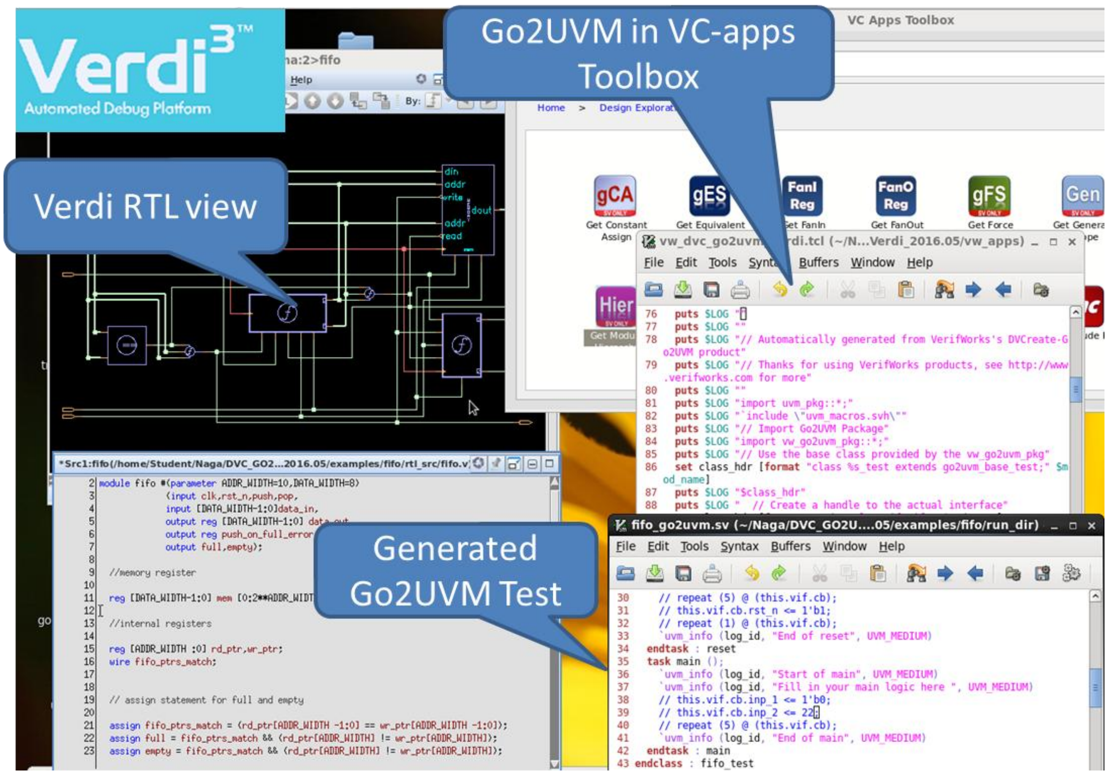
```
Filename: /home/student/Naga/Go2UVM/crash_course_on_UVM_labs/lab4/dut/fifo_go2uvm.sv
  // End of clocking block definition

endinterface : fifo_if
// Automatically generated from VerifWorks's DVCreate-Go2UVM product
// Thanks for using VerifWorks products, see http://www.verifworks.com for more

import uvm_pkg::*;
`include "uvm_macros.svh"
// Import Go2UVM Package
import vw_go2uvm_pkg::*;
// Use the base class provided by the vw_go2uvm_pkg
class fifo_test extends go2uvm_base_test;
  // Create a handle to the actual interface
  virtual fifo_if vif;
  task reset;
    `uvm_info (log_id, "Start of reset", UVM_MEDIUM)
    `uvm_info (log_id, "Fill in your reset logic here ", UVM_MEDIUM)
    // this.vif.cb.rst_n <= 1'b0;
    // repeat (5) @ (this.vif.cb);
    // this.vif.cb.rst_n <= 1'b1;
    // repeat (1) @ (this.vif.cb);
    `uvm_info (log_id, "End of reset", UVM_MEDIUM)
  endtask : reset
  task main ();
    `uvm_info (log_id, "Start of main", UVM_MEDIUM)
    `uvm_info (log_id, "Fill in your main logic here ", UVM_MEDIUM)
```

Figure-9 Go2UVM app within Cadence’s IUS framework

For users familiar with debug API such as NPI (Novas Programming Interface) from Synopsys ([12]), a set of such apps is available as part of “VC Apps” ([13]). Go2UVM app is now available as third-party VC-apps as shown below in Figure-11.



*Src1:fifo(/home/Student/Naga/DVC_GO2...2016.05/examples/fifo/rtl_src/fifo.v

```
module fifo #(parameter ADDR_WIDTH=10,DATA_WIDTH=8)
            (input clk,rst_n,push,pop,
            input [DATA_WIDTH-1:0]data_in,
            output reg [DATA_WIDTH-1:0] data_out
            output reg push_on_full_error
            output full,empty);

  //memory register

  reg [DATA_WIDTH-1:0] mem [0:2**ADDR_WIDT

  //internal registers

  reg [ADDR_WIDTH :0] rd_ptr,wr_ptr;
  wire fifo_ptrs_match;


  // assign statement for full and empty

  assign fifo_ptrs_match = (rd_ptr[ADDR_WIDTH -1:0] == wr_ptr[ADDR_WIDTH -1:0]);
  assign full = fifo_ptrs_match && (rd_ptr[ADDR_WIDTH] != wr_ptr[ADDR_WIDTH]);
  assign empty = fifo_ptrs_match && (rd_ptr[ADDR_WIDTH] != wr_ptr[ADDR_WIDTH]);
```

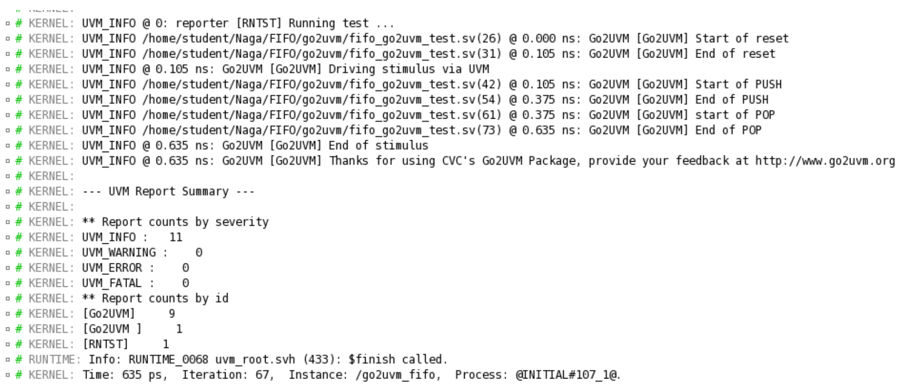
vw_dvc_go2uvm_verdi.tcl (~/N...Verdi_2016.05/vw_apps)

```
  puts $LOG ""
  puts $LOG ""
  puts $LOG "// Automatically generated from VerifWorks's DVCreate-Go2UVM product"
  puts $LOG "// Thanks for using VerifWorks products, see http://www.verifworks.com for more"
  puts $LOG ""
  puts $LOG "import uvm_pkg::*;"
  puts $LOG "`include \"uvm_macros.svh\""
  puts $LOG "// Import Go2UVM Package"
  puts $LOG "import vw_go2uvm_pkg::*;"
  puts $LOG "// Use the base class provided by the vw_go2uvm_pkg"
  set class_hdr [format "class %s_test extends go2uvm_base_test;" $mod_name]
  puts $LOG "$class_hdr"
  puts $LOG "  // Create a handle to the actual interface"
```

fifo_go2uvm.sv (~/Naga/DVC_GO2U....05/examples/fifo/run_dir)

```
    // repeat (5) @ (this.vif.cb);
    // this.vif.cb.rst_n <= 1'b1;
    // repeat (1) @ (this.vif.cb);
    `uvm_info (log_id, "End of reset", UVM_MEDIUM)
  endtask : reset
  task main ();
    `uvm_info (log_id, "Start of main", UVM_MEDIUM)
    `uvm_info (log_id, "Fill in your main logic here ", UVM_MEDIUM)
    // this.vif.cb.inp_1 <= 1'b0;
    // this.vif.cb.inp_2 <= 22;
    // repeat (5) @ (this.vif.cb);
    `uvm_info (log_id, "End of main", UVM_MEDIUM)
  endtask : main
endclass : fifo_test
```

Figure-12Go2UVM app available as part of Verdi VC apps



```
# KERNEL: UVM_INFO @ 0: reporter [RNTST] Running test ...
# KERNEL: UVM_INFO /home/student/Naga/FIFO/go2uvm/fifo_go2uvm_test.sv(26) @ 0.000 ns: Go2UVM [Go2UVM] Start of reset
# KERNEL: UVM_INFO /home/student/Naga/FIFO/go2uvm/fifo_go2uvm_test.sv(31) @ 0.105 ns: Go2UVM [Go2UVM] End of reset
# KERNEL: UVM_INFO @ 0.105 ns: Go2UVM [Go2UVM] Driving stimulus via UVM
# KERNEL: UVM_INFO /home/student/Naga/FIFO/go2uvm/fifo_go2uvm_test.sv(42) @ 0.105 ns: Go2UVM [Go2UVM] Start of PUSH
# KERNEL: UVM_INFO /home/student/Naga/FIFO/go2uvm/fifo_go2uvm_test.sv(54) @ 0.375 ns: Go2UVM [Go2UVM] End of PUSH
# KERNEL: UVM_INFO /home/student/Naga/FIFO/go2uvm/fifo_go2uvm_test.sv(61) @ 0.375 ns: Go2UVM [Go2UVM] start of POP
# KERNEL: UVM_INFO /home/student/Naga/FIFO/go2uvm/fifo_go2uvm_test.sv(73) @ 0.635 ns: Go2UVM [Go2UVM] End of POP
# KERNEL: UVM_INFO @ 0.635 ns: Go2UVM [Go2UVM] End of stimulus
# KERNEL: UVM_INFO @ 0.635 ns: Go2UVM [Go2UVM] Thanks for using CVC's Go2UVM Package, provide your feedback at http://www.go2uvm.org
# KERNEL:
# KERNEL: --- UVM Report Summary ---
# KERNEL:
# KERNEL: ** Report counts by severity
# KERNEL: UVM_INFO :   11
# KERNEL: UVM_WARNING :    0
# KERNEL: UVM_ERROR :    0
# KERNEL: UVM_FATAL :    0
# KERNEL: ** Report counts by id
# KERNEL: [Go2UVM]     9
# KERNEL: [Go2UVM ]     1
# KERNEL: [RNTST]     1
# RUNTIME: Info: RUNTIME_0068 uvm_root.svh (433): $finish called.
# KERNEL: Time: 635 ps,  Iteration: 67,  Instance: /go2uvm_fifo,  Process: @INITIAL#107_1@.
```

Figure-9 Sample log file from *Go2UVM*

V. Summary

Go2UVM is a package derived from UVM library. For first-time users who may not (yet) have the prior knowledge of UVM and OOP's concepts, *Go2UVM* provides a simpler way to verify smaller designs. With free tools providing automation around this *Go2UVM* package, it is really quick for FPGA designers, students and others to get started with UVM the fastest way.Since *Go2UVM* is 100% IEEE 1800 and IEEE P1800.2 compatible, users can easily start with *Go2UVM* and move to a full-fledged UVM environment, as they get mature with the technology. Go2UVM supports all major simulators from popular EDA vendors. Go2UVM is the preferred way for industry engineers running DFT and other related verification traces back in simulation domain. For students and budding engineers, Go2UVM provides the ideal starting point to write quality simulation traces with UVM and thereby making them most appealing and wanted by the semiconductor design houses looking to hire them from the universities.

VI. Conclusion & Future work

UVM is clearly the most-adopted standard in the DV field. With *Go2UVM* and associated "apps" it is set to reach those users hitherto untouched owing to the complexity of UVM for first-time users. As future work, we would encourage academia teaching VLSI to pick up and run with it, create even more apps, donate them etc. We are also working with leading educational institutions to port popular SVA book examples to use *Go2UVM* traces. Another opportunity for budding engineers at colleges would be to create *Go2UVM* tests for some of the open-source designs such as MIAOW GPU (Ref-10, 11).

Acknowledgment

We would like to thank our mentors Mr. **SrinivasanVenkataramanan,** Co-Founder**,** Verification Technologist at VerifWorks& Mrs. **AjeethaKumari,** Serial Woman Entrepreneur at CVC for their continuous motivation and guidance. Last but not least a big thanks to entire CVC Team for overwhelming support.

VII. REFERENCE

[1] Accellera UVM: http://accellera.org/downloads/standards/uvm
[2] UVM SystemC http://accellera.org/activities/working-groups/systemc-verification
[3] *Go2UVM*: http://www.go2uvm.org
[4] VerifWorks: http://www.verifworks.com
[5]https://en.m.wikipedia.org/wiki/Virtual_function
[6] SystemVerilog Assertions Handbook, http://verifnews.org/publications/books/
[7] Digital System Test and Testable Design – Z. Navabi, http://www.springer.com/us/book/9781441975478
[8] DATE: A Package for Test Hardware Evaluation, Zainalabedin Navabi et al. https://www.date-conference.com/files/file/10-ubooth/ub-2.3-p09.pdf
[9] ARM AMBA Specifications: https://www.arm.com/products/system-ip/amba-specifications.php
[10] MIAOW GPU http://www.miaowgpu.org/
[11] Open-source GPU decode block gets open-source Go2UVM Test! http://www.go2uvm.org/2016/07/open-source-gpu-decode-block-gets-open-source-go2uvm-test/
[12] Verdi VC Apps https://www.vc-apps.org/SitePages/Home.aspx
[13] Verdi's open debug platform enables portability across EDA vendors http://verifworks.com/verdis-open-debug-platform-enables-portability-across-eda-vendors/