

# Functional Verification of an HW Block using VERA

Marco Brunelli, Luca Battù, Andrea Castelnuovo, Francesco Sforza

Central R&D, NVM Design Platform  
STMicroelectronics Agrate, Italy

*francesco.sforza@st.com*

## ABSTRACT

During the functional verification of an STBus-based SoC, the behavior of a block (a memory manager) was verified using VERA. The verification environment was setup describing the necessary low-level drivers and implementing different abstraction layers involving random stimulus generation, coverage goals and synchronization features.

Comparisons of VERA versus a “standard” verification flow, based on a VHDL testbench, are reported.

Using the VERA/Scirocco environment as a reference, an attempt was made to connect VERA to a hardware emulator. The connection was established via a SW simulator and the new environment was tested against plain simulation. Problems and results of this experiment are detailed in this paper.

A possible way to enhance VERA features and the improvements needed to better take advantage of the emulator performances are also detailed.

*Keywords: VERA, STBus, Emulation, Code Coverage*

## 1.0 Introduction

Quality of past and present electronic devices depends heavily on how good and fast their verification process can be. Poor verification, for whatever reason, is very likely to result in poor device quality or device failure which, in turn, can impact time-to-market constraints.

In this article a real case has been used to compare the traditional functional verification flow based on custom VHDL testbenches with the application of a Hardware Verification Language based tool (VERA).

The project was used as a test case for VERA on a real design. The major goals of this test were to:

(1) evaluate the cost of moving to a new testbench generation environment and (2) compare the new environment's capabilities against standard VHDL testbenches.

The project is ongoing and current results and future directions are shown. It is also shown how HVL capabilities have been improved by taking advantage of collecting information about the code coverage (in order to drive the pattern generation of VERA) and from speeding up the DUT simulation by using HW emulation.

The rest of this paper is divided as follows: section 2 describes the design under test (DUT); section 3 describes the existing VHDL testbench as received from the design team; section 4 is dedicated to the developed VERA testing environment; section 5 reports the results of a testing session using VERA; section 6 describes two possible ways to enhance VERA capabilities introducing code coverage and emulation in the scope.

## 2.0 DUT description

The design under test is named Free List Manager (FLM), the memory address controller of a SoC still under development. This block is directly connected to the on-chip bus and manages the memory access, delivering free addresses to SW and HW tasks. The FLM is connected to each block that requires interaction with the SDRAM memory.

The FLM design is divided into a control block and a data path. It consists of about 50 Kgates plus about 1Kbyte of RAM divided into three sections.

To better understand the behavior of this block, a short description of the STBus protocol is provided below.

### 2.1 STBus Protocol Basics

The STBus communication protocol is based on different abstraction layers. A transaction (see figure 1.) between an *initiator* (master) and a *target* module (slave) consists of the transmission of a *packet* where each packet consists of a number of *tokens*, transferred on any cycle and controlled by a request/grant (RG) protocol.

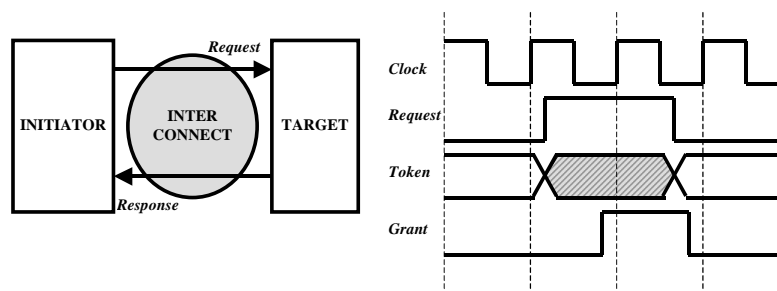


Figure 1. Basic STBus protocol

The request/grant handshake provides an unambiguous synchronization for transferring tokens over the interface.

## 2.2 DUT Functional Description

The FLM block stores and manages pointers to free memory addresses. The packet memory is a global resource shared between all of the sub-blocks of the SoC. The FLM manages the extraction/recycling of buffers in order to prevent resource starvation and to provide memory protection. To address a buffer, the FLM stores its base address in an internal list (see figure 2.).

*Partitions* are virtual areas of the packet memory collecting buffers. The maximum number of partitions available is 64. Partition sizes (number of buffers allowed) are configurable during the initialization phase.

Free buffer locations are stored in an internal LIFO (Last In First Out) of 256 words by 24 bits, which is fed by CPU processes and used by HW/SW tasks. There are two thresholds (MIN and MAX) to warn the CPU (via an interrupt) about the status of this LIFO: if the minimum is reached and there is free space available in the packet memory, the CPU can refill the LIFO with new addresses; if the maximum is reached the CPU process can remove data from the LIFO (see figure 3.).

Partitions must be enabled in the initialization phase. This is done by programming the FLM's internal registers. Once a partition is enabled, it may extract and recycle buffers by interacting with the FLM.

To manage partitions, the FLM has two additional memories: the "counters" memory (64 words by 9 bits) and the "thresholds" memory (64 words by 16 bits). When buffers are extracted from the LIFO, the correct *partition counter* (PC, 8 bits wide) is incremented; when buffers are returned it is decreased. Partition thresholds are assigned during the initialization phase. When the PC reaches threshold 1 a "warning-like" interrupt is generated; when threshold 2 is reached the partition cannot access any other memory buffer and an interrupt is generated (see figure 3.).

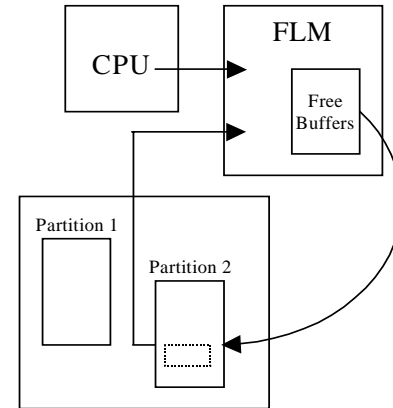


Figure 2. FLM Behavior

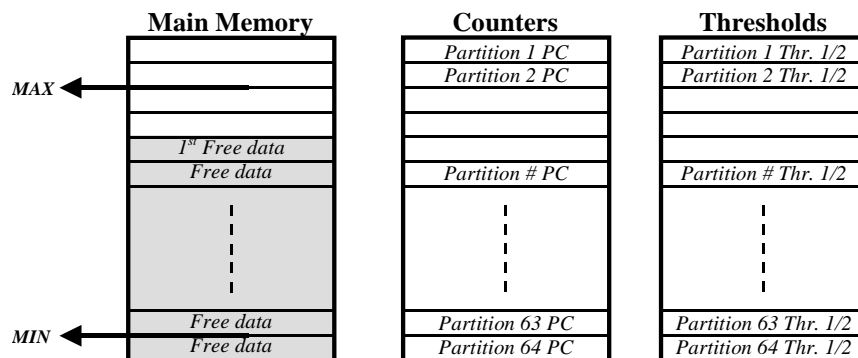


Figure 3 LIFO, counter and threshold memories inside FLM

Partitions can access the FLM LIFO simply by reading and writing registers, using their own partition Ids. The partition-to-FLM interaction takes place over the STBus.

### 3.0 VHDL existing testbenches

The FLM block was delivered by the designers with a simple set of VHDL testbenches that had been designed to just test the main features and functions that can be executed by the DUT (see previous section).

Some actions are mandatory at the beginning of the simulation and they can be viewed as an initialization process:

- all signals at the STBus interface are driven to '0';
- FLM partitions must be activated by initializing the FLM counter memory;
- for each active partition, the FLM stores upper and lower thresholds;

The two global thresholds for the FLM internal memory (LIFO) can be assigned and activated, and partition-related thresholds can be enabled or disabled.

These steps are common to all the VHDL testbenches hereto described. The following is a description of the five main groups of provided tests:

1. Test1: implements a CPU write operation.
2. Test2: implements a CPU write operation and hardware TASK read operations. During read operations several interrupts are generated.
3. Test3: implements a CPU write operation and hardware TASK read operations. During the hardware read operations different interrupts are generated because the partition thresholds are reached.
4. Test4: implements a CPU write and then read operation monitoring a specific interrupt.
5. Test5: implements a hardware CPU write operation but different interrupts are generated.

### 3.1 Why VERA

The testbenches described in the previous section are simple and often incomplete; they perform predefined operations with fixed data and static delays. No conditional decision can be taken; no random values can be chosen to drive all (or a great variety of) possible tasks, both HW and SW. Little intelligence is in the test, i.e. no actions are performed by a CPU task based on the status of the internal LIFO memory (full or empty); nothing is done when partition thresholds (min or max) are reached. Furthermore no value randomization is available to drive a random number of processes with random arguments.

Another key point is the absence of code and functional coverage capabilities, which would allow running the simulation until a desired percentage of the defined goal had been achieved (a typical goal may be to touch all the available states of the FLM internal state machine).

For all these aspects an HVL approach to testbench implementation is the better choice.

### 4.0 VERA testbench

To better understand the implementation of the VERA testbench, figure 4 shows a schematic flow of the testing process.

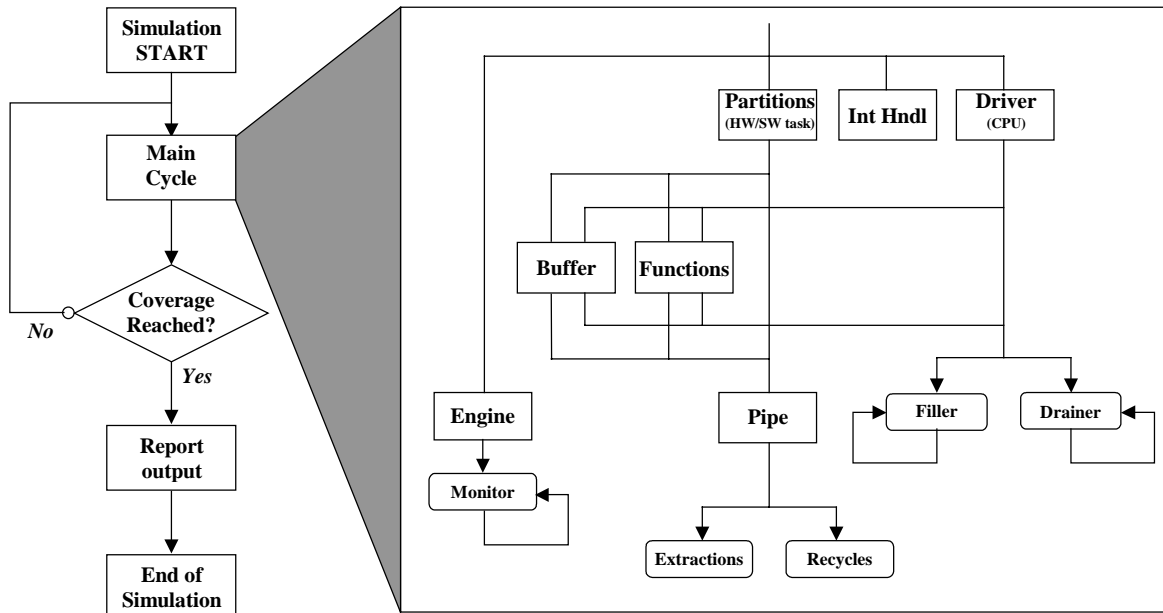


Figure 4: flow schema.

The entire process can be divided into three main parts corresponding to the three different levels inside the architecture plus an STBus interface. The complete flow is described in a bottom-up fashion.

At the lowest level an “Engine” class has been designed. This object runs forever in an infinite loop, fetching incoming packets and then driving them onto the STBus. Before Engine class executes, some initializations are performed to define the signal values for the Bus idle state as described in § 3.0 (see example 1).

Since the three main tasks (INTERRUPT, CPU and HW/SW) have different priorities (in order MAXIMUM, HIGH and LOW) as described in previous sections, care was taken to implement three distinct queues: the first queue has the highest priority (MAXIMUM) and is dedicated to handling the INTERRUPT task; the second queue has an intermediate priority (HIGH) and corresponds to the CPU task; the third queue has the lowest (LOW) and is the HW/SW task dedicated queue, which allows the system testbench to share data packets that need to be retrieved from the FLM. The “Engine” class handles the three queues (*mailboxes*) and the fetching of the packets.

```
@1,RESPONSE_TIMEOUT stb.r_req == 1'b1,
                    stb.r_eop == 1'b1,
                    stb.r_be == p.getBe(),
                    stb.r_opc == p.getOpc();
```

Example 1.

Next, there is the definition and instantiation of the three main processes. They basically correspond to the types of the packets coming from the described queues.

The class named PARTITIONS takes care of defining all the variables and values that are mandatory to completely define a single partition. A partition is the container that holds the logical identifier for each HW or SW task that shares data with the supplier (FLM).

The testbench generates a parameter-defined number of partitions and each one of them *randomly* creates one of the two possible tasks.

The maximum and minimum threshold values for HW/SW tasks belonging to a partition are *randomly* generated according to predefined *constraints* to avoid conflicts and/or mismatches between values (example 2 shows the use of constraints inside PARTITIONS objects).

Furthermore, the PARTITIONS class instantiates the PIPE class. PIPE objects provide the fetching of data from the DUT, i.e. *hw\_extraction*, *hw\_recycle*, *sw\_extraction* and *sw\_recycle*. To this end, suitable packets are generated according to STBus specs. They are put into the low priority *mailbox* ready to be parsed by the Engine.

Example 3 shows how probability weights can be modified at runtime.

The INTHNDL class (Interrupt Handler) is a similar and parallel process inside the testbench architecture. It has been developed to monitor the interrupt line coming from the DUT. Events are used to make decisions inside the VERA code when a change occurs on the interrupt line. The line is appropriately decoded and a suitable event is triggered. The packet prepared for the corresponding operation is put into the highest (MAXIMUM) priority mailbox.

```
constraint c2
{
  threshold2 <= MAX_PART_HIGH;
  threshold2 > threshold1 + MIN_PART_DIFFERENCE;
}
constraint c3
{
  enabled          in { TRUE };
  intEnabled       in { TRUE };
  thresholdEnb     in { TRUE };
  type             in { HW_TASK, SW_TASK };
}
```

*Example 2.*

```
while (!dieSeen)
{
  if (numData <= part.getLowThreshold())
  {
    extraction_type = 100;
    recycle_type    = 0;
  }
  else if (numBuf < part.getUpThreshold())
  {
    extraction_type = 70;
    recycle_type    = 30;
  }
  else
  {
    extraction_type = 0;
    recycle_type    = 100;
  }
  @(posedge stb.grant);
}
```

*Example 3.*

The DRIVER module implements the CPU task. This is the task with the capability to fill up the FLM internal memory with the data that have to be provided to partitions. An internal mechanism allows the CPU task to provide some additional data for the HW/SW tasks if resources dropped too low. In this case the suitable packed fields for the bus transaction are put into the medium (HIGH) *mailbox*. The goal of the Engine is to drive all the incoming packets from the three *mailboxes* to the STBus lines, with respect to the different priorities.

At the top level, a COVERAGE module has been built in order to steer the simulation until some requirements (goals) have been satisfied.

Two main goals have been defined for this class: (1) a functional coverage of the touched states of the DUT internal FSM, including the definition of all the allowed transactions; (2) a coverage for counting the number of partitions created, used and released.

By setting up one or both the coverage queries, the simulation can be run until the goal, e.g. 90% of all state transactions touched, is reached.

Random generation of tasks and of extraction/recycle sequences allows reaching corner cases. Exploration of corner cases can be enhanced by using the “Boundary Conditions” feature, a VERA mode that covers random generation focusing on the boundaries of specified signal ranges.

## 5.0 Test results

Several simulations were run and some unexpected behavior arose under different conditions. In this section a short description depicts what was observed:

- (1) for SW partitions, when an extraction is performed, after some other operations, the returned status is marked as valid but the data extracted is not (‘0’ returned);
- (2) after lengthy simulations, sometimes the DUT returns data for a TASK, belonging to partition  $x$ , flagged as requested by partition  $y$ ;
- (3) it is not possible to create, and hence to enable, a partition with an ID greater than 40.

All these problems were notified to the DUT design team and fixed in the next RTL release.

## 6.0 Improving the environment

The test environment described above is affected by two shortcomings: the intrinsic slow speed of simulation tools and the lack of advanced code coverage functionality. These shortcomings are addressed and explained below.

### 6.1 Code Coverage

A dedicated code coverage tool (with capabilities such as reporting whether all the RTL code has been stimulated or if there are statements, conditions or paths never accessed) was linked to the simulator that, in turn, was driven by VERA.

Unfortunately, since the simulated code needed to be instrumented, the linking step added some overhead to the simulation tool. It increased both code size (about 20%) and simulation time. This resulted in a significant decrease in performance that can be clearly seen in table 1.

	Pure Simulation	Adding Code Coverage
<b>CPU time for a run time of 1.2 ms</b>	8.2 s	36.9 s

*Table 1. Simulation performance with and without code coverage*

Run time grew by 4.5 times, which was partially due to the increase in code size but was mostly due to the interaction between the three tools. On the other hand, it was not only possible to monitor the “functional” coverage but the “code” coverage as well. This way it was possible to see if some portion of the code had not been covered or if some path had not been walked or again if some state of the FSM was not accessed.

Using these metrics it was possible to isolate the part of the FSM tree that was not being exercised at all, which was due to poor exploitation of VERA’s random generation capabilities. The seed generation of random stimuli and some other parameters were changed in order to tune the VERA testbench to reach a coverage target much closer to 100%. Only in this way it was

possible to explore all the significant RTL code, making sure that no corner case had been missed.

```

595 combinational: process (current_state, opcode, stb_opc, counter1, stb_req, int
596
597 constant intf_addr_space:std_logic_vector(11 downto 0):="001010000111";
598
4505 begin
600     case current_state is
601
602     -- reset
603     when ready=>
604
605
606
1931     intf_data_in<=zero32;
1931     intf_mem_wr<="000";
1931     intf_mem_rd<="000";
1931     intf_reg_wr<=zero32;
1931     intf_reg_rd<=zero35;
1931     intf_mem_addr<="00000000";
1931     stb_r_data<=zero32;
614     -- decoding opcode to number of words involved
615     case stb_opc(7 downto 4) is
616     when "0010" =>
819     616a count.limit<=1;
617     when "0011" =>
0     617a count.limit<=2;
618     when "0100" =>
0     618a count.limit<=4;
619     when "0101" =>
0     619a count.limit<=8;
620     when others =>
1112     620a count.limit<=0;
621     end case;
622
623     when write=>
1841     626     intf_reg_rd<=zero35;

```

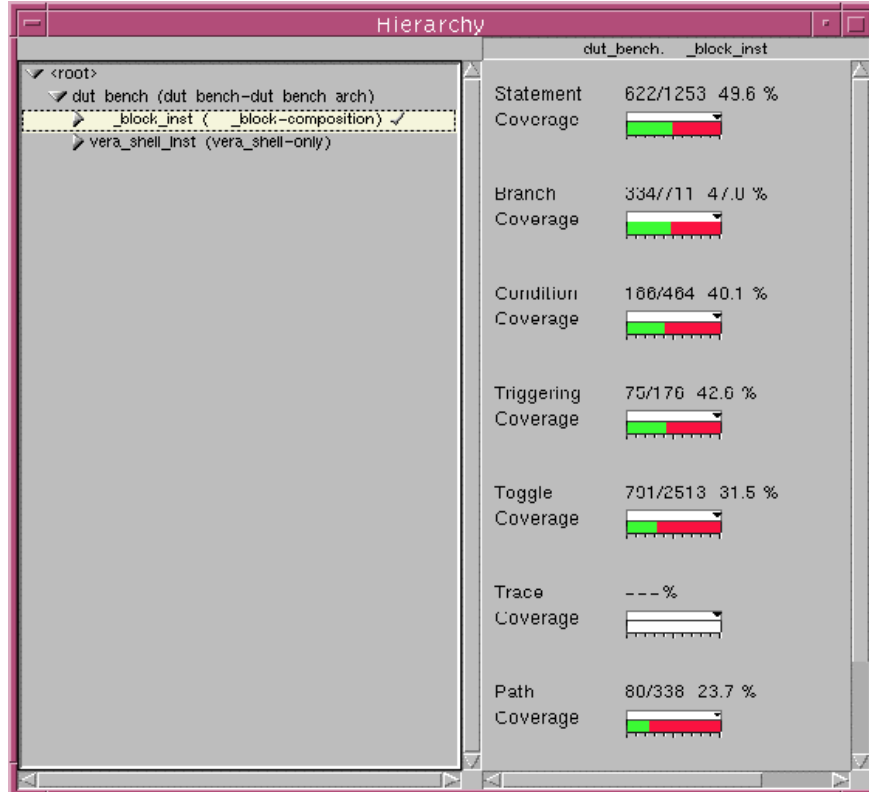


Figure 5. Code coverage metrics collected running with VERA

Another way to test a DUT could be *pure random* stimulus generation. Using this technique no real functionality can be defined (input signals are moving randomly) and hence code coverage is the only way to know whether the code has been properly tested or not.

Comparing coverage results of the original VHDL testbenches against the final VERA testbench, it was observed that the overall coverage percentage increased from about 25% up to about 85%. Not considering coding-style changes for purposes like synthesis (default cases) or debugging (ASSERT), the coverage percentage is even higher.

These results show how code coverage tools can help engineers to steer the VERA testbench in order to test more of the code (and hence improve functional testing of the HW block).

## 6.2 Speeding up simulation

One of the bottlenecks in verifying an RTL design (especially for complete SoC designs) is the slow performance of SW simulators (a few tens of Hz at most).

One possible and effective way to speed up RTL simulation is to replace the SW simulator with an HW emulator. Commercial HW emulators can run from 0.5 up to 25/30 MHz.

A preliminary analysis was performed in order to evaluate the expected advantage of mapping the design onto an emulator. Assuming  $A=E/N$ , where  $A$  is the circuit *activity*,  $E$  is the Toggle Count (i.e. the number of signal events, divided by the number of clock cycles during the given simulation) and  $N$  is the number of signals under control (see [5]), a value of *activity* of 8% (for ~2500 signals) was calculated for the DUT. This figure contributes to justify the expectation of a speed improvement, despite the number of signals at the simulator/emulator interface, since the co-emulation interface is almost synchronous.

Compiling the DUT with a cycle-based simulator (Cyclone), two internal *triggers* were found. This is another confirmation of the expected advantage of using the emulator.

It is not yet possible to link HVL tools directly to commercial emulators; one possible way to overcome this problem is to use a SW simulator as a bridge to interleave between VERA and the emulator.

This obviously results in a loss of performance within the complete system, mainly due to the simulator/emulator interface.

This link usually consists of a PCI connection that handles 64/128 bits up to 33MHz, which results in an inefficient data exchange between the simulator and the emulator. Several tests showed an average speed of few hundreds Hz. This limitation is due to the low level at which communication takes place (i.e. bits are put on the bus without any protocol).

Mapping the FLM block onto the adopted emulator took about 25 minutes (not including the memory modeling<sup>1</sup>), and the mapped design could run up to a maximum frequency of 650KHz. This was the stand-alone working frequency, paced by the simulator/emulator connection. The block was compiled on the emulator keeping full visibility of the signal hierarchy, which means that it is possible to trace all the signals inside the architecture as in a normal SW simulation.

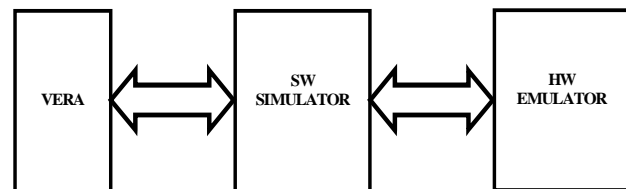


Figure 6. Linking the test environment to HW emulator.

<sup>1</sup> One of the main drawbacks of using HW emulators is that memory models must be replaced by ad hoc models for emulation; this is an overhead for the verification task. To face the problem of the memory mapping (moving from behavioral description to silicon emulation), an ad hoc memory verification environment has been built, still based on VERA, that alleviates the problem (see reference [6]).

Moving to HW emulation has several advantages, such as:

- synthesis sanity check for the DUT
- simulation speed up
- in circuit connection of hard macro blocks

This trial resulted in a significant performance increase, as shown in table 2.

	<b>SW Simulation</b>	<b>HW Co- Emulation</b>
<b>CPU time for a run time of 2.7 ms</b>	~ 22 s	~ 11 s

*Table 2. SW simulation vs. HW emulation performance.*

Run time decreased by a factor of about 2. The result could be even better if the SW simulator bridge were removed.

Despite the emulation and simulation environments being functionally equivalent, it is not possible to access internal signals from the VERA side through the SW simulator interface. For this reason in the emulation environment the functional coverage of the DUT Finite State Machine was removed from the VERA test suite.

## **7.0 Conclusions**

A test environment exploiting the capabilities of an HVL tool such as VERA has been demonstrated. This verification flow proved the added value of VERA versus a plain VHDL test suite. Randomization features, functional coverage, reusability and greater levels of abstraction are some of the key benefits of a VERA testbench. Synchronization features, like mailboxes and semaphores, give the possibility to enhance the high level structure of the testbench.

Analyzing code coverage statistics also allowed the test designers to drive and to tune the VERA testbench in order to better explore the simulation space.

Connecting VERA to an HW emulator via a SW simulator increased simulation performance and synthesis sanity checking. This environment could be the starting point for direct linkage between VERA and commercial emulators, by removing the simulator bridge.

The VERA testbench helped identify some DUT erratic behavior that would have been very difficult to find otherwise.

## 8.0 Acknowledgments

We would like to thank Shawn Honess (Synopsys) for his helpful contribution in the VERA code design. Thanks to Lino Buttà and Jim Nicholas (STM) for allowing us to mention the STBus protocol. Pascal Moniot and Mario Diaznava (STM) gave us the possibility to work on and publish information about the FLM block; this block is under a European patent (see reference [9]).

## 9.0 References

- [1] VERA Users' Manual ver. 4.3.
- [2] Transeda, "VHDLCover Users' Manual ver. 5.001", June 2000.
- [3] Mentor Graphics, "Celaro Users' Manual ver. 2.3\_2", Feb. 2000.
- [4] Sforza et al., "A Design For Verification Methodology", ISQED 2001 accepted paper.
- [5] Sforza et al., "Functional Verification of a SoC - A Compared Evaluation of Different Tools", E-SNUG 2000 proceedings.
- [6] Sforza et al., "Verification of Memory Models for emulation", CICC'01 submitted paper
- [7] M. Keating, P. Bricaud, "Reuse Methodology Manual – Second Edition", Kluwer Academic Publisher, 1999.
- [8] J. Bergeron, "Writing Testbenches – Functional Verification of HDL Models", Kluwer Academic Publisher, 2000.
- [9] P. Moniot, M. Coppola, European Patent 98/16156.