VHDL to SystemVerilog: Constrained Random Verification of a USB 2.0 Host Controller Sub-System

Richard McGee, Paul Furlong
Silicon & Software Systems (S3)
Dublin, Ireland
richard.mcgee, paul.furlong@s3group.com

Fabian Delguste
Synopsys, Inc.
Fabian.Delguste@synopsys.com

ABSTRACT

SystemVerilog has been accepted as an IEEE standard for which all the major CAD and IP vendors have announced support. Many observers view it as becoming a widely accepted industry standard verification language.

It is now possible to build object-oriented and constrained random features into a verification environment without being locked into using one proprietary tool to run that environment. Many users previously deterred from committing to a proprietary solution, along with users of existing proprietary solutions, are seeing benefits in converting their environments to the new standard.

This paper shows how a SystemVerilog test environment with re-usable components can be quickly set up and used to comprehensively test a USB Host Controller sub-system with a mixture of directed and constrained random test cases. The environment has been implemented to replace an existing VHDL testbench by a user with no previous SystemVerilog, OpenVera or *e* language experience. The paper describes a Synopsys Reference Verification Methodology (RVM) implementation, a first step towards the use of the ARM/Synopsys Verification Methodology Manual (VMM).

A VHDL USB 2.0 Host Controller sub-system consisting of USB EHCI and OHCI AHB masters, DMA interface to embedded DRAM, buffering and arbitration logic, will be used as an example.

Readers will learn about the basic concepts of SystemVerilog constrained random testing and how to quickly construct a test environment and simulate using VCS 2005.06. The paper also describes the use of SVA assertions to complement functional coverage results for the system.

Table of Contents

1.0 Introduction 4
2.0 The USB 2.0 Host Controller Sub-System (DUT) 4
2.1.1 Verification Strategy 6
2.1.2 Functionality to be Tested 6
3.0 The Existing VHDL Verification Environment 7
3.1 Flow Overview 7
3.2 Directed VHDL Verification Environment Overview 8
3.2.1 USB DMA Logic VHDL Verification Environment 8
3.2.2 USB Top-Level Verification Environment 8
4.0 The SystemVerilog Verification Environment 10
4.1 SystemVerilog UHU Verification Environment Overview 11
4.1.1 AHB Generator 11
4.1.2 Object-oriented Class Structure 12
4.1.3 AHB Master 13
4.1.4 Leon Generator 14
4.1.5 Leon Master 14
4.1.6 DRAM Read and Write Monitors 14
4.1.7 Scoreboard 15
4.1.8 Building the Environment 16
4.1.9 Constructing the Testbench 17
5.0 Defining the Testcases and Satisfying the Verification Requirements 17
5.1 Constraining the Testcases 18
5.1.1 Test_contention 19
5.1.2 Test_random 20
5.1.3 Test_coherent_e/ohci 21
5.1.4 Test_interrupt 21
5.2 Capturing Functional Coverage using SystemVerilog Assertions 22
5.2.1 Example 1 - F.UHU.4.0 OHCI AHB READ Split Response 22
5.2.2 Example 2 - F UHU.5 for EHCI Core Interrupt Coherency 23
5.3 Reporting the Results 24
6.0 Discussion of Results 26
6.1 Conclusions and Recommendations 26
6.2 Advantages and Disadvantages of SystemVerilog versus a VHDL Directed Approach 27
7.0 Acknowledgements 28
8.0 References 28

Table of Figures

Figure 1 – USB Enabled SoC 4
Figure 2 – USB Sub-System Block Diagram 5
Figure 3 – VHDL UHU DMA Logic Verification Environment 8
Figure 4 – VHDL UHU Verification Environment 9
Figure 5 – RVM Testbench Structure 10

Figure 6 – SystemVerilog UHU Verification Environment..... 11
Figure 7 – DRAM Interface Protocol..... 15
Figure 8 – Unified Report Generator HTML Coverage Reporting..... 25
Figure 9 – S3 NanoFlow CGI Webscript Verification Regression Reporting..... 26

1.0 Introduction

This paper documents an investigation into the feasibility and advantages of switching from using a directed VHDL/Verilog based verification flow to using a SystemVerilog constrained random environment. For the purposes of the investigation a USB 2.0 Host Sub-System in VHDL was selected as the Device Under Test (DUT). The device was previously verified by S3 using a traditional VHDL testbench running directed testcases and worked 100% first time right in Silicon on an image processing System on Chip (SoC) [1]. This was considered a good candidate for the investigation as a number of common SoC elements are present in the design including: two AHB interfaces, a customized Leon processor bus interface, arbitration logic and a DMA interface to embedded DRAM.

The objective of the investigation was to develop a SystemVerilog environment to cover the existing verification plan for the device and in particular to determine:

1. Is there a large ramp up required for our verification engineers in switching to SystemVerilog?
2. If our existing verification flow is resulting in first time right silicon is there a significant advantage in switching to a constrained random flow (e.g. time-saving)?
3. Are existing CAD tools mature enough to support the SystemVerilog features we require?
4. Is co-simulation with existing VHDL designs an issue?
5. How important to constrained random verification is having a well defined verification methodology ?

2.0 The USB 2.0 Host Controller Sub-System (DUT)

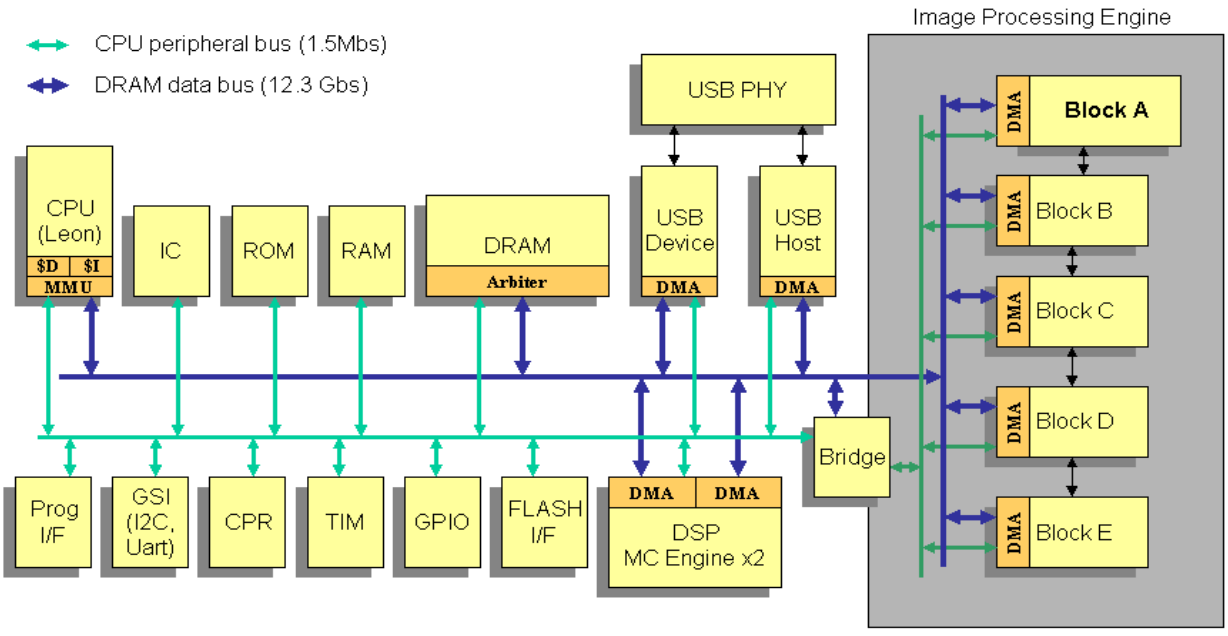

Figure 1 – USB Enabled SoC

A typical SoC with USB 2.0 host and device connectivity including on-chip PHY is shown in Figure 1. The USB host and device can be configured by a Leon CPU [1] through a low-speed CPU peripheral bus. USB data traffic, with rates up of to 480 Mbs in USB 2.0 High Speed mode, can be DMAed into the on-chip embedded DRAM via a separate high bandwidth DRAM data bus. The DRAM arbiter provides guaranteed quality of service in terms of bandwidth and access latency to all its requestors. For the USB connections this means that the SoC should never become a bottleneck as it will always be able to sink and source USB data at the rate of the external USB networks.

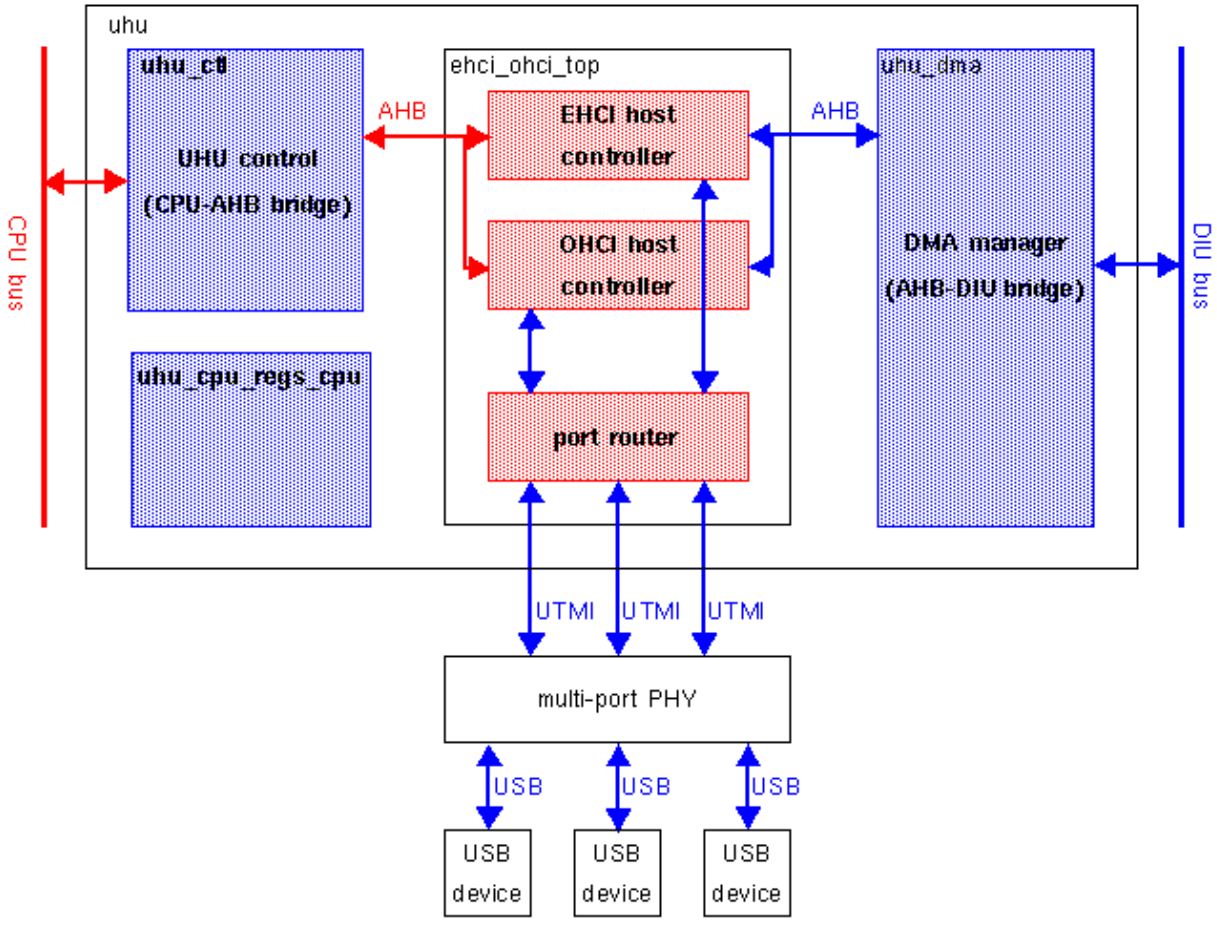

Figure 2 – USB Sub-System Block Diagram

The USB Controller Host Sub-System or UHU, the subject of this paper, is shown in more detail in Figure 2. The host core in the UHU, the *ehci_ohci,* is a USB2.0 compliant 3rd party Verilog IP core from Synopsys. It contains an Enhanced Host Controller Interface (EHCI) controller and an Open Host Controller Interface (OHCI) controller. The EHCI controller is responsible for all High Speed (HS) USB traffic. The OHCI controller is responsible for all Full Speed (FS) and Low Speed (LS) USB traffic. The multi-port PHY provides three downstream USB ports for the UHU.

In addition to the *ehci_ohci* host controller core the UHU also contains

- a connection to the CPU peripheral bus
- CPU configuration registers in *uhu_cpu_regs_cpu*
- an AHB bridge to the *ehci_ohci* allowing its registers to be also configured by the CPU
- a DMA manager, *uhu_dma*, connecting the *ehci_ohci* via a second AHB bridge to the high speed DRAM bus or DIU.

The CPU interface is the master on its AHB interface to the *ehci_ohci* whereas the *ehci_ohci* is the master on the AHB interface to the DRAM DMA logic in *uhu_dma*.

2.1.1 Verification Strategy

The verification of the UHU is split into 2 parts:

1. Exhaustive testing of the UHU CPU configuration and DRAM DMA logic by replacing the *ehci_ohci* by AHB master/slave BFMs.
2. Complete testing of the UHU including the USB *ehci_ohci* host controller cores.

This two-step verification strategy was adopted because exhaustive testing using the complete UHU is difficult since the USB host functionality is partitioned between the hardware UHU and software EHCI/OHCI Host Controller Driver stacks. The development effort for these Host Controller Drivers is considerable and the CPU MIPs requirements large so only simplified drivers were employed. With the simplified host controller drivers it was very difficult to hit all the DMA corner cases so separate testing of the DMA logic was employed. This paper describes the work required to verify the UHU DMA logic with the EHCI/OHCI cores replaced by AHB master/slave BFMs.

2.1.2 Functionality to be Tested

The subset of functionality to be tested as part of this investigation of SystemVerilog is shown in Table 1.

Table 1 – UHU DMA logic Verification Requirements

Point of Functionality	Verification Requirement
Read/Write Burst sizes	
F UHU.1.0	Verify read operation for different AHB burst sizes (1 -1024) for OHCI and EHCI masters.
F UHU.1.1	Verify write operation for different AHB burst sizes (1 -1024) for OHCI and EHCI masters.
F UHU.1.2	Verify different combinations of reads and writes i.e. read/read, read/write, write/write, write/read combinations.
Arbitration	
F UHU.2.0	Verify AHB arbitration fairness between OHCI and EHCI masters.
Read/Write Coherency	
F UHU.3.0	Verify read/write coherency for both OHCI and EHCI masters.
AHB Split Response	
F UHU.4.0	Verify read operation using AHB split responses for OHCI and EHCI slaves. Verify reads split on crossing 256-bit word boundary.
F UHU.4.1	Verify write operation using AHB split responses for OHCI and EHCI slaves. Verify writes split on crossing 256-bit word boundary.

Interrupt Coherency	
F UHU.5.0	Verify for core interrupts any writes in local UHU buffer need to be flushed before interrupt is generated and no new transactions allowed until interrupt is generated.
F UHU.5.1	Verify that any ongoing AHB write transfer is split when a core interrupt is generated and the split is completed when the buffer is flushed and the interrupt has been generated.

These verification requirements need some further explanation of the UHU functionality:

- Read/write burst sizes (F UHU.1.0/1)

  The USB maximum packet size can be up to 1024 bytes so the EHCI/OHCI cores may read or write packets in the range 1-1024 bytes.
- Arbitration (F UHU.2.0)

  Both EHCI and OHCI are separate AHB masters to the DMA logic with a round-robin arbitration policy being implemented. Arbitration is won by the EHCI by default when there have been no previous requests.
- Read/write coherency (F UHU.3.0)

  Read/write coherency is extremely important i.e. a read operation following a write operation to the same DRAM address must always read the data just written.
- AHB Split Response (F UHU.4.0/1)

  In order to prevent long EHCI bursts locking out OHCI DRAM access and vice-versa AHB split response functionality has been implemented. When an AHB burst crosses a 256-bit DRAM word boundary (the embedded DRAM word-size is 256-bits) the AHB access is split and an access from another master is allowed. In this way if both EHCI and OHCI request accesses of long lengths then the resulting accesses will be alternating bursts of 8x32-bit words or 256-bits by each of the EHCI and OHCI masters.
- Interrupt coherency (F UHU.5.0/1)

  Interrupt coherency must also be guaranteed i.e. an interrupt must not be generated in the case of a USB IN transfer until the data has actually been written into the DRAM and therefore can be safely read by the CPU.

3.0 The Existing VHDL Verification Environment

3.1 Flow Overview

The S3 verification methodology used to verify the device involved firstly the creation of a verification plan. This plan in turn contained tables of Verification Requirements such as those shown in Table 1 - bullet point lists of functionality to be covered by the verification test suite. A DUT audit by the verification team was used to generate the verification requirements. This audit took as input: the block specification, relevant standards, available verification IP, input from the block designer and the S3 block verification check list. The resulting requirements went through iterations of review with a view to capturing all the corner case operations where a non-conformance might escape detection by the verification suite.

Functional coverage was measured manually by review of the verification requirements versus the test suite. This was flagged as being very time consuming and was highlighted as an area where automation using assertions for coverage calculation could improve efficiency. Full

coverage was considered as met when all verification requirements were covered by the test suite and statement code coverage was 100% (our goal was 100% branch and statement code coverage but branch coverage was not available with the version of the code coverage tool we were using).

Directed tests were written to cover the functionality described in the verification requirements. In all cases individual tests had to be written to hit the verification requirements, this again was a bottleneck in the process of closing out functional coverage. The directed tests were very procedural-like defining lists of CPU BFM and AHB BFM operations. All tests were self-checking with AHB transactions checked by an AHB monitor. Some limited randomization was used in the generation of address and data values and delays. A total of 20 separate directed tests were required to completely verify the UHU functionality listed in Table 1. Developing these tests took a considerable amount of effort although the same testbench structure was used for all tests.

3.2 Directed VHDL Verification Environment Overview

3.2.1 USB DMA Logic VHDL Verification Environment

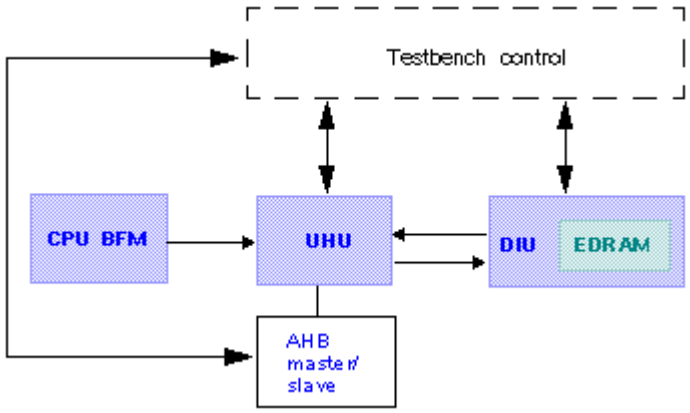

Figure 3 – VHDL UHU DMA Logic Verification Environment

The VHDL verification environment that was used is shown in Figure 3. This comprises the DUT (UHU) together with a CPU BFM, actual RTL for the DRAM arbiter (DIU) and AHB master/slave BFM's replacing the EHCI/OHCI cores. This environment facilitated extensive directed testing of the UHU DMA and CPU interface logic.

3.2.2 USB Top-Level Verification Environment

As noted in Section 2.1.1, a second verification phase included complete testing of the UHU including the USB *ehci_ohci* host controller cores. The testbench is shown in Figure 4. The UHU requires a very complex device driver. The Synopsys task driven verification environment used to test the *ehci_ohci* core standalone was modified to work at UHU level. This includes a USB device model supplied by Synopsys. The testbench controls the driver models by means of transaction-level Verilog task calls. Because of the limited functionality of the device driver, and

also the limited functional coverage of the top-level testcases, it was necessary to exhaustively test the USB DMA buffer logic separately.

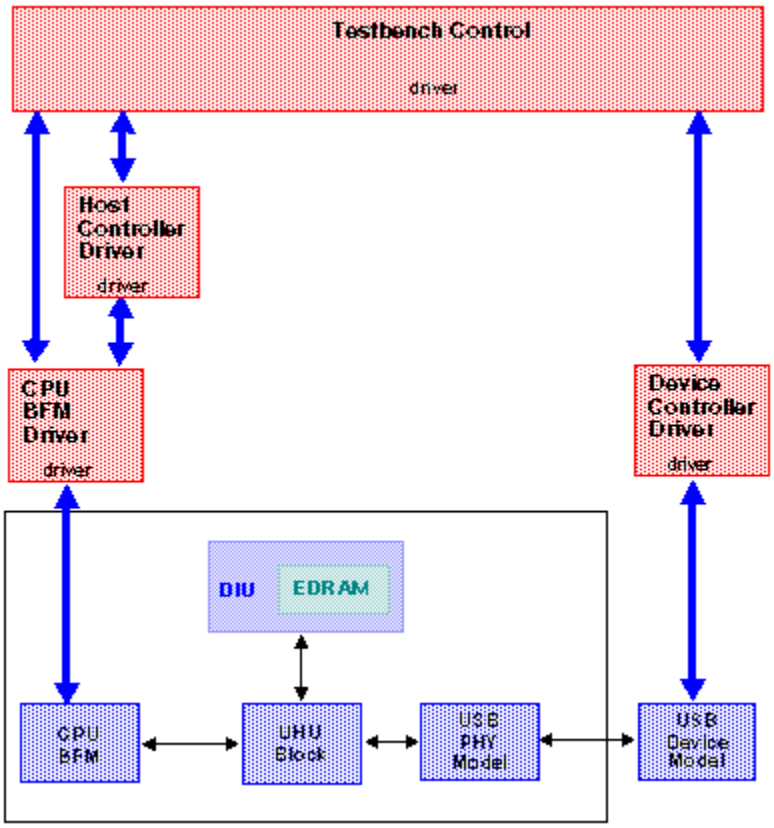

Figure 4 – VHDL UHU Verification Environment

4.0 The SystemVerilog Verification Environment

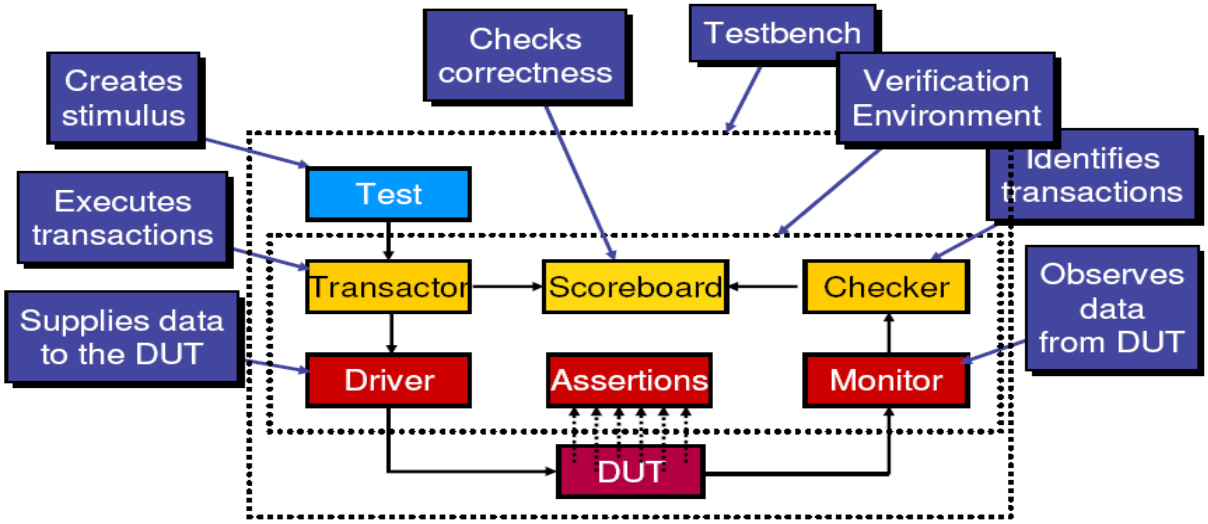

Figure 5 – RVM Testbench Structure

The verification environment developed to allow SystemVerilog testing of the UHU is shown in Figure 6. This environment follows the conventions of the Synopsys RVM (Reference Verification Methodology) SystemVerilog methodology. The RVM is a structured methodology for object-oriented constrained random verification using SystemVerilog. The full RVM makes use of the ARM/Synopsys Verification Methodology Manual (VMM) [2]. The RVM contains guidelines, coding styles and base classes. Here we implement an RVM-lite methodology which does not use the RVM base-classes. RVM-lite is a first step to the implementation of the full RVM. The RVM emphasizes a functional coverage driven verification methodology which matches our use of Verification Requirements

A generic RVM testbench structure is shown in Figure 5 [3]. Testing is performed at the transaction level using a transaction generator called by the testcase, a checker which identifies transactions from the DUT, and a scoreboard to check their correctness. Drivers and monitors map the transactions to the pin-level of the DUT. Assertions check the low-level signaling of the DUT for correct behaviour. The SystemVerilog test environment developed for the UHU follows this RVM testbench structure.

4.1 SystemVerilog UHU Verification Environment Overview

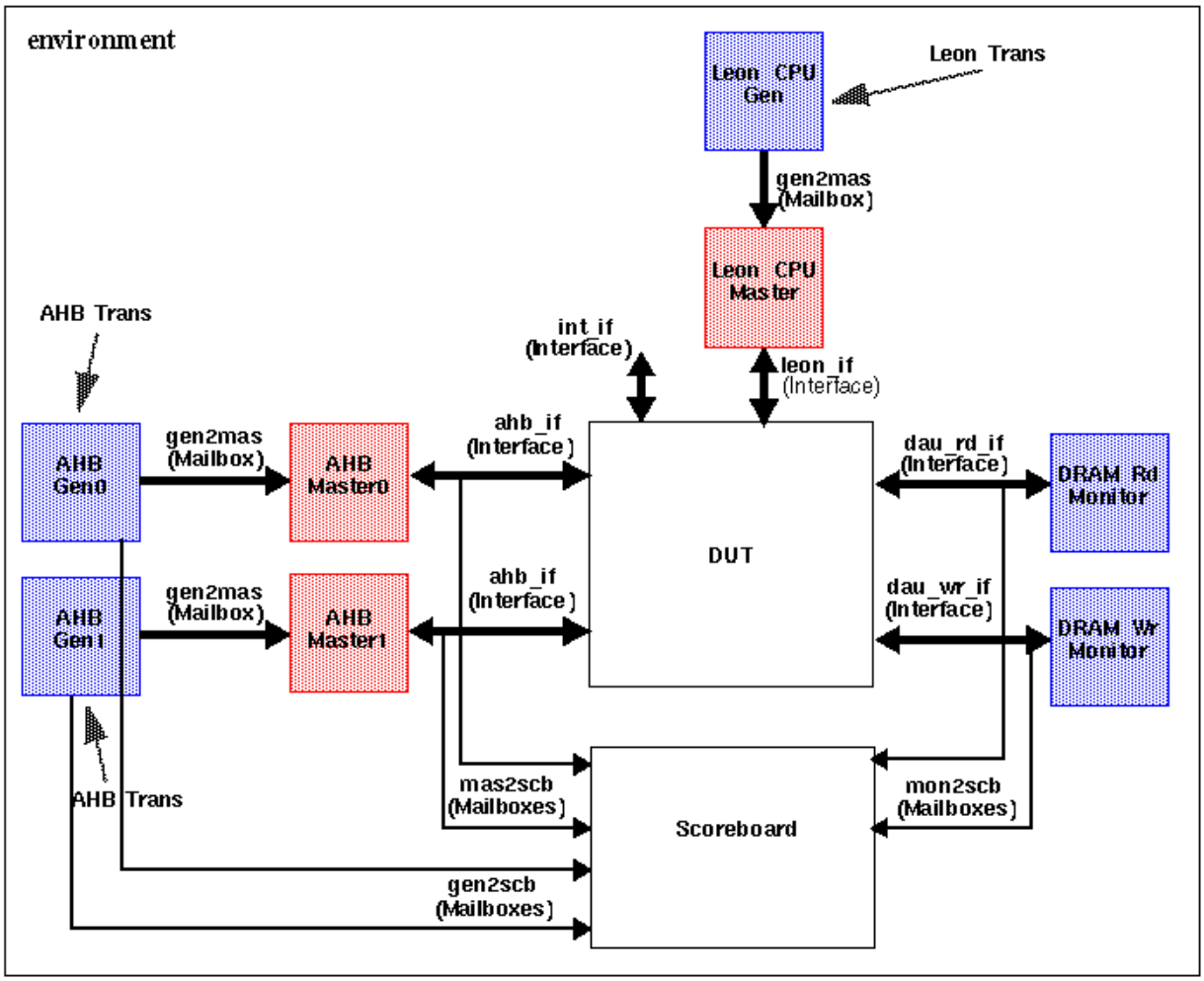

Figure 6 – SystemVerilog UHU Verification Environment

The SystemVerilog UHU verification environment in Figure 6 shows the transaction generators (1 Leon and 2 AHB), read and write DRAM monitors and a scoreboard. The scoreboard checks that transactions on the AHB side result in correct DRAM reads and writes on the DRAM interface. SystemVerilog mailboxes are used to interface the generators with their drivers and to interface to the scoreboard. SystemVerilog interfaces are used at the DUT pin level for the AHB, Leon, interrupt and DRAM read and write pin I/O. The environment is described in more detail in the following sections.

4.1.1 AHB Generator

The AHB Generator creates bursts of AHB transactions. There are 2 AHB generators – 1 for the EHCI and 1 for the OHCI. The *ahb_trans* class is shown below.

```
parameter AHB_ADDR_WIDTH = 32;
typedef bit [AHB_ADDR_WIDTH-1:0] ahb_addr_t;
parameter AHB_DATA_WIDTH = 32;
typedef bit [AHB_DATA_WIDTH-1:0] ahb_data_t;
typedef enum {SINGLE, INCR, WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16} ahb_burst_e;
typedef enum {AHB_READ, AHB_WRITE, AHB_IDLE} ahb_trans_e;
class ahb_trans;
        rand ahb_addr_t addr;
        rand ahb_data_t data;
        rand ahb_burst_e burst;
        rand ahb_trans_e transaction;
        rand integer burstlength;
        … …
        function ahb_trans copy();
         … …
        endfunction: copy
endclass
```

The AHB transactions are all completely random i.e.

- Random transaction type i.e. AHB READ, WRITE or IDLE
- random 32-bit address
- random 32-bit data
- random burst length
- random burst type

Once randomly generated the bursts are posted by an AHB generator to a *gen2mas_e/ohci* mailbox where they will be read by the corresponding AHB master. The AHB generator also posts the count of the total number of AHB words generated to the scoreboard via a *gen2scb_e/ohci* mailbox. This is used by the scoreboard to ensure checking has been performed on all the generated AHB words.

4.1.2 Object-oriented Class Structure

The AHB Generator, similar to the other testbench components, follows an object-oriented class structure:

- Objects are instantiated (actually handles to objects are instantiated) – in this case a random AHB transaction and mailboxes.
- Methods are provided to perform operations on the objects.
- A constructor function *new* performs initialization by allocating memory for the objects and initializing variables and connectivity e.g. of the mailboxes.
- A task *main* has a loop which generates AHB transactions and posts them to the mailbox.

The classes are self contained – they define both the data objects and the operations that can be performed on them. This makes them highly portable and reusable in other verification environments. These concepts seem new at first, from the background of directed VHDL and Verilog verification, but are easy enough to grasp especially if one has had any prior exposure to C++ or SystemC.

```
class ahb_gen;

        // Random AHB transaction
        rand ahb_trans rand_tr;

        //AHB Transaction mailbox
        mailbox gen2mas, gen2scb;

        …
        // Constructor
        function new(mailbox gen2mas, gen2scb, …);
          this.gen2mas    = gen2mas;
          this.gen2scb    = gen2scb;
          rand_tr         = new;
          …
        endfunction

        // Method aimed at generating transactions
        task main();
          while(!end_of_test())
          begin
            // Wait & Get a transaction
            rand_tr = get_transaction();
             … …
            gen2mas.put(rand_tr);
          end // while (!end_of_test())
        endtask
endclass
```

4.1.3 AHB Master

The AHB Master converts AHB READ, WRITE and IDLE transactions received via the *gen2mas_e/ohci* mailbox into cycle accurate address, data and control signals on the *ahb_if* interface connected to the DUT complying to the AHB protocol. SystemVerilog clocking blocks are used in the interfaces to create explicit synchronous timing domains. The AHB Master also posts the generated transactions into separate read and write mailboxes *mas2scb_e/ohci_rd* and *mas2scb_e/ohci_wr* where they can be read by the scoreboard.

The master is a stripped down version of a full AHB master. Some points to note are:

- The AHB master always implements an INCR burst type i.e. an incrementing burst of indeterminate length. This is because we want to test bursts in the range 1 to 1024. The test cases constrain the burst type to INCR.
- A limitation of the AHB master used in this investigation is that it will only generate 32-bit word transactions. So we test bursts in the range 1 to 1024 32-bit words rather than 1 to 1024 bytes but the principle remains the same.
- The transfer type of the first access in a burst, the AHB *htrans* signal, is always NONSEQ for the first transfer in a burst and SEQ for subsequent transfers.
- The AHB Master is designed to handle AHB split transactions. The AHB arbiter in the UHU will generate a split when the AHB address crosses a 256-bit DRAM word

boundary. Recall from Section 2.1.2 “Functionality to be tested” that this is intended to allow equal access to the UHU for both EHCI and OHCI.

4.1.4 Leon Generator

The Leon generator is used to generate Leon READ and WRITE transactions to configure the DUT and to read back status. The transactions are posted to a mailbox *gen2mas_leon* where they can be read by the Leon Master. The *leon_trans* class is shown below.

```
parameter LEON_ADDR_WIDTH = 12;
typedef bit [LEON_ADDR_WIDTH-1:0] leon_addr_t;
parameter LEON_DATA_WIDTH = 32;
typedef bit [LEON_DATA_WIDTH-1:0] leon_data_t;
typedef enum {LEON_READ, LEON_WRITE, LEON_IDLE} leon_trans_e;
class leon_trans;
        randc leon_addr_t addr;
        rand leon_data_t data;
        rand leon_trans_e transaction;
        … …
        function leon_trans copy();
        … …
        endfunction: copy
endclass
```

The Leon transaction address is defined as *randc* i.e. no values are repeated until all the values in the range have been generated. The Leon addresses are constrained within the testcase to include only the addresses of valid configuration registers. In the testcases we are only using Leon WRITE transactions to configure the DUT i.e. we are not reading back status. The Leon generator will generate Leon transactions until the maximum number of Leon transactions is reached, typically set as the number of registers to be configured.

4.1.5 Leon Master

The Leon Master converts Leon transactions received via its *gen2mas_leon* mailbox into cycle accurate address, data and control signals on the *leon_if* interface according to the AMBA APB-like CPU bus protocol used on the SoC.

4.1.6 DRAM Read and Write Monitors

Each DRAM monitor is a reactive transactor i.e. it responds to a request on the DRAM interface with a randomized DRAM transaction. The *dau_trans* class is shown below.

```
parameter DAU_ADDR_WIDTH = 22;
typedef bit [DAU_ADDR_WIDTH-1:5] dau_addr_t;
parameter DAU_DATA_WIDTH = 64;
typedef bit [DAU_DATA_WIDTH-1:0] dau_data_t;
typedef enum {DAU_READ, DAU_WRITE} dau_trans_e;
class dau_trans;
        dau_addr_t addr;
        dau_trans_e transaction;
        rand dau_data_t data[4];
        rand integer ack_delay;
        … …
        function dau_trans copy();
        … …
        endfunction: copy
endclass
```

As the DRAM data bus is 64-bits wide and each DRAM transaction is 256-bits (recall the DRAM word size is 256-bits) the DRAM data element is a 4x64-bit array. The DRAM transaction contains randomized data and a randomized acknowledge delay after which the data and relevant control signals are driven on the DAU interface. The DRAM acknowledge delay is constrained within the monitors as follows:

s = tr.randomize() with {ack_delay > 4; ack_delay < 256;};

The monitors post the DRAM transactions to the scoreboard mailboxes *mon2scb* for comparison with the data that is returned on the AHB interface. The DRAM read and write protocols are shown in Figure 7.

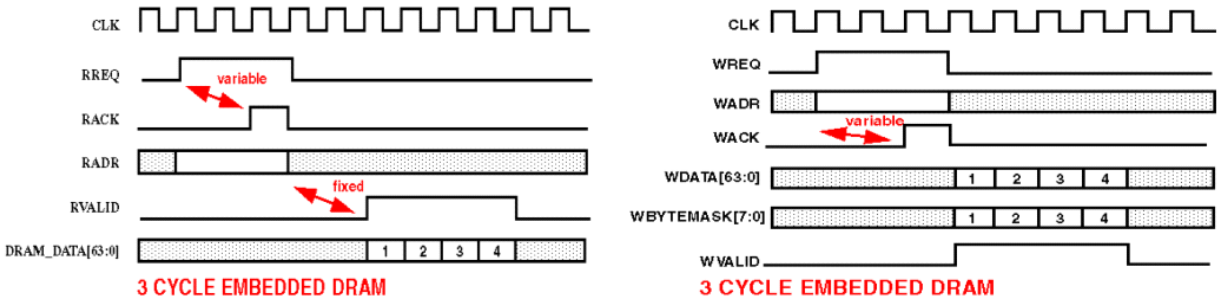

Figure 7 – DRAM Interface Protocol

4.1.7 Scoreboard

The scoreboard checks that writes on the AHB side are reflected as correct DRAM write accesses and that reads on the DRAM side are reflected as correct AHB read accesses.

For writes the scoreboard stalls until a transaction has been posted in the *mon2scb_dau_wr* mailbox by the DRAM write monitor. When a transaction appears it looks to match the transaction address with the top-most element in each of the EHCI and OHCI master write-to-scoreboard (*mas2scb_e/ohci_wr*) mailboxes. It does this by doing a *try_peek* into each of the

AHB master write-to-scoreboard mailboxes. Once a matching transaction address has been found then a *get* is performed and the data fields checked. Mismatches result in errors being generated.
For reads the scoreboard stalls until a transaction has been posted in one of the EHCI and OHCI master read-to-scoreboard (*mas2scb_e/ohci_rd*) mailboxes. When a transaction appears it looks to match the transaction address with the top-most element in the *mon2scb_dau_rd* mailbox posted by the DRAM read monitor. It does this by doing a *try_peek* into the DRAM read monitor to scoreboard mailbox. Again once a matching transaction address has been found then a *get* is performed and the data fields checked. Mismatches result in errors being generated.
A check is also performed to ensure that all words have been transmitted/received correctly using the count of the total number of AHB words generated read from the *gen2scb_e/ohci* mailboxes as noted in Section 4.1.1.

4.1.8 Building the Environment

The complete verification environment shown in Figure 6 is built in an object-oriented manner within the *environment* class.

```
class environment;
        // Transactors
        leon_gen        gen_leon;
        ahb_gen         gen_ohci;
        … …
        // constructor function
        function new(… …);
          gen_leon      = new(… …);
          gen_ohci      = new(… …);
          … …
        endfunction: new
        … …
        virtual task test();
          fork
                gen_leon.main();
                gen_ohci.main();
                gen_ehci.main();
          join
        endtask: test
        virtual task post_test();
          fork
                  wait(gen_leon.ended.triggered);
                  wait(gen_ohci.ended.triggered);
                  wait(gen_ehci.ended.triggered);
                  wait(scb.ended.triggered);
          join
        endtask: post_test

        task run();
                pre_test();
                test();
                post_test();
        endtask: run
endclass
```

The *environment* class instantiates handles to all the individual verification components. It has a constructor function which in turn calls the constructors of each component to initialize them and contains the tasks to invoke the methods of the testcase components.

The AHB and Leon Generators loop generating transactions and posting them to mailboxes until the number of transactions defined in the testcase have been generated. The AHB Masters, Leon Master, DRAM monitors and scoreboards *main()* tasks consist of infinite loops which read or write to mailboxes. The mailboxes are the glue between the various components – they block when full or empty. The testcase is defined to end using the *post_test()* task when the generators have ended and when the scoreboard has completed checking the number of AHB words created by the generators.

4.1.9 Constructing the Testbench

The top-level module of the testbench instantiates both DUT and testcase with the connectivity defined using interfaces. The testcase then instaniates the *environment* class, calls its constructor and runs the test.

```
program automatic test(ahb_if aif0,aif1,leon_if lif, dau_rd_if drif, dau_wr_if dwif, int_if intif);
    `include "env/env.sv" // Top level environment
     environment the_env;

      initial begin
            the_env = new(aif0,aif1,lif,drif,dwif);
           the_env.run();
           $finish;
      end
endprogram
```

The 5 testcases themselves are each described in separate files i.e.

- test_contention.sv
- test_random.sv
- test_coherent_ehci.sv
- test_coherent_ohci.sv
- test_interrupt.sv.

The testcases are compiled and run using the S3 *Nanoflow make* based design environment.

5.0 Defining the Testcases and Satisfying the Verification Requirements

The verification requirements of Table 1 are repeated in Table 2 but now cross-referenced against the testcases and the functional coverage assertions used to measure them. The following sections will discuss constraining the random testcases in order to hit these functional coverage points and capturing the coverage using assertions.

Table 2 – UHU DMA logic Verification Requirements with cross-referenced Testcases and Functional Coverage

Point of Functionality	Verification Requirement	Test Case	Functional Coverage
Read/Write Burst sizes			
F UHU.1.0	Verify read operation for different AHB burst sizes (1 -1024) for OHCI and EHCI masters.	All	Covergroup ahb_cov
F UHU.1.1	Verify write operation for different AHB burst sizes (1 -1024) for OHCI and EHCI masters.	All	Covergroup ahb_cov
F UHU.1.2	Verify different combinations of reads and writes i.e. read/read, read/write, write/write, write/read combinations.	All	write_write_cv write_read_cv read_write_cv read_read_cv
Arbitration			
F UHU.2.0	Verify AHB arbitration fairness between OHCI and EHCI masters.	contention	arb_cv arb_ehci_ohci_ehci_as arb_ehci_ohci_ehci_cv
Read/Write Coherency			
F UHU.3.0	Verify read/write coherency for both OHCI and EHCI masters.	coherent_ohci coherent_ehci	ahb0_write_read_diu_as ahb0_write_read_diu_as ahb1_write_read_diu_as ahb1_write_read_diu_as
AHB Split Response			
F UHU.4.0	Verify read operation using AHB split responses for OHCI and EHCI slaves. Verify reads split on crossing 256-bit word boundary.	contention random interrupt	read_ohci_split_as read_ohci_split_cv read_ehci_split_as read_ehci_split_cv
F UHU.4.1	Verify write operation using AHB split responses for OHCI and EHCI slaves. Verify writes split on crossing 256-bit word boundary.	contention random interrupt	write_ohci_split_as write_ohci_split_cv write_ehci_split_as write_ehci_split_cv
Interrupt Coherency			
F UHU.5.0	Verify for core interrupts any writes in local UHU buffer need to be flushed before interrupt is generated and no new transactions allowed until interrupt is generated.	interrupt	ehci_int_as ehci_int_cv ohci_int_as ohci_int_cv
F UHU.5.1	Verify that any ongoing AHB write transfer is split when a core interrupt is generated and the split is completed when the buffer is flushed and the interrupt has been generated.	interrupt	smi_int_as smi_int_cv ehci_int_write_phase0..7_cv ehci_int_read_phase0..7_cv ohci_int_write_phase0..7_cv ohci_int_read_phase0..7_cv smi_int_write_phase0..7_cv smi_int_read_phase0..7_cv

5.1 Constraining the Testcases

There are two general testcases - *test_contention* and *test_random* which exercise the majority of the verification requirements. Three more specific testcases *test_interrupt*, *test_coherent_ehci, test_coherent_ohci* target the specific interrupt and data coherency verification requirements.

The general approach is to add constraints to the completely randomized verification environment in order to hit the verification requirements. Beyond adding extra constraints to

each testcase the previously constructed verification environment is used in each testcase without modification. This leads to great productivity gains when writing testcases.

5.1.1 Test_contention

This is a test where both AHB masters attempt AHB READ and AHB WRITE transactions at the same time. The burst length of each transaction is in the range 1 to 1024.

To set up this testcase the AHB and Leon generators need to be suitably constrained. We have done this by extending the AHB and Leon base classes within the testcase.

Class *my_leon_gen* extends the Leon generator class so that Leon is constrained to only write to 2 particular configuration register addresses (32'h1C, 32'h20) with data 32'h11. The 2 Leon writes correspond to enabling AHB arbitration for both EHCI and OHCI (address 32`h1C) and enabling DRAM access for both read and write transactions (32'h20). The extended *my_leon_gen* class is shown below. As noted previously, the Leon address is randomized with the *randc* function which cycles through each of the 2 addresses. This is a good example of a random verification environment being constrained to produce directed testcase functionality.

```
class my_leon_gen extends leon_gen;
    …
  // Constraints applied here
  function leon_trans get_transaction();
    int s;
    rand_tr = new();
    s = rand_tr.randomize() with {addr inside {32'h1C, 32'h20};transaction == LEON_WRITE; data  == 32'h11;};
    if (!s)
      begin
        $display("leon_trans::randomize failed");
        $finish;
      end
    get_transaction = rand_tr;
  endfunction
endclass: my_leon_gen
```

Class *my_ahb_gen* extends the AHB generator class so that the transactions are constrained to be READs and WRITEs of burst lengths between 1 and 1024. The extended class is shown below. The constraint on the upper address bits is to ensure the address is within the 20 Mbit range that the USB host controller can process.

```
class my_ahb_gen extends ahb_gen;
  …
 // Constraints applied here
 function ahb_trans get_transaction();
   int s;
   rand_tr = new();
   s = rand_tr.randomize() with {addr[31:22] == 10'b0100000000; burstlength > 0;  burstlength < 1025;
          burst == INCR; transaction inside {AHB_WRITE,AHB_READ};};
   if (!s)
     begin
       $display("ahb_trans::randomize failed");
       $finish;
     end
   get_transaction = rand_tr;
 endfunction
endclass: my_ahb_gen
```

The testcase then instantiates and calls the constructors for the environment and the customized generators. The constructor calls for each of the generators set the number of transactions to generate for each. The environment *run* method is then called to start the testcase.

```
// Top level environment
environment the_env;

// Instanciate the customized generators
my_leon_gen my_leon_generator;
my_ahb_gen my_ehci_generator, my_ohci_generator;

initial begin
        // Instantiate the top level
        the_env = new(aif0,aif1,lif,drif,dwif);

        // Plug the new generators
        my_leon_generator = new(the_env.gen2mas_leon, 2, 1);
        the_env.gen_leon  = my_leon_generator;

        my_ehci_generator = new(the_env.gen2mas_ehci, 100, 1);
        the_env.gen_ehci  = my_ehci_generator;

        my_ohci_generator = new(the_env.gen2mas_ohci, 100 1);
        the_env.gen_ohci  = my_ohci_generator;

        // Kick off the test now
        the_env.run()
        $finish;
end
```

5.1.2 Test_random

This testcase is identical to *test_contention* except the AHB generators create IDLE transactions in addition to AHB READ and AHB WRITE transactions. The constraint in class my_ahb_gen becomes

s = rand_tr.randomize() with {addr[31:22] == 10'b0100000000; burstlength > 0; burstlength < 1025; burst == INCR; *transaction inside {AHB_IDLE, AHB_WRITE,AHB_READ};};*

Since the IDLE transactions are also generated with random burstlengths this testcase exercises the DUT with varying IDLE gaps between AHB transactions.

5.1.3 Test_coherent_e/ohci

The read/write data coherency tests are constrained using SystemVerilog imply statements to generate AHB_WRITE/AHB_READ transaction pairs with a fixed burst length of 8 to avoid AHB SPLIT transactions. For each AHB_WRITE/AHB_READ transaction pair we expect to see a DRAM write operation followed by a DRAM read operation in that order for coherency to be maintained.

```
class my_ahb_gen extends ahb_gen;

int ahb_tr_cnt = 0;

// Constructor
function new(mailbox gen2mas=null, gen2scb=null, int max_trans_cnt, bit verbose);
   super.new(gen2mas, gen2scb, max_trans_cnt, verbose);
endfunction

// Constraints applied here
function ahb_trans get_transaction();
   int s;
   rand_tr = new();
   s = rand_tr.randomize() with
      {
         ahb_tr_cnt==0  ->    {addr[31:22] == 10'b0100000000;
                        addr[4:0] == 5'b00000 ;
                        burstlength == 8; burst == INCR;
                        transaction == AHB_WRITE;
                        }
         ahb_tr_cnt==1  ->    {addr[31:22] == 10'b0100000000;
                        addr[4:0] == 5'b00000 ;
                        burstlength == 8; burst == INCR;
                        transaction == AHB_READ;
                        }
      };
   if (ahb_tr_cnt == 0)  ahb_tr_cnt = 1; else ahb_tr_cnt = 0;
  … …
endfunction
endclass: my_ahb_gen
```

5.1.4 Test_interrupt

The interrupt test is similar to the *test_contention* but also generates core interrupt events to the UHU DMA logic at random times.

5.2 Capturing Functional Coverage using SystemVerilog Assertions

The verification requirements of Tables 1 and 2 were captured using SystemVerilog assertions. Assertions are very good at capturing temporal relationships. A SystemVerilog covergroup was also used for capturing the AHB burstlengths generated and ensuring that they were distributed among various bins across the burstlength range of 1 to 1024. Cross coverage of the burstlength bins against the transaction type (AHB_READ, AHB_WRITE) was used in order to satisfy the verification requirements F UHU.1.0/2.0 of Tables 1 and 2.

Two examples of the use of assertions in capturing functional coverage of verification requirements are explored here. Table 3 shows verification requirement F.UHU.4.0 for the case of OHCI AHB Split response and verification requirements F UHU.5 for EHCI core interrupt coherency.

Table 3 – Selected UHU DMA logic Verification Requirements

Point of Functionality	Verification Requirement	TestCase	Functional Coverage
AHB Split Response			
F UHU.4.0	Verify read operation using AHB split responses for OHCI slave. Verify reads split on crossing 256-bit word boundary.	contention random interrupt	read_ohci_split_as read_ohci_sp
Interrupt Coherency			
F UHU.5.0	Verify for core interrupts any writes in local UHU buffer need to be flushed before interrupt is generated and no new transactions allowed until interrupt is generated.	interrupt	ehci_int_as ehci_int_cv
F UHU.5.1	Verify that any ongoing AHB write transfer is split when a core interrupt is generated and the split is completed when the buffer is flushed and the interrupt has been generated.	interrupt	

5.2.1 Example 1 - F.UHU.4.0 OHCI AHB READ Split Response

```
//assertions to check AHB Split on crossing 256-bit word boundary (corresponds to change in address bit 5);
sequence  ohci_cross_seq;
    (`AHB0.haddr[5] !=  $past (`AHB0.haddr[5]);
endsequence

read_ohci_split_as :  assert property (@(posedge pclk)
            $rose(`AHB0.hgrant) ##1 ((`AHB0.htrans[1] == 1) and !`AHB0.hwrite)
            ##[0:7]  ohci_cross_seq  |=>
            ##[0:$] `AHB0.hreq ##[0:2] (`AHB0.hresp == 2'b11)) check_assert[0][6]++;
            else check_assert[1][6]++;

read_ohci_split_cv : cover  property (@(posedge pclk)
            $rose(`AHB0.hgrant) ##1 ((`AHB0.htrans[1] == 1) and !`AHB0.hwrite)
             ##[0:7]  ohci_cross_seq
             ##[0:$] `AHB0.hreq ##[0:2] (`AHB0.hresp == 2'b11)) check_assert[2][6]++;
```

For this example a sequence is defined for crossing a 256-bit AHB address boundary which corresponds to a change in AHB address bit 5. Then there are two properties - *assert* and *cover*

which are checked throughout the simulation. With this design the AHB *hgrant* signal always asserts 1 cycle before the start of the AHB READ or WRITE transaction.

The *assert* property is used as a checker to ensure that every occurrence of the *antecedent* (the sequence before the "|=>" implication operator) is followed by the sequence after the implication operator. Here the *antecedent* corresponds to finding such a 256-bit address crossing within 8 cycles of identifying an AHB read response (we expect bursts of length 1 to 8 before a split). The assert property then checks that an AHB Split response always follows (*`AHB0.hresp == 2'b11*) the antecedent and will give a message to the simulator log file in the case of a failure.

The *cover* property is used to monitor the number of times the complete sequence has occurred in the simulation. The coverage is reported in the log file at the end of the simulation e.g. the simulator coverage report for *test_contention* is shown below.

```
"hdl/top.v", 161: top.write_write_cv, 1526844 attempts, 7252 match, 0 vacuous match
"hdl/top.v", 162: top.write_read_cv, 1526844 attempts, 7253 match, 0 vacuous match
"hdl/top.v", 163: top.read_write_cv, 1526844 attempts, 6917 match, 0 vacuous match
"hdl/top.v", 164: top.read_read_cv, 1526844 attempts, 7113 match, 0 vacuous match
"hdl/top.v", 184: top.write_ohci_split_cv, 1526844 attempts, 3747 match, 0 vacuous match
"hdl/top.v", 186: top.write_ehci_split_cv, 1526844 attempts, 3285 match, 0 vacuous match
"hdl/top.v", 189: top.read_ohci_split_cv, 1526844 attempts, 3726 match, 0 vacuous match
"hdl/top.v", 191: top.read_ehci_split_cv, 1526844 attempts, 3468 match, 0 vacuous match
"tests/test_contention.sv", 146: top.t1.arb_cv, 1526844 attempts, 10 match, 0 vacuous match
"tests/test_contention.sv", 154: top.t1.arb_ehci_ohci_ehci_cv, 1526844 attempts, 467 match, 0 vacuous match
```

5.2.2 Example 2 - F UHU.5 for EHCI Core Interrupt Coherency

A second example of the use of assertions for functional coverage is the testing of interrupt coherency for the EHCI core interrupt to satisfy the verification requirements of Table 3. Here again there are both *assert* and *cover* properties. These properties are checked if the signals *assert_ehci_int* and *cover_ehci_int* are high and the DUT has not yet asserted its interrupt output *uhu_icu_irq*. The signals *assert_ehci_int* and *cover_ehci_int* are asserted in the testbench when an EHCI core interrupt occurs during an AHB WRITE. If these conditions are satisfied then the sequence *ehci_int_seq* is checked to see whether it has occurred.

ehci_int_seq concisely describes the verification requirements that following a core interrupt during an AHB WRITE the following sequence must occur

- an AHB Split i.e. (*`AHB0.hresp == 2'b11*),
- followed by a DRAM write i.e. a buffer flush (4 consecutive assertions of *uhu_diu_wvalid* according to the DRAM write protocol of Figure 7) ,
- followed by the DUT asserting its interrupt output *uhu_icu_irq.*

and that

- no AHB grant occurs until *uhu_icu_irq* assertion

i.e. no further AHB transactions are allowed until the buffer has been flushed and the interrupt generated.

```
//check if external interrupt event and AHB write in progress that an AHB split occurs
ehci_int_as : assert property (@(posedge top.pclk)
        (assert_ehci_int && !top.intif.uhu_icu_irq) |->
        ehci_int_seq)   test_check_assert[0][0]++; else test_check_assert[1][0]++;

ehci_int_cv : cover property (@(posedge top.pclk)
        (cover_ehci_int && !top.intif.uhu_icu_irq)
        ##0 ehci_int_seq)  test_check_assert[2][0]++;

sequence  ehci_int_seq;
   //AHB split followed by DIU write followed by uhu_icu_irq assertion
   //AND no AHB grant occurs until uhu_icu_irq assertion
        ((`AHB0.hresp == 2'b11)
        ##[1:$] top.dwrif.dau_cb.uhu_diu_wvalid ##1 top.dwrif.dau_cb.uhu_diu_wvalid [*3]
         ##1 (!top.dwrif.dau_cb.uhu_diu_wvalid throughout  !top.intif.uhu_icu_irq [*0:$])
        ##1 top.intif.uhu_icu_irq)
        and
         ((`AHB0.hresp == 2'b11)
        ##1 (!`AHB0.hgrant throughout  !top.intif.uhu_icu_irq [*0:$])
        ##[1:$] top.intif.uhu_icu_irq
        ##[0:$] `AHB0.hgrant);
endsequence
```

5.3 Reporting the Results

In addition to reporting assertion failures and coverage information to log files the Synopsys Unified Report Generator can collate the functional coverage results across multiple testcases and generate *html* reports. These reports can also include code-coverage results as well as functional coverage. An example is shown in Figure 8.

The reader will have noticed the presence of action blocks on both the a*ssert* and *cover* statements of Section 5.2. These capture the number of times an assertion has succeeded or failed and the number of times a coverage point has been hit. At the end of the test ERROR messages (and corresponding INFO messages) are output to indicate if an assertion failed or if the assertion and coverage points were never exercised.

```
INFO COVERAGE: Global Coverage      #0 exercised      7252 times
INFO COVERAGE: Global Coverage      #1 exercised      7253 times
INFO COVERAGE: Global Coverage      #2 exercised      6917 times
INFO COVERAGE: Global Coverage      #3 exercised      7113 times
INFO COVERAGE: Global Assertion     #4 exercised   1526839 times
INFO COVERAGE: Global Assertion     #4 no failures
INFO COVERAGE: Global Coverage      #4 exercised      3747 times
INFO COVERAGE: Global Assertion     #5 exercised   1526839 times
INFO COVERAGE: Global Assertion     #5 no failures
INFO COVERAGE: Global Coverage      #5 exercised      3285 times
```

Category 0

Assertions	Attempts	Matches	Vacuous Matches	Incomplete
top.read_ehci_split_as	1526844	3468	0	0
top.read_ohci_split_as	1526844	3726	0	0
top.write_ehci_split_as	1526844	3285	0	0
top.write_ohci_split_as	1526844	3747	0	0
top.t1.arb_ehci_ohci_ehci_as	1526844	467	0	0

Category 0

Cover properties	Attempts	Matches	Vacuous Matches	Incomplete
top.read_ehci_split_cv	1526844	3468	0	0
top.read_ohci_split_cv	1526844	3726	0	0
top.read_read_cv	1526844	7113	0	1
top.read_write_cv	1526844	6917	0	197
top.write_ehci_split_cv	1526844	3285	0	0
top.write_ohci_split_cv	1526844	3747	0	0
top.write_read_cv	1526844	7253	0	0
top.write_write_cv	1526844	7252	0	1
top.t1.arb_cv	1526844	10	0	0
top.t1.arb_ehci_ohci_ehci_cv	1526844	467	0	0

Figure 8 – Unified Report Generator HTML Coverage Reporting

This reporting mechanism explicitly adds the notion of *required* assertions for a particular testcase. This method of reporting is used by S3 verification teams so that all functional coverage can be reported to the simulation log file with standard INFO and ERROR messages without the need for generating separate coverage databases. S3's NanoFlow design environment contains CGI-based webscripts which are used to parse the log files to generate verification status information for all the testcases relating to a design. The status of an entire chip verification regression, sometimes numbering in the thousands of testcases, can then easily be monitored from a single *html* web-page. Sample S3 NanoFlow web-script screenshots are shown in Figure 9. The percentage functional coverage field facilitates progress tracking and selecting the most efficient testcases during the test suite development.

NANOFLOW Verification Summary : Block uhu

Root directory : /proj/nanoflow/users/paulf/WORK_FCOV/data/s3_fcov_ref1

Block verification status for : uhu

Test Name	Memory	CPU Time	Cycles (pclk)	Real Time	Simulation Messages			Coverage		Simulation Log File	Date	OK?	Required?
					Info	Warning	Error	Code (%)	Functional (%)				
test_contention	48.4M	865.2s	5979737	00:14:27	0	2	0	N/A	100%	test_contention.sim.log	Mar 1, 2006 at 10:12:34	PASS	FALSE
test_coherent_ohci	45.0M	0.7s	1816	00:00:01	0	2	0	N/A	100%	test_coherent_ohci.sim.log	Mar 1, 2006 at 10:10:58	PASS	FALSE
test_coherent_ehci	45.7M	0.6s	1230	00:00:00	0	2	0	N/A	100%	test_coherent_ehci.sim.log	Mar 1, 2006 at 10:24:44	PASS	FALSE
test_random	45.0M	743.2s	3982426	00:12:30	0	2	0	N/A	100%	test_random.sim.log	Mar 1, 2006 at 10:46:00	PASS	FALSE
test_interrupt	44.7M	5.2s	27047	00:00:06	0	2	0	N/A	100%	test_interrupt.sim.log	Mar 1, 2006 at 10:33:56	PASS	FALSE

Global verification status: PASS

Block	Total Tests	Golden Tests Passed	Other Tests Passed	Global P/F?	OK?
pcu	41	41	0	Yes	PASS
tim	1	1	0	Yes	PASS
pss	1	1	0	Yes	PASS
uhu	5	5	0	Yes	PASS
icu	10	0	10	Yes	PASS

Figure 9 – S3 NanoFlow CGI Webscript Verification Regression Reporting

6.0 Discussion of Results

Looking back at Table 2 it is clear that almost all the verification requirements are hit by the *test_contention* general random test case. Only *test_coherent_e/ohci* and *test_interrupt* are required to hit the remaining requirements and in fact these testcases represent only small modifications to the basic testcase. This leads to an economy of effort in testcase development i.e. once the basic verification environment has been constructed writing extra testcases is only a small extra effort. In fact most of the effort in testcase generation goes into writing assertions and ensuring they are correct.

Assertions are extremely useful for checking that certain stimuli conditions have occurred e.g. the *ehci_int_write_phase0..7_cv* assertions are used to check that an EHCI core interrupt has been generated during each possible address cycle of an 8 word AHB Split transaction. Automatic reporting of coverage can be important when late changes to a design mean some testcases no longer exercise some of the coverage points. Reporting ERROR messages when required assertions have not been exercised ensures automatic checking of testcases.

6.1 Conclusions and Recommendations

At the start of this investigation a number of questions were posed. In this section we try to answer these questions and outline any advantages and disadvantages of using a SystemVerilog constrained random verification approach.

1. Is there a large ramp up required for our verification engineers in switching to SystemVerilog?
   - We found it was possible to ramp up SystemVerilog competence in less than 2 weeks without any prior knowledge of constrained random verification provided suitable training with examples are used e.g. the Synopsys SystemVerilog QuickStart training [3]. Previous exposure to OO techniques e.g. C++ or SystemC is definitely a help.

2. If our existing verification flow is resulting in first time right silicon is there a significant advantage in switching to a constrained random flow (e.g. time-saving)?
   - We found that the effort was approximately the same for the two methodologies for the small set of requirements we were targeting. But as the number of scenarios that can be hit with the same verification environment increases then effort should start to reduce with a SystemVerilog environment.

3. Are existing CAD tools mature enough to support the SystemVerilog features we require?
   - Synopsys VCS was easily able to handle this mixed SystemVerilog-VHDL verification without any issues.

4. Is co-simulation with existing VHDL designs an issue?
   - Synopsys VCS supports mixed language designs without any issues.

5. How important to constrained random verification is having a well defined verification methodology ?
   - A well defined methodology is extremely important. The testbench structure of Figure 6 is quite distributed so it should follow a good template. For small designs the RVM-lite methodology is sufficient. For larger designs the full RVM using the RVM-base class standardizes the testbench environment using the best practices incorporated in the VMM base classes based on many years of experience of Vera.

6.2 Advantages and Disadvantages of SystemVerilog versus a VHDL Directed Approach

Advantages of SystemVerilog approach:

- One single reusable test environment.
- By default all tests are completely randomized.
- All requirements can be hit by constraining the environment. The VHDL environment needed approximately 20 procedural directed test cases to hit the requirements fully.
- Automates cross-referencing requirements against testcases through use of assertions. Can easily track progress to 100% functional coverage.
- There is a definite advantage in using completely randomized tests which may test scenarios not thought off.
- Most checkers will be included in the environment for almost all tests unlike in the directed test case where the checker may only be active for that particular directed test.

This has the advantage that wider coverage beyond the precise definition of the verification requirements may be obtained.

Disadvantages:

- Must be aware that environment development takes longer but testcase development effort will be less.
- If directed tests are also to be written, as opposed to constrained directed tests, then the environment must be designed at the outset to support these.
- It is extremely important to review requirements in detail early on in environment development to ensure that the verification environment is sufficient to allow all the verification requirements to be exercised otherwise additional environments may be required which may cost a lot of time to develop.

It is important to emphasis the S3 DUT audit methodology outlined in Section 3.1 as being critical for good verification results in both directed and constrained random verification approaches. The directed VHDL approach is completely dependent on the verification planning as only coverage of the listed requirements is achieved. Verification planning is still important for constrained random for measuring coverage results but constrained random has the bonus of potentially hitting similar corner cases for free.

7.0 Acknowledgements

The authors would like to thank Douglas Fisher, Rob van Osch, Yassine Eben Amine and Roger Parkinson at Synopsys for organizing this investigation of SystemVerilog supported by excellent training and technical support. A large measure of thanks are also due to Dave Buckley and Tony Smith at S3 for work in developing the testbench.

8.0 References

[1] “Silicon Penguins: Using and verifying an open source processor in a 100% first time right commercial ASIC”, Declan Staunton, Paul Furlong, DVCON 2006, San Jose.
[2] “Verification Methodology Manual for SystemVerilog”, Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale, Springer 2005.
[3] “Introduction to SystemVerilog for Testbench – One Day Quick Start”, Synopsys Inc., 2005.