Synopsys Users Group

Tough Bugs Vs Smart Tools - L2/L3 Cache Verification using System Verilog, UVM and Verdi Transaction Debugging

Vibarajan Viswanathan - *Centaur Technology, Austin, TX*

Doug Reed - *Centaur Technology, Austin, TX*

www.centtech.com

Juliet Runhaar - *Synopsys Inc, Austin, TX*

Jun Zhao - *Synopsys Inc, Mountain View, CA*

www.synopsys.com

ABSTRACT

Verdi's Transaction Debug helps record, visualize, analyze and debug transactions at higher level of abstractions. With recorders integrated inside the Verdi UVM libraries and convenient Verdi APIs, UVM sequences and user-level transactions are recorded automatically and available for debug. In this paper, we discuss some of L2/L3 SV/UVM test scenarios and how Verdi Transaction Debug was used to model and debug out-of-order transactions. For example, in a cache miss scenario, L3 forwards the request to external bus where four responses are fetched out of order. Using Verdi APIs, Centaur modeled the request as a Parent Transaction and the four Responses as Children Transactions. Verdi Transaction Debug was then used to quickly find the linked parent (Request) - children (Responses) transactions, which were spread across hundreds of clock cycles, by highlighting the related transactions. This allowed for quickly debugging failing tests due to missing responses or unfinished transactions.

Table of Contents

Table of Figures

1. Introduction

In Processor designs, L1, L2, L3 Caches are prone to corner case bugs pretty late in the design cycle. Even if there are ways to work-around these bugs, it affects the performance of the CPU. Also, fixing these bugs late in the design cycle affects the ability to do better timing closure in order to boost the frequency of operation. At Centuar, SV/UVM constrained random based Unit level benches helped us catch many bugs early on including corner case bugs that were harder to reach in system level. Many of UVM and SystemVerilog constrained random techniques were used to craft the testbench features such as interesting Address generation, Semaphore Controlled Scenarios, Cache-Line Miss Response delay modeling, etc.

Verdi's Transaction Debug platform helps record, visualize, analyze and debug the transactions at high level of abstractions. It is integrated inside Verdi's provided UVM library to record the UVM sequences automatically. Convenient APIs are also provided to record user-level transactions. In this paper, we are going to discuss some of the L2/L3 SV/UVM test scenarios that helped us expose RTL bugs quickly and how Verdi Transaction Debug flow allowed us to identify bugs more easily than traditional debug methods. For example, in a cache miss scenario, L3 forwards the request to the external bus where the response can be fetched from other coherent agents or from the external Memory. The response is broken into multiple objects that come in out of order. In the Verdi debug, the request was modeled as parent transaction and the four response objects were modeled as children. We will also demonstrate how we quickly found these linked objects, espescially when they were spread apart across hundreds of clock cycles, by using Verdi Transaction Debug's powerful graphic features, which highlight children and parent objects when either object is selected.

2. L1,L2,L3 Cache Hierarchy

CPU designs employ multi level caches in order to boost CPU performance. Figure 2-1 shows a typical structure of a cache hierarchy.

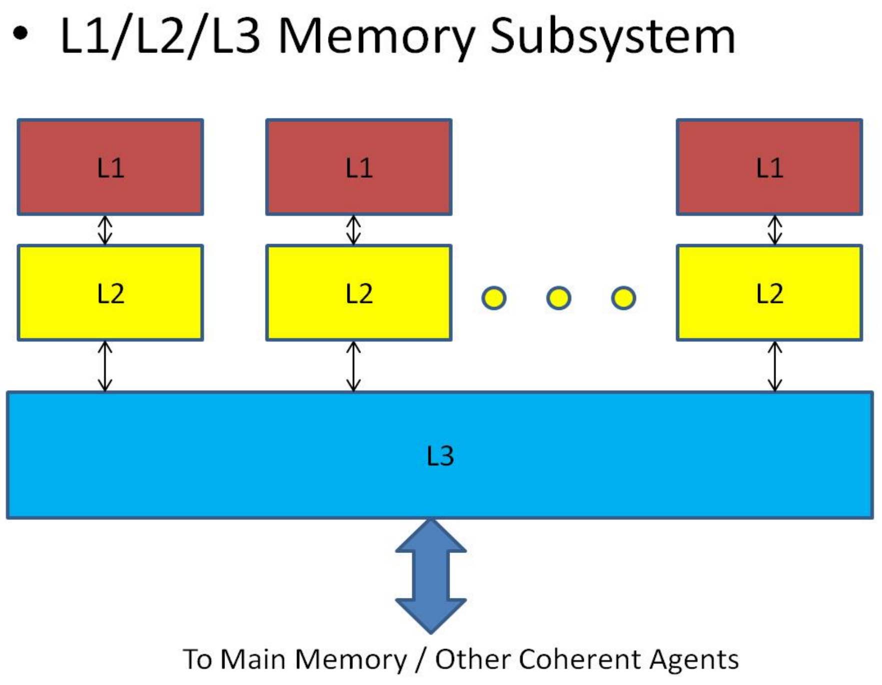

• L1/L2/L3 Memory Subsystem
L1
L1
L1
L2
L2
L2
L3
To Main Memory / Other Coherent Agents

Figure 2-1: L1/L2/L3 Memory Subsystem

2.1 Data Flow

L1, L2, L3 caches all have load/store, evict, and snoop ports. In the case of a L2 Cache-line hit scenario, the data is returned to L1. In the case of a L2 cache miss scenario, the request gets forwarded to L3. If L3 has the line, the data gets delivered to L1. If it is a miss in L3, the request gets forwarded to the coherent bus anticipating the data to come either from other coherent agents or from the main memory.

3. L2 Verification Challenges

Unanimous complexities of the L2/L3 cache, such as queues, concurrent state machines, and synchronizing/intruding events like snoops and evicts, are making L2/L3 prone to complex functional bugs. MESI state bugs, hangs, and deadlocks are examples of such complex bugs.

3.1 Load, Evict and Snoop Dependencies

In the case of L3 cache miss scenario, L3 forwards the Load to the bus. The final data gets returned either from other coherent agents or from the Main Memory. First, other coherent agents are snooped. If it is a miss, then the Load is routed to the final destination which is the Main Memory. A missed load request has multiple broken down responses, which are called Response, Data, Self-Snoop, and HitAssert. As the L3 is waiting for the data to come back, L3 issues the SelfSnoop back to L2 and L1, thereby, signaling the L2 and L1 to mark the MESI state status to either *before* or *after*.

For any other Loads or Snoops to the same address that come *after* the Self-Snoop, L2 must respond with the updated data and MESI. For any other Loads or Snoops to the same address that came *before* Self-Snoop, L2 must respond with the older data and MESI. So, in the *after* scenario, it is likely that the corresponding Data will come shortly after the Self-Snoop. In which case, the L2 will hold the response to those newer Loads or Snoops until the final Data is returned. Moreover, the

Self-Snoops and their corresponding Data can come out of order, which introduces additional complexity.

3.2 Hang, Timeouts, Deadlocks

In the *after* scenario, the Self-Snoops and the Data responses of a Load could come out of order. In situations where Data arrives before Self-Snoop, L1 will wait for the SelfSnoop.

In the *before* scenario, L2 does not have to wait for the Self-Snoop and responds immediately with the older status to any other Loads or Snoops to the same address - that came before Self-Snoop. This feature can cause potential deadlock, especially when the L2 pipes are full and are not able to accept the Self-Snoop and the corresponding pending Load can actually help to free up the L2 pipes. At the least, this feature can adversely increase the Latency thereby degrading the overall System Performance.

In contrast to this, sometimes an Evict or an Evicting-Snoop (snoop that disallows shared MESI state - L2 and L1 are forced to evict the line for this Snoop) could retard the L2 cache and prevent hangs or deadlocks. Such scenarios could easily hide a deadlock bug.

3.3 Coherency Verification

It is important that each cache agent reflects the accurate MESI state for each cacheline that it holds in order to keep track of the latest Data. Bugs can lurk on the MESI state transition logic but these bugs could be exposed with colliding Loads, Snoops, and Evicts to the same address. For example, an Evict request to L2 or an LRU (Least Recently Used) Eviction Mechanisms, unless duly completed amidst the cloud of Loads and Snoops that are pending or are occurring at the same time, could mess up the MESI state.

4. L2 System Verilog, UVM Environment

The L2/L3 cache complexities such as queues, concurrent state machines and synchronizing/intruding events like snoops and evicts are making L2/L3 prone to complex functional bugs, MESI state bugs, hangs and deadlocks. UVM/SystemVerilog constrained random based unit level benches used with SystemVerilog Assertions and debugging technologies are the perfect ammunition to deal with them.

- L2 UVM/SV Test Bench

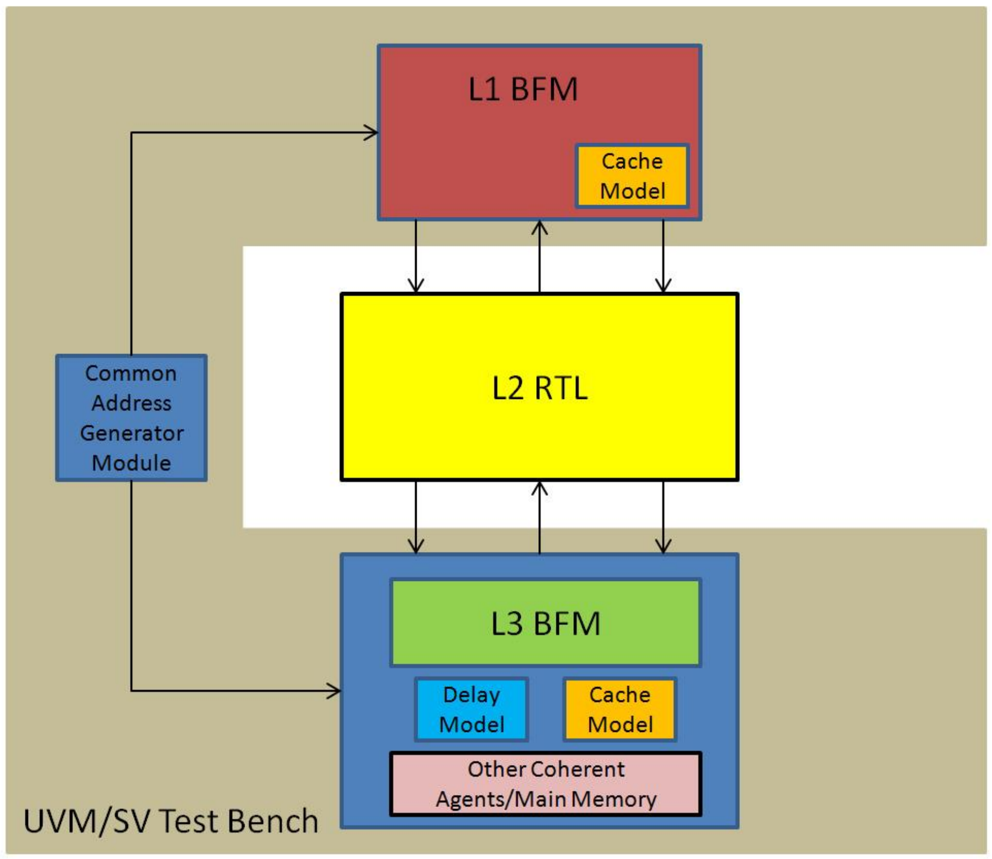


Figure 4-1: L2 UVM/SV/ Test Bench

UVM Bus Functional Models were created for each L2 interfaces. While it took time in pruning those constraints, they were very helpful to quickly generate lots of legal sequences later on. L1, L3 SystemVerilog cache models have helped to track the MESI states accurately. UVM virtual sequences were the key features that enabled to orchestrate stimulus across L1 and L3 interfaces. These are basically the system level scenarios that exposed deadlocks and hangs in the design. Semaphore techniques were extensively used to model arbitration and resource sharing in UVM stimulus generation.

4.1 Unique ID generation using Static Variables

Load, Snoop, and Evict Requests have used individual ID pool. In the sequence items, static variables were used to track ID values from the pool in order to generate Loads, Stores, Snoops, and Evicts with unique IDs. These IDs could not be used until the request was fully completed and released on the driver. This feature used with back to back requests generation to L2 exposed bugs on the outstanding request counts in the L2 Queues/Pipes - L2 was taking few clocks more to release the queue after it had completed the corresponding request which was incorrect.

4.2 Address Generation

A singleton address generate model was created in order to generate interesting colliding addresses across L1 and L3 Interfaces. It has the ability to generate addresses that are random, incrementing, decrementing, and repeating. The constraint class has the configurability to increment, decrement and repeat TAG, CL (CacheLine), and SET Addresses. This helped to walk through a range of addresses. Interesting enough, the tests with TAG, SET, or CL addresses showed different kinds of bugs.

4.3 Latency/Delay Modeling

In order to model out of order responses, SystemVerilog class constraint models were developed to generate random response delays such as C1 Biased, C2 Biased, and C3 Biased. C1 Biased are amount of responses that occur with SHORT delays, C2 Biased are amount of responses that come with MEDIUM delays, and C3 Biased amount of responses that come with LONG delays.

```
class response_delay_biasing;
  rand int index;
  rand int delay [3];
  int c1 = 7;
  int c2 = 3;
  int c3 = 1;

  constraint distribution_c { index dist { 0 := c1, 1  := c2, 2 := c3 }; }
  //constraint delay_c { delay[0] == 70 && delay[1] == 40 && delay[2] == 10 ; }
  constraint delay_c { delay[0] < 20 && delay[1] < 100 && delay[2] < 500 &&
                       delay[0] > 5 && delay[1] > 5 && delay[2] > 5; }

  constraint order_c { solve index before delay ; }

  function new(int i_C1 = 1, int i_C2 = 1, int i_C3 = 1);
      this.c1 = i_C1;
      this.c2 = i_C2;
      this.c3 = i_C3;
  endfunction

  ...........
  ...........
endclass
```

Figure 4-2 shows the delay value distribution of four responses of a Load to L3 in a cache miss scenario. The delays were generated using the SV constraint model. The responses are highlighted as the *Bus_dataresponse_delay*, *hit_response_delay*, *response_delay*, and *self_snooperresponse_delay*. Note that the *Response* component of response should come first and after that the remaining three can come out of order.

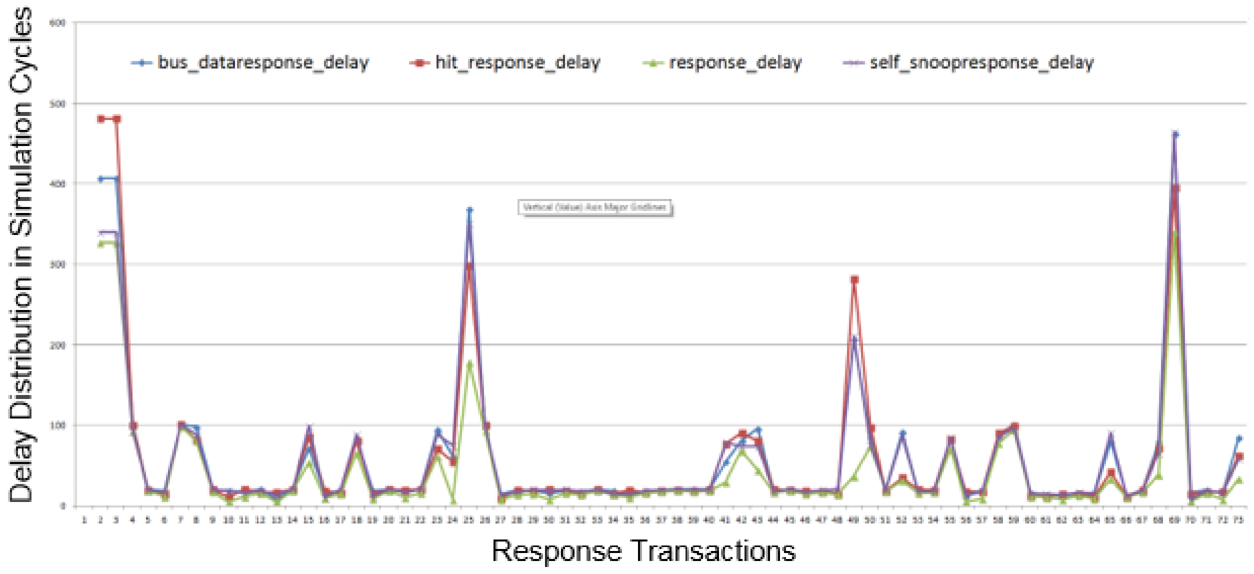


Figure 4-2: Delay value distribution of four responses of a load of L3 in a cache miss scenario

This is where Verdi Transaction debug was very useful to both developments of this delay model and to debug the RTL behavior. Using Verdi APIs, we modeled the request as a Parent Transaction and the four Responses as Child Transactions. Verdi Transaction Debug was then used to quickly find the linked parent (Request) - children (Responses) transactions, which were spread across hundreds of clock cycles, by highlighting the related transactions. This allowed for quickly debugging failing tests due to missing responses or unfinished transactions.

Without Verdi Transaction Debug one would have to first architect a messaging style to log the important attributes of a transaction and its relation to other transactions, then search the simulation log file and postprocess the extracted information, and finally look for the transaction signals and match them to signals of its parent/child/siblings transactions in the waveform viewer . These steps mentioned above require some planning, knowledge of transaction relations, efficient text search mechanisim, and accurate analysis of data obtained from multiple sources such as log files and waveform viewer, hence are time consuming, tedious and prone to human error. Verdi Transaction Debug technology automates this process and offers an accurate and efficient method of debugging at an abstract level.

5. Debug

With the presence of three levels of caches, in order to maintain cache coherency, a simple read/write request from one CPU to a memory location translates into multiple transactions between the CPU's different cache layers, then out to the external bus, and to other CPUs sharing the bus and the memory. These transactions in turn toggle hundreds of hardware signals resulting in a huge quantity of data to analyze and debug at the signal level. This raises the need for a more sophisticated debug tool that is able to capture, display, and analyze abstract data such as transactions.

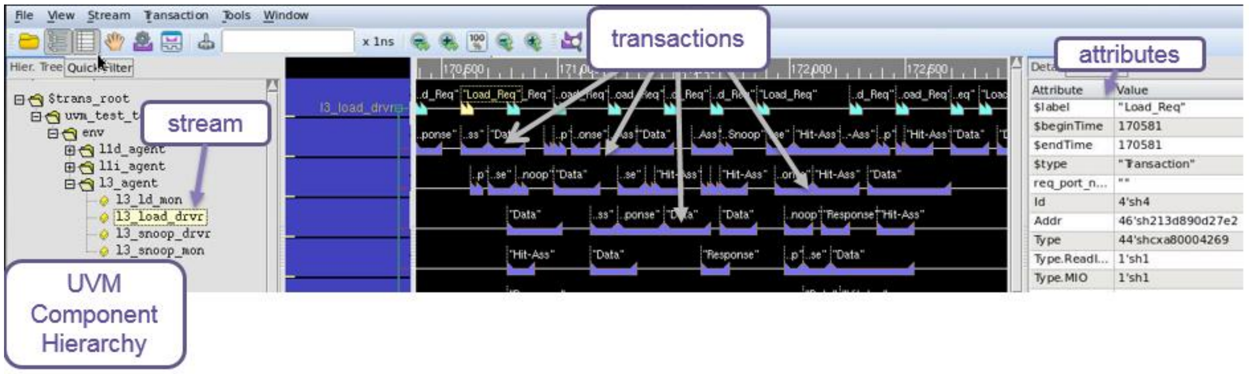
5.1 Verdi's Transaction Debug Platform

Verdi's Transaction Debug Platform is designed to debug at a high abstraction level. The FSDB

(Fast Signal Database), originally designed for debugging HDL level signals, has been extended to include transactions.

A *Transaction* is a high level abstraction-concept which has a start time, an end time, and a list of attributes. An attribute is a characteristic or property of a transaction.

A temporal sequence of transactions with a common origin forms a *Stream*. In Figure 5-1, all *transactions* originating from the same UVM component (in this case a uvm_driver) are grouped into one stream. *Streams* are then organized into the UVM component hierarchy allowing for an easy way to locate a specific stream.



Attribute	Value
$label	"Load_Req"
$beginTime	170581
$endTime	170581
$type	"Transaction"
req_port_n...	""
Id	4'sh4
Addr	46'sh213d890d27e2
Type	44'shcxa80004269
Type.ReadI...	1'sh1
Type.MIO	1'sh1

Figure 5-1: Verdi's view of Transaction Browser, Hierarchy Tree, and Attribute Detail Table

If transactions in a stream are interlinked, then they form a relation. The relations are categorized as parent-child, siblings, or predecessor-successor in the FSDB.

The parent-child relation is the protocol stacking/layering relationship and models the parent/child relationship of UVM sequences (transactions).

The predecessor-successor relation is the temporal ordering/triggering relationship and models a transaction-trace across different test units (e.g. UVM components).

A sibling relation is a relationship among groups of children-transactions that belong to the same parent- transaction.

Later in this paper, we will demonstrate how we used Verdi's Transaction Debug to trace from one transaction to another and find RTL and testbench bugs.

5.2 Recording and Linking Non-Sequencer Transactions

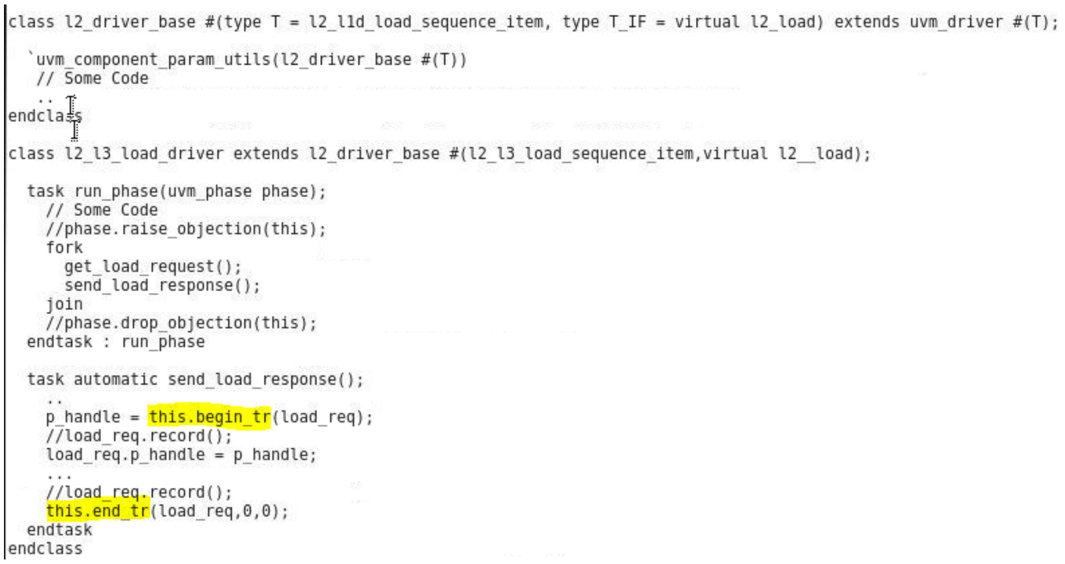
The transaction based FSDB is specifically optimized for efficient recording and fast retrieval of the data. The Verdi platform provides an automatic recording mechanism for recording UVM messages and UVM-sequencer transactions into FSDB. In Centaur's multi core design, a read transaction that is originated by a cpu, could travel through multiple interfaces, from L1, to L2, to L3, to the bus, and to the memory controller before it completes the read task. Hence, visibility into non-sequencer components such as drivers, which drive the transactions into multiple interfaces, and monitors, which monitor the transactions received from multiple interfaces, is crucial for debugging.

In order to record transactions from UVM drivers and monitors, Verdi implements hooks to the UVM APIs *begin_tr(), begin_child_tr,* and *end_tr()*. A *begin_tr()* or *begin_child_tr()* API call must be added in pair with the *end_tr()* API call in the driver and monitors classes.

// Add at beginning of transaction for transaction record
void'(this.begin_tr(tr));

// Add at transaction end for transaction record
this.end_tr(tr);

Figure 5-2 shows the L2 driver code where *begin_tr()* and *end_tr()* were added. *begin_tr(load_req)* first obtains a transaction handle for the *load_req* object and assigns it to the variable *p_handle*. A copy of the handle is saved to the *load_req* before this object is issued to L3. Then SystemVerilog *fork - join_none* constructs were used to generate all four response processors simultaneously. The load request transaction is then completed with the *end_tr(load_req,0,0)* API call.

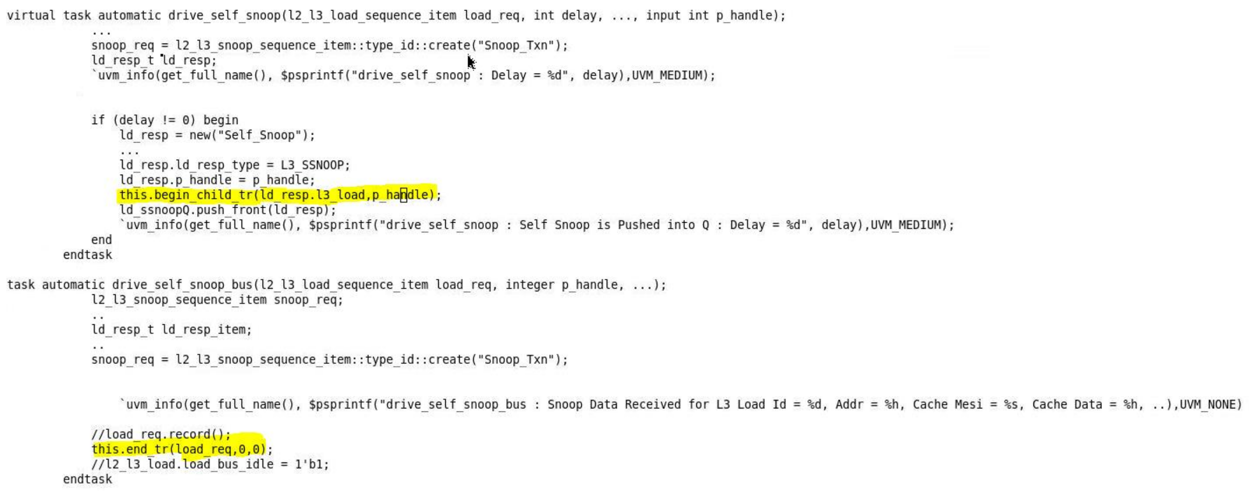


```
class l2_driver_base #(type T = l2_l1d_load_sequence_item, type T_IF = virtual l2_load) extends uvm_driver #(T);

  `uvm_component_param_utils(l2_driver_base #(T))
   // Some Code
   ..
endclass

class l2_l3_load_driver extends l2_driver_base #(l2_l3_load_sequence_item,virtual l2__load);

  task run_phase(uvm_phase phase);
    // Some Code
    //phase.raise_objection(this);
    fork
      get_load_request();
      send_load_response();
    join
    //phase.drop_objection(this);
  endtask : run_phase

  task automatic send_load_response();
    ..
    p_handle = this.begin_tr(load_req);
    //load_req.record();
    load_req.p_handle = p_handle;
    ...
    //load_req.record();
    this.end_tr(load_req,0,0);
  endtask
endclass
```

Figure 5-2: L2 driver code with begin_tr and end_tr

As described earlier in this paper an L2 load request to L3 that is a miss results in four response: Response, Data, Self-Snoop, and HitAssert, which are returned to L2 in any order. Using the *begin_child_tr/end_tr* API, we linked all four responses (child) transaction objects to their corresponding request (parent) transaction object.

Figure 5-3, demonstrates how we created the parent-child relationship between the Load request and the Self-Snoop response. The *begin_child_tr()* API is called in the Self-Snoop task to start a new transaction. The *begin_child_tr()* receives the *p_handle*, handle to the parent request transaction, as an argument, and associates itself to the *p_handle* as a child.



```
virtual task automatic drive_self_snoop(l2_l3_load_sequence_item load_req, int delay, ..., input int p_handle);
            ...
            snoop_req = l2_l3_snoop_sequence_item::type_id::create("Snoop_Txn");
            ld_resp_t ld_resp;
            `uvm_info(get_full_name(), $psprintf("drive_self_snoop : Delay = %d", delay),UVM_MEDIUM);


            if (delay != 0) begin
                ld_resp = new("Self_Snoop");
                ...
                ld_resp.ld_resp_type = L3_SSNOOP;
                ld_resp.p_handle = p_handle;
                this.begin_child_tr(ld_resp.l3_load,p_handle);
                ld_ssnoopQ.push_front(ld_resp);
                `uvm_info(get_full_name(), $psprintf("drive_self_snoop : Self Snoop is Pushed into Q : Delay = %d", delay),UVM_MEDIUM);
            end
        endtask

task automatic drive_self_snoop_bus(l2_l3_load_sequence_item load_req, integer p_handle, ...);
            l2_l3_snoop_sequence_item snoop_req;
            ..
            ld_resp_t ld_resp_item;
            ..
            snoop_req = l2_l3_snoop_sequence_item::type_id::create("Snoop_Txn");


                `uvm_info(get_full_name(), $psprintf("drive_self_snoop_bus : Snoop Data Received for L3 Load Id = %d, Addr = %h, Cache Mesi = %s, Cache Data = %h, ..),UVM_NONE)

            //load_req.record();
            this.end_tr(load_req,0,0);
            //l2_l3_load.load_bus_idle = 1'b1;
        endtask
```

Figure 5-3: L2 Self-Snoop code with begin_child_tr and end_tr

5.3 Recording Transactions into FSDB

To record transactions into FSDB, a set of compilation and simulation switches must be passed to the simulator.

The UVM library shipped with VCS contains the Verdi Transaction *Recorder* and *Catcher* classes. Hence, the only additional options required during compilation are *-ntb_opt uvm* and *-debug_access.* For example:

%vcs -sverilog -debug_access -ntb_opt uvm -l mybuild.log -f mydesign.f

During simulation the required opitons are:

+UVM_VERDI_TRACE
+UVM_TR_RECORD

Here is an example of how to enable ransaction recording with VCS simulator:

% simv + UVM_VERDI_TRACE +UVM_TR_RECORD [other runtime options]

In order to record the transactions into FSDB using other simulators, the user must explicitly specify the Verdi UVM library as shown in the example below:

%vlog +incdir+$VERDI_HOME/etc/uvm $VERDI_HOME/etc/uvm/uvm.sv
$VERDI_HOME/etc/uvm/dpi/uvm_dpi.cc
+incdir+$VERDI_HOME/etc/uvm/verdi
$VERDI_HOME/etc/uvm/verdi/uvm_custom_install_verdi_recorder.sv <compile_options>

The same runtime options which are used with VCS are also used with the other simulators.

The simulator steps described above are used for recording both the automated sequencer component transactions and the non-sequencer component transactions, such as drivers and montiors, that are manully enabled by users via Verdi APIs.

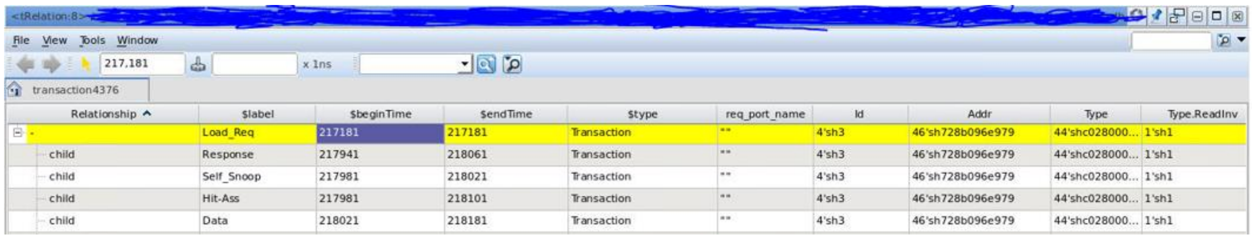
5.4 Debug Scenario 1. Missing Transactions

In a cache miss scenario, L3 forwards a request to the external bus where four responses are fetched out of order. Using the Verdi APIs, we modeled (details provided in section 5.2) the Request from L2 as a parent transaction and the four responses (Response, Data, Self-Snoop, and Hit Assert) received from the external bus as children transactions. Depending on the constraint settings selected for the delays inserted between the response transactions, these transactions were spread across hundreds of clock cycles making it extremely difficult to identify the missing transaction.

Relying on the traditional debug method, for example searching simulation log files, given one knows exactly what to search for, and analyzing the extracted data in text files to match the parent and child transactions is tedious and time consuming.

The Verdi Transaction-Relation Navigator displays the children transactions of a parent transaction in a spreadsheet-like view, with transactions listed in a relation-hierarchy format vertically and attributes for each transaction listed under tabs horizontally. The navigator also provides a filtering and searching mechanism which allows the users to customize the view and to quickly search for transactions.

Figure 5-4 shows a listing of all 'Load_Req' transaction's children and their attributes. In this view we could quickly verify that all the children-transaction IDs and addresses were consistent



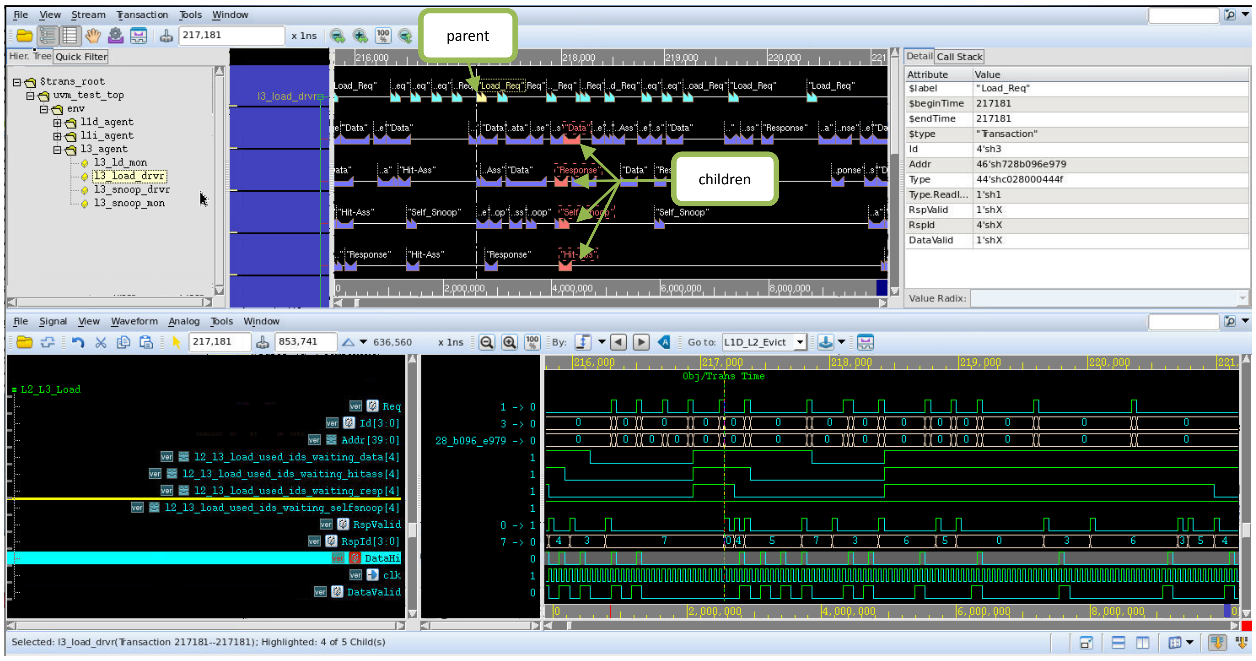
Relationship	$label	$beginTime	$endTime	$type	req_port_name	Id	Addr	Type	Type.ReadInv
-	Load_Req	217181	217181	Transaction	""	4'sh3	46'sh728b096e979	44'shc028000...	1'sh1
child	Response	217941	218061	Transaction	""	4'sh3	46'sh728b096e979	44'shc028000...	1'sh1
child	Self_Snoop	217981	218021	Transaction	""	4'sh3	46'sh728b096e979	44'shc028000...	1'sh1
child	Hit-Ass	217981	218101	Transaction	""	4'sh3	46'sh728b096e979	44'shc028000...	1'sh1
child	Data	218021	218181	Transaction	""	4'sh3	46'sh728b096e979	44'shc028000...	1'sh1

Figure 5-4: A listing of 'Load_Req' transaction's children in Verdi Transaction Navigator

The Relation Navigator cockpit was very helpful in tracing the relations and finding the missing transactions.

The relations between the transactions can also be viewed in the Transaction Browser. The Transaction Browser displays transactions in a simulation in a waveform-like view and provides the capability to unfold overlapping transactions and display them vertically. In the unfold view when a transaction is selected Verdi highlights all other transactions that are linked to it providing a quick and easy way to find parent-child or sibling relations.

In Figure 5-5, all children transactions (Data, Response, Self_Snoop, Hit-Assert) of selected parent transaction (Load_Req) are highlighed providing for an easy and quick way to find interlinked transactions.

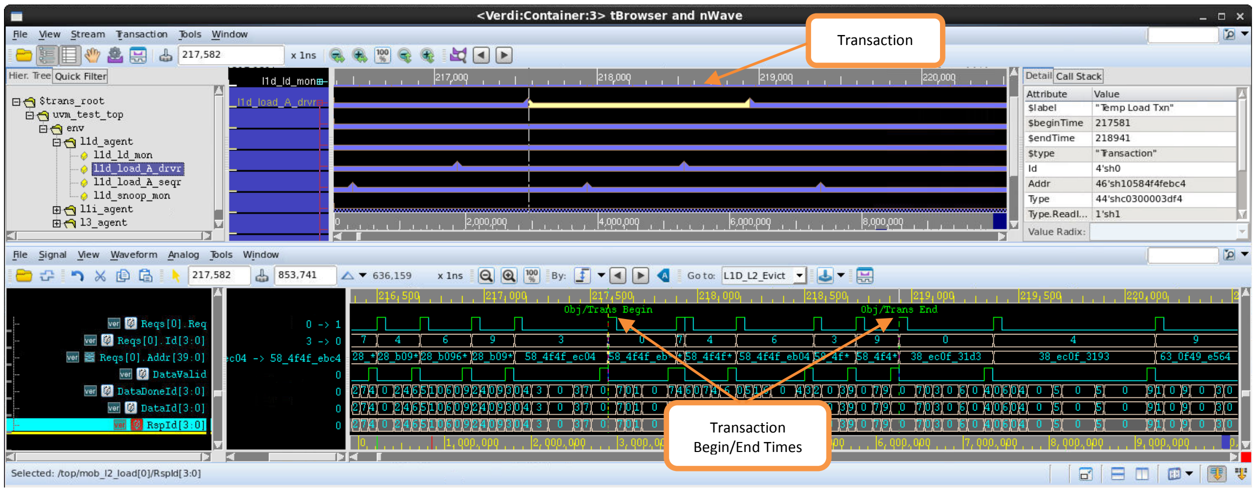

parent
children
Selected: l3_load_drvr(Transaction 217181--217181); Highlighted: 4 of 5 Child(s)

Attribute	Value
$label	"Load_Req"
$beginTime	217181
$endTime	217181
$type	"Transaction"
Id	4'sh3
Addr	46'sh728b096e979
Type	44'shc028000444f
Type.ReadI...	1'sh1
RspValid	1'shX
RspId	4'shX
DataValid	1'shX

Figure 5-5: Parent-Children relation highlighted in Verdi Transaction Browser

5.5 Debug Scenario 2. Correlating Transactions to RTL Signals

In scenarios where simulations were failing due to incorrect handshaking algorithm in the RTL we used Transaction Browser to debug and analyze the transactions and HDL signals together. Verdi provides the option to synchronize the Transaction Browser with the waveform viewer. In Figure 5-6Figure 5-6, the *l1d_load_A_drvr* transaction is selected in the Transaction Browser. Simultanously, in the waveform viewer, two markers labeled O*bj/Trans Begin* and *Obj/Trans End* appear to show the time range associated with the *l1d_load_A_drvr* transaction.


<Verdi:Container:3> tBrowser and nWave
Transaction
Transaction Begin/End Times
Obj/Trans Begin
Obj/Trans End
Selected: /top/mob_l2_load[0]/RspId[3:0]

Attribute	Value
$label	"Temp Load Txn"
$beginTime	217581
$endTime	218941
$type	"Transaction"
Id	4'sh0
Addr	46'sh10584f4febc4
Type	44'shc0300003df4
Type.ReadI...	1'sh1

Figure 5-6: Verdi's debug view of correlating transactions to RTL signals

Traditional debug methods, where we had to search the simulation log files to find the simulation-time window during which a trasaction and its related transactions were active, took a long time.

Verdi's capability to sync the Transaction Browser to the waveform viewer enabled us to quickly determine the simulation time range during which we needed to analyze the RTL signals' behavior.

5.6 Debug Scenario 3. Unfinished Transactions

A number of tests were failing due to a bug in the testbench. Some of the L1 transactions were not completing resulting in simulation hangs.

Verdi Transaction Browser displays the unfinished transactions with a different pattern as shown in Figure 5-7. Also, in the Attribute Detail pane, the end time of an unfinished transaction is suffixed with *(unfinish)* in the value column.

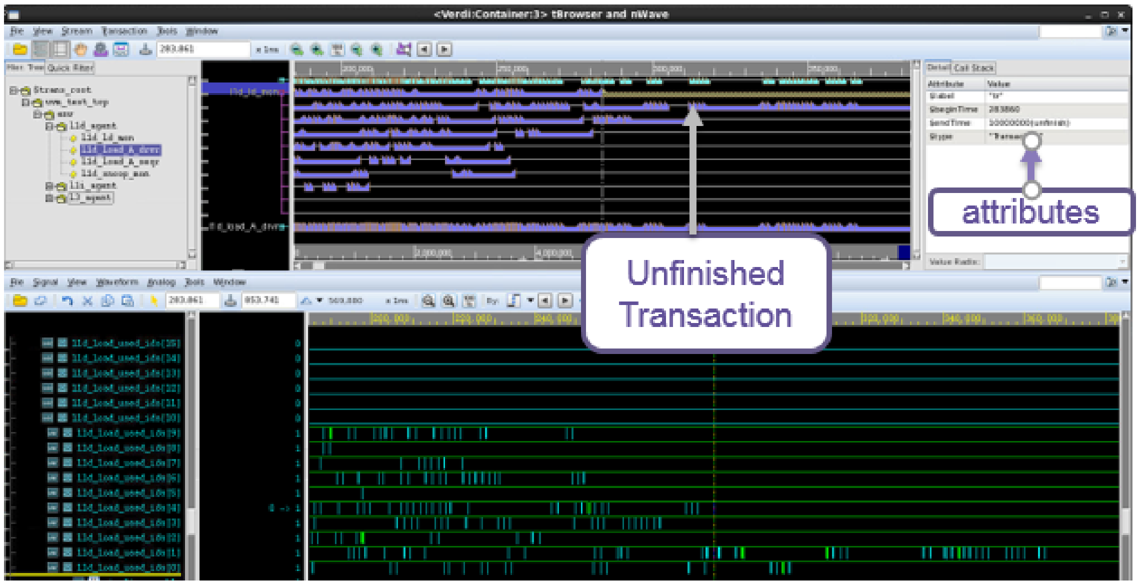
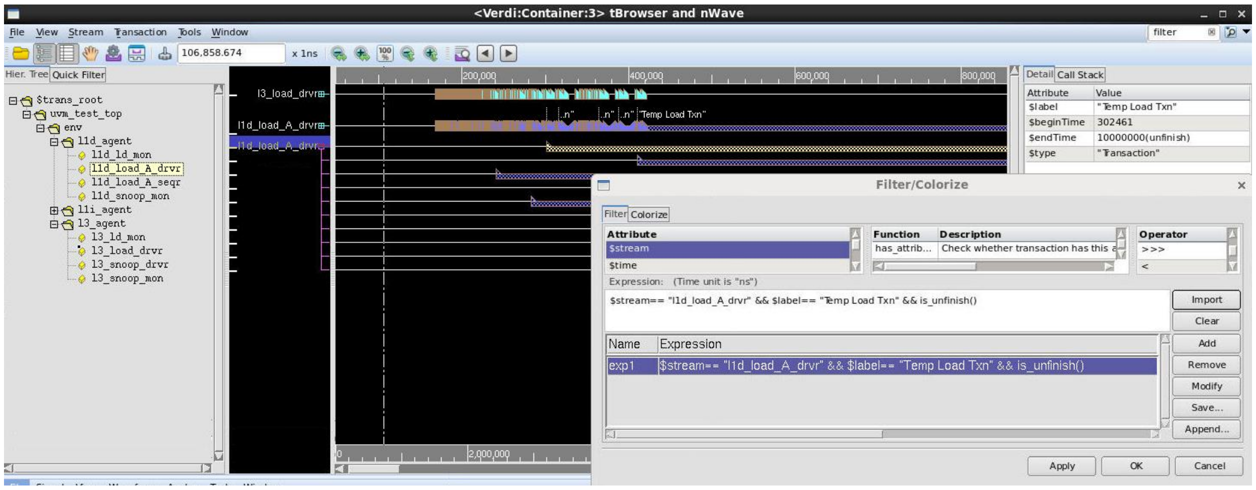


Figure 5-7: View of unfinished transactions in Verdi Transaction Browser

Prior to deploying Verdi Transaction Debug in our flow, we typically spend a long time searching through the UVM printed messages in the log file for a clue to which testbench object or RTL component was causing a hang. Only a few of us are familiar with how to use interactive debug to step throught the code to root cause the hang. For the rest of us the painful process of analyzing the simuation log files was the only option. Interactive debug also was not the ideal choice for debugging hangs that happened hours into a simulation run. Simulations run slower when running in interactive mode. This prolongate the overall debug process.

Visualizing the unfinished transactions in the Transaction Browser was a much quicker way to isolate the unfinished transactions, find the transaction's corresponding source code in the source browser, and quickly correlate the RTL signals and diagnose the problem. We also learned that using Verdi's powerful searching and filtering mechanism we could filter and display only the unfinished transactions in the browser. This was another convenient way to confirm or rule out the existance of unfinished transactions.



Attribute	Value
$label	"Temp Load Txn"
$beginTime	302461
$endTime	10000000(unfinish)
$type	"Transaction"

Attribute	Function	Description	Operator
$stream	has_attrib...	Check whether transaction has this a	>>>
$time			<

Expression: (Time unit is "ns")

```
$stream== "l1d_load_A_drvr" && $label== "Temp Load Txn" && is_unfinish()
```

Name	Expression
exp1	$stream== "l1d_load_A_drvr" && $label== "Temp Load Txn" && is_unfinish()

Figure 5-8: Using the Filter option to filter and display the unfinished transactions of *l1d_load_A_drvr*

We also learned that it would be convenient if Verdi could print warning messages in the simulation log file to indicate unfinished transaction. This would provide for a quick check for any unfinished transactions in the test before we inoke the GUI. An enhancement request was filed, Verdi R&D implemented this new feature, and Synopsys delivered the feature in a newer release.

6. Results

In a full chip environment, it might be easier and natural to create certain complex stimuli, however, due to longer compile and simulation times and complexity of the design, reproducing, debugging and fixing a bug could be very challenging. On the other hand, in a unit level environment, it takes time to craft the components which create interesting sequences, however, once developed, it is much easier to create stimuli, reproduce a failure, and debug and fix a bug than in the chip level. Unit level tests typically are quick to compile and simulate and are less complex than chip level.

Unique ID generation used with back to back Load and Snoop Requests have exposed bugs on the L2 Queues/Pipes allocation and release. Interesting Address generation helped to generate colliding stimuli generation that exposed dead-lock bugs, timeouts, hangs, and corner case bugs. Response Delay model helped to create responses that were either spread or crowded randomly which exposed corner case bugs.

7. Conclusions

UVM Based constrained random unit level testbenches helped us to catch many bugs early, including corner case bugs that were harder to reach in system level. Verdi Transaction Debug was key to quickly debugging these complex bugs. Simple hooks to the UVM APIs begin_tr, begin_child_tr, and end_tr, allowed us to record and link parent and child transactions of the UVM components such as drivers and monitors. Then Verdi Transaction Browser was used to visualize the transactions in a browser view, analyze the transactions and their signal values in the Detailed Attribute pane, and coordinate the Trasactions start and end times to the simulation time in the Verdi waveform brower. Verdi Transaction Brower and Relation Navigator were also usesd to quickly find the linked transactions, which were spread across hundreds of clock cycles, by highlighting the related transactions. This allowed for quickly debugging failing tests due to missing responses or unfinished transactions. One might argue that all this is possible using the traditional debug methods of greping log files and ananlyzing signals in the waveform viewer and then go back and forth between the log file, source code, and the waveform , but it for sure is time consuming and tedious. When your debug is easier you can spend your time and effort on improving your verification in other areas.

8. References

[1] OVM & UVM Techniques for Terminating Tests, Clifford E. Cummings, Sunburst Design, Inc. and Tom Fitzpatrick, Mentor Graphics Corporation, DVCon 2011

[2] Verdi Transaction Debug Platform User Guide Version L-2016.06

[3] *Memory Systems: Cache, DRAM, Disk* Book by Bruce Jacob, David T. Wang, and Spencer Ng

[4] Universal Verification Methodology (UVM) 1.1 Class Reference, June 2011, by Accellera