Applying Transaction-level Debug and Analysis Techniques to DUT Simulated Activity Using Data-Mining Techniques

Leo Chai
Research & Development
nVidia, Inc.
Shanghai, China

Bindesh Patel, Jun Zhao
Research & Development
Synopsys, Inc.
Mountain View, California

Abstract— **Bus activity during RTL simulation is normally captured into a debug database at the bit-level. However, since transactions provide a clearer, bus-protocol view of activity in a design and verification environment, engineers often manually map this bit-level data to abstracted bus transactions. This is the general context for this paper. Significant work and capability already exists to directly record transaction-level traces from higher-abstraction models such as a UVM-based SystemVerilog testbench. However, the focus of this paper is applying these same transaction-level debug and analysis techniques to the DUT (design-under-test) activity.**

Keywords— debug, transacation, testbench, UVM, SystemVerilog, assertion

I. Introduction

For analyzing activity inside a design (DUT), basic bundling of nets into buses for debug and analysis purposes is a common technique but is limited to concatenation and sometimes logical expressions of the said nets at one particular snapshot time. As an example, with these basic bundling capabilities, the user can view the address bus as the actual address (in hex) rather than the individual bits of the address bus. For bus transactions, changes over time also have to be considered. For example, a READ transaction on a DUT bus usually corresponds to a set of signals changing at particular windows of different times. An engineer will typically decode these changes into the READ transaction, mentally or using a notebook, since that is more understandable and applicable to the debug and analysis.

The solution proposed in this paper does what the user was doing manually, but now automatically. This is accomplished with an engine that scans a recorded trace of signals and look for patterns of activity that constitute the transactions that the user wants to abstract out. While the engine itself is straight-forward, a mechanism needs to be provided to the user to specify a temporal sequence of events. We will discuss several possibilities but we will focus on the use of SystemVerilog Assertion (SVA) syntax and semantics to specify the sequence. SVA is highlighted because it has provisions for everything needed to specify any temporal sequence, complex or basic. Additionally, most engineers either already know the language or are able to easily grasp it as it is closely related to Verilog.

The techniques described have been applied to a real design and verification environment and these experiences will also be shared in this paper. We will illustrate some examples as well as present the efficiencies obtained when the proposals in this paper are adopted. We will conclude with proposals for future work to use more advanced visualization and analysis techniques specifically tailored for abstract data.

II. Traditional Signal-based Analysis and Debug

A. Simulation and Data Dumping

Today, most verification flows follow a similar pattern whereby an Engineer runs simulations of his environment using an automated script and/or Makefile which at the end returns some measure of pass or fail. If there is a failure, another simulation is run for the failing test but now with the dumping of debug data enabled. This data is typically in the debugging tool's native format such as the Fast Signal Database (FSDB) from Synopsys. Once the data is generated, the Engineer can load it into the debug tool for further analysis. There may be varying levels of automation in this flow but the main scope does not change.

B. *Debug Methods*

Once the data and design description, typically in a hardware description language (HDL), is loaded into the debug tool, various techniques are employed to find the root-cause of the failure. Standard techniques based on waveform analysis have been around for years. (Figure 1)

terms of time and effort. An example of a recent technology that automates the identification of the root-cause of an anomaly that the Engineer is tracing is Behavior Analysis which uses formal techniques.

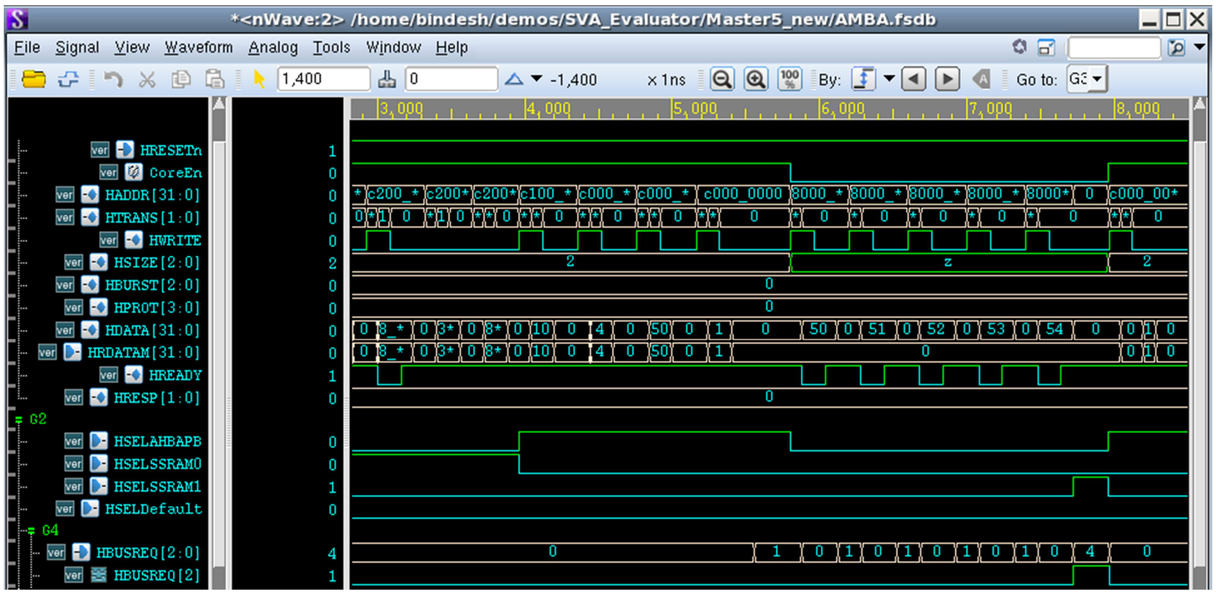


Figure 1. Waveform-based debug and analysis of AMBA bus activity

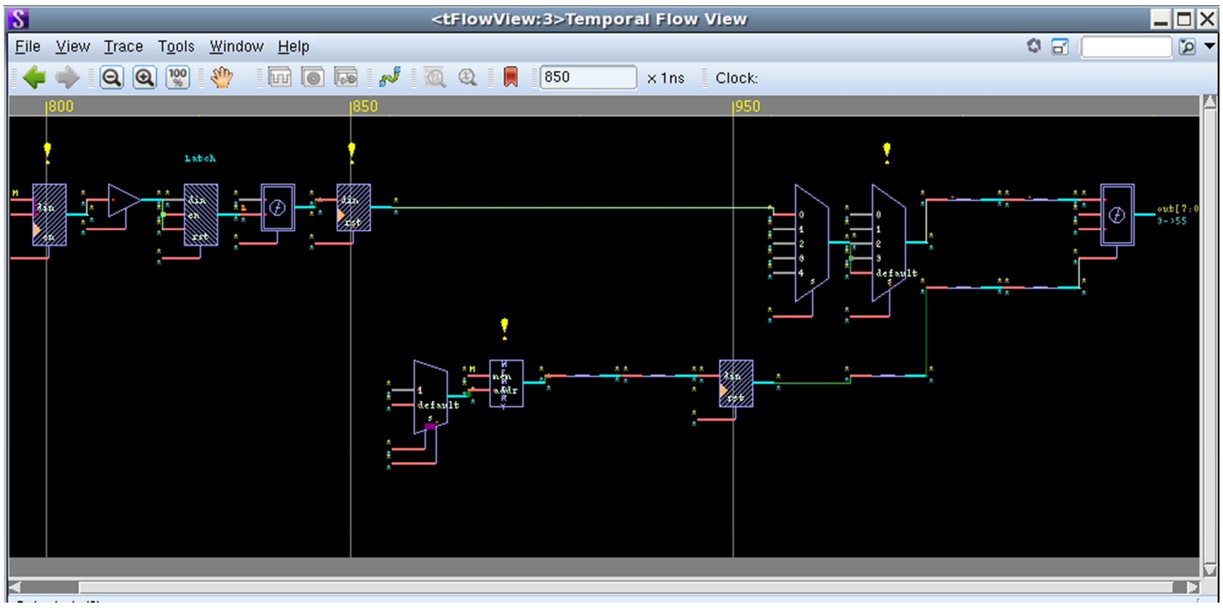

<tFlowView:3>Temporal Flow View
File View Trace Tools Window Help
850
x 1ns
Clock:
800
850
950
Latch
default

Figure 2. Behavioral analysis techniques to do automatic root cause analysis using semi-formal methods

More recently, technologies have advanced to provide new ways and automation which are geared towards getting to the root-cause as quickly as possible. More research and investments in such debug technologies have increased given the generally accepted fact that debug now takes almost half of the verification cycle in

This behavioral analysis technology allows the Engineer to perform root-cause analysis automatically over multiple levels of logic and clock cycles with a single or minimal number of commands. (Figure 2)

III. Introduction to Transaction-based Modeling and Analysis

Transaction-based in electronic design automation (EDA) generally refers to the higher-level communication that occurs between components. These components are typically modeled at a higher abstraction than HDL's and as such references to data are at a corresponding higher level abstraction of transactions. Example methodologies that employ this higher level modeling are SystemC and UVM testbenches. Bus designers and architects refer to a transaction as a temporal event of signal-level activity that does some bus action such as read or a write. For the purpose of this paper, we will refer to a transaction as any encapsulation of multiple events of data activity. The events can be and typically are over time.

A. *Debug and Analysis at the Transaction-Level*

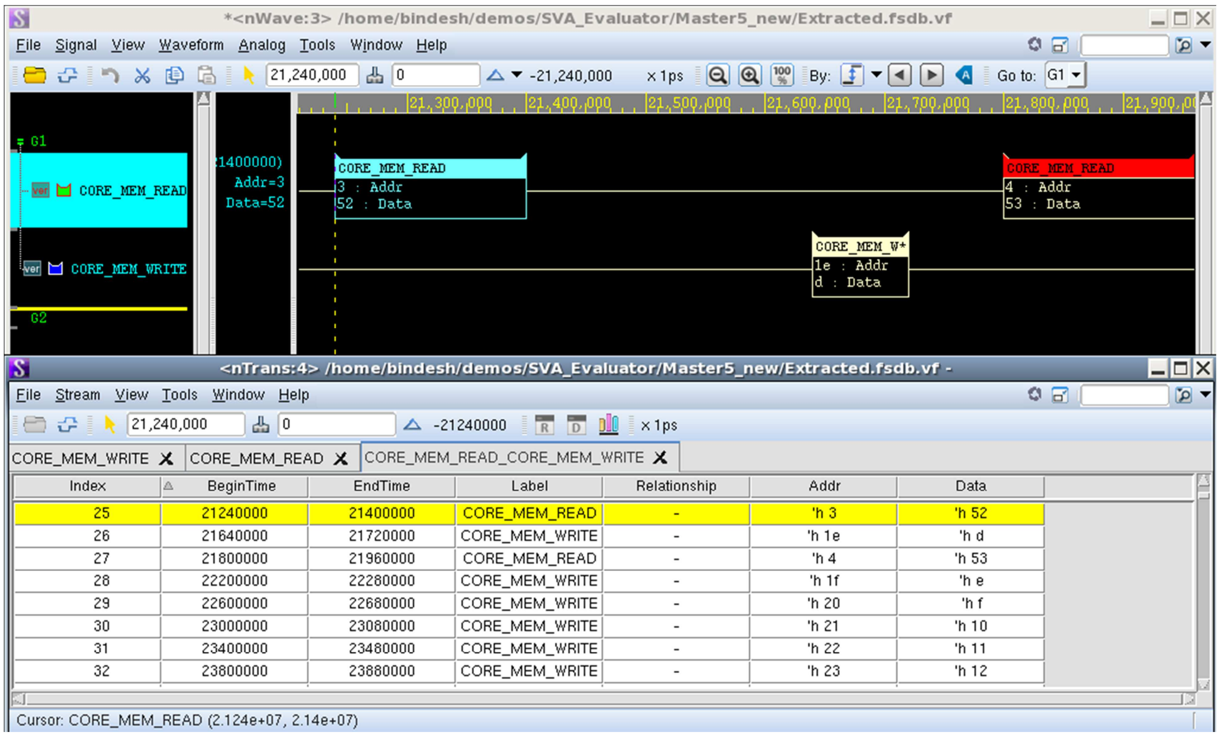
Advanced debug platforms have evolved to record and provide applications for transaction-based debug and analysis. The notion of simple bus bundling that is a collection of signals at one snapshot time was a good starting basis. The next natural step was to extend existing waveform applications to display a transaction. Of course, this depends on the data being recorded and stored into a debug database, such as FSDB.

Figure 3 shows one implementation of showing transaction-level data in a waveform tool.

Additional applications have been developed that are targeted at detailed analysis of transaction-level data. One such application is based on standard spreadsheets – that is, a table view which includes capabilities for filtering and sorting. This is also illustrated in Figure 3.

IV. Feeding Transaction-based Tools

To utilize the transaction debug and analysis capabilities described in the previous sections requires the data to be recorded at that same higher abstraction level. The SCV extension of SystemC has built-in capabilities to plug in a debug recording interface which then allows for transaction-level data to be recorded into a debug database such as FSDB. Users can then view and debug at the transaction-level. More recently, testbench methodologies such as the Universal Verification Methodology (UVM) have gained rapid adoption. These are founded on base class libraries that are built around a transaction-level data object. UVM, for example, allows random sequences of transactions to be easily generated, which in turn feed into the DUT as stimulus. DUT responses are also encoded into transaction-level data by UVM monitors for easier automated analysis by the verification environment. UVM in particular provides hooks in the library where a transaction recording



Index	BeginTime	EndTime	Label	Relationship	Addr	Data
25	21240000	21400000	CORE_MEM_READ	-	'h 3	'h 52
26	21640000	21720000	CORE_MEM_WRITE	-	'h 1e	'h d
27	21800000	21960000	CORE_MEM_READ	-	'h 4	'h 53
28	22200000	22280000	CORE_MEM_WRITE	-	'h 1f	'h e
29	22600000	22680000	CORE_MEM_WRITE	-	'h 20	'h f
30	23000000	23080000	CORE_MEM_WRITE	-	'h 21	'h 10
31	23400000	23480000	CORE_MEM_WRITE	-	'h 22	'h 11
32	23800000	23880000	CORE_MEM_WRITE	-	'h 23	'h 12

Cursor: CORE_MEM_READ (2.124e+07, 2.14e+07)

Figure 3. Transaction debug and analysis tools based on waveforms and spreadsheets

interface can be plugged in that allows the sequencer and monitor transactions to be recorded into a debug database. These are natural applications for transaction-based debug and analysis and flows are provided to generate the data in a highly automated fashion.

A. Data Mining from DUT

What we have discussed so far are flows that can naturally generate transaction-level debug data – natural in the sense that the context that these environments, such as SystemVerilog UVM or SystemC SCV, operate on are themselves based on the notion of a transaction.

Designs themselves, referred to DUTs, are typically described at the Register Transfer Level (RTL) using Verilog or VHDL. The debug data dumped from a simulator usually consists of discrete signal-level activity. Debug tools and simulators are able to capture, store, and present busses to the user as such. The same can be done for ENUM types, with the debug tools providing intuitive views to the user. These primitive levels of encapsulation or bundling are helpful to users.

Most modern SoCs are built around a standard or custom bus architecture, such as AMBA. However, as far as the design itself is concerned, the model is still at RTL and thus the same signal-level dump of the bus signals is generated for debug. An automated mechanism for the simulator or debugger to decode, or more precisely *encode*, bus signal-level activity into the corresponding bus transactions would provide users a more intuitive and clear picture of SoC activity. (Figure 4)

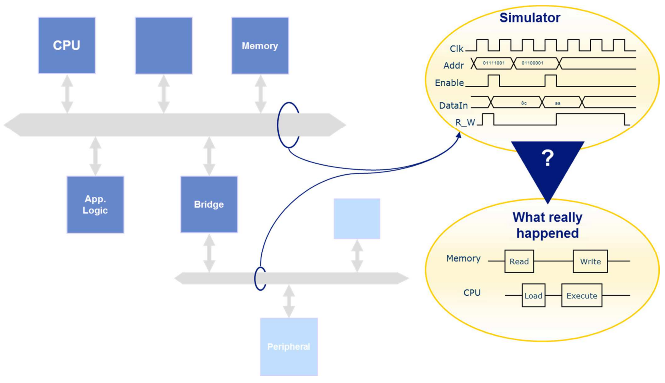


Figure 4. Transaction convey a more intuitive meaning of bus protocol and SoC activity to the user

Buses are defined in terms of transactions anyway and these are the perfect basis of presenting the user a view of bus activity. In fact, users, even when analyzing bus signal-level waveforms, often manually encode the activity into the bus transactions using old-fashioned pen and paper.

So far, we have established that the ideal way to record, store, and view DUT bus activity is by leveraging transaction-based analysis capabilities already available in many advanced debug platforms. An added, but key, complication here is that a bus transaction is comprised of signal events happening over time. Therefore, basic single-time bundling capabilities provided in standard debuggers cannot be used. The user has to be provided a mechanism to generate the data. One solution could be to provide a set of Verilog task-based APIs that the user codes directly into their design so that the data is recorded at the transaction-level during the simulation. These exist in some cases, but are difficult and cumbersome for the user to code. A better solution would be to provide the user a mechanism for extracting, or mining, transaction-level data from signal-level debug data. A GUI-based wizard for the user to describe the sequence of specific signal events that a transaction is composed of can be provided but that can be also cumbersome and the re-usability is questionable. The preferred route is to provide a language that can allow users to specify the transaction semantics.

B. Using SVA to Describe a Transaction

```
sequence core_memory_write;
  logic [10:0] Addr;
  logic [31:0] Data;

  (1) ## 0
  (EN == 1'b1 && WE == 1'b1,
   Addr = ADDR, Data = DI) ##1
  (!(EN == 1'b1 && WE == 1'b1));
endsequence

sequence core_memory_read;
  logic [10:0] Addr;
  logic [31:0] Data;

  (1) ## 0
  (WE == 1'b0 && RST == 1'b0 &&
   RDInvalid == 1'b0, Addr = ADDR) ##1
  (RDInvalid == 1'b0) ##1
  (1, Data = DO);
Endsequence

CORE_MEM_WRITE : assert
     property(@(posedge CLK)
     core_memory_write);

CORE_MEM_READ  : assert
     property(@(posedge CLK)
     core_memory_read);
```

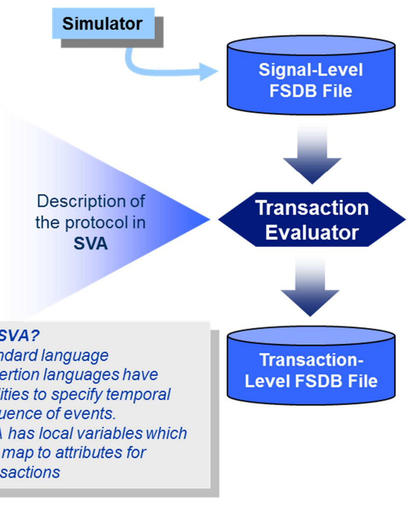
Figure 5. SVA code snippets showing sequences that describe AMBA Read and Write tranactions

When attempting to develop a language or text-based configuration file that can suitably describe transaction activity at the signal-level, it was noted that the standard SystemVerilog Assertion (SVA) syntax can fulfill all requirements. SVA sequences can be used to describe complex temporal sequence of events. Once thusly described, a utility engine can be developed that would mine for the sequence of events from an existing signal-level debug database. Such a utility would be similar to existing post-simulation assertion checker programs that mine assertion failures from a debug database, rather than having the simulator do the checking.

Figure 5 shows a SVA sequence that describes AMBA AHB READ and WRITE transactions.

The extraction utility can mine the read and write transactions based on the signal-level activity of the WE, EN, RST, RDInvalid, Addr, and Data signals in accordance with the temporal sequence specified in the SVA code.

The data-mining utility can be designed to also extract local variables within the sequence as attributes to the transaction. In Figure 5, for example, Addr and Data can be extracted and stored as transaction attributes. We will discuss transaction and attribute display in the next section.



Simulator
Signal-Level FSDB File
Description of the protocol in SVA
Transaction Evaluator
Transaction-Level FSDB File
Why SVA?
• Standard language
• Assertion languages have facilities to specify temporal sequence of events.
• SVA has local variables which can map to attributes for transactions

Figure 6. Flow for transaction data-mining from design debug database

Figure 6 illustrates the flow for dumping a standard design signal debug database that is then processed for transactions.

As a side note, it should be noted that the application of mining transaction data out of a signal-level database isn't necessarily limited to debug and analysis; it can also

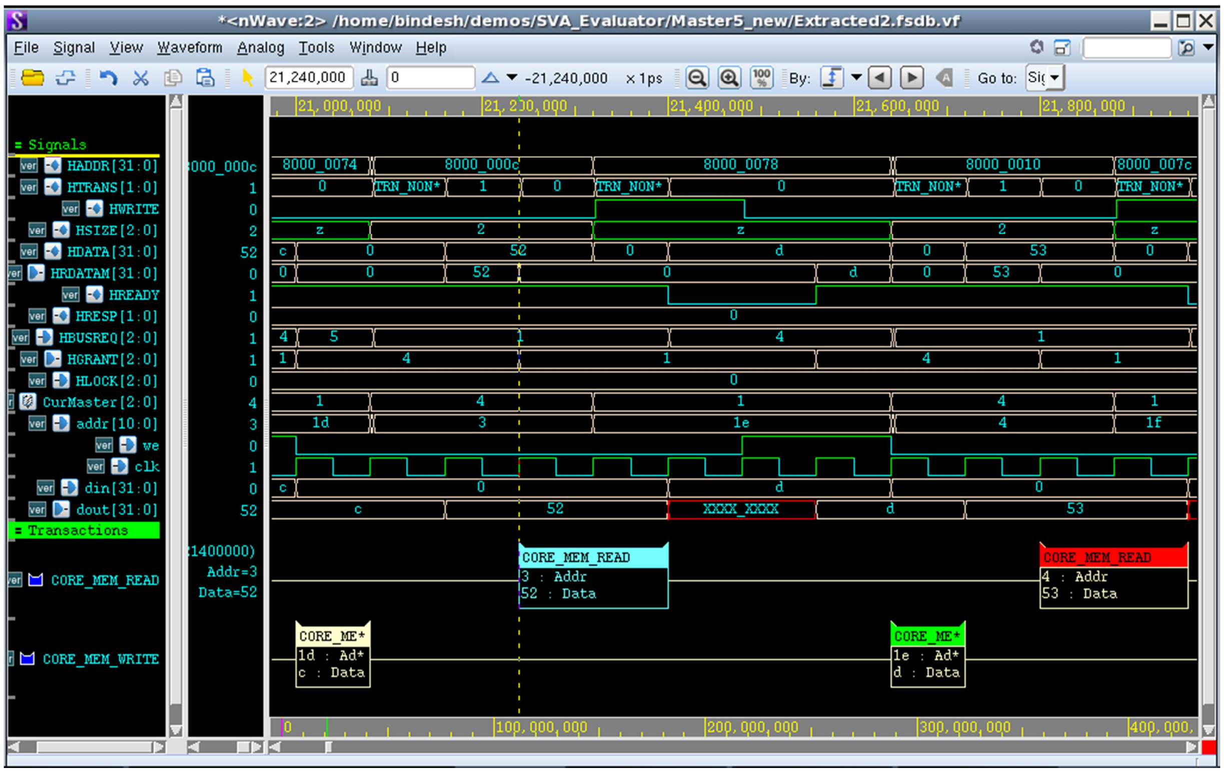


Figure 7. Signal-level and transaction-level for AMBA

be used as a quick-and-dirty means to identify coverage information. Today, users often do this with waveform inspection, a mostly manual process of identifying the interesting sequence of events that the user is interested in covering. Using SVA code and the post-simulation extraction utility, this process can be automated and more importantly, made more deterministic, accurate, and repeatable (i.e. reusable).

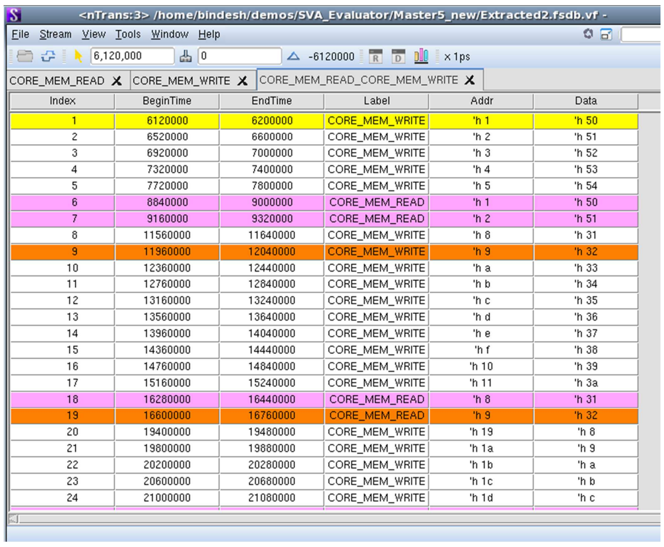
V. Transaction-Based Tools

Once the data is available, the debug platform applications can be used to debug, visualize, and analyze. Currently, two techniques are used, with more future specialized applications in the works. It is clear there is a lot of potential for innovation in this area.

Firstly, waveform tools can be enhanced to now additionally show transaction-level data. Figure 7 shows a signal-level waveform of some AMBA AHB bus activity as well as the extracted transaction-level activity

Figure 7 also illustrates the clarity provided by the transaction-level waveform over the corresponding signal-level activity. Having both the abstract and detailed views side-by-side allows the user to get the high-level view at all times but also have access to the details when needed. The attributes of each transaction, Data and Attr, are also show as support data in the waveform.

The same data can also be imported into a spreadsheet-like tool as show in Figure 8.



<nTrans:3> /home/bindesh/demos/SVA_Evaluator/Master5_new/Extracted2.fsdb.vf -
File Stream View Tools Window Help
6,120,000
0
-6120000
x 1ps
CORE_MEM_READ
CORE_MEM_WRITE
CORE_MEM_READ_CORE_MEM_WRITE

Index	BeginTime	EndTime	Label	Addr	Data
1	6120000	6200000	CORE_MEM_WRITE	'h 1	'h 50
2	6520000	6600000	CORE_MEM_WRITE	'h 2	'h 51
3	6920000	7000000	CORE_MEM_WRITE	'h 3	'h 52
4	7320000	7400000	CORE_MEM_WRITE	'h 4	'h 53
5	7720000	7800000	CORE_MEM_WRITE	'h 5	'h 54
6	8840000	9000000	CORE_MEM_READ	'h 1	'h 50
7	9160000	9320000	CORE_MEM_READ	'h 2	'h 51
8	11560000	11640000	CORE_MEM_WRITE	'h 8	'h 31
9	11960000	12040000	CORE_MEM_WRITE	'h 9	'h 32
10	12360000	12440000	CORE_MEM_WRITE	'h a	'h 33
11	12760000	12840000	CORE_MEM_WRITE	'h b	'h 34
12	13160000	13240000	CORE_MEM_WRITE	'h c	'h 35
13	13560000	13640000	CORE_MEM_WRITE	'h d	'h 36
14	13960000	14040000	CORE_MEM_WRITE	'h e	'h 37
15	14360000	14440000	CORE_MEM_WRITE	'h f	'h 38
16	14760000	14840000	CORE_MEM_WRITE	'h 10	'h 39
17	15160000	15240000	CORE_MEM_WRITE	'h 11	'h 3a
18	16280000	16440000	CORE_MEM_READ	'h 8	'h 31
19	16600000	16760000	CORE_MEM_READ	'h 9	'h 32
20	19400000	19480000	CORE_MEM_WRITE	'h 19	'h 8
21	19800000	19880000	CORE_MEM_WRITE	'h 1a	'h 9
22	20200000	20280000	CORE_MEM_WRITE	'h 1b	'h a
23	20600000	20680000	CORE_MEM_WRITE	'h 1c	'h b
24	21000000	21080000	CORE_MEM_WRITE	'h 1d	'h c

Figure 8. Analyzing Transaction-level data in a spreadsheet-based utility

The advantage of this view is that it allows the user to perform analysis (ala spreadsheet) functions like sorting and flittering based on any column. In this view, the columns show the attributes.

VI. Case Study

A. *Data Mining*

As we have discussed, transaction-level modeling is commonly used to model system-level behavior. At nVidia, we have many standard or user- defined general transaction protocols applied and implemented in various SOC-related projects. However, some knowledge of transaction protocol details is necessary for SOC verification engineers to debug the design with a simulator-dumped debug database. For efficient and practical debug, engineers are forced to study protocol details, which in itself is a non-trivial task. It is especially inefficient for design debuggers, who are usually not familiar with protocol knowledge,

Now, with the help of transaction debug and analysis tools based on protocol waveforms data-mined from a simulator-dumped signal-based database (e.g. FSDB), users have a means to clearly visualize and easily understand the transaction protocol information. After bus signals are recorded into an FSDB file as per our usual process, an extraction engine called Transaction Evaluator in the Verdi platform can be used to process it and extract transaction-level FSDB.

The transaction evaluation engine requires the user to code the protocol so it can recognize a transaction when processing the signal-level FSDB. SVA is used to specify the protocol. This SVA code can be reused whenever the protocol is used again.

Before transactions are extracted using the SVA, the following steps have to be done:

1) *SVA coding for Transaction Dumping*

Add SVA code to the design that will ultimately be used to extract transactions. Here is a summary of coding styles that we apply:

- Use sequence or property block for modeling the transaction, while deep nesting range repetition and unbound range delay, e.g. ##[0:$], are not suggested as it will impact performance.
- SVA local variables, including those declared in the sequence or property of a specific assertion will be recorded as attributes of the extracted transactions; therefore, it's better not to declare local variables with the same name across different sequences/properties so as not to cause confusion during debug.

- Specify the transaction label name of a specific sequence by declaring local.

- Use plusargs as a switch to enable and disable assertion in simulation.

```
property APB_READ;
    logic [ 31 :0] Addr;     // local
variable to record attribute  addr
    logic [ 31 :0] Data;     // local
variable to record attribute  data
    logic [127:0] Client;   // local
variable to record attribute client
name

    @(posedge pclk) disable iff
(disable_ntx_dump)
    ((psel && !penable && !pwrite),
Addr = paddr) |-> ##1
    (((psel && penable && !pwrite &&
pready)[->1]), Data = prdata, Client =
"dtv");
endproperty

APB_READ_nTX  : assert
property(APB_READ);
```

Figure 9. Code snippet from our protocol transaction extraction library

Figure 9 shows code snippet that illustrates our coding style.

2) *Waveform dumping*

Compile SVA module together with design for the simulator, then run a simulation and generate an FSDB file containing design and assertion data as per our normal process.

Verdi also includes an assertion evaluation engine that supports a post-processing mechanism for assertion calculation, which allows for the SVA module not to be compiled in the simulator. However, this post process way is not recommended for complete (i.e all assertions in environment) assertion evaluation due to performance considerations. Rather, it is to be used for a small subset of assertion evaluation. This may be especially useful during assertion code development, since it allows for a potentially quicker code-and-try cycle for assertions. This flow has not been tried at nVidia yet.

3) *Evaluate the assertions for transaction extraction.*

- Load the design and FSDB file into the Verdi platform.

- Open Verdi's Transaction Evaluator

  Invoke "Tools" -> "Transaction" -> "Evaluator" in Verdi's nTrace window. This opens the "Transaction Evaluator" form where all SVA assert signals are listed.

- Enable assertions to be evaluated.

  When the user selects the scope of interest, the transactions are listed in the middle pane. After that, the users has to select the assertions to be extracted and click either the "Add Selected Transaction" button or "Add All Transactions" button to move the selection to the Evaluation Enable List pane.

- Invoke evaluator engine to process.

  Click "Evaluate", the transactions will be extracted

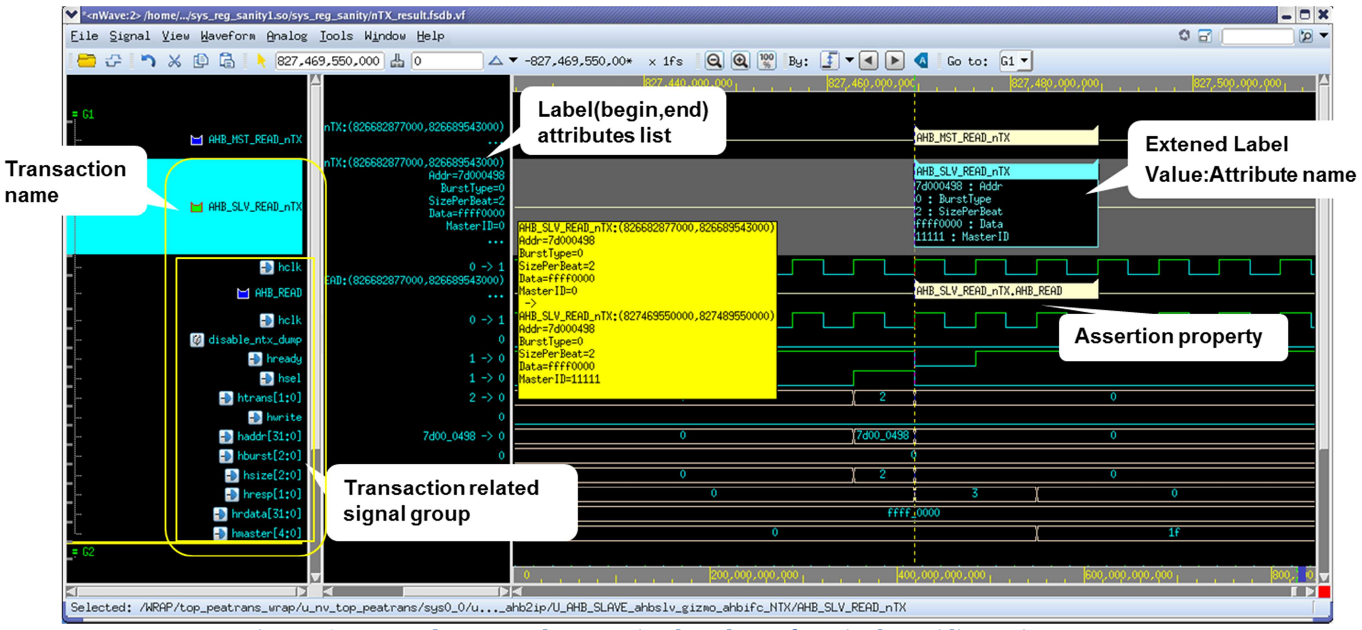


Figure 10. Example extracted transaction-based waveform in the nVidia environment

from the assertion code and saved to the specified file. After the extraction, this FSDB file will automatically be loaded into Verdi and you can start using all transaction viewing and analysis commands for debug in addition to the standard Verdi capability.

- Detailed Transaction View

  Load the transaction FSDB file into nWave the same way as a general FSDB file. A stream name will be shown in the signal pane; begin time, end time, and attributes are shown in the value pane; and the transaction will be shown in the waveform pane as rectangles enclosing all the attributes.

- Linking between Transaction and Signals

  Double click the transaction stream in the signal pane, and then all the signals related to this stream are displayed below the stream. These are the signals used by the transaction evaluator to generate the transaction stream. This way, we have access to this level of detail if and when needed.

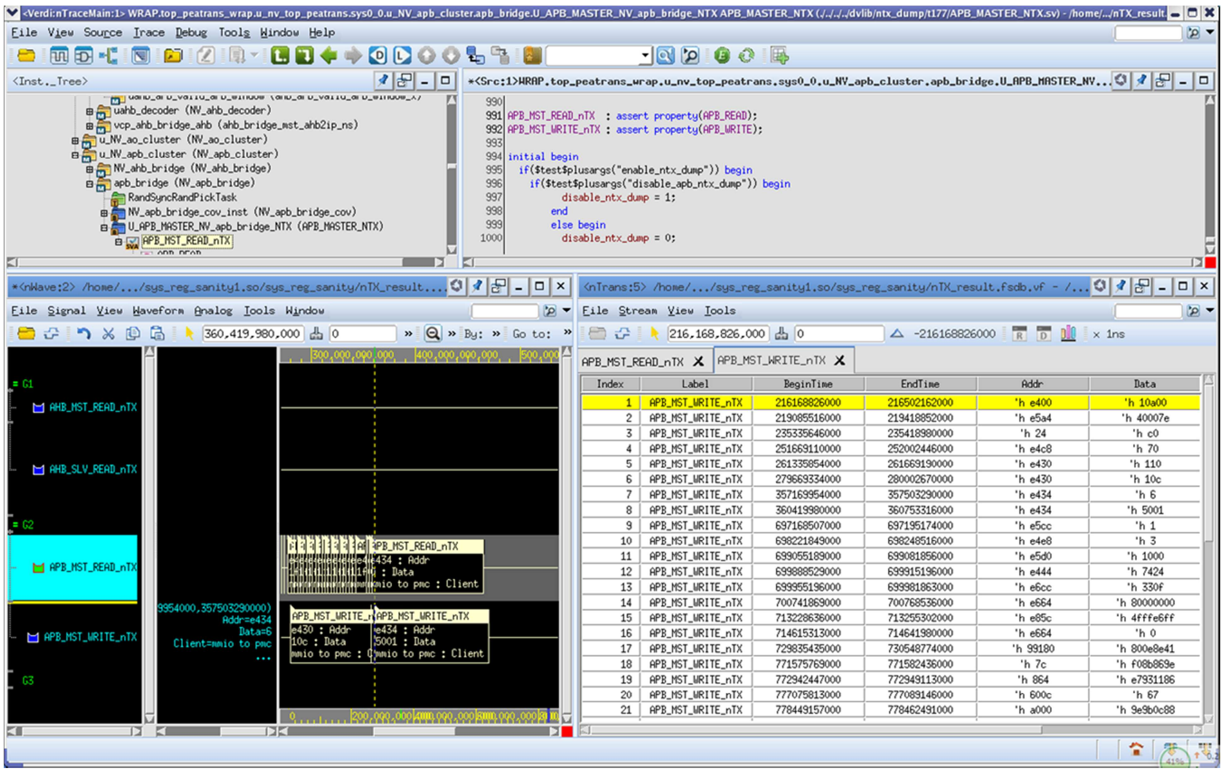
Figure 10 shows extracted transactions in waveform in our (nVidia) environment. As illustrated, the transaction-level waveform provides a clear view into the bus in the context of protocol transaction activity.

In addition to waveform-based debug and analysis, the Verdi platform also includes spreadsheet-based review for deeper analysis. This is shown in Figure 11.

The spreadsheet tool has capabilities similar to a traditional spreadsheet, such as sorting, filtering, etc. Both the waveform and spreadsheet provides filtering and color highlighting of transactions that match a user-specified expression, allowing our users to focus on the transactions of interest.

B. Data Logging

As previously discussed, transaction-level data can be natively dumped from a UVM environment – specifically, sequencer, driver, and monitor transaction-level activity can be recorded for debug and analysis at the transaction-level. This provides much-needed visibility into a UVM testbench during post-simulation debug. This recording is accomplished by way of a recorder module that is specific to the debug database that the user is aiming for (see Figure 12).



<Verdi:nTraceMain:1> WRAP.top_peatrans_wrap.u_nv_top_peatrans.sys0_0.u_NV_apb_cluster.apb_bridge.U_APB_MASTER_NV_apb_bridge_NTX APB_MASTER_NTX (./../../../dvlib/ntx_dump/t177/APB_MASTER_NTX.sv) - /home/.../nTX_result
File View Source Trace Debug Tools Window Help
<Inst._Tree>
*<Src:1>WRAP.top_peatrans_wrap.u_nv_top_peatrans.sys0_0.u_NV_apb_cluster.apb_bridge.U_APB_MASTER_NV...
*<nWave:2> /home/.../sys_reg_sanity1.so/sys_reg_sanity/nTX_result....
File Signal View Waveform Analog Tools Window
360,419,980,000
G1
AHB_MST_READ_nTX
AHB_SLV_READ_nTX
G2
APB_MST_READ_nTX
APB_MST_WRITE_nTX
G3
<nTrans:5> /home/.../sys_reg_sanity1.so/sys_reg_sanity/nTX_result.fsdb.vf - /...
File Stream View Tools
216,168,826,000
-216168826000
x 1ns
APB_MST_READ_nTX
APB_MST_WRITE_nTX

```
uahb_decoder (NV_ahb_decoder)
vcp_ahb_bridge_ahb (ahb_bridge_mst_ahb2ip_ns)
u_NV_ao_cluster (NV_ao_cluster)
u_NV_apb_cluster (NV_apb_cluster)
  NV_ahb_bridge (NV_ahb_bridge)
  apb_bridge (NV_apb_bridge)
    RandSyncRandPickTask
    NV_apb_bridge_cov_inst (NV_apb_bridge_cov)
    U_APB_MASTER_NV_apb_bridge_NTX (APB_MASTER_NTX)
      APB_MST_READ_nTX
```

```
APB_MST_READ_nTX  : assert property(APB_READ);
APB_MST_WRITE_nTX : assert property(APB_WRITE);

initial begin
  if($test$plusargs("enable_ntx_dump")) begin
    if($test$plusargs("disable_apb_ntx_dump")) begin
        disable_ntx_dump = 1;
      end
      else begin
        disable_ntx_dump = 0;
```

Index	Label	BeginTime	EndTime	Addr	Data
1	APB_MST_WRITE_nTX	216168826000	216502162000	'h e400	'h 10a00
2	APB_MST_WRITE_nTX	219085516000	219418852000	'h e5a4	'h 40007e
3	APB_MST_WRITE_nTX	235335646000	235418980000	'h 24	'h c0
4	APB_MST_WRITE_nTX	251669110000	252002446000	'h e4c8	'h 70
5	APB_MST_WRITE_nTX	261335854000	261669190000	'h e430	'h 110
6	APB_MST_WRITE_nTX	279669334000	280002670000	'h e430	'h 10c
7	APB_MST_WRITE_nTX	357169954000	357503290000	'h e434	'h 6
8	APB_MST_WRITE_nTX	360419980000	360753316000	'h e434	'h 5001
9	APB_MST_WRITE_nTX	697168507000	697195174000	'h e5cc	'h 1
10	APB_MST_WRITE_nTX	698221849000	698248516000	'h e4e8	'h 3
11	APB_MST_WRITE_nTX	699055189000	699081856000	'h e5d0	'h 1000
12	APB_MST_WRITE_nTX	699888529000	699915196000	'h e444	'h 7424
13	APB_MST_WRITE_nTX	699955196000	699981863000	'h e6cc	'h 330f
14	APB_MST_WRITE_nTX	700741869000	700768536000	'h e664	'h 80000000
15	APB_MST_WRITE_nTX	713228636000	713255302000	'h e85c	'h 4fffe6ff
16	APB_MST_WRITE_nTX	714615313000	714641980000	'h e664	'h 0
17	APB_MST_WRITE_nTX	729835435000	730548774000	'h 99180	'h 800e8e41
18	APB_MST_WRITE_nTX	771575769000	771582436000	'h 7c	'h f08b869e
19	APB_MST_WRITE_nTX	772942447000	772949113000	'h 864	'h e7931186
20	APB_MST_WRITE_nTX	777075813000	777089146000	'h 600c	'h 67
21	APB_MST_WRITE_nTX	778449157000	778462491000	'h a000	'h 9e9b0c88

Figure 11. Deeper analysis of transaction data using spreadsheet-based tool

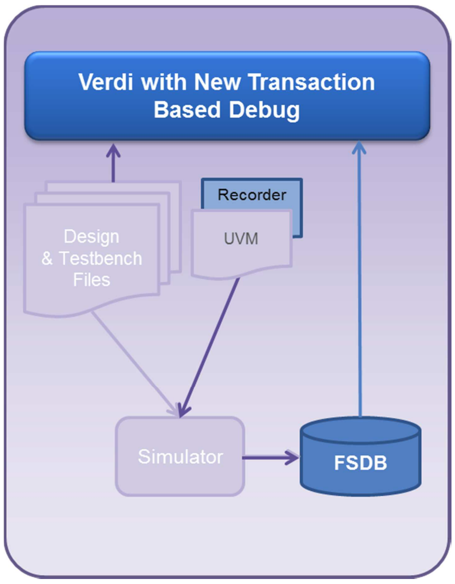


Figure 12. Automated flow for recording transaction activity from a UVM-based environment

The user simply includes this recorded module in their simulation and the data is recorded into a database, such as FSDB. Now, the user can utilize the transaction-based applications that the debugger of choice affords.

VII. CONCLUSION AND FUTURE WORK

While there are natural areas of application for high-level abstraction, or transaction-based, debug such as UVM and SystemC, the concepts and techniques are equally applicable and provide productivity gains for DUT buses. This requires mining transaction-level data from signal-level debug databases. SystemVerilog's assertion (SVA) component is a key aspect of this flow, and provides the syntax and semantics for users to specify or code the protocol transactions. This flow and resulting capabilities have been used in a real-life case at nVidia and proven to provide productivity gains in debug and analysis of SoC-based designs built around buses, standard or custom.

Much of future research and work in this area will be focused on the applications. While existing techniques that have origins in debugging discrete signal-level activity, such as waveforms, have been thus far leveraged, it is clear that any future innovations will have to rely on new and separate platforms geared especially for debug and analysis of high-level abstraction data. Even at the application level, the user will likely want to re-organize the data in various ways (2nd level of data-mining, but now at the application-level). For example, the user may want to see all the transactions that have the same address attribute together. Further possibilities may lie in the analysis aspects, such as performance analysis. Users may want to do performance-related analysis on the various bus transactions, which they may then be useful in optimizing the design itself.

Another area of research interest has the tracing back of transactions, in a somewhat similar fashion to tracing signal activity back logic level by logic level. This is non-trivial in that relationships between transactions have to exist and be recorded into the debug database.

REFERENCES

[1] YC. Hsu, B. Tabbara, YA. Chen, F. Tsai, "Advanced Techniques for RTL Debugging", DAC Proceedings, 2003

[2] UVM User Guide and Reference Manual, http:// http://www.accellera.org/downloads/standards/uvm

[3] R. Chen, B. Patel, and J. Zhao, "UVM Transaction Recording Enhancements", DVCon Proceedings, 2011