2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Facilitating Transactions in VHDL and SystemVerilog

Rich Edelman, Mentor, A Siemens Business, Fremont, CA, US (*rich_edelman@mentor.com*)

***Abstract*—This paper presents transaction recording and debug to the VHDL and SystemVerilog user via many different techniques: including traditional SV coding with bind, VHDL coding with bind and VHDL integration with UVVM. Transaction modeling will also be discussed for IP developers and System modeling.**

Keywords—Transactions, VHDL, Verilog, SystemVerilog, Debug, Verification IP

I. INTRODUCTION

Transactions, transaction recording and transaction debugging techniques have been available for many years, mainly using SystemVerilog or Verilog interfaces. A wider audience using VHDL and transactions can be imagined, with a native VHDL use model and some examples and a transaction recording best practices.

This paper uses various examples from SystemVerilog[1], UVM[2], VHDL[3] and UVVM[4] to illustrate the concepts. The complete source code is available from the author. This is not a SystemVerilog, UVM, VHDL or UVVM training example; the reader should have a solid background in SystemVerilog, UVM or VHDL and UVVM.

The goal of this paper is to educate and enable debugging with higher level transactions while writing very little extra code. For example, a quad AES system with 4 parallel AES engines, with a UVM testbench and virtual sequences, Verilog memory and VHDL memory could be visualized as

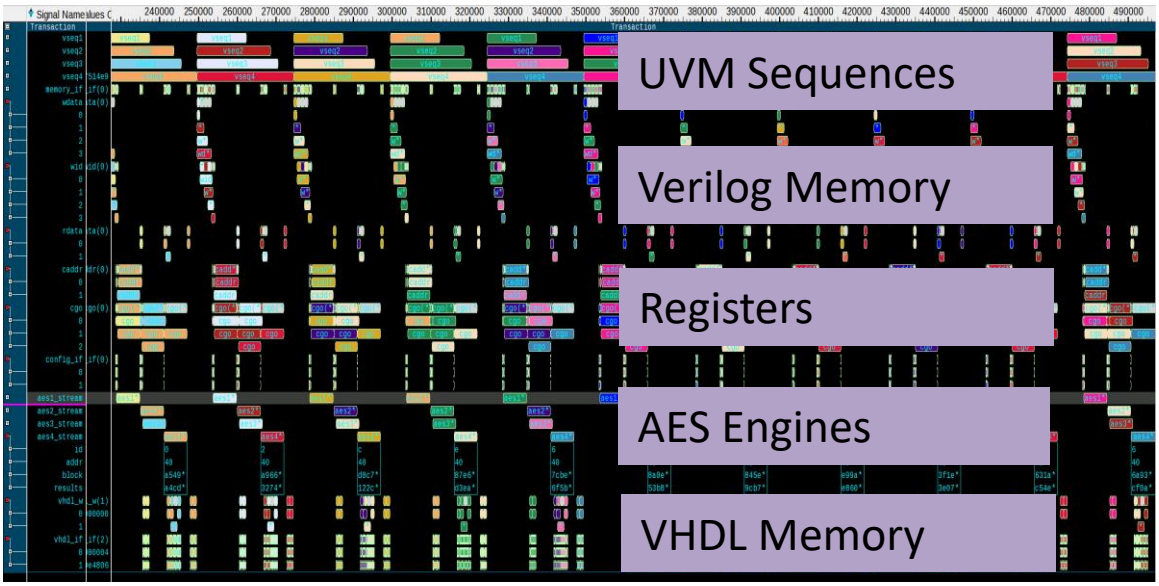


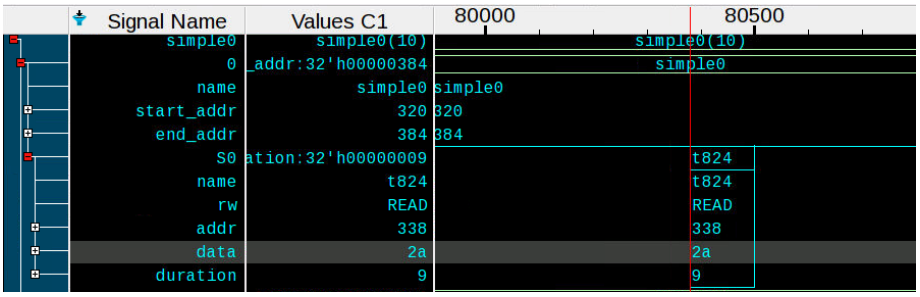
Figure 1 - System Transactions for quad AES System

II. TRANSACTION OVERVIEW

A. *Using Built-in UVM Transaction Recording*

The UVM has a built-in transaction recording mechanism for UVM sequences and UVM sequence items. When a sequence is started, a “begin transaction” occurs. When a sequence ends, an “end transaction” occurs. This automation is very handy to record transactions with very little effort. Unfortunately the transactions recorded are

sub-optimal. Nevertheless, they are still useful and should be considered. They define the UVM transactions for stimulus generation using sequences and sequence items.



In order to record the sequence attributes, the function do_record is implemented in the sequence by the user. Do_record defines which attributes of the class should be recorded as attributes. The do_record below uses the preferred "type-less" macro `uvm_record_field. This macro allows SystemVerilog to figure the actual type and record it automatically[6].

```
class simple_sequence extends uvm_sequence#(transaction);
  `uvm_object_utils(simple_sequence)

  transaction t;
  bit [31:0] start_addr;
  bit [31:0] end_addr;

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("start_addr", start_addr)
    `uvm_record_field("end_addr", end_addr)
  endfunction
```

Figure 2 - Sequence Transaction Recording

In this testbench, the sequence attributes control the sequences items attributes that get generated. In order to record the sequence item attributes, the function do_record is implemented in the sequence item.

```
class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

  rand rw_t       rw;
  rand reg [31:0] addr;
  reg      [31:0] data;

  rand int duration;

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("rw", rw.name())
    `uvm_record_field("addr", addr)
    `uvm_record_field("data", data)
    `uvm_record_field("duration", duration)
  endfunction
endclass
```

Figure 3 - Sequence Item Transaction Recording

The automation provided in the UVM is for sequences and sequence items. Many other places in the testbench can benefit from transaction recording, including drivers, monitors and scoreboards. This is one of the drawbacks of the automation – despite the automation, some of the most interesting transactions are not automated.

B. Using a Transaction Recording API

The Veriog PLI or function call Transaction Recording API was first introduced in 2003[5]. It is designed to be simple, small and lightweight. It can be deployed in UVM, SystemVerilog, Verilog, VHDL and other languages. Any code can be instrumented with these API calls, not just UVM code. The API is small, consisting of 8 entry

points, two of which are almost never used (add_relation, delete_transaction). This is a small, lightweight API, but it is vendor specific. Conveniently, over the years each vendor has implemented relatively the same API. The standard proposal from 2003 was never adopted as a standard, but rather it has become a standard – at least a recommendation for the vendors.

Verilog	**VHDL**
$add_attribute(trHandle, value, attribute_name)	add_attribute(trHandle, value, attribute_name)
$add_color(trHandle, color)	add_color(trHandle, color)
$add_relation(trHandle, trHandle, relation_name)	add_relation(trHandle, trHandle, relation_name)
$begin_transaction(stream, trName, begin_time, parent_tr)	begin_transaction(stream, trName, begin_time, parent_tr)
$create_transaction_stream(stream_name, stream_kind)	create_transaction_stream(stream_name, stream_kind)
$delete_transaction(trHandle)	delete_transaction(trHandle)
$end_transaction(trHandle)	end_transaction(trHandle)
$free_transaction(trHandle)	free_transaction(trHandle)

C. A UVM Monitor

A UVM monitor is instrumented below with the transaction recording API. The monitor starts a thread in the run_phase() task which never ends. It first creates the "STREAM" for the transactions, then loops, trying to recognize (monitor) transactions. When the READY/VALID signals indicate a transaction, then a "BEGIN TRANSACTION" occurs. "ATTRIBUTES" are added to the transaction, and finally an "END TRANSACTION" and "FREE TRANSACTION" occurs.

```
class monitor_WITH_TR extends uvm_monitor;
  `uvm_component_utils(monitor_WITH_TR)

  virtual my_if mif;
  transaction t;
  uvm_analysis_port #(transaction) ap;

  int s, tr;

  function void build_phase(uvm_phase phase);
    ap = new("ap", this);
  endfunction

  task run_phase(uvm_phase phase);
    s = $create_transaction_stream("txn_stream");
    forever begin
      @(posedge mif.CLK);
      if ((mif.VALID == 1) && (mif.READY == 1)) begin
        t = transaction::type_id::create("t");
        if (mif.rw == READ) begin
          tr = $begin_transaction(s, "READ");
          t.rw = mif.rw;
          t.addr = mif.addr;
          @(negedge mif.READY);
          t.data = mif.rd;
          $add_attribute(tr, t.addr, "addr");
          $add_attribute(tr, t.data, "data");
          $end_transaction(tr);
          $free_transaction(tr);
        end
        else if (mif.rw == WRITE) begin
          tr = $begin_transaction(s, "WRITE");
          t.rw = mif.rw;
          t.addr = mif.addr;
          t.data = mif.wd;
          $add_attribute(tr, t.addr, "addr");
          $add_attribute(tr, t.data, "data");
```

```
            $end_transaction(tr);
            $free_transaction(tr);
          end
          `uvm_info(get_type_name(), $sformatf("Got %s", t.convert2string()), UVM_MEDIUM)
          ap.write(t);
        end
      end
    endtask
  endclass
```

Figure 4 – Transaction Recording in a UVM Monitor

The results of this kind of API-driven recording result in a transaction viewing and debug experience that is far superior to the UVM recording API, as can be seen below. In the view below, each transaction is colored according to the "ID" attribute. Transactions with IDs of the same value will be the same color. This coloring algorithm is useful to easily track transactions through switches or across the testbench infrastructure.

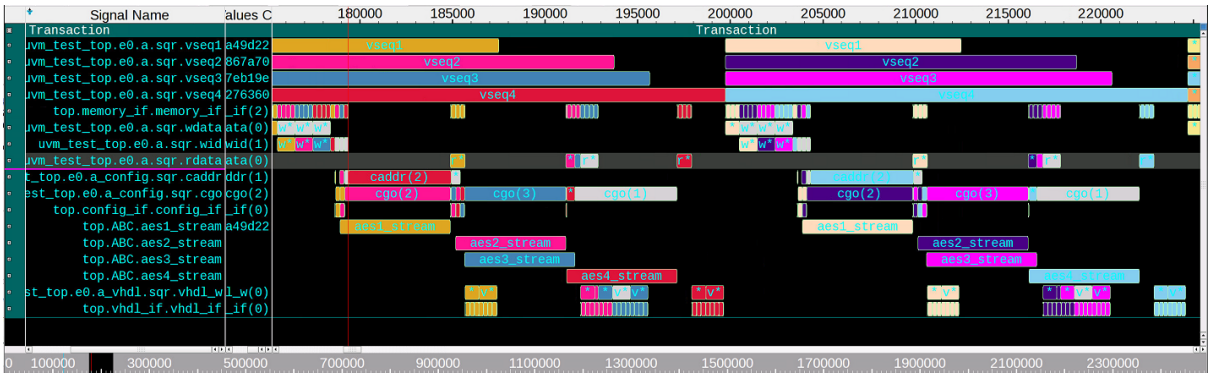


Figure 5 - Transaction Debug View with ID coloring algorithm

III. USING SYSTEMVERILOG TO MODEL TRANSACTIONS

In this modeling style, a Verilog module or interface is written to act as a monitor. This monitor can identify transactions. The monitor will be bound into a Verilog or VHDL DUT or Verilog or VHDL bus structure.

A. UVM Publisher and Subscriber

This use model is a separation of concerns. A monitor as in Section II recognizes the transaction and creates a class object which represents the transaction – but it does NOT do any transaction recording. The created class object (representing a transaction) is "published" to any "subscriber" for further processing. Commonly used subscribers are coverage subscribers and transaction recording subscribers. Instead of instrumenting the monitor with transaction recording code, a subscriber can be written to do the actual recording from the "abstract" class that is published from the monitor using ap.write(t). The reader is encouraged to investigate ap.write(t) and how UVM subscribers and publishers operate. The code above in Figure 4 is such a publisher.

B. Bind module - Bound monitor with direct recording using SystemVerilog $calls

In this modeling style, a transaction recorder of some kind is bound into the RTL using the 'bind' command. The transaction recorder could be Verilog or VHDL. The bind command is a powerful way to add instances to existing code, without changing the actual RTL code.

dut.sv

```
module my_dut(input CLK, output bit READY,  input bit VALID,  ...,
                         output bit READY2, input bit VALID2, ...);
  ...
endmodule
```

The dut defined above has two READY/VALID channels. Below, a monitor will be bound into the DUT, connected to each channel.

tb_top.sv

```
module TB_TOP();

  bind dut MONITOR CHANNEL1(CLK, READY,  VALID,  ...);
  bind dut MONITOR CHANNEL2(CLK, READY2, VALID2, ...);

  DUT dut(CLK,
     READY,  VALID,  ...,
     READY, VALID, ...);
  ...
endmodule
```

The top module defined above instantiates the DUT and two monitors. The monitors are of type MONITOR and are named CHANNEL1 and CHANNEL2 inside the 'dut' instance.

monitor.sv

```
module MONITOR(input CLK, input bit READY, input bit VALID, ...);
  int s;
  int tr;

  initial
    s = $create_transaction_stream("s", "kind");

  always @(posedge CLK) begin
    if ((READY==1) && (VALID==1)) begin
      tr = $begin_transaction(s, rw.name());
      $add_attribute(tr, rw.name(), "rw");
      $add_attribute(tr, addr, "addr");
      if (rw == READ) begin
        $add_attribute(tr, rd, "rd");
      end
      else if (rw == WRITE) begin
        $add_attribute(tr, wd, "wd");
      end
      while ((READY==1) && (VALID==1))
        @(posedge CLK);
      @(negedge CLK);
      $end_transaction(tr);
      $free_transaction(tr);
    end
  end
endmodule
```

The monitor code above is a simple monitor module which creates a stream. Then as READY/VALID define a transfer, a transaction gets recorded.

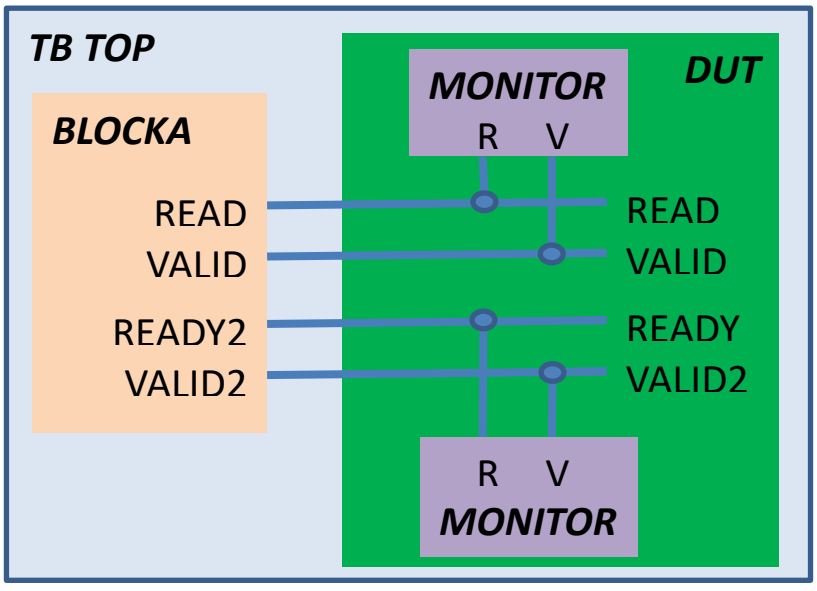


Figure 6 - The block diagram with the bound instances

The block diagram above illustrates the two monitors instantiated inside the DUT. Creating a small monitor, and binding it into the RTL can give high level visibility into internal transfers and behavior.

IV. USING VHDL TO MODEL TRANSACTIONS

In this example, a VHDL instance is created which records transactions. It is a simple example of using the VHDL API, including the parent transaction link, creating a kind of parent-child relation.

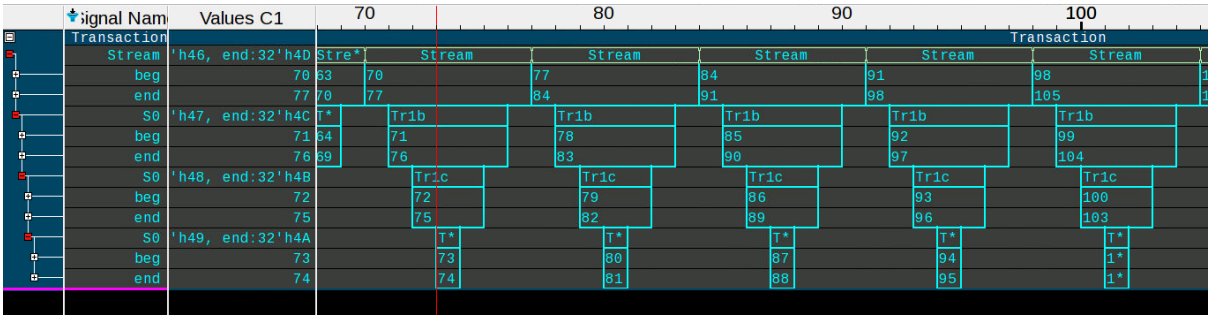


Figure 7 - VHDL Simple Example with Overlapping Transactions

```
entity top is
end;

library IEEE;
    use IEEE.std_logic_1164.all;

library modelsim_lib;
    use modelsim_lib.transactions.all;

architecture arch of top is
begin
    process
        variable stream : TrStream := create_transaction_stream("Stream");
        variable tr1a: TrTransaction := 0;
        variable tr1b: TrTransaction := 0;
        variable tr1c: TrTransaction := 0;
        variable tr1d: TrTransaction := 0;
        variable i : integer := 0;
    begin
        tr1a := begin_transaction(stream,"TR1a");
        add_attribute(tr1a, i, "beg");
        ...
        wait for 1 ns;
        tr1b := begin_transaction(stream,"Tr1b", tr1a);
        add_attribute(tr1b, i, "beg");
        ...
        wait for 1 ns;
        tr1c := begin_transaction(stream,"Tr1c", tr1b);
        add_attribute(tr1c, i, "beg");
        ...
        wait for 1 ns;
        tr1d := begin_transaction(stream,"Tr1d", tr1c);
        add_attribute(tr1d, i, "beg");
        ...
        wait for 1 ns;
        add_attribute(tr1d, i, "end");
        end_transaction(tr1d, true);
        ...
        wait for 1 ns;
        add_attribute(tr1c, i, "end");
        end_transaction(tr1c, true);
        ...
        wait for 1 ns;
        add_attribute(tr1b, i, "end");
        end_transaction(tr1b, true);
        ...
```

```
        wait for 1 ns;
        add_attribute(tr1a, i, "end");
        end_transaction(tr1a, true);
    end process;
end;
```

Figure 8 - VHDL Example

V. INTEGRATING VHDL, UVVM AND TRANSACTION RECORDING

In this modeling style, the third party library, UVVM will have transaction recording integrated. This example instrumented the bisvis_irqc example in the distribution. A more full integration with the internal UVVM library functions and the UVVM VIP is beyond the scope of this paper, but is a natural extension.

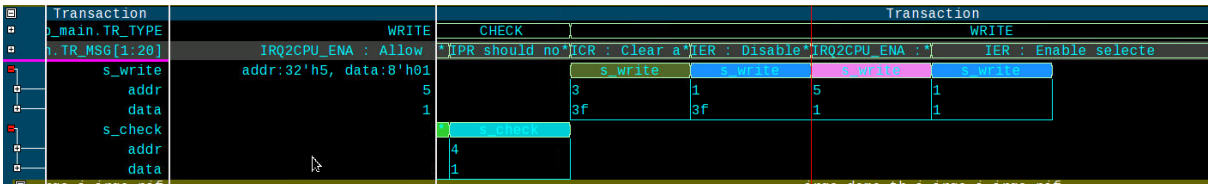


Figure 9 - Transactions from bitvis_irqc test

The example bitvis_irqc code has a VHDL testbench which calls check() and write(). Below, check() and write() get instrumented with transaction recording API calls.

The partial testbench is

```
    check(C_ADDR_IRR,          fit(x"AB"), ERROR, "IRR should now be active");
    check(C_ADDR_IPR,          fit(x"01"), ERROR, "IPR should now be active");
    write(C_ADDR_ICR,          fit(x"FF"),        "ICR : Clear all interrupts");
    write(C_ADDR_IER,          fit(x"FF"),        "IER : Disable all interrupts");
    write(C_ADDR_IRQ2CPU_ENA,      x"01",         "IRQ2CPU_ENA : Allow interrupt to CPU");
    write(C_ADDR_IER,          v_irq_mask,        "IER : Enable selected interrupt");
```

Figure 10 - Source code for bitvis_irqc test recorded above

In the instrumented code, two streams are created – write and check. This is a modeling decision. Using one stream would also be appropriate if desired. Using two streams in this case clearly shows the two sets of activity.

Once the streams are created the handles are used in begin_transaction and end_transaction. A transaction handle is used to manage the attributes and end the transaction.

```
    p_main: process

      variable stream  : TrStream := create_transaction_stream("s_write");
      variable streamc : TrStream := create_transaction_stream("s_check");
      variable tr  : TrTransaction := 0;
      variable trc : TrTransaction := 0;
...
      -- Overloads for PIF BFMs for SBI (Simple Bus Interface)
      procedure write(
        constant addr_value   : in natural;
        constant data_value   : in std_logic_vector;
        constant msg          : in string) is
      begin
        tr := begin_transaction(stream, "WRITE");
        add_attribute(tr, addr value, "addr");
        add_attribute(tr, data_value, "data");

        sbi_write(to_unsigned(addr_value, sbi_if.addr'length), data_value, msg,
              clk, sbi_if, C_SCOPE);

        end_transaction(tr, true);
      end;

      procedure check(
        constant addr_value   : in natural;
        constant data_exp     : in std_logic_vector;
        constant alert_level  : in t_alert_level;
```

```
    constant msg          : in string) is
begin
  trc := begin_transaction(streamc, "CHECK");
  add_attribute(trc, addr_value, "addr");
  add_attribute(trc, data_exp, "data");

  sbi_check(to_unsigned(addr_value, sbi_if.addr'length), data_exp, msg,
        clk, sbi_if, alert_level, C_SCOPE);

  end_transaction(trc, true);
end;
```

Figure 11 - bitvis_irqc/tb/irqc_demo_tb.vhd

VI. TRANSACTION MODELING BEST PRACTICES

Finally, Verification IP developers, testbench developers or even design engineers may be interested in using any of these techniques to instrument their libraries, testbenches of block level testing with transaction recording.

A. Basic Transaction Modeling

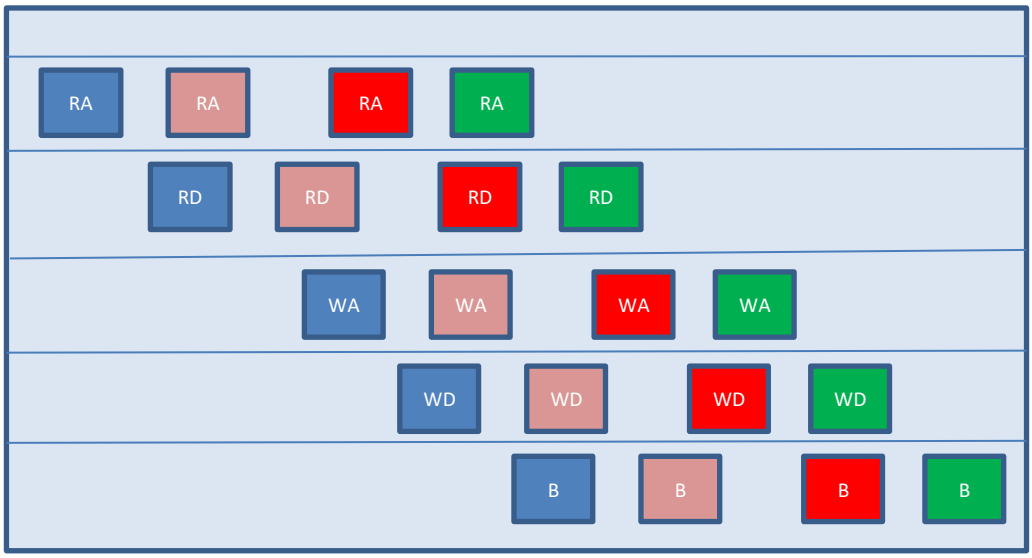
The basic transaction model used has certain characteristics and usages. A transaction has a beginning and an end. A transaction can have name-value pairs (“addr”, “16”) called attributes. The most useful transactions are not zero time transactions. A useful transaction should consume time.

Transactions exist “on” a stream. A stream is simply a container or owner for transactions used as a modeling construct. It may or may not be related to a physical object. For example, a stream could be modeling a bus, in which case it might have RA, RD, WA, RD and B transactions. The stream might be modeling the channels of this bus, in which case there would be five streams – one for each channel. The RA (Read Address) transactions would all occur only on the RA channel. Yet another modeling style for stream between using 1 or N might be a modeling style where the READ transactions are on the READ stream and the WRITE transactions are on the WRITE stream. Figure 12 below illustrates one stream, two streams (READ/WRITE) and five streams (RA, RD, WA, WD and B).

Figure 12 - Modeling transactions using streams

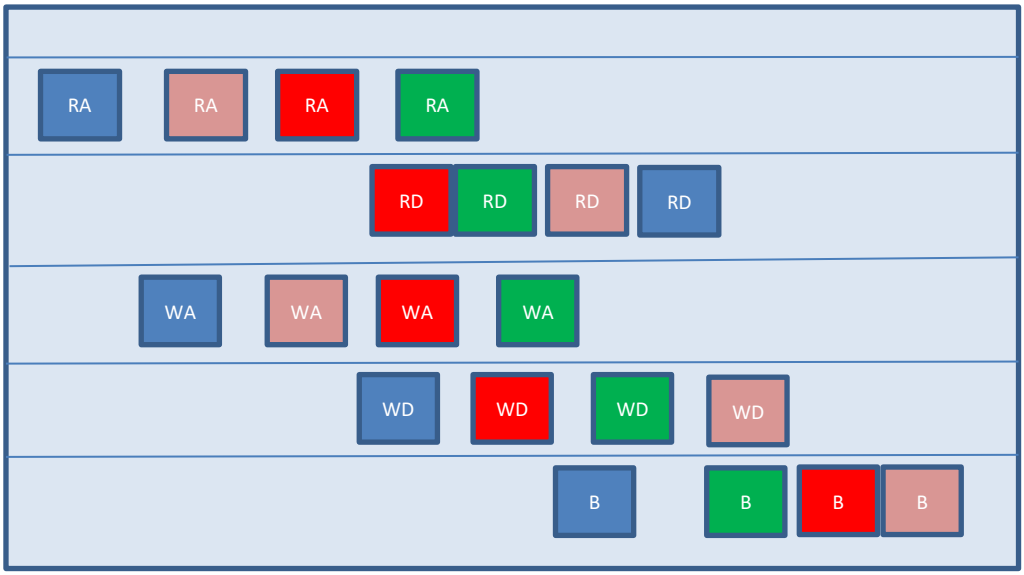
B. *Overlapping Transaction Modeling*

Overlapping transactions pose a modeling issue. The details of overlapping transactions are beyond the scope of this paper. But an interesting characteristic of the underlying stream based model is that it will properly record and display overlapping transactions without any guidance from the user. This is handy, since a given channel (or stream) may not know ahead of time how many transaction will be in-flight at a time.



C. *Out-of-order Transaction Modeling*

Out-of-order transactions pose yet more complexity about managing responses and requests. Out-of-order response management is beyond the scope of this paper. But just as the system adjusts automatically for overlapping transactions, it also adjusts automatically for out-of-order transactions. No matter what order or how many are in flight at once, all the transaction get recorded and displayed without any hints from the API or the user.



VII. CONCLUSION

Verification productivity requires doing more with less. Transactions are an easy way to see "the big picture" with very little effort or code required. This paper has explained and demonstrated multiple ways to record transactions, including the UVM built-in recording, a PLI API for Verilog, a function call API for VHDL, binding, instrumenting and subscriber/publisher.

It is the hope of the author that the reader will adopt some level of transaction recording – perhaps recording transactions representing test snippets and then using add_color() to color the self-checking failures RED. Perhaps loading system memory and initiating system tests by running "compiled code" – annotating the execution states with transactions. Perhaps simply annotating the Verification IP activity to make a black box transparent for debug.

Small changes such as these have helped many others get more visibility into the flow of data across tests and structure; and in the process, improve verification productivity.

VIII. REFERENCES

[1] SystemVerilog LRM, "1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language", https://ieeexplore.ieee.org/document/8299595

[2] UVM 1.1d Reference Implementation, https://www.accellera.org/images/downloads/standards/uvm/uvm-1.1d.tar.gz

[3] VHDL LRM, "1076-2008 - IEEE Standard VHDL Language Reference Manual", https://ieeexplore.ieee.org/document/5981354

[4] UVVM – Universal VHDL Verification Methodology, https://bitvis.no/dev-tools/uvvm/ and https://github.com/UVVM/UVVM

[5] Transaction Recording API Proposal, "Draft Standard for Verilog Transaction Recording Extensions", http://www.boyd.com/1364_btf/report/full_pr/attach/435_IEEE_TR_Proposal_04.pdf

[6] "Transaction Recording Anywhere Anytime", DVCON US 2019, Rich Edelman, https://www.mentor.com/products/fv/resources/overview/transaction-recording-anywhere-anytime-2ef2d0f6-7d85-42f7-881c-87eefa683fe1