2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

SystemVerilog Transactions, UVM and C

Correlation in a functional verification environment

Rich Edelman, Siemens EDA Fremont, CA US (rich.edelman@siemens.com)

Tomoki Watanabe, Siemens Japan, Tokyo, Japan (tomoki.watanabe@siemens.com)

Abstract— **Transaction level modeling and transaction level debug have been in use for years in SystemVerilog and Verilog simulation and verification, but not as available in VHDL, perhaps not used in GLS simulation and C testbenches, and taking new forms in system level modeling. This paper re-introduces and refreshes transaction recording and debug and suggests how each abstraction level can be used productively with worked examples runnable by the reader.**

Keywords— SystemVerilog, Verilog, VHDL, Transaction recording, Transaction debug, C testbench, bind

I. Introduction

As designers build larger and larger chips, it becomes more and more important for verification engineers to think about verification at higher level abstractions. Transactions can be recorded from low level details like clock edges and bus transfers all the way up the abstraction tree to system level interactions and tracking behavioral models in C or other high-level languages. This is important because most of this kind of debug needs to correlate two things. Perhaps two sets of transactions – one from the high level and one from the low level. Perhaps what needs correlated is the low-level signal changes with the top-level abstraction activity. Perhaps the entire abstraction tree must be correlated with itself – to understand who is causing what events and transactions.

The waveform window in a traditional functional design and verification debugger is ideal for this kind of correlation – time-based correlation. The signals and transactions occupy the same timeline and can be reasoned about in the wave window. New views can also be created. Often the actual timing of the transaction is not important, but rather whether a transaction completes before another – the relative position of starting and ending transactions.

This paper will review the various APIs and methods for transaction recording and demonstrate the concepts using an example. That example can be reused in reader code and is open source.

II. Transaction Recording Basics

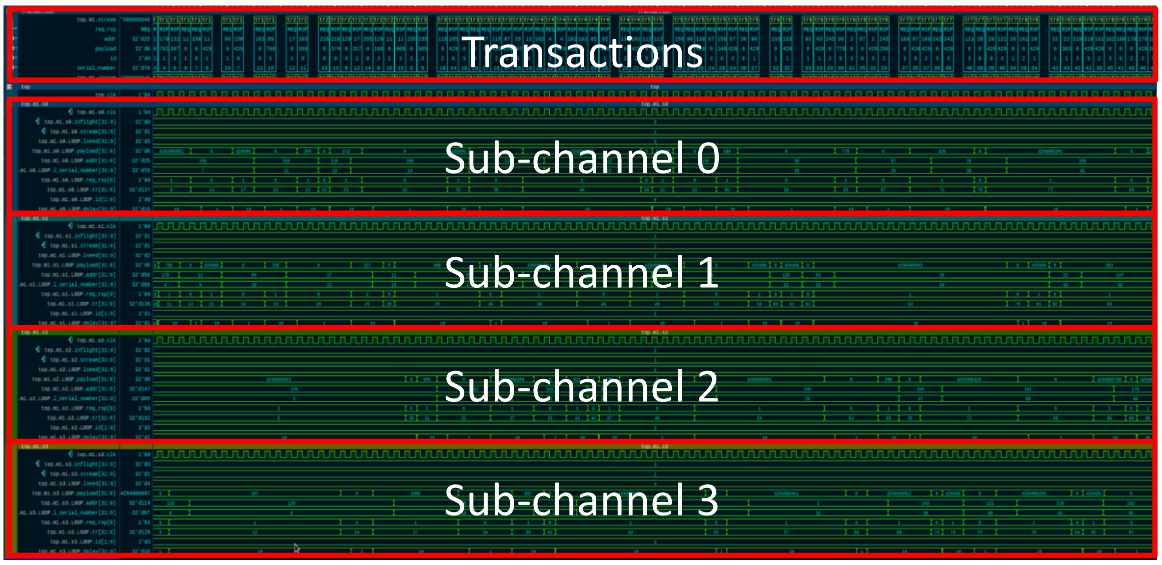
A transaction recording interface is commonly available, including creating a transaction stream, beginning and ending a transaction, adding attributes, adding relationships and signal activity.

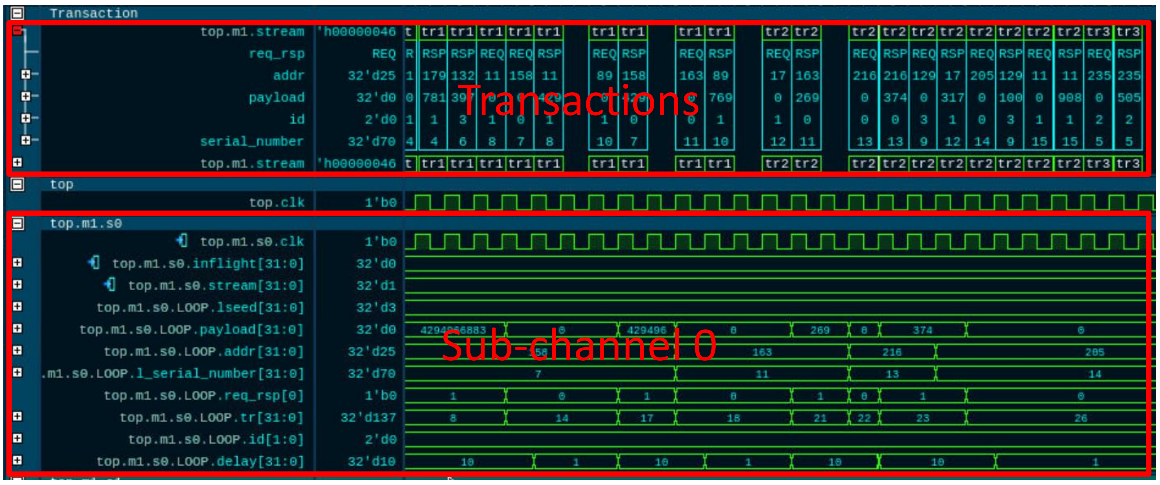
- $create_transaction_stream
- $begin_transaction
- $end_transaction
- $free_transaction
- $add_attribute

An example using such an interface is below. This example does nothing useful, except demonstrate using the API to model a channel with 4 out-of-order transactions in-flight at once.

```
module M(input clk);
  int stream;

  initial begin
    stream = $create_transaction_stream("stream", "kind");
  end

  sub_channel s0(clk, 0, stream);
  sub_channel s1(clk, 1, stream);
  sub_channel s2(clk, 2, stream);
  sub_channel s3(clk, 3, stream);
endmodule
```

```
module sub_channel(input clk, input int inflight, int stream);
  always @(posedge clk) begin: LOOP
    ...
    begin
      req_rsp = REQ;
      tr = $begin_transaction(stream, $sformatf("tr%0d", tr_count));
      $add_attribute(tr, req_rsp, "req_rsp");
      $add_attribute(tr, addr, "addr");
      payload = 0;
      $add_attribute(tr, payload, "payload");
      $add_attribute(tr, id, "id");
      $add_attribute(tr, l_serial_number, "serial_number");
      #10;
      $end_transaction(tr);
      $free_transaction(tr);
    end
    ...
    begin
      req_rsp = RSP;
      tr = $begin_transaction(stream, $sformatf("tr%0d", tr_count));
      $add_attribute(tr, req_rsp, "req_rsp");
      $add_attribute(tr, addr, "addr");
      payload = $random() % 1024; // Fetch the payload
      $add_attribute(tr, payload, "payload");
      $add_attribute(tr, id, "id");
      $add_attribute(tr, l_serial_number, "serial_number");
      delay = 10;
      #delay;
      #10;
      $end_transaction(tr);
      $free_transaction(tr);
    end
  end
endmodule
```





III. Building Your Own Monitor

This section builds on previous work on building monitors. [2] Monitors can be build in many different ways. They need to have a start and an end, attributes and something to “trigger” the start. For example a clock signal, a value change or a condition like “ready & valid”.

A. Building a Verilog based monitor to record transactions.

A Verilog based monitor could be implemented as a module or an interface. The job of the monitor is to recognize a transaction. A monitor usually monitors a lower level abstraction like signal changes on wires and turns those into higher level abstractions – transactions. The monitor can be connected to the “wires” either as a bind, or simply connected to the wires by instantiation in the correct place. Bind is quite convenient, since no existing RTL code needs to change.

The register and memory monitors in following sections are examples of Verilog based monitors.

B. Building a UVM based monitor to record transactions.

A UVM based monitor is normally implemented as a class which is attached to an interface. It could also be an interface (with its wires). This is analogous to the Verilog based monitor attached to wires. A UVM based monitor might produce a “transaction class” and send it to other “subscribers”. Please consult UVM instructions for more information on subscribers in the UVM. For this paper, a monitor is going to record transactions.

Given signals in an interface

```
  reg valid;
  reg ready;
  reg rw;
  reg [3:0] addr;
  reg [3:0] wdata;
  reg [3:0] rdata;
```

A monitor could be written using a forever thread. This thread is in an interface, but could also be in a class. The code below is a simple monitor which prints messages.

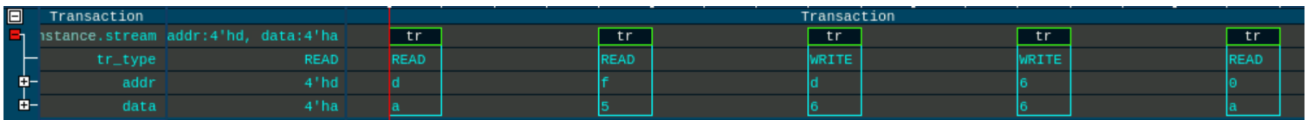
```
forever begin
  @(posedge clk);
  if ((ready == 1) && (valid == 1)) begin
    @(posedge clk);
    if (rw == READ) // READ
      $display("MONITOR: %m: READ  addr=%0d, data=%0d", addr, wdata);
    else // WRITE
      $display("MONITOR: %m: WRITE addr=%0d, data=%0d", addr, rdata);
  end
end
```

Resulting in output like

```
# MONITOR: top.memory_interface_instance.monitor: READ  addr=0, data=1
# MONITOR: top.memory_interface_instance.monitor: READ  addr=13, data=14
# MONITOR: top.memory_interface_instance.monitor: READ  addr=10, data=11
# MONITOR: top.memory_interface_instance.monitor: READ  addr=7, data=8
# MONITOR: top.memory_interface_instance.monitor: READ  addr=4, data=5
# MONITOR: top.memory_interface_instance.monitor: READ  addr=13, data=14
# MONITOR: top.memory_interface_instance.monitor: READ  addr=14, data=15
# MONITOR: top.memory_interface_instance.monitor: READ  addr=15, data=0
# MONITOR: top.memory_interface_instance.monitor: WRITE addr=2, data=10
# MONITOR: top.memory_interface_instance.monitor: WRITE addr=3, data=10
# MONITOR: top.memory_interface_instance.monitor: WRITE addr=0, data=10
```

But with transaction recording added, there is much more information available and it can be viewed in the wave window. The monitor is extended to record transactions:

```
  int stream;
  int tr;

...
    string tr_type;
    stream = $create_transaction_stream("stream", "kind");
    forever begin
      @(posedge clk);
      if ((ready == 1) && (valid == 1)) begin
        @(posedge clk);
        tr = $begin_transaction(stream, "tr");
        if (rw == READ) begin // READ
          $display("MONITOR: %m: READ  addr=%0d, data=%0d", addr, rdata);
          tr_type = "READ";
          $add_attribute(tr, tr_type, "tr_type");
          $add_attribute(tr, addr, "addr");
          $add_attribute(tr, rdata, "data");
        end
        else begin // WRITE
          $display("MONITOR: %m: WRITE addr=%0d, data=%0d", addr, wdata);
          tr_type = "WRITE";
          $add_attribute(tr, tr_type, "tr_type");
          $add_attribute(tr, addr, "addr");
          $add_attribute(tr, wdata, "data");
        end
        @(negedge clk);
        $end_transaction(tr);
        $free_transaction(tr);
      end
    end
```



IV. UVM Transaction Recording

Transaction recording is woven through the UVM. There's automation available which makes it very easy to generate transactions. A transaction or sequence should either be instrumented with do_record () method or the field automation macro method should be used. Most cautious users choose the do_record method. It offers more flexibility and ease of debug, while requiring slightly more code to be written.

The UVM transaction recording is automatic. In addition to "regular" UVM Transaction Recording, this paper will discuss the use of the UVM backdoor access, and how to instrument those access functions for transaction recording.

A. *Instrumenting a transaction*

Implement the do_record() below to record transactions. See the UVM user guide for the field automation method.

```
int gid;
int gserial_number;

class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

  bit  [2:0] id; // 0 to 7
  bit [31:0] serial_number;

  rand int delay;

  rand RW_T rw;
  rand bit [31:0] addr;
  rand bit [31:0] data;

  constraint values {
    addr > 0; addr < 100;
    data >= 0; data < 8;
    delay > 3; delay < 10;
  }

  function new(string name = "transaction");
    super.new(name);
    id = gid++;
    serial_number = gserial_number++;
  endfunction

  function string convert2string();
    return $sformatf("id: %0d %s(%0d, %0d) #%0d", id, rw.name(), addr, data, serial_number);
  endfunction

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

B. *Backdoor Access*

The UVM has built-in calls to access the DUT directly – so called backdoor access. The advantage of backdoor access is its speed. A backdoor access is fast – no bus cycles are used. Almost no simulation. The backdoor access can be a read of a write of values directly from the RTL.

The sequence below generates front-door reads and writes. Then it issues two sets of "writes" to three different registers.

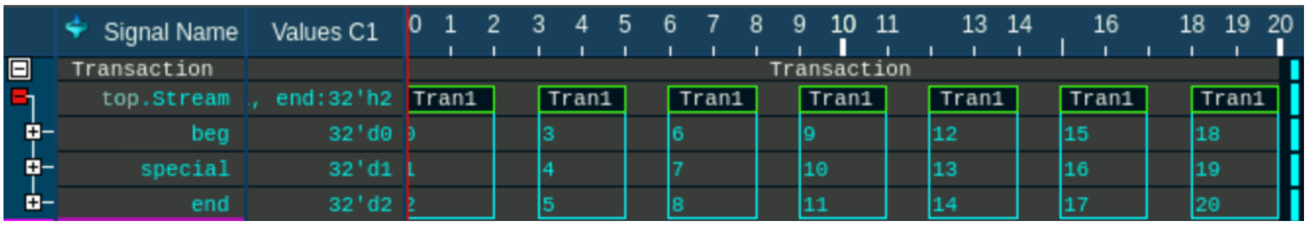
```
class seq extends uvm_sequence#(transaction);
  `uvm_object_utils(seq)

  function new(string name = "seq");
    super.new(name);
  endfunction

  transaction t;

  task body();
    string name;
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)

    for (int i = 0; i < 1000; i++) begin
      name = $sformatf("t%0d", i);
      t = transaction::type_id::create(name);
      start_item(t);
      `uvm_info(t.get_type_name(), "...started transaction", UVM_MEDIUM)
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize Failed")
      finish_item(t);
      `uvm_info(t.get_type_name(), "...finished transaction", UVM_MEDIUM)

    end
      uvm_hdl_deposit("top.memory_instance.regA", 1);
      uvm_hdl_deposit("top.memory_instance.regB", 2);
      uvm_hdl_deposit("top.memory_instance.regC", 3);
      #10;
      uvm_hdl_deposit("top.memory_instance.regA", 2);
      uvm_hdl_deposit("top.memory_instance.regB", 4);
      uvm_hdl_deposit("top.memory_instance.regC", 6);
  endtask
endclass
```

V. VHDL Transaction Recording

VHDL can also log transactions. The PLI routines for Verilog are a library of calls to use. VHDL has a similar mechanism. There is a library of calls. A simple VHDL example is below [3]

```
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 tr: TrTransaction := 0;

        variable i : integer := 0;
    begin
        tr := begin_transaction(stream,"Tran1");
        add_attribute(tr, i, "beg");
        i := i + 1;
        wait for 1 ns;
        add_attribute(tr, i, "special");
        i := i + 1;
        wait for 1 ns;
        add_attribute(tr, i, "end");
        end_transaction(tr);
        free_transaction(tr);
        i := i + 1;
        wait for 1 ns;
    end process;

end;
```

Produces simple transactions as below



VI. Transaction Recording in a Gate Level Simulation

UVM and RTL code is edited and adjusted as verification continues. Adding transaction recording is sometimes “just better debug”. In a gate-level netlist, normally changes are not allowed – the gate-level netlist has been verified in some way. Editing the code is often forbidden. Below a simple module MAJ is defined. And a simple MAJ_monitor is built. Any time the ‘d’ signal – the output from MAJ – changes, a transaction will be recorded.

The MAJ_monitor is “bound” into the MAJ module using a bind statement.

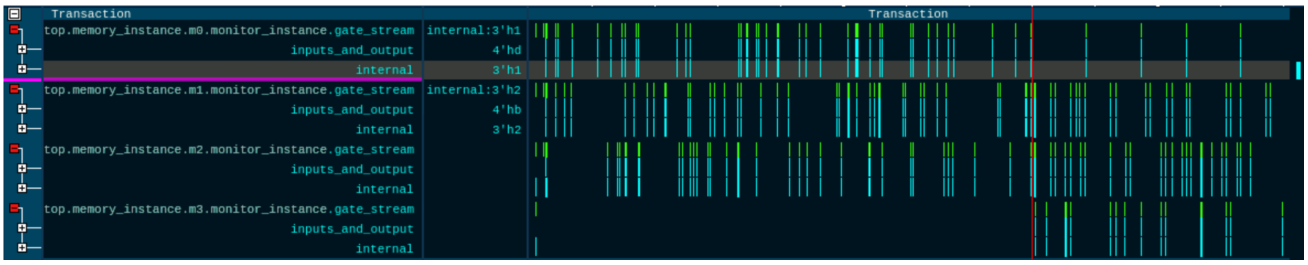
```
module bind_module();
  bind MAJ MAJ_monitor monitor_instance(d, a, b, c, ab, bc, ac);
endmodule
```

The bind statement above creates an instance named monitor_instance in every instance of MAJ, and the bound in module type is MAJ_monitor.

Maj_monitor and MAJ defined

```
module MAJ_monitor(input reg d, a, b, c, ab, bc, ac);
  int stream;
  int tr;

  initial begin
    stream = $create_transaction_stream("gate_stream", "kind");
  end

  always @(d) begin
    tr = $begin_transaction(stream, "MAJ_tr");
    $add_attribute(tr, {d, a, b, c}, "inputs_and_output");
    $add_attribute(tr, {ab, bc, ac}, "internal");
    #5;
    $end_transaction(tr);
    $free_transaction(tr);
  end
endmodule

module MAJ(output d, input a, b, c);
  and #4 ab_and(ab, a, b);
  and #4 bc_and(bc, b, c);
  and #4 ac_and(ac, a, c);
  or  #4   d_or(d , ab, bc, ac);
endmodule
```

Notice that all the monitors create a stream, often in an initial block. Then a thread or process repeats based on a trigger, recording a transaction at each “trigger event”. And we deftly avoid zero delay transactions. The #5 statement above prevents a zero delay transaction. The 5 unit delay is a contrivance – and a convenience for the debugger.



```
Transaction                                                                      Transaction
top.memory_instance.m0.monitor_instance.gate_stream  internal:3'h1
                                  inputs_and_output           4'hd
                                           internal           3'h1
top.memory_instance.m1.monitor_instance.gate_stream  internal:3'h2
                                  inputs_and_output           4'hb
                                           internal           3'h2
top.memory_instance.m2.monitor_instance.gate_stream
                                  inputs_and_output
                                           internal
top.memory_instance.m3.monitor_instance.gate_stream
                                  inputs_and_output
                                           internal
```

VII. Transaction Recording from C

Tests and models can be written in other languages – including C. SystemVerilog provides a very convenient way of interfacing C with UVM and RTL – the DPI-C interface. Please see the UVM instructions for more information about DPI-C. Think of the use of DPI below as the simulator providing a C-callable interface that a C test program or more can call.

The DPI layer below has wrapped the PLI transaction recording interface for export to a C application.

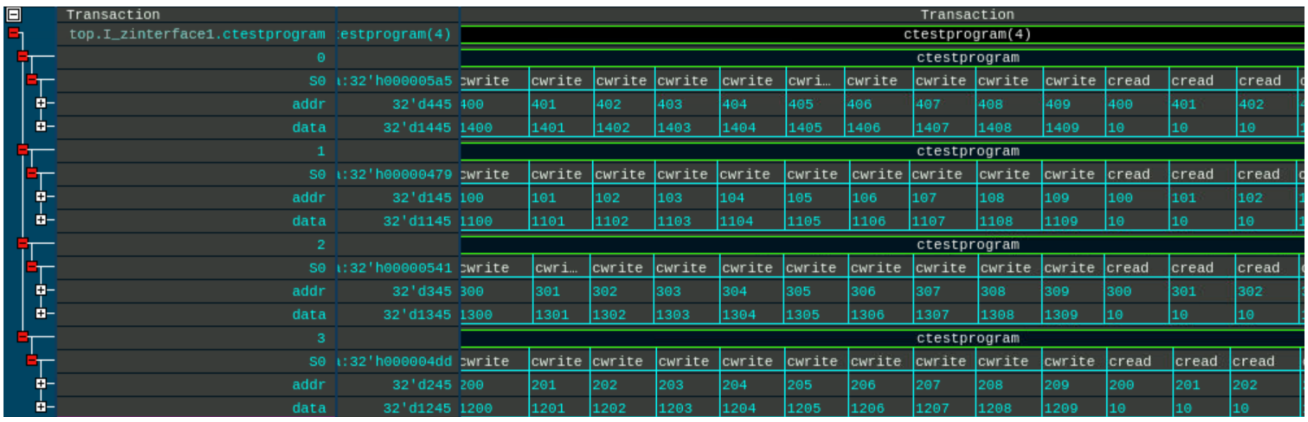
DPI Layer

```
  export "DPI-C" function create_transaction_stream;
  export "DPI-C" function begin_transaction;
  export "DPI-C" function end_transaction;
  export "DPI-C" function free_transaction;
  export "DPI-C" function add_attribute_int;

  function int create_transaction_stream(string name, string kind);
    int stream_handle;
    stream_handle = $create_transaction_stream(name, kind);
    return stream_handle;
  endfunction

  function int begin_transaction(int stream_handle, string name, int begin_time,
        int parent_handle);
    int tr_handle;
    if (parent_handle == -1)
      tr_handle = $begin_transaction(stream_handle, name);
    else
      tr_handle = $begin_transaction(stream_handle, name, , parent_handle);
    return tr_handle;
  endfunction

  function void end_transaction(int tr_handle);
    $end_transaction(tr_handle);
  endfunction

  function void free_transaction(int tr_handle);
    $free_transaction(tr_handle);
  endfunction

  function int add_attribute_int(int tr_handle, int value, string attribute_name);
    $add_attribute(tr_handle, value, attribute_name);
  endfunction
```

The C testprogram

Notice the cwrite and cread transactions. They wrap the call to 'write()' and 'read()' respectively. The write() and read() calls are calling into the C interface offered by the UVM sequences in this case. The transaction logging is in C – it could also have been in the UVM sequence task based layer.

```
testprogram.c
    int stream_handle;
    int transaction_handle;
    int read_transaction_handle;
    int write_transaction_handle;

    original_start_addr = start_addr;

    stream_handle = create_transaction_stream("ctestprogram", "kind");
    transaction_handle = begin_transaction(stream_handle, "ctestprogram", 0, -1);

    // Repeat 10 times, changing the data
    for (dataloops = 0; dataloops < 10; dataloops++) {
      // Repeat 10 times - writing 10, and reading 10
      start_addr = original_start_addr;
      for (loops = 0; loops < 10; loops++) {
        printf("C: ...entering  start_test_program1 (%0d...) <%s>\n", index, name);
        for (addr = start_addr; addr < start_addr+10; addr++) {
          data = addr + 1000 + dataloops;
          write_transaction_handle = begin_transaction(stream_handle, "cwrite",
            0, transaction_handle);
          write(index, addr, data);
          add_attribute_int(write_transaction_handle, addr, "addr");
          add_attribute_int(write_transaction_handle, data, "data");
          end_transaction(write_transaction_handle);
          free_transaction(write_transaction_handle);
          printf("C: ...executed WRITE(%0d, %0d) <%s>\n", addr, data, name);
        }

        for (addr = start_addr; addr < start_addr+10; addr++) {
          read_transaction_handle = begin_transaction(stream_handle, "cread",
            0, transaction_handle);
          read(index, addr, &data);
          add_attribute_int(read_transaction_handle, addr, "addr");
          add_attribute_int(read_transaction_handle, data, "data");
          end_transaction(read_transaction_handle);
          free_transaction(read_transaction_handle);

          printf("C: ...executed READ (%0d, %0d) <%s>\n", addr, data, name);
          if (data != addr + 1000 + dataloops) {
            printf("C: ...ERROR  READ (%0d, %0d) <%s> [wrote: %d, read %d]  \n",
              addr, data, name, data, addr + 1000 + dataloops);
          }
        }

        start_addr += 10;
      }
    }
    end_transaction(transaction_handle);
```



Transaction		Transaction												
top.I_zinterface1.ctestprogram	:estprogram(4)	ctestprogram(4)												
0		ctestprogram												
S0	ı:32'h000005a5	:write	cwrite	cwrite	cwrite	cwrite	cwri…	cwrite	cwrite	cwrite	cwrite	cread	cread	cread
addr	32'd445	400	401	402	403	404	405	406	407	408	409	400	401	402
data	32'd1445	1400	1401	1402	1403	1404	1405	1406	1407	1408	1409	10	10	10
1		ctestprogram												
S0	ı:32'h00000479	:write	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cread	cread	cread
addr	32'd145	100	101	102	103	104	105	106	107	108	109	100	101	102
data	32'd1145	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	10	10	10
2		ctestprogram												
S0	ı:32'h00000541	:write	cwri…	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cread	cread	cread
addr	32'd345	300	301	302	303	304	305	306	307	308	309	300	301	302
data	32'd1345	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	10	10	10
3		ctestprogram												
S0	ı:32'h000004dd	:write	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cwrite	cread	cread	cread
addr	32'd245	200	201	202	203	204	205	206	207	208	209	200	201	202
data	32'd1245	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	10	10	10

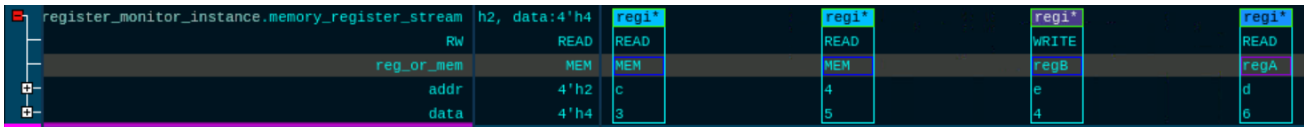
VIII. Transaction Recording for Registers

Registers are bit vectors with special addresses or special characteristics. The REG_monitor below is very simple – it accepts a register value 'r' and a register name. Anytime 'r' changes, it logs a transactions. But this is perhaps too simple. It only logs write transactions.

```
module REG_monitor(input reg [3:0] r, string register_name);
  int stream;
  int tr;

  initial begin
    stream = $create_transaction_stream("reg_stream", "kind");
  end

  always @(r) begin
    bit [3:0] reg_value;
    reg_value = r;
    tr = $begin_transaction(stream, "REG_tr");
    $add_attribute(tr, register_name, "NAME");
    $add_attribute(tr, reg_value, "value");
    #5;
    $end_transaction(tr);
    $free_transaction(tr);
  end
endmodule
```

This simple monitor is bound into the memory.

```
module bind_module_register();
  bind memory REG_monitor reg_monitor_instanceA(regA, "regA");
  bind memory REG_monitor reg_monitor_instanceB(regB, "regB");
  bind memory REG_monitor reg_monitor_instanceC(regC, "regC");
endmodule
```



IX. Transaction Recording for Memories

As the monitor is extended to memories, the monitor is enhanced to be much smarter, and mimic the actual protocol. This monitor is monitoring all the communication on the wires – on the bus. It will record register reads and writes, as well as memory reads and writes.

```
module memory_register_monitor(input clk, input valid, input ready, input rw,
    input bit [3:0] addr, input [3:0] wdata, input [3:0] rdata);

  rw_t rw_as_enum;
  reg [3:0] data;
  string rw_name;
  string name;
  int stream;
  int tr;

  initial begin
    stream = $create_transaction_stream("memory_register_stream", "kind");
  end

  always @(posedge clk) begin
    rw_as_enum = rw_t'(rw);
    if ((ready == 1) && (valid ==1 )) begin
      tr = $begin_transaction(stream, "register");
      case (rw)
      READ: begin
        rw_name = "READ";
        data = rdata;
        case (addr)
        REGA: name = "regA";
        REGB: name = "regB";
        REGC: name = "regC";
        default: name = "MEM";
        endcase
      end
      WRITE: begin
        rw_name = "WRITE";
        data = wdata;
        case (addr)
        REGA: name = "regA";
        REGB: name = "regB";
        REGC: name = "regC";
        default: name = "MEM";
        endcase
      end
      endcase
      $add_attribute(tr, rw_name, "RW");
      $add_attribute(tr, name, "reg_or_mem");
      $add_attribute(tr, addr, "addr");
      $add_attribute(tr, data, "data");
      @(negedge clk);
      $end_transaction(tr);
      $free_transaction(tr);
    end
  end
endmodule
```

This memory_register monitor was instantiated directly in the memory. It could have been bound in, or connected to wires or implemented in an interface. In fact any of the monitors discussed above could have any or these implementation variations.

X. CONCLUSION

The reader of this paper has the background and examples to support their own development of transaction recording and correlation at many abstraction levels. The example is open source, available from the author, and the concepts apply to any of the simulation suppliers, while the details vary slightly.

Using transactions in a verification environment and being able to correlate the results from any abstraction level will allow better verification and increased productivity. Transactions can be used to improve visibility and transparency in a large, complex functional verification environment. The reader will understand how to adapt the concepts and techniques from this paper to their verification environment.

XI. REFERENCES

[1] DVCON Japan 2024, *"Having Your Cake and Eating It Too - Programming UVM Sequences with DPI-C"*, Rich Edelman, [Contact author for a copy]

[2] DVCON US 2013, *"Monitors, Monitors Everywhere – Who is Monitoring the Monitors? SystemVerilog UVM Monitors and Scoreboards"*, Rich Edelman, Raghu Ardeishar

[3] Questa Example – vhdl/transactions/simple, Questa 2025.2