Panning for Gold in RTL Using Transactions

Rich Edelman
Mentor Graphics
rich_edelman@mentor.com

Raghu Ardeishar
Mentor Graphics
raghu_ardeishar@mentor.com

Akshay Sarup
Mentor Graphics
akshay_sarup@mentor.com

Suman Kasam
Qualcomm
skasam@qualcomm.com

ABSTRACT
This paper explains multi-level transaction level monitoring, scoreboarding and coverage collection for existing RTL designs. By applying the ideas in this paper, the reader will understand how to achieve higher level verification on reused or lower level design components. Simple transaction verification is not our only goal. Most systems have a great number of combinations of transactions or sub-transactions. The main contribution of this paper is the generation of those detailed transactions using intelligent monitors, making the transactions visible for other higher level verification components.

Categories and Subject Descriptors
[Hardware Verification]: Functional Simulation and Verification – *class based SystemVerilog, OVM library, UVM library.*

General Terms
Verification, Simulation, Transactions, Verification IP.

Keywords
MVC, VIP, Verification IP, Bind, Transactions.

1. INTRODUCTION
This paper proposes a TLM fabric that is overlaid on the RTL. The fabric is a verification fabric that can be connected to the lower level RTL. The TLM fabric recognizes transactions in the design. This recognition must be complete – a TLM fabric must be able to monitor all the possible transactions, including sub-transactions (like phases) or burst transactions.

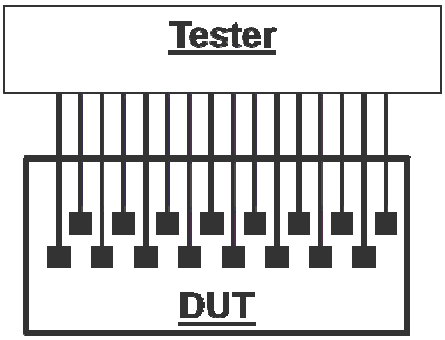

Figure 1 - Bed-of-Nails Testing

The fabric is a collection of smart probes that are bound into the RTL just as a tester might probe internal DUT signals [Figure 1]. When the fabric probes the RTL the signals are made available to monitors that transform the pin level signals into higher level abstractions such as transactions. These monitors are abstraction shifters – they shift from pin wiggles and signals to transactions.

In addition, monitors may be able to create multi-level transactions. These multi-level transactions are transactions which describe the sub-parts of the same transactions – a transfer transaction might be made up of two other phase transactions. A packet transaction might consist of multiple bursts, which consist of multiple transfers, which consist of two phases each.

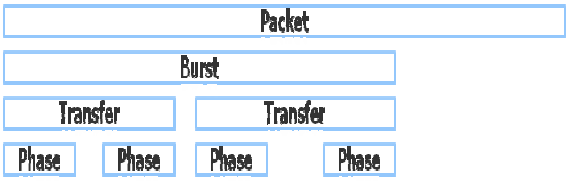

Figure 2 - Multi-level transactions

The multi-level transactions are often hard to observe or rebuild from externally visible signals, but they are available as internal states, or internal signal changes. These internal states and internal signals are the gold which this paper will mine.

Each of the transaction levels are important to understand as part of a complete verification environment. Unfortunately, the detailed monitoring of multi-level transactions and their related internal signals and their transformation can be hard to accomplish. This paper will describe some techniques to make monitoring, scoreboarding and coverage of multi-level transactions easier, but since this area of verification is very design specific, there will always be a heavy requirement of planning and coding to achieve the verification goals.

2. ENABLING TRANSACTIONS
The TLM fabric – the bed-of-nails[5] – will allow new coverage, new model checking and debug. It will do this by making internal states and signals visible. Furthermore, it will transform the internal states into higher level transactions. Exposing the internal states and signals also is a valuable verification tool, but can lead to too much information. Transforming multiple states, state transitions or pin wiggles into transactions allows the information to be captured and communicated throughout the verification environment efficiently.

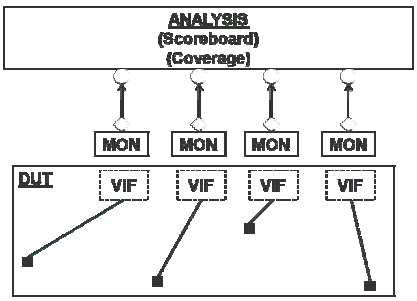

Figure 3 - TLM Fabric

Once a design has been enabled with this TLM fabric, new functionality can be added to the testbench, including coverage of pairs of transactions – like two long burst transactions on two different RTL interfaces. Debugging is also easier since the actual transactions are being recognized and are available.

The fabric consists of verification components - agents which work to recognize the transactions and perform tasks such as coverage collection or scoreboarding. The overlay of the TLM fabric is achieved by "binding" the RTL signals out to a transaction level monitor. The advantage of this approach is that it can be applied at run-time to any part of the design which adheres to the interface protocol.

3. USING VERIFICATION COMPONENTS

Verification components are commonly used to encapsulate an interface or device. A verification component might be used as an AHB [11] master. It knows how to issue stimulus and to respond to AHB related signals. A verification component (agent) contains a monitor which knows how to recognize pin wiggles and generate transactions. It contains a driver, a sequencer, a library of sequences, and other things needed to manage the interface or device [Figure 4].

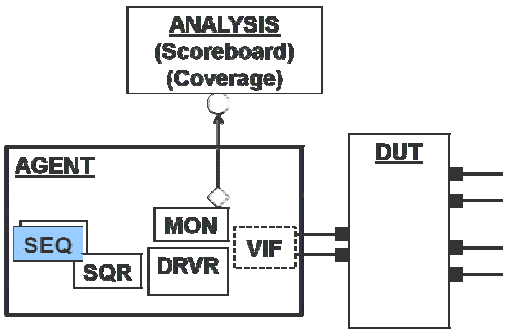

Figure 4 - Verification Component

The TLM fabric will only use the monitoring side of verification components – it is a passive monitor of internal states and signals. The driver, sequencer and sequences will not be used.

A verification component may be connected to the RTL using signals and wires, or may use a SystemVerilog interface construct. An interface construct is common in UVM verification environments, and will be assumed for the verification in this paper.

A SystemVerilog interface is a collection of related signals – like the signals on a bus. It has many properties and capabilities defined in the SystemVerilog LRM [1]. This paper will only consider the interface as a collection of signals.

Verification components are built assuming access to a SystemVerilog interface. Part of building the verification environment is to create an interface, and make it available for the verification component. This interface instantiation and connection to a verification component is assumed by this paper.

4. ANALYSIS COMPONENTS

Analysis components are simple – they perform some kind of analysis. In this paper, analysis components will have any number of analysis_exports. Each of these analysis_exports will be connected to an analysis_port, and specific transactions will be published on these analysis_ports by monitors.

Analysis traffic is generated from monitors, and published to subscribers. Analysis traffic is generated instantaneously – once an analysis transaction is generated it is transferred to an analysis component without consuming any time.

An analysis component can perform any kind of analysis from golden model checking to vector compare to transaction compare to functional coverage collection.

The analysis component can copy the transaction into storage for later processing – as in the case of out-of-order transaction comparison.

The transactions are specific to the functionality of what is being monitored. For example an AHB monitor might be publishing AHB phases, AHB transfers and AHB bursts. Each of these transaction types will be connected to different analysis_exports on analysis components.

5. MONITORS AND INTERFACES

A monitor's job is to monitor. It can monitor many things. It can be simple or complex. A monitor needs to know what to monitor (a value, or state), when to monitor (on posedge clk) and what to do with the monitored data (send a transaction out the analysis port).

Interfaces and bind have been used in many ways [8][9][10]. This paper describes an additional usage – binding interfaces, and providing the bound interfaces to verification component monitors.

5.1 Module-based Monitor

A monitor could be a simple module, connected to the signals to be monitored. In this case the monitor prints the current 'value' on the positive edge of the clock.

```
module monitor(input bit clk,
   input bit[7:0] value);
  always @(posedge clk)
    $display(
      "Value = %0d (%m)", value);
endmodule
```

It has the necessary inputs – at least one piece of data to be monitored – 'value', and a trigger – when to monitor – 'clk'. Unfortunately, while this monitor does a good job monitoring, it is just monitoring a value, and only prints it. If a series of values made up a transaction, it would be nice to collect the values together, create the transaction and send it elsewhere to be processed. This kind of monitor will be introduced below.

To complete our simple module-based monitoring, we construct the testbench 'top' below, and have a piece of hardware – 'computing_element' that needs to be monitored.

```
module top();
  computing_element ce(clk);

  bind ce monitor mon1(clk1, sum);
endmodule

module computing_element(input clk1);
  bit [7:0] sum = 0;

  always @(posedge clk1)
    sum += 1;
endmodule
```

The computing_element 8 bit 'sum' is an internal value. It is not visible to the testbench or the outside world. In order to monitor that value we use the SystemVerilog bind command to instantiate the monitor module within the RTL module being verified (computing_element).

In the bind statement above, the module named 'monitor' is instantiated into the instance named 'ce'. Furthermore, the signal 'clk1' local to the instance of 'ce' is connected to the first port of monitor, and the signal 'sum', local to the instance of 'ce' is connected to the second port of monitor.

This monitor needs to publish a transaction to interested subscribers. An assortment of techniques are available, including direct connection, config setting, and the new resource database setting.

This module based monitoring is provided as background, but is not the subject of this paper.

5.2 Interface-based Monitor

Monitoring internal signals can also be achieved by binding a SystemVerilog interface into existing RTL. This interface is the exact same interface that was used to do block level testing, but now that the system is being tested, the block level interface is no longer externally visible. It is internal, hidden. Using bind, an interface can be instantiated just as the monitor module was above.

```
interface abc_if(input wire clk,
  input wire[7:0]data);
  ...
endinterface
```

The abc_if interface above has two signals shown – the clock and an 8 bit data. The checker below, part of the abc_agent, is a class with a run() task and a virtual interface handle to an abc_if interface. The checker monitors an abc_if, and for each negative or positive edge, it performs some job.

The checker in Figure 5 is connected to the block named Block X and is part of the block level verification plan for that block.

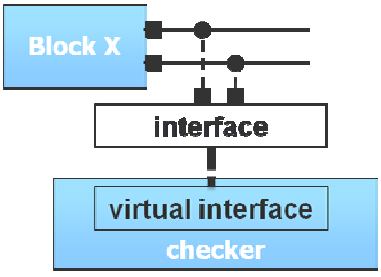

Figure 5 - Block checking using a virtual interface

The skeleton for the checker (a monitor) is below. This checker needs a virtual interface of type abc_if, and then it can do its checking job (monitoring).

```
class checker extends uvm_component;
  virtual interface abc_if vif;
  ...
  task run();
    fork
      forever @(negedge vif.clk) begin
        ...
      end
      forever @(posedge vif.clk) begin
        ...
      end
    join
  endtask
endclass
```

In Figure 6, BlockX from above is combined with BlockY, and reused inside other blocks, SOC A and SOC B, the checker code no longer has direct access to the interface signals that it is checking.

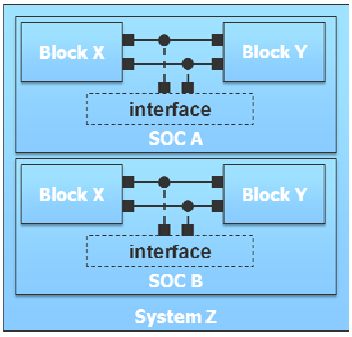

Figure 6 - System checking using a virtual interface

With a simple bind statement an instance of the checker interface can be bound into the lower level SOC A and SOC B blocks. Once bound, the interface becomes available for a checker to connect to.

6. A TRANSACTION LEVEL MONITOR

A transaction level monitor monitors interesting behavior, and creates transactions which get published to interested subscribers. The kinds of interesting behavior monitored includes pin wiggles, counter values, internal states, registers, state-machine current states, state-machine transitions and even other transactions.

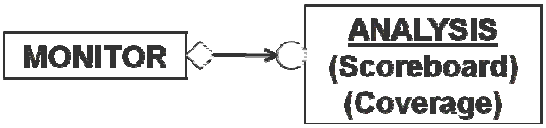

Figure 7 - Monitor and Analysis Component

The monitor will publish the transaction using an analysis port, and the analysis component will subscribe using an analysis export. The publisher/subscriber interface ensures that transactions are delivered without delays (they are transported as function calls, as opposed to task calls).

The monitor recognizes a transaction. Once a transaction has been recognized, a call to new() or the factory creates a transaction. The newly created transaction fields are filled in. Finally, the filled in transaction is published by calling 'ap.write(t)'.

The simple protocol in Figure 8 could be monitored with the code below it. This is a trivial protocol, and a simplistic monitor. Most

real protocols are quite complicated, and most real monitors are quite sophisticated. Writing correct monitors is quite difficult, contributing to a collection of widely available monitors from a variety of specialists. Many times verification teams choose to use multiple monitors from different sources, just to ensure that no behavior is missed.

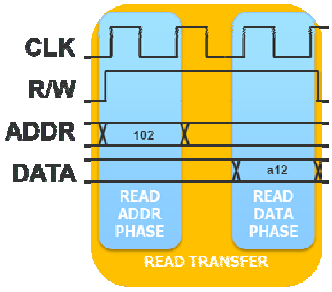

Figure 8 - Simple Read Transfer

This simple protocol is a read transfer made up of two sub-phases, a read addresss phase and a read data phase. The monitor below only publishes a transaction when the read transfer is complete. A more complete monitor would also have the ability to publish the phase transactions are well, since an analysis component might be interested to know that a phase had occurred.

```
class monitor extends uvm_component;
  virtual interface simple_bus vif;
  uvm_analysis_port #(my_transaction) ap;

  function new(string name,
    uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build();
    ap = new("ap", this);
  endfunction

  task run();
    my_transaction t;
    forever @(posedge vif.clk) begin
      t = my_transaction::type_id
        ::create("t");
      t.addr = vif.addr;
      if (vif.idle) begin
        t.rw = IDLE;
        t.data = vif.data;
        t.addr = vif.addr;
      end
      else if (vif.rw) begin
        t.rw = READ;
        @(posedge vif.clk);
        t.data = vif.data;
      end
      else begin
        t.rw = WRITE;
        t.data = vif.data;
      end
      ap.write(t);
    end
  endtask
endclass
```

7. TRANSACTION LEVEL FUNCTIONAL COVERAGE

7.1 Simultaneous Reads

In the figure below, a memory is shared by two different sub-systems. One of the verification goals is to make sure that a large read occurs from BlockX and BlockW simultaneously to the same memory location. This kind of design functionality is exactly what functional coverage in SystemVerilog is for. Functional coverage is the coverage of design specific functionality. Code coverage is an automated coverage where each executable line is counted when it is executed. Functional coverage is different – it must be specified by the user.

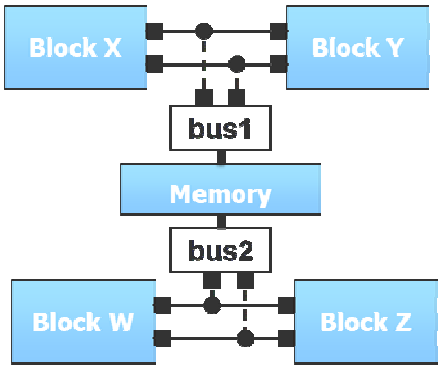

Figure 9 - Memory traffic on two competing busses

In addition to writing the functional coverage, the user must find a way to retrieve the things to be covered – in this case a large packet read transaction from overlapping addresses on two different interfaces.

Functional coverage is one of the most interesting uses for transaction level monitors.

7.2 Data transformation

In Figure 10, when a packet is transmitted on the input of a block a transaction is created. It is the input transaction, and is sent to the coverage unit. As the packet is processed in the RTL, an intermediate packet may be created, and produced. That intermediate packet can be sent to a coverage collector – it is the "generated" result.

In the figure below, BlockX reads via interface 1 and writes via interface 2. BlockX performs a data transformation, for example AES encryption or data compression.

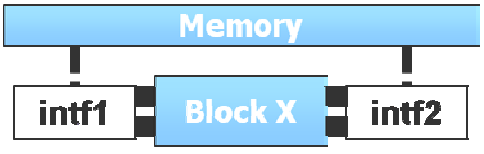

Figure 10 - BlockX transformation

8. TRANSACTION LEVEL SCOREBOARD

Scoreboards check to see if the behavior as observed is what is expected. They check the experimental behavior against a known good behavior.

Transaction based scoreboards are no different. They are built to accept transactions and to check behavior or legality or validity.

8.1 Simple Scoreboard

A simple scoreboard is implemented below which is a uvm_subscriber. The write() routine is called when any connected analysis port publishes a new transaction. The analysis port in our case will be part of the monitor – the monitor recognizes a transaction, constructs a transaction class, and publishes it to its subscribers.

This example subscriber is simple. It just echoes the received transaction to the standard output.

```
class simple_analysis_component
  extends uvm_subscriber#(my_transaction);
  `uvm_component_utils(
    simple_analysis_component)
  ...
  function void write(my_transaction t);
    `uvm_info("SIMPLE",
      $psprintf("Analyzing '%s'.",
        t.convert2string()),
      UVM_INFO)
  endfunction
endclass
```

A more interesting scoreboard might accept a transaction from a monitor, and calculate if it is a legal transaction. For example by calculating the checksum on a collection of data, then comparing with the checksum received in the transaction.

8.2 In-order Scoreboard

A more complex scoreboard might have two sets of transaction streams, an expected stream and an actual stream. Each pair of transactions are compared as they are generated. This is "in-order" comparison.

Below is the in-order comparator for a simple transaction type. This analysis component – a scoreboard – has two analysis exports, and two fifos. When a transaction is available – either on A or B, the write_A() or write_B() function is executed. In each case the transaction passed in is saved into a fifo for later processing. This "front-end" to the scoreboard has infinite length fifos. As transactions appear they are saved "in-order".

The scoreboard has a run() task which pops one transaction from each fifo and does a compare. A more complete scoreboard would check the end of simulation for items still in the fifo. This would indicate that some matching transaction was never produced, or that too many transactions were produced. In either case it is a problem for the verification team.

```
`uvm_analysis_imp_decl(_A)
`uvm_analysis_imp_decl(_B)

class inorder_analysis_component
    extends uvm_component;
  ...
  // Declare the analysis exports that
  // the monitor will publish to.
  uvm_analysis_imp_A#(...) analysis_export_A;
  uvm_analysis_imp_B#(...) analysis_export_B;

  // Fifos to hold the in-order data.
  uvm_tlm_fifo #(my_transaction) fifo_A;
  uvm_tlm_fifo #(my_transaction) fifo_B;
  ...

  function void build();
    fifo_A = new("fifo_A", this, 0);
    fifo_B = new("fifo_B", this, 0);
    analysis_export_A = new("...", this);
    analysis_export_B = new("...", this);
  endfunction

  // When the A analysis_export is written,
  // this write() routine will be called.
  function void write_A(my_transaction t);
    void'(fifo_A.try_put(t));
  endfunction

  // When the B analysis_export is written,
  // this write() routine will be called.
  function void write_B(my_transaction t);
    void'(fifo_B.try_put(t));
  endfunction

  function void compare(my_transaction A, B);
    ...
  endfunction

  task run();
    my_transaction A, B;
    forever begin
      fifo_A.get(A);
      fifo_B.get(B);
      compare(A, B);
    end
  endtask
endclass
```

8.3 Out-of-order Scoreboard

Another kind of complex scoreboard also accepts expected and actual streams, but the transactions are not expected to be in order. This is "out-of-order" comparison. In this case, either the expected or actual transactions would be posted to a list of transactions. These posted transactions must be matched within a window, otherwise the scoreboard will flag the posted transaction as either missed (an expected transaction occurred, but no matching actual) or extra (an actual transaction occurred, but no matching expected). The out-of-order comparator is an exercise for the reader.

9. TRANSACTION LEVEL DEBUG

Using a simple subscriber, as mentioned above, a transaction can be captured. Once captured in the write() routine of the subscriber, many things can be done with it. It can be printed to the standard output. It can be formatted and printed into a logfile. It can be formatted and printed into a "replay-file". It can be recorded into a transaction viewing database using a recording API.

The standard output and logfile solutions are useful debug techniques. The replay-file is a file that is formatted so that it can be read later as either stimulus or as expected behavior. The recording API can be implemented by a user, or a vendor solution can be used.

In all of these cases, once the transaction is captured by the subscriber, it is logged or saved for future debug or golden model checking.

10. BINDING IT ALL TOGETHER

Monitors have been built that monitor interfaces, and analysis components have been built that check functionality or collect coverage.

In order to gain access to the internal signals we are interested in monitoring, we must bind the interface inside the RTL, and register that bound interface – make it available for the monitor.

```
module top();
  bit fast_clk, clk, reset;

  dut dutA(clk,      reset);
  dut dutB(fast_clk, reset);

  env e = new("env", null);

  // Create the probes.
  // Bind a virtual interface into each
  // dut. Name the instances dutA.dut_if
  // and dutB.dut_if. Connect the signal
  // 'clk' and 'counter', as referenced
  // from within the dut.
  bind dutA bus_interface dut_if(
    dut_clk, counter);
  bind dutB bus_interface dut_if(
    dut_clk, counter);

  initial begin
    // Put the probes into a database.
    uvm_resource_db #(
      virtual interface bus_interface)
        ::set("A", "vbus_if", dutA.dut_if);
    uvm_resource_db #(
      virtual interface bus_interface)
        ::set("B", "vbus_if", dutB.dut_if);

    run_test();
  end
```

In the code snippet above, there are two RTL duts (dutA and dutB) and a testbench. The testbench, 'e', will contain two monitors, and perform comparison on the results – one result from dutA and one result from dutB. There are two interfaces created with the two bind statements – dutA.dut_if and dutB.dut_if.

Those interfaces are bound into the RTL, and available for futher use by our testbench. In this example the bound (instanced) interfaces are registered with the resource database [4][7]. This database is really a fancy global name lookup for typed data. We're using it in the most simple way possible. We register a name "A", a variable name "vbus_if" and a value – the virtual interface.

The testbench connect() function, below, is built to lookup the A and B interfaces. The testbench environment code uses the resource 'read_by_name" functionality to retrieve the virtual interface. Once retrieved the environment passes the virtual interface to the monitors – one for A and one for B.

```
function void env::connect();
  // The 'A' Side.
  uvm_resource_db#(
    virtual interface bus_interface)
      ::read_by_name("A", "vbus_if",
        vbus_if_A);
  monA.vbus_if = vbus_if_A;
  ...

  // The 'B' Side.
  uvm_resource_db#(
    virtual interface bus_interface)
      ::read_by_name("B", "vbus_if",
        vbus_if_B);
  monB.vbus_if = vbus_if_B;
  ...
endfunction
```

11. CONCLUSION

When a block is verified using UVM Verification methods, and a verification component is used with a SystemVerilog interface, that verification component can be reused as a monitor when the block is reused in a higher level block or system.

This reuse is accomplished by using 'bind' to bind an interface into the block as it exists as lower-level RTL in a system. Once the interface is bound, the verification component can monitor the interface – just as a bed-of-nails tester might have done.

Once these monitors are in place, the monitors can generate transactions as they observe the internal operations of the reused block. These internal transactions can be checked for correctness, can be used as debugging aids, and can have coverage collected.

Observing these transactions together with similarly published transactions from other blocks in the system can provide a birds-eye view of the system operation, and can allow easy checking of corner cases like – "did we ever check that the bus transfer between BlockA and B works right after Block C has powered up". Observing these transactions together with each other offers the verification engineer greater insight and comfort from his verification environment.

12. REFERENCES

[1] SystemVerilog LRM. www.accellera.org
[2] OVM World Download, www.ovmworld.org
[3] OVM User Guide.
[4] UVM User Guide. (unpublished)
[5] Conrad, James, Mysore, G. and Newberry, B., "A Microcontroller-Based Bed-of-Nails Test Fixture to Program and Test Small Printed Circuit Boards", SouthEast Con, 2006.
[6] OVM Reference Guide.
[7] UVM Reference Guide. (unpublished)
[8] Rich, Dave and Bromley, Jonathan, "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches", DVCON 2008
[9] Baird, Michael, "Coverage Driven Verification of an Unmodified DUT within an OVM Testbench", DVCON 2010
[10] Mallez, Virginie, Peryer, Mark, et. al., "Are You in a "bind" with Advanced Verification?", Verification Horizons, February 2009.
[11] AHB Specifcation. www.arm.com

APPENDIX 1 – INTERNAL RTL MONITORS AND SCOREBOARDS

```
//=======================================
import uvm_pkg::*;
`include "uvm_macros.svh"

// =======================================
// My_transaction
// Simple transaction class with 'data'
// and a timestamp, t.
class my_transaction extends
     uvm_transaction;
  `uvm_object_utils(my_transaction)

  bit [31:0] data;
  time t;
  int id;
  static int g_id = 0;

  function new(string name =
      "my_transaction");
    super.new(name);
    id = g_id++;
  endfunction

  function string convert2string();
    return $psprintf(
      "(id=%0d, data=%0x, @%0t)",
        id, data, t);
  endfunction
endclass

// =======================================
// General Analysis Component.
// Listen for transactions. Re-implement
// write() do to something more
// interesting with them.
class simple_analysis_component
  extends uvm_subscriber#(my_transaction);
  `uvm_component_utils(
    simple_analysis_component)

  function new(string name =
    "simple_analysis_component",
    uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void write(my_transaction t);
    `uvm_info("SIMPLE",
      $psprintf("Analyzing '%s'.",
        t.convert2string()),
      UVM_INFO)
  endfunction
endclass

`uvm_analysis_imp_decl(_A)
`uvm_analysis_imp_decl(_B)
```

```
// =======================================
// INORDER COMPARE
// Simple comparison of transactions
// generated on two streams, in-order.
class inorder_analysis_component
    extends uvm_component;
  `uvm_component_utils(
    inorder_analysis_component)

  // Declare the analysis exports that
  // the monitor will publish to.
  uvm_analysis_imp_A#(my_transaction,
    inorder_analysis_component)
      analysis_export_A;
  uvm_analysis_imp_B#(my_transaction,
    inorder_analysis_component)
      analysis_export_B;

  // Fifos to hold the in-order data.
  uvm_tlm_fifo #(my_transaction) fifo_A;
  uvm_tlm_fifo #(my_transaction) fifo_B;

  function new(string name =
    "inorder_analysis_component",
    uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build();
    // Create 2 infinite size fifos. One
    // for each input stream.
    fifo_A = new("fifo_A", this, 0);
    fifo_B = new("fifo_B", this, 0);
    analysis_export_A =
      new("analysis_export_A", this);
    analysis_export_B =
      new("analysis_export_B", this);
  endfunction

  // When the A analysis_export is written,
  // this write() routine will be called.
  function void write_A(my_transaction t);
    `uvm_info("INORDER-A",
      $psprintf("Got '%s'.",
        t.convert2string()), UVM_INFO)
    void'(fifo_A.try_put(t));
  endfunction

  // When the B analysis_export is written,
  // this write() routine will be called.
  function void write_B(my_transaction t);
    `uvm_info("INORDER-B",
      $psprintf("Got '%s'.",
        t.convert2string()), UVM_INFO)
    void'(fifo_B.try_put(t));
  endfunction

  local function void compare(
    my_transaction A, B);
    `uvm_info("COMPARE", $psprintf("A=%s",
      A.convert2string()), UVM_INFO)
    `uvm_info("COMPARE", $psprintf("B=%s",
      B.convert2string()), UVM_INFO)
    `uvm_info("COMPARE",
      (A.data==B.data)?"PASSED":"FAILED",
        UVM_INFO)
  endfunction

  task run();
    my_transaction A, B;
    forever begin
      fifo_A.get(A);
      fifo_B.get(B);
      compare(A, B);
    end
  endtask

  function void check();
    `uvm_info("CHECK",
      $psprintf("A=%0d, B=%0d",
        fifo_A.used(), fifo_B.used()),
        UVM_INFO)
  endfunction
endclass
```

```
// =======================================
// Monitor.
// On each positive edge of the clock
// create a transaction and send it out
// the analysis port.
class monitor extends uvm_component;
  `uvm_component_utils(monitor)

  virtual bus_interface vbus_if;
  uvm_analysis_port #(my_transaction) ap;

  function new(string name   = "mon",
       uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build();
    ap = new("ap", this);
  endfunction

  task run();
    my_transaction t;
    bit inject_error;
    forever @(posedge vbus_if.clk) begin
      // Recognize transaction on the
      // clock edge...

      // Create new UVM transaction.
      t = my_transaction::type_id
        ::create("t");
      t.data = vbus_if.data;
      t.t = $time;
      $display("NOW=%0t", t.t);

      // Randomly inject a mismatch.
      assert(std::randomize(inject_error));
      if (inject_error)
        t.data += 100;

      `uvm_info("MON", $psprintf(
        "Sending new transaction '%s'.",
          t.convert2string()), UVM_INFO)

      // Send the transaction to any
      // subscribers
      ap.write(t);
    end
  endtask
endclass
```

```
// ========================================
// ENV.
// This env expects to connect to two
// interfaces.
// It will monitor those interfaces.
class env extends uvm_env;
  `uvm_component_utils(env)

  monitor                   monA;
  monitor                   monB;

  simple_analysis_component  simple_ac;
  inorder_analysis_component inorder_ac;

  virtual bus_interface      vbus_if_A;
  virtual bus_interface      vbus_if_B;

  function new(string name   = "env",
       uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build();
    monA = monitor::type_id
      ::create("monA", this);
    monB = monitor::type_id
      ::create("monB", this);

    simple_ac = simple_analysis_component::
      type_id::create("simple_ac", this);
    inorder_ac =
      inorder_analysis_component::
       type_id::create("inorder_ac", this);
  endfunction

  function void connect();
    // Retrieve the bus interface, and
    // pass to the monitor.
    // The 'A' Side.
    uvm_resource_db#(
      virtual interface bus_interface)
        ::read_by_name("A", "vbus_if",
          vbus_if_A);
    if (vbus_if_A == null)
      `uvm_fatal("ENV",
        "Virtual interface A is null")
    monA.vbus_if = vbus_if_A;
    monA.ap.connect(
      simple_ac.analysis_export);
    monA.ap.connect(
      inorder_ac.analysis_export_A);

    // The 'B' Side.
    uvm_resource_db#(
      virtual interface bus_interface)
        ::read_by_name("B", "vbus_if",
          vbus_if_B);
    if (vbus_if_B == null)
      `uvm_fatal("ENV",
        "Virtual interface B is null")
    monB.vbus_if = vbus_if_B;
    monB.ap.connect(
      simple_ac.analysis_export);
    monB.ap.connect(
      inorder_ac.analysis_export_B);
  endfunction

  task run();
    #1000;
    uvm_top.stop_request();
  endtask
endclass
```

```
// ========================================
// Interface.
// Interface to be bound into the DUT.
interface bus_interface(
  input bit clk, bit [31:0] data);
  // Wires...
endinterface

// ========================================
// DUT.
// On the positive edge of the clock, if
// reset is high, clear the counter,
// otherwise increment the counter by 1.
module dut(input wire dut_clk,
           input wire reset);
  bit [31:0]counter = 0;

  always @(posedge dut_clk)
    if (reset == 1'h1)
      counter = 0;
    else
      counter++;
endmodule
```

```
// ========================================
// Top.
// Instantiate two DUTs and an monitor
// environment. Bind two interfaces into
// the DUT to allow internal monitoring.
// Register those bound interfaces.
module top();
  bit fast_clk, clk, reset;

  dut dutA(clk,      reset);
  dut dutB(fast_clk, reset);

  env e = new("env", null);

  // Create the probes.
  // Bind a virtual interface into each
  // dut. Name the instances dutA.dut_if
  // and dutB.dut_if. Connect the signal
  // 'clk' and 'counter', as referenced
  // from within the dut.
  bind dutA bus_interface dut_if(
    dut_clk, counter);
  bind dutB bus_interface dut_if(
    dut_clk, counter);

  initial begin
    // Put the probes into a database.
    uvm_resource_db #(
      virtual interface bus_interface)
        ::set("A", "vbus_if", dutA.dut_if);
    uvm_resource_db #(
      virtual interface bus_interface)
        ::set("B", "vbus_if", dutB.dut_if);

    run_test();
  end

  always begin
    #5 clk = 0;      #5 clk = 1;      end
  always begin
    #1 fast_clk = 0; #1 fast_clk = 1; end

  initial begin
    // Reset, then run. Repeat.
    reset = 1; #20; reset = 0; #100;
    reset = 1; #20; reset = 0; #100;
  end
endmodule
```