Monitors, Monitors Everywhere – Who Is Monitoring the Monitors

Rich Edelman
Mentor Graphics

Raghu Ardeishar
Mentor Graphics

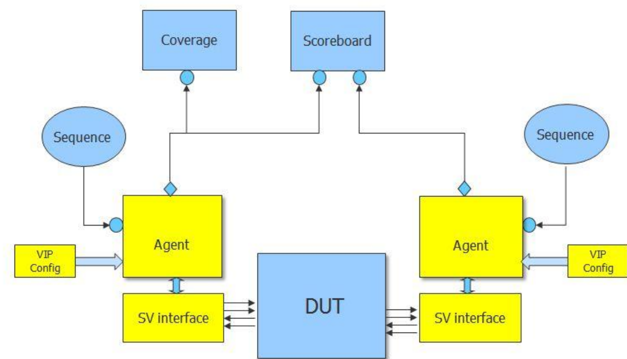
Abstract—The reader of this paper should be interested in predicting the behavior of his hardware or is interested in monitoring his hardware. This paper will review phase-level monitoring, transaction-level monitoring, and general monitoring. In-order and out-of-order transaction-level monitors and UVM constructs for single and multiple port monitors will be demonstrated, including discussion about simple function implementations versus FIFO and threaded implementations. A protocol specific AXI monitor written at the transaction-level of abstraction will be demonstrated. This monitor and scoreboard can model many AXI interactions, but under certain AXI usages, problems arise. For example partially written data may be read by an overlapping READ. This kind of behavior cannot be modeled by the "complete transaction" kind of monitor; it must be modeled by a phase-level monitor. Such a phase-level monitor will be demonstrated. All of these monitoring and scoreboard discussions can be widely applied to many protocols and many monitoring situations.

Keywords—components; transactions; TLP; AXI; phase; monitor; scoreboard; agent

I. INTRODUCTION TO MONITORING

In a verification environment the task of a monitor is to monitor activity on a set of DUT pins. This could be as simple as looking at READ/WRITE pins or as complex as a complete protocol bus, such as AXI or PCIe. In a very simple case a monitor can be looking at a pin or a set of pins and generating an event or raising a flag every time there is a change in signal values. The flag or event can trigger a scoreboard or coverage collector to perform an activity. This sort of monitor is typically very slow and not very useful as it generates a lot of irrelevant data.

At a higher level the monitor can be looking at a set of signals and generating an event when an activity or transaction completes for example a read or write transaction. In this case the monitor has to be smart to recognize the start and end of a transaction not just look at changes in signal values. Once a transaction is complete it not only raises a flag but passes the details of the transaction to any component in the design who wants to observe it. The details can be as simple as the address and data in a read/write transaction or as complex as all the fields in a PCIe transaction like address/data/requester ID/CRC/Error status etc. This kind of monitor is harder to write but yields the best results in quality of data generated as well as speed of execution by the components which use the data – for example a scoreboard. A monitor in a modern UVM based verification environment is usually a component of an agent which is described later.



Generally monitoring is used to generate data for scoreboarding, for debug, for coverage collection, and for further downstream processing. A scoreboard is usually used to predict or check behavior. A typical scoreboard listens to a monitor and is written to receive the same format of data being generated by the monitor. In most cases it is processing two or more streams of data coming in. One stream of data can be coming as a result of a write transaction and the other from a read transaction. The scoreboard in this case will compare if the data being read from a slave address is the same as the data which was written In a simple scoreboard you might only be looking at one specific protocol eg, both the data being written and read is of type AXI. In a more complex case you might generate a write using a AXI but that might get morphed into a USB bulk out. Similarly a read might begin as a USB bulk in and translate into an AXI read. The scoreboard in this case will be dealing with multiple streams of traffic being generated by several different monitors each snooping different sections of the design. The task of writing effective scoreboards has always been a vexing one. Yet, the problem actually begins with the data being sent to the scoreboard from the monitor.

A monitor collecting data at a low level of abstraction (at the wire level) will be slow and will be hard to write. A monitor collecting data at a high level of abstraction (at the "completed" transaction level) may miss details. If you wait for the entire transaction to be complete (e.g., an AMBA

READ or WRITE), you get an efficient, high level of abstraction but miss out on key timing points during the transaction (like the individual AXI phases).

The key to successful monitoring and scoreboarding is to generate and process the data at an appropriate level of abstraction—in this case, the phase level—which will be low enough to gather the data at key intervals and high enough to render the scoreboard efficient and robust.

An agent typically is a collection of the most commonly used components; such as a sequencer, driver, and monitor. A sequencer typically arbitrates among several sequences and sends it to the driver, which is ACTIVE, and converts high-level commands into pin-level activity on the bus. A monitor is PASSIVE; i.e., it does not drive traffic on the bus.

A monitor typically has one or more analysis ports. When a monitor recognizes activity on a bus it generates traffic on the analysis port. An efficient monitor transmits on the level of detail which is useful to a scoreboard or coverage collector listening to it. For example for an AXI scoreboard the monitor recognizes the start of a transaction , waits for the end and only then creates a transaction packet with the relevant details and sends it out the analysis port. In this case it has only the address/data/type of the transaction. Non relevant details like the response delay, latency are dropped. That information is useful to another analysis component like a coverage collector. In that case a different packet is created. By processing on the relevant information you speed up generation and processing of data. The analysis ports are used to transmit activity that the monitor has observed on the bus based on the level of abstraction the user chooses. The analysis ports can be programmed to be of any type based on the complexity of the transaction one wishes to send. It can have the value of one signal or a complete set of wires on a bus. An analysis port is connected to a subscriber, which listens to the data transmitted by the monitor. An example of a subscriber is a scoreboard or coverage collector.

An efficient monitor also looks at pin-level activity on a bus and over a period of time recognizes complete transactions. Those transactions can be represented by transaction-level packets (TLP), which can be processed more effectively. A complete transaction can be broken down into several sub-transactions (which we refer to as phases) that are at a lower level of abstraction than the complete transaction but are at a higher level than the pin-level activity per clock cycle. Once a TLP has been recognized from the bus it is broadcast via an analysis port to subscribers. The key point in designing a monitor is determining what level of abstraction the TLP should be at.

In this paper we will analyze monitors and scoreboards that look at transactions at two levels. The highest level we will call the transaction level and the lower level we will call the phase level.

II. Components of the Environment

A. *Building Monitors*

A monitor is very similar to a driver. While a driver consumes an activity at a transaction level and converts it into wire level activity, a monitor does the reverse; i.e., it consumes wire level activity on a bus and creates a transaction-level packet. A monitor does not drive activity on the bus. A monitor is also protocol aware. It detects protocol related patterns in signal activity usually using a state machine. Monitors extend from *uvm_monitor*. They have at least one analysis port, which is of type *uvm_analysis_port*. When the monitor recognizes a protocol compliant transaction on the bus it assembles a transaction-level packet from the wire-level signals and sends the packet through the analysis port. The broadcast from the analysis port is done using the WRITE method. For example:

```
class axi_monitor extends uvm_monitor;
  `uvm_component_utils(axi_monitor)
  uvm_analysis_port #(axi_tlp) axi_ap;
….
  task run_phase(uvm_phase phase)
    axi_tlp tlp;
    forever @(clk)
      // Detect the tlp from the bus pins

      axi_ap.write(tlp);
  endtask
endclass
```

B. *Building Subscribers*

A subscriber listens to the analysis port on the monitor. A subscriber has to be compatible with the monitor i.e. if the monitor is generating data in a certain format the subscriber has to be able to receive it. You can have one or many subscribers listening on any analysis port. A UVM-based subscriber has a WRITE method implemented. A subscriber extends from *uvm_subscriber* and receives the data from the analysis component when the *<analysis_port>.write* method is called. The WRITE method is polymorphic; i.e., the WRITE method implemented in the subscriber is the actual method that is invoked.

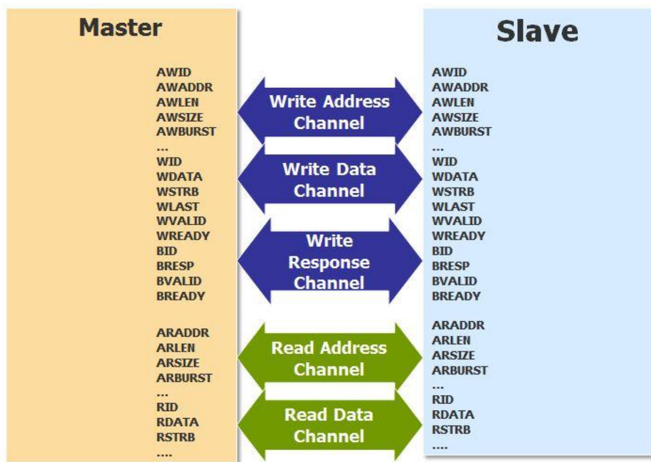
```
class axi_item_listener #( type ITEM_TYPE = int )
  extends uvm_subscriber #(uvm_sequence_item_base );
  typedef ITEM_TYPE item_t;
  `uvm_component_param_utils( uvm_item_listener
                              #( ITEM_TYPE ) );
  function write(input uvm_sequence_item_base t );
    item_t s;
    if( ! $cast( s , t ) )
      uvm_report_fatal( "mvc type error" ,
        $psprintf( "…"  ,
          t.sprint(), get_full_name() ) );
    …
  endfunction
function
endclass
```

C. *Building a Scoreboard*

A scoreboard is a type of subscriber. A scoreboard determines if a DUT is functioning within parameters. A UVM-based

scoreboard is an analysis component that extends from *uvm_subscriber*. When the WRITE task from the monitor is issued it calls the WRITE task (or equivalent) from the *uvm_subscriber*. That way there can be many subscribers — each with their WRITE tfs (tasks/functions) — connected to the monitor. Most scoreboards receive one type of transaction. It can receive many streams of data from different analysis ports but they all have to be of the same type. In that case a single WRITE method would suffice. In case the scoreboard needs to receive more than one type of transaction, it can no longer be a subscriber i.e. it cannot extend from uvm_subscriber. It is now a uvm_component and will have as many different WRITE methods as the different types of transactions it has to receive. For this paper we will be using the AXI bus as a reference. First we will give a quick overview of AXI transactions.

III. AXI PROTOCOL REVIEW



AXI is a point-to-point and burst-based protocol. Point-to-point means that there is data transfer between one master to one slave. There can be more than one master and one slave, but that is done by building an external fabric. This is not inherent to the protocol, as is the case with its sister protocol “AHB”. Every transaction has address and control information on the address channel. The data is transferred between master and slave using a WRITE data channel to the slave or a READ data channel to the master. In a WRITE transaction, all data flows from master to slave. It has a WRITE response channel to allow the slave to signal a response to a WRITE transaction.

The AXI protocol has five channels:

1. Write address channel
2. Write data channel
3. Write response channel
4. Read address channel
5. Read data channel

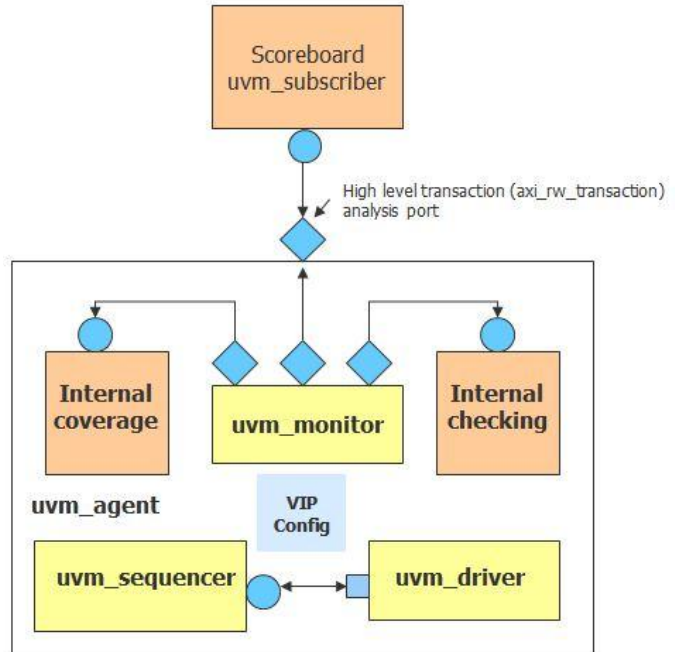
An AXI transaction can be a WRITE or a READ. A WRITE or READ can be of three types: FIXED, INCR, and WRAP.

A FIXED WRITE is equivalent to writing to a FIFO. The WRITE is to a fixed address and each WRITE adds to the FIFO, and a READ pops the data out.

An INCR is a burst WRITE/READ where a starting address and the length of the transfer is specified.

A WRAP is similar to an INCR but the addresses wrap at an address boundary. The rules for calculating the boundaries are detailed in the AMBA AXI specification.

IV. TRANSACTION-LEVEL SCOREBOARD



We will start with an example of an AXI scoreboard that monitors transactions at the highest abstraction level in AXI: the READ or WRITE transaction level. In this case a monitor which is “monitoring” the AXI bus recognizes only one type of transaction. Once the transaction is recognized it is sent out the analysis port to any subscriber which is attached to it. Only one WRITE method needs to be implemented in the subscriber. The READs and WRITEs can be FIXED, INCR, or WRAP. Notice that the transaction-level scoreboard receives only one type of transaction. This can be a READ or a WRITE and is received when the transaction is complete.

```
class axi_scoreboard extends uvm_subscriber;
  typedef axi_master_rw_transaction item_t;
  function do_write(item_t t);
    if(t.read_or_write == AXI_TRANS_WRITE) begin
      process_write(t);
    else
      process_read(t);
  endfunction

  function process_write(item_t t);
```

```
   case(t.burst)
     AXI_FIXED: process_write_fixed(t);
     AXI_INCR:  process_write_incr(t);
     AXI_WRAP:  process_write_wrap(t);

  endfunction
  function process_write_incr(item_t t);
    // Update memory with data in
  endfunction
endclass
```

The above scoreboard updates its memory every time a WRITE transaction is detected on the bus. It waits for a WRITE response from the slave signaling the completion of the transaction before updating the scoreboard memory.

An issue with this approach is when a READ and WRITE to identical addresses overlap. The scoreboard waits for the WRITE response, which has not yet been received. So if a READ is received while a WRITE is in progress, the DUT memory *may* have been updated, but the scoreboard memory does not reflect it. So the scoreboard has to be modified to keep track of old and new memory contents.

Using the scoreboard at the current level — i.e., waiting for the entire READ or WRITE transaction to complete — is inadequate to build a robust scoreboard. We have to drop down in abstraction level to get a better view of the transaction. Now let us take a look at the scoreboard at the phase level.

V. PHASE-LEVEL SCOREBOARD

For this we will take a closer look at the AXI WRITEs and READs. An AXI WRITE transaction can be broken down into three channels or phases.

1. axi_write_address_channel_phase
2. axi_write_data_channel_phase
3. axi_write_response_channel

An AXI READ transaction can be broken down into three channels or phases.

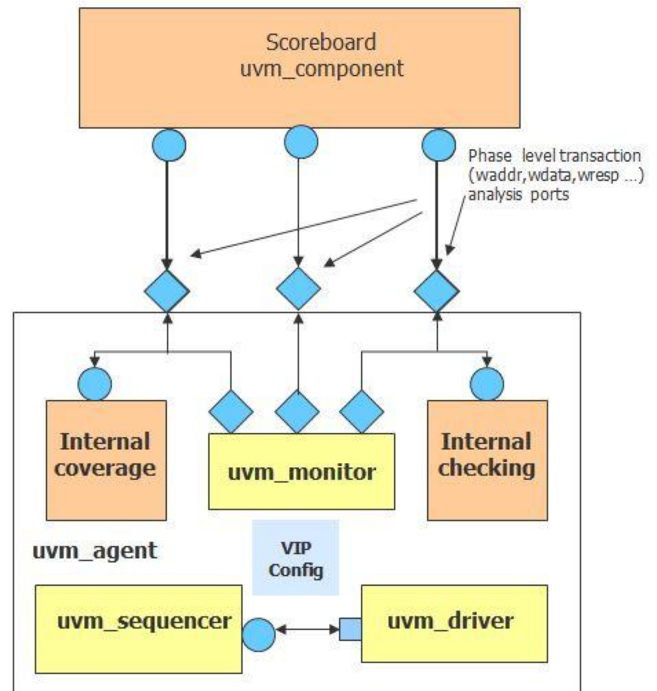
1. axi_read_address_channel_phase
2. axi_read_data_channel_phase

When a WRITE transaction is issued by the AXI master, the master issues the starting address for the transfer along with the burst length (number of beats in the transfer) and other control signals. Once the address is accepted by the slave, the data is received by the slave. A data burst can have more than one "beat" in the transfer, as will be indicated by the control signals. The DUT sends a WRITE response once the WRITE completes. The address and data channel are independent of each other. The data can be sent before or after the address so the scoreboard needs to independently keep track of it.

However, internally the DUT has the option to update its memory with every beat of the transfer or wait till the entire burst is completed. The scoreboard has to reflect either behavior. Hence the scoreboard can no longer wait for the full AXI READ/WRITE TLP to be received in order to process the information.

AXI has five independent channels. READs and WRITEs can overlap. With some hardware, overlapping READs and WRITEs can result in hardware specific behavior, particularly regarding the race condition between those overlapping READs and WRITEs. So the scoreboard has to be modified to receive all five phases of transactions corresponding to each of the five AXI channels.

To accomplish this we need a monitor to generate the different phases of the AXI transaction and the scoreboard to receive it.




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

  uvm_analysis_port #(write_addr_tlp) waddr_ap;
  uvm_analysis_port #(write_data_tlp) wdata_ap;
  uvm_analysis_port #(write_resp_tlp) wresp_ap;
  uvm_analysis_port #(read_addr_tlp)  raddr_ap;
  uvm_analysis_port #(read_data_tlp)  rdata_ap;
….
  task run_phase(uvm_phase phase)
   axi_write_addr_tlp waddr_tlp;
   axi_write_data_tlp wdata_tlp;
   axi_write_resp_tlp wresp_tlp;

   axi_read_addr_tlp raddr_tlp;
   axi_read_data_tlp rdata_tlp;

   forever @(clk)
     // Detect the 5 tlps from the bus pins
```

```
    waddr_ap.write(waddr_tlp);
    wdata_ap.write(wdata_tlp);
    wresp_ap.write(wresp_tlp);
    raddr_ap.write(raddr_tlp);
    rdata_ap.write(rdata_tlp);

  endtask
endclass
```

Since the new scoreboard will now receive five different streams of traffic, it has to be a *uvm_component*. It cannot be a *uvm_subscriber* because a *uvm_subscriber* can have only one WRITE task and we need five WRITE tasks. Therefore, we will use the *`uvm_analysis_imp_decl(phase_name)* macro; a UVM "trick."

UVM provides a workaround by defining the *`uvm_analysis_imp_decl* macro. This macro allows you to declare a specialized *imp*-style analysis export that calls a function named, *write_SUFFIX*, where you specify *_SUFFIX*. Consider :

```
`uvm_analysis_imp_decl(_write_addr_ph)
`uvm_analysis_imp_decl(_write_data_ph)

class axi_phase_scoreboard extends uvm_component;
  uvm_analysis_imp_write_addr_ph
    #(uvm_sequence_item_base,  axi_phase_scoreboard)
                               write_addr_export ;

  uvm_analysis_imp_write_data_ph
    #(uvm_sequence_item_base,  axi_phase_scoreboard)
                               write_data_export ;

  function void write_write_addr_ph(
    uvm_sequence_item_base m);

  function void write_write_data_ph(
    mvc_sequence_item_base m);
…
endclass
```

The scoreboard is connected to the analysis port in the agent. Every time the monitor detects an AXI phase on the bus, it is sent through the appropriate analysis port. The *uvm_component* will have five *write_xxx* methods which will process each phase of the AXI transaction.

Since we are receiving transactions at a lower level than the fully completed transaction, we can maintain separate copies of completed and ongoing READs and WRITEs. When a WRITE response is received from the slave, we will update the memory and remove the old data. Until a WRITE response is received, we will keep a copy of the current memory and the new data that is being written. So if a READ comes along to the address which is being written, it can be compared to either the current contents of the memory or the newly written data.

SUMMARY

We have discussed several different types of monitors and associated components which use the information which the monitors generate like a scoreboard.

The pin level version of the monitor is the simplest to create which typically generates information at almost every clock cycle. This generates too much information and typically is not very useful in a real environment as it makes scoreboarding and coverage collection very complex. It is also very slow.

The other end of the spectrum is a high level transaction level monitor and scoreboard which waits for complex transactions to complete before generating data thru the analysis port. This makes scoreboarding very simple and extremely fast giving a very high level view of the design. However in certain cases lack of information in the high level packet can result in an inflexible scoreboard or coverage collector.

An intermediate approach is a phase level monitor and scoreboard in which the monitor breaks the high level transactions into lower level transactions which are sent thru the analysis port. The scoreboard becomes more complicated but a lot more flexible. The speed is also slightly slower.

VI. References

.

[1] SystemVerilog LRM. www.accellera.org

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

[3] AXI Specifcation.www.arm.com

[4] Panning for Gold in RTL Using Transactions

VII. APPENDIX

```
// ----------------------------------------------------
// Simple one port monitor
//
class one_port_monitor extends
uvm_subscriber#(transaction);
  `uvm_component_utils(one_port_monitor)

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

  function void write(transaction t);
    `uvm_info("1PORT", $sformatf("Got %s",
      t.convert2string()), UVM_MEDIUM)
  endfunction
endclass


`uvm_analysis_imp_decl(_1)
`uvm_analysis_imp_decl(_2)

// ----------------------------------------------------
// Two port in-order monitor
//
class two_port_monitor extends uvm_component;
  `uvm_component_utils(two_port_monitor)

  uvm_analysis_imp_1 #(transaction, two_port_monitor) p1;
  uvm_analysis_imp_2 #(transaction, two_port_monitor) p2;

  uvm_tlm_fifo #(transaction) f1;
  uvm_tlm_fifo #(transaction) f2;

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

  function void build_phase(uvm_phase phase);
    p1 = new("p1", this);
    p2 = new("p2", this);
    f1 = new("f1", this, 0);
    f2 = new("f2", this, 0);
  endfunction

  function void write_1(transaction t);
    `uvm_info("2PORT_1", t.convert2string(), UVM_MEDIUM)
    void'(f1.try_put(t));
  endfunction

  function void write_2(transaction t);
    `uvm_info("2PORT_2", t.convert2string(), UVM_MEDIUM)
    void'(f2.try_put(t));
  endfunction

  task run_phase(uvm_phase phase);
    transaction t1, t2;

    forever begin
      fork
        f1.get(t1);
        f2.get(t2);
      join
      t1.do_check("2PORT", t2);
    end
  endtask

  function void check_phase(uvm_phase phase);
    // If anything exists in the f1 or f2,
    // then they are unmatched.
    transaction t;
    while (f1.try_get(t)) begin
      `uvm_error("CHECK",
        $sformatf("1: transaction=%0s not matched",
        t.convert2string()))
    end
    while (f2.try_get(t)) begin
      `uvm_error("CHECK",
        $sformatf("2: transaction=%0s not matched",
        t.convert2string()))
    end
  endfunction
endclass

// ----------------------------------------------------
// Two port out-of-order monitor
//
class out_of_order_two_port_monitor extends
uvm_component;
  `uvm_component_utils(out_of_order_two_port_monitor)

  uvm_analysis_imp_1
    #(transaction, out_of_order_two_port_monitor) p1;
  uvm_analysis_imp_2
    #(transaction, out_of_order_two_port_monitor) p2;

  transaction stream1[int];
  transaction stream2[int];

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

  function void build_phase(uvm_phase phase);
    p1 = new("p1", this);
    p2 = new("p2", this);
  endfunction

  function void write_1(transaction t);
    `uvm_info("OoO_1", t.convert2string(), UVM_MEDIUM)
    if (stream2.exists(t.id)) begin
      t.do_check("OoO", stream2[t.id]);
      stream2.delete(t.id);
    end
    else
      stream1[t.id] = t;
  endfunction

  function void write_2(transaction t);
    `uvm_info("OoO_2", t.convert2string(), UVM_MEDIUM)
    if (stream1.exists(t.id)) begin
      t.do_check("OoO", stream1[t.id]);
      stream1.delete(t.id);
    end
    else
      stream2[t.id] = t;
  endfunction

  function void check_phase(uvm_phase phase);
    // If anything exists in the stream1 or stream2
    // arrays, then they are unmatched.
    foreach(stream1[i])
      `uvm_error("CHECK",
        $sformatf("1: ID=%0d not matched", i))
    foreach(stream2[i])
      `uvm_error("CHECK",
        $sformatf("2: ID=%0d not matched", i))
  endfunction
endclass
```