VMM PRIMER

Using the Data Stream Scoreboard

Author(s):

Janick Bergeron

Version 1.1 / May 17, 2006

Introduction

The VMM Data Stream Scoreboard is an application package that can be used to simply the creation of a self-checking structure of "data mover" designs. Data mover designs are design that take data on one side, transform it, and then produce data on the other side. Routers, modems, codec, DSP functions, bridges and busses are all data mover designs.

This primer is designed to learn how to create a scoreboard based on the Data Stream Scoreboard foundation classes, and how to integrate this scoreboard in a verification environment. Other primers cover other aspects of developing VMM-compliant verification assets, such as transactors, generators, assertions and verification environments.

This primer assumes that you are familiar with VMM. If you need to ramp-up on VMM itself, you should read the other primers in this series, such as "Writing a Command-Layer Command Transactor".

The DUT used in this primer was selected for its simplicity. Because of its simplicity, it does not require the use of many elements of the VMM Data Stream Scoreboard package. It is sufficient to achieve the goal of demonstrating, step by step, how to use create a scoreboard and use it to verify a design.

This document is written in the same order you would develop a scoreboard, integrate it in a verification environment and verify your design. As such, you should read it in a sequential fashion. You can use the same sequence to create your own scoreboard and use it to verify your design.

The Verification Environment

The Design Under Test used in this primer is an AMBA™ Peripheral Bus (APB), with a single master port and three slave ports. It is a simple address decoding device with the following address map:

Table 1: Address Map

paddr[9:8]	Slave
2'b00	Slave #0, paddr[7:0]
2'b01	Slave #1, paddr[7:0]
2'b10	Slave #2, paddr[7:0]

The simplicity of the APB bus does not require the use of a full-fledge scoreboard to verify the correctness of its operation. Since it is not pipelined and does not support out-of-order execution, a simple global variable would suffice. However, the purpose

of this primer is to show how to use the Data Stream Scoreboard foundation classes, not verify a complex functionality. This simple design serves the purpose quite well. Let's ignore the fact that the design is a trivial bus and assume that it is a complex pipelined bus that can execute multiple transactions simultaneously.

As shown in Figure 1, the design is exercised by an APB master transactor and the responses are provided by three APB slave transactors. These are the same transactors that were created in the primers on writing command-layer transactors. Transactions are created by an atomic generator. As is, this verification environment is not self-checking.

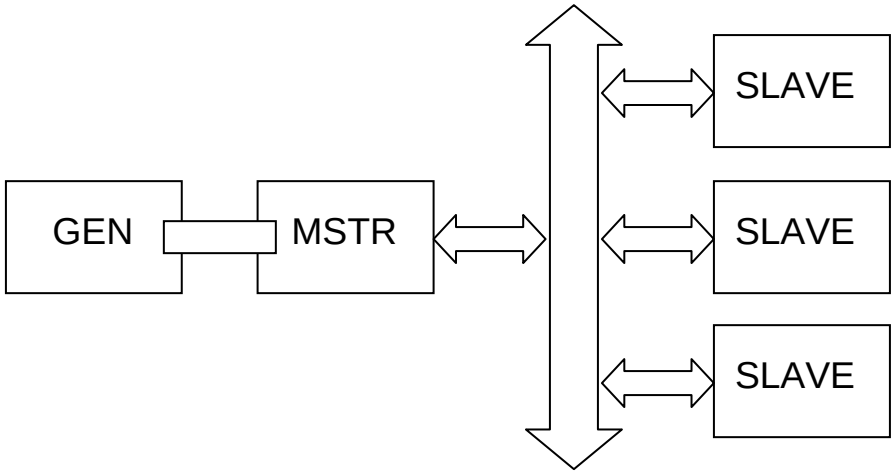

Figure 1: Verification Environment

Beside its simplicity, there is another reason for selecting this design as the DUT for this primer. The name "Data Stream Scoreboard" seems to imply that it is suited only for data networking applications. The documentation and method arguments themselves refer to the data to be verified as "packets". But "packet" and "data stream" are abstract notions. They are used to identify transactions and flows of transaction and do not imply any particular technology or application. Using a bus as the DUT give you the opportunity to break any fallacious mental association between the Data Stream Scoreboard package and data networking applications.

In this example, a "packet" is an APB transaction and a stream is a transaction executed between the master and one of the slaves. There are thus two types of packets in the design: READ and WRITE cycles. And there are three streams, one for each slave.

Step 1: The Self-Checking Strategy

A scoreboard is only an implementation medium. It must be properly used and fit in an overall self-checking strategy to be effective at verifying the response of a design.

One possible strategy would be to use the default RAM-like behavior of each slave and perform a series of WRITE-READ cycles. The correctness would be verified by checking that the data that was read back was indeed the data that was written. But given the nature of the design, this is a rather poor self-checking strategy. It would fail to uncover several classes of functional bugs, such as misconnected address and data busses and incorrect address decoding. It also requires that a READ cycle targets an address that was previously written to, making it impossible to verify the correct operation of two back-to-back WRITE cycles to the same address.

A better strategy is to use the optional response channel in the slave transactor. When present, it allows a higher-level transactor—in this case a response generator—to provide an arbitrary response to any READ cycle. A RAM behavior is only one of the possible responses, one that happens to be built in the slave transactor.

Figure 2 shows the structure of the verification environment with slave response generators attached to each slave. These response generators do not react to WRITE cycles but provide a random response to READ cycles.

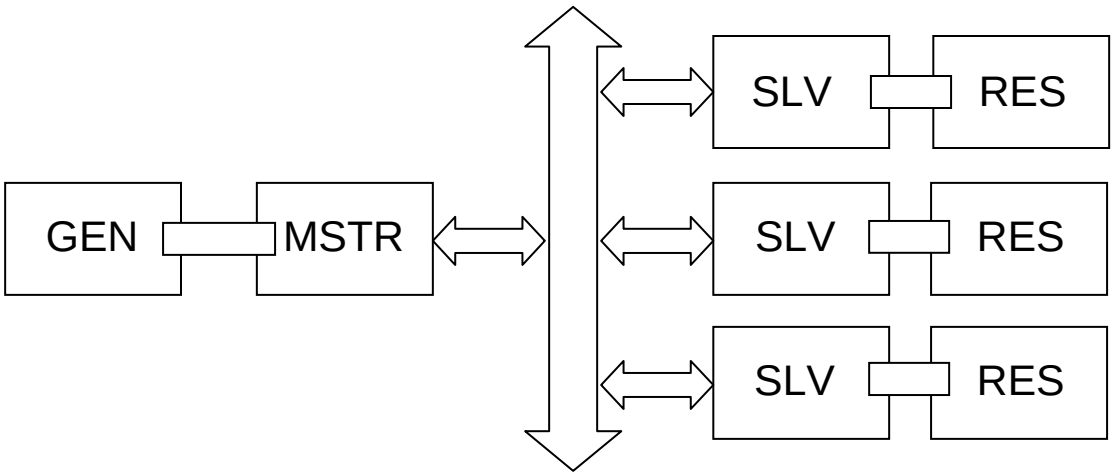

Figure 2: Random Slave Response Environment

The slave response generator can be implemented as a VMM compliant transactor and be reusable. Because the intent of this primer is not to show how to write VMM-compliant random response generator, a simple non-reusable and unconstrainable response generator will be used. It can be coded directly in the environment as a forked-off thread.

File: DateStream_DB/tb_env.sv

```
...
class tb_env extends vmm_env;
   ...
   apb_slave      slv[3];
   apb_rw_channel resp_chan[3];
   ...
```

```
   virtual function build();
      ...
      this.resp_chan[0] = new("Response", "0");
      this.resp_chan[1] = new("Response", "1");
      this.resp_chan[2] = new("Response", "2");
      this.slv[0] = new("Slave", 0, tb_top.s0, ,
                        this.resp_chan[0]);
      this.slv[1] = new("Slave", 1, tb_top.s1, ,
                        this.resp_chan[1]);
      this.slv[2] = new("Slave", 2, tb_top.s2, ,
                        this.resp_chan[2]);
   endfunction: build

   virtual function start();
      ...
      foreach (this.resp_chan[i]) begin
         int j = i;
         fork
            forever begin
               apb_rw tr;
               this.resp_chan[j].peek(tr);
               ...
               if (tr.kind == apb_rw::READ) begin
                  tr.data = $random(); // Poor random strategy!
                  ...
               end
               this.resp_chan[j].get(tr);
            end
         join_none
      end
   endfunction: start
   ...
endclass: tb_env
```

With random responses, it is no longer possible to predict the expected response strictly from the master's interface. It is necessary to carry the expected response from the master to the slave (to verify that the right slave is targeted with the right address and write data) and from the slave to the master (to verify the read data). This requires two scoreboards: one in the master-to-slaves direction, the other in the slaves-to-master direction.

Step 2: Master-to-Slaves

The master-to-slaves scoreboard requires one queue of input transaction and three queues of expected transaction, one per slave. This is implemented using the Data Stream Scoreboard foundation class by defining one input stream and three expected streams. A unique stream identifier must be assigned to each stream. The stream identifier is chosen to facilitate the association of a slave with its corresponding stream during integration. In this case, the stream identifier corresponds to the value of 'j' in the response generator thread.

DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class m2s_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");

      this.define_stream(0, "Master",  INPUT);
      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
   endfunction: new
   ...
endclass: m2s_sb
...
```

The expected transaction, as observed by the slaves, will be different from the one injected by the master. Because each slave only has 256 addressable locations, bits [31:8] of the address will always be zero. Any bits set by the master will be masked. It is thus necessary to perform the same transformation in the scoreboard.

DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class m2s_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");

      this.define_stream(0, "Master",  INPUT);
      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
   endfunction: new

   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      apb_rw tr;

      $cast(tr, in_pkt.copy());
      tr.paddr[31:8] = '0;

      out_pkts = new [1];
      out_pkts[0] = tr;
   endfunction: transform
   ...
endclass: m2s_sb
...
```

By default, the comparison function use by the Data Stream Scoreboard foundation class is the vmm_data::compare() method defined for the transaction descriptor. However, when verifying the response of the master-to-slave path, the data value can only be compared against an expected value for WRITE cycles. It is thus necessary to specify a custom comparisons function in the master-to-slave scoreboard. This is done by overloading the vmm_sb_ds::compare() method.

DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class m2s_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");

      this.define_stream(0, "Master",  INPUT);
      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
   endfunction: new

   virtual function bit transform(input  vmm_data in_pkt,
                                  output vmm_data out_pkts[]);
      apb_rw tr;

      $cast(tr, in_pkt.copy());
      tr.addr[31:8] = '0;

      out_pkts = new [1];
      out_pkts[0] = tr;
   endfunction: transform

   virtual function bit compare(vmm_data actual,
                                vmm_data expected);
      apb_rw act, exp;
      $cast(act, actual);
      $cast(exp, expected);

      if (act.kind == apb_rw::WRITE) begin
         string diff;
         return act.compare(exp, diff);
      end

      return (act.kind == exp.kind) &&
             (act.addr == exp.addr);
   endfunction: compare
endclass: m2s_sb
...
```

Step 3: Integration

Once the functionality of the scoreboard is defined, it must then be integrated with the rest of the verification environment.

The stimulus transaction executed by the master can be extracted using either the pre_cycle() or post_cycle() callback method. Because the post_cycle() method will only be invoked when the transaction will have completed—and hence after the slave has responded—it will be too late to check the transaction that the slave sees against what the master is executing. The pre_cycle() method is thus the proper integration point. The fact that the read-back data is not valid when this callback method is invoked is a non-issue since it is not compared against expected values.

DataStream_SB/tb_env.sv

```
...
class apb_master_to_sb extends apb_master_cbs;
   m2s_sb m2s;
   ...
   function new(m2s_sb m2s, ...);
      this.m2s = m2s;
      ...
   endfunction: new

   virtual task pre_cycle(apb_master xactor,
                          apb_rw     cycle,
                          ref bit    drop);
      this.m2s.insert(cycle, vmm_sb_ds::INPUT);
   endtask: pre_cycle
   ...
endclass: apb_master_to_sb

class tb_env extends vmm_env;
   m2s_sb m2s = new;
   ...
   virtual function build();
      ...
      this.mst = new("Master", 0, tb_top.m, this.gen.out_chan);
      begin
         apb_master_to_sb cbs = new(this.m2s, ...);
         this.mst.append_callback(cbs);
      end
      ...
   endfunction: build
   ...
endclass: tb_env
```

Integrating the checking part is easier since the response generator is implemented directly in the environment. It is only necessary to invoke the proper checking function, identifying the stream this response has been observed on.

DataStream_SB/tb_env.sv

```
...
class tb_env extends vmm_env;
   ...
   virtual function start();
      ...
      foreach (this.resp_chan[i]) begin
         int j = i;
         fork
            forever begin
               apb_rw tr;
               this.resp_chan[j].peek(tr);
               this.m2s.expect_in_order(tr, j);
               if (tr.kind == apb_rw::READ) begin
                  tr.data = $random(); // Poor random strategy!
                  ...
               end
               this.resp_chan[j].get(tr);
            end
         join_none
      end
   endfunction: start
   ...
endclass: tb_env
```

Step 4: Slaves-to-Master

The slaves-to-master scoreboard also requires three queues of input transactions, one per slave. This is implemented using the Data Stream Scoreboard foundation class by defining three input streams. A unique stream identifier must be assigned to each stream. The stream identifier is chosen to facilitate the association of a slave with its corresponding stream during integration. In this case, the stream identifier corresponds to the value of 'j' in the response generator thread.

DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class s2m_sb extends vmm_sb_ds;
   function new();
      super.new("Slave->Master");

      this.define_stream(0, "Slave 0", INPUT);
      this.define_stream(1, "Slave 1", INPUT);
      this.define_stream(2, "Slave 2", INPUT);
      this.define_stream(0, "Master",  EXPECT);
   endfunction: new
```

```
   ...
endclass: s2m_sb
...
```

The completed transactions, as reported by the master, will be different from the one replied by the slaves. The only information that is transferred from a slave to the master is the read back data. All of the remaining information is unmodified. The simplest approach is to leave the response transaction as-is and only compare the data value of READ cycles.

DataStream_SB/tb_env.sv

```
...
`include "vmm_sb.sv"

class s2m_sb extends vmm_sb_ds;
   function new();
      super.new("Master->Slave");

      this.define_stream(0, "Slave 0", EXPECT);
      this.define_stream(1, "Slave 1", EXPECT);
      this.define_stream(2, "Slave 2", EXPECT);
      this.define_stream(0, "Master",  EXPECT);
   endfunction: new

   virtual function bit compare(vmm_data actual,
                                vmm_data expected);
      apb_rw act, exp;
      $cast(act, actual);
      $cast(exp, expected);

      if (act.kind == apb_rw::WRITE) return 1;

      return (act.data == exp.data);
   endfunction: compare
endclass: s2m_sb
...
```

Step 5: Integration

Once the functionality of the scoreboard is defined, it must then be integrated with the rest of the verification environment.

The completed transaction observed by the master can be extracted using the post_cycle() callback method and used to compare against expected transaction if a response was expected.

DataStream_SB/tb_env.sv

```
...
class apb_master_to_sb extends apb_master_cbs;
   m2s_sb m2s;
   s2m_sb s2m;

   function new(m2s_sb m2s, s2m_sb s2m);
      this.m2s = m2s;
      this.s2m = s2m;
   endfunction: new

   virtual task pre_cycle(apb_master xactor,
                          apb_rw     cycle,
                          ref bit    drop);
      this.m2s.insert(cycle, vmm_sb_ds::INPUT);
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw     cycle);
      if (cycle.addr[9:8] == 2'b11) return;
      if (cycle.kind == apb_rw::WRITE) return;
      this.s2m.expect_in_order(cycle,
                               .inp_stream_id(cycle.addr[9:8]));
   endtask: post_cycle
endclass: apb_master_to_sb

class tb_env extends vmm_env;
   m2s_sb m2s = new;
   s2m_sb s2m = new;
   ...
   virtual function build();
      ...
      this.mst = new("Master", 0, tb_top.m, this.gen.out_chan);
      begin
         apb_master_to_sb cbs = new(this.m2s, this.s2m);
         this.mst.append_callback(cbs);
      end
      ...
   endfunction: build
   ...
endclass: tb_env
```

Integrating the checking part is again easier since the response generator is implemented directly in the environment. It is only necessary to invoke the insertion function, identifying the stream this response has been observed on.

DataStream_SB/tb_env.sv

```
...
class tb_env extends vmm_env;
   ...
```

```
   virtual function start();
      ...
      foreach (this.resp_chan[i]) begin
         int j = i;
         fork
            forever begin
               apb_rw tr;
               this.resp_chan[j].peek(tr);
               this.m2s.expect_in_order(tr, j);
               if (tr.kind == apb_rw::READ) begin
                  tr.data = $random(); // Poor random strategy!
                  this.s2m.insert(tr, vmm_sb_ds::INPUT,
                                  .inp_stream_id(j));
               end
               this.resp_chan[j].get(tr);
            end
         join_none
      end
   endfunction: start
   ...
endclass: tb_env
```

Step 6: Congratulations!

You have now completed the development and integration of not one but two scoreboards using the VMM Data Stream Scoreboard application package. You can verify the correct operation of the design by simulating the now self-checking verification environment.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer transactors, verification environments or integrate a Register Abstraction Layer model.