SystemVerilog FrameWorks™ Scoreboard: An Open Source Implementation Using UVM

Dr. Ambar Sarkar
Chief Verification Technologist
Paradigm Works, Inc.
Andover, MA, USA
ambar.sarkar@paradigm-works.com

1 ABSTRACT

Scoreboarding is a critical function required of a verification environment. While much progress has been made in standardizing verification environments with the release of Accellera's Universal Verification Methodology(UVM)[1], no standardized scoreboarding implementation is currently available. In this paper, we describe an open source SystemVerilog scoreboarding utility implemented using the UVM base class library.

Categories and Subject Descriptors
B.7.2 [**Design Aids**]: Hardware.Intergrated Circuits.Design Aids - Verification.

General Terms
Algorithms, Standardization, Languages, Verification.

Keywords
Scoreboard, Open Source, SystemVerilog, UVM, Verification Methodology, Verification Environment

2 INTRODUCTION

Scoreboarding is a fairly straightforward concept used in functional verification environments.

Simply put, when an event of interest is anticipated by the verification environment, details of that event are posted to the scoreboard. Conversely, when an event of interest is actually observed, it is checked against the events already posted on the scoreboard. The mapping of the posted to the checked events is called the transfer function, which can range from the fairly straightforward to the fairly complex. For example, for every posted event, there may be multiple events that are checked and vice versa.

A verification project typically requires the following features from its scoreboarding implementation:

- In-order and out-of-order checking
- Timeout checking
- Hooks for error handling
- Support for complex transfer functions

Interestingly, the soon to be announced UVM 1.0 release[1] does not yet provide a generic implementation of a scoreboard that supports these features. While there is a base class called uvm_scoreboard, it is left up to the user to implement its entire functionality. Users will thus end up creating home-grown versions of the scoreboard, and we feel a generic scoreboarding class implementation is in order.

To address this need, we have contributed the SystemVerilog FrameWorks™ Scoreboard(SVF Scoreboard) package, an open source implementation, to the UVM World website[2]. The contributed package is implemented in SystemVerilog and supports many of the features required of a typical scoreboard. It has been derived from the OVM version [3], which has seen more than 450 downloads by the OVM user community at large. The UVM version is currently being used by several of our own clients. In this article, we discuss some of the features of the SVF scoreboard functionality and show how the verification engineer can quickly integrate scoreboarding into their verification environment. We encourage the reader to download and explore this contribution and welcome feedback [5].

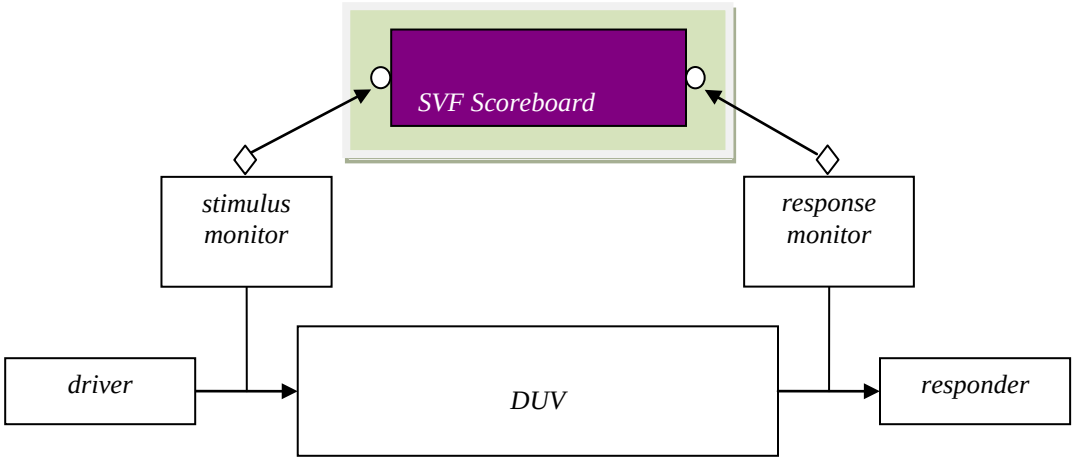

Figure 1. Simple Scoreboard Use Model

3 SVF SCOREBOARD USE MODEL

In this section, we present two typical use models of the SVF scoreboard: a simple and an advanced use scenario. These are generic scenarios and can be customized to any extent depending on the needs of the project.

Figure 1 illustrates a simple use scenario, where an instance of the SVF scoreboard can be pretty much dropped into an existing environment. The SVF scoreboard in Figure 2 is an instance of ***pw_scoreboard*** class and is derived from the *uvm_scoreboard* class. The *uvm_scoreboard* is an empty built-in base component in UVM library. Transactions are posted to the scoreboard as instances of classes derived from the *uvm_transaction* class. Similarly, transactions are checked by passing instances of classes derived from *uvm_transaction* to the scoreboard. The actual comparison takes place by calling the *uvm_compare* method of the posted object with the instance of the checked object as an argument.

In the simple use model, the transactions are being sent from the driver, and expected to be transmitted as driven in a simple in-order fashion through the design under verification (DUV). A monitor sits on the stimulus side, and publishes observed stimulus to its analysis port. The analysis port is connected to the scoreboard and the expected values get posts whenever a transaction is observed. For its counterpart, another monitor sits on the response side, which publishes the observed transactions. When a DUV response is published by this monitor, it gets checked against the posted values in order.

Figure 2 shows an advanced use model of the SVF Scoreboard. Here, the mapping between the posted stimulus to the expected response is not so straightforward and requires some manipulation of

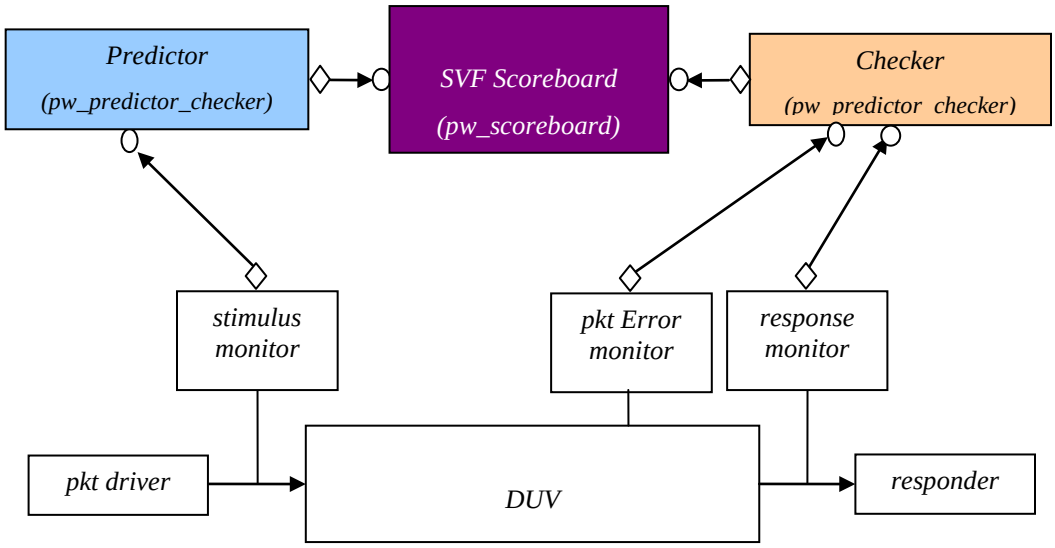

Figure 2. Advanced Scoreboard Use Model

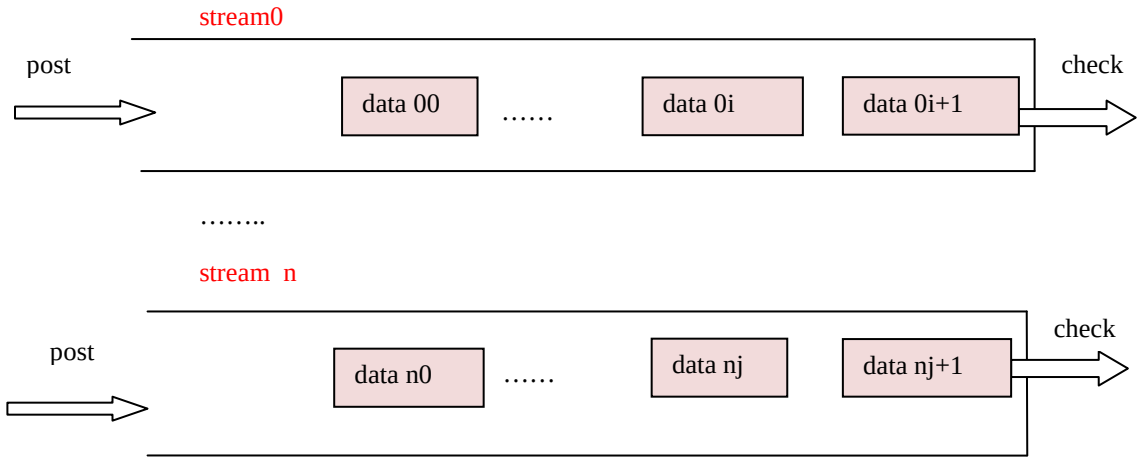

Figure 3. Multi-stream Posting and Checking

both the stimulus and the response data to infer the actual data being checked.

As an example, consider a packet driver that sends in a sequence of packet fragments. The predictor collects the fragments being sent and posts the complete packet data to the scoreboard once it has seen the end of the packet. On the other side, the response monitor sees the outgoing fragments and sends them to the checker. When the checker finally assembles the entire packet, it sends it to the scoreboard for comparison. However, the checker also gets the input from the error monitor and marks whether an error occurred so that the affected data can be compared appropriately with the posted value. This approach of separating the prediction and checking transfer functions in a separate component offers a powerful and generic methodology for scoreboarding. The details of the predictor and checker are described later in Section 4.

4 MULTI-STREAM POSTING AND CHECKING

Self-checking verification environments often need to support the notion of streams. Each stream identifies a sequence of transactions that should appear at the checking end in the same order as they are posted. However, ordering between events does not matter between events in independent streams. For example, data being set from a host to two bulk endpoints may be independent of each other; the host application may not care if one of the endpoints gets some of its transactions before the other. Also, depending on the DUT, the number of streams concurrently active can be fixed or can change dynamically.

The SVF scoreboard supports the notion of an arbitrary number of streams (Figure 3) that can appear dynamically. In-order checking is accomplished by posting objects to the same stream. Out-of-order checking is accomplished by posting objects to different streams. Each stream is identified by a unique number, and objects posted to a given stream are expected to be checked in the same order as they are posted. There is no implied order between objects posted in different streams. Thus, objects can appear in any order with respect to each other if they are posted in separate streams.

5 REPORTING AND STATISTICS

The SVF scoreboard generates error messages when mismatches occur. In addition, it also provides methods to report the activity statistics of the scoreboard. For example, it can report how many events have been posted or checked so far, or how many events are left unmatched. These statistics are available at any time during the test execution, and an error can be generated at the end of a test if elements remain unmatched.

6 ADVANCE TRANSFER FUNCTIONS

When data is transmitted from one interface to a different kind of interface, a complex transfer function may be required to represent the relationship between one data type to another data type. Although the transfer function will be very much design specific, it can still be done in a consistent and systematic manner. By providing hooks to allow design specific transfer functions, the scoreboard can be highly reusable.

SVF scoreboard provides a **pw_predictor_checker** class to handle various transfer function(s) between data being transmitted and data being received. The **pw_predictor_checker** class allows a user-defined transformation of data to take place in a testbench component that is distinct from and feeds the **pw_scoreboard**. This allows the application specific

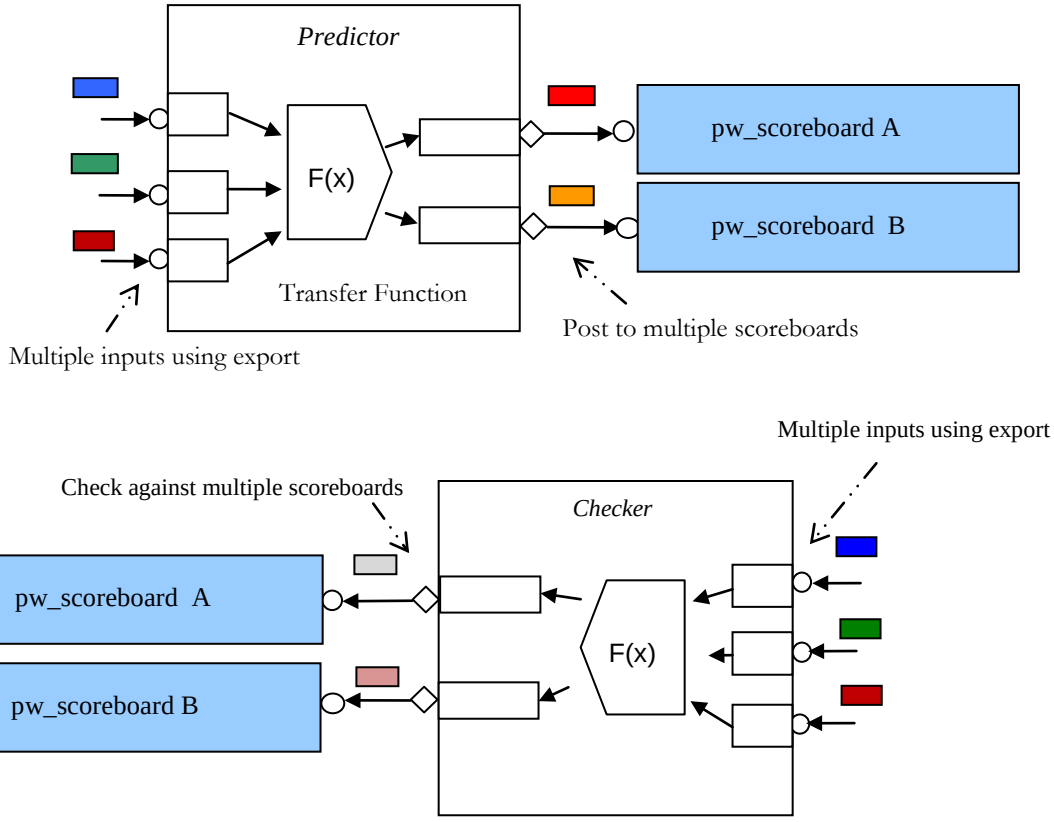

Figure 4. Predictor and Checker Function

complex transfer logic to be encapsulated separately from the generic scoreboarding functions.

Figure 4 respectively illustrates how the predictor and checker objects implement complex transfer functions. The predictor object may receive stimulus data from multiple sources using its exports. Depending on the application need, it then infers the appropriate data and posts it to any of the connected scoreboard instances as needed. On the other hand, the checker object, analogously, receives observed response data from multiple sources through its exports, and then, as the application dictates, forwards the inferred response data to the appropriate scoreboard for checking.

7 PROCEDURAL vs. TLM INTERFACES

TLM ports are the recommended way to communicate between components in a verification environment. Using TLM ports promotes better reuse of the components since it decouples the functionality of the component from how it communicates with others. Thus the same functionality can be easily ported across multiple environments as long as TLM is used as the basic mechanism for communication. The SVF scoreboard supports TLM based communication between the scoreboard and any other components in the verification environment.

However, in certain cases, it may be necessary to access the scoreboard directly without going through the TLM ports/exports. A typical example is the case where events are posted or checked using callbacks. The SVF scoreboard provides **post_sb_data()** and **check_sb_data()** for such purposes. These methods can be called procedurally in the testbench. Although we provide these procedural methods for flexibility, we recommend that users utilize TLM interfaces in order to promote code reuse.

```
// The following shows how the base scoreboard related classes can be hooked up in a UVM testbench
class xbus_demo_env extends uvm_env;

    // Declare a scoreboard.  Both input and output data are of type  xbus_transfer
    pw_scoreboard  #(xbus_transfer, xbus_transfer) pw_sb;

    // Declare a predictor. Number of input ports is '1', number of output  ports is '1'.
    // Input and output data types are  both xbus_transfer
    pw_predictor_checker    #(1,1,xbus_transfer,xbus_transfer)      pw_predictor;
    // Declare a checker.  Number of input ports is '1', number of output  ports  is '1'.
    // Input and output data types are  both xbus_transfer
    pw_predictor_checker    #(1,1,xbus_transfer,xbus_transfer)     pw_checker;
    …
    virtual function void build();
        super.build();
        ….
        // Instantiate the scoreboard
        pw_sb =    pw_scoreboard #(xbus_transfer,xbus_transfer)::type_id::create("pw_sb",this);
        // Instantiate the predictor and checker component
        pw_predictor=  pw_predictor_checker #(1,1,xbus_transfer,xbus_transfer)
                                        ::type_id::create("pw_predictor",this);
        pw_checker=  pw_predictor_checker#(1,1,xbus_transfer,xbus_transfer)
                        ::type_id::create("pw_checker",this);
    endfunction

    function void connect();
        …
        // Connect the prediction side monitor to predictor
        xbus0.master[0].monitor.item_collected_port.connect(pw_predictor.inp_exports[0]);
        // Connect the checking side monitor to checker
        xbus0.slaves[0].monitor.pw_item_collected_port.connect(pw_checker.inp_exports[0]);
        // Connect the predictor and checker to the scoreboard
        pw_checker.sb_aports[0].connect(pw_sb.post_export);
        pw_predictor.sb_aports[0].connect(pw_sb.check_export);
    endfunction

    function void report();
        …
        // Report any outstanding entries
        pw_sb.report_sb(
            1,  // Checks outstanding elements and errors if any
            1   // Prints out outstanding elements;
        );
    endfunction

endclass
```

Figure 5. Hooking up the Scoreboard Components

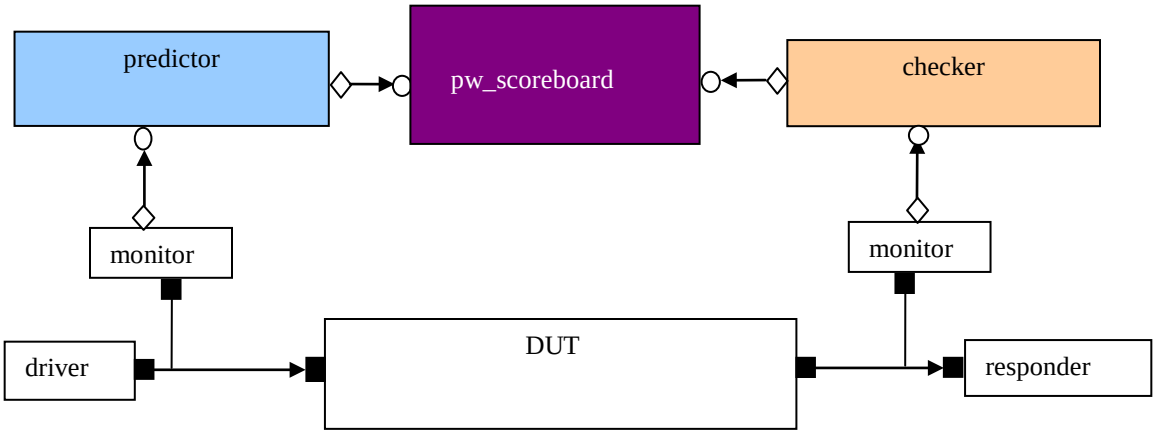

Figure 6. Example of A Typical Scoreboard Hook-up

8 EXAMPLE

Figure 5 shows the basic steps in hooking up the scoreboard for a typical example as shown in Figure 6. In this example, two pw_predictor_checker class instances are created. The instance connected to the monitor at the input side (left) of the DUT works as the predictor. The predictor is responsible for converting the input data type to the expected output data type. The instance connected to the monitor at the output side (right) of the DUT works as the checker. The checker is responsible for comparing the observed output data with the expected data on the scoreboard. The predictor connects to the pw_scoreboard's post_export and the checker connects to the check_export of the scoreboard.

The **pw_predictor_checker** class provides a built-in virtual method **transfer()** to handle different input and output data type**.** By default, input and output data type of the **transfer()** method are the same. Figure 7 shows the default **transfer()** method. Actual transfer functions may be quite complex and dependent upon mirrored images of DUT state and multiple modes or configurations.

Finally, at the end of the test, statistics such as the number of posted and checked events are reported. If any unmatched events are found, an error is generated. As expected, the example above shows how the scoreboard reporting method is called in the **report()** phase of the simulation.

8.1 Advanced Usage Example: Posting Scoreboard Data with Timeout Events

Figure 8 shows the user can set up timeout events associated with the posting of transactions to **post_sb_data**(). The basic approach is to define an uvm_event that gets triggered upon timeout. If a posted event is not matched before the event gets triggered, the scoreboard reports a timeout error.

The code snippet shows how the monitor creates an uvm_event via new and posts it to the scoreboard. This event is associated solely with the specific transaction being posted. Thus, each posted element can have its own timeout, if needed. In this example, if the packet is not matched or unintentionally dropped before 300 time units expire, the scoreboard will report an error similar to this:

```
# UVM_ERROR @ 1108:
uvm_test_top.pwr_demo_sve0.pw_sb[0] [] Timed
out on event : for transaction:a:1 p:2 r:10 l:
10 [16_b9_74_64_fc_cc_c9_b3_b4_fc] parity :
0x1e
```

```
// Transfer function that processes  the arrived transaction. Also specifies the port id at which it arrived
virtual task transfer(T_INP trans, int port_id);

        uvm_report_message( "pw_predictor_checker",   $psprintf("SB: transfer: %d\n", port_id));

        // Implement specific transfer logic


        // Push it to the SB
        uvm_report_message( "pw_predictor_checker:",   $psprintf("%d porti_id", port_id));

        sb_aports[port_id % NUM_SB].write(trans);

endtask // transfer
```

Figure 7. Default Transfer Function

```
class xbus_pw_scoreboard  #( type T_POSTED=uvm_transaction,
                             type T_CHECKED=uvm_transaction)
          extends pw_scoreboard #(T_POSTED, T_CHECKED);


     // This function returns a event that triggers a timeout event. The scoreboard will
     //  generate an error if tjis event is triggered before a match is found
     virtual function uvm_event get_timeout(uvm_transaction posted);
          uvm_event xbus_to_ev;
          xbus_to_ev = new("XBUS_PW_TO");
          fork
               #300;
               xbus_to_ev.trigger();
          join _none
      endfunction
endclass
```

Figure 8. Generating Failure on Matching Timeout

8.2 Advanced Usage Example: Posting and Checking with Support for Dropping Packets

In many networking applications, some amount of packet losses is tolerated. For example, under heavy load conditions, some packets may be dropped due to buffer overruns, as long as they are dropped within specified limits and depending on traffic and other parameters. These scenarios are quite hard to verify, since it is hard to predict which packets can be allowed to be dropped and when it is allowable. The alternative to use directed tests is often sub-optimal, since one may miss a lot of corner cases that may be exposed under such heavy traffic.

Some of the typical techniques to address packet dropping used by engineers are:

a. Marking individual posted packets as droppable and add additional checking logic to ignore such packet

b. Specifying a window and a limit of droppable packets. The checking logic allows some degree of mismatch to occur within certain window of time.

c. Checking the state of the DUT at the precise time when the event is observed and infer if the expected packet was likely to be dropped and the observed packet may be something else that follows.

Figure 9 shows how one can use c. above. By inheriting from the class **pw_scoreboard**, user code can override the **get_canDrop()** method to determine whether a sb_entry is droppable at the time it is checked.

```
class acme_pw_scoreboard extends pw_scoreboard;
          `uvm_component_utils_begin(acme_pw_scoreboard)
   …
   // Allow packets to be dropped before 200ns of simulation time
   virtual function int get_canDrop(uvm_transaction posted);
        if ( $time < 200ns) begin
            uvm_report_info("", "In overloaded get_canDrop, about to return 1.");
            return 1;
       end  else begin
            uvm_report_info("", "In overloaded get_canDrop, about to return 0.");
            return 0;
       end
   endfunction
endclass : acme_pw_scoreboard
```

Figure 9. Implementing Packet Drop.

9 SUMMARY

SVF Scoreboard is a useful package for the practicing verification engineer. It supports the scoreboarding needs of a typical verification project, can be easily extended, and can be quickly integrated into a new or an existing UVM environment. Feel free to download and use it in your current project, and send any feedback to [5].

10 REFERENCES

[1] Accellera Verification Intellectual Property Technical Subcommittee
http://www.accellera.org/activities/vip

[2] UVM World Contribution site:
http://www.uvmworld.org/contributions.php

[3] OVM World Contribution site:
http://www.ovmworld.org/contributions.php

[4] SVF Scoreboard Utility directly URL:
http://tinyurl.com/pw-uvm-scoreboard

[5] SVF Scoreboard Feedback/Issue reporting -
https://sourceforge.net/apps/mantisbt/pwsvf