Keeping Score

Part 1 of 2 - Architectural Considerations for Scoreboard Design

Gordon Allan
Verification Methodologist
Mentor Graphics Corp
Fremont, California
gordon_allan@mentor.com

***Abstract*— Verification environments for today's module, subsystem, and SoC DUTs are complex because the DUTs being verified are increasingly complex.**

Either the functions being performed in hardware are more convoluted in algorithm or data aggregation/multiplexing, or they are more general-purpose, configurable by software to perform a multiplicity of varying tasks within a function domain, or by parameterization to enable reuse in a variety of different products or configurations of an end product.

The complexity is mitigated by techniques and methodologies, some necessary: Constrained Random stimulus, Intelligent Testbench Automation, Coverage-Driven Verification, and by frameworks that assist the more rapid development and debug of those techniques, for example the UVM.

However, one area of testbench complexity is only minimally assisted by tools or OOP libraries: the development of *Scoreboards* and related modeling components used to check expected DUT behavior or output given arbitrary sets of inputs, existing state of the DUT, configuration values, both in terms of DUT parameters and soft configuration or current register/memory state values.

The area of S*coreboarding* may be the last remaining portion of testbench design which relies on Verification Experience and which cannot be solved by a one size fits all methodology.

So how can we accelerate the scoreboard design/development process or at least help it keep pace with the more complex DUT scenarios that we encounter?

In this, the first part of our Scoreboarding paper, we assist the productive application of verification experience to the architectural design of the Scoreboard portion of a verification environment.

We do this by defining a number of architectural patterns which require various possible topologies of the data comparison, which to some extent can reflect the topologies found in dataflows within the DUT, but with the intention of providing a modular scoreboard design.

Keywords: scoreboard, checker, testbench architecture, verification patterns, modeling, systemverilog, OOP, UVM

Sponsored by Mentor Graphics Corp

I. INTRODUCTION

This paper is the first of a two-part series on the topic of "Keeping Score" and covers the architectural design considerations for scoreboards.

The second half will provide examples of the most common architectural arrangements using OOP and SystemVerilog language features as required.

Scoreboarding problems can become as complex as the DUT in question, with various combinations of the following features:

- comparison that has to span time, not resolved instantly or within the inbound protocol data, may require multiple reports
- multiple interfaces, either homogeneous or heterogeneous, contributing to the output, perhaps with config parameterization
- out of order processing, depending on both input state and cumulative DUT state, e.g. to optimize occupancy or latency
- arbitrated priorities, with various generic arbitration or custom selection mechanisms affecting predictability of the DUT outcome
- data aggregation, subdivision, and reservation across time and multiple stimulus, matching such schemes in the design.

Often the challenge is to model just what is required to make the 'pass/fail' comparison, and no more, otherwise the scoreboard can easily grow more complex than the DUT. When this happens, the scoreboard becomes brittle, and requires maintenance, even after the original verification engineer or team has moved on.

Evidence from this author's experience suggests a high proportion of bugs found during the peak of the debug cycle are scoreboard bugs, unless steps are taken to design a scoreboard that is tolerant of variation, within the fixed requirements that are to be verified. A modular architecture reduces brittleness, and simplifies comprehension.

II. SCOREBOARD CONCEPTS

We begin with some definitions and a relative placement of concepts to build upon:

A. *Our Definition of a Scoreboard*

A scoreboard is an element of the self-checking verification environment, responsible for performing comparison tests on observed behavior across two or more interfaces to the design under test, in order to report conformance to the specification and for gathering and storing all the data required in order to perform those tests. It directly influences the pass or fail outcome of the test. It is passive and vertically reusable.

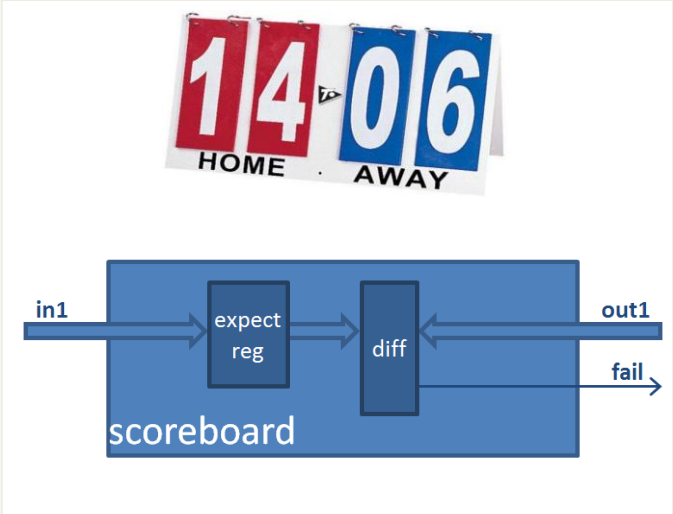

Figure 1. Simple example 'textbook' scoreboard

1) *Simplistic textbook definition*

At the simplest level, a scoreboard is a data structure that stores an expected value for a DUT interface port X based on an observed value on another DUT port A. When the expected value of X is finally observed, the stored value of X can be validated, pass or fail processed, and the data structure readied for future comparisons.

Bergeron[1] limits the definition of a scoreboard as 'the data structure used to hold the expected data for ease of comparison against the monitored output values' and advises developers to consult established software engineering texts to choose relevant structures for the design behavior at hand. He implies that any transformation, consolidation, or preparation of data before presenting it for storage/comparison is done outside of the 'scoreboard' proper. Other authors (Iman[2] and Palnitkar[3]) follow a similar line.

2) *Limitations of simple scoreboard architecture*

We contend that this simple model is only valid for the minority of projects that satisfy the following three criteria: (1) the data flow of the DUT is of a linear nature from port A to port X, and (2) there exists a Golden Reference model or well-defined Transfer Function model specification converting traffic on input A to the corresponding output value on X, and (3) the Verification team is as well versed in Software Engineering and Data Structures technology as they are in Hardware Design & Verification.

Only in this idealistic case can the scoreboard be a simple store-and-compare data structure - comparing apples with apples - observed X versus expected X. This model has limitations which becomes increasingly apparent as we explore typical requirements for data flow in a design.

3) *More realistic definition*

For today's real-world complex module and SoC subsystem designs, we need better guidance in constructing the checker/model/scoreboard subsystem of our verification environment. Typical DUTs will apply much more convoluted transformations from input A (and B, C,...) to output X (and Y, Z,...) leading to some complex interactions between the model/transfer functions (algorithms) and the data structure requirements of the scoreboard (storage). It often becomes impossible to separate the two concerns with a simple TLM pipeline connecting them.

In the worst case, a project team will commence design and coding with the expectation of simplistic scoreboard architecture, and hit limitations, creating spaghetti code along the way to work around them.

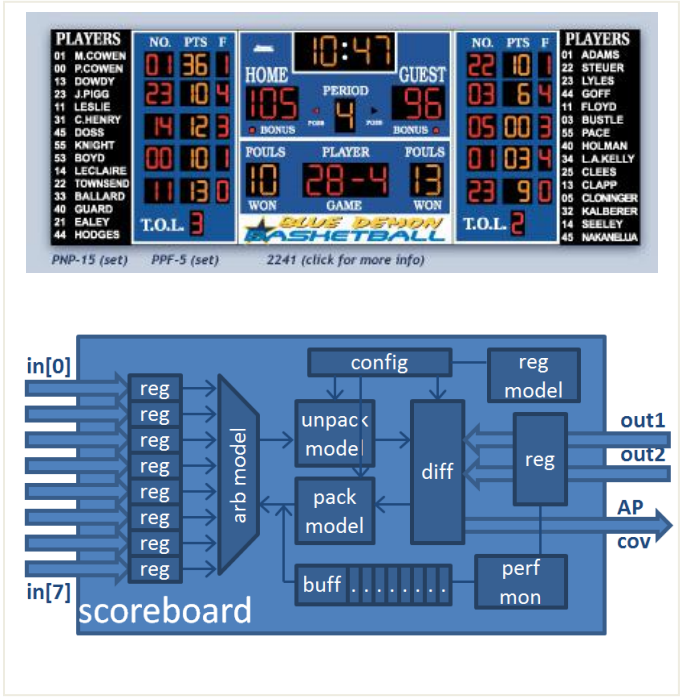

Figure 2. In the real world, a scoreboard does so much more

We seek innovative ways to inform development teams and equip them up front with a toolbox of architectural elements which answer their questions as they develop their self-checking testbench.

B. *Scoreboard versus 'Reference Model'*

Some designs have an available Golden Reference Model, that is, a model which is trusted as an accurate implementation of the specification, usually a behavioral implementation, not necessarily a timing-accurate one.

1) *Model instantated separately from scoreboard*

Such a model is useful if it can be instantiated in the testbench as part of an end-to-end scoreboarding / checking solution.
For a DUT with input A, B and output X, a golden model may compute values of X for any A and B. As such it can be used with a scoreboard to observe the actual A, B, X reports from the DUT interface monitors and check them against the model.

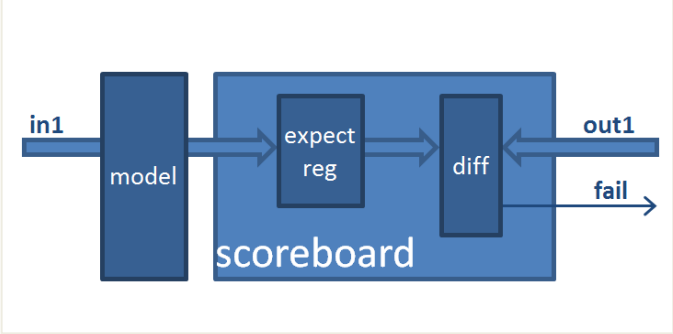

Figure 3. inline modeling outside scoreboard

Some would say that the model in this case lies outside the scoreboard, so that the scoreboard is simplified to comparing like-for-like, i.e. actual observed X versus expected X.

2) *Model tightly bound to scoreboard*

This author prefers to think of the model as an element plugged into the scoreboard, such that the overall component we call Scoreboard (or Checker) is really a dotted line around all storage/comparison elements plus all elements that assist in the preparation of that comparison. We describe later that there can be many different kinds of preparation required in order to achieve that ultimate like-for-like comparison. Scoreboarding encompasses all of that.

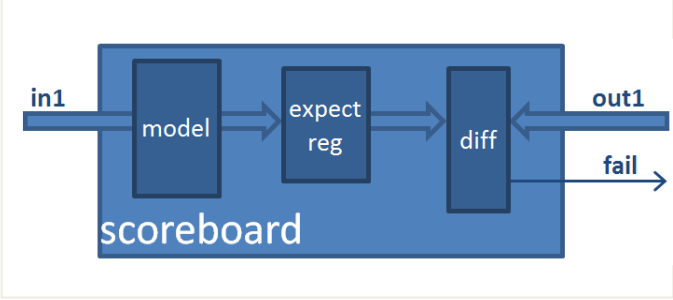

Figure 4. models integrated with scoreboard

It is recognized that two opposing schools of thought exist in the industry. One defense of the 'scoreboard encompasses all' approach, is that the collection of {models, transfer functions, data preparation, scoreboard data structure} are better built as one reusable component, than N separate parts with a fairly complex interconnect. This enables easy vertical or horizontal reuse. Another argument for the inclusive approach is a project planning one: the scoreboard is a complex piece of development, not just an array of handles. If dogma prevails, then try referring to it as a Scoreboard subsystem, or as a Checker (containing models and scoreboards)

C. *Scoreboard versus 'Checker'*

Various elements of a modern verification environment are referred to as 'checkers'. This author's definition of a checker focuses on its output: the pass or fail status of a simulation. All kinds of checker can cause a fail outcome.

Scoreboards are indeed checkers: they take in two or more inputs and check them against each other for consistency and expected cause and effect.

Checkers can also exist within a single interface VIP, performing self-consistency checks while that VIP abstracts a transaction from the pin transitions it observes.

Checkers can also exist within a pin interface, often coded as assertions, performing low-level protocol checks - request is followed by acknowledge, for example.

Checkers can also exist within predictive models such as a register model: the user sets a predicted model value, and whenever the bus reports a read/write to a matching address, the data value is compared with the model.

All kinds of checker are intended to be standalone, passive, reusable. Scoreboards, together with the others mentioned, provide a comprehensive self-checking testbench from low-level pin validation to highest-level expectations.

D. *Scoreboard Methodology vs Bespoke Design*

Is it possible to have a 'methodology' for Scoreboard creation? Methodology is borne out of practice, leading to best practice, leading to extracted patterns and documented techniques.

Each DUT certainly requires a different approach, and so at first the problem of scoreboard architecture and design appears to need a bespoke design approach.

We will explore here, how the typical testbench development cycle can experience pitfalls along the way while adding functionality to the scoreboard. We claim that an overly bespoke incremental approach may be counter-productive, and that a methodology-driven approach, hooking up a set of basic predictable building blocks as an approximation of DUT behavior, may provide fastest time-to-bugs and time-to-closure, even though it is not customized tightly for all nuances of our DUT. More on this paradox as we go.

III. EXAMPLE DUT / SCOREBOARD

We define the following example module design in order to illustrate concepts relating to scoreboard design considerations and implementations. The design is a simplified representation of a computation module that possesses all the required features for illustration of the various aspects of a Scoreboard.

The design has two data input interface channels: A and B, one command input interface C, and two output interfaces X and Y. The DUT also has a configuration register CFG written from some control bus. The behavior of 'output' interfaces X and Y depends on the values of inputs A and B, the command input C, and the value of the configuration register CFG, and the cumulative value of previous stimulus on the DUT.

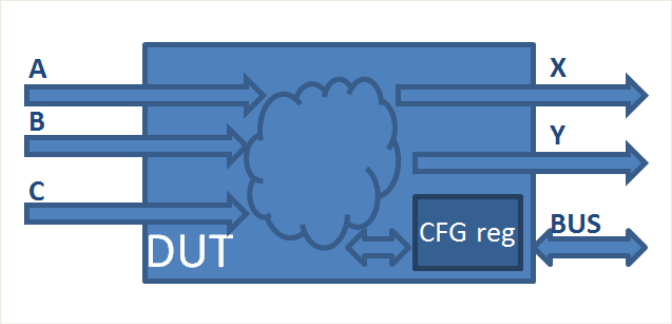

Figure 5. example DUT connections

Inputs A and B are two instances of the same interface, the transaction-level abstraction of which consisting of a data value and other fields. At the pin level, the required clocking, enabling and handshaking are implemented, but are not of concern here.

Input C is a command interface, also with relevant pin-level handshake, directing the design to perform computations on the traffic on inputs A and B leading to outputs X and Y.

Outputs X and Y are abstracted to a data value and other attribute and status fields.

Configuration CFG consists of some hard-coded configuration values and some software-writeable registers on an appropriate bus, which have an effect on the DUT's processing of data A and B and command C.

IV. SCOREBOARD DESIGN CONSIDERATIONS

There are several major behavioral dimensions to a typical scoreboard, we'll enumerate them and discuss in some detail how to implement our logic and storage.

Along the way, we will build and refer to a scoreboard 'language', like an assertion language which represents what capability we might want from our scoreboard data structure, which will help us to design it:

A. *Inputs and Outputs, Cause and Effect*

At the heart of Scoreboard design is the notion of 'cause' and 'effect' as it relates to the design under test inputs and outputs.

When defining our example DUT, we used the terms 'input' and 'output' loosely to refer to the overall direction of information flow across the DUT. Typically each interface on a DUT may consist of various groups of discrete input and output signals to form a handshake.

In a typical testbench, a protocol agent will make sense of those low-level wires and deliver an abstract representation of that transaction to the scoreboard. From the scoreboard's point of view as a 'model' observing and validating DUT behavior, the concept of what is 'stimulus' and what is 'response' may be important, and so each interface must be analyzed and designated as an abstracted DUT input, or an abstracted DUT output - as cause, or effect.

A scoreboard's design begins with this taxonomy, as the choice and implementation of dataflows and temporary storage structures within the scoreboard depends on it.

No matter what the architecture of the scoreboard, the overall flow is always the same: observe some input, derive and store some expected output, time passes, observe some output, check it against stored expectation, pass or fail, repeat.

In scoreboard 'language' terms, we can describe this basic functionality as: 'expect X at some future time', which also implies: 'at end, fail if any expected X was not matched'

B. *Temporal decoupling*

When making sense of 'cause' and 'effect' it is acknowledged that a scoreboard must deal with temporal decoupling of the various DUT behaviors that it can observe and evaluate.

Typically, those behaviors do not arrive simultaneously. Rather, they arrive in sync with their respective protocol interfaces, which may be on different clock domains, or share the same clock signal but with a pipeline latency or n-clock delay with respect to an associated input. That delay may be variable depending on cumulative and environmental factors as well as the attributes of the current DUT traffic.

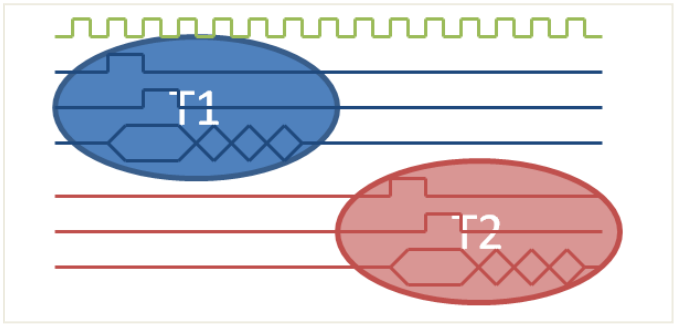

Figure 6. decoupled cause and effect

It is worthwhile to start scoreboard design with the mindset that the 'cause' and 'effect' are completely decoupled, and that the 'effect' may arrive at any time after the 'cause' has been and gone, within its intended specification range.

1) Multiple threads in scoreboard architecture

One approach is to architect scoreboard data structures that can capture event 1 and all its related information, and then move on, and separately await event 2 which may arrive at some unspecified future time. Each update can trigger a comparison or a third thread can look for matches in the data structure, match them up and make sense of them, towards a pass/fail decision.

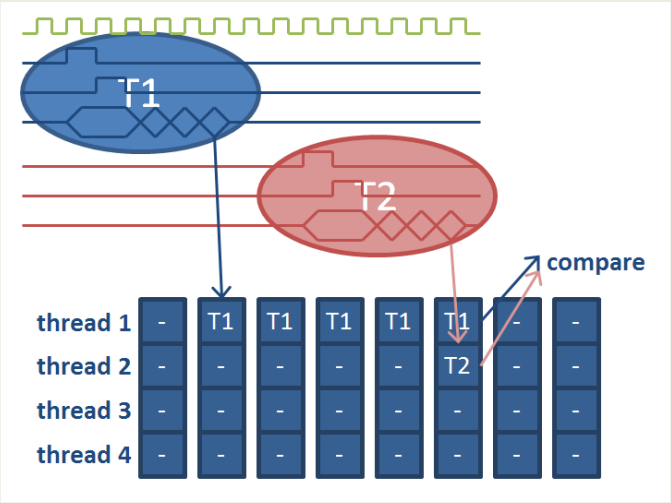

Figure 7. scoreboard threads

In scoreboard 'language', we need the description 'expect X at some future time between N and M' where N and M can be an actual time period, a number of clock cycles, or a count of packets/beats/transfers, or some specific event, as appropriate to the protocol. A catch-all timeout is always useful to avoid the scoreboard filling up with unmatched X values without yet causing a fail and bailing out.

C. Aggregation and Disaggregation

A design may combine successive inputs on one or more interfaces, process them and generate an aggregated output, or it may do the opposite: split an input transaction into a burst of individual output items, each portion distributed across time or across multiple resources. Such is the nature of many bus and communications protocols. The aggregation can often be hierarchical; multiple levels, or split one way and recombined another.

This means that there is no guarantee of a '1 input leads to 1 output' criterion for scoreboard matching. If anticipated, it is best to start the design without any assumption of a 1-to-1 mapping, and instead plan to have matching based on all required constituent parts being observed.

The scoreboard can be designed to store either an expect for the aggregate object, which is incrementally satisfied by component parts as they are observed, or can store a set of expects for the component parts. For most situations one arrangement or the other will be necessitated by factors described below, or will just seem like the natural choice.

Part of the architecture decision may be guided by the protocol, and by the capabilities of the Verification IP for the protocol. It may have appropriate native support for fragmenting and reassembly of the data, so part of the job can be delegated to the agent by configuring or using the appropriate analysis port connection for aggregate or atomic data items as required.

Some complicating factors can arise with transformations that that aggregate or disaggregate. In particular: strict vs variable order, predetermined vs incremental extent, and consecutive vs interleaved succession. See also 'Reverse Modeling' later.

1) Aggregation: Order

In the simple case, the component beats are in a strict predicted order, perhaps implicitly representing addresses, offsets, sub-addresses or lane positions, that increment, decrement, wrap around, or follow some other predetermined algorithmic progression.

Order can also be variable, perhaps due to the far end of the protocol having a latency- or throughput-optimizing algorithm or an element of caching. In this case some identifying information within each data item will guide the reassembly or dispersal of portions of the whole transaction. We discuss more about general reordering concerns as they apply to scoreboarding later, but in this particular case, it is clear that the scoreboard must maintain the entire data structure internally until it is marked complete, and cannot simply use a queue to check/compare/pass/fail each individual data beat as in the strict order case.

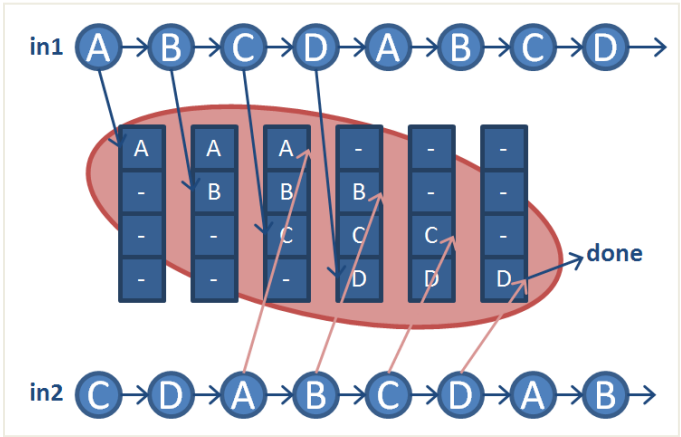

Figure 8. aggregation order

It is usual to define a method to encapsulate any mapping that is involved especially if the indexing can vary based on sizes,

extents or offsets. That method can be within the scoreboard or, if shared with drivers and monitors, can be a helper method in the data item to avoid coding it in multiple components.

2) Aggregation: Extent

Extent refers to the number of data item portions that the combined data item is split into, or composed of. Extent may be a fixed value, configured in advance, specified per transfer or a completely variable value.

In the simplest case, the extent is fixed by the protocol, or it may be predetermined by a configuration value for number of beats or size of enclosing packet, video frame, etc.

A slightly more complex logic is required when the extent can be predetermined on a compound transaction by transaction basis, for example for a burst-based bus protocol where the transaction specifies how many fixed beats will occur.

Finally, a variable extent may be required by the protocol; i.e. the extent is unknown even as the transfer is occurring, and so the scoreboard structures and logic must anticipate several possible outcomes for how to reassemble the data and when to recognize boundaries.

3) Aggregation: Succession

The final factor that can influence or complicate scoreboard architecture for a data aggregation or disaggregation path across the DUT, is the succession of data beats on the interface. A source transaction can be split into a number of destination beats that are normally guaranteed to be consecutive data items on the interface, which makes for easy reassembly and checking.

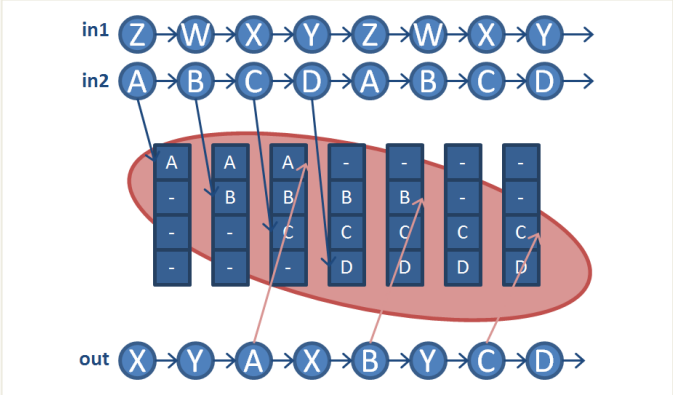

Figure 9. interleaved aggregation

In other cases, the split data items could be non-consecutive, interleaved with items from another queued transaction or with items from other channels, or with maintenance data orthogonal to and interspersed within the main traffic flow.

Again, some of these cases can be mitigated by relying on the monitoring verification IP to split and report only the traffic of interest so that the scoreboard does not need to apply further filtering. The cases of multiple channels, interleaved or arbitrated data beats are still matters for the scoreboard to untangle. We look more at arbitrated data later under 'Modeling', but here suffice to say that the scoreboard data structure will need to accommodate multiple streams of reassembly, and sometimes deal with ambiguity when a data item comes in that could match more than one stream's expected data. Arrangements for tentative matching and backtracking may have to be made.

D. Multiple Heterogeneous Interfaces

So our scoreboard data structure may already be quite a bit more complicated than just a queue or associative array and we have only really considered two inputs - one from the DUT's input A and one from the DUT's output X. Of course a scoreboard may have to process more than just two inputs from the DUT. There are always at least two, in keeping with the 'cause' and 'effect' paradigm, but for any given DUT there may be multiple causes and multiple effects.

In between those inputs and outputs, there is a myriad of possibilities of DUT behavior - combining inputs, selecting among inputs, feeding back from outputs and cumulative state, storage and delay of some inputs for later use. Very rarely does a DUT follow a single path from one input to one output.

To accommodate this, the scoreboard architect has one major decision to make and several minor ones.

The first level of design is to determine whether one scoreboard is required, or if is possible to divide and conquer by having more than one. The following scenarios allow us to consider multiple, smaller scoreboards:

1) More than one independent path across the DUT

Although rather unlikely, a DUT with multiple independent paths can have multiple independent scoreboards - there is no need to combine them into one if no awareness across those domains is required.

Even if there is some interdependence, but mostly the functionality is independent, some design simplifications can be made by having one top-level scoreboard which channels the data according to shared concerns to a number of sub-scoreboards each operating independently. Use every opportunity to divide and conquer in this way when regular or independent structures can be identified.

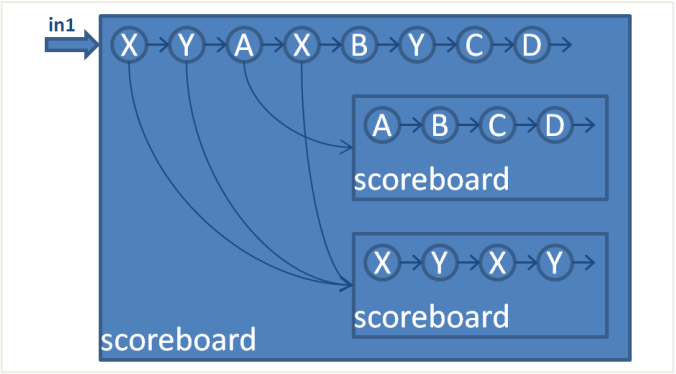

Figure 10. hierarchical scoreboards for independent paths

2) Shared midpoint interface

Divide and conquer also applies if the multiple interfaces of the DUT include a common 'midpoint' interface, for example input A leads to intermediate output X and then is further processed to create final output Y. The intermediate output X could indicate that the DUT is a subsystem consisting of two or more sub-modules in series. The intermediate interface may not be available on DUT pins, but available in a white-box fashion via internal interface, for example an identified coherency point.

In all of these cases we have the opportunity to split the scoreboard into a chain of two, one between A and X and a separate scoreboard between X and Y. Neither need know about the other, although they may share some orthogonal concerns such as configuration.

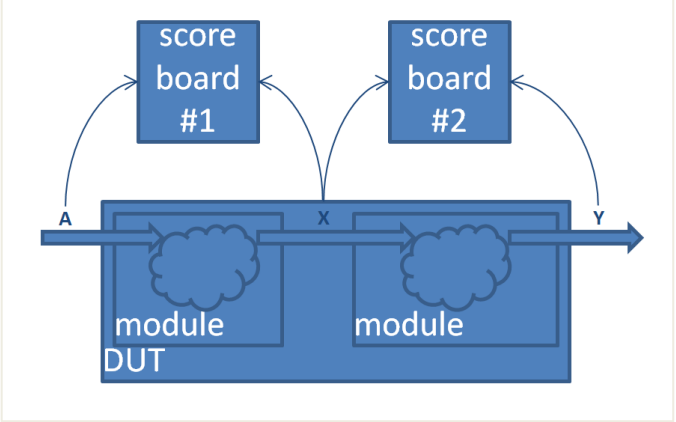

Figure 11. chained scoreboards

The opportunity for streamlined simplified design and enhanced reuse should never be ignored, always investigated

3) Micro-architecture decisions

Once the broad architecture is identified, there may still be multiple interfaces feeding one scoreboard. This informs the shape of the resulting data structures. The following design steps will help:

- Identify which interfaces are orthogonal concerns.
- Identify the dominant paths between interfaces.
- Identify causes and effects, and plan which reported data can be acted upon immediately, which must be stored in fifos for later processing or matching.
- Identify any daemon threads required to maintain internal scoreboard data checking flows.
- Check against the test plan and coverage plan to ensure your data structure can support the required checks and the data scenarios that will require them.

E. Multiple Homogeneous Channels

One special case of the 'multiple interfaces' geometry is the case where multiple peer instances of the same interface type are found, i.e. multiple channels. This situation requires several design considerations - the mapping of input channels to outputs, selection, arbitration, and also structural considerations - is the number of channels a parameter, is a generic N-to-1 or N-to-M approach needed, and finally, a means to identify or track channel identity may be required if that is not coded in the transaction.

For reuse, it is often more efficient to consider the N-to-M mapping case as a generic problem to be solved in the scoreboard architecture, even if N or M are known quantities for the current project. A hard-coded arrangement optimized for two heterogeneous channels takes about the same amount of typing and thinking as an N-channel arrangement, but the latter will be more reusable and arguably less prone to coding errors or latent bugs.

1) Preserving metadata in scoreboard

One aspect of multi-channel is the need to identify items by their channel identity during downstream scoreboard processing. The source channel number would not normally be encoded within a protocol data item. Depending on the hookup to multiple channel monitors, and the processing requirements of the scoreboard, the data items may be merged into a single data structure. In order to preserve source channel number, we must either store only in a per-channel data structure, or use composition to store a wrapper identifying the source of an item; otherwise we need to insist that extra information is added to the base item object. Each of these techniques has limitations and pitfalls in management of the data. Storing in separate data structures loses any concept of 'order' across the channels, which may be important for downstream processing. Insisting on extra attributes added to the data item is not always possible; it may come from third party verification IP.

2) Carrier object approach for metadata

A good solution for both advanced scoreboard functionality and preserving good OOP encapsulation and separation of concerns, is to create a wrapper object: a 'carrier' for the data item, which can also contain the source channel number. In fact this technique can be used for a number of scoreboard requirements that we have already identified in order to implement and convey the 'language' of the scoreboard checks.

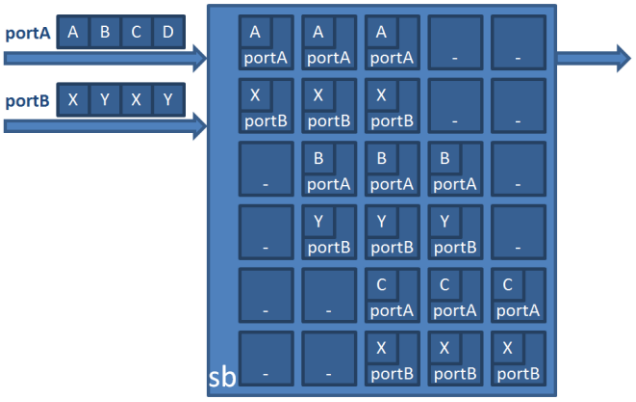

Figure 12. scoreboard with metadata

One other example of the meta-data we may wish to record is a record of the data item's arrival time, or an expiry time by which that item must be checked off, or any other timeout-related information. The scoreboard could keep separate data structures for all those bits of meta-data, but sometimes the carrier object is a clean way forward. The base scoreboard data structure then contains only carrier objects, which in turn reference the underlying data object instance.

This approach introduces some complications too. In OOP terms there is more to do to ensure instances are copied, not referenced, and to ensure clean garbage collection of past-used instances is possible in the simulator. Memory leaks may be common during development. Again, a predefined component approach may be useful here, where the mechanics of populating the data structures is taken care of by a tried and tested reusable API into which the data items, and the carrier meta-data, are slotted.

F. Reordering / Out of Order Processing

When dealing with a DUT that has multiple channels of traffic flowing through it, there is the possibility of a more complex processing order, compared to a linear DUT.

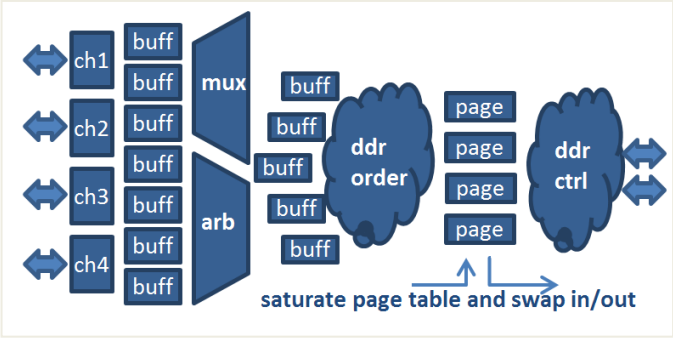

Figure 13. sdram controller reordering

Controllers for communications or memory interfaces are optimized for maximum performance, and this usually means achieving maximal occupancy of the bottleneck resource. This kind of DUT will make selection from available traffic to be processed partly based upon its cumulative state knowledge of the resource being optimized. This will often lead to re-ordering, and so of course the scoreboard needs to be aware of that, to some extent. There is always a design decision to be made about accuracy: either the scoreboard will predict the 'next expected' data item based on modeling accurately the reordering behavior, or it will adopt a more relaxed approach which posts all expected data in a structure, with the proviso that it is all matched in some arbitrary order, perhaps within time bounds.

To make this decision: refer to the verification plan. What aspects of the (re)ordering are items to be tested? Which component is responsible for that checking: the protocol monitor, or the scoreboard? Is the measurement and checking of reordering a protocol validity concern (i.e. if a particular ordering algorithm is not followed, the DUT is broken) or is it only a performance concern (there are multiple valid orderings, but some make more optimized use of resources than others)? If the check is part of the scoreboard spec, it could have a fundamental effect on architectural choices, so ensure the verification plan is complete up front and is implemented by your choices.

G. Dropped Packets and QoS

Another protocol-specific functionality/performance aspect that may influence choice of scoreboard architecture is the ability to handle dropped packets. A communications protocol may tolerate errors or missing expected packets and may have a retry mechanism for clearing up afterwards. This scheme has an implication for the scoreboard. The 'expect X' that was posted in the scoreboard data structure may be satisfied within the expected time, or may not. The scoreboard needs to determine whether that constitutes a 'fail' or whether the protocol allows for recovery of that dropped item.

In this case, the scoreboard may have a separate helper thread that cleans up the data structure based on allowable packet loss. There is no point in the packet remaining in the structure as it would cause a false negative at end of simulation. Also, if it did arrive as a 'stray' input after the expected time, we would want to be notified of that as a 'fail' too.

Some commentators suggest that whether or not a data packet is 'droppable' can be encoded up front from the test stimulus, perhaps with a flag within the data structure. It is more likely, however, that the scoreboard should encode an algorithm measuring this effect as a 'performance' metric and checking that it is within allowable bounds.

Any Quality-of-service metric can be gathered incrementally, or statistics stored for processing at end of test. We will discuss Performance measurement in more detail shortly.

H. Orthogonal Concerns

Having covered architectural aspects of the main flows of traffic across the DUT (and hence across the scoreboard) there

are several orthogonal concerns that a scoreboard needs awareness of to make the right checking decisions.

1) Configuration

While hard configuration can be coded into the logic of the scoreboard affecting all operation, soft configuration is also a concern. If the DUT has configuration registers that affect processing, the scoreboard needs to know about that. In UVM methodology, the way to do that is to provide an abstracted register model, which can keep a shadow copy of the actual DUT configuration by hooking to a bus monitor and tracking register updates. As long as the scoreboard has a handle to the register model, it can frequently refer to the current configured values as it schedules expects of predicted values.

This leads to a dilemma: what to do when configuration changes while there is traffic inflight. It is not possible or desirable to model register bit updates in a clock-accurate manner - the register model may be out of sync with the 'effect' of that register bit inside the DUT by plus or minus several clock cycles. More on 'uncertainty' follows shortly.

2) Low-power modes and clocking

Many DUTs have power and clock management as an orthogonal concern. During the lifetime of a simulation, mode changes may affect the flow of traffic across the DUT, again giving rise to uncertainty in scoreboard matching, if data is dropped or resent.

Changes in clocking should be abstracted out by the monitors feeding the scoreboard, but they may be accompanied by a period of recovery where behavior is different from normal. A good approach is to relax the scoreboard's checking if possible during such a transition.

3) Reset

Similarly, reset provides a number of challenges. The initial reset at start of simulation requires the scoreboard to be primed for operation. There may be a need to mask or ignore some traffic at this time. Some protocols may have a calibration or training phase during which specialized traffic is passed, that may or may not be reconcilable in the scoreboard.

4) Error injection

A common requirement for communications protocol testbenches is to model the effects of known error injection. If the verification IP has this capability or if it is provided via a specialized interface which can perturb normal traffic, then it is usual for the scoreboard to have a mask capability so that it does not fail the simulation for 'correct' handling of the error situation. Error injection is a specialized area and deserving of a paper in its own right.

I. Modeling DUT State

We have established that our particular use of the word 'scoreboard', and the focus of this paper, is the 'complete scoreboarding activity' including any modeling or prediction or transfer functions that are inextricably bound with the scoreboard data structures and which require design engineering effort under the banner of 'scoreboard design, development and debug'. So what kinds of concerns do we need to have solutions for in a scoreboard design?

1) Accuracy and Uncertainty

A recurring theme on the modeling side of the effort is the accuracy of the scoreboard. In a complex DUT which reorders traffic from its inputs to its outputs, the specification and algorithms for such re-ordering may be complex, or may be as simple as 'keep the memory bandwidth fully utilized'.

A scoreboard designer must create an abstraction that represents the specification being verified, and somehow codifies the main rules against which the DUT's behavior and performance can be measured. But there they need to know when to stop. A scoreboard only needs to be as accurate as the specification it is measuring - a frequent pitfall is to build in over-accurate prediction which leads to the overall checker becoming brittle and needing much debug and many iterations during attempts to bring up the whole DUT/testbench quality.

2) Modeling Uncertainty

Several of the areas discussed above as 'orthogonal concerns' alluded to the need to model uncertainty in the scoreboard.

In the simple case, simulation starts, traffic flows, each traffic item has a predictable measurement and check in the scoreboard, at simulation end all checks are complete and nothing is outstanding.

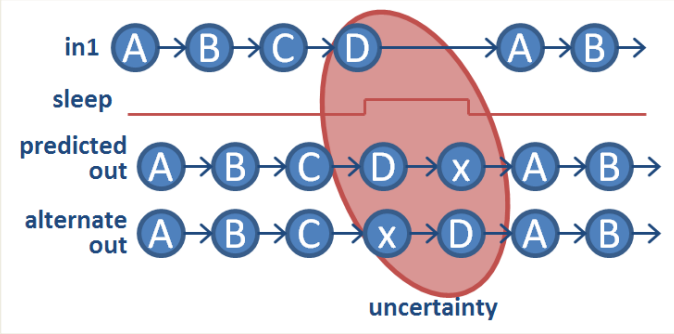

Figure 14. modeling uncertainty

When any configuration changes, mode changes, regular calibration/training phases, resets or error injection/recovery is involved along the way, disrupting the normal flow of traffic cause and effect, some logic is required in the scoreboard to accommodate that.

As ever, it is best to anticipate such a requirement upfront and not wait for the test suite to be developed that introduces that additional complexity.

If the 'carrier object' approach we described earlier is used, additional metadata can be attributed to as-yet-unmatched data items in the scoreboard. One simple approach to modeling uncertainty is to tag data items that were scheduled during a known timeframe prior to, within, and after the 'change event', with an attribute that informs the subsequent checking to apply some 'forgiveness' in the event of mismatch, or timed out or dropped packets. Otherwise they will eventually cause a fail when they time out or at end of simulation they remain unmatched.

Without a metadata capability as above, the simplest option is to turn comparison off and flush out the data structures whenever a change event occurs.

The exact approach may be protocol specific but there is opportunity for a generic scoreboard 'language' and library of modular code to provide this function as a generic or template implementation.

3) Arbitrated Priorities

Wherever there are multiple channel inputs and N-to-1 selection in a DUT, there is inevitably an arbitration algorithm as part of the specification to be verified. In many cases, the arbitration is programmable, mostly because the designers do not know what the required fairness and priority criteria are until the software layer exercises real world traffic.

So we have an often complex arbiter, choosing which channel of traffic gets to use a shared resource next, based perhaps on programmable priorities per channel, or round robin versus priority modes, or high-priority overrides embedded even in the input traffic, or other pluggable arbitration schemes.

A scoreboard needs a strategy for how to deal with that. There are several possible strategies, but only two valid ones: all or nothing. We must either (1) model the arbitration exactly and predict the output precisely, or (2) model the overall 'fairness' criteria for throughput and latency per channel, and measure reality against those rules, leaving the beat by beat prediction relaxed.

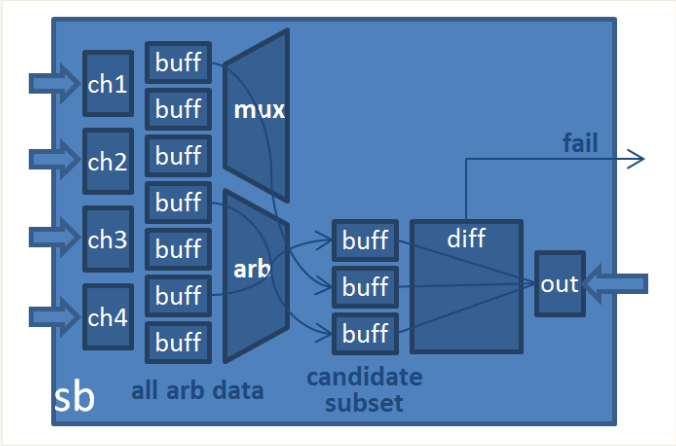

Figure 15. relaxed arbitration checking

This author prefers the latter scheme, because exact modeling hits the 'accuracy' issues mentioned earlier, and can lead to debug gridlock during bringup, when many bugs are found in the scoreboard, each requiring a patch to the model. Also, it is reasonable to use other techniques for addressing the arbiter verification portion of a test plan, such as formal / assertions, and not just rely on fragile procedural code and hitting coverage points.

So what is meant by 'relaxed prediction'? If a scoreboard is not going to model accurately and predict the one and only correct output, it must gather all the 'possibly correct' outputs and store them temporarily, adding attributes to that storage relating to the arbitration if possible, so that it can eventually check those items off as 'done', and measure the arbitration fairness as if it were a performance measurement rather than a strict pass or fail.

This functionality is related to the 'uncertainty modeling' already discussed. It is evident that it is as important to design up front how a scoreboard is going to relax checking as much as how it is going to enforce checking.

4) Cumulative State

When the DUT has cumulative state that affects native operation, this will require modeling also. If it only affects performance optimization, we discuss that shortly.

One important point is that every cumulative state has to begin somewhere, and so some resynchronization may be required after major change events such as reset, again involving specifying regions of uncertainty or relaxed checking until the state is known.

J. Performance

Performance verification is often considered somehow 'different' from Functional Verification, and often separate components are designed and created specifically to equip a test environment with performance measurements and reports.

A good scoreboard will have the means to make those measurements within. After all: they usually boil down to the following categories: bandwidth on an interface, latency across interfaces, and number of packets through / number of packets dropped or retried. All of these statistics are immediately accessible to the scoreboard by definition. Indeed the scoreboard will often measure latency as part of its time-bound matching algorithms between observed inputs and outputs.

All that is required to be added is the relevant post-simulation reporting of performance characteristics: utilization, throughput, percentage hit rates, quality of service, percentage of optimization opportunities missed.

It is often suggested to design a separate Performance monitor for such measurements, although they share a lot of underlying

logic with the scoreboard and so simulation performance can be optimized by leveraging that.

One approach is to have the scoreboard emit objects for further analysis by an external performance monitor that depends on the scoreboard. Vertical reuse is still possible if performance monitoring is required at full-chip level for example, it just requires all the dependent components (monitors, scoreboard and performance monitor) rather than a single entity.

K. *Coverage*

One often overlooked aspect of scoreboard design is support for Functional Coverage. Each individual interface connection can have its own coverage criteria defined and recorded, including some cross-coverage amongst its constituent attributes. However, cross-coverage cannot span time, so it is not possible to define covergroups from one transaction on interface A to another on interface X some clocks later.

This is where the scoreboard comes in - it is in the unique position of requiring storing of transaction A and transaction X internally, arriving at different times, and checking them off against each other. When it does that, it has a simultaneous pair of related transactions that can have some significance to Functional Coverage as defined in our test plan. The two (or more) transactions can be combined into a composite pair object containing instances of both parties. This object can have its own native coverpoints and cross-coverage, and is emitted and triggered by the scoreboard.

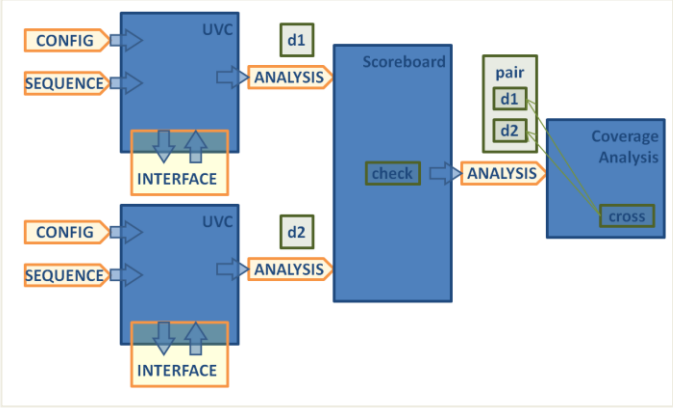

Figure 16. coverage output from scoreboard

The significance is two-fold: first, we now have a mechanism to trigger a desired cross-coverage of a transaction A with one particular attribute and a related transaction X with some other attribute, and secondly and more importantly: the validation of that functional coverage point as being meaningful by virtue of the fact that actual traffic crossed the DUT using those measured values. This aspect of functional coverage is often overlooked: coverpoints of configuration attributes or register settings, for example, can be measured, hit, and yet be completely vacuous unless they are backed up by evidence that actual traffic flowed through the DUT while those attributes were in effect. Scoreboard-generated coverage is more sincere!

V. SUMMARY

There is not a lot of written advice in our industry on the topic of Scoreboarding, in fact in this short paper we have already covered more depth and more aspects of scoreboard architecture than in the cumulative text of any of the verification books on one our typical bookshelves.

There is so much more depth that could be covered. The scoreboard is at the heart of the self-checking testbench - the components around it merely abstract the problem of individual pins wiggling, clocking, handshaking, so that the scoreboard can do checks and comparisons at a high level.

Patterns have yet to emerge to address design and coding of this area, ad hoc design and coding has prevailed. Some example modular solutions exist [4] [5] but most of the innovation is being done in SoC design houses.

We have developed solutions that may lead to standardized methodology in this area, we encourage their documentation. In the second part of this paper, we describe a library and toolkit of technology, design techniques, patterns, modular code, and advice on an approach to the scoreboard development process in a project context, in order to provide solutions for all the situations we describe in this first part.

That part of a verification project which is entitled 'develop and debug' scoreboard covers so much more than just selection and implementation of a data structure. Architect that 'scoreboard subsystem' properly - paying attention to modeling of the data patterns in your DUT - and it will become a useful tool to assist in your quest for bugs in that DUT.

ACKNOWLEDGMENTS

Thanks are due to my Verification Technology colleagues at Mentor Graphics for their contributed experiences.

REFERENCES

[1] J. Bergeron, "Writing Testbenches Using SystemVerilog", NY:Springer, 2006

[2] S. Iman, "Step-by-Step Functional Verification with SystemVerilog and OVM", CA:Hansen Brown, 2008

[3] S. Palnitkar, "Design Verification with e", NJ: Prentice-Hall, 2004

[4] G. Allan, "UVM/OVM Verification Methodology Cookbook", Mentor Graphics Corp, http://verificationacademy.com/uvm-ovm/ScoreBoard

[5] A. Sarkar, "SystemVerilog FrameWorks Scoreboard: An Open Source Implementation Using UVM", DvCon Conference Proceedings, 2011

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software", MA:AddWesley 1995