Handling Windows of Uncertainty: Reducing False Errors from Variable DUT Timing

Jeffery Vance

Verilab, Inc
Austin TX, USA

www.verilab.com

ABSTRACT

A common verification challenge is performing transaction level checks during windows of time when signal values are hard to predict. A testbench may generate predicted values too early or too late relative to the design under test, although the design behavior is nevertheless valid. This generates many false errors, increasing the effort to debug regressions and enhance testbench code. While many solutions exist for solving this problem, most have various deficiencies, typically adding leniency in the checking and increasing code complexity.

This paper proposes a generic window handler as a solution to overcome these issues in many practical situations. The window handler encapsulates timing characteristics of the hard-to-model behavior, reducing complexity of the testbench code without sacrificing rigor in transaction checks. When applied to an example design, this technique eliminated many false errors from simulations, allowing more time to find real bugs and quickly achieving error-free regressions.

Table of Contents

Table of Figures

1. Introduction

Checking design outputs becomes difficult when there is a window of time in which the exact timing of when output value changes are hard to predict. This situation can arise when outputs are checked on a transaction basis, but with imprecise timing criteria or asynchronous relationships to other events. The design requirements may allow for a range of time in which it is difficult to predetermine on which transaction a value will change, but at the end of the window, the testbench and design under test (DUT) must converge on the same deterministic value. The testbench must calculate expected transaction values to compare against the actual values throughout the window. To avoid false errors, the testbench must update the expected value at the same time the actual output changes. Any timing mismatch in the window produces an error since the testbench may be comparing an old value against the latest value from the DUT, or comparing the new value prematurely (before the DUT changes from the old value).

Figure 1 shows an example of predicting a new value (single bit) too early and too late, relative to the DUT. In either case, the design is valid and meets the timing requirements, however many false errors are generated due to the expected and actual values mismatching. The consequence is many hours of manual labor required to debug these errors, verify they are in fact allowable timing mismatches and not design bugs, and then attempt to adjust the testbench parameters to align the predictions. For many designs it is difficult to align the expected values to guarantee no false errors occur for all test cases. It is common to successfully align predictions for a few test cases, only later to discover other cases that still generate false errors. The issue becomes even more difficult if the checking code is to be reusable for multiple design implementations. Each implementation may have different timing characteristics that are still valid.

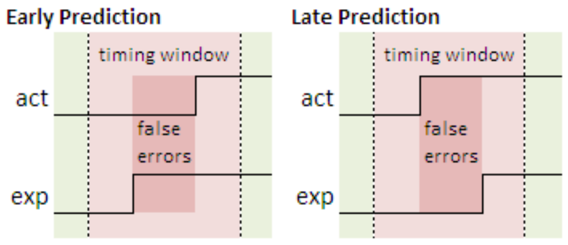

Figure 1 - False errors comparing actual and expected values

The ideal solution is for the testbench to have the flexibility to test to the full allowable window of time, guaranteeing false errors can never occur, but without too much leniency that a bug can slip through the check. Additionally, the solution cannot be so complex that the code is difficult to maintain. This paper presents a window handler solution to achieve these goals. The window handler provides tolerance of time-volatile values that would be compared against predictions that are old or premature for a given transaction. Deviations of non-volatile values are still rejected, ensuring the transaction is checked to the fullest extent it can be predicted. The window handler can be instantiated and configured in a scoreboard or monitor with minimal effort, specifying the transaction type to be checked, indicating which fields are volatile, and providing the timing window parameters such as start event and duration. The testbench can then be applied to multiple design implementations with various timing characteristics without modification.

2. Common Timing Window Solutions

There are many common solutions that have been used by verification engineers to deal with timing windows of uncertainty. This section gives a brief overview of these solutions to highlight how the window handler solution shares characteristics of some while aiming to overcome their deficiencies.

Common solutions generally have two possible issues: lax checking and high complexity. With lax checking, the concern is that the checking is too lenient during the timing window and could potentially miss bugs. Some approaches remove this leniency, but usually at the cost of higher complexity. This complexity is typically due to tight coupling between code that manages the expected data and code that monitors the status of the timing window. Not only does this make the code harder to understand and prone to errors, the code is less re-usable and has high impact from specification changes. Since the timing characteristics of the window are typically part of the check, the checking code must be modified to be re-used for other timing windows. If the timing specifications were to change, testbench updates become burdensome with many changes spread throughout the code.

The following is a list of common timing window solutions:

- **Disable Checking:** All checks associated with indeterminate transactions are disabled or suppressed. The consequence is potentially large periods where no transactions are checked, even if they partially contain deterministic values along with the indeterminate ones. It is common to use methodology class libraries for transaction checking, such as the compare() methods provided by the UVM[1]. UVM does not have a way to temporarily disable comparisons for a subset of transaction fields and manual attempts to do this can be complex.

- **Mask Checking:** Instead of disabling checks, the expected value is set to equal the actual value during the window, guaranteeing the checks will not generate an error. This approach improves on disabling checks since it allows checking of deterministic fields of a transaction to be performed while ensuring the indeterminate values don't give false errors. The disadvantage is this solution typically tightly couples to the checking and window handling code.

- **Avoid Events in Window:** It may be possible to manipulate the input stimulus to prevent outputs from occurring during the window, however bugs can be missed due to functionality not being tested during these special times. This solution is also difficult to guarantee false errors never occur for all regressions. If the timing of inputs are randomized, there is the possibility of random corner cases where an event slips into the timing window, requiring additional effort to repeatedly adjust the test constraints.

- **Queue All Possible Values:** Very often the value being checked can only be one of a few possible values. This could be limited to just two possible states: previous state and expected next state, or it could be a set of values that are equally valid during the window. In this solution, all potential values are put in a queue, generating an error if the actual data does not equal any of them. The disadvantage is often tight coupling between the checking code, the window handling code, and the specific scenario being checked.

- **Track Window Events and Check at End:** Events are recorded and checked at the end of the window. The benefit is all events are checked, however there is the common disadvantage of complexity due coupling between the window handling and checking. An additional disadvantage is checks may be performed many cycles after the transactions occurred. The simulation time of when an error is reported will not coincide with when the error actually occurred. This creates a semantic coupling between the log and the testbench code. Someone reviewing the log must have knowledge of the window handling code in order to understand the sequence of events and debug the errors.

- **Make Testbench Match DUT:** Observe the design implementation timing in a simulation and then hardcode the testbench to expect the exact same timing. The disadvantage is if the requirements allow for a wide window, the testbench is now testing to a stricter timing than what is truly required. Any deviation from the hardcoded timing will cause testbench errors, even if the timing is valid and meets the requirements. This solution also cannot be reused and must be customized for each design implementation.

- **White-box DUT Internal Signals:** Directly monitor an internal signal of the design to use for synchronizing testbench expectations. The disadvantage is less re-use of this solution since the dependency of internal design signals is hardcoded. A change in the design implementation details could break the testbench. Additionally, this solution could result in the testbench making the same assumptions as the DUT regarding the relationship of the internal signals, allowing a bug related to false assumptions to be missed.

The proposed window handler solution aims to overcome all of the issues that exist with the above approaches. The solution is similar to the masking approach, but with additional methods to keep the solution easy to use and portable between projects.

3. Window Handler Solution

The proposed solution is for a generic timing window handler that dynamically adjusts expected values that are checked against actual values during the duration of a window. The handler ensures the DUT does not violate any functional requirements while having the flexibility to allow for various timings within the window. Additionally, the solution is decoupled from the testbench checking code. The complexity of the timing window is abstracted so the primary testbench code is easier to understand and maintain. Another advantage of this decoupling is it allows the timing parameters to be maintained in a single location, rather than spread throughout the testbench code. This reduces impact of design changes, allowing modifications to be made in a central location, and leaving the rest of the testbench code unchanged.

General Approach

In a typical test environment there will be two variables used for checking behavior: one that stores the expected value generated by the testbench and one that stores the actual value read from the design under test. During an unknown timing window, the expected value may be calculated too soon or too late, relative to the actual value. This will produce false errors in any comparisons until the two values agree. This solution adds a third variable: a window compensated expected value. This variable will dynamically adjust the testbench expectations based on the window status and actual output value.

For this approach to work, the following behavior must occur. We will use the variable names ***act*** for the actual observed value, ***exp*** for the expected value, and ***w_exp*** for the window compensated expected value. The ***exp*** variable is managed by the testbench checking code. The ***w_exp*** variable is internally maintained by the window handler.

- The testbench shall always write expected values to ***exp***, but never use it for checking.
- The testbench shall always request the expected value for checking from the window handler.
- The handler shall always use ***w_exp*** as the expected value for checking.
- The window handler shall monitor ***act*** and ***exp***, as well as a timing window start event, using the status to determine how to set ***w_exp***.
- While outside a timing window, the window handler shall set ***w_exp*** equal to ***exp***.
- While inside a timing window, the window handler shall set ***w_exp*** equal to ***act***.
- At the end of a timing window, the window handler shall generate an error if ***exp*** does not equal ***act***.

This solution is effectively assuming that the DUT is always correct during the window and verifying the expected and actual value agree before the window ends. Outside of the timing window, the window handler is transparent, ensuring the standard expected value is always used for checking.

While this is very similar to the common solution of disabling or masking checks during the window, there are two important distinctions. The primary distinction is that the checking code does not have to be concerned with the status of the timing window. The code for checking transactions has no additional complexity compared to the case where no window exists at all. The code would simply poll the handler for assistance in performing the check. Additionally, this solution is intended to segregate volatile fields from non-volatile fields in the checking, rather than treating an entire transaction as volatile.

Single-Transition and Multi-Transition Modes

The window handler has two possible modes of operation: a single-transition mode and a multi-transition mode. In single-transition mode, the actual value is only allowed to change once within the timing window. Additional transitions will result in errors being reported. Multi-transition mode is to accommodate designs that permit multiple changes or glitches in the output during the window. As long as the value stabilizes and matches the expected value by the end of the window, no errors will be generated. The benefit of having these two modes is the ability to adjust the checking leniency to be appropriate to the requirement being verified.

In the case of single-transition mode, once the window handler observes a single output transition and confirms it matches the expected value, there is no need to continue monitoring the window. The timing window can be ended prematurely and the testbench can continue operating as if there is no window. Any additional changes in the output will be caught as a standard out-of-window error. This approach minimizes the amount of leniency in the testbench, ensuring timing windows are only as long as necessary to avoid false errors. Once deterministic behavior is guaranteed, the testbench can drop all leniency related to the window and return to "business-as-usual." The flexibility to dynamically adjust when the timing window ends is an additional benefit over other window handling solutions typically used. This helps to further reduce the chance for a bug to slip through these checks.

Figure 2 shows the possible scenarios of applying the window handler solution to a single bit value being checked. In the case of single-transition mode, the window is ended early once the actual value transitions and equals the expected value. For multi-transition mode, the entire duration of the window must be allowed to finish before returning to normal checking. In the event that there is a mismatch between actual and expected data at the end of the window, an error will be reported.

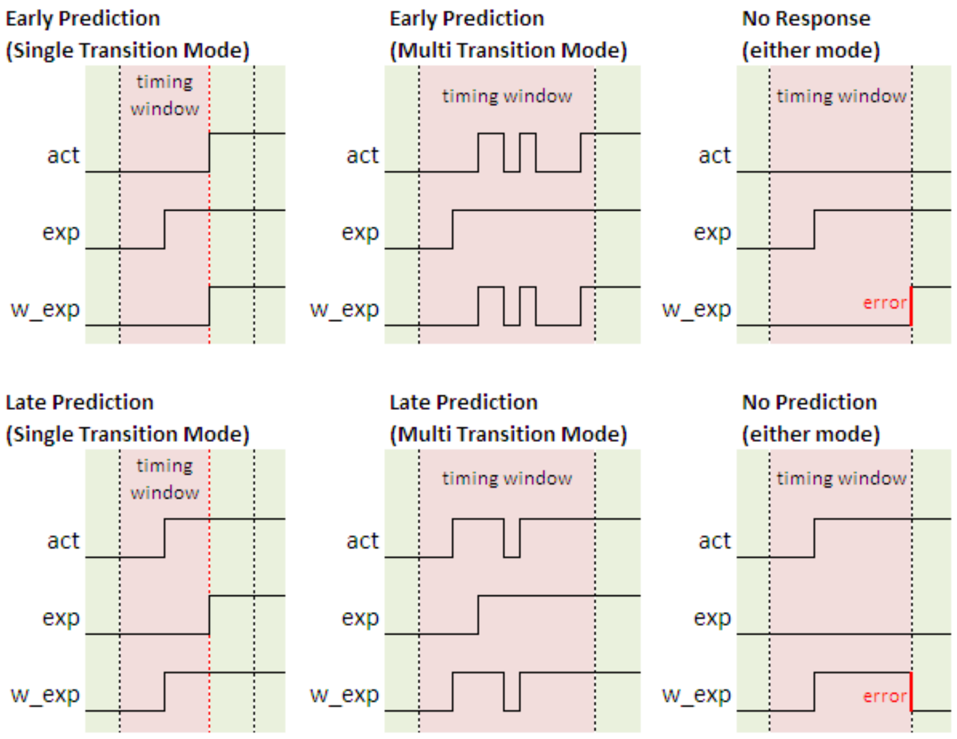

Figure 2 - Window handler checking of a bit

The window handler solution is intended for transaction objects made up of many fields to be checked in the same manner as shown for a single bit in Figure 2. Additionally, transaction objects can distinguish between members so that only the time-volatile fields are adjusted by the handler.

The window handler solution was implemented in SystemVerilog[2] and does not rely on any methodology class libraries. This solution can be used independently or alongside other methodologies such as the UVM. Although SystemVerilog was chosen for implementation, it should be noted that the general concepts described here are applicable to other verification languages.

Limitations

There are some situations where the window handler solution may not be practical for handling timing uncertainty of transactions. The window handler assumes the design output is always correct within a timing window. Any possible value during the window is considered legal until the window ends. For some designs, only a small set of values may be legal and anything else is without a doubt a bug, regardless of the timing. For these situations, it would be more ideal for the handler to run in a mode where a queue of potential valid values are checked against the actual value during the timing window. This is a potential area of future work to expanding the solution.

This solution is only useful for checking transactions where the DUT and testbench are expected to converge on matching values by the end of the window. This solution does not currently handle situations where the testbench and DUT are allowed to diverge. An example is if an interrupt is raised to a microprocessor during an unknown timing window. Depending on the timing, the microprocessor may take different paths of behavior arriving at transaction values that differ from the testbench predictions. This solution does not accommodate any divergence at the end of the timing window.

4. Using the Window Handler

The window handler is intended to be simple to integrate into a scoreboard or monitor testbench component. The complex details of managing the window are encapsulated so all that is needed to is to declare and configure the handler. Checking code can then be written essentially the same as if no timing window were to exist.

Configuration Macros and Methods

The window handler requires macros to be used to declare the name of the handler instance, the transaction type to be monitored, and which fields are volatile during the timing window. The following is an example of declaring a handler instance ***whandler*** for transaction type ***my_tr***. In this example, ***my_tr*** contains three members. Each member declaration must provide a string name to map to the member to be checked. It is therefore recommended the strings match the variable names.

```
`config_win_handler_begin(whandler, my_tr)
   `set_win_field(data1, "data1")
   `set_win_field(data2, "data2")
   `set_win_field(data3, "data3")
`config_win_handler_end
```

This series of macros will instantiate the window handler instance with appropriate parameterized functions for checking this transaction type. The decision to use macros for declaration was made to avoid coupling the window handler solution to the transaction level modeling. This implementation allows the window handler to be applied to any existing SystemVerilog based testbench without modification to the transaction class definitions.

Once declared, public interface methods of the window handler object are used for final configuration. These are as follows:

- `function void` **`set_volatile`**`(string name, bit prev_val_mode=0) –` This function specifies which fields of the transaction are volatile during the window. By default, all values are considered non-volatile, so skipping this function call is the same as not using a window handler at all. The member name must match the string provided by the configuration macro. Additionally, the prev_val_mode parameter configures this field as volatile between only previous and new value, or volatile between any value (default). This is to allow more rigor on checking multi-bit fields that are guaranteed to only deviate between a previous value. If configured this way, the field will only be allowed to deviate between those two values within the window.

- `function void` **`configure`**`(event win_event, time duration, time start_delay=0) –` This configures the window handler with a start event and timing parameters for the window. The duration specifies how long the window lasts after the start event. The start delay specifies if there is a delay between the window event trigger and the actual window start. The default for this value is 0 so that the window starts immediately with the event trigger.

- `function void` **`set_mode`**`(bit multi_trans=0) –` This function sets the mode of the handler to single or multi-transition modes as described in Section 3. The default is single transition mode, therefore this function only needs to be called for multi-transition scenarios.

Public Interface Methods

The following member functions are public interface methods to be used throughout a scoreboard or monitor. All other functions and all member variables are local to enforce proper encapsulation. This is especially important since improper access and manipulation of internal variables could lead to real bugs being missed if the window handler is masking errors at the wrong times.

- `function void` **`start`**`()` – Start window monitoring of events. Normally would call this at the start of a simulation task.
- `function void` **`stop`**`()` – Stop window monitoring. Once stopped, no tolerance will be provided for checking transactions. This may be useful if a check needs to temporarily disable windows.
- `function void` **`set_exp_data`**`(tr)` – Pass a transaction containing expected values to be checked against actual transaction data. This should be called in a scoreboard any time new expectations are calculated. Depending on the status of the window, the values may or may not be used for actual checking.
- `function void` **`set_act_data`**`(tr)` – Pass a transaction containing the actual values received from the DUT.
- `function bit` **`check_act_data`**`()` – Request the handler to compare all expected data against actual data. Expected values of volatile fields will be skipped if in the timing window.
- `function T` **`get_exp_data`**`()` – Get a transaction from the handler with all volatile members adjusted based on window status. Allows scoreboard to perform custom checking instead of relying on the handler to do the comparisons of actual vs. expected data.
- `function bit` **`in_window`**`()` – Returns 1 if currently in the timing window, or 0 if outside the window. If the handler is stopped via the *stop* method, it will always return 0.

This collection of public methods gives a scoreboard or monitor the tools necessary to take full advantage of the window handler. The scoreboard can focus on setting expected data and actual data and how to compare them, leaving the complex timing details for the window handler.

Proper execution of the window handler requires a few rules that must be followed consistently to avoid false errors. For example, it would be typical to apply this in a scoreboard that calculates expected outputs based on observed inputs and compares these values against the latest outputs. In order for the window handler to make proper adjustments, the scoreboard must provide the handler with the latest actual output values immediately as they are received (without consuming simulation time), using the *set_act_data* function. In a UVM environment, this would ideally be in the *write* function of an analysis port from the monitor component for the transaction. Additionally, the scoreboard must provide the window handler with the expected transaction values immediately upon calculation via the *set_exp_data* function. This is important since the window handler is tracking the change of actual events and predictions relative to the timing window bounds and has risk of making false adjustments with late information.

The checking of output transactions from the DUT must be performed immediately as they are received. Again, in a UVM environment, this could be in the *write* function of an analysis port for the transaction, or a function that handles the data without consuming simulation time. Any delay in checking risks crossing a timing window boundary and result in an invalid check operation.

Example with a Scoreboard

Below is a basic example of applying the window handler in a scoreboard component, comparing output transactions to expected values through the window handler. This example continues with the same instance names used earlier.

```
class example_sb;

  whandler.set_volatile("data2", 0);  //Prev value mode OFF
  whandler.set_volatile("data3", 1);  //Prev value mode ON

  event start_event;  //trigger not shown for this example

  //Window lasts 100ms, starting 50us from start_event trigger
  whandler.configure(start_event, 100ms, 50us);

  task run();
    whandler.start();
  endtask

  //On observing an output, check against expected data
  function write_mytr(my_tr out_tr);

    whandler.set_act_data(out_tr);

    if(whandler.check_act_data() == 0) begin
      $display("ERROR: Transaction mismatch");
    end
  endfunction

  //On observing an input, calculate expected outputs
  function write_input(in_tr i_tr);
    my_tr exp_tr;

    //calculate expected transaction based on current state
    calc_exp(exp_tr);
    whandler.set_exp_data(exp_tr);
  endfunction
endclass
```

In this example, the transaction class, *my_tr*, has three members: *data1*, *data2*, and *data3*. The scoreboard configures *data2* and *data3* to be volatile during the timing window, leaving *data1* to default to non-volatile. Additionally, *data3* is configured with previous value mode. This demonstrates how the window handler will treat all three values differently during the window.

- The actual value of *data1* must always equal the expected value, regardless of the window state.
- During the window, the actual value of *data2* can equal any value without generating an error. An error is only generated if the actual and expected values disagree at the end of the window.
- During the window, the actual value of *data3* can only equal the expected value or previous actual value without generating an error. Any other value during the window will generate an error.

This example also demonstrates the major goals of this approach. The scoreboard code is very simple compared to typical solutions of masking checks for periods of time. If the timing characteristics defined for the DUT change, the code performing the checks does not need to be modified. The only necessary change is the parameters passed to the *configure* function. The same change applies for re-using this scoreboard as VIP for other design implementations. Treating each transaction member differently ensures no leniency is introduced when checking the transaction as a whole. The transaction is checked to the fullest extent it can be determined during the window.

Additional Options

The scoreboard example shows a basic way of using the window handler, however there are additional ways of taking advantage of this solution and allowing flexibility for different situations. For example, timing parameters provided to the *configure* function can be further abstracted from the code by keeping them as compiler directives. If there are many window handlers throughout the testbench, all timing parameters can be maintained in a single file that is easy to maintain without disturbing testbench code.

A DUT that requires more flexible timing control of window boundaries has several options. The *start* and *stop* methods can be used at any time to temporarily pause or resume operations of the window handler. Additionally, the handler can be reconfigured with different timing parameters throughout simulation runtime. This can be used to accommodate different modes of operation where the transaction behavior can change.

5. Future Enhancements

The current window handler solution has several limitations, as described in Section 3. There are potential enhancements that would allow this to be applied to more situations. The current implementation only supports transactions composed of integral data types. This could be expanded to support more data types. During a window, the current implementation will allow the actual data to be any value or only the previous actual value (previous value mode). A more flexible solution would be to add a queued mode where a queue of legal values is passed to the handler as a reference. Another potential area of enhancement would be to explore how a window handler could accommodate divergence of values between the DUT and testbench at the end of a window. This could potentially help avoid false errors from the DUT taking a different path from the testbench due to interrupts or other events in the window.

6. Conclusion

This paper has presented a window handler solution for avoiding false errors during windows of transaction timing uncertainty. It has demonstrated how the complex details of timing can be separated from transaction-based checks so the code is easier to write, maintain and reuse. The solution can be applied to existing SystemVerilog based testbenches with minimal modification (source code will be made available from www.verilab.com/resources). A scoreboard example was used to demonstrate how the window handler makes an easy solution to what is typically a complex and time consuming problem. This solution ensures regressions remain free of false errors without sacrificing rigor of checking that is performed. This ultimately allows testbenches to be developed more rapidly and bugs to be discovered sooner.

7. References

[1] Accellera Systems Initiative, "UVM (Universal Verification Methodology," [Online]. Available: http://www.accellera.org/downloads/standards/uvm.

[2] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.