Verification Mind Games

how to think like a verifier

Jeffrey Montesano
Verilab
Montreal, Canada
jeff.montesano@verilab.com

Mark Litterick
Verilab
Munich, Germany
mark.litterick@verilab.com

***Abstract* — Effective verification requires engineers to approach problems using a very different way of thinking compared to that normally applied by designers. Specifically, the verification mindset is focused on finding the bugs that are virtually guaranteed to be in the design by stressing protocols, exploring corner cases, and applying a policy of zero tolerance towards design inconsistencies. Designers on the other hand are concerned with constructing components that correctly implement the intended protocol with appropriate performance and without defects. This paper provides examples of common verification problems that are susceptible to more than one approach, and explores solutions that are consistent with a proper verification mindset, while highlighting the risks associated with a more design-centric approach.**

Keywords—mindset, corner case, debuggability

I. INTRODUCTION

The term mindset can be defined as "an established set of attitudes held by someone", or more precisely, "as a fixed mental disposition that predetermines a person's responses to and interpretations of situations"[1]. The phenomenon is sometimes described as mental inertia, groupthink, or paradigm, and it is often difficult to counteract its effects upon analysis and decision making processes[2].

As an industry, our collective mindset places far too much emphasis on how we do verification and not nearly enough on what we should be doing. For example if you review any job posting for a verification engineer, what you'll find is a list of languages, methodologies, tools and domain knowledge. People are frequently hired because they "know UVM", "have experience with assertions", understand "object-oriented programming" and "constrained-random verification". The hiring process places little to no emphasis on one's judgement or ability to navigate the many difficult decisions that occur over the course of a project – essentially it ignores the single most important determinant of verification effectiveness: the mindset of the engineer. To grasp the weight of this argument, just imagine the absurdity of turning away a great engineering mind like Leonardo da Vinci based on the contents of his toolbox! With some experience most verification engineers can learn to read a specification, build a verification plan, and write decent enough code to implement it. However, where many often fall short of the mark is at the key decision points which are at the foundations of each of these subjects.

Describing a mindset in words is no easy task, and, as such this paper will instead attempt to reveal the verification mindset using a series of critical choices that can arise in verification environments such as:

- Is it the verification environment's duty to accurately replicate the real world?
- Is it acceptable for the testbench and/or testcase to make use of design signals?
- Is it worthwhile to target corner cases that designers consider to be invalid?

Verification engineers often come from the design world or sometimes oscillate between the two roles over the course of their careers. A proper verification mindset is very different from that of a design mindset, and can usually only be arrived at with many years of experience, mentoring, and lessons learned from the school of hard knocks. It permeates every decision, every line of code, every meeting and every interaction throughout the course of a project. It is a way of thinking that is wary of a project veering towards a design-centric verification approach, and is unafraid to boldly say "no, this is not the way to do it!"

At this moment in time, "thinking verification" is both underappreciated and underrepresented in our industry. In this paper we will examine why this topic is so critical to verification effectiveness. We will look at the pitfalls of simply interoperating with a design rather than stressing it; we will describe ways to identify corner cases when reading a design specification; and we will talk about decisions that lead to a testbench with improved debuggability. Overall, through these examples, we will demonstrate what a verification engineer should be doing rather than how they should be doing it, in the hope that this will inspire the verification mindset in the mind of the reader.

II. INTEROPERATING VERSUS STRESSING

One of the most challenging aspects of verification is identifying the thin line that exists between stressing the design and just interoperating with it. While all testbenches must interoperate with the design at some level, only testbenches designed with the proper mindset will know where to draw the line and cry foul at the appropriate moment. Some common situations that illustrate this can be found in designs that

involve clock and data recovery (CDR), handshaking for error conditions, and status indicators for things like FIFO fullness.

A. *Clock Data Recovery*

Some protocols stipulate that data is to be transmitted and received without an accompanying clock signal. In this case, a design implementing such a protocol is required to implement clock-data-recovery (CDR) functionality to extract the clock from the transitions in the data. Developing a verification component (VC) for such a design has a similar requirement – no clock signals are to be transmitted or received from the testbench, just data. Figure 1 shows a design-under-test (DUT) sending and receiving data with such a VC. The DUT takes in a reference clock from a free-running oscillator, which is a slower version of the internal clock with which it drives data. The DUT has a CDR component for receiving data, while the VC has a monitor component for receiving data from the DUT.

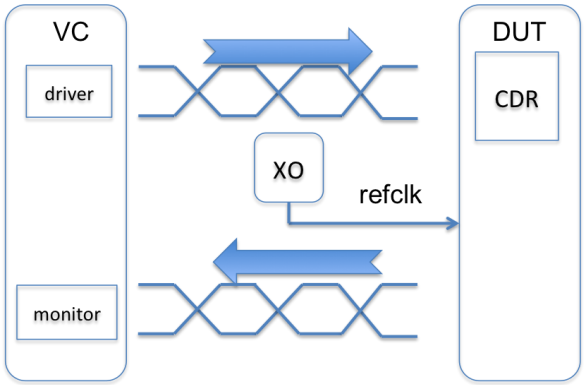

Figure 1 Verification component with design that does CDR

When implementing the VC's monitor, the verification engineer must decide which approach to take in order to receive the data coming from the DUT. Should a CDR algorithm be implemented in the monitor, as is done by the design? Approaching the question from a design-centric mindset would usually lead to the answer "yes, CDR should be done in the monitor", as the VC respects the same protocol as the design it interacts with, and CDR is specified in the protocol. However, when one approaches the question from a verification mindset it leads to just the opposite answer. Let's look at why this is the case.

Doing CDR in the VC's monitor would create a situation where the VC can adapt to the data rate output by the DUT. Paradoxically, the more robust the monitor's CDR algorithm is, the *less* checking the VC will be doing, because it will be increasingly tolerant of invalid data rates coming from the DUT.

An improvement on this would be a CDR algorithm that performs checks on the incoming data rate, but there exists an even better solution: implement a phase-lock-loop (PLL) algorithm based on the same reference clock that the DUT uses as its reference, and use the output of that PLL to sample the incoming data (see Figure 2).

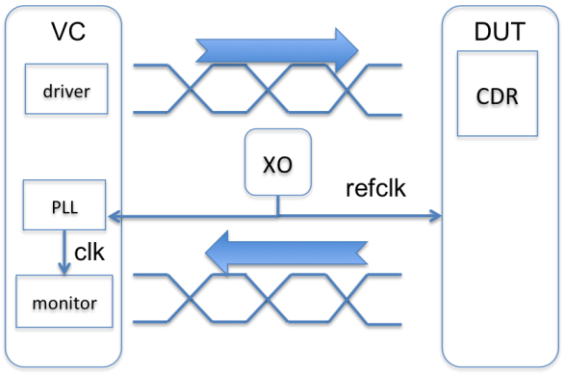

Figure 2 Verification mindset approach for CDR designs

This approach accomplishes the following:

- It verifies that the DUT's data stream is in sync with the reference clock
- It avoids any possibility of the testbench masking a problem because the CDR algorithm is too tolerant
- It has better simulation performance than doing costly checks on data rates
- It is faster and simpler to implement than CDR

At this time, take pause and notice just how different the topic of handling a protocol with CDR is when looked at from a design perspective as compared to a verification perspective. When building an RTL CDR component, the designer strives to build the most robust algorithm possible, capable of interoperating with the widest range of external devices. When building a CDR verification component, the verification engineer instead should strive to build the least robust algorithm possible (or avoid doing CDR at all as suggested above), so it stresses the design and fails on the slightest deviation from the protocol specification. Also note that in the real world, there is no way for the DUT's reference clock to also be reliably sent to another component in the system – otherwise there would be no need for CDR in the first place. This illustrates the point that verification's aim is not to replicate reality, but to verify the design in the most thorough manner possible, even if it means doing so in spite of reality.

Finally, note that doing CDR in a VC with the proper checks in place is not an inherently bad solution; rather, it is an approach that runs counter to the verification mindset we are advocating here.

B. *Handshaking Error Handling*

Oftentimes protocols require that feedback in the form of an acknowledge/not-acknowledge (ACK/NACK) telegram be sent to indicate if an error was detected in the most recently received transfer. Let's look at what happens when creating a VC that implements such a protocol, employing a design-centric approach versus a verification-centric one.

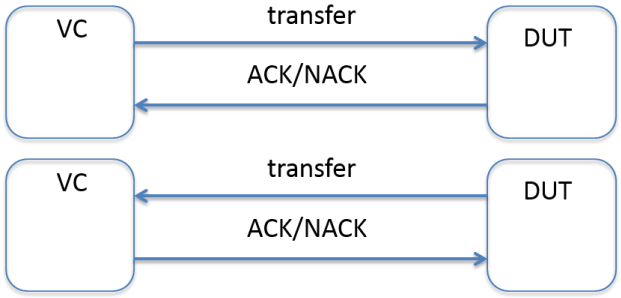

Figure 3 Packet transfers with acknowledgement handshaking

Figure 3 shows transfers and acknowledgements being sent in both directions. As was the case with CDR, a design mindset would lead an engineer to implement the VC according to the protocol specification, that is, to automatically send ACK telegrams when it receives error-free transfers from the DUT and to automatically send NACK telegrams when it receives transfers with errors. This is how a real design would be expected to interact with the DUT after all.

The problem with this approach is that it ignores the fact that under normal simulation conditions (i.e. simulations without any error injection by the testbench), the DUT should *never* have errors in its transfers. This implies that the testbench should *never* have to automatically respond to anything with a NACK. Doing so would potentially mask DUT errors and allow a simulation which should have failed to continue running. The verification mindset recognizes that in the event that the DUT generates a transfer with an unprovoked error, rather than automatically replying with a NACK, the VC's duty is to output an error to the log file and cause the test to fail.

For situations where the DUT is provoked into generating errors (usually restricted to specific directed testcases where the testbench does error injection), the testcase writer must be able to manually control the VC to send a NACK in response. As such, the VC does need to support the ability to send NACKs manually.

To drive the point home, let's take one more example that illustrates the problem of a VC doing automatic handshaking under abnormal circumstances. Let's assume that the protocol states that when design sends a transfer, it must retransmit it again after a given time elapses without a response.

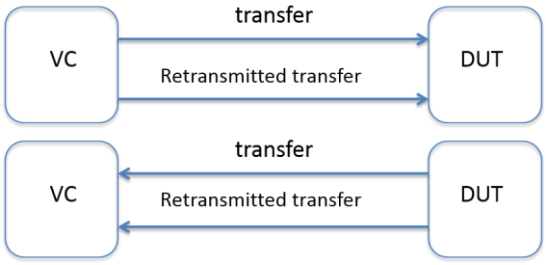

Figure 4 Packet retransmission due to timeout

Upon timing out, a VC built with a design mindset might retransmit the same transfer again, exactly according to the protocol, and in the worst case, without flagging an error for the DUT timeout that occurred. The same VC built with a verification mindset would instead log an error after the transfer timed out, while recognizing that it's a waste of time to implement any automatic retransmission capabilities. Manual retransmission capabilities however are needed to support error injection testcases.

The above examples illustrate the verification mindset's approach to implementing handshaking protocols: a VC is responsible for implementing the protocol, but only up to a point; implementing the full protocol as a design could very well lead to poor verification results and wasted effort.

C. Status Indicators and Clocks

Let's look at an example of a design which indicates buffer fullness with output ports such as "empty" and "full". Any non-error-injecting testcase will want to avoid overflowing and underflowing the design's buffer, and so it will likely want to make use of those flags. However, using the flags without taking the proper precautions is an example of the design mindset at work. It creates the potential for a situation whereby the testbench listens blindly to the DUT, and risks missing out on bugs lurking in the things it is listening to.

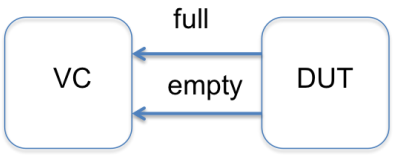

Figure 5 DUT outputting status indicators to testbench

The verification mindset, on the other hand, asks the question "how do we know for sure that the empty/full status indicators don't have any bugs in them?" The answer to this is, "we don't", at least, not until we've done some more work. This could be in the form of a white-box assertion which checks the RTL's read and write pointers versus the status flag values; or it could be done with a monitor in the environment which tracks the number of items written to and read from the buffer; or even with a dedicated testcase that runs in a regression alongside other tests which blindly make use of the flags.

Another classic example of this arises with clock signals – under almost all circumstances, a testbench should never "snoop" the DUT's internal clock signal to use as its own, unless it is doing cycle-by-cycle checks on it (which can hinder simulation performance). Engineers coming from a design mindset fall into this trap because it results in fewer failing tests and faster environment bring-up time, but it also poses a huge risk in terms of masking clocking bugs in the DUT. The proper approach is for the testbench to generate its own clock instead, and for the verification engineer to be prepared to deal with any testcase failures that arise due to clocking bugs (both in the DUT and the testbench). The only circumstance in which clock snooping is a valid approach is when the clock rate is changing according to a random pattern (likely for security purposes) that the testbench cannot predict.

The bottom line is that whenever feasible, the verification environment should only make use of DUT signals if it has first performed checks on them. Anything short of this boils down to a 'blind leading the blind' situation, leaving the door open for bugs to go unnoticed.

III. Outside-The-Box Verification Planning

When someone with a design mindset does verification planning, they most likely will read the specification and come up with scenarios to test each of the specified features. In addition, they will plan on doing error injection for the situations that the DUT can detect (e.g. CRC errors in a communication protocol). While this is a fine approach for finding the more obvious bugs in a design, as we will show in this section, it does not go far enough to find the more obscure bugs which are every bit as critical.

A. Corner Case Identification

1) Function Input Parameters

Imagine a design that draws circles of varying radii, with the radius being an input to the design's "draw" function. A designer might decide to test this by sending a variety of input radii from the smallest possible valid circle to the largest possible one, and, once they all pass, declare that the design is verified. The verification mindset however will do all of that *plus* ask the question "what happens if we try to draw a circle of radius zero?". A designer might very well answer that question with "a circle of radius 0 is invalid, so there's no need to test it". While there's no reason to doubt what the designer has said, the verification mindset doesn't stop there. Whether or not something is valid or invalid is in fact irrelevant; what is important is, can such a scenario *ever* happen, and if the answer it "yes", then it *must* be simulated to ensure that the design recovers from it. A circle of radius zero is as valid a case as simulating a CRC error in a communications protocol: should it happen? No. Should we verify how the device recovers if it does happen? Yes. As it turns out, in a specific circle-drawing design encountered by the authors, unbeknownst to the RTL designer, zero-radius circles were in fact being frequently generated by software under certain conditions. It was the responsibility of the verification engineer to take a higher-level view of things in order to build the best possible verification environment.

2) Register Accesses

Take the following scenario: a design is specified to have a low-power mode that can be activated by writing a '1' to a given register bit. The bit's default value is '0', making the device be in normal mode by default. In the verification planning stage, testing of this low-power mode is put into a directed testcase. When it comes time to implement the test, an engineer creates a test that does the following:

- Write '1' to the low-power bit
- Check that the device enters low-power mode
- Write '0' to the low-power bit
- Check that the device exits low-power mode

The test is added to the regression and passes repeatedly until tapeout. As such, the feature is considered to be verified.

If you haven't realized it yet, the above approach, while appearing sound, has left a critical case uncovered: what happens when a '0' is written to the bit when the bit *is already '0'* (or a '1' when it is already '1')? From a design mindset perspective this might seem like a nonsensical case, but as it turns out, in a design encountered by the authors, simulating this case uncovered a critical bug: writing a '0' to the low-power bit when it was already '0' caused the design to *incorrectly enter low-power mode*.

Outside-the-box verification planning involves looking at all of the things that could conceivably happen, regardless of whether or not the DUT is designed to handle them, and regardless of what designers may say. A verification effort that does not do error injection is quite simply incomplete. As you can imagine, doing error injection in creative ways greatly expands the search space for finding bugs, and so experience and a degree of gut-feeling is required to target those areas most likely to be concealing real bugs. The examples given above illustrate the kinds of places one can look for them.

IV. Prioritizing Debuggability

With the exception of white-box assertions and code comments, RTL code is not usually designed for debuggability. This is to be expected, because RTL code is often used by the same people who develop it (with the notable exception of reusable design IP). Testbenches, on the other hand, are mostly used by engineers who are tasked with writing testcases, who have had no part in the testbench development. For this reason, a well-written testbench puts emphasis on debuggability, so that its users can efficiently write and debug things as issues come up.

Let's look at an example where the verification mindset applies this principle.

A. Protocols with Bi-Directional Ports

Some devices try to save on pins and board trace routing by employing bi-directional ports for data and/or clock signals. The nature of such protocols is that there will always be at least two communicating devices responsible for driving the data and/or clock signals (otherwise there would be no need for bi-directional ports at all). Some devices can be masters (i.e. they initiate transfers), some can be slaves (i.e. they respond to transfers), and some can handle both roles depending on the situation.

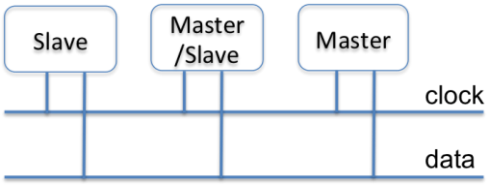

Figure 6 Bi-directional bus with multiple masters and slaves

When building a VC for such a protocol, a design mindset might lead an engineer to follow the letter of the protocol and implement the VC interface with bi-directional ports as follows:

```
interface protocol_if(inout data, inout clk);
endinterface: protocol_if
```

Figure 7 Design-centric approach for bi-directional interface

```
module testbench;
  …
  wire data;
  wire clk;

  protocol_if vc_if(data, clk);
  …
  design_t dut_inst (.data(data), .clk(clk))
  …
endmodule: testbench
```

Figure 8 Design-centric approach for bi-directional testbench

While this approach allows the VC to fulfill the protocol requirements, it offers users of the VC a low degree of debuggability. Let's look at an example to examine why this is the case: in any verification environment of a design with bi-directional ports, at a minimum there will be one such VC connected to one DUT. However in verification environments for some protocols like I2C there is the potential to have multiple VCs connected to multiple DUTs, each acting as masters, slaves, or both (see Figure 9).

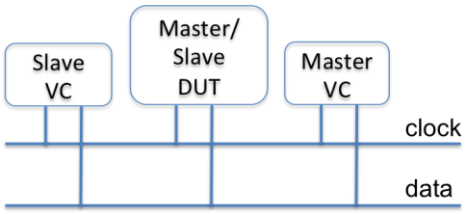

Figure 9 Bi-directional bus with multiple VCs and DUT

Because all of the DUTs and VCs connect to the same bidirectional ports, debugging a waveform quickly becomes a painstaking experience because it is impossible to tell which component is driving what value.

Using a verification mindset, we make use of both unidirectional and bi-directional signals to achieve both ease of debug and adherence to the protocol. This is shown in the following example, in which a pull-up is used on both the data and clock lines:

```
interface protocol_if(inout data, inout clk);

  logic clk_o = 1;
  logic clk_i;
  logic data_o = 1;
  logic data_i;

  assign (highz1,strong0) data = data_o;
  assign (highz1,strong0) clk  = clk_o;

  assign data_i = data;
  assign clk_i  = clk;

endinterface: protocol_if
```

Figure 10 Improved approach for bi-directional interface

(In Figure 10, "(highz1, strong0)" means "when signal is assigned with a '1', it takes on 'Z'; when it is assigned to with a '0', it takes on '0' ").

The testbench then needs to connect to the interface as follows:

```
module testbench;
  …
  wire data;
  wire clk;

  pullup(data);
  pullup(clk);

  protocol_if vc_if(data, clk);
  …
  design_t dut_inst (.data(data), .clk(clk))
  …
endmodule: testbench
```

Figure 11 Improved approach for bi-directional testbench hookup

From this point on, components inside the VC only interact with the unidirectional versions of the signals. So for example driver code will drive the "_o" versions of *clk* and *data*, and will read the "_i" versions of them:

```
class protocol_driver extends uvm_driver;
  …
  task drive_bit(input bit data);
    …
    m_vif.driver.clk_o <= 0;
    # 10ns;
    m_vif.driver.clk_o <= 1;
    if (m_vif.driver.clk_i == 0)
      @(posedge m_vif.driver.clk_i);
    …
  endtask: drive_bit
endclass: protocol_driver
```

Figure 12 VC driver interacting with bi-directional interface

With this approach, a waveform can show what any VC agent is driving and observing at any time, leading to greatly improved debuggability for users of the VC. This illustrates

how the verification mindset puts a high priority on debuggability, by finding solutions that allow users to resolve issues more quickly.

V. Rounding Out The Mindset

There are several remaining topics that need to be touched upon in order to round out the scope of this paper.

A. No Coverage Without Checking

The example presented earlier regarding writing a '0' to a low-power bit when it's already '0' highlights another aspect of the verification mindset, which is to *never do coverage on anything in the absence of doing checks.* This rules out doing register value coverage, because it is misleading at best and a waste of compute resources in a large system-on-a-chip (SOC). Coverage would say "yes we have put the values '0' and '1' in the low-power bit", giving the false sense of security that there are no bugs hiding there.

B. Approach to Debugging

Designers usually use waveforms to debug issues in a simulation, such as failing monitor checkers and assertions. This makes perfect sense, as almost all information regarding the DUT's state can be observed in waves.

This is not true for VCs however, because of the many operations they do are in zero time, and the dynamic data structures they use for things such as sending stimulus. While simulators are always getting better at making this type of information available to both waveforms and other debug windows, the verification mindset recognizes that one of the best tools for doing debug is the logfile itself.

For the logfile to be of use, the VC needs to have an appropriate and consistent messaging scheme (the contents of which are beyond the scope of this paper). Sufficeth it to say that verification engineers in general should *not* be debugging their testbench using waveforms, but rather they should be relying on logfiles. If they are unable to debug things using the logfile, then this is an indication that the testbench's messaging scheme needs improvement.

C. Zoom-In, Zoom-Out Thinking

Something that verification engineers sometimes struggle with when doing system-level verification is the fact that they must be able to switch from low-level thinking (what we will call "zoom-in") to higher-level thinking (what we will call "zoom-out"). When zoomed-in the engineer does tasks such as:

- Understand design specifications
- Write verification plans based on the specification
- Write code to implement the verification plan
- Write testcase code
- Debug failing testcases

Verification engineers must also do a series of tasks while zoomed-out such as:

- Decide which design features to focus on to maximize bug discovery
- Devise creative ways to tease bugs out
- Allocate time so as to get the most important checking and coverage for the effort

The main reason for this is difference is that verifiers need to deal with a larger scope than designers do. In addition, unlike designers, who must complete their design before tapeout, verifiers can continue verifying past tapeout and therefore must prioritize what is to be accomplished before tapeout, what can be put off until afterwards, and what can be dropped altogether. To do this effectively, a verifier needs to be constantly aware of a wide array of information such as system architecture, design hot-spots, project schedule, and client deliverables. The fact that each of these items can change from one week to the next means that zoom-out thinking needs to be done on a regular basis, to ensure that the right things are being focused on. For this reason, it is not enough for a verification engineer to just be proficient in the skills that comprise his or her toolbox; there is a need to stand back, listen, and choose the best path forward in order to truly be effective.

D. What Are We Trying to Accomplish Here?

In the business world, an important question that organizations must continuously ask themselves is "what business are we in?". The notion is that by answering it, a business can identify where it needs to focus its attention, who its competitors really are, and, what types of things it can outsource to third parties.

In the verification world, a similar question is "what are we trying to accomplish here?" Will the task at hand lead to a device with fewer bugs? Is it already being accomplished by other elements in the environment?

A good example of verification engineers *not* asking this question is when they do coverage on DUT outputs. While this looks fine on the surface, when one asks "what are we trying to accomplish here?" the answer that comes back is usually "not very much!". With coverage on the testbench's stimulus and checks on the DUT's outputs, there is usually no need to do coverage on DUT outputs. The result is just wasted effort that should have been spent elsewhere.

Note that doing coverage on internal DUT signals is another matter, and can often be useful towards achieving quality tapeouts.

E. Coverage, Not Testcases

The design mindset sometimes places too much emphasis on testcase writing and evaluating passing rates. This line of thought is based on the belief that writing more testcases translates to more things being tested, and more passing testcases translates to more coverage of the DUT's functionality. It is the way things were done before constrained-random pre-silicon verification came into fashion, and continues to be used in many emulation and validation setups. The benefits of it are that it is easy to explain and understand, and simple to track progress with.

The verification mindset, on the other hand, recognizes that the "holy grail" is achieving coverage closure, provided that

the coverage is accompanied by proper checks (as previously mentioned). It views testcases as nothing more than a vehicle by which coverage closure can be reached. As such, the number of testcases is viewed as being an incidental artifact of the planning process, and the passing rate is viewed as being important only inasmuch as it contributes to coverage (failing tests should not be counted towards coverage).

F. Liaison between Design Architect and Design Engineer

A key difference between designers and verifiers is the level of abstraction at which they operate. Digital designers usually work at a lower level of abstraction, implementing a single module that performs a specific aspect of a larger design or protocol. Verification engineers, on the other hand, work at a higher level, creating a VC that implements all aspects required to interface with and check a given design. While this is well understood, the unique role that this creates for a verification engineer is less so.

The standard situation is one where a design architect writes out a specification, and a design engineer implements a block according to their understanding of it. During the implementation phase, the verifier is in fact the liaison between the two groups that ensures communication and coherency between them. This could involve identifying aspects of the architecture that will lead to increased verification effort such as a lack of consistency, or the introduction of additional modes that are "nice-to-haves" rather than essential features. In such cases, while it's everyone's responsibility to take care to avoid such things, it is the verification effort that will usually be most impacted, and therefore the verification engineer's principle responsibility to speak up and say "here's why we're not going to do it that way".

G. Quitting on First Error

When a simulation encounters an error, the verification engineer must decide to either let the simulation continue running, or exit right there and then. While there are some situations where it can be advantageous to continue running, for the vast majority of cases, quitting immediately is the right thing to do. Let's look at why this is so.

When the testbench discovers an illegal condition in the DUT, there's a decent chance that from that simulation time onward, the testbench is no longer doing proper checking anymore. For example, imagine a protocol with a start-of-frame (SOF) and end-of-frame delimiter.

If the DUT sends a corrupted EOF, the testbench monitor will likely interpret it to be just another data byte, and could eventually raise an error for some type of data length violation. If another SOF is sent before such a length violation is declared, the monitor might interpret it as an illegal symbol in the data stream. At this point, depending on the protocol, the VC could be in a situation where it cannot recover and do its job of checking properly. Note that this is perfectly acceptable. The testbench is *not* a DUT, and does not need to be able to recover from error scenarios. It can be thought of a kind of mousetrap which, upon catching a bug in a given simulation, is closed for catching subsequent ones. The verification mindset recognizes that putting in any effort to make the testbench recover after detecting an error is wasted effort that should be directed elsewhere.

The situation in which it is useful to run beyond an error is when the time comes to debug that error, and simulating longer will provide information to help diagnose the issue. This can be setup using a simulator command-line directive, which would override the default behaviour of quitting on the first error.

VI. Conclusion

In this paper we have demonstrated that the verification mindset is very different from that of a design mindset, and, when properly applied, will lead to improved results with respect to bug-free device tape-outs. The mindset itself is anchored upon finding bugs by any means necessary, even if it means not replicating reality or faithfully implementing a given protocol specification. It recognizes that building debuggability into a testbench, through architectural and messaging decisions, will lead to improved ease of use, and that identifying difficult corner cases requires one to think outside of the design specification. Finally, it involves some softer skills to avoid the pitfall of not seeing the forest for the trees.

Acknowledgments

Special thanks to Mike Warner (Cirrus Logic) for sharing his thoughts and performing valuable reviews.

References

[1] http://www.thefreedictionary.com/mindset

[2] http://en.wikipedia.org/wiki/Mindset