

# Introduction to SVA Assertions for Design Engineers

*by Eric Deal of Cyclic Design, LLC*

Assertions and Assertion-Based Verification (ABV) are a hot topic, but many engineering teams remain unfamiliar with the benefits that assertions bring to the design and verification process. This paper discusses the rationale for using assertions, the benefits of using assertions throughout the design and verification process, and a step-by-step approach to implementing assertions within a design.

This whitepaper is written from the viewpoint of a design engineer. Hopefully both design and verification engineers will see the relevance of (and humor in) these comments from the designer-centric world view.

## Assertions in the Design Process

Implementing assertions within a design requires a conscious decision on the part of the designer to view the design process differently. Typically, designers approach a design in four stages:

1. Gain an understanding of the requirements of the design/algorithm,
2. Design RTL to implement a solution,
3. Perform some simple testing to verify basic functionality.
4. Debug test failures found by verification engineer

There may be variations to the process, such as creating a micro-architectural specification, but these additional steps generally fall into one of the above categories. Adding assertions, however, requires an additional step, which probably explains why designers haven't readily adopted assertion use.

1. Gain an understanding of the requirements of the design/algorithm,
2. Design RTL to implement a solution,
3. **Add SystemVerilog Assertions (SVA)**
4. Perform some simple testing to verify basic functionality.
5. Debug test failures found by verification engineer

As designers begin to understand the power of assertions, they will readily adopt them if for no other reason than pure selfishness: spending time creating assertions minimizes time spent debugging later.

## Benefits of Assertions

Assertions incorporated into a design benefit both the design and verification engineer. Much of a designer's time is actually spent debugging problems found in the design during the verification process. Typical issues that arise during verification (especially for constrained-random testing) include:

- Logic bugs in the design
- Inaccurate behavioral models based on poorly documented interfaces
- Invalid programming or use models

While we (as designers) appreciate the work being done by the verification engineers to test our designs, we also secretly dread the sight of them in our doorway, since that generally means something failed that we now need to debug. In this case, we have to either interrupt our current work or put off debugging the testcase until we get to a good stopping point. Both scenarios result in an interruption of the verification engineer's work, but I suspect many verification engineers dread the latter scenario, since it often means they must interrupt work on testbench development progress to ensure the failing case can be reliably recreated.

In many cases, assertions allow us to break this cycle. They essentially act as red flags during simulation pinpointing failures that may either directly cause test failures or not be detected by passing tests.

Assertions on module interfaces can quickly identify invalid behavior that may be caused by a behavioral model or improper use of the design (invalid register settings, invalid operating modes, etc). Such assertion failures indicate that a problem may be with the testbench, allowing the verification engineer not only to pinpoint the cause of the failure, but also to quickly update the testbench and continue with verification, often without even having to consult the design engineer. In cases where the design is at fault, the design and verification engineers can quickly resolve the issue without extensive debug, because they know where and when the failure occurred.

Internal design assertions also help locate the root cause of failures that directly result in testcase failures. For instance, constrained-random testbenches often find corner cases such as FIFO overflow/underflow conditions that are normally not exercised by directed tests. Debugging failures such as these takes time since the designer has to trace back the failure at the end of a test (which may simply be invalid data in system memory) to the root cause of the failure. ***Including even a few simple assertions on internal FIFOs or interfaces will often locate these failures at the time and point of failure, eliminating the need for a lengthy debug session.***

Lastly, assertions improve reuse since they capture design intent that may be lost as design and verification engineers familiar with the design move to new projects. Some of this information may be captured in micro-architecture documents, but who really reads the manuals unless there is a problem? With assertions present, it is far more likely that problems integrating reusable logic will be found.

Ultimately, designers like to create things. As design engineers begin to understand how assertions reduce debug time and effort, they will realize that time spent creating assertions pales in comparison. Some designers may even come to view adding assertions as a creative challenge.

## ***A Different Mindset***

Adding assertions requires the designer to think of the design from a different point of view. Instead of viewing signals as the implementation of a solution, designers must step back and think about what rules the design, bus/signal protocols, or usage model impose on these signals. These rules must then be coded in an assertion language such as the SystemVerilog Assertion language (SVA), in order to capture the behavior of individual signals or the interaction of several related signals.

It is important to note that assertions should not simply recreate rules forced on signals by a particular implementation. For instance, coding an assertion to recreate the transitions in a state machine is probably not terribly useful. However, coding assertions relating events occurring at a module's interface to the behavior of a state machine can be very powerful.

In order to focus on the process of developing assertions, this paper will focus on creating a simple subset of assertions for ARM's AXI bus interface. In doing this, we simplify the assertion writing

process since assertions will be written for a well-known protocol, and we can use the AXI specification as our list of requirements.

## Creating Assertions

### ***Step 1: Identify Target Modules and Signals***

The first step when adding assertions is to identify the module or group of signals where assertions should be applied. Generally it is easier to describe the behavior of control signals than data signals since algorithmic operations do not lend themselves to simple behavioral rules. Control signals, however, often are responsible for responding to external and internal stimulus, and thus, their behavior and interaction is more readily described. For our AXI example, we will identify the entire set of AXI signals as our target, but for more general applications, designers may choose to focus more time adding assertions at interface boundaries or modules with high levels of control logic.

Adding assertions at the boundary of a module provides an additional benefit. In addition to checking that the internal logic is compliant, they check that the external environment behaves according to these rules. These assertions can therefore check for invalid testbench configurations or protocol violations as well as problems in other blocks that would later be interfaced to this module.

### ***Step 2: Create Rules by Paraphrasing Intent***

Many engineers are overwhelmed by the prospect of adding assertions, mainly because it takes time to think of a rule and then code the associated assertion. Constantly switching between the right-brain activity (exploring associations) and left-brain activity (synthesizing the associated SVA construct) makes the process seem very disjointed.

A better approach is to split the process into two separate steps. First brainstorm as many rules as possible, and simply write each as a phrase much like an RTL comment. Later, take all these phrases and code them as SVAs. By doing this, you will find that you can create a richer set of assertions while streamlining the procedural aspects of coding the assertions.

For our AXI example, we might brainstorm the following rules for the write address bus (these are just a few):

- AWBURST may never be 2'b11 (reserved)
- AWCACHE must never be 4'b010x, 4'b100x, or 4'b110x
- When AWVALID goes high, it must remain high until AWREADY is asserted
- When AWVALID goes high, AWADDR/ID/LEN/SIZE/etc must remain stable until AWREADY
- When an address is accepted (AWVALID & AWREADY) we must see a corresponding response with the same ID (BVALID & BREADY) at some point later in time

Each of the channels shares some of these rules, so many can be reused when creating assertions for the other AXI signals. Many of these rules come directly from the AXI specification – all an engineer needs to do is read the specification and look for phrases that can be interpreted as rules.

### ***Step 3: Code SVA Assertions***

Now that we have a list of rules, we will switch over to our left-brain to code the SVA statements. This is made much easier since we already have a description of the rule and simply need to find the

best way to express the rule in the SVA language. At this point, engineers have a choice to make: they can either code in the SVA by hand or use a third-party tool such as Zazz to automate the task.

Coding directly in SVA, the assertions from step 2 are as follows (just the SVA sequences are shown):

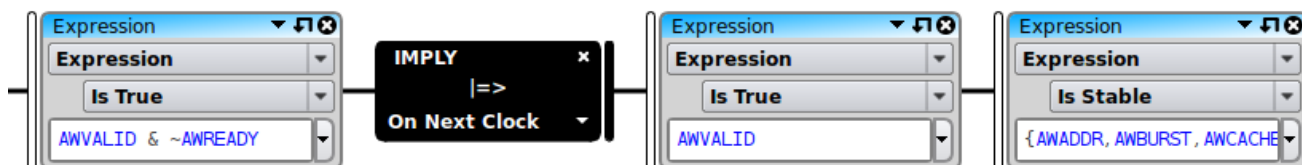
- AWVALID & AWREADY |-> (AWBURST!=2'b11)
- AWVALID & AWREADY |-> (AWCACHE[3:1]!=3'b010) & (AWCACHE[3:1]!=3'b100) & (AWCACHE[3:1]!=3'b110)
- AWVALID & ~AWREADY |=> AWVALID;
- AWVALID & ~AWREADY |=> \$stable({AWADDR,AWID,AWLEN,AWSIZE,AWCACHE,AWSIZE})

The final assertion (not shown above) is more complicated and would probably take someone experienced in SVA to code it. However, all of these assertions are trivial to code using Zazz:

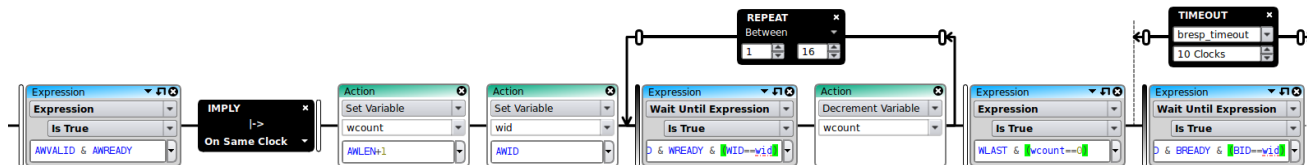
### AWBURST and AWCACHE checking (combined)



### AWVALID high until AWREADY and stable controls (combined)



### Full check of address write through data transfer and write response\*



The full SVA sequence (created by Zazz) is pretty intimidating:

```

property AW_W_B_completion;
int wcount;
int wid;
@(posedge ACLK)
(AWVALID & AWREADY) |-> (1'b1, wcount=AWLEN+1) ##0 (1'b1, wid=AWID) ##0
(
(
(WVALID & WREADY & (WID==wid)) [->1] ) ##0 (1'b1, wcount=wcount-1) ) [*1:16]
)
##0 (WLAST & (wcount==0)) ##0
(
(#[0:bresp_timeout] (1'b1)) intersect
( (BVALID & BREADY & (BID==wid)) [->1] )
);
endproperty

```

## Step 4: Debug Assertions

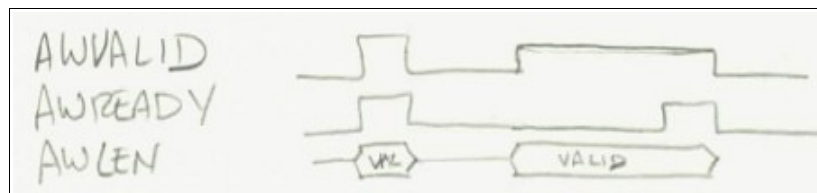
Assertions are just like any other logic construct, so it is imperative that the assertions are coded to accurately describe design behavior; otherwise, both design and verification time will be spent debugging false assertion failures, negating the benefit of using assertions in the first place.

Traditionally, assertions are debugged by running dynamic simulations or using formal tools to exercise the logic and assertions. Failures in the design and assertions are then iteratively corrected as verification proceeds. Often simple assertions require little debug since the rules they convey are fairly absolute. For instance, the assertion “AWVALID must be de-asserted during reset” ( $\sim\text{ARESETn} \rightarrow \sim\text{AWVALID}$ ) is pretty difficult to code incorrectly.

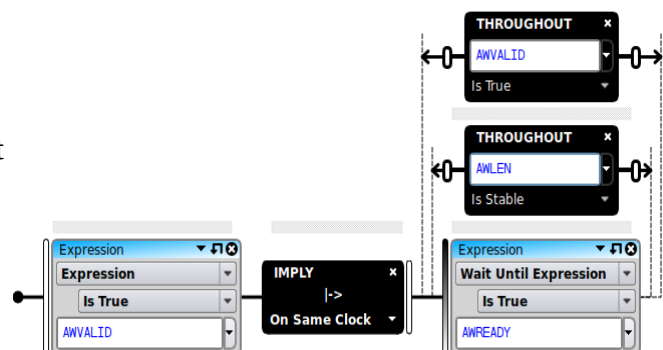
More complex assertions present a different issue. Often when coding an assertion that describes the interaction between two or more signals, the designer envisions a waveform that describes this behavior and then codes an SVA to describe these relationships. The problem is that the SVA is a temporal modeling language, which is inherently different from modeling digital logic\*. This often leads to problems where the designer's intent in the coded SVA does not match the actual behavior of the assertion.

The problem is that a designer needs to close the loop between assertion intent (the mental image of a waveform) and the actual behavior of the assertion (the coded SVA). Zazz addresses this issue by generating a small randomized testbench around each individual assertion, allowing the designer to immediately observe behaviors associated with the assertion, thus closing the loop between intent and behavior. As an added benefit, this process often leads designers to see additional relationships which should be described to more fully constrain behavior, leading to even higher quality assertions.

As an example, let's code the assertion “AWVALID must remain asserted until AWREADY and AWLEN must remain stable throughout this time.” The designer might envision intent as shown in the sketch below:

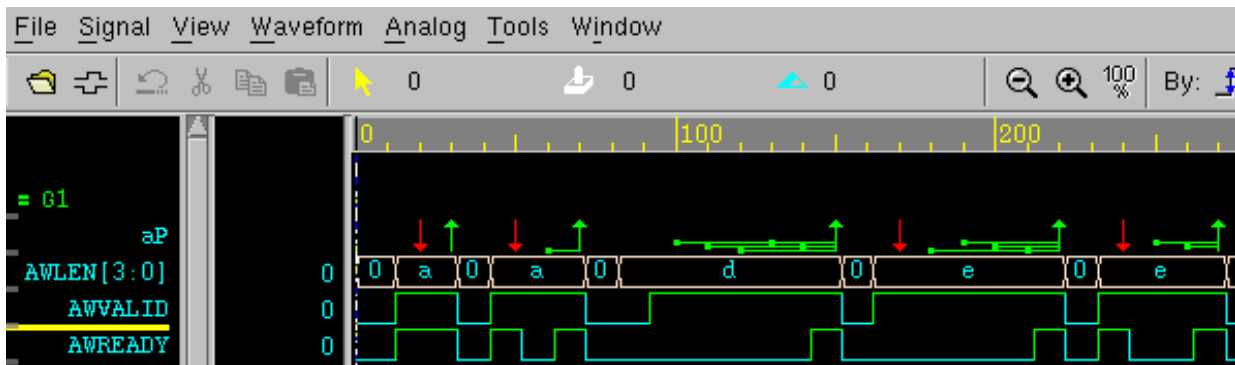


Using Zazz to construct the assertion, it might look like the diagram below\*. It says that when AWVALID is asserted, the assertion should wait until AWREADY is asserted, and then apply two constraints throughout this time: AWVALID must remain true and AWLEN must remain stable (the other control signals are omitted for simplicity).



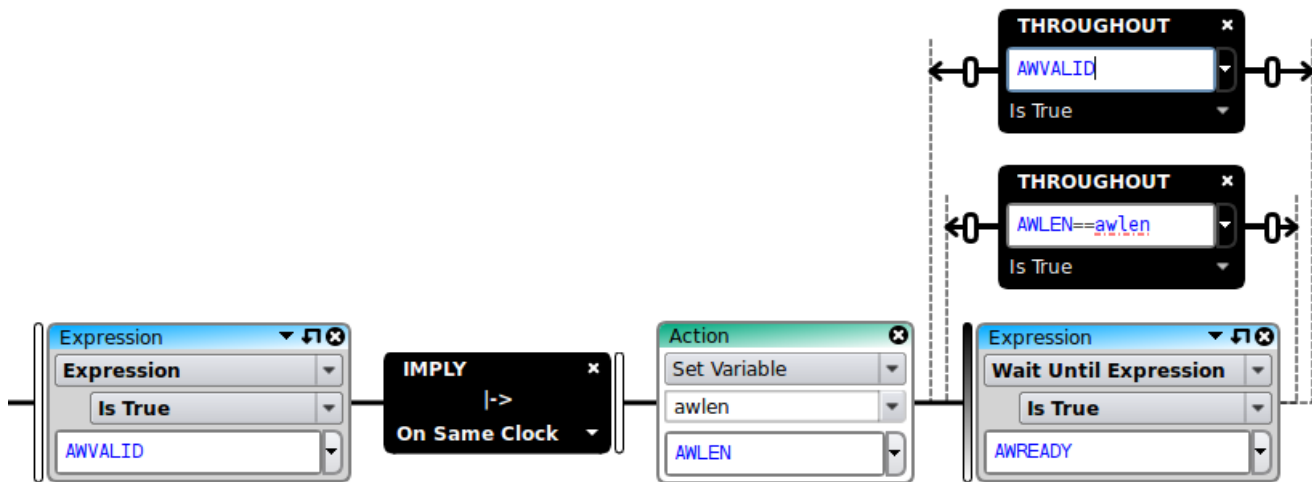
Using the SVA Debug feature, Zazz generates some stimulus and allows the engineer to see how the assertion behaves, as shown in the image below:

\* To keep things simple, the assertion does not account for back-to-back AWVALID cases.

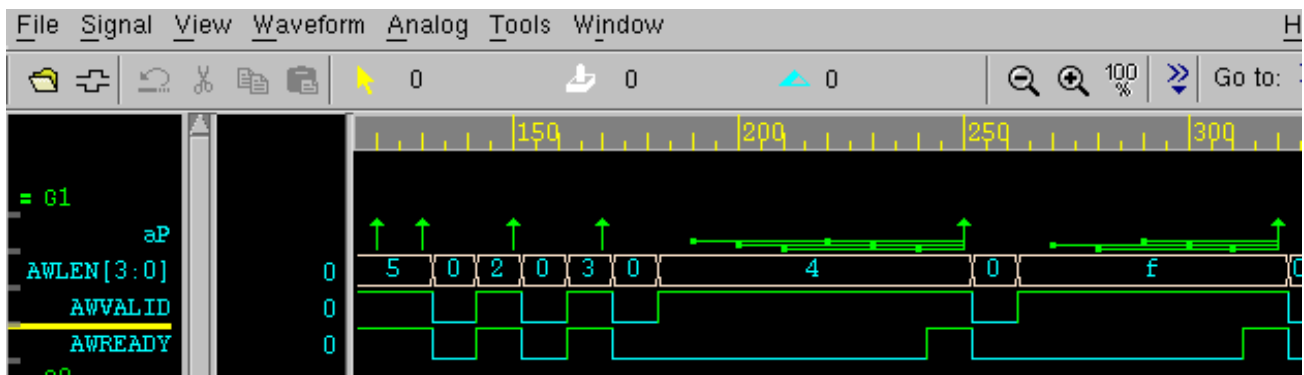


We see that the assertion is failing on several scenarios. The problem is that the \$stable on AWLEN should not apply on the first cycle but instead, only on subsequent cycles. It also shows that the assertion triggers multiple attempts on the same event (which may not be the desired behavior for some assertions).

The corrected assertion and its associated waveform are shown below:



This version sets a temporary variable (awlen) to hold AWLEN and then expects the AWLEN field to match this value until AWREADY is asserted (either immediately or on some subsequent cycle). In addition, the assertion checks that AWVALID remains true during this time period. The resulting waveform no longer displays the assertion failures and the sample waveforms match the original intent. (NOTE: this check should be expanded to check {AWLEN, AWSIZE, ...} instead of just AWLEN).



## Conclusion

While assertions can be a powerful addition to a design team's arsenal, they force designers to spend time early in the design process adding quality assertions. Design engineers may resist adding assertions because of tight time constraints and the complexity of the SVA language. In spite of the time savings for designers during the verification process, adding assertions to the design process has remained a hard sell.

Tools such as Zazz improve ease-of-use to the point that designers can now easily incorporate assertions into designs. The inclusion of high-quality assertions throughout the design will minimize not only the time spent debugging designs, but also the time spent communicating issues between the design and verification engineers. These same assertions may also detect “hidden” failures that do not propagate to an observable point that results in testcase failure, increasing the chances of finding such problems before tapeout.

The current mindset that assertions belong only in the verification process needs to change. As designers begin to understand that assertions are easy to incorporate early in the design process, design teams can realize the benefits of assertions throughout the entire design and verification process.

**An online version of this paper and video tutorial is available at <http://zocalo-tech.com/tutorial>.**

### ABOUT THE AUTHOR

**Eric Deal, President, Cyclic Design LLC** [www.cyclicdesign.com](http://www.cyclicdesign.com)

Eric has 18 years digital logic design and architecture experience at IBM, Rockwell/Conexant, Sigmatel, Multixtor, and most recently with his own IP/consulting company, Cyclic Design, which specializes in error correction for NAND flash. Eric has helped develop several SOC platforms and has experience in clocking, bus connectivity, and development in ECC, encryption, security, image processing, and I/O interfaces. He promoted the use of OVL assertions while at Conexant and was an early adopter of SystemVerilog assertions at Sigmatel. Eric has been a consultant for Zocalo since 2009, and was instrumental in developing the Bird Dog algorithm and the Visual SVA concept.

### ABOUT ZOCALO TECH

Zocalo Tech, Inc., incorporated in 2006, is focused on productivity software for quick and easy creation, use and reuse of assertions that work with popular functional verification flows. Zocalo software, marketed under the name Zazz, has been architected and developed from the ground up with one goal in mind: increased productivity for engineers adopting and utilizing Assertion-Based Verification. For more information, please visit [www.zocalo-tech.com](http://www.zocalo-tech.com).

[www.zocalo-tech.com](http://www.zocalo-tech.com)

[info@zocalo-tech.com](mailto:info@zocalo-tech.com)

512-623-7711