COMPONENTS OF A COMPLETE ASSERTION-BASED VERIFICATION SOLUTION
INTRODUCTION

Assertion-based verification (ABV) is rapidly being adopted by engineering communities to help them improve their design quality and reduce their time to market. There are many different aspects of this technology. In fact, ABV is actually more a methodology than a single tool. Understanding all the components of ABV and how they can be used in a design flow is critical to achieving the benefits that such a capability can provide. This paper will examine the various aspects of ABV and how they can be leveraged to improve design and verification engineering goals. We will start with a brief overview of ABV, followed by a discussion of what capabilities are needed for a basic ABV solution. We will then discuss other components that can be added to make a complete solution.

ABV OVERVIEW

ABV has evolved in response to the problem of not being able to adequately verify a design’s functional operation. As designs have gotten bigger and design cycles have gotten shorter, traditional simulation has failed to deliver adequate functional verification. Simple assertions have been used for years in designs; in both Verilog® and VHDL, designers have used RTL constructs to monitor and report on signal activity, and thus behavior, in their designs. This has evolved over the years to where we now have specific languages, tools, and methodologies that support ABV. Briefly stated, ABV allows design intent to be captured and gives “white box” visibility to all engineers who may work on the design. White box visibility refers to the ability to see various portions of the design, as opposed to “black box,” which allows only the primary inputs and outputs to be observed or controlled. Increased visibility helps engineers understand what has been tested and what has failed.

Once implemented, the user will see a variety of benefits from ABV. The three most important are:

1. Detects incorrect operation (errors) directly, and close to the source. This localization simplifies debugging and repair. (Other verification approaches either completely miss the bug or notice it at a register or primary output, which makes debugging far more difficult.)
2. Detects correct operation and logs these occurrences for coverage analysis. This enables users to optimize tests to improve efficiency and notifies them when they are done with testing.
3. Documents functionality with an “executable specification.” This helps facilitate reuse by ensuring correct usage of IP in future projects.

REQUIREMENTS FOR A BASIC ABV SOLUTION

A complete ABV solution requires more than just tools and language—it requires adherence to a methodology as well. In later sections of this document, we will explore this in more detail. At this point, however, we will introduce the two major capabilities that are required to implement a basic ABV solution:

1. A means of specifying behavior
2. The ability to display and debug results

Assertions need to be added to the design in order for the tools to operate on it. Assertions can be specified in one of two ways:

1. Use a library of pre-defined assertions and instantiate them into the design
2. Use an assertion language to create user-written assertions
In many cases, design teams will do both. Since designers usually want to focus on the RTL, pre-defined and standard assertions (FIFO, flip-flop, etc.) make it easier for them to annotate their design. Verification engineers will use pre-defined verification IP (VIP) where possible, for things such as standard protocol monitors. They will also add user-written assertions as required.

To create user-written assertions, or to develop a library of reusable assertions, a language is required. In the past, HDLs have been used for this. A good example of using an HDL to develop basic assertions is the OVL (Open Verification Library). This is a set of assertions available to users from Accellera (www.accellera.org or www.verificationlib.org). Since these assertions have been developed in both Verilog and VHDL, they are easy to use in most design environments. But this library includes only very basic assertions, and thus may have limited value for specifying complex behaviors.

Developing more complex assertions is necessary for most designs, since users want to check for complex behavior (e.g., a protocol check). Determining that a read operation executed correctly can be difficult to develop in an HDL, since it defines a temporal relationship. Assertion languages were developed to address this limitation.

This paper will not explain the evolution of these languages—it suffices to say that there were several proprietary languages in use before the industry settled on two standard languages: PSL (Property Specification Language) and SVA (SystemVerilog Assertions). PSL is a separate language that can be used with any HDL, including Verilog, SystemVerilog, VHDL, and SystemC®. SystemVerilog Assertions are part of the SystemVerilog language standard.

Once the assertions have been written or instantiated into the design, they can then be used for finding errors and tracking behavior. There are two ways that the assertions can be evaluated: dynamically and statically. Basic assertion analysis is performed along with simulation, since simulation is something that nearly all designers and verification engineers already do on their designs. Static assertion analysis, also known as formal analysis, does not require simulation. We will discuss this more in a later section.

Dynamic evaluation involves the evaluation of assertions side by side with design elements by the simulator. As conditions are satisfied, the assertions are triggered and cause some action to be taken; for example, issuing a message or logging a coverage event.

This capability easily adapts to an existing verification design flow. Assertions can be added to the design and/or testbench, and as simulations are run, the assertions will help find additional bugs and report coverage data. No other changes to the design or testbench are required; neither are additional tools.

**ADVANCED ABV COMPONENTS**

Now that we have reviewed the basic ABV capability, let’s look at more advanced capabilities that can significantly enhance user productivity. There are five main areas we will investigate:

1. Verification IP (VIP)—These are pre-defined modules that allow users to add assertions to the design much more quickly than writing them from scratch.
2. Integrated debug environment—ABV produces a lot of additional information, so advanced debug capabilities are required to understand and respond to this data.
3. Assertion-based coverage—ABV can also produce a lot of coverage data, so an environment that allows the user to analyze it and integrate it with other types of coverage data can significantly improve results.
4. Static ABV—Leveraging formal analysis allows users to find bugs earlier in the design process prior to building testbenches.
5. Assertion-based acceleration (ABA)—Hardware acceleration is often used for integration tests and long latency tests, both of which may find extreme corner case problems with the design. Support for assertions in acceleration helps detect and debug such problems more effectively.
**VERIFICATION IP (VIP)**

Creating user-written assertions for the entire design and testbench can be a significant task. For just a single bus interface there can be dozens or even hundreds of assertions required. Multiply that by the number of interfaces and add in the assertions you need on the design elements, and you can be looking at months or even years of development effort. There are two main types of assertion-based VIP that can help:

1. Function monitors
2. Protocol monitors

Function monitors are modules that can be used on the logic structures of the design. Some of these are available in the OVL; more advanced libraries may include checks for constructs like FIFOs, state machines, and arbiters. They consist of one to many assertions, based on the complexity of the RTL being checked, and they are bundled together in a module such that they can be easily instantiated into the design.

Protocol monitors are essentially a set of assertions that cover the various behaviors of a standard protocol, such as AMBA or PCI-Express. These types of protocols have numerous different operating modes, and each mode has dozens of different things that need to be checked: simple things like the “read” and “write” signals never being on at the same time, and complex things such as the proper sequence of signals to initiate and complete a burst operation.

Building and testing such a complex monitor can be a daunting and time-consuming task. But it needs to be done only once and can then be reused across many designs. Protocol monitors are developed by a variety of sources. Design IP companies are building them to help their customers use IP blocks more effectively and easily. Verification IP companies build them to sell to customers that use a variety of different interfaces and protocols in their designs. EDA companies build them to offer along with their tool sets. And finally, end users build them for proprietary or custom interfaces and protocols that they plan to use across multiple projects.

The most useful forms of VIP will contain not just assertions for error checking, but coverage monitors as well. This allows the monitor to automatically log various types of coverage data, without the user having to add it.

Both types of VIP can be used in a single design. For instance, a design engineer may use a variety of function monitors to check the inner workings of a block or module, while a verification engineer may instantiate protocol monitors on the major interfaces of the design to make sure they are properly exercised during system-level simulation.

**INTEGRATED DEBUG ENVIRONMENT**

When implementing an ABV methodology, it is helpful to keep the amount of changes for the user as minimal as possible. Since most ABV flows begin with adding assertions to the design for use with simulation, keeping the same simulation debug and analysis tools is important to minimize the learning curve. What is really needed is complete and seamless integration of assertion debugging into the existing simulation debugging environment. Engineers are accustomed to using source browsers, waveform displays, and hierarchy browsers to help debug their simulations, and these same capabilities need to be applied to assertions as well.

Assertion debugging requires a variety of information to accurately debug. Some of the data that needs to be seen includes: what signals are involved in the assertion; the current state of the assertion; and when the assertion begins, finishes, or fails. The user also needs control over the assertion language constructs; for instance, being able to turn them on or off, starting and stopping recording of data, etc.
One additional capability that can be helpful is a way to get a brief overview of all the assertions in the design. This would help engineers understand the status of every assertion, the number of times it has been evaluated so far, and how many of those times it has failed. It is important that this capability has the same GUI as the other tools and supports inter-tool communication. This allows users to drag and drop between various windows.

**ASSERTION-BASED COVERAGE**

A big part of assertion-based verification is the collection of coverage data. As assertions are evaluated in the design, they not only flag problems, but they can also track good behavior. For instance, if a particular state of a machine is reached, an assertion can log that fact. Analyzing this type of data helps the user understand how well the design has been tested.

Assertion languages support two main types of statements: assert directives for error checking, and coverage directives for gathering coverage information. The coverage constructs allow a particular behavior to be logged as it occurs. Since there can be hundreds or more assertions in a design, with each being evaluated many times (based on the part of the design it is in and the number of cycles the simulation runs), there can be a lot of coverage data produced. As such, there needs to be an easy way to aggregate and analyze this data. A graphical user interface is most effective here.

Additionally, most simulation users collect other types of coverage data. For instance, many users deploy a code coverage tool that can return statistics about which lines of code have been exercised and which have not. The ability to combine these results with the coverage data returned from ABV can give the user a more accurate picture of their overall coverage. We refer to this combination of coverage as “comprehensive coverage,” since it gives users a more complete view of their designs.

**STATIC ABV**

The availability of standard assertion languages has allowed formal techniques to be applied to the ABV methodology. Formal tools operate on the design without using simulation. These tools analyze the design based on its structure. Most engineers are familiar with another type of formal verification, equivalency checking, the most popular formal technique used in the EDA industry. Equivalency checking compares two design representations without using simulation vectors, but rather by using mathematical proofs. The result is either a passing grade that the two designs are functionally equivalent, or pointers to sections of the logic that are not the same.

Similar techniques can be applied to assertions. The tools can prove whether a given assertion will hold under all possible circumstances. For instance, if there was an assertion that checked to make sure that a “read” must always follow a “write” after three clock cycles, a static ABV tool would analyze the design and expose any design bug that would violate the desired sequence, generating a waveform for that error condition.

While static ABV has not yet reached the adoption levels of dynamic ABV, it has gotten a lot of attention lately. It allows engineers to uncover bugs earlier in the design cycle, thereby reducing the amount of testbench development and simulation required and speeding time to market. Due to its exhaustive nature, static ABV can find corner-case bugs that might be missed by module-level simulation or even by system-level simulation later in the design cycle.

To properly deploy static ABV, there are some base requirements that make its usage far more valuable. First, it must use the same assertion language that dynamic ABV uses. Engineers don’t want to have to write assertions twice for different tools. And second, it must use the same GUI and debug tools as simulation-based debug to ease adoption and simplify debug. Having these two features dramatically lowers barriers to adoption of static ABV.
ASSERTION-BASED ACCELERATION (ABA)

As designs continue to grow, there is more demand for the ability to accelerate simulation. While simulators continue to make incremental improvements in their efficiency, the fact is that the performance increases are much slower than the growth in design size. As such, more and more customers are turning to hardware-based acceleration.

But hardware accelerators must be able to put as much of the design as possible into the accelerator. This pertains to assertions as well, since the assertions are watching for improper behavior as well as tracking coverage. Most accelerators get the design into the hardware through some sort of synthesis or compilation. This needs to be extended to assertion languages.

Additionally, the user interface and debug tools must be consistent between the simulator and accelerator. Users want to be able to view results from these environments interchangeably, as well as be able to combine coverage results from each.

ABV METHODOLOGY

Now that we have discussed the various tool aspects of assertion-based verification, we need to look at the methodology required to support it. The simplest form of deploying ABV is adding assertions to a design, simulating it, and then waiting for the assertions to trigger or show potential design flaws. But even this basic form of ABV requires some methodology decisions: Where are the assertions placed in the design? How many should I use? Where should I place cover statements?

As static ABV and assertion-based acceleration are added to the flow, it becomes even more complex. The types of problems each of these technologies help uncover vary, and therefore the way they are used is important. Figure 1 shows a possible methodology for deploying the three major types of assertion tools.

As the diagram shows, static ABV (formal analysis) can be used at the block or module level. This allows the design engineer to rapidly verify the block so that it can be swiftly passed on to the next level of verification. It also helps the engineer begin verification before there is a complete testbench available. Static ABV runs very quickly at this level since there is a reasonable amount of logic.
As the block is passed up to higher levels of verification, the assertions stay with it. At the multi-block level, static ABV is still a viable solution. There may also be a testbench available at this level, in which case running simulation and/or acceleration—depending upon the design size—will make sense here.

Finally, at the full system level, all forms of ABV are applicable, depending upon the size of the design and the amount of verification necessary. The flow suggested above maximizes the strengths of different tools in a complementary fashion, shortening the overall verification time and effort.

**SUMMARY**

We have defined the tools and languages required to implement both a basic and a more advanced assertion-based verification methodology. These assertions allow the engineer to leverage design knowledge to identify bugs much earlier than with traditional verification methods. By implementing such a flow, designers can achieve a variety of benefits that include (but are not limited to):

- Catching hard-to-find bugs embedded deep in the design
- Finding bugs earlier in the design cycle, thereby making verification more productive
- Isolating bugs more precisely, thus increasing debug efficiency
- Documenting the design more completely for future reuse
- Reducing simulation cycles by finding errors right when they happen

The net result: by using assertions you will find more bugs, faster.