Assertions: Too good to be reserved for verification only.

Written by
Brian Bailey

Abstract

There has been a lot of talk in the industry about the usefulness of assertions as part of a complete verification methodology. The property languages are stabilizing and production grade tools are appearing so that mainstream companies are able to report success stories. But there is something bigger going on here that many vendors are missing: the value that properties can contribute to the fundamental aspect of the design flow. By combining synthesis with data logging techniques, properties can be turned into full on-chip diagnostics, error logging or usage monitoring systems, and the property languages are a perfect starting point for defining these capabilities. This paper will explore the expanded role for properties in both the verification and design domains. It will show examples from a tool called DiaLite from Temento Systems[1].

Copyright © 2006

Introduction

Assertions have emerged over the past couple of years as a significant mechanism for improving the functional verification of complex systems. While assertions are not new, dating back to the 1950's in the software world, it has only been recently that their value in the hardware community has been demonstrated.

An assertion is the execution of a property, where the property describes functionality or temporal relationships in a formal manner that must or must not be exhibited by an implementation. In other words it is a fragment of an executable specification that can be compared against the result of the implementation effort. Several mechanisms have appeared for the creation of these assertions, with the principle ones being OVL[2], which is a library of relatively simple assertions, and two new languages, PSL[3], the Property Specification Language and SVA, a subset of the SystemVerilog[4] language dealing with assertions, both of which are IEEE standards.

Assertions have been demonstrated to be successful in several conference papers[5] and books[6]. Most of these early success stories were reported by large processor companies, who were the early adopters of the technology, and these stories demonstrate the benefit that they received even without the advantages of the new languages. Not only do assertions increase product quality, but they add additional observability into a design, making it easier to detect and diagnose the problems when they are discovered. In short, what is not to like about assertions?

Assertions in a verification flow

When using properties in a verification flow they have one purpose - ensuring that the functionality and timing that they specify are always adhered to in the implementation. These properties can be exercised in two primary ways, either through a traditional simulator, or in a formal property checker. The beauty of a formal property checker is that it does not require a testbench, as it will examine all possible behaviors of the design and attempt to prove that the property will always hold true. Once this has been done, there is no condition under which the failure can ever happen. In some cases the tool will identify a way that it can be violated, and provide the testcase to exercise that failure. The problem can then be investigated and resolved. Formal property checkers cannot always provide this proof, and there may be properties where neither success nor failure can be provided. Under these cases, the verification team has to fall back on a dynamic execution environment, such as simulation. Now the implementation and the properties are subjected to a set of testcases to see if they result in a disagreement. The problem is that no matter how many testcases are run, the simulator can never prove that it will not fail in the product, and running unlimited test cases on large systems takes too long.

Design for Debug

It is a known fact that bug rates will go up. In a DAC 2003 paper[7], Intel talked about the detected bug rates in each generation of the IA-32 architecture. They showed that total detected bugs were going up by 3 to 4X over each successive product generation. They predicted the bug rate for the next design iteration would top 25,000 bugs. Each and every one of those bugs has to be detected, tracked to its root cause, corrected and re-verified. Anything that can be done to reduce the time spent on each bug will clearly save enormous amounts of total effort.

For many companies, the verification process does not finish with simulation, especially if significant parts of the design are going into one or more FPGAs. FPGAs can provide real time exposure to the actual data streams the design is likely to see and enable the discovery of bugs that are minutes or hours into operation, something that is not possible with simulation. But FPGAs have their own problems when compared to simulators, such as the lack of visibility into what is happening deep inside the design. Some FPGA vendors provide ways to access internal signals[8] that can be viewed on a logic analyzer, but this is far from an ideal situation. Others allow you to capture a few events through a JTAG port, but the data rates on this are severely limiting such that real time operation is not possible.

So, what is a designer to do when they want to diagnose problems deep inside a design? The answer is to build the logic analyzer into the design itself, including triggers, data storage and a mechanism to stream this data out for analysis in a traditional debug environment[9]. A number of companies have been working to provide the IP necessary to perform this operation. They, together with software analysis companies and test equipment companies, recently formed a consortium called the Design for Debug consortium[10] to bring about some standardization to the field.

With the right software, assertions that were defined in the verification flow can also become part of this in-hardware verification. The synthesis of assertions is not easy and care has to be taken to ensure that they do not blow up in size such that they finish up taking more space in the FPGA than the actual design. An IBM designer by the name of Rent[11] made an observation about the relationship between the numbers of pins required by a block versus the size of the block. It basically showed that given the constraints of silicon, pins have always been a constraining factor. What this means is that there is normally logic space available on the FPGA that can be put to good use.

In this article we will be showing PSL examples, but similar properties can be described in the other assertion languages. An assertion is made up of three primary pieces, a number of Boolean expressions, which are linked together into a timed sequence and an action. The first two make up the property. When that property is asserted (the action), it tells the simulator or formal tool what it is expected to do with it, namely ensure that the property holds true. The Boolean expressions in PSL can utilize the syntax of either VHDL or Verilog making it a portable language.

The code snippet below specifies that after a reset, which is defined to be when the reset signal goes from a high to a low, the state machine always goes into the idle state.

```
sequence RESET_SEQ = {reset; !reset}; // reset sequence definition
sequence IDLE = {control == 4'b1111}; // IDLE state definition

property TEST_RESET = always {RESET_SEQ} |=> {IDLE};
assert TEST_RESET;
```

This assertion can then be run in a simulator, or if you want to run in an FPGA prototype, the assertion statements would be synthesized and connected into the design, before it goes through the FPGA mapping process. Any failure of this would result in an error being displayed, such as the one reported by the Temento DiaLite tool in figure 1.

Copyright © 2006

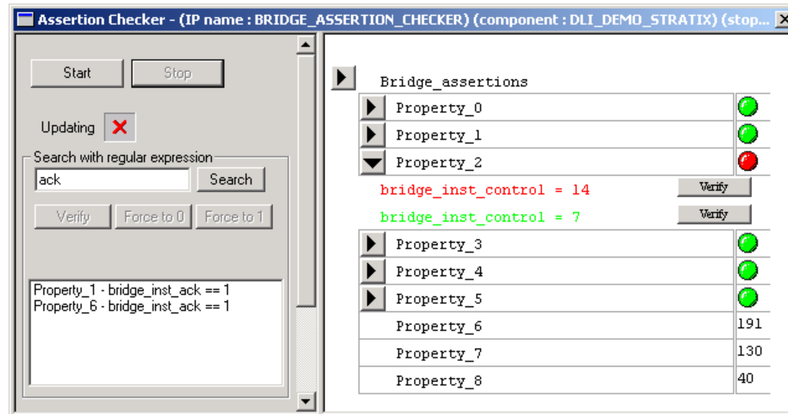

Figure 1: Temento DiaLite tool displaying an assertion failure

The problem can then be debugged by looking at the generated waveform.

Assertions in the design flow

One of the great things about properties is that they allow you to express behaviors in a compact and unambiguous manner. Coupled with the new assertion languages, behaviors have become easier to write and understand. What can be described in just a few lines for a property can consume a whole page or more of RTL description in Verilog or VHDL. For example, consider the following simple example:

```
assert always (req; ack; !halt) |-> next (grant[2])};
```

This says that after you have observed req to be active, followed on the next cycle by ack being active followed on the next clock by halt not being active, then you expect to see grant active for 2 clock cycles. This clearly describes a state machine that implements the behavior and most engineers could probably create it without too many errors, but it would still take longer than writing the single PSL statement.

So, the question has to be asked: why not use property languages for design? The problem has to do with their inability to effectively and completely describe the whole design, and so today this is not possible without language modifications. For example with a state machine it is possible to define the output values associated with each state, but with a property, the complete state machine results in a true or false answer. But what if we constrain the usage of properties to particular tasks? Then it becomes a lot more interesting. Consider such tasks as system monitoring, diagnostics, soft error detection and logging, runtime statistics, or a host of other similar operations. Each of these is based on the detection of one or more events, the collection of data associated with those events, storing the information on chip and then making it available off chip when required for complete analysis. Now this is beginning to look a lot like something that properties can help us define and implement. Of course the design flow will be quite different if you are inserting this logic for built-in self test or diagnostics, as compared to functional verification, and will require different considerations in areas such as coverage measurement. One significant difference is that these assertions are not being used to check for violations in the implementation versus the specification, they are looking for aberrant behavior of the fabricated system, or the detection of conditions under which the design knows how to handle them and possibly correct for them, but the fact that they happened is being logged. All of the logic doing the error detection and logging can be synthesized from the assertions.

Necessary Pieces

Imagine for a moment that we want to build a system that can detect problems on a bus, and whenever a problem is found, it will collect information about what operations were happening just before and after the problem was detected. A number of components are necessary to put together this sample system, including the trigger and trace mechanism, the ability to get the data off the chip and an analysis and display system. Each of these will be described briefly in the next few sections.

Trigger and Trace

The triggers, event capture and memory logging system can all be specified in the Temento Systems DiaLite product. At the bottom of figure 2, a graphical view of some IP blocks which were selected from the library is shown and the data feed between them identified. Not shown in this diagram are the blocks that define what would be recorded in the trace memory neither does it show the controls used to define how much information would be captured before and after the trigger happened.

The leftmost block holds the actual assertions to be defined. In this example, four assertions have been defined, written in PSL. The next box, a User Logic Module, says that we want to trigger the data capture when any of the assertions fire. In other words it provides an OR function on the assertions. The final box is the trace memory. Parameters associated with this box will define the trace depth and other variables associated with this memory block. Other IP blocks would then be instantiated to define which signals or transactions should be captured, and any user defined logic that should first be done on the information before being stored. DiaLite supplies IP blocks such as glitch detectors, range checkers and traffic analyzers that can help to set up this information capture. These analyzers can be quite sophisticated themselves, in that they could be performing complete analysis of a bus or transfer protocol. A vendor who supplies a library of these self contained mini testers can save the designer a lot of time.

```
vunit amba_ahb {

       --AMBA AHB PSL example

       -- define default clock
       default clock is rising_edge ( HCLK );

       -- after reset, HTRANS should be IDLE and HRESP should be OKAY
       property after_reset is
              always( rose ( HRESETn ) -> HTRANS = IDLE and HRESP = OKAY );

       -- after a non sequential transfer type and in burst mode single, transfer
       -- mode can not be sequential or busy
       property after_non_sequential_non_burst is
           always( HTRANS = NONSEQ and HBURST = SINGLE ->
              ( HTRANS /= SEQ and HTRANS /= BUSY ));

       -- after a busy transfer mode when slave is ready, transfer type can not
       -- be IDLE and can not be sequential unless grant is false and ready is true
       property after_busy_and_slave_ready is
           always( HTRANS = BUSY and HREADY ->
              next( HTRANS /= IDLE and HTRANS /= NONSEQ )
              abort (( not HGRANT and HREADY ) or not HRESETn));

       -- maximum wait states: there are never more thant 16 wait states
       property wait_16_or_fewer is
           always( {HREADY; not HREADY}|=>
              {( not HREADY)[*0 to 15]; HREADY} abort ( HRESP /= OKAY ));

       assert after_reset;
       assert after_non_sequential_non_burst;
       assert after_busy_and_slave_ready;
       assert wait_16_or_fewer;

}
```

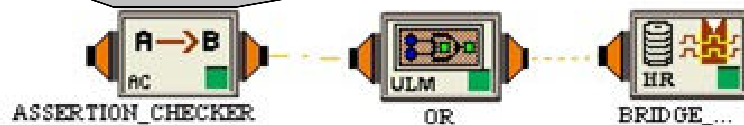

Figure 2: Assertions set up as trace triggers.

Access Mechanism

In our example system, the trace memory would be within the chip and thus their needs be a mechanism to get the data out. This is done through an industry standard mechanism called the Joint Test Action Group (JTAG) Test Access Port (TAP). FPGAs already contain most of the logic to do this, as will most SoCs that contain a processor, so no additional device pins are required. Also the tool will extend the base capabilities to include the access to the memory as well as provide the ability to control some of the embedded instruments. Alternatively, if you have some spare pins on the device, it is possible to provide a much larger external trace memory which can capture a more information than would be possible with internal memory. These external trace storage modules are available from Temento, FS2 and other companies. The extracted data can be stored in any data format, such as the standard VCD file. This can then be viewed and because the condition triggered the storage of the data, it is easy to see exactly what events on the bus, and activity in the system led up to the interesting condition being reached. The Temento display is shown if figure 3.

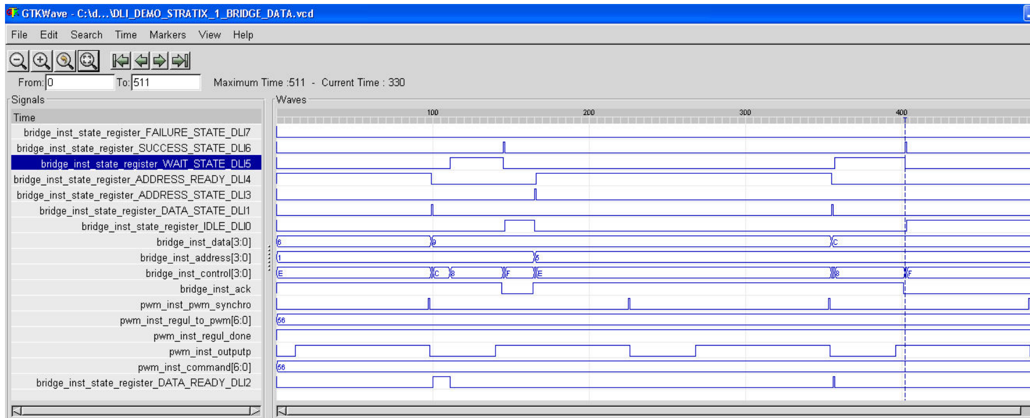

Figure 3: Information from the trace memory can be analyzed and displayed

In a SoC, it is also possible that the trace memory could be implemented as flash storage so that data could be captured and maintained across many power cycles of the system. This would make it a very effective way to record and analyze soft errors in the system.

Conclusions

In this paper, we have taken a very quick look at using assertions in ways that most people would not consider today. Rather than constraining their usage to the verification flow, I believe that the property languages are a powerful way to define significant pieces of the hardware as well, and languages like these are actually more likely to become the system languages of the future rather than the marginally extended capabilities of today's RTL languages. Tools on the market today have all of the capabilities necessary to implement such systems and we should watch out for them becoming significant players in the ESL landscape of the future.

References

---

[1] Temento Systems S.A.; Website http://www.temento.com

[2] Open Verification Library; Accellera; http://www.accellera.org/activities/OVL_VSVA

[3] Property Specification Language Reference Manual Version 1.1; Accellera (www.accellera.org), June 2004.

[4] System Verilog 3.1a Language Reference Manual; Accellera (www.accellera.org), May 2004

[5] Kantrowitz, M.; Noack, L.M..; *I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor*; Design Automation Conference, 1996.

[6] Harry Foster, Adam Krolnik, David Lacey; *Assertion-based design*; Kluwer Academic, 2003.

[7] Tom Shubert; *High Level Formal Verification of Next-Generation Microprocessors*; Design Automation Conference 2003

[8] ChipScope Pro; Xilinx; http://www.xilinx.com/ise/optional_prod/cspro.htm

[9] Chiang, C; Bailey, B.; *Why did my Chip do That?*; DesignCon 2006

[10] Design for Debug Consortium; http://www.designfordebug.org

[11] H. B. Bakoglu; *Circuits, Interconnections, and Packaging for VLSI*; Addison-Wesley, Reading, MA, 1990.

About the author:

Brian Bailey is an independent consultant helping EDA and system design companies with technical, marketing and managerial issues related to verification and ESL. Before that he was with Mentor Graphics for 12 years, with his final position being the Chief Technologist for verification, Synopsys, Zycad, Ridge Computers, and GenRad. He has published three books, *Taxonomies for the Development and Verification of Digital Systems* (Springer, 2005), *The Functional Verification of Electronic Systems: An Overview from Various Points of View* (IEC Press, 2005) and *ESL Design and Verification: A Prescription for Electronic System Level Methodology* (Morgan Kaufmann/Elsevier 2007). He chairs two standards committees and has three patents awarded. He graduated from Brunel University in England with a first-class honours degree in electrical and electronic engineering.

He can be contacted at brian_bailey@acm.org or through his website:
http://brianbailey.us