ENABLING ASSERTION BASED VERIFICATION

WHITE PAPER

NOVEMBER 2010

www.zocalo-tech.com

Contents

OVERVIEW 3

CHALLENGES OF ABV TECHNOLOGY 5

Indentify Assertion Candidates 5

Coding Assertions 5

Assertion Debug 6

Assertion Control 6

ZAZZ™ FROM ZOCALO FOR ENABLING ABV 6

Bird Dog and Metrics 7

Zazz Visual SVA™ 9

Zazz Migrate 12

Assertion Library Support 13

Zazz Infrastructure 14

User Interface 14

Design Database Support 15

Design Viewer 15

CHALLENGES OF IMPLEMENTING ABV 16

CONCLUSION 18

ABOUT ZOCALO 18

Copyright© 2010 Zocalo Tech, Inc

OVERVIEW

Assertions are properties or facts describing the required and forbidden behavior of a design. They are "executable specifications" that are monitored during simulation by *assertion checkers* included in the design file.

- Through usage assertion checkers are referred to as assertions for short.
- The term assertions and properties are often used interchangeably.
- The prevalent language for writing assertions is SystemVerilog Assertion (SVA) language, a specialized subset of SystemVerilog.
- Assertions written in the SVA language are referred to as SVAs.
- Assertion Based Verification (ABV) is the methodology for including assertions in the functional verification process.

If the assertion fails during simulation the user is notified. Depending on the severity of the failure, it could even stop the simulation. If the assertion does not detect a problem there are two possibilities. The design is behaving correctly or the testbench did not cover the property. Therefore, cover properties typically are included with an assertion. The default condition used by Zocalo is to automatically create a cover property for every assertion. Throughout this paper, for brevity, the term assertion means that a corresponding cover property is included.

SystemVerilog to include ABV is viewed as *the evolving standard* that can have major impact on reducing verification time and cost. Various studies have shown that using ABV can reduce debug, now representing 60% of the functional verification time and cost, by 50%. In spite of the promise of ABV, wide scale use has not materialized. ABV is a difficult technology to implement and is perceived as marginally cost effective. If it were easy everyone would have adopted it by now.

The objective of this paper is to:

- Understand the issues limiting the use of ABV.
- Define how Zocalo is addressing them.

Key to this paper is the difference in the terms "*Using Assertions*" and "*Assertion Based Verification*".

- *Using Assertions* is an ad hoc process dependent on the *skill* and *desire* of the designer or verification engineer to provide assertions as part of the functional verification process.
- *Assertion Based Verification (ABV)* is a systematic methodology <u>*requiring*</u> the use of assertions in the functional verification flow *and* the infrastructure to control and manage them.

Most surveys on using ABV reflect *using assertion as* opposed to *ABV*.

ASSERTION USE TODAY

The prevalent approach today is *using assertions on an ad hoc basis* by designers at the module or functional block level and verification engineer's at the higher levels. Assertion use at the functional block level is more prevalent since bugs at that level showing up during system level verification can result in a major schedule hit. Since designers are intimately familiar with the intent of the design and many are capable of coding simple 1 or 2 cycle SVAs, they are more likely to create and add assertions.

Copyright© 2010 Zocalo Tech, Inc

Since assertion use is at their discretion, they are usually added in-line as part of the design file without documentation.

At higher levels up to the system level, the useful SVAs typically require a higher level of complexity than at the functional block level. Temporal (time dependent) properties reflecting concurrency are the norm. Since the verification engineer typically cannot modify the design files, bind files are required. Creating and managing bind files is a labor intensive task. The SVA learning curve can be steep for complex temporal expressions; therefore assertion use, even on an ad hoc basis, is sparse. Certainly, many companies have SVA coding experts that can be assigned for situations where the need for a complex temporal SVA is obvious. However, the broad base of verifications engineers lack the expertise to write this level of assertions.

Only the larger companies are embracing ABV on a systematic basis focusing on more manageable simple assertions at the block level and below. The broader base of chip design companies recognize the promise of ABV, but have not committed because of the effort required.

Per Harry Foster in his paper Assertion-Based Verification: Industry Myths to Realities (Invited Tutorial…2008) defines the situation more bluntly. *"…..it is a myth that ABV is main stream technology."*

……"what differentiates a successful team from an unsuccessful team is process and adoption of new verification methods. Unsuccessful teams tend to approach development in an ad hoc fashion, while successful teams employ a more mature level of methodology that is systematic". ……

Moving from *ad hoc assertion use* to full scale *Assertion Based Verification*, as depicted in Figure 1, represents a major hurdle.

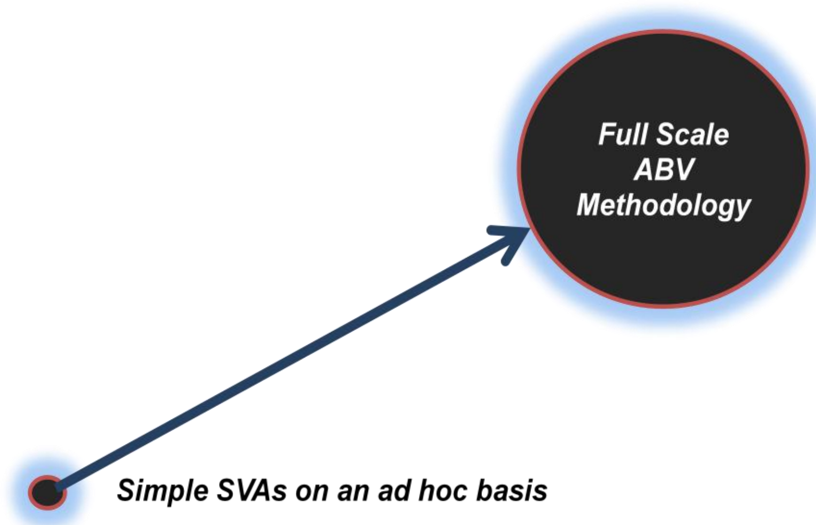

Figure 1 Enabling ABV

Copyright© 2010 Zocalo Tech, Inc

CHALLENGES OF ABV TECHNOLOGY

A systematic methodology for adding assertions requires four steps as shown in Figure 2.

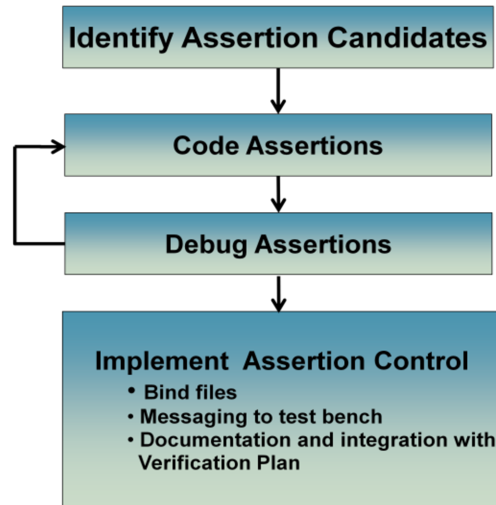

Figure 2 Adding Assertions in a Systematic Methodology

Indentify Assertion Candidates

Although designers are in the best position to identify assertions candidates and use them often, are their choices complete and objective? When using an ad hoc approach, the choices can be skewed to simpler assertions based on assertions coding skills. Also the impact on design time is a major inhibitor to a complete assessment of assertion requirements.

If the designer or verification engineer is unfamiliar with the design as in the case of IP or legacy code, becoming familiar enough with the design to identify the assertion candidates becomes even more difficult and time consuming. Dependent on complexity of the design, this task is measured in days or weeks.

A further challenge is if the company or project is considering or already committed to ABV. How can the effectiveness of ABV use be evaluated in relation to the quantity and quality of assertions used?

Coding Assertions

The user must abstract and code the 'correct' property. The verification engineer and oftentimes the designer must code complex expressions. The SVA language is not intuitive and the temporal expression operators and concurrency are difficult to visualize. The skills required are costly to develop, difficult to retain and inhibit the development of these more useful and powerful SVAs.

Recognizing the difficulty of coding assertions and the requirement for coding consistency for managing them, Accellera, the industry standards organization developed Open Verification Library (OVL). These libraries do not cover complex assertions, but do address a major part of assertion requirements at the block level. OVL represents the first effort to bring consistency and structure to assertion use. Each of

Copyright© 2010 Zocalo Tech, Inc

the major EDA vendors has "OVL like" libraries tuned to their simulation environment and they are typically included with the vendor's simulator license. OVL libraries are available from Accellera at no charge and are compatible with any of the leading simulators.

However, making use of assertion libraries tends to be time consuming and error prone thereby limiting their acceptance. Additionally many designers consider assertion libraries as too inflexible. Since most designers are capable of writing SVAs at the OVL level of complexity, in the world of ad hoc assertion use (i.e. *no bind files, no documentation, minor if any debug),* designers will choose to write simple SVAs over assertion library use.

Assertion Debug

After the user codes a complex SVA property, it is still very raw and unproven. To test the property, the user must create a testbench to validate that the property is correct. For a complex assertion this can be a major task and in most cases impractical. The other approach is to debug the assertion during normal simulation. This places an additional level complexity and lost time. For this reason, the most commonly developed properties continue to be simple 1-2 cycle properties that require minimum debug, if any. Debug is a major factor limiting the development of more useful and powerful SVAs.

Assertion Control

Under an ABV methodology, assertion control includes:

- Assertions managed in separate files (bind files) that can be included with the design file during simulation and removed.
- Documentation of assertions to be included as part of the verification plan.
- Control of assertions ranging from disabling the SVAs, to changing variables such as severity levels preferably from the testbench.
- Reporting error messages/events using the testbench message logger.

The major requirement to accomplish the preceding is packaging assertions in a consistent manner to allow a level of automation.

ZAZZ™ FROM ZOCALO FOR ENABLING ABV

Zocalo Tech believes that automation is the enabler of Assertion-Based Verification and wide scale acceptance is directly proportion to the level of automation. Automation from Zocalo Tech is provided by Zazz that includes the following modules:

- Zazz Bird Dog – Analyzes the design to find the most important candidate signals where properties should be added. Signals that already have legacy properties are recognized and automatically listed as assertion candidates.

- Zazz Metrics - Shows the progress towards goals and the quality of added properties. If legacy properties are part of the design code, they are automatically included as part of the metrics.

Copyright© 2010 Zocalo Tech, Inc

- Zazz Visual SVA – Enables creation and debug of any level of SystemVerilog Assertions (SVAs) complexity without learning the language. Visual SVA completely eliminates the long learning time typically associated with becoming proficient writing properties with the SVA language. Debug is completely independent and outside of the simulation environment. Visual SVA also provides dynamic controllability of assertions and parameter changes plus automatic bind file management and documentation.

- Zazz Migrate – provides a migration path for legacy designs already populated with assertions to assure full compatibility with Zazz.

- Assertion Library Support – including, OVL and libraries from the major EDA and custom libraries.

Bird Dog and Metrics

Bird Dog and Metrics provide the first step towards implementing ABV by automatically identifying and ranking potential assertion candidates. A heuristic algorithm is used that was developed on basis of how a design or verification engineer would evaluate the need for assertions.

The prevalent use of assertions today is designers adding simple one or two cycle SVAs on an ad hoc basis or adding comments or pragma's to enable automatic inclusion of simple assertions to the design. Using only the Bird Dog and Metrics capabilities can significantly increase the level of productivity and results for projects using these approaches.

Bird Dog

Bird Dog is critical for engineers unfamiliar with ABV since it provides guidance on which modules and signals are most likely to benefit from the addition of assertions. Likewise, engineers experienced with assertions are likely to discover new signals as the tool analyzes the signal usage globally versus a designer's tendency to focus on the module level. For legacy code, performing a manual assessment of assertion requirements for a complex functional block can take days versus minutes with Bird Dog.

Since assertions are rules describing how the design should behave, they are most useful for control signals. As Bird Dog elaborates a design, it automatically analyzes the signals for control and data characteristics and further factors in how the signal is used both within the module and across the hierarchy. It assigns a score to each signal and then ranks the signal as an assertion candidate (its Global Rank). They are then placed in buckets of the top 10%, 20%, etc., relative to importance as shown in Figure 3. Typically the top 20 to 30% of the candidates are adequate for populating a design.

Identifying the assertions candidates and targeting a defined number to populate with assertions allows the project to scope the work required. It is emphasized that Bird Dog is not a "magic machine". It does not determine which assertion to use or create assertions. It provides a starting point by rigidly analyzing the design and reporting the results to the user. The user makes the decisions on which assertions to provide. The analogy and thus the name are akin to the role of a bird dog. The bird dog is part of the process of finding, flushing and recovering the birds, but the hunter makes the decisions.

Copyright© 2010 Zocalo Tech, Inc

Once the candidates are identified, Bird Dog provides several views of the signal rankings. The hierarchy browser includes the number of candidates within each level of the hierarchy, enabling engineers to quickly identify the modules that are more control-oriented. Thus, engineers can quickly focus on the portions of the design that benefit from assertions instead of spending time on less critical modules.

After selecting a module from the hierarchy, the user is presented with a list of candidates from within that portion of the design in the candidate browser. Zazz Bird Dog lists each candidate signal along with its associated rank and attributes such as whether the signal is a port on the module. Selecting a candidate causes the source view to display the signal to allow the user to further investigate how it is used in the design.

The Candidate Browser has two options that can be used to help filter signal candidates within the hierarchy. The “Hierarchical” filter selects all candidates in the hierarchy while the “Net View” filter collapses common nets (connected through module ports) into a single instance. These filters can be used to identify the global control signals versus those of interest in the selected module providing engineers the ability to quickly identify the signals most likely to benefit from assertions.

Assertion Candidates

☑ Include All Candidates ☑ Hierarchical View

Signal [36]	Unit	Rank	Bucket	Chkrs	Factors
eop	tx_dequeue	59	+10%		Large fan-out
start_on_lane0	tx_dequeue	26	+10%		Large fan-out
next_ifg_8b_add	tx_dequeue	19	+10%		Large control signal fan-in
next_ifg_4b_add	tx_dequeue	18	+10%		Large control signal fan-in
next_ifg_deficit	tx_dequeue	12	+10%		Large control signal fan-in
txhfifo_rstatus	xge_mac	11	+20%		Large fan-out
reset_xgmii_tx_n	xge_mac	10	+20%		Highly used control signal
next_eop	tx_dequeue	10	+20%		Large control signal fan-in
txdfifo_rstatus	xge_mac	8	+30%		Highly used control signal
frame_available	tx_dequeue	7	+30%		Highly used control signal
txhfifo_wen	xge_mac	6	+30%		Highly used control signal
next_ifg_8b2_add	tx_dequeue	5	+30%		Large control signal fan-in
txhfifo_wstatus	xge_mac	5	+30%		Highly used control signal
byte_cnt	tx_dequeue	5	+30%		Highly used control signal
txdfifo_ren	xge_mac	4	+40%		Highly used control signal / Large fan-out
ifg_deficit	tx_dequeue	4	+40%		Large fan-out / Highly used control signal
shift_crc_eop	tx_dequeue	4	+40%		Large control signal fan-in / Highly used control signal
status_local_fault_ctx	xge_mac	3	-40%:+50%		Highly used control signal
ifg_8b_add	tx_dequeue	3	-40%:+50%		Highly used control signal / Large fan-out
next_txhfifo_wstatus	tx_dequeue	3	-40%:+50%		Highly used control signal / Large fan-out
shift_crc_cnt	tx_dequeue	3	-40%:+50%		Highly used control signal / Large fan-out

Figure 3 Bird Dog Viewer

Metrics

Equally important as identifying assertion candidates is the ability to provide metrics on the number and quality of the assertions present in the design. Metrics provides the project team with an on-going progress report about the quantity and quality of the assertions added during the project versus the target.

As Metrics elaborates the assertion populated design, it also counts the number of assertions touching each signal within the design as shown in Figure 4. A summary of the number of assertions found at each level of the hierarchy is displayed in the module browser, and a count of how many assertions that touch each signal candidate is displayed in the candidate browser. As a result, engineers can quickly identify which modules and signals are in need of further assertions.

Copyright© 2010 Zocalo Tech, Inc

Bird Dog also detects the quality of the assertions based on the density of the assertions within the design and their relationship to the candidate list and complexity.

On a longer term basis, metric reports associated with completed designs provide a basis to measure the value of using assertions. As the company moves from ad-hoc use of assertions or to higher value assertions, Metrics provides a historical base to measure the effectiveness of this ABV growth path.

Design

Search: *Any* | 0 0 | Clear | Options

Hierarchy	Signals	Assertion Candidates	Assertion %Candidates	Instrumented Candidates	Instrumented %Candidates	Checkers	Instantiated Checkers	Density
demo	10124	1025	%10.1	55	%5.4	30	165	0.55
pj cpu	9990	1023	%10.2	53	%5.2	30	165	0.57
iu	5321	472	%8.9	24	%5.1	17	49	0.71
ex	2462	224	%9.1	9	%4.0	6	14	0.67
ifu	796	76	%9.5	5	%6.6	8	25	1.60
rcu	1140	74	%6.5	5	%6.8	3	5	0.60
pipe	440	59	%13.4	5	%8.5	4	5	0.80
ucode	552	49	%8.9					
trap	188	40	%21.3					
hold logic	60	29	%48.3					
fpu	2650	276	%10.4					
cs	440	83	%18.9					
mult1	775	77	%9.9					
exp	461	49	%10.6					
inc	404	43	%10.6					
man	285	42	%14.7					
rsa	325	34	%10.5					
prif	154	30	%19.5					
nxs	42	10	%23.8					
dcu	836	149	%17.8	10	%6.7	7	12	0.70
icu	764	85	%11.1	10	%11.8	7	100	0.70
smu	519	58	%11.2	3	%5.2	3	3	1.00
dtag shell	233	10	%4.3					
pcsu	37	9	%24.3	1	%11.1	1	1	1.00
dcram shell	168	6	%3.6					
icram shell	113	5	%4.4					
itag shell	119	4	%3.4					

Figure 4 Metrics Viewer

Zazz Visual SVA™

Zazz Visual SVA enables design and verification engineers to quickly code and debug assertions without having to deal with the complexities of the SVA syntax. Just as Visual Basic did for the BASIC programming language, Visual SVA raises the level of abstraction for describing properties and completely hides the complexities of using the SVA language. Visual SVA provides major productivity enhancements for creating and debugging from simple to extremely complex properties reflecting concurrency and temporal. Creating properties in Visual SVA is a combination of visually arranging components or controls on a graphical canvas, adding Boolean expressions and then selecting attributes and actions of those components. Additionally all of the "housekeeping" tasks required for implementing ABV are automated.

A temporal view of the SVA property is created using drag-and-drop techniques. Components and controls are selected from a pallet and placed on the property canvas (window). The property canvas, as shown in Figure 5, assumes that time increases from the left side to the right side, just like the familiar waveform viewer. Concurrent execution threads are represented in the vertical direction, again as in a waveform viewer. Visual SVA will automatically connect the components and controls, based upon how they are arranged on the property canvas. Boolean expressions are entered into the expression field utilizing the same expression builder that exists in Zazz OVL. Components and controls have default attributes associated with them, but may be changed by selecting from a dropdown list.

Visual SVA dynamically creates the correct SVA code to describe the property and displays it in a code

Copyright© 2010 Zocalo Tech, Inc

view panel.

Figure 5 is an example of a typical Visual SVA completed code and an example of automatically generated SVA code shown in Figure 6.

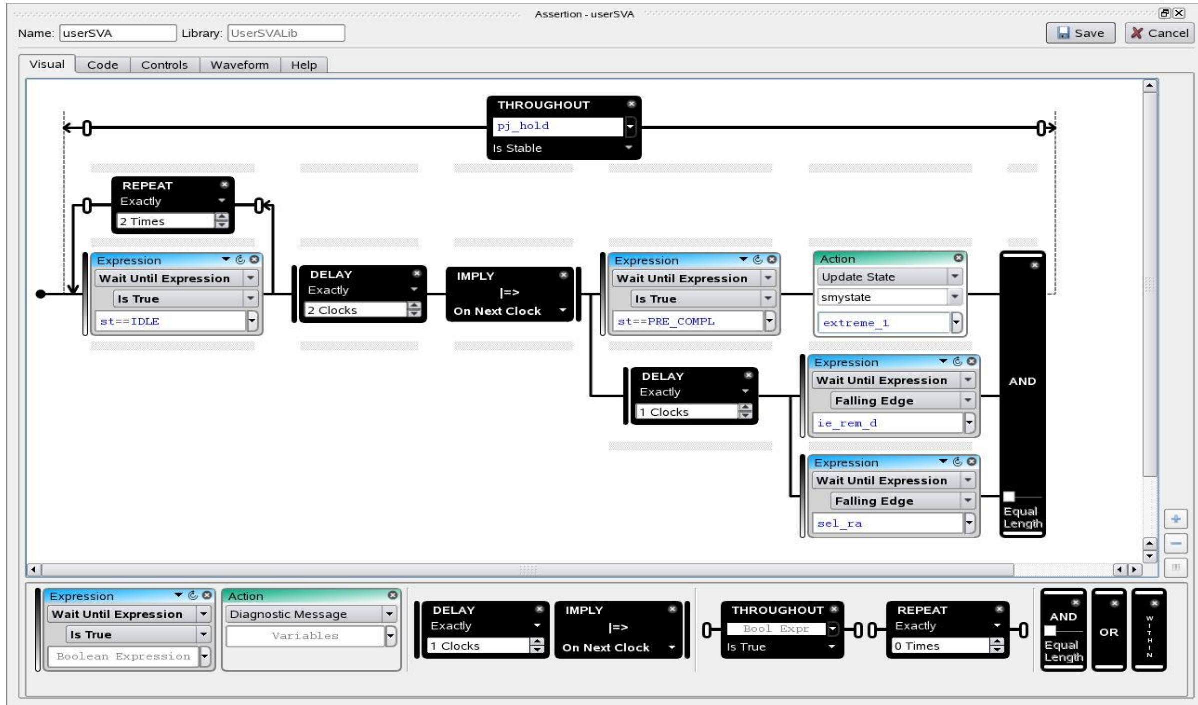

Figure 5 Visual SVA code

Assertion - userSVA

Name: userSVA Library: UserSVALib Save Cancel

Visual | Code | Controls | Waveform | Help

Assertion Code

```
typedef struct {
  logic extreme_1;
} State_mystate;

property userSVA;
 State_mystate mystate;

 (($stable(pj_hold)) throughout ((((st==IDLE)  ) [->2]) ##2 )  |=>
 (
   (((st==PRE_COMPL) [->1] ), mystate.extreme_1=extreme) and
   (##1 ((($fell(ie_rem_d)) [->1])) and (($fell(sel_ra)) [->1] )))
 );
endproperty
```

Operators: Delay, Repeat, not, and, or, intersect, throughout, within, Waveform

Figure 6 SVA Code Example

Copyright© 2010 Zocalo Tech, Inc

When the engineer is finished describing the property, the following capabilities are accessible.

Automatic timeout generation Visual SVA has the ability to insert timeouts to ensure that open-ended temporal expressions that are not satisfied are reported and detected during the course of simulation. Zazz identifies open-ended properties and automatically inserts time out expressions in the form of parameterized static time out or dynamic time out that can be specified at run time.

Auto-generation of SVA testbench and Waveform Visualization Visual SVA automatically generates a testbench for any expression or assertion to aid the user in debugging an assertion. Users can quickly see passing or non-matching cases without having to construct their own testbench or run the assertion in a functional simulation in hopes of finding a passing or failing case. Results can then be displayed as waveforms. (This feature requires an external simulator and waveform viewer).

A typical approach to building an assertion with Visual SVA uses this capability extensively. The users starts with a simple assertion and adds complexity based on interacting with the design hierarchy and recognizing additional scenarios that can take place that need to be checked as part of the property. As complexity of the assertion is increased, at any time by user command, an assertion testbench is automatically created, executed and presented for viewing via the user's wave form viewer. The resulting wave forms allow the user to almost immediately examine the output for the intended results. A typical wave form display is shown in Figure 7.

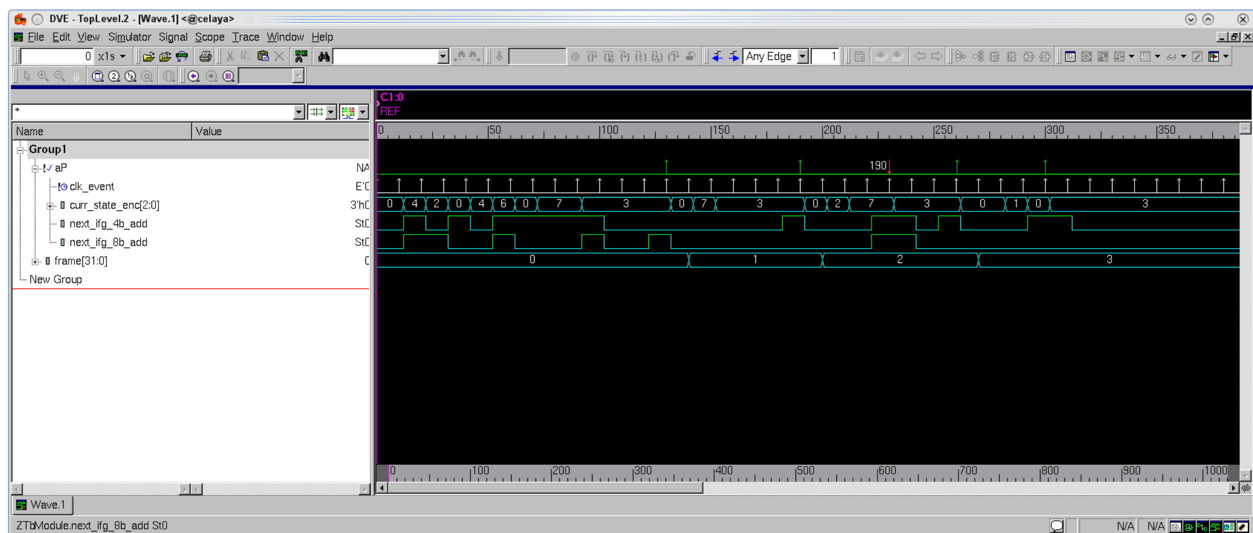

Figure 7 Wave Form Display

Assertion Control and Reuse Visual SVA facilitates assertion control and reuse of any defined SVA property by providing auto-parameterization for anything that must elaborate to a constant. For signal ranges (if a signal with width is selected for inclusion in an expression) then Zazz includes a WIDTH parameter declaration or type declaration. The code is stored in a user library for reuse, modification or extended and saved under the same or a new name.

Copyright© 2010 Zocalo Tech, Inc

Automatic Bind Files and Documentation Visual SVA provides automatic bind file creation and verification plan documentation similar to that described under Zazz OVL. Including the temporal view with the documentation provides a major enhancement to the documentation.

Control Infrastructure Visual SVA auto-parameterization provides control of assertions from within Zazz environment. Future releases will provide integration directly to OVM and VMM allowing controls such as disabling the SVAs, changing their severity levels and reporting error messages/events using the testbench message logger.

Zazz Migrate

For legacy designs that already have assertions, it is important to have a migration path to assure all assertions have a consistent structure compatible with Zazz. Migrate provides this capability by automatic conversion of legacy assertions to full compatibility with Zazz. Migrate is a major requirement for long term users of assertions to move to an expanded capability such as Zazz.

Examples of Migrate are provided in the following screen shots.

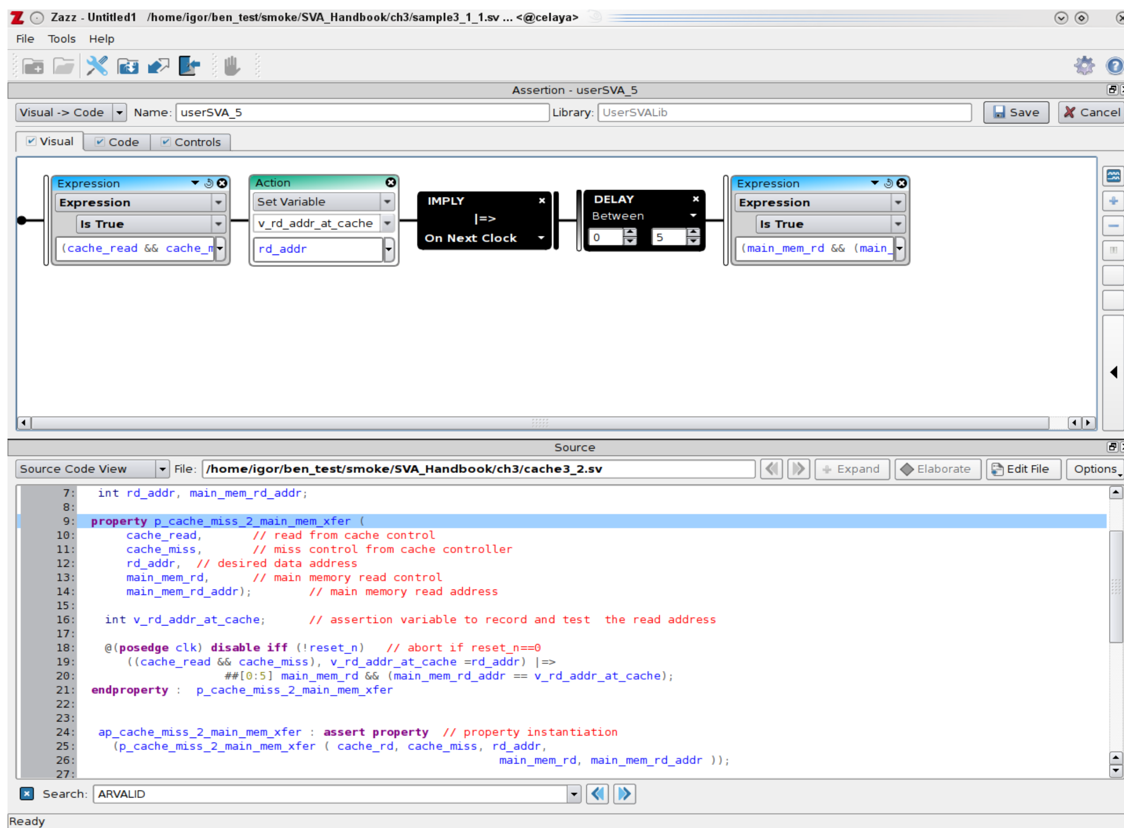

FIGURE 6 LEGACY CODE ON BOTTOM – RENDERED ZAZZ VISUAL VIEW ON TOP

Copyright© 2010 Zocalo Tech, Inc

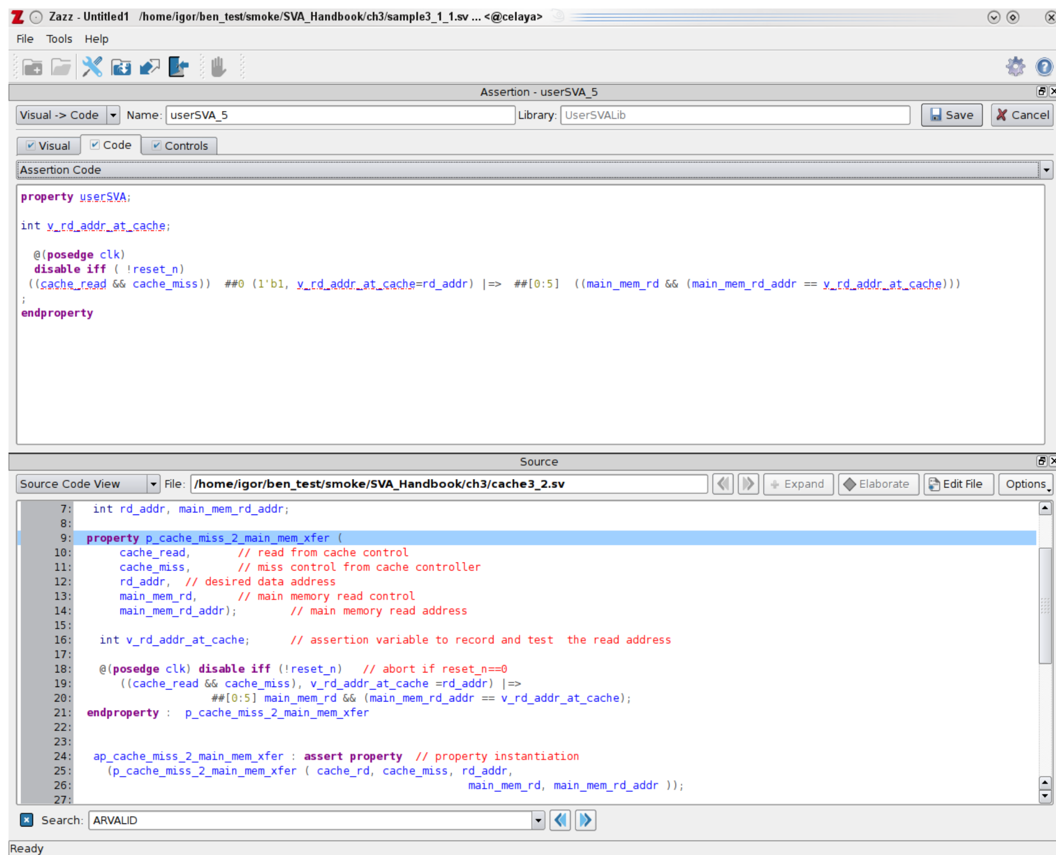

FIGURE 7 LEGACY CODE ON BOTTOM – RENDERED ZAZZ VISUAL VIEW CONVERTED TO CODE FORMAT ON TOP

Assertion Library Support

Support is provided for Accellera Open Verification Library (OVL), Cadence's IAL, Mentor's QVL and Synopsys' SVA _CG. Multiple libraries may be used together. Additionally support for custom assertion libraries is available. Without any level of automation, manual use of assertion libraries to describe a property can take significant time. The probability of an error is high adding additional time to the process.

With Zazz OVL a property can be selected configured and attached in minutes and the most highly used checkers in less than a minute. Bind files and documentation are automatic by-products.

Copyright© 2010 Zocalo Tech, Inc

Zazz Infrastructure

All Zazz capabilities share a common infrastructure.

User Interface

Zocalo considers ease of use to be a major factor in the acceptance of a software tool. EDA tool development has traditionally lacked this emphasis. To this end the Zazz GUI (Graphic User Interface) is based on the latest graphical user interface design tools. The priority is to assure the Zazz GUI provides a highly productive interface between the user and the application.

Major features include:

- All application windows are dock-able and resizable.
- The GUI multithreaded implementation provides both a high level of performance and access to the application during intense computations.
- Designed for multi-headed displays.
- Dynamic SystemVerilog syntax and elaboration error highlighting.

Figure 10 is a display showing access to Bird Dog and Metrics as assertions are being developed in Visual SVA.

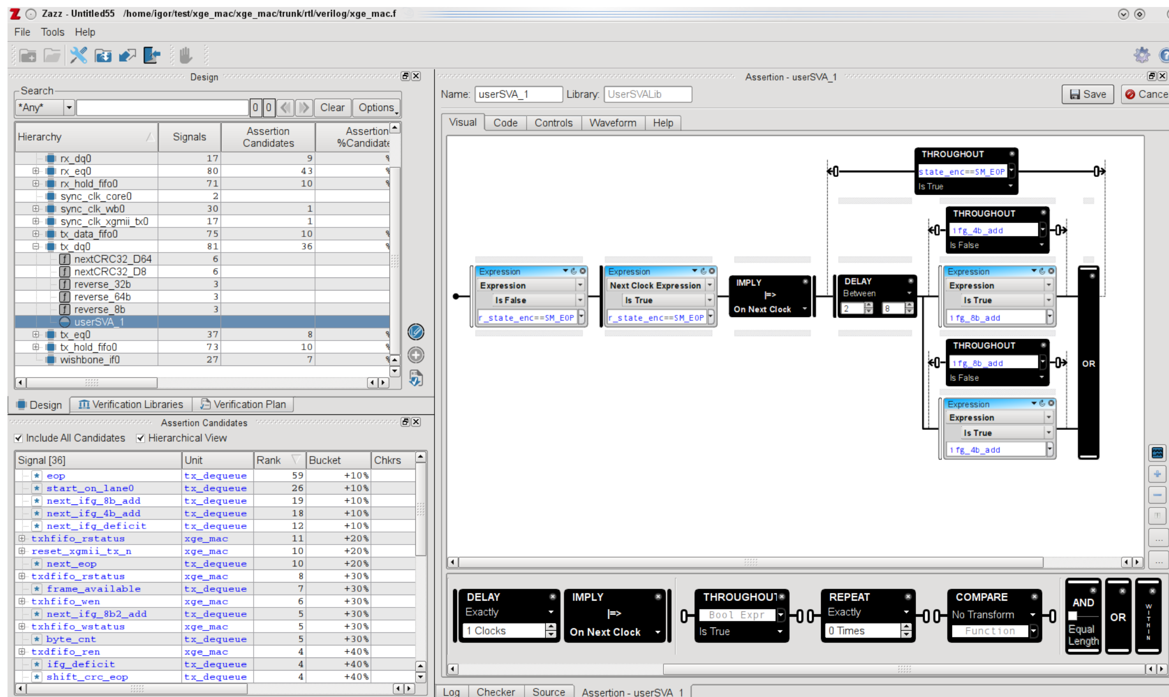

Figure 10 User Interface Example

Copyright© 2010 Zocalo Tech, Inc

Design Database Support

Zazz is Linux-based and supports any mix of Verilog 1995, Verilog 2001 and SystemVerilog design files. Zazz is built on an advanced incremental parser/elaborator and design viewer. An existing design or new design can be read into Zazz and parsed, elaborated, and graphically displayed and modified with the user's editor of choice without leaving Zazz. When a design is modified and saved, it is incrementally parsed and elaborated. The incremental feature, along with the built in modification monitoring provides fast update to the graphical display along with feedback on any errors.

Design Viewer

Zazz's Design Viewer is not the typical design browser. It is an advanced design viewing, navigation, and modification monitoring application. When a design element is selected in the Design View, the Source panel displays the associated source code. This feature gives the user direct access to the RTL regardless of its location in the file system. Direct access can be a major time saver, especially for users not intimately familiar with the entire design file structure. Any of three views of the RTL code may be selected:

- The *Source Code View* is the default view and has an "Edit File" icon for opening an Editor panel/window where the source file may be edited using the designer's favorite editor. Zazz monitors the files being edited, and when one is saved it re-parses and elaborates the design. This feature gives the user immediate feedback on any syntax or elaboration errors.

- The *Processed Code View* is the RTL after processing all of the text substitution macros. Macros can make significant modifications to the RTL that will be elaborated. When viewing the raw source code, determining where the macros were defined and how their definitions were changed by other macros can be very time consuming. The Processed Code View saves that time by directly showing the actual code to be elaborated.

- The *Elaborated Code View* shows the RTL after all parameters have been resolved to their constant value. Signal widths are shown directly as a range of numbers. This negates the need to locate the assigned value of parameters in order to determine the actual width of signals. This view is especially helpful when connecting the ports of assertion checkers since the exact widths of the signals being configured is known. Knowing the exact signal widths helps prevent port size mismatch warnings later in the verification process.

In addition to the standard design hierarchy browsing, Zazz provides a powerful regular expression (regex) search of various name spaces: Module, Instance, Binding, Interface, Task, and Function. For example, when the Module name space is selected with no regex expression, Zazz displays how many modules exist in the design and allows the user to step thru them. With the Module name space selected and a search expression of "ram$" is entered, Zazz displays how many module names end in "ram". The user may step thru the list, selecting each one in turn, with the Source panel displaying the RTL code.

The ability to find syntax or elaboration errors in real time plus the navigation features of searching the design file structure to locate the files of interest to view and/or modify can save untold hours over the life of the project. The Log panel contains links to the errors and warnings encountered. Clicking on a link changes the Log panel to a view of the source file with the error highlighted. If the Edit File icon is clicked then an Editor panel is opened with the cursor close to where the error/warning occurred. Icons

Copyright© 2010 Zocalo Tech, Inc

in the source view in the Log panel allow the user to step to the next or previous error/warning. The Zazz Design Viewer represents a major productivity advantage for the user even when assertions are not being used.

CHALLENGES OF IMPLEMENTING ABV

Verification approaches or methodologies *that increase the probability of designs being correct the first time* and can be easily integrated into the existing functional verification flow should find a ready market. If in addition this technology reduces both verification time and cost it should become a major winner.

ABV is in that class of technology. However, ABV has not been widely adapted because of the time, cost and difficulty of deployment.

Implementing ABV is somewhat of a "chicken and egg" situation. The industry accepts that ABV can reduce debug time by 50% and there is no question relative to its "goodness" for increasing first time design success. However, the implementation phase today is so open-ended and difficult that only "baby steps" have been taken. Even these small steps are useful; however, the industry hasn't come close to attaining the full

As described by a potential customer*, "The main reason we are looking for an 'assertion builder' tool is to help our logic and verification teams create lots of high quality assertion for our full chip and Verilog models. Our team has written some basic assertions, but are very minimal. We need some kind of 'tool' to get us there. We can force people to learn the SVA language, but, as the Zocalo guys stated, this is difficult. People won't do it unless we help them find an easy way"*

The preceding statement is typical of the industry. Clearly the opportunity exists to provide help for implementing ABV. The methodologies are already in place. Effective implementation is the issue. The objective of Zazz is to significantly reduce the difficulty and open ended nature of the technology working seamlessly within the predominant verification flows. Automation will motivate projects to start and continue using ABV, thereby reaping the full benefits of implementation.

Assertions use is at two levels:

- Designer provided assertions at the block level.
- Verification engineer provided assertions at the sub system or systems level.

Assertions added early in the design process by designers, can detect a large percentage of design bugs throughout the design process. Designer provided assertions clearly represent significant leverage to the project. In addition:

- The process of thinking about the intent of the design in order to create assertions forces the designer to review the code more critically gaining a better understanding, thereby catching many design errors.
- These assertions also reduce communication time between design and verification engineers by quickly identifying testbench behavior issues and isolating design bugs.

While designer provided assertions represent a powerful addition to a design team's arsenal, adding quality assertions does take up time during the design process. Designers may resist adding assertions because of tight time constraints and the complexity of the SVA language. In spite of the proven time savings for designers by reducing debug time by up to 50%, adding assertions to the design process has

Copyright© 2010 Zocalo Tech, Inc

remained a hard sell. Zazz drastically reduces the impact on their design time. This reduction in time makes the full benefits of designer provided assertions as an integral part the design flow a reality. This in turn increases the confidence and productivity of the total project. For an in-depth discussion of designer provided assertions challenges and approaches written by a designer who was instrumental in the development of Zazz, see http://cyclicdesign.com/zocalo/tutorial/index.php.

While designer provided assertions appears to be the method of choice to get started in ABV, assertions provided at the sub-system and system levels by verification engineers also provide major benefits for reducing verification time and reduction in re-spin. However, verification engineers have steered clear of ABV because effective assertions at that level typically require more complexity. This in turn requires more expertise with the SVA language and more time to debug. Like any other code, assertions need to be debugged. In the existing environments, debug typically takes place as part of the simulation run. This adds another area of complexity and limits the number of opportunities to debug the assertion based on the turnaround time required between simulation runs of large designs. A verification engineer would be lucky to get five debug iterations opportunities per day with a large design.

A recent evaluation was done by a large company SoC design group. This early evaluation showed the potential for integrating cost effective Assertion Based Verification (ABV) into their flow both by both designers and verification engineers. They decided to use Zazz in the production flow for adding assertion to critical interfaces at the sub-system level prior to prototype tape-out. They were able to create high value assertions seamlessly within their verification flow and the benefits of using Zazz were immediate:

- Visual SVA eliminated the SVA learning curve.
- Visual SVA allowed assertions to be debugged individually and cost effectively outside of the simulation environment.
- The visual output of Visual SVA allowed designers and verification engineers to easily review, understand and communicate the purpose and intent of the assertion. The visual aspect of the tool did away with the “black magic” of assertion use.
- Zazz’s ability to automatically manage the assertions allowed assertions to be used seamlessly within their existing flow.

This production use of assertions on a real and critical phase of the chip design was highly successful. They were convinced that Zocalo’s approach to Assertion Based Verification is correct and intend to promote and expand the use of verification engineer provided assertions. Additionally, they want to explore using even more complex assertion (such as SystemVerilog function calls and return value support) that are an integral part of the Zazz capabilities.

Copyright© 2010 Zocalo Tech, Inc

CONCLUSION

The complexities of Assertion Based Verification relative to identifying assertion candidates, creating more powerful and useful complex properties and assertion management has limited ABV acceptance. Instead the prevalent approach is the use of simple SVAs provided on an ad hoc basis dependent on the skills and desire of the user.

In order to gain the benefits of ABV and wide scale acceptance:

- Automation is required for users.
- Metrics are required by the project for continued assessment of results in relation to using ABV.

Zocalo Tech provides Zazz, a set of capabilities and features, allowing a systematic approach to ABV making the net gains of using ABV readily apparent. All levels of assertions are useful. However, the ability to easily add SVAs early in the design and throughout the verification cycle that reflect temporal behavior and concurrency, is the most cost effective way to *independently* check the behavior of a design and track verification completeness.

Technology from Zocalo that enables cost effective and widespread use of complex SVAs will result in major breakthroughs reducing the time and cost of functional verification and IP integration.

ABOUT ZOCALO

Zocalo Tech, Inc. is a privately funded Austin, Texas EDA company, incorporated in 2006. Zocalo is the Spanish word referring to the *town center*. Zocalo Tech's goal is to be the *center* for wide scale acceptance of Assertion Based Verification.

The Zocalo product set and features, marketed under the name Zazz™, have been architected and developed from the ground up with one goal in mind – increased productivity for engineers adopting and utilizing ABV.

Zazz is a word from the urban dictionary meaning a quantifiable amount of something special. The more you have, the more awesome and successful things will be.

……………………..***put some ZAZZ in your verification environment……………….***

Copyright© 2010 Zocalo Tech, Inc