International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-4 Issue-6, November 2014

Implementation of Formal Verification on Scalable Arbiter

V. S. Jagannatha Rao, Siva Yellampalli

Abstract: In this paper, the Formal Verification (FV) approach is implemented on a scalable arbiter. Arbiters are a critical component in systems containing shared resources. FV is an approach using mathematical proof of ensuring that a design's implementation matches its specification, and utilizes formal analysis techniques targeted at assertions within the RTL, to find design errors. The FV requires, properties and coverage to be written and the same is required to be coded using system verilog assertions (SVA). The key advantage of FV is that it does not require test benches to run and can be used to verify RTL codes very early in the design process. The implementation requires checking RTL design of arbiter, clock initialization, implementation of assertions, proving properties, coverage and tabulating the results, to ensure successful implementation. The results are analyzed by running the incisive formal verifier, (ifv), tool and checking for the properties and coverage which are written in SVA, for pass or fail.

Keywords—Formal Verification FV,Time to market, system verilog assertions –SVA, Bug free silicon, resusbality.

I. INTRODUCTION

With the advent of technology the complexities of the SOC designs in terms of logic functionality is growing leaps and bound. This demands increased verification efforts and thus verification becomes one of the most resource consuming tasks. Ensuring functional correctness on register transfer logic (RTL) designs continues to pose one of the greatest challenges for today's ASIC and SOC design. Formal verification which has been existing from the past couple of decades is now maturing as a verification discipline within the industry. It is now evolving as a main stream methodology for use in industrial deigns, verification, and processes. It has become a specialised side activity with a narrow focus, to achieving a broad based usage as a core verification technology helping to significantly improve design,verification and productivity. With the use of formal verification method the technology has positively impacted as pre-silicon design quality and facilitated root causing of bug escapes to silicon. [1]. Research on formal hardware verification has made steady progress in developing methodologies and tools that try to cope with the growing complexities of systems. The task is to improve the practicality of current formal verification methods for complete state-of-the-art designs. [2]. FV is attracting increasing interest among verification engineers as a tool to deal with the ever increasing cost and complexity of hardware designs and protocols.

Manuscript published on 30 November 2014.
***Correspondence Author(s)**
V. S. Jagannatha Rao, Sr. Lecturer, Department of ECE, SJBIT, Bangalore Centre, Yeshwanthpur, Bangalore, India.
Dr. Siva Yellampalli, Professor, UTL Technologies Ltd, VTU Extension, Bangalore, India.

One application area that has been particularly promising is the use of automatic verification methods to debug device functioning.[3]. Modern verification methodologies for large, complex digital circuits, such as microprocessors, graphics chips and application specific integrated circuits (ASICs), commonly include both dynamic simulation and functional formal verification. Functional formal verification (FV) is usually a parallel verification effort that utilizes formal analysis techniques targeted at assertions within the RTL description. The goal of FV is to obtain proofs that assertions describing functionality are true to find design errors, manifested as counter examples to assertions.[4]. This paper discusses the issues involved with deploying formal verification on a scalable arbiter, and the strategies that may need to be adopted to make this deployment successful. A few common assertions are demonstrated, which are useful at identifying problems in the arbiter's implementation. The traditional technology is simulation with predefined test vector. A good overview on the basic concept and terms of formal and simulation verification is given in [5],[6] and [7] The author in [8] emphasizes that formal verification can help in resolving ambiguity issues in verifying complex hardware designs. For both verification methods the reuse ability is quite important for shortening time-to-market period [9].The integration of FV with simulation and its benefits and risks are covered in.[10]. Arbiters are a critical component in systems containing shared resources. For example, a system containing multiple processors that share a common memory bus would require an arbitration scheme to prevent multiple processors accessing the bus at the same time. There are a number of different arbitration schemes, such as the unfair priority scheme or the fair round-robin scheme, etc. In this paper, a round robin arbiter scheme is used and a few common assertions are demonstrated, which are useful at identifying problems in the arbiter's implementation. Formal analysis is the process of verifying functional (as opposed to electrical, timing,etc) assertions, or finding functional bugs in a design using formal analysis tools. Using sophisticated algorithms, formal analysis tools are able to completely check all possible operating states and all possible legal input sequences of a design. In doing so, they can conclusively verify if an assertion is true over all possible legal states. Formal analysis has gained attention recently for its potential to improve quality and productivity beyond what can be achieved through traditional verification approach, or through the use of simulation based ABV alone. The benefits of FV are:

- Quality is improved as bugs finding becomes easy.
- Productivity is improved.

- Bugs are isolated so that debugging and fixing problems is easier and the turnaround time is faster.
- Bugs are identified more easily and earlier in the design cycle.
- Finding a bug early involves fewer people.
- Improves pre-silicon design quality and facilitated root causing of bug escapes to silicon.
- This helps in eliminating bugs in the early stage of the RTL design, which helps in reducing time to market, with bug free silicon.
- It consumes less time and ensures the correctness of the design for early chip sign off, thus enhancing time to market also. [11]

In this paper the formal verification is implemented on an arbiter to scale it to a 4-bit arbiter, later, it is shown that, the same can be extended to N-bit arbiter. The verification is carried out using the Incisive™ formal analysis tool, (ifv) from Cadence is implemented. Formal Verifier performs assertion-based verification (ABV) of digital RTL designs written in Verilog and using formal analysis (FA) techniques on the arbiter to ensure that it is error free. ABV uses assertions to validate the functional behavior of the arbiter design. Assertions are unambiguous formalized statements about required or expected design behavior. The tool is run to analyze the properties and by default the tool can generate several reports, including the following:

- A summary report of all the properties that were checked.
- A detailed list of the properties that were analyzed, including any assumptions that were used to prove a property. We can filter this list based on status, such as whether the property passed or failed.
- Finally the results are analyzed. The functionalities covered are to be checked using the coverage model which automatically shows what are the functionalities covered.

Problem formulation lies in identifying a FV method with a mathematical approach to write properties and to perform assertion based verification on a scalable arbiter. This helps in eliminating bugs in the early stage of the RTL design, which helps in reducing time to market, with bug free silicon. The approach is a proposed solution which consists of writing codes for the properties using the RTL design of the scalable arbiter and to check the coverage for pass or fail by asserting the properties using the ifv tool. The functionalities to be covered are checked by using system verilog assertion (SVA) coverage block which shows, what are the functionalities covered. The tool is run to analyze the codes written for the properties and by default the tool can generate several reports, which are used for analysis and to arrive at the end result.

II. THE BLOCK DIAGRAM OF THE ARBITER

The proposed arbiter consists of two input round Robin arbiter and can be extended to 'N' number of inputs. Initially assertions are coded for a single arbiter node, which has two inputs and one grant output. The assertions which are coded for the arbiter node can be extended to 'N' number of arbiters. These assertions can generate formal stimulus for any number of inputs and outputs. The assertions should catch any error present while generating a grant. The process is asynchronous, i.e. the assertions are carried out irrespective of clock frequency. The assertions, along with formal stimulus, can also do self checking. The same assertions can also be used for dynamic simulation also. A few extra assertions can be coded to show coverage. The above coverage will show, the possible combination of arbiter inputs and outputs, multiple clock input to grant a request.

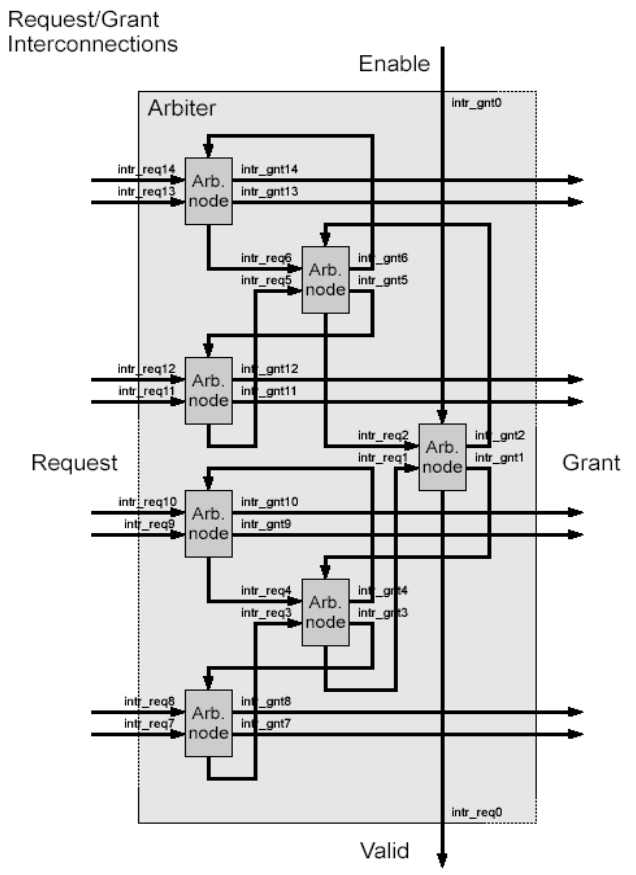


Figure 2.1: Request/grant interconnections of an arbiter

Two synchronous arbiter implementations are provided - 'arbiter' and 'arbiter_x2'. Both are round-robin arbiters with a configurable number of inputs. The algorithm used is recursive in that we can build a larger arbiter from a tree of smaller arbiters as shown in figure-1. The 'arbiter_x2' is a tree of 'arbiter' modules, 'arbiter' is a tree of 'arbiter_ node' modules, and 'arbiter_ node' is the primitive of the algorithm - a two input round-robin arbiter. Both 'arbiter' and 'arbiter_x2' can take multiple clocks to grant a request, and neither arbiter should assert an invalid grant while changing state. An 'arbiter' can take up, to three clocks to grant a req, and 'arbiter_x2' can have up to five clocks. 'arbiter_x2' is only necessary for configurations over a thousand inputs. Presently, the width of both 'arbiter' and 'arbiter_x2' must be power of two due to the way they instantiate a tree of sub-arbiters. Extra inputs can be assigned to zero, and extra outputs can be left disconnected. Parameters for 'arbiter' and 'arbiter_x2' are: Width: The 'width' is width of the 'req' and 'grant' ports, which must be a power of two.

The 'select_width' is the width of the 'select' port, which should be the log base two of 'width'. Ports for 'arbiter' and 'arbiter_x2' are : The 'enable' : which masks the 'grant' outputs. It is used to chain arbiters together, but it might be useful otherwise. It can be left disconnected if not needed. The 'req' : are the input lines asserted to request access to the arbitrated resource.The 'grants': are the output lines asserted to grant each requestor access to the arbitrated resource. The 'select': is a binary encoding of the bitwise 'grant' output. It is useful to control a mux that connects requestor outputs to the arbitrated resource, and is shown in figure-2. It can be left disconnected if not needed.

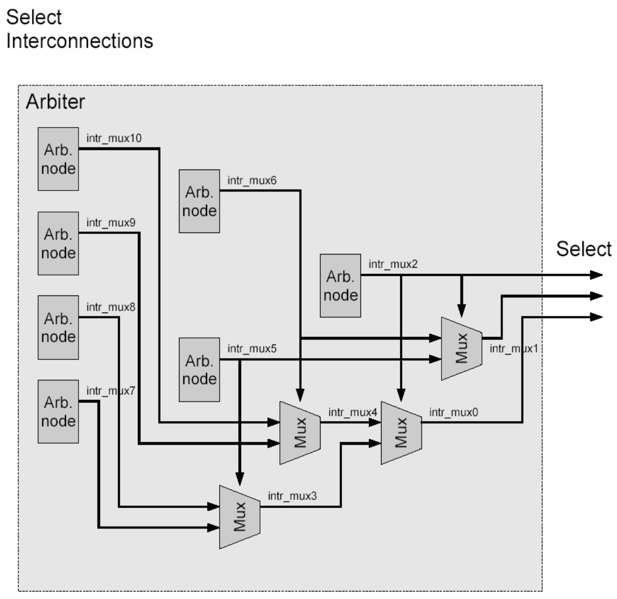


Figure 2.2: Arbiter select interconnections

The 'valid': is asserted when any 'req' is asserted. It is used to chain arbiters together, but it might be otherwise useful. It can be left disconnected if not needed and takes five clocks. 'arbiter_x2' is only: is a two-level tree of arbiters made from registered 'arbiter' modules. It allows a faster clock in large configurations by breaking the arbiter into two registered stages. The variables declaration such as width, grant, select as explained above, for the arbiter is implemented in the following code.

```
module arbiter #( request
        parameter width = 8,
        parameter select_width = 8)
```

A Developing and Using Properties

In formal analysis, properties are the basic unit of expression. Properties are formalized statements about the behaviour of signals over time. They are expressed in a property language, such as PSL, SVA.

Properties can be used to express the:

- Desired behaviour of a design under test,or
- Behaviour of the environment in which the design is embedded.

Properties that express the desired behaviour of a design under test are called assertions. Properties that express the behaviour of the environment are called constraints, because they constrain Formal Verifier to generate only those input sequences that satisfy the properties of the environment. Because of their importance, developing properties is central to using formal analysis. Developing properties is perhaps, one of the most interesting and challenging activities of formal analysis.[3]. Assertion - Formal Verifier can verify user-defined and automatically extracted properties in the design. Formal Verifier automatically extracts properties for doing dead code, pragma, FSM, Xcheck, bus, and range overflow checks on the design. To verify these automatically extracted properties, it is required to add them as assertions, because Formal Verifier verifies only the properties that are added as assertions. The assertion command allows to manage the properties present in the design. This command can be used to specify the properties that will be verified by Formal Verifier and view the verification status of properties in the design.

III. THE FOLLOWING ARE THE DESIGN FUNCTIONALITY REQUIREMENTS FOR IMPLEMENTATION

3.1 Checking RTL design.
3.2 Clocks initialization.
3.3 Tcl file initialization.
3.4 Implementation of assertions
3.5 Proving properties
3.6 Coverage and results

A. Checking RTL design

Formal Verifier does a series of RTL checks on the design and on the properties in the design. These checks enable to detect the problems related to multiple phases of design cycle, while the design is still under development at the RTL level. Such early warnings are a key to avoiding expensive design iterations and meeting quality. The ‘check’ command allows to customize the reports of the pre-defined property checks in the design. This command can be used to view details or summarized reports of the checks done by Formal Verifier on the design.

B. Clocks and Initialization

Once the basic scripts are in place and the block has been successfully read, the next step is to set up the clocking of the block. Cadence recommends Tcl-based initialization is the most common and easiest to use and is used here. To specify any number of inputs as clocks and provide arbitrary periodic waveforms for these clocks, the Tcl command is used. Along with clocking, it is important to set up the block's initialization and reset sequence. There are several techniques, which can be used to accomplish this. The most common and simplest is to use Tcl commands to toggle reset and run simulation for several cycles to propagate reset through the block, proper design initialization is critical in formal analysis. Without proper design initialization, verification can produce false assertion failures. Thus, before assertion verification, the design should be reset to a correct initial state.

C. TCL file initialization

Tcl-based initialization is perhaps the most convenient way to initialize a design. Tcl-based initialization uses an internal event simulator to run simulation on the design.

Tcl commands force reset at appropriate times during simulation. The state of the design after simulation is the desired initial state. This state is then used as the initial state during the verification of assertions. Those state elements (flip-flops and latches) that take the value X at the end of simulation are not initialized, and verification is done assuming all possible initial values for those state elements. Because assertions are checked under all possible initial values, the tool is able to provide a very strong check for initialization correctness. Uninitialized state elements are very common. Some common occurrences of these include address or data holding registers, simple delay registers, memories, and pipeline registers. State elements that should be initialized include state vectors for state machines, registers on handshake signals, and other control-oriented state elements. First time when the set up is initialized, visually the initial values are scanned and a report is generated to decide whether there are any incorrectly uninitialized state elements are present. Although these incorrectly uninitialized state elements would likely be caught by assertion failures later, it is much more time-consuming to identify an initialization problem with an assertion failure than it is to simply scan a list of initial values.

D. *Implementation of assertions*

An assertion is a statement about a design's intended behaviour, generally expressed in terms of properties, which must be verified and helps in error detection, error isolation, and error notification. Formal Verifier can verify user-defined and automatically extracted properties in the design. Formal To verify these automatically extracted properties, it is required to add them as assertions, and hence assertions are written for identified properties in the arbiter RTL design, since Formal Verifier verifies only the properties that are added as assertions. The assertion command allows to manage the properties present in the design. This command can be used to specify the properties that will be verified by Formal Verifier and view the verification status of properties in the design.

E. *Proving properties*

In formal analysis, properties are the basic unit of expression. Properties are formalized statements about the behaviour of signals over time. They are expressed in a variety of property language, such as PSL, SVA, or by a library of properties, such as OVL and IAL, and SVA is used here. Properties that express the desired behaviour of a design under test are called assertions. Properties that express the behaviour of the environment are called constraints, because they constrain Formal Verifier to generate only those input sequences that satisfy the properties of the environment. Because of their importance, developing properties is central to using formal analysis. Developing properties is perhaps, one of the most interesting and challenging activities during formal analysis. After coding the properties it is necessary to indicate which properties are assertions and needs to be to be verified and which properties are to be used as constraints during verification of the assertions. The Tcl commands contains this information.

F. *Coverage and results*

A number of coverage metrics have been developed to determine the effectiveness and quality of the verification process. We summarize several coverage techniques in the following sections. The coverage reports, the desirable behavior that must occur for the verification process to be complete. On the other hand the assertions monitor and report undesirable behavior. The RTL implementation-level functional coverage is covered in this project. The 'ifv' tool (incisive formal verifier tool) with SVA support is required with Tcl support.

IV. IMPLEMENTATION BLOCK DIAGRAM

The intended block diagram for implementation of FV is shown in figure-4.1 below, and each block is explained below. The scalable arbiter RTL code is the input to the ifv tool and is used.Unlike design code, an assertion statement does not contribute in any form to the element being designed.

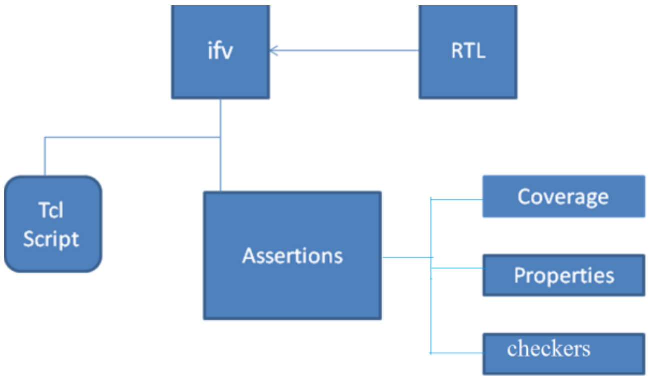


Figure – 4.1: Block diagram of a high level design

The implementation block diagram consists of:

- 4.1 RTL block
- 4.2 The ifv block
- 4.3 The tcl block:
- 4.4 Assertions block:
- 4.5 Property block
- 4.6 Coverage block

A. RTL block

Formal Verifier does a series of RTL checks on the design and on the properties in the design. The RTL block shown in the figure is verified first, for connectivity and coding. The check command allows to customize the reports of the pre-defined property checks in the design. This command summarized reports of the checks done by Formal Verifier on the design.

The syntax of the check command is:

% check -add | -delete | -modify | -show | -summary

B. 'ifv' Block

The Incisive Formal Verifier can be run by issuing one command, the ifv command. This command internally launches the parser, the elaborator, and the formal build to prepare the design, and then launches formal verifier to verify the design and display the output of the verification run.

This model also provides an easy migration path for existing NC and Dynamic ABV (Assertion-based Verification). The advantage of using this model is that no setup is required for the single-step method. All that is required is to launch the tool with its options on the source files. Running ifv automatically creates everything needed to run Formal Verifier, including all directories and libraries.
The ifv command has the following syntax:
ifv [ifv_options] <vhdl_design_files>
<verilog_design_files>+top+<worklib.entity:architecture>

C. TCL block

Formal Verifier's command language is based on Tcl. Formal Verifier is launched with the Tcl file,which contains all the necessary information. This verifies the design and generates a summary of the verification run. In Tcl, many commands have modifiers which tell the tool more about what the command should do.The contents of Tcl file are-start of verification,assertion,prove, assertion summery etc.If any component fails, the next component is not launched. All components share a common log file named formalverifier.log.

D. Assertions Block

Assertions are specified to detect invalid states and invalid transitions in a state machine. For example, in the case of a state machine with a one-hot structure, assertions monitor the state signals to ensure that no two states are ever active simultaneously. An example is shown below.

```
Property grant_is_onehot0;
   @(posedge clock) $onehot0(grant);
  endproperty
 a_grant_is_onehot0: assert property (grant_is_onehot0);
```

Assertions are specified to detect fairness problems within arbiters. While fairness and starvation are difficult verification areas, assertions help reduce the associated bugs. Assertions are used to handle the various corner cases involved with fairness on arbiters. The assertions includes the property, checkers and coverage blocks, and is explained below.

E. Property block

Formal property checking plays an important role in assertion based verification flow. The basic steps required for formal property is introduced. Steps required to perform formal property checking, for example, theorem proving, which include, compiling a formal model of the design creating a precise and unambiguous specification applying an automated and efficient proof algorithm Each of these steps are briefly discussed below. In the first step of the formal property checking process, a formal model of the design is created by compiling a synthesizable Verilog RTL hardware description into a form accepted by the property checker. For this purpose, hardware designs are finite state current systems. For example, the value of the current state of the system can be determined at a particular point in time by examining all state- elements of the system. The *next state* of the system can be computed as a function of the system's current state value and design input values. The following properties have been identified and implemented with assertions and coverage.

- property grant_is_onehot0 – (p1, assertion-a1)
- property grant_is_one_cycle– (p2, assertion-a2)
- property c_grant2– (p3, assertion-a3)
- property c_grant2a – (p4, assertion-a4)
- property c_grant3– (p5, assertion-a5)

An example of implementation of a property is shown below,
Property grant_is_onehot0:
The property declarations and subsequent assertions are carried out in the following code.

```
 @(posedge clock) $onehot0(grant);
   endproperty
   a_grant_is_onehot0: assert property (grant_is_onehot0);
```

In one-hot encoded style, states are defined using parameter declarations and their values are the bit position in the state register-which is a "one" when the FSM is in that particular state. The next-state computation makes the bit corresponding to the next state into a one using the abstract state names as index for the bit position. In this the grant signal is high for one cycle only. The 'posedge clock' declares that the implementation is done at positive edge of the clock. If the grant makes two bit toggling then it is treated as error. For this condition the finite state diagram (FSM) is shown in figure- 4.2, and the implementation is shown below.
The corresponding screen shot for this condition is shown in figure-4.3. The screen shots in figures 4.4 to 4.7 covers the remaining properties. Figure 4.8 shows the coverage property.

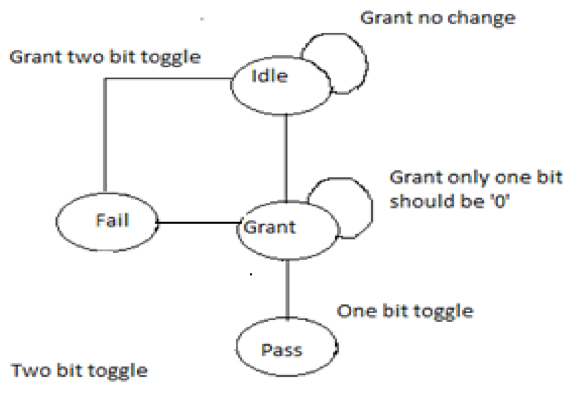


Figure 4.2: FSM of 'grant_is_onehot0

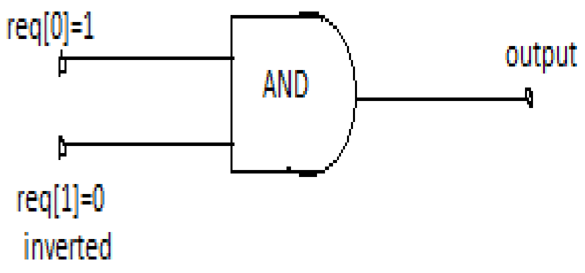
F. Coverage Block

Functional coverage provides the feedback needed to help determine the effectiveness of random environments and to help steer the configurations of these environments. Functional coverage increases observability. A functional coverage methodology provides an automated means for collecting and reporting functional coverage. The impact of changes in the testing strategy can be seen through functional coverage over time. A good functional coverage methodology combines higher level forms of functional coverage related to the specification with lower-level functional coverage for corner cases in the RTL implementation. This is done in coverage block. The coverage monitors and reports undesirable behaviour.

Functional coverage embedded in the RTL design produces an automated method of collecting functional coverage. Since the coverage points are associated with the model description, they are evaluated automatically. Functional coverage provides feedback on identified significant areas of a design. This feedback gives an indication of how effective the project's current set of tests are effective in exercising the features of a design.

Example : req[0] – should have both 0 and 1 and formal verifier needs to generate this condition.

Req[0] && Req[1]



Meaning, we want to cover Req[0], and Req[1]

V. TEST RESULTS

The screen shots in figures 4.4 to 4.7 covers the properties. Figure 4.8 shows the coverage property.

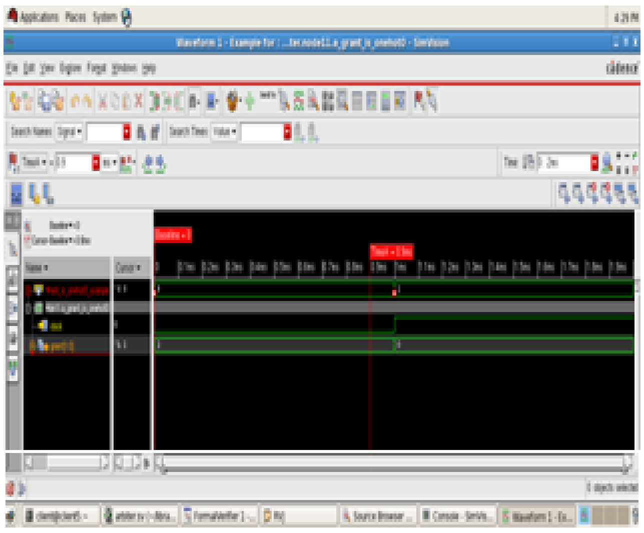


Figure 4.3: A_grant_is_onehot0.png

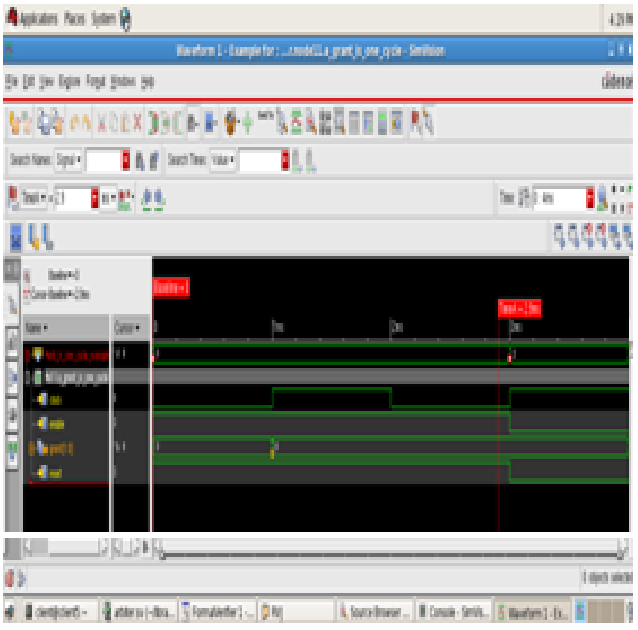


Figure 4.4: grant_is_one_cycle

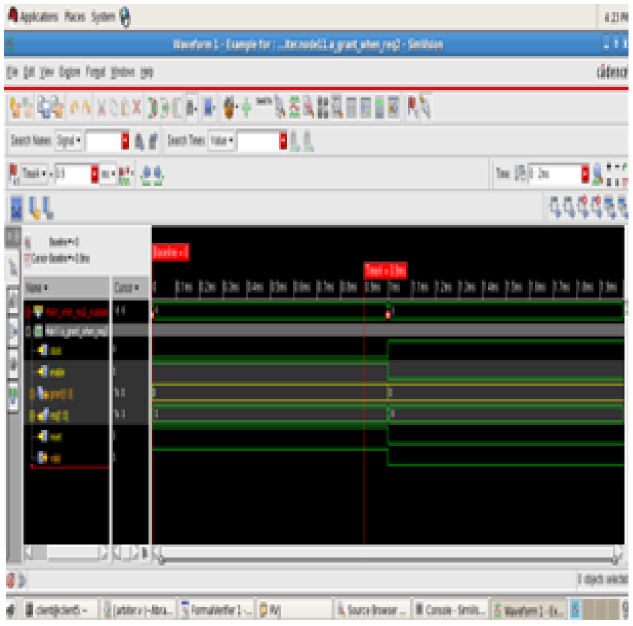


Figure 4.5: A_grant_when_req2.png

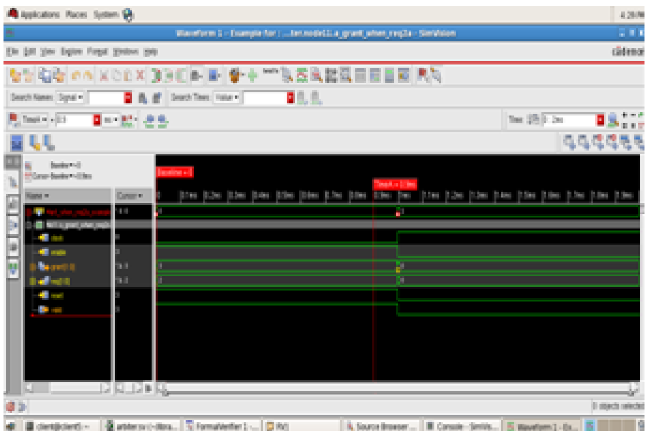


Figure 4.6: a_grant_when_req2a.png

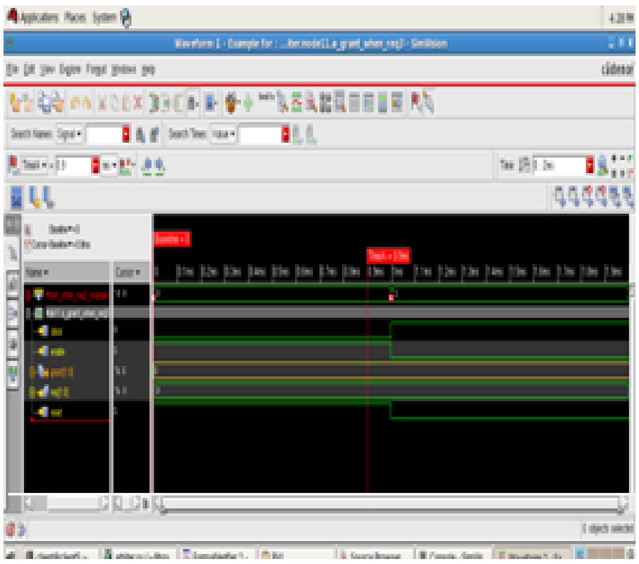


Figure 4.7: a_grant_when_req3. Png

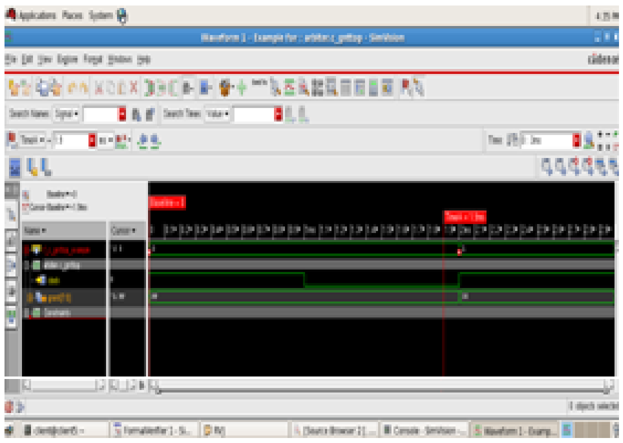


Figure4.8: cov_c_gnttop.png

All the above assertions are run using ‘ifv’ tool, guided by ‘Tcl’ script and the results are tabulated in Table-1, below:

A. Disussion on testing

The Table-1 summarizes the detailed tests carried out on the scalable arbiter. The properties p_1 to p_5 are implemented. The corresponding assertions a1 to a5, generated by the tool are checked with screen shots as shown previously. The coverage of all bits are checked using ‘coverage’ options for the final results and is tabulated in table 1.It is seen that the all the identified assertions for the corresponding properties are passing.

Table-1: Summary of the FV tests

Property	Assertions	Result	Propert-coverage
p1-a_grant_is_onehot0	a1	Passed	Covered
p2-grant_is_one_cycle	a2	Passed	Covered
p3-a_grant_when_req2	a3	Passed	Covered
p4-a_grant_when_req2a	a4	Passed	Covered
p5-a_grant_when_req3	a5	Passed	Covered

VI. CONCLUSION AND FUTURE WORK

The implementation of Formal Verification on a device like scalable arbiter is demonstrated in this paper. The properties and assertions are used to verify the arbiter on its RTL design, thus demonstrating the verification process being carried out at its inception, without the need for a detailed test vector as is done in simulation. This reduces the time, which might otherwise be spent on the detailed test using simulation methods, and serves as the initial checks on the design to verify its intended functionality quickly. This method will not only reduce overall design time but also identifies the bugs if any in the early stage. Formal verification is easy to use and provides significant increases in productivity and quality when used on RTL designs, which fit formal verification tool capacity. The future work includes implementation of formal verification on similar modules, in the areas of complex memory control, cache devices management, etc, to name a few. However, formal verification can be challenging when used on designs that exceed the capacity of the tools. In such cases, the techniques include divide- conquer, scale down the design, smart modelling of constraints, lightweight simulation, and use of advanced tool features. The collection of these techniques enables the full verification easy and saves many man-months of effort.

REFERENCES

1. Alok Sanghavi “What is formal verification?” Technical Marketing Manager Jasper Design Automation, eetasia.com | EE Times-Asia.
2. Nathaniel Ayewah, Nikhil Kikkeri and Peter-Michael Seidel, “Challenges in the Formal Verification of Complete State-of-the-Art Processors”,page no.603-606- IEEE -2005.
3. Alan J. Hu, Masahiro Fujita, Chris Wilson, “Formal Verification of the HAL Sl System, Cache Coherence Protocol,” page 43- IEEE – 1997.
4. C. Richard Ho, Michael Theobald, Martin M. Deneroff, Ron O. Dror, Joseph Gagliardo and David E. Shaw. “Early Formal Verification of Conditional Coverage Points to Identify Intrinsically Hard-to-Verify Logic”, Design Automation Conference, DAC 2008. 45th ACM/IEEE, dated – 8-13 June 2008,page no. 268-271.
5. 2004-2006 Cadence Design Systems, Inc. “Formal Analysis Project Methodology,Incisive Formal Verification,,Cadence Design Systems”,June -2005, page no.1-3.
6. Marjan Sirjani “Introduction to Formal Verification,” Various Contributors, Survey of Formal Verification, IEEE Spectrum , June 1996, pp. 61-67.
7. Kanna Shimizu and David L. Dill, Standford University, ,“Using Formal Specifications for Functional Validation of Hardware Designs“ IEEE, Design & Test of Computers, July/August 2002 (Vol. 19, No. 4), pp. 96-106.
8. ifvuser.pdf, Cadence – “Formal Verifier User Guide,” Product Version 10.2, September 2011.
9. John Lach, University of Virginia, Scott Bingham, University of Virginia, Carl Elks, University of Virginia, “Accessible Formal Verification for Safety-Critical Hardware Design,”, dated – 23-26 Jan. 2006, page no. 29-32.
10. Raj S. Mitra, “Strategies for Mainstream Usage of Formal Verification.” Texas Instruments, Bangalore.DAC 2008, June 9-13, 2008, Anaheim, California, page no.800-805.