Using Static Functional Verification in the Design of a Memory Controller

Winner of Best Paper award, Design Case category, System-on-Chip Conference

Mark Ross
Sachidanandan Sambandan

Cisco Systems

2000
System-on-Chip Design Conference

Using Static Functional Verification in the Design of a Memory Controller

Abstract

This paper presents a study of verifying a memory controller using a static functional verification tool. Static functional verification is a new technology that does not use vectors or dynamic simulation but analyses the behaviors of a design by the use of a property language. This paper presents the design and verification challenges of a controller, and how static verification was used to debug the design, what improvements were seen in methodology, and what was achieved and learned by using a static tool.

Authors/Speakers

Mark Ross

Current Activities

Director of Engineering at Cisco Systems, Inc. developing high-speed network switches. Email: mark@cisco.com

Background

Previously, he led Hardware Engineering at Granite Systems which was acquired by Cisco Systems, Inc. Prior to that, he was the platform architect for the first Ultra-SPARC workstations at Sun Microsystems, Inc. and was responsible for RISC workstation development at NeXT Computer, Inc.

Sachidanandan Sambandan

Current Activities

Director of ASIC Engineering, Force10Networks. Email: sachi@force10networks.com

Background

Until recently, he was Program Manager in the Gigabit Technology Group at Cisco Systems, Inc. Previously he was a Design Manager at Intel Corp. where he worked on the next generation Merced processor and developed a concurrent ASIC design methodology. Mr. Sambandan has been awarded 7 patents in the area of IC memories.

INTRODUCTION

Slide #1 Using Static Functional Verification

Using Static Functional Verification in the Design of a Memory Controller

DesignCon2000

Presented by

Mark Ross, Director of Engineering, Cisco Systems

1

Outline

- Design Overview & Challenges
- Verification Challenges
- Test Plan & Design Flow
- Features of Static Functional Verification (SFV)
- Achieving Functional Closure
- Verification Results using SFV
- Static Coverage Analysis (SCA)
- Conclusions and Summary

2

Slide #2 Overview

A shorter time-to-market and the drive to higher quality are pressures felt by design teams throughout the industry. The time to verify a design typically consumes over 50% of the total design time and effort. New verification technologies for ASIC design are now available as alternatives to vector simulation that has been in use for many years by design teams. This paper presents our experience using static functional verification and how it an alternative or replacement to vector simulation.

Slide #3 Design Overview

Design Overview

- Second generation design
- Four major modules were verified
  - Address decode
  - Memory core
  - Pipeline control
  - Register

3

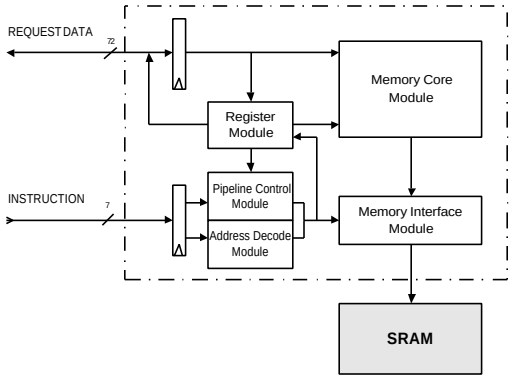
This memory controller (MC) design is used in new high-speed router products being developed at Cisco Systems. It is a second generation of an earlier design.

The basic architecture of the controller consists of the following major blocks:

- Address Decode Module
- Memory Core Module
- Pipeline Control Module
- Register Module

Slide #4 Memory Controller Block Diagram

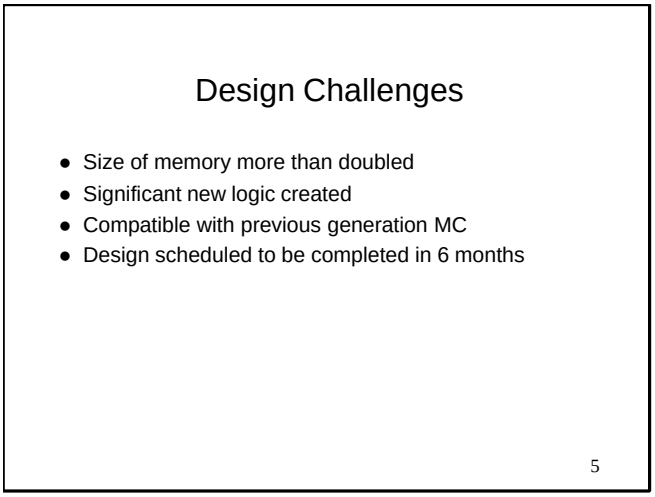
Memory Controller Block Diagram



4

REQUEST DATA is a bi-directional I/O bus that is used for 72-bit reads and writes to the memory. The MC returns Programmed I/O (PIO) read data to the host environment on this bus. It also provides a CPU interface to read/write of the MC's internal registers, and the SRAM. The commands are provided to the MC on the INSTRUCTION bus to perform reads and writes.

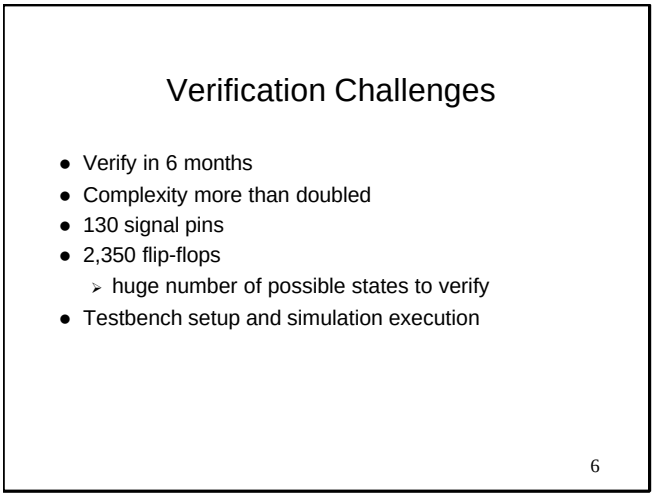
Slide #5 Design Challenges



This design presented new challenges for the design team. It had a number of new features compared to the previous generation of the MC. The size of the memory to be controlled was increased by more than 2X. Additional functionality required significant new logic to be created. The part also had to be compatible with the previous generation MC.

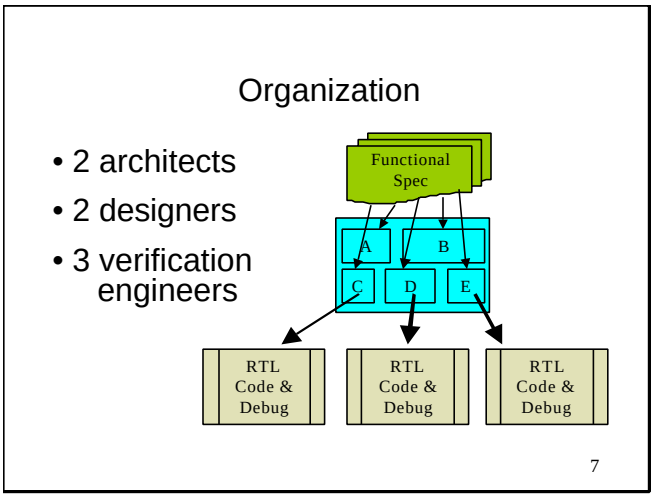
The time for creation, verification and tapeout of this design was scheduled to be 6 months. It was expected that a little over half the time to tape-out would be taken up with verification of the design.

Slide #6 Verification Challenges



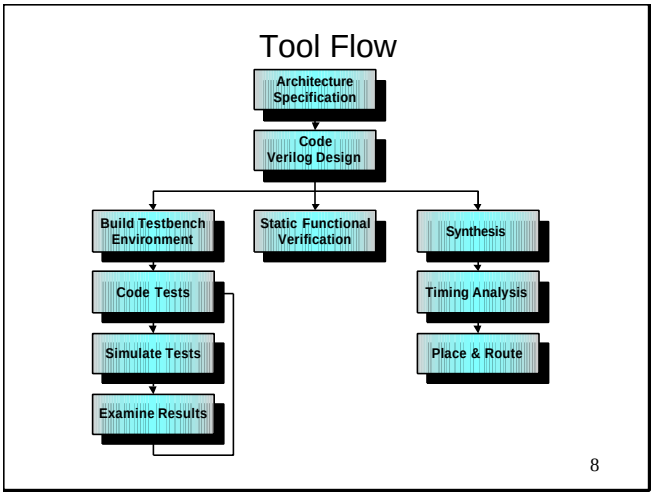
There are 130 signal pins, and over 2,350 flip-flops for this part and therefore a large number of states and modes needed to be verified. To verify correct operation of the various read and write operations would require a huge set of simulation vectors. The previous generation MC took 6 months to verify. The new design was two times more complex due to additional features. An alternative approach that did not require vector simulation was therefore attractive.

Slide #7 Organization



The engineering team included two architects (who are the authors of this paper), two designers and three verification engineers. One author began using the static verification tool to assist with block-level verification. As the design work was completed for each module, the designers began to do verification work as well. The members of team who wrote the specific module fixed any bugs that were found. It should be noted that functional simulation had already been done for several months before SFV was started.

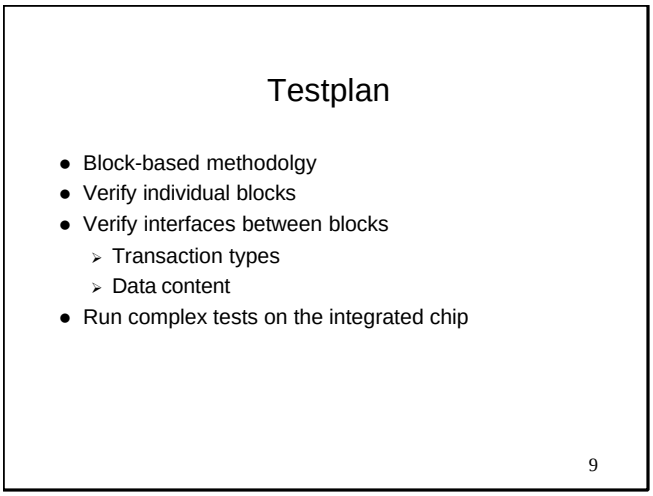
Slide #8 Tool Flow



The tool suite used was Solidify from HDAC, Inc. for static functional verification and Synopsys Design Compiler for synthesis of the design. Other tools used were VCS and

Virsim from Synopsys for simulation and Specman from Verisity for testbench development.

Slide #9 Testplan

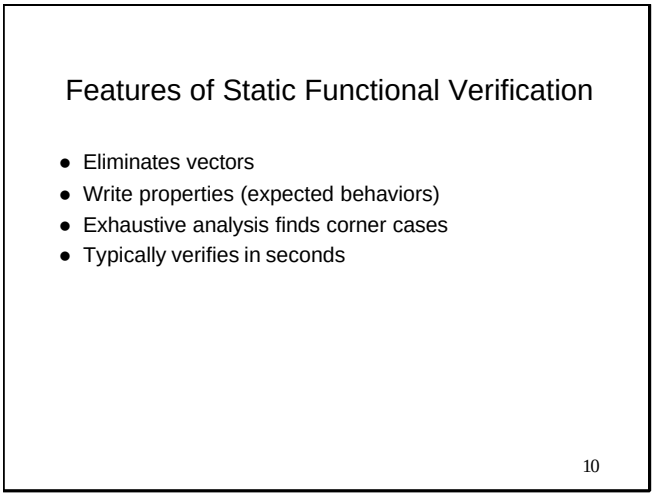


The testplan for the design included running the following tests:

- verifying the individual blocks
- verify the interfaces between the blocks are correct in terms of transaction types and in terms of data content
- run complex vector sets on the chip

Doing a good job at the start of testing means fewer bugs remain to be discovered at the full-chip level by the verification engineers. Every time a bug falls through to the next stage in the design cycle, the cost increases by approximately 10X to find and fix the problem. Using a static functional verification tool at the block level was ideal because of its exhaustive analysis and its ability to find bugs early in the design cycle.

Slide #10 Features of Static Functional Verification

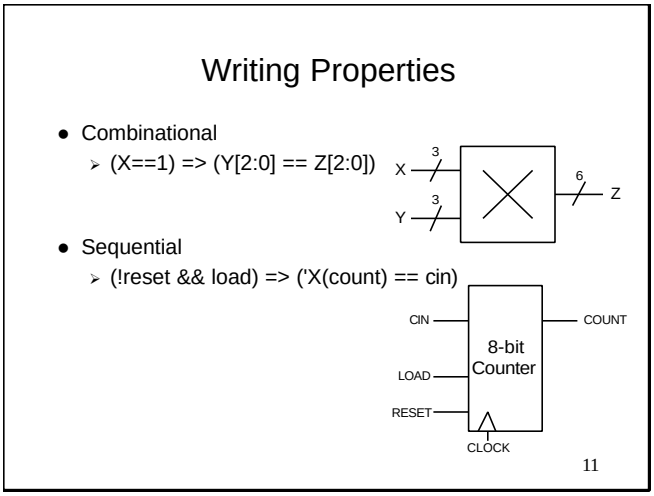


Static verification technology eliminates the need for vectors and provides an exhaustive analysis that is guaranteed to be 100% correct. Static functional verification verifies an RTL or gate-level description satisfies a set of properties or behaviors. The designer writes properties that are based on a functional specification for the particular block or module being verified, e.g., does my finite-state-machine (FSM) only transition to a legal state. The tool then verifies the property holds.

The properties or behaviors typically verify in a few seconds. This is contrast to vector based simulation which requires the setup of test-benches and possibly long analysis times. An additional benefit in not using vectors is much faster debugging compared to vector simulation.

Since static functional verification is an exhaustive analysis, corner cases and unusual operating modes are quickly exposed for review by the designer.

Slide #11 Writing properties



To use an SFV tool requires training in the writing and use of the property language. In this particular study the tool was learned over the course of a week using written tutorial materials. All in all, the learning curve was very low.

Properties can cover both combinational and sequential logic.

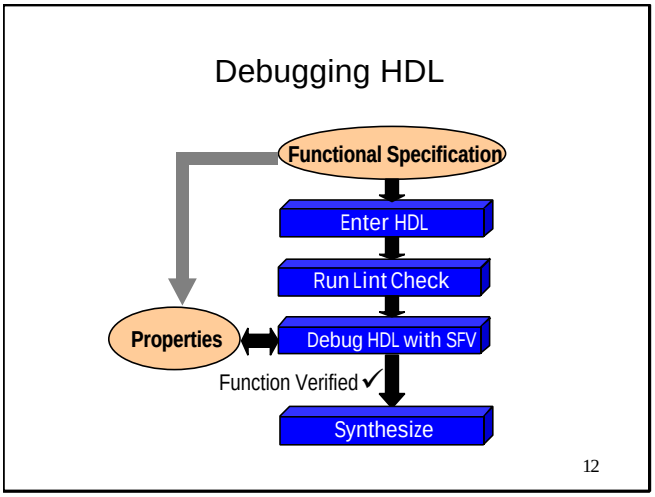
In the first example, consider a 3X3 multiplier circuit with inputs X and Y and output Z. The identity relationship can be written as a simple property: (X==1) => (Y[2:0] == Z[2:0]). This means “when X is 1 then Y equals Z.”

Similarly for sequential circuits, events in the future can also be verified. In the second example, consider an up-down counter with inputs `Reset`, `Load`, counter input `Cin`, and counter output `Count`.

The synchronous load is verified with the property: `(!Reset && Load) => ('X(Count) == Cin)`. It reads "when Reset is not asserted and Load is asserted then Count takes on the value of Cin in the next cycle."

The Synchronous reset is verified with the property `Reset => ('X(Count) == 0)`. It reads "if Reset is asserted then the value of Count in the next cycle is zero."

Slide #12 Debugging HDL

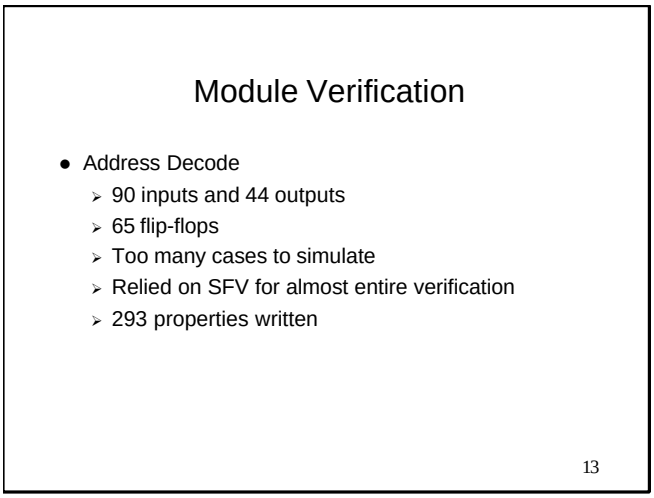


Properties are created using the functional specification for the chip. The behaviors for each block in the design are derived from the relevant sections in the specification. Properties can be analyzed individually or as a group. This allows for quick interactive development of the properties.

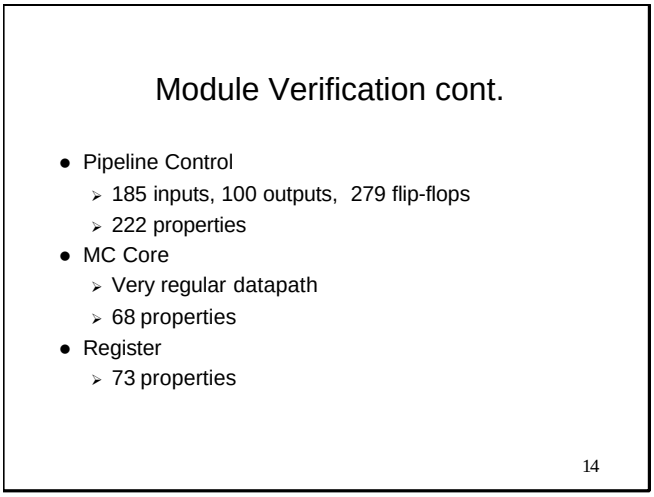
Typically, an initial attempt at a property would be analyzed and the tool would return an exception. If this was due to normal operation, then those particular conditions could be excluded from the property. For instance, the tool might report that the expected behavior might not occur because the reset signal was asserted. The property would be edited to remove the reset condition and then re-verified. If the exception was due to an error in the HDL, then the appropriate fix is performed.

Verification engineers were also working on the design using simulation to verify the design. This ensured that the risk in using a new tool was reduced, and also provided a measure for how fast bugs were found with the static tool.

Slide #13 Module Verification



Slide #14 Module Verification cont.



The functional specification for the MC is 58 pages long, with 11 timing diagrams. There were multiple modes that needed to be verified. The major focus of the static verification work was on the Pipeline Control and the Address Decode modules.

The Address Decode module is used for PIO read, PIO write, and SRAM operations with the host environment. It has over 90 inputs, 44 outputs, and 65 flip-flops. The verification of the Address Decode module was done almost entirely using the SFV tool, because of the large number of possible conditions that needed to be covered. An exhaustive analysis would have taken too long using a vector-based approach. However, some directed simulation tests were done as well to confirm basic operation. In total, 293 properties were written.

In the Pipeline Control module, all MC output signals to the host and SRAM are created. There are over 185 inputs, 100

outputs, and 279 flip-flops for this module. Altogether 222 properties were written.

The MC Core Module is a very regular datapath structure. This block already been analyzed extensively by using directed simulation tests, so fewer properties were written compared to the modules mentioned earlier. There were 68 properties written for this block.

The Register Module was also quite regular and only 73 properties were written for it.

Slide #15 Problems Leading to RTL Changes

Problems Leading to RTL Changes

- Over 20 bugs found
- Problems included
  - SRAM control signals
  - Misaligned register bits
  - Misalignment of addresses to the core
  - Incorrect aliasing
  - Compatibility with previous version of MC

15

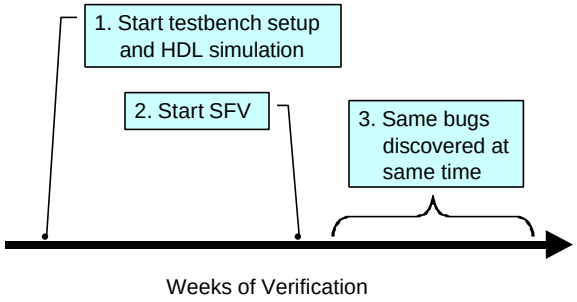
During the course of the testing, there was one show stopper problem identified that was not detected by simulation. The bug related to the control signals that are generated for the SRAMs. RTL changes were implemented to fix this issue.

There were other problems that were also detected and include problems with misalignment of register bits, and addresses to the core, incorrect aliasing, and compatibility with the previous generation of the MC.

It became clear that SFV could start finding bugs very quickly with a design.

Slide #16 Return on Effort

Return on Effort



16

Block level testing using SFV progressed in tandem with full chip verification using testbench simulation. In a number of cases, bugs that were identified through dynamic simulation were also identified using SFV. However, the simulation and testbench effort had started several months prior to the start of using SFV. We believe that if block level verification using SFV had commenced before the dynamic simulations were started, a significant amount of time would have been saved by flushing out most of the block level issues at the beginning.

Slide #17 Integration Testing

Integration Testing

- 10 weeks of verification
- Test suite
  - 45K cycles of random tests
  - 6K cycles of directed tests
  - 6 hours runtime (Sun Ultra 60)
- Several bugs found and fixed

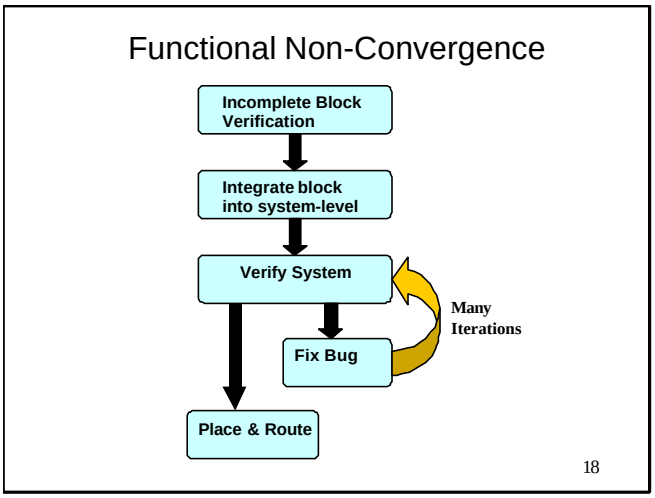
17

The testbench simulation environment was used for integration testing of the MC. The completed test suite took 6 hours to run on a Sun Ultra 60 workstation and consisted of:

- 45K cycles of random tests
- 6K cycles of directed tests

Approximately 10 weeks were spent to verify operation before tapeout. Several bugs, were discovered at integration testing and required fixes to the appropriate modules.

Slide #18 Functional Non-Convergence

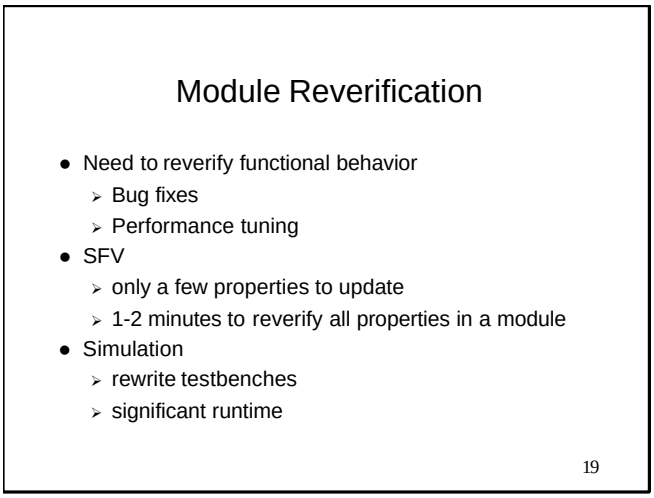


When the various blocks in a system are combined, the verification challenge becomes much greater since many more lines of RTL code need to be simulated. Bugs found at this stage are typically due to incomplete verification of the component blocks, incorrect interface specifications, and miscommunication between members of the design team. Once found, edits at the block level are used to fix the problem. However instead of updating and re-running the block-level testbenches, the patched block is re-inserted into the system for further debugging. Often the effort to create block level testbenches is thrown away, as maintaining block-level testbenches is not trivial. The danger in not re-running verification on the blocks is significant. Applying changes to code often can introduce additional bugs.

Once a change in the design has been introduced, chip or system level verification continues to find any remaining (or new) bugs. By avoiding complete verification at the block level their discovery is postponed until the more difficult and lengthy system simulation is attempted. Later discovery leads to more instability at the system-level and delays achieving confidence that most bugs have been found before committing the design to silicon. This functional non-convergence leads to a longer debug cycle, with less certainty that all design errors have been detected.

The key to achieving functional closure is to perform exhaustive block-level verification before integration, and re-verifying any block-level changes during integration-level testing.

Slide #19 Reverification



Since the design engineers typically do block level verification themselves, saving their time is critical since they also need to drive other issues such as timing closure for the design. When bugs are found, or there are changes in the RTL for performance or architectural reasons, a tool that verifies these changes quickly is very desirable.

Static functional verification demonstrated that the effort to re-verify was low, since only a few properties would be affected by a design change. This is in contrast to a simulation approach where vector testbenches would need to be re-written and re-simulated.

The verification time for each of the properties was typically only a second. This meant that reverifying a module would take about 1-2 minutes. Using exhaustive simulation to verify the modules would have been impossible given the number of inputs and outputs for each module.

Slide #20 Verification Times

Verification Times

Module	Time to Verify (sec)	Properties Written	Secs per Property
Pipeline Control	153	222	0.69
Register	38	73	0.52
MC Core	52	68	0.76
Address Decode	66	293	0.23
Total	**309**	**656**	**0.47**

20

All four blocks in the design were verified in 6 weeks.

How quickly were properties written? It is no surprise the simpler properties were easier to write. On the average about 20 properties a day were written. This was a part-time activity with typically no more than an hour spent each day writing properties.

The MC Core module was a very regular datapath structure and had already been verified using directed testing with simulation. As a consequence an abbreviated set of properties was written to verify that module.

As the table well shows, the verification of the individual properties is on average less than a second. This rapid verification of the behaviors for a design means a designer starts to find bugs and unexpected behaviors more quickly and with less effort.

Slide #21 HDL and Property Comparison

HDL and Property Comparison

Module	Lines of Code	No. of Gates	No. of Prop.	Code lines per Property
Pipeline Control	1,185	6.1K	222	5.3
Register	826	4.2K	73	11.3
MC Core	1,136	24.7K	68	16.7
Address Decode	503	5.0K	293	1.7
Total	**3,650**	**40K**	**656**	**5.6**

21

This table compares the number of lines of HDL, and the gates generated by synthesis, to the number of properties that were written. On average, for every 5 lines of HDL one property was written. In the case of the Address Decode module it was just under 2 lines of HDL per property.

We think the effort to write a property is roughly equivalent to the effort to create a monitor in a testbench environment.

Slide #22 Coverage Report

Coverage Report

Module	Lines of Code	No. of Signals	Percent Uncovered	Coverage Time (sec)
Pipeline Control	1,185	1,265	18.1	9
Register	826	1,857	31.0	2
MC Core	1,136	3,184	63.2	12
Address Decode	503	765	34.8	2
Total	**3,650**	**7,071**	**43.6%**	**25**

22

Coverage analysis determines if any changes in the RTL are caught by the set of properties for that module. It lists uncovered signals and reports what percentage is uncovered. The designer can then write further properties to cover those portions of the circuit. Since coverage is a separate analysis it does not impact the runtime for the verification of properties.

Coverage analysis was not available when this project was being done. In preparation for this paper, coverage analysis was applied to each module, to see how well each was covered by its set of properties.

The Pipeline Control module exhibits the highest level of coverage with only 18% of the signals involved not covered by a property. We are interested in seeing how code coverage will assist in writing properties for future projects. By using coverage analysis, it is very likely that more bugs would be discovered earlier.

Slide #23 Improvements in Methodology

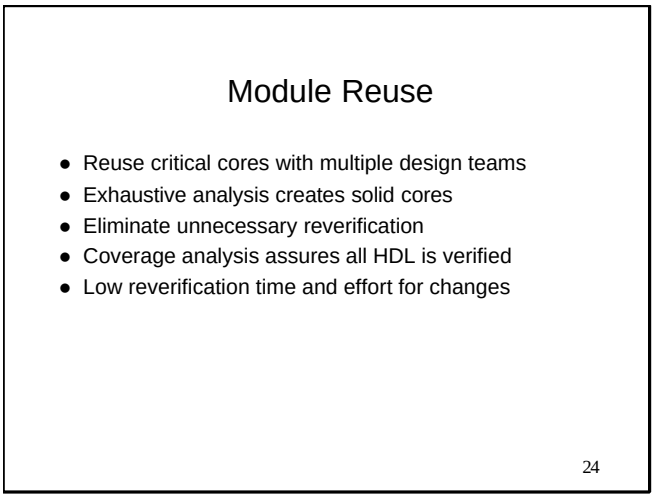
Improvements in Methodology

- SFV eliminates testbench setup cost
- Exhaustive analysis reveals corner cases
- Reverification is quick with low effort
- Fewer bugs expected at chip-integration
- Frees up simulation and testbench resources for use at the chip-level

23

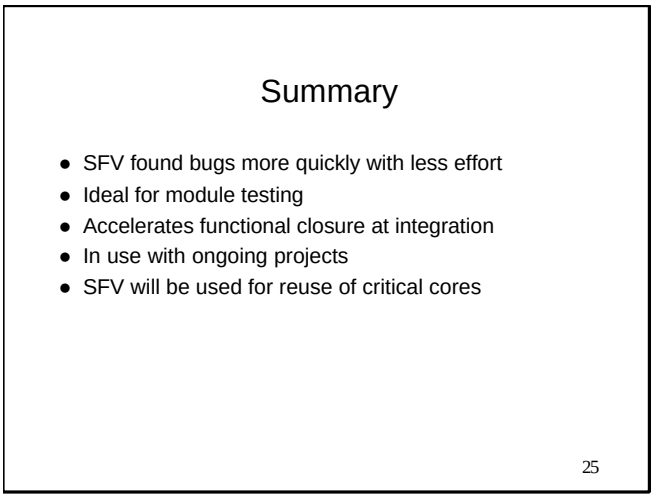
SFV eliminates the setup cost of a testbench environment, so engineers can very quickly start and make progress on verifying their blocks. SFV complements simulation at the system-level by providing a fast means of verifying functional behavior of blocks using an exhaustive analysis. This leads to fewer bugs at system integration. Re-verification of modules is much easier and faster so block level quality improves. Since there is less reliance on simulation & testbenches at the block level, these resources can be focussed on system simulation.

Slide #24 Module Reuse



One future activity that will be undertaken is using SFV on critical cores that will be shared with several design teams. These modules would only need to be verified exhaustively one time. Other design teams would typically not need to re-verify the module unless changes were made to the HDL code. As noted earlier, reverification of a module with SFV proceeds quickly with little rewrite of the properties. If a rewrite were necessary, the properties are typically orthogonal with respect to each other, so only a few of them would need to change. This lowers the effort to reverify. Coverage analysis will also be used to ensure that any changes in the RTL is covered by the property set.

Slide #25 Summary



The experience using static functional verification was very positive. SFV found bugs more quickly with less effort. SFV is ideal for module testing, and accelerates functional closure. SFV is being used with on-going projects and we are continuing to see more bugs being found with less effort than using traditional simulation.

Acknowledgements

We would like to thank Graham Bell for his assistance in preparing this paper.