Nuts and Bolts of Core and SoC Verification

Ken Albin
Motorola, Inc.
7700 W. Parmer Lane, MD PL31
Austin, TX 78729
1-512-996-6351
ken.albin@motorola.com

ABSTRACT

Digital design at Motorola is performed at design centers throughout the world, on projects with different design objectives, executed on different time scales, by different sized teams with different skill sets. This paper attempts to categorize these diverse efforts and identify common threads: what works, what the challenges are, and where we need to go.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids – *simulation, verification.* B.6.3 [**Logic Design**]: Design Aids – *simulation, verification.* B.7.2 [**Integrated Circuits**]: Design Aids – *simulation, verification.*

General Terms

Management, Documentation, Design, Economics, Verification.

Keywords

Verification, monitors, biased-random simulation, testbenches.

1. INTRODUCTION

Digital design at Motorola is performed at design centers throughout the world, on projects with different design objectives, executed on different time scales, by different sized teams with different skill sets. This paper first attempts to categorize these diverse efforts and then identifies some winning verification strategies:

- getting the right people working on the problem
- creating interface monitors, and reusing them at different levels of hierarchy
- the extensive use of constrained random stimulus
- automation to manage/navigate data
- appropriate use of abstraction

The underlying problem that verification engineers must solve is how to deal with the complexity of the design. We know of only two basic techniques for dealing with complexity: divide-and-conquer and abstraction. Most of the strategies discussed below support divide-and-conquer, while some offer opportunities for abstraction as well.

2. VERIFICATION CATEGORIES

Motorola design projects can be grouped into three general categories: functional block, integration, and processor design.

2.1 Functional Block Verification

The most common design projects at Motorola and throughout industry are relatively small units, designed and verified by a few engineers. Examples are programmable timers, UARTs, DMA blocks, interrupt controllers, etc. In some cases, these units are components of a processor design such as a cache controller, floating-point unit, bus interface, etc.

These projects employ classic unit-level verification, with much of the effort spent on creating custom testbenches and protocol or signal level stimulus.

2.2 Integration Verification

With millions of transistors available for designs, there is increasing emphasis on combining existing designs into System on a Chips (SoCs), resulting in a staggering amount of complexity. One recent SoC developed for the communication market has a RISC processor, a DSP, a custom data movement processor, embedded DRAM and more than 60 peripherals.

In order for an SoC to reach market in a reasonable time, a working assumption has to be made that the component designs have been individually verified at the unit level. Based on this divide-and-conquer assumption, the task of integration verification is limited to 1) checking the basic interconnect of the blocks and 2) checking for unexpected interactions between blocks.

Integration verification is enabled by the interface monitors created during unit verification. We currently use these monitors during simulation, but are working towards doing more static analysis using formal verification techniques [1]. Biased-random testing of the integration has also proven to be a powerful technique for discovering unexpected interactions which may take place outside of the obvious interface protocols.

2.3 Processor Verification

At the top of the verification food chain is processor verification. This is a special case of integration verification, distinguished by well-established interfaces and functionality, and substantial existing infrastructure such as compilers, simulators, operating systems, and available application code.

Because many users depend on the specified functionality and interfaces, because the designs are subjected to a wide variety of uses, and because of their complexity, processors require the highest level of investment in verification.

The strategy for processor verification is basically the same as in other integration verifications (thorough unit-level verification, with reuse of monitors in the integration) but additional investment is made in test suites, golden reference models in C or Verilog, and random instruction sequence generators. Existing instruction-level infrastructure is leveraged extensively in creating stimulus.

3. KEYS TO SUCCESS

3.1 Verification Teams

At the unit-level, designers typically verify their own blocks, possibly working with a verification engineer. At this level, the verification engineer provides another set of eyes on the design, simulation environment expertise, and coordination with verification engineers on other units on integration issues.

Processor design teams usually have dedicated verification teams, part of which are assigned to work with block designers while others perform processor-level verification or build and maintain the design/verification infrastructure.

The philosophy for selecting verification engineers varies from one design center to another. It used to be that the least experienced designer was assigned the task of verification, but improvements in CAD tools and SoC complexity has lead to verification dominating schedules and an increasing appreciation for skilled verification engineers.

3.2 Verification Engineer Skill Set

Although it is useful for design engineers to first work in verification (one Motorola design center only brings new designers in through verification), we have found that it is important to have a core verification team who have a fundamentally different skill set than that of designers:

3.1.1 the ability to shift levels of abstraction

While design engineers focus more on optimizing the implementation of their block, verification engineers must be more concerned with the proper behavior of the unit in the context of the integration. Tracking down errors often involves taking an integration level problem and mapping it down to a unit-level behavior.

For example, an error in a processor when the prefetch queue is full and an interrupt is received could be seen at the instruction set level but has to be mapped all the way down to the prefetch controller state machine's implementation before the cause (an incorrect transition) can be identified. Some people are much more adept at this kind of mapping than others.

3.1.2 a mix of hardware and software experience

A mix of hardware and software or mathematical experience seems to correlate with the ability to utilize different levels of abstraction, but verification engineers also must write scripts to analyze designs and automate tasks. Similarly, commercial verification environments require a fair amount of software sophistication to make use of their advanced features (object orientation, parallel programming constructs, etc.).

Finally, verification infrastructure is a significant part of a design project's IP creation, and good software development practices are a key to its successful reuse (e.g., modular design, configuration control, organized releases, etc.).

It is not clear why, but most of the candidates we find with this mixed hardware and software background started in hardware design and expanded out into software. It is more rare to find Computer Science graduates that feel comfortable working on hardware design projects.

3.1.3 the ability to organize large amounts of data

SoC designs are large and complex, and so is the associated verification data. Verification engineers must be able to filter the data (often by crafting special purpose filter scripts) and quickly sort out real problems from noise. They must also be able to switch from one context to another (between blocks, between design versions, etc.).

3.1.4 good or excellent communication skills

Many bugs are not due to deep technical challenges, but rather miscommunication. Verification engineers must be able to clearly describe abstract concepts, create concrete examples, and recognize ambiguity.

Because they must also work cooperatively with other design groups, verification engineers must be able to adapt to different terminology due to cultural, linguistic, and methodology differences.

3.1.5 internally motivated

Although verification is an increasingly important part of any design effort, it is unlikely to ever be considered glamorous. Part of this may be due to cultural/educational biases: university curriculum heavily biased towards design work; management with experience from simpler days when verification was a smaller effort. Another factor may be due to the difficulty in saying when the verification is done or measuring its quality - often the quality will not be evident until sometime much later.

Our most successful verification engineers are those that find personal satisfaction in improving the way things are done.

3.2 Interface Monitors

Figure 1 shows the basic components of a unit-level verification testbench.

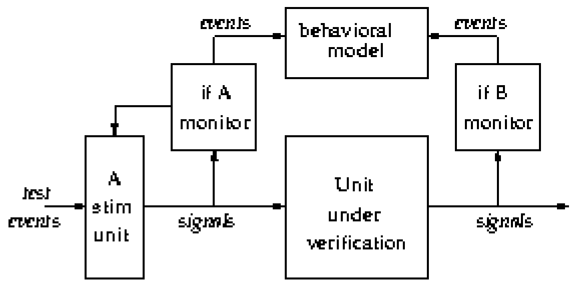

Figure 1. Partitioned unit verification environment.

Ideally, a unit will have a monitor on each interface that turns signal-level behavior into a stream of protocol events. Monitors do not reference internal signals and do not generate stimulus. These monitors are developed based on written specifications of the interfaces and are used by all blocks utilizing the interfaces.

The behavioral module takes one or more streams of interface events and performs some level of checking, ranging from comparing event counts on interfaces to duplicating the entire functionality of the block being verified.

In the unit-level environment, a stimulus block stands in for another design block in the integration. To ensure accuracy, the stimulus block is developed by or at least reviewed by the other block's design team. The stimulus source has traditionally been directed, but a constraint-based biased-random approach [2,3,4,5] has recently produced very impressive results.

There are several commercial tools which address this problem space effectively, but more important than a particular tool is the manner in which the tool is applied. Experience has shown that projects which have followed the strict testbench partitioning shown in Figure 1 have been more successful during integration than projects which were not so careful.

For example, the processor interface monitor of a cache controller may be easier to implement by snooping internal signals, but the monitor would only be usable when the cache controller is present. If the monitor was implemented looking only at interface signals, it could be shared with the processor unit verification team, uncovering protocol violations before integration.

Following the strict partitioning encourages well documented interfaces, facilitates communication between teams working on dependent units, and enables effective reuse of monitors and other infrastructure during integration.

3.3 Extensive use of Random Stimulus

Biased-random instruction sequences have been the backbone of processor verification for many years - utilizing existing infrastructure, biased-random sequences are generated and the design behavior compared with a suitably detailed reference model.

Recently, biased-random protocol-level or signal-level stimulus is seeing more use in unit-level verification. Several commercial tools provide some capability in this area. The engineering challenge is to economically provide a means of recognizing incorrect behavior.

A common approach to determining correctness is to check only a few key properties, but without some way of determining the completeness of these properties, there are likely to be gaps which allow errors to go unnoticed.

A current focus of our efforts in both unit and processor verification is to utilize abstract reference models to check specification-level behavior, combined with monitors to check interface protocols and low-level efficiency concerns such as dead bus cycles. Note that this is the same separation of concerns provided in the unit-level testbench shown in Figure 1.

3.4 Automation to Manage/Navigate Data

The design and verification of SoCs and Cores generates a large amount of data. Many CAD tools can now produce HTML reports and we have found that with a modest amount of effort these HTML pages can be organized and linked, providing an excellent way to track progress and facilitate communication among team members. Examples of data put on our internal website and automatically updated are: regression results, verification plan status, coverage (state machine, source, and functional), and equivalence checking results.

3.5 More Aggressive Abstraction

For some reason, engineers come naturally to the idea of divide-and-conquer, but seldom utilize our only other tool for dealing with complexity, abstraction. We currently do nearly all of our implementation and verification at a very low level. The design itself may often be conceptualized in the designer's mind as an algorithm or architecture, but current CAD tools require the manual mapping of these concepts onto a low-level implementation.

An example of how abstraction can be used much more extensively than just simulation reference models is shown in Figure 2. CAD tools are available which allow the translation of RTL down to mask data to be checked routinely and reliably, so we spend most of our verification effort trying to show that our RTL implements a top-level specification.

The layers shown in the diagram above RTL are an example of how a series of specific abstractions can be made to bridge the gap between implementation and specification. Formal

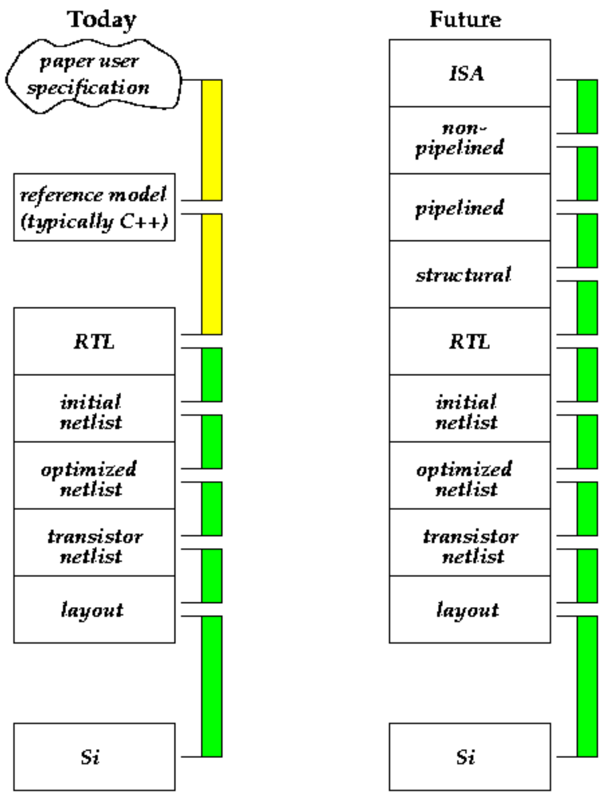

Figure 2. Layers of abstraction in the design flow.

verification of each of these abstractions has already been demonstrated [6,7,8,9] but mostly as research. These techniques are ready to be put into practice.

4. How do you know when you are done?

Many of the verification engineers I know are compulsively honest, so the question "are we done?" sometimes gets them in trouble.

The truth is that despite some impressive progress (e.g., [10]), for a design large enough for anyone to be interested in it, you cannot be truly done: the state space is way too large to exhaustively test it, and formally verifying that the entire design implements the specification is not generally possible today. This is not what project managers want to hear.

Regardless of these truths, designs need to tapeout and some criteria must be used. Some of the criteria used in Motorola are:

- 40 Billion random cycles without finding a bug
- directed tests in verification plan completed
- source and/or functional coverage goals met
- diminishing bug rate
- a certain date on the calendar reached

Most projects use some combination of these criteria, with project specifics determining the weighting. Until we can formally verify the design from the specification down we won't have a completely satisfactory answer.

5. CONCLUSIONS

The fundamental problem that SoC and Core verification efforts deal with is design complexity. We have found two basic tools to deal with complexity: divide-and-conquer and abstraction. Over time standard practices have been worked out that provide the mechanisms and communication necessary to apply divide-and-conquer effectively, but we have largely ignored abstraction. This paper identifies areas where we can begin leveraging abstraction.

6. ACKNOWLEDGMENTS

Special thanks to Carl Pixley for his comments on a draft of this paper.

7. REFERENCES

[1] M. Kaufmann, A. Martin, C. Pixley. Design Constraints in Symbolic Model Checking. CAV'98, pp. 477-487, June 28-July 2,1998

[2] Yuan, Shultz, Pixley, Miller, Aziz. Modeling Design Constraints and Biasing in Simulation Using , ICCAD. November 1999.

[3] Yuan, Shultz, Pixley, Miller, Aziz. Automatic Vector Generaion Using Constraints and Biasing. JETTA, 107-120. 16:1/2, 2000.

[4] Pixley. Integrating model checking into the semiconductor design flow. Computer Design's Electronic Systems journal, pp. 67-74, March 1999.

[5] Pixley, Shultz, Yuan. Integrated Formal and Informal Design Verification of Commercial Integrated Circuits. Proc. of the international Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp. 1061-1067. June 28, 1999.

[6] Burch, Dill. Automatic verification of pipelined microprocessor control. Computer-Aided Verfication (CAV '94), volume 818 of LNCS, pages 68-80. Springer-Velag 1994.

[7] Bryant, German, Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. Computer-Aided Verfication (CAV '99), volume 1633 of LNCS, pages 470-482. Springer-Velag 1999.

[8] Brock, Kaufmann, Moore. ACL2 theorems about commercial microprocessors. FMCAD '96, pp. 275-293. Springer-Velag, 1996.

[9] Srivas, Miller. Formal verification of an avionics microprocessor. Technical Report CSL-95-04, SRI International, 1995

[10] Aagaard, Jones, Melham,O'Leary, Seger. A Methodology for Large-Scale Hardware Verification. FMCAD 2000, pp.263-282, November 2000.