Verification of Configurable Processor Cores

Marinés Puig-Medina
Tensilica, Inc.
3255-6 Scott Boulevard
Santa Clara, CA 95054-3013
mari@tensilica.com

Gülbin Ezer
Tensilica, Inc.
3255-6 Scott Boulevard
Santa Clara, CA 95054-3013
ezer@tensilica.com

Pavlos Konas
Tensilica, Inc.
3255-6 Scott Boulevard
Santa Clara, CA 95054-3013
konas@tensilica.com

ABSTRACT

This paper presents a verification methodology for configurable processor cores. The simulation-based approach uses directed diagnostics and pseudo-random program generators both of which are tailored to specific processor instances. A configurable and extensible test-bench serves as the framework for the verification process and offers components necessary for the complete SOC verification. Coverage analysis provides an evaluation of how well a specific design has been exercised, of the breadth of the configuration space explored, and suggests improvements to the process. The results of the analysis show that our methodology achieves good verification coverage of the processor implementation.

Keywords

Design Verification, Configurable Processor Cores, Test Generation, Co-simulation, Coverage Analysis, System-On-Chip

1. INTRODUCTION

The increased complexity of System-On-Chip (SOC) designs, combined with shorter time-to-market windows, have forced designers to focus on the application specific parts of the system while importing and integrating other pieces of the SOC. One component imported in an SOC design is a processor core that controls the operation of the system. The processor core should contain only the necessary functionality for the given application so that it consumes little power, it occupies a very small area, while still providing high performance. In order to address those requirements, Tensilica has designed a highly integrated, high performance, configurable processor core. The system designer includes into the configured processor only those features required by the application, and extends the processor's functionality by defining and incorporating new instructions without directly modifying the processor HDL.

Designing a system, however, is only a small part of the development cycle. The major bottleneck in an SOC development is the verification of the individual components and of the integrated system as a single entity [4]. Hence, designers of processor cores aimed at SOC designs need a methodology that provides a timely and comprehensive verification of the processor cores while facilitating the system level verification process.

Traditionally, the functional verification of a processor relies on extensive simulation-based testing of the processor's RTL model using a collection of directed and random diagnostic programs [2]. Coverage analysis provides an evaluation of how well the design has been exercised by those programs and assesses the need for further testing [10]. Verification of the Tensilica processors, however, is complicated by their configurability and designer-defined extensibility. These two features create a vast number of different processor instances that need to be verified.

We have developed a robust and flexible methodology for the verification of configurable and designer-extensible processor cores. Our simulation-based approach employs fully configurable directed diagnostics for the architectural and micro-architectural testing of the processor. Two configurable random program generators further enhance the testing. We use coverage analysis to assess how well a single processor instance has been verified and to evaluate the breadth of the configuration space explored. Our test-bench is tailored to specific processor instances, and it provides components useful in the verification of a complete SOC.

The rest of this paper is organized as follows: in section 2 we provide an overview of the Xtensa processor. In section 3 we outline our verification methodology, discuss the generation of diagnostics, and examine the co-simulation process. In section 4 we discuss the test-bench used in our verification environment. In section 5 we address the issue of coverage analysis and the extensions required by configurability. Finally, section 6 summarizes our approach to the verification of configurable and designer-extensible processor cores.

2. XTENSA: A CONFIGURABLE AND EXTENSIBLE PROCESSOR

Xtensa is a new instruction set architecture (ISA) designed to enable configurability, minimize code size, reduce power dissipation, and maximize performance [5]. Xtensa is defined in two parts. The base ISA provides a rich set of instructions guaranteed to exist in all Xtensa implementations. These instructions implement powerful operations such as single cycle compare-and-branch and zero overhead loops. In addition, a large set of configurable options is available to the system designer so that she can customize the processor to a given application.

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2000, Los Angeles, California
(c) 2000 ACM 1-58113-188-7/00/0006..$5.00

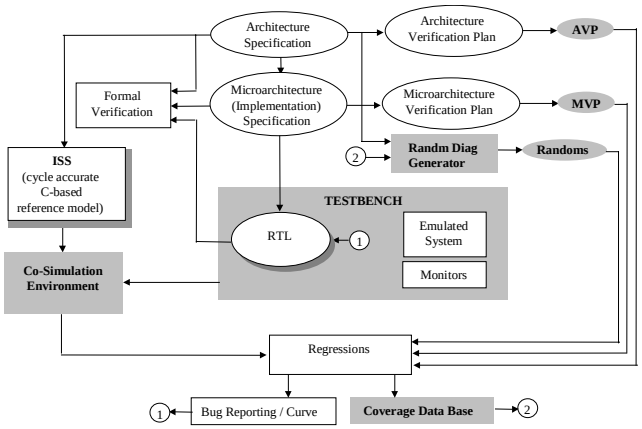

Figure 1: Verification of configurable cores

Xtensa supports configurability in three forms. The simplest form is the inclusion or omission of architectural features such as interrupts, DSP functionality, and JTAG-based on-chip-debug support. A second form of configurability is provided through the numerical scaling of features such as the number and type of interrupts, the size and associativity of caches, and the number of registers. The last and most complex form of configurability involves designer-defined instruction extensions. If the features already available in the processor do not provide the desired functionality, the designer can define new application-specific processor states and instructions in order to speed up key parts of the application and improve the system performance and efficiency. These new states and instructions are defined using the TIE language [5], and they are integrated into the processor RTL and the software tool-chain by the TIE compiler. TIE offers significant advantages to the system designer including ease of expression, seamless integration with the processor, portability across processor implementations, and a significant improvement in application performance.

The configuration of the processor and the generation of both hardware and software are performed through a web-browser. The designer selects and sizes the required features and provides any TIE-based extensions that she wants to integrate into the processor. The Xtensa processor generator uses the provided information to produce customized versions of the processor RTL, the verification suite, and the system test-bench. In addition, the generator creates a customized GNU-based software environment that includes a C/C++ compiler, an assembler, a linker, a debugger, a code profiler, and an instruction set simulator (ISS), all extended with any application specific functionality. A discussion of Xtensa and its properties can be found in [5].

3. A FUNCTIONAL VERIFICATION METHODOLOGY

Functional verification aims at isolating design and implementation flaws so that the released processor design is fully functional; that is, it exhibits the behavior specified by the ISA when it executes a sequence of instructions [6]. However, verification of the Xtensa processor is complicated by two important factors.

First, the configurability and designer-defined extensibility provided by Xtensa translate into a vast number of distinct processor instances that include designer-defined functionality. A traditional

```
TimerCount = numberOfTimers(); // how many timers
timers = timersAttributes();   // read timer attributes
for each timer {
    startTimer(timer);         // start counting
    // Enable the interrupt associated with the i-th timer
    writeInterruptEnableBit(timer→interruptNumber());
    emitInstruction(waiti);    // wait timer to fire
    checkTrapNum(timer);       // check the result
 }
exit();
```

Figure 2: A diagnostic configuration example

verification environment, in which the methodology and the tools are specifically designed for a single implementation of the processor, is too rigid to address the testing requirements of a customizable design. Using such an environment would have required us to develop a prohibitively large number of tools each targeted to a different processor instance. Instead, we have developed an environment that is automatically tailored to a particular processor instance and whose components are independently configured based on the characteristics of the design.

Second, the Xtensa core has been designed so that it can be easily integrated into an SOC. Hence, the processor implementation has to be verified within the context of the complete system. Traditional processor verification environments cannot be easily merged into an SOC verification environment. In contrast, the verification environment for the Xtensa processor has been designed so that its components can be easily integrated into an SOC verification framework.

Figure 1 outlines our verification methodology. The important difference with traditional verification environments is that each component in the environment is completely configurable and it is automatically customized for the processor instance under testing. We implemented the environment using both internally-developed and industry standard tools such as Verilog and VHDL simulators, formal verification tools, and verification languages. The rest of this paper addresses the parts of the environment appearing in shaded boxes. The following two subsections present in more detail two of the more interesting parts of our approach: the test program generation methodology and the co-simulation environment.

3.1 Test program generation

Our verification methodology utilizes several types of test programs to ensure that the design is well exercised. We employ architecture verification programs (AVPs) to test the execution of each instruction in the ISA, and micro-architecture verification programs (MVPs) to test features of the Xtensa implementation. In contrast to traditional AVP and MVP diagnostics, our programs are not constant sequences of instructions used with every new processor design and processor instance. Instead, they are Perl scripts that utilize the configuration information in order to generate high quality tests tailored to a specific processor instance.

The configuration information is used in two ways. First, the script that generates the diagnostic uses the configuration information to better target the resulting test to the processor instance under testing. Depending on the processor options selected by the designer, the generator includes into the diagnostic, or omits from it, sections

```
$install =      $debug→isConfigured() &&
                $timers→areConfigured() &&
                $interrupts→areConfigured();
$Interrupts = 'no';
$RandomBusError = 'no';
```

Figure 3: Database entry for *waiti* diag generation

of the code, and it changes the test parameters within the code. The example shown in Figure 2 illustrates the configuration of a diagnostic based on the availability of the interrupt and timer options. The diagnostic generator computes the number of timers and their associated interrupt levels, and creates a loop that checks whether each timer generates the correct interrupt level when it goes off.

Second, each script is accompanied by a database entry that indicates under which conditions the produced diagnostic should be installed and used (e.g., for what configurations is it valid, which runtime options are required, which runtime options prohibit its use, etc). An example database entry is shown in Figure 3. The entry indicates that the corresponding diagnostic requires the presence of timer counters, interrupts, and of the debug option (which enables instruction-counting and breakpoint exceptions), and that it should only be installed if all three options are available. Furthermore, the database entry indicates that the run-time options of random external interrupts and bus errors would conflict with the diagnostic's operation and, thus, they should not be enabled.

In addition to the directed diagnostics, we also employ random program generators to further enhance the testing of the processor implementation. Random program generators aim at creating tests that would trigger complex interactions across different parts of a processor [1, 7, 10]. We have implemented two configurable pseudo-random diagnostic generators as part of our verification methodology. The generators are automatically customized for each processor instance based on the configuration options selected by the designer. One of the generators, RTPG, was developed using object-oriented Perl, whereas the second one, VSG, is based on the Vera Stream Generator provided with the Vera verification language [9]. Both generators employ instruction list templates that are combined in random sequences to create interesting diagnostic programs [1, 6]. Furthermore, both generators provide a large number of tunable parameters that allow a verification engineer to target a configured random program generator towards complex parts of the processor implementation (e.g., branch unit, load-store unit, etc), as well as towards complex interactions across the entire processor (e.g., branch instructions interleaved with external interrupts and exceptions).

One of the major weaknesses of generating and using random diagnostics in isolation is the potential that the programs will repeatedly cover the same part of the design space and, thus, waste precious simulation cycles, while other parts of the design remain completely uncovered or partially exercised [8]. To avoid such a situation we use the results of the coverage analysis (discussed in section 5) to steer the generators into new directions, hence improving the testing efficiency and verification completeness. Currently, the feedback of the coverage analysis results into the random generators is manual. However, in the future we may implement the infrastructure necessary for the random program generators to automatically use the coverage analysis results in the generation of diagnostics.

3.2 Co-simulation (Cosim)

Co-simulation is the process of running the RTL and the ISS simulators in parallel, and comparing the architecturally visible states at the boundaries defined by instruction retirement and by external event occurrences. We have implemented the comparison process (Cosim) in Vera-VHL [9]. Vera was chosen because it is easy to use, it is portable across HDLs and simulators, and it provides the appropriate abstraction level for developing sophisticated test environments. Cosim acts as the synchronizer and the gateway between the RTL simulator and the ISS, and it also includes monitor and checker tasks that are executed in parallel. A diagnostic program executing under Cosim fails as soon as a mismatch occurs between the RTL and the ISS or when an assertion checker detects a failure. The assertion checkers are Vera tasks that continuously monitor a set of RTL signals or conditions (sequence of events) and terminate the simulation when they detect a violation of the design rules.

There are three major advantages of co-simulation over self-checking diagnostics. First, it allows fine-grain checking through observability of the processor state during the program simulation. In contrast, a self-checking diagnostic can only compare a limited amount of state to pre-computed results and may not detect cases that create erroneous intermediate results while producing a correct final result. Second, constructing a comprehensive self-checking diagnostic is considerably more time consuming and harder to automate, whereas the Cosim mechanism is well suited to random and directed diagnostic generation schemes. Finally, Cosim stops the simulation at, or near, the cycle where the problem appears, which significantly reduces debugging time and effort. The main advantage of self-checking diagnostics over Cosim is their ability to detect problems that exist in both the RTL and the ISS models. However, this advantage can be minimized by thoroughly verifying the ISS model through the execution of extensive test suites and application code before using it as the golden model in the verification of a processor implementation.

One of the biggest challenges of any co-simulation methodology is finding the appropriate synchronization points between models at different levels of abstraction as they respond to asynchronous events. We have designed into Cosim the mechanisms necessary to resolve the ambiguities between the RTL and the ISS models. When an asynchronous event, such as an external interrupt, occurs, Cosim communicates the required state information from the RTL model (which is more detailed) to the ISS model (which is more abstract). In the Xtensa implementation the interrupt latency depends on the non-architectural state of the RTL and cannot be reproduced by ISS since it does not model the Xtensa pipeline. Thus, when an interrupt occurs, Cosim monitors the exact boundary at which it is acknowledged by the RTL model and creates a synchronization boundary between the RTL and ISS. Then, it presents the interrupt information to ISS and compares the resulting state changes due to this event. From this point onwards, the comparison proceeds once again at the instruction retirement boundary. The exchange of information between the two models reduces their independence and could potentially mask off some bugs. To address this issue, we have minimized the information transfered from the RTL model to ISS, and we have augmented the asynchronous event generator programs with monitoring capabilities. For example, when the test program issues multiple interrupts to the processor, the only piece of information conveyed from the RTL to ISS is the instruction boundary at which the interrupt needs to be acknowledged; the type, number, and priority of the actual interrupt is determined independently by each model. At the same time, the test program

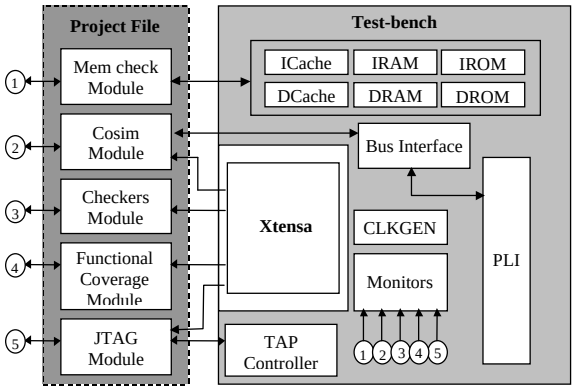

Figure 4: A configurable test-bench

keeps track of the interrupts it has issued and verifies that all of them are acknowledged.

Another co-simulation challenge involves masking off comparisons when the processor state is architecturally undefined. For example, upon reset most of the Xtensa registers are undefined. HDL languages model this value as "x." ISS, on the other hand, uses a sequence of 0's and 1's to represent the corresponding state. Cosim resolves this ambiguity by tagging each register with a valid bit and turning off comparisons until the first change in the register value occurs either in the ISS or in the RTL model.

4. A TEST-BENCH FOR SOC VERIFICATION

In order to verify a configured Xtensa processor we build a system emulation test-bench around the processor (Figure 4). We have developed our test-bench using the Vera *project* mechanism. This mechanism enables individual verification programs (called *modules*) to be independently developed and compiled, and to be dynamically linked into the verification test-bench. The use of modules provides a flexible and extensible environment for verification, and facilitates the integration of our test-bench into an SOC verification environment.

Each module is described by separately defining its interface and its connection to the simulation. The interface definition specifies which signals will be observed and controlled, it maps those signals from the RTL domain into the Vera domain, and it determines the flow of data between the RTL and Vera. The connectivity information, on the other hand, describes the complete HDL hierarchy path for each signal in a module's interface. Because our test-bench is described completely in terms of modules, integrating it into an SOC verification test-bench is as simple as modifying the modules' connectivity information to reflect the new design hierarchy; it is not necessary to recompile the source code and a runtime Vera license is not required.

Our test-bench also contains models of components that enable the simulation of a wide variety of diagnostic programs. The components modeled in our test-bench include caches, processor RAMs and ROMs, a TAP controller for JTAG, and a bus interface that connects the processor to system resources. In addition, we have implemented interrupt and bus error generators, peripherals, and system memory through PLI calls. All the test-bench components, except the processor, are behavioral models implemented at various abstraction levels including RTL, C, and Vera, and they are automatically adapted to the configuration under testing.

Vera-based Coverage				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Exception	100	40	40	100
Interrupt	100	75	79	100
Bubble	100	88	100	100
Bypass	100	100	100	100
MiscEvents	100	14	85	100
FSM				100

ISS Coverage				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Instructions	100	95	88	100
Cache assoc.	N/A	N/A	N/A	N/A
Branch align.	94	48	80	100
Window overflow	95	14	0	100
Hazards	100	15	10	100

HDLScore				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Expression				95

Table 1: Coverage of *proc1* configuration

5. COVERAGE

Our methodology uses a variety of tools and metrics to address coverage from different perspectives. We employ ISS monitors (written in Perl) that check the architectural level coverage by tracking all the configured instructions and architectural features as specified by the ISA. In addition, we use Vera monitors to track the RTL state and to evaluate the correct implementation of micro-architectural features by determining the coverage of fine-grain cycle-dependent events. Furthermore, we use HDLScore, a program-based coverage tool [3], to check how well the RTL model has been exercised. HDLScore gathers and reports several metrics including block, path, and expression coverage. Finally, we use Vera FSM monitors to measure coverage of the finite state machines (FSM) in the design by checking the states of each FSM that were visited and the transitions between states that were followed during the execution of the diagnostics. This approach has the added advantage that assertion checkers, introduced as part of the coverage analysis methodology, can halt the simulation if any illegal state is reached or an illegal transition is followed.

The cumulative reports of each coverage tool for three Xtensa configurations are presented in Tables 1, 2, and 3. These reports are representative of data collected on numerous configurations that are generated and regressed daily. *Proc1* represents a configuration where only part of the available options were included, *proc2* represents a maximum configuration where all available options were included, and *proc3* represents a randomly generated configuration.

The tables present the results of five different types of Vera coverage monitors. The "exception" monitor covers the instructions that cause exceptions and the special instructions used to return from exception handler routines. The "interrupt" monitor tracks external interrupts and their interactions with different processor states. The "bubble" monitor covers the instruction issue and the hazard detection logic. The "bypass" monitor covers operand bypassing

Vera-based Coverage				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Exception	100	20	46	100
Interrupt	73	73	83	98
Bubble	80	80	90	100
Bypass	100	100	100	100
MiscEvents	100	0	70	100
FSM				100

ISS Coverage				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Instructions	100	91	85	100
Cache assoc.	100	100	100	100
Branch align.	94	37	82	100
Window overflow	100	9	0	100
Hazards	100	16	10	100

HDLScore				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Expression				95

Table 2: Coverage of *proc2* configuration

Vera-based Coverage				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Exception	100	20	33	100
Interrupt	100	71	79	100
Bubble	88	77	100	100
Bypass	100	100	100	100
MiscEvents	100	14	42	100
FSM				99

ISS Coverage				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Instructions	100	95	88	100
Cache assoc.	100	100	100	100
Branch align.	99	76	96	100
Window overflow	100	11	0	100
Hazards	100	15	9	100

HDLScore				
Coverage Target	AVP+MVP	RTPG	VSG	Total
Expression				92

Table 3: Coverage of *proc3* configuration

between dependent instructions. Finally, the "miscEvents" monitor accounts for corner cases such as write buffer overflow and simultaneous assertion of exceptions and interrupts.

The tables also present the results of five different types of ISS coverage monitors. The "instructions" monitor checks that all configured instructions have been executed. The "cache associativity" monitor makes sure that all the cache sets have been accessed and that the line replacement algorithm has been verified. The "branch alignment" monitor checks that all permutations of instruction equivalence classes on different address alignments on the target and fall-through paths of branch and jump instructions have been exercised. The "window overflow" monitor examines the type and number of overflow exceptions produced by different instruction classes. Finally, the "hazards" monitor checks for distinct permutations of instructions issued sequentially.

The results show that our methodology achieves good verification coverage of the processor implementation. Furthermore, the coverage achieved for random configurations, an example of which is shown in Table 3, leads us to believe that our methodology will result in similar coverage for any processor configuration. Results from prototypes of a large number of configurations have further raised our confidence on the effectiveness of our verification methodology. We were able to boot and use the VxWorks real-time operating system on our FPGA-based evaluation boards without any problems. In addition, SOC developers who are using Xtensa in their systems, have been developing applications using those evaluation boards. No bugs have been reported by the system developers. Furthermore, an ASIC implementation of a configuration has been on the field for almost a year without any bugs having been found.

The tools we mentioned so far provide coverage analysis for each configured processor instance. To address the issue of configuration space coverage we have used a two-fold approach. First, we have developed a coverage tool that profiles the pre-configured RTL source code and determines which code segments have not been used under a set of configurations. Based on initial results produced by the tool, we augmented the basic set of configurations used in our daily regressions so that the entire source code is now covered. Second, we have developed a random configuration generator and we have used it extensively. The generator utilizes knowledge of the configuration space in order to generate legal random configurations and to test the design using these configurations. Moreover, we have created tools that measure the coverage of key configuration parameters across the verification of different configurations. One such tool is an active configuration matrix that enables the visual analysis of the configuration space explored and suggests ways to improve its coverage.

Measuring coverage is only useful if the results of the analysis are conveyed back to the verification and design teams and they are used to improve the verification process. We use a web-based management tool to handle the large amount of data generated by the analysis process. This tool, which uses a database to store the results of the analyses, provides different views of the coverage of a single configuration as well as multiple views of the combined coverage across configurations. At the same time, it serves as a collaboration framework among hardware designers, verification engineers, and software developers.

6. SUMMARY

We have presented a verification methodology that was designed to address the increased complexity introduced by configurability. We have addressed this issue by extending and enhancing the traditional verification methodology to create a configurable environment that is not only robust but flexible enough to be employed in the verification of configurable processor cores. We have presented a methodology for generating AVP and MVP diagnostics tailored to the configured processor under verification. We have introduced two random generator families that produce high quality diagnostics targeted to a given processor instance. We have also presented our modular and extensible test-bench that facilitates the easy migration of our models and tools into an SOC verification environment. In order to address the questions of how well a single processor has been verified and how well the verification space has

been explored, we have outlined our coverage analysis methodology which is based on Vera and which employs a large number of tools and metrics to approach the problem from different perspectives.

The collection of methodologies and tools we have presented comprise a powerful environment that not only addresses successfully the challenges of configurability, but also supports reuse for ease of integration into an SOC verification methodology.

We are currently working on expanding our coverage analysis framework with even more metrics and tools. Furthermore, we are extending our verification methodology to address the verification of designer-defined extensibility expressed in the TIE language. Finally, we are developing new random diagnostic generators that provide yet another source of high quality test programs. These generators include the capability of automatically using the results of the coverage analysis tools in the construction of the diagnostics.

7. ACKNOWLEDGEMENTS

This paper is based on the creative work of several individuals at Tensilica. The authors are especially grateful to the entire hardware team for building this verification methodology and for their insightful comments on drafts of the paper.

8. REFERENCES

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 279–285, June 1995.

[2] N. Dohm, C. Ramey, D. Brown, S. Hildebrandt, J. Huggins, M. Quinn, and S. Taylor. Zen and the Art of Alpha Verification. In *Proceedings of the International Conference on Computer Design*, October 1998.

[3] L. Fournier, A. Koyfman, and M. Levinger. Developing an Architecture Validation Suite. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 189–194, June 1999.

[4] D. Geist, G. Biran, T. Arons, M. Slavkin, Y. Nustov, M. Farkas, and K. Holtz. A Methodology For the Verification of a "System on Chip". In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 574–579, June 1999.

[5] R. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2), March/April 2000.

[6] A. Hosseini, D. Mavroidis, and P. Konas. Code Generation and Analysis for the Functional Verification of Microprocessors. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, pages 305–310, June 1996.

[7] S. Mangelsdorf, R. Gratias, R. Blumberg, and R. Bhatia. Functional Verification of the HP PA 8000 Processor. *Hewlett-Packard Journal*, 48(4), August 1997.

[8] J. McLeod, N. Azarakhsh, G. Ewing, P. Gingras, S. Reedstrom, and C. Rowen. Panel: Functional Verification - Real Users, Real Problems, Real Opportunities. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 260–261, June 1999.

[9] Synopsys Inc., Mountain View, California. *VeraTM Verification System, User's Manual*, 1999.

[10] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional Verification of a Multiple-issue, Out-of-Order, Superscalar Alpha Processor–The DEC Alpha 21264 Microprocessor. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pages 638–643, June 1998.