Smart Diagnostics for Configurable Processor Verification

Sadik Ezer
Tensilica Inc.
3255-6 Scott Boulevard
Santa Clara, CA 95054-3013
1-408-566-1743
sadik@tensilica.com

Scott Johnson
Tensilica Inc.
3255-6 Scott Boulevard
Santa Clara, CA 95054-3013
1-408-327-7358
scottj@tensilica.com

ABSTRACT
This paper describes a novel technique called Embedded Test-bench Control (ETC), extensively used in the verification of Tensilica's latest configurable processor. Conventional simulation-based verification methodologies that employ assembly programs for testing cannot easily link the diagnostic program to the test-bench for interactive control, consequently resulting in weaker coverage. ETC links the diagnostic program execution and the test-bench functions, thereby increasing the flexibility and power of the diagnostics to create more complex corner cases in fewer simulation cycles and with smaller code size. This method also enables dynamic self-checking and dynamic coverage analysis by either passing or failing the diagnostic based on the coverage goal, or terminating runaway random diagnostics much earlier. The presented simulation results show that ETC augments verification in two major areas: the creation of more maintainable, efficient, and smart diagnostics, and the reduction of the regression time. Some of the techniques presented in this paper can apply to non-processor verification methodologies as well.

Categories and Subject Descriptors
B.5.2 **[Design-aids]:** Verification

General Terms: Design, Verification.

Keywords: Functional Verification, Configurable Processors, Embedded Test-bench Control, Diagnostics, Coverage.

1. INTRODUCTION
Recent studies show that, due to the complexity of current embedded SOC designs, 70% of the design cycle is spent in verification [1]. Several approaches exist to tackle the verification bottleneck, including reducing chip complexity, increasing resources, or increasing verification productivity. The first option is unreasonable as it is well known that design complexity increases exponentially to take advantage of the advances in semiconductor technology driven by the functionality requirements of emerging embedded applications. Increasing resources is not the competitive approach and often times not scalable. Therefore, building an efficient, maintainable, and re-usable verification strategy is essential for quick time to market with a working product.

Permission to make digital or hard copies 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 and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC'05, June 13–17, 2005, Anaheim, California, U.S.A.
Copyright 2005 ACM 1-59593-058-2/05/0006…$5.00.

Formal verification techniques have recently emerged in the realm of pipelined processor verification [3]. However, it is not possible to do complete chip or system level formal verification with the limitations of today's tools. A simulation-based approach still remains the most popular method. In general, this type of verification is accomplished using directed, pseudo-random, or directed random diagnostic programs. To enhance the verification quality of complex processor designs, several large processor companies have successfully employed internally developed or commercially available random instruction sequence generators [7][13]. Automatic test generation for pipelined processors is also a popular research subject [11] [14].

The biggest challenge in simulation-based methodologies is to guarantee the quality and completeness of verification. Therefore, coverage becomes crucial in such methodologies, so much so that some approaches employ purely coverage-directed or coverage-oriented test generation [2]. Choosing a limited set of diagnostics for optimizing coverage versus regression time is a very difficult problem to solve [10].

In summary, simulation-based processor verification techniques require the development of a robust suite of diagnostic programs, combining random and directed tests and a good coverage methodology. In this paper, we describe a methodology that incorporates and augments the ideas mentioned above, to develop high quality and high performance directed and directed-random diagnostics for the verification of the Xtensa LX processor.

2. THE XTENSA LX PROCESSOR
The Xtensa LX processor is Tensilica's newest configurable and extensible processor with many desirable features such as high compute performance using long instruction words, high I/O bandwidth using a new type of wide processor I/O ports, and low-power implementation with fine-grained clock gating [4]. The extensions to the processor are described using the Tensilica Instruction Extension (TIE) language, which enables the designers to create their own instructions, register files, and other specific functionality, tuning the processor for a specific target application [5] [8]. New in Xtensa LX is the Flexible Length Instruction Extensions (FLIX) technology that allows multiple instruction lengths, which can be freely intermixed. FLIX instruction extensions can define multiple operation slots, each independently decoded and executed, enabling high data parallelism.

Also new in the Xtensa LX processor, TIE ports and queues are designer-defined I/O interfaces, which allow an instruction to directly read from and write to the primary I/O ports of the processor. This capability provides a flexible way for execution

units to directly access external devices, and enables low latency, high bandwidth communication.

The combination of variable length instructions, multiple operation slots, customized description of an unpredictable quantity of new processor ports and queue interfaces, added to the numerous contortions of configurations create a verification nightmare for configurable and extensible processor architectures. We address these challenges using a configurable verification infrastructure and configurable diagnostic suites.

3. CONFIGURABLE VERIFICATION INFRASTRUCTURE

Developing a test-bench for functional verification of a configurable processor is challenging in many ways. Not only must the RTL design be configurable, but also so must everything that surrounds it, including the instruction set simulator (ISS), memory models, event generators, assertion checkers, monitors, and test programs. Modularity is also important to be able to deal with the complexity of the whole infrastructure, which needs to be easily maintainable and portable across generations of processor implementations. We have chosen Vera as the primary verification and test-bench language because it allows modularity, is flexible and portable, and has many desirable features such as object oriented programming, temporal assertions, and functional coverage [12].

Our test-bench, shown in Figure 1, is composed of a large number of Vera components including local memory controllers (IRAM, DRAM, IROM, DROM, ICache, DCache, XLMI), system memory controller, co-simulation module, external event generators, JTAG module, TIE port module, coverage module, checkers, and monitors. A separate module outside the test-bench, called the ETC module, enables the assembly-generated Vera code to control the rest of the test-bench modules. This concept will be described in more detail in the following sections.

Our directed diagnostic suites are composed of approximately 300 architecture verification programs (AVPs) to test the execution of each instruction in the ISA, and 400 micro-architecture verification programs (MVPs) to test the specific implementations of the Xtensa LX processor. Like all test-bench components, diagnostic programs are also configurable. They are Perl scripts that utilize the configuration database in order to generate high quality tests tailored to a specific processor instance [6]. Each diagnostic is accompanied by a database entry file, called the 'info' file, that indicates which conditions satisfy the diagnostic coverage goals: a) for what configurations it is valid, b) which runtime options are required, c) which runtime options prohibit its use, d) what should be the probability distribution of random external events, such as bus waits, bus errors, or interrupts.

Several parameters are used to adjust or tune different components of the test-bench to achieve various behaviors; for example, one can change the frequency of the random or periodic toggling of the busy signals in the local memories, or the frequency and type of bus errors injected by the system memory controller. There are a few different ways of setting these parameters, each of which contributes to a complementary aspect of the verification. They can be passed as runtime simulation flags, they can be placed in 'info' files as described above, or they can be controlled from the assembly programs using the ETC technique that will be described in the following section.

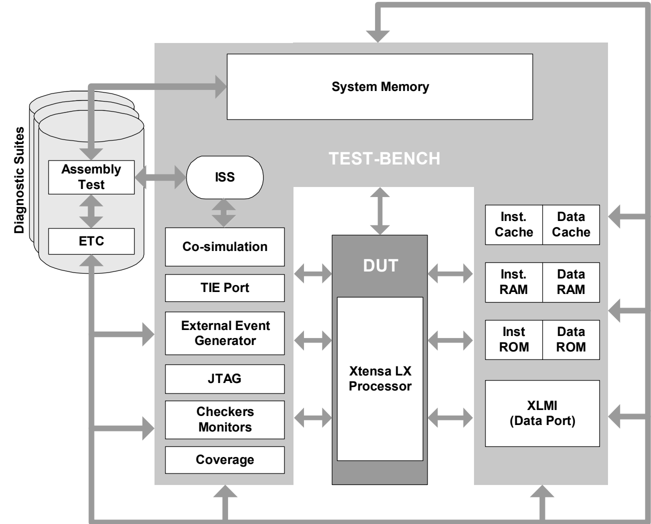

Figure 1. Configurable Xtensa LX Processor Test-Bench

4. EMBEDDED TEST-BENCH CONTROL

As our processor design evolves, generating complex test programs gets more and more complicated. When the test-bench modules that generate random events and the assembly programs are decoupled, it is harder to create many of the corner cases that must be exercised. Furthermore, there are many parts of the test-bench that the diagnostic could use to steer its execution, do self-checking, or terminate the simulation. The method that we have developed and used in our verification infrastructure is called Embedded Test-bench Control (ETC) in assembly programs. This approach enables the assembly code to interact closely with several major components of the test-bench, thus giving the diagnostic writer much more flexibility and power in creating complex scenarios, or getting information dynamically about the current status of the processor to be used in self-checking.

Using ETC, individual diagnostics can easily synchronize execution of a certain code sequence with simultaneously occurring external events. Without ETC, a diagnostic would have to rely on random external events to occur at the desired instant, resulting in wasted simulation cycles and/or loss of coverage.

For example, a diagnostic trying to test contention and arbitration for a local memory port may wish to have a load or store instruction accessing the target memory at the same time that an external master is performing an inbound request (via the PIF) to the same memory. Without ETC, the diagnostic would have to rely on random inbound requests occurring at the right point in time, possibly using a coverage monitor to detect this condition and stop the simulation. By using ETC, the diagnostic can control the external master directly and synchronize that action with its instruction sequence, guaranteeing that the event will occur. This both decreases simulation cycles and improves coverage.

To implement this mechanism, we developed a preprocessor to parse the smart diagnostic's source code, as shown in Figure 2, and find special syntax (**#@**), which indicates the Vera code that is expected to run concurrently with the specified instruction at the commit (W) stage of the pipeline. It is also possible to delay execution of the Vera code for *n* cycles after the instruction is

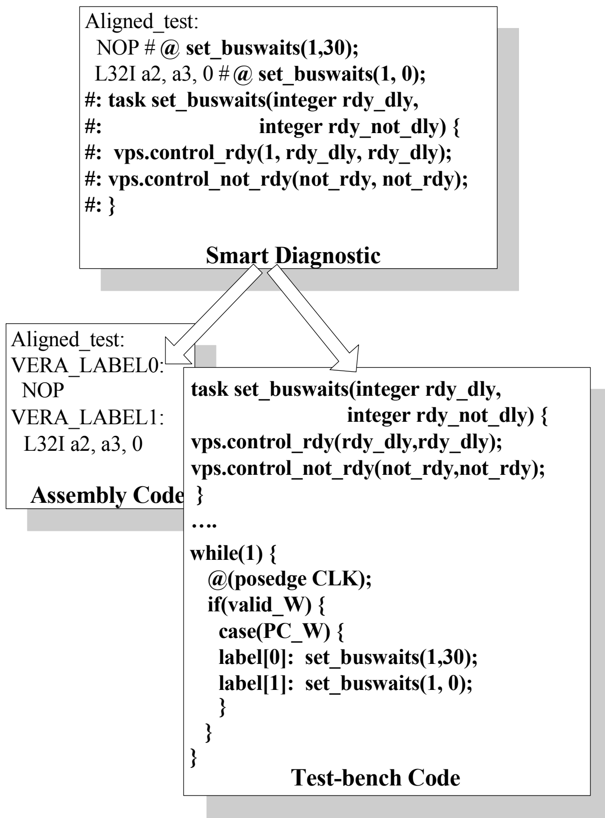

Figure 1. Test Code Generation using ETC

committed using (#@+*n*) syntax. Another syntax, (#:), is used to embed test-bench code that is not tied to the execution of an instruction. The parser builds test-bench code that monitors instruction execution at runtime and matches the W stage program counter (PC_W) values to the addresses of these instructions. When a match occurs, the appropriate Vera code is executed.

This mechanism is complicated by the lack of address information at the time the parser is run: as the assembly and linking process has not yet occurred, the addresses of these instructions have not been determined. To find the appropriate PC values, the parser inserts labels into the assembly source at all such points (see the assembly code shown in Figure 2), so that it may find them again once the diagnostic has been finally linked into an executable. This executable is then analyzed and the addresses of all such instructions are recorded in a text file for use during simulation. The ETC mechanism gives diagnostic writers a new level of control over the verification infrastructure by enabling tight coupling between pipelined instruction execution and external stimuli.

5. USING SMART DIAGNOSTICS

Using ETC in AVPs and MVPs has three major advantages. It provides a mechanism to easily create hard-to-reach scenarios by calling test-bench functions that can generate external stimuli. It gives more flexibility in data checking inside the assembly program by providing the ability to use test-bench functions that can be linked to program execution. Finally, it increases the lifespan of diagnostics across different projects by decreasing the test code size and by providing success status based on the coverage goal, thereby easily identifying broken or obsolete diagnostics that fail to cover their targeted scenario. The following sections give a few detailed examples of verification areas where we obtained the most benefit from this technique.

5.1 Multi-Processor Synchronization Verification

The Xtensa LX processor provides a multiprocessor synchronization option for systems that use shared-memory communication protocols. The synchronization instructions ensure that no load or store access is allowed before all previous load-acquire accesses are performed; similarly, before a store-release access is performed, all previous load, store, acquire, and release accesses must be performed. To test synchronization instructions in a single processor test-bench is a challenging task. The verification of configurations with dual load-store units becomes more interesting because the order of memory accesses across different slots of a FLIX packet with concurrent operations must be checked as well. Checking the memory access order of all load or store instructions separated by acquire and release instructions is achieved using ETC.

Figure 3 shows a program code snippet of a smart diagnostic that verifies synchronization instructions using FLIX packets to generate multiple memory accesses with wide instructions. This particular processor configuration has two slots in its FLIX instructions, each of which may contain a 32-bit load (L32I), store (S32I), or a no-op (NOP) instruction. Many load-acquire (L32AI) and store-release (S32RI) instructions are dispersed among these random FLIX instructions. In the test, after every FLIX packet that contains a memory access instruction in one or both of its slots, the corresponding *add_address* call(s) are added as well, to store the address of the memory access in the current address bucket. When a synchronization instruction is encountered, the bucket number is incremented, so that addresses of all memory access instructions after it are collected in a separate bucket. During simulation, part of the ETC code (not shown) will monitor the memory interface and compare all addresses with the ones collected in different buckets to make sure that the protocol is not violated. This example demonstrates the use of ETC in writing efficient self-checking diagnostics.

```
// Target Addresses
// a4: 0x60000000
// a5: 0x60000040
// a6: 0x60000080
// a7: 0x60010000
  . . .
  MEMW  #@ collect_addresses();
#: task collect_addresses() {
  {S32I a2, a5, 0; NOP;}
#: add_address(basket_id, 32'h60000040);
  L32AI a6, a7, 0  # Load Acquire
#: basket_id++;
  {L32I a2, a4, 4;  L32I a2, a5, 8;}
#: add_address(basket_id, 32'h60000010);
#: add_address(basket_id, 32'h60000060);
  {L32I a3, a5, 8;  S32I a2, a4, 0;}
#: add_address(basket_id, 32'h60000060);
#: add_address(basket_id, 32'h60000000);
  . . .
#: } // end task collect_addresses()
```

Figure 3. MP Synchronization Test

5.2 TIE Ports and TIE Queues Verification

TIE ports and queues are new I/O interfaces in the Xtensa LX processor. There are two types of TIE ports. State exports are designer-defined states that can be made visible at the top-level pins. Import wires enable designer-defined TIE instructions to read inputs from designer-defined primary pins of the processor. TIE queues are push or pop type interfaces with full or empty flow control signals. An input queue definition includes a data signal (IQ), an input control signal (IQ_Empty) and an output control signal (IQ_PopReq). Empty indicates that the external queue has no data to read, and PopReq indicates that the data is being read and popped from the queue. Similarly, an output queue has data (OQ), an input (OQ_Full), and an output (OQ_PushReq) control signals. Full indicates that the external queue cannot be written any more, and PushReq indicates that data is being written to the queue by the processor. With TIE instructions, it is possible to push data to an output queue if it is not full, or pop data from an input queue if it is not empty.

Assertion of empty or full causes instructions that use these queues to stall. A stalled queue instruction should not hold the previous queue instruction from executing; otherwise a deadlock can happen. The test code snippet shown in Figure 4 is an example of a diagnostic which verifies that no deadlock occurs in this scenario. In this test, the write instruction OQWRITE pushes to the output queue OQ, followed by a read instruction IQCAT that pops from the input queue IQ. The test uses ETC cleverly to control stalling of the instructions. While the OQ_Full signal is asserted for only 100 cycles to temporarily stall the write queue instruction, IQ_Empty is asserted indefinitely, and will stall the read queue instruction until it is de-asserted. The ETC code monitors the push request count that increments when the write queue instruction is un-stalled, making sure that the read queue instruction that is stalled forever does not block the previous write queue instruction, therefore creating a deadlock. When the pop request is observed, the IQ_Empty signal is released, resulting in the unblocking of the read queue instruction and the graceful termination of the test. If deadlock does occur, execution will stall indefinitely and some other monitor (not shown) will fire, causing the test to fail. This example shows that ETC can be used both for generating external stimuli and for monitoring the processor states to steer program execution.

```
#Write opcode: IQCAT
   NOP     #@ deadlock();
   NOP.N
   OQWRITE    $a14,$a12    # Write Queue
   IQCAT                   # Read Queue
   EXTW
#:task deadlock() {
#:  integer count;
#:  tiewire.drive_OQ_Full(1'b1, 100);
#:  tiewire.drive_IQ_Empty(1'b1, -1);
#:  while(1) {
#:   count = tiewire.OQ_PushReq_count();
#:   if( count == 1 ) {
#:    tiewire.release_IQ_Empty();
#:    break;
#:   }
#:   @(posedge Xtensa_if.CLK);
#:  }
#:}
```

Figure 4. TIE Queue Deadlock Test

6. DIAGNOSTIC-BASED COVERAGE

Functional coverage is an integral part of our test-bench. Its hierarchical nature encompasses the complete design. We used Vera's functional coverage feature in conjunction with Open Vera Assertions (OVA) used as a temporal language when complex event sequences were needed. Among many desirable features, dynamic coverage feedback capability was essential to our goal-oriented diagnostics.

Unit level coverage monitors were utilized in regressions to point to weakly verified areas in general. Diagnostic-based coverage was a new approach that integrated the coverage goal as a metric to decide the pass or fail status of the tests. This approach had two advantages: to easily identify broken or obsolete diagnostics that did not achieve their intended purposes, and increase the performance of directed-randoms by terminating the program as soon as the coverage goal is reached.

Each goal-oriented diagnostic has a corresponding coverage rule and a query function that can be called periodically during or near the end of the simulation using ETC, which dynamically links the assembly test to the coverage monitors residing in the test-bench. Before our coverage methodology was in place, the diagnostic was run for a length determined by the diagnostic's author to be sufficient to guarantee coverage of the targeted scenario. When diagnostic-based coverage was added, the program is terminated as soon as the coverage goal is reached, thereby decreasing the average simulation time by more than 60%. Figure 5 shows 10 different random simulation results of the same test program on the same processor configuration. This approach was used to do a coverage and simulation performance comparison that is presented in the next section.

7. RESULTS AND CONCLUSIONS

To investigate the usefulness of ETC, we have compared the results of directed or directed-random diagnostics using ETC to the ones that are generated using our random test program generator (RTPG) on two different processor configurations. The first processor (A) is a full-featured configuration of Xtensa LX including all local memories, a wide processor interface, a 7-stage pipeline, and two load-store units. The second processor (B) has one instruction RAM, one data RAM, and a 5-stage pipeline. Both processors use designer-defined TIE instruction extensions accessing TIE ports and queues.

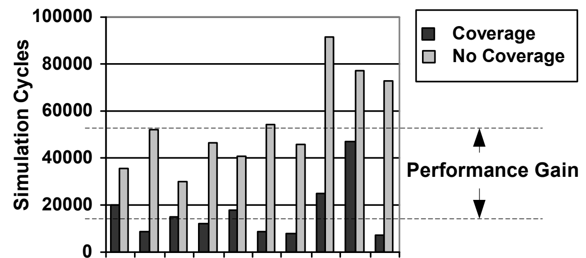

Figure 5. Simulation Performance with Diagnostic-Based Coverage

Table 1. Coverage Results Comparison

Coverage Targets	Processor A		Processor B	
	ETC	RTPG	ETC	RTPG
Exceptions	100	34	100	28
TIE Queue Contention	100	50	100	50
Load/Store/Ifetch Arbitration	96	65	100	100
DMA Interleaved	100	78	100	100
Time spent for ETC simulations is 6% of the time spent for RTPG.				

7.1 Coverage and Performance Improvements

We have simulated on both processors 22 ETC diagnostics that took 231,046 simulation cycles to run, and 100 RTPG diagnostics with random external events (interrupts, bus errors, bus waits, random toggling of queue control signals, random inbound PIF requests, etc.) that took 3,820,219 simulation cycles. Coverage results of selected goals are presented in Table 1. Because ETC diagnostics are coverage oriented, most were able to attain 100% coverage in much less simulation cycles, whereas random diagnostics had long simulation times and poor coverage results.

To demonstrate the performance improvements more clearly with the proposed technique, we have simulated a few selected diagnostics with ETC and with fine-tuned random external events on both processor configurations. The simulation results of three selected diagnostics with and without ETC are presented in Figure 6. The first diagnostic is intended to verify the instruction fetch and external interface units. The diagnostic sends a load to instruction RAM that causes the current instruction fetch request to be canceled. Bus contention is generated during the next fetch and an interrupt causes this second fetch to be canceled as well, thereby forcing the external interface unit to use different IDs for two back-to-back fetch requests. The described scenario is difficult to achieve with random bus waits and interrupts, therefore, ETC is used to slow down instruction fetch responses, allowing plenty of time for the interrupt to cancel the next request.

The TIE queue deadlock test described in Section 5.2 is also used to compare simulation performance with or without ETC. With random toggling of queue control signals IQ_Empty and OQ_Full, it takes many simulation cycles, and fine adjustment of the random weights, to create the deadlock scenario. With ETC, however, it is possible to monitor and assert input signals precisely at the right moment, thus creating the circumstances that might cause a deadlock.

The final test program that we selected is responsible for covering a long inbound processor interface (PIF) request to each configured local memory interrupted by a load or store instruction that accesses the same memory. Although creating many random inbound requests in conjunction with load and store instructions is trivial, making sure that this event happens to all configured memories takes a long time. With ETC, and with help from our configuration database, we created this case very quickly. It was possible to control the type, size, destination address, and exact timing of the inbound PIF requests injected by the system memory controller.

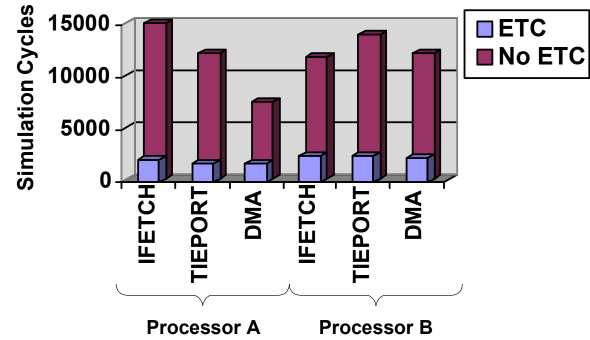

Figure 6. Simulation Performance with ETC

As seen in Figure 6, the simulation performance of the diagnostics with ETC on the same configuration is about the same, and minor variation across configurations can be explained by the difference in the reset code sequence of different processors. On the other hand, when random external events are used instead, the simulation times vary considerably, and in some cases full coverage is not guaranteed. On the average, the simulation performance is reduced by 80% when ETC is used.

Another metric that somewhat represents the difficulty of the test cases better is the coincidence number. The coincidence number is the number of simultaneous or sequential events that need to happen to achieve a corner case. For example for three individual events, A, B, and C, the coincidence number of event (A&B) -> C, A and B followed by C, is three. We have seen that ETC makes it easier to create events with high coincidence numbers in directed or directed-random diagnostics.

An alternative to using ETC is to program the test-bench using magic stores, which have an unpredictable delay between the store and the magic action. ETC is more maintainable because the bus latency does not affect the diagnostic anymore; every new design that has slightly different bus latency does not cause all the old diagnostics to break because the timing is now different. This saves considerable engineering effort in creating new diagnostics or maintaining the old ones.

Although the results presented here seem to point otherwise, we don't intend to favor directed diagnostics using ETC over randoms, and undermine the importance of random testing. One of the most important advantages of random testing is that it can cover scenarios overlooked by the verification engineer. Therefore, the ETC technique cannot be a replacement for random testing, but can be used as a powerful enhancement to the overall verification methodology.

7.2 Conclusions

Verification of configurable processors requires an infrastructure and a methodology that is robust and flexible to address the increased complexity introduced by configurability. Our methodology employs a modular test-bench and diagnostic suites that are tailored to the configured processor under verification. To enhance the performance and coverage of assembly tests in our diagnostic suites, we developed a methodology to seamlessly integrate diagnostics and test-bench modules, thus creating a unified verification environment. This approach gave diagnostic writers a lot of flexibility and power to create complex corner cases in less code size and fewer simulation cycles, lessening the need to utilize

random external events to achieve scenarios that are impossible to generate by assembly programs alone. This methodology not only improved overall coverage and simulation performance of our diagnostic suites, but also increased longevity of diagnostics across generations of processor implementations by enabling a diagnostic-based coverage approach as a metric to decide the final result of the simulation.

7.3 Future Work

This technique currently limits the synchronization of the test-bench code at or after the commit (W) stage of the processor pipeline. It may be possible, but certainly much harder to execute ETC code at earlier stages such as instruction fetch (I), register read (R), or memory read (M) stages, due to the speculative nature of these stages after which the instruction may be replayed or even killed. ETC also has limitations on dynamically scheduled superscalar processors, because multiple instructions may be committed at the same cycle, making synchronization of assembly and test-bench code more difficult. Furthermore, the technique currently supports only one way communication between the test and the test-bench code. It is possible to extend this capability, for instance by enabling backdoor writes to the system memory as a possible communication path back to the diagnostic. Finally, because it mostly relies on hand-written directed diagnostics, there is a potential risk of missing unintentional bug coverage if random diagnostics are omitted entirely from the verification methodology.

We are currently in the process of employing two new random assembly test generators. The first one is a commercial tool developed by a third party vendor, and the second one is an in-house development effort. We believe that it is possible to extend the capabilities of these generators using ETC. We also use several small-scale, task-specific directed-random test generators; examples include random loop generators, data cache stress testing, and random TIE code generators. We are working on a dynamic coverage feedback mechanism using ETC to change weights that control these directed-random generators.

8. ACKNOWLEDGMENTS

Xtensa LX verification involved a large team of people whose names are too many to be listed here. We appreciate the work of everyone who contributed to this effort, especially the hardware team whose ingenious and innovative minds turned the Xtensa LX dream into a reality. We also thank everybody who gave feedback on early drafts of this paper.

9. REFERENCES

[1] Bergeron, J. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[2] Gluska, A. *Coverage-Oriented Verification of Banias.* ACM/IEEE Design Automation Conference, pages 280-285, June 2003.

[3] Bentley, B. *Validating the Intel Pentium 4 Microprocessor.* ACM/IEEE Design Automation Conference, pages 244-248, June 2001.

[4] Jani, D., Ezer, G., and Kim, J. *Long Words and Wide Ports.* Hot Chips, August 2004.

[5] Rowen, C. *Engineering the Complex SOC.* Prentice-Hall PTR, 2004.

[6] Puig-Medina, M., Ezer, G., and Konas, P. *Verification of Configurable Processors.* ACM/IEEE Design Automation Conference, pages 426-431, June 2000.

[7] Behm, M., Ludden, J., Lichtenstein, Y., Rimon, M., and Vinov, M. *Industrial Experience with Test Generation Languages for Processor Verification.* ACM/IEEE Design Automation Conference, pages 36-40, June 2004.

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

[9] Ho, R., and Horowitz, M. *Validation Coverage Analysis for Complex Digital Designs.* IEEE/ACM International Conference of Computer Aided Design, 1996.

[10] Fine, S., Ur, S., and Ziv, A. *Probabilistic Regression Suites for Functional Verification.* ACM/IEEE Design Automation Conference, pages 49-54, June 2004.

[11] Iwashita, H. *Automatic Test Program Generation for Pipelined Processors.* IEEE/ACM International Conference on Computer-Aided Design, pages 580-588, 1994.

[12] Synopsys Inc., Mountain View, California, Vera® *User Guide.* March 2004

[13] Kohno, K., and Matsumoto, N. *A New Verification Methodology for Complex Pipeline Behavior.* ACM/IEEE Design Automation Conference, pages 49-54, June 2004.

[14] Hennenhoefer, E., Typaldos, M. *The Evolution of Processor Test Generation Technology.* Obsidian Software Inc.