Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor -- the Alpha 21164 CPU Chip

by Michael Kantrowitz and Lisa M. Noack

ABSTRACT

Digital's Alpha 21164 processor is a complex quad-issue, pipelined, superscalar implementation of the Alpha architecture. Functional verification was performed on the logic design and the PALcode interface. The simulation-based verification effort used implementation-directed, pseudorandom exercisers, supplemented with implementation-specific, hand-generated tests. Extensive coverage analysis was performed to direct the verification effort. Only eight logical bugs, all unobtrusive, were detected in the first prototype design, and multiple operating systems were booted with these chips in a prototype system. All bugs were corrected before any 21164-based systems were shipped to customers.

INTRODUCTION

The Alpha 21164 microprocessor is a quad-issue, superscalar implementation of the Alpha architecture. The CPU chip required a rigorous verification effort to ensure that there were no logical bugs. World-class performance dictated the use of many advanced architectural features, such as on-chip virtual instruction caching with seven-bit address space numbers (ASNs), an on-chip dual-read ported data cache, out-of-order instruction completion, an on-chip three-way set-associative write-back second-level cache, support for an optional third-level write-back cache, branch prediction, a demand-paged memory management unit, a write buffer unit, a miss-address file unit, and a complicated bus interface unit with support for various CPU-system clock ratios, system configurations, and third-level cache parameters.[1]

Functional verification was performed by a team of engineers from Digital Semiconductor whose primary responsibility was to detect and eliminate the logical errors in the Alpha 21164 design. The detection and elimination of timing, electrical, and physical design errors were separate efforts conducted by the chip design team.[2]

Extensive functional verification prior to releasing the first-pass design to the manufacturing process is a common technique used to ensure that time-to-market goals are met for complex processors. Increasingly, these verification efforts are relying on pseudorandom test generation to improve the quality of the verification effort. These techniques have been in use at Digital for more than seven years and are also used elsewhere in the industry and in academia.[3-6] This paper describes a

functional verification effort that significantly extended pseudorandom testing with extensive coverage analysis and some hand-generated tests to produce working first-pass parts.

GOALS

The verification team had several key goals. Goals for first-pass silicon included ensuring that the first prototypes could boot the operating system and providing a vehicle for debugging of system-related hardware and software. An additional goal was to execute a test to check every block of logic and every function in the chip to ensure that no serious functional bugs remained. The goal for second-pass silicon was to be bugfree so that these chips could be shipped to customers for use in revenue-producing systems. Secondary goals included assisting in the verification of Privileged Architecture Library code (PALcode) and keeping manufacturing test patterns in mind when creating the verification environment and writing tests.

MODELING METHODOLOGY

Several different model representations of the Alpha 21164 CPU were developed for testing prior to prototypes. The verification team primarily used a register-transfer-level (RTL) model of the Alpha 21164 CPU chip. This model accurately represented the detailed logic of the design and delivered very high simulation performance.

Modeling Environment

The design team wrote the RTL model in the C programming language. The model represented all latches and combinatorial logic of the design and was accurate to the clock-phase boundary. The C programming language was chosen because C provides the speed and flexibility needed for a large-scale design. Digital's CAD group designed a user interface for access into the RTL model of the Alpha 21164 CPU. The C command line interface (CCLI) allowed access into the variables used to define signals and to the routines that represented the actual design. It provided the ability to create binary traces of signals for postprocessing analysis and debugging. A standard set of macro-instructions simplified bit manipulation of signals with arbitrary widths.

The use of C also allowed the team to simulate portions of the gate-level design in the structural simulator, CHANGO, and to perform cycle-by-cycle comparisons with various states in the RTL model. These simulations, called shadow-mode simulations, were fully utilized for testing the various functional units of the chip.

Pseudosystem Models

The verification team developed several models to interface to the Alpha 21164 CPU RTL model and to allow testing of interactions with pseudosystems to occur. The C language provided a level of flexibility in the creation of these models that was not available on previous verification projects. One area in which this flexibility was fully utilized was in the formation of a sparsely populated memory model. By using a dynamic tree data structure rather than a static array, the cache, duplicate tag store, and memory system models could be written to support the full range of 64-bit addressing. Hence, tests could be created to use any set of addresses without restrictions. In addition, comparisons with the reference model could be drawn from the entire contents of memory. This significantly enhanced the ability to detect possible errors in the design.

The verification engineers created a system model (the X-box) to simulate transactions on the pin bus. The X-box model provided a means to mimic the real system behavior that the Alpha 21164 CPU would encounter when used with a variety of different platforms. The team used C to develop an X-box model that could be connected to every possible configuration and mode setting of the Alpha 21164 CPU chip. This allowed all modes of the Alpha 21164 CPU to be tested with a single, multipurpose system interface model. The X-box also performed many of the checks needed to ensure the proper operation of the system bus.

STRATEGY

The verification strategy employed multiple techniques to achieve full functional verification of the Alpha 21164 chip. The primary technique used was pseudorandom exercisers. These programs generated pseudorandom instruction sequences, executed the sequences on both the 21164 model and a reference model, and compared the results. A second major technique used focused, hand-generated tests to cover specific areas of logic. Other methods consisted of design reviews, executing existing tests and benchmarks, and a few static analysis techniques. Figure 1 shows the general flow for a single simulation.

[Figure 1 (Design Verification Test Environment) is not available in ASCII format.]

This strategy was deployed in three parts: the try-anything phase, the test-planning phase, and the structured completion phase. Devising a test plan was not the first step. During the early stage of the project, the primary goal was to stabilize the design as quickly as possible. Any major bug that would have had an impact on the architectural definition of the chip was uncovered. Circuit design and layout could then commence without fear of major revisions later. If time had been spent structuring detailed test plans, less time would have been given to actual

testing, and at this point in the design, careful thought was not needed to find bugs.

The main purpose of the try-anything phase was to exercise as much functionality of the design as possible in the shortest time in order to stabilize the design quickly. This phase began even before the RTL model was ready, with the construction of the pseudorandom exerciser programs. The pseudorandom exercisers and the RTL model were debugged together. This produced an atmosphere of intensity and challenge in which everyone was required to interact constantly to help identify the source of problems. This had a secondary benefit of bringing the design and verification teams closer together.

Once the design stabilized and the bug rate declined, the design team began focusing on circuit design and layout, and the verification team took a step back and created a test plan. The purpose of the test plan was to ensure that the verification team understood what needed to be verified. The test plan provided a mechanism for reviewing what would be tested with the design team. The joint review ensured that the verification team did not miss important aspects of the design. The test plan also allowed a way for the design team to raise issues around specific problem areas in the design or areas that employed special logic that were not obvious from the specification. Finally, the test plan provided a means for scheduling and prioritizing the rest of the verification effort.

The test plan consisted of a description of every feature or function of the design that needed to be tested, including any special design features that might require special testing. It did not describe how the test would actually be created. Past experience had indicated that test plans describing the specific sequence of instructions needed to test chip features quickly became outdated. Instead, the test plan focused on the "what," not the "how."

The final verification step was the structured completion phase. During this time, each item from the test plan was analyzed and verified. The analysis consisted of deciding which mechanism was appropriate for covering that particular piece of the design. This might consist of a focused test, a pseudorandom exerciser with coverage analysis, or an assertion checker. As the verification of each item was completed, a review was held with the design and architecture teams to examine what was verified and how it was done. In this way, any problems with the verification coverage were identified.

TEST STIMULUS

Both focused and pseudorandom exercisers were used during the verification of the Alpha 21164 chip. More than 400 focused tests were created during the verification effort, covering a wide

variety of chip functions. Six different pseudorandom exercisers were used. One was a general-purpose exerciser that provided coverage of the entire architecture. Each of the other five exercised a specific section of the chip in a pseudorandom way.

The one general-purpose exerciser used was provided by a separate group and generated pseudorandom streams of instructions, data, and chip state. Its focus was at the architectural level and generated pseudorandom stimulus that would work on any implementation of the Alpha architecture.

Almost all focused design verification tests (DVTs) were written using Alpha assembly code. This provided the right level of abstraction to avoid the need to toggle ones and zeros directly on each pin, yet allowed specific control over the timing of transactions and instruction sequences that would not be possible from a compiled language. The macro-preprocessor feature of the Alpha macro-assembler was used heavily. This allowed the assembly-level programs to be constructed in a modular manner.

PSEUDORANDOM TESTING

Pseudorandom testing offers several advantages in the verification of increasingly complex chips. These include producing test cases that would be time-consuming to generate by hand, and providing the ability to generate multiple simultaneous events that would be extremely difficult to think of explicitly.

Exercisers

In support of the pseudorandom testing strategy, various exercisers were created that focused on different aspects of the chip. The following areas were targeted explicitly:

- Branching
- Data-pattern-dependent transactions
- Floating-point unit
- Traps
- Cache and memory transactions

Fundamentally, each exerciser was the same. The exerciser would create pseudorandom assembly-language code, run the code on the model under test and a reference model, collect results from each, and compare the results from both model runs. Any errors or discrepancies were then reported to the user.

The reference model used, called the ISP model, was a very high-level abstraction of the Alpha architecture written in the C language. The core of this model was created during the

design of the 21064, the first Alpha processor. It was modified slightly to include Alpha 21164 specific features such as internal register definitions. The ISP model integrated the same sparsely populated memory model used in the pseudosystem model in such a way that the freedom in creating addresses could be duplicated.

SEGUE, a text generation/expansion tool, was used extensively to create pseudorandom code and configurations. Each exerciser used SEGUE templates to generate code. Variables were passed to the SEGUE templates that would determine what percentage of certain events or instructions would occur in the resultant code. Users would vary the percentages and create additional templates to target their exercisers to certain portions of the chip. An exerciser could focus only on loads and stores, or templates could be created that would generate trapping code. The verification engineers had the flexibility to create whatever code was needed. The verification engineers worked closely with the designers to understand the details of the logic. As a result, cases could be generated that would thoroughly test the functions being designed into the Alpha 21164 CPU chip.

Configuration Selection

Each test, either pseudorandom or focused, also made use of a configuration control block (CCB) parameter file. The CCB was used to set up the type of system that would be emulated for a given simulation. The parameter file consisted of variables that could be weighted to make certain system events occur or to cause certain configurations to be chosen. Once again, SEGUE scripts were utilized to create the command files that controlled these events. Examples of the type of events that could be chosen were single-bit error-correcting code (ECC) errors, interrupts, the presence of an external cache, the ratio between the system clock and the CPU internal clock rate, cache size and configuration, and other bus-interface timing events. These and other events were varied throughout the course of the project to ensure that the chip could be run in real systems using any given configuration.

The configuration chosen was guided through the use of a parameter file that contained various parameters and weightings to be utilized by SEGUE. Once a configuration was chosen using the parameter file, it was processed to produce two files used in the simulation. The first was a CCLI control file used to set up state internal to the pseudosystem-level model. The second file was loaded into the memory model to be used by the DVT and to provide information accessible through assembly code regarding the configuration type.

Simulation

Once the pseudorandom code and configuration had been generated, the test was loaded into the model under test or into the ISP model to use as the stimulus. A DVT loader was created for both models that would interpret selected data in the CCB and determine the memory locations where the test should be located. The additional information encoded in the CCB included whether the test ran in I/O, where handlers should be placed, and what page mapping was used.

After a DVT was loaded, the simulation would start. A PALcode reset handler was executed first. It read information from the CCB and loaded various registers with the configurations specified. The DVT was executed after the PALcode completed.

Capturing Random Events

In some cases, pseudorandom exercisers were used to capture events that were unlikely to occur and that would have been difficult to obtain by a focused test. By using a new tool (called FIGS), engineers were able to use the pseudorandom exercisers and postprocessing to look for events that were needed to achieve coverage of the various functions in the F-box. When the event occurred, the event could be saved and re-created for future regression testing.

CORRECTNESS CHECKING

A variety of mechanisms were used for checking whether the model behaved correctly. Some handcrafted tests had comparisons built-in to verify that they generated the expected answer. This self-checking mechanism, however, is difficult to include with pseudorandom testing. Three categories of checking mechanisms were developed that could work with pseudorandom or focused tests. These were checks performed during simulation of a model, postsimulation checks done automatically every time a model completes executing, and test-specific postsimulation checks. In all cases, adjusting the checking mechanisms to eliminate reporting false errors was important to keep the debugging time low.

The RTL model was augmented with a wide variety of built-in assertion checkers. These were active any time the model was run; they verified that various assertions and rules of behavior were not violated at any time during the test execution. Assertion checkers ranged from the simple to the complex and were added to the model by both the design and verification teams. Some assertion checkers were added as the initial model was coded, and others were added as needed to ensure that certain situations did not occur. Examples of simple assertion checkers include watching for a transition to an illegal state in a state machine, or watching for the select lines of a multiplexer (MUX) to choose an unused MUX input. More complex assertion checkers were used that

required explicit knowledge about illegal sequences. For example, the system bus had a complicated set of checkers attached to it that checked for violations of the bus protocol.

When a test completed executing on the model, several end-of-run checks were done automatically. One simple check was to verify that the test reached its normal completion point and had not ended prematurely. Complete cache coherency checks were performed to ensure that all three levels of cache contents were consistent with the memory image.

A variety of very powerful end-of-run checks were used. These compared the results of running a test on the model and on the ISP model. Information about the state of the model was saved while the test was executing and then compared with its equivalent from the ISP model. State that was compared in this way included a trace of the program counter (PC), a trace of the updates made to each architectural register, and the final memory image upon completion of a test.

The main problems encountered with this technique were due to inconsistencies between the ISP model and the Alpha 21164 design. The ISP model was used across multiple Alpha design projects. It provided architecturally correct results but had no concept of timing, pipelining, or caching. Several features of the Alpha 21164 implementation were difficult to verify with this reference machine.

In the Alpha architecture, arithmetic traps are imprecise, in that they might not be reported with the exact PC that caused them. Since the ISP model had no concept of timing, it reported traps at a different time than the real design. Thus, the checking mechanisms needed to be intelligent enough to take this possibility into account. Arithmetic traps also presented a problem because the destination register of certain types of traps is unpredictable after a trap occurs. Combined with the imprecise nature of traps, unpredictable values could propagate to other registers, making comparison against the reference machine difficult. Normally, certain software conventions would be followed to control these aspects of the architecture. To achieve the full benefit from pseudorandom testing, however, no restrictions were placed on which registers or instruction sequences could be used. Instead, an elaborate method was devised for tracking which registers were unpredictable at any given time. This information was then used to filter false mismatches.

Optional checks made on a per-test basis could be viewed as more complicated assertion checks. These were performed by tracing internal signals. The specific signals to trace were selected based on the particular postprocessing to be done. Then, by using a library of routines (called SAVES) to simplify accessing and manipulating these signal traces, particular interactions and protocols were verified. These could be viewed as assertion checks, but they were more complicated than the built-in variety.

One example involved representing the behavior of a large section of the design as a single, complicated state machine. The behavior of this state machine could be compared with the I/O behavior of the actual design section. Another example was the representation of the branch-prediction algorithm in a more abstract form than the actual model. The behavior of the abstract algorithm was compared with the behavior of the model itself.

COVERAGE ANALYSIS

The primary difficulty with functional verification is that it is virtually impossible to know when the verification effort is complete. Completing a predetermined set of tests merely indicates that the tests are complete, not that the design has been fully tested. Monitoring the bug rate provides useful information, but a low bug rate might indicate that the testing is not exercising the problem areas. To alleviate this problem and provide increased visibility into the completeness of the verification effort, extensive coverage analysis of the focused tests and pseudorandom exercisers was done. Two types of coverage checking were used: information gathered while a model was executing, and information gathered by postprocessing signal traces.

While a model was executing, information was being stored about the occurrence of simple events. For example, a record was kept on the number of times the machine issued instructions to four pipes simultaneously, the number of times the translation buffers filled up, or the number of times stalls occurred. Since the chip operated in random configurations, a record was also kept about the configuration information such as the B-cache size and timing selected, the system interface options, and timing. At the end of every model run, this recorded information was written to a database to collect statistics across multiple runs.

In addition to these simple coverage checks, more elaborate coverage analysis was done through postprocessing. By using the SAVES library, signal traces were collected while the model was executing; these were later analyzed for the specific occurrence of predefined events. The events were composed of complicated timing relationships among signals. Often, two-dimensional matrices were created, in which each axis of the matrix represented a list of events. Thus, the occurrence patterns of every event in one list could be visualized happening with every event in the second list. For example, it was verified that every type of system command (read, invalidate, set-shared, etc.) occurred followed by every type of bus response (ACK, NOACK, etc.).

Automatic coverage-checking methods were also used. The most common was a state machine coverage analyzer. It was a goal to verify that every state/arc transition in every state machine was being exercised. Programs were automatically generated to search

the trace files for these transitions and record the information about what was and was not covered. This concept was extended to sections of the chip that were not designed as simple state machines. As described above, one large section of the design was represented as a single, monolithic state machine to provide an independent reference for the correct outputs of the section. This conceptual state machine was processed through the coverage analysis tool. Although the transitions that were checked did not map directly to the physical design, they did provide an excellent indication of how well that section of the design had been tested.

The trace analysis tools could accumulate data across multiple simulation runs. The data was analyzed periodically, and areas that were lacking coverage were identified. This allowed the identification of trends in the coverage and provided an understanding as to how well the pseudorandom exercisers were exercising the chip. With this insight, pseudorandom exercisers were modified or new focused tests were created to improve the test coverage. Running pseudorandom exercisers with coverage analysis proved to be a very powerful technique in functional verification.

BUG TRENDS

During the Alpha 21164 CPU verification effort, more than 600 bugs were logged and tracked before first-pass parts were manufactured. Figure 2 shows the bug rate achieved as a function of time for the duration of the project. To track bugs, an action tracking system was set up. Tracking of bugs started after all the subsections of the RTL-level model had been integrated and a small subset of tests was run successfully. Since many areas of the model were ready before others, the action tracking system does not represent all the issues raised. However, it is interesting to look at the trends presented by the data.

[Figure 2 (Bug Rate as a Function of Time) is not available in ASCII format.]

The first trend to consider is the effectiveness of the pseudorandom and focused efforts. As shown in Figure 3, more than half the bugs were found using pseudorandom techniques. Furthermore, one-third of the bugs found by the focused effort were in the error-handling functionality of the design, which had poor pseudorandom test coverage.

Figure 3 Effectiveness of Class of Test

PSEUDORANDOM TEST	61%
FOCUSED TEST	31%
STATIC TEST	1%
OTHER	7%

Bugs were thought to have been introduced in a variety of ways. Figure 4 shows the breakdown of the causes of bugs. The majority occurred in implementing the architectural ideas that were decided upon for the project.

Figure 4 Introduction of Bugs

IMPLEMENTATION ERROR	61%
C PROGRAMMING MISTAKE	17%
PALCODE ERROR	9%
ARCHITECTURAL CONCEPTION	3%
BACK-ANNOTATION OF MODEL (TO MATCH SCHEMATICS)	3%
DOCUMENTATION/SPECIFICATION	2%
SCHEMATIC ENTRY	1%
PROGRAMMABLE LOGIC PROGRAMMING ERROR	1%
POOR COMMUNICATION	1%
OTHER	2%

Figure 5 shows the various detection mechanisms that were used to detect bugs. As in the past, assertion checkers placed in the design to quickly detect when something is not correct are the most successful.

Figure 5 Effectiveness of Bug Detection Mechanisms

ASSERTION CHECKER	34%
SELF-CHECKING TEST	11%
CACHE COHERENCY CHECK	9%
REGISTER FILE TRACE COMPARE	8%
MEMORY STATE COMPARE	7%
MANUAL INSPECTION OF SIMULATION OUTPUT	7%
SIMULATION HANG	6%
ARCHITECTURAL EXERCISER BUILT-IN CHECKS	6%

PC TRACE COMPARE	4%
SAVES CHECK	3%
SIMULATOR BUILT-IN ERROR MESSAGE	2%
OTHER	9%

RESULTS AND CONCLUSIONS

As of September 1, 1994, eight logical bugs were found in the first-pass Alpha 21164 CPU design. Only one of these impacted normal system operation, but it did not occur very often. The first two issues were found while debugging test patterns on the tester; the third was a variation on a known restriction; the fourth occurred in a rare prototype system configuration that was found through pseudorandom simulation testing (which had continued even after the design was released to manufacturing); the fifth was a race condition between two events that rarely were stimulated in simulation; the sixth was a performance-related issue on the pin interface that was found by thinking about the design; the seventh was a very specific set of events that resulted in a system hang; and the last was related to not responding appropriately to an error condition.

These bugs escaped detection for the following reasons:

- An exerciser running on a simulator was slow to encounter the conditions that would evoke the bug. Many conditions needed to occur concurrently, but all of them occurred infrequently.
- An assertion checker did not work properly.
- Comparisons between the RTL model and the structural model missed the bug.

All bugs were fixed before any systems were shipped to customers.

Details of these bugs follow. Included is information about how the bug was detected, a hypothesis on why the bug eluded detection before first-pass chips were fabricated, and lessons learned from the detection and elimination of the bug.

1. One bug was found by an exerciser running on the second-pass RTL model. A cache line victim failed to write back on a B-cache index match because a bypass occurred at the same time. This bug existed only in 32-byte cache mode and B-cache speed configurations of 4, 5, and 6. This bug could have been found in the

first-pass model if this case had been generated pseudorandomly. Running many cases is crucial with a pseudorandom testing strategy. Given unlimited time and computation cycles, this bug might have been found earlier.

2. A second bug was caused by the B-cache read/write timing being off by one cycle. This bug could have caused multiple drivers to drive the data bus at one time. An assertion checker for this bug was in the RTL model, but the checker itself was not working properly. In the future, assertion checkers should be verified by causing the failure to occur and watching to see that it detects the case. In some cases, assertion checkers are written to flag an error for events that should never happen. Forcing an illegal situation to occur can be very difficult.

3. Another bug was found by an exerciser when a WRITE_BLOCK command was preceded by a single-cycle idle_BC signal assertion. This issue was directly related to a specific B-cache speed and was related to another system configuration restriction. This issue caused a restriction to be added, but the design was not changed.

4. If the B-cache sequencer is performing a bypass immediately after a command loads in the B-cache address file and a reference is coming down the S-cache pipe, the B-cache index could change in back-to-back cycles. The index should change only every other cycle. An assertion checker should have been written to test for this situation and make sure it never occurred.

5. The performance-monitoring logic that counted load merges was not counting these events correctly. This bug was not in the RTL model but only in the actual implementation. Possibly, more RTL-to-CHANGO comparisons needed to be run on this section of logic.

6. Because of an LDxL/STxC bug, an invalidate to a locked address was not detected as a hit against the LDxL address. As a result, an STxC passed when it should have failed. This bug could have been detected if a focused test had been written with very specific timing of a FILL and an LDxL hitting the S-cache in consecutive cycles. Gaining control of this interaction on the system bus was not possible, however, and random simulations were relied upon to achieve this case. This was a rare event in the random simulations, but parameters could have been adjusted to make this occur more often.

7. For one specific system configuration, a READ or FLUSH command sent by the system to the Alpha 21164 chip could cause the system to hang. For this to happen, three

specific events, all with very tight timing windows, needed to occur. We could have found this bug during simulation if we had emphasized this type of condition during the pseudorandom testing.

8. When responding to a command, the system had the option of asserting an error signal instead of its normal response. The error signal acted as an interrupt request to the Alpha 21164 chip. Under certain conditions, and for a narrow window of time, this error signal was not properly recognized. Testing of error conditions was a project goal but not a high priority compared to testing normal events. This bug could have been found earlier if additional error-mode tests had been run.

The above issues were fairly minor and all have been fixed in the version of the design that will be released to customers. The use of pseudorandom testing was very successful. Many major, complicated bugs were found over the course of the project that would never have been found using a focused effort. Because of the number of system configurations possible, a verification effort that consisted only of focused testing would have been impossible.

ACKNOWLEDGMENTS

The Alpha 21164 functional verification effort was performed by a team of engineers from the SEG microprocessor verification group. Members of this team included Homayoon Akhiani, David Asher, Darren Brown, Rick Calcagni, Erik DeBriae, Jim Ellis, Bill Feaster, Mariano Fernandez, Jim Huggins, Mike Kantrowitz, Ginger Lin, Chris Mikulis, Lisa Noack, Ray Ratchup, Carol Stolicny, Scott Taylor, and Jonathan White. The CCLI user interface would not have been possible without John Pierce. Walker Anderson provided quality guidance through all phases of the project. The Alpha Architecture Group RAX team (Matt Baddeley, Larry Camilli, Ed Freedman, Joe Rantala, Pravin Santiago, Lucy Tancredi, Steve Torchia), once again, provided and supported an effective verification tool. Lastly, the success of the project and the final quality of the Alpha 21164 chip logical design are as much a tribute to the work of the architecture and design teams as they are to the work of the verification team.

REFERENCES

1. J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor," Digital Technical Journal, vol. 7, no. 1 (1995, this issue): 119-135.

2. W. Bowhill et al., "Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU," Digital Technical

Journal, vol. 7, no. 1 (1995, this issue): 100-118.

3. W. Anderson, "Logical Verification of the NVAX CPU Chip Design," Digital Technical Journal, vol. 4, no. 3 (Summer 1992): 38-46.

4. A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator," IBM Systems Journal, vol. 30, no. 4 (1991): 527-538.

5. A. Ahi, G. Burroughs, A. Gore, S. LaMar, C-Y. Lin, and A. Wiemann, "Design Verification of the HP 9000 Series 700 PA-RISC Workstations," Hewlett-Packard Journal (August 1992): 34-42.

6. D. Wood, G. Gibson, and R. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," IEEE Design and Test of Computers (August 1990): 13-25.

BIOGRAPHIES

Michael Kantrowitz

A principal engineer, Mike Kantrowitz is currently leading the verification effort for a new Alpha microprocessor and developing new verification tools and methods. Prior to this project, Mike was co-leader of the 21164 chip verification, responsible for the instruction fetch and execute units. He has also contributed to the verification of the Mariah, NVAX+, 21064 floating-point unit, and FAVOR vector unit. Before joining Digital in 1988, Mike worked at Raytheon Company. He has a B.S.E.E. from Stevens Institute of Technology and an M.S.E.E. from Worcester Polytechnic Institute. Mike is a member of IEEE.

Lisa M. Noack

A principal engineer, Lisa Noack is currently co-leading the chip verification effort for a new Alpha microprocessor. Prior to this work, Lisa was a co-leader of the 21164 chip verification and was responsible for memory, cache, and system interface units. Lisa has also contributed to the verification of the NVAX+ and NEXMI chips and the PVN module and chip set. Before she joined Digital in 1989, Lisa was employed at Data General Corporation as a design engineer responsible for the system design of I/O subsystems and various gate array design projects. She earned her B.S. and M.S. degrees in computer engineering from Syracuse University.

TRADEMARKS

Digital is a trademark of Digital Equipment Corporation.