# Functional Verification of a Multiple-issue, Out-of-Order, Superscalar Alpha Processor— The DEC Alpha 21264 Microprocessor

Scott Taylor, Michael Quinn,
Darren Brown, Nathan Dohm, Scot Hildebrandt, James Huggins, Carl Ramey
Digital Equipment Corporation

## ABSTRACT

DIGITAL's Alpha 21264 processor is a highly out-of-order, superpipelined, superscalar implementation of the Alpha architecture, capable of a peak execution rate of six instructions per cycle and a sustainable rate of four per cycle. The 21264 also features a 500 MHz clock speed and a high-bandwidth system interface that channels up to 5.3 Gbytes/second of cache data and 2.6 Gbytes/second of main-memory data into the processor. Simulation-based functional verification was performed on the logic design using implementation-directed, pseudo-random exercisers, supplemented with implementation-specific, hand-generated tests. Extensive functional coverage analysis was performed to grade and direct the verification effort. The success of the verification effort was underscored by first prototype chips which were used to boot multiple operating systems across several different prototype systems.

## Keywords

Alpha, microprocessor, verification, validation, architecture, pseudo-random, coverage analysis, 21264

## 1. INTRODUCTION

The Alpha 21264 microprocessor is a highly out-of-order, superscalar implementation of the Alpha architecture. The CPU required a rigorous verification effort to ensure that there were no logical bugs. World-class performance dictated the use of many advanced architectural features, including:

- four-wide instruction fetch
- instruction cache line prediction
- branch prediction (local and global history)
- out-of-order speculative-issue execution pipelines (4 integer pipelines, 2 floating-point pipelines)
- two out-of-order memory references per cycle
- up to sixteen in-flight off-chip memory references

In addition, on-chip caches include a 64-KByte virtually-indexed/virtually-tagged two-way set predicting instruction cache and a 64-KByte virtually-indexed/physically-tagged two-way set associative data cache. Each cache is paired with its own 128-entry fully associative instruction address translation buffer, and the data cache also features cache-hit prediction logic. The chip's pin interface features separate system and second-level (L2) cache interfaces, with support for various CPU-system clock ratios, system configurations, and L2-cache RAM types [1].

Functional verification of the Alpha 21264 design was performed by a team of verification engineers organized as a separate skill-based group. The primary responsibility of this team was to detect any logical errors in the chip design which would result in incorrect operation or negatively impact performance. The correction of these logical errors was the responsibility of the chip architects. The detection and correction of timing, electrical, and physical design errors were separate efforts conducted by the chip implementation team.

The chip verification team primarily utilized simulation-based techniques, which in turn relied heavily upon pseudo-random test generation to improve the quality and efficiency of the verification effort. These techniques have been in use at DIGITAL for more than nine years and are also used elsewhere in the industry and in academia [2-6]. The complexity of the Alpha 21264 design required that significant advancements be made in this area in order to guarantee meeting the key goal of working first-pass parts. This paper describes those advancements within the context of the overall verification effort.

### 1.1 Goals

Conflicting factors such as increased processor complexity and reduced time-to-market goals have resulted in first-pass silicon being relied upon to extensively verify and debug system-related hardware and software. It has become a typical goal that first-pass silicon be capable of booting one or more operating systems across several different prototype platforms. Achieving this first-pass goal provides a more effective mechanism for exercising the processor to its design limits and thereby exposing bugs that would not be readily detected using simulation-based verification techniques. However, as these new bugs are encountered, the original simulation environment can be used to isolate the bug source and to verify the solution. All of this acts to directly support the key goal of second-pass



1-58113-049-x-98/0006/$3.50

silicon, which is to be bug-free so that these chips can be shipped to customers for use in revenue-producing systems.

# 2. MODELING METHODOLOGY

The Alpha 21264 microprocessor was modeled in several levels of abstraction. These abstractions included a performance model, an RTL/behavioral model, a gate-level model, and a three-state simulation model.

The RTL model was written using a DIGITAL proprietary modeling language. This cycle-based simulator contained a command-line interface that enabled the user to access internal processor signals and call subroutines internal to the model. The signal interface also made it possible to dump binary traces of internal signal activity to a file for later processing.

The new simulation engine enabled the Alpha Standard Reference Model (also called the Instruction Set Processor or "ISP"), which was written in C++, to be directly incorporated in the Alpha 21264's RTL model. Verification of the RTL model was often accomplished by comparing its state with that of this reference model. Because these simulation checks could now be done at run-time, rather than as a post-processing task, the speed and extent of the checking was greatly increased.

The simulation engine was also used as a basis for a gate-level simulator. This simulation was also cycle-based, and could be run in lockstep with the RTL simulation. Various signal states could then be compared between the two models; any discrepancies between them would trigger an error condition.

## 2.1 Pseudo-system Models

The verification team created several modules that communicated with the RTL model under test. These routines typically modeled external logic such as main memory, caches, and I/O devices; external agents such as other processors or system demons; and basic pin interface functions such as reset and clocking. Most of this functionality was collectively referred to as the "Xbox." The Xbox implemented all possible system configurations that the Alpha 21264 microprocessor supported. It acted as the interface between the simulation engine, the RTL code, and the user. The Xbox represented the system and drove the pins of the RTL model, providing inputs and checking certain outputs.

Other pseudo-system models were created as well. One example was a "shadow" model of the Alpha 21264's branch prediction logic. Since branch prediction was inherently transparent to the user, it was difficult to verify that the logic was functionally correct. The shadow model was an abstracted version of the RTL logic, and used the same interface signals. These interface values were driven to both the RTL model and the shadow model on a cycle-by-cycle basis. Routines within the shadow model compared the state of the two models and flagged any errors.

# 3. STRATEGY

The verification strategy employed multiple techniques to achieve full functional verification of the Alpha 21264 chip. The primary technique used pseudo-random exercisers coupled with coverage analysis. These exercisers generated pseudo-random instruction sequences, executed the sequences on a model that included both the 21264 model and the ISP reference model, and compared the results during the execution. A second technique used focused, hand-generated tests to cover specific areas of logic. Other methods consisted of model code reviews, box-level testing, and executing Alpha architectural test suites and benchmarks. Figure 1 shows the general flow for a single simulation.

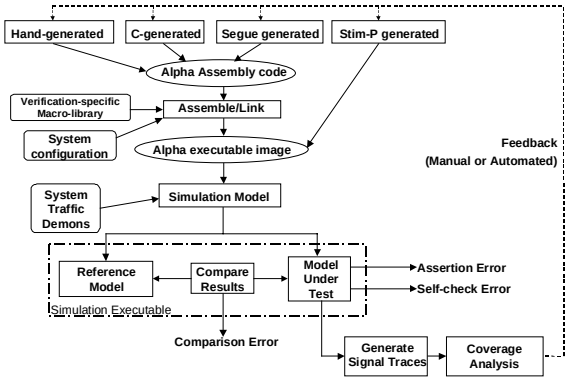


**Figure 1. Verification Test Environment**

This strategy was deployed in four parts: the pre-integration phase, the try-anything phase, the test-planning phase, and the structured completion phase. In the pre-integration phase, testing focused on individual functional units, or "boxes," of the design. These boxes were tested using transactors that drove all inputs and checked all outputs of the functional unit. This allowed for a smooth integration of the full-chip model. Once the full-chip model had been integrated, assembly-level code could be used to test the chip.

The main purpose of the try-anything phase was to exercise the majority of the design in a short period of time. Automated exercisers were the main source of stimulus during this part of the project. This allowed the verification team to spend most of its time debugging failures rather than writing tests. Existing Alpha architectural test suites and benchmarks were also executed to ensure base-level functionality. This debugging period produced an atmosphere of intensity and challenge. Everyone was required to interact constantly and identify problem sources, thus stabilizing the model quickly.

Once the design stabilized and the bug rate declined, 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 initial phases of the project allowed the verification team to gain insight into complex areas of the chip, leading to the creation of a better initial test plan. A set of joint test plan reviews were then held with the design teams for each section of the chip. These joint reviews ensured that the verification team did not miss important aspects of the design. Finally, the test plan provided a means for scheduling and prioritizing the rest of the verification effort.

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 pseudo-random exerciser with coverage analysis, or an assertion checker. As the verification of each section of the chip was completed, a review was held with the design and architecture teams to examine what was verified and how it was done. This review identified any verification coverage holes.

# 4. TEST STIMULUS GENERATION

## 4.1 Focused Testing

Verification of the 21264 relied heavily on pseudo-random test generation and, to a lesser extent, focused testing. About 300 focused tests were written during the verification effort. Additional post-silicon tests were added to increase manufacturing defect detection and isolate critical speed paths. Focused tests covered all major functional areas of the chip and provided repeatable, guaranteed coverage for regression testing.

A subset of the focused test suite was run as part of the model release procedure. For increased coverage the entire test suite was continuously run against the released model using random configurations. This test loop guaranteed that all model releases maintained consistent functionality, as well as providing additional verification coverage. The regression test loop was instrumental in maintaining gate-level and RTL model coherency for the implementation team.

Focused tests had advantages over pseudo-random tests not only in their directed areas of coverage, but in their self-checking abilities. Self-checking code aided in finding bugs in the reference model as well as in the RTL. Most of these tests were used as manufacturing patterns because of their ability to run at various system and L2 cache speed settings. Focused assembly-code tests were readable and directed at specific logic areas, so they were faster to debug.

## 4.2 Pseudo-random Testing

The complexity in the 21264 made pseudo-random testing critical. Defining the logical interactions that needed to be tested was extremely difficult in this chip. Creating these interactions by hand was even more difficult. The pseudo-random testing approach allowed the verification engineer to concentrate effort on defining areas of interest within the chip. Automated pseudo-random test programs called exercisers were created to test the complex interactions within those areas.

### *4.2.1 Pseudo-random Exercisers*

Several dozen different pseudo-random exercisers were used in the verification of the 21264. Most of these focused on specific architectural areas within the chip; others were more generalized. Some areas targeted explicitly were: control-flow, out-of-order instruction processing, superscalar structures, cache and memory transactions, multiprocessing and synchronization code, floating point, traps, illegal instructions, and Privileged Architecture Library (PAL mode).

Fundamentally, each exerciser operated the same way. First, the exerciser created pseudo-random assembly-language code. Next it simultaneously ran the code on the model under test and on the ISP reference model. Any divergence of data or instruction stream between the two models signaled an error.

The out-of-order feature of the 21264 required new testing methods. Code had to be generated in such a way that the chip's full out-of-order potential was tested. This involved tight control of data dependencies in the code, which were matched to the issue depth and superscalar width of the 21264. New tools were developed to combine multiple independent code streams into one test case allowing maximum instruction throughput on the chip while maintaining consistent out-of-order instruction execution.

Many of the exercisers used the test generation engine known as Stim-P. With Stim-P, the user controlled the probability of certain events or instructions rather than specifying the actual code sequence. Stim-P generated pseudo-random Alpha assembly code based on the user's configuration and its own understanding of the Alpha architecture. This allowed each user to "tune" an exerciser to focus on a specific area of the chip. For example, a verification engineer trying to test integer unit traps would have turned up the frequency of integer instructions relative to floating-point instructions and also maximized the probability of generating overflows on a given operation. Similarly, a memory-focused exerciser may have used a reduced number of trapping instructions and an increased probability of re-using a previously generated memory address for loads and stores.

SEGUE, a text generation and expansion tool, was also used to create pseudo-random code. Similar to Stim-P, variables were passed in to determine the focus of a particular test case. SEGUE, however, relied solely on the use of user-defined templates to generate a code stream. Given a basic building block, SEGUE filled in the template with appropriate instructions. An important application of the SEGUE method was to test the 21264's PAL (Privileged Architecture Library) mode. Since PAL mode imposed many restrictions on the content and order of the instruction stream, SEGUE was a natural fit for generating pseudo-random code which followed the specification's coding rules.

Some exercisers combined the Stim-P and SEGUE tools to create specialized test cases. Illegal instructions, for example, were exhaustively generated by SEGUE and run within Stim-P tests. This helped to expose bugs that involved several random events occurring around an illegal instruction. Other hybrid exercisers used these tools to check PAL mode instructions within random user code.

### *4.2.2 Random Configuration Selection*

Each test, whether pseudo-random or focused, made use of certain "configuration parameters" to describe both internal CPU state and the type of system that would be emulated for a given simulation. These parameters consisted of variables that could be weighted to make certain system and/or internal CPU events occur. Interesting "fixed configurations" could also be randomly chosen to better mimic a real system design.

Examples of parameters included single-bit error injection rate, external cache size and configuration, the ratio between the system clock and the CPU internal clock rate, external probe rates, floating point unit enable, branch prediction enable, page mapping method, and PALcode location. In all, there were over eight hundred parameters.

Once again, SEGUE scripts were utilized to create the command files that chose the various configuration parameter values. This was done during the compile stage of a test. This method insured that different configurations were chosen and tested for each simulation. This meant that any test, including focused tests, could be used as a regression test and still provide new verification coverage each time it was run.

Each test could override the randomization of any parameter, which allowed a test to consistently run in a particular mode if necessary. Configuration parameters were also loaded into the memory model to provide information accessible through assembly code regarding the configuration type.

#### *4.2.3 Demons*

Demons were pieces of model code that could query or modify internal or external states of the 21264 chip. These demons were all accessible via macrocode. The access method chosen allowed for quick and predictable demon operation but still allowed their use during test pattern generation time.

There were several varieties of demons. State-modifying demons were used to affect internal or external state of the processor—such as randomly injecting bit errors on fills, or asserting random interrupt requests. System traffic demons were used to simulate probe requests from other processors, such as Invalidate or Set-Dirty requests. There were also a large number of status/query demons, which performed such functions as printing user messages to a log file and counting the number of data cache hits/misses. Special demons were also created to reduce the size of exception flows. One example was a demon that directly looked up page mapping in a hash table structure rather than using macrocode to do the lookup in memory.

## 5. SIMULATION

Once the focused or pseudo-random test case had been generated, the test was compiled to produce one binary image. This image contained the test case along with the random configuration chosen and the PALcode. PALcode is a set of low-level firmware routines designed to handle events such as traps, interrupts, and page-table misses. This PALcode could be customized on a per-test basis.

At the beginning of the simulation, the binary image would be read by a simulation loader. This loader would then place the test throughout memory as directed, set up system configuration variables, and place PALcode and internal configuration state in memory.

After a test was loaded, the simulation would start. A PALcode reset handler was executed first. Information was read from the configuration and various internal registers were loaded with the specified values. The test was executed after the PALcode completed. During simulation, results were compared and any failures detected were reported to the user and stopped the simulation. Upon successful completion of the test, some post-processing steps were performed. Failing cases were saved to disk for debugging, while passing cases were generally discarded.

## 6. CORRECTNESS CHECKING

Several classes of mechanisms were used to ensure the RTL model was functioning correctly. Some focused tests used self-checking, hand-coded by the test author. This approach is time-consuming, prone to error, and not useful beyond a single focused test, so self-checking was kept to a minimum. More general techniques included assertion checkers, reference model comparisons, and memory coherency checkers.

### 6.1 Assertion Checkers

The new simulation engine utilized for Alpha 21264 verification provided many built-in assertion checkers for illegal signal states associated with its RTL modeling primitives, an example of which would be an invalid select combination on a multiplexer. Many other assertion checkers were added to the RTL code to check for illegal events such as an invalid transition in a state machine. These were added to the model by architects or verification engineers. RTL assertion checkers were easy to implement and prevented wasted simulation cycles by halting the simulation when an error occurred. Debugging information was also provided near the location of the error.

Assertion checkers were also included in the Xbox by verification engineers. These checkers tended to be more involved, and tracked the state of the machine. Examples include checkers for queue overruns and violation of the bus protocol.

Another class of assertion checkers was included in coverage analysis tests. The verification engineer would analyze a set of cross-products, and if any elements were illegal, an assertion checker was added. These checks were done as a post-processing step, but were able to provide useful debuging information, most importantly which cycle the violation occurred.

### 6.2 Reference Model Comparisons

A module was built to compare the ISP reference model with the RTL model in real time. Some comparisons were done at the end of a test, such as a full memory comparison. Most comparisons were done in real time; these include:

- Register values at instruction retire time
- Program counter at instruction retire or exception
- Memory ordering at the pins
- System lock flag at instruction retire
- Checks for success/failure of atomic operations (load_locked and store_conditional)

### 6.3 Memory Coherency Checkers

A mechanism was developed that could “snoop” through various levels of the cache hierarchy, including main memory, a second-level backside cache, the first-level data cache, buffers for data arriving on the chip, buffers for data being evicted from the chip, and the pipelines for fills and evicts. This enabled the verification team to check the flat memory state of the ISP against the complex cache hierarchy of the processor. For performance reasons, the entire memory space was not checked every cycle. Instead, selected areas of memory were checked at

specific coherency points such as cache block evictions. These checks were invaluable and found many bugs in the design, but were also difficult to maintain.

# 7. COVERAGE ANALYSIS

The most difficult question within functional verification has always been: "Is the verification effort complete?" One could run test after test and still not have the confidence that all areas of the design were fully tested. One method of answering this question involves watching the bug rate. While this provides some interesting data, watching it fall may just mean that other problem areas are going untested. A more successful method of answering this question is the use of coverage analysis. Coverage analysis plays an important role by providing visibility into the completeness of the verification effort. On the 21264 microprocessor, extensive coverage analysis of both the focused tests and pseudo-random exercisers was done.

Signal traces were collected during a simulation. This data was then used by trace analysis tools to collect the coverage data which was accumulated across multiple simulation runs. That 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 pseudo-random exercisers were exercising the chip. With this insight, pseudo-random exercisers were modified or new focused tests were created to improve the test coverage. Running pseudo-random exercisers with coverage analysis proved to be a very powerful technique in functional verification.

The coverage analysis done can be broken up into four major areas: state transitions, sequence, occurrence, and case.

## 7.1 State Transition Analysis

State transition analysis was used for more complex state machines-ones with dozens of states or arcs. A tool called CADET was used to describe the actual state diagram for the function. The tool then automatically generated the necessary coverage test to check all possible states and arcs. On the 21264 microprocessor, one such state machine checked this way was the instruction cache stream buffer control logic.

## 7.2 Sequence Analysis

CADET could also be used to check sequences of events. A set of possible events happening at time (t) can be "crossed" with a set of events happening at time (t+1). For example, it was verified that every type of system command heading out of the CPU was followed with every type of system command heading into the CPU.

These events were described within CADET graphically which reduced the time needed to create such coverage tests. These events were then organized in sets that described the cross-products. As the data was collected, it could be displayed in a matrix format. Figure 2 shows coverage data for cache blocks transitions from invalid to four possible valid states.

| | V | ~V | ~V | ~V | ~V | Time (t-1) |
|---|---|---|---|---|---|---|
| **Commands** | **~V** | **V** | **V/S** | **V/D** | **V/S/D** | **Time (t)** |
| **RdBlk** | . | ✓ | 7 | ✓ | 0 | |
| **RdBlkMod** | . | . | . | ✓ | 1 | |
| **Clean2Dirty** | . | . | . | . | . | |
| **Share2Dirty** | . | . | . | . | . | |
| **Inval2Dirty** | . | . | . | ✓ | . | |
| **Evict** | ✓ | . | . | . | . | |

V = Valid S = Shared D = Dirty

. = event cannot occur

✓ = more than 100 events of this type were seen

**Figure 2. Sample Cadet Coverage Matrix**

Illegal transitions (such as a data read causing a cache block to become invalid) can be tagged, such that any occurrence of the event would flag an error. Legal transitions increment the value at that matrix location; when the value reaches a pre-determined value (100 in the example), the value is considered "filled." In the example, a transition from invalid to valid-shared-dirty has not occurred. This was a legal, albeit rare, occurrence. The verification engineer could then take this information and tune an exerciser (or write a focused test) to fill in this coverage hole.

## 7.3 Occurrence Analysis

Coverage analysis doesn't always consist of time-related events. Many coverage issues were simple occurrences of "one-time" events. A CADET test might be written to check that a carry-out has been generated for every bit in an adder. Another would make sure each type of interrupt was taken.

## 7.4 Case Analysis

Case analysis consisted of collecting a set of statistics on each simulation run. Types of statistics included exerciser type, cycles simulated, issue rate, instruction types, cache hit/miss rate, specific system configuration and many more. This data was kept in database format using a tool called STATS. This same tool could then generate reports, for later analysis, for specific date ranges and/or exerciser types to gauge an exerciser's verification effectiveness.

# 8. BUG TRENDS

Throughout the 21264 verification effort many bugs were logged and tracked, most before first-pass parts were manufactured. Figure 3 shows the bugs found as a function of time for the duration of the project as given by a bug-tracking system. 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 early or tested at the box level before changes were integrated into the full model, the action tracking system does not represent all the issues raised. However, it is interesting to look at the trends presented by the data.

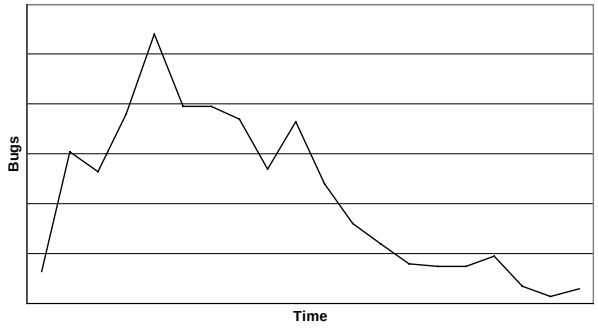


**Figure 3. Bug Detection as a Function of Time**

The first trend to consider is the effectiveness of the pseudo-random and focused efforts. As shown in Figure 4, more than three-quarters of the bugs were found using pseudo-random techniques. Errors found by the focused testing were typically in sections of the design that had sparse pseudo-random test coverage or where self-checking tests could be easily implemented.

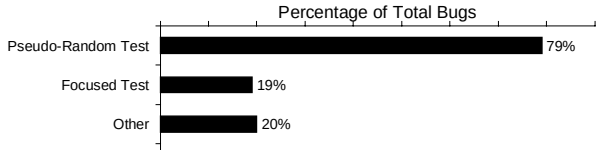


**Figure 4. Effectiveness of Test Category**

Bugs in the design had been introduced in a variety of ways. Figure 5 shows the breakdown of the causes of bugs, of which the majority occurred while implementing the architectural specifications for the project.

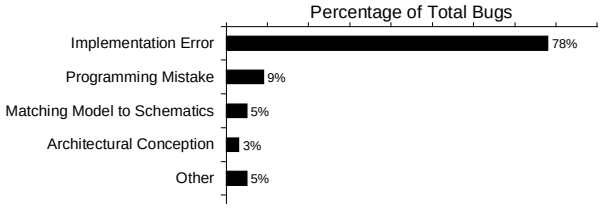


**Figure 5. Introduction of Bugs**

Figure 6 shows the various detection mechanisms that were used to find bugs. Assertion checkers placed in the design to quickly detect when something is not correct were the most successful. However, when viewed collectively, nearly 44% of all bugs were found by reference model comparison, which includes register, PC, and memory state comparisons.

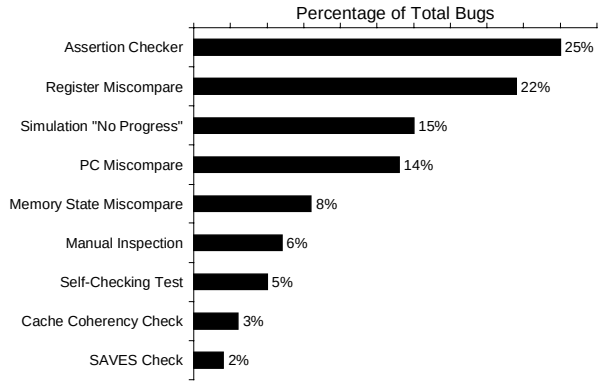


**Figure 6. Bug Detection Mechanisms**

# 9. RESULTS AND CONCLUSIONS

As of December, 1997, eight logical bugs were found in the first-pass prototype Alpha 21264 CPU design. None of these bugs prevented first-pass silicon from being used to boot multiple operating systems on several different prototype platforms. Two of the bugs were found while debugging test patterns on the tester; the others were all found during initial prototype system debugging. These bugs escaped detection via functional verification of the RTL model for the following reasons:

- Three of the bugs were due to the schematics not matching the RTL model, and comparisons between the RTL model and the structural model (generated from the schematics) did not detect the mismatch.
- One bug was due to electrical issues.
- One bug was due to insufficient randomization of the initial state of the RTL model during simulation.
- Two of the bugs were actual logical problems in the RTL model. In both cases checkers had been added to detect that exact problem, however the checkers were either incorrectly or insufficiently implemented.
- The last bug was also a logical problem in the RTL model. However, it involved an extremely complex combination of cross-products that were unlikely to be hit via pseudo-random testing, or even thought of in terms of a focused test. In a very real sense this was exactly the type of bug that first-pass silicon was intended to help find.

All of the above issues have been fixed in the version of the design that will be released to customers. They were all viewed as being relatively minor bugs, with the possible exception of the last one (which again was exactly the type of bug that was considered most likely to be found during first-pass silicon debugging). The success of the Alpha 21264 verification effort once again underscores the value and effectiveness of pseudo-random based testing.

## 10. FOR FURTHER INFORMATION

The authors can be contacted at the following address, c/o Digital Equipment Corporation:

334 South Street
Shrewsbury, MA 01545-4112

Email inquiries may be directed to Scott.Taylor@digital.com or Quinn@ad.enet.dec.com.

## 11. REFERENCES

[1] Linley Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report* (October 28, 1996): 11-16.

[2] Mike Kantrowitz and Lisa Noack, "Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor — the Alpha 21164 CPU Chip," *Digital Technical Journal*, vol. 7, no. 1 (1995): 136-143.

[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.