I'm Done Simulating; Now What?
Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha microprocessor

Michael Kantrowitz Lisa M. Noack

Digital Equipment Corporation
77 Reed Rd
Hudson MA 01749

ABSTRACT

Digital's Alpha-based DECchip 21164 processor was verified extensively prior to fabrication of silicon. This simulation-based verification effort used implementation-directed, pseudorandom exercisers which were supplemented with implementation-specific, hand-generated tests. Special emphasis was placed on the tasks of checking for correct operation and functional coverage analysis. Coverage analysis shows where testing is incomplete, under the assumption that untested logic often contains bugs. Correctness checkers are various mechanisms (both during and after simulation) that monitor a test to determine if it was successful. This paper details the coverage analysis and correctness checking techniques that were used. We show how our methodology and its implementation was successful, and we discuss the reasons why this methodology allowed several minor bugs to escape detection until the first prototype systems were available. These bugs were corrected before any chips were shipped to customers.

OVERVIEW

The DECchip 21164 CPU chip is a quad-issue, super-scalar implementation of the Alpha architecture which required a rigorous verification effort to ensure that there were no logical bugs. World-class performance dictated the use of many advanced micro-architectural features, such as a virtual instruction cache with seven-bit Address Space Numbers, a dual-read-ported data cache, out-of-order instruction completion, on-chip three-way set-associative write-back second-level cache, module-level cache control, branch prediction, demand-paged memory management unit, write buffer unit, miss-address file unit, and a complicated Bus Interface Unit with support for various CPU-system clock ratios, system configurations, and module-level cache parameters. [1]

Increasingly, functional verification efforts are relying on pseudorandom test generation to improve the quality of functional coverage. These techniques have been in use at Digital for more than seven years and are also used elsewhere in the industry and in academia.[2-5] However, the heavy use of pseudorandom testing increases the need for new ways of determining whether the test passed or failed, and new ways of determining exactly what portion of the design the test actually exercised. This paper discusses the correctness checking and coverage analysis mechanisms used by the DECchip 21164 verification team to ensure adequate functional coverage using pseudorandom test generators.

VERIFICATION PROCESS

All verification was done using the process flow depicted in Figure 1. The simulation environment consisted of a register transfer level (RTL) representation of the DECchip 21164 itself, plus a behavioral system model which provided a memory interface and could also mimic the behavior of other processors or I/O devices. This allowed the verification tests to be actual Alpha executable code, instead of needing to apply ones and zeros to the pins of the chip. The system model conformed to the constraints of the Alpha architecture, and was configurable to allow every possible system configuration and mode setting of the DECchip 21164 to be exercised.

The majority of stimulus applied to test the DECchip 21164 was created through pseudorandom methods. 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. Six different pseudorandom exercisers were used on the DECchip 21164 project. One was a general-purpose exerciser that provided coverage at an architectural level. Each of the other five targeted a specific section of the chip in a pseudorandom way.

Test stimulus (either random or focused) was applied to both the Design Under Test and to a reference model. Many different types of mechanisms were used to determine whether the test stimulus executed correctly. These included comparing test results between the Design Under Test and the reference model and enhancing the RTL model with a wide variety of assertion checkers that continuously monitored the model while a test was simulating. Using coverage analysis to estimate how much of the design had been verified was also an important part of the verification flow. Several different techniques for coverage analysis were used. When analyzing the coverage of a particular section of the design, any or all of these techniques were used, as appropriate for that section. The following sections describe the correctness checking and coverage analysis pieces of the verification flow.

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

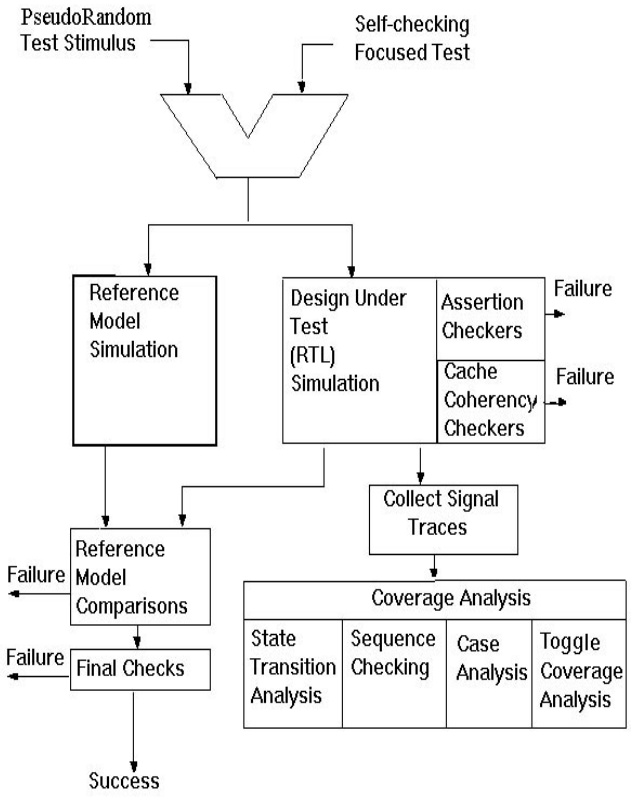

Figure 1: Verification Flow

CORRECTNESS CHECKING

The success or failure of traditional hand-generated focused tests is typically determined by the test itself. The test checks its own answer against a set of pre-determined expected results.

However, with pseudorandom test generation, this self-checking approach did not work; it is very difficult to create a self-checking pseudorandom test. Instead, several alternate mechanisms were used for checking whether the model behaved correctly. The effectiveness of any specific checking mechanism depended on the type of test being run and the type of bug that might occur. Using as many different checking mechanisms as possible, we were able to detect a broad range of bugs. One specific goal was to detect a bug as close as possible to the place where the fault actually occurred, in order to simplify the debugging process.

Reference Model Comparisons

When pseudorandom tests are used, the state of the machine upon test completion is not known, thus it is hard to determine if the test executed successfully. For these cases, we relied heavily on comparisons against a reference model. Both the reference model and the design-under-test supported a full system environment, including not only the CPU chip, but a memory interface and I/O space as well. This allowed us to execute the same test stimulus on both models, and expect the same results.

Ideally, a reference model needs to be fast, correct, and represent all the details of the design. The reference we chose emphasized speed and correctness over detail. It represented a high-level abstraction of the Alpha architecture, written in the C language. The model represented all features visible to software, including the full Alpha instruction set and support for both memory and I/O space. It did not represent internal design details. In particular, it did not represent pipeline stages, parallel functional units, or caches. Representing a higher abstraction level allowed us to produce a reference model that contained very few bugs and was able to execute over 100 times the speed of the detailed RTL model.

The reference model enabled several types of correctness checks. The simplest of these was an end-of-test state comparison. When a pseudorandom test completed, the contents of all memory locations that were accessed during the test, as well as the final state of the integer and floating-point register files, were dumped to a file. These dump files were compared and any differences were flagged for further investigation.

This end-state comparison was of limited usefulness for long tests. Intermediate results may be overwritten and problems with them may never be known. Even if an error is detected, the source of the error may be far away from the detection point at the end of the test. Comparing results between the two models at intermediate points in a test execution, and not waiting until the test completes, can solve both of these problems. However, since our reference model did not exactly match the timing of the design-under-test, these intermediate comparisons were not easily implemented.

Both models accurately represented the Alpha architecture, and only valid architectural comparisons could be made. The additional comparisons we made were checking the PC flow and writes to the integer and floating-point registers. The PC flow immediately signaled any problem with control-flow instructions, while the register write comparison caught problems with the data manipulation instructions. In addition to checking internal state, the memory image was also compared at intermediate points during the test execution. This was complicated by the high level of buffering in the memory subsystem and the on-chip write-back cache of the DECchip 21164. Comparing intermediate memory required monitoring the state of all the internal queues and constructing a consistent memory image that could be compared with the reference model.

The Alpha architecture allows for the generation of unpredictable values under certain circumstances. Since the reference model was not an exact copy of the design-under-test, it could produce a different unpredictable value. To complicate this, unpredictable values could propagate to other registers, making comparison against the reference machine difficult. For example, when an arithmetic trap occurs, the destination register of the instruction which caused the trap may have an unpredictable value. To complicate this further, arithmetic traps are imprecise, meaning they might not be reported with the exact PC that caused them. 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 allowable mismatches between the two models.

Assertion Checkers

Assertion checkers are segments of code added to a model to check that various properties or rules of design behavior are not violated. 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 to choose an unused input. More complex assertion checkers require 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. In all cases, the assertion checkers can only detect a problem after the test has stimulated a particular condition. Their primary purpose is to increase visibility into what a test is doing. The DECchip 21164 verification effort used two categories of assertion checkers. The first was built-in checkers, that were part of the RTL model itself. The second was post-processing checkers that evaluated trace files representing various signal transitions.

The RTL model of the DECchip 21164 was augmented with a wide variety of built-in checkers. The team continually added new checkers to the model, since this was a very effective bug-finding mechanism. The advantage of built-in checkers is that they are always active and monitoring behavior for every cycle that is simulated. If one person on the team adds an assertion checker to the model, everyone else who uses that model will be using that assertion checker as well. Thus, for a large team, the built-in assertion checkers provided a huge amount of added leverage. Many times, an assertion checker caught a bug while the model was being run by someone focusing on a totally different area from which the original writer of the assertion checker was focusing on. An additional benefit of built-in assertion checkers was their ability to detect a bug very quickly and halt the simulation immediately. This simplified the debugging effort immensely.

The disadvantage of built-in assertion checkers is that they slow down the simulation speed. For the majority of checkers, this slow down is negligible, and their bug-finding payback is well worth the impact. However, in cases of particularly complex checkers, the performance impact was unacceptable. In these cases, the checkers were implemented separate from the model, as a post-processing step. While the model is simulating, the only impact is the additional I/O due to tracing the state of internal signals and writing them out to disk. The specific signals to trace were selected based on the particular postprocessing to be done. After simulation, an optional, per-test post-processing step would read the signal trace data and determine whether any of the various assertions were violated. In addition to not impacting simulation performance, it was easier to create more complicated assertion checkers using the post-processing technique. The signal trace file provided information about the future and past state of desired signals. This simplified creating assertion checkers without having to explicitly provide history queues and other complicated data structures inside the model. 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 RTL model. The behavior of the abstract algorithm was compared with the behavior of the model itself.

A specialized form of built-in assertion checker is a cache coherency checker. The DECchip 21164 system supported three levels of caching; a first-level data cache, a second-level, on-chip, combined instruction/data secondary cache, and a third level, off-chip combined backup cache. Each cache was defined to be a subset of the next cache in the hierarchy, complicated by the second-level and third-level caches following a write-back protocol. At regular intervals during a simulation, the cache coherency checkers would be activated, to ensure that the coherency rules were not being violated. This checker alone caught a significant percentage of all bugs.

Self-Checking Tests

The DECchip 21164 verification effort did use some focused, hand-crafted tests which checked their own result against a pre-stored expected result. This was useful for the areas in which the reference model did not accurately match the design-under-test. In particular, performance-enhancing features like bypasses and multiple-issue logic were verified via self-checking tests. As mentioned above, these performance-related features were not included in the reference model in order to keep it simple. Using the cycle counter built into the DECchip 21164, exact timings could be checked and verified against the expected timings. Self-checking tests were also useful for running in an environment where the reference model was not available. For example, when testing prototype hardware, the self-checking focused tests were re-used as diagnostic tests.

When run on the simulation model, though, even a self-checking test used the reference model and assertion check mechanisms. These allowed many more bugs to be detected than the self-check itself could detect.

Figure 2 shows the various detection mechanisms used by the DECchip 21164 verification effort. As can be seen, the assertion checkers were the most effective techniques.

COVERAGE ANALYSIS

The use of pseudorandom testing was highly effective and more productive than creating hand-generated tests. With the correctness checking problem solved, the next major issue was determining what the tests were actually doing. We were using targeted-random testing, so we knew the general areas that the tests were exercising, but we needed more detail on what was being covered. To help with this, extensive coverage analysis was done, mostly as a post-processing step.

During the simulation of the RTL model, a trace of the behavior of various signals was written to disk, in the same way we obtained trace files for assertion checkers. In many cases, coverage analysis and post-processing assertion checking were combined into one step.

Assertion Checkers	34%
Cache Coherency Checkers	9%
Reference Model Comparison	
Register File Trace Compare	8%
Memory State Compare	7%
End-of-Run State Compare	6%
PC Trace Compare	4%
Self-Checking Test	11%
Manual Inspection of Simulation Output	7%
Simulation hang	6%
Other	8%

Figure 2: Effectiveness of Bug Detection Mechanisms

Several different coverage analysis techniques were used. For standard types of coverage checking, a library of analysis routines automated the process significantly. One example where this was possible involved analyzing the coverage of state machines.

State Transition Analysis

Some state machines in the DECchip 21164 were represented as PLA structures. For this case, the PLA representation was used to determine the valid combinations of events that could occur, referred to as a minterm. A minterm consisted of a current state and all active input signals. The following shows an excerpt of a what PLA definition might look like:

```
cs_idle, bus_req_h / ns_bus_req;
cs_bus_req, bus_ack_h / ns_bus_ack;
cs_bus_req, ^bus_ack_h / ns_bus_req, bus_req_h;
```

The tool used to convert the PLA representation to a coverage analysis test was limited in that it would only check the number of times a minterm occurred. This limitation was augmented by other checking mechanisms such as assertion checkers in the model that assured that only a single minterm was asserted at one time.

State transition analysis was also used in areas where the logic was not explicitly implemented as a state machine, but its functionality could be represented by an abstract model of a state machine. This abstract model could be checked for state transition coverage. One area where this concept was applied was in the system and cache interface logic. Various pieces of logic interacted together when processing hits and misses on the Victim Address File (VAF). Rather than performing coverage analysis on each section of logic individually, the choice was made to treat this logic as a single entity and model it as an abstract representation.

Coverage analysis was performed on the abstract model to determine the events that our pseudorandom tests were covering and areas where coverage was inadequate. Since the system and cache interface of the DECchip 21164 is highly programmable, this section of logic was particularly difficult to cover fully. By performing this extensive coverage analysis, we understood the areas that were not well tested, and could target the most important of these for additional testing.

A smaller example of this was an analysis of the state of internal cache blocks vs. commands to the cache. This analysis resulted in the table shown in figure 3:

Cache Block State

Commands	~V	V	V/S	V/D	V/S/D
Nop	.	.	.	.	.
Flti Pt. Load	*	*	.	*	*
Invalidate	*	*	*	*	*
Set Shared	*	*	4	.	.
Read	*	*	.	*	.
Rd Dirty	*	*	*	*	*
Rd Dirty Inv	*	*	*	*	*

V = valid S = shared D = Dirty
. = event cannot occur
* = more than 100 events of this type were seen

Figure 3: Example of a Coverage Analysis Matrix

When this table was generated, an error message was also generated because a Set Shared should not have been issued to a block with V/S status and 4 occurrences of this type were seen. This triggered us to investigate why this illegal combination was happening and a design bug was found.

In the case above, cross-products were based on 2 events occurring with respect to time. However, in the analyses for the DECchip 21164, there were often cases where checking was needed for many events at the same time. For example, the internal trapping logic of the chip was checked by creating a

coverage checking test to see which combinations of traps were being generated at the same time. By looking at the number of times certain traps occurred within the same window (time proximity) of other traps it could be determined whether the testing of the trap logic was sufficient.

When checking coverage on events that can all occur in the same time window, the question raised is "what is enough?" Obviously, for the trap logic case, the first level of having each trap asserted individually was needed. Next would be having each trap asserted with every other trap for the second-level cross-product. But what about the third level where all combinations of any three traps occur? Is this complexity needed? Depending on the number of traps, this number could be small or very large. For the DECchip 21164, our goal was to attain good 1st and 2nd level coverage. Additional levels may have been analyzed to determine what combinations were being generated but complete multi-level coverage was not a goal.

Sequence Checking

Another way that signal traces were utilized was to look for sequences of events in a particular window of time. Post processing tests could be created to look for any combination of events be they state transitions or single signal assertions. Bus interfaces, interrupt assertions, traps and seemingly unrelated events are particularly interesting to look at using this method.

On the DECchip 21164, event sequence checking was used to ensure that the transactor, stimulating the system bus, was fully randomizing events. For example, using the DECchip 21164, systems can acknowledge command transactions with variable timing. Events were described to check that the CACK signal was being asserted at various times. Figure was produced from the post processing traces:

CACK Intervals

	Total
CACK at 1 CLK	0
CACK at 2 CLK	2
CACK at 3 CLK	189
CACK at 4 CLK	271
CACK at 5 CLK	234
CACK at 6 CLK	199
CACK at 7 CLK	199
CACK at 8 CLK	122
CACK at 9 CLK	90
CACK at >9 CLK	266
Total	*

Figure 4: Example of Sequence Checking Analysis

From this output, it was immediately seen that CACK at 1 CLK was an event that was not occurring. Further analysis of why this was not happening triggered the team to choose parameter settings for the pseudorandom methods that would stimulate this event.

Case Analysis

While a model was executing, information was stored about the occurrence of simple events. For example, a record was kept on the number of times that four instructions issued simultaneously, the number of times the translation buffers filled up, or the number of times stalls occurred. Since the configuration in which the chip operated was randomized, a record was also kept about the configuration information such as the Bcache size and speed selected, the system interface options and timing, etc. At the end of every model run, this information was stored to a database which allowed collecting statistics across multiple runs.

Case analysis was used on the DECchip 21164 in a similar way to sequence checking. The occurrence of an event could be determined by looking at the entries in the database. Matrices could be created to show which combinations of events had or had not occurred over the course of all the simulations. For example, on the DECchip 21164 the CPU/SYS clock ratio and the secondary cache block size were programmable. Figure 5 is an example showing whether or not all combinations of these events occurred.

Secondary Cache Size
(in bytes)

		32	64
CPU/Sys Clock Ratio	3	0	13784
	4	0	98341
	5	0	14387
	6	650	28374
	7	787	71843
	8	324	32847
	9	92992	17834
	10	2834	39843
	11	12833	18745
	12	18324	18763
	13	1433	81736
	14	2	13498
	15	0	18327

Figure 5: Example of Coverage Case Analysis

A table like the above, would have indicated that there was a nice distribution of CPU/Sys clock ratios vs. block size when the block size was 64. However, for the 32 byte block size, systems with a CPU/Sys clock ratio or 3, 4, 5, and 15 were not being chosen. A table like this would have triggered the verification engineer to look at the scripts that chose these parameters to find

out why these ratios were not being chosen correctly. Usually, when scripts would be changed to exercise previously unexercised events, additional bugs would be uncovered.

Toggle Coverage

The simulator used in the DECchip 21164 effort was capable of giving a list of signals that were or were not toggling for a given simulation. A toggled signal was one in which a transition from a 0-logic level to a 1-logic level or a 1-logic level to a 0-logic level was detected. Toggle coverage could indicate whether signals were being wiggled, but it did not give a good indication of whether the logic in that section was actually being functionally used.

We utilized toggle coverage only at a gross level on the DECchip 21164. Toggle coverage was checked for various sections within the chip to determine whether or not major areas of the chip were being stimulated. Lists of signals that did not toggle were checked to see whether any patterns emerged or major areas of functionality were not being covered. This sometimes pointed out areas that needed to be stimulated further.

Fault Simulation for Functional Coverage

Fault simulation was not used for functional verification of the DECchip 21164. Fault simulation is very compute intensive, and it targets faults introduced during the manufacturing process, not bugs introduced during the design process. The typical stuck-at fault model is not a useful model of design bugs. For these reasons, we did not use fault simulation during the verification phase of the project.

ESCAPES

Using the above techniques, the DECchip 21164 verification effort was highly successful. First-pass silicon booted the operating system and ran extensive diagnostics and user applications. Even so, we discovered several bugs that escaped our efforts to find them. Examining some of these shows areas where improvements are necessary.

Three bugs were related to bypass mechanisms, where the normal data flow was skipped under very specific timing conditions. Although the three bypasses were unrelated to each other, and in different sections of the chip, it does indicate that our coverage of these bypass conditions was not sufficient. Had we specifically looked for bypass-related coverage, we would have noticed this. To complicate matters, one of these bugs existed only in 32-byte cache mode and B-cache speed configurations of 4, 5, and 6. This indicates that multi-level event coverage is necessary for finding these verification holes.

One bug caused the Bcache read/write timing to be off by one cycle. This was the type of thing we targeted assertion checkers at, and in fact an assertion checker existed to look for this. However, the assertion checker itself was not working properly, thus allowing the bug to evade detection.

CONCLUSIONS

Pseudorandom test generation for design verification has significant advantages over hand-generated focused tests. To realize its full potential, though, the issues of correctness checking and coverage analysis must be addressed. The DECchip 21164 verification effort developed many different techniques for addressing these issues. Selecting the appropriate technique to use for specific areas of the design required good engineering judgment. The verification effort was highly successful, and many bugs were discovered prior to first-pass silicon. Nevertheless, there is still room for improvement in the verification methodology.

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): 119-135.

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

3. 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.

4. 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.

5. 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.