Large-Scale Hardware Simulation: Modeling and Verification Strategies

Douglas W. Clark [1]
Digital Equipment Corporation

June 1990

to appear in the Proceedings of the 25th Anniversary Symposium,
Computer Science Department, Carnegie-Mellon University, Sept. 1990

[1]on leave, 1990-91, at Aiken Computation Lab, Harvard University, Cambridge MA 02138, doug@das.harvard.edu

Abstract

Simulation is a critically important phase of modern computer hardware design. Lacking practical formal methods for proving complex systems correct, designers must run tests on simulation models to demonstrate correctness before chips and boards are fabricated. Using a large-scale model, such as an entire CPU, at a low level of representation, such as the logic gate level, is an efficient way to uncover the inevitable bugs in a complex design. Testing efforts should aim at finding these bugs rather than passing a fixed set of tests. Random generation of automatically checked tests effectively applies computing power to the debugging process. Simulation "demons," which run alongside a test, help to find difficult bugs in the interactions of subsystems. The methodological ideas discussed in this paper have been used in several large projects at Digital.

1 Introduction

Simulation is an indispensable technique in modern computer design. In the days of computers made entirely from off-the-shelf small- and medium-scale integrated circuits it was possible to build a hardware prototype and "debug it into existence," but nowadays the use of highly integrated custom and semi-custom components makes this practically impossible. These components severely constrain hardware prototype debugging because it typically takes a long, and sometimes very long, time to get a new version of a part once a bug is discovered. Repeated re-fabrication of parts can delay a project beyond its economic justification. It is therefore critically important to assure the correctness of complex integrated-circuit designs *before* physical fabrication. Simulation makes this possible.

In this paper I will present a set of opinions and recommendations based on Digital's experience with several large VAX system developments. Methodological recommendations would ideally be validated by controlled scientific experiment: one could imagine setting up two independent teams with the same charter—design a computer system with specified cost, performance, development schedule—but with different simulation methodologies. Unfortunately, in the real world, with real constraints, such an experiment would not be practical. So my views here come not from any systematic evaluation of competing methodologies, but rather from real-world experience.

My subject in this paper is assuring the logical or functional correctness of a newly-designed computer system's hardware. I will not address the companion problem of *timing verification*, for which modern techniques (e.g., [10]) can guarantee correctness. No such guarantee is (as yet) possible in establishing the logical correctness of a complex system, although some progress has been made with simple designs (e.g., [2, 8]). So the term *simulation* in this paper will mean logical simulation only.

There are two fundamental challenges to the effective simulation of complex computer hardware. First is the challenge of speed: simulations are orders of magnitude slower than real hardware, so it is extremely important for simulation to be efficient. Second is the challenge of correctness: since formal methods are not yet practical for verifying the correctness of an entire computer system, some testing scheme must be used to verify, or attempt to verify, the design. To deal with the twin challenges of speed and correctness, I advocate in this paper two basic methodological approaches: first, the use of large-scale models using a detailed representation of the logic; and

second, a testing strategy that is organized around bugs found rather than tests passed.

The next section of this paper argues that computer simulations should use large models—for example, of an entire CPU or even an entire system—at a low level of representation—for example, the level of individual gates and latches. This approach might seem to exacerbate rather than mitigate the problem of simulation speed, but I will try to show that it is the most efficient use of the available resources: computer power, designers' time, and project schedule.

Section 3 considers two testing strategies that attempt to show the correctness of a design. The traditional method, which I call *test-centered*, relies on a fixed list of tests that exercise the design thoroughly. I will argue that this is the wrong way to test a model and advocate instead a more flexible *bug-centered* strategy, which focuses on design bugs and their removal.

In Section 4 several ideas concerning random testing are presented, including the use of simulation *demons*. The conclusion summarizes this paper's recommendations.

2 Levels of Modeling and Representation

In computer system design there is a natural hierarchy determined by physical boundaries: the system may be composed of several cabinets, each of which contains printed-wiring boards, on which sit a number of integrated-circuit chips. Complex chips will contain further levels of hierarchy. The physical (and often logical) hierarchy is a convenient and natural way to organize the design work and the implementation. Specifications and models can be created at a high level, then decomposed into smaller units and parcelled out to the design team for further decomposition or for detailed logic design. At higher levels, descriptions can have various degrees of precision and formality, but at the bottom level—the level communicated to the factory—the requirement is a rigid and exact definition of the actual hardware in all its details. Computer-Aided Design (CAD) tools integrate elements from various levels in the hierarchy for timing analysis and logical simulation.

It is common practice to carry this hierarchical viewpoint into the effort of simulating a system design. In the specification phase of a project, the designers write behavioral models of the hardware for which they were responsible. (A *behavioral* model is a simulatable representation whose input-

output behavior mimics that of the ultimate hardware, but within which hardware detail is suppressed.) At each hierarchical level, these models might be expressed as interconnections of lower-level components, on down to the level of actual circuits. The simulation objective would then be to show that each decomposition was correct—that each component was logically equivalent to its implementation at the next lower hierarchical level. For example, input-output test patterns for a VLSI chip could be captured from its behavioral model and used to test the detailed hardware model as it was developed. A low-level description of one chip could be simulated with behavioral descriptions of other chips in a "mixed-mode" simulation [9, 11, 13]. Execution efficiency might be obtained through independent simulation of parts of the design, driven either by captured patterns, or by direct simulation in the mixed-mode model, since behavioral models are faster than gate-level models as a rule.

I believe that this top-down, hierarchical approach is the wrong way to simulate computer systems. I advocate instead simulating a single integrated large-scale model based on a complete detailed representation of the logic. The most familiar and still quite widely used representation level is that of logic gates and latches, but computer designers increasingly use various kinds of automatic sythesis methods that elevate somewhat the level of logical description [14]. I will use the phrase "gate level" to mean the lowest level of description used by the logic designers, which should be the *highest* level that can convincingly be shown to be logically equivalent to the circuits actually fabricated.

The hierarchical approach requires extra designer time, exposes the design to late discovery of subtle bugs, and hence can delay the project schedule. Before discussing these effects, let me point out three characteristics of most (industrial) development environments that strongly affect methodological decisions:

1. *designer time* is the single most precious resource in the entire effort;
2. *computational resources*—compute cycles, memory, disks—are plentiful, relative to designer time; and
3. *project schedule* is the most important constraint.

These factors mean that project managers should be willing to make tradeoffs of computing resources for designer resources, and that they should want to use calendar time as efficiently as possible.

The first problem with the top-down, hierarchical, mixed-mode approach is that engineers must create and maintain behavioral models with detailed interface specifications, in addition to doing the actual logic design. Development projects should instead substitute computer power for this extra design effort: gate-level models (or the equivalent) should be the only simulatable description of the design. This allows logic design to begin sooner and frees the designer from the responsibility of maintaining two or more models of the same piece of logic.

Another problem with the hierarchical approach is the difficulty of independently simulating separate parts. Unless there is an accurate set of good input-output patterns captured from a higher-level model, the designer must somehow generate inputs and check outputs of the designer's part of the logic. At many levels these "test vectors" can be quite inscrutable and very large. Furthermore, it is easy to see how two designers of interacting pieces of logic might have subtle differences in their understandings of the interface, each believing their design to be correct. An integrated behavioral model can take care of these difficulties only if it is exquisitely detailed: more time, more effort!

A third problem with this approach is accuracy: how can the designer be sure that the logic matches its behavioral description in every detail? And a fourth problem is the sacrifice of partitioning flexibility, about which more below.

Gate-level models of the entire CPU or system address all of these problems. By "system" I mean a substantial collection of hardware: my own group has used models that include multiple CPUs, main memory boards and buses, I/O adapters, I/O buses, simplified models of I/O devices, and even a console interface. Our goal has been to simulate together all of the new hardware the group is designing.

Gate-level models are derived directly from the designers' own specification of the logic (remember that this is sometimes in the form of Boolean equations or other abstract descriptions—one need not draw actual logic gates for everything). Hence there is no possibility of the simulation model disagreeing with the actual design. With multiple-model methods there is always this risk. "If you want two things to be the same," observed one of our engineers, "then only have one thing."

But what about the problem of speed? It is certainly true that a large-scale, gate-level model will run much more slowly than an abstract behavioral model. When will this be a problem? A simulation model is fast enough when it is turning up bugs at least as fast as they can be fixed by the design-

ers. Conversely, a model is too slow only when it becomes the bottleneck in the debugging process. But when during the project is this likely to happen? Digital's VAX experience [3, 4] strongly suggests that it is likely to happen when the bugs are subtle and difficult, requiring complicated stimulus, intensive subsystem interaction, and the simultaneous occurrence of unusual events. And it is precisely then that the big, detailed model is most needed, for only it expresses the exact logical behavior of the system under these extreme conditions.

Gate-level simulation of a large-scale model can be thought of as a brute-force substitution of processing power and main memory capacity for designers' brain power and time [15]. In the industrial environment it is usually easier to apply more computer power than more brain power to this problem.

Partitioning flexibility is lost to some degree in the hierarchical approach. Early specification of the behavior and interfaces of the parts makes it relatively more cumbersome to change the location of some piece of functionality. The brute approach of a single gate-level model allows more flexibility. Not only can changes be made more quickly, but knowing in advance that one will simulate (and timing-verify) this way allows the designers to adopt a looser attitude toward interface definitions, which in turn enables them to start the detailed design work earlier [15]. (Of course, more formal up-front interface design can improve the initial quality of the interface; this benefit is outweighed in my view by the disadvantages of this approach.)

Partitioning flexibility is also important when timing or other physical or electrical design considerations (pins, power, area) force repartitioning. Pure top-down methods may not even be able to express these constraints at higher levels. In some cases, too, when working with advanced technology, physical and electrical parameters can change late in the game; when this happens easy repartitioning is quite valuable.

An extraordinary advantage of a CPU-level (or higher-level) model, whether represented behaviorally or at the gate level, is that it is essentially a very slow computer, and its stimulus is an ordinary computer program. Numerous benefits follow. First, no knowledge of the hardware details is required for a person to run a test program on a simulated CPU. This greatly enlarges the population of competent model testers. Hardware expertise is only required when a test program goes astray. Another benefit is that (for existing instruction-set architectures) there is a ready standard of comparison and a plentiful supply of already written test programs: any program that doesn't run for very long is a candidate. A third benefit is that a program can check its own answer. The problem of checking the output of a simulation is quite

serious, the more so when the volume of tests is large, as it must be for the systems under discussion. Inevitably one needs some other model against which to compare the output. A self-checking program in essence computes some value in two different ways, checking for internal architectural consistency rather than using a separate model for comparison. This procedure will not catch all bugs, of course.

A final advantage of big-model simulation comes from the empirical observation that in using these models, designers often discover bugs in parts of the logic other than the ones they are testing. As I will argue in the next section, these serendipitous bug discoveries should be highly valued.

The orderly construction of a large model will clearly entail *some* simulation of its lower-level parts; I do not mean to prohibit small models absolutely. Indeed, too-early use of a large-scale model can be a very inefficient way to find simple bugs in small components. A modest amount of independent simulation of smaller units should be done to assure basic functionality before integration into a larger model.

Neither do I mean to imply that large-scale models should represent logic at a level of detail *finer* than that used by the designers. At some point, trusted translation tools transform the designers' logical description into an exact physical specification of the circuits. Simulation of individual transistors and the like is certainly required to certify the logical components used by the designers, but it can safely be done in the small. When logic synthesis is used, the correctness of the translation must be rigorously established if a higher-level representation is to be simulated.

One way to think about my dual recommendations is the following. Simulation of large-scale models attacks bugs in the *design* of a system, whereas simulation using low-level representation attacks bugs in the *implementation* of that design. (Things are not quite this pure, of course.) A big model, even if composed of abstract behavioral elements, can uncover design flaws: for example, the bus protocol handles a certain case improperly. Gate-level simulation, even of small pieces of the design, can uncover implementation flaws: for example, these three gates do not express the designer's intent. Together, these approaches constitute a powerful method for doing both at once, a method that efficiently uses the available resources of designer effort, computer power, and calendar time.

3 Passing Tests versus Finding Bugs

So we can now imagine a large-scale model based directly on the designers' lowest-level specification of the design. It can simulate, albeit at a somewhat slower speed, anything the real machine could do. In the design lurk some unknown number of bugs. How best to find them? In this section I will explore two possible approaches and then discuss the question of knowing how much simulation is enough. (Much of the material in this section is drawn from an earlier paper [4].)

Efficient simulation is important because simulation is expensive. In one CPU development project at Digital [5], the simulation ratio, that is, the ratio of the speed of the real machine to the speed of its system-level simulation, was 300 million to one. That means that to simulate just one second of target machine time, the simulation model would have to run around the clock for *ten years*.

The traditional or *test-centered* approach to simulation works as follows. A (long) list of tests is created, one for each identifiable function of the system; these tests should ideally "cover" the design, that is, exercise all of its parts in a balanced way. The tests are then simulated on the model. When tests fail, bugs are uncovered and repaired. Progress is measured against the list: the number of tests that pass is a measure of the goodness of the design. When all the tests pass, the design is regarded as correct, and parts can be released for fabrication.

This approach has several attractive features. It provides a definite plan of attack (the list), which allows a rational allocation of human and computational resources to the simulation task. It provides a clear measure of progress: how many tests pass? The remainder are a measure of the work left to do. And finally, the test-centered approach provides an unambiguous ending criterion: when all the tests run correctly, simulation is complete and hardware fabrication can begin.

So what is wrong with this attractive approach? Simply this: passing all the tests on the list does not demonstrate that any other single test will be able to pass. In essence this approach guarantees that fabricated hardware will pass exactly—and perhaps *only*—those tests on the list.

The test-centered approach simply has the wrong focus, for no amount of testing can guarantee the perfect correctness of a design. As Dijkstra said in another context, "testing can be used to show the presence of bugs, but never to show their absence!" [7] A computer system is just too complex to test completely. Although any computer is in principle just a huge finite-

state machine with a finite number of possible programs that could ever be run, the number of states and state-transitions and programs might as well be infinite. Exhaustive testing is out of the question, and furthermore there is no representative subset of tests which, if passed, will demonstrate perfect correctness. We can aspire to test only a tiny fraction of the system's entire behavior.

Happily, we only need to test a tiny fraction to find all the bugs. The trick is to test the *right* tiny fraction! When simulation begins there is some unknown number of bugs in the design; let B be the number. (Digital's experience suggests that for VAX designs B is on the order of one thousand.) There is a huge but finite number of test programs that could be run. Every bug can be exposed by at least one test, and some tests expose multiple bugs. In principle, no more than B tests need to be simulated to remove all the bugs (neglecting any new bugs that might be introduced in the repair process). There are, of course, many many sets of B or fewer tests, each capable of doing this. The strategic objective of system simulation should be obvious: minimize the number of tests run while maximizing the chance that some one of these special sets of B tests is among the ones actually run. There is little reason to think that any fixed list of tests constructed *a priori* will contain one of the desired sets.

"Regression testing" is a particularly wrong-headed example of the test-centered approach. It means re-running old tests after a change or bug-repair to the design, in order to make sure the design has not "regressed" or lost any abilities it once had. A particularly slavish adherence to this idea would have *all* old tests re-run after *every* design change.

If testing were free this might be all right, but in the hardware simulation environment, testing is far from free. If the tester asked "what test is most likely to uncover a design bug?" the answer would almost never be one of the tests in the regression suite. A test that has passed before is highly likely to pass again. Looking for new tests to uncover new bugs is what the bug-oriented tester would do. Regression testing is a way to get good tests, not a bug-free design; it may also encourage a false sense of confidence in the design's correctness.

"Design verification" is the traditional name for the phase of a project devoted to demonstrating the correctness of a fully specified design. But it should be self-evident that any design is most certainly *not* correct when this activity starts and possibly even when it ends. How can an incorrect design be "verified"? Perhaps a better name for this phase of a project would be *design falsification*. The object should not be to show that the design works

(for it surely doesn't), but to show exactly how it does not.

In other words, the object should be to find bugs, not to pass tests. Design verification is a gradually developing side effect of the process of design falsification: as falsification becomes more and more difficult, the design comes closer and closer to being correct. Testing and debugging should be oriented around the bugs themselves; hence the notion of *bug-centered* simulation. To find a bug it is best to be looking for one. Passing a test—getting the intended result—does not advance this effort one bit.

Indeed, following Myers, who made this observation in the context of software testing [12], we should call such a test a *failure*. If a test gets an incorrect result and thereby uncovers a bug, we should call it a success. This apparently backward way of looking at things is very useful, especially in the hardware simulation environment, where testing is so costly.

A focus on bugs will orient testing away from rigid procedures intended for thorough functional coverage, and towards a flexible approach that applies tests in areas most likely to contain bugs. These include especially:

- areas of unusual logical complexity;
- interfaces between parts designed by different people or groups;
- parts of the design that deal with rarely-exercised functionality (for example, error-detection circuits);
- and, paradoxically, parts of the design that have already yielded many bugs.

Rather then follow a fixed list of test cases, testing efforts should adapt to the empirical situation: testing methods that do not find bugs should be abandoned in favor of methods that do.

Underlying these efforts should be a fundamentally positive attitude toward bugs. Unfortunately it is more natural for design engineers (and their managers!) to see a bug as a defect or a failure. Particularly in times of schedule pressure it is difficult to imagine that finding a bug is a good thing. It is quite understandable that an engineer might be ashamed of a bug, or try to deny it, or hope that its true location is in someone else's logic, or concentrate on portions of the design that function correctly rather than ones that don't.

The design engineer wants to show that the design works; it is quite unnatural to take pleasure in a demonstration that it really doesn't. Yet this is exactly the attitude that engineering management should encourage.

Finding a bug should be a cause for celebration. Each discovery is a small victory; each marks an incremental improvement in the design. A complex design always has bugs. The only question is whether they are found now or later, and *now is always better.*

One way to deal with the natural tendency of a design engineer to resist bugs is to give the testing job to somebody else. In one group at Digital [3] a team separate from the designers helps build simulation models and then creates and runs tests. A similar arrangement is also common in software companies, where the Quality Assurance department or its equivalent is in charge of testing. (Such groups have the luxury of running their tests at full speed, of course.) But having a separate testing team is no guarantee of success: such a team might fall into the test-centered trap itself.

It is quite difficult to predict which testing methods will in fact find bugs. Planning for the design falsification phase of a project is hampered by this fact and by the uncertainty over how many bugs there are. (One Digital engineering manager called this project phase the "Twilight Zone"!) The bug-centered approach urges flexibility in testing. Whatever the plan says, the design or testing team must be prepared to abandon methods that do not find bugs and in their place expand and promote methods that do.

A bug is an opportunity to learn about the design, and can lead the designer to other bugs (called "cousin bugs" by Sherwood [3]). Upon finding a bug, the designer or debuger should try to generalize in various ways by asking questions like these:

- Is this bug an instance of a more general problem?
- Does this same bug occur in any other parts of the design?
- Can the same test find more bugs? Are there any enhancements to the test suggested by this bug?
- Does this bug mask some other bug?
- What other similar tests could be run?
- Why was it not caught earlier? Is there a coverage hole, or is a new test needed?
- How did it get into the design? Was it, for example, due to designer misconception, an ambiguity or error in a design specification, a failure of some CAD tool? (Beware of assigning blame, however.)

The bug-centered strategy, which sometimes goes by the shorthand phrase "bugs are good" at Digital, is slowly taking hold within the company. There remain, however, many groups that operate in a test-centered way, running and re-running their regression tests. The attachment these groups have to the traditional approach comes largely from its clear endpoint: when all the tests pass, testing is done. There may still be bugs, of course, but the management milestone has been met. How can a bug-centered group decide that enough testing has been done? And how can its progress be measured?

To measure progress, bug-centered testers should not count the number of tests passed, but should instead watch the *bug discovery rate*. In several projects my group has used a low-overhead but high-visibility bug-counting method. Engineers simply marked a centrally posted sheet with a tick-mark for each bug they found. Elaborate, comprehensive, computer-aided bug reports were explicitly *not* used, because we felt that the attendant visibility and overhead would discourage candid reporting, particularly of bugs found by their own creators. The bug tallies were updated weekly, and a plot of the rate centrally displayed.

Fabrication release decisions should not be made until this rate drops to a level low enough to establish confidence in the design. With a steady (or increasing!) rate of bug discovery, there is no evidence that the end might be near—none whatever. It is necessary that the rate decline in the face of determined, creative, flexible bug-finding attack. Ideally one would wait until the rate was actually zero for some considerable time. What exactly a "low" rate means is determined by the parameters of the fabrication process and the economics of the project. Low might mean very very low if the fabrication time is long, or cost high. A higher value of "low" can be tolerated if quick turnaround of new designs is available. In designs that contain a number of separately manufactured pieces, one can try to release first those pieces that have been bug-free the longest. Release decisions are in the end a question of judgment and risk: experience with prior projects, confidence in the design, and willingness to take risks with remaining bugs will determine when designs are released. There can be few hard and fast rules in this area.

One rule does seem clear, however: one should not release logic that has *never* been tested! It is important to do some kind of *coverage measurement* during simulation [11]. (The well studied area of *fault coverage* is not what I mean here. Fault coverage addresses the ability of a test to uncover physical failure of the implementation, not the design flaws we are discussing.) For example, one simple measurement is to record which signals have taken on

both logical 0 and 1 values at some point during some test. Now of course mere coverage does not show correctness. A fully-exercised design can still have bugs, so the temptation to stop simulating once some desired level of coverage is reached must be firmly resisted. But a lack of coverage of some piece of logic means that any bugs that might be there have certainly not been found. Coverage measurement should be thought of as a way to find *un*covered logic, rather than as a way to establish confidence in covered logic. Uncovered logic needs new tests.

When the bug discovery rate finally does fall off, it is important to investigate the reason. It may be that there are in fact very few bugs left in the design. But it may also be that the particular testing methods then in use are faltering. If this is so, new methods, perhaps more intensive or complex ones, should be quickly brought to bear. We do not want the bug rate to decline slowly; we would much prefer a precipitate drop from a high rate when there are in fact very few bugs left.

4 Random Testing

When the space of possible behaviors is too vast to test exhaustively (as is obviously the case for computer systems), some method must guide the selection of tests. One way to do this is *directed testing*, in which some human intention controls the selection. This is an important method for areas of the design with a high *a priori* chance of yielding bugs, but if used universally, it leaves open the possibility that unselected tests might reveal bugs. And of course it is quite likely that some behavior not fully considered by the designers, and hence not exercised by the chosen tests, might yield a bug.

Put the other way, any design bug that does show up after fabrication is proof that some important test was never simulated.

The method I advocate for dealing with the problem of a huge space of behaviors is to sample randomly (really pseudo-randomly) from one or more sets of possible tests. A random selection can pick any test with some probability, and the longer a random test-selector runs, the more likely it is that any particular test will be found.

Sometimes it is desirable to have a completely unbiased selection, so that all tests (from some specified set) are chosen with equal probability. This is clearly the right thing to do if one has absolutely no idea where the bugs might be. Sometimes, however, it may be important to bias the selection in

some known way. This bias enables the tester to focus on a particular subset of the test space. (I use terms like "space" and "probability" and "bias" here with conscious informality. No one would actually define the "space" of possible behaviors or enumerate all possible test programs. The ideas become clearer when expressed in this language, but in actual application the details behind the suggestive terminology are never worked out.) For example, to test the floating-point logic one might want to bias the selection of input data toward numerical boundary conditions of various kinds. Or perhaps at the beginning of simulation one might bias the selection toward very simple tests, leaving the more complex choices until bugs in the basic functionality had been found.

What must never be done, however, is to bias the selection is such a way that some important subspace of possible tests is sampled with probability zero!

Random testing implies that a huge volume of tests will be simulated (and also that you'd better have a fearsome computing armamentarium to run the simulations). This in turn has two consequences: first, the tests must be automatically generated; and second, the results of each test must be automatically checked. Manual generation and checking are simply out of the question unless the hardware is trivial and the volume of testing small.

An example of an automatic random tester is the AXE architectural exerciser used at Digital to test VAX implementations, both in simulation and in real hardware [1, 6]. The strict level of architectural compatibility demanded of VAX family members—essentially this: that all legitimate programs must get the same answer on all VAXes without recompiling—yields a ready standard of comparison.

AXE compares two VAX implementations by generating random instructions, running them on both implementations, and noting any differences in the results. It makes a random choice of opcode, picks some random operand specifiers, drops random data into random locations in memory, and arranges for a random selection of architectural impediments such as page faults, arithmetic exceptions, traps, and the like. AXE's random choices can be constrained by the user in various ways; the user might want AXE to pick only certain opcodes, for example. The "case" AXE generates is run on the target machine (a simulation model) and on a known standard VAX (usually the simulation engine itself) and the results are compared in detail. AXE looks at the instruction's actual result, of course, but also checks all register contents, condition codes, relevant memory locations, and the correct handling of any exceptions the case provoked.

AXE, originally developed to certify new hardware, is now also used in every VAX CPU simulation. It has proven to be an excellent bug-finder and is being enhanced to generate multiple-instruction cases.

When the designers suspect that some parts of the system are more likely than others to contain bugs, they can focus testing by biasing the random test selection, as discussed above. Another way to focus is to use a simulation *demon*. A demon is an autonomous source of interference in a simulation. A common type of demon takes the place of one system component while another, usually a CPU, runs a test. The idea is that by generating extra functional interactions, a demon can flush out lurking bugs. It can be implemented as a simple behavioral (that is, not gate-level) simulation of a system component, with its own source of stimulus.

A *Hamming demon*, for example, might randomly drop bad bits into data fetched out of an ECC-protected memory by a simulated CPU. A *bus demon* might generate random (but legitimate) bus transactions while the connected CPU runs a self-checking test that also uses the bus.

Some demons are useful because they bias the random selection of system-level test cases in the direction of high subsystem interaction. Experience teaches that system bugs often occur in these interactions, which lack the architectural clarity of specification found in, say, floating-point hardware or caches or instruction sets. Other demons, like the Hamming demon, force error conditions at unplanned times, another traditional source of bugs.

A demon, in this formulation, does not check its own answers. Instead it relies on the self-checking test running in the CPU to be affected by any bug it turns up. This will not always happen. A thorough check of all the outcomes of a demon's interference might find a failure not visible to the particular test running in the CPU. In some simulation environments, intensive checking of this kind may well be advisable.

Demons can themselves generate random activity, or use a particular type of stimulus generated according to some fixed scheme, since the desired randomness can come from the main self-checking test.

Here are some examples of demons that have been used over roughly the last six years in various VAX simulation efforts. All assume that a self-checking test (often AXE) is running on one processor, and that any errors will be reflected in some failure of the check.

1. *System bus demon.* In shared-memory multiprocessors there are several players on the main system bus: processors, memories, and I/O adapters. Several projects have used bus demons to take the place of

one or more of these players and increase the level of bus traffic in the simulation.

2. *I/O bus demon.* Here the demon sits on a simulated I/O bus pretending to be some sort of I/O device.

3. *Clock demon.* One project used a clock demon to turn off non-memory clocks at random times and for random intervals, mimicking what would happen when actual hardware clocks were stopped and restarted from the system console. (Memory clocks were needed to prevent volatile DRAMs from forgetting.)

4. *Stall demon.* Stalls occur when one part of the computer can't make progress due to a lack of response from, or contention against, some other part. A typical implementation is to block the clocks of the first part while letting the second part proceed until the original reason for stalling is removed. A common example: the first part is the processor, the second part is the cache, and the processor stalls during a cache miss. A stall demon drops in stalls of random (but legitimate) duration at random (legitimate) times.

5. *Error demon.* Some projects have used demons that randomly cause hardware errors in the simulation model. Things like single-bit errors in ECC-protected memory should be transparent, while more serious errors in less protected domains may have more serious consequences. Sometimes such consequences (operating-system intervention, for example) must be checked by more serious methods too.

There is no reason not to run multiple demons simultaneously. When simulation is turning up few bugs, the use of more and nastier demons may expose especially obscure bugs.

5 Conclusion

In this paper I have presented a set of recommendations concerning computer hardware simulation. Methodological ideas, especially those involving big projects with real-world constraints, cannot easily be subjected to rigorous scientific scrutiny. I can claim only logic and experience as supports for my prescription, which in barest essence is this:

- Simulate the lowest level of representation used by the logic designers (e.g., the gate or logic equation level).
- As soon as the basic functionality of the system parts has been tested independently, integrate the parts into a large-scale model such as a CPU or even an entire system. Use this model for most simulation.
- Remember that *bugs are good:* organize the effort around finding the bugs rather than passing the tests.
- Use randomly selected, automatically generated, automatically checked or self-checking tests.
- Use simulation demons to focus stimulus on subsystem interactions, a traditional source of difficult bugs.
- Measure coverage of the logic during simulation and add tests or test methods for areas that lack good coverage. Do *not* stop simulating when some predetermined coverage level is reached.
- Release designs for fabrication only when the bug discovery rate is low (despite determined efforts to raise it), and *not* when some preconceived list of tests is passed.
- Do not stop simulating just because the parts have been released! Simulation is *never* done. A bug found during fabrication is one less bug to find in the prototype.

In a sense one could say that the reason we do extensive simulation of computer system hardware is that we don't know any better. We would much prefer to have the ability to produce bug-free designs to begin with, or to have formal proof methods capable of verifying big systems, or to have rigorous testing regimes that could certify correctness. We can certainly hope that the increasing use of logic synthesis techniques [14] will result in designs with fewer bugs. We can also hope that hardware verification methods of various kinds [2, 8] will be able to handle ever-larger and more complex structures. But for the present, simulation is our lot, and we need to do it well.

References

[1] Bhandarkar, D. Architecture Management for Ensuring Software Compatibility. *IEEE Computer*, Feb. 1982, pp. 87-93.

[2] Bryant, R.E. A Methodology for Hardware Verification Based on Logic Simulation. Tech. Report CMU-CS-87-128, Computer Science Dept., Carnegie-Mellon University, June 8, 1987.

[3] Calcagni, R.E. and Sherwood, W. VAX 6000 Model 400 CPU Chip Set Functional Design Verification. *Digital Technical Journal 2,* 2 (Spring 1990), Digital Equipment Corp., Maynard, MA, pp. 64-72.

[4] Clark, D.W. Bugs are Good: A Problem-Oriented Approach to the Management of Design Engineering. *Research-Technology Management 33,* 3 (May-June 1990), pp. 23-27.

[5] Clark, D.W. Pipelining and Performance in the VAX 8800. *Proc. Second Int. Conf. on Architectural Support for Prog. Lang. and Op. Syst.*, ACM/IEEE, Palo Alto, CA, Oct. 1987, pp. 173-177.

[6] Croll, J.W., Camilli, L.T., and Vaccaro, A.J. Test and Qualification of the VAX 6000 Model 400 System. *Digital Technical Journal 2,* 2 (Spring 1990), Digital Equipment Corp., Maynard, MA, pp. 73-83.

[7] Dijkstra, E.W.D. Notes on structured programming. In *Structured Programming* (Dahl, Dijkstra, and Hoare), New York, Academic Press, 1972, p. 6.

[8] Hunt, W.A., Jr. *FM8501: A Verified Microprocessor.* Ph.D. thesis, Inst. for Computing Science, Univ. of Texas at Austin, Austin, TX, Tech. report 47, Feb. 1986.

[9] Kearney, M.A. DECSIM: A Multi-Level Simulation System for Digital Design. *Proc. Int. Conf. on Computer-Aided Design*, IEEE, New York, Oct. 1984.

[10] McWilliams, T.M. Verification of Timing Constraints on Large Digital Systems. *17th Design Automation Conf.*, ACM/IEEE, June 1980, pp. 139-147.

[11] Monachino, M. Design Verification System for Large-Scale LSI Designs. *IBM J. of Research and Dev. 26,*, 1 (Jan. 1982), IBM Corp., Armonk, NY, pp. 89-99.

[12] Myers, G.J. *The Art of Software Testing*, New York, John Wiley and Sons, 1979.

[13] Samudrala, *et al.* Design Verification of a VLSI VAX Microcomputer. *Proc. MICRO-17*, ACM/IEEE, Oct. 1984.

[14] Thomas, D.E., *et al. Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Boston, Kluwer Academic Publishers, 1990.

[15] Wilhelm, N. Personal communication, 1986.

Acknowledgments. I am grateful to many Digital colleagues who have advised me and/or worked with me on the simulation phase of several CPU projects. Among them I acknowledge particularly Pete Bannon, Debbie Bernstein, Tom Eggers, Jim Keller, Kevin Ladd, Will Sherwood, Bob Stewart, and Neil Wilhelm. Helpful comments on an early draft of this paper were offered by Dileep Bhandarkar, Jim Finnerty, Paul Kinzelman, Richard McIntyre, and Will Sherwood.