Electrical Verification of the HP PA 8000 Processor

Electrical verification applies techniques from both functional verification and reliability and environmental testing to improve the quality of the CPU. Electrical verification checks that the CPU functions correctly under stressful environmental conditions, well outside the normal operating environment.

by John W. Bockhaus, Rohit Bhatia, C. Michael Ramsey, Joseph R. Butler, and David J. Ljung

Early in a product's design life cycle, considerable attention is paid to its functional correctness. This functional verification is carried out on the earliest prototypes and, especially in the case of complex devices such as large VLSI circuits, even earlier through simulation.

As a product approaches customer shipments, testing it against HP's stringent reliability and environmental specifications is a critical task.

In between these two test methodologies, there is a third method that is becoming increasingly important. This is *postsilicon electrical verification*. Electrical verification applies techniques from both functional verification and reliability and environmental testing to improve the quality of a device or product.

The philosophy of electrical verification is different from the other two methods. While it is possible, although not necessarily common, to complete functional verification and reliability testing without finding reasons to change a design, electrical verification's goal is to find a design's weaknesses and fix them. Even a very good design should come out of electrical verification with a higher level of quality.

Like functional verification, electrical verification seeks to exercise as many of a design's logic states, signal paths, and state transitions as possible. However, entering a state, driving a signal, or triggering a transition once is not enough for electrical verification. Combining reliability testing with the coverage of functional verification, electrical verification repeats the functional tests under stressful conditions beyond what a product may ever see in a real application.

Electrical verification goes beyond the limits of reliability and environmental testing, which is typically done only at the system level. Reliability tests are usually done independently of each other. For example, line voltage variations are not applied concurrently with ambient temperature tests. Reliability tests stop at predefined limits. In contrast, electrical verification varies many test parameters at the same time. The ranges of those parameters are continually increased until failures are found.

Electrical verification is not simply a random scattering of tests executed in the hopeful search for some kind of statistical confidence. Instead, the goal is complete coverage of the product's operating space and beyond. This serves two purposes. First, the design is verified over all of the combinations of actual conditions it may encounter. Secondly, failure mechanisms or critical features that may lie outside normal operating limits can be found, identified, and possibly fixed. These items, such as critical timing paths, charge sharing conditions, or marginal driver strengths, can move inside normal limits as a product ages, manufacturing conditions change, or other unanticipated situations arise. Removing them early in the design life cycle ensures a reliable product with a longer life for the customer and avoids a costly in-production change for HP. Furthermore, fixing these electrical failures can increase the yield of the device at a given frequency and can enable higher-frequency and higher-performance upgrades.

An additional purpose of electrical verification is to help determine production test limits and guardbands. By including IC process parameters in the verification effort, test limits can be extrapolated to predict proper function through normal operating conditions.

Fig. 1 is a flowchart of the electrical verification process.

Shmoo Plots

The hunt for electrical bugs begins with the *shmoo plot*. The shmoo was a character in the Lil' Abner cartoon strip that had a changing, bloblike shape. The shmoo plot shows whether the device under test passed or failed as a function of various combinations of electrical parameters applied to it. The name has become part of the engineer's vernacular because the region where a particular device passes or fails, plotted against the parameters applied, may have some of the rounded

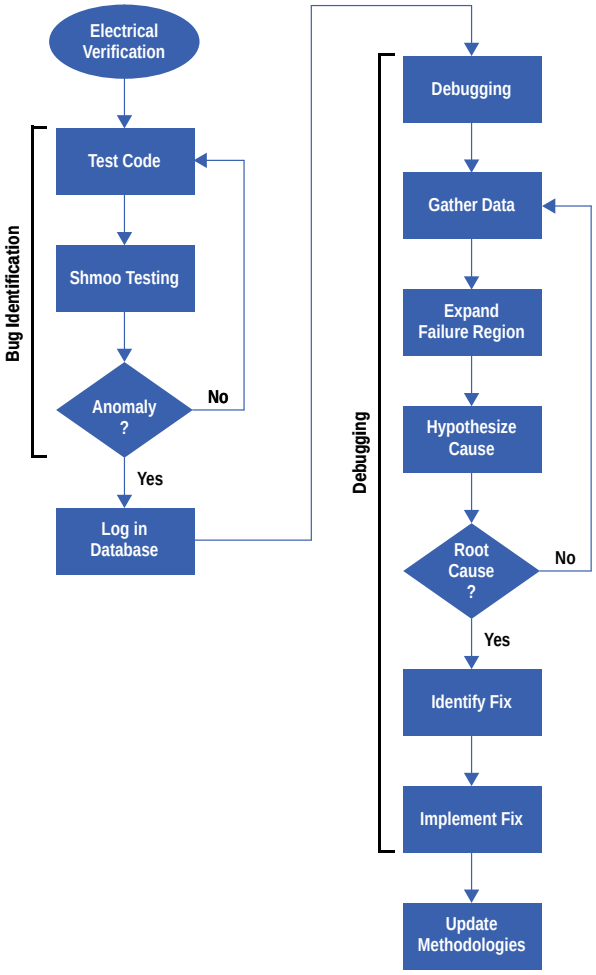

Fig. 1. *The electrical verification process.*

curves and shifting nature of the mythical shmoo. Often, the shape of the plot conveys information about the failures. Many common shapes have been given names (see ***Subarticle 4a***).

The shmoo's name has also become a verb. To shmoo is to run a test repeatedly while varying one or more parameters. These parameters are the axes of the shmoo plot and include both the obvious and the nonobvious.

Obvious shmoo plot parameters include power supply voltage and temperature. In complex systems such as those using the PA 8000, many different supply voltages exist. Supply voltages and temperature are clear choices for shmoo plotting because they so directly affect the operation of electronic devices.

The IC manufacturing process is a key shmoo plot parameter for projects like the PA 8000. In keeping with the goal of methodically covering the shmoo space, testing a large, random sample of parts isn't enough. Instead, prototype parts are manufactured with one or more process metrics intentionally modified. Typical IC shmoo plot parameters are transistor gate length and leakage currents.

Clock frequencies are also good shmoo plot parameters. Increasing the frequency of clocks is a good way to find slow signal paths. However, pushing frequencies higher only tells some of the device's story. Useful information can also be found by seeing how slowly it can go and by testing many frequencies in between the maximum and minimum. Logic races and transmission line reflections are just two potential problems that may lurk in the low frequencies, which the engineer often assumes are "easy."

A shmoo plot parameter that may be less obvious is the software executed on the device under test. Good shmoo plot code seeks to exercise as much of the device as possible. Executing a power-up self-test or booting an operating system may seem complicated, but they are not necessarily good shmoo tests. The PA 8000 shmoo process used a large number of tests that ranged from specially designed to randomly generated.

Test Cases

The success of any verification effort depends to a large extent on the nature and type of test cases that are run on the CPU. The testing code needs to be good enough so that when the systems reach customers the CPU is bug-free. To ensure this, the tests must provide adequate coverage of the design features incorporated into the CPU. Furthermore, because of the complexity of today's processors, it is impossible to imagine all of the interactions that must occur to cause a particular event in the processor. This complexity necessitates the use of random testing. In the postsilicon environment, random testing is aided by the fact that throughput is generally not a problem (compared with the presilicon simulations done on software models, which are millions of times slower than running on the actual hardware) and therefore a huge volume of random testing can be accomplished.

The electrical verification of the HP PA 8000 CPU relied upon the following sources for test cases:

- Directed handwritten tests
- Focused random tests targeted at specific CPU functions
- Random code generators
- Library of worst-case tests for previous bugs
- HP-UX* application code.

In most instances, these test cases were checking for failures in real time. In general, they would set up some initial conditions, run the test code, and perform some checking. Some cases checked for a specific outcome in memory or a general register. Others compared the full architectural state at the end of the test case to some expected final state.

For most of the sources mentioned above, the test cases were leveraged from the presilicon verification effort through the use of various scripts to perform modifications for the postsilicon operating environment. Several benefits can be realized by leveraging the work from presilicon verification. First, leveraging all the tools results in less development time and hence less total work for the postsilicon verification team. Second, by sharing the tests and tools, we get a common environment between presilicon and postsilicon verification. This allows easier modeling of postsilicon failures in presilicon simulation tools and makes the learning curve easier as well. It also provides a path to go back and forth between the two environments, which allows directed test development for postsilicon failures.

Since presilicon verification is targeted at functional correctness, the use of tests leveraged from that effort requires some caution. Electrical verification tests in general need to be much more data-pattern-sensitive than their functional counterparts. For example, a bus may need to be driven from all zeros to all ones or alternating ones and zeros to adequately test for electrical failures, whereas the data patterns in a functional test don't usually influence the logical correctness of the design. Therefore, the random code generators used in the electrical verification effort were modified to provide knobs that allowed the test case writers to control the data patterns used in these tests.

Our library of worst-case test code for all bugs known to date was valuable in ensuring that no known failure mechanisms had gone uncovered as we went through different revisions of the chip. This library of tests was always kept updated and used repeatedly on all CPU parts.

One final source of test cases was our HP-UX stress test suite. Not only is this most similar to what the majority of customers are going to run, but HP-UX testing can also offer the most coverage. Usually this is used as the last set of testing to make sure that all of our test coverage so far has been close enough to the stress that HP-UX code puts the PA 8000 through. The stress test suite consists of a number of applications including the SPEC benchmarks and scripts that make sure that everything is running correctly. The drawbacks are that the run time for HP-UX stress tests is a few hours, which is orders of magnitude longer than the other test types, and that these tests are much harder to debug if failures occur.

Automated Tools

To save time, we created a number of tools that help us automate many of the verification tasks. Each of our test systems was connected to a controller system (see Fig. 2). We then replaced the boot ROMs on the test system with ROM emulators. Our controller system controlled the voltages, frequencies, and temperature and the code that was run on the test system, and it also monitored all of the I/O to and from the test system through the RS-232 port.

We needed to have complete control over the code that was run on the test system. Instead of using the boot ROM to initialize the system so that we could run a full operating system, we used our own framework code called the *CharROM*, short for *characterization ROM* (see Fig. 3). The CharROM installed a subset of the normal initialization code and then ran test code for us. All of our tests were compiled into a separate ROM called a *testROM* which could be uploaded without changing the CharROM. Each testROM contained the test code as well as information on how to run each test, such as verbosity settings and number of iterations. It also contained information about the test that was printed out so that our tools could save information on what tests we were running. With our systems set up this way, we could turn on power and within a few seconds the test system would be printing out initialization information and then test codes.

The CharROM had the flexibility to run tests in batch mode, one at a time according to the testROM, or interactively for debugging. In interactive mode the CharROM let us run tests and modify and view the state of the chip.

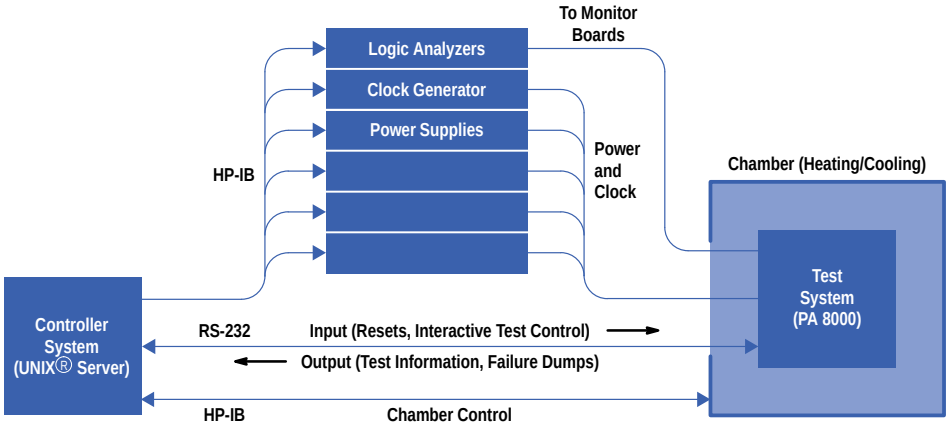

Fig. 2. *Electrical verification test system setup.*

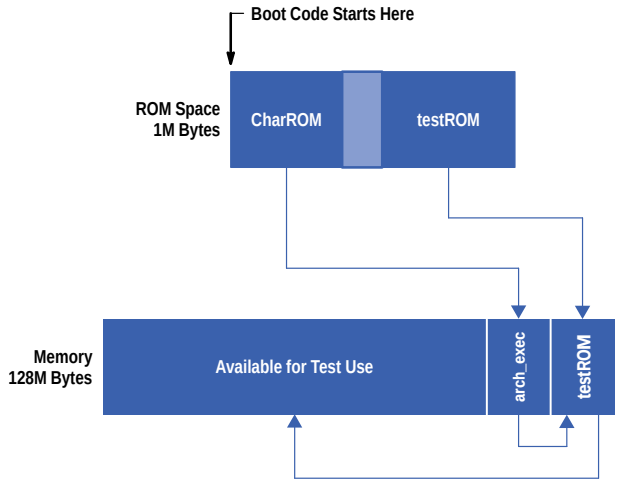

Fig. 3. *The CharROM makes it possible to boot quickly and change tests rapidly. Boot starts at the beginning of the CharROM.* arch_exec *and the testROM are copied to memory for speed reasons. Basic assembly tests are run either in ROM or in memory. Modified phase 1 format tests are unpacked into available memory by* arch_exec*, which runs and checks them.*

The big advantage of the CharROM was that it booted quickly and let us change tests rapidly. This saved a great deal of time during debugging when we needed to run many code experiments. It also eliminated a lot of the boot code that would be needed to run something like the HP-UX operating system. This meant that we were less likely to hit a failure while booting and if we did we could often make quick changes to the CharROM to avoid the failure. This was most obvious when we discovered an electrical bug in a branch instruction that was keeping us from booting. We were able to rewrite the CharROM framework in two days without using the failing types of branches, something that we could never have done with a full-fledged boot ROM.

The most important feature of the CharROM was the control it gave us over exactly what code was being run on the system. Running tests in an environment like HP-UX leaves the tester at the mercy of the operating system, which can switch processes in and out and limit access to privileged operations.

One of the important responsibilities of the CharROM was to initialize the chip state between tests. This helped control repeatability for a given test (so we weren't dependent on a random chip state) and also helped avoid dependencies between tests, that is, a previous test affecting the run of a future test.

Inside the CharROM we had a subframework called arch_exec that could run our modified presilicon test cases. arch_exec takes apart the initial state and sets up the chip accordingly. After the test is run, arch_exec compares the chip state to the expected final state information in the test, automatically showing us any failures. This let us deal with many tests in bulk.

To run our shmoo tests we had scripts that would boot the systems at each point in the shmoo test domain and read the output to decide if the tests had passed or failed. After running a shmoo test or even during a shmoo test we could analyze the output to ignore certain failures or focus on specific failures.

At the completion of a shmoo test, the shmoo script stored all the output in our shmoo database. We used the database to look for specific tests or specific parts so we could avoid duplicating work. This also turned out to be very useful when we needed to take another look at past bugs; we still had all of the shmoo information for the bug work.

Failure Identification

The process of electrical verification of a CPU begins with the task of identifying electrical failures. An electrical failure can be described as the malfunction of a chip under certain but not all operating environments. If the failure occurs regardless of the operating environment, then it is termed a functional failure and is not covered here. Typically, electrical failures can be traced to some electrical phenomenon that occurs only under certain operating environments. Some examples include latch setup or hold time violations, noise issues, charge sharing, leakage issues, and cross talk.

The task of identifying new failure mechanisms involves a number of steps. First, test code must be selected and run in a variety of operating environments. The data collected from this is displayed graphically in shmoo plots and anomalies are noted. Next, the anomalies are checked for repeatability. If the anomaly repeats reliably, the failure signature is analyzed in an attempt to classify the failure or narrow it down to a particular area of the CPU, if possible. The sensitivities to different operating environment variables are also determined to gain further understanding of the failure mode before it is moved into the debugging stage. Each of these steps will be discussed in more detail.

Searching for an Anomaly

Electrical verification of the PA 8000 CPU encompasses a huge test space that is impossible to cover in a reasonable amount of time. This is partly because of the increasing complexity of devices and partly because of the large number of operating environment variables. Operating variables include test code, ambient temperature, several supply voltages, frequency and bus ratios, types of CPU chips (i.e., variations in the fabrication process), different speed grades of cache SRAMs, and many others. A number of techniques were applied in the electrical verification of the PA 8000 CPU that, taken together, effectively covered this large test space.

Initially, the emphasis is placed on varying a large number of the operating variables and exercising the CPU with simple test code. A variety of CPU parts from different corners of the fabrication process are deliberately selected and run under various combinations of temperature and supply voltages to look for failures. For example, known fast PA 8000 CPUs were run in a cold chamber at high supply voltages to look for one class of failures. Similarly, a set of slow CPUs were run in a hot chamber at low supply voltages to look for another class of failures. Experience is always a good guide to the operating variable combinations that are likely to yield failures.

Stress testing is another technique that is applied to induce failures. Stress testing refers to running the CPU with test code on the fringes of the operating environments, under conditions to which an actual system in the field may never be subjected. However, a failure induced in this fashion can often be moved into an operating region that we care about, simply by further experimentation and analysis.

As the process of electrical verification proceeds, the emphasis shifts from running simple test code at a variety of operating points to more complex code sequences at fewer operating points. This can be compared to exploring the test space from a *breadth-first* search to a *depth-first* search. The more complex code sequences are derived from running several random code sequence generators, pseudorandom focused tests, directed tests, and HP-UX application code.

Using one or more of the techniques outlined above, test data is gathered and can be viewed in shmoo plots. These plots are examined and compared to what has been observed in the past for previous test runs on earlier silicon revisions. If the shmoo plots reveal regions of failure not observed before, then we have a *shmoo anomaly*, which needs to be pursued further.

Verifying Repeatability

Once a shmoo anomaly is identified, a number of steps need to be taken to validate and confirm it. It is important that the anomaly be reliably repeatable and be traceable to a CPU malfunction. The steps outlined below are used to satisfy the repeatability requirement.

1. The failing code sequence is rerun several times on the same CPU to confirm failure. This is done to rule out the possibility that an inadvertent change in the operating environment may have induced the failure.
2. In a system verification environment, several other components must be removed from suspicion before the anomalous behavior can be attributed to the CPU. The failing CPU can be placed in a completely different system and the failing code sequence rerun under similar operating conditions to repeat the failure. The failing CPU can also be used in several different processor boards to rule out any dependencies on the processor board characteristics.

3. The next step is to try to locate the failure mode on different but similarly fabricated CPU parts. These could be parts from the same wafer or with similar process characteristics. If the failure mode is not observed on any other CPU, then it is generally considered to be a test escape from the wafer and package screens, meaning there is a defect on this chip that the wafer and package screens did not find. In other words, we have not found an inherent problem with any circuits on the chip. In that case, we will investigate whether we have a coverage hole in our wafer or package screens, rather than move this failure into the debugging phase.
4. If the above three steps are satisfied, then the failure mode is checked for sensitivities to different operating environment variables. Most electrical failures are modulated by one or more of the operating variables, whether it be temperature, supply voltage, or delays on key system clocks. This can not only expand the failure region, but also provide some clues to the type of failure, which can be extremely useful information for the task of debugging.
5. Throughout the process of verifying the repeatability of the failure mode, it is also important to watch the failure signature and check it for consistency. That is, one must ensure that each rerun of the test code is producing the same failure mode.

Classifying the Failure Mode

After a shmoo anomaly is identified and has passed the repeatability requirement, it is time to classify and list the characteristics of the failure mode to see if it is unique or is one that has been observed before. Either classification is important. If it is new, then it needs to be debugged fully and its root cause determined. On the other hand, if it is a repeat failure mode, then this failing code sequence needs to be compared with the current known worst case for that failure mode and understood as well.

Failing code sequences can come from a variety of sources. Typically, each failing sequence will have a certain failure signature and much can be learned from it. Here, we will discuss three types of failures.

The first type of failure comes from self-checking code. A typical example might be code written to exercise and walk known patterns through the cache SRAMs. Such code will check its results and the failure messages will be self-explanatory.

The second type of failure is a final-state error, generally produced by random code generators. Random code generators produce tests that consist of initial CPU state, a sequence of assembly instructions, and an expected final state. When a test terminates, the final state is checked against the expected final state and discrepancies are noted. By analyzing the final-state error messages and looking at the test code sequence, one can infer quite a bit about the nature of the failure and come up with a set of experiments to further zero in on the failure.

The third type of failure is one in the framework code. Framework code is code written to allow tools such as random code generators to run on the hardware. The framework provides the environment for initializing memory, caches, and the architected state of the CPU. Sometimes, especially early in the project, failures will occur in running the framework code. In general, these are hard to debug since the failing code sequence (the framework) could be thousands or millions of instructions long.

The characteristics of the failure mode are determined by noting the sensitivities to different operating environment variables from the repeatability experiments above or through additional experiments at this stage. It is important to do some amount of debugging and failure characteristic determination to rule out most known failure modes to date.

To summarize, the task of bug identification is complete when we have accomplished all of the above and have made a reasonable effort to rule out known problems. We now have a new bug that is ready to be taken through the next task, debugging.

Debugging

The goal of the debugging effort is to determine the root cause of the failure and fix it on the chip in a new revision. The main steps to achieve this goal are gathering data about the failure, expanding the failure region, and hypothesizing the cause of the failure. These steps are all part of an iterative process that can lead us to our goal of complete understanding. As more data is gathered about the failure, a more complete and accurate hypothesis can be formed and a more accurate worst-case vector can be determined. On the other hand, new information may also prove that our initial hypothesis was incorrect. In that case, we go back to the data gathering step to acquire more information about the failure. When all of our data is consistent with our hypothesis, we have the root cause of the failure.

Gathering Data

Once we have determined from the bug identification process that the failure is one that we have not seen before, we need to gather as much data as possible about this new failure. If multiple chips fail in the same way, we may be able to correlate the failure with a specific wafer or lot or to a specific speed grade of the chip. For example, it is possible that only chips with extremely slow FETs will fail. Checking which revisions of the chip fail can tell us whether the failure is related to a recent change in the chip, or whether it has always been there.

The next step is to shorten the instruction sequence that will cause the failure. Often, failures occur in sequences of over 100 instructions, but the failure itself usually requires only a few specific data patterns and some specific instruction timing. Determining where in the instruction sequence the failure is occurring is one step toward isolating the failure. Occasionally, the failing instruction is easy to find. For example, if only one instruction in the case modified the failing register and the inputs to that one instruction did not change during the case, the failure has been isolated to the instruction sequence around that instruction. Usually, it is more difficult. If the failure causes an unrecoverable trap or reads bad data from the cache or main memory, we need more information before deciding where the failure occurred. For example, if a load from cache reads the wrong data, is it because another instruction stored bad data, or were the address lines incorrect during the read, or did the CPU corrupt the data after it was read?

Failures in the framework of a random code generator are quite difficult to debug, especially if the framework is written in a high-level language such as C++. If the failure can be localized to a specific code sequence, which could be quite long, it can usually be ported to a standalone case (no framework involved). From there, the same steps are taken as with any failure sequence to shorten the test case while maintaining the same failure mode.

Monitoring external events may be useful. Logic analyzers attached to external pins, such as the system bus interface and the cache interface, provide a picture of what the CPU is doing when the failure occurs and can help narrow down where in the code it is failing. We may see instruction fetches from main memory, which can tell us what area of the code the CPU is executing. Since the logic analyzer stores many, many previous states, we can look back through the execution of the case to see when bad data starts to appear. If the failure relates to an off-chip path, oscilloscopes can be used to verify the signal integrity of suspect paths. We have used this method when debugging noise-related problems and failures caused by imprecise impedance matching.

We would like to narrow down the code sequence to a very short sequence of instructions that will still fail in the same way as the original case. Creating a very short case such as this is not easy, especially considering that the PA 8000 uses out-of-order execution. Changing the original test case in any way may change the timing of certain events in the case such that it may not fail anymore. In gathering information about the failure, it is important to determine what events contribute directly to the failure and in what sequence the events must occur for the failure to occur. Just removing instructions starting at the beginning of the case may not help. Suppose that a load instruction, which normally would have caused a cache miss and a request to main memory, is removed from the beginning of the case. The behavior of the case will change because the next memory operation that accesses that cache line will cause the cache miss instead. This change in timing may cause two events in the CPU that were concurrent in the original case to be separated by many states in the modified case. Removing instructions may also have the effect of changing register data patterns that may have been required for the failure. If an add instruction that sets up a 0x0F0F0F0F data pattern in a register is removed, that register will contain its initial value instead—different from the pattern set up by the add—and the failure may not occur.

Experiments with the failing code are still very important. Removing irrelevant instructions and data can narrow the search for the failure. It is possible that a large number of instructions in the failing sequence can be removed without affecting the failure. The data patterns in the source registers for one or more instructions might be changed without affecting the failure. Each of these changes narrows the search for the failure mechanism. Very slight changes in the code sequence or data patterns can provide information on what events are necessary for the failure. If we change only one bit in one data pattern and the failure goes away, that is a big indication that the failure requires one bit to be set a certain way. Another useful step to narrow our search is to determine if any specific CPU features are involved in the failure. For example, if we can turn off a specific CPU feature, such as bypassing a register value from a pipeline stage, and the case now passes, we might say the failure is occurring in the bypass logic, or at least the timing of the case requires a bypass.

While we are doing the code experiments, we may use some of the on-chip test and debugging circuitry to get a better picture of the failure. By running the chip in both a passing region and a failing region and comparing the two runs, we can get a picture of where the failure starts.

Using all of the data gathered, we can begin to see the overall picture of the failure. We know under what conditions the failure will occur, including frequency, voltage, temperature, and process parameters. We know a short code sequence that will fail, and what CPU features and timing affect the failure. We have a partial picture of the internal state of the failure by comparing passing and failing runs.

At the same time that we are gathering data about the failure, we are developing a new test case for this failure. This new case will use the known requirements for the failure to occur, including specific instruction sequences and timing, specific register values, and cache hits and misses. When our new case fails, we have all the elements of the failure.

Expanding the Failure Region

In most instances, the particular failure that occurs in a random instruction sequence with random data patterns is not the worst-case failure. We would like to know how severe the problem is. One of our goals is to find the worst-case vector. Failures that were previously outside the operating region usually move into or very close to the operating region with a worse vector. For example, a speed failure may occur at a significantly lower frequency when a worst-case data pattern is used. Or maybe a failure that only occurred at 40°C will now occur at 20°C. Expanding the failure into more general shmoo conditions also helps in gathering more data. (It's not much fun to probe signals in a 40°C oven.)

Electrical problems are often heavily influenced by data patterns. For example, driving different data patterns on an internal bus may increase or decrease the capacitive coupling or delay of a signal that is causing the failure. It is unlikely that the random data pattern used in the original failing case is the absolute worst for this particular failure. Complicating the matter, it may not be clear what other signals could be affecting the failing signal.

There are some cases in which the failing signal may not be appreciably affected by any other signals. In other cases, the failing signal, which might be part of a bus, may be influenced by certain data patterns on that bus. For instance, some of the signals that make up the bus may capacitively couple to the failing signal in that same bus, slowing down the failing signal or inverting its value. Occasionally, the biggest influence on the failing signal is a bus that is functionally unrelated to the failing signal, but is in close proximity physically. The same type of capacitive coupling can occur in this case.

Changing obvious data patterns—instructions themselves, operands from registers, and data results from the ALU or the cache—is the first step. If none of these seem to affect the failure, the layout can be consulted to see what buses or signals run adjacent to or on top of the victim signal. Finding one or more data patterns that influence the failure also allows a better understanding of the failure.

Hypothesizing the Cause

In hypothesizing the cause of the failure, all of the information that has been acquired about the failure will be used.

The minimum code sequence is especially useful to the circuit debugger. First, it provides a list of code sensitivities that either turn the failure on or off or expand the failing region. Second, this code can be simulated by a switch-level simulator to give the debugger full observability into the on-chip circuits being exercised by the test case. By comparing simulations with and without the code sensitivities, the exact effect of the code on the circuits is observed.

The switch-level simulator is the first tool used by the circuit debugger. The exercised circuitry is compared together with the internal state differences from the passing and failing data captures to narrow down the circuits involved. This process yields several potential code experiments that will continue to narrow the playing field. At this stage in the debugging cycle, the circuit debugger is working side-by-side with the system debugger to isolate the failure.

Eventually, the circuit debugger makes a root-cause hypothesis as to the cause of the failing behavior. This hypothesis can frequently be supported by the switch-level simulator. For example, if the hypothesis is that a certain latch fails to make setup, this latch can be forced to fail in the simulator. The resulting simulated failure mode should match the actual failure mode. This is the point when the circuit debugger moves to SPICE as the main debugging tool.

SPICE is used to simulate the isolated failing circuitry in the appropriate failing conditions. In the latch example, the clock and data paths into the latch are accurately modeled in an attempt to reproduce the failure in SPICE under the same conditions as on real silicon. Differences are assumed to be either inaccuracies in the modeling or mistakes in the root-cause hypothesis. Obviously, these differences need to be explained before root cause is declared.

If the failure is frequency dependent, another way to validate the hypothesis is to stretch the specific clock phase during which we believe the failure occurs. By stretching a clock phase, we provide more time for the CPU to do the required work of that phase. For instance, if we have a speed failure related to one specific phase and we lengthen that one phase by 10%, the failure should get better.

We continue to gather data and hypothesize causes of the failure until our hypothesis passes the root-cause test.

Declaring the Root Cause

Often, by inspection of the failing case in the switch-level simulator or SPICE, sensitivities to local circuit behavior are predicted. This prediction can then be verified by targeting code to induce this behavior. For example, we may try to precondition a particular bus with a specific data pattern such that it no longer transitions as in the failing case. Or we may change the instruction timing of the case such that two events no longer occur in the same cycle. If we can turn the failure on and off (the light-switch test) by changing one of the known sensitivities, we have a good understanding of the failure.

Throughout the debugging cycle, all facts and observations are documented thoroughly in a bug database. This database is used to drive experiments to fill in data where it is missing or to explain observations. The entire weight of this data is compared to the root-cause hypothesis for consistency. Any data point in conflict needs to be explained before the root cause of the bug is considered known. This diligence to the data has avoided many premature and wrong root-cause analyses.

Once we have demonstrated a light-switch test and our hypothesis agrees with all of the data, we are at the root cause. We believe we fully understand the failure.

We then revisit the worst-case vector analysis one final time. Even if our final worst-case vector does not move the failure into the operating region, we may fix the problem anyway, because it is hard to determine if a small process shift later in this product's life could move this failure close to or into the operating region.

The next step is to fix the problem. If we can fix the problem with only a change in the metal layers, the turnaround time for new chips is much shorter. To verify that the proposed fix will actually eliminate the failure, we may do a FIB (focused ion

beam) experiment, in which one of the existing chips is modified (metal lines are cut and new ones are deposited) to include the change. The chip is characterized before and after the FIB change to determine how the failure was affected. If the failure was eliminated, we have good confidence in our fix, and we will put our fix into the plan for the next CPU revision.

Creating the Golden ROM

After a bug has been closed, its worst-case test is added to our ROM of all other worst-case tests that have exposed bugs. We call this ROM the *golden ROM*. We use the golden ROM for much of our volume shmoo testing, to serve a number of purposes. It shows where the current bugs were found and can show how a fix or certain chip characteristics could affect these failures. It also lets us know if a bug has been reintroduced, which happens on occasion. As the golden ROM grows in size, it naturally gives us more coverage. Many of our new bugs are found by running the old bug code in our golden ROM. If a test case has important coverage that we do not have in our tester screens, golden ROM tests can be converted into broadside vector tests for our package screens.

Updating the Methodologies

When the root cause or problem circuit has been identified, it often uncovers a flaw in our design methodologies. This implies that other similar circuits may be used on other parts of the chip but haven't been discovered yet. The aphorism "If there's one rat, there are many rats" becomes our motto. A "many rats" investigation is launched to find a tool-based method of extracting similar circuits from the chip database and fixing them if appropriate. Quite often, the failing circuitry has a unique topology that can be searched for with a tool. Finally, this flaw in the design methodologies is documented and the methodologies are updated.

Conclusion

We continue the electrical verification process until we have searched our matrix of variables—temperature, frequency, voltage, process, and test cases—and we can find no more failures that we believe could move into the operating region. This process spans multiple chip revisions, with each new revision fixing one or more failure mechanisms. This process ensures the long-term quality of the product throughout its lifespan.

In addition, we analyze the problems that we found and integrate the solutions to these problems into our design methodologies so that future products can avoid the same pitfalls and potentially reach high quality levels more quickly than previous products.

Acknowledgments

The authors would like to thank everyone who contributed to the successful electrical verification of the PA 8000 CPU. We especially want to acknowledge the hard work of the electrical verification team in the Engineering Systems Laboratory in Fort Collins, Colorado for finding and fixing some tough electrical bugs. We would also like to thank our system partners—the General Systems Laboratory in Roseville, California and the Fort Collins Systems Laboratory in Fort Collins, Colorado—for their help in sourcing electrical bugs.