Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator

Application to the x86 Microprocessors Family

Laurent Fournier
IBM Haifa Research Lab
laurent@vnet.ibm.com

Yaron Arbetman
IBM Haifa Research Lab
yaronar@vnet.ibm.com

Moshe Levinger
IBM Haifa Research Lab
mosh@vnet.ibm.com

Abstract

Even though the importance of microprocessor design verification is widely acknowledged, no rigorous methodology is being commonly followed for its realization. This paper attempts to delineate such a methodology, and shows how it is promoted by Genesys, an automatic pseudo-random test-program generator. The methodology relies on a verification plan which induces smart sets of tests that carry out the verification tasks. The paper reports on an application of this methodology, using Genesys, to verify an x86 design and describes, in particular, how this methodology could have helped to avoid known escape bugs, such as the recent two infamous Pentium Floating Point bugs.

1.0 Introduction

It is widely recognized that functional verification emerges as the bottleneck of the design development cycle. This is due to a combination of several correlated factors: exponential increase in design complexity, tighter time-to-market requirements, and higher quality expectations. In parallel, verification means are not evolving at a matching pace. The cost of the late discovery of the recently found Pentium FDIV flaw (around $475,000,000) demonstrates the implications of having a design that does not totally conform to its architectural specification [2]. It is therefore not surprising that, for a typical microprocessor design project, up to half of the overall resources spent, are devoted to its verification [3, 4]. This paper suggests an overall methodology for the functional verification of microprocessors, and explains how this methodology is promoted by Genesys, a pseudo-random test-program generator developed at the IBM Haifa Research Lab [4]. In addition, it reports some of the insights gained through the application of this methodology to x86 microprocessors.

Common verification practices include tests obtained from several sources [6,7]. Firstly, a small fraction of the tests are usually devised manually to target corner cases, or otherwise hard-to-reach parts of the design. Secondly, the simulation includes running existing sets of tests (legacy tests or commercial test suites) [10]. These tests typically have claimed coverage, and, most importantly, have already been successfully employed in the verification of similar designs. Obviously, this component is available only if the targeted architecture is an established one. For the x86 architecture, such sets of tests are generally available and are being updated along with each architecture upgrade. Thirdly, after the hardware is ready for use, some extensive applications (e.g., operating systems) are run. Finally, automatic test generation, usually mainly random and of restricted scope, might be performed. The overall process should be interleaved with some means of coverage. Coverage is a major constituent of the verification process and therefore deserves further attention. This however is beyond the scope of this paper. In addition, designers should routinely keep in mind the verification implications of their development or modifications, thereby collaborating toward a *verification-aware* design process [5]. It will be shown, in this paper, how the suggested methodology, and Genesys in particular, promotes such a process.

Genesys, a follow-on of the Model-Based-Test-Generator [1,4], enables the combination of randomness and control, thus generating a virtually infinite number of high quality tests. It was primarily developed to minimize the effort to apply it to any architecture, and to allow the usual architectural changes and upgrades to be easily implemented within the tool. Moreover, its most powerful property is its ability to be externally and incrementally enriched by Testing Knowledge (TK) in order to influence the quality of the generated tests [4]. In this way, corner cases can be assigned a suitable probability to occur, whereas their chance of appearing randomly would be practically nonexistent. In fact, Genesys and, in particular, the incremental TK paradigm, is especially well suited to the methodology described in this paper.

The importance of the x86 architecture cannot be overstated, and it seems that this importance will not decrease in the foreseen future. It is therefore of interest to study the results of applying the proposed methodology to the verification of an x86 design.

The remainder of the paper is organized as follows: Section 2 suggests a functional verification methodology. Section 3 presents the Genesys system, and Section 4 describes how it copes with and suits the implementation of the described methodology. A case study, done by analyzing the implementation of the above methodology on an x86 microprocessor, is reported in Section 5. Section 6 concludes the paper.

2.0 General Methodology

The overall strategy relies on the early composition of a Verification Plan. The Verification Plan will ultimately induce sets of tests that realize the verification tasks. This section suggests a framework for composing and implementing a Verification Plan, independently of the tools available for realizing the plan. Section 2.1 describes the Verification Plan, whereas Sections 2.2 - 2.4 survey the different means available for its implementation.

2.1 Verification Plan

It is obvious that any verification activity should be preceded by the composition of a Verification Plan that describes the overall methodology. This includes the description of high level verification goals leading to detailed verification tasks. The plan should be composed from a deep understanding of the architecture and the main properties of the micro-architecture. Available verification means (e.g., existing tests, test generators, etc.) should not be taken into account at this point, since they could bias or limit the scope of the Verification Plan. As the design evolves, new tasks are added to cover emerging implementation details. The plan is therefore a living document reflecting the present state of the design, and can also be incremented upon discovery of a bug.

2.2 Test Repertory

The following subsections define the different types of tests required during the verification process. A summary appears in Table 1.

2.2.1 Periodic Regression Sets (PRS)

Starting early in the design, a *test regression* should be run frequently in order to yield a reasonable confidence in the overall design status. It functions as a kind of *"barometer"* following the design evolution from start to end. Consequently, it should be relatively small so that it can run fast, yet be comprehensive enough to provide useful feedback. These goals are conflicting, but can be simultaneously met by directing a PRS to all the different capabilities of the design, while providing only loose coverage.

The contents of this set should be adapted to the state of the design, starting from very simple tests and evolving as new capabilities are added. Its running frequency is dependent on the number of problems uncovered, and on the time needed to correct them. Typically, such a regression should be routinely run on a daily basis.

Since it is inefficient to run the same regression every day, tests run on successive regressions should be different, yet be able to fulfil the same underlying general purposes. Section 4.1 describes how this goal is easily achieved using a random test-program generator, such as Genesys.

2.2.2 Specific Tests

Generally many tests produced with significant effort are worthwhile keeping for periodic reruns. These tests might be written by the designer himself (sometimes manually) to pinpoint some hard-to-reach corner cases. It would be inefficient not to exploit these tests and lose the Testing Knowledge they include. Section 4.2 shows how Genesys promotes the accumulation and further usage of such Testing Knowledge. Additionally, tests that have uncovered bugs in the design should be preserved. However, the discovery of a bug should induce a broader set of actions, as described in Section 2.4.

2.2.3 Coverage Subsets

To complete the periodic regression set, a library of tests with some known coverage properties, should be available. Different coverage models are possible, the important point being that the successful running of those tests should provide a relatively high degree of confidence. They may be run each time a major design change is performed. They may be partitioned into sets, each having a specific subgoal. In this context, a particular set might be run any time its corresponding part in the design is suspected or undergoes a major change. For example, a set responsible for the Floating Point Unit might be simulated each time this unit is significantly modified. It is also advisable to run those sets periodically (e.g., each weekend) and not only as a result of external events. However, these tests can not be run too often, since such sets of tests, in order to maintain the coverage level, are inherently huge and time-consuming to run. In any event, to maintain efficiency, the use of such coverage subsets should be delayed until the design has reached a relatively stable state.

It is common practice to purchase existing suites of tests claiming coverage, especially for established architecture such as x86. These suites can also be built using a random test generator, such as Genesys (see Section 4.1). However, one should keep in mind that there are always bugs which cannot be found by any existing static set of tests.

2.2.4 Full-Load Testing

Many intricate design mechanisms are difficult to fully exercise using standard tests. For example, causing a complex pipeline to reach full capacity depends on timing dependencies among several internal signals and is not usually accessible by regular tests. It is therefore common to directly trigger these mechanisms by artificially causing them to be in a threshold state. Standard tests can then be exercised in such a context.

2.2.5 Random Testing

When the design reaches high stability and bugs become relatively difficult to find, the testing described in the previous subsections should be mixed with massive unconstrained random testing. A random test generator is clearly needed for this purpose. Randomness is a fundamental property since bug locations are usually unpredictable. However, one should be aware of the limitations of random testing. Indeed, pure random testing is inefficient since the space domain is enormous and the probability of interesting cases is infinitely small. Section 4.3 describes how Genesys avoids the pitfall of pure randomness.

2.2.6 Hardware Testing

Once the hardware is up and running, some testing can be resumed directly on the silicon. Beyond the fact that it is important to check the final product itself, the main advantage of running tests directly on hardware (as opposed to a simulation environment) is that it is much faster. But, since debugging on hardware is typically very inconvenient, testing should also continue in the simulation environment.

Large applications, such as operating systems, are usually run. Although they are important confidence builders, one should keep in mind the limited value of such applications in terms of verification. They tend to repeatedly exercise a very small portion of the design, therefore it would be premature to reach conclusions from running them successfully.

In any event, it is profitable at this point to run very long tests on the hardware. These tests can also originate from a test generator. Again, as for massive random generation, the wealth of the default Genesys Testing Knowledge (see Section 3.3.3) will cause those long tests to have a high verification value.

2.3 Verification Process

At the beginning of the design process, bugs are easy to find. There is therefore no point in bombarding the design with a large number of tests. This would only overwhelm the verification team with a huge number of failures. Similarly, *unsystematic* testing, i.e. running tests without specific properties, is equally inefficient. It induces a non homogenous debugging process, in which some parts of the design arbitrarily receive more attention than others.

Testing should be done incrementally to avoid a non homogenous debugging process, which would complicate the debugging. The initial PRS should be very simple, small and systematic. For example, it is recommended to start with tests that are aimed at single instructions and limited to the simplest processor mode, and then have the test complexity evolves at a pace which matches the design status.

Once relative stability has been reached, coverage subsets can be run. Later, as bugs become rare, massive random testing should begin. Then, when the hardware is running, it will be used for further testing (i.e., long tests, huge applications such as Operating Systems).

The overall process should be interleaved with some means of coverage. It should serve as a feedback mechanism for additional testing and as a reliable measure for deciding when sufficient confidence has been acquired.

2.4 Bug Driven Activity

The discovery of a functional design bug is an important event. There are a few lessons to be gained from it. These stem primarily from the two main properties of design bugs (and bugs in general): temporality and locality. This points out that, on the one hand, the test which revealed the bug should be retained in order to make sure that the bug will not re-occur (temporality) and, on the other hand, additional tests should be built for adjacent design areas (locality). For the latter task, the core of the bug should be understood and serve as a basis for additional tests which would also tackle related areas.

At this point, it is important to point out that the legality of a test often expires. In other words, a test that is perfectly legal at some point in time, may become invalid after changes have been made to the design. Since changes are often made during the design cycle, the lifetime of a test is typically limited. This fact points out that important tests, such as bug finders, might need to be “adjusted”. This can be a very time-consuming task. Section 4.2. shows how this problem can be overcome using Genesys.

Finding a bug can also serve as a feedback mechanism for the Verification Plan. Had the bug been found by some random testing, it might point out that there is a hole in the Verification Plan. It should have induced tasks, and thus tests, covering the discovered failure. This could also lead to the addition of new tasks for other related potential failures.

Section 4.2 shows how Genesys, by generalizing bug data and driving more tests, naturally lends itself to bug driven activity in general, and to the locality property, in particular.

3.0 Genesys

The main objective of this section is to present the basic concepts and properties of the Genesys test-program generator. The general structure of the Genesys system is described in Section 3.1, while Section 3.2 points out its main properties.

3.1 General Structure

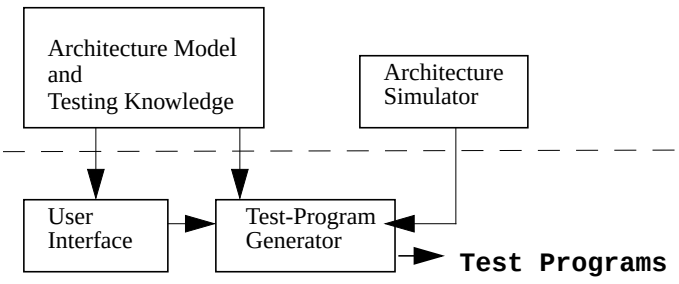

FIGURE 1. System components and interactions

Genesys is a model-based test-program generator which dynamically generates tests using a generation-simulation cycle for each instruction. It has been widely used during the last seven years at many IBM sites all over the USA. It is also used as a major verification means by several other companies [3]. The system has already been employed to model several processor architectures and to verify their implementation.

The system consists of three basic interacting components: a generic, architecture-independent test generator which is the engine of the system, an external specification (the model) which holds a formal description of the targeted architecture, and a behavioral simulator which is used to predict the results of instruction execution (see Figure 1 in which the User Interface is depicted as the fourth block). The user can control test generation by specifying desired biasing towards special events. This biasing can also be saved to a user *directives file*.

The external specification model also allows incorporation of testing knowledge. It is employed in order to generate probing test cases and allows expert users to add knowledge to the system in a local and relatively simple manner.

3.2 Major Properties

The following subsections describe the main properties of Genesys, especially stressing those which will be of primary importance for implementing the verification methodology.

3.2.1 Control

Genesys enables the creation of programs ranging from completely deterministic to totally random programs. By the means of a directives file, control is given to guide the generation to the desired extent, while any parameter not explicitly constrained is randomly set to any consistent value. The ability to target the whole range of test randomness is of key importance for implementing the proposed methodology. Namely, the methodology induces the creation of fully random tests, specific scenarios, and test templates with varying levels of randomness (see Section 4).

3.2.2 Testing Knowledge

Another major feature of Genesys is its ability to easily acquire additional Testing Knowledge (TK). The incorporation of TK into a random generator enables you to adjust the distribution of the probability of the targeted test space. As a simple example, the result of zero for an ADD instruction is typically of special importance while its relative probability to occur randomly is practically inexistent. Informing the test generator that the result of zero is important, and should thus be generated with a reasonable probability, is an example of the process of adding TK to the generator. As a more advanced example, many architectures define instructions with two memory operands (e.g., string instructions in x86). Having the two memory locations overlap is usually an exceptional event rarely occurring randomly, and worthwhile to add as TK to the generator. More generally, adequate weights should be given to corner cases which otherwise would be occurring with negligible probability. In Genesys, TK is in the form of Generation and Validation functions (written in C) and is held in the external specification model. The most noteworthy property of Genesys is that this TK can be incrementally added by the user itself to bias the creation of any test. The TK can be linked to the data, length or address of any resource and it can influence the generation of any event. In such a way, the scope and sharpness of Genesys' test-programs are truly unlimited: they are determined by the investment made by its users. Depending on how it is defined in the specification model, TK can be taken into account during random generation (then called *default* TK) or only when specifically requested (called *specific* TK). The Testing Knowledge paradigm is reported at length in [1].

3.2.3 Model-Based

Genesys relies on a formal and declarative model of the architecture captured separately in its architecture model component. This structure allows to conveniently integrate architecture changes, which are so common during design development. More generally, it enables to adapt Genesys to a wide range of different architectures, with minimal effort. The latest trend of designing a *hybrid* microprocessor (e.g., Merced which mixes x86 with VLIW) emphasizes the importance of having a tool which is easily customized to any architecture.

3.2.4 Short Tests

Contrary to typical test cases which assume initiation from the Reset state of the design, Genesys allows you to start a test from any (legal) state. It enables you to bring the design directly into a state which otherwise would have been relatively hard to reach. The sole output of the generator is a test file which consists of a sequence of instructions starting from a given initial state, and a section of expected results describing the expected values of the various processor resources. Both properties, initial state and expected results, contribute to the production of short, easy-to-debug and incisive tests.

4.0 Implementing the Methodology Using Genesys

The goal of this section is to demonstrate how Genesys integrates and promotes the methodology described in Section 2. Although a large part of this methodology is independent of the particular test generator used, some of the most noteworthy characteristics of Genesys contribute significantly to the successful implementation of the proposed methodology. More specifically, even though all the major properties appearing under Section 3.2 contribute to the overall quality of the verification process, two of them will be shown to be particularly important: first the ability to keep directives files instead of tests, and second the possibility to incrementally and externally add testing knowledge to Genesys. Instead of keeping tests, directives files, which may have varying degrees of control, impart the desired randomness and solve the test legality problem. More exclusive to Genesys, the testing knowledge frame-

work enables to optimize the quality of each of the stages induced by the methodology. In the following subsections, we will follow these stages and explain the role of Genesys in them.

4.1 Getting Started

Given a deep understanding of the architecture and an overview of the principal design properties, the first goal is to compose a Verification Plan. At this point, all of the different expected behaviors, including every corner case, are identified and listed as targets for testing. They should be translated into TK, including default TK, within Genesys, especially if their probability to appear at random is relatively low. In such a way, all the different identified behaviors are easy to target, yet will appear randomly with a reasonable probability.

Then, there is a need for the development of the periodic regression set. As indicated previously, it is desirable for such a test set to be different in successive runs. This can be achieved by composing the set using Genesys directives files instead of static tests. In such a way, a countless number of different tests can be induced. Moreover, randomness will be profitably added to the process.

It should be observed at this point that having *short* tests, due to the initial state and expected results structure, makes the entire debugging process much more efficient. This is true throughout the verification process, but it is of particular importance in the beginning stages. Contrary to standard self-checking tests that start from Reset, simple and short tests should focus on elementary goals. There is no need to get involved in complex Reset sequences and redundant self-checking segments.

Following periodic regression, coverage subsets should become available. A random test generator, capable of generating a countless number of different tests, can assist in the elaboration of such subsets. Given a coverage model, massive generation can be filtered to keep only those tests involving new coverage tasks belonging to this coverage model. As an example for a coverage model, a subset can be built by using standard software techniques to cover the code of an architecture simulator (such as the one present in the Genesys system).

4.2 Testing Knowledge Encapsulation

Both the Verification Plan and the Testing Knowledge evolve together with the design. They must be upgraded on two main occasions. First, when a certain architecture behaviour is being implemented in the design in an unpredictable manner. Nothing in the architecture book hints that this behaviour requires special attention. This event is of particular importance as it is a significant source of numerous escape bugs. It is obviously the designer's duty to convey the information to the verification team. Thus, beyond the elaboration of a few tests by the designer itself, this should cause the Verification Plan to be updated, and several tests to be created around the newly identified corner cases. Genesys, in particular the TK paradigm, provides very efficient groundwork for dealing further with such cases. By directly upgrading Genesys with this additional TK, the feature will continue to be tested randomly, within many different contexts.

Second, when a bug is found by chance, not as a result of the scheduled verification tasks. There is some similarity between new design features and such a bug discovery. In both cases, a new corner case is identified. Therefore, upon bug discovery, the TK should be increased as well. In addition, to tackle the locality property (Section 2.4), once the essence of the bug is distinctly grasped, it should be captured in a Genesys directives file, leaving all other biasing directives random. Consequently, this directives file can be seen as a generic engine to target not only the bug, but all the nearby design regions surrounding the bug as well. Incidentally, keeping directives files (instead, or in addition to tests) also solves the "test legality expiration" problem described in Section 2.4.

4.3 Massive Testing

In later stages of the design process, massive random testing is performed both in the simulation environment and directly on hardware (given an appropriate interface). The *default* TK present in Genesys allows you to tackle all the different design behaviors, even though no effort is done to control the generation process. Genesys allows the combination of quality and quantity, thereby achieving a large number of *smart* cycles.

5.0 Case Study: An x86 Design

Due to its preponderance and complexity, the x86 architecture is a suitable candidate for corroborating the methodology suggested in Section 2. This methodology was applied during the development of an x86 microprocessor within IBM, and relied on the recent availability of Genesys-x86, a Genesys-based pseudo-random test generator for the x86 architecture. Although genericness is one of Genesys' properties, significant effort was needed to support a complex architecture such as the x86. One of the main challenges presented to Genesys by the x86 architecture was the presence of instructions with variable-size and complex addressing modes [8]. Section 5.1 reports the principal insights gained. Section 5.2 demonstrates how Testing Knowledge encapsulation could have prevented the recent Pentium bugs.

5.1 Applying the Methodology to x86 Verification

The main insight gained is that confining verification to (usually purchased) test suites and long applications such as operating systems, is far from sufficient. The full test repertory described in Section 2.2 is needed for thorough verification.

Figure 2 displays the number of bugs uncovered at one of IBM sites by Genesys-x86 tests, in comparison to tests originating from different test sources but mainly commercial test suites (the upper curve being Genesys). Genesys' tests come from both the periodic regression and the massive random testing. It should be noted that Genesys-x86 was employed *after* the test suites had been comprehensively exercised on the design simulation environment. This highlights the fact that the

bugs found by Genesys-x86 might not have been found otherwise, at least not within an appropriate time frame. It is noteworthy to imagine the erroneous conclusions (and their implications) which would have been reached from relying only on bug curves other than Genesys'. It was surprising to witness the simplicity of some of the bugs left by those suites and later discovered by Genesys-x86. For example, a regular jump to high memory (beyond 2^{16}) revealed a bug showing that this simple operation was not exercised by any of the suites. This is partly due to the lack of randomness inherent in any given static set of tests. Whatever their size and claimed coverage, there are always simple bugs which stay untargeted. Of course, Genesys-x86 also revealed many bugs resulting from the complex interactions of different instructions and even units of the design. As an example of a more subtle bug, Genesys generated a test where a MTCR0 changing the processor mode (from protected to real) appeared in a non taken leg of a JMP instruction. This mode switching was not correctly cancelled upon returning to the right instruction stream.

The hardware was functional at first tape-out and successfully ran DOS and other operating systems. As expected, the test subsets coming from Genesys kept finding bugs at a high rate after this stage. Again, the reason is that, even though running large applications such as operating systems is undoubtedly a good confidence builder and a threshold which has to be successfully passed, they do not exercise a significant portion of the architecture. Many areas are left untouched, leaving potential bugs untargeted. In particular, many different initial states proposed by Genesys' tests are never reached.

5.2 Pentium Bugs

It is interesting to check whether the described methodology and Genesys-x86 would have prevented known escape bugs such as the last 2 Pentium FP bugs (the FDIV bug and the FIST bug [9]). Both were implementation dependent bugs which could not have been suspected from merely reading the Pentium architecture book. Consequently, a mandatory precondition for uncovering these bugs is that the unsuspected algorithm implemented by the designer is revealed to the verification team, but this is exactly the requirement described at the beginning of Section 4.2. We see then that such expensive bugs reflect a hole in the verification process. It could be a methodology problem suggesting a lack of coordination between the designer and the verification team, or it could stem from a lack of means to sufficiently integrate the newly revealed algorithm within the verification process. In any event, Section 4.2 explains how the suggested methodology would have dealt with these problems by introducing the implementation-specific information into Genesys' Testing Knowledge.

Nevertheless, there is another less obvious way through which Genesys-x86 might have discovered these bugs. Accumulated experience has shown that there are some cases which are typically more bug-prone in design. For example, although in general it is not clear from the architecture definition, long sequences of 0's or of 1's in source operands or in intermediate results, are often the source of bugs in today's FP designs. This observation can be easily translated into Testing Knowledge within Genesys. Thus, analysis of bugs, in an intra-architecture manner, leads to the definition of what can be called *generic* Testing Knowledge. Incidentally, having the long bit sequence TK in Genesys-x86 causes the tool to generate tests which uncover both Pentium bugs with a reasonably high probability. Table 2 below shows a range of values which cause the Pentium II FIST32 bug ('x' and 'y' can be replaced with '0' or '1') [9]. A similar sequence also caused the FIST16 bug.

The probability of randomly generating such a mantissa for a normal floating point number is $1{:}2^{32}$, as you need a 31 '0' sequence with a '1' before it (the leading '1' is a must for a normal FP number). Using the bit sequence pattern, the probability to hit this type of mantissa drops to approximately $1{:}2^{7}$. For the exponent part, using the bit sequence pattern reduces the probability from $1{:}2^{15}$ to $\begin{pmatrix} 15 \\ 12 \end{pmatrix}$ or 1:910 taking into account the sign bit. Hence, the probability of getting this event using the bit sequence pattern is around $1{:}2^{17}$ (~1:117,000). As a result, one should expect to find this type of FP bug after generating a few hundred thousand FIST32 instructions when the bit sequence pattern is used, whereas, using pure random generation, the probability to hit the bug is $1{:}2^{49}$ which entails the unrealistic need to generate thousands of billions of FIST32 instructions.

6.0 Conclusions

A rigorous methodology for tackling functional microprocessor design verification has been described. The principal contribution of this paper is to set the general guidelines for obtaining a compound functional verification framework, and to describe how this can be optimally implemented by a test-program generator such as Genesys. The goal of the paper is to assist in the composition of verification processes which typically include only part of the testing suggested in this framework. Although related and having a recognized importance, the fields of formal verification and coverage are beyond the scope of this paper.

It has been shown how an incomplete methodology applied to x86 microprocessors, i.e. running only existing suites of tests and long applications, or missing coordination between designers and the verification team, can lead to false conclusions on the design's correctness. In addition, the two main properties of Genesys, namely external Testing Knowledge and comprehensive control through directives files, have been shown to naturally suit the described functional verification methodology.

Type	Description	Design Stage	Goal
Periodic Regression Set (PRS)	A frequently run test regression. Small but comprehensive enough to provide useful feedback on the design status. Should be adapted to the state of the design.	Starting early in the design	Barometer of the design status
Specific Tests	Tests which pinpoint some hard-to-reach corner cases.	Small number of bugs in PRS	Verify corner cases
Coverage Subsets	A library of tests having some known measure of coverage. Should be used each time a major design change is performed. A subset for a given part (such as FPU) may be run when the part changes.	Design is relatively stable	Quality checker. Assurance vs. changes
Full-Load Testing	Standard tests run after the design was artificially put under a full-load state, such as a full complex pipeline.	Unit verification	Verify the unit in a full-load state
Random Testing	Massive unconstrained random testing produced by a random test program generator.	Design is stable	Overall design check. Final search for bugs
Hardware Testing	Running large applications such as operating systems, and massive random generation of long tests.	Hardware is functional	Final verification on product itself

TABLE 1. Type of tests and their usage

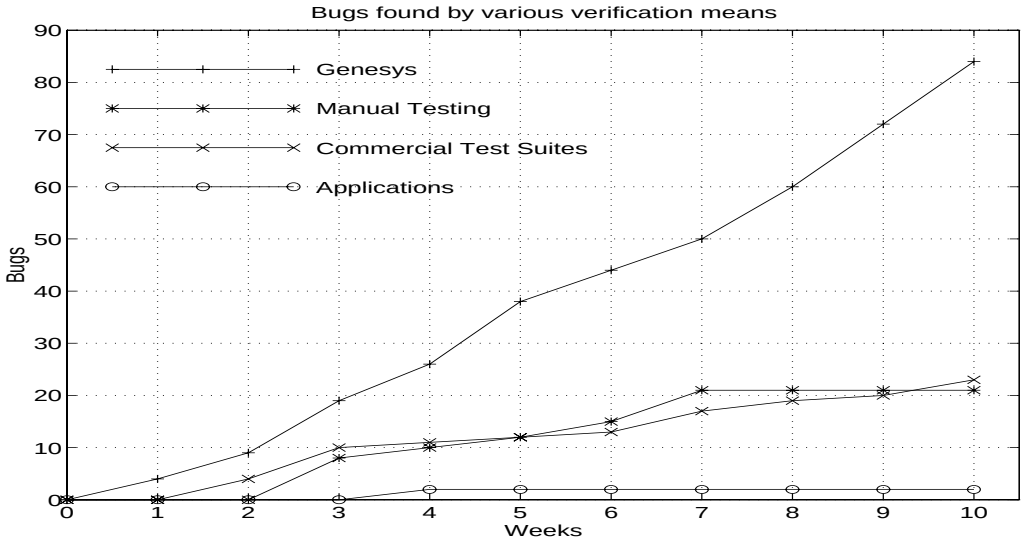

FIGURE 2. Comparative evolution of bugs found at an IBM site (Genesys-x86 vs. other verification means).

Sign	Exponent	Mantissa
1	100000000100100	1xxxxx10000000000000000000000000000yyyyyyyy

TABLE 2. Template for an FP number which uncovers the Pentium II FIST32 bug

References.

[1] Y. Lichtenstein, Y. Malka and A. Aharon, “Model-Based Test Generation For Processor Design Verification”, Innovative Applications of Artificial Intelligence (IAAI), AAAI Press, 1994.
[2] H.P. Sharangpani, M.L. Barton, “Statistical Analysis of Floating Point Flaw in the Pentium Processor”, Intel Corporation, 1994.
[3] F. Casaubieilh et al., “Functional Verification Methodology of Chameleon Processor”, 33rd Design Automation Conference, Las Vegas, June 1996, pp. 421-426.
[4] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, “Test Program Generation for Functional Verification of PowerPC Processors in IBM”, 32nd Design Automation Conference, San Francisco, June 1995, pp. 279-285.
[5] A. L. Sangiovanni-Vincentelli, P. C. McGeer, A. Saldanha, “Verification of Electronic Systems”, 33rd Design Automation Conference, pp. 106-111**.**
[6] J. Monaco, D. Holloway, R. Raina, “Functional Verification Methodology for the PowerPC 604 Microprocessor”, 33rd. Design Automation Conference, pp. 319-324.
[7] M. Kantrowitz, L. M. Noack, “I’m Done Simulating; Now What?”, 33rd Design Automation Conference, pp. 325-330.
[8] D. Lewin, L. Fournier, M. Levinger, E. Roytman, G. Shurek, “Constraint Satisfaction for Test Program Generation”, IEEE 14th Phoenix Conference on Computers and Communications, 1995.
[9] Intel Pentium II flag erratum, Intel home page at: http://developer.intel.com/design/news/flag/tech.htm
[10] G. Ganapathy, R. Narayan, G. Jorden, D. Fernandez, “Hardware Emulation for Functional Verification of K5”, 33rd Design Automation Conference, pp. 315-318.