Developing an Architecture Validation Suite
Application to the PowerPC Architecture

Laurent Fournier, Anatoly Koyfman, Moshe Levinger
IBM Research Lab in Haifa
{laurent, koyfman, mosh}@vnet.ibm.com

Abstract
This paper describes the efforts made and the results of creating an Architecture Validation Suite for the PowerPC architecture. Although many functional test suites are available for multiple architectures, little has been published on how these suites are developed and how their quality should be measured. This work provides some insights for approaching the difficult problem of building a high quality functional test suite for a given architecture. By defining a set of generic coverage models that combine program-based, specification-based, and sequential bug-driven models, it establishes the groundwork for the development of architecture validation suites for any architecture.

1 Introduction

IBM has set up a certification process for designs aimed at implementing the PowerPC architecture. This process grants PowerPC licences to designs that comply with PowerPC architecture. For this purpose, one of the defined criteria is the successful running of a representative set of functional test programs, called the PowerPC Architecture Validation Suite (AVS). Our goal was to build a test set which, on a successfully passing, induces the highest possible confidence that a given design indeed implements the PowerPC architecture.

Although the AVS has been built for the PowerPC architecture, its framework is generic and can be applied to the development of such a suite for any architecture. In fact, this paper focuses on providing the groundwork for the development of architecture validation suites in general, rather than elaborating on details appropriate to a specific architecture (e.g., PowerPC). The PowerPC architecture itself serves as a probing example of the suitability and efficacy of the proposed scheme.

As an architecture suite, the AVS puts aside design-dependent details. However, since it is run on designs and not on architectures, its ultimate purpose is to uncover bugs. Therefore, beyond the obvious covering of the architecture, the goal was to take some generic properties shared by current designs and to include them in the coverage models. For example, superscalar and out-of-order executions of programs are typical aspects of today's designs; the AVS attempts to capture and cover some of their complexity. In short, the AVS, while naturally overlooking specific design dependent properties, extends its expected architecture scope towards generic design properties. This dichotomy, separating design dependent properties from the more generic ones, induces a desired black-box/white-box structure within the verification process.

Numerous suites of tests are commercially available, notably for the x86 architecture [3]. They are typically very expensive and yet they do not usually have a clear quality guarantee. Their main attraction lies in the fact that they have already been used in successful design processes. However, their lack of defined coverage measurements renders them to be of little benefit. They merely constitute a threshold to be passed and do not reveal much regarding the real state of the design [4]. For example, a test possessing a simple branch from low memory to high memory detected a bug in an IBM x86 design after all the existing suites of tests had been successfully run.

In contrast, the AVS uses clear coverage measurements and proposes a new model for architecture validation suites. It consists of test programs yielded by a combination of three different coverage domains: specification-based, program-based and sequential design-bug models. The specification-based model is the best known and most widely used; it requires the test programs to cover a list of predefined tasks. The program-based model uses a behavioral simulator of the design, and applies software coverage techniques to induce coverage subsets. Finally, the sequential-bug model stems from a study of bugs that have escaped in the past [7] and an overall understanding of the generic properties of state-of-the-art mechanisms in today's designs.

The AVS also includes subsets targeted at multiprocessor designs, but their report is beyond the scope of this paper.

The rest of the paper is organized as follows: Section 2 briefly reviews the main tools necessary for construction of an AVS. Section 3 introduces the different coverage models which constitute the foundation of the overall compliance subset. Section 4 elaborates on the properties of the AVS and the results obtained for PowerPC. Some concluding remarks appear in Section 5.

2 AVS Tools

The process of building an Architecture Validation Suite involves creating a set of tests which cover all the targeted coverage models. Given those models, the development of an AVS requires the means to perform three major tasks: test creation, test validation and coverage evaluation. This section briefly surveys the corresponding tools used for PowerPC AVS construction. The focus is on the key capabilities needed for edification of any AVS. Section 2.1 describes the overall scheme used to build the AVS. To create the tests, we used an automatic pseudo-random test generator, Genesys [1][2], which is introduced in Section 2.2. A short description of the test format is included. Section 2.3 discusses the importance of the reference model while Section 2.4 presents the tool used to measure coverage.

DAC 99, New Orleans, Louisiana

2.1 Construction Scheme

An automatic test generator is mandatory to cope with the huge number of tests required to cover all the tasks derived from the coverage models (see Section 4.1). Given such an automatic generator, we adopted the following simple scheme: massive pseudo-random generation of tests is filtered to keep only those tests that contribute to the coverage of new tasks. More optimized schemes for keeping tests might be theoretically appealing **[6]**, but the relatively small gain they provide (in terms of more compact suites) was deemed unnecessary. Due to the richness of the test generator used (Genesys), this simple method enabled us to reach a high coverage percentage (in the order of 95%). For the remaining uncovered tasks, specific tests (in contrast to the ones coming from pseudo-random generation) had to be added. This last step is by far the most expensive, both in time and in the expertise required. Indeed, covering those last tasks typically takes more than 90% of the overall time, while acute expertise both in the architecture and in the generation tool is usually needed as the uncovered cases are often complex corner cases. As a positive side-effect, the detection of such holes serves as feedback used to enhance the generation tool.

2.2 Genesys - the Test-Program Generator

The PowerPC AVS tests were generated by Genesys, a model-based test generator which dynamically generates tests using a generation-simulation cycle for each instruction. The system is termed 'model-based' as a formal model of the architecture lies at its heart.

The system consists of three basic interacting components: a generic, architecture-oblivious 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. The external specification model also allows the incorporation of complex testing knowledge.

Genesys enables the creation of programs ranging from completely deterministic to totally random. 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 output of the generator is a test file which contains 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 at the end of the test. Genesys tests are therefore not self-checking and do not start from reset. This helps provide short, easy-to-debug and incisive tests. In order to accommodate environments not adapted to this kind of tests, the AVS has also been translated into a Self-Checking AVS (SCAVS) composed of self-checking tests starting from reset.

2.3 The Reference Model

There is a need for a tool that verifies the correctness of each test, i.e., a reference model which can distinguish between valid and invalid tests. This is done using a behavioral simulator that implements the PowerPC architecture. Even though such simulators are usually available for each design (see Section 3.1), no reference model is available for the pure PowerPC architecture itself. The distinction is subtle, but worth elaborating upon. Indeed, the transfer from a simulator written for a specific design to a pure PowerPC simulator is a surprisingly far from being a straightforward process. A very deep understanding of the architecture is needed to spot all those cases left undefined by the architecture. Moreover, even after having compiled and implemented all the identified "undefined behaviors", there is no additional reference model for double-checking. New problems can thus be found (and have been found) with each new design that implements an undefined case in an unexpected manner. To make the problem tangible, Table 1 shows an example of a tough-to-detect undefined case.

The development of an accurate pure architecture simulator is a critical step as it is the ultimate validator of all the AVS tests. We have not found a way to automatize this step, however, one of the major lessons learned from this work is that this step should not be underestimated. It is definitely a costly mistake to start building tests before a significant effort has been made on the simulator itself. Indeed, finding a bug in the pure simulator means that all the existing tests should be revalidated, possibly causing some tests to be removed and replaced to preserve coverage. Given the subtle nature of bugs in this context (undefined behavior) and the high cost of correction, it is worthwhile to start by investing a significant period of time on the reference model.

2.4 Comet - the Coverage Measurement Tool

Comet **[8]** is a new coverage tool developed at IBM's Haifa Research Lab. Its main property is the separation between the coverage model definition and the generic coverage analysis engine. This enables the user to employ a single tool for all the coverage models. The user can define the exact coverage model targeted and enjoy at the same time all the benefits of a powerful coverage tool. The definition of the models is written in SQL, while the huge quantity of data is handled by a DB2 database.

3 Coverage Models

Coverage is defined as any metric of completeness with respect to a test selection criterion. The coverage models represent the basic foundation of the suites. Their appropriate selection is therefore the most crucial step in the establishment of a coverage suite **[9]**. Given the models, the derivation of the tests may require some sophisticated tools (see Section 2), but it is a purely technical process. The first step in selecting the models has been to partition the coverage tasks into tasks derived from individual instructions and tasks derived from sequences of instructions. For individual instructions, a combination of program-based and specification-based coverage has been adopted (Sections 3.1, 3.2). The more complex tasks stemming from sequences have been tackled using a bug-model (Section 3.3).

3.1 Program-Based Coverage

Using a program that implements the application to be covered, program-based coverage uses software testing techniques to yield some coverage measurements. A PowerPC behavioral simulator has been used for the development of the program-based coverage subsets of the AVS. The basic rationale behind this approach is that the behavioral simulator can be viewed as a formal, correct and complete representation of the architecture. This is because very early in the verification phase, the behavioral simulator is used heavily, debugged and tested. Assuming then that a behavioral simulator is a representation of an architecture, the problem of measuring the quality of a test subset with respect to the architecture can be reduced to evaluate its quality with respect to the behavioral simulator. Implementation in PowerPC and properties of the program-based approach are described in Sections 3.1.1 and 3.1.2, respectively.

3.1.1 Program-based coverage of PowerPC architecture

The program-based subset of the AVS was obtained by covering

the code of the reference model. We used standard software control flow coverage criteria, such as branch and multi-conditions, using available evaluators. The first observation from applying those models is that relatively small subsets of tests are obtained. This compactness stems from the fact that, in real-life programs, the same block can be covered via many different types of inputs. For example, it might be reasonable to assume that the same block for floating-point exception handling is used for all the floating-point instructions. Furthermore, in some cases, different instructions (such as add and subtract) can use the very same code lines almost all the time. This gave us the incentive to improve those basic models by applying them *separately* to each architecture instruction of the simulator. Using this approach, code, which is shared by multiple instructions (and this is common), must now be covered by each instruction. These models subsequently cover the operation of each instruction. This strategy gives rise to significantly bigger subsets and may catch functionality that is implemented differently in the design. Of course, it added a level of complexity since the problem of knowing whether some code can be reached by some function (instruction) is, in general, undecidable due to data dependency.

3.1.2 Properties of the program-based approach

The program-based approach has the following advantages:

1. **Well-defined measure.** The program-based approach provides a well-defined level of design testing. This property is rarely found in simulation-based verification.
2. **Subset size control.** Software models of gradual complexity can be applied, resulting in subsets of gradual sizes. For instance, the entire spectrum, starting from the simple statement coverage to the unreasonable path coverage, is theoretically available. Our attempt to apply statement coverage for *each* instruction is an example of the advance along the axis of this spectrum. Size is an important property as it is directly linked to the time needed to run the subset. In practice, a hierarchy of subsets can be built, where the appropriate level selected for use is dictated by the amount of time available.
3. **High level of abstraction**. This method maintains a high level of abstraction in contrast to methods based on directly covering the design code. It focuses on the design behavior as specified by the architecture, and is not disturbed by the huge number of details brought by the implementation itself. High level programming languages allow compact representation of the architecture books. Typically, the size of a behavioral simulator is one hundredth of its corresponding design (in terms of lines of code). This allows relatively cheap coverage evaluation.
4. **Correctness and relative completeness.** As stated above, correctness and relative completeness stem from the fact that the simulator has typically been extensively used for other purposes at early stages of the design.
5. **Homogeneity/uniformity of verification.** Program-based coverage naturally induces a homogeneous level of testing throughout all the architecture behavior.

This approach has two main disadvantages: some features present in the architecture might be overlooked because of the dependence on implementation, and missing code can not be found by code coverage.

3.2 Specification-Based Coverage

A more standard way of inducing a partition of the input domain of a program is to review its specification and derive a task list which covers all the listed properties. As programs are usually specified informally (usually by means of a textual document), derivation of the task list must be done manually.

3.2.1 Specification-based coverage of PowerPC architecture

As the PowerPC architecture books span a few hundred pages, establishing a task list has been a tedious process. We first listed the domains that needed to be covered: the instructions, the address translation mechanism, the interrupt mechanism, and so forth. Then we further partitioned the targeted task space by identifying "important" properties within each domain. The importance of these properties is directly underlined by the architecture itself. The following examples are typical properties:

- Following the IEEE classification, the input domain of a floating-point register was divided into 10 types: +/- zero, +/- denormal, +/- normal, +/- infinity, SNaN, and QNaN.
- The address translation scheme defines many types of different behaviors, including different kinds of exceptions (protection violation, page faults, etc.).
- There are many different interrupt types in the PowerPC architecture. Each type includes many different possible causes.

In general, the list of tasks for each domain will include the enumeration of all possible permutations of the identified properties. For instance, the input domain for a 2-input floating-point instruction was divided into 100 groups, covering the cross-product of the 10 possible values for each of the two operands.

3.2.2 Properties of the specification-based approach

The specification-based approach completes the program-based approach by being tightly coupled with the architecture properties. Each of the desired properties is assigned its set of coverage tasks. As a result, the coverage task is independent of a specific implementation.

On the negative side, a manual scan of a processor architecture is a long and error-prone process. Because several people usually develop the task list, the process is exposed to oversights of the reviewers, and classification of the input domains may be done inconsistently.

3.3 Sequential Bug-Model Coverage

3.3.1 Background

Modern computer architectures are complex. A typical architecture includes hundreds of instructions, a few dozen resources (main-memory, general-purpose registers, special-purpose registers), and complex functional units (e.g., floating-point, address translation).

Aggressive performance requirements drive designs to perform complex algorithms in order to improve their instruction per second (IPC) ratio. For example, superscalar mechanisms, in which instructions are routed in parallel through several pipelines, have become popular. Out-of-order and speculative execution of instructions are other examples of now standard, but complex, design mechanisms. Maintaining the correct execution of instructions in such designs is a very challenging and bug-prone process.

PowerPC architecture, among other RISC architectures, favours the implementation of these complex mechanisms by using a fixed-sized instruction set, where each instruction performs a relatively simple task.

There are no comprehensive models for human design errors. It was found though, from an analysis of design bugs which were

uncovered late in the verification process of several PowerPC designs [7], that a relatively large number of bugs are associated with superscalar-related mechanisms. We derived two instruction sequential models for design bugs from this analysis.

The information gathered points out that short sequences are uncovering a high number (45%) of the reported bugs. The sequences include resource dependencies or interleaving instructions from different logical groups with all types of precise interrupts (including no interrupt). Table 2 presents a description of a few bugs belonging to these categories.

Clearly, models requiring the coverage of all these sequences are attractive. Indeed, by focusing not only on the replication of test scenarios for bugs discovered late in IBM designs, such a methodology attempts to generalize the underlying causes for such design bugs, and should therefore be suitable for predicting the location of a large amount of potential bugs in other designs as well.

3.3.2 The bug-models

We included two design-bug sequential coverage models: Interdependency model and Interleaving Instructions with Interrupt model.

The importance of checking sequences of instructions that have resource dependencies cannot be overstressed. The interdependency coverage model lists, as tasks, every possible pair of instructions which use the same resource as an operand. The operand can be either input or output, and the resource can be either a specific register (both general-purpose and special-purpose), or a memory location. The instructions should be 'close' in time, meaning that only up to two instructions can appear between them in the sequential list of instructions. This maximizes the probability of creating hazards in pipelined designs.

In contrast, the need for covering sequences which interleave instructions from different logical groups with all kinds of interrupts is less intuitive. This results from the fact that an interrupt is typically a traumatic event requiring distinctive handling, especially in a design implementing out-of-order execution. Hence, exercising each type of interrupt is obviously important, but the type of instructions causing the interrupt and immediately following it is also important. The instructions were grouped using multiple criteria. We started with the simple architecture classification and added more elaborated reasoning induced from experience with previous designs (e.g., idempotent or instruction duration). The coverage tasks of this model included pairs of instructions from all possible groups, with all possible precise interrupts in between them. Precise interrupts [5] are associated with the execution of a specific instruction, and thus are more difficult to serve in the context of out-of-order implementations. All the instructions which follow the interrupted instruction in the program order must cancel execution, even if their execution has already been started or even completed. Again, the two instructions should be relatively 'close' in time.

3.3.3 Properties of the bug-model approach

Three main properties are distinguished:

- As opposed to program-based and specification-based approaches, the bug-model approach focuses on a different aspect of the compliance validation. Instead of testing the correct execution of the properties defined in the PowerPC architecture per se, it targets complex mechanisms in the design's implementation. These mechanisms are empirically known to be bug-prone.
- Definition of the models, and grouping of the architecture facilities (instructions, interrupts) are done manually, based on empirical statistics, and not on well-defined criteria.
- For the instruction-interrupt interleaving model, a refinement of the partitions (up to a singleton group for each instruction) can yield higher quality coverage models, on account of the much larger test suites.

4 AVS Structure, Properties and Results

4.1 Structure

The AVS consists of various types of tests packaged in several subsets. A subset consists of several test files. All tests in a subset are of the same type; they all consist of target testing of the same area in the PowerPC architecture using the same type of coverage criteria, and they all depend on the same assumptions regarding the alternative behaviors proposed by the architecture (see Section 4.3). The architecture instruction set was separated into four major domains: Floating Point, Fixed Point, Branches and Load/Store. Therefore, when we covered this set using specification-based or program-based coverage criteria, we generated separate subsets for each one of the above-mentioned domains. The tests that cover the two bug models are not included in either of the mentioned domains; they appear in separate subsets.

Separate test suites were completed for 32-bit (about 87,000 tests) and 64-bit (about 150,000 tests) designs. See Table 6 for a complete description of the test suite.

4.2 Properties

It is obvious that no set of tests can yield complete confidence in a design's correctness. No matter how big and comprehensive the set is, it is always easy to show that some potential bugs are left untargeted. Therefore, Architecture Validation Suites cannot replace the full verification process, but they can be seen as an important constituent of this process [4]. The following subsections discuss several significant properties of the AVS.

4.2.1 Black-Box testing

The testing offered by the AVS leaves aside design dependent properties. This mimics the well known black-box/white-box partition of software testing, where specification and details belonging to the code implementation are tested separately. Beyond the structural benefit, this approach in microprocessor designs is not only adequate for multiple designs implementing a given architecture (as in the PowerPC example), but also for successive versions of the same architecture (for example, the x86 architecture). Indeed, the AVS will catch the nucleus of the verification while clearly-defined verification items will be left for each new implementation.

4.2.2 Quality of the AVS

Program-based and specification-based approaches complement one another when covering the behavior of individual instructions. This yields a mature solution for individual instructions. In contrast, the coverage of bugs related to sequences and stemming from intricate (but common) design mechanisms is much more complex to apprehend. The two models proposed here are a preliminary attempt to tackle this very challenging issue. They illustrate the framework which should be greatly enlarged to better cover the abovementioned mechanisms (out-of-order and parallel instruction execution) and to capture other mechanisms as well. Escape bugs should be studied and should induce the definition of additional models. As an example, further studies [7] have shown that many bugs (around 15%) are typically found due to speculative execution of instructions, as in not taken legs of

branches. A description of such a bug appears in Table 3. This points out that there is a need to define a model which captures most of the speculative execution bugs encountered. In general, the framework proposed by the AVS should augment the number of bug-models in order to grasp most of the complex design mechanisms responsible for bugs.

4.2.3 Genericness and alternative behaviors

Architectures typically leave many alternative behaviors for their implementation. Since the AVS is intended to be implementation-independent, the initial trend was to avoid the presence of such behaviors within the suite. In general, the suite should be generic in the sense that any implementation should be able to run it successfully. However, it soon became apparent that restricting the AVS to shared behaviors was very limiting: many important behaviors present in most implementations would be left unchecked in this manner. As an example, the 32-bit architecture enables implementations to work with up to four Gigabytes of physical memory, while it requires only a minimum of eight Megabytes. Restricting all the tests to the small physical memory would clearly significantly decrease the verification scope of the suite. An even tougher example is related to memory alignment exceptions: PowerPC defines a dozen memory alignment exception cases where, in each one, two different behaviors are architecturally correct (access success or interrupt).

Therefore, the following scheme was adopted: The core of the suite includes only tests that can be run on any implementation, i.e., they do not include behaviors where alternative options exist, thereby concurring with the most restrictive constraints. To complete the suite, specific subsets were added to cover the untargeted areas. Those subsets can be run conditionally: only those implementations that fulfil the subset assumptions are required to use them. Table 4 shows the example of memory alignment handling for which each optional case is handled by a separate subset.

There are also cases where the architecture leaves a behavior completely undefined. It is clear that the AVS should not include any test exhibiting such a behavior, as different designs may take different, a priori unknown, decisions.

4.3 Results

The AVS has been run by a dozen different PowerPC designs, both within and outside of IBM. As stated, the primary purpose of the suite was to grant a PowerPC certificate license. However, after a few designs successfully passed the suite, it became known as an efficient bug detecting test suite, and designs within IBM began to incorporate it as an integral part of the verification process itself. In fact, in recent designs, it has become a tape-out criteria. Therefore, its very effectiveness resulted in the loss of its value as a final compliancy checker. Needless to say, designs that used the AVS during the verification process, had no difficulty in passing the AVS barrier, thereby causing it to deviate from its original purpose.

This situation stresses the need for coverage subsets during the verification process, as pointed out in **[4]**. Since the AVS tests were generated by a random test generator, the solution was to generate a second AVS out of the same coverage models, but with potentially different tests. This strategy would ensure that the corrections induced by running the first AVS were globally correct, and not only pinpointed for overcoming the given tests.

As expected, a high number of bugs - in the order of a hundred - were found when the AVS was used as a verification tool. Its usage as a final checker revealed significantly less bugs (around ten). An example of such an escaped bug appears in Table 5.

Running the AVS on different designs uncovered many problems in the AVS itself, mostly due to cases where it was unclear that the architecture had left possible multiple behaviors. Correcting the tests was a very expensive process which underlined the importance of having a correct reference model from the initial stages of test generation (see Section 2.3).

5 Conclusion

The main contribution of this paper is to provide a generic groundwork for the development of architecture validation suites with clearly defined coverage measurements. While the covering of individual instructions is relatively well understood and performed, the covering of sequences of instructions, which becomes critical due to the dependencies introduced by complex design mechanisms, is much less mature. This paper attempts to tackle the sequence issue by studying existing bugs and deriving coverage models that encapsulate not only those bugs but the underlying causes stemming from complex (albeit common) design mechanisms. The two models proposed are a first step towards a more generalized family of models directed toward all generic design mechanisms.

Bibliography

[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] 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, DAC 95, San Francisco, pp. 279-285.

[3] M. Scheitrum and A. Smith, Behavioral Verification and its application to Pentium Class Processors, PCI Developers' Conference, 1995.

[4] Y. Arbetman, L. Fournier, M. Levinger, Functional Verification of Microprocessors Using the Genesys Test Program Generation - Application to the X86 Microprocessor family. DATE 99.

[5] D.A. Patterson and J.L. Hennessy, Computer Organization & Design The Hardware/Software Interface, Morgan Kaufmann, San Francisco, 1994.

[6] E. Buchnick, S. Ur, On Minimizing Regression-Suites using On-Line Set-Cover, EuroStar 97.

[7] Y. Abarbanel-Vinov, S. Ur, Processor Bug Classification and Modelling, Internal IBM Haifa publication.

[8] S. Ur, A. Ziv, R. Grinwald, E. Harel, M. Orgad, User defined coverage - A Tool Supported Methodology for Design Verification, DAC 98.

[9] S. Ur and A. Ziv, Off-The-Shelf Vs. Custom Made Coverage Models, Which is the one for You? STAR98. May 1998.

Appendix. Tables

Book I Par. 4.3.5 (Floating-Point Data Handling and Precision):	**Book I Par. 4.4.1 (Floating-Point Invalid Operation Exception):**
3. Single-Precision Arithmetic Instructions All input values must be representable in single format; if they are not, the result is placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1) is undefined	An Invalid Operation Exception occurs when an operand is invalid for the specified operation. The invalid operations are: • Any floating point operation on a signalling NaN (SNaN)
Question: Should an Invalid Operation Exception (which is a defined event) occur if one of the operands of the single-precision arithmetic instruction is SNaN, not representable in single format, or should the result be undefined?	

Table 1. Hidden undefined behavior

Architecture Type	Category	Short Description
Hybrid PowerPC/X86	Instruction group interleaved with exception (idempotent instruction - interrupt - any instruction).	On an interrupt, the return address register has not been updated with the address of the next instruction. Instead, it received the address of the current instruction, causing it to be executed twice. To find the bug, you need to have the interrupt on an instruction which gives different results when executed twice (idempotent).
32-bit PowerPC	Memory Interdependency	A load bypassing the DCBF (a Cache instruction) can cause stale data to be returned.
64-bit PowerPC	Register Interdependency	The instruction subf. (a type of subtraction) does not update the CR register correctly. As a result, the branch conditional following subf. goes wrong.

Table 2. Bugs inspiring the sequence models selection

Architecture Type	Category	Short Description
64-bit PowerPC	Speculative execution	While the processor switches from real mode to translation mode, it posts a speculative ifetch request. The request is not committed. The address is based upon the processor being in the real mode. The pending request is committed immediately after completion of the mode switching. This may result in there being no address corresponding to the requested address.

Table 3. Speculative execution bug

Event	Alternative1	Alternative 2
The operand of a floating point load or store is not word-aligned.	Interrupt	Correct execution
The operand of a fixed-point doubleword load or store is not word-aligned.	Interrupt	Correct execution
The operand of **lmw, stmw, lwarx, ldarx, ... , eciwx,** or **ecowx** is not aligned.	Interrupt	Undefined results
l**mw, ... , stswi,** or **stswx** instruction and the processor is in Little-Endian mode.	Interrupt	Correct execution
Elementary operand or string load or store crosses a protection boundary.	Interrupt	Correct execution
The operand of **lmw** or **stmw** crosses a segment or BAT boundary.	Interrupt	Undefined results

Table 4. Alternative behavior on unaligned accesses

Two input values will cause a wrong result when used as operands of an Fctid instruction. The values are 432FFFFFFFFFFFFF and C32FFFFFFFFFFFFF (these are two opposite numbers. This case was generated to cover a RESULT=0 coverage model task).

Table 5. Bug found by the AVS

No	Coverage Model	Number of Test Cases												Comments
		B		FL		FP		LS		Other		Total		
		32	64	32	64	32	64	32	64	32	64	32	64	
1	PB	64	88	693	661	622	1476	352	695			1731	2920	No address translation
2	PB	222	235	1303	1418	1792	2200	888	1206			4205	5059	With address translation
3	PB									113	137	113	137	Cache, SCU and Interrupts
4	PB									148	1698	148	1698	Design specific resources
5	SB	795	2635	24994	52470	3338	10823	1966	4194			31093	70122	No interrupts
6	SB									2455	2564	2455	2564	Cache, SCU and Interrupt
7	SB									2494	4199	2494	4199	Design specific resources
8	BD	59	81	3401	3581	14985	19312	2600	2886	19810	31988	40885	57668	Resource Interdependencies
9	BD									6247	7179	6247	7179	

FP	**Fixed Point Instructions**	**32/64**	**32/64 -bit Implementation**	**LS**	**Load/Store Instructions**	**BD**	**Bug-Driven**
B	**Branch Instructions**	**FL**	**Floating Point Instructions**	**PB**	**Program-Based**	**SB**	**Specification-Based**

Table 6. PowerPC AVS structure