Random Self-Test Method
Applications on PowerPC^{TM} Microprocessor Caches

Rajesh Raina
Motorola Inc.
raina@ibmoto.com

Robert Molyneaux
IBM Corporation
bobmoly@ibmoto.com

Somerset Design Center
6200, Bridgepoint Pkwy#4
Austin, TX 78730

Abstract

This paper describes a novel method for generating test stimuli for digital systems. By taking advantage of certain properties of the Design Under Validation, the method can be used to generate test stimuli that is random as well as self-testing. We discuss the requirements and limitations of this method on practical designs. The use of this method for High-Level Design Validation of caches in PowerPC^{TM} microprocessors is also described. The paper concludes by identifying areas where further work is needed.

Topic Areas: High-Level Design Validation, Silicon Validation, Pseudo-Random Testing, Microprocessor Testing.

1. Introduction

Generation, Application & Response-Evaluation of Test Stimuli, are important issues in the effective functional validation of today's complex designs. This paper introduces a novel method for generating test stimuli for a general class of Digital Systems. The method takes advantage of certain properties of the Design being Validated, towards generating test stimuli that is random as well as self-testing.

The most popular form of Design Validation uses Directed Pseudo-Random Test Stimuli [1-4]. This method, as practiced, is summarized in Section 2 along with its strengths and limitations. The most serious limitation - the difficulty in determining the expected response to random stimuli - is discussed.

Random Self-Test principle and method are described in Section 3. This is followed by a discussion on the advantages, applications and limitations of this method.

In Section 4, the Random Self-Test method is applied to PowerPC Microprocessor Caches. The overall utility and potential of Random Self-Test technique is discussed in Section 5, including a comparison with Directed Random Testing method.

2. Directed Pseudo-Random Testing

Digital Systems are tested (for manufacturing defects) and verified (for design flaws) in three primary ways, two of which are shown in Figure 1:

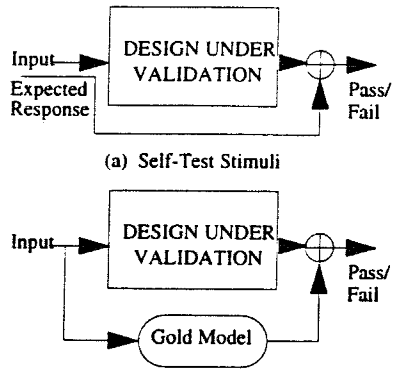


Figure 1: Test Stimuli Generation

(a) With the use of self-test stimuli:

The test/verification engineer manually creates the stimulus and its expected response. The advantage is that the resulting tests are compact and targeted precisely to the areas requiring testing. The disadvantage is that this manual process is very slow; and for large designs, is prohibitively slow.

(b) With the use of pseudo-random stimuli:

The input stimuli is automatically generated, in large volume, from a Pseudo-Random Pattern Generator [5]. The expected response from the design being tested (or validated) is compared with the response, to the same stimulus, of a design known to be good - or the gold model. This may be done cycle-by-cycle or after fixed time intervals (signature comparison). The advantage is that the design can be tested with stimuli that is "several orders of magnitude larger" than what many engineers could ever generate manually. The disadvantage is that a "gold" model is required.

For certain *combinational* designs, pseudo-random testing can be effective, perhaps as a supplement to self-test stimuli or by biasing the input stimulus [6]. However, most large designs - such as Microprocessors, DSPs & Multi-Media

0-8186-8409-7/98 $10.00 © 1998 IEEE

Hardware - are sequential machines. For *sequential* designs, pseudo-random testing can be hopelessly inadequate and inefficient. It could test certain portions of a design repeatedly while leaving large portions of the design untested. Directed pseudo-random testing is able to solve this problem and is described next.

(c) With the use of directed pseudo-random stimuli:

Figure 2 shows the composition of a typical large sequential machine.

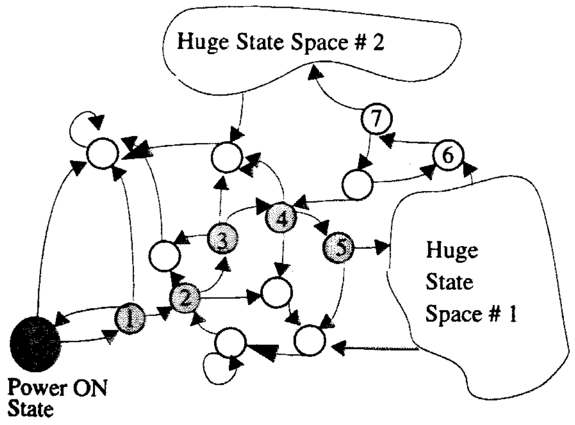


Figure 2: A typical sequential machine

Starting from the Power-ON state, it takes an extremely directed sequence (states 1 through 5) of state transitions to reach the interesting portion of the design marked as "Huge State Space # 1". More direction is needed (states 6 & 7) to enter another portion of the state space. These state spaces could account for 98% of the design functionality. A pseudo-random pattern stimulus, applied at Power-ON, will burn in the start-up states most of the cycles- exercising these states repeatedly - while missing out on the huge state spaces. Not a good investment of test time. Biasing the input stimuli, as described in BIST techniques [6], can help improve the odds of entering the huge state space. For design validation purposes, BIST restrictions are not applicable - consequently - optimized directed sequences (homing sequence) can be used to enter the desired state spaces wherein random testing can be applied.

In practice, a small test will be used to validate the peripheral states of the machine shown in Figure 1. This could even be a manually written self-test. Next, a template will be written where the input stimuli are fixed for the first five cycles after Power-ON. After the fifth cycle, the input would be randomized. The test cases generated from such a template will always land in the huge state space in the quickest possible time after which random testing would begin. Finally, templates will be written to traverse states 1 through 7 in the shortest possible time for each test case, in order to randomly test the huge state space # 2.

Validation and Test engineers spend considerable development time on creating templates for directed random tests. The development effort pays off as the test time is reduced considerably. Imagine testing a microprocessor with purely random test stimuli - 99% of the time, the microprocessor will quickly end up in the machine halt state due to an illegal instruction or an improper instruction sequence! Of course, for comparing the output response, a gold model is required for generating the expected stimuli.

The computer industry has embraced the gold model requirement so strongly that it is almost second nature to invest in gold model development for new digital designs. Furthermore, many new designs are only incremental improvements to existing designs - whereby the tried and tested existing design effectively serves as the gold model.

Despite the advances, and apparent ease of procurement of the gold model, in certain cases - an accurate gold model is hard to get. Sometimes, the effort to develop an accurate gold model begins to reach similar complexities as the development of the real design itself. If the design needs to be tested and verified on real hardware, a gold model is very costly and invariably orders of magnitude slower than the real design. Even when a gold model is available, engineers spend considerable time debugging false fails due to gold model inaccuracies & limitations early during the design phase.

Random Self-Test method is based on a careful study of the particulars of a broad range of digital designs. It identifies a class of designs where testing and design validation can be performed with tests that are random as well as self-testing; thereby obviating the need for a gold model while allowing automatic & hence large volume of test stimulus generation. The class of designs that will benefit from this method include Microprocessors, Microcontrollers & Multi-Media Chips.

In the remainder of the paper, we refer to pseudo-random testing simply as random testing.

3. Random Self-Test Method:

In Section 2, it was explained that when testing/validation is done with random stimuli, a gold model is required to validate the response obtained from the part being tested. The following describes a method by which random stimuli can be applied to parts fulfilling certain conditions, where the response is self-tested without requiring the use of a gold model. This section concludes with a discussion on the limitations of this technique.

Random Self-Test Principle: Let T represent the functionality of the module being tested and let **x** be a random input stimulus.

If

$$y = T(x)$$

and

$$x' = T^{-1}(y)$$

then if x != x' module T or T^{-1} is malfunctional.

We establish functionality for module T exactly the same way as is done with directed random testing. That is, if x = x' for sufficiently large number of random input values (analytically determined), then module T is functional with a certain confidence level [6]. The Random Self-Test principle is illustrated with a block diagram shown in Figure 3.

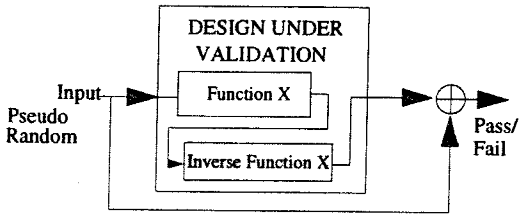


Figure 3: Self-Test with Pseudo-Random stimuli

The two main advantages of this technique are:

(a) Allows powerful random testing without gold model.

(b) Can be used on actual hardware (prototype/emulator).

The drawbacks of this technique are:

(a) Identical errors in T and T^{-1} are masked - Aliasing.

(b) Longer test stimulus is required.

(c) Limited application - every function being tested requires an inverse.

The aliasing problem needs to be studied further in a rigorous fashion. However, a preliminary study indicates that the aliasing problem, in most cases, is comparable to that found in the popular BIST method of Signature Analysis - which itself has been found to be acceptably low [7]. Furthermore, it has been observed that many high performance designs, in order to maximize parallel operation, invariably use separate hardware to implement functions and their inverses [8]. For example, the load data-path and the store data-path in PowerPC 604e^{TM} microprocessor do not share any hardware between them. This allows stores to occur simultaneous to loads. In such designs, the occurrence of identical design errors is reduced and hence the aliasing problem is mitigated even further.

In general, longer test stimulus is required because for every operation the test also needs to use its inverse operation to complete the self-test portion of the test. In most cases though, it would be possible to test an operation and its inverse mutually, in a single test.

For random self-test technique to work, every function being tested must have an inverse - direct or indirect. Therefore, this technique cannot be used effectively on designs with functions having no direct or indirect inverses. Fortunately, for Microprocessors and Microcontrollers - most of the functions are also accompanied with their inverses. For example, the **add** function has the **subtract** function as its inverse. A **rotate-left** function has **rotate-right** as its inverse. A **load-word** has a **store-word** as its inverse. Due to the balanced nature of Microprocessor & Microcontroller designs (there is an undo for everything you do), for many functions, - the inverse function is either directly available or can be indirectly created with a sequence of sub-functions.

The same is true for a new class of designs - the Multimedia chips. In many of these designs a function is accompanied by its inverse. A **compress** is accompanied by an **uncompress**. A **buffer** with a **debuffer**... and so on. This is especially true when testing is performed at the high-level (i.e., System Level). Nonetheless, it is important to be aware of the limitations of this technique.

The following is a simple illustrative example of how the adder function in a Microprocessor can be tested using this technique:

```
   Let  T = addq  and T-1 = subq
lda      r2, const(r31)  ; operand B, a constant, is loaded in r2
get_rv   r1              ; operand A, a random variable loaded in r1
addq     r1,r2,r3
stq      r3, 0(r4)       ; store result of A+B in memory
ldq      r3, 0(r4)
subq     r3,r1,r6          ; (r3 - r1 = r6)
xor      r6, r2, r7      ; compare actual (r6) with expected (r2) ...
bne      r7, error       ; and branch on error
```

Through each pass of the above sequence, the add function is tested with a new random variable (loaded in r1) as operand A. For every value of the random stimuli for operand A that is added to a constant (operand B), the inverse function will always return back the constant value for comparison. This sequence can be repeated millions of times. A subsequent test sequence could randomize operand B and hold operand A as constant.

It is clear that this technique - when applicable - can be used effectively with relatively low development costs for automatic validation of designs using random stimulus without requiring a gold model. Furthermore, this technique can also be used on actual hardware - where a comparable gold model is harder to develop.

Functions with indirect inverses:

A given design may not provide direct inverses for all its functions. Fortunately, in most such cases, the actions performed by a given function can be undone with a *sequence* of functions. The sequence of functions thus serves as an indirect inverse to the function being tested.

An example of a function with an indirect inverse is the Cache Load. This function is associated with Cache memories used in processor-class designs. A Cache Load moves data requested by the CPU from main-memory location **r**, into the cache location **s**, so that subsequent accesses to this location are faster.

This function typically has no direct inverse such as a Cache Unload that would move data back from cache location **s**, to main-memory location **r**. However, by performing another Cache Load to a *different* memory location **q**, that maps to the *same* cache location **s**, one is able to initiate the Victim Allocate operation (a.k.a. Castout [9]) that simply moves existing data from cache location **s** back to main-memory location **r** before moving new data into **s**.

Therefore, in the above example, the sequence of functions that serves as an indirect inverse to the "Cache Load address-**r**" function is -

```
Compute memory location q
    such that q =/= r and q maps to s.
Initiate Cache Load q
```

Functions with no direct or indirect inverse:

Certain functions or operations do not support a deterministic inverse for all input stimulus values. As an example, a *Saturated Add* function - used in DSP applications - does not have a deterministic inverse for all its input values [10].

A saturated add function adds two or more operands like a regular add function, upto a saturation point that is less than the maximum added value for the operand range. An add function without the overflow/carry bit, is an example of a saturated add (Figure 4).

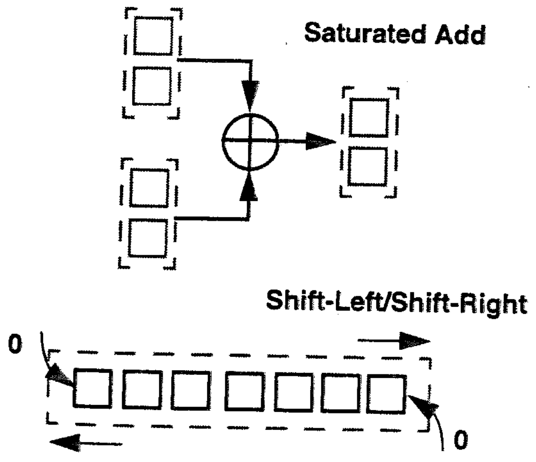


Figure 4: Functions with no inverse

A shift-left or a shift-right, where the shifted out bit is lost, are also operations without a deterministic inverse (Figure 4). At a first glance, these two might appear as inverses of each other. However, rotate-left and rotate-right are inverses but not shift-left/shift-right.

We described two functions that inherently do not support corresponding inverse functions. Graphically, a function may be viewed as a mapping from its input space to output space. All mappings from input to output that are *one-to-one*, theoretically have a corresponding inverse. The mappings from input to output that are *many-to-one*, cannot have a deterministic inverse. This is shown in Figure 5. It may be noted that correctly defined functions do not have *one-to-many* mappings from input to output space.

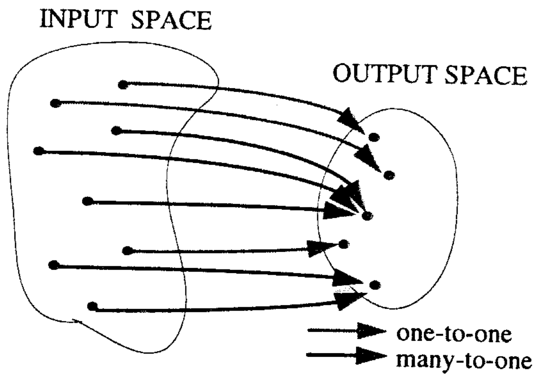


Figure 5: Function with many-to-one mappings

Random Self-Test approach may be used for partially validating such functions, but cannot be used for complete validation. Partial validation may be accomplished if the range of input stimuli is restricted to all *one-to-one* mappings from the input to output space. In the saturated add case, this implies restricting input stimuli to not cause the result to reach the saturation point. Partial validation may also be accomplished if the result, after applying the inverse function, is partially compared with the expected result. In the N-bit Shifter case, this implies comparing only N-r rightmost bits after applying an r-bit Shift-left followed by the corresponding r-bit Shift-right; where r is a random integer between 0 & N.

It has been observed that practical hardware implementations typically support inverse functionality for all functions where possible. An interesting question arises in cases where a design implements a function without its inverse, but for which an inverse is possible [11]. Should an inverse function be added to the implementation, even if it not required by the design, simply to aid random self-test? The authors hope to encounter a practical design case with such a situation and hence be in a position to assess its feasibility. The answer obviously depends on the *cost* of implementing a bare-bones inverse functionality for a given function and, to a certain extent, the *perceived* dependence on random self-test method for validating this function.

4. Applications on PowerPC Microprocessor Caches

This Section describes the use of Random Self-Test technique with two examples derived from the PowerPC Microprocessor family. The first example is the use of Random Self-Test for exercising the Data Cache and the second example describes an Instruction Cache thrasher. Figure 6 shows the conceptual view of the on-chip cache organization in PowerPC 604e microprocessor. It features 32K byte Instruction and Data caches. From the architectural level, the Instruction cache is read-only. The Data cache is Write-through - implying a CPU write is written to both the Data cache and the Main Memory.

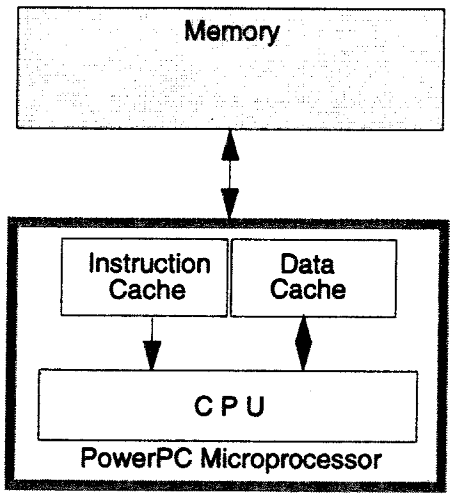


Figure 6: On-Chip caches on PowerPC microprocessors

4.1 Random Self-Test on the Data Cache

Consider a Data Cache with the following features:

- 8-way set associative
- Write through capability

The four major functional operations for the Data Cache are:

- Cache Load
- Address Tag Compare (Hit/Miss)
- Victim Allocate/Castout
- Write Allocate

Readers not familiar with cache design and operations are recommended reading any textbook on Computer Design, such as [12, 14].

A template for exercising the Data Cache with Random Self-Testing stimuli is shown in Figure 7 in a high-level flowgraph for easy understanding.

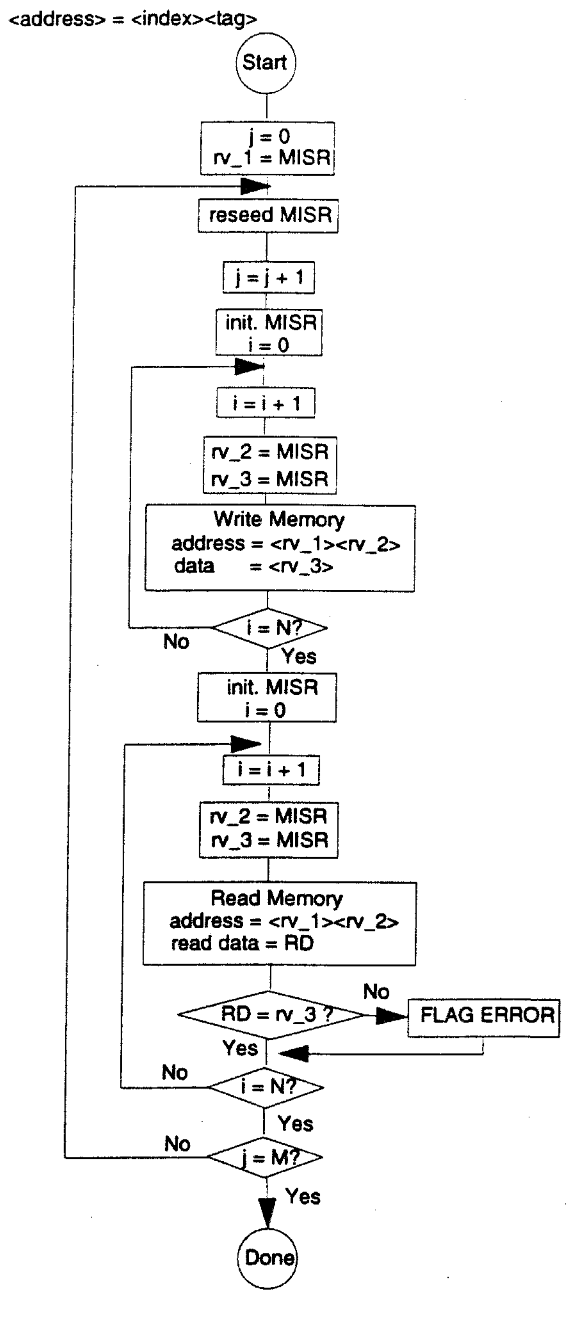


Figure 7: A Random Self-Test Flowgraph for PowerPC microprocessor Data Cache

MISR (Multi Input Shift Register) is used as a source for random stimulus. In the first inner loop, the random data is written to addresses with random tags but fixed index. Since the cache supports Write Allocate, all 8 ways of a particular index will quickly fill up and then exercise the Victim Allocate (Castout) function extensively. At the end of the first inner-loop, a total of N tag-random addresses will be initialized with random data of which the last 8 will reside in the cache and the remainder in main memory. Before the second inner loop is executed, the MISR is re-initialized such that the same N tag-random memory locations are read back and compared with expected data. In this loop, the Cache Load and Victim Allocate will be exercised extensively. The outer loop repeats the testing performed by inner loops on random index values thus complementing the tag-compare testing.

In practice, 6 to 8 templates could be developed for exercising all functions of the cache with variations in addressing, read/write sequencing, cache on/off testing and data-width granularity. Additionally, a practical validation/test environment will turn on background activity such as external interrupts & DMA requests to stress the cache design, in a fashion similar to Directed Random Testing.

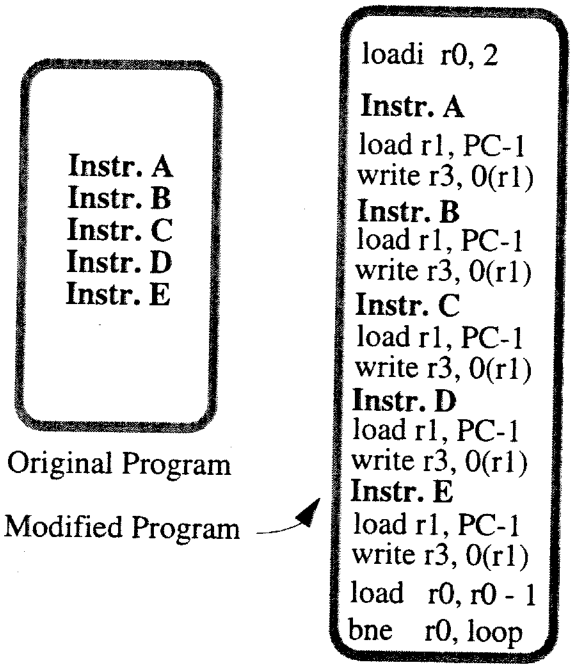
One may have noticed that the example template described here will pass when the data cache is completely non-functional (i.e., the CPU gets a cache miss all the time and read/write operations access main-memory). This is equivalent to the data cache being turned off. In general, such a gross design error will be validated with manually written tests as described in Section 3 (also see Figure 2). Manual test are short and easy to write for such cases. The biggest utility of random testing is towards validating design functionality for obscure machine-states. Due to large number of variables involved, manual test generation is not practical.

4.2 Instruction Cache Thrasher

A template for exercising the Instruction Cache is given below. The features of the Instruction Cache are the same as the Data Cache - except for Write Allocate and Write Back, which are not applicable (Instruction Cache is read only).

```
Take any existing program
Modify by appending following two instructions
after each program instruction
    load r1, PC-1    ; r1 <-- Program Counter - 1
    write r3, 0(r1) ; write r3 to location (r1)
```

Register r3 is initialized with an illegal instruction. The modified program is enclosed in a loop such that it runs twice. That's it. This simple template (dubbed Icache-thrasher) is capable of self-testing the Instruction Cache quite extensively using existing programs or randomly generated programs. An example of this procedure is shown in Figure 8.



```
loadi r0, 2
Instr. A
load r1, PC-1
write r3, 0(r1)
Instr. B
load r1, PC-1
write r3, 0(r1)
Instr. C
load r1, PC-1
write r3, 0(r1)
Instr. D
load r1, PC-1
write r3, 0(r1)
Instr. E
load r1, PC-1
write r3, 0(r1)
load r0, r0 - 1
bne r0, loop
```

Figure 8: Example use of Instruction Cache Thrasher

After each instruction (of the original program) is executed the first time through the program loop, it is loaded into the instruction cache. The following two instructions simply overwrite the instruction in main-memory with an illegal instruction. By the time the first loop completes, all cache-resident instructions should have an illegal instruction in the corresponding main-memory location. The second loop through the program validates all the major cache functions such as Cache Load, Victim Allocate and Tag Compare.

In practice, the Icache-thrasher considers a few more details in structuring the modified program to improve its efficiency, scope and diagnostic ability. For example, Register r3 may be initialized with the opcode of a write instruction, instead of an illegal instruction as described in the example. The write instruction is able to provide enough unique information to simplify diagnosis of failed functions or cache locations. Alternately, the illegal-instruction exception-handler can be rewritten to process every failed function/location before returning execution to the program.

Because the templates are self-testing and the stimulus is directed-random, one is able to run a given test on real hardware without need for a on-the-fly reference model or the pre-computed expected response. This is a big advantage. Ten seconds of Random Self-Testing based on 10-12 templates for both caches, yields 3 Billion cycles on a 300 MHz Microprocessor!

5. Discussion

Many digital designs, when viewed from a higher level, say system level, offer inverse functions for many of its supported functions. The Random Self-Test method described in this paper can be used with a certain degree of effectiveness in validation testing of such digital systems.

Although the proposed method may be used for manufacturing testing, the primary utility of this method is design validation testing on actual silicon [13]. For designs where this method can be used effectively, the economical advantages are enormous. When a gold model is not required, at-speed validation can proceed on actual silicon without need for creating expected responses from a gold model. There may be economic utility for this method even when a subset of a given design functionality is amenable for random self-testing.

Clearly this method is not suitable for validation or testing at very low levels of a design, because logical operations such as AND/OR/NAND do not have inverses. Furthermore, functions with no inverses, as described in Section 3, cannot be tested across the entire functional space. Floating Point operations also belong to this category because rounding-operation is a many-to-one mapping from input to output space.

Table 1 compares the Random Self-Test (RST) method against Directed Random Testing (DRT) on eight important features. We have seen that DRT requires a reference (gold) model while RST does not. This is the biggest advantage in favor of RST. For effective application, both methods require effort in developing templates for test stimuli. We believe template development effort for RST will generally be higher than DRT because the template must also include inverse functionality - and in many cases, only an indirect inverse may be possible.

Aliasing exists in both methods. Aliasing may originate from two sources in DRT - (a) the design and its gold model may have identical design error; (b) the effect of a design error may not be captured in the machine-state signature which is observed periodically for comparison with the corresponding signature of the gold model. RST has one source of aliasing - identical errors in function and its inverse will be masked. This is mitigated in high-performance designs that typically implement functions and their inverses with different hardware. We assign advantage to DRT because, the already accumulated, empirical data does not indicate aliasing to be causing design error escapes.

Both methods use pseudo-random test stimuli. Pseudo-random testing is inherently poor in efficiency. We assign advantage to DRT because the tests used in RST will be longer, and hence less efficient, due to every function requiring the inverse operation to complete the test.

The scope of DRT is as broad as the completeness of the reference model will allow. Fairly complete and robust reference models exist for many designs. A prior implementation of a given design may also serve as a reference model. On the other hand, we have seen that RST method cannot be applied to functions with no inverses. This is the primary disadvantage of RST.

Feature	Directed Random Testing (DRT)	Random Self-Test (RST)	Adva ntage
Reference Model	Required	Not Required	RST
Template Dev. Effort	Medium	High	DRT
Aliasing	Yes	Yes	DRT
Test Efficiency	Low	Low	DRT
Scope	Broad	Limited	DRT
Completion Criteria	Probabilistic/ Coverage Analysis	Probabilistic/ Coverage Analysis	even
Test Speed on Hardware	As Fast as Reference Model	As Fast as Hardware	RST
Existing Infra-structure	Stable/Vested	Nil	DRT

Table 1: Comparison of Test Methods

One of the biggest problems with random testing methods is the difficulty in establishing completion criteria. This is especially true for validation testing [1-3]. DRT and RST must use similar probabilistic and/or coverage-analysis methods to establish criteria and determine when testing is completed. Both methods are even in this regard.

Continuing design validation on actual silicon is an accepted and prudent industry practice for large and complex designs. With DRT method, the speed at which test stimulus can be applied is limited by the speed at which the expected response is computed from the reference model. The simulation speed for the reference model, for a typical PowerPC microprocessor, is 50,000 cycles/second. Therefore DRT can go no faster than 50K cycles/second. Because RST method does not require a reference model, it can operate as fast as real hardware. PowerPC microprocessors operate in the range of 180-400 million cycles/second. The tremendous speed-advantage enjoyed by RST is a derivative advantage of not requiring the gold model.

In terms of existing infrastructure, DRT enjoys a stable and vested environment. RST has not been used heavily quite yet.

Built-In Self-Test for Memories (M-BIST) implicitly uses the Random Self-Test principle. A data pattern is **written** to memory. Later the data pattern is **read** from memory and compared with what was written earlier. The **Write** and **Read** operations constitute the function & inverse pair. Although, many practical M-BIST systems use fixed data patterns, such as checkerboard, walking ones, etc, - the data pattern can very well be randomized for each write operation as long as the M-BIST system can conveniently remember what it wrote.

The Random Self-Test method described in this paper may also be viewed as a generalization of the M-BIST scenario described above.

Of the eight discernible features, we view "Reference Model" and "Test Speed on Hardware" as the biggest advantages of Random Self-Test over Directed Random Testing; and "Scope" as the biggest disadvantage of Random Self-Test. For the remaining features, Directed Random Testing may hold an advantage, but only by virtue of having been in use and practice for a long time.

4. Conclusions

The paper presents a novel method for generating test stimuli for validation of Digital Systems. The test stimuli is random as well as self-testing. The method takes advantage of the fact that Digital Systems, feature an inverse function for many of its supported functions. This is especially true when validation testing is done at a high-level (i.e., complete system level). The paper describes the use and limitations of this technique on microprocessor class designs using PowerPC microprocessor caches as examples.

It is hoped that the paper is successful in generating research and development interest in this promising area.

Acknowledgments:

The authors thank the PowerPC 604e microprocessor validation and test teams - Dawit Belete, Robert Bailey, Charlie Beh, Hemendra Talesra, Tom Austin, Tien Dinh, Suksoon Yong, Terri Fukuhara, Laura Powell, Richard Yin and Nathan Steinke for providing assistance with PowerPC microprocessor implementation details. The authors are also thankful to our esteemed colleagues - Heeru Thadani, Javier Prado, Jim Golab, Brad Burgess, Carl Pixley, Marvin Denman, Mike Snyder and Magdy Abadir for reviewing this work and providing constructive suggestions. We thank the GLS-VLSI'98 anonymous reviewers for providing insightful and useful critique. Finally we are thankful to Carlos Gutierrez, Mark McDermott, Moises Cases and Naras Iyengar for providing continued management support to this project.

References:

1. Hosseini A., D. Mavroidis & P. Konas, "Code Generation and Analysis for the Functional Verification of Microprocessors," *Proc. of the 33rd Design Automation Conference*, pp. 305-310, June 1996.
2. Monaco J., D. Holloway & R. Raina, "Functional Verification Methodology for the PowerPC 604^{TM} Microprocessor," *Proc. of the 33rd Design Automation Conference*, pp. 319-324, June 1996.
3. Kantrowitz M. & L. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor," *Proc. of the 33rd Design Automation Conference*, pp. 325-330, June 1996.
4. Aharon A., et. al., "Test Program Generation for Functional Verification of PowerPC Processors in IBM", *Proc. of the 32nd Design Automation Conf.*, pp. 279-285, June 1995.
5. Wagner K.D., C.K. Chin & E.J. McCluskey, "Pseudorandom Testing," *IEEE Trans. on Computers*, Vol. C-36, No. 3, pp. 332-343, March 1987.
6. Waicukauski J.A., V.P. Gupta & S.T. Patel, "Fault Detection Effectiveness of Weighted Random Patterns," *Proc. of the Int'l Test Conf.*, pp. 245-255, 1991.
7. Raina R. & P.N. Marinos, "Signature Analysis with Modified Linear Feedback Shift Registers," *Proc. of the 21st Fault Tolerant Computing Symp.*, pp. 88-95, June 1991.
8. Feedback from reviewer#1; GLS-VLSI '98.
9. Denman M., P. Anderson & M. Snyder, "Design of the PowerPC 604e^{TM} Microprocessor," *Proc. of the IEEE Compcon*, pp 126-131, 1996.
10. Pixley C., B. Burgess *et. al.*, Patent Disclosure Meeting, Somerset Design Center (Motorola/IBM), Austin, TX, Feb. 1997.
11. Feedback from reviewer#2; GLS-VLSI '98.
12. Stone H., "High-Performance Computer Architecture," Addison-Wesley Publishing Company, 1987.
13. Golab J., R. Raina, T. Dinh and N. Steinke, "Silicon Validation Methodology of the 0.25 micron PowerPC 604e^{TM} Microprocessor," *Proc. of the DesignCon - On-Chip Systems Design Conference*, Jan 1998.
14. PowerPC 604e RISC Microprocessor User's Manual, Motorola Press. June 1996.
    Web Address - http://www.mot.com/PowerPC/