Accelerating Microprocessor Silicon Validation by Exposing ISA Diversity

Nikos Foutris
Dept. of Informatics & Telecom.
University of Athens, Greece
nfoutris@di.uoa.gr

Dimitris Gizopoulos
Dept. of Informatics & Telecom.
University of Athens, Greece
dgizop@di.uoa.gr

Mihalis Psarakis
Dept. of Informatics
University of Piraeus, Greece
mpsarak@unipi.gr

Xavier Vera
Intel Barcelona Research Center
Barcelona, Spain
xavier.vera@intel.com

Antonio Gonzalez
Intel Barcelona Research Center
Barcelona, Spain
antonio.gonzalez@intel.com

ABSTRACT

Microprocessor design validation is a time consuming and costly task that tends to be a bottleneck in the release of new architectures. The validation step that detects the vast majority of design bugs is the one that stresses the silicon prototypes by applying huge numbers of random tests. Despite its bug detection capability, this step is constrained by extreme computing needs for random tests simulation to extract the bug-free memory image for comparison with the actual silicon image.

We propose a *self-checking* method that *accelerates* silicon validation and significantly increases the number of applied random tests to improve bug detection efficiency and reduce time-to-market. Analysis of four major ISAs (ARM, MIPS, PowerPC, and x86) reveals their inherent diversity: *more than three quarters of the instructions can be replaced with equivalent instructions*. We exploit this property in post-silicon validation and propose a methodology for the generation of random tests that detect bugs by comparing results of equivalent instructions. We support our bug detection method in hardware with a light-weight mechanism which, in case of a mismatch, *replays* the random test replacing the offending instruction with its equivalent. Our bug detection method and corresponding hardware significantly accelerate the post-silicon validation process. Evaluation of the method on an x86 microprocessor model demonstrates its efficiency over simulation-based and self-checking alternatives, in terms of bug detection capabilities and validation time speedup.

Categories and Subject Descriptors

C.4 [Computer Systems Organization]: Performance of Systems – *reliability, availability, and serviceability*. B.8.1 [Hardware]: Performance and Reliability – *reliability, testing, and fault-tolerance*. B.8.2 [Hardware]: Performance and Reliability – *performance analysis and design aids*.

General Terms

Design, Performance, Reliability, Verification

MICRO'11, December 3–7, 2011, Porto Alegre, Brazil.

Keywords

Microprocessor Silicon Validation, Design Debug, ISA Diversity

1. INTRODUCTION

Aggressive technology scaling and extreme chip integration make microprocessor validation a daunting task and a major bottleneck in the development of new architectures. The pressure on the validation team to deliver correct design in the marketplace on time is higher than ever although the combination of correctness and timeliness seems almost infeasible given the complexity of microprocessor designs ("infinite" validation space) and the available time-to-market windows [20], [23].

Pre-silicon design verification is mainly based on *simulation* at different levels of abstraction [6]. Despite its maturity and the tremendous utilization of computing resources, it is impossible to guarantee that all design bugs have been fixed before tape-out [18] because only a small number of functional scenarios can be simulated during pre-silicon verification. Statistics show that 12% of design bugs slip into first silicon prototypes [14] and almost 50% of microprocessor chips require extra unplanned tape-outs [3]. An ineffective validation process easily leads to product delays or even product recalls and a severely tarnishing in the reputation for the company. Thus, an effective post-silicon validation approach that promptly detects and eliminates design bugs before volume production can make the difference between success and failure of a modern microprocessor product [26].

The goal of post-silicon validation is to detect anything that may lead to incorrect operation: logic bugs, electrical or process-related bugs and mask-related manufacturing defects. Post-silicon validation runs a comprehensive collection of test programs on silicon prototypes in a real system environment. Huge numbers of tests run 24 hours per day for up to a year, feeding the microprocessor chip with a varied assortment of test scenarios at various frequency, voltage, and temperature operating ranges. Every time a bug is detected the validation team is fed with the failure data from the execution of the test to debug the design. When a sufficient number of bugs are detected and fixed, a new batch of prototypes (step) is manufactured and validation continues on the new samples. Post-silicon validation ends when time-to-market window demands volume production to start.

In this paper, we propose a post-silicon validation methodology for microprocessors with two major objectives: *apply more tests to silicon prototypes and detect bugs earlier*. The methodology does so, by exploiting the inherent *diversity* of microprocessor instruction sets (existence of equivalent ways to perform

operations) to mitigate the very expensive and time consuming simulation step.

2. POST-SILICON VALIDATION

2.1 Challenges

Several years of experience of microprocessor manufacturers have shown that the combination of design complexity with shrinking time-to-market windows, lead to numerous important design bug escapes (errata bugs) in production silicon despite the extremely large efforts of the validation team. All processors have several known errata bugs: the Intel Pentium® FDIV bug, the AMD Opteron™ REP MOVS bug, the IBM PowerPC® bug that caused some instructions to execute at lower frequencies only [30], and the recent Intel Sandy Bridge chipset bug in the SATA port [16]. The rate of errata bugs has more than doubled in the latest generation of Intel processors [11]. More than half of the bugs that slip into volume production have no fixes and for those that a fix exists, the in-field workaround is often too costly [5]. Thus, effective silicon validation methods are needed to ensure that forthcoming architectures do not suffer from severe bug escapes due to the following challenges. By effectively addressing these challenges the number of escaping bugs is expected to be reduced.

Simulation Limitations – Simulation offers excellent control and monitoring capabilities throughout the entire design, but the limited simulation throughput has always been a bottleneck in the microprocessors industry. Expensive server farms devote huge amounts of time and energy for simulation but only a small portion of the different modes of operation can be thoroughly excited before silicon. Table 1 summarizes the throughput of simulation, emulation and actual hardware execution [13].

Table 1. Simulation, emulation and silicon throughput

Approach	Throughput (instructions/sec)
System simulation	$\sim 10^3$
RTL simulation	$10^1 - 10^3$
Emulation	$\sim 10^5$
FPGA prototyping	$\sim 10^6$
Silicon	$10^7 - 10^9$

Validation tests applied to prototype chips range from random instruction tests (RIT) [9] to user applications [8]. The silicon validation phase that is based on RITs contributes tremendously to the detection of design bugs; 71% of the bugs in Intel's Core™ 2 Duo found in post-silicon validation are detected by RITs [9]. In a RIT-based validation a huge number of random instruction sequences (trillions of random instructions in total) are executed and aim to cover all possible architecture and micro-architectural scenarios defined by the programmer's reference manual.

RIT-based post-silicon validation is tightly coupled with a necessary simulation step and thus suffers from the simulation throughput problems [29]. A typical RIT-based validation flow involves the execution of tests on a golden reference model, which is an instruction-level accurate model of the processor (an architectural simulator) to produce the correct (golden or expected) responses. Correct responses must be known to compare them with the actual prototype responses; in case of a mismatch, bug-hunting begins.

Effective post-silicon validation of the future must mitigate the simulation of random instruction tests to save time, resources, and budget while not limiting bugs detection. *How does one know in a simulation-less method (without needing "golden responses") that a random test ran correctly in the silicon prototype?* Our methodology mainly contributes to this challenge proposing a self-checking technique.

At this point, it is important to emphasize on a fundamental requirement we set to ourselves for the proposed methodology: *keeping original random tests unmodified*. Several methods for the generation of effective random instruction tests have evolved throughout the last years [2], [3], [7]. Important problems in random tests generation have been addressed, such as avoidance of the creation of fixed patterns, which introduce interference into the test. Interference can spoil the test scenario and may hinder creation of more general patterns, thus reducing bug detection coverage. An example of interference is the case where a data processing operation (e.g. integer addition) is always paired with a data movement instruction (e.g. data loading). In this case, there is a high probability that the fixed instruction sequence spoils the contents of caches and leads to cache misses that the validation scenario is not expecting. *Our methodology leaves original RITs unmodified to fully utilize their bug detection capabilities*.

Post-Processing (Triaging) – Triaging is the step in which validation data (from silicon execution of random tests) is analyzed to identify common failure modes, which indicate potential design bugs. In a traditional RIT-based validation flow, for each failing random test a detailed simulation reproduces the test to identify the root cause of the mismatch. As reported in [17], there is a large possibility that multiple fails in the validation data are due to the same bug. However, the validation team will necessarily simulate all failing tests before recognizing the cause of failure. Such delays in bug identification can seriously impact time-to-market and hinder the detection of other bugs. Thus, after validation data is collected (memory array dumps from random tests execution), it is crucial to eliminate or reduce the "dirty" or "misleading" debugging records inside them and cluster common failure modes more effectively before the debug process begins. *How can the random tests execution in a self-checking method provide more useful validation data to the debug engineers?* We contribute to this challenge by providing detailed information about the offending instructions.

Blocking Bugs – Another major issue of post-silicon validation is dealing with *blocking* bugs. Mostly in the first stages of silicon validation (first prototypes), there are several bugs with blocking behavior [19]. A bug is a blocking one, if it can potentially mask out the discovery of other bugs, by stalling the execution of the subsequent tests (the rest of the validation plan goes wasted), because no workaround is possible for that bug. In such a case, validation proceeds only after bug-fixing, re-design, and new silicon prototypes arrive to the validation teams [12]. Volume production may be seriously delayed and the overall development cycle and time-to-market will be prolonged if multiple such silicon re-spins are necessary before a design is considered sufficiently bug-free. *How can a self-checking RIT-based method help reducing the effect of blocking bugs?* We contribute to this challenge by providing equivalent instructions workarounds for the offending instructions.

2.2 Scope of the proposed methodology

The proposed methodology is applied at the post-silicon validation phase of the microprocessor validation cycle for the detection of logic and electrical bugs. The proposed approach aims to detect failures in silicon prototypes (potentially corresponding to actual silicon bugs) through the comparison of equivalent instructions responses. Furthermore, it refines the

validation data provided to the debug process, by replaying the failing random tests. The contributions of the proposed methodology to post-silicon validation are the following.

- We introduce a novel methodology for the generation of *enhanced random instruction tests* (RITs) able to detect design bugs by comparing the results of equivalent instruction sequences. The methodology is therefore *self-checking* (does not need golden responses to compare with). A bug can make either an instruction or its equivalent to fail but a mismatch in the comparison denotes the existence of a bug in either case. Bugs can escape only when they affect the equivalent instructions in the same way. By generating equivalent instructions that do not activate the same hardware areas in the processor logic we minimize this probability. The identification of equivalent instruction sequences is a key enabler for the execution of subsequent random tests despite the existence of bugs, so our method inherently supports bypassing of *blocking bugs*. When an offending instruction is identified and while debug engineers look for fixes before new prototypes are produced, RIT-based validation can continue normally by avoiding the use of the problematic instruction and replacing it with its equivalent.
- We propose a light-weight hardware mechanism that records the mismatch between the results of two equivalent instructions to support *subsequent identification of the offending instruction*. Furthermore, the hardware mechanism contributes to the *reduction of validation data* forwarded to the debug engineers. After it records the mismatch location, the hardware mechanism replays the RIT replacing the offending instruction by its equivalent sequence to take full advantage of the RIT's failure detection capabilities: it allows continuation of the RIT execution and identification of additional failures. By utilizing the information about the exact location of mismatches during post-processing triage, the debug engineer can identify instructions or instruction classes that fail often and focus root cause analysis to particular structures of the microprocessor. Note that the hardware mechanism is complementary to the bug detection method (checking of equivalent instructions) and is optional.

3. ISA DIVERSITY ANALYSIS

Our bug detection philosophy is based on the existence of inherent equivalences (i.e. diversity) within instruction sets. We define ISA diversity as the extent to which operations of an ISA can be performed equivalently by more than one different ways. If the same input data is applied to equivalent instructions or instruction sequences, they will produce identical results, although they *activate different logic paths in the processor logic*. It is exactly this activation of different parts of the processor that enable bug detection by comparison (self-checking).

To identify the extent of ISA diversity in microprocessors, we analyze four popular instruction set architectures: ARM, MIPS, PowerPC, and x86 and provide examples of diversity and statistics for each ISA.

3.1 Diversity Examples

Table 2 lists examples of equivalent instruction sequences for ARM, MIPS, PowerPC, and x86 ISAs. In most cases, more than one equivalent instruction sequences exist for each original instruction, but only one alternative appears in Table 2. We use the actual assembly instruction mnemonics of each ISA to describe the equivalent code. For uniformity, we use the same generic names (RA, RB, RC, etc.) for general purpose registers. Note that whenever an equivalent code modifies a register – which is not affected by the original instruction – its value has to be saved before and restored after the execution of the equivalent code (the save and restore instructions are omitted).

Table 2. ISA diversity examples of four popular ISAs

ARM ISA diversity

Original Instruction	Equivalent Sequence	Description

mvn RA, RB
move not
	sub RC, RC, RC
	sub RC,RC,#1
	eor RA, RC, RB
		Uses exclusive OR operation and an all 1's mask stored in RC to invert the bits of RB.

mlas RA, RB, RC, RD *multiply and accumulate*	mul RA, RB, RC adds RA, RA, RD	Splits the complex operation into a multiplication and an addition.
smuad RA, RB, RC *dual 16-bit signed multiply with add*	smulbb RA, RB, RC smultt RD, RB, RC add RA, RA, RD	Executes two signed 16-bit multiplications in the bottom and top halves of the source registers RB and RC and then adds the intermediate products.
stmia RA!, {RB-RD} *store multiple increment after*	str RB, [RA] str RC, [RA, #4]! str RD, [RA, #4]!	Executes multiple single register store instructions which (except the first one) update the index.

MIPS ISA diversity

Original Instruction	Equivalent Sequence	Description
slt RA, RB, RC *set on less than*	sub RD, RB, RC srl RA, RD, 31	Executes subtraction and checks if the result is negative (R0 = zero register).
lw RA, addr(RB) *load word*	lhu RA, addr(RB) lhu RC, (addr+2)(RB) sll RC, RC, 16 or RA, RA, RC	Executes two load halfword unsigned instructions and places the second halfword to upper bytes.

MIPS ISA diversity (*cont.*)

lui RA, imm_{16}
load upper immediate
	addi RB, R0, 0xFFFF
	and RA, RA, RB
	addi RB, R0, imm_{16}
	sll RB, RB, 16
	or RA, RA, RB
		Resets the upper half of RA and then uses add and shift instructions to load the constant (R0 = zero register).

srlv RA, RB, RC
s*hift word right*
	rotrv RA, RB, RC
	sub RC, R0, RC
	addi RC, RC, 32
	sllv RB, RB, RC
	xor RA, RA, RB
		Executes rotate right instruction and then left shift to mask upper bits in the rotated result (R0 = zero register).

PowerPC ISA diversity

Original Instruction	Equivalent Sequence	Description
eqv RA, RB, RC *equivalent*	andc RD, RB, RC andc RE, RC, RB nor RA, RD, RE	Executes the logic operation: ¬ ((RB & ¬RC) | (¬RB & RC))
rldimi RA, RB, SH, MB *rotate left doubleword immediate then mask insert*	rldicr RC, RB, SH,63-SH clrldi RC, RC, MB rldicl RB, RB, MB,63-SH rotrdi RB, RB, MB or RA, RB, RC	Performs successive rotate and mask operations using other rotate instructions.

lwaux RA, RB, RC
load word algebraic with update indexed
	lwzx RA, RB, RC
	add RB, RB, RC
	extsw RA, RA
		Loads word first, then updates RB with the new address and finally extends sign.

cntlzd RA, RB
count leading zeros
	addi RA, 0, 0
	Loop: add RA, RA, 1
	rldcl RC, RB, RA, 63
	beq Loop
	addi RA, RA, -1
		Implements a loop that uses rotate and mask operations to count leading zeros. Each loop iteration rotates reg RB left by RA locations and clears the 63 upper bits (and update flags). If the result is zero continues, otherwise exits.

x86 ISA diversity

Original Instruction	Equivalent Sequence	Description

add RA, [m32]
integer addition
	fild [m32]
	mov m32, RA
	fiadd [m32]
	fistp m32
	mov RA, [m32]
		Moves data from integer register file to FP-stack and uses the FP addition instead of integer addition.

mov RA, [m32] *load data from memory into register*	push [m32] add ESP, 0x4	Next instruction that uses RA operand should load the data from the stack. Restore stack pointer (ESP).
clc *clear carry flag*	mov RA, 0x0 bts RA, 0x0	Sets RA to zero and performs a bit test and set instruction which clears the carry flag.

jmp target
jump to target address
	mov RA, 0x1
	cmp RA, 0x1
	je target
		Sets RA to a value and performs a compare instruction which activates the ZF flag. The conditional jump (je) is then used instead of jmp.

cvtdq2pd RA, [m64] *convert packed dword integers to packed double-precision FP values*	cvtsi2sd RA, [$m64_{low}$] cvtsi2sd RB, [$m64_{high}$] movlpd m64, RA movhpd m64, RB movlpd RA, [m64] movhpd RA, [m64]	Executes two convert dword integer to scalar double precision FP values instructions followed by two load operations. The intermediate results (low 32-bits, high 32-bits) are merged into the same register.

fadd [m64]
floating-point addition
	movlpd RA, [m64]
	fst m64
	movhpd RA, [m64]
	mov $m64_{low}$, 0x0
	mov $m64_{high}$, 0x0
	haddpd RA, [m128]
	movlpd m64, RA
	fld [m64]
		$m64_{low}$ and $m64_{high}$ are consecutive memory addresses filled with zeros. Modifies operands values and replaces the floating point operation by a packed double-fp horizontal addition.

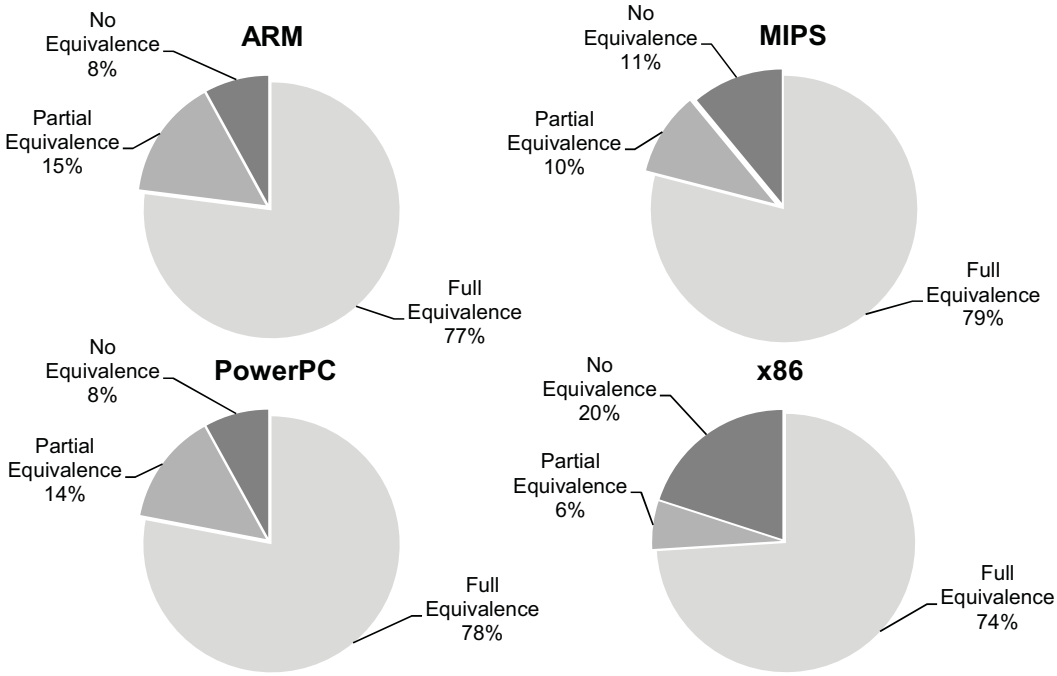

Figure 1. Diversity statistics of four popular ISAs

3.2 Diversity Statistics

We classify the instructions of the ARM, MIPS, PowerPC, and x86 ISAs in 3 categories.

(a) *Full Equivalence*: instructions for which there are one or more equivalent ways to realize their operation. This category includes the vast majority of arithmetic and logic instructions, data transfer instructions, compare instructions, and a large number of control flow instructions.

(b) *Partial Equivalence*: instructions for which there are partially equivalent ways to realize their operation. This is due to: (i) different modes of operation for these instructions some of which cannot be realized differently, (ii) inherent loss of accuracy in the operation of an instruction (floating-point conversions). This category includes some of the instructions not included in the previous category, a number of floating point instructions, and some data transfer instructions that involve system storage areas.

(c) *No Equivalence*: instructions with no equivalences. This category includes mainly the privileged instructions that access system resources, complex control operations, input and output instructions, interrupts, exceptions and very complex arithmetic instructions (of CISC architectures).

Figure 1 shows the statistics of our analysis for the four popular instruction set architectures where all instructions are classified in the three categories. For each ISA, our statistics present the percentage of different instruction *mnemonics* that fall in each category and not the different opcodes. In many cases (particularly in x86) the same mnemonic includes several tens of different opcodes. For the x86 ISA, we focused on the general purpose instructions set and not the special ISA extensions. It is evident from Figure 1 that all four ISAs have a large amount of instructions in the full equivalence category.

4. SILICON VALIDATION BY EXPLOITING ISA DIVERSITY

The proposed post-silicon validation methodology consists of four stages:

(1) Generation of the ISA diversity database.

(2) Generation of enhanced random instruction tests.

(3) Hardware replay mechanism.

(4) Post-processing (triage).

We propose a novel, self-checking, diversity-based, hardware supported framework to *accelerate* post-silicon validation and improve its quality. An overview of the framework is shown in Figure 2, where our major contributions are highlighted. A detailed analysis of each feature of the methodology follows.

(1) ISA Diversity Database Generation. The fundamental first step of the proposed framework is the identification of ISA diversity, i.e. microprocessor instruction equivalences. Identification of equivalent instruction sequences and population of the ISA diversity database heavily depends on the designer's knowledge about the underlying architecture (detailed knowledge of the architecture, micro-architecture and microcode of the design). For this reason it is very likely to provide high quality results given the insights that the design team has on the microprocessor architectural details. The database contains for each instruction a list of equivalent instruction sequences. (In our case, this step took approximately one and a half man month to study in detail the x86 ISA and identify its diversity; effort was much less for the RISC ISAs we studied – see Section 3).

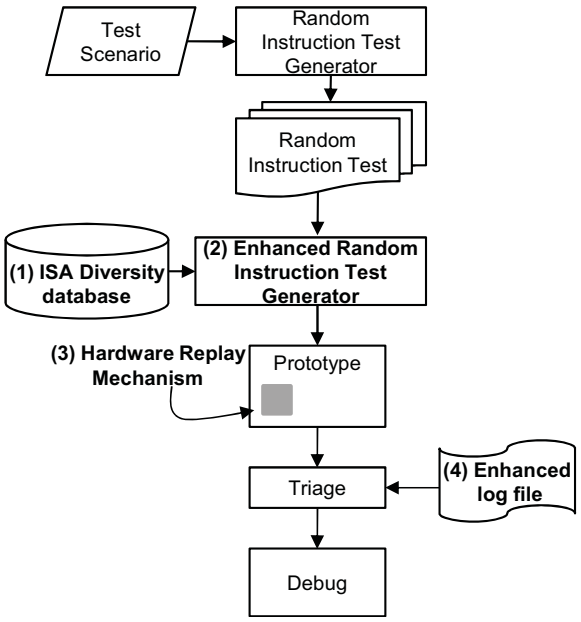

Figure 2. Proposed silicon validation framework

(2) Generation of Enhanced Random Instruction Tests. The validation flow is fed with effective Random Instruction Tests (RITs) already generated (but not simulated) by sophisticated random generators that all microprocessor companies internally use. We pair each RIT with an Equivalent RIT (ERIT), to implement our basic bug detection concept: bugs are detected in our method by comparing the execution results of a RIT and its ERIT (a mismatch indicates a potential silicon bug). An ERIT is automatically generated from a RIT replacing its instructions with their equivalent counterparts that have been stored in the ISA diversity database. When the database contains more than one entry with equivalent instructions for an original instruction, our approach randomly picks one of the alternatives. Instructions without equivalents are simply duplicated (in this case, only electrical bugs related to these instructions can be detected).

The enhanced RITs that our methodology automatically generates have the following structure:

```
/* Original RIT code starts here. */
   . . . . . .
/* End of Original RIT code. Original RIT responses have been
stored to memory. */
/* Equivalent RIT (ERIT) code starts here. */
   . . . . . .
/* End of Equivalent RIT (ERIT) code. ERIT responses have been
stored to memory. */
/* Checking code starts here. Compare RIT to ERIT responses. */
   . . . . . .
/* End of Checking code. */
```

Each enhanced RIT consists of the following:

(a) An *original RIT* which is left unmodified; we assume each RIT consists of a few thousands of instruction cycles as reported in the literature (each RIT we use in our experimental evaluation consists of 4K instruction cycles).

(b) An *equivalent RIT* (ERIT) generated as described above using the ISA diversity database. For each instruction in the RIT, an equivalent instruction or instruction sequence is inserted.

(c) A *checking code* that compares the stored results of the original RIT and the equivalent RIT to identify mismatches as indications of potential silicon bugs.

Bug detection in our method takes place by recording mismatches during silicon execution (through the checking code); therefore our method provides *immediate bug detection*. On the contrary, in a typical RIT-based flow, mismatches (due to bugs) are only detected very late and off-line when dedicated servers (i.e. validation host machines) compare the memory dumps (i.e. memory locations that are accessed or modified by the test, register files and any other data structures that can be scanned out from the silicon prototype) of the actual silicon execution with the expected memory dump contents from simulation. Figure 3 outlines the bug detection concept for the traditional and the proposed RIT-based silicon validation flow for microprocessors.

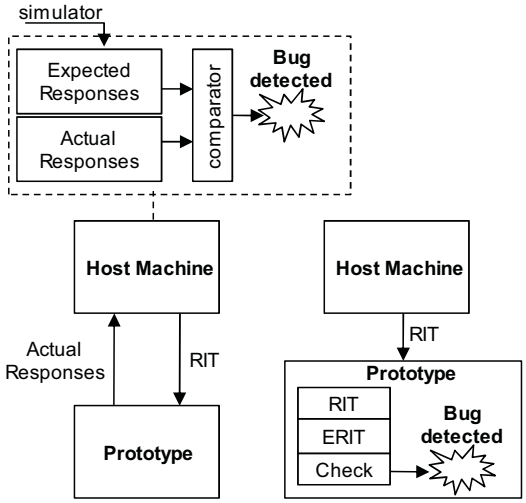

Figure 3. Traditional (left) vs. proposed (right) RIT-based validation flow

(3) Hardware Support for Validation. In our method we take advantage of the fast bug detection that takes place during RIT execution on the prototype chip, and we support it with a hardware mechanism that is part of the microprocessor design (Figure 2). The hardware mechanism *records* the failing comparisons and *pins* the execution points where mismatches happen. Moreover, when a mismatch is detected, the hardware mechanism allows *replay* of the RIT by replacing the execution of the offending instruction with its equivalent. If the offending instruction is the original one and its equivalent is bug-free the enhanced RIT replay produces the useful logging information we want. If this is not the case (i.e. the equivalent instruction is the offending one) the enhanced RIT replay does not produce useful logging information. Bugs in the instructions of the ERIT will be identified subsequently by other RITs that are explicitly generated to test them.

With our hardware replay described below, the test can continue execution and more bugs can be detected. Replay can happen several times for a single enhanced RIT as long as it detects more mismatches (potential bugs). In a typical RIT-based flow, after the first mismatch the remaining execution of the test is most probably useless since many subsequent responses are corrupted (since an output value of one operation can propagate to

the input value of the subsequent operations). Other bugs that could possibly be detected by the remaining of the test are left undetected. In the replaying of the test using our hardware mechanism, the *mismatch is bypassed*, subsequent responses are not corrupted and if the remaining test can detect another mismatch (more bugs) it is allowed to do so. The last execution of the enhanced RIT is mismatch-free and detailed logging information is available for post-processing.

To analyze the hardware mechanism, we assume that an enhanced RIT (original RIT + ERIT + checking code) will be executed on the prototype chip. The original RIT includes k store instructions that write the results of the computation to the memory. For a typical RIT of ~4K instructions, it is realistic to assume that k is between 500 and 1000. Similarly, the ERIT (equivalent RIT) includes k store instructions that write the results of the equivalent computations. The checking code compares the results stored by instruction *store*[i] and *estore*[i], where *store*[i] is the i^{th} store of the RIT and *estore*[i] is the i^{th} store of the ERIT, with $i = 1...k$.

The basic concept of the mechanism is that when a mismatch is detected between *store*[i] and *estore*[i], during replay, instead of executing the "buggy" code between *store*[i–1] and *store*[i], the processor executes the equivalent code between *estore*[i–1] and *estore*[i]. The mismatch has been bypassed.

The hardware mechanism is shown in Figure 4 and operates as follows:

First run of the enhanced RIT. The checking code finishes with the first mismatch among the k responses of RIT and the k responses of ERIT stored in variable *mid* (mismatch id), with *mid* between 0 and k. If *mid* = 0 (i.e. the queue is empty), then there is no mismatch and the chip passes the enhanced RIT; validation continues with the next RIT. If *mid* > *0* (i.e. the queue is not empty) the enhanced RIT will be replayed because *store*[*mid*] and *estore*[*mid*] instructions generated different results. Moreover, during the first run of the enhanced RIT the addresses of all store instructions (of the RIT and the ERIT) are saved in the *store-addr* and *estore-addr* buffers of the hardware mechanism to facilitate replay. Each of these two structures has a size of k words (addresses) for a total of 2k words. The *store-addr* and *estore-addr* buffers record the address of the store instructions themselves and not the address they store the data to. The contents of the two buffers are needed to replay the enhanced RIT by manipulating the contents of the *program counter* – PC (or Instruction Pointer – IP) in hardware, as explained below.

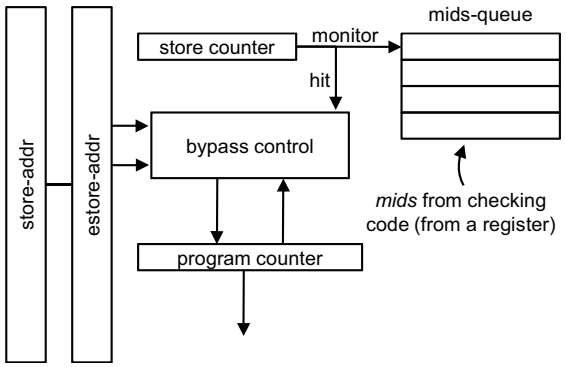

Figure 4. Hardware mechanism structure

Subsequent runs of the enhanced RIT. Every mismatch produced in previous runs is stored in an entry of the *mids-queue*. The hardware mechanism counts the number of stores (monitoring instruction decoding) in the new run of the RIT in a *store-counter* (which is reset at the beginning of each run of the enhanced RIT). When *store-counter* gets equal to *mid* – 1 (i.e. before execution of the corrupted sequence) where *mid* is the mismatch id at the head of the *mids-queue* a "bypass" must happen (managed by the *bypass control* component), i.e. execution of the RIT code between *store*[*mid*–1] and *store*[*mid*] must be "replaced" by the execution of the ERIT code between *estore*[*mid*–1] and *estore*[*mid*] (this is why store-counter must be equal to *mid*–1). The "replacement" is done on-the-fly as follows using the store addresses information saved in buffers *store-addr* and *estore-addr*:

- After instruction *store*[*mid* –1] finishes, *PC* gets the address of the instruction following *estore*[*mid*–1].
- After the instruction before *estore*[*mid*] finishes, *PC* gets the address of instruction *store*[*mid*].

Therefore, instead of executing the "buggy" code between *store*[*mid*–1] and *store*[*mid*], the processor executes the equivalent code between *estore*[*mid*–1] and *estore*[*mid*]. The mismatch has been bypassed.

During each replay run, this bypass process is repeated for each *mid* saved in the *mids-queue*. At the end of each "replay" run a new non-zero *mid* may be produced (in the checking code) and stored in the *mids-queue*. This means that in the subsequent replay execution one more bypassing will take place because the RIT found one more mismatch. Eventually, the last "replay" (in which multiple bypasses took place) will produce a *mid* = 0 denoting a mismatch-free execution. Table 3 summarizes the operation of the hardware replay mechanism.

Table 3. Hardware mechanism operation

inputs: Set of *RITs*: original random instruction tests
Set of *ERITs*: equivalent random instruction tests
output: Log information {*mids-queue*}

for all RITs **do**
 execute RIT; **save** *store addresses* to *store-addr* buffer*;*
 execute ERIT; **save** *store addresses* to *estore-addr* buffer*;*
 execute checking code; **compare** *RIT*/*ERIT* responses;
 update *mids-queue*: add entry if mismatch found;
 if (*mid* = 0) **then**
  TestPassed;
 else
  while (mid > 0) **do**
   store-counter = 0;
   replay(RIT, ERIT);
    if (*store-counter* hits a *mid* – 1 in *mids-queue*) **then**
     PC ← *estore*[*mid* – 1] + 4;
     execute equivalent operation;
     PC ← *store*[*mid*];
    end if
    execute checking code; **compare** *RIT*/*ERIT* responses;
    update *mids-queue*: add entry if *new* mismatch found;
  end while
 end if
end

The logging information produced for the debug engineer is the queue of the mismatch identifiers *mids-queue*. For example, if at the end of the execution of an enhanced RIT (including all replays) the *mids-queue* contains integers 10, 25, 130, and 0, this means that mismatches have been detected between *store*[10] and

estore[10], between *store*[25] and *estore*[25], and between *store*[130] and *estore*[130], and that the fourth "replay" was mismatch-free. With this logging information in hand, the debug engineer can directly locate the code portions with mismatches and focus on them for root-cause analysis. The debug engineer can also easily identify the RIT code replaced by the equivalent ERIT code for each bypass (it is of course the code before the stores with mismatches). We note also that the *mids-queue* contents must be also saved in memory so that after the end of the test they are passed as the methodology log to the debug team. Inside the hardware mechanism itself, *mid* information is passed by the checking code in one of the processor registers and the hardware mechanism records it in the *mids-queue*.

The size of the hardware mechanism depends on the size of the *store-addr* and *estore-addr* buffers (we assume a maximum size of 1000 address entries words for each), the size of the *mids-queue* (we assume a maximum of 10 entries; it is very unlikely for a single RIT to detect more bugs), the *store-counter*, and the *bypass control* logic that includes multiplexers and comparators. The hardware mechanism is deactivated after the end of post-silicon validation and thus it does not consume any power and does not affect performance in the field.

(4) Post-processing (triage). As we mentioned earlier, it has been reported in the literature that the same bug can corrupt a large number of RITs. When the debug engineer is fed with many failing RIT memory dumps that are due to the same bug (in the same instruction, operation, or structure) the debug phase takes unnecessarily long time in order to determine if that failure is unique. This is an inherent inefficiency of traditional RIT-based flow since mismatches are detected much later in the server that compares the memory dumps. Our method offers a very important advantage to the post-processing (triage) phase: validation data provided by our hardware mechanism can help clustering of failure modes. This can be achieved through post-processing of the enhanced logs generated by the hardware mechanism (list of mismatch identifiers, *mids*, i.e. stores that saved different results to memory).

The list of mismatch identifiers (*mids*) is the log information our method provides. An integer *m* in the log (an entry in the *mids-queue*) means that: (a) the m^{th} pair of stores produced a mismatch, i.e. *store*[*m*] and *estore*[*m*] produced different results; (b) the code between *store*[*m*–1] and *store*[*m*] has been replaced by the code between *estore*[*m*–1] and *estore*[*m*] and the RIT continued. These two pieces of information can help the debug engineer identify the offending instructions and work on them.

Apart from the instruction bypassing realized by the hardware mechanism, our methodology and corresponding logging data provide a *fast workaround* solution necessary to allow validation to continue running subsequent RITs: buggy instructions can be avoided in subsequent RITs by using their bug-free equivalents from the ISA diversity database.

5. EXPERIMENTAL EVALUATION

We assess our validation methodology by performing a comprehensive bug injection experiments campaign on a superscalar, out-of-order, single-core x86-compatible model in the *PTLsim* simulator [35]. PTLsim supports the full x86-64 instruction set of Pentium 4 (and subsequent), Athlon 64, and similar machines with all extensions (x86-64, SSE/SSE2/SSE3, MMX, x87).

For the experimental evaluation of our microprocessor validation methodology, we set up the tool chain outlined in Figure 5.

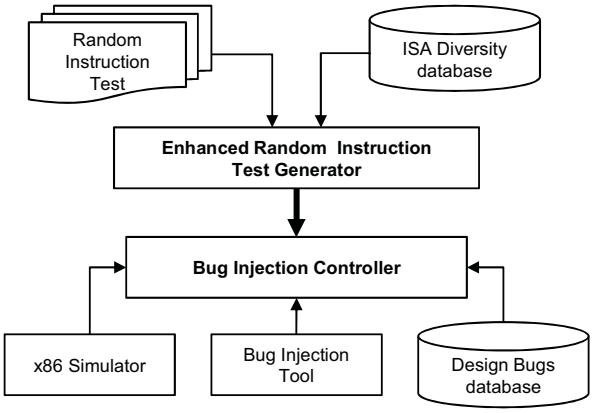

Figure 5. Experimental setup and tool chain

The experimental framework consists of the following:

(a) The PTLsim *architectural simulator* for the x86 microprocessor.
(b) Our *RIT enhancement tool* described previously that gets original RITs and produces enhanced RITs applying the equivalence-based methodology utilizing the ISA diversity database.
(c) A *bug injection tool* that injects both logic and electrical bug at various locations throughout the entire processor. The bug injection tool uses our bug database which has been populated with bugs of either type.

For a given set of bugs in the design bugs database and a given set of enhanced RITs produced by our methodology, the experimental framework executes the enhanced RITs and records if a bug is detected or not.

We have injected both *logic* and *electrical* bugs to model different design bug conditions throughout the entire x86 architecture. For electrical bugs injection, we follow [21] which assumes that an effective and realistic way to model electrical bugs is to model them as transient bit flips at the microprocessor's flip-flops. On the other hand, logic bugs have a permanent effect and we model them through a modification in the semantic correctness [31] of the architectural simulator's source code. Table 4 summarizes the types of the injected logic bugs.

Table 4. Types of logic bugs injected into the simulator

Semantic Modification	Correct Instance	Buggy Instance
wrong operator	a = b + c	a = b – c
wrong conditional statement	if (a > b)	if (a ≥ b)
wrong signal assignment	a ← b + c	a ← d
conceptual error	if (a > b) then a ← c	if (a > b) then a ← c + b
wrong constant assignment	a = 0x000F	a = 0x0002

Table 5 presents a summary of the logic and electrical bugs we injected. In total, 1025 *design bugs* were injected, 802 of them are logic and 223 are electrical, covering all pipeline stages and hardware components of the x86 microprocessor. We injected electrical bugs mostly in the components that integrate large memory arrays, i.e. branch prediction unit, register file, etc. since the memory-dominated modules are more vulnerable to electrical bugs due their high density. The bit-flips are activated randomly in any position of a data structure. On the other hand, we injected

logic bugs mostly in the control-related components where design errors in the complex conditional decisions are more likely to occur.

Table 5. Injected design bugs distribution in the components of the x86 processor model

Pipeline Stage	Component	Logic bugs	Electrical bugs	Total bugs
Fetch/ Decode	Branch Predictor	71	16	**87**
	Prefetcher	29	12	**41**
	Instruction Decoder	100	–	**100**
	Microcode	62	–	**62**
	Instruction Buffer	–	18	**18**
Issue/ Execute	Integer Arithmetic	95	–	**95**
	FP Arithmetic	97	–	**97**
	Jump logic	46	–	**46**
	Load/Store logic	66	21	**87**
	Issue Queue	42	–	**42**
	Scheduler	32	–	**32**
	Register File	61	63	**124**
Retire	Reorder Buffer	101	41	**142**
Instruction & Data		–	52	**52**
Total		**802**	**223**	**1025**

We compare our methodology with the traditional RIT-based validation flow. Moreover, we perform the same set of experiments for two other self-checking validation approaches presented in the literature that also aim to mitigate the time-consuming simulation step of RIT-based validation: (a) *Reversi* [33], according to which each instruction is followed by a reverse instruction; a bug is detected when the final result is not equal to the initial one (i.e. it has not been reversed correctly); (b) *QED or instruction duplication* [15], according to which each instruction is duplicated and electrical bugs are detected when execution of duplicated instructions gives different results. For each of the three methods (Reversi, QED, and ours), we use the same original RIT as input and we enhance it according to the basic idea of each method.

Each of the original RIT sequences we use as an input consists of 4K instruction cycles and was generated by the tool generously provided to us by the authors of [33]. In our experiments we used 154 original RITs produced by the RIT generator, thus we ran a total of 616K random instructions in the simulator for each of the 1025 injected bugs. Our RIT enhancement methodology increases, on average, the RIT code size by 6 times compared with the original RIT size (min=4.3X, max=9X, for the 154 RITs). The total number of instructions for the full campaign of 154 enhanced RITs (includes the original RITs, the equivalent RITs, and the checking code) is 3.7M of x86 instructions. The corresponding increase in RIT size by Reversi is on average 4 times (i.e. total number of instructions is approximately 2.5M instructions) and by QED is on average 3 times (i.e. total instructions is approximately 1.9M).

The results of our bug injection experiments are shown in Figure 6. Our methodology detects all 1025 bugs injected into the simulator (bug coverage 100%) because we stopped generation of more RITs when all the injected bugs were detected. The traditional post-silicon validation flow detects 928 bugs (coverage 90.54%). This difference, against the proposed method, is explained by the activation of more hardware areas by the equivalent RIT. The instruction reversing method (Reversi) detects 903 bugs (coverage 88.10%) because there are cases where an instruction cannot be inverted. Furthermore, the flexibility of the ISA diversity concept to deploy equivalent instructions which activate totally different path in processor's logic provides us with the ability to avoid bug masking conditions (e.g. integer addition and subtraction happen on the same module, while in our method the equivalent addition take place on the floating point logic). Finally, the duplicated instructions approach (QED) detects 210 bugs (coverage 20.49%) because it can only detect electrical bugs, since a logic bug will act in an identical way in both original and duplicated instruction.

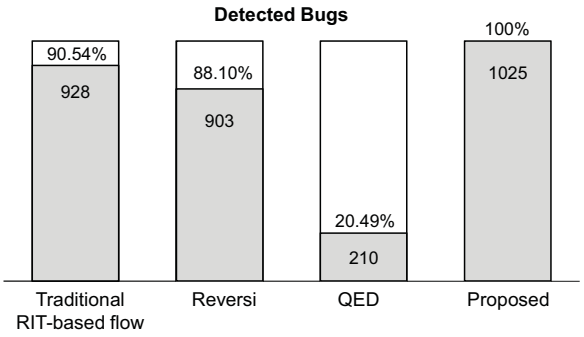

Figure 6. Design bugs coverage (1025 bugs injected in total) for the four different methods

For a complete validation plan (trillions of instructions), we expect our approach to have the same bug detection capability with the traditional RIT-based flow since our bug detection capability relies on the original RITs which are carefully generated by sophisticated industrial random generators. The advantage of our method is that mitigating the time-consuming simulation step it is able to apply more RITs and thus detect potential bugs earlier. Our methodology compares favorably with the other two self-checking approaches Reversi and QED. This is because Reversi is based on instructions whose effect can be reversed; this is not possible in many cases, and thus a number of bugs are not detected. QED on the other hand is based on instruction duplication and is very effective only for electrical bugs but not for logic ones.

Another advantage of the proposed method is that, it refines the validation data using the hardware replay mechanism. During our bug injection experiments, we observed that the average number of different bugs that were detected by a single RIT is about 4. Therefore, we integrated the hardware replay mechanism in the PTLsim simulator and conducted a second set of experiments: we injected all the bugs at the beginning of the simulation and executed all RITs with the highest bug detection capability. The proposed hardware mechanism detected all the injected bugs (through bypassing the offending instructions with their equivalents in the replay executions). This is a significant benefit of the proposed framework compared to the traditional flow which requires more tests to detect the same number of bugs.

A second observation deduced by our bug injection experiments is that the average number of RITs affected by a single bug in each component is 67 (out of 154 RITs we applied). This observation highlights the requirement for fast workaround solutions, since a bug may corrupt more than one test. However the number of corrupted tests highly depends on the location of

the activated bug. For instance, a bug injected in a functional unit will probably corrupt all subsequent tests that utilize this unit, while a bug injected in a cache memory cell may not affect the subsequent tests due to the local use of the specific memory location. The proposed technique contributes to this requirement providing equivalent instructions workarounds for the early detected bugs.

Table 6 presents a comparison in terms of validation time (all timing measurements are on an Atom N270 with 1 GHz clock frequency) for the traditional RIT-based flow, Reversi, QED and the proposed method. Note that in post-silicon validation stage numerous (hundreds of millions for almost a year) random tests are generated; therefore Table 6 represents only a snapshot of the whole process. We discuss the different parts of the total validation time in the following.

Table 6. Validation times

Time (sec)	Trad. RIT	Reversi	QED	Proposed
Generation	4.460	6.310	5.530	7.680
Simulation	51.000	–	–	–
Execution	0.027	0.110	0.071	0.176
Total	55.487	6.420	5.601	7.856

In a typical microprocessor post-silicon validation flow, the total validation time consists of the following parts.

1. *Generation time*: The time required to generate the random tests in the host machine. In our experiments, we generate 154 random tests, each one consisting of 4K instructions (summing up to 616K instructions for the traditional RIT flow, 2.4M instructions for Reversi, 1.8M instructions for QED, and 3.7M instructions for the proposed method).
2. *Upload time*: The time required to upload the test from the host machine to the prototype for execution. This is typically performed through a standard PCIe interface (or any other host debug interface). Given that the size of a random test is only a few kilobytes the PCIe throughput guarantees a nearly zero upload time and we do not include this time in Table 6.
3. *Simulation time*: Only the traditional RIT-based flow needs to be simulated, since the other three approaches are self-checking.
4. *Execution time*: The actual silicon execution time.
5. *Download time*: The time required to download the responses (memory locations affected by the random test) of the test from the prototype to the host machine using the PCIe or other host interface. For the three self-checking methods the responses are downloaded only in the case of a failing RIT, while for the traditional RIT-based flow responses' downloading always follows each test execution. Since PCIe throughput guarantees a nearly zero download time, we don't include it in Table 6.
6. *Compare time*: The time required to compare the actual responses with the expected ones (golden signatures). This is again very fast and we do not include it in Table 6.

The timing measurements demonstrate that the proposed method is much faster than the traditional RIT-based flow: more RITs can be applied in the same time. In addition, the longer test execution time of the proposed method compared to the two self-checking alternatives is due to the longer random tests it uses. However, this is alleviated by the improved bug detection capability of our method as shown in Figure 6.

Note that the speedup offered by our methodology by the mitigation of simulation applies only to the validation of instructions that have equivalents (more than three quarters of the ISAs – see Section 3). For the remaining instructions, the classic simulation-based RIT flow must be followed and thus our methodology is complementary to current industry practice.

Figure 7 roughly visualizes the timing of a traditional RIT-based flow and the timing of the proposed flow to give a clear idea of the timing advantages of the proposed method. In the host machine, we assume that generation (G) of random tests, uploading (U), simulation (S), downloading (D) and comparison (C) of the actual responses with the expected can take place in parallel. The prototype starts execution of legacy tests available from pre-silicon verification (or from previous architectures) and then executes the newly generated post-silicon random tests. Although, in our experiments the upload, download and compare times are negligible because of the high throughput of PCIe interface, in Figure 7 we include them for demonstration purposes. The upload and download times can be significant if a slower interface than PCIe is used or if the size of the tests is much larger than a few kilobytes.

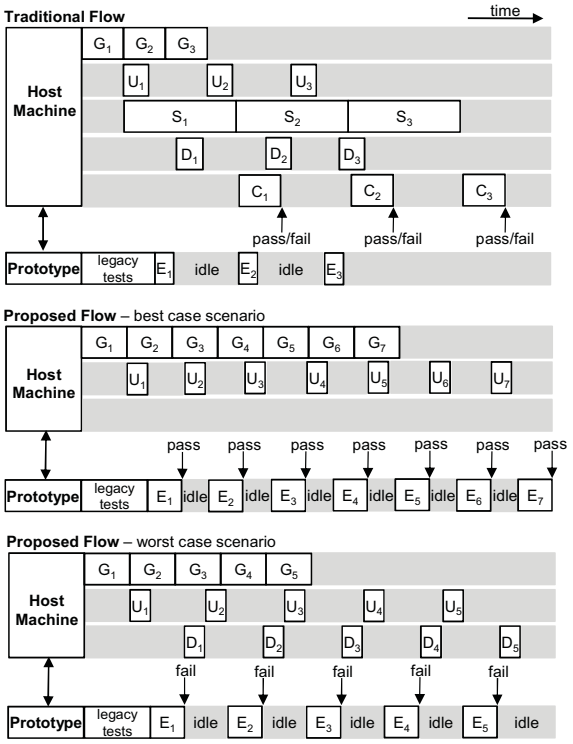

Figure 7. Traditional vs. proposed RIT-based validation

Figure 7 shows that the proposed methodology mitigates the simulation phase and also downloads only the responses of the failing RITs to the host machine. In a sense, the silicon prototypes are *better utilized* with our method and they execute *more random tests during the same time*. For example, Figure 7 shows that the proposed flow executes five random tests in the worst case (all failing) and seven random tests in the best case (all passing) while the traditional RIT-based flow executes only three. Thus, it accelerates post-silicon validation significantly.

6. RELATED WORK

Self-checking methods. Previous studies [27], [33] have proposed the generation of reversible test programs, where the program's final state is known a priori, as a way to avoid the simulation step of golden signature production. However, generating reversible operations is not always an easy task and in some cases is partially or totally infeasible, like in the case of floating point operations. Another recent approach [15] targets to minimize the error detection latency of electrical bugs by duplicating instructions.

Software diversity. Previous approaches have adopted the concept of software diversity, as a zero-overhead alternative of design diversity, to build fault-tolerant systems. The key idea is to modify the executed code when a hard fault is present, without spoiling the original code functionality [22]. Independent generation of programs has been also proposed as a fault tolerant approach [10]. Construction of programs with duplicated instruction and diverse data operands has been proposed as a way to detect temporal and permanent faults in the field [24]. Software implemented fault tolerance aims to provide soft error tolerance by instruction duplication [28]. Our method, for first time, utilizes the concept of ISA diversity for efficient post-silicon validation.

Online bug detection. Previous approaches propose the use of dedicated hardware to detect and recover from bugs in the field [4], [11], [30], [32], [34]. Semiconductor industry needs bugs detected as soon as possible before massive production of the microprocessor chip. The proposed post-silicon validation methodology aims to satisfy this requirement.

Design for debug. Recent studies [1], [25] propose hardware modifications in order to improve bugs root cause analysis. Our method does not aim to enhance the debug phase directly, but rather it attempts to extract as useful as possible debug information from random test execution "replay". Therefore, it collaterally favors the subsequent debug phase by providing useful logs.

7. CONCLUSION

Effective post-silicon validation for modern architectures must minimize the simulation bottleneck for random instruction tests to save time, resources, and budget while not limiting bug detection capabilities. We have presented a novel, self-checking, hardware supported framework to accelerate and improve the quality of post-silicon validation by exploiting diversity (instructions equivalence) in instruction set architectures. Our analysis for ARM, MIPS, PowerPC, and x86 instruction sets shows that despite their differences, modern ISAs can perform an operation with many equivalent ways. We exploit this ISAs property to generate random tests that detect bugs by comparing results of equivalent instructions. Moreover, we support our bug detection method in hardware with a light-weight mechanism which records mismatches and replays the random tests after bypassing the offending instruction. Hardware-based replay allows the RIT to detect as many failures (potential silicon bugs) as it can and it provides exact information on the offending instructions. We evaluate the methodology experimentally on an x86 model with a comprehensive bug injection campaign. Our methodology successfully detected all injected bugs (1025 bugs in total) while the traditional RIT-based approach and the two self-checking methods we compared with did not. Furthermore, the proposed methodology accelerates the post-silicon validation process by increasing the prototype utilization since a larger number of random tests are executed.

8. ACKNOWLEDGMENTS

The authors wish to thank Ilya Wagner of Intel for generously providing the code of Reversi random test generator [33] and for his help in using it. Also, the authors thank Todd Schelling of Intel for his valuable inputs regarding the microprocessor post-silicon validation flow.

9. REFERENCES

[1] M.Abramovici, P.Bradley, K.Dwarakanath, P.Levin, G.Memmi and D.Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In ACM/IEEE Design Automation Conference (DAC), 2006.

[2] A.Adir, E.Almog, L.Fournier, E.Marcus, M.Rimon, M.Vinov and A.Ziv. Genesys-pro: Innovations in Test Program Generation for Functional Processor Verification. IEEE Design & Test of Computers (D&T), 21(2):84-93, 2004.

[3] A.Adir, S.Copty, S.Landa, A.Nahir, G.Shurek, A.Ziv, C.Meissner and J.Schumann. A Unified Methodology for Pre-silicon Verification and Post-silicon Validation. In ACM/IEEE Design, Automation & Test in Europe Conference (DATE), 2011.

[4] T.Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In ACM/IEEE International Symposium on Microarchitecture (MICRO), 1999.

[5] A.Avizienis and H.Yutao. Microprocessor entomology: A Taxonomy of Design Faults in COTS Microprocessors. In IEEE Dependable Computing for Critical Applications (DCCA), 1999.

[6] F.Bacchini, R.Damiano, B.Bentley, K.Baty, K.Normoyle, M.Ishii, and E.Yogev. Verification: What Works and What Doesn't. In ACM/IEEE Design Automation Conference (DAC), 2004.

[7] M.Behm, J.Ludden, Y.Lichtenstein, M.Rimon and M.Vinov. Industrial Experience with Test Generation Languages for Processor Verification. In ACM/IEEE Design Automation Conference (DAC), 2004.

[8] T.Bojan, F.Igor and M.Robert. Intel's Post Silicon Functional Validation Approach. In IEEE High Level Design Validation and Test Workshop (HLDVT), 2007.

[9] T.Bojan, I.Frumkin and R.Mauri. Intel® First Ever Converged Core Functional Validation Experience: Methodologies, Challenges, Results and Learning. In IEEE International Workshop on Microprocessor Test and Validation (MTV), 2007.

[10] L.Chen and A.Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In IEEE Fault Tolerance Computing Symposium (FTCS), 1996.

[11] K.Constantinides, O.Mutlu and T.Austin. Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation. In ACM/IEEE International Symposium on Microarchitecture (MICRO), 2008.

[12] E.Daoud and N.Nicolici. Embedded Debug Architecture for Bypassing Blocking Bugs During Post-silicon Validation. In IEEE Transactions on Very Large Scale Integration Systems (TVLSI), 19(4):559-570, 2011.

[13] J.Goodenough and R.Aitken. Post-silicon is Too Late Avoiding the $50 Million Paperweight Starts with Validated Designs. In ACM/IEEE Design Automation Conference (DAC), 2010.

[14] N.Hakim. Introduction to Post-silicon Validation, presentation. Intel Corp. 2010, http://embedded.eecs.berkeley.edu/eecsx44/lectures/ /NagibHakim-PostSiValidation.pdf

[15] T.Hong, Y.Li, S-B.Park, D.Mui, D.Lin, Z.A.Kaleq, N.Hakim, H.Naeimi, D.S.Gardner and S.Mitra. QED: Quick Error Detection Tests for Effective Post-silicon Validation. In IEEE International Test Conference (ITC), 2010.

[16] Intel finds flaw in Sandy Bridge chipsets, halt shipment. http://www.techreport.com/discussions.x/20326, 2011.

[17] D.Josephson, S.Poehhnan and V.Govan. Debug Methodology of McKinley Processor. In IEEE International Test Conference (ITC), 2001.

[18] D.Josephson. The Good, the Bad, and the Ugly of Silicon Debug. In ACM/IEEE Design Automation Conference (DAC), 2006.

[19] D.Josephson. The Manic Depression of Microprocessor Debug. In IEEE International Test Conference (ITC), 2002.

[20] J.Keshava, N.Hakim and C.Prudvi. Post-silicon Validation Challenges: How EDA and Academia Can Help. In ACM/IEEE Design Automation Conference (DAC), 2010.

[21] R.McLaughlin, S.Venkataraman and C.Lim. Automated Debug of Speed Path Failures Using Functional Tests. In IEEE VLSI Test Symposium (VTS), 2009.

[22] A.Meixner and D.Sorin. Detouring: Translating Software to Circumvent Hard Faults in Simple Cores. In IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2008.

[23] S.Mitra, S.A.Seshia and N.Nicolici. Post-silicon Validation Opportunities, Challenges and Recent Advances. In ACM/IEEE Design Automation Conference (DAC), 2010.

[24] N.Oh, S.Mitra and E.J.McCluskey. ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions. In IEEE Transactions on Computers (TC), 51(2):180-199, 2002.

[25] S-B.Park, T.Hong and S.Mitra. Post-silicon Bug Localization in Processors Using Instruction Footprint Recording and Analysis (IFRA). In IEEE Transactions on Computer-Aided Design (TCAD), 28(10):1545-1558, 2009.

[26] P.Patra. On the Cusp of a Validation Wall. In IEEE Design & Test of Computers (D&T), 24(2):193-196, 2007.

[27] R.Raina and R.Molyneaux. Random Self-test Method – Applications on PowerPC^{TM} Microprocessor Caches. In IEEE Great Lake Symposium on VLSI, 1998.

[28] G.Reis, J.Chang, N.Vachharajani, R.Rangan and D.August. SWIFT: Software Implemented Fault Tolerance. In ACM/IEEE International Symposium on Code Generation and Optimization (CGO), 2005.

[29] H.Rotithor. Postsilicon Validation Methodology for Microprocessors. IEEE Design & Test of Computers (D&T), 17(4):77-88, 2000.

[30] S.Sarangi, A.Tiwari and J.Torrellas. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. In ACM/IEEE International Symposium on Microarchitecture (MICRO), 2006.

[31] S.Sudhakrishnan, L.Su and J.Renau. Processor Verification with *hwBugHunt*. In IEEE International Symposium on Quality Electronic Design (ISQED), 2008.

[32] S.Sudhakrishnan, R.Dicochea and J.Renau. Releasing Efficient Beta Cores to Market Early. In ACM/IEEE International Symposium on Computer Architecture (ISCA), 2011.

[33] I.Wagner and V.Bertacco. Reversi: Post-silicon Validation System for Modern Microprocessors. In IEEE International Conference on Computer Design (ICCD), 2008.

[34] I.Wagner, V.Bertacco and T.Austin. Using Field-Repairable Control Logic to Correct Design Errors in Microprocessors. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 27(2):380-393, 2008.

[35] M.Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2007.