Reversi: Post-Silicon Validation System for Modern Microprocessors

Ilya Wagner and Valeria Bertacco
University of Michigan
{*iwagner, valeria*}*@umich.edu*

***Abstract*— Verification remains an integral and crucial phase of today's microprocessor design and manufacturing process. Unfortunately, with soaring design complexities and decreasing time-to-market windows, today's verification approaches are incapable of fully validating a microprocessor before its release to the public. Increasingly, post-silicon validation is deployed to detect complex functional bugs in addition to exposing electrical and manufacturing defects. This is due to the significantly higher execution performance offered by post-silicon methods, compared to pre-silicon approaches. Validation in the post-silicon domain is predominantly carried out by executing constrained-random test instruction sequences directly on a hardware prototype. However, to identify errors, the state obtained from executing tests directly in hardware must be compared to the one produced by an architectural simulation of the design's golden model. Therefore, the speed of validation is severely limited by the necessity of a costly simulation step.**

In this work we address this bottleneck in the traditional flow and present a novel solution for post-silicon validation that exposes its native high performance. Our framework, called *Reversi*, generates random programs in such a way that their correct final state is known at generation time, eliminating the need for architectural simulations. Our experiments show that Reversi generates tests exposing more bugs faster, and can speed up post-silicon validation by 20x compared to traditional flows.

I. Introduction

Verification remains an unavoidable, yet quite challenging and time-consuming aspect of the microprocessor design and fabrication process. With shortening product timelines and increasing time-to-market pressure, processor manufacturing houses are forced to pour more and more resources into verification. The problem is exacerbated by the appearance and growing adoption of multi-core chips. The design effort in these systems is lower than that of a single-core chip of a similar size, since cores are replicated from a single design. The verification effort is however, higher, because in addition to validating the cores, inter-core communication must also be verified. Therefore, with processor complexity increasing rapidly, and verification speeds lagging behind, bugs, such as the AMD Opteron REP MOVS error [2] and functional problems in the Intel's Core 2 Duo [3, 4], continue to slip into production silicon.

Hardware verification can be divided into two phases: pre- and post-silicon. Pre-silicon verification employs two major families of solutions: simulation-based tools and formal techniques. Although formal solutions can be used to prove key design properties, such as absence of deadlock, proper ALU and FPU functionality, *etc.*, they suffer from the state explosion problem and can ultimately be used only on small design modules. For example, in the verification of the Intel Pentium 4 processor, formal methods were used only on floating-point units, schedulers and instruction decoders [8]. Simulation approaches, on the other hand, do not have such strict limitations, but neither can provide hard guarantees of correctness: only those behaviors that have occurred during the simulation can be validated. Nevertheless, simulation remains the method of choice for pre-silicon verification due to its scalability.

Post-silicon validation relies on a concept similar to simulation: the hardware prototype executes as many randomly generated input vectors as possible. However, there are a few key differences between this approach and pre-silicon validation. First, the execution on a hardware prototype is several orders of magnitude faster than any functional simulator, therefore, significantly more test vectors can be checked. However, this high speed comes at the price of limited observability: the internal state of the prototype cannot be easily or fully observed, forcing the engineers to diagnose errors from the architectural state of the system. Tests in the post-silicon domain consists of directed tests checking specific features of the processor, compatibility checks, such as operating system boot-up and tests with legacy software, as well as automatically generated random tests [8, 16]. Due to the unpredictable outcome of these random programs, engineers must simulate them on a known-correct model of the design to obtain the correct final state of the hardware prototype to identify discrepancies, potentially revealing a bug. While tests can be run at-speed on the hardware, test generation and simulation constitute the bottleneck in this process, limiting it to the performance level of pre-silicon simulation. Consequently, design houses are forced to spend enormous computational resources on test generation and simulation servers [16].

Traditional post-silicon testing solutions differ from validation in that they rely on a structural model of the design to determine the correct behavior of the silicon part under test and detect electrical and manufacturing defects. However, functional errors in the design will be present in both the hardware prototype and the structural model generated from RTL, thus testing is not viable to find this kind of bugs.

In this paper we take the first step towards a novel high-throughput post-silicon validation methodology, which allows for test generation to match the performance of execution of the silicon prototype. By cleverly crafting randomized tests with known final outcome, we address the bottleneck of the traditional post-silicon flow, while leveraging its high design coverage.

A. Contributions of This Work

The main contribution of this paper is the development of a novel test generation framework, called *Reversi*, for post-silicon processor validation. Our goal is to exploit the full performance potential of silicon prototypes and eliminate the costly simulation step required to obtain a known-correct final state. To this end, tests are generated by Reversi in such

a way that at the end of the execution, the initial state of the machine is restored. Therefore, the final state of such a *reversible program* is known *a priori* and, the simulation phase of the validation process is bypassed. Since our program generation algorithm is agnostic to any particular instruction set, it can be easily ported between processors with different instruction and feature sets. Moreover, the absence of the simulation step in our framework allows for tests to be generated directly by hardware residing on the same system board as the prototype, eliminating the need for costly test generation servers. Consequently, validation speed becomes only limited by the speed of communication between the prototype and the testing board. Moreover, once the system under test is sufficiently validated, the test generator can run directly on it. In this latter case, tests can be produced in one portion of the chip's cores and transferred to other cores for execution. If the generator cores were flawed, they would not produce proper reversible programs, exposing the issue.

We evaluated our framework against a traditional post-silicon validation flow based on a constrained-random test generator paired with an architectural simulator. Our experimental results demonstrate that reversible programs can expose more complex processor bugs faster than traditional methods and, at the same time, boost the performance of the testing process by 20x.

The remainder of this paper is organized as follows. Section II reviews prior work in post-silicon validation with random instruction generators. Sections III and IV present the Reversi solution and detail the construction of complex program structures. Section V provides a comparative evaluation of our approach, while Section VI concludes the paper.

II. Prior work

Hardware verification with constrained-random test generation has been a focus of both academic and industrial research for a long time. Most efforts, however, have been dedicated to the pre-silicon verification domain, where test length is relatively short compared to real-life applications. One of the most prominent industry tools in this family is Genesys-Pro [7] developed by IBM. This tool provides advanced capabilities for test generation (biasing primitives, templated specification language, *etc.*) However, it is designed primarily as a pre-silicon tool. Genesys is not capable of producing tests with known final states, and hence its use in the context of post-silicon validation would require a simulator to compute such states. Several other industrial solutions [1, 14] provide similar features, but again require a simulator to generate the final processor state.

As reported by Rotithor in [16], test generation engines targeting the post-silicon domain share some of their properties with the tools mentioned above: test scenarios have a templated format allowing for fast generation of randomized programs. Note however, that the setup described in [16] calls for a number of servers to build the tests and known-correct design models to simulate these tests and obtain the correct final state, which will be compared with the results of the prototype execution. Therefore, the framework in [16] still requires a simulation-based checker in order to expose bugs, unless they manifest themselves more explicitly, *e.g.,* as a deadlock or early test termination.

There also exists a variety of testing solutions that combine ATPG (automatic test patter generation) [11] with techniques for silicon state acquisition such as scan [13], JTAG [12], cycle breakpoint [9] or on-chip-logic analyzers [10]. Unfortunately, ATPG approaches are only capable of exposing electrical and manufacturing defects. A functional error, on the other hand, cannot be flagged by these solutions, since it is present not only in the hardware under test, but in the structural model used by the test generator as well. Unlike these approaches, our solution relies on a functional, high-level specification of the hardware to expose design defects.

Finally, Raina and Molyneaux presented in [15] a solution based on the use of instructions and their inverses for processor verification. However, their work used the reversibility scheme for cache verification, rather than for processor cores.

III. Reversi Test Generation System

Typically post-silicon functional validation in industry has been conducted with two types of tests: parameterized directed tests and constrained-random (or pseudo-random) tests. Although the former ones can provide high coverage, they require significant human effort to be developed. The pseudo-random tests, constrained to produce only valid instruction sequences, can be generated automatically, however often suffer from lower coverage. More importantly, the final state of the processor after executing a random test sequence is unknown. Therefore, engineers must resort to simulating the design's golden model to compute the final processor state and check it against state dumps of the actual hardware prototype (as illustrated in Figure 1.a).

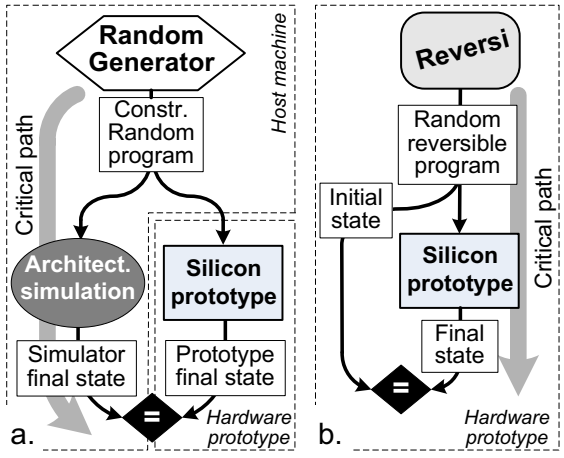

Fig. 1. **A typical post-silicon validation flow vs. a Reversi-based flow. a.** In a typical post-silicon methodology, random tests are produced by a test generator and fed to both a golden model simulator and a silicon prototype. Bugs are flagged by differences between the prototype's and simulator's final states. Both test generation and simulation are done on a host machine at relatively slow speed. **b.** A Reversi-based flow does not require a simulator: random reversible programs can be generated on a tester board or on the hardware prototype itself. Bugs are flagged by differences between final and initial states of the prototype.

Unfortunately, as was mentioned above, the simulation of the golden model is several orders of magnitude slower than the hardware execution, therefore, the computation of the

final state becomes a bottleneck for the entire effort. We address this issue in our methodology by developing a post-silicon solution which fully exploits the performance of the hardware under test. We designed a test generator, called *Reversi*, that produces tests whose outcome is known by construction. This allows us to bypass the simulation step and speed up the overall validation flow (Figure 1.b).

The main observation that we made in developing Reversi is that many instructions in a processor's ISA have counterparts, *i.e.,* operations whose functionality is the inverse of the former, such as restoring a value in a particular register, clearing a set of flags, *etc*. Moreover, if no single instruction exists to reverse the action of another, one can devise a small program sequence to be used to the same effect. This was the case, for example, for the integer multiply instruction in one of the ISAs that we used in our experimental evaluation. No instruction for integer division was implemented, but we could resort to software emulation of division to revert the effect of multiplication. Note that, if the emulation routine exposed any error in the hardware prototype, the result of the multiplication would not be reversed correctly. The presence of inverse functions enables us to design programs that include every instruction in an ISA, and for which the final register values match exactly the initial ones. In other terms, if $\bar{x}$ is a vector representing the processor state, and each F_i / F_i^{-1} pair represents a distinct function (either an ISA instruction or an instruction block) and the corresponding inverse, then a program generated by Reversi applies the following sequence of functions to the state $\bar{x}$:

$$\bar{x} = F_1^{-1}(F_2^{-1}(...(F_n^{-1}(F_n(...(F_2(F_1(\bar{x})..) \quad (1)$$

A. Reversible and Non-reversible Instructions

In order to create reversible programs, we first analyze each ISA and identify inverse instructions (or instruction sequences) for each of the operations. By applying these operations in the manner discussed above we can modify the state of the processor and then properly restore it (in the absence of bugs). This allows us to create a *block database* containing pairs of *functional blocks*: for each operation block, there is a corresponding inverse block. Each block contains either a single instruction or a small program sequence. An operation block modifies the value of a register, called the *focus register*, while its inverse restores its initial value. The ID of the focus register for each block is a parameter set by Reversi dynamically during test generation. Therefore, the same block may appear in the test program multiple times, each time modifying a different register, which allows a varied set of programs to be created. Note that blocks operate only on a single focus register at a time to maintain the reversibility of our program and track the correctness of its execution. Thus, for instructions with multiple operands, only one of the registers is the focus register, while other operands are randomly generated by Reversi according to the instruction format. The flexible and robust structure of the block database allows the Reversi algorithm to be agnostic to the functionality of individual blocks and the underlying ISA, making our framework readily adaptable to different processor architectures. Moreover, since blocks in Reversi may contain multiple instructions, we can populate the database with complex functions, including loops, procedure calls, *etc.*, and create elaborate tests representative of real software. In the remainder of the section we discuss individual classes of instructions and implementation details of operation and inverse block verifying them.

TABLE I- **Reversi blocks for arithmetic and logic instructions**

Instruction	Operation Block	Inverse Block
add	*add*	*sub*
sub	*sub*	*add*
inc	*inc*	*dec*
dec	*dec*	*inc*
xor	*xor*	*and/or emulated xor*
not	*not*	*nand emulated not*
neg	*neg*	*-1 mult emulated neg*
and	*and/or emulated xor*	*xor*
or	*and/or emulated xor*	*xor*
mult	*mult*	*emulated division*
rol	*rol*	*ror*
ror	*ror*	*rol*
sll	*store lost bits, sll*	*srl, restore lost bits*
srl	*store lost bits, srl*	*sll, restore lost bits*
sra	*1.store lost bits,*	*1.rol*
	2.create mask	*2.apply mask*
	3.sra	*3.restore lost bits*

Arithmetic and logic instructions. The design of blocks containing arithmetic and logic instructions is summarized in Table I and is fairly straightforward, since the majority of these operations have a simple inverse directly in the ISA. For example, *add* can be reversed by *sub*, *inc* by *dec*, *ror* (rotate right) by *rol* (rotate left) and so on. If an instruction does not have a counterpart in the ISA, a small routine can be used to emulate its inverse. Some Boolean logic instructions, such as *and* and *or*, do not have direct inverses, however, these operations can be used to construct an *xor* logic function, which can then be reversed by an *xor* instruction. Such structure is also beneficial for verification of the *xor* instruction itself, since operation and inverse block in this case exercise different hardware modules. Situations where the same processor modules are used in the function and its inverse should be avoided to prevent bugs being masked by faulty hardware.

Some ISA operations, for example *sll* and *srl* cause some of the data bits to be lost. In order to be able to restore fully the initial value of the focus register, we must mask out these bits and store them in the scratchpad memory before applying the operation. When the program reaches the inverse block, it first applies the reverse operation (*i.e.,* shift in the opposite direction in this case) and then loads and restores the bits from memory. Finally, the outcome of an instruction may depend on the sign or value of the focus register, which is not known at generation time. For example, shift-arithmetic right (*sra*) will preserve the sign of the value by replicating its most significant bit. Blocks verifying such value-dependent operations can be built to execute differently based on the operand's value, saving and restoring all the bits required to deterministically retrieve the initial data.

Load/store instructions. In Reversi the correctness of load and store instructions is checked by copying a data structure: a region of memory is initialized with random values and load/store pairs are used to copy it to a new location. We do not require that the copy preserves the order of the bytes, rather, we treat the data structure as a pool of values, which can appear out of order at destination (see Figure 2). This allows programs generated by Reversi to closely resemble real software applications where loads bring data from memory to the processor, and stores copy results of the computation back. Moreover, because of their random nature, Reversi programs contain a variety of cache and memory access patterns, that can expose corner-case bugs in the memory subsystem. To check the correctness of the final state of the memory, we simply compare an *xor-hash* of the memory values before and after test execution. This approach allows Reversi to expose load/store related issues such as illegal memory accesses and/or data corruption.

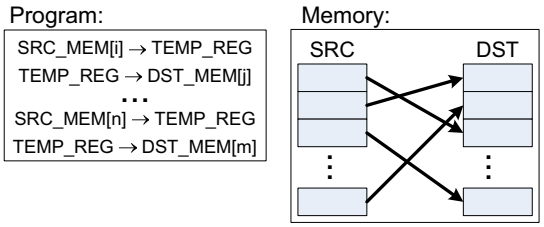

Fig. 2. **Blocks for load and store instructions.** Load/store pairs are used to copy bytes from source to destination data structures. Bytes may be reshuffled, but their xor-hashes must match.

Branch instructions. The block database of Reversi also contains templates that test branches with different properties: forward/backward, taken/nottaken *etc.* For example, an operation block for a forward taken branch (Figure 3.a) contains a store operation that saves the value of the focus register to scratchpad memory, followed by a load that overwrites the focus register with a predetermined constant and then by the branch itself. Although the constant is generated randomly, its value is dependent on the type of the tested branch. For example, a template for a *beq* (branch if equal) instruction, overwrites the focus register with a random constant and then loads a temporary register with the same value to test the branch. With reference to Figure 3.a, the destination of the branch is located in the inverse block, which also contains a load operation restoring the focus register and a return jump. Therefore, if all control flow instructions are executed correctly, the value of the focus register after execution of the block is preserved. If, however, the branch is not taken by a faulty hardware, the focus register would not be restored. Note that the inverse block is only accessible via the proper branch and is skipped otherwise. If, however, the unconditional branch in Figure 3.a is not taken due to a bug, the *halt* instruction is executed and the test stops without fully reverting processor's state. The structure of the blocks for forward not-taken branch (Figure 3.b) is similar to the one described above differing in the position of the restoring load. We use a similar technique to detect other faulty control flow operations, initializing all unused locations in the program to *halt* instructions.

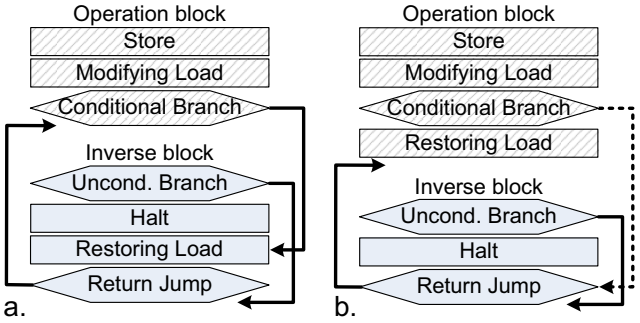

Fig. 3. **Branch operations. a.** Block pair for forward taken branch. The operation block includes a modifying load, a branch and the return label, while the inverse block contains a restoring load and a return jump. **b.** Structure of a forward not-taken branch. The dashed line indicates the program flow for a case when the branch is taken by the faulty hardware.

Control register manipulation. In many modern processors there exist several special control registers. In general terms we can classify them into two groups: *mode control* registers, that can only be accessed by special instructions and specify the machine's mode of operation; and *Execution flag* registers, that cannot be changed by the user but are affected indirectly by executed instructions. For instance, a register enabling/disabling the first level cache is a mode control register, while a register storing the ALU overflow bit or comparison bits from the comparator (*equal, greater than, etc.*) is an execution flag register.

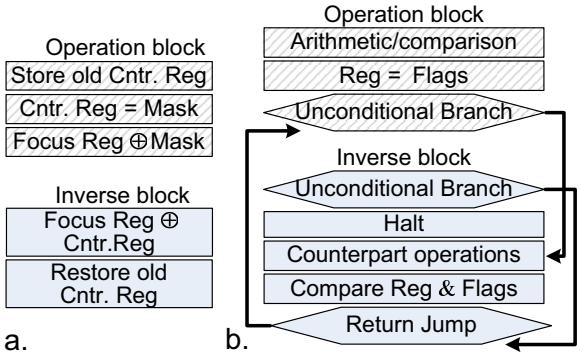

Fig. 4. **Handling of instructions affecting control flags. a.** Block pair for testing *mode control* registers. An erroneous register operation is reflected in the focus register's value. **b.** Block pair for instructions affecting *execution flag* registers. The operation block includes an arithmetic/comparison instruction setting the flag bits and copying the resulting flag vector. The inverse block performs the counterpart action and compares resulting flags with the vector from the operation block.

Reversi exploits the fact that the value of the mode control registers can be modified only through specific instructions, and must remain unchanged throughout other parts of program execution. To test the proper operation of a control register, the following sequence of steps is taken (Figure 4.a). In the operation block the old value of the control register is first stored to memory, then the new bit-mask is loaded to the control register and also *xor*ed with the focus register. In the inverse block, Reversi first accesses the value of the control register and *xor*s it with the focus register, restoring the previous mode of operation from memory afterwards. Thus, if the control register was erroneously modified between the execution of the operation and the inverse block, the focus register's value would reflect this error.

Reversi can also check the correctness of execution flag registers, because instructions affecting them (arithmetic, logic and comparison) have counterparts in terms of which flags they set. For instance, if *comp $r1, $r2* sets the *greater-than* bit, then *comp $r2, $r1* must set the *less-than* bit. Similarly, knowing that $a+b>c \equiv b>c-a$, we can check that an *add* operation sets the overflow bit in the execution flag register correctly. In this case

add $r1, $r2, $r3 # overflow *i.e. $r3 > MAX*

can be checked through subtraction and comparison:

sub MAX, $r1, $r3
comp $r2, $r3 # must set greater-than bit

where *MAX* is the largest number that can be stored in the register. Thus, individual bits of the flag register can be used to check other flag bits. The Reversi block structure for execution flag register validation is presented in Figure 4.b. The operation block executes a comparison or an arithmetic operation that affects the flags, stores the flags values in a register and jumps to the inverse block. The inverse block then executes the counterpart operations, obtains the resulting flags and checks if they correspond to previously computed ones. Recalling the example above, first the *add* executes in the operation block and then, in the inverse block, we check that if an overflow bit was set by the *add*, then a greater-than bit is set by the *comp* instruction.

Floating point instructions. Floating point operations present a unique challenge to Reversi due to the their inherent imprecision: In the majority of cases the result of the computation is rounded, making it impossible to restore the operands exactly. To address this issue we must recognize this intrinsic approximation and take into account the relative error that is introduced by these operations. We do so by constructing a table indexed by the exponents of the operands and checking the relative error after every operation against the expected boundaries. Although this solution will not lead to a strictly reversible program, the approach is still viable for floating point error detection.

Limitations Although the Reversi framework allows to create high-coverage tests with a verifiable final state, the proposed approach has a few limitations. The most important one stems from the fact that Reversi relies on the existence of inverse functions that can fully and precisely restore the internal state. For example, if the integer division operation was implemented in such a way that the remainder is lost, the value of the dividend could not be restored precisely. Similarly, floating point instructions do not exhibit such precision, however, they can still be partially verified by the approach described above. Unfortunately, input and output operations are inherently irreversible and cannot be easily covered by Reversi. For example, external interrupts cannot be expected to arrive at a certain time and cannot be "undone" by the core. In addition, Reversi may fail in targeting special execution cases for instructions whose output depends on the operands' values. For instance, a divide-by-0 operation may trigger an exception or set an error bit. Due to the randomness of operand values generated by Reversi, it is unlikely that a zero divisor occurs. To address this, the Reversi database can be augmented with specialized blocks to exercise corner case situations.

B. Reversi Generator

As described above, reversible programs generated by Reversi consist of sequences of operation and inverse blocks instantiated from the block database. However, a single sequence of blocks only alters a single focus register, therefore, to create complex programs, Reversi generates multiple block sequences (called *stacks*), each altering a different focus register. The stacks are then interleaved into a complex reversible test program, as Figure 5 illustrates.

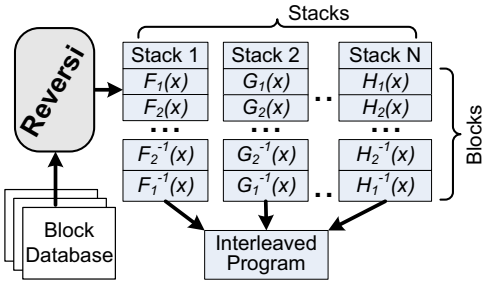

Fig. 5. **Reversi operation.** Given a database of functional blocks, Reversi produces a set of stacks, consisting of blocks and inverse operations assembled in reverse order. Each stack operates on a single focus register, modifying it in such a way that its final value matches the initial one. The stacks are then interleaved into a program with predictable outcome.

Stack generation. During the test generation, Reversi randomly selects functional blocks from the database and creates a user-specified number of *stacks*, each consisting of several blocks and their inverses. Each stack has only one focus register selected at random. The blocks are then arranged so that inverse blocks follow operation blocks in inverse order (see Eq. 1). On a properly working processor the focus register should be restored to its original value once a stack execution completes. Reversi also allocates a set of temporary registers to each of the stacks, based on the requirements of its blocks. We chose to allocate completely disjoint sets of registers to each stack to simplify the interleaving.

Stack interleaving. After the required number of stacks is generated, Reversi interleaves them by selecting instructions from all stacks and chaining them together to form a single test program. Note that some instructions may be grouped together into "atomic operations", meaning that the interleaving phase cannot insert instructions between them. The atomicity indicator is provided in the block definition in the database. To balance the selection algorithm, we attribute different probabilities of selection to each stack, based on its length, so to avoid a long tail from a single stack at the end of the program. The probability of selecting the next instruction from a given stack j is:

$$P_j = \frac{|stack_j|}{\sum_i |stack_i|}$$

where $|stack_j|$ is the number of atomic operations in $stack_j$, and all P_j's are adjusted after each removal of an atomic

operation. Note that the requirements of using disjoint sets of registers in each stack limits the total number of stacks that we can have in Reversi. We chose to forego more complex dynamic register set partitioning (as in some compiler techniques) in favor of faster test generation.

The test program includes one last routine that calculates the final *xor-hash* of the destination memory data structure. When the program terminates the final values of the focus registers and the hash of the destination memory are compared to the initial state computed by Reversi during the generation to determine if the test executed successfully. It is also important to note that Reversi programs can provide more aid in debugging than traditional randomly generated programs. If the test results indicate that there is a bug in the processor, a validation engineer can quickly check if the exposing instruction sequence is located in an individual stack, by re-running the program without interleaving. Insights into the nature of the bug can also be found by "peeling" operation and inverse blocks from the program. Therefore, a reversible program exposing a bug can be dramatically shortened to alleviate debugging. In contrast, in a traditional flow a costly re-simulation is required to obtain the new golden state after each change of the test program.

IV. Example

This section presents an example of a program generated by Reversi for a simple instruction set presented in Table II. Two stacks for this ISA using focus registers \$r7 and \$r11 are shown in Figure 6.a and 6.b. For both stacks the function blocks are indicated in the left column and boxes mark atomic actions. The stack in Figure 6.a contains simple arithmetic/logic operations, while the stack in 6.b includes logic instructions, load/store pairs and forward taken conditional branches. Sets of register IDs for both stacks are allocated dynamically by Reversi and are disjoint. Initial focus register values (*reg_val1* and *reg_val2*), constants (*const1-const3*) and location accessed by the loads and stores in the program are also selected at random.

TABLE II- **Example ISA**

Instruction	**Semantics**
halt	Stop the execution
add \$r1, \$r2, \$r3	\$r3=\$r1+\$r2
sub \$r1, \$r2, \$r3	\$r3=\$r1-\$r2
neg \$r1, \$r2	\$r2= -\$r1
ld \$r1, var	\$r1=MEM[var]
st \$r1, var	MEM[var]=\$r1
beq \$r1, \$r2, label	PC=(\$r1==\$r2) label : PC+1
Register \$r0 is hardwired to the value 0	

An interleaving of the stacks into a program is shown in Figure 6.c. Conditions that must hold after this program executes are: \$r7=*reg_val1*, \$r11 = *reg_val2* and $\bigoplus$*src_mem* =$\bigoplus$*dst_mem*. So, by using the resulting values of the focus registers \$r7 and \$r11 and the xor-hash of the *dst_mem* data structure, we can quickly determine if the program has exposed any functional bugs. Note also that the branch in block G_2 was generated by Reversi to be taken. Thus, during correct operation, the execution should modify the value of

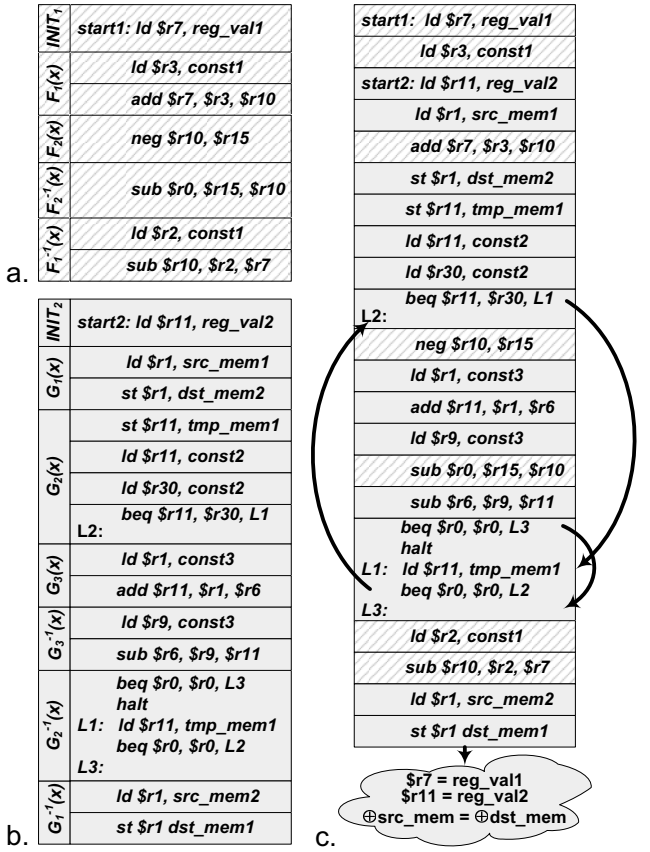

Fig. 6. **Test program for the example ISA. a.** Stack with arithmetic/logic operations. **b.** Stack with arithmetic operations, load/store pairs and forward taken branches. **c.** Interleaving of atomic operations in stacks a. and b. and exit condition of the test.

\$r11 and jump to the label $L1$. Then the processor restores the value of the focus register and takes the unconditional branch returning to $L2$. When operating properly, the processor should not visit line $L1$ again and skip directly to $L3$. Moreover, if the branch in G_2 is not taken, then the exit condition described above does not hold, exposing a bug.

V. Experimental Evaluation

In this section, we first present our experimental evaluation platform and two of our Reversi setups. Then, we evaluate the performance of these setups against a traditional solution based on a constrained-random instruction sequence generator. Finally, we investigate bug-finding capabilities of Reversi in our last experiment.

A. Experimental Framework

To evaluate the performance of our Reversi approach, we created two reversible instruction block databases: one implementing a subset of the Alpha instruction set and another implementing a subset of the x86 ISA. The database for the Alpha instruction set contained 17 distinct blocks for arithmetic and logic functions testing a range of instruction formats (reg/reg and reg/imm) and 5 blocks for each type of compare instructions. In addition to that, the database included 3 blocks for load and store instructions, an unconditional jump block and 16 branch blocks containing 4 distinct

branching instructions, each in four possible modes (fw/bw and taken/nottaken). Similarly, the x86 block database contained 32 logic-arithmetic blocks testing multiple instruction formats (reg/reg, reg/imm, reg/ mem, mem/reg), 3 load-store blocks, 1 compare block and 40 branch blocks. Reversi itself is implemented as an optimized program in C that created and interleaved a specified number of stacks and contained routines to set a random initial state and perform the final check. The blocks are partially pre-assembled in binary, and Reversi is responsible for setting the appropriate bit-fields with register IDs, randomly generated constants, *etc.*

To compare Reversi with a traditional post-silicon validation flow (Figure 1.a), we created an assembly-level constrained-random test generator. In addition, for the architectural simulation phase of the traditional post-silicon flow we used M5 2.0b3 [5] and Bochs-2.3.5 [6] for Alpha and x86 systems, respectively. Test generation and simulation for both Reversi and the traditional post-silicon flow was performed on a 3.2GHz Pentium 4 machine with 2GB of memory.

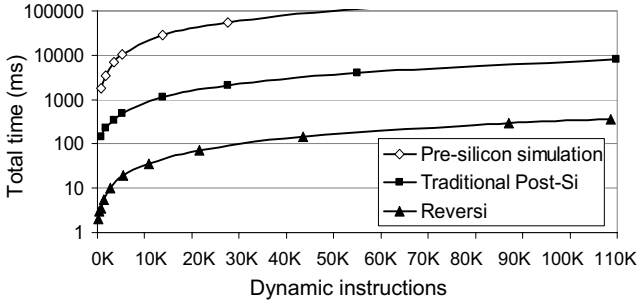

Fig. 7. **Total testing time for a traditional post-si flow and Reversi: Alpha instruction set.** The total time for the traditional post-silicon flow includes the test program generation, simulation and execution time. The time for Reversi includes test generation and execution. For comparison we plot the speed of a pre-silicon validation technique based on RTL simulation.

B. Performance Evaluation

In our first experiment we compared the validation performance of the traditional post-silicon flow with the Reversi flow. In this case the total time for the traditional flow consisted of i) the time to create a program on the constrained-random test generation, ii) the time for the instruction set simulator (either M5 or Bochs) to obtain the golden state and iii) the execution time on the silicon prototype. For the Reversi flow we need to include i) the Reversi generation time and ii) the time to execute on silicon. Performance for the Alpha design was measured over shorter program sequences, while x86 used longer testing programs. The results of the experiments are presented in Figures 7 and 8. In Figure 7 we also plot the performance of a typical pre-silicon simulator (using a behavioral Verilog model of the Alpha design) for comparison. As these figures demonstrate, the Reversi-based approach flow provides a 19.5x and 21.5x performance improvement for Alpha and x86 designs, respectively. It should be noted that in addition to eliminating the simulation step from the flow, Reversi is more efficient because it operates on pre-assembled blocks. In a traditional approach, on the other hand, the generator must frequently solve fairly complex constraints to produce valid and meaningful tests. Moreover, due to the presence of branching instructions and PC-relative branches, the program generator must produce tests in assembly language and then call an assembler to convert it to machine codes. Reversi, however, does not need an external assembler, since it implements internally all functions required to generate the binary code.

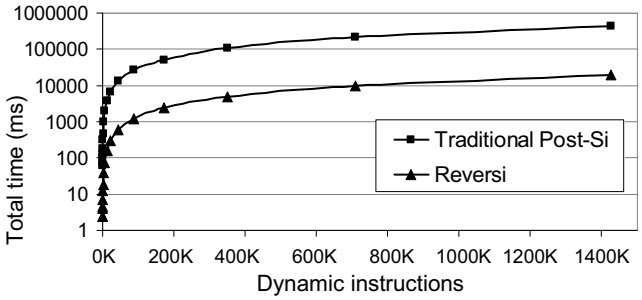

Fig. 8. **Total testing time for a traditional post-si flow and Reversi: x86 instruction set.** The total time for the traditional post-silicon flow includes the test program generation, simulation and execution time. The time for Reversi includes test generation and execution.

C. Design Error Coverage

In the second experiment, we use an RTL implementation of a 5-stage pipeline running Alpha ISA to create 20 designs, each containing a single bug from Table III. During the test, both the traditional flow and Reversi generated code of increasing length until the bug was exposed. Note that, in order to identify an error with a randomly generated program, we first need to compute the correct final state by running it on a known-correct model. We run the experiment 10 times with different random seeds and calculate the minimum, average and maximum time required for the traditional flow and Reversi to expose the fault (Figure 9).

TABLE III- **Bugs introduced in Alpha design.**

Bug	Description
ld_st_addr	load to store address forwarding fault
regfile_rd	faulty internal forwarding in register file read port A
fwd_mem	error in forwarding dependency resolution
fwd_reg31	forwarding through register 31 (const 0)
ucbr_cbr	unconditional branch after conditional branch fails
fwd_wb	unnecessary forwarding from wb stage
regfile_wr	invalid write access to register file
flush	pipeline flush on specific register file access
srl	invalid execution of logical right shift
scmp_cbr	invalid forwarding from signed compare to a branch
cbr_st	backward conditional branch after a store is not taken
ld_st_data	load to store data forwarding fault
ucmp_cbr	invalid forwarding from unsigned compare to a branch
back_cbr	specific backward conditional branch is never taken
add_over	incorrect handling of overflow on add
loop	incorrect execution of looping sequence
jsr	incorrect handling of jsr with invalid address
back_ucbr	fault in backward unconditional branch
sh_back_br	fault in branch resolution for short backward branch
ld_arith	invalid execution of a load followed by arithmetic

As the results in Figure 9 demonstrate, Reversi can find all errors faster than the traditional post-si flow. Furthermore, some of the bugs, such as *loop, jsr* and *sh_back_br*, were not exposed by the post-si flow in any of the runs. We believe that this was due to the unique nature of the programs generated by Reversi - they are designed so that only correctly operating hardware produces an easily verifiable result.

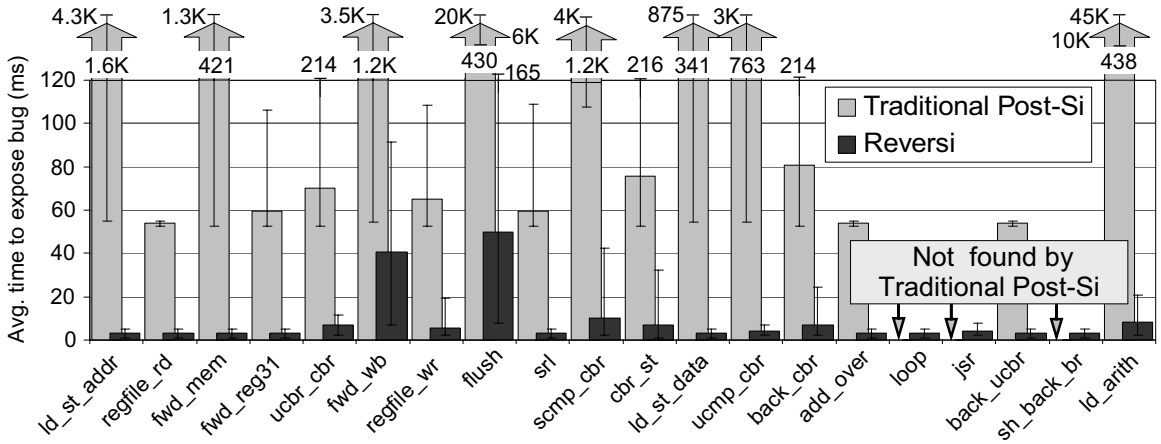

Fig. 9. **Average time to discover bugs in the traditional post-silicon flow and Reversi.** The experiments were run 10 times with different random seeds and the minimum, average and maximum times to expose each bug are plotted. Note that bugs *loop, jsr* and *sh_back_br* were not exposed by a traditional post-silicon flow based on a constrained-random test generator.

Thus, incorrect operations can be detected immediately at execution completion. Moreover, Reversi creates complex programs with multiple interleaved execution flows that exercise all instructions in the ISA, exposing these corner-case bugs.

It's worth observing that, in several experiments with the traditional flow (such as *fw_wb*), a shorter random program exposed a bug, while a longer sequence of instructions did not. This is possible due to the random nature of the test: later instructions may overwrite registers/memory locations that contain incorrect values, thus eliminating the evidence of the bug. Therefore, a longer random program does not necessarily find more bugs than a shorter one. Reversi programs, on the other hand, are designed so that any behavior corrupting the processor state is propagated to the exit point and exposed.

VI. Conclusions and Future Work

In this paper we presented a novel post-silicon validation methodology that exploits the performance potential of hardware prototypes and bypasses the design simulation step required by traditional flows. Test programs that our Reversi framework generates work to explore complex execution scenarios and, most importantly, have identical initial and final architectural states eliminating the need for a simulator to check the correctness of the test. The programs are built from sequences of functional blocks, which modify the state of the machine, and they are combined with inverse blocks to undo earlier operations and restore the original machine state. Individual blocks are parameterized and may consist of one or several instructions, selected randomly from a block database during test generation. Reversi handles all types of instructions: arithmetic (integer and floating point), logic, memory accesses, control flow and control register operations. As our results demonstrate, Reversi creates programs capable of finding more bugs faster than traditional constrained-random test generation techniques. Moreover, due to the omission of the architectural simulation step, Reversi can generate and run tests 20x faster than tools based on a traditional post-silicon flow.

In the future, we plan to optimize Reversi to only require minimal resources, such as OS primitives, I/O drivers, *etc.*, so that we can run it on the same board as the device under test. The programs in this case can be generated by a more reliable or thoroughly tested previous generation processor more efficiently than in our experiments. We also foresee the possibility of running our generator on a subset of the cores of a multi-core device-under-test. This would allow Reversi to achieve generation speeds that significantly exceed the performance of today's methods and approach a throughput comparable to actual silicon.

References

[1] Constrained-random test generation and functional coverage with Vera. Technical report, Synopsys, Inc, Feb. 2003.

[2] *Revision Guide for AMD Athlon 64 and AMD Opteron Processors*, Aug. 2005.

[3] *Intel Core2 Duo Desktop Processor E6000 and E4000 Sequence Specification Update*, Nov. 2007.

[4] *Intel Core2 Extreme Quad-Core Processor QX6000 Sequence and Intel Core2 Quad Processor Q6000 Sequence*, Nov. 2007.

[5] The M5 simulator system, Nov. 2007. http://www.m5sim.org.

[6] The open source IA-32 emulation project, Sept. 2007. http://bochs.sourceforge.net/.

[7] A. Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, Mar. 2004.

[8] B. Bentley and R. Gray. Validating the Intel Pentium 4 microprocessor. *Intel Technology Journal, Q1*, pages 1–8, 2001.

[9] K. H. Bierman et al. *U.S. Patent no. 7133818: Method and apparatus for accelerated post-silicon testing and random number generation*, Nov. 2006.

[10] T. Litt. Support for debugging in the Alpha 21364 microprocessor. In *International Test Conference*, Oct. 2002.

[11] M. L. Bushnell, V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory & Mixed-Signal VLSI circuits*. Springer, 2000.

[12] M. Melani *et al*. An integrated flow from pre-silicon simulation to post-silicon verification. In *Research in Microelectronics and Electronics 2006, Ph. D.*, pages 205–208, June 2006.

[13] P. T. Barch et al. *U.S. Patent no. 5923836: Testing integrated circuit designs on a computer simulation using modified serialized scan patterns*, Nov. 2006.

[14] R. Emek et al. X-Gen: A random test-case generator for systems and SoCs. In *International Workshop on High Level Design Validation and Test*, pages 145–150, Oct. 2002.

[15] R. Raina and R. Molyneaux. Random self-test method - applications on PowerPC microprocessor caches. In *Proceedings of the Great Lakes Symposium on VLSI*, Feb. 1998.

[16] H. Rotithor. Post-silicon validation methodology for microprocessors. *IEEE Design & Test of Computers*, 17(4):77–88, Oct. 2000.