Selfie5: An autonomous, self-contained verification approach for high-throughput random testing of programmable processors

Yehuda Kra, *Student Member, IEEE*, Naama Kra and Adam Teman, *Member, IEEE*
EnICS Labs, Faculty of Engineering, Bar-Ilan University, Ramat-Gan, Israel

***Abstract*—Random testing plays a crucial role in processor designs, complementing other verification methodologies. This paper introduces Selfie5, an autonomous, self-contained verification approach that utilizes the device under verification (DUV) itself to generate, execute, and verify random sequences. This approach eliminates the overhead associated with testing environment interfaces, resulting in a substantial increase in throughput, a critical aspect for achieving comprehensive coverage. The utility can be deployed to FPGA prototypes, emulation platforms and fabricated ASICs and run at-speed to execute billions of tested scenarios per hour, while ensuring the reproducibility of captured failures in an observable simulation environment. This paper describes the Selfie5 approach, algorithms and utility, while also providing detailed insights into successful deployment of the utility for a RISC-V implementation. When deployed on a 16 nm test SoC featuring a RISC-V processor, Selfie5 delivered a testing throughput of 13.8 billion tested instructions per hour, which is 69 × higher than other published works.**

***Index Terms*—Random verification, post-silicon validation, self-contained, high-throughput testing, RISC-V, open source utility**

I. Introduction

FUNCTIONAL verification is an essential aspect of any processor design cycle. Multiple testing strategies are applied at different stages of the project. At the initial stage, high-level architectural models are used to validate the proposed design architecture. As the design progresses, specification compliance test suites are regularly verified on an implementation-driven detailed model. Towards design release, massive random testing is performed by prototyping the design on FPGA boards or by mapping it to hardware emulation platforms, such as Cadence Palladium. Finally, massive post-silicon testing continues on the manufactured prototype during system bring-up. To successfully complete this process, it is vital to apply the most effective verification method for each design phase, addressing complementary objectives.

For programmable hardware blocks, such as general-purpose microprocessors, basic instruction set architecture (ISA) compliance is mostly achieved at a relatively early stage of the design. ISA compliance proves that each applied instruction implements its specified functionality when examined on its own. However, a large percentage of the potential design bugs reside in the internal control of the processor and within processes that communicate with the memory and system components. These include incorrect handling of dynamic scenarios, such as pipeline hazards, cache misses, cache coherence, system stalls, instruction ordering, interrupts, and many more.

The number of instruction type permutations combined with execution of dynamic events is enormous, such that directed testing alone is insufficient to effectively identify all potential failures. Therefore, random testing is an essential component of a complete verification plan. Unfortunately, the simulation runtime limits the testing throughput. FPGA-prototyping and hardware emulation platforms help overcome this bottleneck; however, they are still limited by communication channels and remote generation and checker interface overhead.

In this work, we propose Selfie5, a novel verification approach that is completely autonomous and self-contained. In other words, the device under verification (DUV) (e.g., microprocessor or other programmable block) itself is used to generate, execute and verify random control sequences without the need for external components during execution. The proposed test approach is entirely run on the DUV, including test generation and checkers, and therefore communication overheads with these components are completely eliminated. Furthermore, Selfie5 can be run at all levels of verification, e.g., simulation, FPGA-prototyping, hardware emulation, and post-silicon. And since it is self-contained, execution on a post-silicon evaluation board can be carried out at the target speed of the DUV platform, thus enabling achieve maximum throughput, while also dynamically stressing the hardware.

In this paper, we overview the concept of Selfie5 and describe the algorithms that enable its novel, ISA-independent, self-testing approach. We demonstrate the utility with a RISC-V adaptation module to run the testing on both an FPGA-platform and a number of fabricated systems-on-chip (SoCs) featuring RISC-V cores. When deployed to a 16 nm SoC, running at 1 GHz, Selfie5 was able to deliver a testing throughput of 13.8 billion instructions per hour, which is 69 × higher than other reported approaches, such as [1], reporting 200 million instructions per hour. The massive testing capabilities were used to verify these designs and successfully exposed true design bugs, which escaped compliance and other common testing strategies.

II. Beyond State-of-the-Art

Methods of random testing for both pre-silicon verification and post-silicon hardware validation are well established and discussed in the literature [1]–[7]. ISA compliance testing for programmable hardware is also commonly applied, such as RISC-V compliance testing that has been gaining interest in recent years [4], [5]. However, few publications have suggested the idea that the DUV can be effectively utilized to validate itself to achieve the highest testing throughput.

In multiple publications, Herdt et al. [1]–[3] have contributed advanced processor testing methods. In [2], RISC-V ISA compliance tests are automatically generated and in [3], this is complemented with a fuzzing-based approach for negative testing coverage. A reference instruction set simulator (ISS) is used in [1] to verify an endless instruction stream that dynamically evolves during hardware model simulation. However,

979-8-3503-4859-0/DATE24/© 2024 EDAA

the throughput is limited by the trade-off between simulation runtime and the level of abstraction of the model, while also requiring high observability of the DUV state, which may not be efficient in a post-silicon validation framework.

Several inspiring publications based on experience with the IBM Power processors address the transition from pre-silicon verification to post-silicon validation [8]–[11]. A unified methodology is suggested for sharing test templates for random testing generation in both environments, promoting pre-silicon prototyping to maximize observability and controllability, while benefiting from the high testing throughput of post-silicon.

Another method, suggested by [12], generates test programs out of templates. However, manually created test templates may limit the randomness of the test flow, reducing coverage of potential hidden failing execution scenarios.

The approach we suggest achieves high-throughput random self-testing on the DUV, both during FPGA emulation and upon the post-silicon validation prototype. The method completely eliminates communication overhead with the off-DUV test bench, yet failing scenarios are easily identified and isolated for fast turn-around reproduction and analysis in a fully observable simulation setup. The next sections will present the overall utility architecture, followed by a detailed description of the components and algorithms that enable this functionality.

III. Selfie5 Architecture Overview

Selfie5 is a self-contained verification utility that utilizes the DUV processor compute power to generate random tests, execute the tests, and check the test output for correctness. The utility is compiled, linked and loaded onto the target platform, and then can run continuously at target speed until a bug is found. The Selfie5 architecture is presented in Fig. 1. The utility is divided into two main sections: the Front-End, which generates the test and expected results, and the Back-End, which maps the tests into the target platform ISA and executes them.

The Front-End is ISA-independent and can be applied to any programmable platform that supports branches and a stored-program memory. It consists of the **Utility Manager**, which issues the test seeds and iteratively launches the tests, followed by the **Flow Generator** module. The Flow Generator creates an execution order graph, described in the abstract Flow syntax, made up of three commands:

- `STEP`, proceed to the next consecutive instruction;
- `GOTO`, redirect the program to another location;
- and `FINISH`, terminate the test.

These abstract commands are randomly generated in a sequence that ensures that a program that starts from the input to the graph, covers all the nodes in the graph, and eventually reaches the `FINISH` command.

The Flow code is then passed on to the **Generic Code Generator** (GCG) module that translates the execution order graph into a sequence of generic, ISA-independent commands in the Gcode syntax. Each `STEP` or `GOTO` instruction is assigned a randomly selected non-branching or branching generic instruction, ensuring that the flow order is preserved. The GCG module maintains a set of virtual generic registers and a data

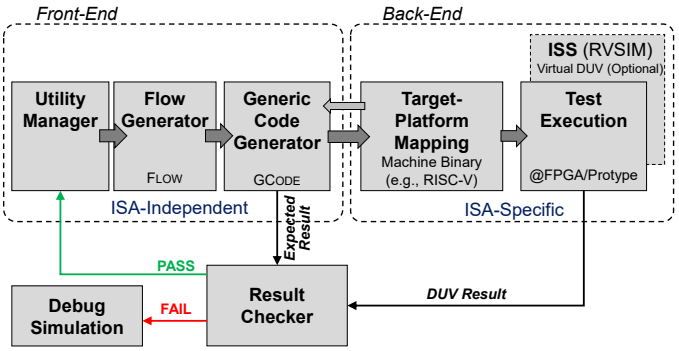


Fig. 1. Selfie5 Utility Architecture.

memory image, which are tracked and retained for the checking stage. Due to branch destination dependence on the platform-specific ISA (e.g., variable-length instructions or mapping one Generc Code (Gcode) command to multiple ISA instructions), some of the generic virtual registers contain placeholders that are only filled in during Mapping (e.g., actual return address linking).

The resulting Gcode is passed to the Back-End of the Selfie5 utility, where ISA-specific instructions are assembled and deployed. First, the **Target-Platform Mapping** (TPM) module translates the pseudo-assembly Gcode language into a DUV-specific ISA. The TPM module ensures that each translated Gcode instruction yields the exact expected impact on the registers and memory images.

The resulting ISA-specific instruction sequence is transferred from the TPM module into the **Test Execution** module. The Test Execution module loads this binary into an allocated memory section and directs the DUV's Program Counter (PC) to that location to natively execute the generated code sequence. Upon completion, the post-execution image (DUV memory and virtual registers content) is compared against its expected content, which was earlier generated by the GCG module.

Even though both the checker and the test sequence are executed by the DUV, they originate from exclusive independent sources at different levels of abstraction. While the final result is provided by the DUV running at the ISA-specific machine code level, the expected results are obtained at the Gcode abstraction interpretation level. This prevents the concern of self-testing error cancellation.

As a final note, the Selfie5 utility can also interface with an ISS, which can replace the actual DUV model for utility development and debugging purposes. In this work, we implemented a RISC-V ISS, called RVSIM. However, the ISS module, which is resource-intensive at run time, is not needed and is excluded during operational testing, as the expected results are efficiently calculated by the GCG module.

IV. Selfie5 Front-End Components and Algorithms

A. Flow Generator

The Selfie5 approach requires generating an executable random assembly code section for testing purposes. This code is functionally meaningless, but needs to feature predictable behavior to enable final state checking. In addition, the generated code must be constrained to avoid out-of-range branch targets and memory access locations, it needs to prevent infinite

Algorithm 1 Execution flow generation algorithm

```
1:  procedure FLOW_GEN(N)
2:      i ← 0 ; cmd_idx ← 0
3:      L[0 : N − 1].is_avail ← TRUE
4:      while i < N − 2 do                          ▷ Generate Loop
5:          L[cmd_idx].is_avail ← FALSE
6:          if NOT L[cmd_idx + 1].is_avail then
7:                                  ▷ Next unavailable, must jump.
8:              L[cmd_idx].cmd ← GOTO
9:          else
10:             L[cmd_idx].cmd ← RAND([GOTO, STEP])
11:         if L[cmd_idx].cmd == STEP then
12:                                 ▷ STEP: Continue to next location.
13:             next_idx ← cmd_idx + 1
14:         else                    ▷ GOTO: Jump to random location.
15:             next_idx ← RAND_AVAIL(0, N − 2)
16:             L[cmd_idx].dest ← next_idx
17:         cmd_idx ← next_idx
18:         i ← i + 1
19:     ▷ Iteration N − 2 should jump to the end.
20:     L[cmd_idx].cmd ← GOTO
21:     L[cmd_idx].dest ← N − 1
22:     L[N − 1].cmd ← FINISH
```

loops and must eventually reach an execution exit point to return control to the test utility upon completion. A timeout mechanism is insufficient as a trigger for test completion, as it is subject to system-level factors, such as real-time stalls and interrupts, and therefore, cannot be guaranteed to end with a predictable post-execution register and memory state.

The responsibility of Selfie5's Flow Generation module is to guarantee a fully predictable entry point, exit point, and program flow. To achieve this, the Flow Generation module outputs a list of operations in FLOW code, which supports three abstract commands: `STEP`, `GOTO`, and `FINISH`. The `STEP` command is a placeholder for a non-redirecting instruction that will later be assigned to some ALU or memory access (load/store) instruction, or to a guaranteed non-taken branch condition. The `GOTO` command is a placeholder for execution redirection to a non-consecutive address. It can be mapped to a non-conditional jump instruction or a conditional branch that is guaranteed to be taken. Accordingly, the produced FLOW code must satisfy the following constraints:

1) A test will be constructed of N commands, accessed in a randomly generated order[1].
2) Each command location (L) must be visited once and only once.
3) The flow starts at location $L[0]$ and ends at location $L[N]$.

The process described in Algorithm 1 was designed to meet these constraints. The algorithm assigns an abstract command (`STEP` or `GOTO`) for each of the N instructions (code locations). At each iteration of the Generate Loop (Line 4), the current instruction is randomly selected to be a `STEP` or `GOTO` command (Line 10). In the case of a `STEP`, the instruction index is simply incremented by one (Line 11); however, if the next index has already been visited, the current instruction is automatically assigned a `GOTO` command (Line 8). In the case of a `GOTO`, a redirect destination is randomly chosen from the available locations (Line 15) and the instruction index is updated to be this destination (Line 17). To ensure that the algorithm terminates at the last instruction ($L[N-1]$), the Generate Loop only runs for $N-2$ iterations and the final instruction index is automatically assigned a `GOTO` redirecting to the last location (Line 20), which is set to be a `FINISH` command (Line 22).

A Flow Generation example is provided in Fig. 2. The N iterations of the algorithm are shown in order in Fig. 2(a), noting the current location (LOC column), the choice of a `STEP` or `GOTO` command (CMD column), and the chosen redirection destination (DEST column) for `GOTO` operations. The corresponding FLOW code ordering is shown in Fig. 2(b) with arrows annotating the progression of the algorithm across command locations until terminating at the `FINISH` command. Fig. 2(c) and Fig. 2(d) show the outputs of the GCG module and TPM module, respectively, as covered hereafter.

B. Generic Code Generator (GCG)

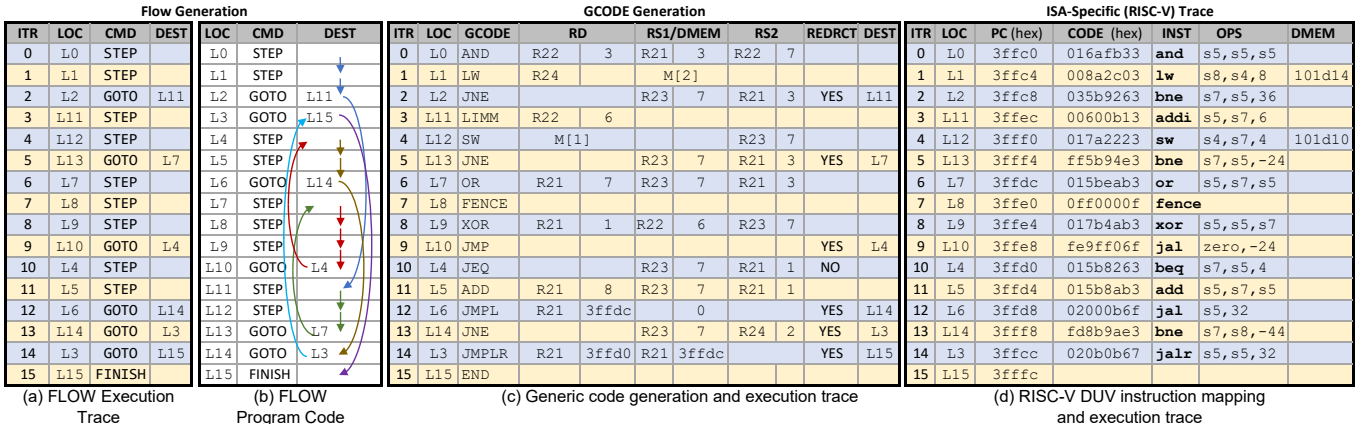
The purpose of the Generic Code Generator module is to convert the abstract `STEP` and `GOTO` commands from the FLOW code into random machine-independent instructions that represent the functionalities of a generic programmable block. This is achieved by traversing over the sequence in *execution order* (rather than location order) and generating a (random) GCODE instruction for each FLOW instruction. The set of GCODE instructions currently follows constructs common to RISC processors, but can be easily extended or modified according to the specific features and ISA of the target DUV. The execution of every GCODE instruction is then emulated, and the state of the virtual registers and data memory is stored. This information is used both to generate GCODE instructions that maintain the FLOW execution order, as well as to provide the required state at the end of the test for ensuring correct execution and identifying failures (bugs).

Algorithm 2 describes the conversion process from FLOW commands ($L[idx]$) to GCODE instructions ($G[idx]$). Initially, all registers and the memory space are assigned with random values (Line 2). `STEP` instructions are mapped into randomly selected non-redirecting instructions, including ALU, memory access (Line 9), or guaranteed non-taken conditional branches (NTCBs) (Line 12). The only constraint is that the memory and register operands should be within their legal range. `GOTO` instructions are mapped into either non-conditional jumps (NCJs) (Line 26) or conditional branches that are guaranteed to be taken (TCBs) (Line 21).

For a TCB or NTCB branch instruction, two source register operands and a relation condition operator are randomly selected. The operator is assigned to the GCODE instruction if the condition yields a `TRUE` result for a TCB or a `FALSE` result for a NTCB. Otherwise, the complementary condition is assigned (e.g., *not-equal* instead of *equal*) (Lines 15 and 24).

Once the current GCODE instruction has been successfully recorded, it is simulated and the virtual register and data memory state are updated. The algorithm then iterates to the next command in the FLOW execution. The final state of the

[1] N is a configurable parameter that sets the size of the code block to be executed during each test.



ITR	LOC	CMD	DEST
0	L0	STEP	
1	L1	STEP	
2	L2	GOTO	L11
3	L11	STEP	
4	L12	STEP	
5	L13	GOTO	L7
6	L7	STEP	
7	L8	STEP	
8	L9	STEP	
9	L10	GOTO	L4
10	L4	STEP	
11	L5	STEP	
12	L6	GOTO	L14
13	L14	GOTO	L3
14	L3	GOTO	L15
15	L15	FINISH	

LOC	CMD	DEST
L0	STEP	
L1	STEP	
L2	GOTO	L11
L3	GOTO	L15
L4	STEP	
L5	STEP	
L6	GOTO	L14
L7	STEP	
L8	STEP	
L9	STEP	
L10	GOTO	L4
L11	STEP	
L12	STEP	
L13	GOTO	L7
L14	GOTO	L3
L15	FINISH	

ITR	LOC	GCODE	RD		RS1/DMEM		RS2		REDRCT	DEST
0	L0	AND	R22	3	R21	3	R22	7		
1	L1	LW	R24		M[2]					
2	L2	JNE			R23	7	R21	3	YES	L11
3	L11	LIMM	R22	6						
4	L12	SW	M[1]				R23	7		
5	L13	JNE			R23	7	R21	3	YES	L7
6	L7	OR	R21	7	R23	7	R21	3		
7	L8	FENCE								
8	L9	XOR	R21	1	R22	6	R23	7		
9	L10	JMP							YES	L4
10	L4	JEQ			R23	7	R21	1	NO	
11	L5	ADD	R21	8	R23	7	R21	1		
12	L6	JMPL	R21	3ffdc		0			YES	L14
13	L14	JNE			R23	7	R24	2	YES	L3
14	L3	JMPLR	R21	3ffd0	R21	3ffdc			YES	L15
15	L15	END								

ITR	LOC	PC (hex)	CODE (hex)	INST	OPS	DMEM
0	L0	3ffc0	016afb33	**and**	s5,s5,s5	
1	L1	3ffc4	008a2c03	**lw**	s8,s4,8	101d14
2	L2	3ffc8	035b9263	**bne**	s7,s5,36	
3	L11	3ffec	00600b13	**addi**	s5,s7,6	
4	L12	3fff0	017a2223	**sw**	s4,s7,4	101d10
5	L13	3fff4	ff5b94e3	**bne**	s7,s5,-24	
6	L7	3ffdc	015beab3	**or**	s5,s7,s5	
7	L8	3ffe0	0ff0000f	**fence**		
8	L9	3ffe4	017b4ab3	**xor**	s5,s5,s7	
9	L10	3ffe8	fe9ff06f	**jal**	zero,-24	
10	L4	3ffd0	015b8263	**beq**	s7,s5,4	
11	L5	3ffd4	015b8ab3	**add**	s5,s7,s5	
12	L6	3ffd8	02000b6f	**jal**	s5,32	
13	L14	3fff8	fd8b9ae3	**bne**	s7,s8,-44	
14	L3	3ffcc	020b0b67	**jalr**	s5,s5,32	
15	L15	3fffc				

Fig. 2. Flow Program and conversion trace example. (a) FLOW execution order (N successive cycles). (b) FLOW program line count (location) order. (c) GCODE generation and execution trace. (d) ISA-specific instruction mapping to RISC-V machine code.

Algorithm 2 FLOW to GCODE conversion algorithm

```
1:  procedure FLOW_TO_GCODE
2:      Randomly initialize registers and data memory
3:      i ← 0, idx ← 0                    ▷ i=Iteration Count, idx=inst. index
4:      while i < N do                                  ▷ Conversion Loop
5:          RD, RS1, RS2 ← RAND_REG()
6:          if L[idx].cmd==STEP then
7:              type ← RAND([STEP, NTCB])
8:              if type == STEP then
9:                  STEP_OP ← RAND(LIMM, LW, XOR, etc.)
10:                 G[idx].op ← STEP_OP(RS1, RS2)
11:             else if type == NTCB then
12:                 COND_OP ← RAND(JEQ, JNE, etc.)
13:                 if COND_OP(RS1, RS2) == FALSE then
14:                     G[idx].op ← COND_OP(RS1, RS2)
15:                 else G[idx].op ← COND_OP(RS1, RS2)
16:             idx ← idx + 1
17:         else if L[idx].cmd == GOTO then                 ▷ Redirect
18:             G[idx].dest ← L[idx].dest
19:             type ← RAND(TCB, jmp, jmpl, jmplr)
20:             if type == TCB then                         ▷ Branch Taken
21:                 COND_OP ← RAND(JEQ, JNE, etc.)
22:                 if COND_OP(RS1, RS2) == TRUE then
23:                     G[idx].op ← COND_OP(RS1, RS2)
24:                 else G[idx].op ← COND_OP(RS1, RS2)
25:             else                                        ▷ Non-conditional Jumps
26:                 if type == jmp then G[idx].op ← JMP
27:                 else                                    ▷ Linked Jumps
28:                     R[RD] ← returnAddress()
29:                     G[idx].linkReg ← RD
30:                     if type == jmpl then
31:                         G[idx].op ← JMPL
32:                     else                                ▷ Indirect Jump
33:                         G[idx].dest ← calculateOffset()
34:                         G[idx].op ← JMPLR
35:             idx ← L[idx].dest
36:         Simulate execution of current instruction
37:         Update register and data memory state
38:         i ← i + 1
```

virtual registers and data memory after N iterations is used for checking the correctness of the execution on the target platform.

Fig. 2(c) provides an example of such a conversion, based on the FLOW generation of Fig. 2(a) and (b). For instance, the GOTO command of instruction ITR 2 (at LOC L2) is mapped into a *Jump-Not-Equal* (JNE) conditional branch. This branch is guaranteed to be taken (TCB), since the randomly selected

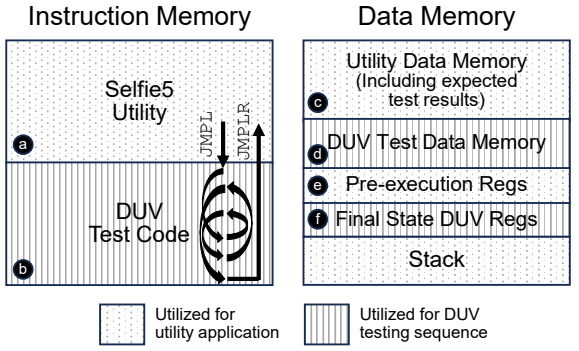


Fig. 3. Test Flow generation.

registers (R23 and R21) were initialized to the random values of 7 and 3, respectively, and therefore, are not equal. Similarly, the mapping of the STEP command of ITR 10 (at LOC L4) to a *Jump-Equal* (JEQ) branch instruction is applied, as the randomly selected R21 and R23 registers are not equal, such that the branch is guaranteed not to be taken (NTCB).

As previously emphasized, the GCODE program is executed along with its creation. After mapping each instruction, the virtual register or memory destination is updated according to the execution flow. For example, in the case of Fig. 2(c), the value of R21 was randomly initialized to 3, as can be seen at ITR 2, but is then updated in ITR 6 and ITR 9, such that its value is 1 when it is fed to the JEQ instruction of ITR 10. This approach also enables checking the final state of the registers and memory versus the expected state according to the simulated execution during test generation.

Note that linked jumps (i.e., jal and jalr commands in RISC-V) require special handling that is covered in lines 26 to 34 of Algorithm 2. The algorithm differentiates between three types of non conditional jumps:

1) JMP: Simple direct jumps to an absolute program index.
2) JMPL: Linked jumps, where the current program location (PC) is saved into the destination register (Line 28).
3) JMPLR: Indirect jumps, where the destination address is calculated as an offset from a base register (Line 32).

The actual value of the program counter, however, is only derived during the TPM stage, as it is ISA-dependent, as elaborated upon below. Therefore, the GCG module maintains placeholders for absolute values of the return address along

that are updated following their calculation along with branch destination values during Target-Platform Mapping.

V. Back-End Components and Verification Flow

A. Target-Platform Mapping (TPM)

During target-platform mapping, each GCODE instruction is mapped to a corresponding instruction in the target ISA; if several instructions provide equivalent functionality, one of them is randomly selected. Each GCODE instruction is assigned a sequentially increasing address and the destination fields of the redirection instructions (taken branches) are calculated accordingly. Mapping of the GCODE generated in Fig. 2(c) to RISC-V machine code is shown in Fig. 2(d) along with its disassembled instructions and changes to data memory state.

Note that certain GCODE instructions may require more than one ISA instruction to complete. In addition, various instruction sizes may be supported (e.g., compressed instructions in RISC-V). Therefore, the translation between GCODE location and ISA-instruction address must be calculated by the TPM module. Examples of this can be seen in Fig. 2(d) at ITR 12 and 14 for the RISC-V `jal` and `jalr` instructions, respectively.

B. DUV Test Execution

The Selfie5 utility, described in the previous sections, is coded in a high-level language (C/C++), compiled to the target-platform ISA, and the binaries are loaded onto the target-platform. As such, the utility code, which is independently coded at a higher level of abstraction, is not used for hardware verification, and therefore is not biased by the implementation to be tested. The verification is done by executing the randomly generated code within the Test Execution module. This is applied using the memory mapping and invocation scheme illustrated in Fig. 3. Since the utility is coded in C/C++ and the generated DUV test is provided as machine code, assembly-coded functions are used to maintain the target ISA application binary interface (ABI) and calling convention standards.

The instruction memory is partitioned into two parts, the first ⓐ storing the application code of the Selfie5 utility, and the second ⓑ which is loaded with the randomly generated DUV test code during each iteration.

In addition to the stack, the data memory is partitioned into four sections: the utility data memory ⓒ, the DUV Test data memory ⓓ, the pre-execution registers ⓔ, and the final state (virtual) registers ⓕ. Prior to invoking the DUV test, the registers that will be affected by the test are saved in ⓔ, and the program jumps to ⓑ to execute the test sequence. Upon completion of the test, the resulting register content is copied to memory ⓕ. The pre-test register content is then restored, and the program returns to the utility application context ⓐ for result checking. The virtual registers and memory content are compared against the expected content, as captured during the test generation during the GCODE stage.

An important issue to point out is the testing within the overall system context. Since the Selfie5 verification approach is to apply massive stress to reach corner cases in the control logic, accurate management of interrupts is key. Such interrupts impact the memory and register contents, and therefore, the Selfie5 result checker will confirm proper entry into and exit from the interrupt context. Accordingly, random interrupts are optionally applied at the system level during Selfie5 testing to verify the correct handling of interrupts under corner cases by confirming that the expected post execution state of both the interrupted and interrupting code is not affected.

TABLE I
Utility Composition and Throughput

Utility Stage	Cycles per tested instr.	%
FLOW Gen.	92	35%
GCODE Gen.	114	43%
Target ISA Map	38	15%
Execute & Check	17	7%
Total	261	100%

Testing Throughput Giga tested Instr/Hour	
40 MHz FPGA	0.551
1 GHz ASIC	13.8

As an alternative to DUV testing on an FPGA or ASIC prototype, an ISS can be used to support more efficient debugging, as well as for further development of the utility in absence of a specific hardware DUV. The current version of the emulator, RVSIM, is a RISC-V machine code interpreter, coded in C and adopted from the open-source MPRC-RISCV-simulator [13]. However, for the sake of testing throughput, the ISS should be disabled during massive DUV testing and enabled only for debugging and development purposes.

C. Result Checker and Failure Handling

Following test execution, the context is passed back to the Selfie5 utility, which compares the expected state of the registers and data memory, generated during the GCODE stage, versus their content produced by the DUV. A successful test will flag the Utility Manager to generate a new test, and the process will be repeated. As long as the DUV operated correctly, the potentially endless random testing generation, execution, and checking will continue. A failed test will stop the execution and the test seed that caused the failure will be output. Since each random test is self-contained with no dependence on the leftover state of previous tests, this seed allows for complete reproduction of the failing test in a simulation environment for analysis and debugging within a short turnaround time.

In a less desirable scenario, a bug in the DUV could cause the system to fail without finishing the Selfie5 test iteration. This could be caused, for example, by the test branching out of the tested code section, leading to unpredictable behavior. To address such cases, a watchdog timer is set when each test is launched and the seed of the most recently launched test is stored in a protected memory location or in a SoC-level register. If a test does not complete within a reasonable expected duration, the recently recorded seed is indicated and used in a simulation environment for reproducing the failing case.

VI. Implementation, Usage and Comparison

The Selfie5 utility was used to massively test an internally developed RISC-V core [14] within an SoC platform upon a Cyclone-IV FPGA board, as well as three fabricated ASICs [15], [16]. This section will present the verification performance of the utility and a comparison with similar works followed by a use-case example that exposed a bug in silicon.

TABLE II
COMPARISON OF FEATURES WITH RELATED WORK

Comparison Factor		This Work	Herdt [1]	Tran [6]	Chupilko [13]	Adir [10]
Observability	Failure Reproduction	✓				✓
	Coverage Guided	×	✓	✓	✓	✓
Supported Environments	Instr. Set Sim. (ISS)	✓	✓	✓	✓	✓
	Simulation	✓	✓	✓	✓	✓
	Accelerators	✓	×	×	✓	✓
	FPGA	✓	×	×	✓	✓
	Post-silicon	✓	×	×	✓	✓
Development Stages	Design	×	✓	✓	✓	✓
	Sign Off	✓	✓	✓	✓	✓
	Post-silicon	✓	×	×	✓	✓
DUV Scope	Core level	✓	✓	✓	✓	✓
	System level	✓	×	×	×	✓
	ISA compliancy	×	×	✓	✓	
	Pipeline Control	✓				
Random-ization	Registers, Memory	✓	✓	✓	✓	✓
	Literals, Opcodes	✓	✓	✓	✓	✓
	Conditional Branches	✓	✓	✓	✓	✓
	Uncond. Br/Jmp	✓	✓	✓	✓	✓
	Linked Jumps	✓	✓	✓	✓	✓
	Indirect Jumps	✓	✓	×	✓	✓
	Interrupts	✓	×			
Testing throughput	Tested Instr/Hour	13.8G	200M			
	Generation	High	Simulation Based		Constrained by load time per test	
	Execution	Self Gen.				
	Failure Reproducing	Simul.				
Test Legalization Method		Construction			Templates	

* Note that characteristics not specified in the reference are left blank in the table.

A. Utility Throughput and Comparison

The binary of the Selfie5 utility is loaded onto the target platform, and from that point, autonomously generates random tests, copies their machine code to the designated space in the instruction memory, executes the tests, and checks them for correctness. The length of the generated tests is configurable (4–256 instructions) and randomized to test various sequences. Table I breaks down the average number of CPU cycles for the various stages in the utility flow for a 32-instruction test. While the actual test execution comprises less than 10% of the total operation, since this is applied at-speed on the target platform, the total test throughput is very high, reaching 551 Million test instructions per hour (IPH) on the FPGA prototype running at 40 MHz and 13.8 Billion IPH on the 16 nm SoC at 1 GHz.

Table II presents features of Selfie5 alongside other methods that target the similar objective of high-throughput verification [1], [5], [9], [12]. Selfie5 delivers 69 × higher throughput than the work by Herdt, et al. [1], which is the only other publication to report this figure-of-merit. This high-throughput compensates for the lack of dynamic coverage-guided feedback that exists in other approaches and is the only work that clearly supports verification of random interrupt injection.

B. Use Case Example

To emphasize the capabilities of the Selfie5 approach, we overview a bug that successfully passed through the functional verification process and was exposed by Selfie5 in silicon.

An internally developed RISC-V core [14] was designed and sent for fabrication as part of a 65 nm test chip, prior to the development of Selfie5. To support the dual-issue capability of the core, a register file with five read ports and three write ports was developed with a controlled placement approach [16]. While the silicon samples successfully ran several benchmarks, Selfie5 brought the system to a failing corner case within minutes of run time.

Thanks to the failure reproduction capabilities of Selfie5, the scenario was easily replicated in simulation. The debug exposed an unexpected case, when two consecutive instructions write to the same register. While such a scenario is meaningless, it can be generated by standard compilers (in rare cases) and can occur in manually crafted assembly code. The result of the two instructions should be that the latter value is stored in the destination register. However, the optimized, semi-custom implementation of the register file was designed assuming that the destination registers of the write ports were mutually exclusive. Upon dual-issue execution of the two instructions, the same destination register was simultaneously applied to two write ports, causing the wrong value to be stored. Based on the bug exposure, this scenario was immediately worked around in software and fixed in hardware for subsequent tapeouts.

VII. CONCLUSIONS

This paper presented, Selfie5, an autonomous, self-contained verification approach and utility that provides high-throughput random testing of programmable processors at the simulation, emulation, and post-silicon validation stages. Selfie5 was demonstrated on three fabricated test-chips containing RISC-V cores, providing massive stress testing, which exposed bugs that passed functional verification tests. The Selfie5 utility code is available open source at [17], including RISC-V target-platform modules and the RVSIM ISS.

REFERENCES

[1] V. Herdt *et al.*, "Efficient cross-level testing for processor verification: A RISC-V case-study," in *IEEE FDL*, 2020, pp. 1–7.

[2] ——, "Towards Specification and Testing of RISC-V ISA Compliance," in *DATE'20*, 2020, pp. 995–998.

[3] ——, "Closing the RISC-V compliance gap: Looking from the negative testing side," in *DAC'20*, 2020, pp. 1–6.

[4] S. Ahmadi-Pour *et al.*, "Constrained random verification for RISC-V: Overview, evaluation and discussion," in *MBMV'21*, 2021, pp. 1–8.

[5] D. D. Tran *et al.*, "RISC-V Random Test Generator," in *ACOMP*, 2021.

[6] K. Uday Bhaskar *et al.*, "A universal random test generator for functional verification of microprocessors and SoC," in *IEEE ICVD*, 2005.

[7] P. Mishra *et al.*, "Post-silicon validation in the SoC era: A tutorial introduction," *IEEE Design & Test*, vol. 34, no. 3, pp. 68–92, 2017.

[8] A. Adir *et al.*, "A unified methodology for pre-silicon verification and post-silicon validation," in *DATE'11*, 2011, pp. 1–6.

[9] ——, "Leveraging pre-silicon verification resources for the post-silicon validation of the IBM POWER7 processor," in *DAC'11*, 2011.

[10] ——, "Threadmill: A post-silicon exerciser for multi-threaded processors," in *DAC'11*, 2011, pp. 860–865.

[11] A. Nahir *et al.*, "Optimizing test-generation to the execution platform," in *ASP-DAC*, 2012, pp. 304–309.

[12] M. Chupilko *et al.*, "Test program generator MicroTESK for RISC-V," in *IEEE MTV*, 2018, pp. 6–11.

[13] Y. Luo *et al.*, "MPRC-RISCV-simulator," https://github.com/Oliver-Luo/MPRC-RISCV-simulator, 2023 [Online; accessed 4-Sep-2023].

[14] Y. Kra *et al.*, "HAMSA-DI: A Low-Power Dual-Issue RISC-V Core Targeting Energy-Efficient Embedded Systems," *IEEE TCAS-I*, vol. 71, no. 1, pp. 223–236, 2024.

[15] E. Garzón *et al.*, "A RISC-V-based Research Platform for Rapid Design Cycle," in *ISCAS 2022*, 2022, pp. 2614–2615.

[16] H. Marinberg *et al.*, "Efficient Implementation of Many-Ported Memories by Using Standard-Cell Memory Approach," *IEEE Access*, vol. 11, pp. 94 885–94 897.

[17] Y. Kra *et al.*, "SELFIE5 verification utility with RISC-V adaptation," 2024. [Online]. Available: https://github.com/enics-labs/Selfie5