Automated Generation of Software-Based Self-Test from ATPG Patterns targeting Transition Delay Faults in the decode stage of the RISC-V based microcontrollers

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 July 2026, accepted 10 August 2026, date of publication 12 August 2026, date of current version 24 August 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3723024

RESEARCH ARTICLE

Automated Generation of Software-Based Self-Test From ATPG Patterns Targeting Transition Delay Faults in the Decode Stage of the RISC-V-Based Microcontrollers

NIMA KOLAHIMAHMOUDI, (Student Member, IEEE), FRANCESCO ANGIONE, (Member, IEEE), AND PAOLO BERNARDI, (Senior Member, IEEE)
Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Turin, Italy
Corresponding author: Nima Kolahimahmoudi (nima.kolahimahmoudi@polito.it)

ABSTRACT Software-Based Self-Tests (SBSTs), grouped in Software Test Library, are functional testing approaches used as a flexible solution during manufacturing tests to enhance the quality of outgoing silicon or online testing, as functional safety mechanisms. While SBSTs have proven to be effective for stuck-at fault testing, their development for transition delay fault models (TDFs) remains challenging, requiring skilled test engineers and time-consuming manual efforts. This paper presents a novel methodology for automatically generating SBSTs oriented to test TDF models in the decoder unit of microprocessors by leveraging already generated scan-based test patterns. The proposed approach analyzes scan-based patterns to extract instructions and register file values, processes this data to create code portions that maintain proper timing for TDF excitation and propagation, and handles generated control-flow instructions. The methodology considers pipeline depth to ensure fault visibility in processor registers and includes proper code relocation through linker scripts. The generated SBST can serve as input for further optimization through genetic or ATPG-driven methods to enhance fault detection capabilities or optimize characteristics such as code size and execution time. Experimental validation is conducted on open-source RISC-V CPU cores such as CV32E40P from the Open Hardware group, the Berkeley Out-of-Order Machine (BOOM), and Rocket core from UCBerkley, synthesized using a 45nm technology library. Experimental results are primarily aiming at maximizing the TDF coverage on the decode unit of the CPUs, since a fault in the decode unit may affect the entire behavior of the CPUs. Afterward, the generated SBST is fault graded on the entire datapath logic of all the CPUs. The experimental results demonstrate the effectiveness of transforming scan-based test patterns into correct functional tests while having moderate fault detection capabilities with low manual efforts compared to manually developed SBSTs.

INDEX TERMS Functional testing, SBST, hardware testing, transition delay faults, manufacturing testing, functional test generation, riscv architecture, microprocessor testing, design-for-testability reuse.

I. INTRODUCTION

The Software Test Libraries (STLs) are a common functional testing technique. STLs are self-test procedures that exploit CPU-based instructions to verify the functional behavior of System-on-Chips (SoCs) [1], [2], [3], [4], [5], [6].

An STL comprises a collection of Software-Based Self Tests (SBSTs) designed to test specific portions of logic circuits, primarily CPU submodules, during manufacturing test and mission mode test (online testing) by exploiting

The associate editor coordinating the review of this manuscript and approving it for publication was Poki Chen.

CPU-based instructions [7]. SBSTs are an effective testing solution due to their advantages in terms of area (requiring minimal storage memory), at-speed testing, and software flexibility compared to hardware approaches such as built-in self-tests [8].

Skilled test engineers develop SBSTs manually, which begins by setting up the fault simulation campaigns and proceeds through an iterative and time-consuming process. These manually developed test routines for different modules require a thorough knowledge of the design under test and the Instruction Set Architecture (ISA). Test engineers are responsible for the development of the program for the entire CPU core, either from scratch or by reusing already known SBSTs from the same family of devices [3]. Although the development of SBSTs can be guided by high-level metrics [9], it still remains a labor-intensive and time-consuming process. The reason behind this challenging development is that, with every new CPU design, it is necessary to develop or adapt SBSTs and grade their quality in terms of fault detection capabilities. To address these challenges, different works have attempted to generate SBSTs automatically in the literature. Some of these works exploit Automatic Test Pattern Generators (ATPG) [10] with functionally driven constraints, feedback-based approaches based on evolutionary optimizers for generating routines combined with extensive fault simulation campaigns [11], or randomized-based methods [12].

Although the generation of SBSTs is quite a mature topic in the literature, it has been mainly explored for the stuck-at fault (SAF) model and not for timing-related fault models, such as transition or small delay fault models. The challenges in generating SBSTs for the transition delay fault model (TDF), either manually or automatically, reside in the problem of applying a specific sequence of test vectors to the module under test. This sequence of test vectors should excite the transition, and in the next test vector, propagate the difference to observable points (primary outputs). Functionally speaking, developing an SBST that can apply correctly sequenced specific test vectors is extremely challenging, even with automated methodologies [13]. This difficulty arises from the inherent problem of not being able to control all the values of CPU registers and their evolution to/from different states.

The proposed methodology (available on GitHub [14]) aims to alleviate the manual effort required to develop SBSTs for Transition Delay Faults (TDF) in CPU decode stages, thereby reducing development time in the following ways:

1) Providing an automated framework to develop and grade SBSTs during early design stages.
2) Offering a scalable development solution by repurposing multi-cycle, ATPG-generated, scan-based TDF patterns.

The proposed methodology reuses scan-based patterns, generated for manufacturing scan tests, and scan-chain descriptions, the latter of which is used to identify the position of the required registers to extract from the scan patterns. After the registers of interest (i.e., registers used by the SBST during the test execution) have been identified, their values during the scan pattern application can be extracted. The logic simulation process applies the extracted scan patterns to the device under test and dumps the register values for each pattern application. Processing the sequence of extracted register values provides a sequence of code portions (i.e., for each scan pattern, in each capture cycle, a new value in the user-observable CPU register is present). Execution of the generated code portions applies the generated multi-cycle scan patterns to the user-accessible registers, following the identical timing during the scan test. Thus, the excitation and propagation of TDFs should follow the same sequence as the scan test.

Depending on the depth of the core's pipelines and the number of capture cycles produced by the ATPG, a single scan pattern, or multiple scan patterns can be extracted. The instruction register values are the instructions that can be the user input instruction to the microprocessor. The extracted register values from the logic simulation of the scan patterns are in binary values. These patterns should pass through a sanity checker, in which protected addresses and registers are filtered out, branch offsets are cut off to the maximum allowable by the ISA, and instruction classes are identified. Control-flow instructions (jumps/branches) require special handling; they are processed to generate an additional linker script to locate the generated code portions in the correct memory locations during execution. This process should be without or with minimal modification in the artificial control flow of the binary values produced by the ATPG for scan-based tests. Finally, the test code is embedded in the final SBST structure, including the prologue, epilogue, test setup code, and a custom exception handler, which are ISA-dependent.

The experimental results are carried out on three different open source RISC-V CPU cores, the CV32E40P [15], the BOOM [16], and the Rocket [17], synthesized with a 45 nm open source technology library [18].

The three case studies have a rising complexity in terms of logic gates and number of TDFs, starting from the CV32E40P core with 17,463 gates and 138,513 TDFs, followed by the Rocket core with 256,485 logic gates and 1,006,888 TDFs, up to the BOOM core with 1,800,591 logic gates and 13,302,082 TDFs. The scan patterns needed to reach a fault coverage of around 96% for TDFs in the entire core are, respectively, 1,278 for the CV32E40P core, 4,919 for the Rocket core, and 23,865 for the BOOM core. As the complexity of the microprocessors grows, the number of required scan patterns to reach the same fault coverage grows. This increasing trend results in more scan-based patterns, hence producing a longer SBST program.

The primary objective of the proposed methodology is to automate the generation of SBST programs for testing the decode stage of CPUs. ATPG complexity is reduced by generating scan test patterns specifically for the decode stages

of the selected case studies. For the CV32E40P, there are 2,124 TDFs, requiring 18 scan patterns to achieve 96% fault coverage. The Rocket core contains 42,212 TDFs, requiring 4,919 scan patterns for 96% coverage, while the BOOM core has 18,392 TDFs, requiring 157 scan patterns to reach the same threshold. The high number of patterns required for the Rocket decode stage case study results from the lack of hierarchical separation of its decode stage; the Rocket core RTL utilizes a flattened LUT for instruction decoding within the design. Consequently, the ATPG process cannot target a specific unit and must be configured for the full core, leading to a slight increase in ATPG computation time. This demonstrates the feasibility of the proposed methodology for both flattened and hierarchical designs.

The SBSTs generated by the proposed methodology reaches a fault coverage of TDFs of 69.3 % for the CV32E40P core (on 2,124 TDFs), 44.69 % for the Rocket core (on 42,212 TDFs) and 38.02% for the BOOM core (on 18,392 TDFs). The experimental results on the decode stage of the various case studies have been obtained in about one week of development time (without including the fault simulation campaign or the ATPG computing time, but purely the development time of the SBST).

In order to assess the ripple effect of the low-effort generated SBST by the proposed methodology, fault simulation campaigns are executed on the entire cores with a reduced fault list compared to the number of faults from the ATPG for speeding up the fault simulation campaigns and the limitations of the generated SBST into stimulating, some modules, such as the Control and Status register module, the cache controller, the memory protection unit, the transaction look-aside buffer, etc (for Rocket and BOOM cores). The generated SBST by the proposed methodology reaches a fault coverage of 61.81% for the CV32E40P core (on 138,513 TDFs), 22.57 % for the Rocket core (on 812,494 TDFs), and 4.73 % for the BOOM core (on 7,701,736 TDFs).

Although the fault coverage achieved by the proposed methodology decreases due to the complexity of the core, it is important to highlight the low required manual efforts and reduced development time. The only manual effort required is to identify the specific registers for extraction from the scan patterns, map their hierarchical position within the design, and prepare those necessary input files for the proposed methodology. This step can be mitigated by AI tools or by using utility scripts already included in the proposed methodology to extract scan chain flip-flops. In the latter case, the user merely needs to define the appropriate regular expression based on the target architecture and the registers of interest. For a new ISA, potential effort involves adapting the files that govern SBST generation and instruction function decoding (the available ones are RISC-V based), as well as creating the linker script, which may be compiler-dependent (the available ones are GNU-toolchain based).

The proposed methodology is compared to manually developed SBST in Software Test Libraries (STLs), primarily in terms of development time and achieved fault coverage on the different case studies. Execution time in clock cycles and code size are reported for completeness. Regarding the CV32E40P core, the generated SBST has an execution time of 14,967 clock cycles and a code size of 18.2 kilobytes, and it has been generated in approximately 13 hours compared to weeks of development of the already existing fourteen STL [13]. On the decoder unit, the generated STL shows a fault coverage of 69.3%, which is 11.06% bigger than that of the manually developed. On the full core, the generated STL shows a promising fault coverage of 61.81%. Even though the fault coverage is not the highest amongst the other manually developed STLs, it provides a solid working base to develop the STLs much faster.

As there are no publicly available STLs for the BOOM and the Rocket case studies, the generated SBST and manually developed STLs (adapted from the STLs of the CV32E40P core) are compared. The generated SBSTs for the Rocket and the BOOM core have a comparable fault coverage with manually developed STL, but they have been generated respectively about three times faster for the Rocket core and about two times faster for the BOOM core with minimal manual efforts.

The rest of the paper is structured as follows: The section II, presents the background about Automatic Test Pattern Generator and Software-Based Self Test, Section III shows the proposed methodology, while Section IV presents the experimental results, and Section V draws some conclusions.

II. BACKGROUND

A. AUTOMATIC TEST PATTERN GENERATOR

Scan-based testing is a crucial approach for achieving high fault coverage in large System-on-Chip (SoC) designs. The Automatic Test Pattern Generation (ATPG) is a powerful solution for test pattern generation oriented to test digital circuits. Figure 1 illustrates a sequential circuit, including both combinational and Flip-Flops, along with the flow used for testing manufactured devices.

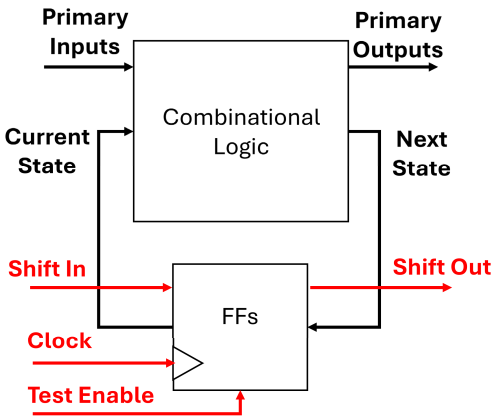

FIGURE 1. Sequential circuit example for scan-based test generation.

The principle of scan-based circuits is to interconnect all the flip-flops in the sequential circuit and exploit them as a

single shift register. This approach reduces the complexity of test generation to that of a purely combinational logic circuit that has a high fault coverage. Thus, scan-based patterns are based on loading precomputed values into the scan chain, executing those values at speed through the combinational logic, and subsequently capturing the computed values. These values are then shifted out for comparison with the golden computed values generated by the ATPG, while new patterns are shifted in.

While ATPG aims to achieve complete fault coverage, many generated patterns can result in overly broad test configurations. These patterns may configure the circuit in states that are not representative of its actual operational mode [19]. This phenomenon, known as over-testing, can lead to increased test time and higher computational resource consumption [20]. Furthermore, in large SoC designs, ATPG may leave certain faults undetected [21], meaning that the scan-based pattern set is not always exhaustive, especially for timing-oriented testing [21].

Traditional ATPG focused primarily on the stuck-at fault model, which addresses static manufacturing defects. However, modern high-performance chips are susceptible to timing-related defects that do not manifest under static conditions. Multi-cycle scan-based testing has emerged as the standard approach to detect these delay-induced failures [22].

In a multi-cycle environment, the ATPG tool utilizes a sequence of clock pulses to evaluate the dynamic behavior of the circuit. This is fundamentally different from single-cycle testing in traditional scan-based testing, which only captures a snapshot of the logic state.

The process involves two critical phases: the launch cycle and the capture cycle. The launch cycle initiates a transition at a source flip-flop, while the capture cycle records the response at a destination flip-flop after a specific time interval. The timing of these pulses must match the functional frequency of the chip to ensure at-speed execution [23].

Multi-cycle testing is primarily designed to address transition delay faults (TDFs) and path-delay faults (PDFs). To generate effective patterns, ATPG tools typically employ one of two methodologies:

1) **Launch-off-Shift (LOS):** The capture transition is launched during the last shift operation of the scan chain.
2) **Launch-off-Capture (LOC):** The capture transition is launched using a functional clock pulse after the scan loading is complete [24].

While LOS offers higher fault coverage and simpler pattern generation, it requires a high-speed scan enable signal.

B. SOFTWARE-BASED SELF-TEST

Software-Based Self-Tests (SBSTs) utilize CPU-based assembly to verify the functional behavior of microprocessor-based systems [1], [2], [3], [4], [25], [26], [27], [28].

SBSTs have been effectively used for manufacturing testing [7] or in online testing as a functional safety mechanism [29]. For example, SBSTs require minimal silicon area as they are stored in memory, unlike lockstep mechanisms [29] for online testing. In manufacturing testing, SBSTs can complement scan-based tests by identifying faults that may have been missed by other testing strategies [30]. The SBST's introduction in manufacturing testing permits a broader, field-like, at-speed test, which is crucial for capturing faulty behavior due to timing-related faults such as TDF or small delay faults [7].

Furthermore, software-based test solutions can be updated to address flaws and to meet safety and execution requirements. The effectiveness of an SBST is evaluated based on its fault coverage, which measures its ability to detect and propagate faults to observable points, and it is graded through fault simulation campaigns.

SBSTs are collected in the Software Test Library (STL), scheduled by the operating system [31]. These STLs assess the correctness of the operation at the end of the test program, utilizing a signature computation [4]. A signature is an arithmetic accumulation in a single register of a custom subset of the overall registers and/or memory words.

Figure 2 depicts a generic structure for SBST. This structure mainly consists of a data section for storing the initial test data of the program and the final golden signature.

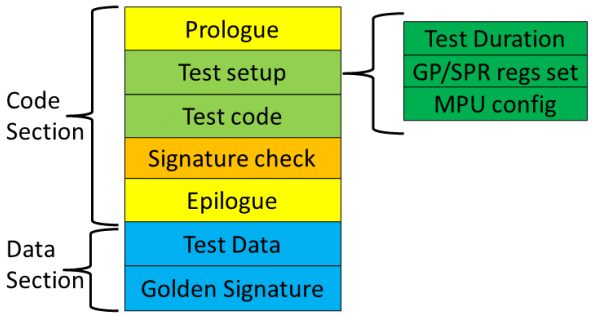

FIGURE 2. Generic structure of an SBST.

The code section is further subdivided into a setup phase to prepare the execution of the test code. This phase includes load instructions targeting general/special purpose registers (GPRs/SPRs), the instruction targeting the module under test, followed by the signature computation. This section is surrounded by a prologue and epilogue (ISA-dependent) code to save the current state of the registers before entering the test routine.

C. STATE-OF-THE-ART FOR SBSTs GENERATION

SBST has become a crucial method for testing microprocessors and embedded CPUs, providing significant benefits such as at-speed testing, non-intrusiveness, and applicability, especially in the field. The automatic generation of SBST programs is particularly vital due to the increasing complexity of modern processors and the necessity to minimize manual development efforts [32].

Modern SBST methodologies can be classified into functional and structural approaches [32]:

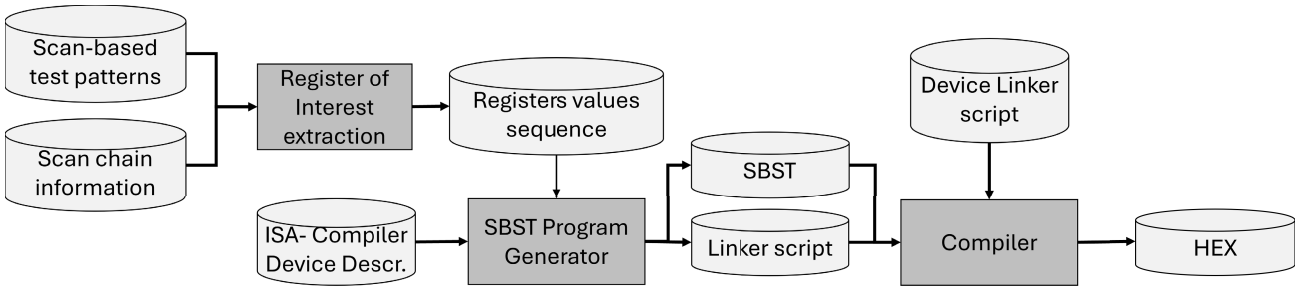

FIGURE 3. Proposed methodology workflow.

- **Functional Approaches:**
  - *Randomizer-Based Methods:* Generate random instruction sequences with architectural constraints (e.g., *FRITS* [12]).
  - *Feedback-Based Methods:* Utilize coverage metrics to guide test generation and employ evolutionary algorithms for optimization [11].
- **Structural Approaches:**
  - *Hierarchical Methods:* Focus on individual processor modules, generating module-specific test patterns.
  - *RTL-Based Methods:* Combine RTL and ISA information to generate deterministic algorithms and ATPG targeting specific fault models.

Recent SBST generation techniques employ hybrid approaches that combine various testing strategies [33]:

- **Deterministic Structural SBST:** Utilizes high-level test development and gate-level ATPG.
- **Verification-Based Self-Test:** Focuses on control-oriented components using functional coverage metrics.
- **Directed Random Test-Program Generation (RTPG):** Supplements other methods with biased random instruction sequences.

State-of-the-art SBST generation frameworks typically include [10]:

- **Validity Checker Module (VCM):** Ensures generated test programs comply with processor constraints and validates instruction sequences for architectural compliance.
- **Automatic Test Pattern Generation (ATPG):** Generates functional test sequences using formal methods, employing bounded model checking (BMC) and SAT-based techniques to optimize fault coverage.
- **Test Program Optimization:** Utilizes evolutionary algorithms for optimizing test sequences, implementing feedback-based coverage improvement.

Prominent companies such as Intel and Sun Microsystems have implemented SBST methodologies for their microprocessors, achieving fault coverage exceeding 92% and significantly reducing test development time [34].

The automatic generation of SBST has advanced considerably, focusing on hybrid methodologies and support for advanced architectures. The proposed methodology advances the state of the art by providing a generation framework that takes as input existing and generated scan-based test patterns used during manufacturing structural tests oriented to the TDF model. It automatically synthesizes the scan-based test patterns into a sequence of assembly instructions, ready for deployment for functional testing oriented to TDFs for reducing the development time of SBST.

III. PROPOSED METHODOLOGY

This work focuses on automatically generating an SBST program for detecting TDFs in the decode stage of microprocessors, starting from multi-cycle scan-based structural test patterns to reduce the manual efforts in developing an SBST in the early development stages. Figure 3 presents the proposed methodology workflow. This workflow is based on two main steps:

A) Reuse of existing ATPG-generated scan-based TDF-oriented multi-cycle test patterns and scan chain information to extract registers of interest values in the scan test patterns. This phase produces a sequence of register values that are used as input to the next phase.

B) Generate SBST program by taking the sequence of register values (in binary format) and the ISA description of binary opcodes and instruction constraints. It generates the SBST program by correctly assembling different code portions, such as ABI-compliant prologues and epilogues, loading the test data in the registers, signature computation macros, and compiler-specific directives, as well as an additional linker script for correctly placing the SBST in memory.

The flow begins by identifying specific registers of interest and extracting their binary values through logic simulation of ATPG patterns. These extracted values, for each scan pattern, represent the state of the CPU during each capture cycle and are transformed into sequences of instructions. The length of this sequence of instructions is strictly dictated by the core's pipeline depth and the number of capture cycles, ensuring that the software execution accurately mimics the timing required to excite and propagate Transition Delay Faults (TDFs).

Once the binary data is extracted, it undergoes a rigorous sanity check to transform raw bits into valid executable

TABLE 1. A simplified view of the registers of interest values file after the logic simulation of the scan patterns.

Chunk	Cycle	Program Counter	Instruction	Reg1	...	Reg31	FP Reg0	...	FP Reg31
0	0	0x10da6911	0xffbffffbf	0x2b4badb	...	0xc01d1f25	0xebb75ecafa4ac9f7	...	0xc724aedb4b49d873
0	1	0x97e554c1	0xc204a38e	0x068eacf1	...	0x726b94b8	0x406e635373ea69f8	...	0xe50e3f5f423eb871
0	2	0x17e47c27	0x358b81C8	0x244e219e	...	0x7824d8dc	0x12b5344bce64e7a7	...	0x0e8d3395a7d3c7e4
1	0	0xc1ae9795	0xa31ea44f	0x509eb82f	...	0xc8a53e4e	0xa59b8d2e4c1f309a	...	0xd8f63ab9072c1e45
Last Chunk	...	...	...	...	...	...	...	...	...

code. This stage filters out protected registers or restricted memory addresses and trims branch offsets to ensure they remain within the limits defined by the ISA. Because the ATPG-generated sequences do not follow natural program logic, control-flow instructions require a specialized linker script; this script relocates code segments to specific memory addresses, allowing the artificial execution flow to persist without crashing the processor.

The final stage involves embedding this generated sequence into a complete SBST. This includes the necessary ISA-specific components such as the prologue and epilogue for context switching, setup code to initialize the test environment, and a custom exception handler to manage any unexpected behavior during the test execution.

In the following subsections, every aspect of the workflow is explored in detail.

A. REGISTERS OF INTEREST EXTRACTION

The registers of interest extraction phase is the preliminary phase responsible for extracting the information for generating the SBST.

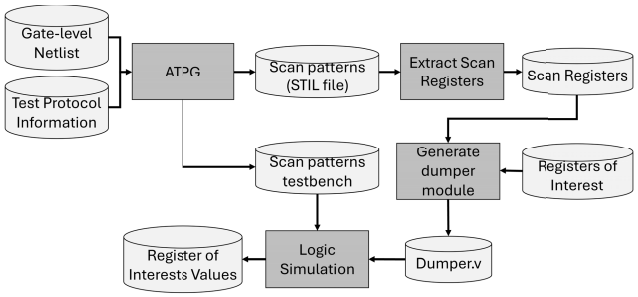

FIGURE 4. The flow for extracting registers of interest.

This phase is depicted in Figure 4, and it takes as input the STIL file [35] produced by the ATPG containing the scan patterns for a specific fault model (TDF in this case). The first step is to extract the scan registers from the STIL file by exploiting its standardized format [35] together with its hierarchical position in the gate-level netlist. Afterward, the generate dumper module step takes the scan registers in the gate-level netlist and user-defined register of interest definition (in the middle right in Figure 4) to generate a verilog module (called *Dumper.v*) capable of dumping the value of user-defined registers for each capture cycle during a scan pattern application (hereinafter named *Chunk*) during a logic simulation with a testbench (generated by the ATPG) capable of applying the scan patterns to the gate-level netlist.

A pseudo code example of the *Dumper.v* is presented in Algorithm 1, in which the values of the user-defined registers are captured (line 5-7) and dumped to a file (line 8).

Algorithm 1 Pseudocode for Verilog Capture Module

Input: Clock, Reset, Scan Inputs, Control Signals
Output: Registers of Interests values (CSV file)
1: **Initialize** Internal variables
2: **Initialize** Create CSV file and print the header
3:
4: **Always** @(posedge capture signal) and !reset **do**
5: **for all** Register R_i in User-defined Registers
6: **Capture** data of R_i in $V[i]$
7: **end for**
8: DumpCSV(V)
9: **end Always block**

Table1 shows a simplified example of the output file during this phase, for each chunk (scan pattern), each capture cycle, the value inside the program counter, instruction registers, general purpose registers, and eventually, 64-bit floating point registers. It is important to mention that registers may change values between cycles of the same scan pattern since new value can be loaded.

In Figure 5, a clarification example for the register extraction phase is presented. As mentioned in the Background section, by exploiting the scan-based registers, precomputed values by the ATPG can be loaded into the registers in a shift-register-like manner. When the shift-in phase is complete, the scan patterns can be applied to the combinational logic through a capture phase in which the logic circuit operates normally, using Launch-on-Capture (LoC) or Launch-on-Shift (LoS) approaches [36] and an increased number of capture cycles to maximize the detection of transition delay faults [8], [36]. Afterward, the results are shifted out for comparison.

Independent of the capture phase strategy (LoC or LoS), the registers are preloaded during the shift-in phase with their precomputed values to be applied to the combinational logic. Therefore, these values are applied during the capture phase in a functional-like manner to the logic and can be used as a starting point for generating functional test programs, such as SBSTs.

Furthermore, the ATPG scan pattern generation for the TDF model can have capture cycles starting from two cycles (mandatory for detecting transitions) up to a reasonable number, usually under twenty, to avoid excessively long test

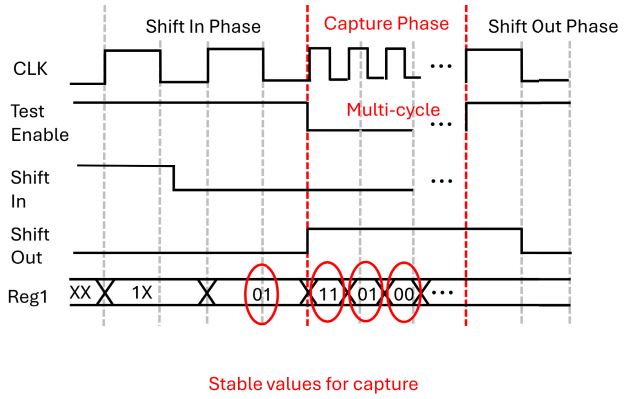

FIGURE 5. Timing diagram of 2-bit scan registers during scan-based structural tests.

times [37]. It is crucial to highlight that by knowing the number of pipeline stages in the CPU, ATPG generation can be constrained to have the exact capture cycles of the pipeline stage in order to excite and propagate, functionally, TDFs to the commit/write-back stage of the pipeline during the capture phase.

The extracted registers of interest values are used as input data for the SBST program generation phase.

B. SBST PROGRAM GENERATION

In order to effectively generate an SBST program, the following phase combines different information, starting from the register of interests sequences of values extracted from the scan-based test patterns. Following the Instruction Set Architecture (ISA) description, the memory map of the device, and compiler-dependent information (such as assembler directives and linker script syntax).

The SBST program generation flow is divided into different passes, as represented in Figure 6:

1) Test setup pass: This pass is responsible for extracting the initial values to be loaded into the user's registers for each unbreakable code portion (i.e., the *Chunk*).
2) Instruction decode and sanitize pass: This phase is responsible for checking and identifying the instruction classes, such as illegal instructions, control-flow instructions (i.e., branches) used for the linker generator pass, or verifying the allowable ranges for load and store instructions based on the device description.
3) Linker generator pass: This pass utilizes the knowledge of control-flow instructions provided by the previous phase to generate a linker script. This script relocates the test setup and code to specific addresses in memory, ensuring that the control-flow instructions generated by scan-based test patterns and their related timing are not disrupted.
4) Code generator pass: This is the final pass of the SBST generation flow. It collects the identified test code and test setup code portions, generating the SBST by sequentially combining the ISA-dependent SBST description (e.g., the definition of the ABI prologue and epilogue) with the code portions.

Algorithm 2 Test setup creation for each Chunk.

Input: Register of Interests values, ISA
Output: List of test setup and instructions for each chunk

```
1:  ISA = loadISA(ISA)
2:  CSVFile = Register of Interests values
3:  Registers = ReadHeader(CSVFile)
4:  Chunks = readCSV(CSVFile)
5:  for C_i in Chunks do
6:    for cycle in C_i.Ncycles do
7:      regs=extractValues(Registers, C_i.cycles[cycle])
8:      if cycle==0 then
9:        C_i.test_setup(ISA.LoadRegFunc,regs['gpr','fp'])
10:     end if
11:     C_i.instrs.append(regs['instruction'])
12:   end for
13: end for
```

The SBST program generator flow follows these aforementioned steps to create the information necessary for generating a ready-to-compile SBST. This resulting SBST is composed of the sequence of timing-unbreakable identified code portions from each scan-based test pattern and capture cycle (referred to as chunks) and the associated linker script (shown as the right outputs in Figure 6).

1) TEST SETUP PASS

In this pass, the flow is responsible for extracting, from the registers of interest values of the scan-based test patterns extracted from the register of interest extraction phase, the values that need to be preloaded into the user-accessible registers before the execution of every *Chunk*, or test code. This phase handles both the loading and setting of various values in the registers. Additionally, it ensures that exceptions are efficiently handled, if needed, during the execution of the test code to prevent any disruptions in the flow of executed instructions or alterations to their associated timing, by setting a custom defined (ISA-dependent) exception handler. This process is ISA-dependent, as it requires knowledge of the specific instructions used to load immediate values into the registers. This process is presented in Algorithm 2.

The Algorithm 2 outlines the systematic transformation of raw dumped scan data from the registers of interest values file into a structured format. The process initiates by loading the ISA definitions and parsing a CSV file containing the registers of interest, where the header is used to map specific registers. The core logic operates through a nested loop structure that iterates through Chunks, i.e., a scan pattern application and its internal cycles (line 5-6).

Within each chunk, the algorithm examines every individual clock cycle to retrieve the binary state of the registers via the extractValues function. A critical conditional check occurs during the initial cycle of each chunk; at this point,

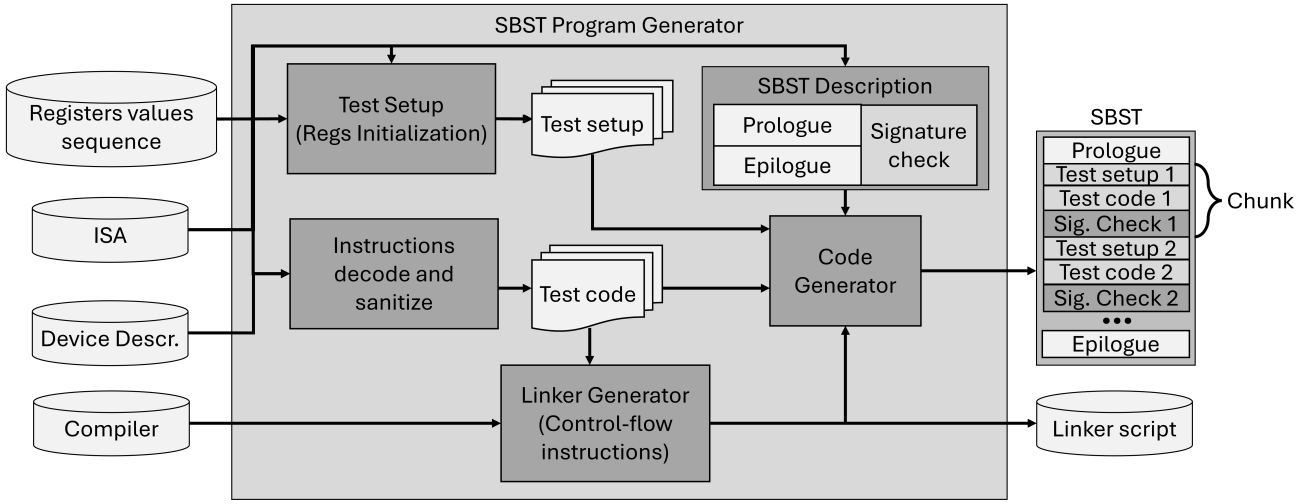

FIGURE 6. SBST generation flow.

the GPR and FP (if present) register values are captured to define the *test_setup*, i.e., the intial values to be loaded before executing the Chunk's instruction using the ISA dependent function for loading immediate values into the registers (line 9). As the loops progress through subsequent cycles, the algorithm identifies the binary values of the instruction in the instruction register and appends them to an internal data structure to be used in the next pass (line 11).

The final result is a chronological list of instructions and setup parameters that allow a software-based test to replicate the hardware conditions found in the original scan patterns.

2) INSTRUCTION DECODE AND SANITIZE PASS

The CPU register containing the currently executed instruction may hold different values across scan-based patterns during the capture cycles, as it is typically unconstrained during the ATPG process. This unconstrained generation produces instruction opcodes of various types, which must be analyzed and, if necessary, adjusted. Algorithm 3 presents the steps needed for instruction decode and sanitize pass.

The algorithm translates raw binary instructions into a structured set of decoded instructions, specifically focusing on identifying and correcting control flow offsets within Chunks.

The primary procedure iterates through every instruction contained within a given Chunk (line 1-2). For each binary instruction, the algorithm utilizes an ISA module to perform a two-step transformation. First, the *ISA.Decode* function identifies the specific category of the instruction—such as arithmetic, memory access, or control flow—and translates the binary bits into a readable format. Following this, the *ISA.Sanitize* function refines the instruction by focusing on its destination or memory reference.

Within the *ISA.Sanitize* function, the algorithm checks if the instruction belongs to the control flow category, which includes operations like jumps or branches. If it does, the logic extracts the raw memory offset and applies an alignment procedure to ensure it matches the required architecture boundaries. To prevent memory errors or out-of-bounds execution, the algorithm compares the processed offset against the total available memory size, capping the value if it exceeds the hardware limits.

Algorithm 3 The algorithm used for instruction decode and sanitize pass.

Input: Chunks, ISA
Output: Decoded instructions, type and offset for each Chunk

```
1:  for C_i in Chunks do
2:    for bin_instr in C_i.getInstructions() do
3:      tmp, type = ISA.Decode(bin_instr)
4:      instr, offset = ISA.Sanitize(tmp, type)
5:      if type == control_flow then
          C_i.offsets.append(offset)
6:      endif
7:      C_i.test_code.append(instr, type)
8:    end for
9:  end for
    ISA.Decode(bin_instr):
10: type = ISA.getType(bin_instr)
11: instr = ISA.DecodeBin(bin_instr)
12: return decoded_instr, type
    ISA.Sanitize(bin_instr, type):
13: if Type == Control Flow Instruction or L/S then
14:   offset = ISA.ExtractOffset(bin_instr)
15:   offset = OffsetAlignment(offset)
16:   if Offset > Memory Size then
17:     Offsett = offsetCapping(offset)
18:   end if
19: end if
20: return decoded_instr, type
```

Once an instruction is fully decoded and validated, the algorithm updates the Chunk test code (line 7). If the instruction involves control flow, its sanitized offset is recorded in a specific list for that Chunk. Finally, the formatted instruction text is appended to the Chunk's list of decoded instructions.

The algorithm involved in this pass categorizes and sanitizes the instructions, derived from the sequence of register values, that need to be executed according to the ISA opcode description. The categorization includes the following types:

- **Control-flow instructions**: These instructions are used during the linker generation phase to correctly relocate the SBST in memory. Additionally, instruction jumps to ATPG-computed addresses must be verified for alignment, offset, and sign. The offsets higher than the allowed jump offset are capped to offsets allowed by the ISA. This minimal change guarantees the closest timing behavior to the patterns applied during the scan test. Moreover, offsets (negative or positive) may target out of the boundary of the memory region or target a memory region in which there is a peripheral as these CPUs are generally used inside a SoC, so reading/writing in that region is impossible. In order to avoid such problems, it is necessary to modify the control-flow instructions to jump into allowed memory arrays. It is crucial to align the offset of the control flow instructions to avoid misaligned instruction fetches that can cause exceptions.
- **Arithmetic Instructions**: These instructions generally do not pose issues during program execution. However, it is essential to preload the required data into the registers to ensure proper execution.
- **Privileged Instructions**: These instructions change the CPU behavior; they can be executed by implementing a custom trap handler that eventually catches exceptions, and at the end of the chunk, their original value must be restored.
- **Load/Store instructions**: These instructions involve source addresses that access data memory for reading or writing values. Since ATPG generates these addresses randomly, they may fall outside the valid memory range, potentially causing exceptions. To handle such cases, a custom exception handler is implemented.
- **Illegal instructions**: Some instructions extracted from the scan-based patterns generated by ATPG may be unimplemented or illegal. The execution of such instructions must be managed by a custom exception handler, which ensures the program resumes execution at the next instruction in the SBST.

For the sake of generality some ISAs may include compressed instructions, which are designed to reduce code size and increase the number of executed instructions within a given memory footprint. These compressed instructions typically use a more compact format, such as 16-bit representations, compared to standard 32-bit instructions.

In the context of ATPG, the process may generate sequences that mix compressed and non-compressed instructions. For example, a compressed instruction might be followed by a non-compressed instruction, or vice versa. This mixing can lead to misaligned instructions in memory, which may cause instruction-related exceptions during execution. Such exceptions arise because the processor may attempt to fetch or decode instructions at incorrect boundaries, violating alignment requirements or accessing unintended memory regions.

To address these challenges, it is essential to ensure proper alignment of instructions during this pass. This may involve verifying the memory layout of generated patterns and implementing mechanisms to handle or correct misaligned instructions, thereby preventing exceptions and ensuring smooth program execution. In this scenario, there are two possible solutions:

- One solution is to break the pattern into two instructions and deal with the specific instructions. The only limitation in this case is that once a pattern becomes two instructions, they are only applied to the first half of the instruction input. So, the pattern is not applied to the instruction input as it is applied during the scan test, as presented in the Figure 7.

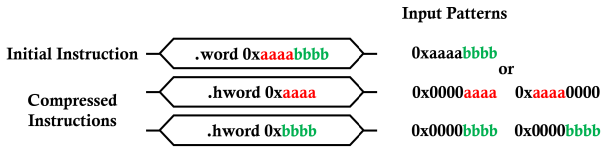

FIGURE 7. Breaking compressed instruction pattern scenario.

- Another solution is to force a minimal number of bits of the patterns to a specific value to avoid having a compressed instruction. In this case, by changing one or two bits, there will be a pattern which does not contain two compressed instructions as presented in Figure 8.

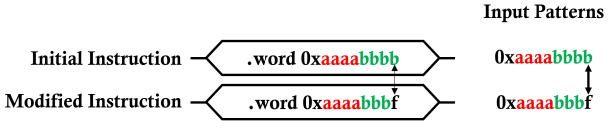

FIGURE 8. Modifying compressed instruction pattern scenario.

Among the proposed solutions for handling the compressed instructions, modifying the pattern by one bit is a better solution, because this solution is conserving the pattern application timing. Moreover, the instruction extraction procedure becomes simpler by handling one instruction instead of two.

3) LINKER GENERATOR PASS

When a control flow instruction is encountered within a sequence of register values, the program execution jumps

Algorithm 4 The algorithm used for the Linker generator Pass.

Input: Chunks with computed offsets, Compiler, ISA
Output: Linker Script Compiler-Dependent

```
CC = SetCompiler(Compiler)
linker_sections_text = [ ]
current_jump_address = jump_sections_start
for all C_i in sort(Chunks, key = offset) do
    for all offset in C_i.offset do
        section_sizes[C_i] = C_i.CalculateCodeSize(ISA)
        section_start = current_jump_address
        section_end = section_start + section_sizes[C_i]
        current_jump_address = section_end
        linker_section = CC.emitLinkerSection
        (section_start, section_end)
        linker_sections.append(linker_section)
    end for
end for
CC.WriteLinkerScript(linker_sections)
```

using an ATPG-computed value that may or may not be valid. This results in an instruction flow for the potential SBST that is significantly more fragmented than that of a traditional SBST. Consequently, every code chunk may be relocated to a different position within the instruction memory.

To prevent the ATPG-generated instruction flow from being disrupted and to maintain the original timing between instructions, the flow is slightly altered after the offsets are sanitized. Furthermore, additional code—specifically the test setup routine—must be included for each chunk. This addition requires a further modification of the jump addresses. To manage this, a jump table is generated to store the starting address and the size of each individual chunk.

The resulting linker script serves as an auxiliary configuration to relocate the necessary chunks. It defines specific memory sections where the code is stored and subsequently relocated during startup to prevent memory overlaps. Finally, the starting addresses associated with each chunk can be rescheduled as needed to optimize the memory layout and ensure no conflicts occur. This solution is presented in Algorithm 4.

It maps code chunks to precise memory locations based on computed offsets. The process begins by selecting a target compiler configuration, which determines the syntax and rules used for the final script generation. Once the environment is set, the algorithm initializes a tracking mechanism for memory addresses, starting at a designated base point for jump sections.

The core logic revolves around a nested iteration through sorted data chunks. To ensure a predictable memory layout, the chunks are first ordered by their specific offsets. For every control flow offset identified within a chunk, the algorithm utilizes the ISA definitions to calculate the exact size the code will occupy in memory. This size is then used to define a contiguous memory block, where the start address is pulled from the current jump pointer and the end address is determined by adding the calculated code size.

As each section is defined, the memory pointer is incremented to the end of the current section to prevent overlaps for the subsequent entry. The compiler-specific module then transforms these calculated address ranges into formal linker script directives. These directives are collected sequentially until all chunks and their associated offsets have been processed. The algorithm concludes by aggregating all generated sections and writing them into a final, unified linker script file ready for use in the compilation toolchain.

4) CODE GENERATOR PASS

The code generator pass collects the identified test setups (registers initialization) and the associated test code, which may include arithmetic instructions, illegal instructions, control flow instructions, load/store instructions, and, optionally, compressed instructions. Based on the SBST description provided as an input file, along with the ISA, it generates the final SBST. Each Chunk (i.e., a scan pattern and its subsequent capture cycles) is converted into an atomic triple *Test Setup*, *Test Code*, and *Signature Check*.

The pseudo code for generating the final SBST is provided in Algorithm 5. By providing as an aditional input the SBST description, as illustrated in Figure 6, which includes details of ISA-dependent features such as the ABI prologue and epilogue for assembly functions, as well as the signature computation and verification. Additionally, it may contain supplementary information related to the SBST, such as comments or specific ISA- and device-related characteristics. These characteristics could include, for example, the setup of a watchdog timer for executing the SBST.

The algorithm outlines the systematic construction of a SBST (SBST) source file, transforming pre-processed data from the scan patterns into a complete, compilation-ready source code. The process begins by loading the SBST description and ISA parameters to ensure the generated code is compatible with the target hardware. It populates the source file with essential structural components, including the informational header, the epilogue for exit procedures, and the initial hardware setup. A critical safety check is performed during this initialization phase: if any illegal instructions are detected within the chunks, the algorithm automatically includes a routine to set up an exception handler, ensuring the system can manage unexpected execution faults without crashing.

The core of the test logic is organized by a scheduler that coordinates the execution of individual Chunk. The algorithm iterates through each chunk, sequentially writing the necessary test setup code, the primary test instructions, and a mechanism for partial signature computation. This partial signature is vital for tracking the intermediate status of the hardware and detecting faults as they occur. After all chunks are processed, a final signature calculation is

Algorithm 5 SBST generation algorithm.

Input: Chunks with decoded instruction, ISA, SBST definition

Output: SBST source code in SBST_File

```
SBST = LoadDescription(SBST, ISA)
SBST.WriteInfoHeader(SBST_File)
SBST.WriteEpilogue(SBST_File)
SBST.WriteInitialization(SBST_File)
if any(illegal instruction in Chunks) then
    SBST.WriteExceptionHandlerSetup(SBST_File)
end if
SBST.WriteChunkScheduler(SBST_File)
for C_i in Chunks do
    SBST.Write(C_i.test_setup,SBST_File)
    SBST.Write(C_i.test_code,SBST_File)
    SBST.WritePartialSignatureComputation(SBST_File)
end for
SBST.WriteFinalSignatureCalculation(SBST_File)
if any(illegal instruction in Chunks) then
    SBST.WriteExceptionHandlerRestore(SBST_File)
end if
SBST.WritePrologue(SBST_File)
SBST.WriteAdditionalData(SBST_File)
SBST.WriteExceptionHandler(SBST_File)
```

appended to provide an aggregate result of the self-test's success or failure.

At the end, the algorithm performs clean-up and auxiliary tasks. If an exception handler was previously established, code is added to restore the original system state and the actual exception handler logic is appended to the end of the file. The process concludes by writing the formal prologue, integrating any additional data required for the test environment, and finalizing the SBST source file for compilation.

IV. EXPERIMENTAL RESULTS

This section presents the experimental setup on which the proposed methodology and the results are carried out. Afterwards, it presents the CPUs that were used to perform our experiments, and it introduces the other SBSTs generated for testing the CPU cores. To conclude, fault coverage experimental results are presented, and these results are compared to the available STLs for testing the CPUs.

A. EXPERIMENTAL SETUP

Figure 9 presents the experimental setup used for validating the proposed methodology.

The Register-transfer level (RTL) design of the case study is synthesized using an open-source cell library, and the scan chain is inserted using *Design Compiler*, from *Synopsys*. The scan-based test patterns are generated using *TestMax*, from *Synopsys*, and they feed the input data for the proposed methodology.

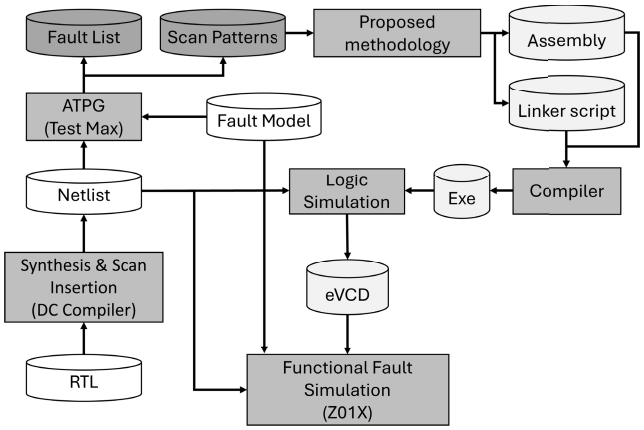

FIGURE 9. Experimental setup.

After extracting the registers of interest from scan-based test patterns, the SBST with its related linker script is generated and compiled for the Logic Simulation using *QuestaSim*, from *Siemens* or *VCS*, from *Synopsys*, depending on the case study, which generates the extended Value Change Dump (eVCD) file. The stimulus for the functional fault simulation with *Z01X*, from *Synopsys*, is based on the eVCD generated by the Logic simulation.

B. CASES OF STUDY

In order to validate the proposed methodology for SBST effortless generation for the decode unit, it is applied to three different CPUs (synthesized using the 45nm Silvaco Open Cell library [18] and a clock frequency of 100 *MHz*):

1) **CV32E40P:** This is an in-order 4-stage RISC-V RV32IMFCX pulp CPU architecture. This CPU is based on RI5CY from PULP-Platform and developed by OpenHW Group [15].
2) **Rocket Core:** Rocket is a 5-stage in-order scalar processor core, also developed at UC Berkeley and SiFive [17].
3) **RISC-V BOOM:** The Berkeley Out-of-Order RISC-V Processor (BOOM) is an open source RV64GC RISC-V core developed in the Chisel hardware construction language [16].

In Table 2 a comprehensive comparative analysis of three RISC-V processor cores is presented focusing on their physical synthesis characteristics and the effectiveness of ATPG for TDFs. The data is structured to show the complexity, highlighting how the scale of a design impacts its the ATPG results.

The synthesis outcomes illustrate a significant range in design scale. The CV32E40P serves as the smallest study case with 17,463 total cells and a higher clock frequency of 100 MHz. In contrast, the BOOM core represents a high-complexity design with over 1.8 million cells, nearly ten times the size of the Rocket core. Across all three designs, the ratio of combinational to sequential logic remains relatively

TABLE 2. Synthesis and ATPG results for the case studies.

Core	Synthesis			ATPG				
	# Total Cells	# Sequential cells	# Combinational cells	# Faults	# Scan Patterns		# Scan Cells	Fault coverage [%]
					Decode	Full Core		
CV32E40P	17,463	2,131	14,942	138,513	18	1,278	2,131	96.31
Rocket	256,485	44,031	209,827	1,006,888	4,919	4,919	42,878	96.87
BOOM	1,800,591	282,939	1,512,893	13,302,082	157	23,865	208,198	96.02

TABLE 3. Experimental results and comparison of manually developed STLs and the SBSTs generated using the proposed methodology.

Core	# Faults		Manually developed STL				Proposed methodology			
	Decode Unit	Full Core	Fault Coverage [%]		Code Size [MB]	Dev. Time [h]	Fault Coverage [%]		Code Size [MB]	Dev. Time [h]
			Decode	Full Core			Decode	Full Core		
CV32E40P	2,124	138,513	58.24	67.97	1.66	560	69.30	61.81	0.018	13
Rocket	42,212	812,494	47.69[1]	22.90	1.06	562	44.69[1]	22.57	1.351	50
BOOM	18,392	7,701,736	27.97	1.73	1.06	562	38.02	4.73	0.043	112.5

[1] Due to the flattened RTL design, the Rocket core functional fault simulation targets the full logic at the top level rather than a standalone decode unit.

consistent, with combinational cells forming the vast majority of the silicon area. For instance, in the BOOM core, there are approximately 1.5 million combinational cells compared to roughly 283,000 sequential elements.

Regarding the ATPG results, the number of identified faults scales proportionally with the total cell count, reaching a peak of over 13.3 million faults for the BOOM core. To address these faults, the ATPG process generated a set of scan patterns that also increases with complexity, ranging from 1,278 patterns for the smallest core to 23,865 for the largest. An interesting observation in the scan cell data shows that for the CV32E40P, every sequential cell is utilized as a scan cell, whereas in the Rocket and BOOM designs, a small percentage of sequential cells are excluded from the scan chains.

Despite the massive disparity in the number of total cells and fault sites, the overall effectiveness of the testing process is remarkably uniform. The fault coverage for all three cores is maintained within a very narrow and high range, specifically between 96.02% and 96.87%. This consistency suggests that the architectural differences between a small embedded core like the CV32E40P and a large out-of-order core like BOOM do not significantly hinder the ability of standard scan-based ATPG tools to achieve high-quality transition delay fault detection but they increase the computing ATPG time to reach those fault coverage results.

The high number of patterns required for the Rocket decode stage case study results from the lack of hierarchical separation of its decode stage (they are the same number for the full core ATPG process); the Rocket core RTL utilizes a flattened LUT for instruction decoding within the design. Consequently, the ATPG process cannot target a specific unit and must be configured for the full core, leading to a slight increase in ATPG computation time. This demonstrates the feasibility of the proposed methodology for both flattened and hierarchical designs.

The proposed methodology is compared in the case of CV32E40P with an instruction self test library (developed for verification purposes of the core) and manually developed STLs, each one of them developed into a period of 20 days by 2 test engineers. The manual STLs were originally developed for the PULPino platform [15] and adapted to the new CV32E40P. The first three STLs are re-adapted for the Rocket and BOOM core for comparison purposes.

TABLE 4. STLs charachterization.

STL	Execution Time [Clock Cycles]	Code Size [kB]
STL1	14,048	191.08
STL2	8,754	92.17
STL3	76,274	810.92
STL4	4,472	84.42
STL5	2,280	27.96
STL6	4,002	31.08
STL7	4,114	47.32
STL8	5,602	80.18
STL9	5,376	97.17
STL10	2,332	47.93
STL11	6,236	35.52
STL12	8,467	95.9
STL13	1,506	32.98
STL14	6,628	115.48
STLs Cumulative	146,560	1,696.18
Instruction Self test Library (ISL)	753	8.6

C. FAULT COVERAGE ON THE DECODE STAGE

The primary objective of the proposed methodology is to automate the generation of SBST routines for the instruction decode stage, a component typically characterized by high manual development overhead. As shown in Table 3, the methodology exhibits high effectiveness in this specific unit across all three architectures. For the CV32E40P, the generated SBST achieves a TDF coverage of 69.30%, representing an 11.06% absolute improvement over existing manually developed STLs. This enhancement suggests that the automated repurposing of ATPG patterns can

identify excitation sequences that are non-intuitive for human engineers.

In the more complex cores, the results remain competitive with manual efforts while requiring significantly less development time. The Rocket core achieved 44.69% coverage, while the BOOM core, despite its sophisticated out-of-order execution logic—reached 38.02% coverage in the decode unit. The lower coverage in the BOOM core compared to the CV32E40P is attributed to the inherent difficulty of controlling internal pipeline states functionally; however, the fact that the automated approach outperformed manual adaptation (27.97%) by nearly 10% highlights the scalability of the proposed framework for complex decoders, especially when there exists parallel decoders in Out-of-Order CPUs, such as the BOOM core.

It is important to highlight the difference in the number of faults between the functional fault simulation in Table 3 and the ATPG in Table 2. This discrepancy stems from the specific units targeted within the core. While the ATPG targets all submodules inside the core, the functional fault simulation excludes cache controllers, branch prediction tables, and virtual memory units from the fault list. These units are outside the scope of the generated SBST and would require more specific testing algorithms.

D. FAULT COVERAGE ON THE CORE

To evaluate the ripple effect of the decode-targeted SBSTs, fault simulation campaigns were extended to the entire CPU core. The primary goals of STLs and SBSTs are to consolidate submodule-specific tests into a comprehensive suite; therefore, analyzing core-wide coverage is essential to determine the effectiveness of the proposed methodology. These results are detailed in Table 3.

For the CV32E40P, the generated SBST reached a full core coverage of 61.81%. While this is slightly lower than the 67.97% achieved by specialized manual STLs, the proposed SBST was generated in only 13 hours compared to the 560 hours for developing 14 STLs. In the case of the Rocket core, which contains approximately 812,494 TDFs (six times the density of the CV32E40P), the methodology reached 22.57% coverage. For the BOOM core, the most complex architecture with 7.7 million TDFs (roughly 54 times the density of the CV32E40P), the coverage is 4.73%. Despite the exponential increase in architectural complexity and TDF count, the SBSTs for these cores were generated in only 50 and 112.5 hours, respectively, demonstrating a highly scalable development-to-coverage ratio.

The inherent gap between the coverage achieved by the functional SBST and the ideal scan-based ATPG results stems from several functional constraints:

- Register Accessibility: Unlike scan-based testing, which shifts values directly into all Flip-Flops (FFs), functional instructions cannot force specific values into internal pipeline registers that lack direct ISA-level accessibility.
- Control Flow Disruptions: Illegal instruction sequences produced by the ATPG can trigger interrupts or exceptions, breaking the exact pattern sequence required to excite and propagate specific TDFs.
- Instruction Constraints: Functional requirements, such as correcting ATPG-generated memory accesses to addressable ranges and handling the specific alignment of compressed instructions, necessarily alter the test patterns from their original scan-based form.

Nevertheless, the proposed methodology serves as an efficient ''first-pass'' generation tool that provides substantial collateral coverage for logic outside the decode unit. By significantly reducing the initial manual effort, this approach allows test engineers to focus their time on developing targeted routines for specific modules—such as the Cache Controller or Memory Protection Unit—thereby drastically accelerating the overall SBST development time.

E. COMPARISON WITH MANUALLY DEVELOPED STLS

To evaluate the efficiency of the proposed methodology, a comprehensive comparison was conducted between the generated SBST programs and the manually developed STL designed oriented to TDF in terms of development time, memory footprint, and achieved fault coverage.

As detailed in Table 3, the proposed SBST for the CV32E40P features a remarkably small code footprint of 0.018 MB, which is significantly lower than the 1.66 MB required for the cumulative manual STL suite. While it has an execution time of 14,967 clock cycles, which is higher than individual manual tests, this is attributed to the overhead of frequent interrupt handler entries triggered by the illegal instruction patterns—a necessary trade-off for the high fault coverage achieved.

For the RISC-V Rocket and BOOM architectures, the increased hardware complexity necessitates a larger number of ATPG patterns, depending on whether the architecture is flattened or not. Consequently, the program sizes increase to 1.35 MB and 0.043 MB, respectively. To our knowledge, these represent the first automated SBSTs generated for these complex architectures. Notably, the development time for these cores (50 hours for Rocket and 112.5 hours for BOOM) represents a massive reduction compared to the 562 hours (including the manual development and porting time to the Rocket/BOOM cores) of intensive engineering required for the manual alternatives, as shown in Table 3. For the BOOM core, the proposed methodology actually outperforms the manual effort in both the decode unit (38.02% vs 27.97%) and the full core (4.73% vs 1.73%). This 10.05% improvement in the decode unit, achieved in roughly one-fifth of the manual development time, confirms that the proposed automated flow is not only faster but can be more effective than manual engineering for high-complexity, out-of-order architectures.

Moreover, in order to provide a full picture of the achieved fault coverage between the proposed methodology and manually developed STLs, table 5 provides a detailed

TABLE 5. Fault coverages for different pipeline stages and cumulative by different SBSTs for the CV32E40P CPU core.

Stage name	Fetch stage		Decode stage		Execute stage		CSR		Load Store Unit		Full Core	
#Faults	14,078		61,737		34,546		22,178		5,974		138,513	
STL	#Det. Faults	FC [%]	#Det. Faults	FC [%]	#Det. Faults	FC [%]	#Det. Faults	FC [%]	#Det. Faults	FC [%]	#Det. Faults	FC [%]
STL1	7,714	54.79	32,585	52.78	22,212	64.30	0	0	2,665	44.61	65,176	47.05
STL2	7,648	54.33	32,146	52.07	27,792	80.45	0	0	2,559	42.84	70,145	50.64
STL3	7,884	56.00	44,634	72.30	30,428	88.08	0	0	2,621	43.87	85,567	61.78
STL4	7,589	53.91	48,018	77.78	20,333	58.86	0	0	2,871	48.06	78,811	56.90
STL5	7,375	52.39	34,051	55.15	29,004	83.96	0	0	2,755	46.12	73,185	52.84
STL6	7,416	52.68	34,148	55.31	29,188	84.49	0	0	2,806	46.97	73,558	53.11
STL7	7,491	53.21	33,252	53.86	25,524	73.88	0	0	2,621	43.87	68,888	49.73
STL8	7,739	54.97	50,173	81.27	28,919	83.71	132	0.60	3,487	58.37	90,450	65.30
STL9	7,589	53.91	36,911	59.79	27,769	80.38	0	0	2,681	44.88	74,950	54.11
STL10	7,463	53.01	34,151	55.32	26,937	77.97	0	0	2,705	45.28	71,256	51.44
STL11	7,579	53.84	34,904	56.54	28,429	82.29	0	0	2,693	45.08	73,605	53.14
STL12	7,570	53.77	34,471	55.84	22,750	65.85	0	0	2,665	44.61	67,456	48.70
STL13	7,443	52.87	31,001	50.21	12,408	35.92	0	0	2,621	43.87	53,473	38.61
STL14	7,594	53.94	32,510	52.66	29,464	85.29	0	0	2,621	43.87	72,189	52.12
STLs Cumulative	**8,110**	**57.61**	**51,649**	**83.66**	**30,743**	**88.99**	**132**	**0.60**	**3,516**	**58.86**	**94,150**	**67.97**
ISL	**7,032**	**49.95**	**18,461**	**29.90**	**10,124**	**29.31**	**0**	**0**	**2,858**	**47.84**	**38,475**	**27.78**
Proposed methodology	**10,009**	**71.10**	**51,845**	**83.98**	**16,998**	**49.20**	**1,957**	**8.82**	**4,812**	**80.55**	**85,621**	**61.81**

breakdown of coverage across the CV32E40P pipeline stages between the proposed methodology and the full suite of STLs.

The Table 5 reports the total number of TDFs, the number of detected TDFs, and the Fault Coverage for the full CPU core and each submodule. The SBST program generated using the proposed methodology has a better fault coverage than most of the other STLs developed by hand, considering the full core coverage. The proposed SBST excels in the Fetch (71.1%), Decode (83.98%), and Load-Store (80.55%) units. A significant highlight is the 8.82% coverage achieved in the Control Status Registers (CSRs), which is nearly 15 times higher than the 0.6% achieved by manual efforts.

V. CONCLUSION

This paper presented an automated methodology for the generation of Software-Based Self-Test (SBST) routines specifically targeting Transition Delay Faults (TDFs) within the decode stages of modern CPUs. By effectively repurposing multi-cycle, ATPG-generated scan patterns and mapping them into functional instruction sequences, the proposed approach significantly alleviates the manual burden and deep microarchitectural expertise traditionally required for STL development. The evaluation across three RISC-V cores (CV32E40P, Rocket, and BOOM) demonstrates that while architectural complexity and the degree of hierarchical design influence the final fault coverage, the methodology consistently provides a high-quality baseline in a fraction of the time required for manual authorship.

The experimental results highlight a significant shift in the development paradigm, where a functional test library for the CV32E40P core was generated in approximately 13 hours, achieving a decode-stage TDF coverage that surpassed manually developed libraries by over 11%. Although the absolute fault coverage on the full-core level decreases for complex out-of-order designs like the BOOM core due to inherent controllability and observability constraints, the methodology remains a scalable and efficient starting point for early SBST development. Ultimately, this work provides a practical solution for the industry's need to reduce time-to-market while maintaining high reliability against timing-related faults in increasingly dense and sophisticated SoC designs.

REFERENCES

[1] P. Bernardi, R. Cantoro, A. Floridia, D. Piumatti, C. Pogonea, A. Ruospo, E. Sanchez, S. De Luca, and A. Sansonetti, "Non-intrusive self-test library for automotive critical applications: Constraints and solutions," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 920–923.

[2] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, pp. 88–99, Jan. 2005.

[3] D. Piumatti, E. Sanchez, P. Bernardi, R. Martorana, and M. A. Pernice, "An efficient strategy for the development of software test libraries for an automotive microcontroller family," *Microelectron. Rel.*, vol. 115, Dec. 2020, Art. no. 113962.

[4] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "On the automatic generation of SBST test programs for in-field test," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2015, pp. 1186–1191.

[5] Cypress. *AN204377-FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library*. Accessed: Sep. 23, 2023. [Online]. Available: http://www.cypress.com/file/249196/download

[6] MicroChip. *16-Bit CPU Self-Test Library User's Guide*. Accessed: Sep. 23, 2023. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf

[7] V. Gangaram, D. Bhan, and J. K. Caldwell, "Functional test selection for high volume manufacturing," in *Proc. 7th Int. Workshop Microprocessor Test Verification (MTV)*, Dec. 2006, pp. 15–19.

[8] J. Rearick, "Too much delay fault coverage is a bad thing," in *Proc. Int. Test Conf.*, 2001, pp. 624–633.

[9] F. Angione, D. Appello, P. Bernardi, A. Calabrese, S. Quer, M. S. Reorda, V. Tancorre, and R. Ugioli, "A toolchain to quantify burn-in stress effectiveness on large automotive system-on-chips," *IEEE Access*, vol. 11, pp. 105655–105676, 2023.

[10] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "A flexible framework for the automatic generation of SBST programs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3055–3066, Oct. 2016.

[11] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, "Automatic test program generation: A case study," *IEEE Design Test Comput.*, vol. 21, no. 2, pp. 102–109, Mar. 2004.
[12] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS—A microprocessor functional BIST method," in *Proc. Int. Test Conf.*, 2002, pp. 590–598.
[13] R. Cantoro, F. Garau, P. Girard, N. Kolahimahmoudi, S. Sartoni, M. S. Reorda, and A. Virazel, "Effective techniques for automatically improving the transition delay fault coverage of self-test libraries," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2022, pp. 1–2.
[14] F. Angione, P. Bernardi, N. Kolahimahmoudi, and CAD & Reliability Group. (2026). *Scan2SBST (Version 1.0.0)*. Accessed: Mar. 11, 2026. [Online]. Available: https://github.com/nimakolahi/SBST
[15] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.
[16] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "*Sonic*BOOM: The 3rd generation Berkeley out-of-order machine," in *Proc. 4th Workshop Comput. Archit. Res.*, May 2020, pp. 1–7.
[17] K. Asanovic et al., "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, Apr. 2016.
[18] Si_2. (2025). *Open Cell and Free PDK Libraries*. Accessed: Jun. 12, 2025. [Online]. Available: https://si2.org/open-cell-and-free-pdk-libraries/
[19] C. Hobeika, C. Thibeault, and J. F. Boland, "Illegal state extraction from register transfer level," in *Proc. 8th IEEE Int. NEWCAS Conf.*, Jun. 2010, pp. 245–248.
[20] I. Pomeranz, "Built-in generation of functional broadside tests using a fixed hardware structure," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 1, pp. 124–132, Jan. 2013.
[21] H. H. Chen, "Beyond structural test, the rising need for system-level test," in *Proc. Int. Symp. VLSI Design, Autom. Test (VLSI-DAT)*, Apr. 2018, pp. 1–4.
[22] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Cham, Switzerland: Springer, 2013.
[23] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI Test Principles and Architectures: Design for Testability* (Systems on Silicon). 2006.
[24] X. Liu, M. S. Hsiao, S. Chakravarty, and P. J. Thadikaran, "Novel ATPG algorithms for transition faults," in *Proc. 7th IEEE Eur. Test Workshop*, 2002, pp. 47–52.
[25] A. Ruospo, D. Piumatti, A. Floridia, and E. Sanchez, "A suitability analysis of software based testing strategies for the on-line testing of artificial neural networks applications in embedded devices," in *Proc. IEEE 27th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jun. 2021, pp. 1–6.
[26] A. Ruospo, G. Gavarini, A. Porsia, M. S. Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale, "Image test libraries for the on-line self-test of functional units in GPUs running CNNs," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2023, pp. 1–6.
[27] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. S. Reorda, M. Grosso, and O. Ballan, "On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors," in *Proc. 14th Int. Workshop Microprocessor Test Verification*, Dec. 2013, pp. 52–57.
[28] K. Christou, M. K. Michael, P. Bernardi, M. Grosso, E. Sanchez, and M. S. Reorda, "A novel SBST generation technique for path-delay faults in microprocessors exploiting gate- and RT-level descriptions," in *Proc. 26th IEEE VLSI Test Symp. (VTS)*, Apr. 2008, pp. 389–394.
[29] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 744–754, Mar. 2016.
[30] P. Bernardi, M. Restifo, M. S. Reorda, D. Appello, C. Bertani, and D. Petrali, "Applicative system level test introduction to increase confidence on screening Quality," in *Proc. 23rd Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2020, pp. 1–6.
[31] F. Angione, P. Bernardi, R. Cantoro, N. Di Gruttola Giardino, D. Piumatti, M. S. Reorda, D. Appello, and V. Tancorre, "On the integration and hardening of software test libraries in real-time operating systems," in *Proc. IEEE 24th Latin Amer. Test Symp. (LATS)*, Mar. 2023, pp. 1–6.
[32] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Design Test Comput.*, vol. 27, no. 3, pp. 4–19, May 2010.
[33] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos, "Hybrid-SBST methodology for efficient testing of processor cores," *IEEE Design Test Comput.*, vol. 25, no. 1, pp. 64–75, 2008.
[34] M. Tripp, S. Picano, and B. Schnarch, "Drive only at speed functional testing; one of the techniques Intel is using to control test costs," in *Proc. IEEE Int. Conf. Test*, 2005, pp. 129–136.
[35] *IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data*, IEEE Standard 1450-1999, Sep. 1999.
[36] A. Krstic, J.-J. Liou, K.-T. Cheng, and L.-C. Wang, "On structural vs. Functional testing for delay faults," in *Proc. 4th Int. Symp. Quality Electron. Design*, 2003, pp. 438–441.
[37] Y. Liu, J. Rajski, S. M. Reddy, J. Solecki, and J. Tyszer, "Staggered ATPG with capture-per-cycle observation test points," in *Proc. IEEE 36th VLSI Test Symp. (VTS)*, Apr. 2018, pp. 1–6.

NIMA KOLAHIMAHMOUDI (Student Member, IEEE) received the master's degree in electronic engineering from the Polytechnic University of Turin, in 2022. He is currently pursuing the Ph.D. degree with the Electronic CAD and Reliability Group, Department of Control and Computer Engineering, under the supervision of Professor Paolo Bernardi. His main research interests include test and reliability of automotive systems-on-chips, including analog and digital, and mixed-signal circuits and embedded memories.

FRANCESCO ANGIONE (Member, IEEE) received the M.Sc. degree in embedded systems track from the Politecnico di Torino, in 2020, and the Ph.D. degree in system-level-test techniques for automotive SoCs. He is currently a Computer Engineer. He is also a Postdoctoral Researcher with the Electronic CAD and Reliability Group, Politecnico di Torino. His main research interests include real-time operating systems, computer architectures, and their dependability.

PAOLO BERNARDI (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science, in 2002 and 2006, respectively. He is currently an Associate Professor with the Politecnico di Torino, where he is involved with the Electronic CAD and Reliability Research Group. His current research interests include system-on-chip tests and reliability, especially in the direction of high-quality automotive devices. He was recently acting as the Topic Chair for European Test Symposium (ETS), the Design and Diagnosis of Electronic Circuits Symposium (DDECS), and the International On-Line Test Symposium (IOLTS). He is the General Chair of the Test Technology Educational Program (TTEP) and the Program Chair of the Automotive Reliability and Test (ART) Workshop held in conjunction with the International Test Conference.