Constraint-Based Automatic SBST Generation for RISC-V Processor Families

Tobias Faller*, Nikolaos I. Deligiannis†, Markus Schwörer*,
Matteo Sonza Reorda†, Bernd Becker*
*University of Freiburg, Department of Computer Science - Freiburg, Germany
†Politecnico di Torino, Department of Control and Computer Engineering (DAUIN) - Turin, Italy

***Abstract*—Software-Based Self-Tests (SBST) allow at-speed, native online-testing of processors by running software programs on the processor core, requiring no Design for Testability (DfT) infrastructure. The creation of such SBST programs often requires time-consuming manual labour that is expensive and requires in-depth knowledge of the processor's architecture to target hard-to-test faults. In contrast, encoding the SBST generation task as a Bounded Model Checking (BMC) problem allows using sophisticated, state-of-the-art BMC solvers to automatically generate an SBST. Constraints for the BMC problem are encoded in a circuit called Validity Checker Module (VCM) and applied during SBST generation.**

In this paper, we focus on presenting a VCM architecture and a constraint set that allows building SBSTs that make minimal assumptions about the firmware, targeting hard-to-test faults in the ALU and register file of multiple scalar, in-order RISC-V processor families. The VCM architecture consists of a processor-specific mapping layer and a generic constraint set connected via a well-defined interface. The generic constraint set enforces the desired SBST behaviour, including controlling the processor's pipeline state, memory accesses, and with that executed instructions, register state, and fault propagations. Using a generic constraint set allows for rapid SBST generation targeting new RISC-V processor families while keeping the generic constraints untouched. Lastly, we evaluate this approach on two RISC-V processor families, namely the DarkRISCV and a proprietary, industrial core showing the portability and strength of the approach, allowing for rapidly targeting new processors.

***Index Terms*—Software-Based Self-Test, Functional ATPG, Automatic SBST, Microprocessor Test, RISC-V**

I. Introduction

Software-Based Self-Test (SBST) programs allow for at-speed, native testing of processors in the field while not requiring any kind of Design for Test (DfT) infrastructure. SBSTs in form of Self Test Libraries (STLs) are in use by many core and semiconductor companies which provide STLs together with their devices for safety-critical applications. Even though SBSTs might bring many benefits, their creation often is time-consuming and costly, requiring manual labour of a skilled developer that knows about the intricacies of the processor's micro-architecture at hand. The manually-written program has to be constructed to make hardware faults visible, requiring reasoning about the micro-architecture's behaviour under fault influence which is neither intuitive nor fast to comprehend. Additionally, the SBST program has to be constructed for its environment, making the SBST creation a complex process that - until now - has to be repeated for every new design.

The introduction of the license-free RISC-V [1] instruction set architecture (ISA) facilitated the creation of a vast amount of new processor cores featuring different micro-architectures, base instruction sets, and extensions posing new challenges for SBST creation. Especially the shortened development cycles and the automated high-level synthesis of whole processor families that provide ISA extensions depending on the targeted use-case requires a new adaptive, automated approach.

Bounded Model Checking (BMC) has been shown to allow semi-automatic generation of SBSTs for processors using manually constrained automatic test pattern generation (ATPG) [2]–[5]. Extending that, [6] introduced an abstraction of the applied constraints by introducing the so-called *Validity Checker Module* (VCM). The VCM allows for specifying constraints as a circuit written in a Hardware Description Language (HDL) and is used during SBST generation to apply constraints to the processor. By using a VCM the development of constraints for complex SBST scenarios is simplified.

However, specifying SBST constraints for whole processor families requires an even higher abstraction level. We present a VCM architecture together with an exemplary constraint set that allows for targeting multiple processor cores in an automated way. Reusability of constraints is provided by having a configurable constraint set that is specified in a processor-agnostic way. These constraints are mapped onto the processor at hand by a well-defined interface that relates signals and behaviour between the processor and the interface. The here presented VCM architecture together with an example constraint set is the main contribution of this paper. To the best of our knowledge, such a structured, generic approach has not been presented before. Note that the architecture allows for generating SBSTs that show a different behaviour compared to the example constraint set presented here.

The example constraint set constructs an SBST run by the firmware during idle times. It consists of only arithmetic instructions and targets hard-to-test faults in the ALU and register file. The SBST generation makes minimal assumptions about the firmware and is designed to be completely independent of the firmware's instruction memory and state. It computes a checksum in an architecture register which is later verified by the firmware. If a mismatch is detected the firmware can take the appropriate measures according to its use case.

The rest of the paper is organized as follows: Section II introduces the VCM concept and its application in SBST generation. Section III presents a VCM architecture with a constraint set that allows the generation of the aforementioned SBST. In Section IV we present the results for two example processor families, and lastly, we draw some conclusions.

II. Background

A. Validity Checker Module (VCM)

The Validity Checker Module (VCM) is a circuit written in a Hardware Description Language (HDL). It encodes a set of functional constraints that are applied to the Circuit Under Test (CUT). The VCM is synthesized into a gate-level representation and is encoded together with the miter circuit of the gate-level CUT (a circuit containing the fault-free and faulty CUT) into a single BMC problem. As shown in Figure 1, the VCM observes the miter circuit via its inputs. The VCM's inputs are connected to the miter circuit's internal and external signals. This allows for observation of the CUT's state and environment under fault-free and faulty conditions at the same time. The VCM's logic validates the CUT's state and behaviour according to its encoded constraints. The VCM has validity outputs that indicate the validation result of each constraint as Boolean value. An output value of 1 indicates that the corresponding functional constraint is held. When generating the BMC problem, all VCM validity outputs are constrained to always have an output value of 1. This enforces the functional constraints of the VCM and with that a valid behaviour of the CUT.

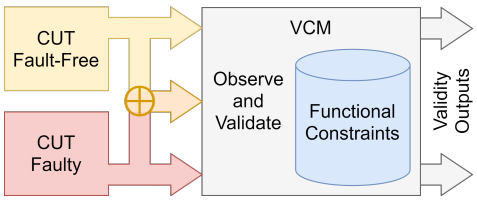

Figure 1. VCM observes miter circuit (left) and validates constraints (right)

The original concept of the VCM as Boolean constraint specification has been significantly extended. Support for detecting `DON'T CARE` values in the miter circuit is added by providing additional `IS DON'T CARE` inputs for the VCM. This is required for processing `DON'T CARE` values in the VCM as it operates on a purely Boolean gate-level. Then, going beyond the core concept of using the VCM as only a constraint specification, additional special inputs and outputs are provided. The special inputs do not correspond to miter circuit signals and are instead used to pass binary encoded parameters into the VCM. These parameters enable and disable constraints in the VCM and allow for configurability without the need for multiple gate-level variants originating from the same HDL code. During the BMC process, these inputs are constrained according to a user-defined configuration. Special so-called result outputs are implemented that allow the VCM to produce binary encoded data on its outputs. This data is later extracted from the solution of the BMC problem. With that, observations of the CUT's state can be pre-processed via the VCM and the applied configuration and later be used for the following steps in the test generation.

III. Approach

The SBST program generation is implemented in our Automated Test Pattern Generation (ATPG) framework named FreiTest that was derived from PHAETON [6]. Even though the core concept of the VCM originates from PHAETON, the framework has been fully rewritten and redesigned for RISC-V SBST generation. This includes the circuit import, VCM handling, the fault simulation, data export and visualization, as well as Conjunctive Normal Form (CNF) generation and the whole ATPG process itself.

The SBST generation is a computationally expensive and complex task. Therefore, we split the problem of SBST generation into smaller steps that are executed as shown in the listing below.

1) Processor and VCM gate-level description import
2) Fault list generation
3) Reset sequence generation
4) Testability check for each fault
5) Instruction sequence generation using BMC
6) Instruction sequence elimination
7) Instruction sequence concatenation to an SBST
8) SBST evaluation and statistics export

During steps 3, 4, 5 and 8 the VCM is used to apply constraints to the processor or to evaluate the processor's behaviour. In Section III-A we will focus on the VCM's architecture that enables porting the steps that are described in the following section to new processor cores and families. Following that, in Section III-B we will focus on the constraint sets that are used during these four VCM-using steps.

For SBST generation the processor and VCM are expected to be present as gate-level description Verilog sources. The VCM and processor sources are previously synthesized from RTL to a gate-level description using the synthesis tool of choice and a target technology library. An adapter library that maps library cells to basic Verilog primitives is required for FreiTest to support new target libraries.

Step 1) The gate-level sources are read into a graph structure in FreiTest. Step 2) A fault list is generated based on the processor's gate-level structure. Step 3) A reset sequence is generated that initializes all architecture registers of the processor to a known state. Step 4) Multiple testability checks are performed for each fault of the fault list. The checks are performed to find faults for which no SBST can be generated, for instance, faults that to be detected require a reset of the processor or a non-functional state in general. Step 5) After the testability check, all fault statuses except untestable faults are reset and the generation of instruction sequences for the SBST starts. An instruction sequence is a short sequence of instructions that is created to make the fault

effect visible. A BMC problem that generates an instruction sequence is constructed and solved. Subsequently, when a solution is found a fault simulation is performed that tests all so far untested faults. If they are detected, they are dropped. Step 6) After all instruction sequences have been generated, a reverse fault simulation is performed. This removes duplicates, unnecessary or dominated instruction sequences that have been generated during the parallel instruction sequence generation. Step 7) The full SBST is constructed by concatenating all instruction sequences into a single sequence of instructions. Step 8) Finally, the fault list is reset to its original state and a fault simulation is performed to evaluate the built SBST program to compute a final, accurate fault coverage.

A. VCM Architecture

To support a fast adaptation of the SBST generation to new processors a VCM architecture that abstracts from processor implementation details was devised. This architecture is shown in Figure 2 where the processor is depicted on the left side. On the right side is the VCM which consists of a mapping layer (orange) that is connected to the processor's essential components (yellow). The mapping layer connects the processor's signals to a well-defined interface that interacts with the generic constraints that are shown in red on the right. The generic constraints encode the valid SBST and processor behaviour, including the valid RISC-V instructions. For decoding and validating the executed RISC-V instructions an embedded decoding module is included. This decoder module's source code is automatically generated from the formal specification used by the MINRES DBT-RISE-RISC-V instruction set simulator [7] and is adjusted to the supported instruction set of the processor core at hand. The instruction set extensions A, M, F, and D are directly available through the formal specification. For custom extensions, the decoding module can be extended by specifying opcodes of supported RISC-V instructions in JSON format.

The mapping layer is responsible for translating the processor's signal lines to the generic interface. By applying constraints to the generic side of the interface the constraints are propagated to the processor at hand. For the different steps of SBST generation, the processor signal mappings listed below are implemented:

- Processor control (reset, halt, run)
- Processor state (resetting, halted, running)
- Pipeline state (bubble, flush, halted)
- Program counter
- Architecture register file (`x1` to `x31`)
- Instruction bus transactions
- Data bus transactions

The instruction and data bus interfaces are both mapped using a bus mapping module written in Verilog. It is exchanged according to the processor bus at hand to reduce adaption efforts. Currently, the bus protocols Advanced High-performance Bus (AHB), Open Bus Interface (OBI), PicoRV32, and DarkRISCV protocols are implemented. Each bus interface is mapped to a set of nine generic signals that monitor transactions. These are a transaction active signal, a transaction commit signal that signals that the transaction was acknowledged, an address signal, a read enable and write enable signal, a read and write byte mask that signals the size of a transaction, and a signal for the read and written value.

For SBST generation the signals listed above are mapped in five variants corresponding to the miter-related inputs available in the VCM. This includes the fault-free and faulty signal, the `DON'T CARE` input for the fault-free and faulty signal, and a difference signal that observes fault propagations for the processor signal. This enables the VCM to implement many different constraints including checking for signals to be well-defined (not `DON'T CARE`), checking for expected signal values and behaviours, and constraining and enforcing fault propagation.

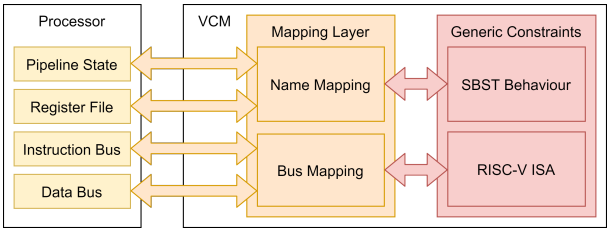

Figure 2. Interaction of processor (left) and VCM (right) with mapping layer in between (middle)

B. Constraint Set

Building on top of the VCM architecture a configurable, generic constraint set is implemented. The constraints are translated through the mapping layer and are applied to the processor. Depending on the SBST generation step different subsets of constraints are enabled to enforce a desired behaviour of the processor. We will now give a more detailed overview of the constraint subsets:

Step 3) The first subset of constraints is used to generate a reset sequence for the processor. The reset processor control signal is forced to be activated for at least one timeframe. Once the reset signal has been deactivated it is disallowed to be enabled again. Only `ADDI` instructions are allowed by enforcing the opcode on instruction bus read transactions. The data bus is constrained to not allow transactions during the reset sequence. As the BMC target, a defined program counter and a fully defined register file are enforced through constraining the `DON'T CARE` signals.

Step 4) The second subset is used for testability checking. This step finds untestable faults that are not relevant in a functional scenario. Table I shows the four stages that apply increasingly complex constraints to the processor. During each stage, a BMC problem is solved for each fault. If for a fault no solution exists, the fault is marked untestable and excluded from further processing. If a timeout occurs or the maximum unrolling depth is reached, the next step is executed and no change to the fault status is done.

Constraints marked with a star are enforced via the VCM while the rest are enforced through FreiTest directly. The first constraint subset is equivalent to a classical full-scan ATPG that only enforces a fault activation and propagation but no functional constraints. This step finds untestable faults caused by signal reconvergences. The next step additionally constraints the processor to be running via the processor state signals and allows only valid RISC-V opcodes on read transactions of the instruction bus. This step finds untestable faults that are only relevant for disallowed situations, e.g. when illegal instructions are executed or the processor resets. The third step additionally enforces the SBST initial state (reset sequence end) to be applied. This step finds faults requiring unreachable states via unbounded model checking features of the BMC solver. These faults are untestable due to unreachable states of the processor like for example unused encodings of pipeline registers. The fourth and last step applies all the previous constraints combined with a directed fault propagation. The fault propagation is enforced by enforcing a difference signal for either the data bus, the instruction bus, or the register file.

Table I
CONSTRAINTS FOR TESTABILITY CHECK

Constraint	1	2	3	4
Initial state	-	-	X	X
Processor running *	-	X	X	X
RISC-V instructions *	-	X	X	X
Fault activation and propagation	X	X	X	X
Directed fault propagation *	-	-	-	X

[1] Combinational full-scan [2] Combinational partial SBST
[3] Sequential partial SBST [4] Sequential SBST

Step 5) The most sophisticated constraint subset is used for the instruction sequence generation since each instruction sequence has to be constructed in a way that it tests its targeted fault but also allows for concatenation to build the final SBST program. The here presented example subset is meant to be run in between firmware idle times to check for degradation and makes minimal assumptions regarding the firmware including the instruction memory, data memory, and the register file.

The final SBST program in Figure 3 is built from multiple instruction sequences. It is executed by the firmware which first saves the register file to the data memory, then jumps to the SBST program. The SBST program computes a checksum in the architecture register `x1` and then jumps back to the firmware which does a state restore excluding the checksum register `x1`. The checksum is verified by the firmware and appropriate action is taken if the verification fails.

Figure 4 shows the structure of an instruction sequence generation. The initial state is set to the reset sequence end state to start with a valid pipeline state. Then, a scramble sequence is generated that permutates register `x1` to reduce the likelihood of fault effect cancellations. Then, arbitrary instructions follow for fault activation and propagation. Finally, the fault effect is propagated to the `x1` register to update the checksum.

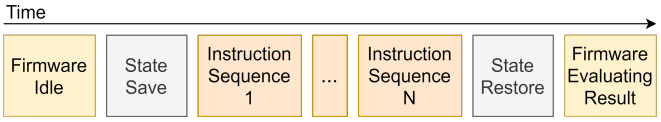

Figure 3. SBST execution from left to right: Firmware starts SBST, firmware context is saved, instruction sequences are run, firmware context is restored, firmware evaluates SBST result

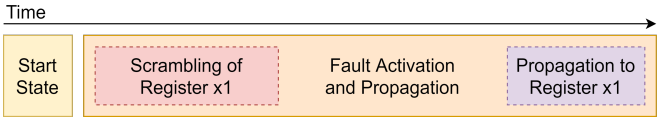

Figure 4. Instruction sequence BMC problem with initial state (yellow), scrambling (red) and final propagation to the register `x1` (purple)

The scramble sequence enforces pre-defined instructions for four time frames rotating register `x1` by one bit and then clearing the temporarily used register `x2`. This results in the fault propagation spreading evenly to all bits of the register and reduces fault effect cancellations. The instruction sequence is enforced by directly constraining the first four instruction fetch transactions on the instruction bus.

A fault propagation to register `x1` is enforced via the difference signals for register `x1` in the BMC target. To be independent of previous checksum values only a limited set of interactions with register `x1` are allowed. This requires constraining instruction fetch transactions that interact with register `x1`. First, only `XOR` and `XORI` instructions are allowed to interact with register `x1` while the destination and exactly one source register is register `x1`, ensuring that the register is updated like a checksum.

Further constraints are applied to ensure a functional SBST program and the independence of instruction sequences. This includes enforcing the processor to be running without interrupts. All transactions on the data bus are forbidden by constraining data transactions. A read or write on the data bus would either make the SBST dependent on or change the firmware state which is disallowed. Instruction fetches are constrained to allow only valid RISC-V opcodes. The program counter is constrained to increase linearly creating a single, linear instruction stream. The SBST program is later extracted by monitoring instruction fetches. Further, jump and branch instructions, as well as instructions that are dependent on the value of the program counter, e.g. `AUIPC` are forbidden making the SBST's memory location irrelevant. Registers have to be initialized before they are used to further aid starting state independence. This is enforced by keeping a list of initialized architecture registers. Uninitialized registers are forbidden to be used as a source operand. Once a register is written to by an executed instruction, the register is marked as initialized and can be used as a source operand for following instructions.

With that multiple measures are made for instruction sequence independence from the program memory position, the firmware state, and the start state of the processor. The constraint set is summarized in the following:

- Processor is running without interrupts
- Data bus is neither read nor written
- Program counter increases linearly
- Only supported RISC-V instructions
- No control flow instructions
- Registers are initialized before being read
- Scramble sequence applied to register x1
- Only XOR and XORI instructions using register x1
- Test result is propagated to x1

Steps 7) and 8) After all instruction sequences are generated they are concatenated to a single SBST program and evaluated via a fault simulation. During fault simulation, the VCM is monitoring the processor and its environment and extracts instructions and fault propagations to the register `x1` and the program counter. The results are obtained from the VCM's result outputs and evaluated for fault coverage. A difference in register `x1` at the SBST end marks the fault as detected.

IV. Evaluation

The SBST generation has been evaluated for two processor families with four configurations in total. The DarkRISCV (3-stage pipeline) and a proprietary, industrial core (5-stage pipeline) have been chosen to show the effectiveness but also the limitations of the presented constraint set and were synthesized for the Nangate 45nm PDK [8]. The BMC depth has been set to 15 timeframes and the timeout to 5 minutes. All experiments have been conducted using an AMD Threadripper 3970X system (32 cores, 64 threads) with 256 GB of RAM.

As a benchmark, a stuck-at fault model was used. The integration of advanced fault models (delay / cell-aware faults) is not completed yet but will be evaluated in future publications.

The need for in-field testing and reaching a minimum fault coverage is mandated by safety standards, e.g. ISO 26262. Therefore, the fault coverage FC and test coverage TC are evaluated for the built SBST programs. These metrics are defined as follows:

$$\mathrm{FC} = N_{\text{testable}} / N_{\text{faults}} \tag{1}$$

$$\mathrm{TC} = N_{\text{testable}} / (N_{\text{faults}} - N_{\text{untestable}}) \tag{2}$$

Table II contains the resulting fault and SBST statistics, and test generation times. It can be seen that fault coverages of roughly $80\,\%$ are achieved for the DarkRISCV processor with a program size of $16\,\mathrm{kB}$ to $26\,\mathrm{kB}$. Even though the SBST program has been constructed to target mainly units that can be tested with arithmetic instructions it can be seen from the test coverage that only roughly $20\,\%$ of the processor remains untested.

However, the evaluation of the proprietary core paints a different picture. Here, only the ALU and register file show a fault coverage over $93\,\%$ while the test coverage surpasses $99.5\,\%$. Other units require an extended constraint set to test. Through the strict constraint set multiple components are not testable, e.g. the exception unit in the instruction fetch (IF) stage, the decoding, bypassing, and hazard detection logic in the instruction decode (ID) stage, the CSRs in the execute (EX) stage and the memory (MEM) stage as no data transactions are allowed.

The SBST generation ranges from $16\,\mathrm{h}$ to $100\,\mathrm{h}$. However, $15\,\%$ to $50\,\%$ (not shown in table) of the time was spent on fault simulation of the final SBST program in FreiTest and can be optimized. Additionally, by moving the scramble sequence out of the BMC problem and prepending it manually the runtime of the BMC could be significantly reduced. This shows that the overall generation runtime has the potential to be optimized.

Comparing the program sizes with existing STLs [9] shows that the generated SBST programs require optimizations to reach the compactness of company-provided STLs ($46\,\mathrm{kB}$ our approach vs $5.8\,\mathrm{kB}$ company-provided). However, regarding that each instruction sequence currently only uses 4-byte instructions and contains a scrambling sequence that might not be required in most cases shows the room for optimizations.

The SBST programs have been additionally evaluated with Z01X by Synopsys and a custom testbench containing an accurate memory model. This allows simulating advanced fault propagations, e.g. propagation chains from the program counter to the instruction memory to the register file. A custom strobing module evaluates the behaviour of the SBST program and evaluates the contents of register `x1` after the SBST program has reached its end. The faults in Table III are either classified as *Detected* if the content of the register is fully known at the program end and it differs from the golden value, or *Maybe Detected* if the register is dependent on an unknown state like unspecified instruction memory regions or uninitialized (CSR) registers, or *Undetected* if no difference from the golden signature emerges. Custom fault statuses were used to signal exceptions during the SBST execution under fault influence. This includes cases where the end of the SBST is not reached (*End Not Reached*), the firmware state is modified or the fault detection is dependent on the firmware state (*Data Bus Used*), or the processor raises an exception or traps (*Exception Occured*). Structurally untestable faults are shown in the *Untestable* column. All remaining behaviours that are not classified are put into the *Unknown* category.

The evaluation using Z01X shows that the detected faults for the DarkRISCV processors are roughly $5\,\%$ below the fault coverage of our tool FreiTest. However, roughly $6\,\%$ of faults have shown to be dependent on uncontrolled factors like the firmware state and could potentially be detected. No exceptions occur since the processor has no exception handling. Two to four percent of faults do not fall into any described behaviour (*Unknown*).

The proprietary, industrial core shows that a $3.3\,\%$ of the faults create an exception. This reduces the percentage of detected faults. However, the number of undetected faults for the ALU and register file show that almost all detectable faults are detected by the built SBST program, validating the assumption of the SBST being able to make hard-to-detect faults visible and propagate them to the checksum register. To detect faults in other modules (exception unit, CSRs, IF,

Table II
FREITEST RESULTS FOR RISC-V PROCESSOR CORES

Processor and ISA		Fault Coverage	Test Coverage	Generated Sequences	Program Instructions	Generation Time	Testable Faults	Untestable Faults	Aborted Faults	Solver Timeouts
DarkRISCV	**RV32E**	75.85 %	79.10 %	587	4,112	16.78 h	17,301	933	4,576	2
	RV32I	82.42 %	84.96 %	934	6,526	46.80 h	28,056	1,015	4,968	3
	RV32I_Zicsr	79.18 %	82.17 %	811	5,645	54.18 h	28,222	1,299	6,124	7
Proprietary	IF Stage	6.54 %	14.01 %	13	76	15.59 h	749	1,688	9,018	6,242
RV32I	ID Stage	40.29 %	45.26 %	58	479	4.08 h	1,546	421	1,870	1,417
_Xunknown	└Register File	93.40 %	99.67 %	730	7,196	34.19 h	16,536	1,112	56	3
	EX Stage	11.66 %	13.65 %	67	514	29.82 h	2,060	2,575	13,027	10,260
	└ALU	99.70 %	99.89 %	428	3,087	5.36 h	7,848	15	9	3
	MEM Stage	12.88 %	18.94 %	17	94	2.45 h	303	753	1,297	985
	WB Stage	25.63 %	26.67 %	13	59	0.84 h	132	20	363	261
	Miscellaneous	4.27 %	5.21 %	15	84	4.06 h	135	569	2,455	2,029
	Sum	45.40 %	51.06 %	1,341	11,589	96.39 h	29,309	7,153	28,095	21,200

Table III
Z01X RESULTS FOR RISC-V PROCESSOR CORES

Processor and ISA		Detected	Maybe Detected	Undetected	End Not Reached	Data Bus Used	Exception Occured	Unknown	Untestable
DarkRISCV	**RV32E**	69.24 %	6.32 %	12.74 %	3.46 %	0.05 %	0.00 %	3.39 %	4.80 %
	RV32I	75.70 %	6.62 %	8.47 %	2.26 %	0.03 %	0.00 %	2.16 %	4.76 %
	RV32I_Zicsr	71.23 %	6.53 %	11.39 %	2.35 %	0.03 %	0.00 %	3.63 %	4.82 %
Proprietary	Instruction Fetch Stage	3.43 %	0.12 %	78.16 %	1.46 %	0.01 %	7.22 %	3.40 %	6.20 %
RV32I	Instruction Decode Stage	40.57 %	0.05 %	70.67 %	0.91 %	0.09 %	7.44 %	7.60 %	4.30 %
_Xunknown	└Register File	91.23 %	0.00 %	0.12 %	0.00 %	0.00 %	0.08 %	3.33 %	5.24 %
	Execute Stage	11.69 %	0.00 %	75.24 %	0.24 %	0.00 %	1.13 %	6.15 %	5.55 %
	└Arithmetic Logic Unit	93.75 %	0.00 %	0.03 %	0.00 %	0.00 %	5.89 %	0.20 %	0.12 %
	Memory Stage	14.42 %	0.00 %	76.57 %	1.05 %	0.00 %	0.12 %	2.20 %	5.64 %
	Writeback Stage	23.88 %	0.10 %	73.69 %	0.68 %	0.00 %	0.05 %	1.31 %	0.29 %
	Miscellaneous	4.25 %	0.14 %	83.03 %	0.37 %	0.03 %	7.65 %	0.70 %	3.85 %
	Sum	38.51 %	0.04 %	48.86 %	0.51 %	0.01 %	3.34 %	3.90 %	4.83 %

ID, and MEM stages) however, the constraint set has to be extended.

V. CONCLUSION

SBSTs allow at-speed, native online-testing of processors without requiring DfT infrastructure. Creating SBST programs is a complex process that - until now - has to be repeated for every new processor design. In this paper, we presented a VCM architecture and a constraint set that allows building SBSTs in particular targeting the ALU and register file of multiple scalar, in-order RISC-V processor families. We evaluated the VCM via BMC-based SBST generation for multiple processor cores. The results were analyzed for fault coverage, runtimes, and size and showed that high functional fault coverages are achievable while the program size and generation time shows potential for future optimizations.

ACKNOWLEDGMENT

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0132 and by the Italian ICSC National Research Centre for High Performance Computing, Big Data and Quantum Computing within the NextGenerationEU program.

REFERENCES

[1] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213*, 2019.

[2] Y. Zhang *et al.*, "Automatic test program generation for out-of-order superscalar processors," in *2012 IEEE 21st Asian Test Symposium*.

[3] R. Cantoro *et al.*, "Effective techniques for automatically improving the transition delay fault coverage of self-test libraries," in *2022 IEEE European Test Symposium (ETS)*, 2022.

[4] A. Ruospo *et al.*, "On-line testing for autonomous systems driven by risc-v processor design verification," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2019.

[5] P. Bernardi *et al.*, "Software-based self-test techniques of computational modules in dual issue embedded processors," in *2015 20th IEEE European Test Symposium (ETS)*, 2015.

[6] A. Riefert *et al.*, "A flexible framework for the automatic generation of sbst programs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, 2016.

[7] *Minres dbt-rise-riscv*, https://github.com/Minres/DBT-RISE-RISCV.

[8] Silvaco, *Open-cell 45nm freepdk*, https://si2.org/open-cell-library/.

[9] A. Ruospo *et al.*, "A suitability analysis of software based testing strategies for the on-line testing of artificial neural networks applications in embedded devices," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021.