Delft University of Technology
Faculty of Aerospace Engineering
Faculty of Electrical Engineering, Mathematics & Computer Science

ARM Limited, Cambridge

Konstantinos Karavelas

CPU Engineering: System Test Libraries team

Prototyping an efficient and cost-effective method to detect and mitigate random faults in Commercial-Off-The-Shelf processors

Detailed Design Report
January – October 2020

CONFIDENTIAL

TU Delft
Mekelweg 5
2628 CD Delft, The Netherlands

TU Delft supervisors
Alessandra Menicucci
a.menicucci@tudelft.nl
Arjan van Genderen
a.j.vangenderen@tudelft.nl

ARM Limited Cambridge
110 Fulbourn Road
Cambridge, UK

ARM Supervisors
Mukesh Lahori
mukesh.lahori@arm.com
Ivan Di Prima
ivan.diprima@arm.com

Preface

This report is the product of the first 8-months of work at ARM Limited in Cambridge, for the purpose of completing a MSc. thesis in Computer Engineering and Aerospace Engineering. The thesis was scheduled to last from January 2020 to October 2020, as part of the mandatory ET4300 Master thesis course at TU Delft. The report presents a summary of all the design activities performed during the course of the graduation project, including the challenges, the engineering choices and the achieved outcome. Its purpose is to document the whole process and provide additional information missing from the Main Report, given the confidential agreement between TU Delft and ARM Limited.

Contents

List of Figures

List of Tables

Glossary

AI Artificial Intelligence. 4

AMBA Advanced Microcontroller Bus Architecture introduced by ARM. 4

API Application Programming Interface. 9

AXI Advanced eXtensible Interface part of AMBA. 4, 6–9, 11, 12, 14, 21–23

BIU Bus Interface Unit. 4–7, 9, 11, 12, 14, 20–23, 25

COTS Commercial Off-The-Shelf components. 2, 4

CPU Central Processing Unit. 3, 5, 15, 17, 25

CubeSat Miniaturized satellite for space research that is made up of multiples of 10 cm × 10 cm × 10 cm cubic units. 2

DI Data Integrity checks. 15–17, 23

DCCIMVAC Data Cache line Clean and Invalidate by Address system register. 11

DCU Data Cache Unit. 4–8, 11–13

DF Detected finish/stop. 20, 21

DD Dropped Detected. 20, 21

ECC Error Correction Code(s) used to control errors in data channels. 4, 25

FM Simplified C-based model of processor. 17, 18, 24, 29

FPGA Field-Programmable Gate Array. 2

FPU Floating-Point Unit. 4

FSM Finite State Machine. 7, 12, 13

ICU Instruction Cache Unit. 4–7, 11

IEC 61508 International functional safety standard. 20

IoT Internet of Things. 4

ISO 26262 Automotive functional safety standard. 20

LEO Low Earth Orbit. 2

LFB Linefill Buffer. 7–9, 11–13, 21–23

LSU Load/Store Unit. 5–8, 11

ML Machine Learning. 4

NoC Network-on-Chip offering point to point communication in a modern SoC. 6

ND Not Detected. 20, 21

NPU Neural Processing Unit. 4

Questa Questa Advanced Simulator from Mentor. 14

RAR Read-After-Read hazard. 7

RAW Read-After-Write hazard. 7, 12

RO Read-Only register. 16

RTL Register-Transfer Level. 14, 17, 18, 24, 28, 31

RTOS Real-Time Operating System. 15, 17

RW Read/Write register. 16

SEE Single Event Effect. 3

STB Store Buffer. 5, 6, 8, 11, 12

STL System Test Libraries. 2–7, 9–15, 17, 19, 20, 22, 23, 25, 31

TCM Tightly Coupled Memory. 4

THREADX THREADX RTOS Real-Time Operating System. 17

TSMC Taiwan Semiconductor Manufacturing Company. 8

VLSI Very Large Scale Integration. 20

WC Write-Clear register. 16

WO Write-Only register. 16

ZOIX ZOIX Fault Simulator software from Synopsys. 20, 21

1 Background information

This Section contains introductory information on the scope of the thesis project, as well as some information on the purpose of this report. Section 1.1 elaborates on the purpose of the Detailed Design Report and provides some background information on the project, whereas Section 1.2 reinstates the thesis Objective for the sake of completeness. Finally, Section 1.3 gives a brief outline of the current report.

1.1 Introduction to the thesis project

This confidential report acts as a supplement to the main thesis report and provides additional information on the design of the STL routines, the Verification Software and the regressions framework. The goal is to document all intermediate steps taken during the design process, as well as include information that is considered confidential or trade-secret and could not be presented in the Main Report.

The goal of the thesis project is to research and design an efficient and cost-effective method to improve random error detection and mitigation of COTS microprocessors, with increased compatibility across devices. The aforementioned goal was formulated after a Literature Study was performed on the current status of mitigation methods in space embedded systems. The conclusion of the review was that the majority of existing methods rely on hardware techniques which are applicable during the design of a specific microprocessor model, with limited to no compatibility across other models. In addition, it was observed that several COTS processors used in CubeSat missions implement one or two mitigation mechanisms, exhibiting poor radiation performance compared to their radiation-hardened counterparts [2]. Several other methods proposed in academia for usage in radiation-hardened or industrial FPGAs [3], can indeed bridge the gap between the two extremes, however they are mostly tailored to a specific processor model and they may require exhaustive validation and verification, a task which is not trivial.

The rise of the CubeSat community and the New Space age of private companies launching spacecraft and other payloads in Low Earth Orbit (LEO) and beyond, meant that a wider academic, industrial and enthusiast community was building low-cost, low-mass and small-volume spacecrafts made out of Commercial Off-The-Shelf components (COTS). Given the increased costs and lead times of radiation-hardened processors, several Space Agencies started experimenting with COTS components, in an effort to take advantage of the superior computational performance, reduced costs and power consumption [4]. The goal of such effort was to determine the most reliable components that could be used in one or more subsystems, reducing the number of radiation-hardened components used, but without compromising the success of the mission. This highlights the importance of having effective, flexible and low-cost solutions for error mitigation in commercial processors, in an effort to bridge the radiation performance gap towards the more expensive options.

1.2 Thesis objective

By taking the aforementioned into account, it was decided to initiate a MSc. thesis project with the following objective:

"To research and prototype an efficient and cost-effective method to detect and mitigate random faults in Commercial-Off-The-Shelf ARM microprocessors, with increased compatibility across processor models."

To that end, the thesis was carried out at ARM, as part of the STL team responsible for developing software tests that are able to detect random errors. They have been proposed for usage in safety critical embedded systems, however they could be potentially used in space applications for detecting errors caused by SEEs. The main objective can be broken down into several sub-objectives, which have also been included here for the sake of completeness:

- Familiarize with the ARM architecture and the current infrastructure for STL development.
- Select a reference CPU model and a target submodule as a proof of concept. Start designing tests with portability in mind.
- Tune performance, to match coverage expectations and iterate as necessary.
- Port all tests to a different processor, within the same architecture extension and verify desired behaviour and coverage.
- Develop a verification flow to prove that each STL routine achieves expected coverage across multiple hardware configurations, without corrupting the system's initial state.
- Provide a unified tool-flow, enabling faster deployment and turnaround time.

Given the wide range of activities needed to satisfy all sub-objectives, it is necessary that one or more tasks are done in parallel. Therefore, even though the list of objectives offers an intuitive methodology with respect to the steps that need to be taken, there have been some alterations in the order and depth, according to intermediate results and scheduling.

1.3 Structure outline

The Detailed Design Report has the following structure. Section 2 will elaborate on the selected platform that will be used to develop and test the aforementioned proof-of-concept, whereas Sections 3 and 4 will present the core design work performed during this 8-month long thesis. The outcome of the work will be presented in Section 5, elaborating on any divergences from the expected results. Finally, Section 6 will elaborate on the conclusions and recommendations applicable to the Detailed Design Report only, since the main conclusions and future work recommendations will be presented in Chapter 7 and 8 of the Main Report.

2 Platform of Choice

This Section will include the technical specifications on the platform used as a testbench for the prototype software flow. The main modules of the selected processor will be highlighted and their design specifications explained. Section 2.1 presents the processor of choice in the context of this thesis, whereas Section 2.2 focuses on the selected module. Finally, Section 2.3 elaborates on the details of the chosen processor Revision and implementation technology.

2.1 Cortex-M55 processor

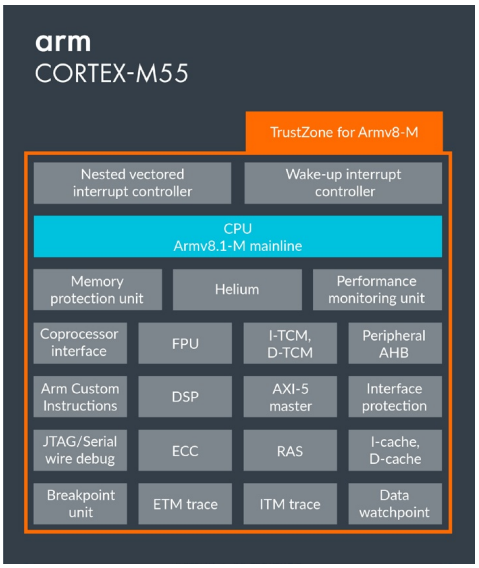
The goal of this project is to prototype efficient STL routines in order to enhance the radiation performance of COTS processors. One of the main goals of this work is to provide a flexible and portable solution, that could have an impact on a wide range of end devices. To that end, a state-of-the-art microprocessor design was used, targeting the next generation of embedded devices. The Cortex-M55 is one of the first processors of its kind, implementing the ARMv8-M architecture with the goal of bringing AI endpoint capabilities to the next generation of IoT devices and applications. This opens up opportunities for reusing the STL routines developed for this thesis, to future M-class processors, hence satisfying the portability aspects of this work. An overview of the main specifications is visible in Table 1.

Specifications	
Architecture	ARMv8-M 32-bit
Pipeline	4-stage
Floating-Point Unit (FPU)	✓
Instruction cache (ICU)	up to 64KB with ECC
Data cache (DCU)	up to 64KB with ECC
Bus Interface (BIU)	AMBA AXI 5 64-bit master

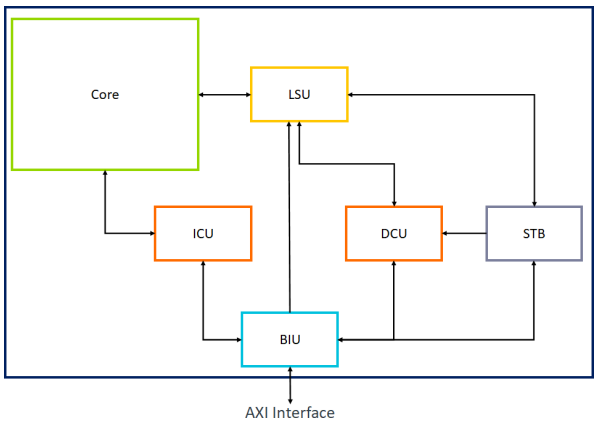
Table 1: Main specifications of the Cortex-M55 core [1].

The Cortex-M55 is a single core design with a 4-stage pipeline, coupled with instruction and data caches that support Error Correction Codes (ECC). It is accompanied by a Floating-Point Unit (FPU), a Neural Processing Unit (NPU) suitable for Machine Learning (ML) workloads and a coprocessor interface, giving integrators the ability to embed their own hardware accelerators. In addition, it includes onboard Tightly Coupled Memories (TCMs) for faster program execution, without the need to access the slower main memory.

The block diagram of the Cortex-M55 processor is visible in Figure 1a. It consists of the previously mentioned functional units, surrounding the main execution core, as well as some debug and trace units necessary for software development. Not all of the aforementioned units are included in the default configuration of the processor, as many of them can be omitted depending on the end application of the partner or system integrator. Some of these units are used for transferring the requested data to and from the core block, while others are either used for interfacing with external hardware (e.g. coprocessor) or providing system-level control. The major modules necessary for computation are visible in the diagram of Figure 1b.



(a) Abstract block diagram showing all available modules for the standard configuration.



(b) Simplified block diagram showing the major modules only.

Figure 1: Block diagrams of Cortex-M55.

These modules comprise of the Core, the Load & Store Unit (LSU), the two caches, the Store Buffer (STB) and finally the Bus Interface Unit (BIU). The Core which includes all arithmetic and logic processing capability fetches instructions from the ICU and performs computations on data arriving via the LSU. The latter has dual-issuing capability, meaning that it can fetch 2x 32-bit data from the available sources, such as the DCU for cache data, or the BIU for main memory or peripheral data. It has local storage and it's main purpose is to fetch and store data as quickly as possible, in an effort to hide the main memory latency from the Core. The final module is the Store Buffer, which collects write data in the form of store commands from the Core (via the LSU), as well as evicted data from the DCU. After grouping and merging the aforementioned data, it sends them to the BIU so that they can be committed to main memory. The STB has a local storage capacity of 5-slots, each containing 64-bits (5x 64-bits).

In the past, STLs have been developed for the Core module of previous projects and can therefore be ported to newer designs to some extent, assuming of course that they were written for the same architecture. On the other hand, the memory subsystem and more specifically the interfacing units towards the outside of the CPU have been far less explored. Nevertheless, these units play a pivotal role in guaranteeing the correct functionality of any core, since corrupted data arriving in the main execution pipeline will almost certainly generate faults or system hangs.

2.2 Bus Interface Unit (BIU)

Given the aforementioned, the Bus Interface Unit (BIU) was selected as the target module for STL development, since it is an integral part of memory subsystem of every Cortex-M processor and has not been exercised yet. It is also a challenging module to work with, due to the following reasons; it is located far enough from the main core, such that it can only be implicitly targeted with regular load/store assembly commands, while at the same it exhibits the typical non-deterministic behavior associated with every memory subsystem. An overview of the BIU module is visible in Figure 2.

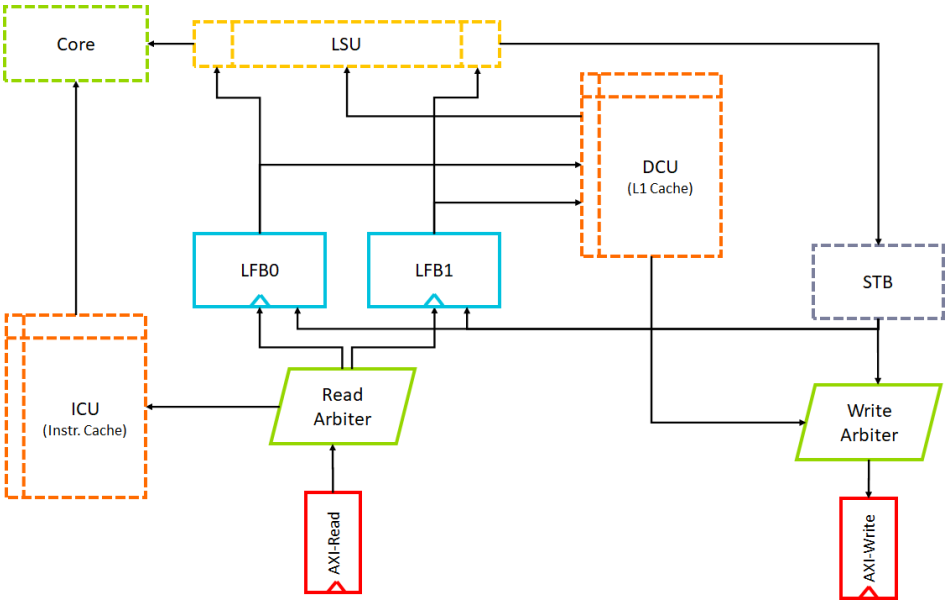


Figure 2: BIU main components and neighbouring modules. The latter are visible with dashed lines.

The BIU provides a connection to the outside world, by interfacing with the Network-on-Chip (NoC) in the form of an AXI Interface. It also routes data to/from neighbouring modules such as the ICU and DCU, the main Core via the LSU and the STB. Data and instructions arriving via the AXI interface are temporarily stored and routed to the correct consumer module (ICU, DCU or LSU), while data moving out of the core, in the form of store commands, follow a different path from the LSU to the STB. Subsequently, data are grouped and merged as necessary, before being committed again to the BIU. The latter contains the following submodules:

- *AXI Read/Write Registers:* These registers receive and send data over the AXI Interface, using an AXI Master device. The aforementioned device uses five independent transaction channels implemented with FIFO queues; two for Read operations (AR for address, R for data readout) and three for Write operations (AW for address writes, W for writing data and B for write acknowledgments) [5]. The registers are 64-bit wide and can accommodate up to 4 consecutive transfers in burst mode.
- *Read/Write Arbiters:* These units contain the necessary logic to allow for arbitration of multiple sources and consumers of data. Given the nature of the BIU, it is expected that

there can be multiple data requests at the same time, therefore the required arbitration is implemented with fixed priorities. For reads, data have priority over instruction fetches to the ICU, whereas fore writes, evictions from the DCU have priority over data from the Store Buffer.

- *Linefill Buffers (LFBs):* These two buffers receive a linefill-worth of data (256-bits) and deliver the requested words of 32-bits to the pipeline, via the dual channel LSU. At the same time, they can also allocate a linefill of data in the DCU, meaning that future requests from the same addresses will be directly served from the cache. Finally, they can accept and merge store data from the Store Buffer module, whenever there are pending load requests to the same addresses.

- *Hazarding & Eviction logic:* These modules are responsible for detecting hazards between successive data linefills, such as Read-After-Write (RAW), or Read-After-Read (RAR) hazards to the same address as the one currently in progress. They are also able to detect hazards associated with cache evictions, since data words may not allocate in the cache until the data they are replacing have been evicted. The modules have multiple connections with almost all of the aforementioned modules, therefore they have been omitted from Figure 2 for the sake of simplicity.

- *Prefetcher logic:* This module looks for patterns in incoming addresses and tries to make a prediction on the next batch of addresses that could be requested. It therefore fetches data from the main memory before they are actually needed, in an effort to hide the memory latency from the Core. The Prefetcher contains two FSMs which can process two prefetch streams simultaneously, by looking for patterns of constant matching strides. If one of the two FSMs detects three consecutive linefills of constant stride, it attempts to prefetch the next one and allocate it in one of the two available LFBs. Given the complexity and number of connections, the module has been omitted from Figure 2.

In order to effectively stress the BIU, the STL parts will need to be developed in such a way, so that they target one or more of the aforementioned submodules in a meaningful way. This allocation of parts to logic modules is also in line with the general STL architecture as described in Section 3.1. In addition, it also helps in breaking the problem of targeting multiple interrelated modules, into manageable parts with defined interfaces and functionality.

2.2.1 Principle of operation

A regular cacheable load command, results in data being delivered from the AXI Read Register to the Linefill Buffers, starting allocation always form LFB0. In case, LFB0 is full, then the linefill is allocated in LFB1. While inside the Linefill Buffers, two simultaneous activities can take place. First, a request is sent to the DCU and upon approval, the data are allocated to the correct set and way of the cache. At the same time, if there is a pending request from the pipeline, the corresponding data word is delivered to one of the two ports of the LSU. The former activity is always performed, while the latter is performed only on load commands, depending on the program flow.

A non-cacheable load will deliver data from the AXI Read Register directly to the LSU, using a dedicated bypass path which is not visible in Figure 2. That way, no allocation can take

place neither in the LFBs nor the cache. On the contrary, a regular cacheable store command will follow the path from the LSU towards the STB, which collects all store requests, to one or more available slots. Subsequently, the stores are merged and forwarded to the LFBs in order to be allocated in the cache. Any stores to non-cacheable addresses are grouped and then forwarded to the Write Arbiter, where they are temporarily stored together with any evicted linefills from the DCU. The arbitration system will then select which of the two sources will commit its contents to the Write Register of the AXI bus, thus fulfilling the respective requests.

2.3 CPU Revision

As previously mentioned, this work will be carried out using the Cortex-M55 processor as a reference platform, in order to develop the proof-of-concept. Since the selected processor's development schedule will exceed the time frame of this thesis, it is important to select a specific version and a target implementation technology early on. As a result, all of the work will be carried out on the same underlying platform, avoiding any divergences in the reported results. For that reason, the *Revision-0* or simply *r0* was selected as the baseline for this project, which has almost all of the features implemented and is available for Limited or Early Release Access. In addition, the processor can be implemented with one or more target technologies, each one presenting different area, power and performance characteristics. In the context of this project, a TSMC $40nm$ LP technology was used for the netlist during the fault simulation step, as will be presented later on in Section 5.1

3 STL Design

This Section corresponds to Chapter 4 of the Main Report and includes the detailed design of all work performed during the thesis. To simplify the process, certain STL parts with similar characteristics targeting neighbouring modules have been grouped together. More specifically, Section 3.1 will introduce the concept of STLs and the existing framework, which form the starting point of this work. Section 3.2 describes the tests developed to stress the AXI registers, along with the accompanying arbiters. Furthermore, Sections 3.3 and 3.4 elaborate on the parts specifically designed to target the LFBs, although by definition all tests would exercise the buffers implicitly up to a certain extent. In addition, Section 3.5 provides insight into the parts stressing the Hazarding logic and Section 3.6 describes the test dedicated to the Prefetcher module. Finally, Section 3.7 presents an overview of all developed parts.

3.1 Overview: STL architecture

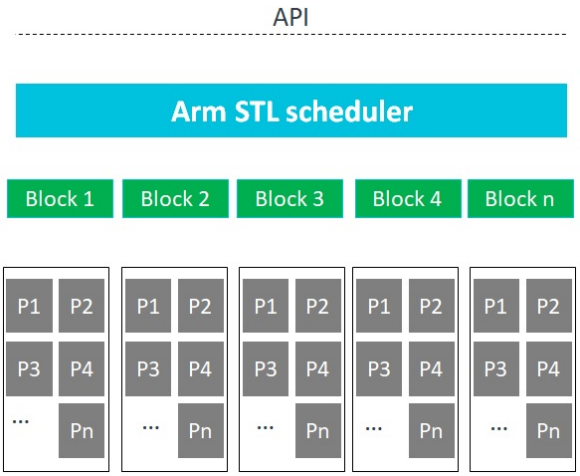
STLs are efficient assembly test routines designed to stress one or more subsystems inside a modern ARM processor. They can be executed during startup or run periodically with the use of a scheduler designed in-house. A C-based Application Programming Interface (API) gives the prospective system designers and integrators an easy-to-use and flexible interface to combine multiple STL routines and form larger test suites. An overview of the aforementioned infrastructure is visible in Figure 3a.

Each STL suite corresponds to one block, containing one or more parts. In essence, every part is targeting a specific submodule or feature within a functional unit of the processor. They are designed in a fully self-contained way and have a bounded execution time of < 6000 clock cycles. As a result, they are deterministic in nature allowing their integration into larger blocks, which in turn are interruptible and relocatable in memory. The prospective user can then schedule one or more blocks of preference, depending on the application requirements of the target computing system. An example of periodic STL scheduling can be found in Figure 3b, where execution is interleaved between the main process and a selected STL part, given a user defined period. In this scheduling scheme, a fixed number of parts is executed per system call and depending on the duration of the part, there might be periods of idle time. The aforementioned STL framework served as the basis for the thesis project, since the tests developed for the BIU module would need to be compatible with the scheduling platform, so that they can be integrated together with other STLs and form bigger test routines. The tests developed in the context of this thesis comprise of a single block, whose parts will be presented in the following Section.

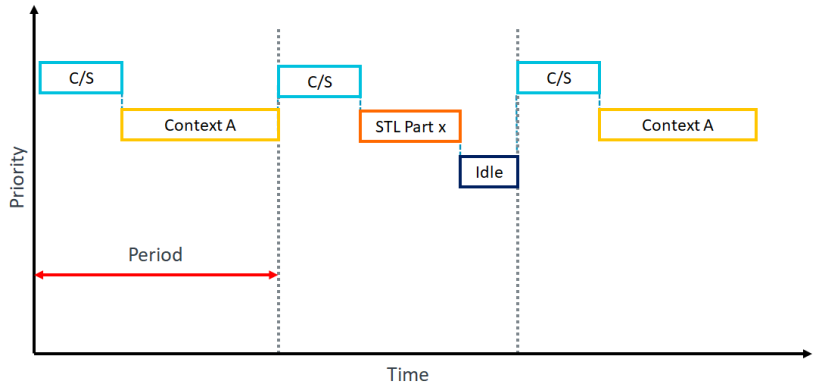
3.2 AXI Interface part

The first part in the STL suite focuses on stressing the interfaces of the BIU with the outside world and namely with the AXI bus. The test consists of three parts which will be presented below:

- *Data Generation:* During the first step, a series of data blocks mapped to specific pre-allocated addresses are generated and will be used locally throughout the test. The data blocks are equal to three linefills and consist of series of decrementing complementary data patterns, using cacheable complementary addresses. Using complementary data



(a) STL software architecture and application interfaces [6].



(b) Online Time mode (OLT) STL execution of a given part with the user's application. Context switches are labelled as *C/S*. Adapted from [6].

Figure 3: The STL framework, which acts as the starting point for this work.

and address patterns effectively increases the likelihood of having a stuck-at-fault detected, due to the increased toggling of the data, address and command lines in between two successive loads. This subsequently offers increased coverage in the AXI Interface Registers and arbiters, as well as in both LFBs.

- *Cache evictions:* Given that the data series are generated as a sequence of store commands, they will follow the store path from the LSU to the STB and consequently allocate in the DCU, via the two LFBs. This is part of the normal operation of the module, however during the specific scenario the goal is to stress the AXI interface and not the cache. Hence, all data are then evicted from the cache, using the DCCIMVAC system register together with the target addresses [7]. This is done in order to force the data to main memory (via the AXI Bus), so that the next load request would read data using the process described previously for regular cacheable loads in Section 2.2.1.

- *Load & Check:* After all evictions are completed, a series of load commands request all data from the aforementioned complementary addresses. Data move from main

memory back to the BIU, using the load path presented in Section 2.2.1. The series of commands is written in such a way so that both LFBs are used interchangeably, starting from LFB0. Finally, all data are checked for consistency inside the aforementioned Load & Check loop, that compares each word within a linefill with its predetermined value. If the comparison succeeds then no errors were detected and the process continues until completion. Otherwise, a failed comparison signals a detected stuck-at-fault and the program execution is interrupted, branching to a specific fail routine. This in turn restores the context and notifies the scheduler about the detected fault.

This is the first test in the STL routine and completes in 953 cycles.

3.3 Linefill Buffers part

There are two parts which target explicitly the two LFBs, focusing on the load and store paths respectively. This test focuses on the load path and has a similar structure with the previous one, apart from the load and checking sections of the code.

In this case, loading and checking of data occurs in two ways:

- **Directly,** as data arrive in the LFBs, before any allocation in the DCU can take place. This is achieved using a series of load-multiple and compare commands against predetermined values. That way, all words within a linefill are delivered to the LSU directly via the LFBs, providing increased coverage on the BIU-LSU interface. Since the BIU module also serves instruction fetches to the ICU, there can be cases where load patterns are interrupted, in order to fetch the next batch of instructions. In this test, special care was taken in having the code aligned in a way, so that the series of loads and comparisons execute without any interruption. This is achieved using the *.align* directive, which is supported by the ARM Compiler [8].
- **Indirectly**, after data have been allocated in the cache, using Load & Check loops, similar to the previous part. Due to the slower execution of the loop, the LFBs have enough time to allocate their contents in the DCU. Therefore, all requested data are eventually delivered from the cache. This effectively checks for faults indirectly in the buffers, since a fault there would have been forwarded to the cache as well. At the same time, the interface between the LFBs and the DCU is also checked, providing additional coverage gains.

This is the second part of the STL routine and requires 595 cycles to complete.

3.4 Store Buffer part

This test is also used to stress the two LFBs, but contrary to the previous one, it focuses on the merge path from the STB to the LFBs, as seen in Figure 2. As in all the previous cases, it generates locally a series of data patterns, which are then written to the STB and subsequently generate a series of linefill requests to the LFBs interchangeably. This happens because all cacheable stores need to allocate to the DCU, hence the merge path and associated logic initiate the aforementioned allocations to the LFBs. The execution continues with the necessary Load & Check loop, in order to determine whether there was a fault in any of the incoming data. This is the third part of the STL routine and completes in 532 cycles.

3.5 Hazarding & Evictions part

This part focuses on the logic responsible for detecting hazardous conditions. It generates and commits data to main memory, in the same way as described in Section 3.2. The rest of the test consists of the following two sections:

- *Introduction of hazards:* After all data have been written to main memory, hazards are introduced by modifying selected words within the committed linefills. A load loop subsequently reads all linefills from memory, while the hazards are still in the STB. In order to resolve the RAW hazard, the new data should be merged in the LFBs together with the incoming stream of original data. This sequence checks for faults directly in the hazarding logic of the LFBs, as well as in the interfaces between each LFB and the STB.
- *Creating evictions:* The second section of this test builds on the first one, by creating linefills mapped into L1 cache sets, that contain the dirty and valid data of the previous section. As a result, evictions will need to happen, further increasing the toggle coverage of the test. The evicted data will follow the path from the DCU to the Write Arbiter and consequently to the AXI-Write channel. Finally, a Load & Check loop checks all linefills for consistency, both the ones allocated in cache, as well as the evicted ones. The latter check is necessary in order to make sure that evictions did indeed happen. If data loaded from the respective addresses contain the original data, then this signals that evictions did not occur, pointing to a fault in the hazarding or eviction logic of the BIU.

This is the fourth part of the STL routine and completes in 992 cycles.

3.6 Prefetcher part

The final part of the BIU tests focuses on the Prefetcher logic and the associated Finite State Machines FSMs and internal buffers. The structure of the part is similar to the aforementioned ones, with the difference of using selected address ranges which will trigger the Prefetcher module to start allocating subsequent linefills.

The Prefetcher Unit is enabled only when three load requests, from addresses with fixed strides, are detected. In this case, the concept of a stride is defined as a specific bit-field within every address (bits[12:5]), that is used as a means of detecting neighbouring addresses which are one stride apart. An internal buffer is shared between the two FSMs and is used to track all linefill requests made. In order to keep the process going, several loads are used to keep the Prefetcher active, until its internal buffer is full. The commands are also written in such a way as to stress both LFBs interchangeably. Once again, a two-part Load & Check sequence checks the data for consistency, directly as they allocate in the LFBs and indirectly through a subsequent load loop from the DCU. This is the fifth part of the STL routine and completes in $1,405$ cycles.

3.7 Overview

During the development of the STL tests, a state-of-the-art hardware simulation environment was used from Mentor, called Questa [9]. In order to develop tests, the first step was to study the BIU specifications and consequently the respective RTL module. Then, a series of sample tests would provide the first feeling of the module's behavior, on a cycle-by-cycle basis. To check whether the developed code actually exercised the target submodule in the correct way, a waveform viewer was used to determine whether the actual behavior corresponds to the desired one. The process continued in an incremental fashion, adding more tests targeting different submodules of the BIU. An overview of all tests, along with their execution times in cycles, is given in Table 2.

STL tests	Cycles
AXI part	953
Linefill Buffers part	595
Store part	532
Hazarding & Evictions part	992
Prefetcher part	1,405

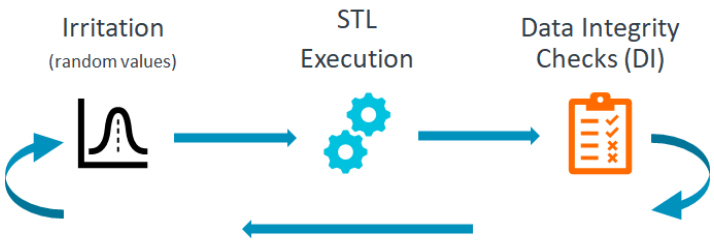
Table 2: Overview of all BIU STL-parts.

4 Verification Design

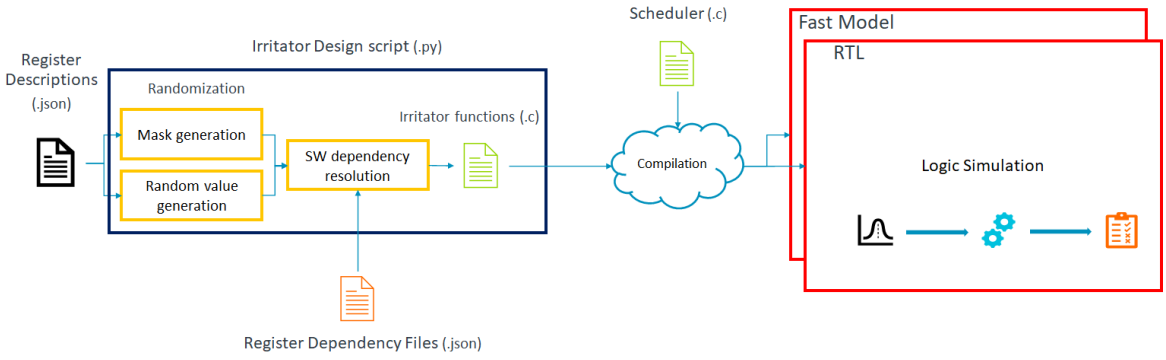
This Section also corresponds to Chapter 4 of the Main Report, but will include the whole verification effort made in order to validate the correct functionality of the aforementioned STLs. Details on the process, the steps and the interfaces with the work performed for STL Design will be given in Sections 4.1 and 4.2, whereas Section 4.3 will present the implemented framework necessary to support regressions.

4.1 Verification Process

An integral part of the STL Design process is to verify that the developed tests can execute on the target platform, without corrupting the state of the system. At the same time, parts should be able to run on multiple hardware configurations of the same CPU, either in the context of a Real-Time Operating-System (RTOS) or as part of a baremetal application. The aforementioned factors impose a very high number of possible initial system states. In each of these cases, STLs should be able to execute correctly, without corrupting the system's architectural state. Since the number of possible configurations is exponentially high, there needs to be an effective and automated way of performing testing. This is achieved through the process of verification, an overview of which can be seen in Figure 4a.



(a) Simplified overview of the Verification Process.



(b) Irritator Design steps and integration into the verification flow.

Figure 4: STL Verification Process for Cortex-M55.

The rationale behind the verification effort is based on the following three simple steps:

- *Irritation:* The first step attempts to re-create the many different initial states that a potential system could be set in, while running arbitrary software. More specifically, several possible architectural states are generated by writing randomly constrained values to memory-mapped system registers, which in turn control the processor's configuration and behavior. In total, there are more than 150 system registers in the Cortex-M55 processor, which are read and written using regular load/store assembly commands to the respective memory addresses.
- *STL Execution:* Parts are scheduled either in the context of an RTOS or baremetal application, as previously seen in Figure 3b. In any case, part execution should neither corrupt the context of the interrupted client application, neither it should change the configuration of the CPU, by accidentally corrupting the values stored in one or more system registers.
- *Data Integrity (DI) Checks:* The final step reads back the values of the system registers, and compares it with the original ones, prior to STL execution. Any inconsistencies in values, point to a badly written STL part, which corrupted the initial system state, by altering one or more stored values.

The Verification Process needs to be performed for all different combinations of the applicable system registers, in order to verify that the tests can execute in a variety of target systems configurations. The Cortex-M55 implements more than 150 system registers as described in the ARMv8-M architecture manual, each one having one or more bitfields dedicated to the control of a specific function. In total, the number of possible combinations increases substantially, making any effort for manual checks futile. Hence, it is only possible to thoroughly verify the correct functionality of STLs, in an automated way. At the same time, the automated process should not be custom-tailored to a specific CPU or project, since a lot of rewriting and re-verifying would be required for every new project. As a result, it is imperative that the current flow is abstracted and generalized to the extent possible, in order to enhance its portability and enable support for future M-Class processor designs with minimal overhead. This will also help in reducing the number of human errors, introduced while performing modifications and at the same time it would reduce the development effort. The aforementioned would also satisfy the project's objective of providing a unified and flexible flow, as described in Section 1.2.

4.2 Irritator Design

One important area of potential improvement is the design of the irritation software, which is the first step in the Verification Process, as presented in Section 4.1. Previously, irritation was done by creating intermediate representations of system register descriptions and hard-coding certain constraints in the main verification script, so that constrained random values could be generated for the respective bitfields. In order to make the procedure as abstract as possible, the whole flow is re-written, using basic parts of the old script and modifying core functionalities such that the end result satisfies the aforementioned objectives. The general architecture of the new design is visible in Figure 4b. The process involves the following steps:

- *Register Description:* The input of the flow is given in the form of *.json* files delivered by the Design Team, containing a standardized description for each system register. This description includes the available bitfields, their supported values or ranges, as well as

any *Reserved* bits. These bits are defined in the ARMv8-M architecture and correspond to bits whose values should not be written by any software. Writing can cause undesired effects such as system crashes or hangs.

- *Irritator Design script:* This script is an important part of the whole verification flow, as it parses the aforementioned register description files and determines which registers to randomize. The ARMv8-M architecture defines several types of system registers which can be Read/Write (RW), Read-Only (RO), Write-Only (WO) or Write-Clear (WC) to name a few. The developed script automatically detects which registers can be randomized and parses all of their bitfields in sequence. This is an important step in order to determine which bits will be randomized with constraint values. Additional functionality was added in order to be able to interpret correctly the desired functionality of each bitfield. Sometimes bits can be *Reserved* or WO, while other times the register description file uses different syntax to refer to the same underlying functionality. In addition, support was added for multiple bit dependencies, as in some cases, certain system registers can only obtain specific values depending on the value of another related register. These dependencies need to be provided in the form of separate *Register Dependency Files*, but only for the affected registers.

- *Irritator functions:* The Python script generates a C-file, which contains three irritation functions. The first one is used to initialize a register struct, which contains all pre-computed random values, as well as the memory addresses used to access the registers. The second function randomizes all system registers, by reading the initial value of the selected register, applying an appropriately sized mask to any *Reserved* bits and finally, by inserting the precomputed random values to the remaining bits. All registers are initialized to zero during boot-time. Finally, the last generated function performs DI checks at the end of the logic simulation and is used during the consistency-check step, as seen in Figure 4b. All generated functions are called from within the main scheduler file which in turn will schedule one or more STLs to run during simulation. The necessary code is compiled into a single binary which is used as a payload during logic simulation.

- *DI checks:* After irritation and test execution are completed, the values of all system registers are read back, in order to check whether they have been corrupted during STL execution. Each register value is compared to the randomized value written during the irritation step. Any divergences hint to a potentially badly designed STL part, or to a bug in the Irritation Script.

The aforementioned flow is compatible with both the Fast Model (FM) or RTL representation of Cortex-M55. The Fast Model is a C-based description of the processor which is faster but less accurate than the RTL simulation. The flow provides the necessary portability required, since it is based on a single Python file, which generates everything else. It provides a structured way of generating constrained random values for system registers, having a single requirement on the *Dependency Files*. These files need to be hand-written, only when changing architectures. As a result, two implementations of the same instruction set could be accommodated this way, without any need to change the aforementioned files. Moreover, the list of dependent registers in a given architectural set is rather short. Therefore, creating the *Dependency Files* by hand is considered a good trade-off compared to having the whole framework re-adapted or re-written from scratch.

4.3 Regressions

Given that the Cortex-M55 CPU has more than 150 memory-mapped system registers [10] and that each register has one or more bitfields dedicated to separate functions, the total number of possible states increases dramatically. Therefore, the process depicted in Figure 4b is performed in regressions, which are multiple parallel runs, each one with a different random seed used for register irritation. This way, many errors in STL design that would otherwise remain masked or undetected using only logic or fault simulation, can be captured and corrected. As a result, the Verification Process increases the portability and robustness of STL tests, making sure that they will work in a number of different system configurations, targeting a wide range of applications.

More specifically, STL tests would need to be tested both in the context of a RTOS, as well as in the context of a baremetal application. This, in combination with the system register randomization, provides a wide spectrum of possible system states. In order to test exhaustively all possible states, several hundred or thousands iterations will be required. Regressions are performed on both the FM and RTL descriptions of the processor, since each one provides its own benefits. The FM description is faster and thus is used to perform long iterations in order to cover larger parts of the exploration space. In addition, it uses the THREADX real-time operating system [11] to schedule one or more STLs, hence providing the aforementioned testing context. On the contrary, the RTL version uses a baremetal scheduling framework and is significantly slower, requiring up to 4 hours to run 15-iterations on a subset of registers. However, its increased accuracy makes it ideal fort testing the most interesting cases identified during the FM-regressions.

All regressions are run on a remote cluster in a batch of 15 iterations. The Verification Process visible in Figure 4b was further expanded to support a series of interfaces with configuration scripts and makefiles, in order to be transformed into a completely automated process. The main regression script is scheduled to run periodically, either in a daily or usually weekly fashion. The script takes as a parameter the desired model (FM or RTL) and calls all the necessary makefiles and configuration code in sequence, to first compile the source software and then the selected FM or RTL version of Cortex-M55. Each time, a different seed is generated during the code compilation, which will be used by the Irritator script to compute the random values. This ensures that each iteration is different from the previous one and makes it possible to recreate the random values of a selected iteration, in case there is an error. All outputs from the different steps of the process are logged in a file, to facilitate the debugging process when necessary.

5 Results

This Section will present the outcome of the thesis and corresponds to Chapter 6 of the Main Report. It includes detailed results on the fault coverage metrics as well as examples from the regression work. These were omitted from the Main Report, since they contain sensitive information. An elaborate explanation will be given and any deviation from the intended outcome will be highlighted. Section 5.1 focuses on fault simulations and Section 5.2 on the outcome of the verification work.

5.1 ZOIX Fault Coverage

The developed STL parts, as described in Section 3, were tested for fault coverage, using a certified version of Synopsys ZOIX Simulation software [12]. This is a proprietary Fault Simulation Environment, that uses a fault injection mechanism to test modern VLSI systems for permanent or transient faults [13]. The tool performs logic simulation of the given hardware design running the payload software, effectively checking for compliance with ISO 26262 and IEC 61508 safety standards. Section 5.1.1 will elaborate on the permanent fault simulation results, whereas Section 5.1.2 on transient fault simulation. A sample of the generated coverage reports is visible in Appendix A.

5.1.1 Permanent Faults

Table 3 summarizes the results obtained when introducing stuck-at-faults only in the BIU unit, using the *r0* gate-level (netlist) representation of Cortex-M55. This provides the most accurate results possible in a simulation environment, since it is using a representation of the design very close to the actual physical implementation of the chip. The fault universe is split into two main categories of faults, namely the Testable and Untestable faults. The former, as their name suggests, consists of all the faults that can be tested in the given design and are taken into account during the final fault coverage calculation. The latter, consist of faults being structurally undetectable, given that they can not be detected by any workload. They are either tied to a supply or ground net (V_{DD} or GND) or being blocked by another signal being tied to them [13]. Hence, Untestable faults are not taken into account during the fault coverage calculation, since they do not influence the behaviour of the circuit.

The results of permanent fault simulation visible in Table 3, present the achieved coverage for the major modules only. The coverage number is calculated, using the following formula:

$$Coverage(\%) = \frac{DD + DF}{DD + DF + ND} \cdot 100\% \tag{1}$$

where, Dropped Detected (DD) represents the number of detected faults, Detected finish/stop (DF) is the number of faults triggering a timeout and Not Detected (ND) is the number of undetected faults [13]. The DF fault category represents faults that can be detected with the use of a timer or Watchdog mechanism. The total fault coverage is calculated using the data from Table 3 and substituting them in Equation (1), as follows:

$$Coverage(\%) = \frac{38,816 + 1,728}{38,816 + 1,728 + 83,093} \cdot 100\% \approx 32.79\% \tag{2}$$

BIU module	**Testable Faults**			***Total***
	DD	DF	ND	
LFB 0	7,593 (58.12%)	113 (0.86%)	5,358 (41.01%)	13,064
LFB 1	7,394 (56.42%)	217 (1.66%)	5,495 (41.93%)	13,106
Write Arbiter	8,468 (38.51%)	333 (1.51%)	13,188 (59.98%)	21,989
Read Arbiter	750 (34.03%)	1 (0.05%)	1,453 (65.93%)	2,204
AXI Interface	2,853 (62.74%)	16 (0.35%)	1,678 (36.90%)	4,547
Hazarding logic	475 (4.40%)	191 (1.77%)	10,122 (93.83%)	10,788
Prefetcher logic	39 (0.33%)	332 (2.79%)	11,548 (96.89%)	11,919
Test Coverage	38,816 (31.40%)	1,728 (1.40%)	83,093 (67.21%)	**123,637**

Table 3: Snapshot of the permanent fault simulation results for the major modules only, using ZOIX 2017.12-SP2. Testable faults are listed as Dropped Detected (DD), Not Detected (ND) and Detected finish/stop (DF). Untestable faults are not listed, since they do not contribute to the final coverage calculation.

The largest modules in terms of size and number of faults are the Write Arbiter and the two LFBs. This was expected, since the core BIU functionality is implemented in the aforementioned buffers, whereas the Write Arbiter has internal buffering to store evicted data or non-cacheable writes to main memory. The Hazarding & Prefetcher logic are similar in size, however only the prefetch datapath contains local buffering, which could explain its size in faults. This highlights how complex the Hazarding module is, which is only a few thousand faults short from the core modules of the BIU. From the results, it is evident that the current tests achieve good coverage numbers on the two LFBs, detecting more than 50% of the total faults. Similar performance, albeit lower, is achieved for the Write Arbiter, where 38% of the faults were detected. Both AXI Read/Write Registers, listed as AXI Interface in Table 3, are covered up to almost 63%, however this module is relatively small in size. The same holds true for the Read Arbiter, which amounts to $2,204$ faults. Therefore, even though coverage is satisfactory on both modules, it is expected that their contribution to the total BIU coverage would be rather small.

On the other hand, performance in two major modules inside the BIU, namely the Hazarding and Prefetcher logic, is not satisfactory. Even though both tests as described in Sections 3.5 and 3.6 seem to exercise the subsystems up to a certain extent, it is obvious that the results can be improved substantially, by creating more advanced test scenarios. However, this might be more challenging to accomplish when compared to writing tests for the LFBs, given the inherent difficulty of writing advanced hazard tests. Overall, the total fault coverage is $> 30\%$, which is a satisfactory first result, given the fact that the BIU has not been targeted before. From there on, several iterations will be required in order to increase coverage further.

5.1.2 ZOIX Transient Faults

Transient fault modeling was performed with a newer version of ZOIX (2019.06-SP2-3), which supported the use of injection intervals. Given the nature of transient faults, as explained in Chapter 5 of the Main Report, transient fault injection was performed using the transient toggle model. Hence, faults were placed in all possible locations of the BIU at every

time interval, by inverting the value that a non-faulty machine would have at the exact same location and time [13]. This created a large fault universe, which could reach up to $10.8M$ faults for $250ns$ intervals. The fault simulation process is using a distributing computing framework, partitioning the total number of faults into jobs and assigning them to available machines. Given the number of licenses, as well as the large number of faults, decreasing the injection interval further was not possible. As a result, the transient fault analysis started from $250ns$, gradually increasing to several microseconds for the duration of the STL tests. Table 4 summarizes the fault coverage metrics, using an interval of $1000ns$.

BIU module	**Testable Faults** DD	ND	***Total***
LFB 0	25,807 (9.17%)	165,562 (58.81%)	281,501
LFB 1	33,270 (11.82%)	159,414 (56.63%)	281,501
Write Arbiter	26,954 (5.56%)	331,937 (68.52%)	484,436
Read Arbiter	7,743 (16.97%)	30,486 (66.80%)	45,640
AXI Interface	12,119 (11.13%)	27,407 (25.17%)	108,884
Hazarding logic	4,056 (1.66%)	236,212 (96.55%)	244,663
Prefetcher logic	1,748 (0.64%)	213,380 (78.67%)	271,232
Test Coverage	164,627 (6.05%)	1,954,281 (71.77%)	**2,723,078**

Table 4: Transient fault simulation results for the major modules only, using an interval of $1000ns$. Only Dropped Detected (DD) and Not Detected (ND) faults are listed, as the rest have been omitted for simplicity. The total achieved coverage, including DF faults is 6.83%.

From the results, it is obvious that the total number of faults has increased substantially. This is due to the fact that the fault simulation injected a fault every $1000ns$ which is quite high. The coverage results are generally much smaller than the corresponding ones in Table 3, but this was expected due to the nature of transient faults and the inherent difficulty of detecting them using a software method. Once again, the two LFBs exhibit relatively high coverage numbers up to almost 12% and the same holds true for the Read Arbiter and AXI Interface, which have a coverage of approximately 17% and 11% respectively. The same cannot be said for the Write Arbiter, which experienced a big drop when compared to the aforementioned modules. Finally, as in the previous case, both the Hazarding & Prefetcher logic have quite low coverage numbers, which once again point to the fact that the developed STLs are not exercising them enough.

As mentioned in Chapter 6 of the Main Report, several transient fault simulations have been performed in order to characterise the STLs across a wide spectrum of injection intervals. It was observed that reducing the injection frequency (increasing interval), resulted in a coverage drop especially towards the end of the BIU tests at $160,000ns$. For that reason, several other tests have been performed at absolute intervals, in an effort to better quantify the average achievable coverage. The tests produced similar results to the ones presented in Table 4 and their total coverage numbers have been documented in the Main Report. Hence for the sake of simplicity, the detailed coverage numbers on a per module basis for all simulation runs will be omitted.

5.1.3 Porting to Revision-r1

The STL tests were developed using a specific version of the Cortex-M55 processor and target netlist. More specifically, the *r0* version was used for development and fault simulation experimentation, hence the results are applicable to the specific version. When the tests were ported to the *r1* version, there was a difference in the reported coverage. More specifically, the STL tests achieved 20.37% on stuck-at-fault coverage, which amounts to a significant drop. This was attributed to the fact that all modules increased in size, having additional logic being added which resulted in a different netlist. More specifically the Write Arbiter, which is still the largest module, increased by $3,755$ faults. Furthermore, the Hazarding & Prefetcher logic also increased in size considerably, having approximately $2,000$ and $1,500$ additional faults respectively. An overview of the changes in fault count is presented in Table 5.

BIU module	*Revision-r1*	*Revision-r0*	**Faults difference**
LFB 0	13,838	13,064	774
LFB 1	13,774	13,106	668
Write Arbiter	25,744	21,989	3,755
Read Arbiter	2,930	2,204	726
AXI Interface	5,255	4,547	708
Hazarding logic	12,694	10,788	1,906
Prefetcher logic	13,411	11,919	1,492
Total	138,559	123,637	14,922

Table 5: Comparing r0 and r1 Revisions of Cortex-M55, in terms of permanent faults. Results are presented for the major submodules only.

It is evident that the changes in hardware impacted greatly the total number of stuck-at-faults reported during fault simulation. All major submodules increased in size and the total number of faults increased almost by $15,000$ faults. Putting things into perspective, this corresponds to an additional Linefill Buffer or Prefetcher module. Furthermore, such changes in module size have also an influence on the cycle-by-cycle behavior, meaning that certain STL parts might not be able to stress the corresponding submodule, the same way they did before. These changes are expected to have an effect on the transient fault coverage as well, which may experience an even larger drop. Hence, it is evident that the developed tests are sensitive to hardware changes, which was expected to some extent, given the nature of STLs. Since the design of the processor is still ongoing, such changes will be frequent and more work will be required in order to increase the fault coverage on the new Revision.

5.2 Verification Design

As mentioned in Section 4, the verification work focused on developing the master Python script, which generates all the necessary files for simulation. Prior to integrating the Verification Software in the main flow, several logic simulations were performed while only irritating the system registers and performing DI checks. This was necessary in order to make sure that the newly written software was functioning as expected and generated the appropriate

constrained random values. In addition, many dependencies were uncovered which were later encoded into the Dependency Files. In total, 105/154 registers could be randomized, with the remaining ones having complex hardware dependencies. In order to proceed with the rest of the work and develop an end-to-end verification flow, it was decided not to include hardware dependency support in the Python script, at least for now, but instead proceed with the regressions. Having a complete flow working with less functionality was more important than fully developing a specific subsystem of the whole flow.

An overview of the Verification Software capabilities is visible in Table 6. The design supports both the FM and the RTL description of the Cortex-M55 processor and is able to perform scheduled regressions of 15 iterations each, both locally and on the remote cluster. The aforementioned number of iterations was considered a good trade-off between adequate randomization and execution time for a given regression run. Almost all of the originally planned functionality has been implemented, with the exception of hardware dependencies and randomization features. In the future, the script will be able to randomize the hardware parameters as well before performing logic simulations, changing for example the amount of cache memory between successive iterations. In total, 41 registers have been included in the Verification Process, due to the fact that the FM presents some divergences in behaviour and configuration, compared to the RTL. More specifically, some of the debug and trace modules presented in Section 2.1 have not been implemented, hence any system registers associated with these modules are missing. Therefore, the total number of registers was reduced to a basic set that is supported by both versions.

Verification features	**Status**
System register parsing	✓
Mask generation	✓
Random seed/value generation	✓
Dependency resolution (software only)	✓
FM/RTL Simulation	41/105 registers integrated
Regression support	15 iterations with random seeds

Table 6: Overview of the capabilities of the Verification Software.

The log files of the completed regression runs can be up to 60,000 lines long, therefore they can not be included in this Section. However, screen captures from both RTL and FM regression runs are visible in Appendix A.

6 Conclusion & Future Work

This Section will elaborate on the conclusions and recommendations that are applicable for the Detailed Design Report. The main conclusions with respect to the objective of the thesis, as well as some general points for future work, are presented in Chapters 7 and 8 respectively of the Main Report.

6.1 STL parts

The results summarized in Section 5 are quite encouraging in terms of random fault detection on the BIU module. As already mentioned in Section 2.2, creating test scenarios that stress the BIU to a great extent is challenging at best, due to the nature and position of the module within the processor's hierarchy. Given the aforementioned, it was estimated that a $> 30\%$ fault coverage would be a good starting point, especially for a module targeted for the first time.

Moving forward, it is obvious that more work is necessary in order to stress the last two big submodules within the BIU, namely the Hazarding & Prefetcher logic. Given their size in terms of total faults, as well as the low achieved coverage, it is expected that there are more gains to be had from the aforementioned modules, that would greatly impact the total coverage of the BIU and in turn the CPU's. This is particularly relevant for the r1 Revision of the CPU, which includes an even larger version of the aforementioned modules. In addition, almost all modules will need to be revisited, since the change in size will almost definitely have an impact on the module behavior. As a result, the stress patterns of some tests may have been influenced and further analysis is required in order to determine the extent at which the existing tests will need to be adapted.

6.2 Porting to another CPU

Another major point in the Purpose Statement of Section 1.2 is the ability to execute the designed tests and verify their behavior, using another processor within the same architectural extension. Even though the tests are compatible with any M-class processor, requiring few changes on the memory address patterns, it is expected that any porting activity will require additional time and some modifications, in order to match the specifications of the new processor. As seen in Section 5.1.3, porting the STL tests to a different Revision of the same CPU resulted in coverage fluctuations. Hence, even though the portability aspect of this work is to some extent achieved, additional development and testing time is required, in order to ensure that the tests work with different processor models with minimal coverage drops.

6.3 Verification Support

Finally, on the verification side, Section 5.2 presents the work performed in order to build the verification flow and add support for automated regressions. This was considered an important milestone, since the tests' correct execution could be verified across a wide spectrum of system configurations. In the future, several new features could be added in order to enhance the existing framework and enable additional functionalities. More specifically, support should be added for hardware dependencies, since the values of certain bits in several system registers are dependent on the hardware configuration, such as the existence of ECC

memory. Furthermore, multiple hardware configurations will need to be dynamically generated, so that the tests are also verified across all possible versions of Cortex-M55. By taking the current end-to-end developed framework and expanding it further to support even more registers, additional dependencies could be uncovered. This activity will not only benefit the verification design, but also help develop more robust and reliable STL tests, across multiple modules or processor models.

References

[1] ARM. Arm Cortex-M55 Processor. *Product Brief*, 2020.

[2] Steven M. Guertin, Mehran Amrbar, and Sergeh Vartanian. Radiation Test Results for Common CubeSat Microcontrollers and Microprocessors. *IEEE Radiation Effects Data Workshop*, 2015.

[3] Christian M. Fuchs, Nadia M. Murillo, Aske Plaat, Erik van der Kouwe, and Peng Wang. Towards Affordable Fault-Tolerant Nanosatellite Computing with Commodity Hardware. *IEEE 27th Asian Test Symposium*, 2018.

[4] Kenneth A. LaBel and Michele M. Gates. Single-Event-Effect Mitigation from a System Perspective. *IEEE Transactions On Nuclear Science*, 1996.

[5] ARM Limited. AMBA AXI and ACE. *Protocol Specification*, 2019.

[6] Mukesh Lahori and Kausar Johar. SW-based, Run-time Diagnostic Tests to Minimize System-level Effort. *ARM TechCon*, 2018.

[7] ARM Limited. ARMv8-M Architecture. *Reference Manual*, 2017.

[8] ARM Limited. ARM Compiler Version 6.14 Reference Guide. 2020.

[9] Mentor. Questa Advanced Simulator. `https://www.mentor.com/products/fv/questa/`. [Online; accessed September 2020].

[10] ARM Limited. ARM Yamin Processor. *Revision: r0p0*, 2019.

[11] Express Logic. THREADX RTOS Real-Time Operating System. `https://rtos.com/solutions/threadx/real-time-operating-system/`. [Online; accessed September 2020].

[12] Synopsys. Z01X Functional Safety Assurance. `https://www.synopsys.com/verification/simulation/z01x-functional-safety.html`. [Online; accessed September 2020].

[13] Synopsys. Z01X Simulator. *Safety Verification User Guide*, 2020.

A Appendix

The Appendix contains terminal and screen captures, in an effort to present the developed work in a better way. It aims to provide a means of visualizing the achieved results, in order to better convey the main concepts presented in the previous Sections.

```
07-Sep-2020 13:58:43    # STL-Part MTID= 2 STID= 2 Part Took:  1805 cycles
07-Sep-2020 13:58:43    # STL-Part MTID= 2 STID= 2 Part Took:  1969 cycles
07-Sep-2020 13:59:03    # STL-Part MTID= 2 STID= 2 Part Took:  1972 cycles
07-Sep-2020 13:59:03    #
07-Sep-2020 13:59:03    #  Register: <name>, Readval: <0x...>, Irritval: <0x...>, Currval: <0x...>
07-Sep-2020 13:59:23    #  ------------------------------------------------------------------------
07-Sep-2020 13:59:43    #  ACTLR, 0x0, 0x8018410, 0x8018410 / 0x8018410, 0x8018410 MATCH
07-Sep-2020 13:59:43    #  AFSR, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:00:03    #  AIRCR, 0xFA050000, 0x5FA0020, 0xFA050020 MATCH
07-Sep-2020 14:00:23    #  CCR, 0xB0201, 0xB0201, 0xB0201 / 0xB0000, 0xB0000 MATCH
07-Sep-2020 14:00:42    #  CFSR, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:00:42    #  CPDLPSTATE, 0x202, 0x31, 0x31 / 0x31, 0x31 MATCH
07-Sep-2020 14:01:02    #  CSSELR, 0x0, 0x1, 0x1 / 0x1, 0x1 MATCH
07-Sep-2020 14:01:22    #  DAUTHCTRL, 0x0, 0x106, 0x106 / 0x106, 0x106 MATCH
07-Sep-2020 14:01:42    #  DCRDR, 0x0, 0x2964007F, 0x2964007F / 0x2964007F, 0x2964007F MATCH
07-Sep-2020 14:02:02    #  DCRSR, 0x0, 0x1001E, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:02:02    #  DEBR0, 0x0, 0x80019CBB, 0x80019CBB / 0x80019CBB, 0x80019CBB MATCH
07-Sep-2020 14:02:22    #  DEBR1, 0x0, 0x8000162D, 0x8000162D / 0x8000162D, 0x8000162D MATCH
07-Sep-2020 14:02:42    #  DEMCR, 0x100000, 0x1300450, 0x1300450 / 0x1200450, 0x1200450 MATCH
07-Sep-2020 14:03:02    #  DFSR, 0x0, 0x34, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:03:22    #  DSCSR, 0x0, 0x1, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:03:22    #  EVENTSPR, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:03:42    #  FPCAR, 0x0, 0x2A168710, 0x2A168710 / 0x2A168710, 0x2A168710 MATCH
07-Sep-2020 14:04:02    #  FPCCR, 0xC0000004, 0x9400077E, 0x9400077E / 0x9400077E, 0x9400077E MATCH
07-Sep-2020 14:04:22    #  FP_CTRL, 0x10000080, 0x10000082, 0x10000080 / 0x2, 0x2 MATCH
07-Sep-2020 14:04:42    #  HFSR, 0x0, 0x80000002, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:05:02    #  ICSR, 0x0, 0x42000000, 0x400000 / 0x42000000, 0x42000000 MATCH
07-Sep-2020 14:05:22    #  IEBR0, 0x0, 0x40015B82, 0x40011B82 HW_DEP: ECC not active
07-Sep-2020 14:05:42    #  IEBR1, 0x0, 0x800111C1, 0x800111C1 / 0x800111C1, 0x800111C1 MATCH
07-Sep-2020 14:05:42    #  NSACR, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:06:02    #  NVIC_ISER, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:06:22    #  NVIC_ICER, 0x0, 0x1, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:06:41    #  NVIC_ITNS, 0x0, 0x1, 0x1 / 0x1, 0x1 MATCH
07-Sep-2020 14:06:41    #  RFSR, 0x0, 0x10003, 0x10003 / 0x10003, 0x10003 MATCH
07-Sep-2020 14:07:01    #  SAU_CTRL, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:07:21    #  SAU_RNR, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:07:41    #  SAU_RBAR, 0x0, 0x800BCFC0, 0x800BCFC0 / 0x800BCFC0, 0x800BCFC0 MATCH
07-Sep-2020 14:08:01    #  SAU_RLAR, 0x0, 0x939A82A2, 0x939A82A2 / 0x939A82A2, 0x939A82A2 MATCH
07-Sep-2020 14:08:21    #  SCR, 0x0, 0x18, 0x18 / 0x18, 0x18 MATCH
07-Sep-2020 14:08:41    #  SFAR, 0x0, 0x3A5D49EA, 0x3A5D49EA / 0x3A5D49EA, 0x3A5D49EA MATCH
07-Sep-2020 14:08:41    #  SFSR, 0x0, 0xF, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:09:01    #  SHCSR, 0x0, 0x988, 0x988 / 0x988, 0x988 MATCH
07-Sep-2020 14:09:21    #  STIR, 0x0, 0x0, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:09:21    #  SYST_CSR, 0x0, 0x4, 0x4 / 0x4, 0x4 MATCH
07-Sep-2020 14:09:41    #  SYST_CVR, 0x0, 0x21EB10, 0x0 / 0x0, 0x0 MATCH
07-Sep-2020 14:10:01    #  SYST_RVR, 0x0, 0xFEE716, 0xFEE716 / 0xFEE716, 0xFEE716 MATCH
07-Sep-2020 14:10:13    #  VTOR, 0x0, 0x9835CB80, 0x9835CB80 / 0x9835CB80, 0x9835CB80 MATCH
07-Sep-2020 14:10:13    # ** TEST PASSED OK ** (Time:             2404385)
07-Sep-2020 14:10:13    # ** Note: $finish    : /tmp/sbist/projects/yamin/rtl/shared/../logical/testbench/execution_tb/verilog
07-Sep-2020 14:10:13    #    Time: 2404385 ns  Iteration: 1  Instance: /tbench/genblk1/u_sbist_sim_ctl
07-Sep-2020 14:10:13    # End time: 14:10:13 on Sep 07,2020, Elapsed time: 0:14:14
07-Sep-2020 14:10:13    # Errors: 1, Warnings: 3
07-Sep-2020 14:10:13    2166:# ** TEST PASSED OK ** (Time:             2372475)
07-Sep-2020 14:10:13    4335:# ** TEST PASSED OK ** (Time:             2335745)
07-Sep-2020 14:10:13    6504:# ** TEST PASSED OK ** (Time:             2392405)
07-Sep-2020 14:10:13    8673:# ** TEST PASSED OK ** (Time:             2519465)
07-Sep-2020 14:10:13    10842:# ** TEST PASSED OK ** (Time:             2518965)
07-Sep-2020 14:10:13    13011:# ** TEST PASSED OK ** (Time:             2517975)
07-Sep-2020 14:10:13    15180:# ** TEST PASSED OK ** (Time:             2267505)
07-Sep-2020 14:10:13    17349:# ** TEST PASSED OK ** (Time:             2298685)
07-Sep-2020 14:10:13    19518:# ** TEST PASSED OK ** (Time:             2546905)
07-Sep-2020 14:10:13    21687:# ** TEST PASSED OK ** (Time:             2519385)
07-Sep-2020 14:10:13    23856:# ** TEST PASSED OK ** (Time:             2514415)
07-Sep-2020 14:10:13    26025:# ** TEST PASSED OK ** (Time:             2393715)
07-Sep-2020 14:10:13    28194:# ** TEST PASSED OK ** (Time:             2524725)
07-Sep-2020 14:10:13    30363:# ** TEST PASSED OK ** (Time:             2537045)
07-Sep-2020 14:10:13    32532:# ** TEST PASSED OK ** (Time:             2404385)
07-Sep-2020 14:10:13    STL PASSED
```

Figure 5: Capture of the RTL regression log, after completing 15 iterations. A check at the end reports the number of successful simulations and the corresponding line in the log.


```
08-Sep-2020 04:04:56
08-Sep-2020 04:04:56    -------------------------
08-Sep-2020 04:04:56    Final number of parts ran: 10
08-Sep-2020 04:04:56
08-Sep-2020 04:04:56     Register: <name>, Readval: <0x...>, Irritval: <0x...>, Currval: <0x...>
08-Sep-2020 04:04:56     ---------------------------------------------------------------------
08-Sep-2020 04:04:56     ACTLR, 0x00000000, 0x00018ccc, 0x00018ccc / 0x00018ccc, 0x00018ccc MATCH
08-Sep-2020 04:04:56     AFSR, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     AIRCR, 0xfa050000, 0x05fa0630, 0xfa050630 MATCH
08-Sep-2020 04:04:56     CCR, 0x00000201, 0x000b0701, 0x000b0701 / 0x000b0500, 0x000b0500 MATCH
08-Sep-2020 04:04:56     CFSR, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     CPDLPSTATE, 0x00000000, 0x00000233, 0x00000233 / 0x00000233, 0x00000233 MATCH
08-Sep-2020 04:04:56     CSSELR, 0x00000000, 0x00000001, 0x00000001 / 0x00000001, 0x00000001 MATCH
08-Sep-2020 04:04:56     DAUTHCTRL, 0x00000000, 0x0000000f, 0x0000000f / 0x0000000f, 0x0000000f MATCH
08-Sep-2020 04:04:56     DCRDR, 0x00000000, 0xd330c386, 0xd330c386 / 0xd330c386, 0xd330c386 MATCH
08-Sep-2020 04:04:56     DCRSR, 0x00000000, 0x0001001e, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     DEBR0, 0x00000000, 0x800289e1, 0x00000000 HW_DEP: ECC not active
08-Sep-2020 04:04:56     DEBR1, 0x00000000, 0x0003b984, 0x00000000 HW_DEP: ECC not active
08-Sep-2020 04:04:56     DEMCR, 0x00100000, 0x01300350, 0x01300350 / 0x01200350, 0x01200350 MATCH
08-Sep-2020 04:04:56     DFSR, 0x00000000, 0x00000014, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     DSCSR, 0x00000000, 0x00000002, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     EVENTSPR, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     FPCAR, 0x00000000, 0x58460380, 0x58460380 / 0x58460380, 0x58460380 MATCH
08-Sep-2020 04:04:56     FPCCR, 0xc0000004, 0x940001dc, 0x940001dc / 0x940001dc, 0x940001dc MATCH
08-Sep-2020 04:04:56     FP_CTRL, 0x10000080, 0x10000083, 0x10000080 / 0x00000003, 0x00000003 MATCH
08-Sep-2020 04:04:56     HFSR, 0x00000000, 0x80000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     ICSR, 0x00000000, 0x03000000, 0x00400000 / 0x03000000, 0x03000000 MATCH
08-Sep-2020 04:04:56     IEBR0, 0x00000000, 0x8001299a, 0x00000000 HW_DEP: ECC not active
08-Sep-2020 04:04:56     IEBR1, 0x00000000, 0x000076f4, 0x00000000 HW_DEP: ECC not active
08-Sep-2020 04:04:56     NSACR, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     NVIC_ISER, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     NVIC_ICER, 0x00000000, 0x00000001, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     NVIC_ITNS, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     RFSR, 0x00000000, 0x00010003, 0x00010003 / 0x00010003, 0x00010003 MATCH
08-Sep-2020 04:04:56     SAU_CTRL, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     SAU_RNR, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     SAU_RBAR, 0x00000000, 0x532f1e60, 0x532f1e60 / 0x532f1e60, 0x532f1e60 MATCH
08-Sep-2020 04:04:56     SAU_RLAR, 0x00000000, 0x5b05d023, 0x5b05d023 / 0x5b05d023, 0x5b05d023 MATCH
08-Sep-2020 04:04:56     SCR, 0x00000000, 0x0000001c, 0x0000001c / 0x0000001c, 0x0000001c MATCH
08-Sep-2020 04:04:56     SFAR, 0x00000000, 0xbc38e661, 0xbc38e661 / 0xbc38e661, 0xbc38e661 MATCH
08-Sep-2020 04:04:56     SFSR, 0x00000000, 0x000000af, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     SHCSR, 0x00000000, 0x00010c91, 0x00010c91 / 0x00010c91, 0x00010c91 MATCH
08-Sep-2020 04:04:56     STIR, 0x00000000, 0x00000000, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     SYST_CSR, 0x00000007, 0x00000004, 0x00000004 / 0x00000004, 0x00000004 MATCH
08-Sep-2020 04:04:56     SYST_CVR, 0x0000beab, 0x00756817, 0x00000000 / 0x00000000, 0x00000000 MATCH
08-Sep-2020 04:04:56     SYST_RVR, 0x0000ea5f, 0x0069274b, 0x0069274b / 0x0069274b, 0x0069274b MATCH
08-Sep-2020 04:04:56     VTOR, 0x00000000, 0xb1f0e680, 0xb1f0e680 / 0xb1f0e680, 0xb1f0e680 MATCH
08-Sep-2020 04:04:56    STL PASSED
08-Sep-2020 04:04:56
08-Sep-2020 04:04:56    ######### END #########
08-Sep-2020 04:04:56    tools/annotatetrace -image FM/rtos_0.axf -input FM/tarmac.cpu0.log -output FM/tarmac.cpu0_annotated.log
08-Sep-2020 04:05:00    Reading source info data from 'FM/rtos_0.axf'
08-Sep-2020 04:05:00    make[1]: Leaving directory `/tmp/sbist/projects/yamin'
08-Sep-2020 04:05:00    3962:STL PASSED
08-Sep-2020 04:05:00    7935:STL PASSED
08-Sep-2020 04:05:00    11903:STL PASSED
08-Sep-2020 04:05:00    15877:STL PASSED
08-Sep-2020 04:05:00    19846:STL PASSED
08-Sep-2020 04:05:00    23812:STL PASSED
08-Sep-2020 04:05:00    27780:STL PASSED
08-Sep-2020 04:05:00    31746:STL PASSED
08-Sep-2020 04:05:00    35714:STL PASSED
08-Sep-2020 04:05:00    39680:STL PASSED
08-Sep-2020 04:05:00    43648:STL PASSED
08-Sep-2020 04:05:00    47616:STL PASSED
08-Sep-2020 04:05:00    51582:STL PASSED
08-Sep-2020 04:05:00    55552:STL PASSED
08-Sep-2020 04:05:00    59519:STL PASSED
08-Sep-2020 04:05:00    STL PASSED
```

Figure 6: Capture of the FM regression log, after completing 15 iterations. A check at the end reports the number of successful simulations and the corresponding line in the log.

```
14884 #-------------------------------------------------------------------------------
14885 # Faults listed:       Prime: 0        Total: 0
14886 #-------------------------------------------------------------------------------
14887 # Fault Coverage Summary
14888 #
14889 #                                    Prime             Total
14890 #-------------------------------------------------------------------------------
14891 # Total Faults:                      78065            123637
14892 #
14893 # Dropped Detected                DD    23747  30.42%     38816  31.40%
14894 # Detected $finish/$stop          DF     1030   1.32%      1728   1.40%
14895 # Dropped Potential               PD        0   0.00%         0   0.00%
14896 # Not Detected                    ND    53288  68.26%     83093  67.21%
14897 #
14898 # Untestable Tied                 UT     2082              2133........
14899 # Untestable Blocked              UB     3744              3744........
14900 #
14901 # Faults not detected yet:
14902 #   Not simulated yet:                      0   0.00%         0   0.00%
14903 #   Simulated 1 to 5 times:             54318  69.58%     84821  68.60%
14904 #   Simulated 6 to 10 times:                0   0.00%         0   0.00%
14905 #   Simulated > 10 times:                   0   0.00%         0   0.00%
14906 #
14907 # Faults potentially detected:
14908 #   Possible 1 to 5 times:                  0   0.00%         0   0.00%
14909 #   Possible 6 to 10 times:                 0   0.00%         0   0.00%
14910 #   Possible > 10 times:                    0   0.00%         0   0.00%
14911 #
14912 # Test Coverage                              31.74%            32.79%
14913 # Fault Coverage                             29.53%            31.30%
14914 #-------------------------------------------------------------------------------
```

(a) Screen capture of the permanent coverage report for Cortex-M55 *Revision-r0*.

```
14885 #-------------------------------------------------------------------------------
14886 # Faults listed:       Prime: 0        Total: 0
14887 #-------------------------------------------------------------------------------
14888 # Fault Coverage Summary
14889 #
14890 #                                    Prime             Total
14891 #-------------------------------------------------------------------------------
14892 # Total Faults:                    2723078           2723078
14893 #
14894 # Dropped Detected                DD   164627   6.05%   164627   6.05%
14895 # Detected $finish/$stop          DF    21315   0.78%    21315   0.78%
14896 # Dropped Potential               PD        0   0.00%        0   0.00%
14897 # Illegal Access                  IA   528483  19.41%   528483  19.41%
14898 # Impossible x-state              IX    54372   2.00%    54372   2.00%
14899 # Not Detected                    ND  1954281  71.77%  1954281  71.77%
14900 #
14901 # Untestable Blocked              UB      489               489........
14902 #
14903 # Faults not detected yet:
14904 #   Not simulated yet:                      0   0.00%        0   0.00%
14905 #   Simulated 1 to 5 times:           2558451  93.95%  2558451  93.95%
14906 #   Simulated 6 to 10 times:                0   0.00%        0   0.00%
14907 #   Simulated > 10 times:                   0   0.00%        0   0.00%
14908 #
14909 # Faults potentially detected:
14910 #   Possible 1 to 5 times:                  0   0.00%        0   0.00%
14911 #   Possible 6 to 10 times:                 0   0.00%        0   0.00%
14912 #   Possible > 10 times:                    0   0.00%        0   0.00%
14913 #
14914 # Test Coverage                              6.83%             6.83%
14915 # Fault Coverage                             6.83%             6.83%
14916 #-------------------------------------------------------------------------------
```

(b) Screen capture of the transient coverage report for Cortex-M55 *Revision-r0*, using a $1000ns$ injection interval.

Figure 7: Sample of two coverage reports.

```
# RAM: ----------------------------------------------------------
# RAM: ITCM flat memory model
# RAM: Image :    tests/debugdriver.bin
# RAM: (C) COPYRIGHT 2019 Arm Limited or its affiliates.
# RAM: memory width =          32 bits
# RAM: memory size  =        1024 kB
# RAM: image max    =      262144 lines
# RAM: ----------------------------------------------------------
# RAM: ----------------------------------------------------------
# RAM: D0TCM flat memory model
# RAM: Image :          bin/stl_biu.bin
# RAM: (C) COPYRIGHT 2019 Arm Limited or its affiliates.
# RAM: memory width =          32 bits
# RAM: memory size  =         256 kB
# RAM: ----------------------------------------------------------
# RAM: ----------------------------------------------------------
# RAM: D1TCM flat memory model
# RAM: Image :          bin/stl_biu.bin
# RAM: (C) COPYRIGHT 2019 Arm Limited or its affiliates.
# RAM: memory width =          32 bits
# RAM: memory size  =         256 kB
# RAM: ----------------------------------------------------------
# RAM: ----------------------------------------------------------
# RAM: D2TCM flat memory model
# RAM: Image :          bin/stl_biu.bin
# RAM: (C) COPYRIGHT 2019 Arm Limited or its affiliates.
# RAM: memory width =          32 bits
# RAM: memory size  =         256 kB
# RAM: ----------------------------------------------------------
# RAM: ----------------------------------------------------------
# RAM: D3TCM flat memory model
# RAM: Image :          bin/stl_biu.bin
# RAM: (C) COPYRIGHT 2019 Arm Limited or its affiliates.
# RAM: memory width =          32 bits
# RAM: memory size  =         256 kB
# RAM: ----------------------------------------------------------
# TracePort Logger (c) 2019 Arm Limited or its affiliates.
# STL-Part MTID=        2 STID= 1 Part Took:   953 cycles
# STL-Part MTID=        2 STID= 2 Part Took:   595 cycles
# STL-Part MTID=        2 STID= 3 Part Took:   532 cycles
# STL-Part MTID=        2 STID= 4 Part Took:   992 cycles
# STL-Part MTID=        2 STID= 5 Part Took:  1405 cycles
# ** TEST PASSED OK ** (Time:              162705)
# ** Note: $finish    : /projects/pd/pj02774_yamin/users/konkar01/sbist_trans/
#    Time: 162705 ns  Iteration: 1  Instance: /tbench
# End time: 06:42:43 on Sep 27,2020, Elapsed time: 0:00:36
# Errors: 0, Warnings: 0
```

Figure 8: Terminal capture of STL execution during RTL simulation.