POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Optimization of Test Architecture in RISC-V Based System-on-Chip

In collaboration with:
STMicroelectronics srl

Supervisors
prof. Riccardo Cantoro
prof. Michelangelo Grosso
Iacopo Guglielminetti

Candidate
Mauro Lubrini

Academic Year 2024-2025

Abstract

This thesis optimises test architecture for RISC-V System-on-Chip designs by improving observation coverage for Software-Based Self-Test (SBST) on the open-source CVA6 processor. As silicon now drives appliances, vehicles, smartphones, and medical devices, robust System On Chip (SoC) and thorough testing are essentials. Designs that are easier to test could rise the yield and shorten the time-to-market; periodic online tests executed during normal operation improve reliability and enable at-speed checks. Developed with STMicroelectronics, this work extends an earlier CVA6 study that boosted SBST by inserting observation monitors at random Register Tranfer Level (RTL) points. Here, randomness is replaced with a principled selection of internal signals whose fault effects are masked and cannot reach primary outputs. Synopsys SpyGlass identifies such locations; although usually applied to place scan elements, its suggestions are repurposed as direct strobe points in functional mode. The study evaluates whether SpyGlass-detected nodes improve coverage over random insertion and over having no added points. Four workflows target the Execution Stage of the RISC-V. First, SpyGlass runs at RTL; the design is synthesised; a functional fault simulation then runs while an assembly SBST program exercises the core. Second, after synthesis, SpyGlass analyses the netlist while sequential Automatic Test Pattern Generation (ATPG) patterns are generated; coverage is assessed by comparing Good and Faulty Machines, with selected points routed to extra primary outputs. Third, to isolate point-placement effects, the same ATPG pattern set is reused across SpyGlass, random, and no-point cases. Finally, the Execution Stage is flattened by separating combinational from sequential logic so SpyGlass can assess the combinational portion as a single block; the same simulations are repeated. Results depend on circuit typology. In the purely combinational Aritmetic Logic Unit (ALU), SpyGlass-selected points deliver a clear uplift, with coverage rising by up to 6.58% over random selection. In the unmodified Execution Stage, where sequential and combinational logic interleave, the advantage is negligible and coverage trends overlap. When the stage is flattened to expose its combinational core, the improvement reappears: coverage with SpyGlass-guided points is higher by 18,83% than with random points under the same stimulus. This indicates that SpyGlass is effective on combinational regions, while sequentially rich blocks may require additional measures. In conclusion, the thesis validates a method for selecting observation points to enhance SBST on CVA6. It documents flows at RTL and post-synthesis and clarifies where the method has the most impact: in combinational logic or in designs refactored to make that structure explicit. Combining structural analysis with modest refactoring supports higher at-speed, in-field coverage; modules dominated by state may benefit from additional architectural hooks or alternative stimuli.

Contents

1 Introduction 7
- 1.1 Objectives and motivations . . . 8
- 1.2 Structure of this thesis . . . 8

2 Background 11
- 2.1 Risc-V ISA . . . 11
- 2.2 CVA6 architecture . . . 11
  - 2.2.1 Frontend . . . 14
  - 2.2.2 Issue Stage . . . 17
  - 2.2.3 Instruction Decode Stage (ID) . . . 18
  - 2.2.4 Execute stage . . . 19
  - 2.2.5 Commit Stage . . . 24
  - 2.2.6 CSR file . . . 24
  - 2.2.7 Controller . . . 24
- 2.3 The discipline of Design For Testability (DFT) . . . 25
  - 2.3.1 Design for Testability (DfT) testing techniques . . . 26
  - 2.3.2 Test pattern generation . . . 32
  - 2.3.3 Fault models . . . 36
  - 2.3.4 Fault simulation . . . 38
  - 2.3.5 Design for Testability (DfT) testing metrics . . . 41
- 2.4 Previous work and starting point . . . 42

3 Approaches 45
- 3.1 RTL observation points insertion and SBST based fault simulation workflow 46
  - 3.1.1 Design analysis and optimal test point insertion at RTL level . . . 46
  - 3.1.2 Random test points selection and insertion at RTL level . . . 47
- 3.2 Post-synthesis observation points insertion and ATPG based fault simulation workflow . . . 48
  - 3.2.1 Post-synthesis design analysis and optimal test point insertion . . . 48
  - 3.2.2 Post-synthesis random test points selection and insertion . . . 49
- 3.3 Fault simulation with unique precomputed ATPG test patterns workflow . 50
  - 3.3.1 Fault simulation with optimal test point insertion . . . 50
  - 3.3.2 Fault simulation with random test point insertion . . . 50

4 Implementation 53
4.1 SpyGlass analysis . . . 53
4.2 Observation points insertion at RTL level and SBST based fault sim . . . 58
4.2.1 Synthesis . . . 58
4.2.2 Fault simulation run . . . 67
4.2.3 Random buffer insertion . . . 75
4.3 Post-synthesis observation points insertion and ATPG based fault simulation 79
4.3.1 TestMAX ATPG working principle . . . 80
4.3.2 ATPG Fault simulation . . . 82
4.3.3 ATPG with random test point insertion . . . 85
4.3.4 Hierarchy flattening . . . 90
4.4 Post-synthesis observation points insertion and fixed functional test patterns fault sim . . . 92

5 Results 97
5.1 SBST with observation point inserted at RTL level . . . 98
5.1.1 Observation points on ALU . . . 98
5.1.2 Observation points on Execution Stage . . . 99
5.2 ATPG with observation points inserted at netlist level . . . 100
5.2.1 Observation points on Execution Stage . . . 101
5.2.2 Observation points on ALU . . . 102
5.2.3 Observation points on flattened Execution Stage . . . 103
5.3 Fault simulation with ATPG test pattern, observation point at netlist level 104
5.3.1 Observation points on ALU . . . 105
5.3.2 Observation points on Execution Stage . . . 106
5.3.3 Observation points on flatten Execution Stage . . . 107

6 Conclusion 109
6.1 Results analysis and conclusion . . . 109
6.2 Final conclusions . . . 111
6.3 Future works . . . 112

A Matlab scripts 113
A.1 Graph plots . . . 113
A.2 Data extrapolating functions . . . 122

B General scripts 127
B.1 Reports . . . 127
B.1.1 SpyGlass optimal observation points on Execution Stage . . . 127
B.1.2 Random selection of test points in the netlist . . . 133

Acronyms

ALU Aritmetic Logic Unit

ATE Automated Test Equipment

ATPG Automatic Test Pattern Generation

BGA Ball Grid Array

BHT Brench History Table

BIST Built-In Self-Test

BS Joint Test Action Group

BS Boundary Scan

BTB Branch Targhet Buffer

CMOS Complementary Metal–Oxide–Semiconductor

CPU Central Processing Unit

CSR Control Status Register

CUT Circuit Undet Test

DfT Design for Testability

DRC Design Rule Check

DTLB Data Translation Lookaside Buffer

DUT Device Under Test

ECL Emitter-Coupled Logic

EDA Electronic Design Automation

ELF Executable and Linkable Format

FIFO First In First Out

FM Faulty Machine

FPU Floating Processing Unit

FSM Finite State Machine

FU Functional Unit

GM Good Machine

GUI Graphic User Interface

HDL Hardware Description Language

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IP Intellectual Propriety

IPC Instruction Per Cycle

IQ Instruction Queue

ISA Instruction Set Architecture

JETAG Joint European Test Action Group

LRM Language Reference Manual

LSB Least Significant Bit

LSU Load Store Unit

MMU Memory Management Unit

MSB Most Significant Bit

ODE Output Data Evaluator

OS Operating System

PA Physical Address

PC Program Counter

PCB Printed Circuit Board

PGA Pin Grid Array

PI Primary Inputs

PMP Physical Memory Protection

PO Primary Outputs

PPA Power Performance and Area

PTW Page Table Walker

RAM Random Access Memory

RAS Return Address Stack

RISC Reduced Instruction Set Computer

RTL Register Tranfer Level

SBST Software-Based Self-Test

SFF Scan Flip-Flop

SGDC SpyGlass Design Constraints

SoC System On Chip

STIL Standard Test Interface Language

TAP Test Access Port

TCK Test Clock

TCL Tool Command Language
TDI Test Data Input
TDO Test Data Output
TLB Transition Lookaside Buffer
TMS Test Management Signal
TPG Test Pattern Generator
TRST Test Reset
TTL Transistor Transistor Logic
VA Virtual Address
VIPT Virtually Indexed Physically Tagged

Chapter 1

Introduction

Today's world is powered by tiny components made of silicon. They control everything, from home appliances to cars, smartphones, Internet of Things (IoT) devices, medical equipment, and many more. Nowadays, more than ever, it is important to have reliable and robust SoC capable of working a relentless number of times without having issues: in particular in safety-critical applications like in the automotive or in the medical industry, where a small issue could lead to critical consequences. Ensuring these tiny devices are thoroughly tested before they're released to the market is a crucial part of the integrated circuit design process. Having a design that is easy and highly testable can allow a company to have a higher yield during production and a faster time to market, increasing the income and the profit. Another key aspect in the design of an SoC is the ability to execute periodic online tests run in the background during the processor's normal working activity, so the architecture is automatically tested within itself, without the need for any additional test equipment. This methodology can lead to incredible benefits, starting from the improvement of reliability over time by checking at the processor's activity throughout its lifetime, to the execution of at-speed testing, essential for modern high-speed SoCs. In this context, Design for Testability (DfT) plays a fundamental role, studying new ways to improve testing and the related procedures during the design of an SoC. With circuits becoming increasingly complex, testing becomes more difficult, making the need for more advanced and efficient methods of analysis and testing even more crucial.

Lots of different testing techniques have been developed during the course of these years, but nowadays, more than ever, the need for efficient, low-power, and compact SoCs lead to the development of new methodologies that can reduce the area overhead induced by additional testing elements or modules implemented inside the design. In particular, as the geometries shrink and delay effects become more relevant, an at-speed test is needed, but due to the increment core frequencies of modern SoC, new high-speed interfaces and more expensive test equipment are required. Additionally, for more advanced multi-core architectures, the testing time scales with the number of cores if the test is not executed in parallel. These challenges led the semiconductor companies to find new testing methods. Starting from the 1980s, functional self-testing (also known as SBST) gain more and more interest in the semiconductor industry, allowing to run test programs directly on the device, running them at it's native clock frequency and removing the need of expensive

and cumbersome test equipment. [1]

1.1 Objectives and motivations

This thesis, developed in collaboration with ST-Microelectronics, builds upon a previous work that used a Software-Based Self-Test (SBST) approach to improve the test coverage of an open-source RISC-V architecture developed by ETH Zürich and the University of Bologna called CVA6. Test coverage refers to the computation of detected faults relative to the testable fault count. Given a fault model, which is the mathematical representation and characterization of real defects within a circuit, a list of faults inside the circuit is created. A set of test stimuli is then simulated on the list, proving which faults are controllable or observable by them, i.e., detected by the stimuli. This thesis builds upon previous work that used a SBST approach to functionally simulate the design and by inserting some observation monitors improve the test coverage of the RISC-V architecture. The original study inserted these set of additional observation monitors at random points, mainly on the highest hierarchical modules. In this thesis work, an alternative methodology is proposed and implemented for strategically selecting the optimal locations of these observation points. The coverage results of this new approach will then be compared against the previous random method to determine its effectiveness. To the best of our knowledge, no commercial tools are available for such a purpose. However, a commercial static analysis tool for RTL circuits, Spyglass by Synopsys, was used in this thesis to evaluate its potential in helping to discover the optimal location of observation points. The goal of this thesis is to demonstrate that the points detected by Spyglass are useful to increase the test coverage in functional mode, even though Spyglass is originally intended to suggest optimal locations for scan-enabled flip-flops during scan-chain insertion. The locations pinpointed by Spyglass are then used as additional observation points during fault simulation. This research work aims to validate this methodology for the improvement of the test coverage and its effectiveness over a random insertion of test points.

1.2 Structure of this thesis

- **Chapter 2**: This chapter starts with a brief introduction to the CVA6 architecture to contextualize the device under test. Next, a general overview on different testing techniques, fault models, and fault classifications providing the reader with some fundamental concepts needed to understand the methods and the results of this thesis. In conclusion, a summary of the former thesis work on which the present work builds upon.

- **Chapter 3**: This chapter is dedicated to explaining at a high level the different workflows developed during this thesis work. A general overview, with no specific implementation details, of the different methodologies is proposed. The idea is that

these workflows could be theoretically mapped to any possible working environments and toolchains.

- **Chapter 4**: This chapter contains a detailed description of the different workflows implementation. Some attention has been given to the software tools used during this research, explaining at a high level how they work and how they have been set up to perform the different tasks constituting each methodology. Here are explained the reasoning and the motivations behind each test procedure.
- **Chapter 5**: In this chapter, the reader can find all the results obtained by all the different tests mentioned previously in chapter 3. Comparisons and comments are carried out to provide a better understanding of the results showing the pros and cons of each different approach.
- **Chapter 6**: This final chapter is dedicated to the conclusions on the carried work and results, with some possible proposal on future works that could be undertaken starting from the results obtained.

Chapter 2

Background

2.1 Risc-V ISA

RISC-V is an open and royalty-free standard Instruction Set Architecture (ISA) based on the Reduced Instruction Set Computer (RISC) architecture. Developed in 2010 by the University of Berkeley, RISC-V has experienced an exponential growth in popularity over the course of the last few years. Its modularity, scalability, simplicity and open nature make it the best choice for simple embedded systems, but also for high-performance microprocessors running on mobile, desktop, and servers. In an industry where the main semiconductor companies own their proprietary architectures and Intellectual Propriety (IP) (Such as ARM, MIPS, x86 from Intel), the RISC-V architecture proposes an open source alternative usable both academically and commercially without any royalties. Moreover, RISC-V is strongly supported by a vast community of developers who maintain and extend the ISA through time. [2]

2.2 CVA6 architecture

The CVA6 is a RISC-V single issue, in order Central Processing Unit (CPU). It implements the 64-bit RISC-V ISA and the I, M, A and C (compressed) extensions; furthermore can also fully support a Unix-like operating system. It is an industrial evolution of ARIANE architecture created by ETH Zürich and the University of Bologna. It is written in SystemVerilog and maintained by the OpenHW Group. The CVA6 can be configured as a 32 (RV32) or 64-bit (RV64) processor core, and it implements L1 caches and optional Memory Management Unit (MMU), Physical Memory Protection (PMP), and Floating Processing Unit (FPU). Its flexibility and modularity allows designers to modify the processor to the required Power Performance and Area (PPA) metrics or extend it with additional peripherals which allows the CVA6 to implement new and advanced functionalities. In this thesis work, the 64-bit version of the CVA6 (CV6A_MMU) has been used. This variant of the architecture implements an optional MMU capable of managing efficiently the interfacing with the external Random Access Memory (RAM) memory.

Moreover, it can also handle a 64-bit memory address space and execute the RV64I Base Integer Instruction Set. [3], [4], [5], [6]

The CVA6 core has the following characteristics:

- **Multiple Issue Pipelines:** The CVA6 features a 6-stage pipeline allowing the core to issue continuously in sequence new instructions per clock cycle; so in the case of a sequentially executed program (without any branches), this feature significantly increases the throughput. Nevertheless, in case of a non perfectly sequential program, the CVA6 has some countermeasures against pipe flushing to improve the instruction issuing, by implementing a Brench History Table (BHT).

- **Out-of-Order Execution:** Thanks to this feature, the processor is capable of executing any fetched instruction as soon as the corresponding execution unit is available, without waiting for previous instructions to be committed. This allows the processor to run faster and more efficiently.

- **Instruction and Data Caches:** The processor has two separate caches, one dedicated to the data and the second to the instructions. Exploiting the locality of reference principle, these two caches allows the required data to be much closer to the core, reducing the number of accesses to the main memory and so the overall latency related to that.

- **Optional Hardware Accelerators:** One of the defining features of the RISC-V architectures is their modular design, and the CV6 allows connecting to the core additional hardware accelerator or IP, extending its functionalities to target specific tasks.

Figure 2.1 presents a diagram illustrating the architecture organization.

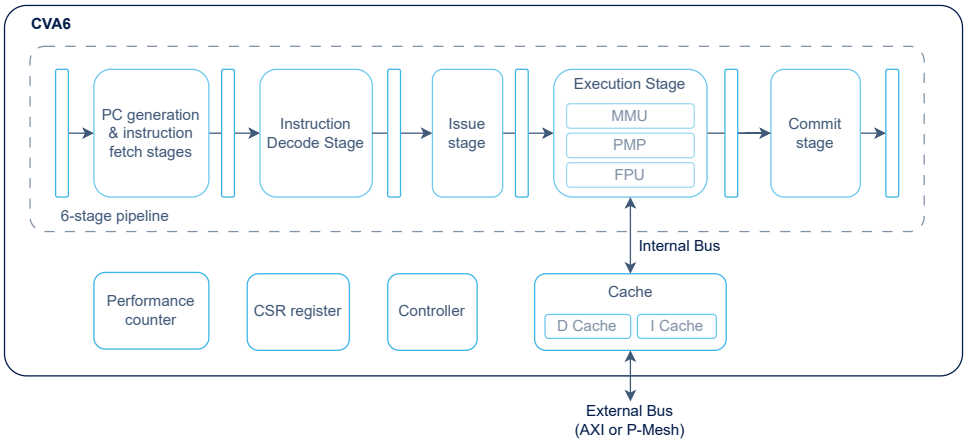


Figure 2.1: CVA6 architecture

- **CVA6 core:** The core of the CVA6 IP containing all the sub-modules of the architecture.
- **L1 write-through cache:** Provides frequently used data to be accessed more rapidly, avoiding frequent memory accesses.
- **FPU:** allows to extend the computational capability with floating-point calculations.
- **MMU:** stands for Memory Management Unit and manages the memory interface and translates virtual addresses used by programs into physical addresses in memory, enabling virtual memory, process isolation, and efficient multitasking.
- **PMP:** stands for Physical Memory Protection, a hardware feature that divides a system's physical memory into regions with configurable permissions to control access.
- **Control Status Register (CSR)**: for system configuration and status reporting.
- **Performance counters:** Additional hardware counters used to detect and analyze the system performance.
- **AXI interface:** Module to manage the communication with external interfaces through AXI protocol.
- **Controller:** Generates and elaborates all the control signals that manage all the different stages and modules of the core.

After introducing how the CVA6 architecture has been conceptualized in its organization, the following sections propose a more detailed analysis of the main modules which compose the core, a quick list is depicted in Figure 2.2.

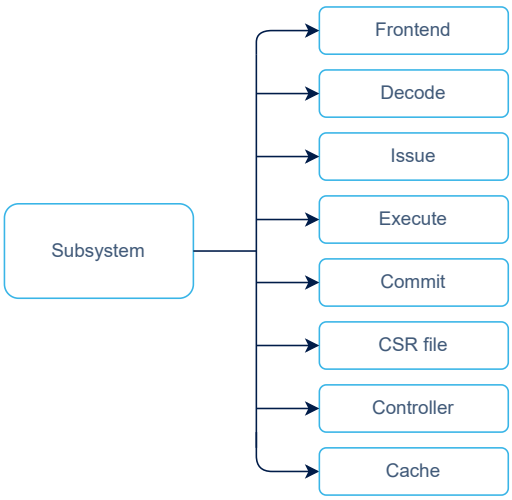


Figure 2.2: CVA6 submodule division

2.2.1 Frontend

The frontend of the CV64A6 processor is the entry point of the pipeline. It is responsible for Program Counter (PC) generation, instruction fetching, and prediction mechanisms that enable efficient instruction supply to the decode stage. Designed as a modular subsystem, it integrates branch prediction units, instruction buffering, and scanning logic to maximize throughput while maintaining low power and area efficiency. It provides the essential capabilities for speculative execution, compressed instruction support, and robust pipeline feeding even in the presence of frequent control-flow changes.

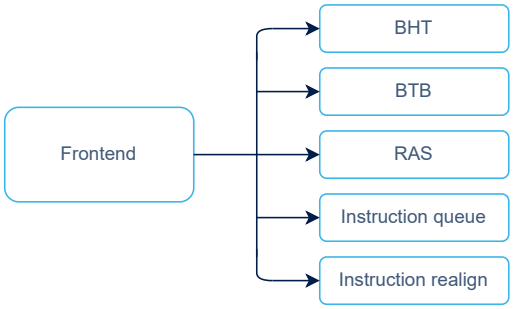


Figure 2.3: Frontend units division

Breaking down the main subsection of this module, we have:

- **PC Generation and Control Flow Management:** this subsection creates the PC value that points to the next instruction. This component strongly interacts with the prediction units to speculate on the possible branch that will be taken afterwards.
- **Branch Prediction Subsystem:** this subsection is where all the speculative and predictive actions are taken. In this unit is possible to find the Branch History Table BHT, Branch Targhet Buffer (BTB), Return Address Stack (RAS).
- **Instruction Handling:** this section uses the Instruction Realign unit to align the fetched instruction words and uses the Instruction Queue module to store the decoded instructions while they wait to be issued from the issue stage.

Branch History Table (BHT)

The BHT is a PC direction predictor and it predicts whether a conditional branch is taken or not taken, keeping the fetch stage supplied with the most likely branch path; target addresses are provided elsewhere by the BTB or the RAS. This separation keeps the predictor fast enough for per-cycle use across all instruction "slots" in a fetch bundle.

Each BHT entry is a tiny record comprising a valid bit and a two-bit saturating counter. The prediction rule is quite simple: if the entry is valid and the counter's most significant bit is one, the branch is predicted to be taken; if the Most Significant Bit (MSB) is zero,

it is predicted not to be taken. The second bit acts as hysteresis, preventing the predictor from flipping direction on a single anomalous outcome; this rule is better depicted with a simple diagram in Figure 2.4. The conventional encoding applies: 00 represents strongly not taken, 01 weakly not taken, 10 weakly taken, and 11 strongly taken. Because only the MSB participates in the decision, the predictor offers a clean one-cycle datapath while still damping noise via the Least Significant Bit (LSB).

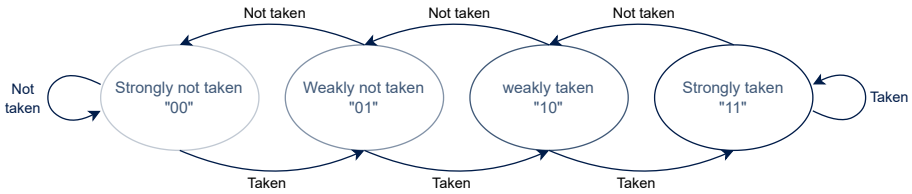


Figure 2.4: BHT rule diagram

Updates arrive from the backend when a branch resolves. Provided updates are marked valid, the BHT locates the corresponding entry using the resolving instruction's PC, sets the entry valid, and adjusts the counter towards the observed outcome. A taken result increments the counter unless it is already 11; a not-taken result decrements it unless it is already 00. This is the standard "two-bit" learner, proven to work well on loop bodies and other locally correlated control-flow patterns typical of embedded workloads.

Branch Target Buffer (BTB)

The BTB is responsible for predicting the destination address of control-flow instructions so that the processor can redirect instruction fetch without waiting for branch resolution in the execute stage. While the BHT decides whether a branch is likely to be taken, the BTB provides the actual target address, ensuring that the frontend can immediately fetch from the predicted path. Together with the RAS), which specializes in handling function returns, the BTB forms the target half of the prediction mechanism. Each BTB entry contains a valid bit, a tag, and a target address. On lookup, if the tag matches and the entry is valid, the BTB produces a hit and provides the predicted target address for that instruction slot. This address is then passed to the PC logic, which priorities targets as follows: for returns, the RAS takes precedence; otherwise, a BTB hit supplies the target; if neither applies, the frontend proceeds sequentially. In this way the BTB integrates cleanly with both the BHT and the RAS: the BHT determines whether a branch is taken, the BTB supplies the target if the branch is indeed taken, and the RAS handles the special case of function returns. Updates to the BTB occur when control-flow instructions resolve in the backend. If the instruction has a stable and determinate target, such as a taken conditional branch, a direct jump, or a function call, the BTB entry is written with the instruction's PC tag and the resolved target. The valid bit is set, and the entry becomes eligible for future predictions. On reset, the entire BTB is invalidated: all valid bits are cleared and no predictions will be made until new control-flow instructions are resolved and populate the table. On a flush, which may be triggered by a misprediction or an

exception, the table is also cleared to prevent stale targets from being used. This design keeps recovery simple and predictable, with the cost that the predictor must warm up again after a flush, repopulating entries as instructions retire.

Return Address Stack (RAS)

The RAS is a specialized branch prediction structure dedicated to handling function calls and returns. Unlike the Branch History Table BHT, which predicts the direction of conditional branches, or the BTB, which stores general branch and jump targets, the RAS is designed to exploit the highly structured nature of subroutine calls and returns. In RISC-V, a call instruction such as `jal` or `jalr` saves the return address (the address of the instruction following the call) in the link register (`ra`), and a return instruction typically jumps back to the value stored in the `ra` register. So the RAS maintains a small stack of predicted return addresses, allowing the frontend to supply the correct target for a return instruction without waiting for the backend to resolve it. In other words, the RAS operates as a last-in, first-out stack. When the frontend fetches a call instruction, it pushes the predicted return address (current PC + instruction length) onto the stack. When the frontend later encounters a return instruction, it pops the top entry and uses it as the predicted target address. This mechanism is extremely effective for nested and recursive subroutines, since each call has a matching return, and the addresses can be predicted with high accuracy. In practice, the RAS dramatically reduces the cost of returns, which otherwise would often be mispredicted by a generic BTB due to their indirect nature. Integration with the rest of the frontend is straightforward. The RAS supplies its prediction only when the instruction being fetched is identified as a return. For other control-flow types, the BTB or sequential PC logic is used instead. In the prioritization logic of the frontend's next-PC selection, the RAS target typically has the highest priority for return instructions, ensuring that it overrides any BTB entry that might also be present for the same slot. The RAS is updated speculatively at fetch time (on calls) and corrected if necessary when the backend resolves the instruction and signals mispredictions or flushes. On a pipeline flush or reset, the stack pointer and contents are cleared to prevent stale return addresses from being used after recovery.

Instruction Realign

The Instruction Realign unit in the frontend is the first stage that processes raw instruction words arriving from memory and prepares them for the rest of the fetch pipeline. Its role is essential because it takes the fixed-width words delivered by the instruction fetch interface and it realigns them into a continuous stream of correctly aligned RISC-V instructions. This becomes particularly important in a processor like the CV64A6 that supports both standard 32-bit instructions and 16-bit compressed instructions, which may cross natural word boundaries. Without a realignment stage, instructions could be misinterpreted, misaligned, or split incorrectly across fetch beats, leading to incorrect decoding. At the interface level, the realign unit accepts instruction words fetched from memory, typically 32-bit aligned and delivered in bundles depending on the frontend configuration. The fetch stage itself is agnostic to whether the instructions are compressed or not: it simply supplies

the raw instruction words indexed by the current program counter. The challenge is that compressed (RVC) instructions are 16 bits wide, and they can occur in any position relative to 32-bit alignment. For example, a 16-bit instruction might be followed immediately by another 16-bit instruction, together fitting into a single 32-bit word, or it might be followed by a full 32-bit instruction, which then crosses the boundary between two words. The job of the realign logic is to stitch these pieces together into clean instruction packets, so that each downstream unit sees a valid and complete instruction starting at the correct program counter.

Instruction Queue (IQ)

The Instruction Queue (IQ) is the final buffering stage before instructions are handed over to the decode and issue logic. Its purpose is to decouple the timing of instruction fetch from the demands of the backend pipeline, creating elasticity between the two halves of the processor. By providing a small buffer that sits between the frontend and the decode stage, the queue allows the frontend to continue fetching instructions even when the backend is temporarily stalled, and conversely ensures that the backend can keep working on buffered instructions when the fetch side experiences a hiccup, such as a cache miss or a redirection. Architecturally, the Instruction Queue is a First In First Out (FIFO) structure sized to hold a handful of instructions. It supports both the 32-bit RISC-V standard instructions and the 16-bit compressed instructions. Because the frontend fetches instructions in aligned bundles, the queue must be able to accept multiple instructions in a single cycle and then present them to the backend in order and one at a time. This means that the queue's internal organization is more sophisticated than a simple one-word FIFO: it must track instruction boundaries, manage validity for each slot, and keep instructions aligned correctly for the decode logic.

2.2.2 Issue Stage

The Issue Stage is situated between decode and execute. Each cycle it accepts a decoded micro-op, checks for hazards and Functional Unit (FU) availability, fetches or forwards source operands, and fires exactly one instruction to the selected FU. At every issued instruction it hands a transaction ID that is going to be used later when any results or flags are returned. The issue stage interacts with different functional units independently; so it means that it has to check for their readiness each time an instructions request the use of one of them. It also receive and store their write-back data unconditionally, while the instructions are issued in order (contrary to the write back which can happen out-of-order). The issue stage is composed of two main units: a scoreboard and the issue read operands.

Scoreboard

The scoreboard is essentially a FIFO buffer with one read and one write port, both paired with a valid and an acknowledge signal. The instruction decode stage writes directly to

the scoreboard only if the latter is not already full. The commit stage instead looks for already ended instructions and updates the architectural state.

Issue Read Operands

The Read-Operands unit runs in the same cycle as issue and decides whether the head instruction can be run while sourcing the required operands. It consults the scoreboard's clobber maps to detect RAW/WAW hazards and uses a forward-first policy: for each source (rs1, rs2) prefers a forwarded value from in-flight or just-written results; only if no newer value exists does it read the register file. WAW is blocked unless the commit stage writes the same rd in the same cycle of issue.

2.2.3 Instruction Decode Stage (ID)

The instruction decode has the purpose of distinguishing instructions from the data that the instruction fetch stage outputs. This unit decodes the instructions and sends them to the issue stage. With the introduction of compressed instructions, the Instruction Decode also has to realign instructions that fall off the word boundaries: for example, if a compressed instruction (16-bit wide) is followed by a non-compressed instruction (32-bit wide), the latter is split in half across two consecutive words. In case of this event, the instruction is fully decoded only after two memory accesses; therefore, a properly sized fetch FIFO is needed. Furthermore, the information coming from the branch predictor is used to output the correct instruction and send it to the issue stage.

Compressed Decoder

The Compressed Decoder is used to decompress all compressed instructions (RV32C). It is composed of a purely combinational circuitry which takes a 16-bit compressed instruction and expands it to the 32-bit version.

Decoder

The decoder takes the raw instruction (also 16-bit compressed instructions) data and decodes it accordingly, transforming it into a scoreboard entry that the issue stage can consume:

- PC: PC of instruction;
- FU: functional unit to use;
- OP: operation to perform in each functional unit;
- RS1: register source address 1;
- RS2: register source address 2;
- RD: register destination address;

- Result: for unfinished instructions, this field also holds the immediate;
- Valid: is the result valid;
- Use I Immediate: should we use the immediate as operand b;
- Use Z Immediate: use zimm as operand a;
- Use PC: set if we need to use the PC as operand a, PC from exception;
- Exception: exception has occurred;
- Branch predict: branch predict scoreboard data structure;
- Is compressed: signals a compressed instructions, we need this information at the commit stage if we want jump accordingly e.g.: +4, +2;

The scoreboard entry is fundamental since it controls operand selection, dispatch and execution.

2.2.4 Execute stage

The execution stage (EX) of the CV6 pipeline is where instructions are actually executed. After an instruction has been issued, the execution stage steers it to its appropriate functional unit, waits for the result (in case the unit is multi-cycle), and feeds back both data and control outcomes to the rest of the core. In CVA6 this stage is deliberately modular, and its subdivision is described in Figure 2.5: an integer ALU for single-cycle integer ops; a Branch Unit that resolves control flow and produces the definitive next PC; a CSR buffer that arbitrates access to Control and Status Registers; hardware multipliers and dividers for the M instruction extension; a Load/Store path (with an explicit load unit) that computes effective addresses and talks to memory; an floating point unit and a custom-extension endpoint (CV-X-IF) that lets designers attach accelerators.

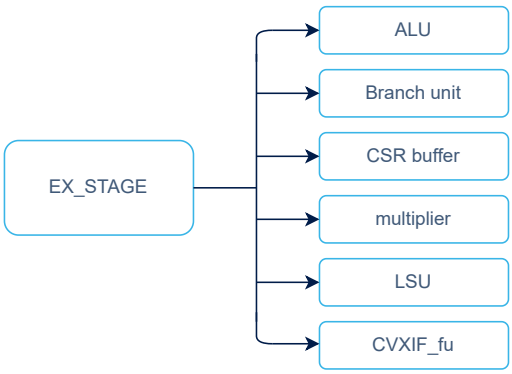


Figure 2.5: Execution Stage (EX) units division

Aritmetic Logic Unit (ALU)

The ALU performs all base integer arithmetic and logical operations (subtraction, addition, shifts and comparisons) in a single cycle so no sequential logic is present inside it. It takes two operands plus control bits from decode, and produces one result per cycle for write-back unless the instruction is routed to a different functional unit, or a custom hardware accelerator extension). Operand A is selected from the integer register file or, for PC-relative instructions, from the current PC forwarded into the execution stage; operand B is chosen between the second source register and one of several immediate decoded from the executed instruction (I/U/B/J/S forms), so the same adder can serve ADD/ADDI as well as AUIPC and the link-address computation for JAL/JALR. With a front-end muxing, the ALU covers ADD/SUB, logical AND/OR/XOR, shifts (SLL/SRL/SRA), set-less-than (SLT/SLTU), and the immediate variants; LUI/AUIPC are realized by feeding the 20-bit U-immediate into the datapath and selecting the appropriate source on the other side.

Multiplier

The multiplier in the execution stage implements the RV64M multiply subset and delivers the product bits required by the four architectural instructions MUL, MULH, MULHSU, and MULHU. Functionally, it accepts two XLEN-wide operands from the issue/execute wrapper, forms a full 2×XLEN-bit product, and selects either the lower XLEN bits (for MUL) or the upper XLEN bits with the correct sign (for MULH/MULHSU/MULHU) before write-back. In the CV64a6 configuration (64-bit) this is a 128-bit internal product; the low 64 bits implement MUL, while the high 64 bits implement the three "H" variants according to signed×signed, signed×unsigned, and unsigned×unsigned semantics defined by the RISC-V M extension. The multiplier is paired in the EX stage alongside the divider. The decode/issue logic routes an instruction to the mult wrapper, which steers it to the multiplier for product or the divider for division computations. The wrapper exposes a single result and a "busy/ready" status to the execution stage so that the pipeline stalls only while a mult/div operation is in flight since multiplication is performed in two cycles while the division is performed by a simple serial divider which needs 64 cycles in the worst case.

Floating point unit (FPU)

The FPU provides IEEE-754 compliant computation for the RISC-V floating-point extensions: it can handle single precision (RVF), double precision (RVD), and, when configured, reduced-precision formats such as FP16, FP16ALT (bfloat16-like), and FP8. At the interface, the FPU receives three operands (A, B, C), a rounding mode, format selectors, and a decoded operation. The FPU accepts these via a ready/valid handshake and returns a result together with the five exception flags (inexact, underflow, overflow, divide-by-zero, invalid). A small two-state Finite State Machine (FSM) in the wrapper ("READY/STALL") holds inputs when the new result is not ready, so the rest of the pipeline only ever sees a single clean signal.

Branch unit

The branch unit in the CVA6 execution stage is the block that finalizes control-flow decisions, computes the correct next PC, and reconciles the frontend speculation. It takes as inputs the current instruction's PC, a flag indicating whether that instruction is 16-bit (compressed) or 32-bit, the decoded operation and operands, the ALU's boolean branch-condition result, and the predictor's per-slot speculation (control-flow type and predicted target). From these, it builds two addresses: `next_pc`, which is simply `PC + instr_len` (advancing by 2 or 4 bytes depending on whether the instruction is compressed), and a potential prospect `target_adress`, which is formed by adding a base and an immediate. With those addresses available, the branch unit resolves speculation and drives redirection. For conditional branches, it uses the ALU's comparison result to choose between the computed `target_address` (taken) and `next_pc` (not taken), marks that decision as the resolved outcome, and sets the output control-flow type to "branch" so the BHT can be trained. A misprediction is flagged precisely when the ALU's outcome disagrees with what the predictor classified as a branch in this slot. For register-indirect jumps `jalr`, it declares a misprediction if the predictor did not mark this slot as a control transfer or if the predictor's target does not match the computed `target_address`. In that case the control-flow type is reported as "jump-register" so the BTB can learn the site.

Load store unit

The Load Store Unit (LSU) is an essential block which houses different functional units responsible for interfacing with the data memory. In particular, it houses the MMU which contains the Data Translation Lookaside Buffer (DTLB), the Page Table Walker (PTW), and the Transition Lookaside Buffer (TLB). To solve as fast as possible any possible TLB misses, the LSU arbitrates the accesses to the data memory between loads and stores; it prioritizes the PTW lookup. The LSU issues load requests as soon as possible, while the stores' requests are kept back as long as the scoreboard does not decide to issue a commit signal: having a single commit point, the processor is constrained to behave with an in-order commit. This way of behaving is referred to as "posted-store" because the store request is kept waiting in the store queue for the commit signal to be high and the memory interface not being in use. So during a load procedure, the LSU has to check the store buffer for potential aliasing and in the case it finds uncommitted data, it should stall to wait for the data commitment.

This means that the LSU should follow these rules:

- Two loads to the same address are allowed and are returned in the order they were issued.
- Two stores to the same address are allowed, and the scoreboard will issue and store them in order. If the commit signal is off, the stores happen in the store buffer; otherwise, they are done inside the main memory.
- A store followed by a load to the same address is permitted only if the store has already been committed (marked as committed in the store buffer). Otherwise, the LSU stalls until the scoreboard commits the instruction.

In case of misaligned accesses (so words not aligned with 64-bit, 32-bit, or 16-bit boundaries), a misaligned exception is thrown, and the exception handler will resolve the load or store.

The design of the LSU is split in 6 main units, Figure 2.6:

1. LSU Bypass
2. D$ Arbiter
3. Load Unit
4. Store Unit
5. MMU (including TLBs and PTW)
6. Non-blocking data cache

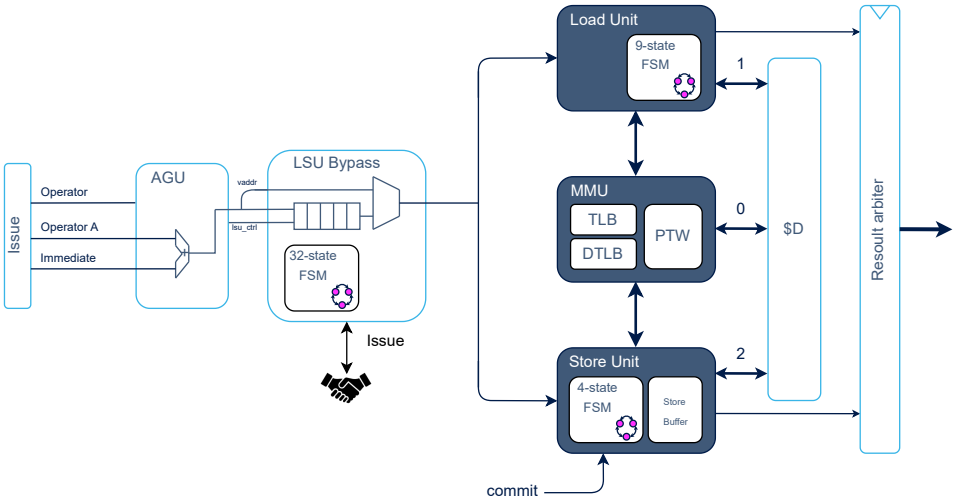


Figure 2.6: Load Store Unit (LSU) scheme

LSU Bypass

The LSU bypass is a small auxiliary block that shields the issue stage from the LSU's late "go" or "no-go" decision. Because the LSU talks to relatively slow SRAMs and must perform sequential, time consuming work, address generation, address translation, and store-buffer check for potential aliasing. The true readiness of a load/store is often known only very late due to TLB misses and store aliasing, which are the most common causes. Driving that late ready signal directly back to the issue stage would create a long critical path and slow the whole core, and the issue stage cannot stall once it has issued an instruction. The bypass solves this by interposing a tiny FIFO so that issuing can hand off one more request even if the LSU is not yet ready, allowing the LSU's ready signal to

be delayed by one cycle, easing the timing. When the LSU is free, requests are bypassed straight through.

Load Unit

The load unit issues loads as early as possible (since, contrary to storing, issuing does not alter the architectural state), but before that it checks the store buffer to avoid reading out of date data (that would create aliasing). For speed, it compares only the 12-bit page offset of the load against all pending stores: this keeps the check on 12 bits (rather then 64) and it is performed before translation since at that point Physical Address (PA) and Virtual Address (VA) share the same offset. If any offset matches, the load stalls until the store drains its content. This unit also performs address translation and uses a Virtually Indexed Physically Tagged (VIPT) D$ access scheme, in this way tag resolution reduce the number of cycles needed for load accesses. On a TLB miss, the load may need to kill its in-flight D$ request to let the PTW proceed, since a blocking cache can hold the port; so implementing a non-blocking cache would solve this problem.

Store Unit

The store unit manages all stores by simply calculating the target address and setting the appropriate byte enable bits. Other than that, it also performs address translation and interacts with the load unit to communicate if any store waiting in its buffer matches any load request.

Store buffer buffer

The store buffer tracks all stores using two queues: a speculative queue for not-yet-committed stores and a commit (non-speculative) queue for committed ones. On a flush, only the commit queue is preserved; the speculative queue is fully cleared. Each queue has its own full flag: the speculative full flag blocks the LSU bypass module (blocking new LSU requests), while the commit full flag stalls the commit stage. When the commit stage retires a store, it asserts the signal `lsu_commit`, moving that entry from the speculative queue to the commit queue. The commit queue, as soon as a store has been committed, it tries to commit the oldest store in the memory, cache grants permitted. Address handling is physical in the commit queue (translation is final by commit time). Speculative entries may have unresolved or stale translations; if translation structures change (e.g., TLB shootdown), the pipeline flushes, which also clears the speculative queue, ensuring only correctly translated stores reach the memory system.

Memory Management Unit (MMU)

The MMU performs virtual-to-physical translation and access control for both instruction fetch and data. Internally, it has an ITLB, a DTLB, and a shared hardware page-table walker (HPTW). There are two separate main paths in the MMU: the Instruction fetch stage and the LSU path. The instruction fetch path accepts virtual addresses: if address translation is disable, it passes straight to the I$, otherwise, it delays the I$ request until

a valid TLB translation (ITLB hit is combinational), or it returns a page/access fault. In the case of faults, the fetch path uses a small exception FIFO to return exceptions with valid responses. The data path (LSU side) is a response-request interface managed with a handshake protocol: the load/store units request translation and, on a DTLB hit, get the result one cycle later (added for timing). Since the D-cache is VIPT, this extra cycle does not hurt Instruction Per Cycle (IPC), but it makes the memory request more cumbersome since any possible exceptions could abort memory access. In case of a load exception, the load unit is responsible to kill the memory request sent a clock cycle earlier.

CSR buffer

The CSR buffer module holds temporarily the CSR address at which the instruction is going to read/write. As the CSR instruction alters the processor architectural state, so this instruction has to be buffered until the commit stage decides the best time instant to execute the instruction.

2.2.5 Commit Stage

The commit stage is the last stage in the pipeline. It takes incoming retiring instructions and updates the architectural state. It does this by writing in the CSR register, committing stores, and writing back data to update the architectural state. Other than the retiring instructions, the commit stage also manages different exceptions that arise from three different sources: the first exception can come from any of the previous four pipeline stages, the second exception can come from during commit, and the third one from an interrupt (Interrupts are considered only during the commit stage, ensuring a precise and reliable event) Lastly, the commit stage controls the overall stalling of the processor by blocking any other instruction commitment generating a back-pressure in the pipeline.

2.2.6 CSR file

The Control and Status Register (CSR) subsystem in the RISC-V core implementation. CSRs are special-purpose registers that control processor behavior, manage exceptions and interrupts, store privilege levels, and configure the memory management unit. [6]

2.2.7 Controller

The controller in CVA6 is the small, centralized unit that decides when the pipeline must be flushed or halted and who “owns” the next PC. From the backend it watches for branch mispredicts, exceptions, returns from exception, and PC requests. From the architectural side it observes CSR-triggered flushes and halts and on a mispredict it performs a light recovery dropping only unissued scoreboard entries.

2.3 The discipline of Design For Testability (DFT)

Design for Testability (DfT) is a discipline that aims to develop and implement techniques tailored to add and improve testability features to hardware designs. This is an essential aspect of the design process of a device because it can help to build a chip in which defects, even the most isolated ones, more easily detected, reducing effort, time and cost in testing the architecture. In practice, DfT group together architectural choices, design rules, and dedicated on-chip structures (e.g., scan chains, test points, built-in self-test controllers, boundary-scan cells, and embedded monitors) that increase controllability and observability while keeping functional impact and overhead within acceptable limits. DfT therefore could shorten the time spent in debugging procedures, improve yield, and reduce overall time-to-market by enabling efficient production screening and systematic diagnosis. [7], [8]

Rudimentary DfT techniques were developed starting in the 1940s–50s, when engineers came up with ways to probe voltages and currents on internal nodes in analogue computers. As digital computers quickly developed, DfT techniques evolved in new directions, introducing additional circuitry or physical probing that allowed much easier control and observation (controllability/observability) of a design's internal state. By the late 1970s–80s, scan design had become standard practice, enabling ATPG to reach deep sequential logic through scan chains instead of relying on complex functional sequences. In the 1990s, Built-In Self-Test (BIST), boundary-scan access for board-level interconnect test, and at-speed structural testing using transition and path-delay fault models became very popular and have remained widely used. More recently, SBST and Cell-Aware fault-modelling methodologies have grown in popularity, complementing conventional stuck-at and transition models by focusing on realistic defect mechanisms and, in the case of SBST, exploiting software-driven test programs to exercise hardware at speed. Today, more than ever, the dramatic increase in the number of Integrated Circuit (IC)s produced and their rapid growth in architectural and physical complexity have pushed traditional testability techniques to their limits, making them unable to satisfy modern industry demands on cost, quality, safety, and performance. This has led to the need for more advanced and efficient methodologies to test and detect any possible faulty chips produced, while also enabling faster diagnosis and yield improvement. Although design complexity has grown exponentially and the manufacturing cost per transistor has decreased over time, the testing cost per transistor has actually increased, making the testing phase a substantial portion of the total cost of production of a device. So discovering new and more efficient testing techniques, while improving their effectiveness in detecting and diagnosing faulty devices, has become a central topic in the semiconductor industry. Within this context, newer approaches such as SBST aim to reduce the need for additional test-specific hardware by leveraging on-chip processors and microcontrollers to execute carefully designed test programs. This enables at-speed testing using the native clock signal, avoiding the over-stimulation of the architecture, making test procedures faster, more efficient, and less expensive (no additional test-specific hardware is needed) than classical methodologies such as scan chains.[1]

In modern design flows, a strong focus has been placed on Electronic Design Automation (EDA) software that implements advanced DfT techniques and algorithms to analyze,

insert, and verify sophisticated test structures in complex designs. Contemporary tools automate scan insertion and compression, power-aware pattern generation. They sustain high defect coverage and actionable diagnostics while containing test time and data volumes, and they provide a scalable solution as devices continue to integrate more cores, memories, high-speed interfaces, and heterogeneous IP.

Two different approaches can be followed in testing an architecture: the first being the structural testing, which focuses on the analysis of the Device Under Test (DUT) internal structure; it mainly focuses on detecting internal physical defects and anomalies, adopting their corresponding fault model, and it makes large use of DfT techniques to ease and improve the accuracy and efficiency of the internal analysis. In opposition to that, functional testing tries to adopt alternative methodologies to verify the DUT by testing the logic functionalities, trying to detect, without the use of DfT techniques (like scan-chains analysis), any anomalies inside the architecture. The two approaches can be complementary, as one does not exclude the other. For example, a structural test using scan chains can detect faults that otherwise, with a functional test running the ISA of the microprocessor, would never be excited; on the other hand, running a functional test can allow for the detection of logic faults or timing issues, something that a structural analysis could not detect.

2.3.1 DfT testing techniques

In the DfT discipline, various methodologies can be employed to enhance a design's test coverage. These techniques, developed over time, vary in effectiveness, applicability, and their impact on PPA metrics. The introduction of additional hardware components, for example, may increase area and power consumption, and in some cases—such as scan-chain methods—can also affect timing. Often, these methodologies are tailored to specific circuits or scenarios, resulting in varying degrees of efficiency, speed, and reliability when testing an architecture. The following sections describe three of the most widely used techniques.

Scan based

Scan-based technique controls and observes the design's internal state by forcing or sampling values at specifically selected internal nodes. It does this by adding (or converting existing) flip-flops into multiplexed Scan Flip-Flop (SFF) and linking them in series to form a scan chain, which acts as a sort of shift register that can be loaded with a given input pattern or it can be unloaded to read the internal state of the logic. The basic block that constitutes a scan chain is the SFF, which is a modified version of the standard D flip-flop, but capable of receiving the input data from two different paths (Figure 2.7).

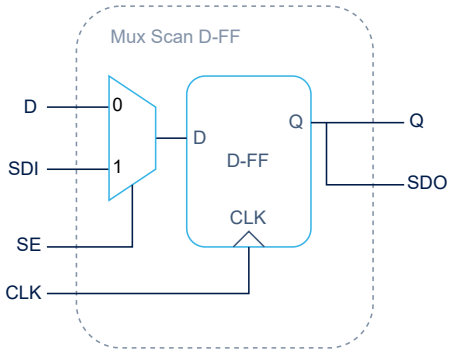


Figure 2.7: Multiplexed Scan Flip-Flop

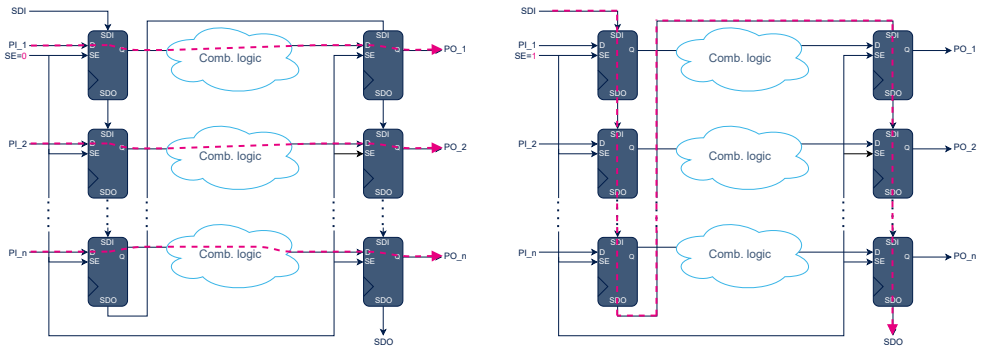
The cell itself has the following ports:

- **Data (DI):** functional input from the upstream combinational logic;
- **Scan Data In (SDI):** serial input which takes the data from the preceding SFF; for the first SFF of the chain, the data arrives from the external scan-in stream.
- **Scan Enable (SE):** it drives the internal multiplexer and selects which input is captured on the clock edge, switching between DI during capture mode or SDI during scan mode;
- **Output (Q):** functional output to the downstream combinational logic; also reflects the captured/shifted value;
- **Scan Data Out (SDO):** serial output (typically the same value as Q or a buffered replica) feeding the next scan SFF; for the last SFF, this is connected to the scan-out stream.

Scan chain has two operating modes:

- **Normal Mode:** This is used to operate the circuit in functional mode, and it is activated when the Scan Enable signal is disabled (SE=0). In this case, the serial connection between the scan elements is interrupted, and the data entering the SFFs come from the functional inputs (Figure 2.8a).
- **Scan Mode:** This mode is used to load or unload the chain and is activated by asserting the Scan Enable signal (SE=1), which detaches the functional inputs/outputs from the internal logic and establish the serial connections between each scan cell. In this way, the SFF works in series, taking the data from the previous SFF and feeding it to the following (Figure 2.8b).

The main idea is to control internal nodes by preparing a vector of ones or zeros (mainly ATPG is used for this purpose) which, by enabling the SE signal (SE=1), is shifted inside



(a) Scan chain in normal mode, $SE = 0$ (b) Scan chain in scan mode, $SE = 1$

Figure 2.8: Scan chains working modes

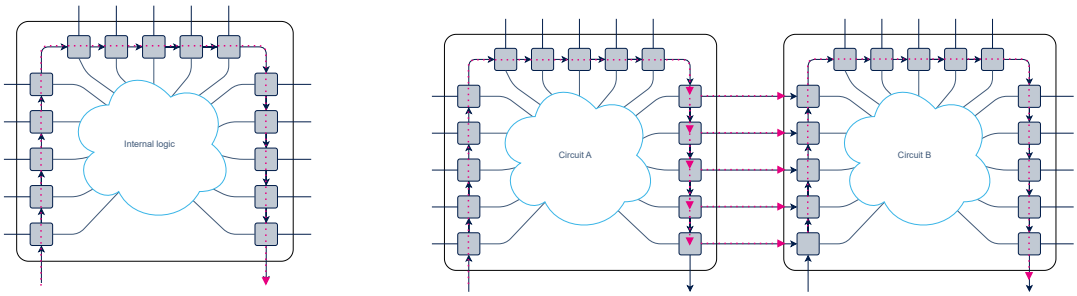
the scan chain from the scan-in input; this procedure can take several clock cycles, as many as the number of scan elements in the chain. Once the entire chain has been filled, the Scan Enable signal is disabled (SE=0) and the circuit starts to operate in the standard mode, but with the internal nodes set by the values of the formerly shifted pattern. To read the internal state, the procedure is the same: by enabling the SE signal (SE=1), the internal nodes of the logic are sampled by the SFFs, and the values are sent out serially to the scan-out stream, ready for analysis.

This DfT technique is very versatile, easy to implement, and has a low pin overhead, allowing for an easy control/observation of the internal state of a circuit for debug purposes or for sequential circuits testing in a combinational manner. On the contrary, the addition of scan elements increases the area overhead, and the multiplexers introduce additional delay to the signal path. [8]

Boundary scan

Boundary Scan (BS) is a standard developed during the 80s, started under the name of Joint European Test Action Group (JETAG), then later changed in Joint Test Action Group (BS) (with the addition of American companies), and finally in 1990, the Institute of Electrical and Electronics Engineers (IEEE) formalized it with the IEEE 1149.1 standard. BS was shared between different IC manufacturers and provides an efficient and unique way to test the devices directly on the Printed Circuit Board (PCB), allowing to access to their pins without the need for physical contact with ATE probes (which, with modern packaging like Ball Grid Array (BGA), Pin Grid Array (PGA), is often very difficult to establish). BS supports both single device or IC interconnections testing: the first can be done by shift in the values to apply the circuit Primary Inputs (PI), then to validate the results, the Primary Outputs (PO) are latched and shifted out from the chain (Figure 2.9a); the latter, instead, involves scanning into circuit A the values to apply at the interconnections, then circuit B latches the values received from A and finally B scans

out the values (Figure 2.9b). Other than the two "test mode" , the BS can also act in "normal mode", allowing for the normal behavior of the device, receiving/retrieving the data from the PI and PO. [8]



(a) Single chip test

(b) Interconnection test

Figure 2.9: Scan chains working modes

Figure 2.10 shows the general architecture of a chip that implements BS. The boundary scan structure is composed of different components, starting from the Test Access Port (TAP), which is the interface of the BS which contains the following signals:

- **Test Data Input (TDI)**: it is used to load serially the data and the instructions to the BS;
- **Test Data Output (TDO)**: it is a three-state output used to serially unload the content of the BS register or the internal registers;
- **Test Clock (TCK)**: independent from the system clock, at the rising edge it samples the input signals Test Management Signal (TMS) and TDI, while at the falling edge it enables the output signal TDO;
- **Test Reset (TRST)**: optional signal used to asynchronously reset the TAP controller.

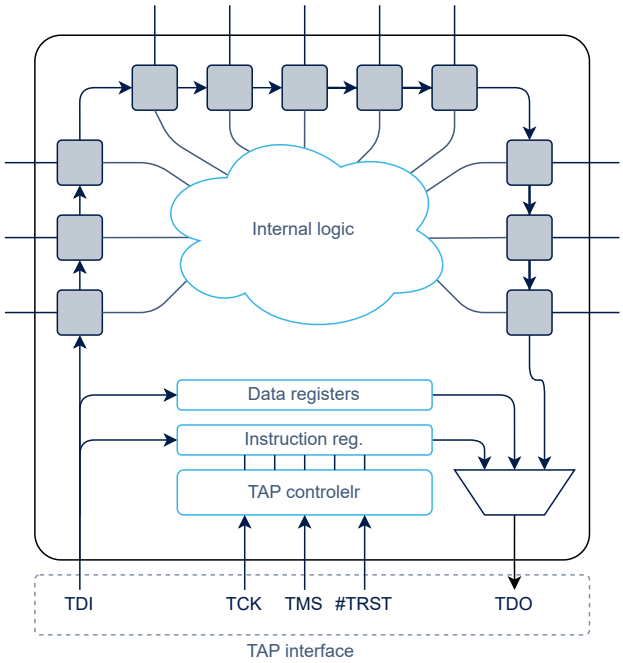


Figure 2.10: Boundary Scan (BS) internal architecture

The remaining components of the BS are the Instruction Register, which stores the instructions that define which actions that the BS system should perform and the data registers, a set of registers of which two mandatory (Boundary Scan Register and the Bypass Register), and others optional like the Device ID registers which stores information about the device or any additional user defined registers. Then an important component in the BS architecture is the TAP controller, which is composed of a finite state machine that reads the input signals coming from the TAP interface and the instruction stored inside the Instruction Register, and manages accordingly the BS system. Two different operative modes are possible depending on the uploaded instruction:

- **Non-invasive modes:** the BS does not influence, by any means, the normal behavior of the device.

- **Pin-permission modes:** the BS acts on the internal logic, forcing the values of the I/O pins which, in that case, are detached from the rest of the system.

The "non-invasive" supported instructions are:

* `bypass`: is a mandatory instruction coded with all ones (11...11) which disables all the registers except the bypass one which creates a one-bit connection between TDI and TDO. This instruction allows mainly used to test only one IC at a time;

- ❋ `idcode`: this instructions is optional and is used to access the ID Register to read the identification code of the IC. It allows to automatically understand the devices mounted on an unknown board;

- ❋ `usercode`: this is an optional instruction used to connect the ID register to TDI and TDO and it allow to load a user defined code;

- ❋ `ecidecode`: this is an optional instruction which allows to read the serial number of the IC;

- ❋ `sample/preload`: is a mandatory instructions which connects the BS register to TDI and TDO without detaching the internal logic from it, so in this way the register respectively capture the values of the I/O pins during the normal IC activity or it can preload the data into the BS cells.

The "pin-permission" supported instructions are:

- ❋ `extest`: is a mandatory instruction coded with all zeros (00...00) which is used to isolate the ICs from the rest of the system in order to test the interconnections on the PCB through the values loaded in the BS cells;

- ❋ `intest`: this is a very common instruction used to connect the BS register to TDI and TDO in order to perform a low speed static test of the internal logic (only if the chip supports single-step operations);

- ❋ `runbist`: it allows the execution of a self-test (BIST) producing a Go/Nogo value;

- ❋ `highz`: it forces all the outputs and bidirectional pins to high impedance for allowing external Automated Test Equipment (ATE) forcing values on them;

- ❋ `clamp`: it clamps the output to the values stored int the BS register.

Built In Self Test (BIST)

Built-in self-test (BIST) is a DfT technique that has been developed to overcome the increasing cost of ATE and the long testing time associated with ATPG. The idea is to embed the testing equipment inside the device, making the test procedure more efficient with a lower of cost, since no expensive ATEs are needed, and a faster testing time, since at-speed testing are possible. This technique also aims to improve the quality and effectiveness of the tests. BIST can be used as a final test at the end of the chip manufacturing, and during the normal operation of a device, allowing for performing in-field tests during its lifetime. Besides its advantages, BIST can lead to an increase in area and power consumption since additional modules have to be implemented.

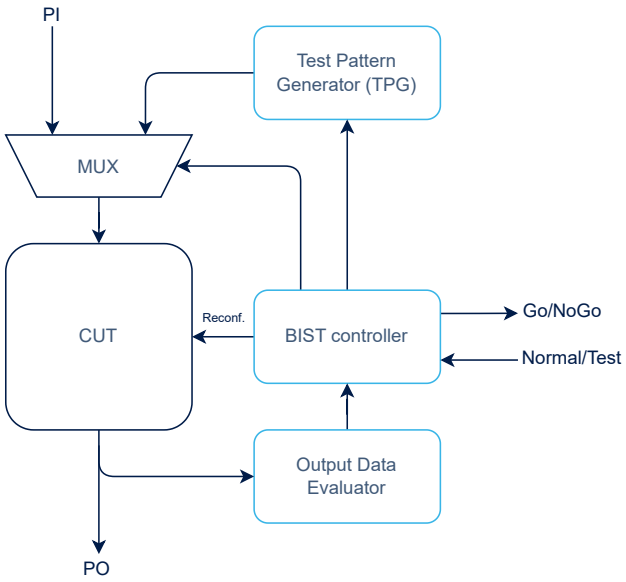


Figure 2.11: BIST general scheme

Figure 2.11 shows the main working scheme, which is composed of several elements: A multiplexer is used to switch between the PI and the test patterns generated by an appropriate module called Test Pattern Generator (TPG). The Circuit Undet Test (CUT) generates the results, which are sent to the PO and to the Output Data Evaluator (ODE), whose role is to compare the results of the CUT with the expected values. The BIST controller then takes the feedback provided by the ODE and manages the behavior of the whole testing architecture: it re-configures the CUT to improve the controllability or the observability of the device, then it controls the TPG module to manage the patterns generation accordingly to the ODE feedback, and finally it generates the Go/NoGo signal which is used to send to the external environment the CUT working status: "Go" means that the CUT is working correctly, while "NoGo" means that a failure has been detected. A signal called Normal/Test that comes from the outside sets the working modality of the BIST structure, activating the normal operating mode or the test mode. During test mode, the CUT works using the patterns generated by the TPG, and its outputs are analyzed by the ODE to detect any failures. On the contrary, while working in normal mode, the CUT takes the inputs from the PI and returns the outputs on the PO. [8]

2.3.2 Test pattern generation

Generating a set of test patterns that deeply exercises the design is a crucial aspect of the testing process. Indeed, depending on how effectively the test stimulus can excite the various faults in the design, a higher or lower coverage can be obtained. So it is fundamental to generate the patterns so that they can extensively simulate all the parts of a system. These patterns not only have to excite the faults, but they also have to

propagate the faults to the PO so that they can be observed, so the justification of the downstream logic should also be taken into account during the patterns generation. Sometimes, certain parts of the design result in hard-to-test locations, since the vectors used to stimulate the circuit hardly succeed in controlling or observing the faults in that particular sites; to overcome this issue, it is possible to directly control or observe those faults by inserting some additional elements like scan chains or more simply a direct connection (test point).

Test patterns can be produced either randomly or by following certain criteria, for example, by using specific algorithms. In the following sections, two of the most common methods for generating test patterns are discussed.

Automatic Test Pattern Generation (ATPG)

Figure 2.12 shows a general scheme depicting the basic ATPG flow: The Fault Manager takes the design description and generates a list of faults, collapsing the equivalent ones. Then the fault list is sent to the Test Pattern Generator, which analyzing the topology of the circuit from the design description files, it generates a set of test patterns targeting the faults present in the list.

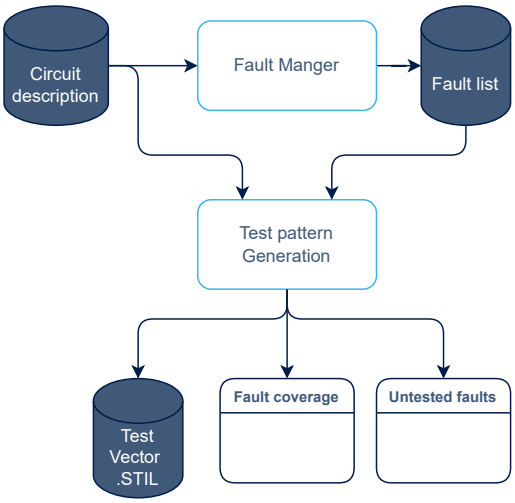


Figure 2.12: ATPG generic scheme

The test vector generation starts by selecting a fault in the fault list and generating a test pattern (or multiple, depending on the design and on the type of fault). Then a fault simulation is performed, and if the pattern is able to detect the selected fault, then that fault is dropped from the fault list and a new fault is selected; otherwise, the fault is kept and a new pattern is generated up until either the fault is detected or an exit condition is reached; in that case the fault is labeled as "aborted". In general faults can be labeled in different ways:

- **Untestable:** it is proven that due to the layout of the circuit, the fault cannot be controlled or observed, so its dropped from the fault list;

- **Tested:** at least one pattern was able to detect it, and once this happens, the fault is dropped from the fault list;

- **Aborted:** this happens when, after a certain number of trials, some threshold metric is reached (CPU time, memory space, number of iterations). In this case, no test pattern was capable of detecting the fault, and its presence is kept in the fault list.

Sometimes ATPG algorithms can optimize the test vector list by exploiting don't care values or running a second campaign of fault simulations with the generated patterns, but in the opposite order, eliminating in this way the ones that do not provide any coverage improvement. The algorithms that generate the test vectors are different and they can change accordingly to the type of circuit (combinational, sequential, memories), the fault model (stuck-at, bridge, delay), or the type of simulation strategies (topological ATPG, which exploits the structural knowledge of the circuit, or functional ATPG, which on the contrary exploits the functional knowledge). Generating a pattern for a single stuck-at fault in a purely combinatorial circuit is considered to be an NP problem, while for sequential ones the complexity increases exponentially, and even if some studies have been carried out on this topic, the ATPG effectiveness on sequential logic is still limited to very small circuits.

To approach ATPG, different methodologies can be undertaken:

- **Exhaustive testing:** it generates all the possible input combinations, and is recommended only for small circuits with a limited number of inputs. This is very effective but can also be quite time-consuming.

- **Pseudo-exhaustive testing:** involves dividing the circuit into smaller sub-modules, each of which is then subjected to exhaustive testing individually.

- **Algebraic methods:** starting from the Boolean function of the faulty and the good machine, supposing that at least one output between the two should differ, by combining the two functions, the input vectors that generates that discrepancy are retrieved. The result should be a Boolean function capable of representing all the test vectors that activate that fault. Even if this is a complete method, the memory required to store those Boolean functions is often too big.

- **Topological method:** is the most used approach in commercial ATPG tools, and it consists of two main steps. First is the excitation phase, in which a fault is forced in its location and, through logic backtracking, the input pattern that excites that fault is determined. The second is the observation phase, where the fault is propagated along all possible paths and, using backtracking, the input patterns that enable its propagation are identified. The most common algorithms used for this methodology are the D-Algorithm, the PODEM and the FAN algorithm. [8]

ATPG is exceptional in generating test patterns capable of testing combinational circuits, but it suffers from a higher degree of complexity when used to generate a set of patterns to test sequential circuits.

In particular, in case the ATPG algorithm has to exercise a fault state machine, the patterns generated have to take care of three things:

- initialized the FSM to a known state;
- excite the fault;
- propagate the fault to the PO in order to be detected.

Two main methods are adopted to perform ATPG on a sequential circuit: the one first is called the "Time-Frame expansion" method, which consists of replicating the combinational part of a sequential circuit as many times as the number of test vectors needed to test a given single stuck-at fault. Then, the same fault is placed in each replicated block, and a test for the multiple stuck-at faults is performed using combinational ATPG. The second approach is the "Simulation-based" method, which instead tries to run a sequential ATPG across the circuit. Under the assumption of working with a synchronous circuit, two possible cases exist: the first one is a cycle-free circuit with no feedback loop between the flip-flops, and this is the simplest case, where the complexity for the test pattern generation targeting a single fault can take a maximum of $(sequential\_depth)+1$ time frames. In case of a circuit that has feedback loops between the different flip-flops, working with a 9-valued[1] logic, it could take up to 9^{Nff} time frames (Nff represents the number of flip-flops); in this case, the complexity increases exponentially with the number of flip-flops. [9]

Software-Based Self-Test (SBST)

One very effective methodology for functionally testing the architecture is the Software-based Self-Test (SBST). This approach involves the use of a test code, which is first loaded inside the microprocessor's program memory, then executed, and finally the results are stored inside the internal data memory, ready to be either verified from the outside or directly analyzed by the microprocessor itself. SBST provides many benefits since it enables the architecture to self-test, enabling online and at-speed testing, ensuring a higher reliability of the system and a more accurate analysis executed under normal working conditions. The downsides of this technique is the difficulty in designing a test program capable of extensively and deeply exercising the design, which can take into account all the possible exceptions and corner cases; indeed, a bad program could lead to bad coverage performances. Another drawback is that each SBST test program is tailored to a specific architecture, so it cannot be easily reused for other designs. The SBST test program can be either loaded once into the internal processor's flash memory, ensuring that the test program is always ready to be executed at any time to perform a periodic check of the system for reliability purposes, or it can be loaded at every test execution

[1] 9-valued logic consists in: 0, 1, 1/0, 0/1, 1/X, X/1, 0/X, X/0, X.

in the RAM, but this is done mainly for final manufacturing tests. The execution of this online test can be triggered in different ways. One possibility is to run the test at the system reset, or power on; it can also be triggered by an interrupt or an exception, either raised internally or externally or, in case it is present, the Operating System (OS) can managed the testing scheduling, performing a cyclic and periodic test of the device. [8]

2.3.3 Fault models

Many different fault models have been developed over the years. Those models aim to simplify the testing process by modeling real physical faults[2] in abstract logical ones. The most used one is the Stuck-At fault model. But there are many more, each one specifically tailored to test and detect a specific type of fault, like short or bridge model, delay fault model, or cell-aware test model. In the following sections, some of the most common fault models are analyzed in details.

Stuck-at model

Within all the possible fault models, Stuck-At is one of the most common ones: it models faults assuming that the value at a specific node of the circuit is either fixed to one (stuck-at-1) or zero (Stuck-at-0), independently of the signal value that drives it. In Figure 2.13, an example is shown where the inverter output port is fixed at 1 (stuck-at-1) even if the value at its input is 1. To test a stuck-at fault, two things have to be done, exciting the fault and observing it:

1. To excite a fault, the input test vector should induce a value at the fault location that is opposite with respect to the stuck-at. In Figure 2.13, it is possible to see how by setting the value 1 to the input a that induces a 0 at the output of the inverter (location of a stuck-at-1 fault), generating a discrepancy between the faulty and good circuits.

2. To observe a fault, the input test vector should justify all the rest of the logic not affected by the fault, so that the latter can propagate to at least one PO to be detected, so that at the end, by comparing the faulty and the good circuit results, a difference is observed. In the example of Figure 2.13, the test vector `X111` allows for propagating the s-a-1 to the output U.

[2]For physical defects, we refer to any physical imperfections within the IC, such as a metal drop that short-circuits two lines, a defect on the mask that results in an incorrect photo-impression on the wafer or a different doping concentration that leads to a transistor malfunction.

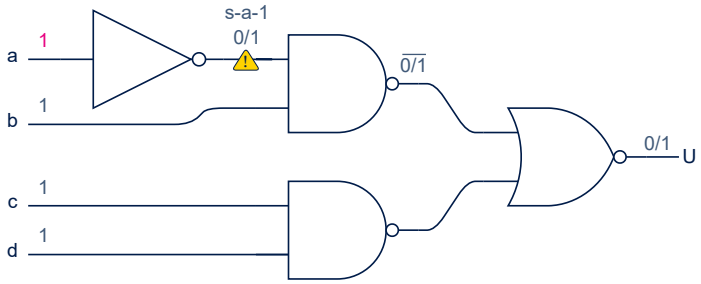


Figure 2.13: Stuck-at-1 example

Faults that cannot be either controlled or observed by any possible input test vector are labeled as "unstable", and they can be considered "safe" since the system outputs are unaffected by their existence so they can be dropped from the fault list. This type of fault is usually obtained from redundant logic and the detection of them can be made by looking at the topology of the circuit, finding gates that are either not reachable from PI, not connected to PO, or nodes connected to fixed values.

Another aspect to take into consideration is the fault equivalence, which can be exploited to collapse the fault list, reducing in that way the number of test patterns that are needed to test all the contained faults. There are two types of equivalence: the test equivalence and the functional equivalence. [8]

- Two faults a and b are said to be "test equivalent" if all the test vectors that test a also test b, and vice versa.
- Two faults a and b are said to be "functional equivalent" if the faulty function of A returns the same result as the faulty function of B.

Short (Bridge) model

Short (or Bridge) model is used to mock-up possible shorts that can happens in the interconnections of the circuit. Depending on the used technology and sometimes on the downstream logic from the fault location, the value imposed by this fault can change. For example, working with a Transistor Transistor Logic (TTL) logic (which is a zero-dominant logic), a short that bridges two connections will force a one only if both lines have a high value; otherwise, a zero. This can be modeled with an AND gate at the fault location. On the contrary, working with Emitter-Coupled Logic (ECL) (a dominant technology), the fault can be modeled with an OR port. With Complementary Metal–Oxide–Semiconductor (CMOS) technology, the imposed value depends on the final state of the downstream circuit, so it cannot be defined in advance.

In general, given a circuit with N nodes, the number of possible bridge faults is:

$$\#F_{bridge} = \frac{N(N-2)}{2} \tag{2.1}$$

So this fault model is quite heavy for large circuits. In those cases, the best approach is to divide the circuits into subpart and then analyze the most critical ones.[8]

Delay fault model

The Delay fault model is used to detect defects that are critical for the timing characteristics of the circuit. These defects are usually produced by process variations, which lead to variation of the delays of the combinational paths of the circuit, increasing the chance of misses in the sampling of the signal or unstable outputs due to unmet timing constraints. These faults happen when the systems run at their normal working frequency, so to test these defects, an at-speed test must be performed.[8]

Cell-aware test model (CAT)

While the majority of fault models act on the inputs/outputs of the cells, the cell-aware test model (CAT) focuses on identifying possible defects inside the physical structure of the cell. This is a more complex approach to testing since it involves analog simulations to characterize all the cells of the library, and for every faults inside the cells, a list of inputs controls and outputs observable values, that are able to test that specific fault, are created. [8]

2.3.4 Fault simulation

Fault simulation is a procedure that enables designers to evaluate the testability of a design. A fault list containing all possible detectable faults is created on the basis of the design layout and the selected fault model. Then, these hypothetical faults are injected inside the model of the faulty DUT, to perform the fault simulation on a given collection of test patterns in order to see whether the injected faults are detected at the circuit outputs. To quantify the effectiveness of this simulation, the test coverage metric is computed at the end by considering the percentage of detected faults over the total number of all possible detectable faults. The goal is to increase as much as possible the test coverage by detecting the highest number of faults possible.

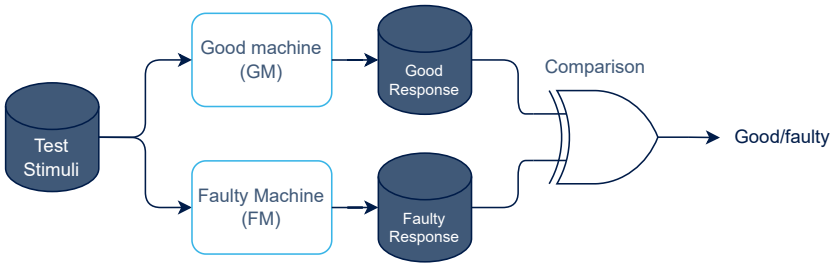


Figure 2.14: Basic test principle scheme

Figure 2.14 depicts the basic flow of a fault simulation campaign. The process begins

with the creation of a set of test patterns to exercise the DUT; these patterns may be generated using various methods, using various techniques like ATPG, random, or pseudo-random approaches. The patterns are then applied to the two models of the circuit under test: the Good Machine (GM), representing the fault-free design, producing the good response, and the Faulty Machine (FM), which is the fault-injected design producing the faulty response. After running a simulation on both machines, the respective results are collected and compared to see if any discrepancies are present; if this is the case, it means that the test patterns used to simulate both models are capable of detecting the injected faults so these are labeled as detected. On the contrary, in case no patterns produced a difference in the two models' responses, the injected faults are labeled as undetected; in this case, further measures have to be implemented trying to detect as much as possible of those faults, maybe by generating a new set of test patterns or by introducing additional control or observation points.

Fault classification

Faults can be labeled in different ways accordingly to characteristic, observability, controllability, and position inside the circuit. Five main classes of faults exist, and each one has its own peculiarities.

DT - Detected: these are faults that have been identified as "hard detected". They are divided into two subcategories:

- **Detected by Simulations (DS):** are those faults that are hard to detect by simulations of the ATPG patterns, and at least one of them causes the fault to be placed in that class and retained.
- **Detected by Implications (DI):** are faults detected by implication analysis, and they can come from:
  - Faults on pins in the scan chain path.
  - Faults on ungated circuitry that connect to the shift clock lines of scan chains or set/reset lines of scan cells.

PT - Possibly Detected: are divided into two subcategories.

- **ATPG unstable, Possibly detected (AP):** these faults are simulated in the faulty machine with "X" rather than "1" or "0", and no ATPG patterns are capable of detecting them, so these faults are removed from the active fault list;
- **Not analyzed, Possibly detected (NP):** are faults whose analysis is not complete, or it cannot be proven that the fault is always simulated with an "X", but it is still possible that some patterns are capable of detecting them.

UD - Undetectable: this class includes faults that cannot be detected under any conditions. Since they have no logical effects on the circuit behavior, these faults are not inserted in the fault list. There are different subcategories of this class:

- **Undetectable Unused (UU):** refers to faults that are located in points that have no connectivity to any external observable point. Usually, in the creation of the simulation model, the unused circuitry that produces these faults is removed;
- **Undetectable Unobservable (UO):** similar to the UU fault, this is located on unused gates with fanout;
- **Undetectable Tied (UT):** this fault is located on pins that are tied to the same value as the fault;
- **Undetectable Blocked (UB):** this fault is blocked from its propagation to observable outputs due to a tie logic.

AU - ATPG Untestable: this set of faults can neither be hard detected from ATPG patterns nor proved to be redundant. Since these faults have the potential to cause failures, they are considered the same as untested faults.

- **ATPG Untestable, Not Detected (AN):** these faults are removed from the active fault list, so there is no opportunity for them to possibly be detected. Usually, these condition happens because:
  - Faults are untestable due to certain types of constraints.
  - Faults are detected through sequential patterns.
  - Faults require an unresolvable state to be detected.

ND - Not Detected: In this class, the faults are not being controlled nor observed yet, but there is a chance that by increasing the ATPG effort, these results would be labeled in a different category.

- **Not Controlled (ND):** this is the initial fault state for every fault at the beginning of the ATPG process. In this case, no pattern has been capable of controlling the fault state yet.
- **Not Observed (NO):** even though the fault site is controllable, no ATPG pattern has been capable to observe the fault yet.

Standard Test Interface Language Standard

In fault simulation often runs on a set of precomputed input test patters, that are simulated to exercise the DUT. These test vectors can be generated in multiple ways using different approaches and algorithms. Once they are created they are often exported in a standard format called Standard Test Interface Language (STIL) which is an IEEE standard designed to carry digital test information between test-generation tools (e.g., ATPG) and ATE. It represents everything needed to define manufacturing-time digital tests. Patterns are expressed as sequences of cycled waveform executed over time; tools can emit STIL directly or use it as an intermediate, tool-agnostic format. This standard has been widely adopted, primarily due to its flexibility and extensibility, as it can describe complex constructs and be easily translated to reconstruct data. It is also a very

portable format, allowing for high-volume vector transport in a small form factor. STIL files are built from top-level blocks with a "define-before-use" rule and a simple domain (named-block) referencing model. Untitled blocks are global, while named blocks must be referenced to use their contents. The spec also details the expected ordering to make references resolvable.

A STIL file is composed of different sections: the first statement is `STIL` followed by the language version used; then it follows the `header{}`, which contains optional metadata about the file. In our case, it includes information on the file's name, the simulation date, the uncollapsed stuck fault summary report, and a list of all the added primary output ports. Next, the `signal{}` section includes the list of all the primary I/O signals of the DUT module, followed by the `SignalGroups{ }` in which multiple signals are grouped and defined as an ordered set of signals to be referenced in subsequent operations. Usually, signals belonging to buses are grouped, and groups may be global or domain-scoped and require explicit reference when domained. The `Timing { WaveformTable ... }` is a block in which "WaveformTables" are defined. Each WaveformTable describes the waveform to be applied to each signal in a vector. The `PatternBurst{ }` block defines a collection of pattern names to be executed sequentially. All patterns defined in a single `PatternBurst` are executed under a similar context, the context being defined by the subsequent `PatternExec statement`. The `PatternExec{}` block defines how `PatternBurst` and timing information is assembled to create the set of tests to execute. Finally, the `pattern_data{}` block is defined, which constitutes the bulk of data in the STIL data set, and is generally processed one-vector-at-a time. Here, the different test patterns generated during ATPG are applied to the primary input ports "`V { "_pi" = ...}`" and the results are read from the primary output ports "`V { "_po" = ...}`"[3].[10]

2.3.5 DfT testing metrics

There are different metrics that can be used to quantify the effectiveness of a DfT technique, such as: fault coverage (FC), test coverage, test time (TT), test cost (TC).

Fault Coverage is defined as the number of faults detected by the test over the total number of possible faults in the system and it refers to a given fault model (e.g., Stack-at).

$$\text{Fault Coverage} = \frac{\text{Detected faults}}{\text{Total faults}} \tag{2.2}$$

Test Coverage is defined as the number of detected faults over the total number of detectable faults. This metric quantifies how effectively a given DfT technique increases the number of detected faults relative to those detectable in the specific test run." This is the metric used to evaluate the effectiveness of the proposed methodology during this thesis work.

$$\text{Test Coverage} = \frac{\text{Detected faults}}{\text{Detectable faults}} \tag{2.3}$$

[3]V = Vector statement. It applies one test cycle and sets the WaveformChars for the listed signals.

In particular for software like TMAX from Synopsys the test coverage is computed in this way:

$$TC = \frac{DT + (PT \times PT_{credit})}{All\_faults \cdot UD \cdot (AN \times AU_{credit})} \tag{2.4}$$

Where DT represents the number of detected faults, PT is the number of possibly detected faults, UD is the number of undetectable faults, while AN is the number of untestable not-detected faults[4]; PT_{credit} and AU_{credit} are some weights that by default are set respectively to 0.5 and 0.[11]

Test Time refers to the time needed to perform a test run, and is defined as the number of the test patterns (n_{tp}) times the time each of them takes to be tested (t_p).

$$TT = n_{tp} * t_p \tag{2.5}$$

Test Cost refers to the total cost for a given test, which is defined as the sum of the costs of: test equipment (C_{equip}), test development (C_{dev}) and test process ($C_{process}$).

$$TC = C_{equip} + C_{dev} + C_{process} \tag{2.6}$$

2.4 Previous work and starting point

As outlined in the introduction, this thesis is a sequel to the work of a previous student who developed an automated workflow for inserting observation monitors at the RTL level to improve observation coverage in SBST. In that workflow, the designer compiles a `.json` file, listing all the monitors to be instantiated, declaring their type and precise location in the RTL design. A bash script then parses the `.json` and injects the corresponding modules into the SystemVerilog sources, producing a design ready for synthesis and simulation. While this approach brought modest gains of only a few percentage points in observation coverage, it lacked a systematic method for deciding where monitors should be placed. The placement was left to the designer's intuition, which often could potentially led to sub-optimal choices and limited improvements in test coverage.

The central aim of the present thesis is to address this gap by identifying hard-to-observe regions of the design in a systematic way and then merging the existing insertion flow to insert them at RTL level. More in detail, the thesis investigates the use of a software from Synopsys called "Spyglass" to highlight low-observability nodes, electing candidate insertion points, and selects a minimal set of locations that is expected to maximize coverage improvements for a given area/overhead budget. Once these sites are pinpointed, the established `.json`-driven workflow should be used to instantiate the monitors at RTL level. Since this is a novel approach, a series of tests and analyses should be performed to validate the effectiveness of this methodology and ensure that it actually brings benefits and improvements in test coverage. If the results of this analysis are positive, the final objective would be to implement the already well-tested flow, created

[4]For more detail see Section:2.3.4

by the former student, and improve it by exploiting the information obtained from the SpyGlass analysis to insert a more accurate way of the set of monitors used to observe the internal states of the design. This thesis work, which will be described in the following chapters, will primarily focus on conducting a series of tests to validate the effectiveness of this methodology.

In summary, this thesis aims to validate the proposed approach by evaluating that the use of Spyglass reliably identifies hard-to-observe points within the design and by quantifying their incremental contribution to the observation metric. The effectiveness is evaluated by comparing the observation coverage obtained with this methodology against the one obtained by a random insertion of the same set of observation points inside the design.

Chapter 3

Approaches

This thesis work tries to find a methodology for strategically selecting the optimal locations to insert these observation points, preferably on the RTL description of the circuit. The optimal points to be observed are internal signals showing logical errors, which cannot be propagated to the primary outputs (PO) due to some layer of logic that obstructs or hides the faults. Two different approaches have been studied and compared under different conditions. The first one exploits the use of an EDA tool to perform a static circuit analysis and identifies the most critical points to observe, while the second one involves a random selection of internal points. In this chapter, a high-level overview of the different approaches investigated in this thesis is presented. Since this section provides only a general description of the steps involved, the workflows here described could, in principle, be applied in any environment using alternative software offering comparable functionalities. In this thesis work, tools from the Synopsys suite were employed, and their specific implementations are discussed in detail in Chapter 4, while in this chapter, the workflows are outlined without reference to any particular software.

In this thesis work, has been decided to perform fault simulations in functional mode, by applying a set of functional test stimuli at the primary input of the DUT and observing the system response from the PO. Input test vectors can be created in different ways, from ATPG algorithms to assembly test programs run directly on the architecture; no scan-based approaches are planned in this thesis work. To facilitate the fault detection, some additional test points can be used to observe a specific node inside the DUT. These test points can be managed in different ways: they can be collected and processed by some additional modules like monitors, which could process or compress the collected data, they can be used directly as supplementary observation points, like additional probing signals for the fault simulation tool, or connected directly to additional PO of the DUT. In this thesis work, the latter approach has been followed, using directly the information sampled by these additional test points without any conditioning of the data, sending them directly to the fault simulation tool, or creating additional PO connected to these points.

The general flow used to perform each test is depicted in the Figure 3.1: First, (1) the design is analyzed by a static circuit analysis tool to retrieve a set of optimal locations to strobe internal signals during the fault simulation (for now, only observation points are

used). The next steps (2) consists of parsing the observation points locations and insert them in the strobe list of the simulator; depending on the particular type of approach, this last step can be either done by drawing from the optimal test point list (3.a) or by selecting a certain number of random points from an external source like a fault list (3.b). After these steps, a fault simulation is performed (4), and finally, the results are collected and analyzed using MATLAB to plot them (5).

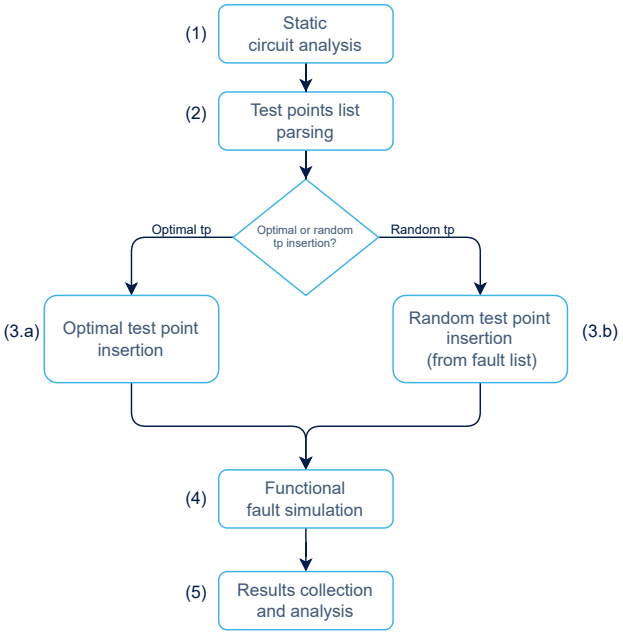


Figure 3.1: General test workflow

3.1 RTL observation points insertion and SBST based fault simulation workflow

Since the previous thesis work focused on the direct implementation of monitors at the RTL level, the first approach to test the proposed methodology has been to apply it directly to the RTL design so that the points found by the static circuit analysis tool could be directly used with the monitor insertion procedure previously developed. In this section, the workflow established to test the performance of the proposed methodology applied at the RTL level is outlined.

3.1.1 Design analysis and optimal test point insertion at RTL level

This workflow (Figure 3.2) involves the use of a static circuit analysis tool (in this thesis work SpyGlass has been adopted) to analyze the CUT to list a series of hard-to-test

locations in which the faults they hold are difficult to detect during the fault simulation; for this reason, these points are worth being observed by inserting a test point directly at the mentioned location. During the synthesis process, the list of optimal points is used to implement those test points, which are later exploited during fault simulation to aid in fault detection. In particular the inserted test points signals are brought out from the hierarchy up to the DUT's top level, where the fault simulator tool can read it's value and use it to perform the fault analysis. Fault testing is performed by launching an SBST simulation test program capable of accurately testing the CUT. The system's response is then used to perform the fault simulation, returning a test coverage metric.

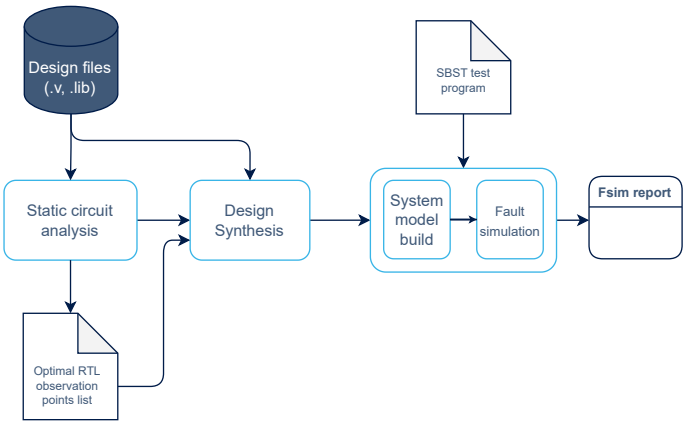


Figure 3.2: General workflow diagram for design analysis and TP insertion at RTL level

3.1.2 Random test points selection and insertion at RTL level

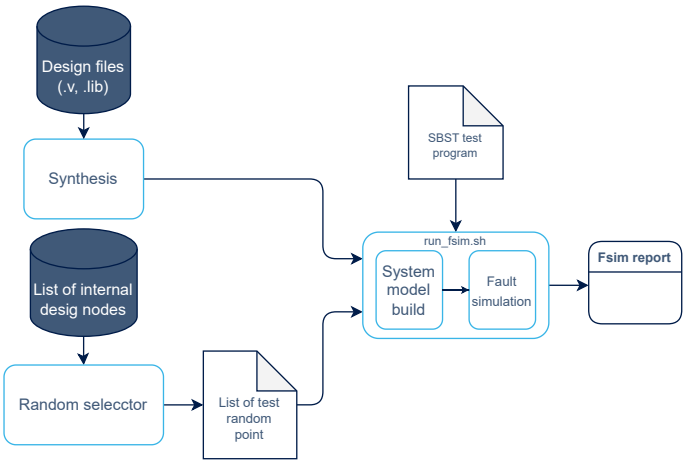


Figure 3.3: General workflow diagram for random TP selection and insertion at RTL level

The workflow for a random selection of test points (Figure 3.3) involves the use of a custom-made program capable of reading a list of CUT's internal points, which are then randomly picked and used as test points during fault simulation. The list of internal locations can be either self-made or taken from other external sources: in this thesis work, for simplicity's sake, the list has been retrieved from the fault list produced during previous fault simulations, so that the locations holding faults are automatically reported without the need for additional work. Apart from this detail, the rest of the workflow is identical to the previous one: the design is synthesized with the defined, randomly selected test points and then tested through an SBST program.

3.2 Post-synthesis observation points insertion and ATPG based fault simulation workflow

After studying the RTL-based methodology, a new approach was followed to achieve more relevant and meaningful results, which is also the procedure suggested by Synopsys guidelines. Following the RTL-based approach, the analysis of the systems happens in pre-synthesis, and this could lead to some problems related to the non-synthesized logic: indeed, any blocks that are generated iteratively, through the construct `generate` or any logic function that is described in a behavioral manner, are only instantiated during the synthesis process. Fortunately, in our specific case, using SpyGlass, a quick synthesis is performed during the design analysis, unfolding all the logic hidden behind the unsynthesized Hardware Description Language (HDL) files. Nevertheless, this is something to take into consideration, and for this reason, instead of working at the RTL level, the analysis and the relative test point insertion are now performed post-synthesis directly on the design netlist.

3.2.1 Post-synthesis design analysis and optimal test point insertion

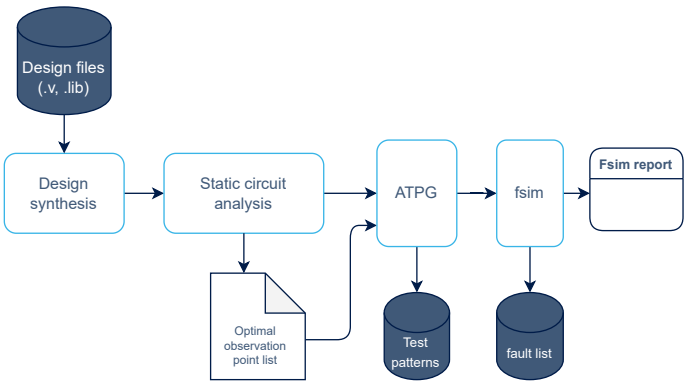


Figure 3.4: General workflow diagram for post synthesis design analysis and TP insertion

Concerning the fault simulation, a new method to exercise the architecture has been adopted. Instead of relying on the use of a test program to perform an SBST, a set of test patterns, generated in ATPG, is fed at the input of the CUT. Conscious of the fact that the ATPG procedure is not ideal for the test of sequential logic, an attempt was made to observe if this technique could provide sufficiently significant results. In Figure 3.4, a general workflow to follow in order to proceed with the previously mentioned methodology is represented: starting from the synthesis, the netlist is provided, and a static analysis is performed on it, retrieving a list of optimal observation locations inside the netlist, which is then modified by implementing additional signals connecting the aforementioned test points to a new set of supplementary PO. The modified netlist is then loaded into the ATPG tool, which generates a set of test patterns using full sequential algorithms to test in functional mode at thePI the CUT. The system response read from the PO is then used to compute the fault simulation and retrieve the test coverage of the CUT.

3.2.2 Post-synthesis random test points selection and insertion

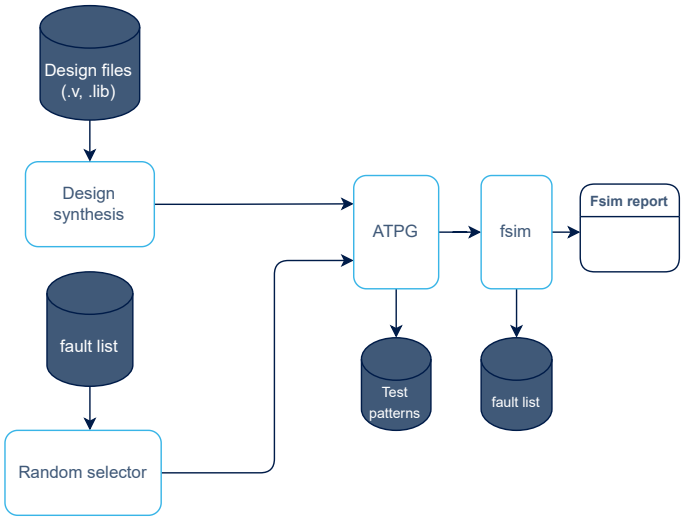


Figure 3.5: General workflow diagram for post synthesis random TP selection and insertion

Similarly to the previous case, the workflow for a random selection of test points (Figure 3.5) involves the use of a custom-made program capable of reading a list of CUT's internal points, which are then randomly picked and used as test points during fault simulation. The list of internal locations can be either self-made or picked from other external sources. The design netlist, is modified by with the aforementioned test locations which are connected directly to an additional set of CUT's PO. Finally an ATPG campaign is run on the modified netlist and a fault simulation is performed returning the test coverage metric.

3.3 Fault simulation with unique precomputed ATPG test patterns workflow

To achieve a more consistent and comparable analysis across simulations, it is crucial to use a single, common set of test patterns. Therefore, a third approach has been followed: performing fault simulation using the exact same stimulus conditions for every test case. This means that the input test patterns for functional fault simulation are not generated independently via ATPG for each simulation run but instead, it is used an external common precomputed set (usually given in STIL or bin format). This workflow allows for loading these external functional test patterns directly into the fault simulation, bypassing the need for an internal pattern generation procedure; so this approach requires that the test pattern set is available beforehand.

3.3.1 Fault simulation with optimal test point insertion

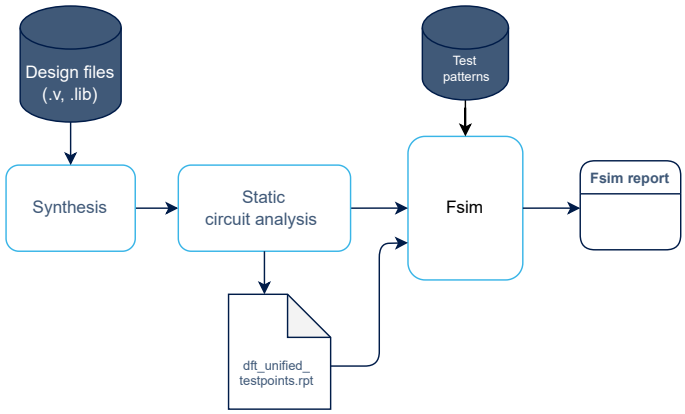


Figure 3.6: General workflow diagram for fault simulation and optimal TP insertion

Figure 3.6 depicts the proposed workflow for analyzing the post-synthesis netlist and running fault simulation on external test patterns. Similar to the previous methodology, the process begins by creating a synthesized netlist of the design, which is then analyzed using a static circuit analysis tool to identify the designated optimal observation points. The netlist is then modified in the aforementioned locations by connecting them to an added set of CUT's PO. Finally, a functional fault simulation is run on the provided set of external test patterns, which are fed into the CUT's PI, while the extended set of PO is used by the fault analysis tool to read the system response and compute the test coverage.

3.3.2 Fault simulation with random test point insertion

The workflow depicted in Figure 3.7 is very similar to the previous one (3.2) with the only difference in the test point selection. Now, instead of relying on a circuit static analysis

tool to suggest an optimal location for test point insertion, a random pick is performed across the whole CUT. These points, as before, are used to modify the netlist, which at the end will implement an extended set of PO connected to the randomly selected positions. Finally, a functional fault simulation is run on the provided set of external test patterns, which are fed into the CUT's PI, while the extended set of PO is used by the fault analysis tool to read the system response and compute the test coverage.

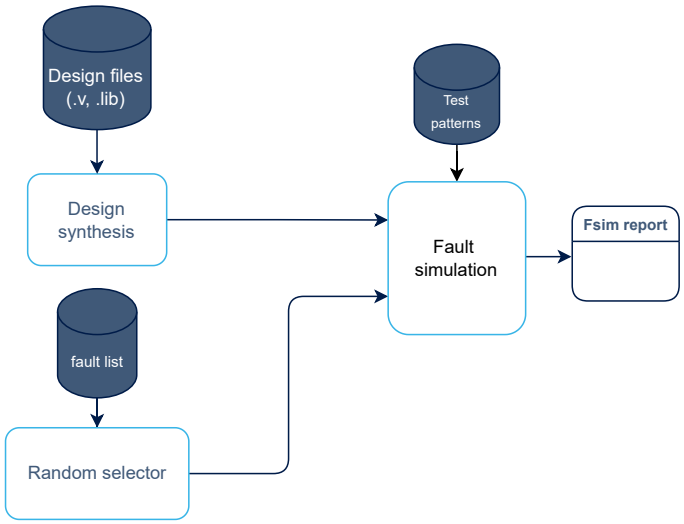


Figure 3.7: General workflow diagram for fault simulation and random TP insertion

Chapter 4

Implementation

In this chapter it will be presented a detailed overview of the steps needed to set up the analyses and simulations, from how the scripts were created to how they are executed. Multiple tools from the Synopsys suite were used; for each tool, a concise yet complete explanation of its principles of operation is provided. A particular emphasis has been placed on reproducibility: during this thesis work, a strong effort has been put into developing a workflow that could be potentially replaceable to other working environments and architectures with a few changes in the scripts. Here the flows described at a high level in Chapter 3 are recalled and outlined in more detail, explaining all the implementation procedure and steps needed to set put the different methodologies. At the heart of this thesis work, there is the use of SpyGlass, a static circuit analysis tool from the Synopsys suite. Other software from the same company has been adopted, like VCS, VC-Z01X to perform SBST tests and TestMax ATPG to run ATPG based fault simulation.

4.1 SpyGlass analysis

SpyGlass is a software developed by Synopsys that uses advanced algorithms and analysis techniques to aid designers and provide them with insights about their designs. This tool is very powerful since it can be integrated at any stage of the design process (from RTL to post-synthesis netlist). SpyGlass offers a vast selection of checks that can be used to test and validate the current work, ensuring that specific constraints (such as timing or power) are met. SpyGlass can also check for consistency and reliability, looking for design choices that could potentially lead to a reduction of performance, reliability, or efficiency of the SoC. SpyGlass is capable of doing so by running the so called “goals”, which are a set of rules specifically intended to inspect and test a specific feature of the design. These goals can be highly configurable, allowing the designer to customize the tool to fit its specific use case. [12]

In this case, SpyGlass has been used to aid the testing process by analyzing the design and suggesting optimal points in which to insert eventual observation monitors. The set of goals used for this work belongs to the DfT-submethodologies, which helps in the optimization of the design for DfT purposes. There is, in particular, a specific rule called

"`Info_random_resistance`" which provides the capability to identify parts of the design containing hard-to-test faults (either hard-to-detect or hard-to-control) which may cause ATPG to abort.

SpyGlass setup

To set up a SpyGlass analysis two main files are needed to be created: a project file which contains the design files (HDL, Libraries), the simulation options, the goal with its related parameters, and a SpyGlass Design Constraints (SGDC) containing constraints on signals, black-box, and clocks.

Project script (Project_tp_insertion.prj)

```
1  ###############################################################################
2  ## Progect File for Spyglass:
3  ## In this file you will find all the TCL commands needed to perform the
   ↪  specified goals
4  ## Author: Mauro Lubrini
5  ###############################################################################
6
7  ##Data Import Section ------------------------------------------------
8  read_file -type verilog ../pd/synth/cva6_synth.v
9  read_file -type gateslib ../pd/synth/tech/NangateOpenCellLibrary.lib
10 read_file -type sgdc sg_setup/cva6.sgdc
11
12 ## Select the module on which run the goal -----------------------------
13 set_option top ex_stage_16_864949
14 ##-------------------------------------------------------------------------
15
16 ##Common option section ----------------------------------------------
17 set_option incdir ../vendor/pulp-platform/common_cells/include/
18 set_option incdir ../vendor/pulp-platform/common_cells/src/
19 set_option incdir ../vendor/pulp-platform/axi/include/
20 set_option incdir ../common/local/util
21 set_option incdir ../core/cache_subsystem/hpdcache/rtl/include
22
23 set_option sortmethod du
24 set_option sort yes
25 set_option allow_non_lrm 1
26
27 set_option language_mode mixed
28 set_option enable_gateslib_autocompile yes
29 set_option enable_gateslib_autocompile yes
30 set_option enableSV09 yes
31
32 ##set_option projectwdir ./output_dir
33 ##set_option projectcwd  ./
34
```

```
## ignore multiple assignment bitwise or violations (W415a) in for loop
set_parameter ignore_bitwiseor_assignment yes
## ignore multiple assignment violations (W415a) in if/else or case
set_parameter ignore_if_case_statement yes
set_option mthresh 20000

##Goal Setup Section ---------------------------------------------------
current_methodology /software/synopsys/spyglass/V-2023.12-SP1/SPYGLASS_H
↪ OME/GuideWare/2023.12/block/rtl_handoff

##current_goal dft/dft_scan_ready
##set_parameter dftGenerateStuckAtFaultReport all
##run_goal

current_goal dft/dft_dsm_random_resistance
set_parameter dftGenerateStuckAtFaultReport all
set_parameter dft_rrf_tp_type observe
set_parameter rme_active 1
##set_parameter dft_insert_ta_tp on
set_parameter dft_insert_rrf_tp on
set_parameter rme_wrap_generate_loop on
set_parameter dft_pattern_count 64000
set_parameter dft_rrf_tp_count 200
set_parameter dft_rrf_tp_effort_level high
set_parameter dft_rrf_tp_report_final_coverage on
run_goal
```

`Project_tp_insertion.prj` is the project file that contains all the commands necessary to run a goal. To run a SpyGlass analysis the content of this file should be copy and paste inside the SpyGlass Graphic User Interface (GUI) shell. The script is composed of different sections:

- **Data Import Section:** In this section, all the design files, such as the HDL files, the libraries, and the constraints file `cva6.sgdc`, are imported. Note that for the SystemVerilog files, it can be either declared a file list of HDL files (to run SpyGlass at RTL level), or it can be declared the synthesized netlist (to run SpyGlass directly on post-synthesis).

- **Select the module:** This option must be changed accordingly to the module that SpyGlass have to analyze and in which to detect fault-resistant nodes. SpyGlass explores the full hierarchy below the selected module; therefore, the recommended observation points are not limited to that module and may be placed in any of its submodules.

- **Common option section:** In this sections are defines general files like headers, package files, and interface definitions. Also in this section, general project options are set, such as: `set_option sortmethod du`: which is used to sort automatically the HDL files; `set_option allow_non_lrm 1`: which is used to enable

parsing of not standard Language Reference Manual (LRM)[1] constructs in Verilog parser; `enableSV09 yes`: sets the SystemVerilog version, enabling SystemVerilog IEEE Std 1800-2009 compatibility; `set_option mthresh 20000`: specifies the bit-count threshold for the compilation of net or variables in a design unit which, by default is set to 4096, but if too low, an error during the SpyGlass run arises, suggesting a safe value (in this case 20000).

- **Goal Setup Section:** In this section, the specific goal and its parameters are defined. Different options are set:
  - `current_methodology`, which defines the sets of goals and rules tailored for a specific scope in the chip design process;
  - `current_goal dft`: defines the specific goals to run. In a single project files multiple goals can be defined to run in sequence;
  - `dftGenerateStuckAtFaultReport all`: generates a detailed fault report about stuck-at faults;
  - `dft_rrf_tp_type observe`: defines the typology of the test point to reports. In this case, only observable points are searched;
  - `dft_pattern_count 64000`: defines the number of patterns used to test the design;
  - `dft_rrf_tp_count 200`: defines the number of optimal test points to find in the whole specified module;
  - `dft_rrf_tp_effort_level high`: defines the effort that the simulation has to adopt to find optimal test points. The higher the effort, the better the analysis at the expense of the run time;
  - `run_goal`: launch the actual goal run.

SpyGlass design constraints script (cva6.sgdc)

```
1  ##########################################################################
2 ## File Name : SpyGlass Constraints File(sgdc file)
3 ## Purpose   : To define additional design information like as clock,
  ↪  reset, case analysis setting etc
4 ##             which needed for correct, accurate and meaningful design
  ↪  analysis using SpyGlass.
5 ## Type      : This is not a tcl compatible file and has it's own
  ↪  command/construct, which can be specified here.
6 ## Usage     : This file is read via SpyGlass Project File(.prj).
7 ##
```

[1]The SystemVerilog Language Reference Manual (LRM) was specified by the Accellera SystemVerilog committee. More detail at: https://ece.uah.edu/ gaede/cpe526/SystemVerilog_3.1a.pdf.

```
## Note : The contents of this file is automatically created using command line inputs provided to 'aipk_read'
##        script, SG designread results & clock/reset auto-inferrencing feature in SpyGlass.
## Note : User must review the auto-inferred/generated SG constraints below before going to design analysis
##        step(using 'aipk_run' script) and update/refine the constraints appropriately as required.
 ########################################################################

## Declare Top-level Design Name
current_design cva6

##### CLOCKS DECLARATION  ################################################
## Syntax: 'clock -name <clock_port/net> -domain <clk-domain> -period <value> -edge {values} -testclock -atspeed'
## Auto-inferred definite clocks ( Add correct domain name, clock period value etc)
clock -name "cva6.clk_i" -domain domain0 -tag SG_AUTO_TAG_1 -testclock -atspeed -period 10 -edge {0 5}

##### ENDING CLOCKS DECLARATION  #########################################

##### RESETS DECLARATION  ################################################
## Syntax: 'rest -name <reset_port/net> -value <0|1> [-sync]'
## Auto-inferred asynchronous definite resets
reset -name "cva6.rst_ni" -value 0

##### ENDING RESETS DECLARATION ##########################################

##### DFT CONSTRAINTS DECALARATION #######################################
## Syntax: 'test_mode -scanshift -name <port/net-name> -value <0|1>'
## Auto-inferred Testmode definitions (for scanshift mode only) for given reset signal declaration
test_mode -scanshift -name "cva6.rst_ni" -value 1

##### ENDING DFT CONSTRAINTS DECALARATION ################################


#### BLACK-BOX CONSTRAINTS DECLARATION ###################################

## Constraint for Black-Box: unread
## Clocks in Black-Box:

#### ENDING BLACK-BOX CONSTRAINTS DECLARATION ############################
```

This constraints file with extension `.sgdc` contains the set of constraints that SpyGlass should consider during the analysis. This particular template has been taken from the

official CVA6 GitHub repository[2]. The file starts with `current_design cva6` defining the top module of the design hierarchy. Next, `clock -name "cva6.clk_i" ...` declare the clock signal with its name, time period, and additional timing parameters. Then the reset signal is defined with `reset -name "cva6_rst_ni" ...`, and since is an active low signal, its active value is set to "0". Finally `test_mode -scanshift ...` set the reset signal value during scan-shifts to "1" (not active) which is something useful if a scan chain methodology is used, but in our case this is irrelevant since no scan chains are implemented.

4.2 Observation points insertion at RTL level and SBST based fault sim

Since the previous thesis work focused on the direct implementation of monitors at the RTL level, the first approach to test the proposed methodology has been to apply it directly to the RTL design so that the points found by SpyGlass could be directly used with the monitor insertion procedure previously developed. In this section, the workflow established to test the performance of the proposed methodology applied at the RTL level is outlined in Figure 4.1.

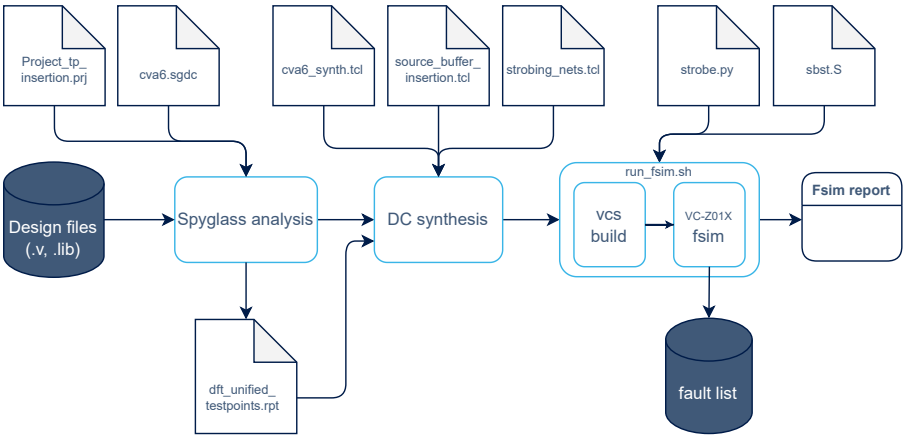


Figure 4.1: Workflow diagram

4.2.1 Synthesis

Once the SpyGlass analysis has been successfully completed, before launching the fault-simulation, since SpyGlass has been run on RTL level, the design should be synthesized to create a netlist to be fed to the simulator. But during the synthesis process, due to the various optimizations that Synopsys Design Compiler performs, the locations pinpointed

[2]CVA6 GitHub repository: https://github.com/openhwgroup/cva6.

by SpyGlass in the RTL design often are eliminated or changed by name. The workaround to this problem was to insert a series of unitary buffers (BUF_X1) acting as placeholders, exactly in the points pinpointed by SpyGlass. Then, during the synthesis process, those buffers are set as "don't touch" so that in case of further optimization steps, these buffers are still kept in their original position and with their name untouched. Later, when declaring the list of additional strobe points for the fault simulation, the only thing to do is to search for those buffers in the synthesized netlist and attach a strobe point at their output pin.

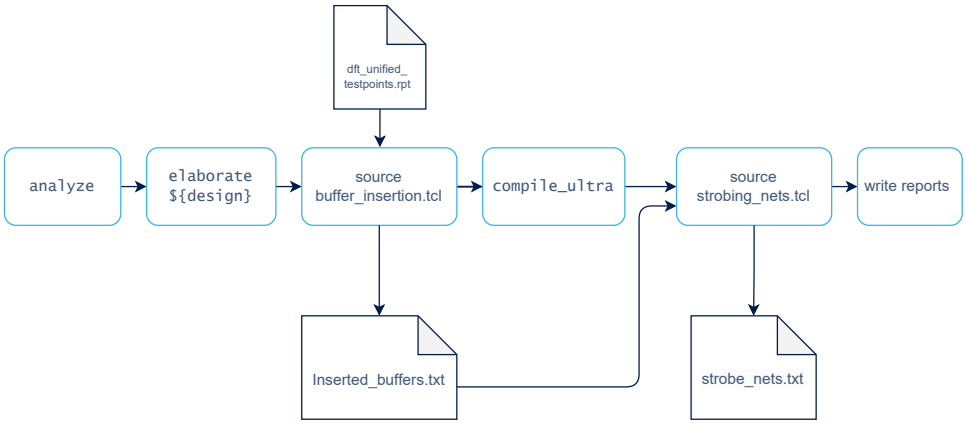


Figure 4.2: Synthesis flow diagram

Figure 4.2 shows the synthesis flow, which extends the standard procedure with two additional steps: buffer insertion and strobe net extraction. In the first one, the Tool Command Language (TCL) script `buffer_insertion.tcl` parses the SpyGlass test-point list, inserts placeholder buffers into the RTL design, and writes their hierarchical design paths to `Inserted_buffers.txt`. In the second one, the TCL script `strobe_list.tcl` reads `Inserted_buffers.txt` where it resolves the listed design paths to each buffer's output pin in the post-synthesis netlist, and creates `strobe_list.txt`, which is later used to edit the VC-Z01X strobe list `.sff` file. This second step cannot be performed directly during the buffer insertion because net names change after synthesis; a separate post-synthesis pass is therefore required to produce a syntax-correct list that matches the post-synthesis naming.

Synthesis script (cva6_synth.tcl)

This is the modified version of the TCL script, which runs the synthesis process. This file is invoked during the execution of Design Compiler by the Makefile target: `make cva6_synth` run in the command window inside the *./pd/synth/* directory.

```
1 # Copyright 2021 Thales DIS design services SAS
2 #
```

```
# Licensed under the Solderpad Hardware Licence, Version 2.0 (the
  ↪  "License");
# you may not use this file except in compliance with the License.
# SPDX-License-Identifier: Apache-2.0 WITH SHL-2.0
# You may obtain a copy of the License at https://solderpad.org/licenses/
#
# Original Author: Jean-Roch COULON - Thales
#


source -echo -verbose scripts/dc_setup.tcl

set clk_name main_clk
set clk_port clk_i
set clk_ports_list [list $clk_port]
set clk_period $PERIOD
set input_delay $INPUT_DELAY
set output_delay $OUTPUT_DELAY
set CVA6_REPO_DIR "../../"
set TARGET_CFG $TARGET

set_app_var search_path "../../vendor/pulp-platform/common_cells/include/
  ↪  $search_path"

# Enable GHM (guide_hier_map) flow (to aid Formality computing formal
  ↪  verification, suggested by the SW)
set_app_var hdlin_enable_hier_map true

sh rm -rf work
sh mkdir work
define_design_lib ariane_lib -path work

set hdlin_keep_signal_name all

#=========================================
# ANALIZE .sv
#=========================================
analyze -vcs "-sverilog -f Flist.cva6_synth" -library ariane_lib

#=========================================
# ELABORATE THE RTL DESIGN
#=========================================
elaborate ${DESIGN_NAME} -library ariane_lib

set_verification_top

uniquify

```

```
48 #Insert the buffer in the points designated by spyglass
49 source buffer_insertion.tcl
50
51 link
52
53 create_clock [get_ports $clk_port] -name $clk_name -period $clk_period
54
55 set_dont_touch to keep sram as black boxes
56 set_dont_touch gen_cache_wt.i_cache_subsystem/i_wt_dcache/i_wt_dcache_me
   ↪ m/gen_tag_srams[*].i_tag_sram
57 set_dont_touch gen_cache_wt.i_cache_subsystem/i_wt_dcache/i_wt_dcache_me
   ↪ m/gen_data_banks[*].i_data_sram
58 set_dont_touch
   ↪ gen_cache_wt.i_cache_subsystem/i_cva6_icache/gen_sram[*].data_sram
59 set_dont_touch
   ↪ gen_cache_wt.i_cache_subsystem/i_cva6_icache/gen_sram[*].tag_sram
60
61 write -hierarchy -format ddc -output
   ↪ ${DCRM_ELABORATED_DESIGN_DDC_OUTPUT_FILE}
62
63 write -f verilog -hierarchy -output ./netlist/cva6_netlist.v
64
65 change_name -rule verilog -hier
66
67 # Prevent assignment statements in the Verilog netlist.
68 set_fix_multiple_port_nets -all -buffer_constants
69 #constraint the timing to and from the sram black boxes
70 set_input_delay -clock main_clk -max $input_delay
   ↪ gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_dcache_mem/gen_tag_s
   ↪ rams_*__i_tag_sram/gen_cut_*__i_tc_sram_wrapper/rdata_o[*]
71 set_input_delay -clock main_clk -max $input_delay
   ↪ gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_dcache_mem/gen_data_
   ↪ banks_*__i_data_sram/gen_cut_*__i_tc_sram_wrapper/rdata_o[*]
72 set_input_delay -clock main_clk -max $input_delay
   ↪ gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen_sram_*__data_sram/g
   ↪ en_cut_*__i_tc_sram_wrapper/rdata_o[*]
73 set_input_delay -clock main_clk -max $input_delay
   ↪ gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen_sram_*__tag_sram/ge
   ↪ n_cut_*__i_tc_sram_wrapper/rdata_o[*]
74
75 set_output_delay $output_delay -max -clock main_clk
   ↪ gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_dcache_mem/gen_tag_s
   ↪ rams_*__i_tag_sram/gen_cut_*__i_tc_sram_wrapper/addr_i[*]
76 set_output_delay $output_delay -max -clock main_clk
   ↪ gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_dcache_mem/gen_data_
   ↪ banks_*__i_data_sram/gen_cut_*__i_tc_sram_wrapper/addr_i[*]
```

```
77 set_output_delay $output_delay -max -clock main_clk
   ↪ gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen_sram_*__data_sram/g
   ↪ en_cut_*__i_tc_sram_wrapper/addr_i[*]
78 set_output_delay $output_delay -max -clock main_clk
   ↪ gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen_sram_*__tag_sram/ge
   ↪ n_cut_*__i_tc_sram_wrapper/addr_i[*]
79
80 # Check the current design for consistency
81 check_design -summary > ${DCRM_CHECK_DESIGN_REPORT}
82
83 #=========================================
84 # COMPILE THE GATE LEVEL DESIGN
85 #=========================================
86 compile_ultra -no_boundary_optimization
87
88 change_names -rules verilog -hierarchy
89
90 #=========================================
91 # WRITE REPORTS
92 #=========================================
93
94 #Extract strobe points from the output nets of the inserted buffers
95 source strobe_list.tcl
96
97 write -format verilog -hierarchy -output
   ↪ ${DCRM_FINAL_VERILOG_OUTPUT_FILE}
98 write -format verilog -hierarchy -output ${DESIGN_NAME}_synth.v
99 write -format ddc     -hierarchy -output ${DCRM_FINAL_DDC_OUTPUT_FILE}
100
101 report_timing -nworst 10  >  ${DCRM_FINAL_TIMING_REPORT}
102 #report_timing -through gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_
   ↪ dcache_mem/gen_tag_srams_*__i_tag_sram/#gen_cut_*__i_ram/rdata_o[*]
   ↪ >>  ${DCRM_FINAL_TIMING_REPORT}
103 #report_timing -through gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_
   ↪ dcache_mem/#gen_data_banks_*__i_data_sram/gen_cut_*__i_ram/rdata_o[*]
   ↪ >>  ${DCRM_FINAL_TIMING_REPORT}
104 #report_timing -through gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen
   ↪ _sram_*__data_sram/gen_cut_*__i_ram/#rdata_o[*] >>
   ↪ ${DCRM_FINAL_TIMING_REPORT}
105 #report_timing -through gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen
   ↪ _sram_*__tag_sram/gen_cut_*__i_ram/#rdata_o[*] >>
   ↪ ${DCRM_FINAL_TIMING_REPORT}
106 #report_timing -through gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_
   ↪ dcache_mem/gen_tag_srams_*__i_tag_sram/#addr_i[*] >>
   ↪ ${DCRM_FINAL_TIMING_REPORT}
107 #report_timing -through gen_cache_wt_i_cache_subsystem/i_wt_dcache/i_wt_
   ↪ dcache_mem/#gen_data_banks_*__i_data_sram/addr_i[*] >>
   ↪ ${DCRM_FINAL_TIMING_REPORT}
```

```
#report_timing -through gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen
  ↪  _sram_*__data_sram/addr_i[*] >>  #{DCRM_FINAL_TIMING_REPORT}
#report_timing -through gen_cache_wt_i_cache_subsystem/i_cva6_icache/gen
  ↪  _sram_*__tag_sram/addr_i[*] >>  $#{DCRM_FINAL_TIMING_REPORT}

#report_area -hier -nosplit > ${DCRM_FINAL_AREA_REPORT}
#write_parasitics -output ${DCRM_FINAL_SPEF_OUTPUT_FILE}
write_sdc ${DCRM_FINAL_SDC_OUTPUT_FILE}
write_sdf ${DESIGN_NAME}_synth.v.sdf

exit
```

In the script, the four main synthesis steps have been highlighted by proper comments. Starting from the "analysis step", `analyze` reads the design HDL files and stores the resulting templates into the specified library. With the `-vcs "-sverilog -f Flist.cva6_synth"` is specified the format of the files to be analyzed: in this case, they are SystemVerilog files collected inside a filelist.
Next in the "elaboration step", `elaborate ${DESIGN_NAME} -library ariane_lib` elaborates the design, `uniquify` creates a unique design for each duplicated cell instance and `buffer_insertion.tcl` is invoked to perform the buffer insertion at RTL level in the design. The process continues launching `compile_ultra` to gate-level compile the design and `change_names` names the netlist elements (ports, cells, and nets) by following Verilog syntax. Finally, in the "write report" section, by invoking `strobe_list.tcl`, the output file containing the list of inserted buffers generated by `buffer_insertion.tcl` is parsed, and syntactically corrected accordingly to Verilog rules; this is needed to have a list of test points that the fault simulator can later retrieve by looking at the synthesized netlist.

Buffer insertion script (buffer_insertion.tcl)

This script is used in the synthesis flow to insert placeholder buffers at the RTL. It reads the SpyGlass test points report `dft_unified_testpoints.rpt` and adds a simple buffer at each indicated location. Because some points may reference to unconnected nets, the script first checks connectivity and skips any unused nets, as they do not help in fault detection. It produces two outputs: a list of all inserted buffers with their hierarchical paths (`Inserted_buffers.txt`), and a report summarizing the connection count for the nets used (`connected_nets.txt`).

```
##############################################################################
# Mauro Lubrini
#
# Description: This TCL script is used to insert buffers in the positions
# defined by SpyGlass. A BUF_X1 buffer is appended to the net's path.
# The notation used to name the buffer is "add_buf<x>" where x is the
# index of that buffer. Two .txt files are generated:
# (1.) connected_nets.txt: lists all the nets that at least one
  ↪  connection
```

```
# (2.) Inserted_buffers.txt: a list of all inserted buffer and the
# path in the hierarchy
################################################################################

#Variables-------------------------------------------------------------------
set buf_type "BUF_X1"
set bufname ""
set connected_nets ""
set buffer_inserted ""
set i 1
#To be changed accordingly
set preamble "ex_stage_i/"

#Path files------------------------------------------------------------------
set in_file_path "../../spyglass/spyglass-1/ex_stage/dft/dft_dsm_random_resistance/spyglass_reports/dft/dft_unified_testpoints.rpt"
set out_file_path "./connected_nets.txt"
set out_file_path1 "./Inserted_buffers.txt"

set file_i_id [open $in_file_path r]


#Run throuh all lines of the test point report from Spyglass
while {[gets $file_i_id line] != -1} {

        #Select only the lines containing the tp list (they start with a number)
        if {[regexp {^[0-9]} $line]} {

                #Exteract only the net path
                set net "$preamble[lindex [split $line] 6]"
                #Check the number of connection of that path
                set number_of_conn [sizeof_collection [all_connected $net]]
                #Select only the nets which have >=1 connections to them
                if {$number_of_conn > 0} {

                        #Buffer insertion
            set bufname [format "add_buf%d" "$i"]
                        insert_buffer $net $buf_type -new_cell_name "$bufname"

                        #Set don't touch
                        set net_path [join [lrange [split $net "/"] 0 end-1] "/"]
                        set_dont_touch [get_cells $net_path/$bufname]

                        # Inser in the reports
```

```
52                          append connected_nets [format "%d - $net -
                            ↪  #%d#\n" "$i" "$number_of_conn" ]
53                          append buffer_inserted [format "%d\)$bufname =>
                            ↪  $net_path/$bufname\n" "$i" ]
54                          # debug
55                          puts "=>$bufname inserted."
56
57                          # To give to each net a sequential increasing
                            ↪  index
58                          incr i
59                  }
60          #To give to each net the same index in the Syglass list
61                  #incr i
62          }
63  }
64
65  #Print output files
66  set file_o_id [open $out_file_path w]
67  puts $file_o_id $connected_nets
68  set file_o_id1 [open $out_file_path1 w]
69  puts $file_o_id1 $buffer_inserted
70
71  #Close files
72  close $file_i_id
73  close $file_o_id
74  close $file_o_id1
```

The script starts by defining the variables, with `buf_type`, which sets the type of buffer (one should select those present in the used library), and `preamble`, which is used to fill the gap between the top-level module of the CVA6 (`cva6/`) and the module analyzed by SpyGlass. This is done because in the SpyGlass report, the nets' paths are referenced, not starting from the absolute top hierarchy, but from the first level of the top module analyzed.[3] Next, a while loop which scans all the SpyGlass test point report and parses each line retrieving each net's hierarchical path. Once the path has been isolated in `$net`, with the command `$insert_buffer` a buffer is placed in the location indicated. Next, with the command `set_dont_touch [get_cells ...]` Design Compiler is asked to keep the declared cell untouched during the other synthesis steps. Finally, it updates the two output reports: one listing every inserted buffer (with its hierarchical path) and another reporting the connectivity of the processed nets.

[3]For example, if the design hierarchy is: **cva6/ex_stage/alu/...** and SpyGlass analyzes the execution stage, the nets listed in the report would start with the first level under the execution stage, so: **alu/....**

Strobe selection script (strobe_list.tcl)

This script is run post-compilation during the synthesis flow. It is responsible for creating a list of strobe points (`strobe_list.txt`) to be inserted in the VC-Z01X configuration `.sff` file by the script `strobe.py` during the fault simulation process.

```
1  ##############################################################################
2  # File: strobe_list.tcl
3  # Author: Mauro Lubrini
4  # Date: 23/05/2025
5  # Description: This TCL script is used to retrieve the output net name
6  # of the inserted buffers to be inserted in the .sff configuration file
7  # of VC-Z01X. The net names are written in strobe_list.txt, which is
8  # later parsed by a strobe.py during the fault simulation flow.
9  ##############################################################################
10
11
12 #Variables------------------------------------------------------------------
   ↪  --
13 set strobing_list ""
14
15 #Path
   ↪  files-------------------------------------------------------------------
16 set in_file_path "./Inserted_buffers.txt"
17 set out_file_path "./strobe_list.txt"
18
19 set file_i_id [open $in_file_path r]
20
21 #Run throuh all lines of the test point report from Spyglass
22 append strobing_list [format "<< Spyglass selected test points >>\n"]
23
24 while {[gets $file_i_id line] != -1} {
25
26         if {"$line" ne ""} {
27                 #Exteract only the net path
28                 set buf_path_raw [lindex [split $line] 2]
29
30                 #use the post elaborate format
31                 set path [string map {[ _ ] _ . _} [string tolower
                   ↪  $buf_path_raw]]
32                 set strobe_path [string map {/ .} [join [ list "uvmt_cva
                   ↪  6_tb.cva6_dut_wrap.cva6_tb_wrapper_i.i_cva6" $path
                   ↪  "Z"] "."]]
33
34                 puts "DEBUG_2----------------------------"
35                 puts "$buf_path_raw"
36                 puts "$path"
37                 puts "$strobe_path"
38
```

```
39                  if {"$strobe_path" ne ""} {
40                          append strobing_list [format "$strobe_path, "]
41                  } else {
42                          puts "ERROR, NO OUTPUT NET FOUND FOR
                             ↪  $strobe_path!\n"
43                  }
44          }
45  }
46
47  #Print output files
48  set file_o_id [open $out_file_path w]
49  puts $file_o_id $strobing_list
50
51  #Close files
52  close $file_i_id
53  close $file_o_id
54
```

The script is composed of a while loop, which runs through all the elements in the `Inserted_buffers.txt` file and, for each of them, it reads the hierarchical path and modifies it following the SystemVerilog syntax. This is necessary to have a list of paths that can be used to redirect to points inside the synthesized list. In particular, in the variable `path` is saved the path read from `strobe_list.txt` changing the squared brackets (`[`, `]`) with underscores (`_`), and the slashes (`/`) substituted with dots (`.`). Then, in the variable `strobe_path`, the preamble `"uvmt_cva6_tb.cva6_dut_wrap.cva6_tb_wrapper_i.i_cva6"` is inserted before the content of `path`, and the character ".Z" is appended at the end of it; this is because we want to connect each strobe point to the output pins of the inserted buffers. Finally, the output file `strobe_list.txt` is written.

4.2.2 Fault simulation run

To run the fault simulation, a specific bash script (`run_fsim.sh`), already present in the CVA6 environment, must be executed. This script launches an automated process that recalls other scripts to set up the simulation environment. For this initial approach, the same type of SBST simulation performed in the previous thesis work was used, employing an assembly program to exercise the architecture[4]. The tools used for the fault simulation are VCS combined with VC-Z01X: the first is used to gate-compile the design and build the executable model, while the second performs the actual fault simulation, running the compiled assembly program.

[4]The whole simulation environment has been developed and set up by the CVA6 community available on the GitHub repository: https://github.com/openhwgroup/cva6.

Preparaton to the fault simulation

To launch the fault simulation with VC-Z01X, some configuration files have to be modified. In particular is needed to define which module of the architecture should be fault-injected, simulated, and which are the strobe points that VC-Z01X uses to read the fault propagation.

Step 1: Change the `.sff` file by selecting the correct module in which to inject the faults.

strobe.sv

```
1 FaultGenerate
2 {
3     #NA [0,1] { PORT
      ↪  "uvmt_cva6_tb.cva6_dut_wrap.cva6_tb_wrapper_i.i_cva6.**" }
4     NA [0,1] { PORT "uvmt_cva6_tb.cva6_dut_wrap.cva6_tb_wrapper_i.i_cva6
      ↪  .ex_stage_i.**" }
5     #NA [0,1] { PORT "uvmt_cva6_tb.cva6_dut_wrap.cva6_tb_wrapper_i.i_cva
      ↪  6.ex_stage_i.alu_i.**" }
6 }
```

The `faults.sff` defines in which specific module of the design (and all its related sub-modules) VC-Z01X will inject the faults. This should be selected accordingly to the desired top module on which the fault simulation will be performed. A specific line has to be inserted, starting with the type of fault (in this case `NA [0,1]` means to inject stuck-at-0 and stuck-at-1 faults) and then with the module design path. Multiple modules can be declared simultaneously, but for our purposes, only one module at a time is simulated (all the others are commented).

Step 2: Change the DUT in the Makefile[5], which we want to simulate: Inside this Makefile, there is a line that specifies which is the top module to use as DUT for the fault simulation. This should match the same module previously defined in the `.sff` file.

./verif/sim/Makefile

```
164 ###########################################################################
165 # UVM specific commands, variables
166 ###########################################################################
167 ALL_VCS_FLAGS = $(if $(VERDI) || $(FSIM), -kdb -debug_access+all -lca,)
    ↪  -sverilog -full64 -timescale=1ps/1ps $(if $(FSIM),
    ↪  $(CVA6_FSIM_SRC_PATH)/strobe.sv -fsim -fsim=dut:uvmt_cva6_tb.cva6_d
    ↪  ut_wrap.cva6_tb_wrapper_i.i_cva6.ex_stage_i -fsim=portfaults
    ↪  +vcs+fsdbon ,) $(if $(SAFE), +nospecify, )
```

[5]The cited Makefile is located in the project directory: ./verif/sim/Makefile.

```
export VCS_WORK_DIR =
    ↪  $(CVA6_REPO_DIR)/verif/sim/vcs_results/default/vcs.d
SIMV = $(VCS_WORK_DIR)/simv
```

The option to look for inside the Makefile code is in the "UVM specific commands, variables" section, and the line to change is this one:
`-fsim=dut:uvmt_cva6_tb.cva6_dut_wrap.cva6_tb_wrapper_i.i_cva6.<...>`

Step 3: Declare the strobe points for VC-Z01X in the configuration strobe file `strobe.sv`[6] This file defines the points at which VC-Z01X reads the system response and the faults propagation from a fault simulation. By default, VC-Z01X uses the primary output of the declared TOPLEVEL to read fault propagation, but additional strobe points can be inserted. To ease the strobe insertion, a python script (`strobe.py`) is invoked in `run_fsim.sh` before launching the fault simulation, and is used to read the elected observation point listed inside the `strobe_list.txt` and automatically insert them in the `strobe.sv` file.

```
`timescale 1ps / 1ps

`ifndef TOPLEVEL
        `define TOPLEVEL uvmt_cva6_tb.cva6_dut_wrap.cva6_tb_wrapper_i.i_
        ↪  cva6.ex_stage_i
  //TODO: INSERT HERE THE STROBE LIST OF SIGNALS
`endif

module strobe;

integer cmp;

initial begin
  #10;
  @(posedge `TOPLEVEL.rst_ni);
  $fs_inject();
end

always @(negedge `TOPLEVEL.clk_i) begin
  cmp = $fs_compare(`TOPLEVEL);
  if (1 == cmp) begin
    $fs_drop_status("ON", `TOPLEVEL);
  end else if (2 == cmp) begin
    $fs_drop_status("PN", `TOPLEVEL);
  end
end

```

[6]The cited file is located in the project directory: ./fmeda/fsim/src/strobe.sv.

```
27 always @(negedge `TOPLEVEL.clk_i) begin
28   cmp = $fs_compare(`STROBELIST);
29   if (1 == cmp) begin
30     $fs_drop_status("ON", `STROBELIST);
31   end else if (2 == cmp) begin
32     $fs_drop_status("PN", `STROBELIST);
33   end
34 end
35
36 endmodule
37
```

This configuration file begins by defining the list of signals to observe. This can be done in two ways: either by defining a module (VC-Z01X will implicitly consider its primary outputs) or by listing each net's hierarchical path explicitly. With `ifndef TOPLEVEL ...` it's defined the top module under test whose primary outputs are read by VC-Z01X. To implement additional observation points, a new list of signals containing the paths to the internal design nets to strobe should be inserted; for this purpose, a `TODO` comment is used as a placeholder for `strobe.sv` to place `` `define STROBELIST...(list_of_paths) ``. Next, it defines the clock, which synchronizes sampling of the outputs. On each negative clock edge, with the command `cmp = $fs_compare()`, VC-Z01X compares the Good Machine (GM) against the Faulty Machine (FM) values of the listed signals.

- If `cmp=0` it means that the GM and the FM values are identical, so no fault is detected;
- if `cmp=1` it means that there is a discrepancy between the GM and FM values (GM=0 or 1 while FM is the opposite), so a fault is detected, and in this case is labeled as an "observed not diagnosed" (ON);
- if `cmp=2` it means that GM have know values (either 0 or 1) while the FM have a value in an unknown state (either X or Z) and in this case the fault is labeled as "Potentially Observed Not Diagnosed" (PN). [13]

strobe.py

This Python script is used to read the list of additional points to observe in the design (`strobe_list.txt`) and automatically insert them into the configuration file `strobe.sv`. This process is not done directly on top of `strobe.sv`, but a second SystemVerilog file (`template.sv`) is used as a template, so that each time a new set of observation points needs to be implemented, a clean file is available to be modified. Summing up, this program takes `template.sv` and writes in the designated location the list of observation points paths, and then saves it as `strobe.sv`.

```
1 """
2 File: strobe.py
3 Author: Mauro Lubrini
```

```
Date: 27/05/2025
Description: This script is used to read the list of strobepoints
generated during the syhtesis process from the file strobe_list.txt
and update the strobe list of the strobe.sv. "teamplate.txt" is
used as reference model forthe actual file strobe.sv
"""

FILE_PATH = "../../../pd/synth/strobe_list.txt"
new_file = []

# open the file containing the strobepoints
strobe_list = open(FILE_PATH, "r")
strobes = strobe_list.readlines()

# open the teamplate file
teamplate = open("template.sv", "r")
lines = teamplate.readlines()

for i, line in enumerate(lines):
    new_file.append(lines[i])
    if '//TODO: INSERT HERE THE STROBE LIST OF SIGNALS' in line:
        new_file. append("  `define STROBELIST ")
        for j, signal in enumerate(strobes):
            if (j != 0):
                new_file.append(strobes[j])

        new_file.append("\n")

# Creating the strobe.sv file
with open("strobe.sv", "w") as outfile:
    outfile.writelines(new_file)

teamplate.close()
strobe_list.close()
```

The script starts by defining the directory containing the strobe list to read (`strobe_list.txt`) and initializing a new list (`new_file=[]`) that stores the content of the new file to be created. Then the template file is opened and a for loop reads all its lines, looking for the comment "`//TODO: INSERT ...`", which pinpoints the exact location in which to insert the hierarchical path of the additional observation points. Once this line is found, the content of `strobe_list.txt` is inserted, and the loop continues running through the lines of the template file, appending each one of them to the `new_file` list, which finally is printed as a file under the name of `strobe.sv`

run_fsim.sh

As already mentioned above, the CVA6 environment is equipped with a quite intricate simulation environment that, depending on the options set, can perform different activities. In our case, the environment is set to run a fault simulation using Synopsys VCS and VC-Z01X. The process starts by executing `run_fsim.sh`.

```
#!/bin/bash

# Define the script to be run
SCRIPT="run_fsim_sbst_nogui.sh"
SCRIPT2="strobe.py"

# File to extract the coverage percentage from
REPORT_FILE="./verif/sim/fsim_v.rpt"

# Log file to store the iteration and coverage information
LOG_FILE="fsim.log"

# Path to the sbst.S file
SBST_FILE="./verif/tests/custom/sbst/sbst.S"

# Path to the backup file
BACKUP_FILE="./verif/tests/custom/sbst/sbst_backup.S"

# Clear the log file if it exists
> "$LOG_FILE"

# Initialize maximum, mean and sum coverage to zero
OBS_COVERAGE=0

# Main
echo "------------------------------------------------------------------------"
echo "Running fsim on the ALU with observation points insertion" | tee
  ↪  -a "$LOG_FILE"
echo "------------------------------------------------------------------------"

#update the strobe.sv file with the strobes found with Spyglass
cd fmeda/fsim/src
python3 "$SCRIPT2"
cd ../../../

# Start the fault simulation
./$SCRIPT
if [ $? -ne 0 ]; then
  echo "Script encountered an error during iteration $i" | tee -a
    ↪  "$LOG_FILE"
  exit 1
fi
```

```
41
42 if [ -f "$REPORT_FILE" ]; then
43   # Extract numerical coverage value
44   OBS_COVERAGE=$(tac "$REPORT_FILE" | grep -m 1 "Observational Coverage"
     ↪  | grep -oE '[0-9]+\.[0-9]+')
45   echo "Observational Coverage: $OBS_COVERAGE%" | tee -a "$LOG_FILE"
46 else
47   echo "Report file $REPORT_FILE not found." | tee -a "$LOG_FILE"
48   exit 1
49 fi
50
51 echo "Script has been run successfully!" | tee -a "$LOG_FILE"
```

The script starts by defining the scripts to be run, the path for the coverage report, and the simulation log file. Then `strobe.py`[7] is run to insert the strobe points in the fault sim configuration file of VC-Z01X. Next, the actual simulation is run by `run_fsim_sbst_nogui.sh`, which sets up the simulation environment and options. The remaining lines of code are used only to extract and display the numerical coverage value from the simulation results.

run_fsim_sbst_nogui.sh

```
1  #!/bin/bash
2
3  if [ -z ${VCS_HOME} ]; then
4      echo "\$VCS_HOME variable not set! Exiting..."
5      exit 1
6  fi
7
8  ### ENVIRONMENT SETUP ###
9  source set_env.sh
10
11 ### SELECT GATE LEVEL SIMULATION ###
12
13 export DV_SIMULATORS=vcs-gate
14
15 ### SELECT TARGET CVA6 CONFIGURATION ###
16
17 export TARGET_CFG="cv64a6_imafdc_sv39"
18
19 ### ENABLE FAULT SIMULATION WITH VC-Z01X ###
20
21 export FSIM=1
22
```

[7] In case of random strobe, the script randomize_strobe_selection.py is run right before strobe.py.

```
### DISABLE GUI ###

export TRACE_COMPACT=1 && unset VERDI

### RUN SBST WITH DEFINED CONFIGURATION
source ./verif/regress/sbst.sh

exit 0
```

`run_fsim_sbst_nogui.sh` is used to configure the simulations options, starting with checking if the `VCS\_HOME` and the simulation environment is all set up. Then, with `export DV_SIMULATORS=vcs-gate` is selected the VCS gate-level simulation option, and with `export TARGET_CFG=cv64a6_imafdc_sv39` the specific CVA6 configuration. Next `FSIM=1` enables the fault simulation branch, and with `export TRACE_COMPACT=1 && unset VERDI` the GUI is disabled. Finally, the sbst driver `sbst.sh` is launched.

sbst.sh

This script acts as a driver for the execution of the assembly program `sbst.S`. It installs the necessary tools, cleans the build directories, and selects the directed assembly program to run. Then it invokes `cva6.py` and defines its options.

```
# where are the tools
if ! [ -n "$RISCV" ]; then
  echo "Error: RISCV variable undefined"
  return
fi

# install the required tools
source verif/regress/install-cva6.sh
source verif/regress/install-riscv-dv.sh
source verif/regress/install-riscv-compliance.sh
source verif/regress/install-riscv-tests.sh

if ! [ -n "$DV_SIMULATORS" ]; then
  DV_SIMULATORS=vcs-gate
fi

make clean
make -C verif/sim clean_all

cd verif/sim

# src0=../tests/custom/sbst/sbst_main.c
src0=../tests/custom/sbst/sbst.S
srcA=(
        ../tests/custom/common/syscalls.c
```

```
        # ../tests/custom/sbst/sbst.S
        ../tests/custom/common/crt.S
)
cflags=(
        -fno-tree-loop-distribute-patterns
        -static
        -mcmodel=medany
        -fvisibility=hidden
        -nostdlib
        -nostartfiles
        -lgcc
        -O3 --no-inline
        -I../tests/custom/env
        -I../tests/custom/common
        -I../tests/custom/sbst/
        -DNOPRINT
)

set -x
python3 cva6.py \
        --target hwconfig \
        --hwconfig_opts="--default_config=cv64a6_imafdc_sv39
        ↪  --isa=rv64imafdc --NrLoadPipeRegs=0" \
        --iss="$DV_SIMULATORS" \
        --iss_yaml=cva6.yaml \
        --asm_tests "$src0" \
        --gcc_opts "${srcA[*]} ${cflags[*]}" \
        --linker ../tests/custom/common/test.ld

```

cva6.py

The `cva6.py` is a very intricate Python script that acts as an "orchestrator" for the fault simulator; it reads a YAML file, which routes the top-level Makefile in `verif/sim` carrying the VCS/VC-Z01X simulation. The process compiles the assembly `sbst.s` file into an Executable and Linkable Format (ELF) which can be executed later by VC-Z01X. It then uses VCS to build and compile the DUT merging the UVM testbench with the FSIM instrumentation, creating an executable gate-level snapshot of the design. Then, having set `FSIM=1` the fault simulation branch is taken, VC-Z01X is launched running the golden and fault-injected campaigns on that VCS-built snapshot using the same SBST compiled ELF, performing in this way the fault simulation. Finally the coverage report is then written into the file `verif/sim/fsim_v.rpt`.

4.2.3 Random buffer insertion

As already said in previous sections, the aim of this work is to compare the proposed methodology with the case of a random choice of test points. So, a list of all the nets

inside the module under test is needed to draw a random set of strobe points. The idea was to exploit the fault list generated during the previous fault simulation, which contains a list of faults and their relative locations in the design. For this purpose, `randomize_strobe_selection.py` is launched while running the fault simulation flow. This python script is invoked in `run_fsim.sh` right before starting the fault simulation, and it selects from the fault list a set of random points inside the netlist substituting their path to the ones contained in the `strobe_list.txt` file. So the next time the strobe list of VC-Z01X is created, instead of drawing the SpyGlass suggested list, a set of randomly selected points is used.

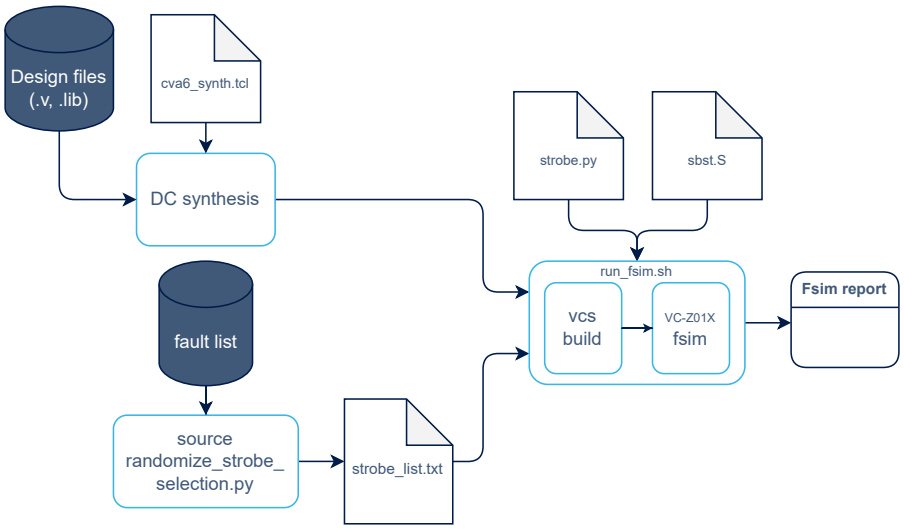


Figure 4.3: workflow diagram

run_fsim.sh

Similar to the SpyGlass-driven case, the fault simulation is run by invoking `run_sbst.sh`, but this time, `randomize_strobe_selection.py` is run before launching the strobe insertion (`strobe.py`) and fault simulation (`run_fsim_sbst_nogui.sh`). The rest of the procedure remains unchanged with respect to the one previously described.

```
1 #!/bin/bash
2
3 # Define the script to be run
4 SCRIPT="run_fsim_sbst_nogui.sh"
5 SCRIPT2="strobe.py"
6 SCRIPT3="randomize_strobe_selection.py"
7 TP_REPORT="./verif/sim/fsim_v.rpt"
8 TOT_REPORT="./verif/sim/Report_backup/sim_collection.txt"
9
```

```
# File to extract the coverage percentage from
REPORT_FILE="./verif/sim/fsim_v.rpt"

# Log file to store the iteration and coverage information
LOG_FILE="fsim.log"

# Path to the sbst.S file
SBST_FILE="./verif/tests/custom/sbst/sbst.S"

# Path to the backup file
BACKUP_FILE="./verif/tests/custom/sbst/sbst_backup.S"

# Clear the log file if it exists
> "$LOG_FILE"

# Initialize maximum, mean and sum coverage to zero
OBS_COVERAGE=0

echo "------------------------------------------------------------------------"
echo "Running fsim with random observation points: iteration $i" | tee
↪  -a "$LOG_FILE"
echo "------------------------------------------------------------------------"

# Insert a new random set of tp
cd pd/synth
python3 "$SCRIPT3"
cd ../../

#update the strobe.sv file
cd fmeda/fsim/src
python3 "$SCRIPT2"
cd ../../../

# Start the fault simulation
./$SCRIPT
if [ $? -ne 0 ]; then
  echo "Script encountered an error during iteration $i" | tee -a
  ↪  "$LOG_FILE"
  exit 1
fi

if [ -f "$REPORT_FILE" ]; then
  # Extract numerical coverage value
  OBS_COVERAGE=$(tac "$REPORT_FILE" | grep -m 1 "Observational Coverage"
  ↪  | grep -oE '[0-9]+\.[0-9]+')
  echo "Observational Coverage: $OBS_COVERAGE%" | tee -a "$LOG_FILE"
else
  echo "Report file $REPORT_FILE not found." | tee -a "$LOG_FILE"
```

```
  exit 1
fi

echo "Script has been run successfully!" | tee -a "$LOG_FILE"
```

Random buffer selection script (randomize_strobe_selection.py)

```
"""
File: strobe.py
Author: Mauro Lubrini
Date: 27/05/2025
Description: This script is used to read the fault list generated during
the first fault-sim and then select randomly N-strobing points from that
list (N=STROBE_NUMBER)
"""

import random

# Variables and constants
INFILE_PATH = "../../verif/sim/fsim_v.rpt"
OUTFILE_PATH = "./strobe_list.txt"
STROBE_NUMBER = 100

csn_lines = 0
fault_list = []
selected_index = []
strobe_list = []


infile = open(INFILE_PATH, "r")
lines = infile.readlines()

# Extrapolate all the fault injection points from the fault list
for i, line in enumerate(lines):
    if ('PORT' in line):
        elements = line.split()
        if (len(elements)==4):
            fault_list.append(elements[3].rstrip("\"}").lstrip("\"") + ",
            ↪  ")
        elif (len(elements)==6):
            fault_list.append(elements[5].rstrip("\"}").lstrip("\"") + ",
            ↪  ")

list_length=(len(fault_list))

# Create the strobe list
if (list_length>=STROBE_NUMBER):
```

```
39      strobe_list.append("<< Random selected test points>>\n")
40      for j in range(STROBE_NUMBER):
41          rand_index = random.randrange(0, list_length-1, 1)
42          if rand_index not in selected_index:
43              selected_index.append(rand_index)
44              strobe_list.append(fault_list[rand_index])
45              j += 1
46              print(rand_index)
47  else:
48      print("ERROR: the faultlist is smaller than the number of strobe we
        ↪  want to insert")
49      print("       please select a value for STROBE_NUMBER lower than",
        ↪  list_length )
50
51  # Create the list of random strobing points
52  with open(OUTFILE_PATH, "w") as outfile:
53      outfile.writelines(strobe_list)
54
55  infile.close
```

The `randomize_strobe_selection.py` script is used to read the fault list generated during the first fault-sim and then select randomly N-strobe points from that list. The script begins by importing the Python's standard library module `import random`. Next, it defines `INFILE_PATH` which is the input fault list path, `OUTFILE_PATH` which is the output file path, and finally `STROBE_NUMBER` which is the variable that sets the number of random locations to be selected from the fault list; note that this value should be set equal to the number of observation point extracted by SpyGlass[8]. A first loop parses the fault list, extrapolating for each fault injection location its corresponding hierarchical path, and saving it in a temporary list (`fault_list`). Finally, a second loop is used to select random entries from `fault_list`, which are later written in the output text file.

4.3 Post-synthesis observation points insertion and ATPG based fault simulation

In this new approach, following the workflow described in Section 3.2, the architecture has been exercised through ATPG and then fault simulated using Synopsys TestMAX ATPG. As before, SpyGlass has been run on the synthesized netlist following the same synthesis and analysis procedure described in the previous section. Regarding the test procedure, the flow is now quite streamlined. Figure 4.4 depicts the main steps: after the design synthesis and SpyGlass analysis on the netlist, the ATPG fault simulation is run by launching `run_ATPG.sh`, which calls TMAX running the TCL script `Full_seq_ATPG.tcl`, which automatically performs an ATPG fault simulation, also integrating the candidate

[8]This is important to keep simulations under comparable conditions between random and SpyGlass selected test points.

test points found by the previous SpyGlass analysis. Performing new ATPG runs each time a new fault simulation is performed could lead to non-equal and non-identical scenarios, which means that the comparisons between the random test point insertion and the SpyGlass selected one could show some discrepancies due to differences in the set of input vectors, so in the way the design is exercised for the two fault simulations. This observation could lead to the thinking that the previous work of comparing different simulations being run on different sets of test patterns is corrupted and meaningless, but actually, a test like that allows us to appreciate how the implementation of test points inserted in different locations could help and ease TMAX ATPG algorithms in generating effective test vectors.

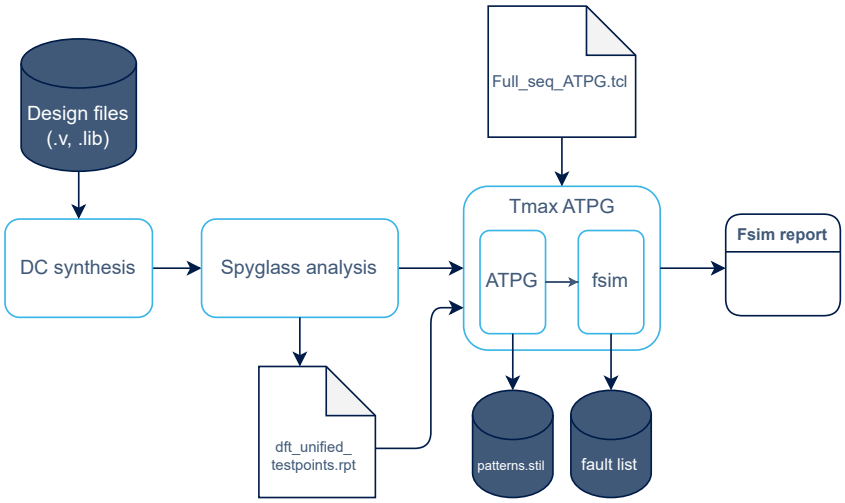


Figure 4.4: ATPG fault sim flow diagram

4.3.1 TestMAX ATPG working principle

TestMax ATPG is a software from Synopsys that delivers high-coverage test patterns with high efficiency in utilizing hardware resources. The typical TMAX workflow Figure 4.5 is composed of four main blocks:

BUILD-T: in this section, the design libraries and gate-level netlist are read, and the design model is built. An in-memory design image is created for the ATPG algorithm to run the simulations. Here, some Design Rule Check (DRC) rules are checked for any violations in the netlist (N rules) or in the built model (B rules). In this step, eventual black boxes can be defined.

DRC-T: in this step, the clock and the scan-chains information are verified. In these steps, some information, such as how many scan chain pins, how the scan mode is enabled in the design, what are the initialization sequences for the design, what are the clocks

and resets, and many more, is provided through a STIL file, which can be either provided from the external or it can be created directly by TMAX. Here are some rules checked:

- S rules check for scan chain or shift violations;
- C rules check for clocks or capture violations;
- Z rules check for internal tristate buses and bidirectional paths;
- R rules check for scan compression violations;
- X rules for combinational feedback loops;
- V rules for vector statements in the SPF;

Only if the DRC is passed, the TEST mode is entered to proceed with the ATPG procedures. In cases of severe violations, the error is reported, and the DRC run is aborted at the point of the error.

TEST-T: in this section, first the ATPG run should be prepared, so the fault list and the fault model are selected. After that, the ATPG procedure is launched. The goal of the ATPG is to create a set of test patterns that can be used by an ATE to distinguish the good machine from the faulty machine behavior. When launching the ATPG run, TMAX will cycle through all the ATPG engines enabled; different choices of engines can be chosen:

- **Basic scan:** it gives a very fast way to perform ATPG, giving fast results on a full scan design with minimum non-scan cells. Basic-scan patterns are generated by combinational ATPG, and they contain a scan load, a force of all primary inputs, a measure of all primary outputs, a clock pulse, and a scan unload.
- **Fast-Sequential:** this is good for design with some non-scan logic or some shadow logic around memories, guaranteeing better results. In this case, it allows up to 10 capture clock pulses during the capture phase to exercise deep sequential logic. In this case, at each cycle, it contains a scan load, a force of all primary inputs, a clock pulse, and at the end, the last clock cycle contains a single measure of the primary outputs and a single scan unload.
- **Full-sequential:** this is the engine used during this thesis work and is recommended to test full sequential logic; this mode supports multiple capture cycles between scan load and unload (no limit imposed), increasing the test coverage in sequential design. In this case, the patterns are generated in full sequence through a non-threaded ATPG.

Then, post ATPG, the coverage results can be analyzed, and additional measures can be implemented to increase the test coverage by trying to move faults from the ND class

to the DS one[9]. The generated patterns can also be exported in different formats (binary, STIL, WGL) for later usage.

VERIFICATION: Finally, the generated patterns can be tested through a Verilog simulator, which is created automatically by TMAX through the command `write_testbench` upon the patterns saved in the STIL file previously created. This testbench applies the test stimulus to the DUT and checks the responses against the expected ones. It can be used in combination with Xcelium to print the waveform responses of the device being tested. [11]

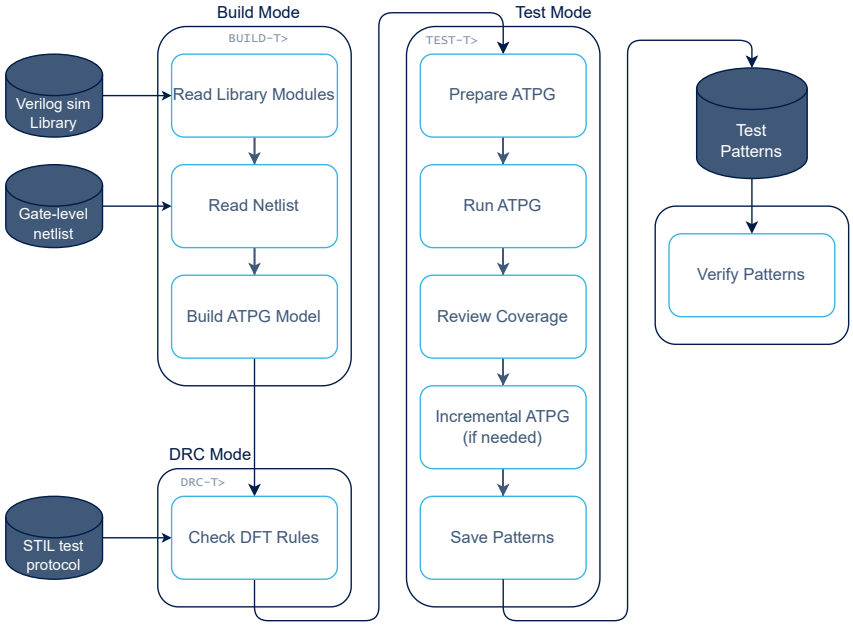


Figure 4.5: TMAX ATPG design flow

4.3.2 ATPG Fault simulation

The ATPG approach abandons the previous convoluted simulation environment for a simpler approach, which uses only one software. For this purpose, Synopsys TestMax has been adopted to both perform the ATPG and the fault simulation based upon that set of patterns. To start the fault simulation `run_ATPG.sh` must be run on the command shell and automatically starts TMAX in batch mode, running the `Full_seq_ATPG.tcl` script. At the end of the simulation, a report called `atpg_logfile` shows the faults and coverage metrics.

[9]Consult Section 2.3.4 for mode detail

run_ATPG.sh

```
#!/bin/bash

SCRTIPT="Full_seq_ATPG.tcl"

tmax2 "$SCRTIPT" -shell
```

Full_seq_ATPG.tcl

```
#############################################################################
# Mauro Lubrini
# Date: 5/06/25
# Description: Tcl script to perform full_sequential_ATPG on the defined
# DESIGN_TOP module. SpyGlass test points are inserted as additional PO
#############################################################################
set NETLIST "../pd/synth/cva6_synth.v"
set TECHLIB1 "../pd/synth/tech/NangateOpenCellLibrary.v"
set TECHLIB2 "../pd/synth/tech/NangateOpenCellLibrary_fixed.v"
set DESIGN_TOP "ex_stage_16_864949"
#set DESIGN_TOP "alu__864949"

set in_file_path "../spyglass/spyglass-1/$DESIGN_TOP/dft/dft_dsm_random_resistance/spyglass_reports/dft/dft_unified_testpoints.rpt"
set out_file_path "./connected_nets.txt"
set connected_nets ""


# -- Basic ATPG command sequence
set_messages -log atpg_logfile -replace

# -- Read design and libraries
read_netlist $NETLIST

read_netlist $TECHLIB1 -library
read_netlist $TECHLIB2 -library

#############################################################################
# -- tp insertion and connection to PO
set file_i_id [open $in_file_path r]

while {[gets $file_i_id line] != -1} {

        #Select only the lines containing the tp list (they start with a number)
        if {[regexp {^[0-9]} $line]} {
        if {"$line" ne ""} {

```

```
37                  #Exteract only the net path
38                  set net "[lindex [split $line] 6]"
39          puts "Debug-----------------------------"
40          puts "$net"
41          append connected_nets "$net\n"
42
43                  # connect tp points to po
44                  add_net_connections po "$net"
45              }
46          }
47  }
48
49  #Print output files (for debug purpuses)
50  set file_o_id [open $out_file_path w]
51  puts $file_o_id $connected_nets
52
53  #Close files
54  close $file_i_id
55  close $file_o_id
56
57  ##########################################################################
58
59  # -- Build design model
60  set_build -nodelete_unused_gates
61  run_build_model $DESIGN_TOP
62  #to check the number of connections to PO
63  report_net_connections > report_connections.txt
64
65
66  # -- Define clocks and pin constraints
67  add_clocks 0 clk_i
68  add_clocks 1 rst_ni
69
70  # -- perform DRC checks
71  run_drc
72  report_rules -fail
73
74  # -- set the number of threads
75  set_atpg -num_threads 0
76  set_simulation -num_threads 0
77
78  # -- set simulation options
79  set_faults -model stuck
80  add_faults -all
81  set_patterns -internal
82  set_atpg -full_seq_atpg -patterns 1000
83  run_atpg -auto_compression full_sequential_only
    ↪
84
```

```
# -- save fault list and patterns
write_pattern patterns.stil -format STIL -replace
write_patterns patterns.bin -replace
set_faults -fault_coverage
report_summaries sequential_depth
report_faults -all -verbose
```

This script performs the full sequential ATPG on the module under test designated as `DESIGN_TOP`. It begins with defining the design netlist, technology libraries, and the hierarchical top module. It then defines the path to the SpyGlass designated test point report (`dft_unified_testpoints.rpt`) and an output file (`connected_nets.txt`) which lists the inserted strobe points connected to primary outputs to verify correct implementation in the design. Then, after reading the design files, the script continues with the test point insertion section: a loop reads and parses the entire SpyGlass test points list, looking at each cycle for a new observation point and its hierarchy path, saving it in the `$net` variable. It then updates the `connected_nets` list and, with the command `add_net_connections po`, creates a new primary output connected directly to the point indicated by the `$net` path. Next, once the new test points have been inserted, the design is built and the model is run. With the command `report_net_connections > report_connections.txt`, a new file reporting the added primary output connection is created and is used as a secondary check. The script continues with defining the clock and reset constraints, setting `add_clocks 0 clk_i` and `add_clocks 1 rst_ni` to specify the off state of the two signals. Then, after performing the DRC, the simulation options are set, starting with the number of threads, which in this case are set to 0 (single thread), as full sequential ATPG requires single-threaded execution. Next other simulation option are set: `set_faults -model stuck`: select the stuck-at fault model; `add_faults -all`: adds faults at all potential fault sites in the design to the fault list; `set_patterns -internal`: specifies that the pattern source is internally generated; `set_atpg -full_seq_atpg -patterns 5000` it enables the full-sequential ATPG algorithm and sets the maximum number of patterns generated during ATPG, then it terminates the simulation; `run_atpg -auto_compression full_sequential_only`: it forces the full-sequential pattern generation. Then the final part of the script sets the simulation reports and writes the STIL file containing the test patterns generated during the ATPG; this file can be useful for independent fault simulation using an external set of input test patterns.

4.3.3 ATPG with random test point insertion

Figure 4.6 depict the procedure for running ATPG with randomly selected strobe points which is very similar to the previous one, with only minor adjustments. Since the random strobe selection requires a fault list, if one is not already available, this step can only be performed after completing a fault simulation using the SpyGlass generated strobe points. As before, the simulation is started by launching `run_ATPG.sh`. The script `randomize_strobe_selection.py` is first invoked to parse the fault list and selects a number of random points. Then, TMAX is launched running the `Full_seq_ATPG.tcl`

script. This time, instead of using the SpyGlass selected points, the script reads the Python generated strobe list and adds these as additional primary outputs during the fault simulation.

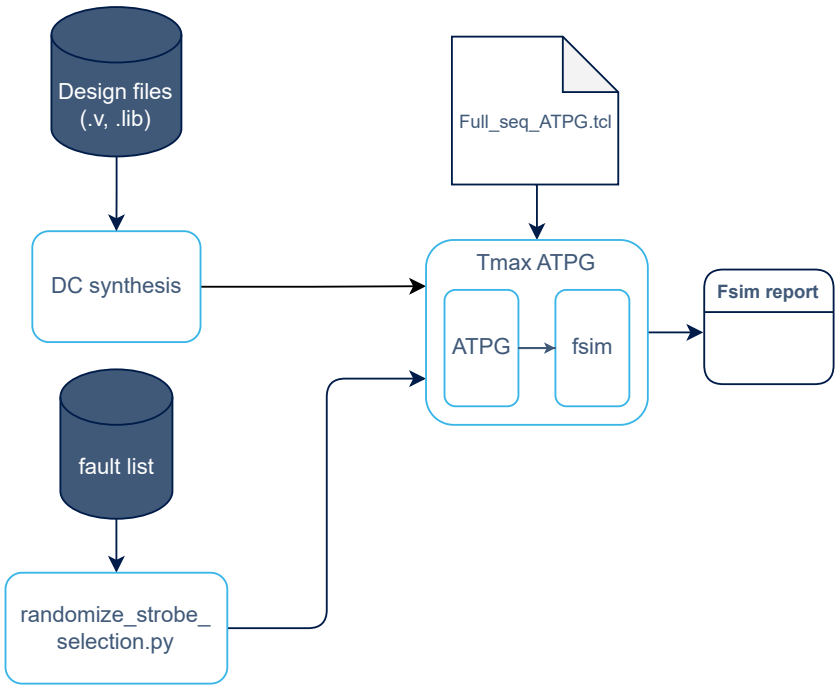


Figure 4.6: ATPG fault sim flow diagram

run_ATPG.sh

Starting point to execute the fault simulation process. Before launching TestMAX, the random test point selector script is run to prepare a list of random test point to be later be adopted.

```
#!/bin/bash

SCRTIPT="Full_seq_ATPG.tcl"

python3 randomize_strobe_selection.py
tmax2 "$SCRTIPT" -shell
```

randomize_strobe_selection.py

The `randomize_strobe_selection.py` script is used to read the fault list generated during the first fault-sim and then select randomly N-strobe points from that list.

```
"""
File: strobe.py
Author: Mauro Lubrini
Date: 27/05/2025
Description: This script is used to read the fault list generated during
the first fault-sim and then select randomly N-strobing points from that
list (N=STROBE_NUMBER)
"""

import random

# Variables and constants
INFILE_PATH = "./fault_list.txt"
OUTFILE_PATH = "./random_strobe_list.txt"
STROBE_NUMBER = 200

fault_list = []
strobe_list = []
repeated_cells = []
j=0

infile = open(INFILE_PATH, "r")
lines = infile.readlines()

# Extrapolate only the port pins from the fault list
for i, line in enumerate(lines):
    elements = line.split()
    fault_list.append(elements[2] + "\n")

list_length=(len(fault_list))

# Randomize the selection of the ports pins to be used as strobe points
if (list_length>=STROBE_NUMBER):
    print("Random Indexes selected:----------------------------")

    while j < STROBE_NUMBER:
        rand_index = random.randrange(0, list_length-1, 1)
        if (("/".join((fault_list[rand_index].split("/") )[:-1])) not in
         ↪  repeated_cells):
            strobe_list.append(fault_list[rand_index])
            repeated_cells.append("/".join((fault_list[rand_index].split
             ↪  ("/") )[:-1]))
            j += 1
            print("/".join(fault_list[rand_index].split("/")[:-1]))
            print(rand_index)
else:
    print("ERROR: the faultlist is smaller than the number of strobe we
     ↪  want to insert")
```

```
46      print("       please select a value for STROBE_NUMBER lower than",
        ↪  list_length )
47
48  # Create the list of random strobing points
49  with open(OUTFILE_PATH, "w") as outfile:
50      outfile.writelines(strobe_list)
51
52  infile.close
```

The script begins by importing the Python's standard library module `import random`. Next, it defines `INFILE_PATH` which is the input fault list path, `OUTFILE_PATH` which is the output file path, and finally `STROBE_NUMBER` which is the variable that sets the number of random locations to be selected from the fault list. A first loop parses the fault list, extrapolating each fault injection location its corresponding design path, and saving it in a temporary list (`fault_list`). Finally, a second loop is used to select random entries from `strobe_list`, which are later written in the output text file. Since the fault list reports the faults injected at the pins (input or output ports) of cells, in case the random selection picks two locations of the same cell (for example, it selects both an input and the output port of the same cell), the fault simulation stops due to an error. The workaround to this problem is to create at the beginning an empty list (`repeated_cells`) and at each location randomly picked, the name of the visited cells is appended inside the list, so that the next time a different port of the same cell is selected, this choice is discarded and a new random selection is performed. In conclusion, the script writes a text file, `random_strobe_list.txt`, listing a random subset of nets in the top-level module's design hierarchy. This file is later read by `Full_seq_ATPG.tcl` to insert those nets as additional test points for the fault simulation.

Full_seq_ATPG.tcl

```
1  ##########################################################################
2  # Mauro Lubrini
3  # Date: 5/06/25
4  # Description: Tcl script to perform full_sequential_ATPG on the defined
5  # DESIGN_TOP module. Random test points are inserted as additional PO
6  ##########################################################################
7
8  set NETLIST "../../pd/synth/cva6_synth.v"
9  set TECHLIB1 "../../pd/synth/tech/NangateOpenCellLibrary.v"
10 set TECHLIB2 "../../pd/synth/tech/NangateOpenCellLibrary_fixed.v"
11 set DESIGN_TOP "ex_stage_16_864949"
12 #set DESIGN_TOP "alu__864949"
13 #set DESIGN_TOP "ex_stage_16_864949"
14
15 set in_file_path "./random_strobe_list.txt"
16 set connected_nets ""
17
```

```
18
19 # -- Basic ATPG command sequence
20 set_messages -log atpg_logfile -replace
21
22 # -- Read design and libraries
23 read_netlist $NETLIST
24
25 read_netlist $TECHLIB1 -library
26 read_netlist $TECHLIB2 -library
27
28 ##############################################################################
29 # -- tp insertion and connection to PO
30 set file_i_id [open $in_file_path r]
31
32 while {[gets $file_i_id line] != -1} {
33
34         if {"$line" ne ""} {
35
36         #Exteract only the net path
37         puts "Debug----------------------------"
38         puts "$line"
39         append connected_nets "$line\n"
40
41         # connect tp points to po
42         add_net_connections po "$line"
43         }
44
45 }
46
47 #Close files
48 close $file_i_id
49
50 ##############################################################################
51
52 # -- Build design model
53 set_build -nodelete_unused_gates
54 run_build_model ex_stage_16_864949
55 #to check the number of connections to PO
56 report_net_connections > report_connections.txt
57
58
59 # -- Define clocks and pin constraints
60 add_clocks 0 clk_i
61 add_clocks 1 rst_ni
62
63 # -- perform DRC checks
64 run_drc
65 report_rules -fail
66
```

```
67 # -- set the number of threads
68 set_atpg -num_threads 0
69 set_simulation -num_threads 0
70
71 # -- set simulation options
72 set_faults -model stuck
73 add_faults -all
74 set_patterns -internal
75 set_atpg -full_seq_atpg -patterns 5000
76 run_atpg -auto_compression full_sequential_only
   ↪
77
78 # -- save fault list and patterns
79 write_pattern patterns.stil -format STIL -replace
80 write_patterns patterns.bin -replace
81 set_faults -fault_coverage
82 report_summaries sequential_depth
83 report_faults -all > fault_list.txt
```

This script is the same as the one for the SpyGlass selected test points. The only differences are in the test point insertion section, where this time, instead of the SpyGlass test point report, the `random_strobe_list.txt` is parsed.

4.3.4 Hierarchy flattening

From an analysis of the suggested test points returned by SpyGlass analysis (Appendix B.1.1), it is evident that, inside the Execution Stage, the tools target mainly the ALU module. As already exposed in the introduction to the CVA6 architecture in Section 2.2.4, the ALU module is a purely combinational circuit, while the execution stage, having multiple submodules equipped with an FSM machine, is a sequential circuit. This difference is likely affecting SpyGlass capability to identify optimal locations for observing the internal state of the system. Indeed, examining the SpyGlass test point report on the execution stage module reveals that most of the selected test points are still related to the ALU rather than other modules. This leads to the hypothesis that SpyGlass algorithms are more effective on combinational circuitry. To validate this idea, a new approach has been followed: all the execution stage design has been flattened and then separated by grouping the combinational part from the sequential one. So, as the final result, depicted in Figure 4.7, there will be only two new modules inside the execution stage: one composed only by combinational circuitry (like buffer, logic gates, multiplexers) and another one sequential, containing only memory elements (like registers, flip flops, lathces).

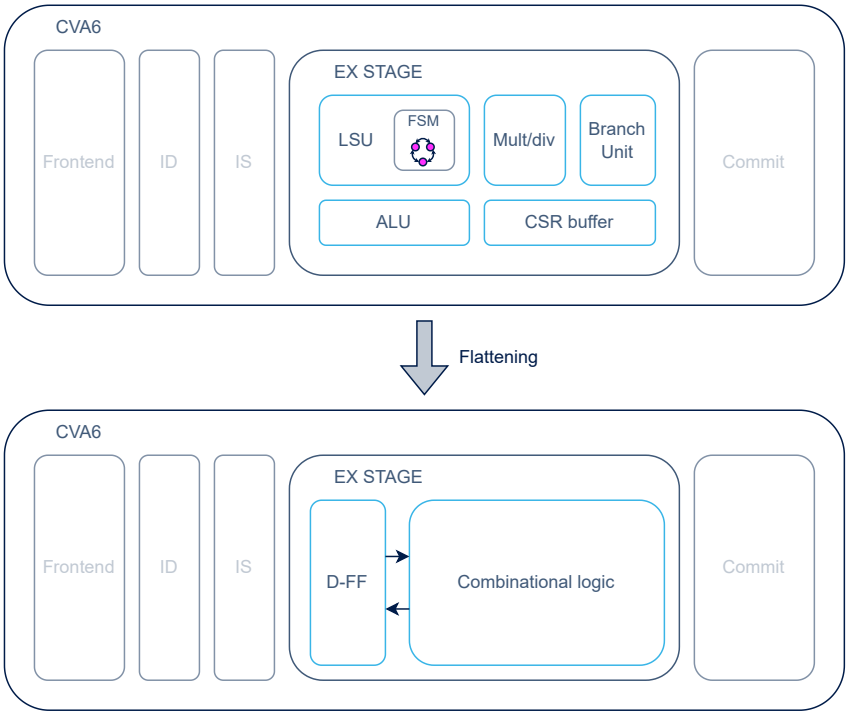


Figure 4.7: Flattening of the execution stage

By doing so, it is possible to make SpyGlass analyze the whole execution stage logic as it is a single big combinational block. Therefore, the optimal test point analysis will be executed only on the combinational block, while fault simulation through ATPG is performed throughout the entire execution stage, including the sequential block as well.

Synthesis flattening

The flattening process is performed during synthesis using Design Compiler with a few additional line inserted in the compilation step. In the following is represented a piece of code from `cva6_synth.tcl` showing the additional lines of commands added.

```
80 #========================================
81 # COMPILE THE GATE LEVEL DESIGN
82 #========================================
83 compile_ultra -no_boundary_optimization
84
85 change_names -rules verilog -hierarchy
86
87
88 #Perform flattening
89 current_instance ex_stage_i
```

```
ungroup -all -flatten -prefix C_ -simple_names
group -logic -design_name comb_block
```

After launching `compile_ultra`, the flattening of the execution stage takes place. With `current_instance` the `ex_stage_i` module is selected, and with the command `ungroup` all the modules inside the `ex_stage_i` instance are flattened out. In particular, to recognize more easily the components being flattened, they have all been assigned with "C_" as a prefix. Next, with the command `group -logic` all, the combinational logic is grouped together under the name of `comb_block`.

4.4 Post-synthesis observation points insertion and fixed functional test patterns fault sim

To have a more equal and coherent comparison between simulations, the use of a univocal set of test patterns is preferred. Holding the test patterns fixed in each scenario, while running the three simulations, it shows which one of the two strobing methods works best under the same simulation conditions, performing a comparison uninfluenced by any possible asymmetry in the simulation run. The main workflow remains the same as the ATPG one previously described, with the only difference that now fault simulations are performed on the same set of test patterns formerly generated from an ATPG campaign and exported as a STIL file (Figure 4.8). This pattern database is imported directly into TMAX during the fault simulation flow and used without eliminating the need to run a new ATPG campaign. The fault-simulation procedure itself remains unchanged: test points are inserted (or, in the no-test-points case, a masking procedure is applied), and the simulation is then executed without any prior ATPG run.

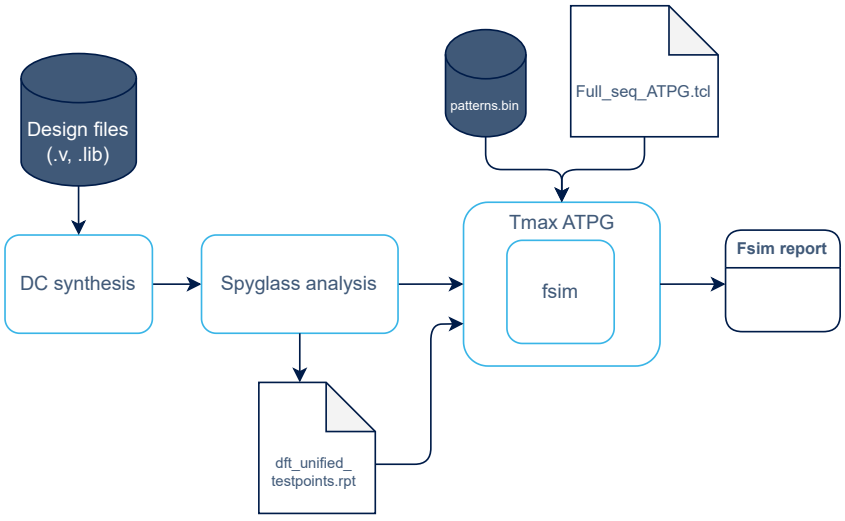


Figure 4.8: Fault simulation flow

The same procedure is also followed for the randomly selected test point fault simulation. The workflow (Figure 4.9) is identical to that of the ATPG process, with the only difference being that now the fault simulation uses the same test patterns across all three fault simulations[10].

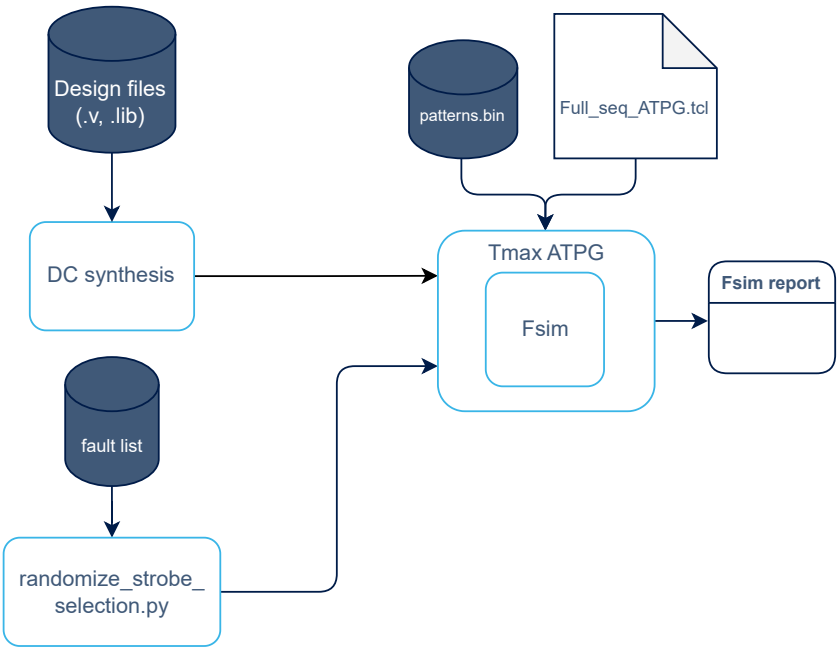


Figure 4.9: Fault simulation flow

Full_seq_ATPG.tcl

This code is used to run a sequential fault simulation using an external set of input test pattern

```
1 ##########################################################################
2 # Mauro Lubrini
3 # Date: 5/06/25
4 # Description: Tcl script to perform the fault simulation given a set of
  ↪  input patterns
5 ##########################################################################
6 set NETLIST "../pd/synth/cva6_synth.v"
7 set TECHLIB1 "../pd/synth/tech/NangateOpenCellLibrary.v"
```

[10]For "three simulations" is intended the fault simulation on the raw design, the fault simulation on the design with SpyGlass selected test points and the fault simulation executed with a random set of test points.

```
set TECHLIB2 "../pd/synth/tech/NangateOpenCellLibrary_fixed.v"
#DESIGN_TOP: is the top module on which I want to perform my ATPG
#DUT is the module on which I run spyglass on, so the one in which I want to insert the tp
set DESIGN_TOP "ex_stage_16_864949"
set DUT "ex_stage_16_864949"
#set DESIGN_TOP "alu__864949"

set in_file_path "../spyglass/spyglass-1/$DUT/dft/dft_dsm_random_resistance/spyglass_reports/dft/dft_unified_testpoints.rpt"
set out_file_path "./connected_nets.txt"
set connected_nets ""


# -- Basic ATPG command sequence
set_messages -log atpg_logfile -replace

# -- Read design and libraries
read_netlist $NETLIST

read_netlist $TECHLIB1 -library
read_netlist $TECHLIB2 -library

##########################################################################
# -- tp insertion and connection to PO
set file_i_id [open $in_file_path r]

while {[gets $file_i_id line] != -1} {

        #Select only the lines containing the tp list (they start with a number)
        if {[regexp {^[0-9]} $line]} {
        if {"$line" ne ""} {

                #Exteract only the net path
                set net "[lindex [split $line] 6]"
        append connected_nets "$net\n"

                # connect tp points to po
                add_net_connections po "$net"
            }
        }
}

#Print output files (for debug purpuses)
set file_o_id [open $out_file_path w]
puts $file_o_id $connected_nets

```

```
#Close files
close $file_i_id
close $file_o_id

################################################################################

# -- Build design model
set_build -nodelete_unused_gates
run_build_model $DESIGN_TOP
#to check the number of connections to PO
report_net_connections > report_connections.txt

# -- Define clocks and pin constraints
add_clocks 0 clk_i
add_clocks 1 rst_ni

# -- define scan chains & STIL procedures, perform DRC checks
run_drc
report_rules -fail

# -- perform the fault simulation using external patterns
set_faults -model stuck
add_faults -all
set_patterns -ext patterns.bin
run_fault_sim -sequential

# -- reports
set_faults -fault_coverage
report_patterns -profile
report_faults -summary
report_summaries
#report_faults -all -verbose
```

The script is very similar to the one previously described in the past section, it has a central loop which reads an external file containing the list of strobe points to insert and it connects those points to additional primary outputs. What changes is the commands used to perform the fault simulation: starting with `set_faults -model stuck` the stuck-at fault model is selected, then with `add_faults -all` faults are added at all potential fault sites in the design to the fault list. Next, with `set_patterns -ext patterns.bin` the binary version of the STIL file is imported, and finally with `run_fault_sim -sequential` the fault simulation is started specifying the use of a simulation algorithm in sequential mode. The patterns generated during ATPG have a trend of application which appears to be an alternating sequence between primary output and primary input; sometimes between the two toggles with `V { "clk_i"=P; }`[11] and more rarely the reset signal with `V { "rst_ni"=P; }`. The fact that the clock signal doesn't switch at each input pattern

[11] P is defined in the WaveformTable as: {P{ '0ns' D; '50ns' U; '80ns' D; }}.

application leads to believe that the ATPG algorithms with those patterns try to target mainly the combinational logic.

Masking.tcl

Since the patterns generated during ATPG have a length that depends on the number of primary outputs of the design under test, the STIL file generated from the ATPG with test points (so with additional PO) and the one without test points have different pattern lengths, leading to an incompatibility that, if the same patterns are adopted, generates an error during the fault simulation. So, the workaround to this problem was to always consider the design with the additional PO and its set of patterns, then to fault simulate the design without test points, a masking of the supplementary PO is performed; `masking.tcl` does exactly this, it retrieves the added PO from `report_connections.txt` and masks them with the command `add_po_mask`. In this way, even if the STIL files account for an extended set of PO, only the standard outputs of the DUT are used to compute the fault analysis.

```
################################################################################
# Mauro Lubrini
# Date: 5/06/25
# Description: Tcl script to perform the masking of the added PO
################################################################################

set in_file_path "report_connections.txt"
set file_i_id [open $in_file_path r]

while {[gets $file_i_id line] != -1} {

    if {"$line" ne ""} {
    #Exteract only the net path
    set port "[lindex [split $line] 5]"
    puts "Debug-----------------------------"
    puts "$port"

    # connect tp points to po
    add_po_mask "$port"
    }

}

#Close files
close $file_i_id
```

Chapter 5

Results

In this Chapter, the focus is on analyzing the results obtained from the fault simulations executed during the different workflows described in Chapter 3.

Is worth to remember that the main goal of this thesis is to assess the effectiveness of static circuit analysis tool, like SpyGlass, in identify optimal locations to insert test points for the sake of improving test coverage metric. So to validate this and confirm the effectiveness of the proposed methodology, in each different study cases, the SpyGlass approach should outperform the random one, otherwise the methodology cannot be considered a valid solution for the test point insertion problem. To validate the proposed methodology, the SpyGlass-guided approach must outperform the random insertion method Failure to demonstrate superior performance would invalidate the method as a viable solution for the test point insertion problem.

For each proposed study cases, three tests are performed: one for SpyGlass suggested test points, one for the case with no test points, and one for the randomly selected ones; so in most of the graphs, three curves, representing the three different test cases, are showed, with each curve having a specific color code: the dark blue curve represents the results obtained with the insertion of the SpyGlass test points, the light blue curve, instead, represents the results obtained without any additional test points, while the pink one is related to the randomly selected test points case. For the random case, to obtain statistically fair and representative data, allowing for meaningful analysis of variability and significance, ten different simulations, each one with new randomly selected test points, were performed and then merged by computing the mean value. In addition to that, to have a better statistical representation of the random case, the standard deviation between the ten simulation run has been computed. This statistical spread is then depicted on the graph using conventional visualization methods, such as error bars or a shadowed band around the mean. The formula used to compute the standard deviation is:

$$S = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}|A_i - \mu|^2} \tag{5.1}$$

Where N is the number of element in the dataset, A_i is the i-t element of the data set and μ is the mean value defined as:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} A_i \tag{5.2}$$

In this case, where the standard deviation is plotted, the dataset is composed by the test coverage computed in that point by the N simulation run.

5.1 SBST with observation point inserted at RTL level

In this set of simulations, SpyGlass analysis has been performed at the RTL level to look for the optimal location to insert the observation points (more details in Section 4.2) and the DUT has been exercised by running an SBST simulation where a sequence of random assembly instructions, selected from the CVA6 ISA (rv64imafdc[1]), has been executed. Finally using the system response obtained from the SBST procedure, a fault simulation has been performed through a tool from Synopsys called VC-Z01X.

5.1.1 Observation points on ALU

For this run of simulations, the test points were allowed to be inserted only in the ALU module, which is contained inside the Execution Stage. Graph 5.1 shows the two curves that plots the coverage obtained with the test points inserted with SpyGlass (blue curve), and the coverage obtained by randomly selecting the same number of test points (pink curve); on the y-axis it is represented the test coverage, while on the x-axis the number of supplementary test points. In this case, five different simulations with 0, 50, 100, and 169 test points have been computed. By analyzing the RTL files directly, SpyGlass was unable to detect more then 169 optimal test locations. The graph features error bars for the random curve, showing the standard deviation computed on the 10 test coverage results computed with Equation 5.1. In Table 5.1, the numerical results of the latter fault simulations with the difference (delta Δ) between the two are shown.

Graph 5.1, shows a clear separation between the SpyGlass and the random one, with the latter having a higher observation coverage. Indeed, standard deviation of the random case never overlap the SpyGlass curve, statistically confirming the separation between the two curves. This seems to affirm the effectiveness of the method as it reached 47.08% of maximum test coverage, using 169 test points, with an improvement over the random approach of 6.58%.

[1] rv64 means that the ISA instructions supports 64-bit address space, while the following letters indicate the ISA extension instruction types: “i” stands for integer, “m” for integer multiplication and division, “a” for atomic instructions, “f” for single-precision floating-point, “d” for double-precision floating-point and “c” for 16-bit compressed instructions

Inserted test points	*0*	*10*	*50*	*100*	*169*
SG tp cov [%]	36.84	43.73	44.21	45.13	47.08
Random tp cov [%]	36.84	38.20	38.63	38.15	40.50
Δ [%]	0	5.53	5.58	6.98	6.58

Table 5.1: Table representing the coverage results of Fig.5.1

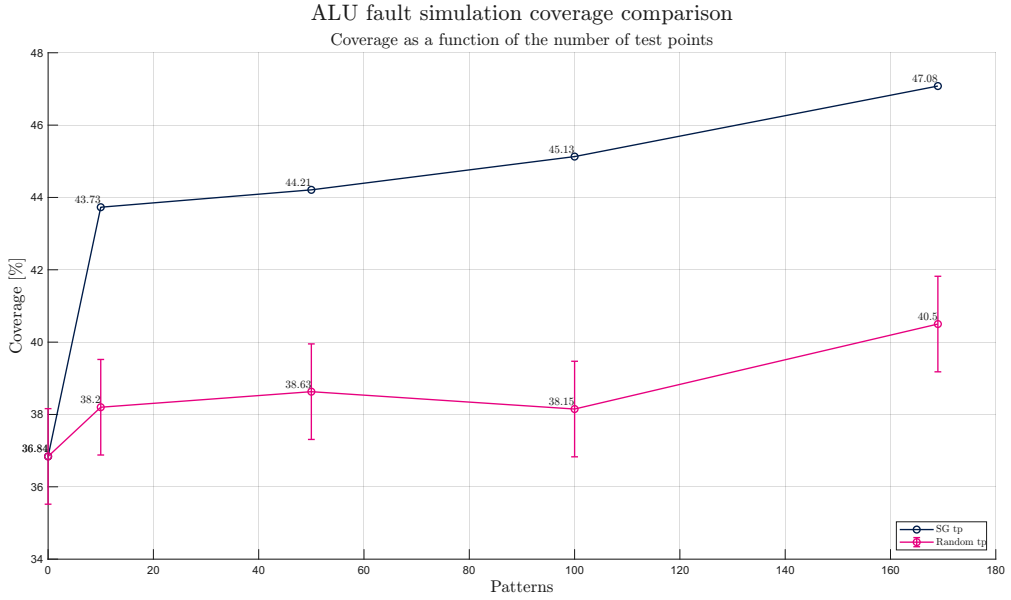


Figure 5.1: Coverage comparison between SG and random test points insertion in the ALU

5.1.2 Observation points on Execution Stage

For this run of simulations, the test points were allowed to be inserted in whole Execution Stage module. Graph 5.2 shows the two curves that plot the test coverage obtained with SpyGlass suggested test points (blue curve), and the coverage obtained by randomly selecting the same number of test points (pink curve). On the y-axis it is represented the test coverage, while on the x-axis the number of supplementary test points has been inserted. In this case, five different simulations with 0, 50, 100, and 250 test points have been computed. In this case, by analyzing the RTL files directly, SpyGlass was unable to detect more then 250 optimal test locations. The graph features error bars showing the standard deviation of each data set computed as in Equation 5.1. In Table 5.2, the numerical results of the latter fault simulations with the difference (delta Δ) between the two are shown.

By reading Graph 5.2, the following consideration can be made: the first being that the SpyGlass curve largely overlap within the standard deviation band of the random test

point insertion case, meaning that it is not possible to declare one method better than the other; the second is that the maximum coverage reached during fault simulation is extremely low for both simulations. The coverage value obtained in these simulations is not sufficiently high to consider these results significant and reliable. Most likely, a problem lies in either the effectiveness of the stimulus applied to the DUT, which hides the differences between the two different simulations or in the effectiveness of the test point placement.

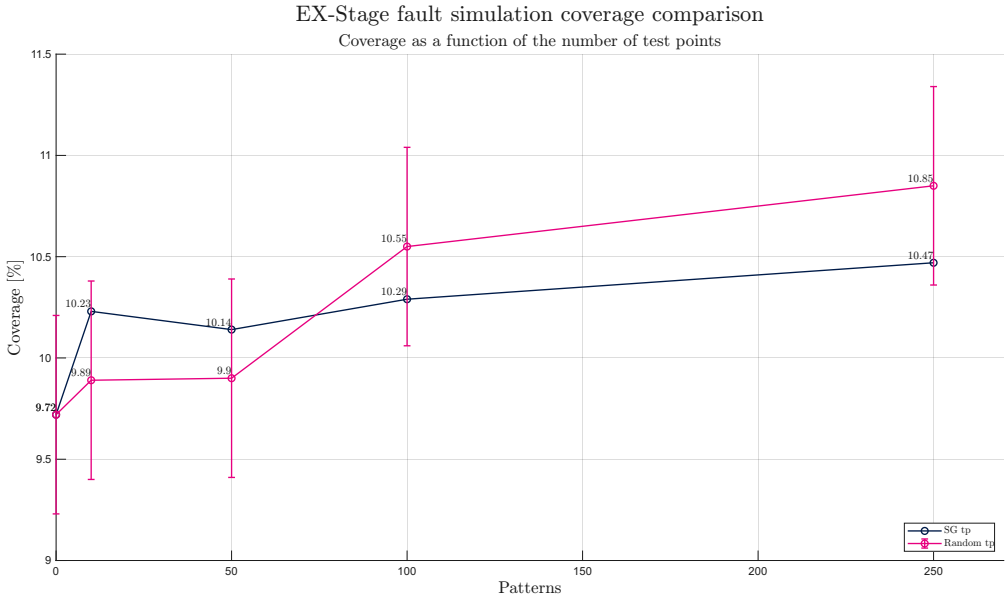


Figure 5.2: Coverage comparison between SG and random test points insertion in the ES

Inserted test points	*0*	*10*	*50*	*100*	*250*
SG tp cov [%]	9.72	10.23	10.14	10.29	10.47
Random tp cov [%]	9.72	9.89	9.90	10.55	10.85
Δ [%]	0	0.34	0.24	−0.26	−0,38

Table 5.2: Table representing the coverage results of Fig.5.2

5.2 ATPG with observation points inserted at netlist level

In this set of simulations, a new approach has been explored, running an ATPG campaign to compute the input patterns to test the DUT. So the option devised was to rely on the capability of TestMax to generate, using full sequential algorithms, a sufficiently good

set of test patterns capable of raising to higher values the test coverage obtained during fault simulations. In the following sections, three case studies were assessed: the first one is a fault simulation executed with the test points allowed to be inserted only within the ALU, the second one with the test points inserted within the whole Execution Stage module, and lastly, after a separation of the combinational and sequential logic, a fault simulation was performed on the entire Execution Stage but with the test points restricted only within the combinational circuit. In each different scenarios a total of 200 test points have been implemented while the target number of ATPG generated test pattern has been set to 5000; however, depending on the location of the inserted test points, many ATPG campaigns stalled much earlier. The random test point insertion curve was calculated by averaging the test coverage from ten separate fault simulations, each using a new, randomly selected set of test points. For all the following study case, the test patterns has been applied directly to the Execution Stage PI and the system response read at its PO.

Regarding the ALU, since it is a simple and purely combinational block, regardless the usage of additional test points, was able to reach nearly 100% in test coverage so, for this reason, no further investigation has been carried out on the ALU module itself since it would be impossible to distinguish any discrepancies between the different approaches.

5.2.1 Observation points on Execution Stage

During these simulations, the additional test points were allowed to be inserted in all the Execution Stage module. Graph 5.3 represents on y-axis the test coverage, while on x-axis, the number of test patterns generated during ATPG campaigns. It worth remembering that in this study case, the ATPG runs were executed on each individual netlist relative to that specific approach with the eventual test point already been implemented. The target value for the generated patterns was set to 5000, but accordingly to the different configuration of the DUT, some ATPG campaign stalled before reaching that value; in this case, the limiting simulation was the random test point insertion which stopped at around 4500 patterns[2]. Contrary to expectation, Graph 5.3 demonstrates that test points selected by SpyGlass provided negligible improvement in test coverage, essentially overlapping with the base case, while surprisingly, the random selection of test points yielded a slightly higher coverage with a gap to the SpyGlass curve of 4.63% after 4500 test patterns.

[2]To avoid misunderstanding, it's important to clarify that "random simulation" refers to the mean value computed across ten different sets of simulations. This means the maximum number of patterns reached by the random simulation is limited by the shortest simulation among those ten.

	SG tp [%]	*Random tp* [%]	*No tp* [%]
Maximum coverage	67.53	72.16	67.68
Δ from max NO tp	−0.15	4.48	-
Δ from max RD tp	−4.63	-	-

Table 5.3: Table representing the coverage results of Fig.5.3

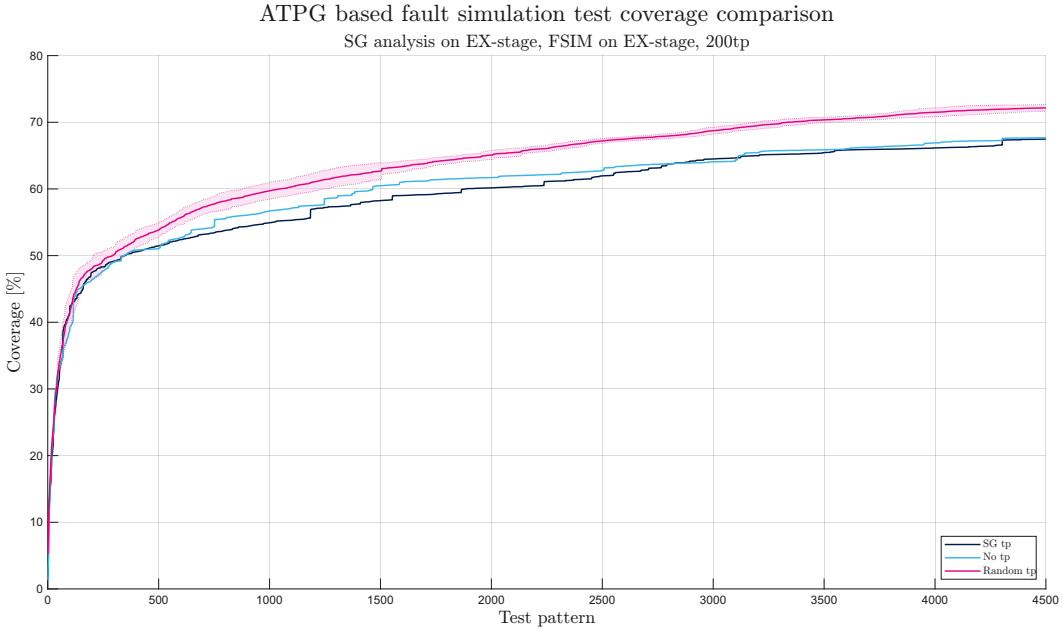


Figure 5.3: ATPG based fault simulation of the EX-Stage, with TP inserted in the EX-Stage

5.2.2 Observation points on ALU

In this simulation campaign, the test points insertion has been allowed only in the ALU to assess its weight and relevance in the test coverage computation of the Execution Stage. Given that SpyGlass primarily targets locations within the ALU, while ignoring others, it is worth investigating whether this placement strategy is justified by the ALU containing the majority of hard-to-observe faults. This confirmation would validate the high concentration of test points in that specific module. Proving this would validate the effectiveness of adding supplementary test points only confined inside the ALU and their contribution in the overall Execution Stage coverage computation. Graph 5.4 shows on the y-axis the test coverage, while on the x-axis the number of test patterns generated during the ATPG campaign. The pattern target value was set to 5000, but the comparison has been limited by the SpyGlass run, which stopped at 4728 patterns. The results demonstrate that the ALU itself does not add any relevant benefits in the test coverage computation as

all three curves are overlap. This results demonstrate that test point insertion restricted exclusively within the ALU yields to no improvement in the test coverage metric; this holds true for both the SpyGlass and random insertion methods.

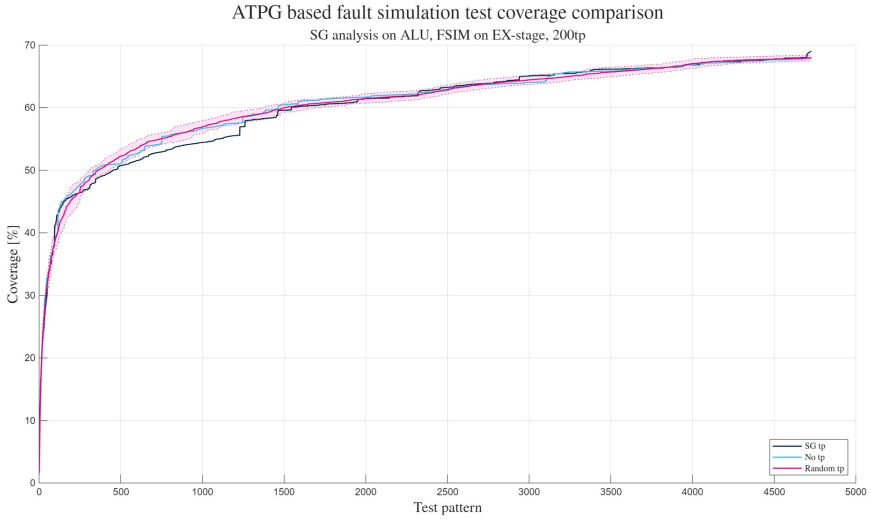


Figure 5.4: ATPG based fault simulation of the EX-Stage, with TP inserted in the ALU

	SG tp [%]	*Random tp* [%]	*No tp* [%]
Maximum coverage	69.02	67.98	67.88
Δ from max NO tp	1.14	0.1	-
Δ from max RD tp	1.04	-	-

Table 5.4: Table representing the coverage results of Fig.5.4

5.2.3 Observation points on flattened Execution Stage

The final evaluation of this ATPG approach was conducted on a flattened design (for more information see Section 4.3.4). The primary objective was to validate SpyGlass's effectiveness on combinational logic. With the Execution Stage fully flattened, SpyGlass is able to analyze the entire module's combinational logic rather than being limited to the ALU one, allowing for a better placement of the test points enhancing the test coverage evaluation. Graph 5.5 shows on the y-axis the test coverage, while on the x-axis the number of test patterns generated during the ATPG campaign. The target value for the generated patterns was set to 5000, but the comparison has been limited by the SpyGlass at 4482 patterns. The results show an interesting crossover behavior: up to 3000 test patterns, the trend mirrors previous findings with the random approach being superior

with respect to the SpyGlass one; but beyond 3000 patterns, ATPG algorithms seem to exploit the SpyGlass test points better, adding a significant contribution in increasing the fault coverage, making it surpass the random curve. So, beyond a critical pattern count, the optimal SpyGlass test points seem to show a slight but clear improvement over the random ones. While this gap is not impressive, the curve seems to have an upward trend, hopefully leading to a more significant divergence.

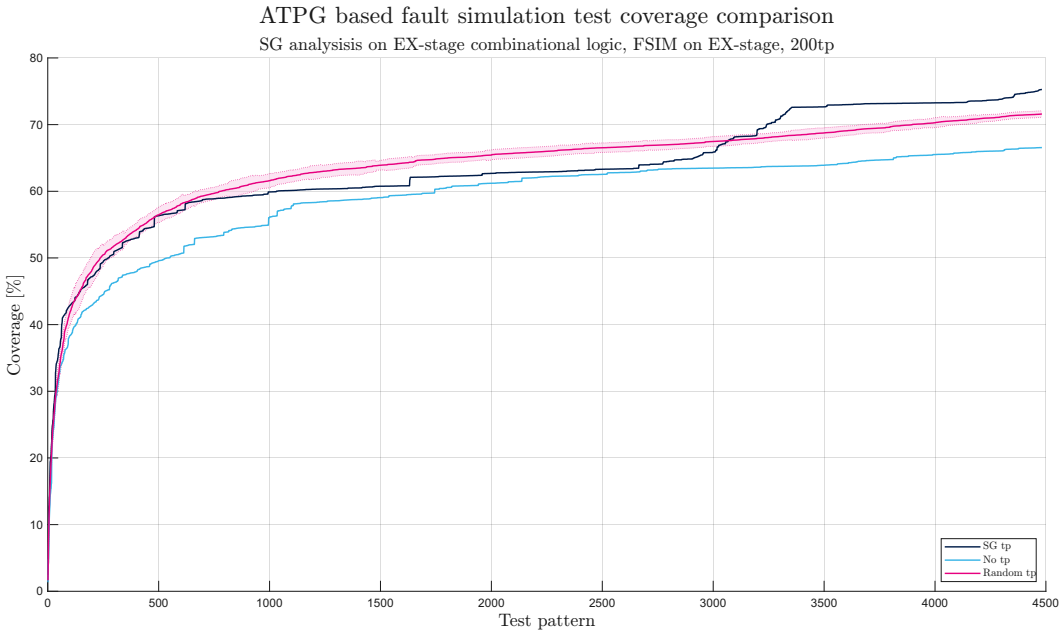


Figure 5.5: ATPG based fault simulation of the EX-Stage, with TP inserted in the EX-Stage combinational logic

	SG tp [%]	*Random tp* [%]	*No tp* [%]
Maximum coverage	75.27	71.59	66.56
Δ from max NO tp	8.71	5.03	-
Δ from max RD tp	3.68	-	-

Table 5.5: Table representing the coverage results of Fig.5.5

5.3 Fault simulation with ATPG test pattern, observation point at netlist level

This campaign test is very similar to the one executed before in Section 5.2.2, with the only difference being in the input stimuli adopted. The input patterns are kept fixed

for each of the three fault simulations within the same test scenario. This is achieved by exporting the generated patterns obtained from the previous ATPG campaigns and load them as external functional test patterns while launching the fault simulation. By holding the pattern set fixed, we can isolate the intrinsic effectiveness of the inserted test points, avoiding confounding from pattern-specific effects that might underuse them. The test patter chosen to run the fault simulation in each one of the following study cases, are taken from an ATPG campaign executed on a modified netlist containing a random set of test points. The reason behind this choice is to avoid having a set od test pattern polarized towards a specific test point configuration. Indeed, since at every new fault simulation run, a new set of test point is inserted, the specific test vectors adopted would not be created in any of the simulated configurations, being in this way a more neutral and general set of test patterns. These test patterns has been applied directly to the Execution Stage PI, while the system response has been read at its PO.

5.3.1 Observation points on ALU

Graph 5.6 shows on the y-axis the test coverage, while on the x-axis, the number of test cycles executed during the fault simulation. The simulations in this scenario confirm the results found previously in Section 5.2.2, where the additional test points inserted only in the ALU, whether SpyGlass or random test points insertion is used, brings no benefits in the improvement of the Execution Stage observation coverage. Zooming on the curves, it is possible to observe that for the whole length of the curves, the randomly selected test points have higher coverage with respect to the SpyGlass one, but the difference between the two curves is so minimal that it is impossible to state that one is better than the other. Moreover, comparing the latter two curves with the case without added test points, we can see how, for the purpose of incrementing the Execution Stage coverage, the test points inserted only in the ALU are quite ineffective.

	SG tp [%]	*Random tp* [%]	*No tp* [%]
Maximum coverage	67.5	67.18	66.83
Δ from max NO tp	0.67	0.35	-
Δ from max RD tp	0.32	-	-

Table 5.6: Table representing the coverage results of Fig.5.6

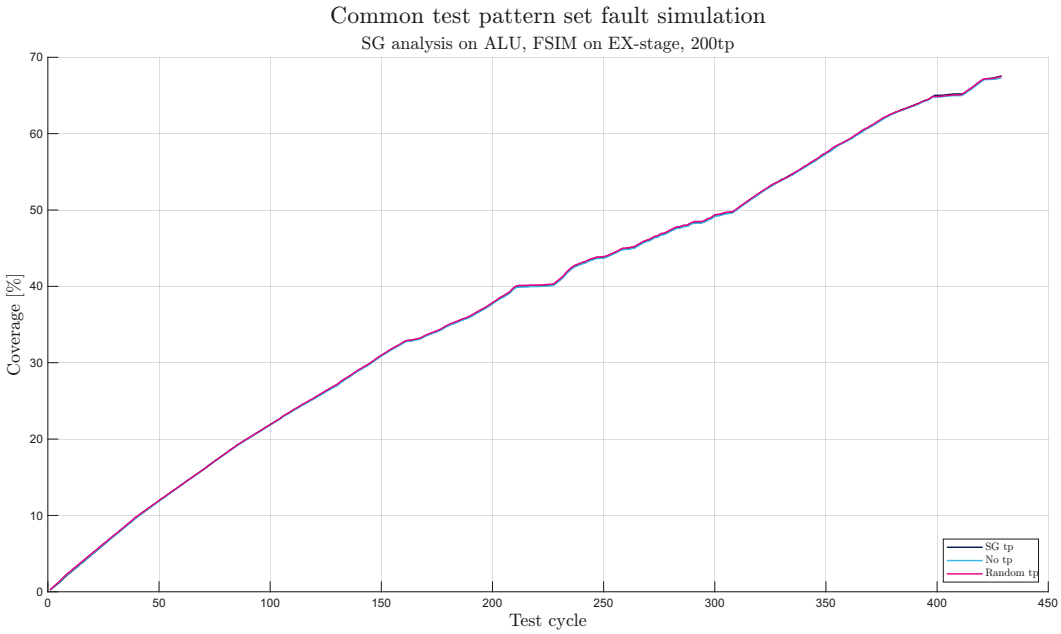


Figure 5.6: Fault simulation of the EX-Stage run on common test patterns with TP inserted in the ALU

5.3.2 Observation points on Execution Stage

Graph 5.7 shows on the y-axis the test coverage, while on the x-axis, the number of test cycles executed during the fault simulation. The data clearly demonstrates that the random test point insertion is superior, providing a 11.62% increment in test coverage relative to the coverage achieved using the SpyGlass approach. This aligns with the trend observed from the former simulation campaign of Section 5.2. Analyzing the SpyGlass-selected optimal test point list (Appendix B.1.1) is possible to observe that most of them are related to the ALU module and only a few of them are picked from other locations in the Execution Stage. So it is no surprise that, similarly to Graph 5.6, the SpyGlass test coverage is basically overlapped with the standard test coverage with no additional observation points inserted.

	SG tp [%]	*Random tp* [%]	*No tp* [%]
Maximum coverage	41.39	53.01	41.22
Δ from max NO tp	0.17	11.79	-
Δ from max RD tp	-11.62	-	-

Table 5.7: Table representing the coverage results of Fig.5.7

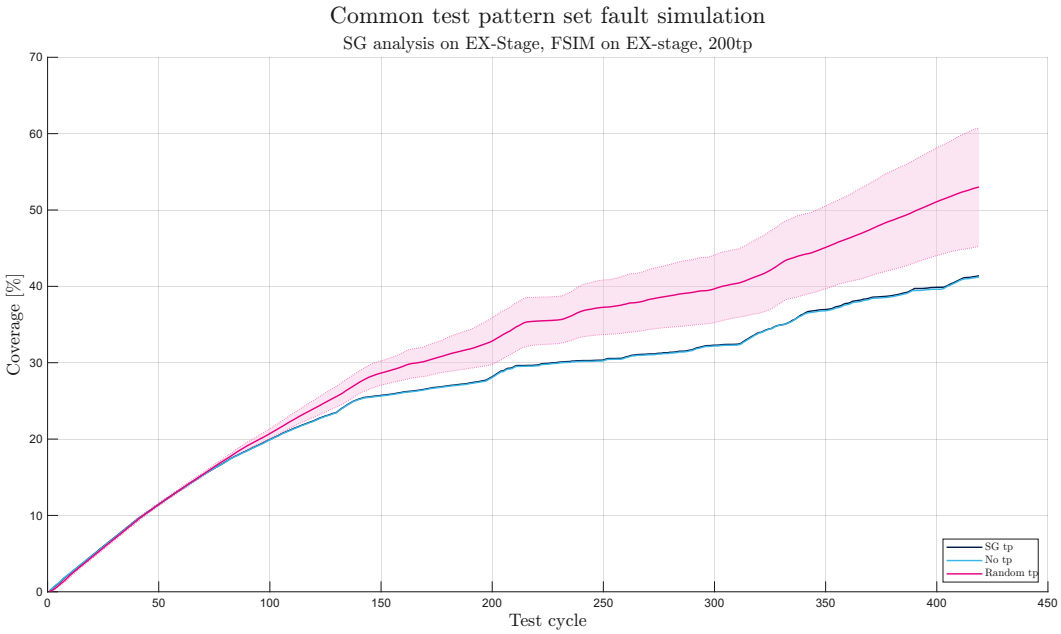


Figure 5.7: Fault simulation of the EX-Stage run on common test patterns with TP inserted in the EX-Stage

5.3.3 Observation points on flatten Execution Stage

Similar to before, a fault simulation has been performed on the flattened Execution Stage, with the combinational and sequential logic separated. Graph 5.8 shows on the y-axis the test coverage, while on the x-axis the number of test cycles executed during the fault simulation, and for this test case, the results seem promising. With respect to the random insertion method, the use of SpyGlass-selected test points resulted in an evident increase in test coverage, reaching an improvement of 6.41% over the random mean value curve, suggesting that SpyGlass successfully identified critical, difficult-to-observe nodes where testability measures have the greatest impact. If a deeper analysis is performed, it is possible to see that the standard band deviation of the random methodology is actually pretty close, if not even overlapped, with the SpyGlass curve, suggesting a not clear separation between the two methodologies.

	SG tp [%]	*Random tp* [%]	*No tp* [%]
Maximum coverage	58.71	52.30	42.45
Δ from max NO tp	16.26	9.85	-
Δ from max RD tp	6.41	-	-

Table 5.8: Table representing the coverage results of Fig.5.8

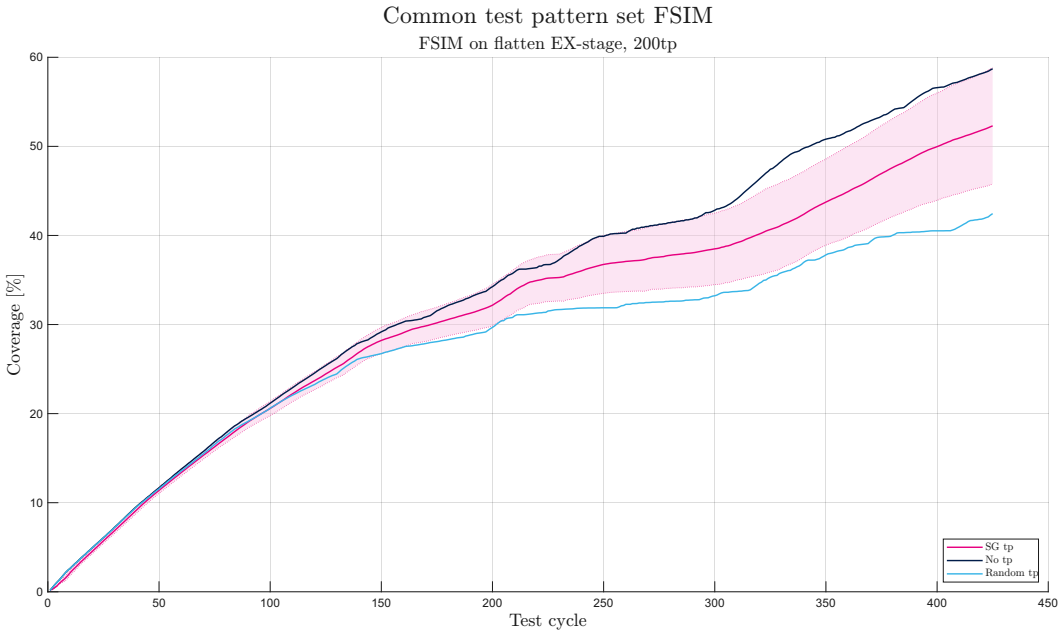


Figure 5.8: Fault simulation of the EX-Stage run on common test patterns with TP inserted in the EX-Stage combinational logic

Chapter 6

Conclusion

This chapter presents the final conclusion and discussion of the obtained results, including a potential justification for the findings. The Synopsys team provided crucial support in justifying some results, which was essential for the final analysis and commentary on SpyGlass performance.

6.1 Results analysis and conclusion

To summarize, in this thesis work, a study on evaluating the effectiveness of a static circuit analysis tool for strategically inserting test points has been pursued. The goal was to determine if this methodology could provide a better solution compared to the simple random selection of points in the test point placement strategy. Different workflows have been arranged, trying to investigate the effectiveness of the proposed methodology. Starting from the first results coming from the RTL approach Section 5.1, Graph 5.1 showed promising results where the use of SpyGlass to analyze the ALU module allowed for obtaining notably higher test coverage with respect to the random insertion of test points. The two curves are clearly separated, so the results seem to validate the efficacy of the proposed methodology. Moving to a much more complex design like the Execution Stage, simulations did not show such promising results. Indeed, from Graph 5.2 is possible to observe the two curves being basically overlapped with no notable distinction between the two; moreover, the absolute test coverage value reached in both cases is extremely low, being in the order of 10%. Very likely, a not sufficiently accurate SBST test program affects the fault detection, providing an overall low test coverage as the architecture cannot be sufficiently tested leading to many faults uncovered ultimately hiding the differences between the two different approaches. The insufficient stimulation of the Execution Stage by the test program used to perform the SBST fault simulation could be solved by creating an ad-hoc list of assembly instructions targeting specific modules of the Execution Stage; even if this may seem straightforward, is a trivial job, since the Execution Stage (described in section 2.2.4), other than computing numerical calculation, it is also responsible for managing branches and memory accesses. So, in order to exercise the Execution Stage properly, it would require to write a code capable of generating every possible data and

control hazards. The coverage being so low also means that probably, many components of the DUT are not even reached, and if test points are inserted in those locations, they are clearly useless to detect faults, so it is not possible to evaluate the effectiveness of the inserted test points since with this approach is not clear if these are actually being used during fault simulation or not.

To better test the Execution Stage, instead of an SBST simulation, a new methodology has been pursued by performing an ATPG campaign capable of exercising the architecture much more extensively (Section 5.2); indeed, from the result obtained, a higher overall test coverage of the Execution Stage could be reached. This approach allowed for better testing of the architecture, giving the opportunity to appreciate the differences between the different methodologies. After inserting test points throughout the Execution Stage and performing a fault simulation on it, Graph 5.3 clearly shows that the random approach, even if slightly, yielded to better results with respect to the more sophisticated SpyGlass method. SpyGlass results, in fact, were similar to the no-test-points baseline and this suggests that, in this case, SpyGlass was not capable of effectively selecting a set of locations capable to enhance the detection of faults. This result can be justified by examining the SpyGlass test points list (Appendix B.1.1), where compared to a random selection of test points (Appendix B.1.2), it is evident that most of the selected locations are restricted solely to the ALU, being only 12 out of 200 outside the module. This can lead to two possible hypotheses: the first being that the ALU is a critical module inside the Execution Stage for the test coverage computation, but this is very unlikely, since the result shows exactly the opposite; while the second being that SpyGlass is very good at analyzing combinational logic rather than sequential one, so all its efforts are targeted mainly towards the ALU. Under this last hypothesis, it is clear why the random approach is the best one, as it allows for selecting points all over the Execution Stage rather than being limited only to the ALU (An example in: Appendix B.1.2). This, of course, brings an improvement in the observation coverage because a higher variety of points are used and more places of the Execution Stage are strobed. To confute the relevance of the ALU module in the computation of the Execution Stage test coverage, an additional experiment has been performed: in Graph 5.3, the simulations have been executed by restricting the locations chosen as additional test points only within the ALU module. Whether they are selected by SpyGlass or randomly chosen, the use of additional test points showed no additional benefit in test coverage improvement. So, now it is clear why SpyGlass suggests the optimal test points mainly on the ALU module rather than in other parts of the design: the ALU being a purely combinational block is intrinsically more suited to be analyzed from SpyGlass's algorithms, which end up suggesting only locations relative to that module. Indeed SpyGlass assumes to have full accessibility to the I/O of the sequential circuits (something typical and true in a scan-based approach), but in functional testing, this assumption is not true anymore. For this reason, a third test has been pursued, dividing the internal Execution Stage logic between the sequential and combinational parts and making SpyGlass analyze only the latter. In this way, the tool is used in its best possible working condition; indeed, Graph 5.5 shows a potential benefit in using SpyGlass over a random approach: for the first sets of patterns, up to 3000, the test coverage was still dominated by the randomly selected test points, then, from that point forward, the ATPG algorithms were capable to exploit the SpyGlass selected test

points and improve strongly the test coverage, creating a step which overcomes the random case. Still, Spyglass methodology cannot provide extraordinary benefits compared to the random one, as they only differ by 3.69%. To better evaluate the benefits of using specific tools to analyze and aid the test point selection, a final test is made by performing a fault simulation using a fixed set of ATPG-generated patterns for all the three fault simulation cases performed in each scenarios. In the case of the test points restricted to the ALU module (Graph 5.6), the behavior aligns with the expectations: being the ALU just a small piece in the whole Execution Stage architecture, the additional implementation of test points with any given methodology, cannot actually provide an improvement in the test coverage results, so all the three curves are overlapped. Concerning the case where the test points are allowed to be spread all across the Execution Stage (Graph 5.7), the behavior of the curves reflects the comments stated before, where the adoption of the random test points approach leads to better results since the selected locations, contrary to SpyGlass analysis, are not localized to only one location. Indeed, the random method shows a better localization of the observation points, leading to a test coverage improvement over the SpyGlass method of 11.79%. The tool, as expected, does not provide any additional aid in fault detection, and it yields to results quite identical to those in the case with no additional test points. Finally, a last test has been performed by computing the fault simulation on the flattened Execution Stage with separation of the combinational and sequential logic. This case could, for the first time, validate the benefits of using SpyGlass as a tool to detect possible optimal observation points, as their test points could provide a higher coverage. Indeed, Graph 5.8 shows that the test coverage reached with the SpyGlass elected test points performs better than the random one by 6.41%. However, looking at the standard deviation band of the random methodology, a deeper analysis shows that the SpyGlass curve often runs very close, and even overlaps, the upper band limit. This suggests that, while the random methodology provides lower test coverage on average, the probability that a random set of test points could achieve a coverage level comparable to the systematic SpyGlass approach is not a remote possibility. So the adoption of a tool like SpyGlass could theoretically help in the test point insertion when performing a functional testing, but its improvements, for what observed from the results, are not quite significant with respect to a more simple random insertion strategy.

6.2 Final conclusions

It is clear that a static analysis tool like SpyGlass can be effective only in certain conditions, specifically when working on purely combinational circuits. In fact, as already stated, SpyGlass's DfT functionality is to analyze the circuit for scan chain insertion, which implies that the tool performs its analysis under the assumption that each flip flop is fully controllable and observable (typical assumptions of a scan-based approach). Therefore, unlike combinational circuits, running functional simulations on sequential circuits reveals that the tool's foundational assumption no longer holds true, limiting SpyGlass's effectiveness in those contexts.

In conclusion, this thesis proved that the use of tools like SpyGlass could potentially

help to improve the test coverage, but only under specific conditions and use cases. SpyGlass performs its analysis under the assumption that each flip-flop in the design has maximum statistical probability to be controlled or observed, but this supposition does not hold in a functional approach, where is not certain that result coming from a cone of logic, controlled by a given functional pattern, is actually sampled by a downstream flip-flop. To obtain better and more reliable results, this proposed procedure should be supported with additional techniques that can perform deeper and more detailed statistical analysis on the controllability and observability of each flip flop of the design, maybe by adopting reinforcement learning methodologies or similar algorithms.

6.3 Future works

Future work should pursuit a few key areas to enhance the effectiveness of this static analysis. First, investigate the statistical aspects of SpyGlass tool by attempting to retrieve significant data from the CUT's internal statistics. This information could then be used to guide the SpyGlass analysis in complex sequential modules, where the current static assumptions are inadequate. Another work that could be assessed is the controllability study of the circuit: as this work focused only on the observation aspect of fault analysis, a similar, dedicated case study could be performed for the controllability aspect. Analyzing controllability in detail could lead to better insights and methods, ultimately helping to achieve higher test coverage results. Finally, once the statistical and controllability procedures are fully validated, they could be integrated into existing RTL monitor insertion workflows. This work will require assessing the significant challenge of managing SystemVerilog iterative module generation and post-synthesis algebraic functions logic instantiation.

Appendix A

Matlab scripts

A.1 Graph plots

Coverage_plot.m

```
% Initialization and configuration -------------------
clear all
clc
close all

stmdarkBlue = '#03234B';
stmlightBlue = '#3CB4E6';
stmpink = '#E6007E';
stmyellow = '#FFD200';
linewidth = 1.2;
linewidth_contour = 0.25;
stmpink_rgb = [0.9764705882352941, 0.7490196078431373,
  ↪  0.8784313725490196];

% Variables ---------------------------------------------
fontsize = 15;
fontsize_title = 20;

% 1) DEFINE THE FILE PATH ========================================
% File path----------------------------------------------

%Fault on EX, ATPG on EX_STAGE 200tp, 5000 pattern
path13 = ".\7_ATPG_ex_200tp\ATPG_5k_ex_sb_clean.txt";
path14 = ".\7_ATPG_ex_200tp\ATPG_5k_ex_nsb_clean.txt";
filelist5kex = [".\7_ATPG_ex_200tp\5k_1_clean.txt", ...
".\7_ATPG_ex_200tp\5k_2_clean.txt",
  ↪  ".\7_ATPG_ex_200tp\5k_3_clean.txt", ...
".\7_ATPG_ex_200tp\5k_4_clean.txt",
  ↪  ".\7_ATPG_ex_200tp\5k_5_clean.txt", ...
```

```
".\7_ATPG_ex_200tp\5k_6_clean.txt",
    ↪  ".\7_ATPG_ex_200tp\5k_7_clean.txt", ...
".\7_ATPG_ex_200tp\5k_8_clean.txt",
    ↪  ".\7_ATPG_ex_200tp\5k_9_clean.txt", ...
".\7_ATPG_ex_200tp\5k_10_clean.txt"];

%Fault on ALU, ATPG on EX_STAGE 200tp, 5000 pattern
path15 = ".\4_ATPG_alu_200tp\ATPG_5k_alu_sb_clean.txt";
filelist5kalu = [".\4_ATPG_alu_200tp\5k_1_clean.txt", ...
".\4_ATPG_alu_200tp\5k_2_clean.txt",
    ↪  ".\4_ATPG_alu_200tp\5k_3_clean.txt",...
".\4_ATPG_alu_200tp\5k_4_clean.txt",
    ↪  ".\4_ATPG_alu_200tp\5k_5_clean.txt",...
".\4_ATPG_alu_200tp\5k_6_clean.txt",
    ↪  ".\4_ATPG_alu_200tp\5k_7_clean.txt",...
".\4_ATPG_alu_200tp\5k_8_clean.txt",
    ↪  ".\4_ATPG_alu_200tp\5k_9_clean.txt", ...
".\4_ATPG_alu_200tp\5k_10_clean.txt"];

%ATPG on flattened architecture, 200tp
path16 = ".\6_ATPG_flattened_arch\ATPG_flatt_5k_sb_clean.txt";
path17 = ".\6_ATPG_flattened_arch\ATPG_flatt_5k_nsb_clean.txt";
filelist5kflat = [".\6_ATPG_flattened_arch\5k_1_clean.txt", ...
".\6_ATPG_flattened_arch\5k_2_clean.txt",
    ↪  ".\6_ATPG_flattened_arch\5k_3_clean.txt", ...
".\6_ATPG_flattened_arch\5k_4_clean.txt",
    ↪  ".\6_ATPG_flattened_arch\5k_5_clean.txt", ...
".\6_ATPG_flattened_arch\5k_6_clean.txt",
    ↪  ".\6_ATPG_flattened_arch\5k_7_clean.txt", ...
".\6_ATPG_flattened_arch\5k_8_clean.txt",
    ↪  ".\6_ATPG_flattened_arch\5k_9_clean.txt", ...
".\6_ATPG_flattened_arch\5k_10_clean.txt"];

%fsim EX with ATPG pattern, 200tp
path18 = ".\8_fsim_ex\fsim_ex_sb_clean.txt";
path19 = ".\8_fsim_ex\fsim_ex_nsb_clean.txt";
filelist_ex_fsim = [".\8_fsim_ex\5k_1_clean.txt", ...
".\8_fsim_ex\5k_2_clean.txt", ".\8_fsim_ex\5k_3_clean.txt", ...
".\8_fsim_ex\5k_4_clean.txt", ".\8_fsim_ex\5k_5_clean.txt", ...
".\8_fsim_ex\5k_6_clean.txt", ".\8_fsim_ex\5k_7_clean.txt", ...
".\8_fsim_ex\5k_8_clean.txt", ".\8_fsim_ex\5k_9_clean.txt", ...
".\8_fsim_ex\5k_10_clean.txt"];

%fsim ALU with ATPG pattern, 200tp
path20 = ".\9_fsim_alu\fsim_alu_sb_clean.txt";
path21 = ".\9_fsim_alu\fsim_alu_nsb_clean.txt";
filelist_alu_fsim = [".\9_fsim_alu\5k_1_clean.txt", ...
".\9_fsim_alu\5k_2_clean.txt", ".\9_fsim_alu\5k_3_clean.txt", ...
```

```
".\9_fsim_alu\5k_4_clean.txt", ".\9_fsim_alu\5k_5_clean.txt", ...
".\9_fsim_alu\5k_6_clean.txt", ".\9_fsim_alu\5k_7_clean.txt", ...
".\9_fsim_alu\5k_8_clean.txt", ".\9_fsim_alu\5k_9_clean.txt", ...
".\9_fsim_alu\5k_10_clean.txt"];

%fsim ex flatt with ATPG pattern, 200tp
path22 = ".\10_fsim_flatt\fsim_ex_flatt_sb_clean.txt";
path23 = ".\10_fsim_flatt\fsim_ex_flatt_nsb_clean.txt";
filelist_ex_flatt_fsim = [".\10_fsim_flatt\5k_2_clean.txt", ...
".\10_fsim_flatt\5k_3_clean.txt", ".\10_fsim_flatt\5k_5_clean.txt", ...
".\10_fsim_flatt\5k_6_clean.txt", ".\10_fsim_flatt\5k_7_clean.txt", ...
".\10_fsim_flatt\5k_8_clean.txt", ".\10_fsim_flatt\5k_9_clean.txt", ...
".\10_fsim_flatt\5k_10_clean.txt"];


% 2) EXTRACT DATA FROM THE DEFINED FILES ==========================
% data extraction

%ATPG ex
ex_200tp_sb = funct_data_extractor(path13, [1, Inf]);
ex_200tp_nsb = funct_data_extractor(path14, [1, Inf]); %valid also for
  ↪  alu
rand_ex_200tp = mean_calculator_ATPG(10, 4500, filelist5kex);

%ATPG alu
alu_200tp_sb= funct_data_extractor(path15, [1, Inf]);
rand_alu_200tp = mean_calculator_ATPG(10, 4728, filelist5kalu);

%ATPG flatt
flatt_5k_sb = funct_data_extractor(path16, [1, Inf]);
flatt_5k_nsb = funct_data_extractor(path17, [1, Inf]);
rand_flatt_200tp = mean_calculator_ATPG(10, 5000, filelist5kflat);

%fsim ex
fsim_ex_sb = fsim_data_extractor(path18, [1, Inf]);
fsim_ex_nsb = fsim_data_extractor(path19, [1, Inf]);

%fsim alu
fsim_alu_sb = fsim_data_extractor(path20, [1, Inf]);
fsim_alu_nsb = fsim_data_extractor(path21, [1, Inf]);

%fsim flatt
fsim_ex_flatt_sb = fsim_data_extractor(path22, [1, Inf]);
fsim_ex_flatt_nsb = fsim_data_extractor(path23, [1, Inf]);

%fsim rand
fsim_ex_rand = mean_calculator_fsim(10, 433, filelist_ex_fsim);
fsim_alu_rand = mean_calculator_fsim(10, 429, filelist_alu_fsim);
```

```
112 fsim_ex_flatt_rand = mean_calculator_fsim(8, 431,
    ↪  filelist_ex_flatt_fsim);
113
114
115 % 3) GRAPHS ============================================================
116 %%PLOT CON STDEV=======================================================
117
118
119 %---------------------------------------------------------------
120 %---------------------------------------------------------------
121 % ATPG ex
122
123 x = (1:1:4500);
124 m = rand_ex_200tp(x);
125 std1 = (m-std_dev_ATPG_ex(x));
126 std2 = (m+std_dev_ATPG_ex(x));
127
128 Figure20=figure(20);
129 set(Figure20, 'defaulttextinterpreter', 'latex');
130 hold on;
131 c0=plot(x, [std1(x);std2(x)], ':', 'Color', stmpink, 'LineWidth',
    ↪  linewidth_contour);
132 c1=fill([(x), fliplr(x)], [std1, fliplr(std2)], stmpink_rgb,
    ↪  EdgeColor='none', FaceAlpha='0.4');
133 c2=plot(x, ex_200tp_sb.coverage(x), 'Color', stmdarkBlue, 'LineWidth',
    ↪  linewidth);
134 c3=plot(x, ex_200tp_nsb.coverage(x), 'Color', stmlightBlue,
    ↪  'LineWidth', linewidth);
135 c4=plot(x, rand_ex_200tp(x), 'Color', stmpink, 'LineWidth', linewidth);
136 title("Different ATPG runs fault simulations","Fontsize",
    ↪  fontsize_title);
137 subtitle("Spyglass on EX-stage, FSIM on EX-stage, 200tp", "Fontsize",
    ↪  fontsize);
138 xlabel("Test pattern", "Fontsize", fontsize);
139 ylabel("Coverage [\%]", "Fontsize", fontsize);
140 legend([c2 c3 c4], "SG tp", "No tp", "Random tp", ...
141     'Location', 'southeast','Interpreter', 'latex');
142 grid on;
143
144
145 %---------------------------------------------------------------
146 %ATPG alu
147
148 x = (1:1:4728);
149 m = rand_alu_200tp(x);
150 std1 = (m-std_dev_ATPG_alu(x));
151 std2 = (m+std_dev_ATPG_alu(x));
152
```

```
153 Figure21=figure(21);
154 set(Figure21, 'defaulttextinterpreter', 'latex');
155 hold on;
156 c0=plot(x, [std1(x);std2(x)], ':', 'Color', stmpink, 'LineWidth',
    ↪ linewidth_contour);
157 c1=fill([(x), fliplr(x)], [std1, fliplr(std2)], stmpink_rgb,
    ↪ EdgeColor='none', FaceAlpha='0.4');
158 c2=plot(x, alu_200tp_sb.coverage(x), 'Color', stmdarkBlue,
    ↪ 'LineWidth', linewidth);
159 c3=plot(x, ex_200tp_nsb.coverage(x), 'Color', stmlightBlue,
    ↪ 'LineWidth', linewidth);
160 c4=plot(x, rand_alu_200tp(x), 'Color', stmpink, 'LineWidth', linewidth);
161 title("Different ATPG runs fault simulations","Fontsize",
    ↪ fontsize_title);
162 subtitle("Spyglass on ALU, FSIM on EX-stage, 200tp", "Fontsize",
    ↪ fontsize);
163 xlabel("Test pattern", "Fontsize", fontsize);
164 ylabel("Coverage [\%]", "Fontsize", fontsize);
165 legend([c2 c3 c4], "SG tp", "No tp", "Random tp", ...
166     'Location', 'southeast','Interpreter', 'latex');
167 grid on;
168
169 %-------------------------------------------------------------
170 %ATPG Flatt
171
172 x = (1:1:4482);
173 m = rand_flatt_200tp(x);
174 std1 = (m-std_dev_ATPG_alu(x));
175 std2 = (m+std_dev_ATPG_alu(x));
176
177 Figure22=figure(22);
178 set(Figure22, 'defaulttextinterpreter', 'latex');
179 hold on;
180 c0=plot(x, [std1(x);std2(x)], ':', 'Color', stmpink, 'LineWidth',
    ↪ linewidth_contour);
181 c1=fill([(x), fliplr(x)], [std1, fliplr(std2)], stmpink_rgb,
    ↪ EdgeColor='none', FaceAlpha='0.4');
182 c2=plot(x, flatt_5k_sb.coverage(x), 'Color', stmdarkBlue, 'LineWidth',
    ↪ linewidth);
183 c3=plot(x, flatt_5k_nsb.coverage(x), 'Color', stmlightBlue,
    ↪ 'LineWidth', linewidth);
184 c4=plot(x, rand_flatt_200tp(x), 'Color', stmpink, 'LineWidth',
    ↪ linewidth);
185 title("Different ATPG runs fault simulations","Fontsize",
    ↪ fontsize_title);
186 subtitle("Spyglass on flattened EX-stage comb logic, FSIM on EX-stage,
    ↪ 200tp", "Fontsize", fontsize);
187 xlabel("Test pattern", "Fontsize", fontsize);
```

```
188 ylabel("Coverage [\%]", "Fontsize", fontsize);
189 legend([c2 c3 c4], "SG tp", "No tp", "Random tp", ...
190     'Location', 'southeast','Interpreter', 'latex');
191 grid on;
192
193
194 %-------------------------------------------------------
195 %EX FSIM
196
197 x = (1:1:419);
198 m = fsim_ex_rand(x);
199 std1 = (m-std_dev_fsim_ex(x));
200 std2 = (m+std_dev_fsim_ex(x));
201
202 Figure23=figure(23);
203 set(Figure23, 'defaulttextinterpreter', 'latex');
204 hold on;
205 c0=plot(x, [std1(x);std2(x)], ':', 'Color', stmpink, 'LineWidth',
    ↪  linewidth_contour);
206 c1=fill([(x), fliplr(x)], [std1, fliplr(std2)], stmpink_rgb,
    ↪  EdgeColor='none', FaceAlpha='0.4');
207 c2=plot((x), fsim_ex_sb.coverage(x), 'Color', stmdarkBlue,
    ↪  'LineWidth', linewidth);
208 c3=plot((x), fsim_ex_nsb.coverage(x), 'Color', stmlightBlue,
    ↪  'LineWidth', linewidth);
209 c4=plot((x), fsim_ex_rand(x), 'Color', stmpink, 'LineWidth', linewidth);
210 title("Fixed test pattern set FSIM","Fontsize", fontsize_title);
211 subtitle("FSIM on EX-stage, 200tp in EX-stage", "Fontsize", fontsize);
212 xlabel("Test cycle", "Fontsize", fontsize);
213 ylabel("Coverage [\%]", "Fontsize", fontsize);
214 legend([c2 c3 c4], "SG tp", "No tp", "Random tp", ...
215     'Location', 'southeast','Interpreter', 'latex');
216 grid on;
217
218
219 %-------------------------------------------------------
220 %ALU FSIM
221
222 x = (1:1:429);
223 m = fsim_alu_rand(x);
224 std1 = (m-std_dev_fsim_alu(x));
225 std2 = (m+std_dev_fsim_alu(x));
226
227 Figure24=figure(24);
228 set(Figure24, 'defaulttextinterpreter', 'latex');
229 hold on;
230 c0=plot(x, [std1(x);std2(x)], ':', 'Color', stmpink, 'LineWidth',
    ↪  linewidth_contour);
```

```
231 c1=fill([(x), fliplr(x)], [std1, fliplr(std2)], stmpink_rgb,
    ↪ EdgeColor='none', FaceAlpha='0.4');
232 c2=plot((x), fsim_alu_sb.coverage(x), 'Color', stmdarkBlue,
    ↪ 'LineWidth', linewidth);
233 c3=plot((x), fsim_alu_nsb.coverage(x), 'Color', stmlightBlue,
    ↪ 'LineWidth', linewidth);
234 c4=plot((x), fsim_alu_rand(x), 'Color', stmpink, 'LineWidth',
    ↪ linewidth);
235 title("Fixed test pattern set FSIM","Fontsize", fontsize_title);
236 subtitle("FSIM on EX-stage, 200tp in ALU", "Fontsize", fontsize);
237 xlabel("Test cycle", "Fontsize", fontsize);
238 ylabel("Coverage [\%]", "Fontsize", fontsize);
239 legend([c2 c3 c4], "SG tp", "No tp", "Random tp", ...
240     'Location', 'southeast','Interpreter', 'latex');
241 grid on;
242
243
244 %-----------------------------------------------------------------
245 %EX FLATT FSIM
246
247 x = (1:1:416);
248 m = fsim_ex_flatt_rand(x);
249 std1 = (m-std_dev_fsim_flatt(x));
250 std2 = (m+std_dev_fsim_flatt(x));
251
252 Figure25=figure(25);
253 set(Figure25, 'defaulttextinterpreter', 'latex');
254 hold on;
255 c0=plot(x, [std1(x);std2(x)], ':', 'Color', stmpink, 'LineWidth',
    ↪ linewidth_contour);
256 c1=fill([(x), fliplr(x)], [std1, fliplr(std2)], stmpink_rgb,
    ↪ EdgeColor='none', FaceAlpha='0.4');
257 c2=plot(x, fsim_ex_flatt_rand(x), 'Color', stmpink, 'LineWidth',
    ↪ linewidth);
258 c3=plot(x, fsim_ex_flatt_sb.coverage(x), 'Color', stmdarkBlue,
    ↪ 'LineWidth', linewidth);
259 c4=plot(x, fsim_ex_flatt_nsb.coverage(x), 'Color', stmlightBlue,
    ↪ 'LineWidth', linewidth);
260 legend([c2 c3 c4],"SG tp", "No tp", "Random tp", ...
261     'Location', 'southeast','Interpreter', 'latex');
262 title("Fixed test pattern set FSIM","Fontsize", fontsize_title);
263 subtitle("FSIM on flatten EX-stage, 200tp", "Fontsize", fontsize);
264 xlabel("Test cycle", "Fontsize", fontsize);
265 ylabel("Coverage [\%]", "Fontsize", fontsize);
266 grid on;
```

RTL_data_extractor.m

```
% Initialization and configuration -------------------
clear all
clc
close all

stmdarkBlue = '#03234B';
stmlightBlue = '#3CB4E6';
stmpink = '#E6007E';
stmyellow = '#FFD200';
linewidth = 1.2;
linewidth_contour = 0.25;
stmpink_rgb = [0.9764705882352941, 0.7490196078431373,
    0.8784313725490196];


% Variables -------------------------------------------
fontsize = 15;
fontsize_title = 20;

tp_number_alu=[0, 10, 50, 100, 169];
tp_number_ex=[0, 10, 50, 100, 250];
alu_coverage_sg=[36.84, 43.73, 44.21, 45.13, 47.08];
alu_coverage_rd=[36.84, 38.20, 38.63, 38.15, 40.50];
ex_coverage_sg=[9.72, 10.23, 10.14, 10.29, 10.47];
ex_coverage_rd=[9.72, 9.89, 9.90, 10.55, 10.85];

err_alu_rd = std(alu_coverage_rd) * ones(size(alu_coverage_rd));
err_ex_sg = std(ex_coverage_sg) * ones(size(ex_coverage_sg));
err_ex_rd = std(ex_coverage_rd) * ones(size(ex_coverage_rd));

%alu
Figure1=figure(1);
set(Figure1, 'defaulttextinterpreter', 'latex');
hold on;
plot(tp_number_alu, alu_coverage_sg, '-o', 'Color', stmdarkBlue,
    'LineWidth', linewidth);
errorbar(tp_number_alu, alu_coverage_rd, err_alu_rd, '-o', 'Color',
    stmpink, 'LineWidth', linewidth);
title("ALU fault simulation coverage comparison","Fontsize",
    fontsize_title);
subtitle("coverage as a function of the number of test points",
    "Fontsize", fontsize);
xlabel("Patterns", "Fontsize", fontsize);
ylabel("Coverage [\%]", "Fontsize", fontsize);
legend("SG tp", "Random tp", ...
    'Location', 'southeast','Interpreter', 'latex');
grid on;
```

```
% Annotate each point with its value
for i = 1:length(tp_number_alu)
    text(tp_number_alu(i), alu_coverage_sg(i),
    ↪ num2str(alu_coverage_sg(i)), 'VerticalAlignment', 'bottom',
    ↪ 'HorizontalAlignment', 'right');
end

% Annotate each point with its value
for i = 1:length(tp_number_alu)
    text(tp_number_alu(i), alu_coverage_rd(i),
    ↪ num2str(alu_coverage_rd(i)), 'VerticalAlignment', 'bottom',
    ↪ 'HorizontalAlignment', 'right');
end


%ex
Figure2=figure(2);
set(Figure2, 'defaulttextinterpreter', 'latex');
hold on;
plot(tp_number_ex, ex_coverage_sg, '-o', 'Color', stmdarkBlue,
↪ 'LineWidth', linewidth);
errorbar(tp_number_ex, ex_coverage_rd, err_ex_rd, '-o', 'Color',
↪ stmpink, 'LineWidth', linewidth);
title("ALU fault simulation coverage comparison","Fontsize",
↪ fontsize_title);
subtitle("coverage as a function of the number of test points",
↪ "Fontsize", fontsize);
xlabel("Patterns", "Fontsize", fontsize);
ylabel("Coverage [\%]", "Fontsize", fontsize);
legend("SG tp", "Random tp", ...
    'Location', 'southeast','Interpreter', 'latex');
grid on;
xlim([0 270]);

for i = 1:length(tp_number_ex)
    text(tp_number_ex(i), ex_coverage_sg(i),
    ↪ num2str(ex_coverage_sg(i)), 'VerticalAlignment', 'bottom',
    ↪ 'HorizontalAlignment', 'right');
end

% Annotate each point with its value
for i = 1:length(tp_number_ex)
    text(tp_number_ex(i), ex_coverage_rd(i),
    ↪ num2str(ex_coverage_rd(i)), 'VerticalAlignment', 'bottom',
    ↪ 'HorizontalAlignment', 'right');
end
```

A.2 Data extrapolating functions

funct_data_extractor.m

```
function atpg_gate_masked_clean = funct_data_extractor(filename, dataLines)
%IMPORTFILE Import data from a text file
%  ATPG_GATE_MASKED_CLEAN = IMPORTFILE(FILENAME) reads data from text
%  file FILENAME for the default selection.  Returns the data as a table.
%
%  ATPG_GATE_MASKED_CLEAN = IMPORTFILE(FILE, DATALINES) reads data for
%  the specified row interval(s) of text file FILENAME. Specify
%  DATALINES as a positive scalar integer or a N-by-2 array of positive
%  scalar integers for dis-contiguous row intervals.
%
%  Example:
%  atpg_gate_masked_clean = funct_data_extractor("C:\Users\mauro\Documents\AAA_Universita\Tesi\Test\atpg_gate_masked_clean.txt", [2, Inf]);
%
%  See also READTABLE.
%
% Auto-generated by MATLAB on 18-Jun-2025 23:06:45

%% Input handling

% If dataLines is not specified, define defaults
if nargin < 1
    dataLines = [1, Inf];
end

%% Set up the Import Options and import the data
opts = delimitedTextImportOptions("NumVariables", 10);

% Specify range and delimiter
opts.DataLines = dataLines;
opts.Delimiter = " ";

% Specify column names and types
opts.VariableNames = ["num_pattern", "x11608", "x801578", "x0_0_0", "coverage", "time_s", "Var7", "Var8", "Var9", "Var10"];
opts.SelectedVariableNames = ["num_pattern", "coverage", "time_s"];
opts.VariableTypes = ["double", "string", "string", "string", "double", "double", "string", "string", "string", "string"];

% Specify file level properties
opts.ExtraColumnsRule = "ignore";
opts.EmptyLineRule = "read";
```

```
opts.ConsecutiveDelimitersRule = "join";
opts.LeadingDelimitersRule = "ignore";

% Specify variable properties
opts = setvaropts(opts, ["x11608", "x801578", "x0_0_0", "Var7",
    ↪  "Var8", "Var9", "Var10"], "WhitespaceRule", "preserve");
opts = setvaropts(opts, ["x11608", "x801578", "x0_0_0", "Var7",
    ↪  "Var8", "Var9", "Var10"], "EmptyFieldRule", "auto");
opts = setvaropts(opts, ["num_pattern", "coverage", "time_s"],
    ↪  "TrimNonNumeric", true);
opts = setvaropts(opts, ["num_pattern", "coverage", "time_s"],
    ↪  "ThousandsSeparator", ",");

% Import the data
atpg_gate_masked_clean = readtable(filename, opts);

end
```

fsim_data_extractor.m

```
function fsim_po_clean = fsim_data_extractor(filename, dataLines)
%IMPORTFILE Import data from a text file
%  FSIM_PO_CLEAN = IMPORTFILE(FILENAME) reads data from text file
%  FILENAME for the default selection.  Returns the data as a table.
%
%  FSIM_PO_CLEAN = IMPORTFILE(FILE, DATALINES) reads data for the
%  specified row interval(s) of text file FILENAME. Specify DATALINES as
%  a positive scalar integer or a N-by-2 array of positive scalar
%  integers for dis-contiguous row intervals.
%
%  Example:
%  fsim_po_clean = fsim_data_extractor("C:\Users\mauro\Documents\AAA_U⌋
    ↪  niversita\Tesi\Test\fsim_po_clean.txt", [2, Inf]);
%
%  See also READTABLE.
%
% Auto-generated by MATLAB on 18-Jun-2025 23:39:00

%% Input handling

% If dataLines is not specified, define defaults
if nargin < 1
    dataLines = [1, Inf];
end

%% Set up the Import Options and import the data
opts = delimitedTextImportOptions("NumVariables", 8);
```

```
27
28 % Specify range and delimiter
29 opts.DataLines = dataLines;
30 opts.Delimiter = ["\t", " "];
31
32 % Specify column names and types
33 opts.VariableNames = ["Var1", "faults", "x1577", "x1", "x2",
   ↪  "x811609", "coverage", "time_s"];
34 opts.SelectedVariableNames = ["faults", "coverage", "time_s"];
35 opts.VariableTypes = ["string", "double", "string", "string",
   ↪  "string", "string", "double", "double"];
36
37 % Specify file level properties
38 opts.ExtraColumnsRule = "ignore";
39 opts.EmptyLineRule = "read";
40 opts.ConsecutiveDelimitersRule = "join";
41
42 % Specify variable properties
43 opts = setvaropts(opts, ["Var1", "x1577", "x1", "x2", "x811609"],
   ↪  "WhitespaceRule", "preserve");
44 opts = setvaropts(opts, ["Var1", "x1577", "x1", "x2", "x811609"],
   ↪  "EmptyFieldRule", "auto");
45 opts = setvaropts(opts, "coverage", "TrimNonNumeric", true);
46 opts = setvaropts(opts, "coverage", "ThousandsSeparator", ",");
47
48 % Import the data
49 fsim_po_clean = readtable(filename, opts);
50
51 end
```

mean_calculator_ATPG.m

```
1  function [mean_vector, std_dev] = mean_calculator_ATPG(number_sets,
   ↪  num_data, path_list)
2  %MEAN_CALCULATOR Summary of this function goes here
3  %   Detailed explanation goes here
4  sum = 0;
5  mean_vector = [];
6  data = [];
7  raw_data = zeros(num_data, 0);
8
9  for i = 1:number_sets
10     data = funct_data_extractor(path_list(i),[1, inf]).coverage;
11     raw_data(:,i) = data(1:num_data);
12 end
13
14 for j = 1:num_data
```

```
    for k = 1:number_sets
        sum = sum + raw_data(j, k);
    end
    mean_vector(j) = sum/number_sets;
    sum = 0;
end

std_dev = std(raw_data, 0, 2)';

end
```

mean_calculator_fsim.m

```
function [mean_vector, std_dev] = mean_calculator_fsim(number_sets,
    num_data, path_list)
%MEAN_CALCULATOR Summary of this function goes here
%   Detailed explanation goes here
sum = 0;
mean_vector = [];
data = [];
raw_data = zeros(num_data, 0);

for i = 1:number_sets
    data = fsim_data_extractor(path_list(i),[1, inf]).coverage;
    raw_data(:,i) = data(1:num_data);
end

for j = 1:num_data
    for k = 1:number_sets
        sum = sum + raw_data(j, k);
    end
    mean_vector(j) = sum/number_sets;
    sum = 0;
end

std_dev = std(raw_data, 0, 2)';

end
```

Appendix B

General scripts

B.1 Reports

B.1.1 SpyGlass optimal observation points on Execution Stage

dft_unified_testpoints.rpt

```
#+--------------SPYGLASS DFT UNIFIED TEST POINTS REPORT---------------+#
# date : 2025-08-23
# TestMAX Advisor : T-2022.06-SP2-01

# test points for target "random_resistant"
# start time : 11:43:31
# Requested Test Points : 200
# Threads requested : 8
# Threads created   : 7 (excluding main thread)

# Design : "ex_stage_16_864949"
# Initial Random Pattern Test Coverage : 6.84
# Stuck At Test Coverage : 14.87
# Target Random Pattern Test Coverage : 99.99
# Requested Test Points : 200
# Random Pattern Count : 64000
# Effort Level : high
1  observe    alu_i/n4155   #Gain : 0.00796 Cov : 6.85071 S@TC : 14.9
2  observe    alu_i/n7590   #Gain : 0.00904 Cov : 6.85975 S@TC : 14.9
3  observe    alu_i/n6863   #Gain : 0.01076 Cov : 6.87051 S@TC : 14.9
4  observe    alu_i/n7425   #Gain : 0.00930 Cov : 6.87981 S@TC : 14.9
5  observe    alu_i/n3573   #Gain : 0.00927 Cov : 6.88908 S@TC : 14.9
6  observe    alu_i/n7639   #Gain : 0.01124 Cov : 6.90032 S@TC : 14.9
7  observe    alu_i/n7137   #Gain : 0.01075 Cov : 6.91107 S@TC : 14.9
8  observe    alu_i/n4109   #Gain : 0.01124 Cov : 6.92230 S@TC : 14.9
9  observe    alu_i/n4193   #Gain : 0.01013 Cov : 6.93244 S@TC : 14.9
```

```
27 10  observe    alu_i/lz_tz_wcount[0]   #Gain : 0.01162 Cov : 6.94406
   ↪  S@TC : 14.9
28 11  observe    alu_i/n3290   #Gain : 0.00728 Cov : 6.95134 S@TC : 14.9
29 12  observe    alu_i/n1473   #Gain : 0.01055 Cov : 6.96189 S@TC : 14.9
30 13  observe    alu_i/n3881   #Gain : 0.00754 Cov : 6.96943 S@TC : 14.9
31 14  observe    alu_i/n4194   #Gain : 0.00834 Cov : 6.97777 S@TC : 14.9
32 15  observe    alu_i/n1410   #Gain : 0.01230 Cov : 6.99007 S@TC : 14.9
33 16  observe    alu_i/n7887   #Gain : 0.01230 Cov : 7.00237 S@TC : 14.9
34 17  observe    alu_i/gen_bitmanip_i_clz_64b/n57   #Gain : 0.00955 Cov :
   ↪  7.01192 S@TC : 14.9
35 18  observe    alu_i/gen_bitmanip_i_clz_64b/n30   #Gain : 0.00825 Cov :
   ↪  7.02017 S@TC : 14.9
36 19  observe    alu_i/n7385   #Gain : 0.00712 Cov : 7.02729 S@TC : 14.9
37 20  observe    alu_i/n1347   #Gain : 0.00703 Cov : 7.03432 S@TC : 14.9
38 21  observe    alu_i/n3739   #Gain : 0.00710 Cov : 7.04143 S@TC : 14.9
39 22  observe    alu_i/n5977   #Gain : 0.00732 Cov : 7.04874 S@TC : 14.9
40 23  observe    alu_i/n4426   #Gain : 0.00782 Cov : 7.05657 S@TC : 14.9
41 24  observe    alu_i/n6942   #Gain : 0.00623 Cov : 7.06280 S@TC : 14.9
42 25  observe    alu_i/n4146   #Gain : 0.00678 Cov : 7.06957 S@TC : 14.9
43 26  observe    alu_i/n3593   #Gain : 0.00675 Cov : 7.07632 S@TC : 14.9
44 27  observe    alu_i/n3022   #Gain : 0.00842 Cov : 7.08474 S@TC : 14.9
45 28  observe    alu_i/n4196   #Gain : 0.00660 Cov : 7.09135 S@TC : 14.9
46 29  observe    alu_i/n1374   #Gain : 0.00849 Cov : 7.09983 S@TC : 14.9
47 30  observe    alu_i/n7293   #Gain : 0.00652 Cov : 7.10636 S@TC : 14.9
48 31  observe    alu_i/n7584   #Gain : 0.00637 Cov : 7.11273 S@TC : 14.9
49 32  observe    alu_i/n1469   #Gain : 0.00688 Cov : 7.11961 S@TC : 14.9
50 33  observe    alu_i/n5081   #Gain : 0.00574 Cov : 7.12535 S@TC : 14.9
51 34  observe    alu_i/n3846   #Gain : 0.00630 Cov : 7.13165 S@TC : 14.9
52 35  observe    alu_i/n7262   #Gain : 0.00673 Cov : 7.13839 S@TC : 14.9
53 36  observe    alu_i/n3620   #Gain : 0.00624 Cov : 7.14462 S@TC : 14.9
54 37  observe    alu_i/n7200   #Gain : 0.00609 Cov : 7.15071 S@TC : 14.9
55 38  observe    alu_i/n3737   #Gain : 0.00564 Cov : 7.15635 S@TC : 14.9
56 39  observe    alu_i/n5800   #Gain : 0.00643 Cov : 7.16278 S@TC : 14.9
57 40  observe    alu_i/n1471   #Gain : 0.00742 Cov : 7.17020 S@TC : 14.9
58 41  observe    alu_i/n1375   #Gain : 0.00849 Cov : 7.17869 S@TC : 14.9
59 42  observe    alu_i/n5075   #Gain : 0.00536 Cov : 7.18405 S@TC : 14.9
60 43  observe    alu_i/n1718   #Gain : 0.00606 Cov : 7.19011 S@TC : 14.9
61 44  observe    alu_i/gen_bitmanip_i_clz_64b/n14   #Gain : 0.01153 Cov :
   ↪  7.20165 S@TC : 14.9
62 45  observe    alu_i/n6218   #Gain : 0.00655 Cov : 7.20820 S@TC : 14.9
63 46  observe    alu_i/n6194   #Gain : 0.00598 Cov : 7.21417 S@TC : 14.9
64 47  observe    alu_i/n3173   #Gain : 0.00611 Cov : 7.22028 S@TC : 14.9
65 48  observe    alu_i/n1394   #Gain : 0.01188 Cov : 7.23216 S@TC : 14.9
66 49  observe    alu_i/n7430   #Gain : 0.00564 Cov : 7.23780 S@TC : 14.9
67 50  observe    alu_i/n2473   #Gain : 0.00604 Cov : 7.24384 S@TC : 14.9
68 51  observe    alu_i/n7757   #Gain : 0.00990 Cov : 7.25375 S@TC : 14.9
69 52  observe    alu_i/n1692   #Gain : 0.01006 Cov : 7.26380 S@TC : 14.9
70 53  observe    alu_i/n7649   #Gain : 0.00518 Cov : 7.26898 S@TC : 14.9
```

```
71 54  observe    alu_i/n7701   #Gain : 0.00599 Cov : 7.27498 S@TC : 14.9
72 55  observe    alu_i/n5976   #Gain : 0.00687 Cov : 7.28185 S@TC : 14.9
73 56  observe    alu_i/n6139   #Gain : 0.00520 Cov : 7.28705 S@TC : 14.9
74 57  observe    alu_i/n6870   #Gain : 0.00586 Cov : 7.29291 S@TC : 14.9
75 58  observe    alu_i/gen_bitmanip_i_clz_64b/n71   #Gain : 0.00681 Cov :
   ↪ 7.29971 S@TC : 14.9
76 59  observe    alu_i/n2664   #Gain : 0.00512 Cov : 7.30483 S@TC : 14.9
77 60  observe    alu_i/n7426   #Gain : 0.00580 Cov : 7.31064 S@TC : 14.9
78 61  observe    branch_unit_i/n249   #Gain : 0.00509 Cov : 7.31573 S@TC :
   ↪ 14.9
79 62  observe    alu_i/gen_bitmanip_genblk1_i_clz_32b/n47   #Gain :
   ↪ 0.01056 Cov : 7.32629 S@TC : 14.9
80 63  observe    branch_unit_i/n497   #Gain : 0.00510 Cov : 7.33139 S@TC :
   ↪ 14.9
81 64  observe    branch_unit_i/n623   #Gain : 0.00509 Cov : 7.33648 S@TC :
   ↪ 14.9
82 65  observe    branch_unit_i/n755   #Gain : 0.01036 Cov : 7.34684 S@TC :
   ↪ 14.9
83 66  observe    branch_unit_i/n756   #Gain : 0.01064 Cov : 7.35748 S@TC :
   ↪ 14.9
84 67  observe    alu_i/n2230   #Gain : 0.00449 Cov : 7.36197 S@TC : 14.9
85 68  observe    alu_i/n4487   #Gain : 0.00498 Cov : 7.36694 S@TC : 14.9
86 69  observe    alu_i/n7682   #Gain : 0.00487 Cov : 7.37181 S@TC : 14.9
87 70  observe    alu_i/n1370   #Gain : 0.00533 Cov : 7.37714 S@TC : 14.9
88 71  observe    branch_unit_i/n248   #Gain : 0.00509 Cov : 7.38224 S@TC :
   ↪ 14.9
89 72  observe    alu_i/n5697   #Gain : 0.00469 Cov : 7.38692 S@TC : 14.9
90 73  observe    alu_i/lz_tz_count[1]   #Gain : 0.00516 Cov : 7.39208
   ↪ S@TC : 14.9
91 74  observe    alu_i/gen_bitmanip_i_clz_64b/n45   #Gain : 0.00704 Cov :
   ↪ 7.39912 S@TC : 14.9
92 75  observe    alu_i/n6244   #Gain : 0.00439 Cov : 7.40351 S@TC : 14.9
93 76  observe    alu_i/n7075   #Gain : 0.00533 Cov : 7.40885 S@TC : 14.9
94 77  observe    alu_i/n7370   #Gain : 0.00428 Cov : 7.41313 S@TC : 14.9
95 78  observe    alu_i/n4108   #Gain : 0.00556 Cov : 7.41869 S@TC : 14.9
96 79  observe    alu_i/n1719   #Gain : 0.00606 Cov : 7.42476 S@TC : 14.9
97 80  observe    alu_i/n1720   #Gain : 0.00606 Cov : 7.43082 S@TC : 14.9
98 81  observe    alu_i/n6740   #Gain : 0.00414 Cov : 7.43496 S@TC : 14.9
99 82  observe    alu_i/n5460   #Gain : 0.00449 Cov : 7.43944 S@TC : 14.9
100 83  observe    branch_unit_i/n250   #Gain : 0.00509 Cov : 7.44454 S@TC :
   ↪ 14.9
101 84  observe    branch_unit_i/n750   #Gain : 0.00509 Cov : 7.44963 S@TC :
   ↪ 14.9
102 85  observe    branch_unit_i/n754   #Gain : 0.01064 Cov : 7.46027 S@TC :
   ↪ 14.9
103 86  observe    alu_i/n5654   #Gain : 0.00451 Cov : 7.46479 S@TC : 14.9
104 87  observe    alu_i/n6951   #Gain : 0.00496 Cov : 7.46975 S@TC : 14.9
105 88  observe    alu_i/n6149   #Gain : 0.00425 Cov : 7.47400 S@TC : 14.9
```

```
106 89  observe    alu_i/n4996   #Gain : 0.00428 Cov : 7.47828 S@TC : 14.9
107 90  observe    alu_i/n6388   #Gain : 0.00417 Cov : 7.48245 S@TC : 14.9
108 91  observe    alu_i/n4354   #Gain : 0.00437 Cov : 7.48681 S@TC : 14.9
109 92  observe    alu_i/n4530   #Gain : 0.00421 Cov : 7.49103 S@TC : 14.9
110 93  observe    alu_i/gen_bitmanip_i_clz_64b/n55   #Gain : 0.00485 Cov :
    ↪ 7.49588 S@TC : 14.9
111 94  observe    alu_i/n7600   #Gain : 0.00450 Cov : 7.50038 S@TC : 14.9
112 95  observe    alu_i/n1717   #Gain : 0.00436 Cov : 7.50475 S@TC : 14.9
113 96  observe    alu_i/n7517   #Gain : 0.00447 Cov : 7.50922 S@TC : 14.9
114 97  observe    alu_i/n6083   #Gain : 0.00431 Cov : 7.51352 S@TC : 14.9
115 98  observe    alu_i/n6144   #Gain : 0.00434 Cov : 7.51786 S@TC : 14.9
116 99  observe    alu_i/n3818   #Gain : 0.00413 Cov : 7.52199 S@TC : 14.9
117 100  observe    alu_i/n4366   #Gain : 0.00537 Cov : 7.52736 S@TC : 14.9
118 101  observe    alu_i/n4554   #Gain : 0.00436 Cov : 7.53173 S@TC : 14.9
119 102  observe    alu_i/n6503   #Gain : 0.00464 Cov : 7.53637 S@TC : 14.9
120 103  observe    alu_i/n7926   #Gain : 0.00382 Cov : 7.54019 S@TC : 14.9
121 104  observe    alu_i/n5803   #Gain : 0.00512 Cov : 7.54531 S@TC : 14.9
122 105  observe    alu_i/n2448   #Gain : 0.00449 Cov : 7.54980 S@TC : 14.9
123 106  observe    alu_i/n6046   #Gain : 0.00608 Cov : 7.55587 S@TC : 14.9
124 107  observe    alu_i/n4781   #Gain : 0.00439 Cov : 7.56026 S@TC : 14.9
125 108  observe    alu_i/n5903   #Gain : 0.00415 Cov : 7.56441 S@TC : 14.9
126 109  observe    alu_i/n5514   #Gain : 0.00409 Cov : 7.56850 S@TC : 14.9
127 110  observe    alu_i/n7187   #Gain : 0.00406 Cov : 7.57256 S@TC : 14.9
128 111  observe    alu_i/n7647   #Gain : 0.00415 Cov : 7.57671 S@TC : 14.9
129 112  observe    alu_i/n7265   #Gain : 0.00408 Cov : 7.58078 S@TC : 14.9
130 113  observe    alu_i/n7696   #Gain : 0.00395 Cov : 7.58474 S@TC : 14.9
131 114  observe    alu_i/n5458   #Gain : 0.00373 Cov : 7.58847 S@TC : 14.9
132 115  observe    alu_i/n5253   #Gain : 0.00420 Cov : 7.59267 S@TC : 14.9
133 116  observe    alu_i/n4741   #Gain : 0.00400 Cov : 7.59666 S@TC : 14.9
134 117  observe    alu_i/n7510   #Gain : 0.00399 Cov : 7.60065 S@TC : 14.9
135 118  observe    alu_i/n3257   #Gain : 0.00414 Cov : 7.60479 S@TC : 14.9
136 119  observe    alu_i/n4882   #Gain : 0.00399 Cov : 7.60878 S@TC : 14.9
137 120  observe    alu_i/n5277   #Gain : 0.00382 Cov : 7.61261 S@TC : 14.9
138 121  observe    alu_i/n7048   #Gain : 0.00359 Cov : 7.61619 S@TC : 14.9
139 122  observe    alu_i/n4223   #Gain : 0.00400 Cov : 7.62019 S@TC : 14.9
140 123  observe    alu_i/n7239   #Gain : 0.00363 Cov : 7.62382 S@TC : 14.9
141 124  observe    alu_i/n3726   #Gain : 0.00346 Cov : 7.62729 S@TC : 14.9
142 125  observe    alu_i/n5553   #Gain : 0.00380 Cov : 7.63108 S@TC : 14.9
143 126  observe    alu_i/n7784   #Gain : 0.00385 Cov : 7.63494 S@TC : 14.9
144 127  observe    alu_i/n1413   #Gain : 0.00406 Cov : 7.63900 S@TC : 14.9
145 128  observe    alu_i/n3995   #Gain : 0.00383 Cov : 7.64282 S@TC : 14.9
146 129  observe    alu_i/n7824   #Gain : 0.00359 Cov : 7.64641 S@TC : 14.9
147 130  observe    alu_i/n7058   #Gain : 0.00380 Cov : 7.65021 S@TC : 14.9
148 131  observe    alu_i/n6561   #Gain : 0.00355 Cov : 7.65377 S@TC : 14.9
149 132  observe    alu_i/n7113   #Gain : 0.00406 Cov : 7.65783 S@TC : 14.9
150 133  observe    alu_i/n3445   #Gain : 0.00383 Cov : 7.66167 S@TC : 14.9
151 134  observe    alu_i/n2475   #Gain : 0.00351 Cov : 7.66518 S@TC : 14.9
152 135  observe    alu_i/n4396   #Gain : 0.00389 Cov : 7.66907 S@TC : 14.9
153 136  observe    alu_i/n3813   #Gain : 0.00321 Cov : 7.67228 S@TC : 14.9
```

```
154 137  observe    alu_i/n5097   #Gain : 0.00357 Cov : 7.67585 S@TC : 14.9
155 138  observe    alu_i/n6780   #Gain : 0.00382 Cov : 7.67967 S@TC : 14.9
156 139  observe    alu_i/n7174   #Gain : 0.00419 Cov : 7.68386 S@TC : 14.9
157 140  observe    alu_i/n3324   #Gain : 0.00334 Cov : 7.68721 S@TC : 14.9
158 141  observe    alu_i/n7328   #Gain : 0.00352 Cov : 7.69072 S@TC : 14.9
159 142  observe    branch_unit_i/n251   #Gain : 0.00509 Cov : 7.69582 S@TC
    ↪  : 14.9
160 143  observe    n582   #Gain : 0.00341 Cov : 7.69923 S@TC : 14.9
161 144  observe    alu_result[28]   #Gain : 0.00344 Cov : 7.70266 S@TC :
    ↪  14.9
162 145  observe    alu_i/n7028   #Gain : 0.00350 Cov : 7.70617 S@TC : 14.9
163 146  observe    alu_i/n4860   #Gain : 0.00365 Cov : 7.70982 S@TC : 14.9
164 147  observe    alu_i/n7111   #Gain : 0.00361 Cov : 7.71343 S@TC : 14.9
165 148  observe    alu_i/n5802   #Gain : 0.00330 Cov : 7.71674 S@TC : 14.9
166 149  observe    alu_i/n7938   #Gain : 0.00342 Cov : 7.72015 S@TC : 14.9
167 150  observe    alu_i/n7378   #Gain : 0.00361 Cov : 7.72376 S@TC : 14.9
168 151  observe    alu_i/n7512   #Gain : 0.00347 Cov : 7.72724 S@TC : 14.9
169 152  observe    alu_i/n7770   #Gain : 0.00377 Cov : 7.73100 S@TC : 14.9
170 153  observe    alu_i/n3567   #Gain : 0.00346 Cov : 7.73446 S@TC : 14.9
171 154  observe    alu_i/n3143   #Gain : 0.00341 Cov : 7.73788 S@TC : 14.9
172 155  observe    alu_i/n6240   #Gain : 0.00333 Cov : 7.74121 S@TC : 14.9
173 156  observe    alu_i/n7821   #Gain : 0.00339 Cov : 7.74460 S@TC : 14.9
174 157  observe    alu_i/n7059   #Gain : 0.00332 Cov : 7.74792 S@TC : 14.9
175 158  observe    alu_i/n5023   #Gain : 0.00332 Cov : 7.75124 S@TC : 14.9
176 159  observe    alu_i/n7779   #Gain : 0.00366 Cov : 7.75490 S@TC : 14.9
177 160  observe    alu_i/n2678   #Gain : 0.00340 Cov : 7.75830 S@TC : 14.9
178 161  observe    alu_i/n3797   #Gain : 0.00360 Cov : 7.76190 S@TC : 14.9
179 162  observe    alu_i/n6994   #Gain : 0.00316 Cov : 7.76506 S@TC : 14.9
180 163  observe    alu_i/n7110   #Gain : 0.00327 Cov : 7.76833 S@TC : 14.9
181 164  observe    alu_i/n3482   #Gain : 0.00355 Cov : 7.77187 S@TC : 14.9
182 165  observe    alu_i/n4361   #Gain : 0.00340 Cov : 7.77527 S@TC : 14.9
183 166  observe    alu_i/n6883   #Gain : 0.00310 Cov : 7.77837 S@TC : 14.9
184 167  observe    alu_i/n7324   #Gain : 0.00336 Cov : 7.78174 S@TC : 14.9
185 168  observe    alu_i/n3869   #Gain : 0.00321 Cov : 7.78495 S@TC : 14.9
186 169  observe    alu_i/n1042   #Gain : 0.00320 Cov : 7.78815 S@TC : 14.9
187 170  observe    alu_i/n5571   #Gain : 0.00330 Cov : 7.79145 S@TC : 14.9
188 171  observe    alu_i/n6022   #Gain : 0.00290 Cov : 7.79435 S@TC : 14.9
189 172  observe    alu_i/n6352   #Gain : 0.00304 Cov : 7.79739 S@TC : 14.9
190 173  observe    alu_i/n3166   #Gain : 0.00286 Cov : 7.80025 S@TC : 14.9
191 174  observe    alu_i/n5080   #Gain : 0.00330 Cov : 7.80355 S@TC : 14.9
192 175  observe    alu_i/n6248   #Gain : 0.00344 Cov : 7.80699 S@TC : 14.9
193 176  observe    alu_i/n5454   #Gain : 0.00292 Cov : 7.80991 S@TC : 14.9
194 177  observe    alu_i/n3568   #Gain : 0.00311 Cov : 7.81302 S@TC : 14.9
195 178  observe    alu_i/n5184   #Gain : 0.00305 Cov : 7.81607 S@TC : 14.9
196 179  observe    alu_i/gen_bitmanip_i_clz_64b/n91   #Gain : 0.00476 Cov :
    ↪  7.82083 S@TC : 14.9
197 180  observe    alu_i/n5372   #Gain : 0.00302 Cov : 7.82385 S@TC : 14.9
198 181  observe    alu_i/n5971   #Gain : 0.00346 Cov : 7.82731 S@TC : 14.9
```

```
199 182  observe    alu_i/n7297   #Gain : 0.00323 Cov : 7.83054 S@TC : 14.9
200 183  observe    alu_i/n6353   #Gain : 0.00306 Cov : 7.83360 S@TC : 14.9
201 184  observe    alu_i/n7106   #Gain : 0.00324 Cov : 7.83685 S@TC : 14.9
202 185  observe    alu_i/n4516   #Gain : 0.00334 Cov : 7.84018 S@TC : 14.9
203 186  observe    alu_i/n4224   #Gain : 0.00329 Cov : 7.84348 S@TC : 14.9
204 187  observe    alu_i/n6899   #Gain : 0.00316 Cov : 7.84664 S@TC : 14.9
205 188  observe    alu_i/n4269   #Gain : 0.00291 Cov : 7.84956 S@TC : 14.9
206 189  observe    alu_i/n3909   #Gain : 0.00335 Cov : 7.85291 S@TC : 14.9
207 190  observe    alu_i/n4754   #Gain : 0.00290 Cov : 7.85581 S@TC : 14.9
208 191  observe    alu_i/n5296   #Gain : 0.00294 Cov : 7.85876 S@TC : 14.9
209 192  observe    alu_i/n3452   #Gain : 0.00312 Cov : 7.86188 S@TC : 14.9
210 193  observe    alu_i/n7417   #Gain : 0.00341 Cov : 7.86528 S@TC : 14.9
211 194  observe    alu_i/n7894   #Gain : 0.00308 Cov : 7.86836 S@TC : 14.9
212 195  observe    alu_i/n1675   #Gain : 0.00271 Cov : 7.87108 S@TC : 14.9
213 196  observe    alu_i/n3910   #Gain : 0.00309 Cov : 7.87416 S@TC : 14.9
214 197  observe    alu_i/n3174   #Gain : 0.00300 Cov : 7.87716 S@TC : 14.9
215 198  observe    alu_i/n5904   #Gain : 0.00300 Cov : 7.88016 S@TC : 14.9
216 199  observe    alu_i/n6508   #Gain : 0.00289 Cov : 7.88305 S@TC : 14.9
217 200  observe    alu_i/n5171   #Gain : 0.00308 Cov : 7.88613 S@TC : 14.9
218 # Current random pattern test coverage is approximately 7.89% with 200
    ↪  test points.
219 # Test point search completed : 200 (requested count : 200) test points
    ↪  identified.
220 # runtime : 19.1165
221 # end time : 11:43:51
222
223 #+-------------------- TARGET-WISE TEST POINT SUMMARY ----------------+#
224 # target : random_resistant
225 #   requested test point count : 200
226 #   reported test point count  : 200
227 #   runtime                    : 19.1165
228 # test points for shadow_wrapper
229 #   reported test point count  : 0
230 #   runtime                    : 0.0000
231 #+--------------------------------------------------------------------+#
232
233 #+------------------- UNIFIED TEST POINT SUMMARY --------------------+#
234 # total test points   : 200
235 # total runtime       : 19.1165
236 #+--------------------------------------------------------------------+#
```

B.1.2 Random selection of test points in the netlist

random_strobe_list.txt

```
lsu_i/i_store_unit/store_buffer_i/u1393/B2
lsu_i/i_store_unit/store_buffer_i/speculative_queue_q_reg_3__address__18_/CK
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_2__i_pmp_entry/u481/A2
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/content_q_reg_14__ppn__39_/RN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u2290/ZN
alu_i/u8016/B1
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u7625/B
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/content_q_reg_4__reserved__8_/Q
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_2__i_pmp_entry/u26/ZN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u1570/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_1__i_pmp_entry/u679/B1
lsu_i/i_pipe_reg_load/d_o_reg_175_/QN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_2__i_opgroup_block/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_noncomp/u337/A1
i_mult/i_multiplier/u7465/B
i_mult/i_multiplier/u486/A1
alu_i/u3721/B1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_3__i_pmp_entry/i_lzc/u171/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/tags_q_reg_0__vpn0__6_/RN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_block/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/nrbd_nrsc_u0/control_u0/genblk4_1__iteration_div_sqrt/u188/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/content_q_reg_11__ppn__6_/D
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_2__i_opgroup_block/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_noncomp/u249/A3
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_0__i_pmp_entry/i_lzc/u170/A2
alu_i/u6103/A2
alu_i/u7071/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/u69/A1
```

```
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/i_pmp_ptw/gen_pmp_genblk1_5__i_pmp_entry/u196/A
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u6985/CI
i_mult/i_multiplier/u2708/A
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u4726/B1
alu_i/u1606/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/tags_q_reg_7__vpn2__2_/RN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_block/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/preprocess_u0/u387/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_7__i_pmp_entry/u258/ZN
i_mult/i_multiplier/u2118/B1
lsu_i/lsu_bypass_i/mem_q_reg_1__vaddr__61_/Q
branch_unit_i/u1139/A2
i_mult/i_multiplier/u6076/B2
i_mult/i_multiplier/u8272/A
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/u226/A1
lsu_i/i_store_unit/store_buffer_i/u1026/A2
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_block/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lane_instance_i_fpnew_cast_multi/u2068/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_0__i_pmp_entry/u78/Z
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u5887/A1
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u5806/A1
i_mult/i_multiplier/u6712/B
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_6__i_pmp_entry/u180/ZN
lsu_i/u677/A2
i_mult/i_multiplier/u2523/B
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_6__i_pmp_entry/u35/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u3953/ZN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u4937/A
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u3935/B1
```

```
53 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/nrbd_nrsc_u0/c
   ↪ ontrol_u0/genblk4_0__iteration_div_sqrt/u565/A
54 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/nrbd_nrsc_u0/c
   ↪ ontrol_u0/u1746/B
55 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u1277/B1
56 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_cast_multi/u2219/A2
57 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_4__i_pmp_entry/
   ↪ u225/A
58 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/i_fpnew_rounding/u85/A2
59 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/i_pmp_ptw/gen_pmp_genblk1_7__i_pmp
   ↪ _entry/u276/A
60 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_1__i_pmp_entry/
   ↪ u416/ZN
61 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u9112/S
62 i_mult/i_multiplier/u7180/A2
63 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/inp_pipe_op_q_reg_1__1_/Q
64 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u3751/B
65 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_cast_multi/u236/A1
66 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_3__i_pmp_entry/
   ↪ u274/Z
67 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/i_arbiter/u188/A1
68 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/preprocess_u0/
   ↪ u72/A
69 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_0__i_pmp_entr
   ↪ y/u368/B1
70 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u4969/S
71 alu_i/u1324/ZN
```

```
72 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u14741/ZN
73 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_3__i_pmp_entry/
   ↪ u263/ZN
74 lsu_i/lsu_bypass_i/u1368/B2
75 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u3674/A2
76 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_5__i_pmp_entr
   ↪ y/u112/A2
77 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/content_q_reg_2__ppn__11_/D
78 alu_i/u1572/B1
79 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/u40/A1
80 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u1169/ZN
81 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u930/A2
82 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u1342/B1
83 lsu_i/u798/A2
84 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u675/B2
85 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u2591/A
86 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_cast_multi/u3133/A1
87 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u7592/ZN
88 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/nrbd_nrsc_u0/c
   ↪ ontrol_u0/genblk4_2__iteration_div_sqrt/u527/ZN
89 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u2800/ZN
90 i_mult/i_multiplier/u11354/B
91 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u2536/ZN
92 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
   ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
   ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/preprocess_u0/
   ↪ u299/ZN
93 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_2__i_pmp_entry/
   ↪ u156/A
94 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
   ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
   ↪ s_0__active_lane_lane_instance_i_fma/u12830/ZN
```

```
lsu_i/lsu_bypass_i/u478/A
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/fpu_norm_u0/u8
    ↪ 02/ZN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u11366/A
i_mult/i_multiplier/u8243/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_7__i_pmp_entr
    ↪ y/u97/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/tags_q_reg_13__vpn2__4_/D
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/inp_pipe_operands_q_reg_1__0_
    ↪ _13_/D
i_mult/i_multiplier/u12176/B
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u2550/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/tags_q_reg_14__vpn2__6_/CK
lsu_i/lsu_bypass_i/mem_q_reg_0__vaddr__12_/CK
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u2638/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/content_q_reg_1__a_/D
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/inp_pipe_operands_q_reg_1__0_
    ↪ _18_/Q
alu_i/c373/A1
alu_i/u1464/A2
alu_i/u7508/B1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u5221/B1
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u5324/Z
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u3302/C1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_6__i_pmp_entry/
    ↪ u644/A2
i_mult/i_multiplier/u8561/ZN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u8559/A
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_5__i_pmp_entry/
    ↪ u546/B1
lsu_i/i_load_unit/u188/ZN
alu_i/u443/A
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u5037/A
```

```
122 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u831/A2
123 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u4354/ZN
124 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u3934/ZN
125 alu_i/u2860/ZN
126 lsu_i/i_store_unit/store_buffer_i/speculative_queue_q_reg_2__address__
    ↪ 26_/D
127 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_cast_multi/u3771/A1
128 gen_cvxif_cvxif_fu_i/u270/A2
129 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u2128/A1
130 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/i_pmp_ptw/gen_pmp_genblk1_6__i_pmp
    ↪ _entry/u129/C1
131 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u2211/A
132 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/tags_q_reg_13__asid__13_/CK
133 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u632/ZN
134 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u12333/A1
135 lsu_i/lsu_bypass_i/u93/A1
136 i_mult/i_multiplier/u12719/ZN
137 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_4__i_pmp_entr
    ↪ y/u23/A2
138 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/tags_q_reg_15__asid__5_/RN
139 i_mult/u205/A1
140 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u5504/A1
141 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_3__i_pmp_entr
    ↪ y/u531/A
142 lsu_i/gen_mmu_sv39_i_cva6_mmu/misaligned_ex_q_reg_cause__0_/RN
143 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_cast_multi/u148/A
144 i_mult/i_multiplier/u7550/B
145 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u882/B2
146 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_3__i_pmp_entry/
    ↪ u222/A2
```

```
147 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/fpu_norm_u0/u3
    ↪ 38/B1
148 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u10431/ZN
149 lsu_i/u338/A2
150 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u2422/B
151 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_2__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_noncomp/i_class_a/u29/ZN
152 alu_i/u6586/A2
153 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u4087/B
154 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/tags_q_reg_4__vpn0__1_/Q
155 lsu_i/i_store_unit/store_buffer_i/u2767/A2
156 i_mult/i_multiplier/u8421/B2
157 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u5825/ZN
158 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u2721/A2
159 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u15609/ZN
160 lsu_i/i_pipe_reg_store/u40/A
161 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/nrbd_nrsc_u0/c
    ↪ ontrol_u0/u1904/A2
162 alu_i/u4729/A2
163 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u14130/A
164 alu_i/u7720/A
165 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u13725/A2
166 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_dtlb/u2921/A4
167 u73/ZN
168 lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/i_pmp_ptw/gen_pmp_genblk1_6__i_pmp
    ↪ _entry/u562/ZN
169 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u5315/B1
170 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_2__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_noncomp/u7/Z
171 i_mult/i_multiplier/u11702/B
```

```
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_2__i_pmp_entry/u40/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/i_pmp_ptw/gen_pmp_genblk1_6__i_pmp_entry/u371/B2
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_1__i_pmp_entry/i_lzc/u28/Z
lsu_i/lsu_bypass_i/u423/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_if/gen_pmp_genblk1_7__i_pmp_entry/u227/ZN
fpu_gen_fpu_i/u278/A2
alu_i/u4390/B1
alu_i/u7854/A1
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/mid_pipe_spec_res_q_reg_1__sign_/CK
alu_i/c43/A1
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u370/A2
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/i_pmp_ptw/gen_pmp_genblk1_6__i_pmp_entry/u373/ZN
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/u4442/S
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_pmp_data/gen_pmp_genblk1_6__i_pmp_entry/u164/ZN
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_block/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lane_instance_i_fpnew_cast_multi/u2232/A1
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_block/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/fpu_norm_u0/u539/A2
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_block/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/nrbd_nrsc_u0/control_u0/genblk4_0__iteration_div_sqrt/u64/A2
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u7716/A
alu_i/u4403/ZN
lsu_i/i_store_unit/store_buffer_i/u2664/ZN
lsu_i/i_load_unit/u621/A2
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_itlb/content_q_reg_2__ppn__14_/Q
lsu_i/gen_mmu_sv39_i_cva6_mmu/i_ptw/i_pmp_ptw/gen_pmp_genblk1_6__i_pmp_entry/u561/A2
fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_block/gen_parallel_slices_0__active_format_i_fmt_slice/gen_num_lanes_0__active_lane_lane_instance_i_fma/u210/A1
```

```
196 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_3__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_cast_multi/u384/A1
197 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/preprocess_u0/
    ↪ u1327/ZN
198 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_1__i_opgroup_b
    ↪ lock/gen_merged_slice_i_multifmt_slice/gen_num_lanes_0__active_lan
    ↪ e_lane_instance_i_fpnew_divsqrt_multi/i_divsqrt_lei/preprocess_u0/
    ↪ u684/A
199 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u9960/B2
200 fpu_gen_fpu_i/fpu_gen_i_fpnew_bulk/gen_operation_groups_0__i_opgroup_b
    ↪ lock/gen_parallel_slices_1__active_format_i_fmt_slice/gen_num_lane
    ↪ s_0__active_lane_lane_instance_i_fma/u7947/ZN
```

Bibliography

[1] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010.

[2] Wikipedia contributors. Risc-v — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=RISC-V&oldid=1304464205`, 2025. [Online; accessed 17-August-2025].

[3] Jean-Roch COULON Thales. Cv32a60x design document — openhw group, thales dis france sas. `https://docs.openhwgroup.org/projects/cva6-user-manual/07_cv32a60x/design/design.html`, 2025. [Online; accessed 24-August-2025].

[4] ETH Zurich and University of Bologna. Cva6: An application class risc-v cpu core — openhw group. `https://docs.openhwgroup.org/projects/cva6-user-manual/index.html`, 2025. [Online; accessed 18-August-2025].

[5] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology, july 2019.

[6] Ultraembedded. Risc-v core architecture. `https://deepwiki.com/ultraembedded/riscv/2-risc-v-core-architecture`, 2025. [Online; accessed 24-August-2025].

[7] Wikipedia contributors. Design for testing — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Design_for_testing&oldid=1277262927`, 2025. [Online; accessed 26-August-2025].

[8] Fulvio Corno Maurizio Rebaudengo and Matteo Sonza Reorda. Testing and fault tolerance.

[9] Michelangelo Grosso. Digital testing basics, 25/09/2023.

[10] *IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data.*

[11] *TestMAX ATPG and TestMAX Diagnosis User Guide.*

[12] *SpyGlass DFTMethodology GuideWare2.0 UserGuide.*

[13] *VC Z01X User Guide.*