



Meinhard Wolfgang Kissich, BSc

Model Checking in Digital Design to Enhance Traditional Functional Verification

A Practical Approach on a Lidar Evaluation Kit

Master's Thesis

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Advisors

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Dipl.-Ing. Bernd Janger (ams OSRAM AG)

Institute of Technical Informatics

Graz, May 2022

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

There is no doubt that thorough function verification is essential for developing a digital system. While Assertion-Based Verification has been widely adopted in simulation for designers and verification engineers to deal with increased design complexity and cost, model checking still seems to be a field reserved for formal verification experts. However, formal verification appears to be an ideal technique to check aspects of the design intent at the block level in early verification when the fully-featured test environment is not ready. Thereby the iterative phases lead to a deep insight into the design and can find hard-to-detect corner-case bugs.

Formal property verification is investigated from a designer's perspective and applied to the digital design of a Lidar evaluation kit. After a general introduction to formal verification highlights its differences and advantages to dynamic verification, a broad overview of state-of-the-art methodologies and verification flows is given. The subsequent work focuses on deriving properties in industry-standard SystemVerilog Assertions and applying the proposed lightweight verification methodology. In the evaluation, the design complexity and possible countermeasures are discussed. There are many pitfalls to consider that might even be contradictory to simulation. The applied verification found weaknesses in the specifications and led to an unprecedented design insight.

Kurzfassung

Funktionale Verifikation ist ein essenzieller Bestandteil der Entwicklungsphase integrierter digitaler Systeme. Um trotz steigender Komplexität eine ausreichende Evidenz zu angemessenen Kosten zu erreichen, entstehen neue Konzepte wie Assertion-Based Verification. Assertions bieten eine deklarative Beschreibung des Systemverhaltens, das durch dynamische Verifikation oder formale Methoden überprüft werden kann. Obwohl Assertions in Simulationen sehr geläufig sind, scheint formale Verifikation ein Bereich zu sein, der dezidierten Experten zugeschrieben wird. Die Vorteile von Model Checking können allerdings auch für Designer sehr attraktiv sein, da gewisse Aspekte von Modulen vor Fertigstellung der finalen Testumgebung verifiziert werden können. Im Gegensatz zur Simulation wird dabei der gesamte Zustandsraum untersucht. Dies vereinfacht das Auffinden von Corner-Case Fehlern und führt durch iterative Verfeinerung der formalen Umgebung zu einem detaillierten Einblick in das Design.

Diese Arbeit diskutiert formale Verifikation aus der Sicht eines Designers anhand der Implementierung eines Lidar Evaluierungskits. Nach einer kurzen Einführung in die formale Verifikation wird ein Überblick über moderne Methodiken und Anwendungsbereiche gegeben. Die folgenden Kapitel leiten Systemeigenschaften in SystemVerilog Assertions ab und diskutieren die gewonnenen Erkenntnisse, sowie den Umgang mit Komplexität und Limitierungen der formalen Tools. Der gezeigte Bug-Hunting Ansatz konnte Unklarheiten in der Spezifikation aufzeigen, die durch Simulation übersehen werden könnten.

Acknowledgment

This thesis would not have been possible without encouragement and support from many people.

First, I would like to thank my advisors, Prof. Christian Steger and Dipl.-Ing. Bernd Janger, who supported me during the thesis, gave scope for development and guidance when needed.

I would also like to extend my gratitude to all colleagues at ams OSRAM for supporting me on various topics that made this work possible.

Specifically, I would like to thank Steven Dennis, B.Sc (Hons), and Dr. Ernst Haselsteiner, whose experience and patience in answering numerous questions have been invaluable throughout this thesis and broadened my mind and thinking of engineering.

I am also grateful to the team from YosysHQ, which first made SVA property checking accessible to me to gather knowledge as a prerequisite for this work.

Many thanks to my study colleagues and friends for their support, countless discussions, collaboration in joint projects, and joyful time.

Finally, my deep and sincere gratitude to my partner and family for continuous encouragement and all their love throughout my years of study.

Meinhard Kissich
Graz, May 2022

Contents

1. Introduction	3
1.1. Motivation	3
1.2. Goals	4
1.3. Outline	5
2. Background	6
2.1. System Failure	6
2.1.1. Intel FDIV	6
2.1.2. Ariane 5	7
2.2. Verification Approaches	7
2.2.1. Assertion-Based Verification	7
2.2.2. Simulation	8
2.2.3. Emulation	9
2.2.4. Formal Verification	9
2.3. Model Checking	11
2.4. SystemVerilog	12
2.4.1. Brief History	12
2.4.2. Simulation: Event Regions and Scheduling	12
2.4.3. Assertions	15
2.4.4. Concurrent Assertions	17
2.4.5. Suffix Implication	18
2.4.6. Environment Modeling and Constraints	18
2.4.7. Cover Points	20
3. Related Work and State-of-the-Art	21
3.1. Applications	21
3.1.1. Error Correcting Code	21
3.1.2. RISC-V Formal	22
3.1.3. OpenRISC 1200 Security	22
3.1.4. AHB-Lite to I2C Bridge	23
3.2. Verification Flows	23
3.3. Debugging Assertions	23
3.3.1. Visualization	24
3.3.2. Structural Debugging and Fan-In Analysis	24
3.3.3. Mutation Based	24

Contents

3.4.	AutoSVA	25
3.5.	Missing Assumptions	26
3.6.	Completeness	26
3.6.1.	Coverage	26
3.6.2.	Gap-Free Verification	27
3.7.	Formal Efficiency	27
3.7.1.	Suitable Designs	28
3.7.2.	Efficient Property Design	28
3.7.3.	Abstraction and Other Techniques	30
3.8.	Formal Verification for Safety-Critical Systems	30
3.8.1.	Automotive Standards	31
3.8.2.	Applicaton of Formal Verification	31
4.	Design Specifications	33
4.1.	Project Introduction	33
4.2.	System View	34
4.3.	Digital System Requirements	35
4.3.1.	Pinout	35
4.3.2.	Operating Modes	37
4.4.	Derived Architecture	40
4.4.1.	Debounce	42
5.	Verification	45
5.1.	Verification Procedure	45
5.1.1.	Bind Files	45
5.1.2.	Verification Objective	46
5.1.3.	Proposed Methodology	46
5.1.4.	Flow	48
5.1.5.	General Considerations	49
5.2.	Debounce	50
5.2.1.	Environment	50
5.2.2.	Cover Points	51
5.2.3.	Assertions	54
5.3.	Core	62
5.3.1.	Environment	62
5.3.2.	Cover Points	64
5.3.3.	Assertions	64
6.	Evaluation	67
6.1.	Runtime Considerations	67
6.1.1.	Auxiliary Code	68
6.1.2.	Data Types and Vector Sizes	70

Contents

6.2. Runtime and Complexity	73
6.2.1. Debounce	74
6.2.2. Core	76
6.3. Coverage	77
6.3.1. Debounce	77
6.3.2. Overlooked Bugs	78
6.3.3. Core	80
6.4. Efficiency	82
7. Conclusion	83
7.1. Results	83
7.2. Conclusion and Future Work	84
Bibliography	86
A. Core Module Verification	93
B. Detailed Evaluation Results	97

List of Figures

1.1.	Cost of a revealed bug as the project evolves.	4
2.1.	Simulation traces through the abstracted state space.	8
2.2.	Exploration of the state space in formal verification.	10
2.3.	Exemplary Kripke structure.	11
2.4.	Verilog-2001 scheduling event regions.	13
2.5.	SystemVerilog-2017 scheduling event regions without PLI.	15
2.6.	Assertion types in SystemVerilog-2017.	16
2.7.	Sampled values in concurrent assertions.	17
2.8.	Abstraction of the DUT's environment in functional verification. . .	19
2.9.	Constraints on internal signals in simulation.	19
2.10.	Assumptions on internal nets in formal verification.	20
2.11.	Abstracted state space representation of a cover property.	20
3.1.	Annotated assertions in simulation.	24
3.2.	Structural debugging in OneSpin DV.	25
3.3.	Fan-in analysis of an SVA property.	25
4.1.	High-level view on the Lidar transmitter evaluation board.	34
4.2.	Primary interaction channels between the components.	35
4.3.	FPGA pinout.	35
4.4.	Timing diagram for control signals in mode M_0 , M_1 and M_3	39
4.5.	Timing diagram of the main FSM in mode M_2	40
4.6.	Refined FPGA architecture.	41
4.7.	<i>Debounce</i> module: operation without disable and mode change. . . .	43
4.8.	<i>Debounce</i> module: behavior on dynamic reset.	43
4.9.	<i>Debounce</i> module: behavior on disable.	44
4.10.	<i>Debounce</i> module: behavior of mode_change input.	44
5.1.	Usage of the bind directive to connect the checker to the DUT. . . .	46
5.2.	Flowchart of the proposed verification methodology.	47
5.3.	Formal verification flow and tool interaction.	49
5.4.	Sequence diagram of a simple formal verification run.	50
5.5.	Witness for cover property cov_4 from listing 5.3.	51
5.6.	Witness for cover property cov_chg_disabled from listing 5.7.	53

List of Figures

5.7. <i>Debounce</i> module: visualization of assertion <code>asrt_req_data_if</code>	55
5.8. <i>Debounce</i> module: concept to verify the request based on a shift register. . .	57
5.9. <i>Debounce</i> module: concept to verify the request based on a counter. . .	57
5.10. <i>Debounce</i> module proof step: request did not violate the debounce attempt.	59
5.11. <i>Debounce</i> module proof step: possible raise of follow-up requests. . .	59
5.12. <i>Debounce</i> module proof step: correct raise of follow-up requests. . .	60
5.13. <i>Debounce</i> module proof step: correct raise of follow-up requests by considering a sequence of two requests.	60
6.1. Concept to prove the debounce delay with pure SVA.	68
6.2. Comparison of the runtime to prove the debounce delay with pure SVA or auxiliary code.	70
6.3. Comparison of the runtime when an integer is used instead of a bit vector.	72
6.4. Impact of adding unnecessary counter bits on the runtime of proving the debounce delay for 128 debounce cycles.	73
6.5. Assertion complexity for 6 debounce cycles.	74
6.6. Assertion complexity for 128 debounce cycles.	75
6.7. <i>Debounce</i> module: RAM utilization for proving the properties on a model with 6 or 128 debounce cycles, respectively.	75
6.8. <i>Core</i> module: RAM utilization for proving the properties.	76
6.9. Auto-checks of initialization values in the formal tool.	78

List of Tables

1.1. Thesis goals.	5
2.1. SVA property types and usages.	16
4.1. FPGA pinout description.	36
4.2. FPGA operating modes.	37
4.3. Operating mode M_0	37
4.4. Operating mode M_1 : differences to M_0	38
4.5. Operating mode M_2	38
4.6. Operating mode M_3	39
4.7. Submodules of <i>core</i>	41
6.1. <i>Debounce</i> module assertions runtimes for 128 debounce cycles. . . .	76
6.2. <i>Core</i> module assertions runtimes.	77
6.3. <i>Debounce</i> module structural coverage.	77
6.4. <i>Core</i> module structural coverage.	80
6.5. <i>Core</i> module structural coverage details.	81
6.6. <i>Core</i> module structural coverage analysis.	81
B.1. Runtimes with pure SVA and auxiliary code proven with Yices 2. . .	98
B.2. Runtimes with pure SVA and auxiliary code proven with Z3. . . .	98
B.3. Runtimes for different declarations of the counter variable; Yices 2. .	99
B.4. Runtimes for different declarations of the counter variable; Z3. . . .	99
B.5. Runtimes when the state-holding vector is artificially enlarged on a model with 128 debounce cycles.	100

Listings

2.1. Verilog simulation race condition; example 1.	14
2.2. Verilog simulation race condition; example 2.	14
2.3. Example of a concurrent assertion in SVA.	17
2.4. SVA property using suffix implication to trigger a check.	18
3.1. Decomposition of assertions to lower the complexity.	29
5.1. Default clocking and default disable statements.	49
5.2. <i>Debounce</i> module: assumptions on the environment.	50
5.3. <i>Debounce</i> module basic cover properties.	51
5.4. <i>Debounce</i> module: cover request raise.	52
5.5. <i>Debounce</i> module: cover attempt termination.	52
5.6. <i>Debounce</i> module: cover sequence without request.	52
5.7. <i>Debounce</i> module: cover an address change while the module is disabled.	53
5.8. <i>Debounce</i> module: cover a mode change.	53
5.9. <i>Debounce</i> module: prove reset state.	54
5.10. <i>Debounce</i> module: release from reset.	54
5.11. <i>Debounce</i> module: pulse length of a request.	54
5.12. <i>Debounce</i> module: update output address on a request.	55
5.13. <i>Debounce</i> module: output address stable without a request.	55
5.14. <i>Debounce</i> module: request <i>If (a)</i> part.	58
5.15. <i>Debounce</i> module: request <i>If (b)</i> part.	58
5.16. <i>Debounce</i> module: prove debounce delay with auxiliary code.	58
5.17. <i>Debounce</i> module: not a second request occurs without precondition.	61
5.18. <i>Debounce</i> module: no request while module is disabled.	61
5.19. <i>Debounce</i> module: delay after enabling.	61
5.20. <i>Debounce</i> module: terminate attempt when disabling.	62
5.21. <i>Debounce</i> module: invalidate address on a mode change.	62
5.22. <i>Core</i> module: assumptions on the environment.	63
5.23. <i>Core</i> module: cover properties.	64
5.24. <i>Core</i> module: <code>mon_x_from_fpga</code> in mode <i>M2</i>	64
5.25. <i>Core</i> module: <code>eues</code> in mode <i>M2</i>	65
5.26. <i>Core</i> module: <code>lshot_en_x_from_fpga</code> for inactive driver in mode <i>M2</i>	65
5.27. <i>Core</i> module: <code>lshot_en_x_from_fpga</code> during SPI transfer in mode <i>M2</i>	65

5.28. <i>Core</i> module: 1shot mask in mode M_2	65
5.29. <i>Core</i> module: SPI frame status byte.	66
5.30. <i>Core</i> module: SPI frame padding.	66
5.31. <i>Core</i> module: SPI frame command.	66
6.1. Prove debounce delay without auxiliary code.	68
6.2. Modified auxiliary code to prove input data stability.	69
6.3. Define <code>sva_stable_cnt</code> as an integer instead of a bit vector.	70
6.4. Instrumented auxiliary code to inspect the runtime for different declarations of <code>sva_stable_cnt</code>	71
6.5. Instrumented auxiliary code to inspect the impact of different vector lengths on the runtime.	72
6.6. Undetected statement in <i>debounce</i> module.	78
6.7. Assertion to check the reset state of the debounce register.	78
6.8. Malicious D-type flip-flop design.	79
6.9. Malicious D-type flip-flop checker.	79
A.1. <i>Core</i> module: properties for modes M_0 and M_1	93
A.2. <i>Core</i> module: shadow model for <code>1shot_en_mask</code>	94
A.3. <i>Core</i> module: simple SPI receiver shift register.	96
B.1. Open <i>debouncer</i> implementation.	97

List of Abbreviations

ABV	Assertion Based Verification
ASIL	Automotive Safety Integrity Level
ATPG	Automatic Test Pattern Generation
BMC	Bounded Model Checking
CEX	CounterEXample
COI	Cone of Influence
DUT	Device Under Test
DUV	Device Under Verification
DV	Design Verification
ECC	Error Correction Code
FEV	Formal Equivalence Verification
FI	Fault Injection
FPGA	Field Programmable Gate Array
FPV	Formal Property Verification
FSM	Finite State Machine
FV	Formal Verification
GUI	Graphical User Interface
HDL	Hardware Description Language
HDVL	Hardware Description and Verification Language

List of Abbreviations

IP	Intellectual Property
ISA	Instruction Set Architecture
LEC	Logical Equivalence Checking
Lidar	Light detection and ranging
LRM	Language Reference Manual
LUT	LookUp Table
OEM	Original Equipment Manufacturer
PLI	Programming Language Interface
PSL	Property Specification Language
RPi	Raspberry Pi
RVFI	RISC-V Formal Interface
SBC	Single-Board Computer
SPFM	Single-Point Fault Metric
UNR	Coverage UNreachability
UVM	Unified Verification Methodology
VCSEL	Vertical Cavity Surface Emitting Laser
VIP	Verification IP

1. Introduction

This chapter emphasizes the importance of a thorough verification process and outlines some of the topics discussed in this work. Furthermore, the thesis' goals are announced, and an outlook on the subsequent chapters is given.

1.1. Motivation

In recent years, system complexity has grown significantly, affecting hardware designs [1] as well as software designs. More than half of the 500 software development professionals in [2] pointed out that the amount of code has been increased by a factor of 100 within the last ten years. This trend also becomes apparent in safety-critical systems [3] where deviations from the intended functionality can lead to dangerous situations and human fatality. Section 2.1 depicts two often-cited examples of a system failure due to an overlooked fault introduced in the development phase. Besides the substantial loss of money, a bad reputation and liability are an incomplete list of consequences. Thus, functional verification is essential [4, 5] and faults must be eliminated as early as possible during the development phase.

Design complexity also affects the verification process [6, 7], which has become more expensive than the design development itself [1, 6, 8]. While traditional verification based on simulation is still primarily used, the complexity requires new methodologies. Since the introduction of modern formal specification languages such as Property Specification Language (PSL) [9] and SystemVerilog Assertions (SVA) [10], Assertion-Based Verification (ABV) has been widely adopted in industry and brought several advantages [1, 6, 11–13]. As assertions provide an unambiguous temporal description of the design intent, they can be formally proven on a given model known as Formal Property Verification (FPV).

Although Formal Verification (FV) is not utilized in every project [14], the benefits are clear: FPV exhaustively proves a property, which maybe is impractical in any reasonably-sized implementation with dynamic verification. Simulation always suffers from the risk of missing one particular input sequence that would trigger the fault [15]. Moreover, the formal tools provide an explicit CounterEXample (CEX) for each violated assertion that can be used for debugging [11]. As FV does not require classical test cases, verification can start early and drive parallel

development. Overall, it can lead to a shorter time-to-market [16], uncover bugs earlier [17], and prevent product delays [18]. Figure 1.1 depicts the commonly known increase of the cost of a found bug as the project progresses to highlight the importance of finding bugs early.

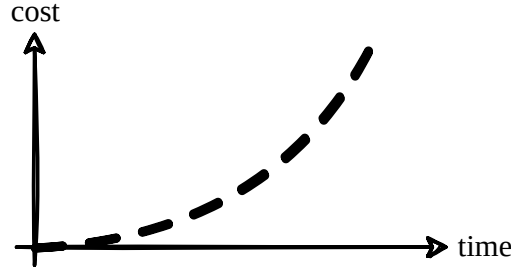


Figure 1.1.: Cost of a revealed bug as the project evolves.

Besides functional verification, applying FV gives a deep understanding of the design gained from iterative steps of analyzing counterexamples and refining the model. There are many more state-of-the-art applications of FV in semiconductor design. It is utilized to check equivalence of designs after optimization [18], validate safety mechanisms against random hardware faults in safety-critical systems [14] and prevent hardware trojans [19] to name a few.

Many semiconductor companies such as Bosch, Infineon, Renesas [20] and Intel [16] rely on formal verification in their development processes.

1.2. Goals

The hugely growing design complexity and shorter time-to-market become apparent in the endeavor to optimize verification. The verification phase needs to be thoroughly planned and optimized for cost and time without cutting back on the completeness or quality. Overlooked faults may lead to a bad reputation, financial losses, or human injury. FV seems to be promising to achieve a higher level of evidence. However, many verification flows seem to base on traditional dynamic verification. This work aims to give an introduction to FPV in an industry environment, investigate the current barriers in model checking and apply formal methods to a practical digital design for a Light detection and ranging (Lidar) evaluation kit. It shall be discussed whether formal verification is a suitable technique to find bugs early. The goals are summarized in table 1.1.

Table 1.1.: Thesis goals.

Goal	Description
<i>Goal_1</i>	Introduce ABV and discuss the differences between simulation and formal verification.
<i>Goal_2</i>	Discuss model checking on a practical digital design using an industry-standard language to model the formal environment and temporal properties.
<i>Goal_3</i>	Give an overview of related work and projects that used FPV.
<i>Goal_4</i>	Summarize state-of-the-art methodologies for efficiently applying FPV and techniques to overcome limitations on real projects.
<i>Goal_5</i>	Propose a verification flow for bug hunting based on FPV and apply it to an actual digital design.
<i>Goal_6</i>	Evaluate the gathered results and discuss the advantages and difficulties of applying formal verification early during the design phase.

1.3. Outline

The thesis is structured into the following parts: Chapter 2 provides background information on the covered topics. It starts with emphasizing the importance of thorough verification by outlining two system failures due to insufficient functional verification. Subsequently, *assertion-based verification* is introduced in static and dynamic verification. Assertions are a practical way to describe the system intent that can be proven on a given model of the design. A primer on how this can be solved mathematically is given. The chapter concludes with an introduction to SystemVerilog Assertions, SystemVerilog scheduling semantics, and an outlook on modeling a formal test environment.

Chapter 3 describes related work and summarizes state-of-the-art techniques and methodologies. It shows some practical projects whose verification flow benefited from FPV, which is the technique also used throughout this thesis. Then, methodologies to efficiently debug properties and add missing assumptions to the formal testbench are referenced. It addresses completeness and metrics to get evidence of the verification quality. As the state(-space) explosion is a considerable burden in practical model checking, techniques to overcome the complexity are discussed. Finally, an outlook on formal methods for verifying functional safety in the automotive domain is given.

In chapter 4 the verification target is presented. It starts with a system-level overview of the developed Lidar evaluation kit. Afterward, the required specifications for the verification are derived in detail.

Chapter 5 addresses the property modeling to enhance dynamic verification by model checking. The chapter is split into two parts, and each of which covers the properties and considerations of one module to be verified.

An evaluation of the implemented properties and efficiency considerations is given in chapter 6. The thesis concludes by summarizing the results and giving an outlook on future work in chapter 7.

2. Background

This chapter gives background information on the topics discussed throughout the further work. Section 2.1 highlights the importance of a reliable functional verification process and briefly outlines two system failures caused by incorrectness of the design. Section 2.2 introduces *assertion-based verification* and its importance for finding bugs close to their source. Moreover, the relation between assertions and *function property verification* is clarified. In section 2.3 *model checking* as the mathematics behind formal property verification is briefly described. Finally, section 2.4 gives an insight into *SystemVerilog Assertions*, its scheduling semantics, and modeling of properties for formal verification and simulation.

2.1. System Failure

There is no doubt that functional verification is an essential part of the product development cycle. Different approaches have been established to find bugs early and cope with various limitations (see section 2.2). Many designs rely on simulation-based verification [8, 21] with the significant disadvantage of only verifying the correctness of a selected set of input stimuli (either hand-crafted or randomly generated). For most designs, the number of states and potential stimuli is enormous. Thus, simulation cannot exhaustively prove the correctness of a system in all corner cases [8, 14, 15].

When a defect is overlooked and gets activated in the field, it might lead to a system failure with either *direct* or *indirect consequences* [3]. A long list of widely known system failures that led to a massive loss of money, bad reputation, or even human fatality in the case of safety-critical systems already exists. Two of which are briefly described below.

2.1.1. Intel FDIV

The FDIV bug refers to a systematic fault encountered in Intel Pentium processors in 1994 [22]. It was discovered by Thomas Nicely, who recognized deviations in a sum of twin prime reciprocals. Specific floating-point numerator and denominator pairs caused incorrect division results. He could trace back the issue and assumed a bug in the floating-point arithmetic unit of the processor. Soon a model was found that showed errors of up to 61 ppm by the computation $4195835.0/3145727.0$. The error originated from five missing lookup table entries in the silicon implementation of the radix-4 SRT algorithm [23]. A detailed insight into the mathematical background is given in [22].

The cost of the bug was estimated to amount to over \$400 million and sped up the emergence of new verification techniques such as formal verification [16, 24].

2.1.2. Ariane 5

Although the failure was caused by a software bug, the Ariane 5 crash is an excellent example of insufficient verification. On 4th June 1996, just 39 seconds after H_0 (approximately 35 seconds after lift-off), the Ariane 5 space launch vehicle disintegrated. The failure was traced to a re-used software part from Ariane 4. Higher velocities in Ariane 5 caused an overflow in a data-type conversion. Due to a maximum workload, there was no precondition check executed on the type conversion [25, 26].

2.2. Verification Approaches

2.2.1. Assertion-Based Verification

Assertion-Based Verification is an effective technique increasingly used to cope with design complexity and shorter time-to-market [1, 12, 13, 27–29]. In ABV, the design code is instrumented with assertion properties to check the intended behavior. This has multiple advantages illustrated throughout this work. Assertions describe design aspects that must hold if the design intent is matched [16, 18].

While the implementation usually describes *how* the system behaves, assertions target *what* the system shall do, introducing an abstraction layer which is ideal for verification [30]. Unlike assertions in software, which tend to be more straightforward and check boolean expressions [18], assertions in hardware design have to describe signals over time. They must be written in a machine-friendly syntax to be checked by the tools. This is typically done by a declarative verification language like Property Specification Language (PSL) [9] or SystemVerilog Assertions (SVA) [10]. The latter is an integral part of the SystemVerilog language, which has native support for formally expressing temporal behavior with *concurrent assertions* (see section 2.4). As assertions have highly expressive semantics that the tools and designers can understand, they are seen as an executable part of the design documentation [11, 13, 18] that adds fine-grained details possibly not included in the design specification. Unlike comments, assertions indicate errors when not updated on design changes.

The expressiveness of assertions also leads to complexity. Even short-looking assertions can express fairly complex behavior. Often only a small portion of the available features of specification languages is used, and assertions are kept short [11]. There is a high chance that a verification run fails because of erroneous assertions instead of a design bug. Section 3.3 deals with state-of-the-art methods to ease assertion debugging.

There are many well-fitting places for assertions in hardware designs depending on the performed checks and the person who added the assertion. A digital designer may add an embedded assertion to check specific aspects of the design intent or input and parameter ranges. A verification engineer, however, may consider the design as a black box and connect (*bind*, see section 5.1.1) a *checker* to the design to verify its correctness. Binding checker modules to the design increases re-usability. Once a checker is written for commonly used behavior (e.g., a bus protocol), it can be re-used in many subsequent designs, commonly referred to as Verification IP (VIP) [31]. Assertions may also be re-used for simulation, formal verification, and emulation [12, 18].

2.2.2. Simulation

Simulation is the traditional way of evaluating the functional correctness of the implemented design. The checked module or system is usually referred to as the Device Under Test (DUT) or the Device Under Verification (DUV). It is instantiated in a testbench that applies stimuli at the DUT's inputs over the simulation time. The simulation engine performs computations that mimic the behavior of actual hardware. Section 2.4.2 gives an overview of how this can be achieved with sequentially executed instructions and how accidentally created race conditions can be prevented. A monitor probes the DUT's output ports and compares them against a reference, possibly computed by the high-level reference model.

Assertions are additional checks of either the internal (white-box assertions) or the external (black-box assertions) behavior of the DUT. Embedded assertions are close to the fault's origin and can detect errors that do not propagate to the DUT's outputs. The increased observability leads to shorter traces and simplified localization of a bug [31].

Figure 2.1 abstractly depicts how stimuli may steer the design state from a set of initial states through the state space. Each arrow reflects one simulation trace. The cross-hatched area represents a set of illegal states described by an assertion. As can be seen, assertions are only checked on the particular trajectory.

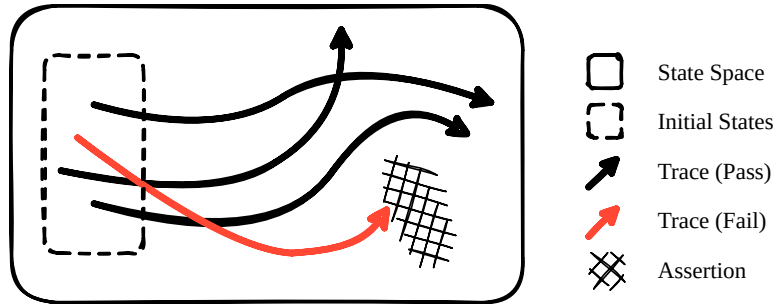


Figure 2.1.: Simulation traces through the abstracted state space.

One main concern in simulation is unstimulated errors, i.e., defects that are not activated by a simulation trace and remain undetected [15, 32]. As simulation typically cannot exhaustively test all combinations of input vectors, there are different approaches to measure the verification quality and decide when to terminate testing.

Input stimuli can be created in different ways, such as directed testing, where test-cases are hand-crafted, or (constraint-)random testing to automatically generate a large set of test vectors [33]. Although randomly generated stimuli can reach a higher overall coverage in a shorter time [34], corner cases might not be hit, and designers have to add directed test cases to reach them [4].

As designers often face recurring problems, there have been many attempts to unify the verification environment. In 2010 the Unified Verification Methodology (UVM) [35] was published that is widely adopted and became the de-facto standard [36] in industry. A good primer is given in [37].

2.2.3. Emulation

Simulation might be too slow to simulate large sequences such as an operating system boot. In emulation, the DUT is synthesized and run on configurable hardware. Formally specified assertions may be synthesized along with the design to overcome speed limitations [18].

2.2.4. Formal Verification

Within the scope of this work, formal verification refers to applying mathematical concepts to formally prove properties, i.e., temporal description of an intended design behavior on a model of a system. This type of static ABV is well known as Formal Property Verification (FPV) [34].

Formal Property Verification

Unlike simulating one test case after another, formal verification exhaustively checks properties on the entire state space (although there are practical limitations). Thus, FPV has several advantages over simulation-based verification. If the proof passes, there does not exist *any* trace that violates this assertion under the given assumptions. Thus, a bug cannot remain unstimulated.

Moreover, neither a classical testbench nor input stimuli are required. The formal tool uses mathematics to explore the state space of a system instead (see section 2.3). This helps to find bugs early in the development process when the simulation environment is not ready yet [13] and pushes parallel development. Crafting formal properties can also help discover conceptual flaws in the specification [17].

Another notable advantage is the automatic generation of a counterexample on a violated assertion. The crafted CEX is usually the shortest trace [38] leading to the assertion violation, which eases analysis. It can be visually inspected or reproduced in simulation to debug the DUT. Thus, the task of writing a suitable testbench that triggers all assertions is fully transferred to the formal tool. As the tool can find any trace that leads to a violation (which can be pretty creative for humans), measures must be taken to exclude illegal input sequences. Typically, this involves modeling the environment with assumptions along with assertions (see section 2.4.6).

Figure 2.2 illustrates the differences to simulation: No individual traces are simulated; instead, the properties are checked one by one for the entire model of the implementation. If an assertion violation is reachable from the set of initial states, a trace exists to trigger the assertion. Assumptions may restrict the considered reachable state space¹. The generated CEX provides a basis to fix the design, environment model, or assertion for the next iteration run.

¹Assumptions are neglected in the illustration. Thus, all reachable states are considered in the proof.

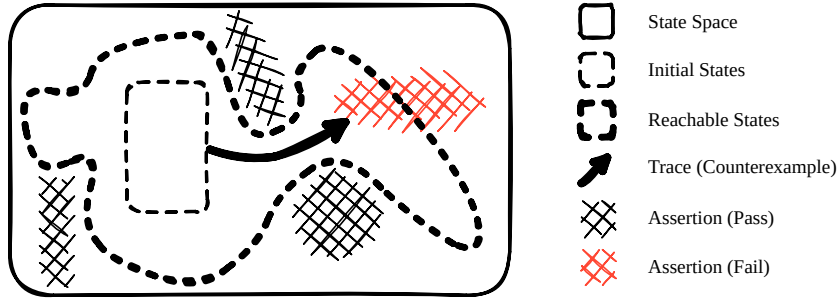


Figure 2.2.: Exploration of the state space in formal verification. Adapted from [16, Figure 4.2].

While the view on FV up to this point was quite optimistic, many practical limitations exist that need to be overcome. Most importantly, FV can only cope with a small model as the number of states grows exponentially to the number of state-holding elements. This phenomenon is known as *state space explosion* [4, 34, 39]. The model size that can be formally proven depends on the particular architecture and properties, but it shall be kept in the range of thousands [18] to 40.000 [16] state-holding elements as a rule of thumb. Section 3.7 discusses methods to deal with this limitation. Moreover, the specifications must be complete and unambiguous to derive formal properties.

Bug Hunting

FPV may be applied at different extents and with different expectations. It ranges from invoking the formal tool for exploring a single execution path² to an exhaustive formal proof of the complete specifications. [16] refers to the first as *design exploration*. Instead of manually writing a cycle-accurate testbench to apply all inputs to reach a target state, the formal property only expresses *what* to reach. Afterward, the tool comes up with a suitable trace for reaching the desired state, i.e., *how* to reach this state, if there exists a path to the desired state.

On the other hand, a full formal proof can replace simulation by exhaustively proving all specification details. As this might come with massive effort and cost, *bug hunting* [16] (also called *lightweight verification* [18]) is a more relaxed approach to applying FPV. It is aimed to prove interesting aspects and hunt hard-to-find bugs with less effort and cost than a full proof. Unlike a full proof, it cannot replace other verification techniques but enhances the confidence that the system is correct.

Other Applications Formal Verification

Besides FPV, many other applications of FV in digital design exist. Formal Equivalence Verification (FEV)³ can check whether two models are equivalent. Use cases are the comparison of the RTL model against a synthesized gate-level netlist, or the original vs. an

²E.g., check whether a cover property is reachable if no stimuli can be found that covers a particular sequence.

³It is also referred to as Logical Equivalence Checking (LEC) [40].

optimized implementation [18]. FV may also be used to evaluate the impact of random hardware faults (see section 3.8) and prevent hardware trojans [19]. A larger set of practical applications can be found in [16] and [18].

2.3. Model Checking

This section briefly demonstrates how FPV boils down to a graph search problem efficiently solved by model checking. The following considerations are limited to reactive and synchronous systems, i.e., systems that continuously interact with the surrounding environment and whose state holding elements are updated on a common clock signal. [8] is a highly recommended resource for a more profound insight into model checking.

Model checking is applied to a system model instead of the implementation itself. The description in hardware design is usually given in an HDL such as VHDL or (System)Verilog and is then converted into a unified formalism. For reactive systems, it is necessary to model the internal states rather than just the input-output behavior. The usual representation is a *Kripke* structure which is the input for most model checking techniques [41]. Research shows that the conversion into a finite-state Kripke structure or, likewise, first-order logic is possible [8].

A Kripke structure is defined as five-tuple⁴ structure $M = (S, S_0, R, AP, L)$ where S is a set of states, S_0 are the initial states ($S_0 \subseteq S$), R is a left-total transition relation ($R \subseteq S \times S$), AP is the set of atomic propositions and L is a labeling function $L : S \rightarrow 2^{AP}$ [8]. Loosely speaking, L maps each state to a set of atomic propositions that are asserted in the corresponding state, e.g., signals that are high in hardware circuits. Figure 2.3 depicts an exemplary Kripke structure.

Let s_i , $\{i \in \mathbb{N}_0 | i \leq 7\}$ be the numbered states from left to right (ignoring the vertical offset) and the text within each state the set of asserted atomic propositions $L(s_i)$. In this case, the Kripke structure may be described as follows: $S = \{s_0, s_1, \dots, s_7\}$, $S_0 = \{s_0, s_1\} \subseteq S$, $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_1), \dots, (s_7, s_5)\}$, $AP = \{a, b\}$ and $L = \{s_0 \mapsto \emptyset, s_1 \mapsto \{b\}, \dots, s_7 \mapsto \{a\}\}$. As can be seen, only the five leftmost states are reachable.

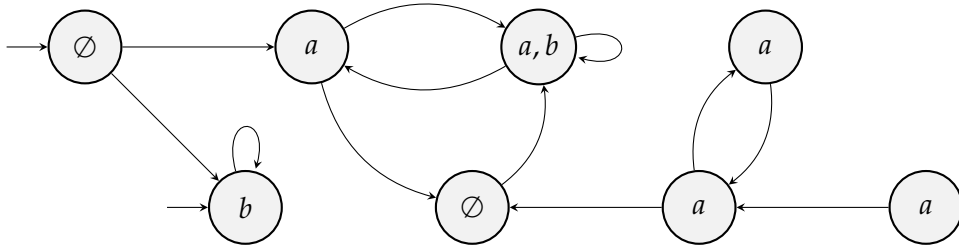


Figure 2.3.: Exemplary Kripke structure.

⁴In some literature a Kripke structure is defined as a quadruple [5].

Given the model of the system, one can formally express specifications in temporal logic and check whether the model satisfies the formula. The formulas may be expressed with different temporal logics such as CTL* or fragments such as CTL or LTL [8]. The model satisfies the property if the language defined by the property is a superset of the language defined by the model [18]. The Kripke structure either models the entire system or a sufficiently detailed approximation to prove the intended property. Approximation might be required to handle complexity and overcome state space explosion, which is one of the main challenges in model checking [5, 42]. Due to research of new algorithms and techniques, it is possible to solve this problem for a remarkable number of states, which made it applicable in verifying industry applications (see section 3.1).

2.4. SystemVerilog

2.4.1. Brief History

The Hardware Description Language (HDL) *Verilog* was introduced in winter 1983/84 [43] (1984 [44]). In 1989 rights were shifted to *Cadence*⁵, who made Verilog publicly available in the following year. Five years later, in 1995, Verilog became an IEEE standard [44].

SystemVerilog is a superset of Verilog and was enriched to support features for testing and verification natively. Thus, SystemVerilog became a Hardware Description and Verification Language (HDVL) [18]. It was first standardized in IEEE 1800-2005 and referred to Verilog version IEEE 1364-2005 [43]. From 2009 both standards are combined [18, 43] with the current version being IEEE 1800-2017 [10]. A short notation for the corresponding standards is used for readability, e.g., SystemVerilog-2017 instead of SystemVerilog IEEE 1800-2017.

2.4.2. Simulation: Event Regions and Scheduling

It is crucial to investigate how SystemVerilog is simulated. On the one hand, SystemVerilog Assertions are frequently written once for both simulation and formal verification⁶. On the other hand, assertions are often debugged through simulation.

One difficulty is correctly simulating the behavior of concurrent hardware with sequentially executed software running on the CPU. To cope with parallel execution in hardware, Verilog simulation is based on discrete *events* [45].

Events are categorized into *update events* and *evaluation events*. Update events are caused by a changed value or signal. Evaluation events concern the execution of processes sensitive to a changed value by an update event.

Simulation time is the discretized understanding of time by the simulator. It is retained until no pending events are queued, and only then the simulation time is increased.

⁵<https://www.cadence.com>

⁶However, optimization for simulation and formal verification might be contradictory and not be achievable at the same time.

Verilog Event Queues

Verilog-2001 uses five event queues. Four are handled in the current time slot (*active*, *inactive*, *NBA*, *monitor*) and one is for future events. The latter one is further split into *future inactive events* and *future NBA* [45]. Simulating the events happens iteratively. First, all events in the *active region* queue are executed. While executing, they might trigger new events. Once the *active region* queue is empty, all events from the *inactive region* queue are moved to the *active region* queue. The *active region* queue is executed again. This is repeated until both *active region* and *inactive region* are empty. Afterward, all events in the *NBA* queue are executed. Again, new events might be released. When the *NBA* queue is empty, there are two options: Either all queues are empty, or new events have appeared. In the latter case, the whole cycle is repeated. If all queues are empty, the simulation time is increased [18].

Figure 2.4 depicts the scheduling regions in Verilog-2001. Either the *active*, *inactive*, or *NBA* region can spawn new events that lead to re-wind of the execution in the current simulation time step. Note that the *monitor* step is referred to as *postponed* in SystemVerilog [46]. The Language Reference Manual (LRM) [45] provides a pseudo-code snippet of the scheduling algorithm simulators shall implement.

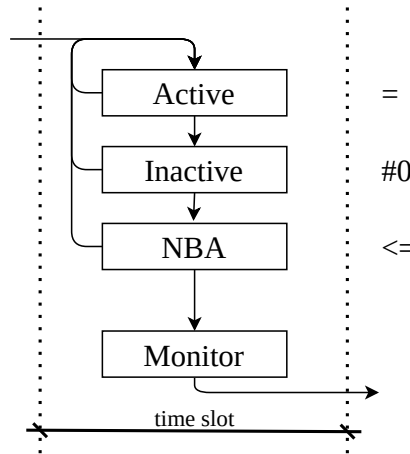


Figure 2.4.: Verilog-2001 scheduling event regions [18, 45].

Race Conditions

A *race condition* in the context of Verilog simulation relates to unexpected simulation output because of an ambiguous command execution order [45]. Besides simulation-induced race conditions, [46] also discusses hardware races from the physical behavior of hardware components. The following discussion is limited to simulation-induced race conditions.

The LRM gives a descriptive example of a race in Verilog simulation illustrated in listing 2.1. The assignment in line 4 causes an update event for *p*. The simulator may execute line 5 before the update event or vice versa. Both execution orders are valid according to the LRM. $\$display(p)$ may print either 0 or 1 [45].

2. Background

```
1 assign p = q;
2 initial begin
3     q = 1;
4     #1 q = 0;
5     $display(p);
6 end
```

Listing 2.1: Verilog simulation race condition; example 1. Source: [45, p. 66].

Another example is given in [18]. In the assignment in listing 2.2 the new value of *a* depends on the execution order of the simulator when *clk* and *b* rise at the same time. This can happen when a testbench module sets both signals to high simultaneously. The simulator may schedule those events in arbitrary order. Thus, either the clock rises first, in which case *a* gets the old value of *b* (0), or *b* rises first and *a* gets 1.

```
1 always @(posedge clk) a <= b;
```

Listing 2.2: Verilog simulation race condition; example 2. Source: [18, Example 3.6].

Coding guidelines help to reduce simulation-based race conditions. The proposed guidelines in [47] can eliminate “90-100% of the most common Verilog simulation race conditions” [47, p. 22]. These guidelines are also summarized in [46]. SystemVerilog enhances the available number of event queues and thereby provides countermeasures against race conditions. program blocks are introduced to avoid most races between the DUT and the testbench [18]. As statements in programs are executed in a different event region set, assignments to signals by the testbench cannot interfere with the design [48].

SystemVerilog Event Queues

Figure 2.5 illustrates the simulation event regions in SystemVerilog-2017 required to understand the evaluation of assertions later on. In addition to the depicted regions, another eight regions exist to interact with, e.g., external scripts. These are referred to as Programming Language Interface (PLI). There are two natural groupings of the event regions: The *active region set* consisting of the *Active*, *Inactive* and the *NBA* regions and the *reactive region set* composed of the *Reactive*, *Re-Inactive* and the *Re-NBA* regions⁷. Due to this split, the design events (i.e., in the DUT) and the testbench events (i.e., the program applying stimuli and probing results) are separated. Subsequently, most races are prevented between design and testbench code. The further discussion focuses on the effect on writing assertions. A detailed insight is given in the LRM.

⁷PLI regions are neglected.

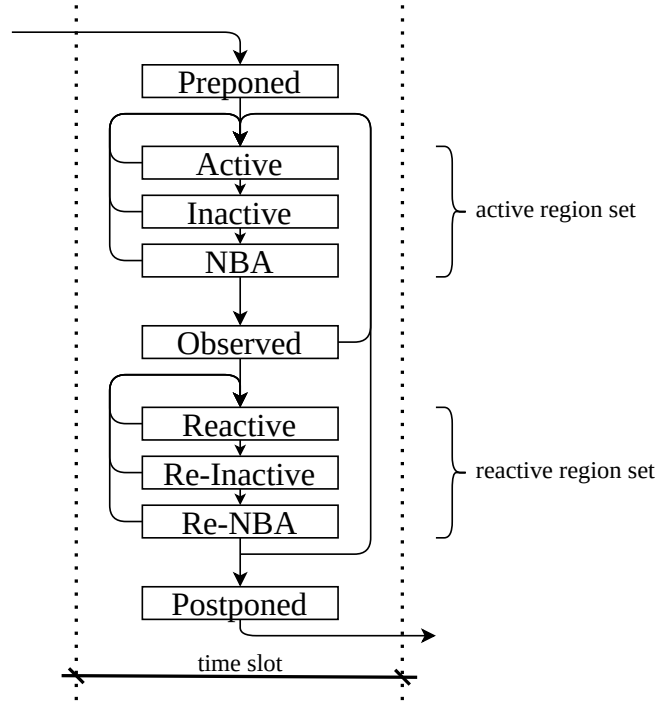


Figure 2.5.: SystemVerilog-2017 scheduling event regions without PLI regions. Adapted from [10, Figure 4-1].

2.4.3. Assertions

SystemVerilog Assertions are an integral part of the SystemVerilog language and add tremendous advantages for verification. SVA allows describing the system's temporal behavior in an expressive and human-readable way, while tools can check the same statements in simulation and formal verification. The term *assertion* must be treated with care as it might be interchangeably used for the `assert` keyword or the set of the SVA statements `assert`, `assume`, `cover`, `restrict`, `expect`⁸.

Assertions can be categorized into *immediate* and *concurrent* assertions. Immediate assertions can be subcategorized into *simple immediate* and *deferred immediate*. Again, the latter is split into *observed deferred* and *final deferred*. It is common practice (and also done in [18]) to abbreviate immediate assertions. In the following, *immediate assertion* refers to *simple immediate assertion*, *observed assertion* to *observed deferred assertion*, and *final assertion* to *final deferred assertion*. Figure 2.6 depicts the correlation between the different types of assertions. The subcategories of immediate assertions differ in the event region in which they are reported (notice figure 2.5).

⁸The `expect` statement is added for completeness and not relevant for further investigation.

2. Background

Table 2.1.: SVA property types and usages.

Type	Usage
<code>assert</code>	Checks the correctness of the design: Must hold in FV and simulation.
<code>assume</code>	Models the environment: Limits the state space in FV; acts like an <code>assert</code> in simulation.
<code>cover</code>	Checks if a property is covered: Trace was observed in simulation; trace is possible in FV.
<code>restrict</code>	Limits the state space in FV (like <code>assume</code>) but ignored in simulation.
<code>expect</code>	Used in testbench code to synchronize on a particular event.

While immediate assertions are being affected by zero-delay simulation glitches, deferred immediate assertions (deferred assertions and final assertions) delay the action block, i.e., reporting a failure (statements evaluated to *o*, *x*, or *z*). Deferred assertions are updated like simple immediate assertions, but their result is buffered in the *deferred assertion report queue*. This intermediate result may be overwritten once evaluation signals change. The evaluation result *matures* in the *observed region*, and the action block is handled in the reactive region for observed assertions. Subsequently, glitches within one region set are prevented. However, if the program (reactive region set) changes a signal and causes a reevaluation of the active region set, the assertion is re-evaluated. *Final assertions* are handled in the postponed region. Thus, glitches caused by the interaction of active and reactive region sets are prevented as well [18].

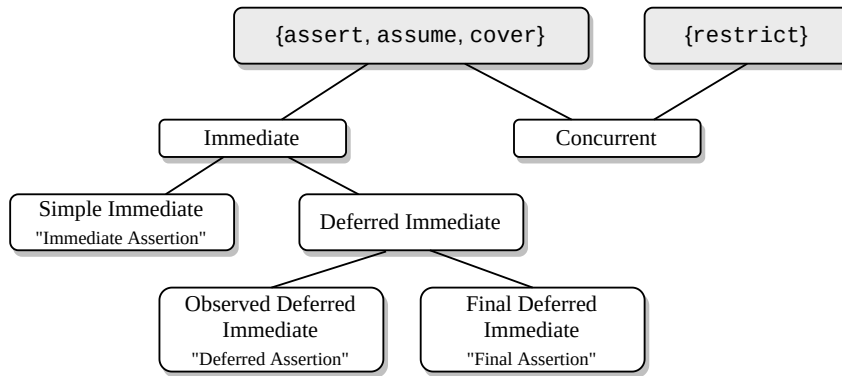


Figure 2.6.: Assertion types in SystemVerilog-2017 [10].

Immediate assertions are typically embedded in procedural code and “are primarily intended to be used with simulation” [10, p. 364]. All assertions designed in the practical part (see chapter 5) are concurrent assertions in a separate checker module bound to the DUT.

2.4.4. Concurrent Assertions

Concurrent assertions are more powerful and can describe temporal behavior, i.e., how signals behave over time. An example would be: *Signal a goes high, one clock cycle after signal b fell*. In this example, an essential concept of concurrent assertions is already present: Time is expressed relative to a clocking event, e.g., the positive edge of signal `clk`. Listing 2.3 expresses the verbal example from above formally as an SVA property.

```
1 asrt_ab: assert property ((@posedge clk) $fell(b) |> $rose(a));
```

Listing 2.3: Example of a concurrent assertion in SVA.

A label (`asrt_ab` in this example) is usually added to keep track of a variety of assertions combined in a module or checker. Tools usually link the labels to the verification results. In simulation, an evaluation attempt is started at each leading clock edge (`@posedge clk`). There might be multiple threads spawned for each evaluation attempt depending on the number of possible paths⁹. In formal verification, the check boils down to a graph search problem.

One of the essences of the previous discussion on the SystemVerilog scheduling semantics is the sampling of values in concurrent assertions. It is crucial when simulating the design and for understanding counterexamples after a formal verification run. Signals evaluated in concurrent assertions are sampled at their clocking event in the *preponed* region. Informally speaking, the samples are captured before any signal changes are applied in the current simulation time step. Figure 2.7 depicts a signal `sig` in a synchronous design. At some clock edge ①, the signal `sig` rises. However, the assertion is evaluated with the *old* value of `sig` as it is captured *before* that change is applied. It takes until the next leading clock edge ② to evaluate the assertion with `sig` being high. Some tools draw the signal change, e.g., the rise of `sig` at ①, with a short delay and limited slew rate to ease inspection.

Instructions in the action block are executed in the *reactive* region, however. When instructions relate to signals from the property, their value might differ from the value taken for evaluating the assertion, i.e., at ① the value in the action block is high. `$sampled(sig)` may be used instead to get the sampled value from the evaluation attempt.

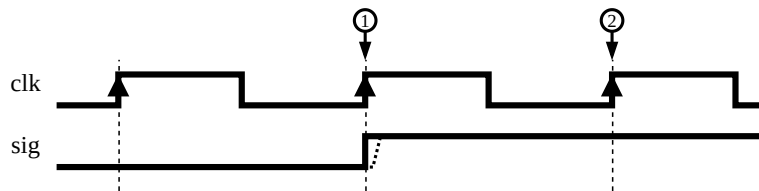


Figure 2.7.: Sampled values in concurrent assertions.

⁹In this example, there is just one path.

2.4.5. Suffix Implication

Although the exact syntax and operators are out of scope and details can be found in the LRM, one operator is particularly interesting when evaluating assertions. Often, assertions are checked after a trigger – a specific precondition – occurred. In terms of SVA, this can be done by the *suffix implication*. Only when the *antecedent* matches, the *consequent* is checked. The example in listing 2.4 shows an assertion that formally describes: *If a is high and in the second cycle after that, b is high followed by a low c, then c must be high for two contiguous cycles beginning from the next cycle.*

```
1 asrt_suffix_impl: assert property ((@posedge clk) a ##2 b ##1 ~c |=> c[*2]);
```

Listing 2.4: SVA property using suffix implication to trigger a check.

As will be see, this statement might be affected by *vacuity* which must be considered in verification. Informally, this statement observes whether the antecedent has a match. Only then it checks the consequent from the *next* clock cycle as a non-overlapping implication ($|=>$) is used. However, the antecedent (a at time t, b at t+2 and $\neg c$ at t+3) might not be reachable in FV or does not occur in the simulation trace. In this case, the consequent is never checked, and the assertion has a *vacuous pass*. One of the possible reasons could be over-constraining. A practical anecdote on why this might be problematic is given in [18]:

What would you say about a civil engineer who constructed a bridge, and to the question “Will this bridge withstand if a heavy truck crosses it?” he will answer “Of course, it will. There was no truck crossing it until now, and the bridge hasn’t collapsed yet”? [18, pp. 123–124].

To keep track of vacuous passes, many commercial tools automatically generate cover statements for assertion antecedents [16].

2.4.6. Environment Modeling and Constraints

The set of possible input values and sequences is usually determined by the surrounding environment, i.e., the neighboring components. During the verification phase, the environment may not be included in the DUT and needs to be modeled to apply only legal input scenarios. The specification might forbid (explicitly or implicitly) some values, combinations, or sequences. Only under these considerations, the system is designed to function as intended. Figure 2.8 illustrates the described environment. One part of the system is the DUT (black). Many neighboring components (grey) are abstracted to cope with complexity¹⁰. Modeling the environment is done differently in simulation and formal verification, as elucidated in the following sections.

¹⁰Another reason might be unavailable models, e.g., when a third-party IP is instantiated.

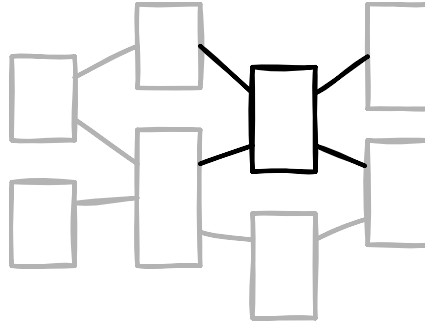


Figure 2.8.: Abstraction of the DUT's environment in functional verification.

Simulation

Test cases are either directed or derived for some randomness. In the first case, the designer must apply only correct stimuli. Assumptions are treated as assertions that check the validity of the applied signals. Constraint expressions [10] can be used to derive randomized test cases within some given rule set (constrained-random verification). Often, constraints are applied to input signals of the DUT. When internal nodes are constrained, it must be considered that the effect is only propagated towards the DUT's outputs [16]. Figure 2.9 illustrates constraints made by the simulation environment on an internal signal of the DUT.

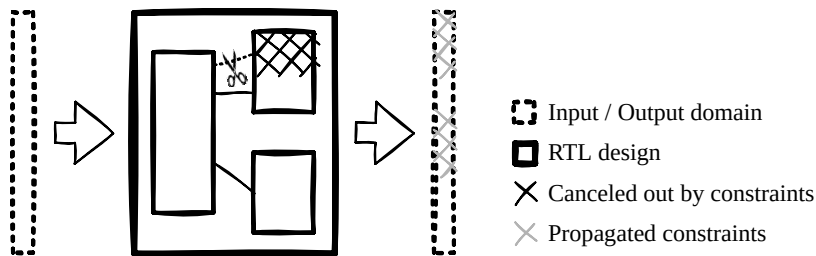


Figure 2.9.: Constraints on internal signals in simulation.

Formal Verification

Assumptions are made on signals to constrain the state space taken into account by the formal tool. Correctly modeling the environment in sufficient detail is considered the main effort [18] in FV. While assumptions are often related to the input ports, they can also be made on internal nets. The constraints are propagated in *any* direction to the DUT's ports as depicted in figure 2.10. Contradictory to simulation, constraints on internal signals also propagate towards the input. Valuations that lead to a violation of the assumption are excluded from the considered input domain.

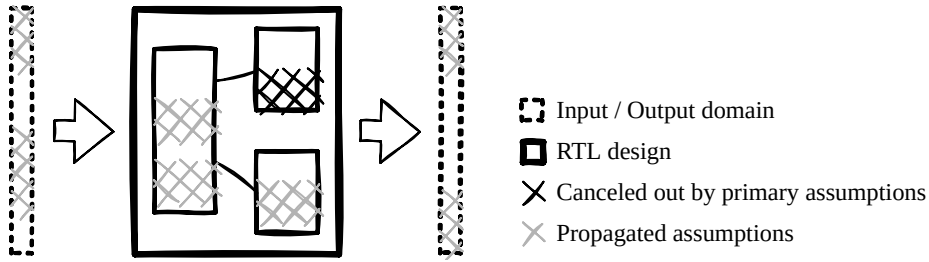


Figure 2.10.: Assumptions on internal nets in formal verification.

2.4.7. Cover Points

Cover properties (see section 2.4.3) are also treated differently in simulation and FV. They are instantiated to inspect interesting states or sequences in a design. In a simulation environment, the stimuli are applied by the surrounding testbench. The DUT operates accordingly, and its internal state is steered through the state space. Cover properties monitor the signals within its property declaration and notify how often the sequence of interest occurred. The testbench must guarantee to apply stimuli that reach the cover property.

When a formal tool is started to process cover properties, the tool itself tries to reach the cover property. If it is reachable, a trace (*witness* [18]) is provided that leads to the cover point. There are multiple situations in which this is useful: FV might be invoked to see if the cover property is reachable when no stimulus can be found in simulation that reaches it¹¹ [16]. Although FV explores the entire state space and cover statements might seem unnecessary at first glance, they prevent over-constraining the model [18]. It is advised to start FPV by writing a set of proper cover properties [16].

Figure 2.11 illustrates the difference between cover properties in simulation and FV. In simulation, the design state is steered through the state space and might hit the cover point after crossing many other states (left). However, FV checks whether the cover point is reachable and, in this case, provides a directed witness (right).

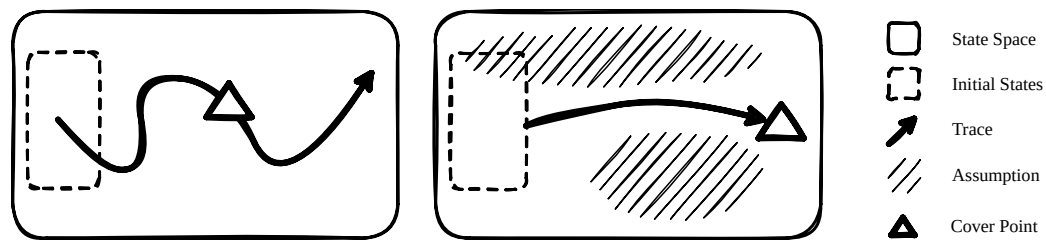


Figure 2.11.: Abstract state space representation of a cover property in simulation (left) and formal verification (right).

¹¹It is also known as Coverage Unreachability (UNR) and is supported by commercial tools [49].

3. Related Work and State-of-the-Art

This chapter outlines related work and gives an overview of state-of-the-art research and applications of FV. Moreover, it discusses techniques to lower complexity and overcome limitations in model checking.

Section 3.1 outlines some projects that successfully enhanced their verification flow by FV. It is shown that FV is a legitimate technique to uncover hard-to-find bugs in real-world applications and explore corner case scenarios. Section 3.2 links to proposed verification flows in literature for FV. SVA is highly expressive, and errors are quickly introduced when writing assertions. Thus, section 3.4 shows an approach to derive formal properties from annotated HDL code automatically. Moreover, section 3.3 presents techniques to efficiently debug formal properties. Besides incorrect assertions, also missing assumptions can lead to assertion violations. An interesting approach to detect those and add them to the formal environment is outlined in section 3.5. Section 3.6 deals with the question of whether enough assertions have been added to the checker and present a methodology that achieves a gap-free verification. As complexity and state space explosion is the main limiting factor of practical model checking, section 3.7 summarizes some measures to allow convergence in larger designs. Finally, section 3.8 gives an introduction to functional safety in the automotive domain and the strengths of FV to reach higher safety levels.

3.1. Applications

This section refers to related work and mentions projects that benefited from formal verification as part of their verification flow.

3.1.1. Error Correcting Code

In [32] Devarajegowda et al. verified an Error Correction Code (ECC) module for correctness by their proposed methodology. The examined logic is part of an automotive microcontroller to prevent random hardware faults. Section 3.8 gives an outlook on fault metrics to be achieved and further use cases for formal verification in the automotive domain. The ECC module encodes two input vectors of 256 bits and 19 bits, respectively. Subsequently, the state space has a size of $2^{256} \cdot 2^{19}$. As the tool did not converge within 96 hours for proving the applied properties, techniques were used to lower the complexity. As already pointed out in section 2.3, state space explosion is one of the main limitations in model checking. For proving the ECC logic, two measures were applied: case-splitting and partitioning the data path (see section 3.7 for other approaches to lower the complexity). Afterward, the proof completed in a feasible time and successfully uncovered a bug in the design specification.

3.1.2. RISC-V Formal

RISC-V is an open-source Instruction Set Architecture (ISA) [50] that is increasingly considered in academia and industry. A vast number of RISC-V cores are freely available on the web. [51] provides an extensive list of the available cores with the corresponding license and implementation language. [52] compares a small sub-set of open-source implementations for different metrics. Besides, many companies implement their own RISC-V microprocessors [53] or consider an open-source variant [54]. Subsequently, it is essential to functionally verify the design against the ISA specification and rule out any deviations in corner cases.

There is active research in new verification methodologies and frameworks – many focusing on open-source tools. While dynamic verification based on an evolutionary optimizer can achieve high coverage [55], there are interesting approaches to formally verify RISC-V cores. One framework presented by Claire Wolf (formerly Clifford Wolf [56, 57]) from Symbiotic EDA¹ got special attention in the open-source community.

The developed *riscv-formal* framework [58] can be used to apply a formal end-to-end verification against a formally specified ISA model. The design needs to implement a RISC-V Formal Interface (RVFI) to exchange information with the verification framework, such as details on each retired instruction. The proofs are based on assertions and split into small chunks to achieve a feasible complexity. Moreover, approximation techniques are used whenever possible. *Instruction checks*, e.g., prove the correct interpretation and execution of individual instructions and can black-box the inter-instruction state. As the framework primarily targets BMC, a reasonable trace length must be chosen. It further needs to be ensured that the specifications to be proven are fully covered in the formal model [53].

[59] combines formal verification based on *riscv-formal* and dynamic verification to successfully uncover hard-to-detect bugs in a 5-stage 32-bit RISC-V processor. Two out of the three discovered bugs were found by the formal verification part of the proposed methodology.

3.1.3. OpenRISC 1200 Security

A different approach is taken in [60]. Security-related properties are derived from the specification and literature of the OpenRISC 1200 (*or1200*) processor to detect critical design faults. The properties are converted into SVA and fed into a state-of-the-art model checker. Thirteen properties are checked on the underlying implementation relating to potential vulnerabilities such as privilege escalation, control flow instructions, exception handling, interrupts and the debugging unit interfering with normal operation. Analyzing the CEX could reveal implementation weaknesses that potentially lead to security hazards.

¹<https://www.symbioticeda.com/>

3.1.4. AHB-Lite to I2C Bridge

Foster et al. [61] apply FPV based on their proposed verification flow to an *AHB-Lite to I2C* bridge for read and write accesses. In their implementation, write requests are buffered in a FIFO. Read accesses are prioritized unless there is a pending write request to the same address. As the design involves memory and an interface to a serial bus, the design has a high complexity for the formal tool. Methods such as compositional reasoning are applied to formally prove the design for correctness.

3.2. Verification Flows

In literature, there are many recommendations for a formal-based verification flow. [21] applies FPV to a 4-bit counter similar to the approach in this work. However, the proposed methodology not described in detail. Kumar et al. [60] propose a similar approach to uncover security vulnerabilities in a processor design. [18] is more precise in the proposed flow and differentiates between an exhaustive formal verification and more lightweight approaches. All proposed flows include an iterative phase of inspecting the counterexample on a failed proof and re-running the formal tool after modifications are applied.

Foster et al. [61] provide additional guidance to set up a hierarchical top-down verification strategy. Their methodology addresses that it might initially not be known whether the entire design is suitable for FV. Their work considers a mix of FV and dynamic verification. Besides, the main three components are covered: assertions, assumptions, and cover properties. The latter is different from dynamic verification as FV does not need to measure *input space coverage*. Their verification foresees initially over-constraining the model and gradually relaxing the assumption to the actual operations. This supports verification while ongoing development and helps track the verification process.

The proposed methodology in this work is given in section 5.1.3 and builds upon the literature from above.

3.3. Debugging Assertions

As stated in section 2.2.1, ABV is a vital part of state-of-the-art hardware verification flows. SystemVerilog Assertions are highly expressive to model temporal system behavior, which results in compact descriptions. E.g., each assertion may start an evaluation attempt at each leading clock cycle and spawn multiple threads that each can have a match. Additionally, properties can be nested and include other properties, sequences, and function calls. Thus, assertions might become very hard to debug, which is a significant bottleneck [11]. This section summarizes achievements and research to help debug and fix SVA properties.

3.3.1. Visualization

Many commercial tools annotate visualized traces to help inspecting SVA properties. Annotations include started attempts, spawned threads, matches of properties and sub-expressions. An example with assertions from chapter 5 is shown in figure 3.1. It depicts four assertions whose state changes over the simulation time. Within the shown time frame, no assertion is violated. Further examples can be found in [39].

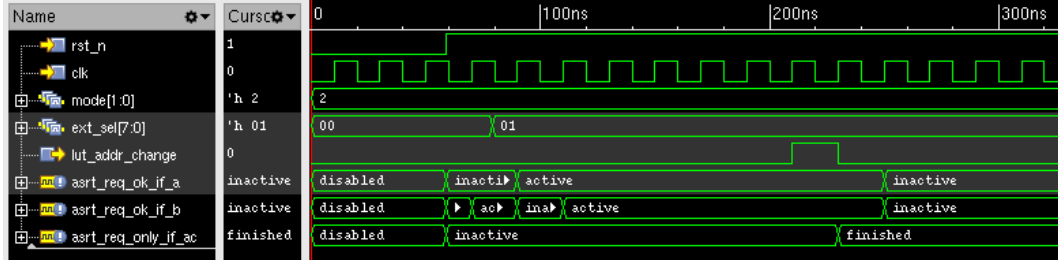


Figure 3.1.: Annotated assertions in simulation.

3.3.2. Structural Debugging and Fan-In Analysis

OneSpin’s² formal tool provides additional guidance for debugging assertions that has been adopted in many industrial projects, e.g., [15]. The techniques are presented in [11] and can find design bugs or nail down defects in the assertion property. It consists of structural debugging and a temporal fan-in analysis. If an assertion fails, the CEX trace is plotted (see figure 3.2, bottom right). The tool highlights the parts that do not conform to the temporal description. The structural debugging displays the high-level SVA statement and marks the parts where something went wrong. The statement’s sub-expression(s) can then be hierarchically explored, and the tool precisely annotates the logical values for each clock cycle from the CEX (see figure 3.2, left). The example shown in figure 3.2 is based on a modified assertion from section 5.2. Siegel et al. [11] demonstrate the feature on SVA statements for an arbiter.

A second valuable feature is the fan-in analysis that identifies the signals on which a sub-expression depends. Figure 3.3 shows an example of the same assertion from above.

3.3.3. Mutation Based

Keng et al. [62] propose a methodology for efficiently debugging assertions based on mutation. Different modifications are applied if an assertion fails. They use a mutation model based on typical industry errors, which also can be customized. Based on these mutations, a set of mutated assertions is generated.

The CEX obtained from verifying the initial assertion is then simulated on the mutants. Violated and vacuously passing mutants are eliminated from this set without much computational overhead. After that, the initial verification flow is invoked to filter candidates

²<https://www.onespin.com/>

3. Related Work and State-of-the-Art

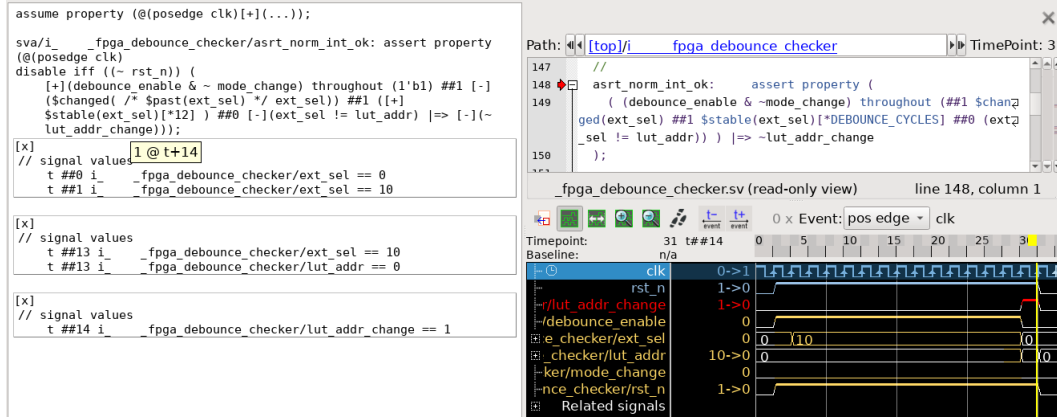


Figure 3.2.: Structural debugging in OneSpin DV.

Fanin Nets	Value	Kind	Module	Hierarchical Time
asrt_norm_int_ok	N/A	System ...	_fpg...	i _fp... 1
> debounce_enable	0	Input	_fpg...	i _fp... 1
> ext_sel	0	Input	_fpg...	i _fp... 1
> lut_addr	0	Input	_fpg...	i _fp... 1
> lut_addr_change	0	Input	_fpg...	i _fp... 1
> mode_change	0	Input	_fpg...	i _fp... 1
> rst_n	0	Input	_fpg...	i _fp... 1

Figure 3.3.: Fan-in analysis of an SVA property.

further. The methodology is tested by artificially introduced errors in RTL code and has successfully found alternatives that can be verified on the model. Due to the strict filtering, the cardinality of the found choices is practical and computed in a suitable amount of time. The manageable set of verifiable modifications can be compared to the original temporal description and used to find modeling issues.

The idea of mutation to correct SVA is taken up by Mostafa et al. [63]. Their methodology aims to apply mutation more directed instead of randomly by a generic fault model.

3.4. AutoSVA

One way of avoiding mistakes when writing assertions is not to write assertions. Orenes-Vera et al. [64] present a methodology to automatically derive end-to-end SVA properties and auxiliary code from an annotated RTL implementation. Their proposed framework focuses on verifying the control logic of transactions (e.g., request/response). Due to the automatic neglectation of the data path, the considered state space and, thus, complexity is lowered. By annotating the interface of a module with specific Verilog comment blocks, the AutoSVA script automatically creates a set of properties (liveness and safety) and additional files for a full formal testbench. The focus is on verifying forward progress. As an alternative to explicit annotation, implicit definitions may be made by matching a proposed naming convention. The framework has been tested on seven RTL modules and showed persuasive results with highly reduced effort to set up the formal environment.

3.5. Missing Assumptions

Formal tools can get overwhelmingly creative for a human mind in finding a CEX. It investigates *all* legal input sequences (even unlogical ones in the given context) to disprove the assertion, i.e., break the assertion. Even with accurate assertions and no design bugs found in the previous simulation, the tool might find a CEX. While the environment is clear from the testbench in simulation, the environment needs to be modeled in FV (see section 2.4.6). If assumptions are missing, the tool might apply an input sequence that cannot happen in the actual application. One difficulty in accurately modeling the environment is that assumptions are often not explicitly mentioned in the requirements but instead given by the interaction with the other system components [65]. Research showed that missing assumptions lead to a significant share of verification failures [66]. Keng et al. [65] proposed a methodology to inspect for missing assumptions automatically.

Their proposed methodology is implemented in C++ based on the *Minisat* SAT engine and works in synergy with a proprietary tool for formal checks. While formal tools only provide *one* CEX, their implementation starts by generating different meaningful counterexamples. Subsequently, the user is given a set of CEXs leading to different failures of the original assertion. The implemented algorithm is provided as pseudo-code and is based on the idea that two CEXs are different when the failure does not share any minimal unsatisfiable subset.

The generated CEXs may be used to check and debug the assertions. Due to different failure modes, the designer might get a more profound intuition about why the assertion is violated. A filtering function is set up that can filter assumption candidates or test if user-defined assumptions are sufficient to satisfy the initially failed assertions. An experiment on different designs is performed, and a couple of the assumption candidates are included in their work. While some seem pretty reasonable, caution is advised with respect to assumptions that lead to a vacuous pass.

3.6. Completeness

Similar to dynamic verification, the question of when to stop testing also arises in FV. Unlike simulation, FV cannot overlook an error by not hitting the right trace through the state space that activates the fault³. However, FV only proves the specified properties. As a result, it needs to be inquired whether enough and correct assertions have been written that do not lead to gaps in the verification [15, 18].

3.6.1. Coverage

Feng et al. [67] give a thorough overview of formal coverage metrics used in industrial applications. *Static Assertion Cone of Influence Coverage* apportions the potentially explored design. It analyzes which design parts may influence each assertion and takes the union over all assertions. Due to fast results, it is an adequate initial indicator but cannot give

³Assuming that the model is not over-constrained.

evidence for verification quality. The Cone of Influence (COI) gets huge quickly and cannot ensure that all the contained design parts are indeed checked [68].

Proof Core Coverage gets more precise regarding the COI actually covered by the formal engine, and it can rule out false-positive coverage on bugs outside the proof core but within the COI. The price, however, is increased computation time.

Mutation-based coverage metrics may also be used but have several disadvantages, such as that it is computationally expensive, requires a fault model, and can lead to vacuity [68].

OneSpin uses a technology called *Quantify*⁴ that analyses the observed statements of checked assertions [68].

3.6.2. Gap-Free Verification

Bormann et al. [15] address the problem of verification completeness. Both formal and dynamic verification may suffer from undetected errors due to gaps in the verification process. They grouped escaped errors into three categories: unstimulated, overlooked, and falsely accepted. The presented methodology prevents the first two groups and minimizes falsely accepted errors by independent modeling and easy reviewability.

The main question in formal verification is whether enough assertions to prove the design intent are written. The presented *GapFree Verification*⁵ methodology automatically analyses the completeness. Each input scenario must be described by a sequence of transactions, i.e., a chain of properties. This includes the input values, a starting state, and the expected output and follow-up state. Moreover, it is specified in which order they can succeed each other.

Two projects by Infineon are presented that applied the proposed methodology. First, a single-pipelined, 32-bit protocol processor with 40 instructions whose verification solely relied on formal verification. It took four engineer-months to complete the verification, which is half of the time a simulation-based verification took in a similar project. Thereby, the evidence on correctness gained through formal verification is much more substantial.

The second project is the Infineon TriCore2. A combination of simulation and FV was used, and FV detected 89 out of 259 errors. An analysis showed that only 40 could have been found without FV. Moreover, FV found 67 errors in the specifications.

3.7. Formal Efficiency

Formal verification can exhaustively verify temporal logic on a given design model. This can be seen as running exponentially many randomized test cases to cover the entire state space [16]. However, FV is generally considered an extremely tough problem. Proving SVA properties is EXPSPACE-complete when intersections of regular expressions are used and PSPACE-complete without intersections of regular expressions and local variables [69].

⁴*Quantify* is a trademark by OneSpin Solutions.

⁵*GapFree Verification* is a trademark by OneSpin Solutions.

Formal tools usually invoke a set of engines based on different heuristics to achieve a result [16]. Considerations on efficiency have to be made, however, to allow the engines to come up with a converging proof. This section focuses on how engineers can cope with the complexity of modern design in FV when using the provided formal tools.

3.7.1. Suitable Designs

Not all designs are equally suited for FV. Foster et al. [61] classify designs to predict whether FV or simulation might be a better choice and gives concrete examples. The first differentiation is made between *concurrent* and *sequential* designs. In the former, many streams of data are handled that possibly collide with each other such as an arbiter. These may be considered for FV. In sequential designs, streams of data are separated and do not interfere. In this case, dynamic verification against a high-level model might be a preferable choice.

Moreover, designs can be categorized into *control-oriented* and *data-oriented*. The latter can again be split into whether data is manipulated or transported only. While FV might be a good choice for control-oriented modules, it is usually inefficient for data transformation blocks due to the large state space within the COI. Data transport blocks can be considered for FV as the data path may be abstracted, leading to fewer observed states.

Examples of well-suited modules are interrupt controllers, memory controllers, DMA controllers, power management units, and standard interfaces. Data-transformation modules such as floating points units and graphics shading units, by contrast, might not be verified with FV. However, this does not imply that FV of data transformation blocks is generally unattainable. Ram et al. [70] presented an approach to formally verify a floating-point unit against the IEEE 754 standard.

3.7.2. Efficient Property Design

Writing efficient properties largely contributes to the overall complexity. Efficiency targets, however, may differ for FV and dynamic verification and may even be contradictory. When properties are meant to be used in both verification techniques, priority shall be given to optimizing for FV, as it is more critical [18].

The complexity of assertions highly depends on the COI, i.e., the amount of circuitry influencing the property [16]. The tool may disregard circuit parts outside the COI, and the remaining complexity correlates to the parts influencing the property. Reduction techniques, however, can reduce the resulting complexity of the COI [61], which may complicate a priori complexity estimations. When the proof does not complete, one might inspect the COI as a first step. Commercial tools can analyze and sort signals and assumptions by their influence on the complexity of the assertion to be proven [71].

Generally, assertions shall be kept small and include just a few signals with little sequential depth. For this reason, assertions can be decomposed [72] as shown in listing 3.1. Assertion `asrt_x` requires two signals (`x1` and `x2`) to be high in the consequent. The assertion can be split into two equivalent assertions (`asrt_x1` and `asrt_x2`) that both must hold. Similarly,

asrt.y can be split into two smaller assertions. Simplifying the assertions support the formal tool and possibly also the designer to debug failing assertions.

```
1 asrt_x:  assert property (@(posedge clk) a |-> (x1 & x2));
2
3 asrt_x1: assert property (@(posedge clk) a |-> x1);
4 asrt_x2: assert property (@(posedge clk) a |-> x2);
5
6 asrt_y:  assert property (@(posedge clk) (a | b) |-> y);
7
8 asrt_y1: assert property (@(posedge clk) a |-> y);
9 asrt_y2: assert property (@(posedge clk) b |-> y);
```

Listing 3.1: Decomposition of assertions to lower the complexity for a conjunction in the consequent [72] and disjunction in the antecedent.

The complexity and resulting runtime of the tool depend not only on the number and structure of assertions but also on the assumptions made on the system. Reducing assumptions can positively influence complexity. This is especially true when assumptions are removed that introduce new signals to the COI, i.e., signals that are not already part of the COI of the assertion [73].

However, also adding assumptions may reduce the complexity [71]. Assumptions can instruct the tool to neglect design parts that are not important for the proof, such as operating modes that are not considered [16]. It needs to be ensured that the model is not over-constrained (underapproximated) as incorrect assumptions might lead to false-positive assertions [18]. [16] summarizes that *simple* assumption crosses out the complexity of the design while detailed and complex assumptions add even more complexity due to the fine-grained incisions in the state space. Already proven assertions may also be converted into assumptions to support subsequent proofs.

Auxiliary Code

Auxiliary code (also called *HDL helper code*) is an ordinary SystemVerilog code that is written to support verification. It may be used to lower the complexity of SVA properties or express behavior that is not possible with pure SVA. Auxiliary code is more efficient for the formal tool as there cannot be multiple threads spawned, and there is only one instance. Thus, the state space and complexity are reduced [74].

Moreover, auxiliary code can express overlapping behavior that is impossible with pure SVA. E.g., it can be used to describe that every request receives *its* grant. A single grant would satisfy all pending requests if the request signal were placed in the antecedent with the grant signal in the consequent. It cannot be expressed that each request must receive a separate grant. Auxiliary code, however, can keep track of the different requests raised with a counter [74, 75].

3.7.3. Abstraction and Other Techniques

When targeting larger designs, more sophisticated considerations might be required for a converging proof. Among the techniques used in industry are *black boxing*, *cut-points*, *pruning*, *case-splitting*, *shadow modeling*, and *concolic testing* [16, 18].

Black boxing can be instructed by commands to the formal tools and abstracts a design part. Its output signals become free variables to the DUT that can take any value. Usually, this technique is applied to modules that contribute primarily to the state space size, such as memories. Similarly, cut-points are used to break certain nets in the design [76]. When the formal proof holds on the more general model, it also holds on the original model [61].

Case-splitting is applied to prove independent functionality in different verification runs. If the behavior does not correlate, assumptions can temporarily over-constrain the model to neglect all functionality not part of the current proof and reduce the state space size. One must ensure that the union of all proof parts implies the overall proof and no holes are left out [16]. Assumptions may also be used only to consider a sub-set of functionality in early verification [61].

Design parts may also be replaced by an abstracted version that lowers the complexity of the proof. In [77] a counter is replaced by a simpler state-machine. The substitution can be done non-intrusively by tool commands such as cut-points to open the original connections.

It needs to be ensured that the model is not over-constrained inadvertently by adding too general assumptions. In this case, the restrictions are too tight and do not allow the actual behavior in the final product. Thus, the tool might return a misleading positive proof result. The designer might think everything is fine, although there might be an error in the actual product that is not checked during verification. [61] compares this conceptually to insufficient functional coverage in dynamic simulation. Manually written cover properties and automatically generated cover properties for assertion antecedents support detecting an over-constrained environment. Moreover, assumptions might be checked by using them as assertions on the neighboring components if possible, known as *Assume-Guarantee Paradigm* [18, 61].

3.8. Formal Verification for Safety-Critical Systems

It does not suffice for a digital design to be proven systematically correct in safety-critical systems. Although preventing systematic faults is essential, measures must be taken to detect faults in the field and be tolerant against some of the occurring faults [78, 79]. It correlates with the definition of functional safety as given by Chris Hobbs: "Something must continue to *function* in order to keep the system safe" [6, p. 16].

Unlike *systematic faults*, which are introduced while developing the system, *random hardware faults* occur at any time during operation. They can be caused by wear-out, radiation, and power supply glitches. Miniaturization in modern process technologies and tight power budgets increases the likelihood of their occurrence [14].

3.8.1. Automotive Standards

Many standards regulate functional safety in different areas. For the automotive domain, ISO 26262⁶ is the applicable series of standards. It is derived from IEC 61508, a more general standard for "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems" [80]. ISO 26262 "is intended to be applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems and that are installed in series production road vehicles, excluding mopeds" [81, p. 1]. It addresses functional safety, defined as the "absence of unacceptable risk due to hazards caused by malfunctioning behavior of E/E systems" [81, p. 14] and is split into 12 parts. ISO 26262 guides the complete product lifecycle.

Interpreting and implementing an ISO 26262 complied flow is a comprehensive task and requires much expertise. Although complying with functional safety standards is a significant factor in development costs, suppliers, semiconductor companies, and tool providers must conform to the ISO 26262 standard to be accepted by the Original Equipment Manufacturer (OEM) [20, 82]. A complete overview of the product development phases and required work products is out of the scope of this work and a detailed introduction with many practical anecdotes is given, e.g., in [6]. Although the focus is on software, fundamental ideas and processes are also crucial for hardware development. [82] and [83] provide a brief overview of the development phases and then focus on hardware faults to be considered, which are covered in part 5 of ISO 26262.

3.8.2. Application of Formal Verification

Safety mechanisms might be added to detect faults and prevent system failure due to random hardware faults in safety functions. Formal verification can support the development in two ways [20].

- (a) Safety mechanisms are required to be systematically correct. A fault shall be reported if and only if a fault is present. Thus, bugs in the safety mechanism can lead to a safety goal violation.
- (b) Formal verification can be invoked to prove properties under fault conditions.

ISO 26262 is very explicit about fault metrics. One of which is the Single-Point Fault Metric (SPFM) that "reflects the robustness of the item to single-point and residual faults either by coverage from safety mechanism or by design (primarily safe faults)." [84, p. 41]. For Automotive Safety Integrity Level (ASIL) B, C, D, the SPFM needs to be larger or equal to 90%, 97%, or 99%, respectively. Loosely speaking, the SPFM is the ratio of the single point faults (i.e., not covered by the safety mechanism) plus the residual faults (i.e., escape safety mechanism) to all faults. It is crucial to classify faults and identify safe faults, i.e., faults that cannot violate the safety goal to achieve an acceptable ratio. There are many research papers discussing fault classification and safe fault detection. Many relying on formal methods [20, 78, 79].

⁶All references to ISO 26262 refer to the current version, ISO 26262-x:2018(E). For the sake of readability, the version is omitted.

In [79], researchers present a methodology that combines the strengths of Automatic Test Pattern Generation (ATPG), Fault Injection (FI) simulation, and formal methods to relieve the burden of functional safety verification. The developed *Fault Checker* can be configured for different tools and executes formal and simulation-based verification in parallel. The ATPG generates an environment for the FI simulation, and formal methods are applied to classify safe faults. The tool generates a combined report and highlights discrepancies to increase the tool's confidence level. An anomaly can be assumed when the formal tool classifies a fault as safe while the simulator detects it. The combination of techniques yielded a detection rate of over 99% on their designs under test.

Recently, Felipe et al. [78] have proposed another methodology to classify residual faults and identify safe faults without manual analysis. Their work emphasizes the detection of *determined-safe faults* by formal verification. *Structural-safe fault* can be classified by automatically generated formal properties that prove that a fault cannot propagate to the output for any input stimuli. Thus, it can never cause a violation of the safety goal and is considered a safe fault. Unlike structural-safe faults, determined-safe faults can propagate to the output but do not affect the safety function. Potential candidates are determined by simulation and code coverage. Formal tools can then exhaustively check properties defined on the gathered candidates. E.g., for a counter that is never read, all faults affecting the counter value can be classified as determined-safe in that operation mode. The proposed methodology has been validated on an SJA1000 CAN controller. It increased the diagnostic coverage by around 6% to a final ratio of 93.04%.

4. Design Specifications

Chapter 4 presents the design requirements and specifications of the developed evaluation kit for Lidar transmitters. Firstly, section 4.1 gives a brief overview of the overall goals and project intentions. Then, in section 4.2, the project is investigated at a system level. After the fundamental concepts have been delineated, section 4.3 focuses on the digital design, which is subject to verification. Finally, the architecture and sub-modules composing the FPGA's functionality are depicted and described in section 4.4. Many timing diagrams support the verbal description to provide an unambiguous basis for verification. The given details focus on the aspects covered in chapter 5. Project specifications not required for further investigation may be skipped.

4.1. Project Introduction

The designed system is part of an evaluation kit to explore and test different Lidar Vertical Cavity Surface Emitting Laser (VCSEL) transmitters in various scenarios. It aims to give customers a hands-on experience and to explore new use cases. The initial development targets an automotive Lidar driver IC. The VCSEL die has 72 addressable rows bonded to the driver IC. Two different versions are available with 905nm or 940nm wavelength. Although the target Lidar transmitter is designed according to the ISO 26262, the requirements of the evaluation kit on functional safety and eye safety are less strict. It is exclusively operated by professionals in a laboratory environment. The focus is on developing a generic board that can be used beyond the initially considered use cases. It shall be seen as a generic basis that can quickly be adapted to different Lidar transmitters in future applications and showcases. At the time of writing this thesis, the Lidar kit is planned to be used for internal evaluations and by selected customers. Its usage might be extended in the future.

Figure 4.1 provides an abstracted high-level view on the circuit board being developed. It consists of a Single-Board Computer (SBC), configurable digital hardware (a Field Programmable Gate Array (FPGA) in this case), components for the Lidar transmitter unit, and a mounting mechanism, i.e., precisely defined mounting holes for the optics.

The board's operation can be split into two categories: In the non-real-time operating mode, the drivers can be configured from a PC. Settings can either be adjusted from a Graphical User Interface (GUI) or by executing scripts. One of the settings selects the active VCSEL row¹, which is illuminated on a trigger pulse. Besides setting the active VCSEL row from the GUI, it shall be possible to change the row at a much higher (and deterministic) rate than possible from a PC with a generic operating system. Thus, pins are exposed on the circuit board to address the row by applying a binary encoded electrical signal. In the

¹No more than one row is active at each point in time.

real-time operating modes, the FPGA encodes the applied address and updates the drivers' configuration data accordingly. This task is considered to be hard-real-time. The trigger to illuminate² the selected row happens at the rising clock edge and turns it on for a pre-set pulse width.

The evaluation kit is split into two circuit boards: the mainboard (*Tx Main*) and the board containing the VCSEL die (*Opto*).

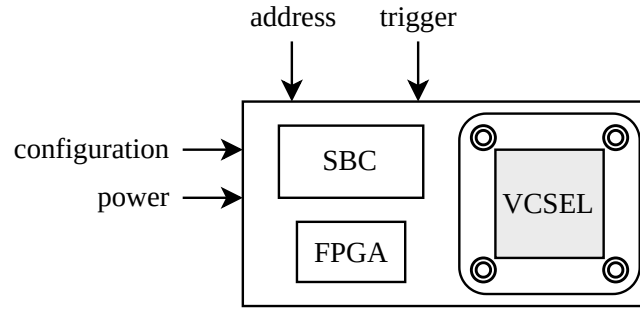


Figure 4.1.: High-level view on the Lidar transmitter evaluation board.

4.2. System View

One of the main intentions of the mainboard design is adaptability to different use cases. It shall provide a basis for quickly implementing and testing various scenarios in future development. Thus, the mainboard is equipped with a generic Raspberry Pi (RPi) Zero single-board computer together with an FPGA. The RPi takes on resource-intensive tasks that can be programmed at a high level. The FPGA provides real-time capabilities that are not available with the RPi. It may directly implement an application specific digital design or forms a synergy with a soft-core microcontroller. Figure 4.2 depicts the main communication channels and interactions. Hardware components and signals not relevant for further discussion are neglected.

The RPi shall handle the higher-level software communication with the GUI. In future extensions, the wired communication to the PC might be replaced by a webserver spawned by the RPi. In this case, the GUI may be provided by browsing a website without any additional software needed. For now, the RPi shall act as a USB to SPI/GPIO bridge under the control of the customer's PC.

This work focuses on verifying the digital design implemented in the FGPA. The following section consists of a precise explanation of the digital system's specifications.

²Often referred to as *shooting*. Hence, some signal names carry the name *shot*.

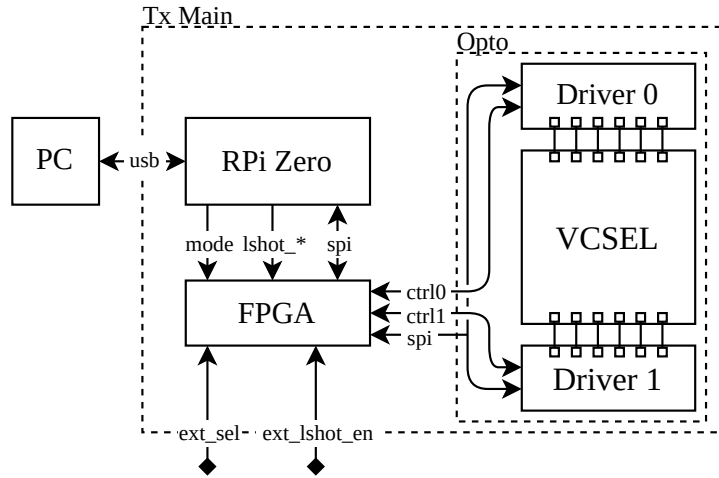


Figure 4.2.: Primary interaction channels between the components.

4.3. Digital System Requirements

In this section, the requirements of the digital system are elaborated in more detail. Some specification details are postponed to the architecture decomposition to avoid text duplications. This consideration is made as the main focus is verification, not deriving a design from design requirements.

4.3.1. Pinout

The abstracted system-level view on the FPGA interface from figure 4.2 is refined in figure 4.3. All signals entering and leaving the FPGA are explicitly indicated, and grouped signals are split up. Table 4.1 lists the signal names together with a short description. Signals that are named similar for multiple instances but are not used as a vector in the implementation are abbreviated using an x and vector notation to denote all instances. E.g., `lshot_en_x_from_fpga[0:1]` refers to `lshot_en_0_from_fpga` and `lshot_en_1_from_fpga`.

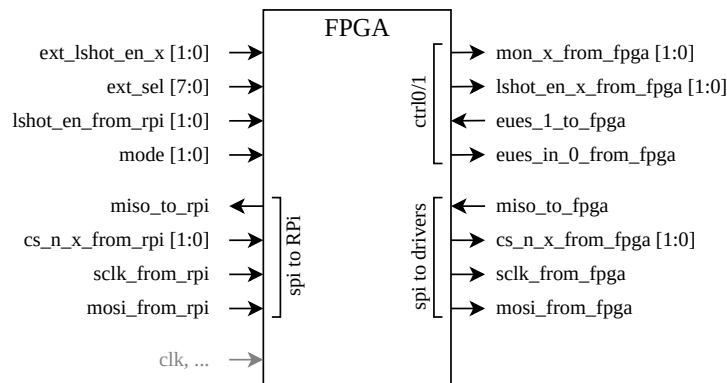


Figure 4.3.: FPGA pinout.

4. Design Specifications

Table 4.1.: FPGA pinout description.

Name	Width	Direction	Description
ext_lshot_en_x	2	in	External signals to request driver unlock.
lshot_en_from_rpi	2	in	RPi output to request driver unlock.
lshot_en_x_from_fpga	2	out	Unlock driver $\{0, 1\}$ for pulsing.
ext_sel	8	in	Row address input; customer numbering schema.
mon_x_from_fpga	2	out	Driver control signal. ³
eues_out_1_to_fpga	1	in	Driver status signal.
eues_in_0_from_fpga	1	out	Driver control signal.
mode	2	in	Select FPGA operating mode.
miso_to_rpi	1	out	SPI: Data from FPGA to RPi; only active in operating modes M_0 , M_1 and M_3 .
cs_n_x_from_rpi	2	in	SPI: Chip-select for driver $\{0,1\}$ in modes M_0 and M_1 . Inactive in mode M_2 . Chip-select for FPGA device in mode M_3 .
sclk_from_rpi	1	in	SPI: Clock; inactive in mode M_2 .
mosi_from_rpi	1	in	SPI: Data from RPi to FPGA; inactive in mode M_2 .
miso_to_fpga	1	in	SPI: Data from drivers to FPGA; inactive in mode M_3 .
cs_n_x_from_fpga	2	out	SPI: Chip-select from FPGA; inactive in mode M_3 .
sclk_from_fpga	1	out	SPI: Clock to drivers; inactive in mode M_3 .
mosi_from_fpga	1	out	SPI: Data to drivers; inactive in mode M_3 .

³As the usage is not relevant for the verification, details are neglected for the scope of this work.

4.3.2. Operating Modes

The RPi can set four operating modes via the mode input vector which are listed in table 4.2. For convenience, the modes are abbreviated by M_0 , M_1 , M_2 and M_3 for $M_0_ROUTE_THROUGH$, $M_1_EXT_STREAM_TRG$, $M_2_EXT_ROW_SEL$, and $M_3_SER_CTRL$, respectively.

Table 4.2.: FPGA operating modes.

Mode	Description
$M_0_ROUTE_THROUGH$	SPI packets can be exchanged between RPi and VCSEL drivers. No real-time change of the active row is possible.
$M_1_EXT_STREAM_TRG$	Similar to mode M_0 . The differences are listed below.
$M_2_EXT_ROW_SEL$	The FPGA becomes the SPI master and can actively start an SPI communication to the drivers. Each time the row address at <code>ext_sel</code> changes, the FPGA generates an SPI packet including the requested row and sends it to the correct VCSEL driver.
$M_3_SER_CTRL$	The FPGA acts as SPI slave to the RPi. Addressable <i>config/status</i> registers may be read and written.

Mode 0

In operating mode $M_0_ROUTE_THROUGH$ the drivers can be configured from the GUI. Subsequently, the RPi receives data from the PC and generates SPI frames to update the content of the configuration registers of the VCSEL drivers. SPI communication is routed through the FPGA to the drivers (bidirectionally), and the FPGA does not intercept the communication. The main requirements are listed in table 4.3.

Table 4.3.: Operating mode M_0 .

Nr	Description
$M_0_1.0$	The FPGA shall connect the SPI bus interfaces and allow uninterrupted access to the drivers' configuration registers. All SPI lines (<code>cs_n_x_from_rpi</code> , <code>mosi_from_rpi</code> , ...) shall directly be assigned to their output counterparts (<code>cs_n_x_from_fpga</code> , <code>mosi_from_fpga</code> , ...). The SPI clock shall be assigned to the output as well. Propagation delays may be neglected for low clock frequencies.
$M_0_2.0$	<code>ext_sel</code> and <code>ext_lshot_en_x</code> shall be ignored.
$M_0_3.0$	<code>lshot_en_x_from_fpga</code> shall be assigned to <code>lshot_en_from_rpi</code> .
$M_0_4.0$	<code>mon_x_from_fpga</code> shall be zero.
$M_0_5.0$	<code>eues_in_0_from_fpga</code> shall be assigned to <code>eues_out_1_to_fpga</code> .

Mode 1

Mode *M1_EXT_STREAM_TRG* is similar to *Mo*. Differences are listed in table 4.4.

Table 4.4.: Operating mode *M1*: differences to *Mo*.

Nr	Description
<i>M1_1.0</i>	<i>lshot_en_x_from_fpga</i> shall be assigned to the conjunction of <i>lshot_en_from_rpi</i> and <i>ext_lshot_en_x</i> .
<i>M1_2.0</i>	<i>eues_out_1_to_fpga</i> shall be zero.

Mode 2

In mode *M2_EXT_ROW_SEL*, the FPGA acts as SPI master. The RPi shall not start communication while this mode is active.

Table 4.5.: Operating mode *M2*.

Nr	Description
<i>M2_1.0</i>	On a change of the external address (<i>ext_sel</i>), a <i>debounce</i> attempt is started. Only when the address is stable for at least a pre-defined number of cycles (e.g., 40 cycles) the internal address is updated, and a request is raised to emit an SPI frame.
<i>M2_2.0</i>	<i>lshot_en_x_from_fpga</i> shall be zero for the inactive driver (only the driver selected by the address is considered to be active). For the active driver, <i>lshot_en_x_from_fpga</i> shall be the conjunction of <i>lshot_en_x_from_rpi</i> and an internal <i>int_lshot_en_mask</i> signal from the control FSM. Details are given in figure 4.5.
<i>M2_3.0</i>	<i>eues_in_0_from_fpga</i> shall be assigned to zero.
<i>M2_4.0</i>	<i>mon_x_from_fpga</i> shall be <i>lshot_en_x_from_fpga</i> .

Mode 3

M3_SER_CTRL is meant to read status information and write configuration data to the FPGA.

4. Design Specifications

Table 4.6.: Operating mode M_3 .

Nr	Description
$M_3_{1.0}$	The FGPA is the SPI slave and registers can be accessed by a frame consisting of the register address and data (or padding in case of a read).
$M_3_{2.0}$	<code>cs_n_0_from_rpi</code> is used to address the FPGA. <code>cs_n_1_from_rpi</code> shall not be set to zero in this mode.
$M_3_{3.0}$	Both <code>cs_n_x_from_fpga</code> shall be high.
$M_3_{4.0}$	<code>lshot_en_x_from_fpga</code> shall be assigned to <code>lshot_en_x_from_rpi</code> .
$M_3_{5.0}$	<code>eues_in_0_from_fpga</code> shall be assigned to zero.
$M_3_{6.0}$	<code>mon_x_from_fpga</code> shall be zero.

Mode Change

The RPi controls the selected mode and shall only change between modes when allowed. Mode M_2 shall not be left while the FPGA master transmits data to the VCSEL drivers. Thus, the mode shall be stable from any trigger attempt to the expected end of the emitted frame. Although the timing is not precisely known to the RPi, sufficient protection time can be added as the modes are meant to be changed upon human request. Mode M_3 shall not be left while the RPi reads or writes register data. All other modes can be changed without restrictions.

Timing Diagrams

The following timing diagrams clarify possible ambiguities due to verbal description. In figure 4.4 the control signals in modes M_0 , M_1 and M_3 are illustrated. `lshot_en_x_from_fpga` is taken over from its input counterpart in modes M_0 (a \rightarrow b) and M_3 (h \rightarrow i). In mode M_1 it is composed of a boolean conjunction of `ext_lshot_en_x` and `lshot_en_x_from_rpi` (e \rightarrow g). `eues_in_0_from_fpga` gets `eues_1_to_fpga` in M_0 (c \rightarrow d) and is constantly low in modes M_1 and M_3 .

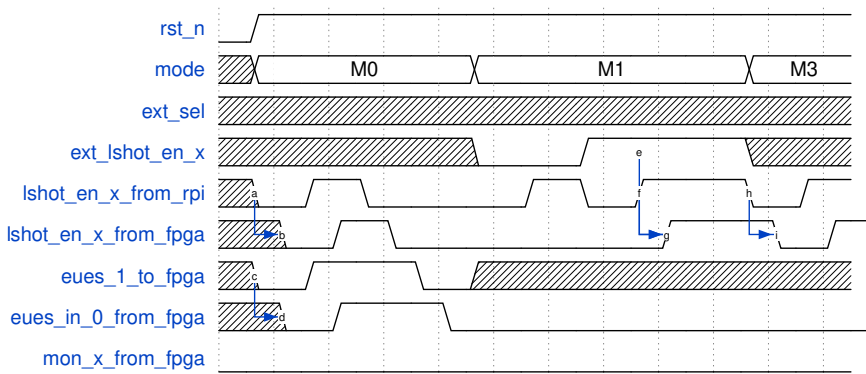


Figure 4.4.: Timing diagram for control signals in mode M_0 , M_1 and M_3 .

Figure 4.5 depicts the main Finite State Machine (FSM) in mode *M2*. After the mode is activated (and `ext_sel` is different to `00h`), or `ext_sel` changed, a debounce attempt is started at the next rising clock edge (marked with a). The debounce attempt succeeds after `DEBOUNCE_CYCLES`⁴ clock cycles. One cycle afterwards, a request is raised and `lut_addr` is updated (`b` → `c`). At that clock cycle, the FSM leaves the idle state. First, the `int_lshot_en_mask` is set low in case it was high before (`f` → `g`). Then, it remains low until the frame is fully sent and a post-delay is passed. On each change of the active mode, `int_lshot_en_mask` is set low and remains in this state until the first frame has been sent. After turning off the mask, a frame is generated and sent to the corresponding driver. The receiver is determined by `chip_id` – the LSB of the translated address. After the frame has been sent and the post-delay passed, the mask is set again and the FSM returns back to idle one cycle later.

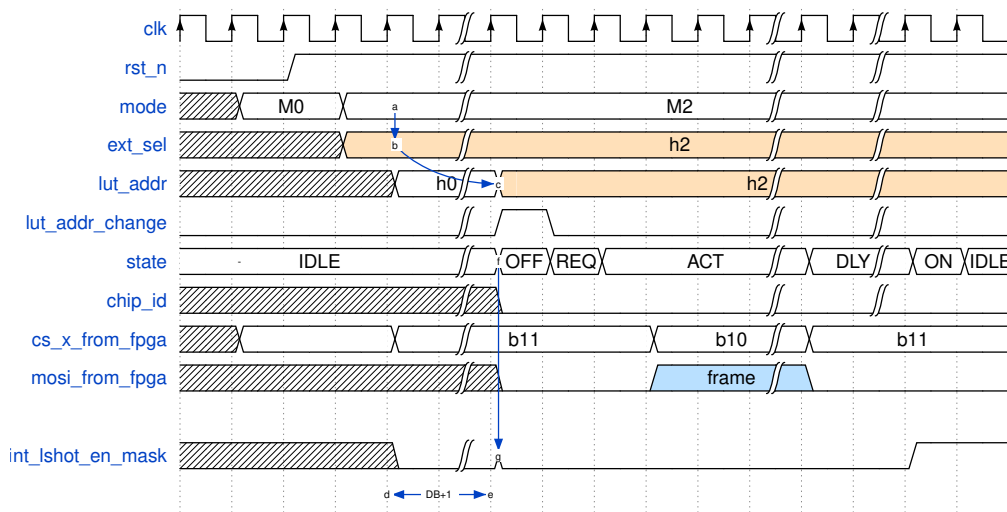


Figure 4.5.: Timing diagram of the main FSM in mode *M2*.

4.4. Derived Architecture

The specifications given in section 4.3 are refined to get a concrete partitioning of the digital design. Figure 4.6 depicts the modules and architecture of the implementation. Table 4.7 gives a brief description of the building blocks in the *core* module and their usage.

Chapter 5 focuses on the verification of the *debounce* module and the top-level *core* module. For this reason, the *debounce* module is further described in section 4.4.1 as a cycle-accurate specification is required. The applied checks on the *core* module can be proven without further details.

⁴Abbreviated with *DB*.

4. Design Specifications

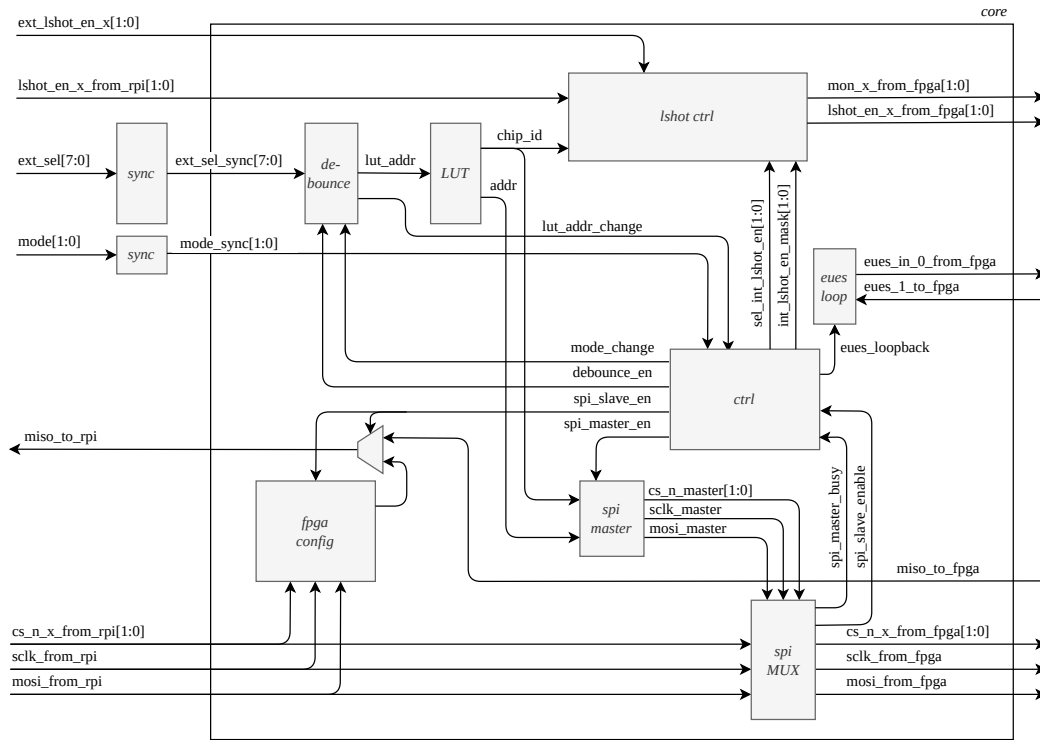


Figure 4.6.: Refined FPGA architecture.

Table 4.7.: Submodules of *core*.

Module	Description
<i>debounce</i>	Debounces the applied address before a request to update the drivers' information is raised. See section 4.4.1 for a cycle-accurate specification.
<i>LUT</i>	The Lookup Table (LUT) translates the customer addressing scheme to the internally used addresses. Moreover, it extracts the target driver (<i>chip_id</i>), i.e., the receiver of the frame.
<i>lshot ctrl</i>	Generates certain control signals such as <i>lshot_en_1_from_fpga</i> and <i>eues_in_0_from_fpga</i> .
<i>eues loop</i>	This block is also related to control signals for the drivers. Depending on the mode, the incoming <i>eues</i> signal is either looped back or interrupted.
<i>ctrl</i>	The control logic block includes the main FSM to generate the control signals for other modules depending on the active mode.

continues on next page

<i>spi master</i>	The <i>Spi Master</i> is activated in mode M_2 by the control unit. It waits for a request and builds the SPI frame from the translated address and some fixed parameters. Then, the frame is transmitted to the target driver selected by <code>chip_id</code> .
<i>spi MUX</i>	The <i>Spi MUX</i> selects the SPI master that is connected to the VCSEL driver. It can be either the internal SPI master or the incoming bus lines from the RPi. As priority is given to the internal SPI master, the RPi shall not communicate while the FPGA's SPI master is active. Otherwise, in-flight communication might be interrupted. Moreover, the MUX can disconnect the drivers from the bus, which is required in mode M_3 .
<i>fpga config</i>	The <i>fpga config</i> block implements the internal SPI slave and config registers that can be read and written. The block is deactivated by the <i>ctrl</i> block in any other mode than M_3 . If mode M_3 is active, it listens to the same chip-select signals as driver zero. <i>Spi MUX</i> disconnects the drivers in this mode to avoid a conflict.

4.4.1. Debounce

The *debounce* module ensures that the applied address (`ext_sel`) is stable for a pre-defined amount of time before a request to update the active row is raised. Moreover, it offers some control signals to deactivate the operation and re-trigger requests. On each request, the SPI master generates and sends a frame to the VCSEL driver to activate the row selected by `ext_sel`. Any address apart from FF_h is considered legal.

Figure 4.7 depicts the timing diagram of a typical operation in mode M_2 . `rst_n` and `debounce_enable` are set to high to release the module from reset and enable it, respectively. `mode_change` is set low and can be ignored in the shown diagram. Once `ext_sel` changes its value, i.e., one or more of its bits are different to the last clock cycle, the module internally starts a debounce attempt. When the value has been stable for DB clock cycles, a request is raised at the next rising clock edge, i.e., $DB+1$ cycles after the latest change of `ext_sel`. `lut_addr` is set to zero on a reset. When the module is released from reset, a change of `ext_sel` is assumed when its value is different to the initial value of `lut_addr`. This point in time is marked with a. After $DB+1$ cycles, the address A is taken over to the `lut_addr` output and the change is indicated by a pulse on the request line (`lut_addr_change`) marked with b and c, respectively. The input address changes again at f. As it has not been stable for long enough (shorter than DB cycles), no request is triggered at h. `ext_sel` changes at g to the value C and back to B after exactly DB cycles. This is just long enough to trigger a request in the next cycle and set `lut_addr` to C ($g \rightarrow 1$).

4. Design Specifications

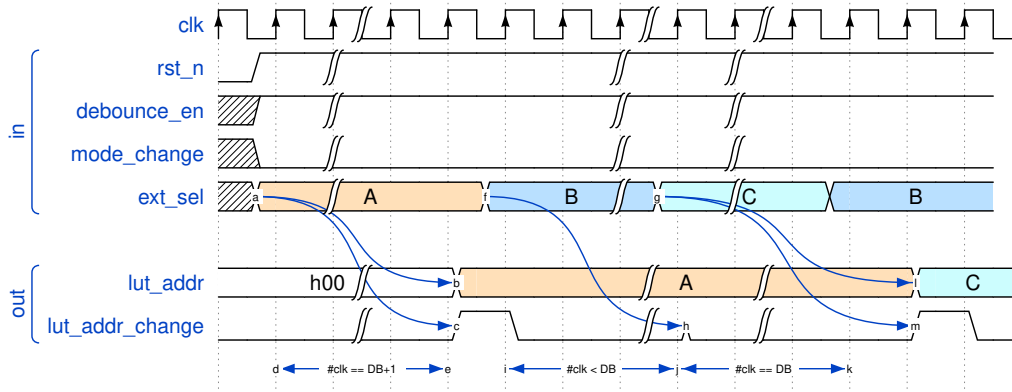


Figure 4.7.: *Debounce* module: operation without disable and inactive *mode_change*.

The behavior on a dynamic reset is illustrated in figure 4.8. As soon as *rst_n* is pulled down, *lut_addr* is set to zero independent of its previous value ($i \rightarrow j$), and *lut_addr_change* remains low. When reset is released, a request is raised after $DB+1$ cycles if *ext_sel* is stable for DB cycles beginning with the rising edge of *rst_n* ($d \rightarrow h$).

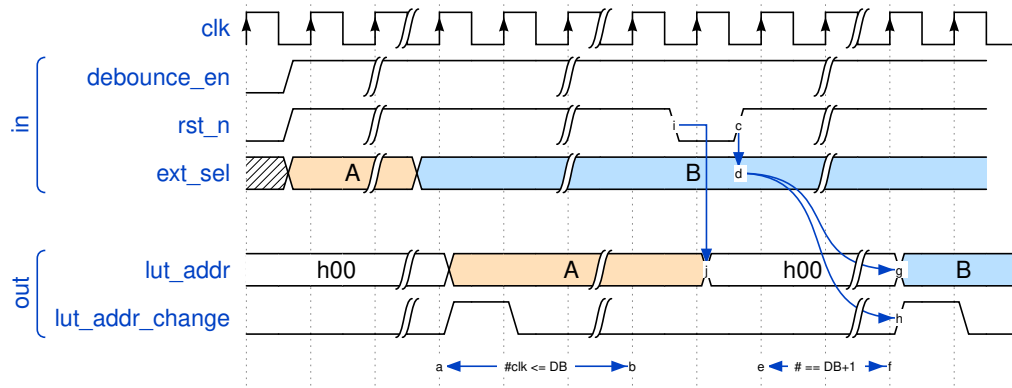


Figure 4.8.: *Debounce* module: behavior on dynamic reset.

Figure 4.9 depicts the behavior when the module is disabled and re-enabled. Once *debounce_enable* goes low, any attempt is immediately terminated ($a \rightarrow b$). The currently present state at *lut_addr* is preserved. When enabling the module again there are two different cases: Either *ext_sel* has been stable while *debounce_enable* was low, or it has changed during this period. In the first case, no request is triggered on that value of *ext_sel*⁵, even if the original attempt was terminated. If it has changed, a new attempt starts at the rising edge of *debounce_enable* (marked with e). It might trigger a request at j . Thus, a request is emitted not earlier than DB cycles (strictly) after the rising clock edge. In case *ext_sel* changes, the active attempt is terminated, and a new attempt started.

⁵It needs to change again to start a new debounce attempt.

4. Design Specifications

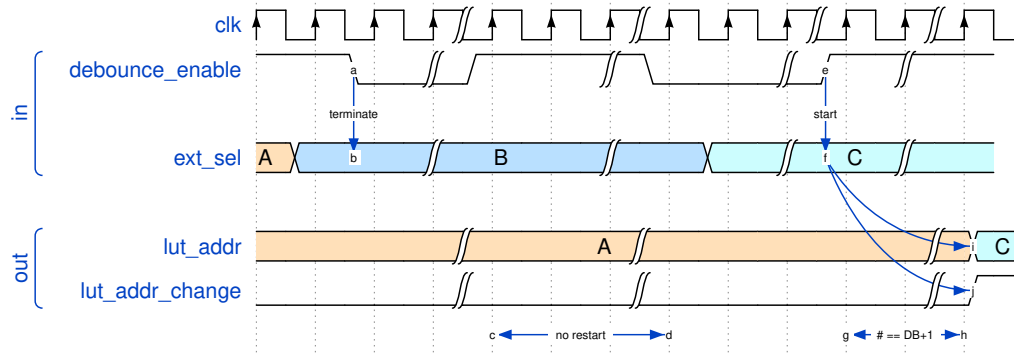


Figure 4.9.: *Debounce* module: behavior on disable.

One input has been neglected up to now: mode_change. The behavior is shown in figure 4.10. It is used to re-trigger a request on a stable ext_sel vector. On the rising edge of mode_change, the output lut_addr gets invalid (a → b). Thus, the input appears as if it has changed (regarding to the lut_addr). A new attempt is started like on a changed value of ext_sel. The time is started with the falling edge of mode_change as shown in k → l → m. mode_change shall not be high for more than one clock cycle in a row.

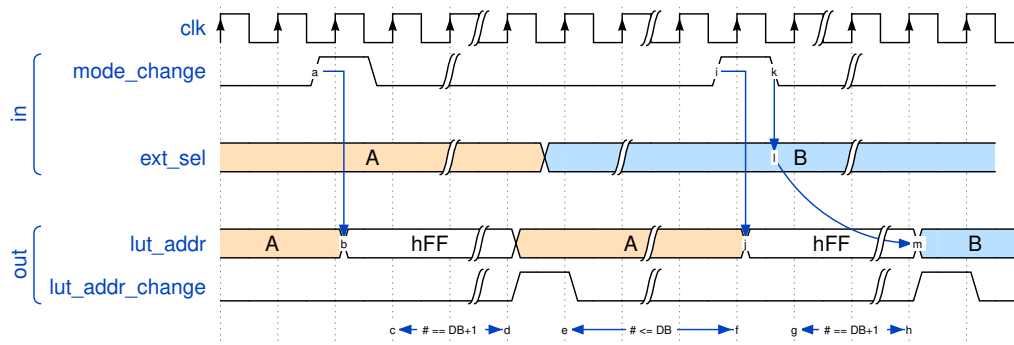


Figure 4.10.: *Debounce* module: behavior of mode_change input.

5. Verification

In this chapter, properties to verify two SystemVerilog modules from the project described in chapter 4 are modeled. Section 5.2 addresses the *debounce* module and section 5.3 the top-level *core* module. Both modules are defined in section 4.3 and section 4.4. The latter requires valid output data from the *debounce* module to abstract the debounce delay. Deriving verification properties includes the formal description of safety-properties, assumptions on the environment, and cover points to avoid over-constraining. Moreover, this chapter documents efficiency considerations made while modeling the properties. An evaluation and scalability analysis are offered in chapter 6.

5.1. Verification Procedure

5.1.1. Bind Files

Formal properties derived from the specifications are kept in a separate file. The *checker module* is then bound to the DUT. Thus, the design is kept clean or includes just a few assertions written by the designer¹. The verification engineer can independently derive formal properties from the specification without being biased by existing properties. Usually, the checker module has the same ports as the DUT. However, all ports are *inputs* as the checker observes the inputs and outputs from the DUT and shall not actively drive any of the signals. By having access to these ports, black-box assertions can be created. In addition, internal nets can be made accessible to the bound checker module to express white-box assertions. Especially for more complex modules, assertions based on intermediate signals can simplify the resulting property and prevent code duplication. Moreover, designers can check their intent by observing internal signals.

Binding the checker to the target module is equivalent to instantiating the checker module right before `endmodule` of the target design [10]. However, the connection can be made by a separate *bind* file that can be easily replaced or omitted to loosen or tighten the applied checks. The bind directive is visualized in figure 5.1.

¹Also, the designer might use a separate checker file instead of mixing the design and verification statements.

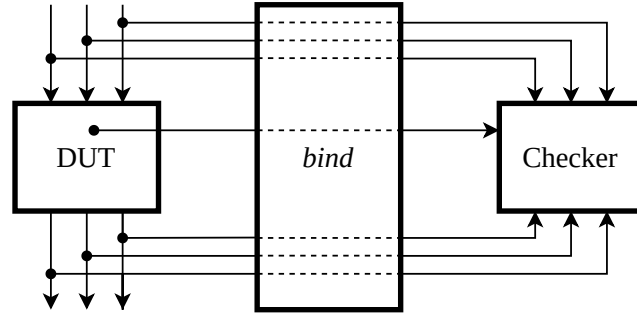


Figure 5.1.: Usage of the bind directive to connect the checker to the DUT.

5.1.2. Verification Objective

The applied FV shall be considered as an addition to traditional simulation-based verification and detect possible corner-case bugs. Furthermore, it shall lead to a deep understanding of the implementation. The formal environments of two modules are documented in the subsequent sections. As the *debounce* module is well suited to formal verification, most parts of its specifications are described formally – resulting in an almost full proof. For the *core* module, critical parts of the specifications are checked in a bug hunting approach. These checks abstract the debouncer as it is proven separately.

To give the reader a compact overview of the applied assertions, the documentation throughout this chapter linearly lists all applied SVA properties. Details on setting up a hierarchical Design Verification (DV) plan is out of the scope of this work. A good starting point for creating a complete formal DV plan is given in, e.g., [61]. Moreover, the verification flow includes many more steps of iteratively adding and refining cover points, assertions, and assumptions that are not explicitly documented. [16] refers to these iterations as *wiggling* which is a natural phase of the verification process.

5.1.3. Proposed Methodology

Figure 5.2 illustrates the proposed methodology which builds upon recommendations in literature (see section 3.2). It aims to give more guidance to the engineer by explicitly noting the options for debugging during the wiggling process. The step-by-step flow shall assist in overcoming some of the difficulties of FV.

First, a blank checker module is created and bound to the DUT. Assumptions that are explicitly mentioned in the specifications may be added at this point. All further assumptions are added iteratively when analyzing the CEX of failing assertions.

It is practical to start by writing cover points to inspect the design. Cover points describe the typical behavior of the system to see whether the DUT *can* fulfill the requirements. In other words, it is at least possible to cover the positive sequence specified by the property. If a cover point is not reachable, the specifications cannot even be satisfied. However, this does not necessarily point out a bug. Possible reasons include: (a) a bug in the cover property,

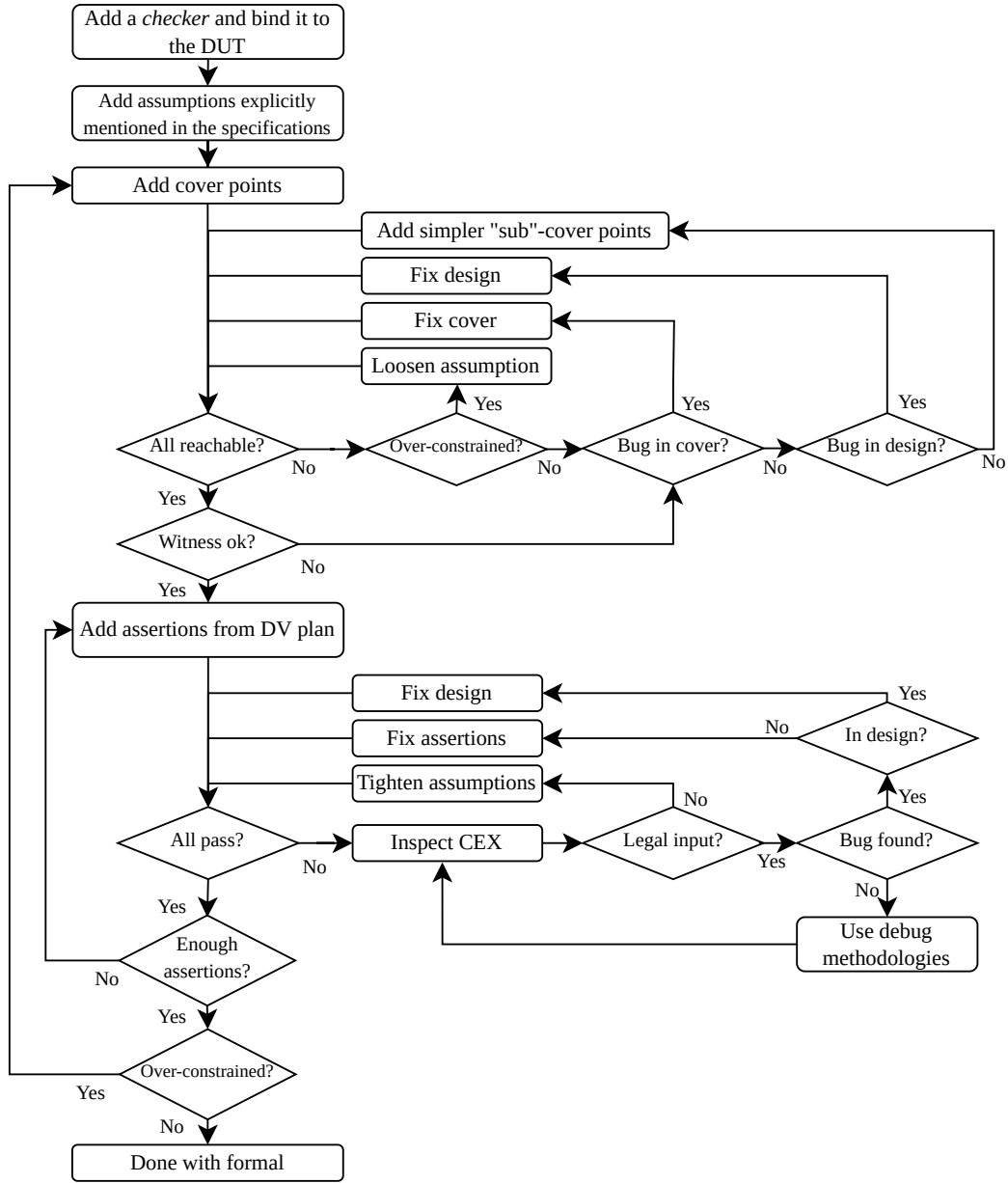


Figure 5.2.: Flowchart of the proposed verification methodology.

e.g., semantic errors or wrongly interpreted specifications, (b) the assumptions are too tight and over-constrain the system, or (c) there is a bug in the design itself. If the cause cannot be found, the cover property can be split up into a set of less complex cover properties to narrow down the root cause. Unlike proving assertions, there is no CEX that supports debugging.

When all cover properties are reachable, inspecting the witness may hint at bugs early in the verification process. E.g., some sequence matches after two cycles from reset but is only expected to match after ten cycles. In this case, there might be a weakness in the reset

state. As cover points describe *what* to reach, it might also be interesting to inspect *how* it is reached and take a look at whether the trace matches the expectation. Moreover, it might also be helpful to leverage cover properties and FV to develop directed test cases in dynamic verification. In this case, the formal verification flow can be terminated at that point.

Afterward, assertions are derived from the specifications or taken from the DV plan and added to the checker module. Most likely, assertions do not pass in the first run. The model checking algorithm comes up with a CEX that can be inspected to get an intuition about the failure. It might be caused by (a) too loose assumptions, (b) a bug in the design, or (c) a bug in the assertion property. If the assumptions are too loose, they can be tightened. As an empty model satisfies all assertions, it must be ensured not to exclude the behavior of the actual product. To prevent accidental over-constraining, a final step checks all cover properties from the beginning again. Missing assumptions or a bug might not be obvious and easily detectable. In this case, debugging methodologies as summarized in section 3.3 may be considered. However, it is common to have many iterations of refining assumptions and fixing bugs until the assertion passes.

Once all properties pass, the question of whether enough assertions have been added arises. There are different metrics and approaches to support this decision. Some of which are presented in section 3.6. Moreover, the question of completeness also depends on the desired depth of FV. While a full formal proof relies on the completeness, bug hunting may omit most design parts on purpose. Instead, an interesting subset of the functionality is proven.

Finally, the cover points must be re-examined to prevent over-constraining by tightened assumptions during the wiggling phase. Tools usually support this by generating implicit cover properties on assertion antecedents. However, the tool might be re-run on the hand-written cover points as well.

5.1.4. Flow

The proposed methodology from above is implemented based on OneSpin DV-Verify. However, a similar flow can also be implemented with other commercial formal verification tools. Figure 5.3 gives an insight into how the scripts and tools interact with each other. The engineer calls a *csh* (C Shell) script and passes a path to a file list as well as an indicator of whether to start in GUI or batch mode. The script then calls OneSpin with a Tcl script to be executed after startup. It handles loading the source files (DUT and checker) and applying all settings to prove the formal properties. Moreover, it can be specified whether to prove all formal properties or a selected subset.

Starting OneSpin in GUI mode is recommended during the wiggling phase as the graphical view has many features that simplify inspection and debugging as described in section 3.3. After inspecting the results, properties may be added or modifications may be applied before rerunning the tool, depending on the current progress. At a later stage, the additional *Quantify* process might be run to inspect the achieved coverage. If OneSpin is executed in batch mode, the results are written to the hard disk. Verification results may also be exported to a coverage database to be tracked by a third-party tool [85].

The DUT may also be loaded by a simulator to verify certain design aspects in dynamic verification. In the illustrated example, the same checker file is used. However, a different checker file may be loaded explicitly for simulation. Further details on simulation-based verification are beyond the scope of this work.

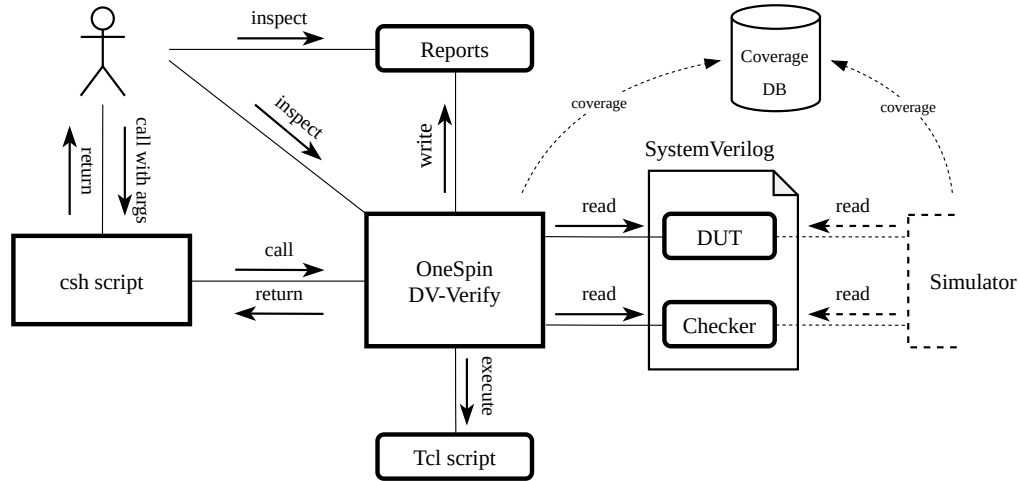


Figure 5.3.: Formal verification flow and tool interaction.

Figure 5.4 gives a more detailed insight into the sequential execution order of a basic formal verification run. In the illustrated example, the engineer calls the csh script with a file list and an indicator to start in GUI mode. The csh script sets the environment variables that the subsequent Tcl script can read. Afterward, OneSpin is called. A Tcl script is given as a parameter that starts executing after OneSpin has started. A minimal Tcl script sets some fundamental options and loads the specified files. Additional arguments might be given to override SystemVerilog *parameters* or influence conditional compilation. After elaborating and compiling the DUT, the tool can be instructed to prove a set of formal properties. Depending on the complexity, it might take a considerable amount of time to complete that process. The user might start to inspect the results – such as a CEX – right after the first check has finished. After all checks have been completed, a Quantify run may be started. Upon finishing the Tcl scripts, the program waits for the user to close the main windows or exit the program by a Tcl command before returning to the calling script.

5.1.5. General Considerations

If not mentioned otherwise, the default clocking and default disable from listing 5.1 are used throughout this chapter.

```

1  default clocking cb @(posedge clk);
2  endclocking
3
4  default disable iff (~rst_n);

```

Listing 5.1: Default clocking and default disable statements.

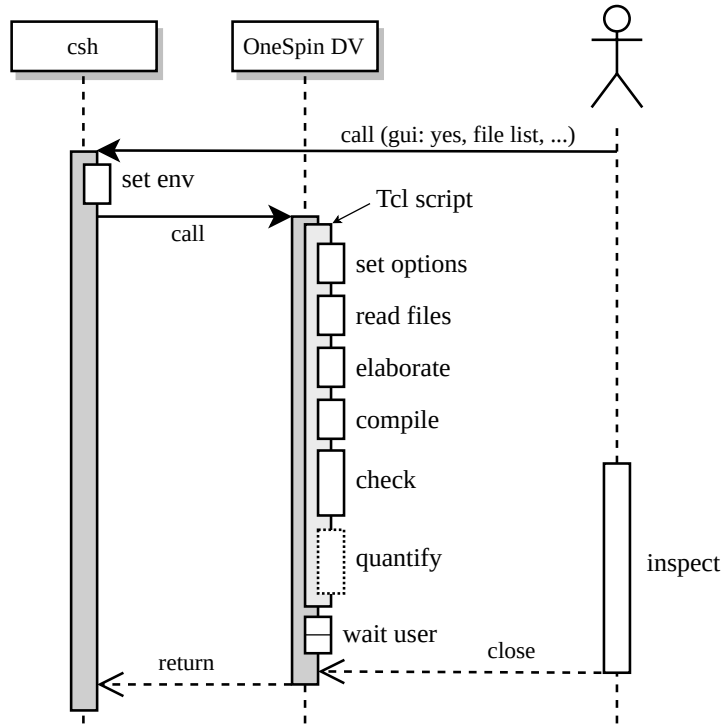


Figure 5.4.: Sequence diagram of a simple formal verification run.

5.2. Debounce

In this section, assertions to verify the *debounce* module are implemented. The specifications can be found in section 4.4.1. All checks are performed with six debounce cycles ($DB = 6$). Chapter 6 discusses formal efficiency and gives an insight into this choice.

5.2.1. Environment

The environment is constrained by two assumptions shown in listing 5.2. The first one ensures that FF_h is not considered to be used as a valid address. This reflects the specification and is required to obtain a correct system operation. The second assumption ensures that the *mode change* indicator is not high for more than one contiguous clock edge.

```

1 assume property (ext_sel != 'hff');
2 assume property (mode_change | => ~mode_change);

```

Listing 5.2: *Debounce* module: assumptions on the environment.

5.2.2. Cover Points

Before writing assertions, the reachability of the usual operation is checked to prevent over-constraining and vacuity. The first set of cover properties shown in listing 5.3 reflects some fundamental characteristics: E.g., a request can be triggered, or the output can take on some values. `cov_1` and `cov_2` start with an initial one-cycle delay (`##1`) as the succeeding sequence parts need to be one cycle away from reset. The functions `$rose` and `$fell` inspect the value of the past cycle to determine a signal change. Thus, it shall only be evaluated from the second cycle onwards.

After running the tool, it returns that all cover points can be reached. The comment after each cover property denotes the number of cycles from reset until the property is matched. All of them seem reasonable. The first request is triggered after $DB + 1$ cycles and rises from cycle 7 to 8. The property in the second line matches one cycle later, which is expected as a request shall have a width of one cycle. The cover properties `cov_3` and `cov_4` check whether different addresses can be observed at the `lut_addr` output, i.e., the debounced version of the applied `ext_sel` vector. As can be seen in figure 4.6 that signal is further routed to a LUT to obtain the internally used address. `cov_3` matches with the first raised request² as `h0h` is the initial value. `cov_4` takes two debounce attempts to have a match. The witness is shown in figure 5.5.

```

1 cov_1: cover property (##1 $rose(lut_addr_change));           // 7
2 cov_2: cover property (##1 $fell(lut_addr_change));           // 8
3 cov_3: cover property ((lut_addr == 'h00) ##[+] (lut_addr == 'h10)); // 8
4 cov_4: cover property ((lut_addr == 'h50) ##[+] (lut_addr == 'h00)); // 14

```

Listing 5.3: *Debounce* module basic cover properties.

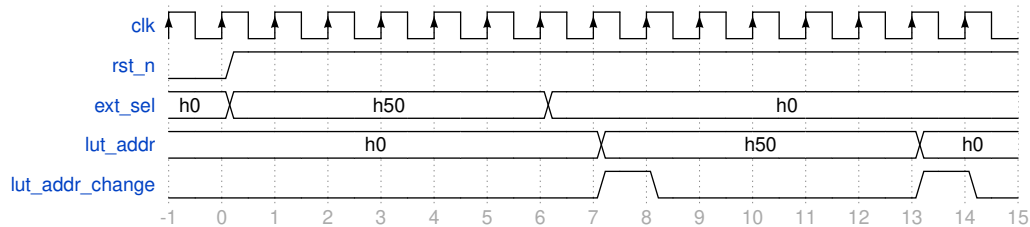


Figure 5.5.: Witness for cover property `cov_4` from listing 5.3.

Listings 5.4 - 5.8 depict more detailed cover points created after inspecting the cover properties shown above. `cov_int_raise` in listing 5.4 ensures that a request can be triggered and that the output address is updated after `ext_sel` has been stable for $DB+1$ cycles in normal operation. `debounce_enable` is high and `mode_change` low throughout the entire sequence. Moreover, `ext_sel` changes one cycle away from reset. After $DB+1$ cycles there must exist a state in which `lut_addr`, i.e., the internal address, changes and a request is

²The update can only be observed in the subsequent cycle because of value sampling (see section 2.4.4).

raised. This property would fail in an assertion as there is no expression part to hold `ext_sel` stable over the debounce time³.

```

1 cov_int_raise: cover property (
2   (debounce_enable & ~mode_change)
3   throughout ( ##1 $changed(ext_sel) ##(DEBOUNCE_CYCLES+1)
4               ($changed(lut_addr) & lut_addr_change) )
5 );

```

Listing 5.4: *Debounce* module: cover request raise.

Property `cov_termination` in listing 5.5 checks whether a debounce attempt can be terminated by disabling the module. `mode_change` is held low and `debounce_enable` is held high so as to not interfere with the intended check. Moreover, `ext_sel` shall always take a different value to the internal one to ensure a starting debounce attempt. The attempt is started by a change of `ext_sel` one cycle away from reset. One cycle before the attempt succeeds, the initial considerations on `mode_change` and `debounce_enable` are relaxed as the `throughout` block is left. The module is disabled by pulling down `debounce_enable`. `ext_sel` is still kept stable not to terminate the attempt by a changed address. No request shall be raised in the two subsequent cycles to check whether the termination happened.

```

1 cov_termination: cover property (
2   ( (~mode_change && debounce_enable
3     && $stable(lut_addr) && (lut_addr != ext_sel) )
4   throughout
5     (##1 $changed(ext_sel) ##1 $stable(ext_sel)[*(DEBOUNCE_CYCLES-1)]) )
6
7   ##1 ($stable(ext_sel) && ~debounce_enable && ~lut_addr_change))
8   ##1 ~lut_addr_change
9 );

```

Listing 5.5: *Debounce* module: cover attempt termination.

`cov_no_int` in listing 5.6 shows a trace where no request is emitted when the module is enabled and `ext_sel` is stable. `mode_change` is kept low and `ext_sel` is kept stable for the entire sequence. Then, the module is disabled one cycle away from reset (`debounce_enable` low). In the next `DB+2` cycles, the module is enabled (`debounce_enable` high). No request shall be raised (`lut_addr_change` low) and the internal address shall not be updated (`ext_sel` stable) over this timespan.

```

1 cov_no_int: cover property (
2   (~mode_change && $stable(ext_sel))
3   throughout
4   ( ##1 ~debounce_enable
5     ##1 ( debounce_enable &&
6           $stable(lut_addr) &&
7           ~lut_addr_change ) [*(DEBOUNCE_CYCLES+2)] )
8 );

```

Listing 5.6: *Debounce* module: cover sequence without request.

³In addition, this assertion would require some antecedent to not start an attempt at every leading clock edge.

In `cov_chg_disabled` (listing 5.7), a trace is found where `ext_sel` changes while the module is disabled. Moreover, `mode_change` is kept low throughout the sequence to indicate a stable mode. Then `debounce_enable` is pulled low. One cycle later `ext_sel` changes while the module is still disabled. In the next cycle the module is activated again and `ext_sel` is kept stable for `DB` cycles. As `ext_sel` changed while the module was disabled, a request shall be raised after the debounce time. The found witness is depicted in figure 5.6 and shows two interesting aspects. A request shall be emitted after `DB+1` cycles, when `ext_sel` has been stable for `DB` cycles. However, `cov_chg_disabled` checks `lut_addr_change` after `DB+2` cycles due to the sampling of values in concurrent assertions (see section 2.4.4). Thus, the sampled value at `DB+1` cycles is the *old* value. Another interesting finding is that `lut_addr_change` goes high at cycle 9 although the module is disabled in the same cycle (`debounce_enable` goes low). This is interesting and not precisely specified in requirements.

```

1 cov_chg_disabled: cover property (
2   ~mode_change throughout (
3     ~debounce_enable ##1 ~debounce_enable && $changed(ext_sel)
4     ##1 (debounce_enable && $stable(ext_sel)) [*](DEBOUNCE_CYCLES) ]
5     ##2 lut_addr_change )
6 );

```

Listing 5.7: *Debounce* module: cover an address change while the module is disabled.

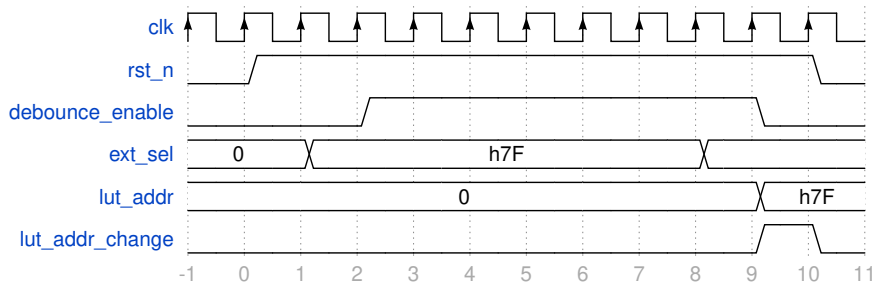


Figure 5.6.: Witness for cover property `cov_chg_disabled` from listing 5.7.

Finally, `cov_mode_chg` in listing 5.8 inspects a changing operating mode. After `mode_change` was high for one clock cycle, a new debounce attempt is started, which succeeds after `DB+1` cycles after the rising edge of `mode_change`. `debounce_enable` is kept low and `ext_sel` is kept stable over the entire sequence so as to not interfere with the aspect to be shown.

```

1 cov_mode_chg: cover property (
2   ($stable(ext_sel) && debounce_enable)
3   throughout
4   ( (~mode_change)[*2] ##1 mode_change
5     ##1 (~mode_change)[*DEBOUNCE_CYCLES] ##1 lut_addr_change )
6 );

```

Listing 5.8: *Debounce* module: cover a mode change.

5.2.3. Assertions

This section lists the modeled safety properties along with the design objective and a brief description.

Reset Behavior

asrt_rst_state When the module is in reset, `lut_addr_change` and `lut_addr` are zero.

```
1 asrt_rst_state: assert property (disable iff (rst_n)
2   ~lut_addr_change && lut_addr == 'bo
3 );
```

Listing 5.9: *Debounce* module: prove reset state.

asrt_rst_release When `rst_n` rises, no request is emitted within the first *DB* cycles.

As the default `disable` disarms the assertions, it is overwritten by `default disable (0)` to always be active.

```
1 asrt_rst_release: assert property (disable iff (0)
2   $rose(rst_n) |-> ~lut_addr_change[*DEBOUNCE.CYCLES]
3 );
```

Listing 5.10: *Debounce* module: release from reset.

Request Length and Data

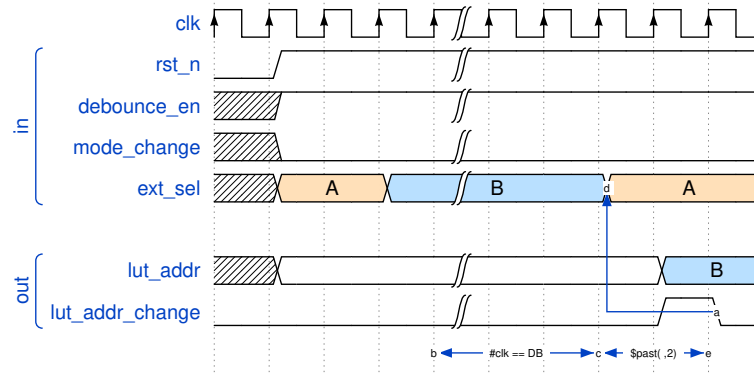
asrt_req_len The request pulse (`lut_addr_change`) shall be high no longer than one clock cycle.

```
1 asrt_req_len: assert property (lut_addr_change |=> ~lut_addr_change);
```

Listing 5.11: *Debounce* module: pulse length of a request.

asrt_req_data_if When there is no mode change (`mode_change` is low), `lut_addr` shall get `ext_sel` at each request.

When `mode_change` does not cause a re-trigger, the output address shall be updated if and only if a request is raised. Assertion `asrt_req_data_if` (listing 5.12) describes the *If* part and checks whether `lut_addr` is correctly updated on a request (`lut_addr_change`). Figure 5.7 visualizes the intent of the assertion. If a request is raised, the *high* request signal is sampled at the next rising clock edge (marked with a). From that instant in time, a debounce attempt succeeded two cycles in the past: one cycle from the succeeded attempt to the raised request; another to the sampled request. Thus, the **\$past** function uses its optional parameter to get the value of two cycles before ($a \rightarrow b$).

Figure 5.7.: *Debounce* module: visualization of assertion `asrt_req_data_if`.

```

1 asrt_req_data_if: assert property (
2   ~mode_change ##1 $rose(lut_addr_change)
3   |-> lut_addr == $past(ext_sel, 2)
4 );

```

Listing 5.12: *Debounce* module: update output address on a request.

asrt_req_data_only_if When there is no mode change (`mode_change` is low), `lut_addr` shall be stable if no request (`lut_addr_change`) is raised.

Assertion `asrt_req_data_only_if` completes the proof by checking the *Only If* part. If both checks succeed, subsequent properties only need to consider the correct level of the request signal and can safely neglect the address values. This simplifies the written property and the complexity the tool faces as the data path no longer needs to be considered. The signals `mode_change` and `lut_addr_change` in the antecedent are observed in a different cycle. `mode_change` is a primary input, and the action is taken at the subsequent clock cycle. However, the output `lut_addr_change` is raised at the same clock edge the output data is updated.

```

1 asrt_req_data_only_if: assert property (
2   (~mode_change ##1 ~lut_addr_change)
3   |->
4   $stable(lut_addr)
5 );

```

Listing 5.13: *Debounce* module: output address stable without a request.

Request Occurrence

There are multiple possible ways to prove the correct occurrence of a request (`lut_addr_change`). In general, the *If* and the *Only If* directions need to be proven to ensure that a request is raised when anticipated but at no other instant of time. The following considerations assume that the module is enabled and released from reset.

If If `ext_sel` changes or `mode_change` is high and afterwards (a) `ext_sel` is stable for enough cycles, and (b) `mode_change` is lower over the debonuce attempt, a request shall be raised. No request shall be raised, however, if the output address is equal to the applied one.

Only If If a request is raised, **\$past**(`ext_sel`) shall have been stable for enough cylces after it had changed or **\$past**(`ext_sel`) shall have been stable for enough cylces after `mode_change` had been high. During the debounce attempt, `mode_change` shall be low.

As pointed out in section 3.7, properties may be decomposed into smaller fragments to better cope with complexity. Subsequently, the *If* and *Only If* parts can be expressed by the conjunction of two or three shorter assertions, respectively.

If (a) If `ext_sel` changes and then `ext_sel` is stable for enough cycles, a request shall be raised. However, no request shall be raised if the output address is equal to the applied one.

If (b) If `mode_change` is high (and then low over the whole debounce attempt) and then `ext_sel` is stable for enough cycles, a request shall be raised.

Only If (a) If a request is raised, **\$past**(`ext_sel`) shall have been stable for long enough.

Only If (b) If a request is raised, `ext_sel` changed, or `mode_change` shall have been high before the debounce attempt started.

Only If (c) If a request is raised, `mode_change` shall have been low during the debounce attempt.

In the following, different approaches to implementing the proof are discussed conceptually. There might be some additional logic required to precisely reflect the implementation's behavior.

Figure 5.8 depicts the concept of using a shift register to keep track of the system's state over the debounce attempt. Each time `ext_sel` changes, or a request is triggered by a `mode_change` pulse, a *one* is shifted into the shift register. Otherwise, a *zero* is appended at the rising clock edge. The logic checks whether the MSB of the shift register is high whereas all other bits are zero. In this case, a request shall be raised in the subsequent cycle. Shift registers are a common technique to argue over longer sequential depths efficiently. However, a potential drawback of this logic is its similarity to the implementation of the design. By having access to the DUT's code, an error might accidentally be carried over to the verification code.

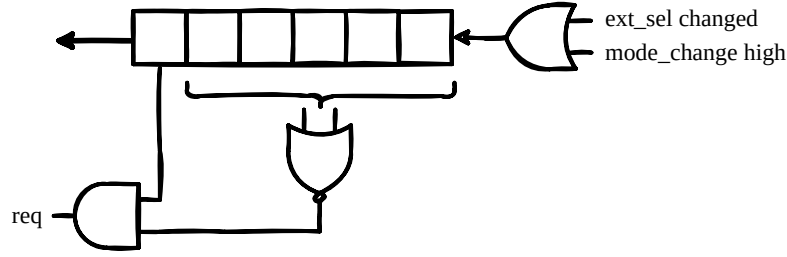


Figure 5.8.: *Debounce* module: concept to verify the request based on a shift register.

Another approach uses a counter, as shown in figure 5.9. If `ext_sel` has been stable and `mode_change` is low, the counter is incremented. The counter needs to be limited by some value larger than *DB* to avoid an overflow. If the *and* gate outputs a zero, the counter is reset. By comparing the counter value against the debounce duration, one can check the request signal in the *If* and *Only If* direction.

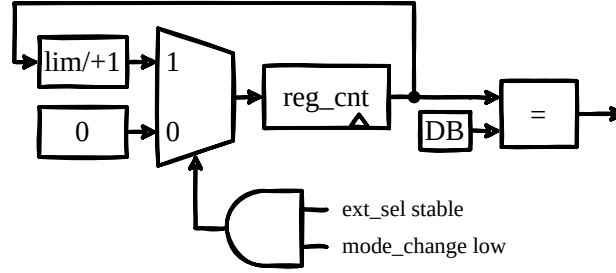


Figure 5.9.: *Debounce* module: concept to verify the request based on a counter.

In a bug hunting approach, it highly depends on the application to decide which parts of the specifications shall be proven with formal verification. The chosen depth also influences the effort put into generating the model. The implemented assertions used in this scope combine different techniques such as pure SVA and auxiliary code modeling a counter. Thus, the resulting complexity and runtimes can be compared. Moreover, thinking about the problem in different ways might support detecting possible weaknesses in the design and spot verification gaps.

asrt_req_ok_if_a If the mode is stable (`mode_change` is low) and if the module is enabled (`debounce_enable` high), a request shall be raised if the address (`ext_sel`) is different to the current one (`lut_addr`) and `ext_sel` is stable for *DB* cycles.

This builds the *If (a)* part of the request-correctness proof. Once the precondition is satisfied, a request shall be raised. In `asrt_req_ok_if_a` (listing 5.14), the module is assumed to be enabled (`debounce_enable` high) and in a fixed mode (`mode_change` low). If that holds, `ext_sel` changes and is then stable for *DB* clock cycles, then `lut_addr_change` shall be high in the following cycle. Additionally, it is checked that `ext_sel` is different from the current internal address in the antecedent. The property should be provable in a reasonable time but is expected to be inefficient for a longer debounce duration due to many⁴ repetitions.

⁴but finite

```

1  asrt_req_ok_if_a: assert property (
2    (debounce_enable & ~mode_change)
3    throughout
4    (##1 $changed(ext_sel) ##1 $stable(ext_sel)[*DEBOUNCE_CYCLES]
5    ##0 (ext_sel != lut_addr))
6    | => lut_addr_change
7  );

```

Listing 5.14: *Debounce* module: request *If (a)* part.

asrt_req_ok_if_b If the module is enabled (`debounce_enable` high), a request shall be raised if `mode_change` was high for one cycle and the address (`ext_sel`) is stable for *DB* cycles afterwards.

Assertion `asrt_req_ok_if_b` (listing 5.15) can be implemented similarly to assertion `asrt_req_ok_if_a` from above. A different assertion below checks the correct invalidation of the output address when `mode_change` is high. Thus, all legal input addresses at `ext_sel` are different to the output address at `lut_addr`.

```

1  asrt_req_ok_if_b: assert property (
2    ( (debounce_enable && $stable(ext_sel))
3    throughout
4    (##1 mode_change ##1 (~mode_change)[*DEBOUNCE_CYCLES]) )
5    | => lut_addr_change
6  );

```

Listing 5.15: *Debounce* module: request *If (b)* part.

asrt_req_only_if_ac A request shall only occur if `ext_sel` is stable for long enough and `mode_change` was low during that interval.

The property makes use of auxiliary code (listing 5.16) and combines the parts *Only If (a)* and *Only If (c)*. The implemented counter accumulates the cycles since the latest change of `ext_sel` or high `mode_change`.

```

1  bit [7:0] sva_ext_sel;
2  bit [$clog2(DEBOUNCE_CYCLES+1)-1:0] sva_stable_cnt;
3
4  always_ff @(posedge clk or negedge rst_n) begin
5    if (~rst_n) begin
6      sva_ext_sel    <= 'bo;
7      sva_stable_cnt <= 'bo;
8    end else begin
9      sva_ext_sel    <= ext_sel;
10     if ((ext_sel == sva_ext_sel) && ~mode_change) begin
11       if (sva_stable_cnt == DEBOUNCE_CYCLES) begin
12         sva_stable_cnt <= sva_stable_cnt;
13       end else begin
14         sva_stable_cnt <= sva_stable_cnt + 'b1;
15       end
16     end else begin
17       sva_stable_cnt <= 'bo;
18     end
19   end
20 end

```



```

21
22 asrt_req_only_if_ac: assert property (
23   lut_addr_change |-> $past(sva_stable_cnt) >= (DEBOUNCE_CYCLES-1)
24 );

```

Listing 5.16: *Debounce* module: prove debounce delay with auxiliary code.

Assertion `asrt_req_only_if_ac` compares the counter value with the debounce duration to check whether the debounce attempt indeed was successful. The `$past` function is called as the request is raised one cycle after the debounce time is over and `ext_sel` might change at that cycle without affecting the request in the next cycle (see figure 4.7). As the debounce time is respected, the first raised request must be valid.

Figure 5.10 illustrates the evidence gained from `asrt_req_only_if_ac`. As it proves that `ext_sel` has been stable for long enough and `mode_change` was low during that period, the *first* request cannot violate the specifications if the attempt was indeed started. However, it might be illegally raised without a trigger condition, i.e., `ext_sel` has never changed since reset and `mode_change` has never been high since reset. For the given application, no requirement is violated by an erroneously raised first request even when `ext_sel` did not change and is equal to the output value. For any other input address, a debounce attempt is started after the reset is released anyhow. Follow-up requests, however, must be caused by a valid trigger condition.

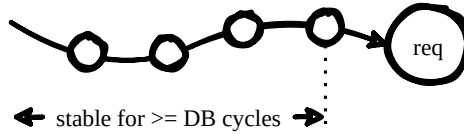


Figure 5.10.: *Debounce* module proof step: request did not violate the debounce attempt.

Assertion `asrt_req_only_if_ac` does not detect whether multiple requests without precondition are raised. E.g., there could be another request raised three cycles after the first request without any change of `ext_sel` or `mode_change`. Figure 5.11 depicts how legal follow-up requests may be reached. After either `ext_sel` changed or `mode_change` was high, the trajectory moves through the state space until there is a state after $DB+1$ cycles that raises the request.

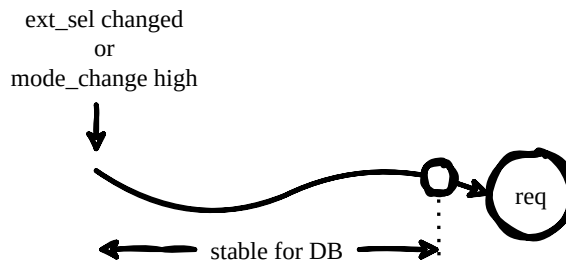


Figure 5.11.: *Debounce* module proof step: possible raise of follow-up requests.

There are different ways to implement the missing *Only If (b)* part. One possibility would be to use the auxiliary code from listing 5.16 and increase the upper counter bound by one cycle to indicate an outdated attempt. Thus, no two requests can be triggered by the same – not re-triggered – debounce attempt. However, there is another neat way to express the behavior in pure SVA. Figure 5.12 illustrates the idea. As the first request is legal⁵, all subsequent requests are also legal if either `ext_sel` changed, or `mode_change` was high.

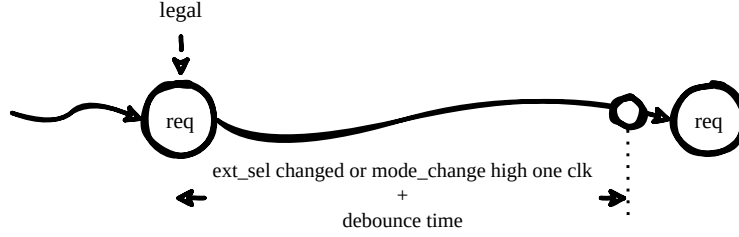


Figure 5.12.: *Debounce* module proof step: correct raise of follow-up requests.

The idea may be rephrased into: There is no sequence of two requests without a change of `ext_sel` or `mode_change` high in-between as shown in figure 5.13.

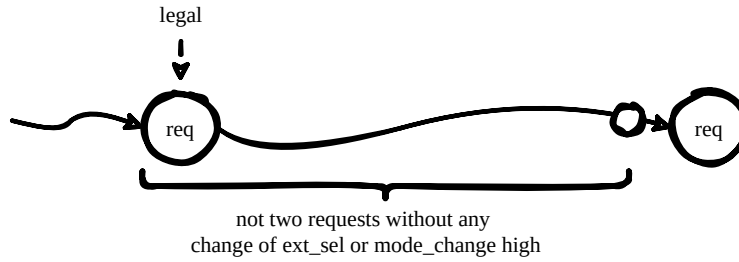


Figure 5.13.: *Debounce* module proof step: correct raise of follow-up requests by considering a sequence of two requests.

asrt_req_once Within two occurrences of `lut_addr_change`, it is not possible that `$past(mode_change)` was continuously low and `$past(ext_sel)` did not change value.

Assertion `asrt_req_once` elegantly expresses the considerations from above. This assertion is named differently from the other ones, as it does not fully cover *Only If (b)*. As discussed, the first attempt might have started without a trigger condition.

The *goto* operator is syntactic sugar and can be expressed by an equivalent alternative. In particular `lut_addr_change[->n]` is equivalent to `(!lut_addr_change[*] ##1 lut_addr_change)[*n]`. As infinite repetitions are considered efficient in FV [18] and `n` is small in the implemented statement, the overall complexity is assumed to be lower than exploring the entire history over the debounce attempt.

⁵Although it might not be legally triggered, it cannot endanger the system's operation.

```

1 asrt_req_once: assert property (
2   not( $past((~mode_change && $stable(ext_sel))) )
3   throughout lut_addr_change[->2]
4 );

```

Listing 5.17: *Debounce* module: not a second request occurs without precondition.

Disable and Mode-Change

asrt_en_disabled When the module is disabled (`debounce_enable` low), no request shall be raised, i.e., `lut_addr_change` is low.

There are different views on verifying this statement caused by an ambiguous verbal specification. One could express the statement as an invariant of the form (`debounce_enable` || !`lut_addr_change`) as there must not be a request while `debounce_enable` is low. Another intuitive representation would be an implication expressing that no request is followed by a disabled module: !`debounce_enable` |-> !`lut_addr_change`. However, the specification also allows an alternative interpretation that no active attempt is allowed to succeed after disabling the module⁶. This was the actual design intent and expressed by a non-overlapping implication as the request is raised with a delay of one cycle after the attempt succeeded. Property `asrt_en_disabled` in listing 5.18 shows the formal statement from above.

```

1 asrt_en_disabled: assert property (~debounce_enable |=> ~lut_addr_change);

```

Listing 5.18: *Debounce* module: no request while module is disabled.

asrt_en_init_dly After activating the module, no request shall be released within *DB* cycles.

By proving property `asrt_en_init_dly` it is ensured that after activation, the debounce time is restarted even if `ext_sel` changed some time ago in the inactive phase.

```

1 asrt_en_init_dly: assert property (##1 $rose(debounce_enable)
2   |=> (~lut_addr_change) [*DEBOUNCE_CYCLES]
3 );

```

Listing 5.19: *Debounce* module: delay after enabling.

asrt_en_terminate Disabling the module shall terminate all attempts.

The property ensures that after `debounce_enable` falls, there is no request (even if `debounce_enable` rises again) until `ext_sel` changes or `mode_change` indicated a changed operating mode. However, the property could be safely neglected as it is redundant. `asrt_en_disabled` already proves that no request can be raised in a reset phase. Moreover, `asrt_en_init_dly` checks that *DEBOUNCE_CYCLES* is respected after activating the module. Thus, there is no way of not terminating an attempt on deactivation. If the inactive

⁶As a request is raised one cycle after the attempt succeeded, there can be a request while the module is disabled.

phase is long, then the potential request would occur while `debounce_enable` is low and caught by `asrt_en_disabled`. If the inactive phase is short, then the potential request would be raised when the module is active again. In this case, however, the debounce time since the rise of `debounce_enable` would not be respected. `asrt_en_init_dly` would detect that.

```

1  asrt_en_terminate: assert property (
2      ##1 $stable(ext_sel) && $fell(debounce_enable)
3      |=>
4      ~lut_addr_change until ($changed(ext_sel) or mode_change)
5  );

```

Listing 5.20: *Debounce* module: terminate attempt when disabling.

asrt_mode_invalidate After a mode change, the output address shall be invalid (all ones).

The property checks if a `mode_change` happens at least one cycle away from reset. If this happens, then `lut_addr` shall be all ones beginning from the next cycle.

```

1  asrt_mode_invalidate: assert property (
2      (##1 mode_change) |=> (lut_addr == '1)
3  );

```

Listing 5.21: *Debounce* module: invalidate address on a mode change.

5.3. Core

In this section, all formal properties to verify the *core* module, i.e., the top-level module without synchronizers, are derived from the requirements and implemented. Its specification is given throughout section 4.3.

5.3.1. Environment

As with the *debounce* module in section 5.2, assumptions on the environment are created in an iterative approach. Coming up with an adequate model to constrain and refine the assumptions is usually spanned across the entire wiggling phase.

The applied assumptions are denoted in listing 5.22. `assu_addr` constrains the set of valid addresses as mentioned in the specification. `assu_m2_rpi_a` and `assu_m2_rpi_b` ensure that the RPi does not start an SPI transfer while mode *M2* is active, as it might be interrupted by the FPGA's SPI master.

Changing between different operating modes is considered next. Modes *M0* and *M1* can be entered or left at any time as there is no state to remember for the FPGA. Subsequently, mode *M2* can be entered from either mode *M0* or mode *M1* at any time. When mode *M2* is active, the FPGA might start an SPI transfer to the drivers that spans over multiple clock cycles. Changing the externally applied mode vector to indicate a mode change request will not lead to an update of the internal mode register until the SPI transfer has been completed. All mode-dependent assumptions could take the internal mode

register as a reference (white-box) or reference to an adequate helper model that reflects the same behavior as the actual implementation (black-box). The latter is done here by a straightforward model. `sva_active_mode` mimics the internal mode register and takes over any mode request at the next rising clock edge. As outlined above, this is not true when a transfer is in-flight in mode *M2*. Thus, `assu_mode_chg` is added to consider this case. It does not allow a mode change when mode *M2* is selected and the FPGA master is active – indicated by `sva_master_not_idle`. A white-box signal is used for simplicity to indicate if the master is active. However, it alternatively may be modeled as (a) *some time after the latest ext_sel change* or (b) a more detailed shadow model of the main FSM.

Mode *M3* is beyond the proofs considered within this scope. As the RPi is the SPI master, it shall not change the mode within an ongoing transfer. The FPGA's SPI slave is selected by `cs_n_0_from_rpi`. Assumption `assu_m3_rpi_cs` expresses that the other chip-select (`cs_n_1_from_rpi`) shall not be activate in this mode.

```

1  sequence seq_mode_act(m);
2    (sva_active_mode == m);
3  endsequence
4
5  assu_addr: assume property (ext_sel != 'hff);
6
7  bit [2:0] sva_active_mode;
8  always_ff @(posedge clk or negedge rst_n) begin
9    if (~rst_n) begin
10      sva_active_mode <= 'bo;
11    end else begin
12      sva_active_mode <= mode;
13    end
14  end
15
16  assu_m2_rpi_a: assume property (
17    seq_mode_act(M2_EXT_ROW_SEL) |-> {cs_n_0_from_rpi, cs_n_1_from_rpi} == 'b1);
18
19  assu_m2_rpi_b: assume property (
20    seq_mode_act(M2_EXT_ROW_SEL) |-> {mosi_from_rpi, sclk_from_rpi} == 'boo);
21
22  bit sva_master_not_idle;
23  assign sva_master_not_idle = (oslo_fpga_core.i_oslo_fpga_ctrl.state !=
24    oslo_fpga_core.i_oslo_fpga_ctrl.IDLE);
25
26  assu_mode_chg: assume property (
27    ##1 sva_master_not_idle && $past(mode == M2_EXT_ROW_SEL) |-> $stable(mode));
28
29  assu_m3_rpi_cs: assume property (mode == M3_SER_CTRL |-> cs_n_1_from_rpi);

```

Listing 5.22: Core module: assumptions on the environment.

5.3.2. Cover Points

The cover points shown in listing 5.23 include some simplistic covers such as to select each mode and some more advanced properties to trigger a request to each of the drivers (cov_m2_frm_driver0 and cov_m2_frm_driver1). More properties might be added to inspect the design and detect critical misbehavior early in the verification flow.

```

1  cov_m0:  cover property (seq_mode.act(M0_ROUTE_THROUGH));
2  cov_m1:  cover property (seq_mode.act(M1_EX_STREAM_TRG));
3  cov_m2:  cover property (seq_mode.act(M2_EXT_ROW_SEL));
4  cov_m3:  cover property (seq_mode.act(M3_SER_CTRL));
5
6  cov_m2_frm_driver0: cover property (
7    (seq_mode.act(M2_EXT_ROW_SEL) ##1 (mode == M2_EXT_ROW_SEL)) throughout
8    (##[+] $fell(cs_n_o_from_fpga) ##[+] $rose(cs_n_o_from_fpga))
9  );
10
11 cov_m2_frm_driver1: cover property (
12   (seq_mode.act(M2_EXT_ROW_SEL) ##1 (mode == M2_EXT_ROW_SEL)) throughout
13   (##[+] $fell(cs_n_1_from_fpga) ##[+] $rose(cs_n_1_from_fpga))
14 );

```

Listing 5.23: Core module: cover properties.

5.3.3. Assertions

Modes M0_ROUTE_THROUGH and M1_EX_STREAM_TRG

The safety properties for modes *M0* and *M1* are straightforward and only check the SPI forwarding and some combinatorial logic on the control signals. They can be found in listing A.1 in Appendix A.

Mode M2_EXT_ROW_SEL

The modeled safety properties are described below. All references to the specifications assume that mode *M2* is active.

asrt_m2_mon_x mon_x_from_fpga shall be the same as the swapped instances of lshot_en_x_from_fpga.

```

1  asrt_m2_mon_0: assert property (
2    seq_mode.act(M2_EXT_ROW_SEL) |-> (mon_0_from_fpga == lshot_en_1_from_fpga)
3  );
4
5  asrt_m2_mon_1: assert property (
6    seq_mode.act(M2_EXT_ROW_SEL) |-> (mon_1_from_fpga == lshot_en_0_from_fpga)
7  );

```

Listing 5.24: Core module: mon_x_from_fpga in mode *M2*.

asrt_m2_eues eues_in_0_from_fpga shall be zero.

```

1  asrt_m2_eues: assert property (
2      seq_mode_act(M2_EXT_ROW_SEL) |-> (eues_in_o_from_fpga == 'bo)
3  );

```

Listing 5.25: Core module: eues in mode M2.

asrt_m2_lshot_act_x lshot_en_x_from_fpga shall be zero for the inactive driver.

One of the drivers covers each even row, the other driver each odd one. Thus, sva_active_driver_id is assigned to the address' LSB.

```

1  bit sva_active_driver_id;
2  assign sva_active_driver_id = lut_addr[o];
3
4  asrt_m2_lshot_act_o: assert property (
5      seq_mode_act(M2_EXT_ROW_SEL) ##o (sva_active_driver_id == 'bo)
6      |-> ~lshot_en_1_from_fpga
7  );
8
9  asrt_m2_lshot_act_1: assert property (
10     seq_mode_act(M2_EXT_ROW_SEL) ##o (sva_active_driver_id == 'b1)
11     |-> ~lshot_en_o_from_fpga
12 );

```

Listing 5.26: Core module: lshot_en_x_from_fpga for inactive driver in mode M2.

asrt_m2_lshot_inact_mstr During a transfer, both lshot_en_x_from_fpga signals shall be low.

```

1  asrt_m2_lshot_inact_mstr: assert property (
2      seq_mode_act(M2_EXT_ROW_SEL) ##o sva_fpga_trans_act
3      |->
4      ({lshot_en_1_from_fpga, lshot_en_o_from_fpga} == 'bo)
5  );

```

Listing 5.27: Core module: lshot_en_x_from_fpga during SPI transfer in mode M2.

asrt_m2_lshot_act_mask_x For the active driver lshot_en_x_from_fpga shall be lshot_en_x_from_rpi except for when the FPGA communicates with the drivers. Concretely, from state OFF to state ON⁷ both lshot_en_x_from_fpga shall be low.

```

1  asrt_m2_lshot_act_mask_o: assert property (
2      seq_mode_act(M2_EXT_ROW_SEL)
3      ##o ((sva_active_driver_id == 'bo) && sva_lshot_mask)
4      |-> lshot_en_o_from_fpga == lshot_en_o_from_rpi
5  );
6  asrt_m2_lshot_act_mask_1: assert property (
7      seq_mode_act(M2_EXT_ROW_SEL)
8      ##o ((sva_active_driver_id == 'b1) && sva_lshot_mask)
9      |-> lshot_en_1_from_fpga == lshot_en_1_from_rpi
10 );

```

Listing 5.28: Core module: lshot mask in mode M2.

⁷See figure 4.5.

Figure 4.5 depicts the timespan in which `int_mask_lshot_en` is low and thus `lshot_en_x_from_fpga` is low. Listing A.2 in Appendix A shows how `sva_lshot_mask` is modeled. Moreover, `sva_lshot_mask` is compared with the internal signal `int_lshot_en_mask` to check the model. `asrt_m2_lshot_act_mask.x` uses the modeled mask to prove that the active driver's lshot is correctly enabled.

asrt_spi_frm_status The *status* byte of a frame shall be all zeros except for the driver ID in bit zero.

Listing A.3 in Appendix A models a simplistic SPI receiver based on a shift register. The shift register has the length of a frame and uses a packed structure to access the frame parts. Thus, the individual section can be checked at each finished transmission. Although the complexity and runtime of data path checks are expected to be high, the computation time is shown to be feasible for the given design. The frame's content is checked right at the end of a transmission (`sva_fpga_trans_act` falls) before any data in the shift register is lost. The corresponding frame part can be selected in the structure by using the dot-operator as used throughout listings 5.29 to 5.31

```

1  asrt_spi_frm_status: assert property (
2      seq_mode_act(M2_EXT_ROW_SEL) ##1 $fell(sva_fpga_trans_act)
3      |->
4      ($past(sva_spi_frame.status) == {7'bo, sva_active_driver_id})
5  );

```

Listing 5.29: Core module: SPI frame status byte.

asrt_spi_frm_pad All padding bytes in a frame shall be zero.

```

1  asrt_spi_frm_pad: assert property (
2      seq_mode_act(M2_EXT_ROW_SEL) ##1 $fell(sva_fpga_trans_act)
3      |-> ($past(sva_spi_frame.padding) == 'bo)
4  );

```

Listing 5.30: Core module: SPI frame padding.

asrt_spi_frm_cmd The frame's command byte shall be set correctly.

```

1  asrt_spi_frm_cmd: assert property (
2      seq_mode_act(M2_EXT_ROW_SEL) ##1 $fell(sva_fpga_trans_act)
3      |-> ($past(sva_spi_frame.cmd) == CMD_SEL_ROW)
4  );

```

Listing 5.31: Core module: SPI frame command.

Similarly, all other data of a frame can be proven. For computationally expensive parts, such as the CRC byte, dynamic verification against a reference implementation is more suitable.

6. Evaluation

In this chapter, the modeled safety properties from chapter 5 are evaluated.

First, section 6.1 analyzes complexity considerations based on an open-source debouncer module similar to the *debounce* module of the Lidar evaluation kit. The evaluation is performed with the formal tool by YosysHQ. Different approaches such as auxiliary code instead of pure SVA are compared and interpreted for the given implementation.

Afterward, section 6.2 inspects the complexity and runtime of the formal proofs applied to the implementation of the digital design from chapter 5. All checks related to this project have been proven on a 12-core Intel Xeon Gold 6136 CPU with 32 GB RAM using OneSpin 360 (R), version 2021.1.2 - build May 6, 2021.

Finally, section 6.3 evaluates the achieved coverage and discusses potential gaps in the verification.

6.1. Runtime Considerations

This section investigates some of the considerations taken to cope with complexity. All comparisons are made on an open-source debouncer implementation shown in listing B.1 in appendix B. Although the design is straightforward and smaller than the required verification code, it features the same core functionality as the one from the Lidar evaluation kit. Thus, it is ideally suited for evaluating of some of the applied assertions on an open and reproducible design.

For the same reason, software around the open-source Yosys framework by YosysHQ¹ has been selected for the formal proofs. The tool is run on an Intel Core i5-8350U CPU @ 1.70GHz with 8 GB of RAM. The proof is based on k-induction with minimal depth and default options unless mentioned otherwise. Yices 2 and Z3 are used as solvers whereas the faster completing one is taken for the plots. However, the comparison aims at conveying a feeling for the complexity variations obtained from different verification choices, and it does not want to compare different solvers.

¹<https://www.yosyshq.com/>

6.1.1. Auxiliary Code

To prove the assertion `asrt_req_only_if_ac`, auxiliary code is used as depicted in listing 5.16. The assertion is triggered when a request is raised (`lut_addr_change` high). Then, it argues over the history of `ext_sel` by comparing to a counter value. Thus, it can be determined whether `ext_sel` has been stable for enough cycles. The auxiliary code increments the counter (`sva_stable_cnt`) if `ext_sel` was stable during the last cycle. Otherwise the counter is reset to zero. Subsequently, the assertion only needs to compare the counter value at each raised request with a constant². If the counter value is larger than or equal to $DB-1$, the request is valid.

The same behavior can be checked with pure SVA instead of auxiliary code. Assertion `asrt_req_only_if_ac` associated with the used auxiliary code combines two checks. It ensures that `ext_sel` was sufficiently stable and that no `mode_change` pulse occurred during the succeeded debounce attempt. The proof can be decomposed into two SVA statements by following the decomposition advice to better cope with complexity. One assertion checks the stability, and the other one that `mode_change` has been low. The exemplary SVA statement `asrt_req_only_if_a_sva` checks the stability of `ext_sel`. A similar property can be implemented for the second part.

As the `$stable` function can only check the current valuation against that *one* cycle in the past, the approach depicted in figure 6.1 is implemented. Once a request is raised (marked with a)³, the `$past` function jumps back to one cycle after the latest legal change of `ext_sel` ($a \rightarrow c$). This can be achieved by using the optional parameter of `$past` for the number of cycles to jump back. When `$stable` is called at that point, `ext_sel` is compared with its value one cycle before c , i.e., the first cycle of the debounce attempt ($c \rightarrow d$). As `$stable` always compares two cycles, the expression repeats $DB-1$ times. Listing 6.1 shows the SVA property from the description above.

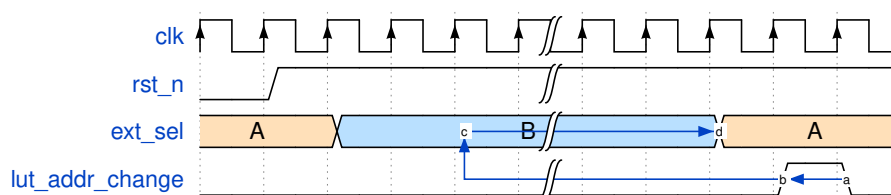


Figure 6.1.: Concept to prove the debounce delay with pure SVA.

```

1  asrt_req_only_if_a_sva: assert property (
2    lut_addr_change
3    |->
4    $stable($past(ext_sel, DEBOUNCE_CYCLES)) [*(DEBOUNCE_CYCLES-1)]
5  );

```

Listing 6.1: Prove debounce delay without auxiliary code.

²Strictly speaking, the *past* counter value due to the delay of `lut_addr_change` by one cycle.

³The sampled value of the request only becomes high at the cycle after the DUT sets the request (at b).

To compare the runtime differences between the SVA statement and using auxiliary code, the HDL helper code from listing 5.16 is slightly modified to match the port namings of the open debouncer module. Moreover, the code has been instrumented as shown in listing 6.2 to run script-based measurements.

```

1  bit sva_din_dly;
2
3  bit [($clog2(`NR_CYCLES)-1):0] sva_stable_cnt;
4
5  always_ff @(posedge clk_i) begin
6      if (~rst_in) begin
7          sva_din_dly    <= 'bo;
8          sva_stable_cnt <= 'bo;
9      end else begin
10         sva_din_dly    <= din;
11         if (din == sva_din_dly) begin
12             if (&sva_stable_cnt) begin
13                 sva_stable_cnt <= sva_stable_cnt;
14             end else begin
15                 sva_stable_cnt <= sva_stable_cnt + 'b1;
16             end
17         end else begin
18             sva_stable_cnt <= 'bo;
19         end
20     end
21 end
22
23 `ifdef USE_AUX
24     asrt_req_only_if_a: assert property (
25         @(posedge clk_i)
26         req |-> ($past(sva_stable_cnt) >= (`NR_CYCLES-1))
27     );
28 `endif
29
30 `ifdef USE_SVA
31     asrt_req_only_if_a_sva: assert property (
32         @(posedge clk_i)
33         req |-> $stable($past(din, `NR_CYCLES))[*](`NR_CYCLES-1)
34     );
35 `endif

```

Listing 6.2: Modified auxiliary code to prove input data stability.

Figure 6.2 plots the runtime of the formal tool to prove `asrt_req_only_if_a_sva` and `asrt_req_only_if_a`, i.e., the assertion with pure SVA and the assertion based on auxiliary code, respectively. The module is parameterized with different debounce delays to estimate how the runtime scales for larger designs. Each point on the x-axis represents a new verification run with a different number of debounce cycles. The y-axis depicts the runtime in seconds for each of the properties at each run. Yices 2 has been used for the plotted measurements. Detailed results can be found in table B.1 and table B.2 in the appendix B.

For a small number of debounce cycles, the performance gap is neglectable. However, the difference increases significantly when the design becomes more complex and the sequential depth more extensive. Although this behavior is expected, such considerations during property design can differentiate between a converging and non-converging proof. More details on auxiliary code can be found in section 3.7.2.

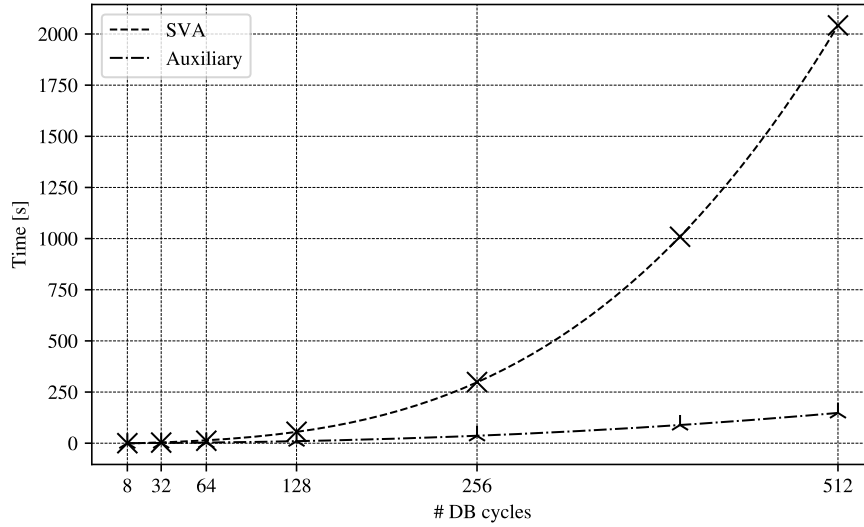


Figure 6.2.: Comparison of the runtime to prove the debounce delay with pure SVA or auxiliary code.

6.1.2. Data Types and Vector Sizes

When reexamining listing 5.16, one can notice that a *bit vector* is used instead of an *integer* type. This consideration has been made to keep the number of added state-holding elements low. Unnecessary bits might lead to longer execution times due to an enlarged state space. To inspect that choice, the same assertion is also proven with the variable defined as an integer (see listing 6.3).

```

1 // bit [ $clog2 (DEBOUNCE_CYCLES+1) - 1:0 ] sva_stable_cnt ;
2 int sva_stable_cnt ;

```

Listing 6.3: Define `sva_stable_cnt` as an integer instead of a bit vector.

Moreover, it can be noticed that the vector used in the original declaration is one bit larger than it needs to be. Instead of counting up to *DB* cycles, the same property can be proven by clipping at *DB-1* cycles. In this case, also the `if` statement to clip at the maximum counter value needs to be adjusted.

Listing 6.4 shows the modified and instrumented code to perform the comparison from above on the open debouncer implementation. An additional assumption has been added to neglect signed comparisons in the formal property. As k-induction is used, the original `if` statement in line 26 has been modified to compare against *larger equal* `NR_CYLCES` instead of an *equal*. It helps to let the proof converge faster but does not change the proof intent.

```

1  `ifdef USE.INT
2      assu_int_pos: assume property (@(posedge clk_i) sva_stable_cnt >= 0);
3  `endif
4
5      bit sva_din_dly;
6  `ifdef USE.VEC
7      bit [$clog2(`NR.CYCLES+1)-1:0] sva_stable_cnt;
8  `elsif USE.OPT
9      bit [$clog2(`NR.CYCLES)-1:0] sva_stable_cnt;
10 `else
11     int sva_stable_cnt;
12 `endif
13
14     always_ff @(posedge clk_i or negedge rst_in) begin
15         if (~rst_in) begin
16             sva_din_dly    <= 'bo;
17             sva_stable_cnt <= 'bo;
18         end else begin
19             sva_din_dly    <= din;
20             if (din == sva_din_dly) begin
21         `ifdef USE.OPT
22             if (sva_stable_cnt >= `NR.CYCLES - 'b1) begin
23         `else
24             if (sva_stable_cnt >= `NR.CYCLES) begin // modified for k-induction
25         `endif
26             sva_stable_cnt <= sva_stable_cnt;
27         end else begin
28             sva_stable_cnt <= sva_stable_cnt + 'b1;
29         end
30         end else begin
31             sva_stable_cnt <= 'bo;
32         end
33     end
34 end
35
36     asrt_req_only_if: assert property (
37         @(posedge clk_i)
38         req |-> ($past(sva_stable_cnt) >= (`NR.CYCLES-1));

```

Listing 6.4: Instrumented auxiliary code to inspect the runtime for different declarations of `sva_stable_cnt`.

Figure 6.3 depicts the results of the Yices 2 solver. The detailed results are listed in table B.3 and table B.4 (appendix B). The formal tool is run multiple times for different debounce durations denoted on the x-axis. The y-axis depicts the runtime in seconds for either the vector-type, the optimized vector-type, or the integer-type declaration of `sva_stable_cnt`. It shows that the vector-types converge faster than the integer. This demonstrates how important it is not to add unnecessary state-holding elements and how easily such mistakes can be introduced. Even the single bit that could be saved in the optimized vector can lead to a significant performance increase in larger designs. In this work, the unideal vector size from listing 5.16 was only realized by adding the code instrumentation for evaluating the runtimes.

The salient point is that suboptimal choices do not yield wrong results but silently consume much more CPU time due to the added complexity. Even on a small design such as the *debounce* module with a couple of tens of lines of code, the difference is noticeable.

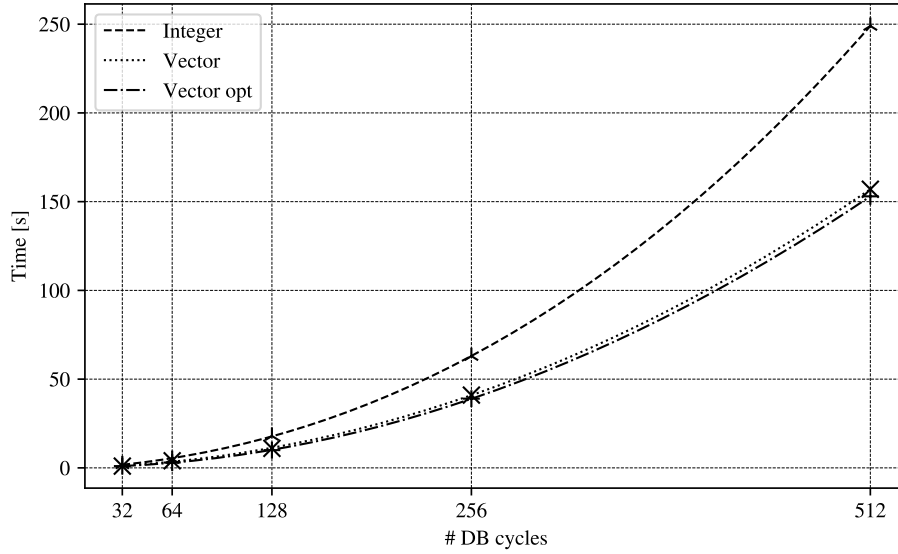


Figure 6.3.: Comparison of the runtime when an integer is used instead of a bit vector.

Additional vector bits are added to the counter vector to investigate the impact of an unnecessarily enlarged state space on the runtime. The experiment is run for two counter bounds: In the first case, the maximum counter value is cropped at $DB-1$ for any vector size ('limited counter value'). In the second case, the counter value can go up until the maximum value of the corresponding vector length is reached, i.e., until all bits are *one* ('unlimited counter value'). Listing 6.5 shows the adapted auxiliary code for the measurements. The experiments are performed with a debounce length of 128 cycles.

```

1  bit sva_din_dly;
2  bit [($clog2(`NR_CYCLES)-1+(`ADD_BITS)):0] sva_stable_cnt;
3
4  always_ff @(posedge clk_i) begin
5      if (~rst_in) begin
6          sva_din_dly    <= 'bo;
7          sva_stable_cnt <= 'bo;
8      end else begin
9          sva_din_dly    <= din;
10         if (din == sva_din_dly) begin
11             `ifdef USE_LIM
12                 if (sva_stable_cnt >= `NR_CYCLES - 'b1) begin
13                     `else
14                         if (&sva_stable_cnt) begin
15                             `endif
16                             sva_stable_cnt <= sva_stable_cnt;
17                         end else begin
18                             sva_stable_cnt <= sva_stable_cnt + 'b1;
19                         end
20                     end else begin
21                         sva_stable_cnt <= 'bo;
22                     end
23                 end
24             end
25         end

```

```

26  asrt_req_only_if: assert property (
27    @(posedge clk_i)
28    req |-> ($past(sva_stable_cnt) >= (`NR_CYCLES-1))
29  );

```

Listing 6.5: Instrumented auxiliary code to inspect the impact of different vector lengths on the runtime.

Figure 6.4 plots the results from the Yices 2 solver. Details can be found in table B.5 in appendix B. The x-axis shows the number of *added* bits from zero (minimum-sized) to 256 additional bits. On the y-axis, the runtime of each verification run is plotted.

In both scenarios, the runtime increases significantly compared to the minimum-sized vector. Moreover, the execution time correlates with the state space size in both cases. Although there are considerably fewer *good* states in the *limited* case (i.e., the counter value is cropped at $DB-1$), there is no apparent deviation to the *unlimited* counter value (i.e., maximum is all ones in the vector).

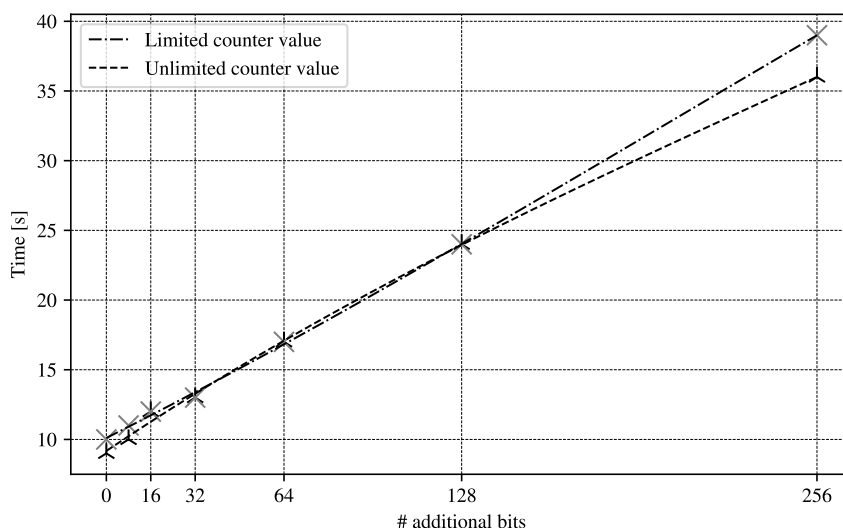


Figure 6.4.: Impact of adding unnecessary counter bits on the runtime of proving the debounce delay for 128 debounce cycles.

6.2. Runtime and Complexity

The proofs on the Lidar evaluation kit are run with default options in batch mode unless mentioned otherwise. The analysis part is split into a subsection addressing the *debounce* module and a subsection dedicated to the *core* module.

6.2.1. Debounce

Figure 6.5 depicts the complexity of the properties proven on the *debounce* module for a debounce duration of six cycles. *Vars* indicates the state-holding elements in the COI whereas *Nodes* correlates to the number of operators to express the property [86]. The complexity may be taken as a rough estimate of the expected runtime. However, a low reported complexity does not necessarily imply a low runtime.

The assertion `asrt_req_len` observes one signal within a sequential depth of one cycle. It checks whether `lut_addr_change` is low in the cycle after it has been high. As expected, the resulting complexity is quite low. `asrt_req_ok_if_a`, however, is far more complex and considers many signals over a larger time frame. This is also reflected in the complexity reported by the formal tool. In section 6.1.1, the use of auxiliary code is discussed. As shown, the property with pure SVA, `asrt_req_only_if_a_sva`, has a much higher complexity than the property which uses auxiliary code, `asrt_req_only_if_a`.

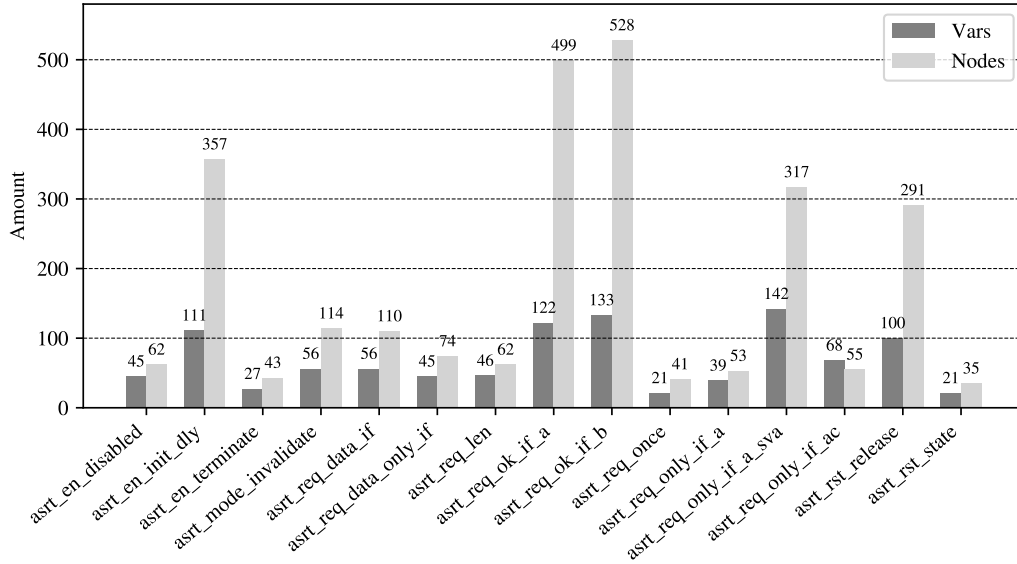


Figure 6.5.: Assertion complexity for 6 debounce cycles.

Figure 6.6 plots the assertion complexity for an increased number of debounce cycles. Instead of six cycles, the debounce module is parametrized with 128 cycles debounce duration. Comparing the charts emphasizes writing properties that scale well for larger designs, when applicable. While properties with a small temporal depth mostly still have low complexity, `asrt_req_ok_if_a` and `asrt_req_ok_if_b` might become infeasible to prove on larger designs.

Figure 6.7 depicts the RAM utilization while proving the SVA statements. All properties are proven with six *DB* cycles and 128 *DB* cycles shown in grey and light-grey, respectively. Unexpectedly, some properties consume less memory when proven on the larger design. Table 6.1 lists the required runtimes for all assertions on the *debounce* module that report a longer execution time than zero seconds on the model with 128 debounce cycles.

6. Evaluation

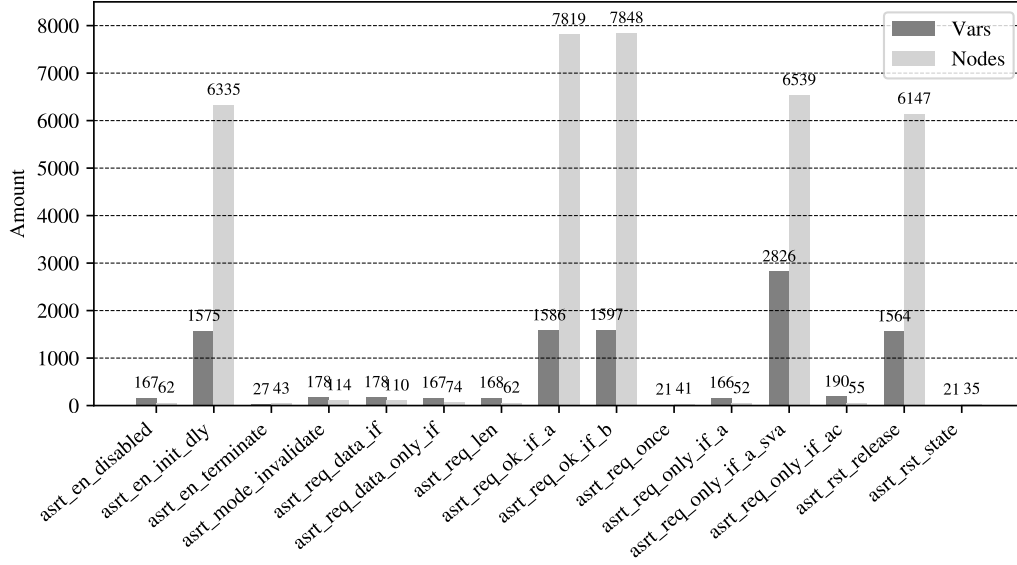


Figure 6.6.: Assertion complexity for 128 debounce cycles.

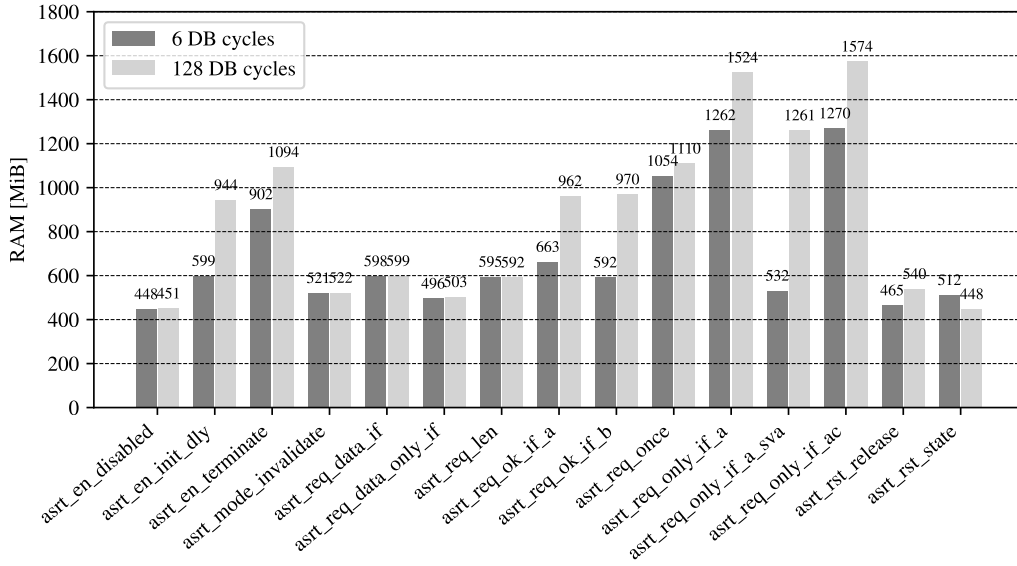


Figure 6.7.: Debounce module: RAM utilization for proving the properties on a model with 6 or 128 debounce cycles, respectively.

6. Evaluation

Table 6.1.: *Debounce* module assertions runtimes for 128 debounce cycles.

Assertion	Runtime [s]
asrt_req_once	1
asrt_req_only_if_a	9
asrt_req_only_if_a_sva	39
asrt_req_only_if_ac	10

6.2.2. Core

All properties used to verify the *core* module have a similar reported complexity of 35-36 *Vars* and 111-115 *Nodes*. Figure 6.8 summarizes the RAM utilization of each property when verified in batch mode.

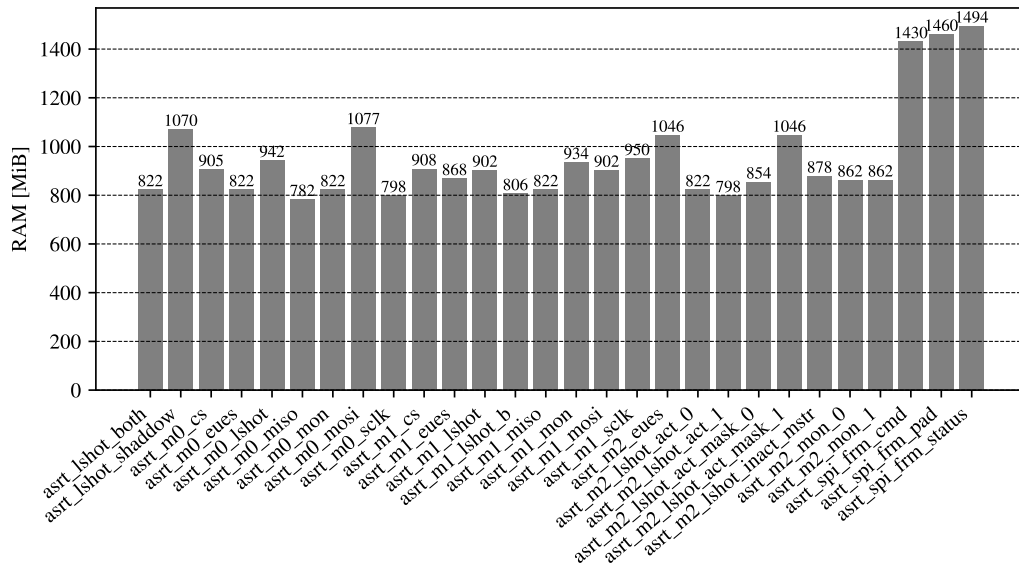


Figure 6.8.: *Core* module: RAM utilization for proving the properties.

Table 6.2 lists the assertions that reported a runtime larger than zero seconds. Most of the properties can be proven almost instantaneously. Properties regarding the data path such as `asrt_spi_frm_cmd`, `asrt_spi_frm_pad`, and `asrt_spi_frm_status`, however, take considerably longer to be proven. This highlights the suitability of formal verification for control-flow-orientated logic and might require abstraction for data paths in larger designs. However, the runtimes are still feasible for the small design.

Table 6.2.: *Core* module assertions runtimes.

Assertion	Runtime [s]
asrt_spi_frm_cmd	380
asrt_spi_frm_pad	100
asrt_spi_frm_status	386
asrt_lshot_shaddow	1

6.3. Coverage

In this section, the coverage reported by the *Quantify* run is inspected. The obtained structural coverage can be used to argue about the completeness of the proof and find possible gaps. It marks statements and branches according to whether an induced mutation is detected (observation coverage). If a statement is marked as *detected*, a mutation is at least reported by one assertion. If it is marked as *undetected*, a mutation will not get uncovered. *Statements* refer to mutations in that line of code. *Branches* are marked as detected if they include larger or equal to one detected statement. Moreover, the Quantify results contain a reachability analysis (simulation coverage) that indicates whether a Quantify target has been active in a witness [86].

To emphasize the power of formal verification once more: An assertion, if proven, holds for *any* input stimuli. Unlike simulation, there cannot be an overlooked stimulus that would trigger that assertion. Subsequently, no metrics for the completeness of the covered input domain are required.

6.3.1. Debounce

The coverage analysis on the *debounce* module is run and completed within six seconds due to the simplicity of the module. The results are depicted in table 6.3. The inspection of the coverage analysis shows that the undetected parts are all related to resetting registers when *rst.n* is high such as shown in listing 6.6.

Table 6.3.: *Debounce* module structural coverage.

	Statements	Branches
detected	92.31%	92.86%
undetected	7.69%	7.14%

```

1  if (~rst_n) begin
2      r.debounce <= 'b0;

```

Listing 6.6: Undetected statement in *debounce* module.

The coverage metric can be increased by adding a dedicated property for the reset state. Once the internal register is made accessible to the checker by adding it as an input port, a property can check its state. Assertion `asrt_rst_state_int` in listing 6.7 checks the reset state of the debounce register (`r.debounce`). Its disable condition is inverted to the other checks that observe the design during operation. However, the effort for these assertions might not be necessary. The formal tool adds auto-checks to inspect the reset states, as shown in figure 6.9. Moreover, the reset state of `r.debounce` cannot influence the design behavior. If it is unequal to zero, there are two cases to consider: In the first case, `ext_sel` has its initial state. At some point, the debounce register would have a state where an interrupt is raised, but it is suppressed as `ext_sel` equals the output address (`lut_addr`). In the second case, `ext_sel` changes and causes an update of the debounce register that overrules the initial state.

```

1  asrt_rst_state_int: assert property (disable iff (rst_n) r.debounce == 'bo);

```

Listing 6.7: Assertion to check the reset state of the debounce register.

Init	General	Stick	Dependencies
Name	Status	Result	
!	!	<=	!
lut_addr	hold	initialized to 8'h00	
lut_addr_change	hold	initialized to 1'b0	
r_debounce	hold	initialized to 6'h00	
r_ext_sel	hold	initialized to 8'h00	

Figure 6.9.: Auto-checks of initialization values in the formal tool.

Reaching 100% coverage is a good sign but does not guarantee that no assertions are missing and that the verification does not have any gaps. A minimal example illustrates this downside of the reported coverage in the subsequent section.

6.3.2. Overlooked Bugs

Formal coverage is an adequate metric to inspect which lines have been covered by assertion and help locate missed-out parts of the design. However, it does not guarantee that there are no gaps in the verification. In listing 6.8 the implementation of a D-type flip-flop is shown whose function is not as intended. It has an enable input (`en`), a data input (`d`), and a data output (`q`). Whenever `en` is high, the input `d` shall be assigned to the output `q` at the next rising clock edge. Otherwise, `q` shall be stable. There is, however, a malicious signal that abstracts a more complex error in a real design. `malicious` is assigned to the inverted enable signal. Thus, it is *low* in the subsequent cycle after `en` was high and the data is correctly assigned to the output. However, in any following cycle, the output is inverted, as `malicious` is high.

6. Evaluation

```
1 module malicious_dff (
2     input logic clk ,
3     input logic rst_n ,
4     input logic en ,
5     input logic d ,
6     output logic q
7 );
8 logic data;
9 logic malicious;
10
11 assign q = data ^ malicious;           // detected
12
13 always_ff @(posedge clk or negedge rst_n) begin
14     if (~rst_n) begin                 // detected
15         malicious <= 1'b0;           // detected
16         data <= 1'b0;                // detected
17     end else begin                   // detected
18         data <= en ? d : data;       // detected
19         malicious <= ~en;            // detected
20     end
21 end
22 endmodule // malicious_dff
```

Listing 6.8: Malicious D-type flip-flop design.

Listing 6.9 shows the two assertions that have been added to the design. Assertion `asrt_rst_state` checks the correct initialization. Moreover, assertion `asrt_check` proves that the data is correctly assigned to the output after `en` was set. By applying these two assertions 100% structural coverage is achieved as `asrt_rst_state` checks the reset if and `asrt_check` has all other lines in its COI. Thus, a mutation would be detected by the set of assertions.

```
1 module malicious_dff_checker (
2     input logic clk ,
3     input logic rst_n ,
4     input logic en ,
5     input logic d ,
6     input logic q ,
7     // White-Box
8     input logic malicious ,
9     input logic data
10 );
11 default disable iff (~rst_n);
12
13 default clocking cb @(posedge clk);
14 endclocking // cb
15
16 asrt_check: assert property (en ==> (q == $past(d)));
17
18 // This would fail
19 //
20 //asrt_forgotten: assert property ((~en) ==> $stable(q));
21
22 asrt_rst_state: assert property (
23     disable iff (rst_n) (~malicious & ~data) );
```

Listing 6.9: Malicious D-type flip-flop checker.

Nevertheless, the verification is not complete as it is not proven that the output remains stable if `en` is low. This check is labeled with `asrt_forgotten` and might be forgotten during verification. As expected, the assertion fails when it is added to the checker. The example highlights that 100% structural coverage does not imply completeness, and thorough DV planning and other metrics are required.

6.3.3. Core

For the *core* module, the Quantify run took 30.7 hours to complete and was run in GUI mode. At first glance, the reported structural coverage does not seem worth the effort to construct the formal properties. However, the results must be inspected in more detail to make a valid grading. Some of the *core* functionality has been neglected in the lightweight formal proof, which negatively influences the overall coverage metrics.

Table 6.4.: *Core* module structural coverage.

	Statements	Branches
detected	28.46%	26.96%
undetected	71.54%	73.04%

Table 6.5 breaks down the coverage results to the individual sub-modules as shown in the design specifications in figure 4.6. The modules `registers` and `spi_slave` compose the *fpga config* block from the architecture diagram (figure 4.6). Moreover, `spi_shift_reg` and `crc` are part of the *spi master*.

The coverage results are split into two groups, *statement* coverage and *branch* coverage, and categorized as detected (Det), undetected (unDet) and excluded (Excl). In addition to the categorization from above, *excluded* refers to code that is not considered in the Quantify run, such as dead-code or code not active in the current parametrization. Going through the list gives more insight into the covered parts. Table 6.6 comments on the achieved coverage metrics.

6. Evaluation

Table 6.5.: Core module structural coverage details.

Module	Statements			Branches		
	Det	UnDet	Excl	Det	UnDet	Excl
crc	0%	100%	0%	0%	100%	0%
ctrl	92.6%	7.4%	0%	93.3%	6.7%	0%
debounce	23.1%	76.9%	0%	21.4%	78.6%	0%
lut	1.4%	98.6%	0%	0%	100%	0%
packet	50%	50%	0%	100%	0%	0%
registers	0%	100%	0%	0%	100%	0%
spi_master	66.7%	33.3%	0%	77.8%	22.2%	0%
spi_shift_reg	77.8%	22.2%	0%	33.3%	66.7%	0%
spi_mux	44.4%	0%	55.6%	0%	0%	100%
spi_slave	0%	96.8%	3.2%	0%	100%	0%

Table 6.6.: Core module structural coverage analysis.

Module	Analysis
crc	No statements in the CRC module are covered as the output data is not observed with any of the formal properties. While some frame parts are checked (like the command and the padding), the CRC byte is not within this scope.
ctrl	The control logic is part of the applied formal checks to a large extent. Not covered lines address the reset state and <i>default</i> branches in the case statements that can be safely validated by inspection.
debounce	The <i>debounce</i> module is mainly skipped on purpose as it is already verified separately.
lut	The address translation is not intended to be covered. One line that selected the correct driver from the address' LSB, is implicitly covered, however.
packet	The packet is partly covered. Assertions <code>asrt_spi_frm_status</code> , <code>asrt_spi_frm_pad</code> , and <code>asrt_spi_frm_cmd</code> check the frame parts. Other parts such as the CRC byte are not checked.
registers	The registers are part of operating mode M_3 which is excluded from verification.
spi_master	The SPI master verification is limited to observing some of the sent frame parts. Subsequently, a correct start and end are partly included.

continues on next page

<code>spi_shift_reg</code>	All lines of the shift registers are included apart from some conditions for the reset state.
<code>spi_mux</code>	The SPI MUX reports some <i>excluded</i> lines. These refer to optional behavior which was disabled in the verification run. Thus, the circuit parts are considered dead code and excluded from the coverage analysis. All the active code is observed.
<code>spi_slave</code>	Similar to the registers, the SPI slave is only active in operating mode M_3 and thus not considered in the verification run.

6.4. Efficiency

All properties can be proven within a reasonable time. Thus, no further optimization is required. One technique that might be a first choice if the performance is too low, is *case-splitting*⁴. Many properties use the `throughout` operator to match a certain precondition, such as a selected mode or *enable* signal level. Examples are `cov_int.raise`, `cov_termination`, `cov_no_int`, as well as `arst_req_once` from section 5.2 and `cov_m2_frm_driver0`, and `cov_m2_frm_driver1` from section 5.3. Instead of differentiating between the operating conditions, e.g., *mode x is selected* or *the module is enabled and signal x is low* by the `throughout` operator, the cases could be proven separately. Preconditions are assumed by `assume` statements, and the single formal run is split into a set of runs each proving a fraction of the overall functionality. Although the `throughout` operator is considered efficient in FV [18], compositional reasoning of the different modes eases the understanding of the properties to implement further techniques if required. A brief overview of possible choices is given in section 3.7.

⁴See section 3.7.3.

7. Conclusion

This chapter summarizes the results of the formal verification endeavor during this thesis and outlines future work.

7.1. Results

By applying FPV, specific design properties could be proven correct in all scenarios and under all input stimuli. Although FV requires neither a testbench nor input stimuli, creating the formal environment results in a considerable amount of work. Moreover, accurate and unambiguous specifications are needed to model the properties and enter the formal verification flow. The deep insight into the design gained during the verification process came with the cost of many iterative cycles of fixing assertions, adapting assumptions, and inspecting the design.

While the power of formal methods is considering the entire state space in one run, the effort to exclude illegal input sequences must not be neglected. Sequences that cannot appear in the actual application must be declared and excluded by proper assumptions. Adequate assumptions evolved during this iterative phase and could not accurately be written a priori. From an engineer's perspective, the tools can get very 'creative' in breaking assertions with illegal inputs. However, the imposed iterations of inspecting the CEX and adapting the properties also uncovered many corner-case situations that might be overlooked in dynamic verification and hand-crafted test cases.

Although the verified design seems as straightforward as a *debounce* module, FV was able to reveal weaknesses in the specifications and find unintended behavior.

Assertion `asrt_en_disabled` showed that disabling the debounce module does not guarantee that no subsequent request is raised. Each successful debounce attempt is delayed by one clock cycle before the request line is asserted. A valid request might be raised one cycle after disabling the module. The behavior was ambiguously defined in the specification. It could be interpreted as either (a) no new debounce attempt shall succeed or (b) no request shall be raised if the module is disabled. The implementation behaves as the first interpretation and can be changed to the latter by *and*-ing the *enable* signal (`debounce_enable`) to the request output (`lut_addr_change`).

Another notable behavior has been shown by the cover property `cov_chg_disabled`. Initially, the behavior on dynamically disabling the module was not documented in the required amount of detail. FV supported understanding of the design and different scenarios when the `ext_sel` vector changes before or during a disabled phase. Performing this process also sharpened the mind to think of other corner-case situations.

In the top-level *core* module, many aspects of control-related logic of the different operating modes could be formally proven. As the COI in these cases is narrow, the formal tool came up with a proof result without any difficulties and without requiring major abstraction techniques. Auxiliary code was used to implement a shift-register-based SPI receiver to observe the transmitted frame's content. Although the complexity was still feasible, it became apparent that wide data paths and larger sequential depths highly complicate the proof.

Moreover, it has been shown that debugging assertions and fixing assumptions is not always that straightforward. However, modern commercial tools support the wiggling phase with various features. Besides, there are many noteworthy methodologies in the research domain that keep their focus on the open-source spirit.

The proven DUT is relatively small, and the performed checks only required minor complexity considerations. Still, larger state spaces, such as those caused by data paths, massively impact the runtime and performance of the formal tools. Abstraction techniques might be required to cope with complexity and obtain a converging proof in larger designs. Engineers must know the limitations and design the formal environment accordingly. Even aspects that can be neglected in dynamic verification, such as the chosen data type, can cause a significant performance decrease in FV. In a bug hunting approach, a trade-off is needed regarding which aspects to prove in simulation and which by FV. When applied adequately, the overall verification flow can highly benefit from formal methods both in a full formal proof and by applying checks to certain design aspects.

When FV is meant to replace simulation-based verification, thorough planning is required not to leave gaps. It has been shown that even 100% structural coverage by a set of assertions does not ensure the correctness of the design. However, there exist methodologies to detect and close holes in a formal verification environment, such as the *Gap-Free Verification* as outlined in section 3.6.2.

7.2. Conclusion and Future Work

During this work, formal verification has been introduced from a designer's perspective. Chapter 1 gave a motivation for the importance of functional verification and outlined the thesis' goals. After the introduction of ABV in section 2.2.1, the significance of assertions for dynamic and static verification was discussed. In section 2.3 a brief introduction to the mathematics behind model checking was given before the focus was set on SystemVerilog Assertions to express formal properties. Chapter 3 summarized related work as well as state-of-the-art methodologies that support FPV. As the main limiting factor of model checking is the state space explosion, section 3.7 discussed techniques to keep the complexity feasible. The runtime evaluation in section 6.1 demonstrated how easily mistakes can be made that significantly impact the execution time and might prevent convergence.

After the design specifications have been declared in chapter 4, chapter 5 focused on property modeling. It showed how properties can be kept in a separate checker file and bound to the DUT. The proposed verification flow was presented and applied to the implemented digital design. It highlighted the importance of creating suitable cover

properties to get an initial feeling for the design and rule out severe bugs first. Writing and correcting the properties might take numerous iterations. Missing assumptions or errors in formal properties were frequent sources of failing proofs. Although the wiggling phase might initially appear tedious, it leads to a deeper understanding of the specifications and the design, which in turn may lead to the identification of system-level corner cases. As the constructed statements are proven on the implementation, they unambiguously describe properties of the design and might be taken as a part of the design documentation.

The work showed how FPV can enhance traditional dynamic verification to check the design intent when a suitable testing environment may not be ready. Even when a full formal proof may not be feasible, FV is an appropriate technique for lightweight verification and proving certain aspects. FV found weaknesses and ambiguities in the specification that may lead to edge-case bugs at the system level.

While the proposed methodology gives satisfactory guidance for applying FPV to small designs, topics to verify larger designs are not discussed. This includes hierarchical verification planning as it might not be known which parts of the design are better suited to simulation or FV in the beginning. Moreover, in more complex designs, the tool might not be able to prove a property thoroughly. When a full proof does not converge due to a large state space, Bounded Model Checking (BMC) can be used to prove the design up to a certain number of cycles from the reset states. A bounded proof is computationally cheaper and can provide suitable verification results if the bound is selected carefully. Future work will investigate hierarchical verification planning, the evidence gained from a bounded proof, and how a reasonable depth can be determined for a given design.

Bibliography

1. El-Kharashy, H., Khami, M., Sala, A. & Korany, M. *A novel assertions-based code coverage automatic CAD tool in IEEE EUROCON 2017 -17th International Conference on Smart Technologies* (2017), 277–281.
2. Research, D. *The Emergence of Big Code* 2020.
3. Singh, N. *Using Event-B for Critical Device Software Systems* (Jan. 2013).
4. Lyu, Y. & Mishra, P. *Automated Test Generation for Activation of Assertions in RTL Models in 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2020), 223–228.
5. Clarke, E., Klieber, W., Nováček, M. & Zuliani, P. *Model Checking and the State Explosion Problem* 1–30 (Jan. 2012).
6. Hobbs, C. *Embedded Software Development for Safety-Critical Systems, Second Edition* (CRC Press, 2019).
7. Wille, R. et al. *Identifying a Subset of System Verilog Assertions for Efficient Bounded Model Checking in 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools* (2008), 542–549.
8. Clarke, E., Grumberg, O., Kroening, D., Peled, D. & Veith, H. *Model Checking, second edition* (MIT Press, 2018).
9. IEC 62531:2012(E) (IEEE Std 1850-2010): Standard for Property Specification Language (PSL). *IEC 62531:2012(E) (IEEE Std 1850-2010)*, 1–184 (2012).
10. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, 1–1315 (2018).
11. Siegel, M., Maggiore, A. & Pichler, C. *Untwist your brain - Efficient debugging and diagnosis of complex assertions in* (Aug. 2009), 644–647.
12. Boule, M., Chenard, J.-S. & Zilic, Z. *Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis in 8th International Symposium on Quality Electronic Design (ISQED’07)* (2007), 613–620.
13. Sonny, A. T. & Lakshmiprabha, S. *OVL, PSL, SVA: Assertion based verification using checkers and standard assertion languages in 2013 International Conference on Advanced Computing and Communication Systems* (2013), 1–4.

14. OneSpin Solutions GmbH. *When Correct Is Not Enough: Formal Verification of Fault-Tolerant Hardware V.14*. May 2020.
15. Bormann, J. et al. *Complete Formal Verification of TriCore2 and Other Processors* in (2007).
16. Seligman, E., Schubert, T. & Kumar, M. V. A. K. *Formal Verification: An Essential Toolkit for Modern VLSI Design* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2015).
17. Kern, C. & Greenstreet, M. R. Formal Verification in Hardware Design: A Survey. *ACM Trans. Des. Autom. Electron. Syst.* **4**, 123–193 (Apr. 1999).
18. Cerny, E., Dudani, S., Havlicek, J. & Korchemny, D. *SVA: The Power of Assertions in SystemVerilog* (Springer International Publishing, 2016).
19. Jianfeng, Z. *Case Study: Discovering Hardware Trojans Based on model checking* in (Nov. 2018), 64–68.
20. OneSpin Solutions GmbH. *Using Formal to Verify Safety-Critical Hardware for ISO 26262 V.2*. May 2020.
21. Sarikhada, R. M. & K Shah, P. *Speed up the validation process by formal verification method* in *2020 IEEE International Conference for Innovation in Technology (INOCON)* (2020), 1–4.
22. Edelman, A. The mathematics of the Pentium division bug. *SIAM review* **39**, 54–67 (1997).
23. Moler, C. *A tale of two numbers* 1995.
24. Dill, D. L. & Rushby, J. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer* **29**, 23–24 (1996).
25. Ben-Ari, M. The Bug That Destroyed a Rocket. **33**, 58–59 (June 2001).
26. esa. *ARIANE 5 Flight 501 Failure Report by the Inquiry Board* July 1996.
27. Li, Y., Wu, W., Hou, L. & Cheng, H. *A Study on the Assertion-Based Verification of Digital IC* in *2009 Second International Conference on Information and Computing Science* **2** (2009), 25–28.
28. Kuang-Chien Chen, K.-C. C. *Assertion-based verification for SoC designs in ASIC, 2003. Proceedings. 5th International Conference on* **1** (2003), 12–15 Vol.1.
29. Cortez, J. & Torres, D. *Design and verification based on assertions: some statistics* in *2005 2nd International Conference on Electrical and Electronics Engineering* (2005), 132–135.
30. Foster, H. Applied Assertion-Based Verification: An Industry Perspective. *Found. Trends Electron. Des. Autom.* **3**, 1–95 (Jan. 2009).
31. Datta, K. & Das, P. *Assertion based verification using HDVL* in *17th International Conference on VLSI Design. Proceedings.* (2004), 319–325.

32. Devarajegowda, K., Servadei, L., Han, Z., Werner, M. & Ecker, W. *Formal Verification Methodology in an Industrial Setup* in 2019 22nd Euromicro Conference on Digital System Design (DSD) (2019), 610–614.
33. Spear, C. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features* (Springer US, 2006).
34. DasGupta, P. *A Roadmap for Formal Property Verification* 1–251 (Jan. 2006).
35. IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, 1–458 (2020).
36. Borchers, K., Montenegro, S. & Dannemann, F. *Volatile Register Handling for FPGA Verification Based on SVAs Incorporated into UVM Environments* in 2019 IEEE Aerospace Conference (2019), 1–7.
37. Bromley, J. *If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language* in *Proceedings of the 2013 Forum on specification and Design Languages (FDL)* (2013), 1–7.
38. Grimm, T., Lettnin, D. & Hübner, M. *A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip* May 2018.
39. Cohen, B., Venkataramanan, S., Kumari, A. & Piper, L. *SystemVerilog Assertions Handbook: ... For Dynamic and Formal Verification* 4th (CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2015).
40. Krishnegowda, D. *A primer on Logical Equivalence Checking (LEC) using Conformal* Oct. 2021.
41. Zhu, J., Wang, H., Xu, Z. & Xu, C. *A new model for model checking: Cycle-weighted Kripke structure* Mar. 2010.
42. Xin-feng, Z., Jian-dong, W., Bin, L., Jun-wu, Z. & Jun, W. *Methods to Tackle State Explosion Problem in Model Checking* in 2009 Third International Symposium on Intelligent Information Technology Application **2** (2009), 329–331.
43. Thomas, D. *Logic Design and Verification Using SystemVerilog: (Revised)* (CreateSpace Independent Publishing Platform, 2016).
44. Doulos. *The Verilog Golden Reference Guide* (Doulos, 1996).
45. IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001*, 1–792 (2001).
46. Cummings, C. & Salz, A. *SystemVerilog Event Regions, Race Avoidance & Guidelines* Dec. 2007.
47. Cummings, C. *Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill* Jan. 2000.
48. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, 1–1315 (2013).

49. *Jasper UNR App* Accessed: 2022-04-30. Cadence Design Systems. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-coverage-unreachability-app.html.
50. Waterman, A. *et al. The RISC-V Instruction Set Manual* 2014.
51. riscv.org. *RISC-V Cores and SoC Overview* Commit d1165434ede91b29779bcc75f814d0bce30829e7. 2021. <https://www.github.com/riscvarchive/riscv-cores-list>.
52. Dörflinger, A. *et al. comparative survey of open-source application-class RISC-V processor implementations* This version contains updates for CVA6 SPEC CPU2017 scores. 2021.
53. Chupilko, M., Kamkin, A. & Protsenko, A. *Open-Source Validation Suite for RISC-V in 2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)* (2019), 7–12.
54. Cockrell, M. *Evaluation of RISC-V for Pixel Visual Core* 2018. <https://content.riscv.org/wp-content/uploads/2018/05/13.15-13.30-matt-Cockrell.pdf>.
55. Schiavone, P. D. *et al. An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study in 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (2018), 43–48.
56. #374 - An Interview with Claire (née 'Clifford') Wolf <https://theamphour.com/374-an-interview-with-claire-nee-clifford-wolf/>. Accessed: 2022-03-08.
57. Wolf, C. X. Accessed: 2022-03-08. Dec. 2019. <https://twitter.com/oe1cxw/status/1212071128296042496>.
58. Wolf, C. *Formal Verification of RISC-V cores with riscv-formal* Accessed: 2022-03-08. Dec. 2018.
59. Rojas, C., Morales, H. & Roa, E. *A Low-Cost Bug Hunting Verification Methodology for RISC-V-Based Processors in 2021 IEEE International Symposium on Circuits and Systems (ISCAS)* (2021), 1–5.
60. Kumar, B., Jaiswal, A. K., Vineesh, V. S. & Shinde, R. *Analyzing Hardware Security Properties of Processors through Model Checking in 2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)* (2020), 107–112.
61. Foster, H. *Guidelines for creating a formal verification testplan* in (2006).
62. Keng, B., Safarpour, S. & Veneris, A. *Automated debugging of SystemVerilog assertions in Design, Automation Test in Europe* (2011), 1–6.

63. Mostafa, M., Safar, M., El-Kharashi, M. W. & Dessouky, M. *System Verilog Assertion Debugging Based on Visualization, Simulation Results, and Mutation* in 2014 15th International Microprocessor Test and Verification Workshop (2014), 55–60.
64. Orenes-Vera, M., Manocha, A., Wentzlaff, D. & Martonosi, M. *AutoSVA: Democratizing Formal Verification of RTL Module Interactions* in 2021 58th ACM/IEEE Design Automation Conference (DAC) (2021), 535–540.
65. Keng, B., Qin, E., Veneris, A. & Le, B. *Automated debugging of missing assumptions* in 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC) (2014), 732–737.
66. Matsuda, A. *Overcoming the challenges of formal verification and debug* (ed Vennsa Technologies Inc.) Accessed: 2022-02-25. May 2011. <https://www.eetimes.com/overcoming-the-challenges-of-formal-verification-and-debug>.
67. Feng, X., Muchandikar, A. & Chen, X. *The lights in the Tunnel: Coverage Analysis for Formal Verification* in (2016).
68. Tusinschi, N. & Solutions, O. *Mitigating verification risk by understanding the coverage* in (Apr. 2021).
69. Bustan, D. & Havlicek, J. *Some Complexity Results for SystemVerilog Assertions* in. 4144 (Aug. 2006), 205–218.
70. Ram, R. *et al. FORMAL VERIFICATION OF FLOATING-POINT HARDWARE WITH ASSERTION-BASED VIP* in (2018).
71. Hupcey III, J. *How to Reduce the Complexity of Formal Analysis – Part 1 – Finding Where Formal Got Stuck and Some Initial Corrective Steps to Take* Accessed: 2022-03-22. Aug. 2018. <https://blogs.sw.siemens.com/verificationhorizons/2018/08/08/how-to-reduce-the-complexity-of-formal-analysis-part-1-finding-where-formal-got-stuck-and-some-initial-corrective-steps-to-take/>.
72. Hupcey III, J. *How to Reduce the Complexity of Formal Analysis – Part 3 – Assertion Decomposition* Accessed: 2022-03-22. Sept. 2018. <https://blogs.sw.siemens.com/verificationhorizons/2018/09/11/how-to-reduce-the-complexity-of-formal-analysis-part-3-assertion-decomposition/>.
73. Hupcey III, J. *How to Reduce the Complexity of Formal Analysis – Part 2 – Reducing the Complexity of Your Assumptions* Accessed: 2022-03-22. Aug. 2018. <https://blogs.sw.siemens.com/verificationhorizons/2018/08/22/how-to-reduce-the-complexity-of-formal-analysis-part-2-reducing-the-complexity-of-your-assumptions/>.
74. Avery, M. *3 Common Scenarios Which SVA Cannot Describe - Why Auxiliary HDL code is Needed* Accessed: 2022-03-31. Cadence Design Systems, 2019. <https://www.youtube.com/watch?v=gVIgJ6tcNjo>.

75. Kottapalli, V. *Formal Verification of a MESI-based Cache Implementation* in (July 2017).
76. Hupcey III, J. *How to Reduce the Complexity of Formal Analysis – Part 5 – Memory Abstraction* Accessed: 2022-03-22. Oct. 2018. <https://blogs.sw.siemens.com/verificationhorizons/2018/10/23/how-to-reduce-the-complexity-of-formal-analysis-part-5-memory-abstraction/>.
77. Hupcey III, J. *How to Reduce the Complexity of Formal Analysis – Part 4 – Counter Abstraction* Accessed: 2022-03-22. Sept. 2018. <https://blogs.sw.siemens.com/verificationhorizons/2018/09/28/how-to-reduce-the-complexity-of-formal-analysis-part-4-counter-abstraction/>.
78. Da Silva, F. A. et al. *Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs* 2020.
79. Silva, F. A. d., Bagbaba, A. C., Hamdioui, S. & Sauer, C. *Efficient Methodology for ISO26262 Functional Safety Verification* 2019.
80. *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General requirements (see Functional Safety and IEC 61508)* Standard (International Electrotechnical Commission, Geneva, CH, Apr. 2012).
81. *Road vehicles - Functional safety - Part 1: Vocabulary* Standard (International Organization for Standardization, Geneva, CH, Dec. 2018).
82. Nardi, A. & Armato, A. *Functional safety methodologies for automotive applications* 2017.
83. Chang, Y.-C., Huang, L.-R., Liu, H.-C., Yang, C.-J. & Chiu, C.-T. *Assessing automotive functional safety microprocessor with ISO 26262 hardware requirements* 2014.
84. *Road vehicles — Functional safety — Part 5: Product development at the hardware level* Standard (International Organization for Standardization, Geneva, CH, Dec. 2018).
85. OneSpin Solutions GmbH. *OneSpin 360 Assertion Results Integration. Integrating assertion results between OneSpin 360 and Simulation* 2016.
86. OneSpin Solutions GmbH. *User Manual: OneSpin 360™ Version 2021.1.2* (May 2021).

Appendix

Appendix A.

Core Module Verification

Appendix A lists some additional SystemVerilog code for the *core* module verification in section 5.3.

Listing A.1 shows the safety-properties mentioned in section 5.3.3, which are used to verify modes *Mo* and *M1*. The properties primarily check whether the SPI bus is routed through the FPGA correctly and whether the control signals are generated as intended.

```
1  asrt_mo_mosi: assert property (seq_mode_act(Mo.ROUTE.THROUGH)
2    |-> (mosi_from_fpga == mosi_from_rpi)
3  );
4
5  asrt_mo_sclk: assert property (seq_mode_act(Mo.ROUTE.THROUGH)
6    |-> (sclk_from_fpga == sclk_from_rpi)
7  );
8
9  asrt_mo_cs: assert property (seq_mode_act(Mo.ROUTE.THROUGH)
10    |-> ({cs_n_1_from_fpga, cs_n_o_from_fpga} ==
11        {cs_n_1_from_rpi, cs_n_o_from_rpi})
12  );
13
14 asrt_mo_miso: assert property (seq_mode_act(Mo.ROUTE.THROUGH)
15    |-> (miso_to_rpi == miso_to_fpga)
16  );
17
18 asrt_mo_lshot: assert property (seq_mode_act(Mo.ROUTE.THROUGH)
19    |-> ({lshot_en_1_from_fpga, lshot_en_o_from_fpga} ==
20        {lshot_en_1_from_rpi, lshot_en_o_from_rpi})
21  );
22
23 asrt_mo_mon: assert property (seq_mode_act(Mo.ROUTE.THROUGH)
24    |-> (~|{mon_o_from_fpga, mon_1_from_fpga})
25  );
26
27 asrt_mo_eues: assert property (seq_mode_act(Mo.ROUTE.THROUGH)
28    |-> (eues_in_o_from_fpga == eues_1_to_fpga)
29  );
30
31 asrt_m1_mosi: assert property (seq_mode_act(M1.EX.STREAM.TRG)
32    |-> (mosi_from_fpga == mosi_from_rpi)
33  );
34
35 asrt_m1_sclk: assert property (seq_mode_act(M1.EX.STREAM.TRG)
36    |-> (sclk_from_fpga == sclk_from_rpi)
37  );
```

```

38
39 asrt_m1_cs: assert property (seq_mode_act(M1_EX_STREAM.TRG)
40   |-> ({cs_n_1_from_fpga , cs_n_o_from_fpga} ==
41       {cs_n_1_from_rpi , cs_n_o_from_rpi}))
42 );
43
44 asrt_m1_miso: assert property (seq_mode_act(M1_EX_STREAM.TRG)
45   |-> (miso_to_rpi == miso_to_fpga))
46 );
47
48 asrt_m1_lshot: assert property (seq_mode_act(M1_EX_STREAM.TRG)
49   |-> (lshot_en_o_from_fpga == (lshot_en_o_from_rpi & ext_lshot_en_o)))
50 );
51
52 asrt_m1_lshot_b: assert property (seq_mode_act(M1_EX_STREAM.TRG)
53   |-> (lshot_en_1_from_fpga == (lshot_en_1_from_rpi & ext_lshot_en_1)))
54 );
55
56 asrt_m1_mon: assert property (seq_mode_act(M1_EX_STREAM.TRG)
57   |-> (~{|mon_o_from_fpga , mon_1_from_fpga}))
58 );
59
60 asrt_m1_eues: assert property (seq_mode_act(M1_EX_STREAM.TRG)
61   |-> (eues_in_o_from_fpga == 'bo))
62 );

```

Listing A.1: Core module: properties for modes *Mo* and *M1*

In listing A.2 the `sva_lshot_mask` signal is modeled. It implements an abstracted version of the actual FSM that generates the signal. Thus, the DUT's implementation can be compared with a more straightforward model that is independently derived from the specification. Figure 4.5 in the design specifications chapter depicts the timing of the mask signal. As the phases *ACT* (*WAIT_TRANS*) and *DLY* (*WAIT_DLY*) span over many cycles, the signal cannot be efficiently verified with pure SVA.

Assertion `asrt_lshot_both` checks that both channels share the same mask signal in the actual implementation. Property `asrt_lshot_shadow` compares the generated signal of the model with the actual implementation in the design.

```

1 bit sva_lshot_mask;
2 bit [$clog2(Delay.CYCLES) - 1:0] sva_lshot_dly_cnt;
3 bit [$clog2(Delay.CYCLES) - 1:0] sva_lshot_dly_cnt_next;
4
5 typedef enum logic [1:0] {IDLE.OFF, IDLE, WAIT_TRANS, WAIT_DLY}
6   t_sva_shadow_lshot;
7
8 t_sva_shadow_lshot sva_lshot_state;
9 t_sva_shadow_lshot sva_lshot_state_next;
10
11
12

```

```

13 always_comb begin
14     sva_lshot_mask      = 'bo;
15     sva_lshot_state_next = sva_lshot_state;
16     sva_lshot_dly_cnt_next = sva_lshot_dly_cnt;
17
18     case (sva_lshot_state)
19         IDLE_OFF: begin
20             if (lut_addr_change == 1'b1)
21                 sva_lshot_state_next = WAIT_TRANS;
22             end
23
24         IDLE: begin
25             sva_lshot_mask = 'b1;
26             if (lut_addr_change == 1'b1)
27                 sva_lshot_state_next = WAIT_TRANS;
28             end
29
30         WAIT_TRANS:
31             if (sva_old_fpga_trans_act && ~sva_fpga_trans_act) begin
32                 sva_lshot_state_next = WAIT_DLY;
33                 sva_lshot_dly_cnt_next = DELAY_CYCLES;
34             end
35
36         WAIT_DLY: begin
37             sva_lshot_dly_cnt_next = sva_lshot_dly_cnt - 'b1;
38             if (sva_lshot_dly_cnt_next == 'bo)
39                 sva_lshot_state_next = IDLE;
40             end
41
42         default: sva_lshot_state_next = IDLE ;
43     endcase // case (sva_lshot_state)
44 end
45
46 always_ff @(posedge clk or negedge rst_n) begin
47     if (~rst_n || mode != M2_EXT_ROW_SEL) begin
48         sva_lshot_state <= IDLE_OFF;
49         sva_lshot_dly_cnt <= 'bo;
50     end else begin
51         sva_lshot_state <= sva_lshot_state_next;
52         sva_lshot_dly_cnt <= sva_lshot_dly_cnt_next;
53     end
54 end
55
56 asrt_lshot_both: assert property (
57     int_lshot_en_mask[1] == int_lshot_en_mask[0]
58 );
59
60 asrt_lshot_shaddow: assert property (
61     seq.mode_act(M2_EXT_ROW_SEL) |-> (sva_lshot_mask == int_lshot_en_mask[0])
62 );

```

Listing A.2: Core module: shadow model for lshot_en_mask

Listing A.3 depicts the simplistic SPI receiver to check the frame's content at a rising edge of chip-select, i.e., at the end of each transmission. The frame is of the type `sva_spi_frame_t`, which is a *packed* array to reinterpret the shift register and decompose the frame into its sub-parts. Safety properties to prove the frame's data are shown in section 5.3.3.

```

1  typedef struct packed {
2      bit      [7:0]  status;
3      bit      [7:0]  cmd;
4      bit  [5:0][7:0]  data;
5      bit      [7:0]  crc;
6      bit  [9:0][7:0]  padding;
7  } sva_spi_frame_t;
8  sva_spi_frame_t sva_spi_frame;
9
10 always_ff @(negedge clk or negedge rst_n) begin
11     if (~rst_n) begin
12         sva_spi_frame <= 'bo;
13     end else if (sclk_from_fpga) begin
14         sva_spi_frame <= {sva_spi_frame[$bits(sva_spi_frame) - 1:0], mosi_from_fpga};
15     end
16 end

```

Listing A.3: Core module: simple SPI receiver shift register.

Appendix B.

Detailed Evaluation Results

Appendix B contains implementation details and results from the evaluation in section 6.1.

Open Debouncer Implementation

Listing B.1 shows the simplistic debouncer implementation that is used to evaluate different considerations when designing the formal environment.

```
1  module debounce
2  (
3    input  wire  clk_i ,
4    input  wire  rst_in ,
5    input  wire  din ,
6    output reg   req
7  );
8
9    reg          din_dly ;
10   reg  [`NR_CYCLES-1:0] shft_dly ;
11
12   always @(posedge clk_i) begin
13     if (~rst_in) begin
14       din_dly  <= 'bo;
15       shft_dly <= 'bo;
16       req      <= 'bo;
17     end else begin
18       din_dly  <= din;
19       shft_dly <= {shft_dly[`NR_CYCLES-2:0], (din ^ din_dly)};
20       req      <= shft_dly[`NR_CYCLES-1]
21                 && (shft_dly[(`NR_CYCLES-2):0] == 'bo);
22     end
23   end
24
25 endmodule
```

Listing B.1: Open *debouncer* implementation.

Results: Pure SVA vs. Auxiliary Code

Table B.1 and table B.2 list the runtimes of the properties `asrt_req_only_if_a` (*AUX*) and `asrt_req_only_if_a_sva` (*SVA*), which are both proven with the solvers Yices 2 and Z3. Both properties verify the same design aspect. `asrt_req_only_if_a` uses auxiliary code whereas `asrt_req_only_if_a_sva` is implemented with pure SVA. The first column shows a different parametrization of the debounce duration. Subsequent columns depict the runtime to prove the correlated property. Further details can be found in section 6.1.1.

Table B.1.: Runtimes with pure SVA and auxiliary code proven with Yices 2.

<i>DB</i> cycles	Runtime <i>AUX</i>	Runtime <i>SVA</i>
#	sec	sec
8	0	0
16	0	1
32	1	3
64	3	12
100	6	33
128	10	56
256	36	299
400	89	1009
512	148	2042
800	406	7699
1024	732	16528

Table B.2.: Runtimes with pure SVA and auxiliary code proven with Z3.

<i>DB</i> cycles	Runtime <i>AUX</i>	Runtime <i>SVA</i>
#	sec	sec
8	0	0
16	0	1
32	1	7
64	9	124
100	44	1391
128	96	4739
256	1143	timeout

Results: Vector vs. Integer as Counter Variable

Table B.3 and Table B.4 depict the detailed runtimes when the counter variable is declared as either an integer, a vector, or an optimized vector¹. The runtimes refer to proving assertion `asrt_req_only_if_a` and are evaluated for the solvers Yices 2 and Z3. As with the evaluation from above, different debounce durations are used to inspect the scalability. A further discussion is given in section 6.1.2.

Table B.3.: Runtimes for different declarations of the counter variable; Yices 2.

<i>DB cycles</i>	<i>Runtime Vector</i>	<i>Runtime Optimized Vector</i>	<i>Runtime Integer</i>
#	sec	sec	sec
32	1	1	2
64	3	3	5
128	11	10	18
256	37	34	59
512	154	146	259

Table B.4.: Runtimes for different declarations of the counter variable; Z3.

<i>DB cycles</i>	<i>Runtime Vector</i>	<i>Runtime Optimized Vector</i>	<i>Runtime Integer</i>
#	sec	sec	sec
32	1	1	2
64	9	12	18
128	77	98	118
256	914	971	1354

¹The original implementation used a vector that was one bit larger than required. Thus, the *optimized* version inspects the runtime difference by saving one bit.

Results: Different Vector Lengths

To investigate the impact of an enlarged state space in some more detail, section 6.1.2 discusses the effect of different vector sizes on the obtained runtime of the formal tool. It shall be shown what impact a loosely chosen (approximated towards larger sizes) vector length of verification helper code can have on the runtime. Subsequently, the minimum-sized vector is made longer by adding bits to its size (that are not required to prove the assertion). Moreover, two upper bounds of the maximum value are compared. Details are given in section 6.1.2. The runtimes shown in table B.5 are obtained from a model with 128 debounce cycles. Yices 2 and Z₃ are used as solvers.

Table B.5.: Runtimes when the state-holding vector is artificially enlarged on a model with 128 debounce cycles.

Added bits #	<i>Limited Counter</i>		<i>Unlimited Counter</i>	
	Runtime Yices 2 sec	Rt. Z ₃ sec	Rt. Yices 2 sec	Rt. Z ₃ sec
0	91	10	94	9
8	93	11	99	10
16	87	12	116	12
32	123	13	124	13
64	134	17	146	17
128	160	24	202	24
256	240	39	299	36
400	372	58	471	52
512	514	100	540	80