A THESIS SUBMITTED FOR THE DEGREE OF DR. RER. NAT.

# COQ meets CλaSH

## PROPOSING A HARDWARE DESIGN SYNTHESIS FLOW THAT COMBINES PROOF ASSISTANTS WITH FUNCTIONAL HARDWARE DESCRIPTION LANGUAGES


BY

FRITJOF BORNEBUSCH

Faculty 3 - Mathematics and Computer Science


Supervisor: Prof. Dr.-Ing. Robert Wille
First Examiner: Prof. Dr. Rolf Drechsler
Second Examiner: Prof. Dr.-Ing. Görschwin Fey

Disputation on: 16.06.2021

# DISCLAIMER

I hereby declare that

- this dissertation is my own original work,
- it has been completed without claiming any illegitimate assistanc eand
- I have acknowledged all sources used (both, verbatim and regarding their content).

---

Fritjof Bornebusch, Bremen 19 April 2021

# ABSTRACT

Over the last few decades, electronic circuits have more and more become a part of our lives, and their area of application expands continuously. As a result, these circuits are getting more and more complex through these areas of application. They are synthesized from hardware designs, which describe their functional and timing behavior at a higher level of abstraction. Since safety-critical systems such as cars rely on these designs, it is essential to address the increasing complexity as it directly impacts the design's correctness.

This dissertation investigates the increasing complexity of hardware designs by proposing and evaluating a hardware design synthesis flow that automatically propagates verification results from a formal specification to an implementation. A model at the Electronic System Level (ESL) is automatically extracted from a specification at the Formal Specification Level (FSL), and this model is synthesized into an implementation at the Register-Transfer Level (RTL). This automatic propagation contrasts with the established hardware design flow, which relies on manual realizations at the ESL and RTL levels. Due to the missing propagation of verification results, these manual realizations rely on the required test benches' quality. Consequently, similar verification tasks are repeated at different levels. The proposed synthesis flow combines the proof assistant Coq with the functional hardware description language CλaSH to achieve the automatic propagation of verification results. This combination avoids test bench generation for the model and the implementation.

Furthermore, the proposed flow allows the investigation of problems in the model or the implementation already at the FSL level, e.g., arithmetic overflows for *finite* integer types. The established hardware design flow models *infinite* integer types at the FSL level, which do not realize an overflow behavior. To specify finite integer types at the FSL level, *dependent types* are used. Based on these types, a generalizable overflow detection scheme is presented to detect arbitrary arithmetic overflows. The scheme's impact on the final implementation's performance and consumed space is evaluated by comparing implementations that realize operations using this scheme with those that do not.

To investigate the general implementation's performance synthesized by the proposed flow, a 32-bit MIPS processor is formally specified and verified. Its final implementation is compared with a functional equivalent processor synthesized by a state-of-the-art hardware acceleration framework. This comparison shows the proposed synthesis flows' potential and opens the door for further research on synthesizing hardware implementations that satisfy correctness properties and take performance under consideration.

# ZUSAMMENFASSUNG

In den letzten Jahrzehnten wurde elektronische Schaltkreise immer mehr Teil unseres Lebens und ihre Anwendungsgebiete erweitern sich kontinuierlich. Aufgrund dieser Anwendungsgebiete werden diese Schaltkreise immer komplexer. Sie werden aus Hardware-Entwürfen synthetisiert, welche ihr funktionales und zeitliches Verhalten auf höherer Abstraktionsebene beschreiben. Da sicherheitskritische Systeme wie Autos auf diese Entwürfe setzen, ist es wichtig, diese steigende Komplexität zu adressieren, da sie sich direkt auf die Korrektheit des Entwurfs auswirkt.

Diese Dissertation untersucht die zunehmende Komplexität von Hardware Entwürfen, indem sie einen Hardware-Entwurf Synthese Ablauf vorstellt und evaluiert, der automatisch Verifikationsergebnisse von einer formalen Spezifikation zu einer Implementierung propagiert. Ein Modell auf der Elektronischen System Ebene (engl. ESL) wird automatisch aus einer Spezifikation auf der Formalen Spezifikations Ebene (engl. FSL) extrahiert, und dieses Modell wird in eine Implementierung auf der Register-Transfer Ebene (engl. RTL) synthetisiert. Diese automatische Propagation bildet einen Kontrast mit dem etablierten Hardware-Entwurfs Ablauf, der auf eine manuelle Realisierung auf der ESL und RTL-Ebene setzt. Durch die fehlende Propagation von Verifikationsergebnissen, verlassen sich diese Realisierungen auf die Qualität der verwendeten Test Fälle. Als Konsequenz werden ähnliche Verifikationsaufgaben auf mehreren Ebenen wiederholt. Der hier vorgestellte Synthese Ablauf kombiniert den Beweis Assistenten Coq mit der funktionalen Hardware Beschreibungssprache CλaSH, um die automatische Propagation von Verifikationsergebnissen zu erreichen. Diese Kombination macht die Generierung von Test Fällen für das Modell und die Implementierung überflüssig.

Weiterhin erlaubt der vorgestellte Synthese Ablauf die Untersuchung von Problemen des Modells oder der Implementierung bereits auf der FSL-Ebene, z.B. arithmetische Überläufe von *endlichen* ganzzahligen Typen. Der etablierte Hardware-Entwurfs Ablauf modelliert *unendliche* ganzzahlige Typen auf der FSL-Ebene, die kein Überlaufverhalten realisieren. Um diese endlichen ganzzahligen Typen auf der FSL-Ebene zu spezifizieren, werden *Dependent Types* verwendet. Basierend auf diesen Typen wird ein generalisierbares Überlauf Erkennungs Schema vorgestellt, um beliebige arithmetische Überläufe zu erkennen. Der Einfluss des Schemas auf die Leistungsfähigkeit und den verbrauchten Platz der finalen Implementierung werden evaluiert, indem Implementierungen, die Operationen unter Verwendung dieses Schemas realisieren, mit solchen verglichen werden, die das nicht machen.

Um die generelle Leistungsfähigkeit von Implementierungen zu untersuchen, die von dem vorgestellten Ablauf synthetisiert werden, wird ein 32-bit MIPS Prozessor for-

mal spezifiziert und verifiziert. Die finale implementierung wird mit einem funktional äquivalenten Prozessor verglichen, der von einen Hardwarebeschleunigungssystem synthetisiert wurde, das auf dem neusten Stand ist. Dieser Vergleich zeigt das Potential des hier vorgestellten Synthese Ablaufs und öffnet die Tür für weitere Forschung zur Synthese von Hardware-Implementierungen die Korrektheitseigenschaften erfüllen und zusätzlich die Leistungsfähigkeit berücksichtigen.

# ACKNOWLEDGEMENTS

I want to thank Rolf Drechsler for making this dissertation possible and giving me the opportunity to present my ideas to the world and to work in an inspiring environment. I also want to thank Christoph Lüth and Robert Wille for offering me assistants whenever I needed it and for always being patient with me. Your help was invaluable and will never be forgotten.

Furthermore, I want to thank the entire University of Bremen's Group of Computer Architecture and the DFKI's CPS department for being friendly colleagues and giving me a good time. My special thanks go to Buse Ustaoglu, Christina Cociancig, Martin Ring, Saman Fröhlich, Pascal Pieper, Tobias Brandt, Rhea Rinaldo, and especially Christina Plump for giving feedback and for the countless and constructive discussions we had.

Finally, I want to thank my mother for letting me study, my brother, my roommates, and my friends who were there for me in my lazy times, and for the support you gave me during my studies and my doctorate.

*Do or do not. There is no try.*

— Master Yoda

# CONTENTS

Listings III
List of Figures VI
List of Tables VII
Acronyms IX
1 INTRODUCTION 1
1.1 Research Motivation 1
1.2 Research Topics and Questions 4
1.3 Research Contribution 8
1.4 Organization 12
1.5 Further Contributions 12
2 PRELIMINARIES 15
2.1 Proof Assistants 15
2.2 Dependent Types 20
2.3 High-level Synthesis Flows 22
2.3.1 SystemC 23
2.3.2 Acceleration-oriented Synthesis Flows 23
2.3.3 Functional Hardware Description Languages 24
3 PROPOSING AN AUTOMATIC HARDWARE DESIGN SYNTHESIS FLOW USING FORMAL SPECIFICATIONS 27
3.1 Introduction 27
3.2 Motivation 29
3.2.1 Introductory Example 29
3.2.2 Considered Problem 32
3.3 General Idea 34
3.3.1 Proof Assistant Based Hardware Design Flows 34
3.3.2 Functional Hardware Description Languages 36
3.3.3 Proposed Hardware Design Synthesis Flow 37
3.4 Specification of Hardware Designs 39
3.4.1 Finite Integer Types 39
3.4.2 Combinational Circuits 40
3.4.3 Synchronous Sequential Circuits 41
3.5 Synthesis of Hardware Design Specifications 51
3.5.1 Model Extraction from Specifications 51

3.5.2 Synthesis of Implementations 53
3.6 Evaluation 54
3.6.1 Implementation Effort 55
3.6.2 Quality of the Results 56
3.7 Scalability 57
3.8 Conclusion 58
4 DETECTING ARITHMETIC INTEGER OVERFLOWS IN FORMAL SPECIFICATIONS 61
4.1 Introduction 61
4.2 Motivation 62
4.2.1 Introductory Example 63
4.2.2 Considered Problem 67
4.3 Related Work 70
4.4 Specification of Arithmetic Integer Overflow Detection 72
4.5 Proving Properties about Arithmetic Integer Overflows 74
4.6 Proposed Integer Overflow Detection Scheme 83
4.6.1 Proving Properties 84
4.6.2 Exceptions 85
4.6.3 Closure of Functions 86
4.7 Evaluation 90
4.7.1 Integer Overflow Detection Specifications 90
4.7.2 Results 92
4.7.3 Discussion 93
4.8 Summary 94
5 CASE STUDY: PERFORMANCE ASPECTES OF A FORMALLY SPECIFIED 32-BIT MIPS PROCESSOR 97
5.1 Introduction 97
5.2 Motivation 99
5.2.1 The LegUp Synthesis Flow 99
5.2.2 Considered Problem 102
5.3 Correctness-oriented Synthesis Flows 103
5.4 Formal Specification and Verification of a 32-bit MIPS Processor 104
5.4.1 Specification of Sequential Hardware Designs 104
5.4.2 Construction of Instructions 105
5.4.3 Proving of Properties 108
5.5 Evaluation 117
5.5.1 Results 117
5.5.2 Discussion 119
5.6 Conclusion 120

6 CONCLUSION AND OUTLOOK 121
6.1 Conclusion 121
6.2 Outlook 124
Bibliography 125
A APPENDIX FOR CHAPTER 3 141
A.1 Traffic Light Controller Specification 141
A.2 Theorems about the Traffic Light Controller 142
B APPENDIX FOR CHAPTER 4 143
B.1 Lemmas and Theorems about Overflow Detection 143
C APPENDIX FOR CHAPTER 5 145
C.1 32-bit MIPS Processor Specification 145
C.2 Theorems proved by Custom Proof Methods 150
C.3 General Theorems 164

# LISTINGS


2.1 Formal proof about Nat in Gallina . . . 17
2.2 *head* function with dependent types . . . 21
2.3 Mac circuit in CλaSH . . . 25
3.1 Safety property for traffic light controller in OCL . . . 31
3.2 Tick function of the SystemC model . . . 33
3.3 Combinational MAC hardware design in Gallina . . . 40
3.4 Theorem and proof about the combinational MAC hardware design in Gallina and $\mathscr{L}_{tac}$ . . . 40
3.5 Specified Moore machine in Gallina . . . 42
3.6 Theorem and proof about the Moore machine in Gallina and $\mathscr{L}_{tac}$ using the empty list . . . 43
3.7 Theorem about the Moore machine in Gallina and $\mathscr{L}_{tac}$ using the non empty list . . . 43
3.8 Specified Mealy machine in Gallina . . . 44
3.9 Theorem and proof about the Mealy machine in Gallina and $\mathscr{L}_{tac}$ using the empty list . . . 45
3.10 Theorem and proof about the Mealy machine in Gallina and $\mathscr{L}_{tac}$ using the non empty list . . . 46
3.11 Specification of the traffic light controller in Gallina. . . . 48
3.12 Theorem and proof in Gallina and $\mathscr{L}_{tac}$ representing the transformed safety property . . . 49
3.13 Automatic extracted CλaSH model . . . 52
4.1 OCL constraints for the traffic light controller . . . 65
4.2 Tick function of the SystemC model . . . 66
4.3 Safety property in OCL for the traffic light controller . . . 67
4.4 Inductive option type in Gallina . . . 72
4.5 Safe_mult function in Gallina . . . 74
4.6 Theorem and proof about multiplication in $\mathbb{Z}$ maps multiplication in $\mathbb{Z}_{>-1}/2^{32}\mathbb{Z}$. . . . 74
4.7 Theorem and proof about the safe_mult function in Gallina and $\mathscr{L}_{tac}$ detecting the overflow. . . . 76
4.8 Theorem and proof about the safe_mult function in Gallina and $\mathscr{L}_{tac}$ showing the result of the arithmetic operation. . . . 77

4.9 Safe_add function in Gallina . . . 78
4.10 Traffic light controller in Gallina . . . 78
4.11 Safety property about the traffic light controller in Gallina and $\mathscr{L}_{tac}$ . . 80
4.12 Safety property about the traffic light controller in Gallina and $\mathscr{L}_{tac}$ detecting the overflow . . . 82
4.13 Proposed overflow detection scheme . . . 83
4.14 Functions of the option monad in Gallina . . . 87
4.15 Theorem and proof about the option monad Gallina and $\mathscr{L}_{tac}$ . . . 88
4.16 Theorem and proof about the option monad in Gallina and $\mathscr{L}_{tac}$ . . . 89
4.17 Specified safe_mult_signed function in Gallina . . . 91
4.18 Specified safe_add_signed function in Gallina . . . 92
5.1 Extract of the 32-bit Microprocessor without Interlocked Pipeline Stages (MIPS) processor model . . . 101
5.2 Function type of the mealy function specified in Gallina . . . 104
5.3 Function type of the mips function in Gallina . . . 105
5.4 Extract of the 32-bit MIPS processor specification in Gallina . . . 107
5.5 Theorem and proof about the ADDU instruction in Gallina and $\mathscr{L}_{tac}$ . 109
5.6 proveRFormat tactic in $\mathscr{L}_{tac}$ . . . 111
5.7 Theorem and proof about the ADDIU instruction in Gallina and $\mathscr{L}_{tac}$ . 112
5.8 proveIFormat tactic in $\mathscr{L}_{tac}$ . . . 113
5.9 Theorem and proof about the J instruction in Gallina and $\mathscr{L}_{tac}$ . . . 114
5.10 proveJFormat tactic in $\mathscr{L}_{tac}$ . . . 115
5.11 Theorem and proof about the NOP instruction in Gallina and $\mathscr{L}_{tac}$ . . 116
A.1 Theorem and proof about all traffic lights . . . 141
A.2 Theorem and proof about all traffic lights . . . 142
B.1 Theorem about type conversion from Z to finite integer . . . 143
B.2 Lemmas for proving the *multUnsigned32mapsMultZ* theorem. . . . 143
B.3 Theorems including proofs for the safe_add function. . . . 143
C.1 32-bit MIPS specification . . . 145
C.2 Theorems and proofs about the MIPS processor's R Format instructions 150
C.3 Theorems and proofs about the MIPS processor's I Format instructions 158
C.4 Theorems and proofs about the MIPS processor's J Format instructions 163
C.5 Theorems and proofs about the MIPS processor . . . 164

# List of Figures


Figure 1.1 Sketched established correctness-oriented and HLS flows 5
Figure 1.2 The contribution's relation 9
Figure 2.1 Sketched theorem proving flow using proof assistants 16
Figure 2.2 Sketched High-Level Synthesis flow 22
Figure 3.1 Sketched established hardware design flow 30
Figure 3.2 SysML traffic light controller 31
Figure 3.3 Sketched hardware design synthesis flow using proof assistants 36
Figure 3.4 Sketched hardware design synthesis flows using functional HDLs 37
Figure 3.5 Proposed hardware design synthesis flow 38
Figure 4.1 Sketched semantic gap in the established correctness-oriented hardware design flow 63
Figure 4.2 SysML class diagram of the traffic light controller 64
Figure 5.1 Sketched LegUp synthesis flow for pure hardware designs 100
Figure 6.1 Addressed established hardware design and HLS flows problems 122

# List of Tables

Table 3.1 Benchmark evaluation comparing consumed space and maximum clock frequency 57
Table 4.1 Benchmark evaluation comparing consumed space and maximim clock frequency 93
Table 5.1 Benchmark evaluation comparing consumed space and performance 118

# Acronyms

**CiC** Calculus of Inductive Constructions. 8, 15, 37
**CNF** Conjunctive Normal Form. 3, 13
**CVE** Common Vulnerabilities and Exposures. 6
**CWE** Common Weakness Enumeration. 6

**DSL** Domain-Specific Language. 4, 97

**ESL** Electronic System Level. 3–5, 7, 10, 22, 23, 27, 28, 30, 32, 34, 37, 39, 56–59, 71, 97, 100, 102, 121, 122

**FHDL** Functional HDL. 36, 37
**FPGA** Field-Programmable Gate Array. 30, 54–56, 59, 90, 94, 95, 100, 103, 117, 118
**FSL** Formal Specification Level. 1, 4–8, 10, 21, 27, 28, 32, 34, 37, 39, 51, 56, 58, 59, 61, 62, 68, 69, 72, 83, 94, 98, 103, 121–124
**FSM** Finite State Machine. 31, 64

**HDL** Hardware Description Language. 24, 28, 30, 37, 53, 56, 98, 123, 125
**HDSL** Hardware DSL. 28, 35, 36, 119
**HLS** High-Level Synthesis. 3–5, 7, 15, 22–24, 97–99, 117, 121–123

**ISA** Instruction Set Architecture. 6, 99, 124

**LLVM** Low Level Virtual Machine. 99, 102

**MIPS** Microprocessor without Interlocked Pipeline Stages. VI, 97–106, 117
**MPX** Memory Protection Extensions. 6

**OCL** Object Constraint Language. 4–6, 8, 10, 13, 27–33, 36, 55, 58, 61–68, 70, 71, 78, 79, 83, 94, 121–123

**RISC** Reduced Instruction Set Computer. 99
**RTL** Register-Transfer Level. 3–5, 7, 8, 10, 22, 25, 27–30, 32, 34–39, 51, 54, 56–59, 61, 93, 98–100, 102, 103, 117–122

**SAT** Boolean Satisfiability Problem. 2, 3, 12
**SMT** Satisfiability Modulo Theories. 2, 3, 58

**SysML** System Modelling Language. 4–6, 8, 10, 13, 27–33, 36, 55, 58, 61–64, 66–71, 79, 83, 94, 121–123

**UML** Unified Modeling Language. 4

# 1

# INTRODUCTION

## 1.1 RESEARCH MOTIVATION

Over the last few decades, electronic circuits have more and more become a part of our lives. Their area of application extends from cars over medicine to toothbrushes. These circuits are synthesized from hardware designs that describe their functional or timing behavior, e.g., the behavior of a car's control unit that triggers the airbag in the case of an accident. This large number of different application areas leads to an ever-growing complexity of hardware designs since designers need to adapt these designs to current developments, e.g., an increasing number of sensors in a modern car. However, the increasing complexity escalates the number of potential failures in the design and subsequently in the synthesized circuit. Errors in a deployed circuit might lead to severe problems and often result in high costs for the manufacturer to fix them. Furthermore, the increasing complexity directly impacts the safety of hardware designs. In 1996, the Ariane-5 self-destructed during its maiden flight because of an arithmetic integer overflow in the flight control system. An arithmetic integer overflow occurs when the result of an arithmetic integer operation is too big for being represented by the underlying integer type. These overflows are still under current research [DLRA15, MGE17, MMS+18].

Options to address the hardware design's increasing complexity include their consideration at a higher abstraction level for a better design understanding, e.g., at the Formal Specification Level (FSL) [DSW12]. This level allows the formal specification and verification of correctness properties. A hardware design's formal specification hides implementation details, such as the clock frequency, to understand the design's behavior better. A formal specification allows the *formal verification* of correctness properties, e.g., to guarantee that the car's airbags are triggered if and only if in the case of an accident. Formal verification proves that a design satisfies its requirements (properties) using formal methods of mathematics.

Simultaneously, as the complexity increases, the need for ever-faster circuits also increases. The large amount of data gathered by sensors, e.g., in a modern car, requires fast processing. Imagine an autonomous car that has to keep track of the current speed,

possible obstacles, or the vehicles' distance in front. A quick evaluation of all data gathered by the different sensors is essential to react to these different circumstances. The need for ever-faster circuits results in a *trade-off* between correctness-oriented and performance-oriented hardware designs.

On the one hand, circuits need to satisfy correctness properties, which one might expect to harm performance. These correctness properties satisfaction involves the execution of additional functionality, e.g., the validation of input values. On the other hand, the large amount of data that needs to be processed, as mentioned above, requires a performance-oriented circuit.

Much work has been done in the areas of formal specification and verification of hardware designs [PSC+06, LFD+19, CP88, Gup92, KG99, LFD+19]. The foundation of formal verification is mathematical reasoning [SH13, HT15] contrasting traditional verification techniques like simulation-based verification [KC11], which uses stimuli generation [WGHD09, NRJ+07]. Stimuli are input assignments generated to try to ensure that the final implementation corresponds to its specification. The hardware design processes these input assignments, and every assignment's output is compared with the expected value in a simulation process. Confidence in the correspondence between specification and implementation increases with the number of stimuli applied to the hardware design during the simulation process. In general, the application of all possible stimuli (exhaustive testing) is not feasible as the number of potential stimuli is too large due to the resulting state space explosion [Val96]. As a result, simulation-based verification attempts to detect as many defects in a design as reasonably possible. With mathematical reasoning, on the other hand, e.g., proof by induction, the entire search space is covered and, thus, it is guaranteed that a design is free of defects, i.e., formally verified. Theorem provers implement formal verification and range from fully-automated ones to interactives [ES03, dMB08, BC04, NPW02]. Interactive theorem provers also called proof assistants, help the designer to find a formal proof for a given verification task by human-machine collaboration. In contrast, automatic theorem provers try to find the formal proof themselves. In general, finding formal proofs is challenging, whether the process is interactive or automatic [Chl13]. Due to their high automation level, automatic theorem provers, also called solvers, like Z3 [dMB08], Yices [Dut14], or MiniSAT [ES03], are widespread to formally verify hardware designs in both academia and industry [GLH18]. The goal of automatic theorem provers is to find a solution for a given problem or prove that no solution exists by automated reasoning. More precisely, the given problem represents a decision problem and is described either as an Satisfiability Modulo Theories (SMT) instance or as an Boolean Satisfiability Problem (SAT) instance. An SMT instance describes the decision problem by decidable background theories using a subset of classical first-order logic, e.g., integer numbers or the theory of data structures such as lists

or arrays [GdM09, dMB11]. A SAT instance, on the other hand, describes the decision problem by propositional logic in Conjunctive Normal Form (CNF) [MMZ+01]. A verification task is often reduced to an SMT or SAT instance, and the automatic prover tries to solve the represented decision problem subsequently [dMB11]. However, the decision problems represented by SMT or SAT formulas are NP-complete, meaning that finding a solution that solves a verification task might take much time, depending on the given task [CES+09]. This underlying NP-completeness limits the application of automatic theorem provers. In contrast to automated theorem provers, proof assistants like Coq [BC04, Chl13, SBF+20] or Isabelle/HOL [NPW02] use higher-order logic to specify a functional behavior and formally verify it. Since higher-order logic is too expressive for automatic reasoning, the designer guides the proof assistant manually through the proof, hence the term human-machine collaboration. Proof assistants prove sophisticated verification tasks, such as the four-color theorem [Xia09] or the Kepler conjecture [HAB+15], due to higher-order logic usage. They address the limitations of automatic theorem provers regarding finding proof for a given verification task by relying on humans' interaction to find it. Since higher-order logic is more expressive than propositional logic or first-order logic used by SAT and SMT solvers, proof assistants are more generally applicable. This generalizability indeed comes with the price of a lower automation level, but the formal verification of hardware designs using proof assistants gains more and more attention in academia [CVS+17, KSHY19, FSS15, KKN+20].

Coming back to the synthesis of circuits from hardware designs mentioned above. The High-Level Synthesis (HLS) design flow has been proposed [NSP+16, HTHT09, Tak16, CGMT09, GR94, GDWL92, HLW+20] to synthesize hardware designs described at a higher abstraction level. This design flow describes the automatic synthesizes of hardware design models realized at the Electronic System Level (ESL) [MBP07] into implementations at the Register-Transfer Level (RTL), e.g., in Verilog. Realizing hardware designs at the ESL level allows design space exploration [CGMT09], the synthesis of performance-oriented implementations [HLW+20], the realization of virtual prototypes [HGPD20], and Hardware/Software CoDesign [Tei12].

Different HLS flows have been proposed, such as SystemC design synthesis [Arn00, Gro02], LegUp [CCA+13], Bambu [PF13], DWARV [NSO+12]. SystemC is a C++ class library and models hardware designs, e.g., virtual prototypes, for early testing or verifying particular properties [Mar03, HGLD18, GD19, HGW+20]. It is also used in the Hardware/Software Codesign development process [Tei12]. Apart from these, SystemC provides a synthesizable but restricted subset [Inc16, FHT06, SWD13]. A hardware model realized in SystemC following the subset guidelines can be synthesized into an implementation, e.g., in Verilog [CHM07]. The subset prohibits the synthesis of hardware design models in general due to its restriction. The HLS flow implemented

by LegUp, Bambu, and DWARV, are focused on hardware acceleration. In contrast to SystemC, these frameworks use a Domain-Specific Language (DSL) embedded into the C programming language. The acceleration is achieved by adding performance optimizations like functional or loop pipelining [CCA+13, HHL91]. These optimizations are added automatically by analyzing the given hardware design before synthesizing it. However, the correctness of hardware designs synthesized by following the HLS flow is still important.

Already proposed iterative methods combine formal specification and verification of hardware designs with the HLS flow. These methods start with a specification of a hardware design at the FSL level [DSW12], e.g., using the System Modelling Language (SysML) [Obj15] and the Object Constraint Language (OCL) [Obj12]. SysML is an Unified Modeling Language (UML) [Obj17] profile designed for systems engineering applications. UML's profile mechanism allows the generic extension of UML models for a particular domain. SysML class diagrams specify the design's structure, while OCL constraints specify the design's functional behavior. OCL constraints consist of preconditions, postconditions, and invariants and can be verified by automatic theorem provers [BCBO18, PWPD18]. After verifying the specification, the next step is to realize an ESL. SystemC is the "de-facto-standard" for describing a hardware design at this level [BDBK09]. While the SysML class structure automatically translates into a SystemC class structure, the OCL constraints do not automatically translate into executable SystemC code. Since an automatic translation process is missing, a manual realization of the model is required. Due to the restricted synthesizable subset, the RTL level implementation reimplements the SystemC model in general, as mentioned above. Since the translation process from the specification to the model and finally to the implementation is not fully automatable, generated test benches ensure the correct behavior of both the model and the implementation [CMK12, WGHD09, WPK+16]. However, both correctness and performance of electronic circuits are essential to address the needs required by these circuit's different application areas we have today.

## 1.2 Research Topics and Questions

This section describes the research topics of this dissertation. For this reason, Figure 1.1 shows the sketched established correctness-oriented hardware design flow, which uses SysML/OCL and SystemC and the established HLS flows described in the previous section. These established flow's problems are discussed in detail resulting in the research questions addressed in this dissertation, using this figure.

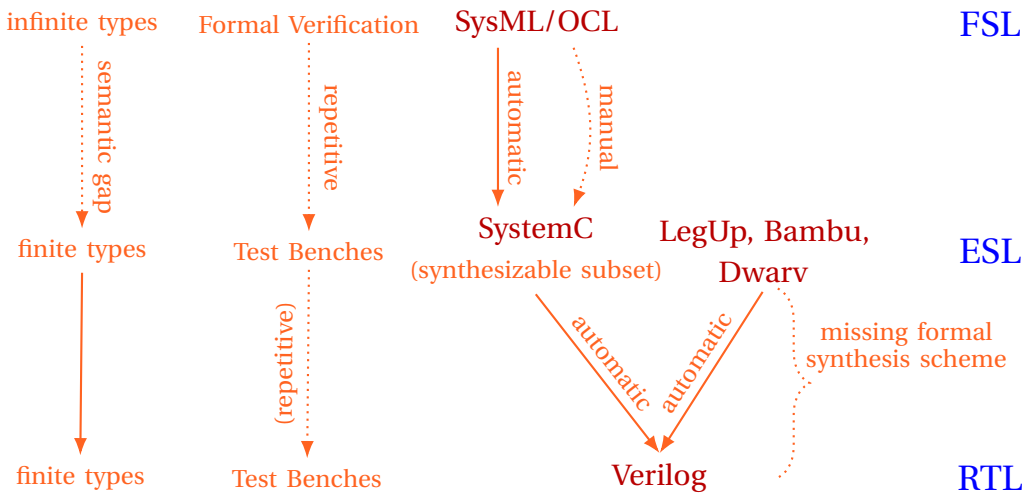


**Figure 1.1 :** *Sketched established correctness-oriented hardware design and HLS flow. The dotted lines indicate the problems addressed in this dissertation. The established correctness-oriented hardware design flow specifies infinite integer types at the FSL level but finite at the ESL and RTL level. Verification results cannot be propagated automatically, due to the manual realization of the SystemC model. Note that the test benches at the RTL level are* only *required if the SystemC model is manually translated to an RTL implementation because of the restricted synthesizable subset. The LegUp, Bambu, and DWARV frameworks provide a larger synthesizable subset because they aim to accelerate hardware design implementations in general. The established HLS flows do not provide a formal synthesis scheme.*

This dissertation's first research topic investigates the established correctness-oriented hardware design flow's missing propagation of verification results from the formal specification to the final implementation. To ensure the correct behavior of these realizations test bench generation is required. In general, test benches are required for SystemC models due to the restricted synthesizable subset. Hence, the model's and implementation's correct behavior depends on the test benche's quality rather than on the formal specification's verification results. Since exhaustive testing is not feasible, this is a problem for circuits deployed in safety-critical systems. These circuits must satisfy correctness properties that guarantee their formally specified behavior.

The second research topic investigates the established hardware design flow's *semantic gap* between the SysML/OCL specification and the SystemC model regarding the represented integer types at both levels. This semantic gap results in properties that hold for the specification but not for the model and the implementation. The

SysML/OCL specification describes infinite integer types, e.g., the type *Int*, which models the mathematical type $\mathbb{Z}$. However, the SystemC model describes finite integer types, e.g., the type *sc_int<32>*, representing the quotient ring $\mathbb{Z}/2^{32}\mathbb{Z}$. Operations defined on these types are not semantically equivalent, i.e., they do not share the same semantic behavior. For example, there is no equivalence relation between the addition operation defined for $\mathbb{Z}$ and $\mathbb{Z}/m\mathbb{Z}$, where $m \in \mathbb{Z}_{>-1}$, as $\mathbb{Z}/m\mathbb{Z}$ models a wrap-around behavior which $\mathbb{Z}$ does not. This missing equivalence relation between infinite and finite integer operations is a problem since properties formally verified for the specification do not necessarily hold for the model and implementation. As a result, problems related to finite integer types cannot be addressed in the SysML/OCL specification because it models infinite integer types [Obj15].

This dissertation investigates the semantic gap between the different integer type representations for two reasons:

1. Arithmetic integer overflows may lead to severe problems. For example, the self-destruction of the Ariane-5 rocket, mentioned above, was caused by a hidden integer overflow. This example shows that these overflows might lead to fatal consequences in safety-critical systems, and their detection is necessary to appropriately respond to them. The detection of integer overflows is still part of current research, as it is quite challenging to detect them, both manually and automatically [DLRA15]. According to the Common Weakness Enumeration (CWE)[1], integer overflows and integer wrap-around behavior are still under the "Top 25 Most Dangerous Software Weaknesses" [Tea]. These behaviors are often addressed in software, but their roots are in the description of finite integer types in hardware. For this reason, it makes it comprehensible to detect arithmetic integer overflows and integer's wrap-around behavior directly in hardware designs rather than letting the software detecting them. Furthermore, there has been an effort to detect errors directly in hardware that are normally handled in software. In 2015, Intel® proposed an extension for there Instruction Set Architecture (ISA) called Memory Protection Extensions (MPX), to detect malicious pointer references [OKB+18].

2. Investigating this semantic gap might lead to the specification of machine integers at the FSL level. Machine integers are integer values represented as finite bit-vectors by a computer, allowing them to perform arithmetic and logical operations. Non-trivial hardware designs, such as processors [HTHT09], are described using these types, representing their instructions. Having these integers

[1] CWE contains a community-maintained list of software and hardware weaknesses published by the MITRE corporation, which also maintains the Common Vulnerabilities and Exposures (CVE). The foundation of this list are the CVE's: https://cve.mitre.org/.

at the FSL level allows the formal specification and verification and maybe even synthesizing these non-trivial hardware designs to an RTL implementation.

The problem of arithmetic integer overflows, the integer's wrap-around behavior and the widespread application of machine integers motivates the semantic gap's investigation between integer representations, rather than investigating the semantic gap between other data types, e.g., floats.

The third and final research topic investigates the established HLS flow's missing formal synthesis schemes [Gro02, CCF+16, NSO+12, PF13]. A synthesis scheme defines how to translate a term of the model into terms of the implementation [EK95, BK13]. This scheme ensures the correct propagation of the semantics from the model to the implementation. If such a synthesis scheme is missing, it is unclear whether the model's semantics are propagated correctly to the implementation and how to track and verify properties denoted at the FSL level in the RTL implementation.

This dissertation aims to answer three research questions based on the research topics discussed above.

1. *Can an alternative hardware design flow to the established one be developed to propagate verification results automatically?* As discussed above, the established hardware design flow's fundamental problem is the missing propagation of verification results. An alternative hardware design flow that allows this propagation would avoid the generation of test benches for the model and the implementation. Properties verified at the FSL level also hold at the ESL level and the RTL level. Furthermore, the model and the implementation must not be reimplemented manually, which is also an error-prone process. The avoidance of both the test bench generation and the model's and implementation's manual reimplementation also accelerates the deployment of hardware designs.

2. *If the semantic gap can be closed, can the alternative hardware design flow be used to address low-level problems such as arithmetic integer overflows at the FSL level?* The established hardware design flow describes infinite integer types at the FSL level and finite ones at the ESL level, as discussed above. Since integer overflows might lead to severe problems, they should be detected early in the hardware design process. If these overflows could be detected at the FSL level, properties about them can be formally verified and propagated to the RTL implementation leveraging the alternative hardware design flow.

3. *Can the alternative hardware design flow be used to gauge the trade-off between performance-oriented and correctness-oriented hardware designs?* As discussed above, in general, there is always a trade-off between correctness-oriented and

performance-oriented synthesized implementations. Since there is no detailed quantitative analysis available yet that addresses this trade-off, investigating it helps the designer gauge the impact of correctness-oriented synthesis flows on the implementation's performance.

## 1.3 RESEARCH CONTRIBUTION

This section briefly presents the contributions that address the research questions asked in the previous section. It shows how the individual contributions are related and which contribution addresses which question and how. Figure 1.2 shows the relation between the individual contributions and how they are related to each other. The frame around the second and third contributions indicates that these two address one research question.

1. The contribution published in [BLWD20c] is the presentation and evaluation of an alternative hardware design flow that synthesizes a hardware design's formal specification automatically in an implementation at the RTL level. It addresses the first research question, asked in Section 1.1. In contrast to the established hardware design flow described in the previous section, which starts with a formal specification in SysML/OCL, the proposed synthesis flow starts with a formal specification for the proof assistant Coq [BC04, Chl13].

   Coq describes a functional behavior in the Calculus of Inductive Constructions (CiC). This calculus combines higher-order logic with a richly-typed functional programming language. To perform formal verification, Coq provides interactive proof methods, decision, and semi-decision algorithms [BC04]. A built-in tactic language allows user-defined proof methods [Del00]. Coq's abstraction level allows proving properties about mathematical objects. Therefore, integer types in Coq model infinite types, such as $\mathbb{Z}$ or $\mathbb{N}$. In contrast, hardware designs rely on finite data types as these types cannot expand once synthesized in hardware. To address the problem of describing finite integer types in Coq, the CompCert integer library is leveraged [LBK+16]. This library allows the specification of finite integer types of arbitrary sizes and provides mathematical operations like basic arithmetic operations. The specification of finite integer types at the FSL level closes the semantic gap between infinite and finite integer types, the established hardware design flow has.

   After verifying the specification, an executable CλaSH model [BKK+10] is automatically extracted from that specification and subsequently synthesized in an

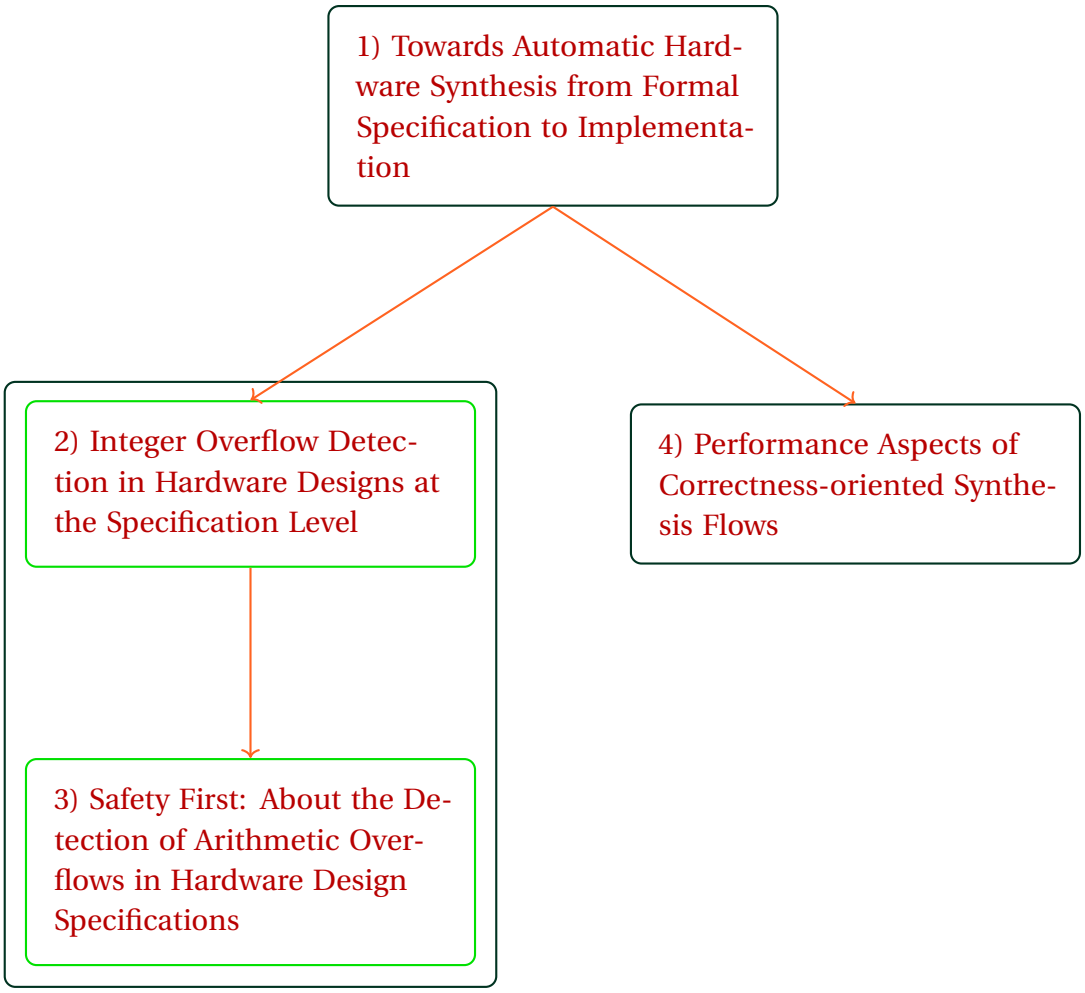


**Figure 1.2 :** *The first contribution proposes an automatic hardware design synthesis flow based on the proof assistant Coq and the functional hardware description language CλaSH. The second contribution uses the proposed synthesis flow as its foundation to formally specify and verify an arithmetic overflow detection scheme. Build on this; the third contribution extends the detection scheme by proposing a method to cascade overflow detecting operations and presents an impact analysis of the scheme. The fourth contribution also uses the proposed synthesis flow as its foundation and uses it to formally specify and verify, and subsequently synthesize a 32-bit MIPS processor.*

implementation, e.g., in VHDL [CS19], Verilog [CS05], or SystemVerilog [CS18] by the CλaSH compiler. This contribution extends Coq's extraction backend by CλaSH to automatically extract the model. Coq's extraction backend already provides a software model's extraction in a functional language, e.g., in Haskell or Ocaml, of a formally verified specification [Chl13]. The synthesis of a model in an implementation is not part of this contribution as the CλaSH compiler already provides it.

The combination of Coq and CλaSH allows the specification of both combinational and synchronous sequential hardware designs and the automatic propagation of the semantics from the specification down to the final implementation. This propagation ensures that properties verified on the FSL level hold for the model and the implementation. Different specified benchmarks for both combinational and sequential hardware designs evaluated the presented synthesis flow.

2. The contribution published in [BLWD20a] is the presentation and evaluation of an arithmetic integer overflow detection scheme using the first contribution as its foundation. It addresses the second research question, asked in Section 1.1. The semantic gap regarding the integer types between the SysML/OCL specification and the SystemC model, as described in Section 1.1, motivates this contribution. By leveraging the CompCert library, the presented synthesis flow addresses problems regarding finite integer types already at the FSL level. In contrast to the established flow where those problems first occur at the ESL level. Arithmetic overflows on finite integer types are one of those problems [DLRA15, CKK+12, CH13].

   A generalizable arithmetic overflow scheme is described at the FSL level using the CompCert integer library. This scheme enables the detection of arithmetic overflows already at this level. Using the proposed synthesis flow, verification results about these overflows propagate to the final implementation. The proposed detection scheme allows the unique distinction between an occurred overflow and the result of an arithmetic operation. A dedicated data type that defines only two constructors: one for the occurred overflow and one for the operation's result ensures this distinction. Formally verified properties about functions that use this data type show that an overflow is detected. Benefit from the above-proposed synthesis flow, these properties hold for the RTL implementation as well.

3. The contribution published in [BLWD20b] extends the second contribution's proposed overflow detection scheme. Due to specified boundaries, there are

cases where that scheme is not necessary as an overflow cannot occur. This contribution considers such cases and proposes a theorem that if it is proven, the arithmetic operation can be used safely, i.e, without considering an overflow. Since the proposed overflow detection scheme requires a different function type, i.e., the two operands have a different type than the return value, this contribution also proposes a method to close operations implementing this type. Furthermore, the proposed detection scheme was applied to different arithmetic operations with different integer types to evaluate this scheme's impact on the final implementations. The impact regarding the consumed space and maximum clock frequency was investigated by synthesizing a hardware design using basic arithmetic operations on one side and their overflow detecting counterparts on the other side.

4. The contribution published in [BLWD21] is the presentation of a case study. This case study compares two 32-bit MIPS processor implementations regarding their performance [HTHT09]. The first implementation was formally specified, verified, and subsequently synthesized using the proposed hardware design synthesis flow mentioned above. The second implementation was synthesized from a model by the state-of-the-art hardware acceleration framework LegUp [CCA+13, CCF+16]. This contribution addresses the third research question, asked in Section 1.1, and considers the *trade-off* between correctness-oriented and performance-oriented hardware design synthesis flows. LegUp was chosen as a representative of hardware acceleration frameworks as it synthesizes, on average, the most performance-oriented implementations [NSP+16]. The case study's evaluation shows that LegUp synthesizes the faster implementations, but the implementations synthesized by the proposed synthesis flow are comparable. As a result, it gives a first impression how to gauge that trade-off and shows the proposed synthesis flow's potential when the need for correct and fast synthesized hardware designs arises.

These four contributions are the foundation of this dissertation. The proposed hardware design synthesis flow addresses different problems regarding the formal verification of hardware designs. Addressing these problems shows the flow's potential for further research in synthesizing formally verified hardware designs, which also consider performance.

## 1.4 Organization

This dissertation is structured as follows: Chapter 2 describes the preliminaries necessary to follow and understand the individual chapters in detail. Chapter 3 describes the automatic hardware design synthesis flow. The foundation of this chapter is the first contribution, described in Section 1.3. Chapter 4 describes the formal specification and verification of the detection scheme proposed for arithmetic integer overflows after introducing the synthesis flow. The foundation of this chapter is the second and third contributions described in Section 1.3. Chapter 5 describes the case study that compares a specified, verified, and finally synthesized 32-bit MIPS processor with an implementation of the same processor provided by the state-of-the-art hardware acceleration synthesis framework LegUp. The foundation of this chapter is the fourth and final contribution, described in Section 1.3. Chapter 6 summarizes and concludes the entire dissertation and gives an outlook on further research topics.

## 1.5 Further Contributions

During this dissertation, further contributions have been published. Since these contributions are outside the scope of this dissertation, they are only mentioned briefly.

1. The publication [BWD17] is about the analysis of techniques implemented by modern SAT solvers in the context of self-verification. It evaluates the development of lightweight SAT solvers, focusing on memory consumption. First, it discusses which core techniques of modern SAT solvers impact total memory consumption. Based on that, different experimentally evaluated configurations were defined. The results clearly show that disabling the learning of conflict clauses yields the best solution concerning memory requirements. Simultaneously, the runtime performance is not entirely unacceptable – although a significant amount of instances could not be solved anymore when conflict clauses are deactivated. Overall, this motivates to re-think established SAT solving strategies for applications such as self-verification, which do not necessarily rely on the best possible runtime performance but may have severe hardware limitations. Future work focuses on developing SAT solvers and verification methods following this direction. However, more detailed evaluations shall be conducted first before considering more sophisticated configurations and a wider variety of instances.

2. The publication [RBL+19] is about the verification of embedded systems *after* deployment. The partitioned system search space consists of one part, which infrequently changes (the *configuration variables*), and one that frequently does. By conducting the verification with the configuration variables set to their actual values, the search space reduces drastically, making verification feasible even on an embedded system's limited resources. The basic idea is independent of the specific modeling languages (SysML, OCL, CλaSH) and prover tools (Yices, MiniSat) used for evaluation. As long as the verification conditions can be translated into a format where the variables can be tracked to be instantiated, which is suitable to automatic proof (CNF in this case), the approach is viable and competitive because of the exponential reduction of the search space. For example, on a more high-powered system, it might be able to prove the instantiated verification conditions in bit-vector logic rather than CNF. The verification of bit-vector logic will enable the verification of even more sophisticated systems. Note that the verification flow proposed here does not *replace* the existing verification flow; it *enhances* it. All well-known powerful tools may be used at design time to prove verification conditions as before, and still use verification after deployment to tackle the verification conditions we could not prove during design — combining the best of both worlds. Overall, the evaluations and discussions show that, following the proposed idea, a novel verification methodology does not necessarily need to rely on incremental improvement of existing tools and methods but tackles complexity from a wholly different and more successful angle.

3. This publication is about partial verification [RBL+20]. In this work, a complementary approach to tackle the verification gap is proposed. Instead of aborting the entire verification process and getting no result due to time limitations, it is proposed to set some variables to a fixed value to get at least a partial verification result. This publication proposed a systematic verification runtime analysis that shows *how many* and *which* variables to fix for maximum verification runtime reduction. Experimental evaluations based on a proof-of-concept implementation confirmed the potential. These evaluations demonstrated that the proposed analysis method does not only yield a partial verification result, but also gets the most out of it. Considering that further analysis methods can be implemented on top of this methodology, this publication provides a promising basis for future work in this direction as an alternative to existing verification methods.

# 2

# PRELIMINARIES

This chapter describes the foundations to understand and follow the further chapters of this work. Section 2.1 describes how *proof assistants* [YD19, Geu09a] specify and verify a particular behavior. This chapter's focus is on proof assistants that use higher-order logic like Coq [BC04, Chl13] or Isabelle/HOL [NPW02]. It does not consider proof assistants like ACL2 [NK09] that use first-order logic. Section 2.2 describes *dependent types* [BG01, Hof97]. These types specify the fixed-size vectors used in hardware designs. Some proof assistants like Coq allow the specification of user-defined dependent types of arbitrary sizes [Chl13]. Finally, Section 2.3 describes the HLS [NSP+16, Tak16, CGMT09, GR94, GDWL92, HLW+20] synthesis flow. The focus is on the SystemC design synthesis flow [Gro02, Mar03] as the "de-facto" HLS standard [BDBK09], acceleration-oriented flows implemented by LegUp [CCA+13], Bambu [PF13], and DWARV [NSO+12] and flows that rely on functional hardware description languages like CλaSH [BKK+10].

## 2.1 PROOF ASSISTANTS

Proof assistants formally specify a functional behavior to prove properties specified as theorems or lemmas [1] by human-machine collaboration [Geu09b]. They mostly use higher-order logic for specification and verification, e.g., the well-known proof assistants Coq [BC04] or Isabelle/HOL [NPW02]. Proof Assistants like ACL2 [KM96], which uses first-order logic, are not considered in this dissertation. A proposition $\phi$ is formally proven if and only if $\phi$ is derivable in the proof assistant's logic, e.g., CiC, which is the formal language Coq is based on. CiC combines higher-order logic with a richly-typed functional programming language. As higher-order logic is too expressive for automated theorem proving, the proof assistant is guided manually through the proof process by tactics. Tactics realize proof methods [Del00, Wen02] that

[1] Theorems and lemmas have no semantic difference in Coq. The reason to call a proposition a theorem or a lemma is purely psychological and reflect the designers opinion about a particular proposition. In general, a theorem is a mathematical proposition that is to be proven by mathematical reasoning. A lemma is a minor result that helps to prove a theorem.

allow the proof assistant to construct a formal proof. Figure 2.1 sketches the formal specification and verification using proof assistants.

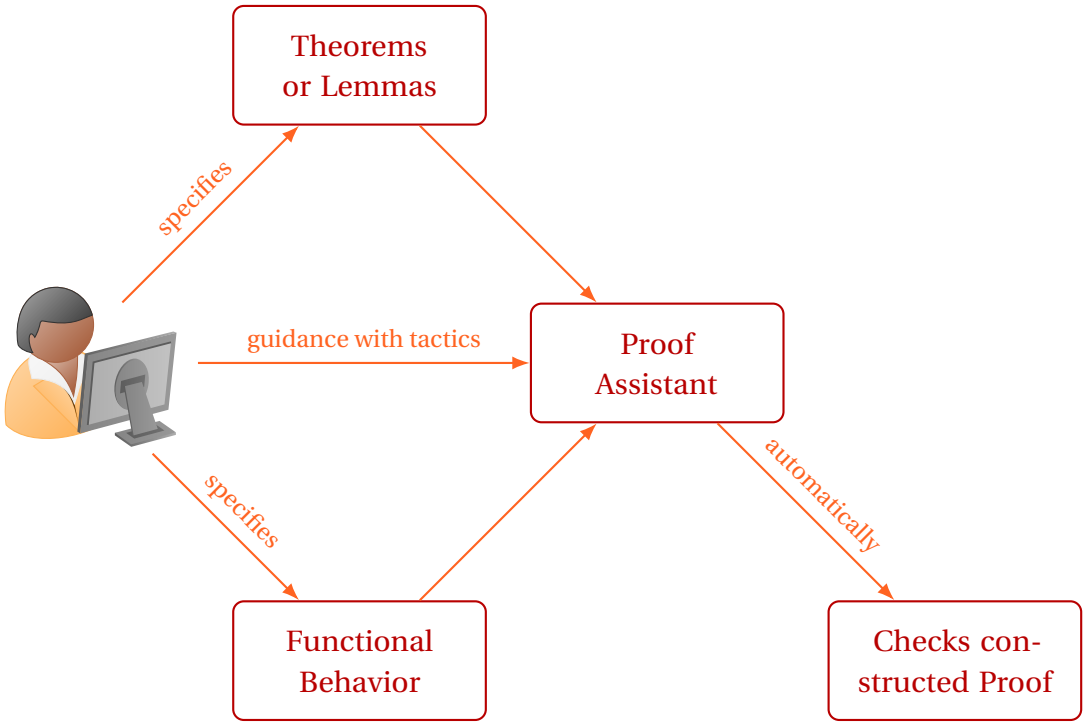


**Figure 2.1 :** *Sketched theorem proving flow using proof assistants. The user specifies the theorems or lemmas, and the functional behavior formally in the proof assistant's logic. Afterward, the user guides the proof assistant through the proof using a collection of tactics, which constructs the formal proof. This constructed proof is machine-checked by the proof assistant.*

Having a formal proof that is machine-checked by a computer, one question remains: *how can that proof be trusted?* To answer this question, we look at *how* a formal proof is machine-checked by the proof assistant. Proof assistants like Isabelle/HOL and Coq use a formal system called type theory as their mathematical foundation [NPW02, BC04]. In this theory, propositions are represented as types as they are essentially the same – *propositions as types* paradigm [Wad15]. The proof of a proposition is the same as constructing a term of the proposition's type in type theory. Listing 2.1 shows the application of this paradigm in Coq. Coq is used as an example since this dissertation utilizes this proof assistant. It provides a formal specification language called Gallina and a tactic language called $\mathscr{L}_{tac}$.

```
1  Inductive nat : Set :=
2    | O : nat
3    | S : nat -> nat.
4
5  Theorem addAssoc :
6    forall x y z: nat, (x + y) + z = x + (y + z).
7  Proof.
8    intros.
9    induction x.
10   - reflexivity.
11   - simpl.
12     rewrite IHx.
13     reflexivity.
14 Qed.
15
16 Print addAssoc.
17 (* addAssoc =
18 fun x y z : nat =>
19 nat_ind (fun x0 : nat => x0 + y + z = x0 + (y + z))
20         eq_refl
21         (fun (x0 : nat) (IHx : x0 + y + z = x0 + (y + z)) =>
22            eq_ind_r (fun n : nat => S n = S (x0 + (y + z)))
23            eq_refl IHx
24         ) x
25      : forall x y z : nat, x + y + z = x + (y + z)
26 *)
```

**Listing 2.1 :** *Formal proof about the type* nat*, which models the type* $\mathbb{N}$*, provided by Coq's standard library – Coq.Init.Datatypes. The Theorem* addAssoc *denotes that addition for* nat *is assoziative. The* Print *command prints the given object.*

The type *nat* provided by Coq's standard library is used to demonstrate the *propositions as types* paradigm. This type defines two constructors; *O* and *S*, as seen from Line 2 to Line 3 of Listing 2.1. *O* represents $0 \in \mathbb{N}$, and *S* represents a function that returns the successor of a value of type *nat*. The *nat* type is defined inductively, meaning it defines a list of constants and functions, called constructors, to create an element of a type [Chl13]. The *nat* type models $\mathbb{N}$ recursively in Coq, based on the Peano axioms [MB00]. For example, the value $2 \in \mathbb{N}$ is represented as *S (S O)*.

Coq represents a formal proof as a collection of unproven subgoals. Each subgoal has a local context and a conclusion. The context contains variables, local definitions, and hypotheses. The subgoal is a statement that is to be solved by its local context. It is also possible to use constants from the global environment, e.g., other proved lemmas or theorems, to solve a subgoal. A proof is complete if all subgoals are solved. To demonstrate how to formally verify a proposition about *nat* we prove the associativity of addition, denoted by the theorem *plusAssoc*. This theorem denotes the proposition's type:

$$\forall xyz : nat, x + y + z = x + (y + z)$$

The type is defined between the keywords *Theorem* and *Proof*, as seen in Line 6 of Listing 2.1. Between *Theorem* and the : sign stands the theorem's identifier. To solve the proof's subgoals, the user must provide a proof script defined between the keywords *Proof* and *Qed*[2]. From this proof script Coq generates a proof object internally to check if it constructs a term of the proposition's type. The proof script consists of a collection of proof methods (tactics) in $\mathscr{L}_{tac}$, e.g., intros or reflexivity[3]. Coq provides pre-defined tactics but $\mathscr{L}_{tac}$ also allows the specification of user-defined proof methods.

The first tactic applied to prove the lemma *addAssoc* is the *intros* tactic, as seen in Line 8 of Listing 2.1. This tactic introduces variables like *x*, *y*, and *z* to the subgoal's local context. The next applied tactic is *induction* which requires a parameter to perform induction on, as seen in Line 9 of Listing 2.1. This tactic splits the current subgoal: $x + y + z = x + (y + z)$ into one subgoal for the $x = O$ and one for $x = S\ x$, derived from the *nat* type's constructors. These cases represent the base case and the induction step. The - sign is used in the proof script to separate these cases, as seen in Line 10 of Listing 2.1. The first subgoal is solved by applying the tactic *reflexivity*. This tactic checks if the subgoal has the form *t=u* and whether *t* and *u* are convertible. The first tactic to solve the second subgoal is *simpl*, as seen in Line 11 of Listing 2.1.

[2] *Qed* is an abbreviation for the latin phrase "quod erat demonstrandum", which means "what was to be shown". Traditionally *qed* shows the end of a mathematical proof.

[3] To see the full list of tactics provided by the Coq proof assistant and their specified behavior, visit: https://coq.inria.fr/refman/coq-tacindex.html.

This tactic tries to reduce a term based on different reduction techniques, e.g., $\beta$-reduction [Chl13] or the expansion of transparent constants. Transparent constants are unfoldable, i.e, their identifier can be replaced with their definition in the subgoal's context or in the subgoal itself. Function definitions are transparent constants while theorems or lemmas are opaque constansts, meaning they cannot be unfolded. The *simpl* tactic reduces the subgoal:

$$Sx + y + z = Sx + (y + z)$$

to the following:

$$S(x + y + z) = S(x + (y + z))$$

The next tactic applied is called *rewrite*, as seen in Line 12 of Listing 2.1. This tactic rewrites a subterm with another subterm. Since the induction's base case ($x = 0$) is already proven, Coq added the Induction Hypothesis for x (IHx):

$$IHx : x + y + z = x + (y + z)$$

to the second subgoals context. The subgoal is rewritten by this hypothesis to:

$$S(x + (y + z)) = S(x + (y + z))$$

The *reflexivity* tactic solves the second subgoal as both sides of the = sign are convertible, as seen in Line 13 of Listing 2.1. Since all subgoals are solved the proof is finished.

After the proof script is complete, we look at the constructed proof object, seen in Line 17 of Listing 2.1. With the inductive definition of *nat* comes automatically defined elimination principles, e.g., the induction principle *nat_ind*. This principle is a function that denotes the following type:

$$\forall P : nat \rightarrow Prop, P0 \rightarrow (\forall n : nat, Pn \rightarrow P(Sn)) \rightarrow \forall n : nat, Pn$$

If a property *P n* holds for *P 0*, and it holds for *P (S n)* under the assumption that *P n* holds, then it holds for all *n*. The $a \rightarrow b$ notation denotes a function mapping *a* to *b*. If a proposition holds, it means that this proposition evaluates to *True*, which is a constructor of *Prop*. For each inductive type *t*, Coq generates an induction principle *t_ind*. The property *P* is denoted as a function (fun) seen in Line 19 of Listing 2.1. The function that evaluates *P O* to *True* is *eq_refl*. The *eq_refl* function defines equality on types and is used by the *reflexivity* tactic. This tactic solves a subgoal if *x* and *y* have the same type. For example, *S (S 0)* and *O* are not of the same type because different constructors are used. As seen in Line 21 of Listing 2.1, the property *P*'s function body

is used as a function argument denoted as *IHx*. The function *eq_ind_r* denotes the following type:

$$\forall (A : Type)(x : A)(P : A \rightarrow Prop), Px \rightarrow \forall y : A, y = x \rightarrow Py$$

If a property $P\,x$ holds for all $y$, where $y$ is equal to $x$, this property holds for $P\,y$. The *eq_ind_r* function defines equality rewriting and is used by the *rewrite* tactic. This function is polymorphic (A : Type), meaning that *A* is inferred at compile-time. The *addAssoc* function is gradually constructed from the applied tactics. Coq's type checker checks in every step if the function satisfies the required proposition's type. If not, the proof is not finished, and the user has to change the proof script.

To trust a constructed proof means to trust the proof assistant's type checker. The prototypical way to trust the type checker is to implement the *deBrujin criterion*, also called the *LCF approach* [Pol97, Geu09a], named after the LCF proof assistant [Mil79]. A proof assistant satisfies this criterion if it generates independently checkable proof objects. These proof objects can be checked independently by a program that a skeptic user can "quickly" write if there are any doubts regarding the type checker's correctness. For Coq, the type checker's correctness was formally specified and subsequently verified [SBF+20]. Satisfying the *deBrujin criterion* boils down to trusting only the proof assistant's type checker; instead of the entire system, a formal proof is correct.

## 2.2 DEPENDENT TYPES

A *dependent type* allows the definition of a type that relies on an additional value. Describing hardware designs using *dependent types* is not new and started back in the 90s [HD92, BMH07]. For instance, type *Vec A n* defines a vector of length *n* with components of type *A*. We call *Vec A n* a *dependent type* if the definition of *Vec A* depends on *n*. Since the length is part of the type definition, the information about its length is lifted into the type level, i.e., the length of a vector is evaluated at compile-time. A famous example to demonstrate the usage of dependent types is the *head* function. This function returns the first element of a vector. The question is: *what is the first element of a vector of length zero?* Listing 2.2 demonstrates how to use dependent types to address this question.

The *head* function takes a vector of the dependent type *Vector.t A (S n)* as an explicit argument and returns an element of the type *A*. Type *A* is a polymorphic type (A:Type), i.e., its actual type is inferred at compile-time. Parameters enclosed in curly brackets are implicit and give Coq additional information about this parameter. In contrast to explicit parameters, implicit ones are not part of the function call. Since

```
1  Definition head {A : Type} {n:nat} (v : Vector.t A (S n)) : A :=
2    match v with
3      |  x :: _ => x
4    end.
5
6  Definition v0 : forall A : Type, t A 0:= nil.
7  Definition v3 : Vector.t nat 3 := [1;2;3].
8
9  Compute (head v0). (* The term "v0" has type
10                        "forall A : Type, t A 0"
11                        while it is expected to have type
12                        "t ?A (S ?n)". *)
13
14 Compute (head v3). (* = 1 : nat *)
```

**Listing 2.2 :** *Specification of the* head *function using dependent types in Gallina. This function only accepts vectors of length greater than zero. The vectors* v0 *and* v3 *are examples to demonstrate the usage of dependent types to specify this behavior.*

the vector requires a value of a constructed type using the constructor *S*, Coq needs information about the type of *n*. The second implicit parameter denotes that *n* has the type *nat*, described above. Using *S n* to specify the vector's type prevents applying the *head* function to a vector using the *nat* constructor *O*, i.e., this function can never be called with a vector of length zero. This prevention is evaluated at compile-time. The definition of the vectors *v0* and *v3* demonstrate this behavior. *v0* denotes a vector of length zero. In this case, the actual type of *A* cannot be inferred since *nil* is the empty list, leaving the *v0*'s type polymorphic. *v3* denotes a vector of length three containing values of type *nat*. Applying the *head* function to these vectors results in a compile-time error for *v0* because the constructor *O* is not the required constructor *S* and the returning of the first element of *v3*.

Proof assistants, like Coq, allow the description of user-defined *dependent types*. Thus, *finite* types can already be used at the FSL level, where usually infinite types such as $\mathbb{Z}$ or $\mathbb{N}$ are used.

## 2.3 HIGH-LEVEL SYNTHESIS FLOWS

This section gives an overview about different HLS flows[4]. HLS defines a hardware design flow that automatically synthesizes a hardware description at the ESL level to an implementation at the RTL level, e.g., a hardware design description embedded in C or C++ [CCA+13, Gro02] to VHDL. Figure 2.2 illustrates the HLS hardware design flow.

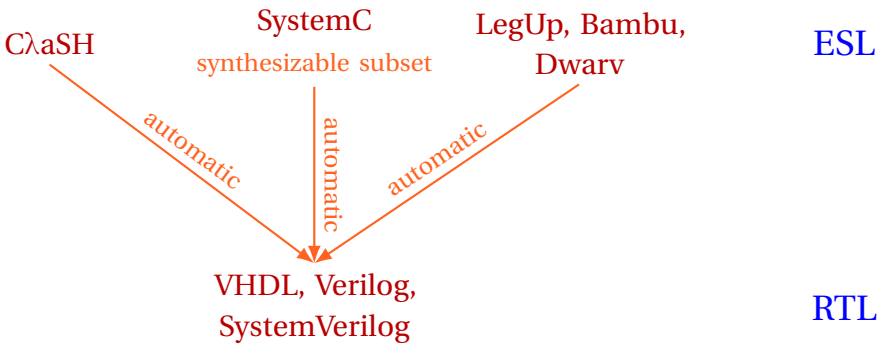


**Figure 2.2 :** *Extract of Figure 1.1 showing the sketched HLS synthesis flow. This extract was extended by CλaSH as it implements a HLS flow but is not seen as an established one. A model at the ESL level is automatically synthesized into an RTL implementation. The model's higher abstraction creates a better understanding and facilitates the development of larger designs.*

The different HLS flows differ in their input languages, design goals, and optimizations applied during the synthesis process. SystemC is focused on modeling hardware designs to test or verify particular properties early in the design process [Mar03, GD19] and in the Hardware/Software Codesign development process [Tei12]. Acceleration-oriented HLS flows like LegUp focus on synthesizing implementations with increased performance [CCA+13]. Flows like CλaSH focus on describing less error-prone and more understandable models using functional languages [Hug89].

[4] Note that this section *only* describes the HLS flows essential to follow the rest of this dissertation. The explanation of all would go beyond its scope.

### 2.3.1 SystemC

SystemC is a modeling and simulation language for hardware designs and is the "de-facto-standard" at the ESL level [BDBK09]. It defines a set of C++ class libraries that extend the C++ language elements. A hardware design describes a collection of modules that communicate through input and ouput ports or function calls [Arn00]. A simulation kernel allows the event-driven simulation for real-time communication between the individual modules.

SystemC supports two modes. The described model is compiled into an executable binary using a standard C++ compiler in the first mode. This mode is used for creating virtual prototypes [Mar03, HGW+20, GD19] or for Hardware/Software CoDesign, as mentioned above. Virtual prototypes model a particular behavior, e.g. protocols, periphirals, or processors, for early testing or verification [GD19, HGW+20]. In the Hardware/Software CoDesign development process, both hardware and software are developed concurrently to decrease development time and optimize or satisfy constraints like the final product's performance or power. This mode maps all extended SystemC elements to native C++ language elements. In the second mode, a model is synthesized in an RTL implementation. This mode requires a model in a restricted subset (a hardware description language embedded into C++) that is synthesizable [Inc16, FHT06, SWD13]. The synthesis of SystemC models requires a particular compiler to transform a model to an implementation in a hardware description language, e.g., in Verilog [CHM07]. The standard C++ compiler only supports the transformation to binary code and thus only the first mode is supported.

### 2.3.2 Acceleration-oriented Synthesis Flows

Due to the need for ever-faster hardware design implementations, HLS flows such as LegUp [CCA+13], DWARV [NSO+12] and Bambu [PF13] have been proposed. These flows require a model described in a description language embedded into the C programming language. A hardware design is synthesized from this model, and additional performance optimizations such as loop pipelining and functional pipelining are applied during the synthesis process [CCF+16, HHL91].

If a model contains unsynthesizable language constructs, such as dynamic memory allocation, LegUp automatically generates a hybrid hardware/processor architecture [CCA+13]. In contrast, DWARV and Bambu only support synthesizable language constructs [NSO+12, PF13]. The hardware/processor architecture splits the original

model into a hardware and a software part. This architecture contains a synthesized MIPS processor, which executes the software part and communicates with the hardware part.

A dedicated benchmark suite provides a quantitative analysis of the accelerated implementations synthesized by the above tools [HTHT09, NSP+16]. This benchmark suite defines hardware models implemented in a hardware description language embedded into C. These benchmarks range from a MIPS processor, encryption, and image/video processing algorithms to floating-point arithmetic.

### 2.3.3 FUNCTIONAL HARDWARE DESCRIPTION LANGUAGES

The idea of describing hardware designs functionally and simulate them started back in the 1980s [TH19], with languages like $\mu$FP [She84] and Daisy [Joh83]. A functional HDL description offers a higher degree of abstraction than HDLs embedded into C or C++, making them more comfortable reading and understanding [Hug89]. Pure functions realize combinational circuits, which map inputs to outputs without any internal state. These functions offer formal semantics due to their mathematical foundation. Pure functions combined with a state machine, either a Mealy machine or a Moore machine [DH89] realize sequential circuits. The state machine allows a time-controlled execution and represents the clock in functional sequential hardware designs. Compared with SystemC and the acceleration-oriented flows, the application of HLS flows using functional Hardware Description Language (HDL)s can be seen as an exception [NSP+16, Tak16].

CλaSH is considered a representative of the functional HDLs to describe how hardware designs are modeled and subsequently synthesized because this dissertation benefits from this HDL. It is a functional hardware description language that borrows its syntax and semantics from the pure functional programming language Haskell. Listing 2.3 shows a model of the multiply-and-accumulate circuit in CλaSH. This circuit multiplies to 32-bit signed integer values *x* and *y* and adds the result to a 32-bit signed integer accumulator *acc*, as seen in Line 7 of Listing 2.3.

Based on the *ma* function's type, the *macT* and *mac* function's types are inferred by the *CλaSH* compiler. The function *macT* realizes the transfer function required by the *mealy* function, seen in Line 1 of Listing 2.3. The *macT* function maps a state (acc) and an input ((x,y)) to a tuple consists of the new state (acc') and the output (o), as seen in Line 9 of Listing 2.3. The initial accumulator is 0. CLaSH allows the description of combinational and synchronous sequential hardware models, as described above. Asynchronous sequential models, which are not governed by a

```
1  mealy :: (s -> i -> (s, o)) -- transfer function
2        -> s                  -- current state
3        -> i                  -- current input
4        -> (s,o)              -- (new state, output)
5
6  ma :: Signed 32 -> (Signed32, Signed32) -> Signed32
7  ma acc (x,y) = acc + x * y
8
9  macT acc (x,y) = (acc',o)
10   where
11     acc' = ma acc (x,y)
12     o    = acc
13
14 mac = mealy macT 0
```

**Listing 2.3 :** *Model of the sequential multiply-and-accumulate circuit in CλaSH. The* ma *function realizes the actual behavior, while the* macT *implements the function type needed for the* mealy *machine.*

global clock signal, are not supported. A described circuit can be simulated and synthesized in an RTL implementation in a low-level hardware description language. The supported languages are Verilog [CS05], SystemVerilog [CS18], and VHDL [CS19].

# 3

# Proposing an Automatic Hardware Design Synthesis Flow using Formal Specifications

This chapter's foundation is the work published by Bornebusch et al. [BLWD20c]. It proposes an alternative hardware design synthesis flow to the established hardware design flow. The proposed flow combines the proof assistant Coq [BC04, Chl13] with the functional hardware description language CλaSH [BKK+10]. This combination allows the automatic extraction of a CλaSH model from a formal specification in Coq, which is subsequently synthesized by CλaSH's compiler in an implementation at the RTL level, e.g., in VHDL. As a result, the proposed synthesis flow automatically propagates verification results from the FSL level to the RTL level due to these automatic translations.

## 3.1 Introduction

Nowadays, hardware designs and the circuits synthesized from these designs are in almost every part of our lives and getting more complex over time. The increasing complexity leads to a rising number of potential failures. A hardware design flow must consider the increasing complexity to reduce potential failures in hardware designs at the beginning. Iterative methods have been proposed to address the problem of increasing complexity, e.g., considering hardware designs at a higher abstraction level. The established correctness-oriented hardware design flow specifies a hardware design at the FSL level [DSW12] in SysML/OCL [Obj15, Obj12], which can later be verified using automatic theorem provers [DHJ+10, PWPD18]. After the hardware design has been specified, a SystemC [Gro02, Tak16] model at the ESL level [MBP07] is realized manually. As a result, test benches are required to ensure the model corresponds to the SysML/OCL specification [WGHD09, WPK+16]. Again, test benches ensure that the final implementation corresponds to the SystemC model [CMK12].

In terms of implementation effort, the established design flow requires the manual conduction of almost all realization steps between the individual abstraction levels. More precisely, while a SystemC model's class structure can indeed be automatically generated from a SysML class diagram, its behavior, described by OCL constraints, must be implemented manually. The final RTL implementation has to be implemented manually due to the limitations of the SystemC's synthesizable subset [Inc16, FHT06, SWD13]. These manual translation steps make the established design flow's implementation effort time-consuming. Due to the manual translation steps, the model at the ESL level and the implementation at the RTL level rely on the generated test benche's quality. Any verification result obtained in higher abstraction levels does not need necessarily hold for lower abstraction levels. Furthermore, the SystemC's synthesizable subset does not define a formal synthesis scheme [EK95, BK13]. In general, it is unclear 1) how the implementation is generated from the model in detail; 2) whether the model's semantics are correctly propagated down to the implementation; and 3) how to track and verify properties stated at the FSL level in the RTL implementation.

Consequently, the established hardware design flow requires the repetition of similar design tasks at different abstraction levels. This design flow's fundamental problem is the lack of an automatic translation process between the individual levels. Proof assistant based design flows emerged to address this problem [BDN09, BC04, CVS+17, FSS15]. A formal specification of a hardware design is specified and verified by a proof assistant and automatically translated into an RTL implementation afterward. These design flows require no test benches at the RTL level due to the automatic propagation of verification results. However, following this flow, the hardware design is specified in a restricted Hardware DSL (HDSL). The HDSL is embedded into the proof assistant's specification language to specify hardware designs. As the HDSL focuses on particular hardware designs, these approaches cannot be treated as an alternative to the established hardware design flow. In contrast to these flows, the established one allows the specification and synthesis of arbitrary hardware designs.

This chapter proposes an alternative hardware design flow that combines the proof assistant based design flow with functional hardware description languages HDL [BKK+10, BCSS98]. Proof assistants like Coq [BC04] provide the extraction of formal specifications into executable code [Let08], which was extended to extract a CλaSH model [BKK+10] that again generates an RTL implementation. Both translation steps are automatic, i.e., no test benches need to be generated at the ESL and RTL level. The combination of proof assistants and functional HDLs provides a hardware design synthesis flow that does not require a restricted HDSL and eliminates the established design flow problems, as discussed above.

The rest of this chapter is structured as follows: First, the established hardware design flow is motivated and discussed in Section 3.2. Section 3.3 evaluates existing approaches regarding the lack of automation of the established flow and describes the proposed hardware design synthesis flow. Section 3.4 describes how both combinational and sequential hardware designs are formally specified and verified, while Section 3.5 describes how a specification is synthesized into an implementation. Section 3.6 evaluates the proposed design flow and presents the obtained results. Finally, Section 3.8 summarizes and concludes this chapter.

## 3.2 MOTIVATION

This section briefly reviews the established hardware design flow. Based on this review, its fundamental problem and the resulting motivation for a new hardware design synthesis flow to realize a given formal specification as a final implementation at the RTL level are discussed. This section introduces a running example to illustrate the established flow and the alternative flow proposed later in this chapter.

### 3.2.1 INTRODUCTORY EXAMPLE

Formal specifications, such as SysML/OCL [Obj15, Obj12], provide an abstract description of arbitrary hardware designs to be realized. These descriptions allow for formal verification [DHJ+10, PWD16] and provide a valid starting point for the correctness-oriented hardware design flow sketched in Figure 3.1.

This flow provides a formal specification in a modeling language such as SysML, which describes the desired hardware design structure, while constraints in OCL describe the design's behavior.

Afterward, a corresponding SystemC [Gro02] model has to be realized. Code skeletons can straightforwardly and automatically be derived from a SysML class diagram. The resulting SystemC model's functional behavior, described by OCL constraints, must be reimplemented manually since no executable model can be derived automatically from these constraints. Since manual implementations are error-prone, test bench generation ensures that the model corresponds to the specification [WGHD09, WPK+16]. Thus, the model significantly relies on the quality of these test benches. The resulting manual realization of the SystemC model and the generation of good-quality test benches is a time-consuming process. To finally synthesize

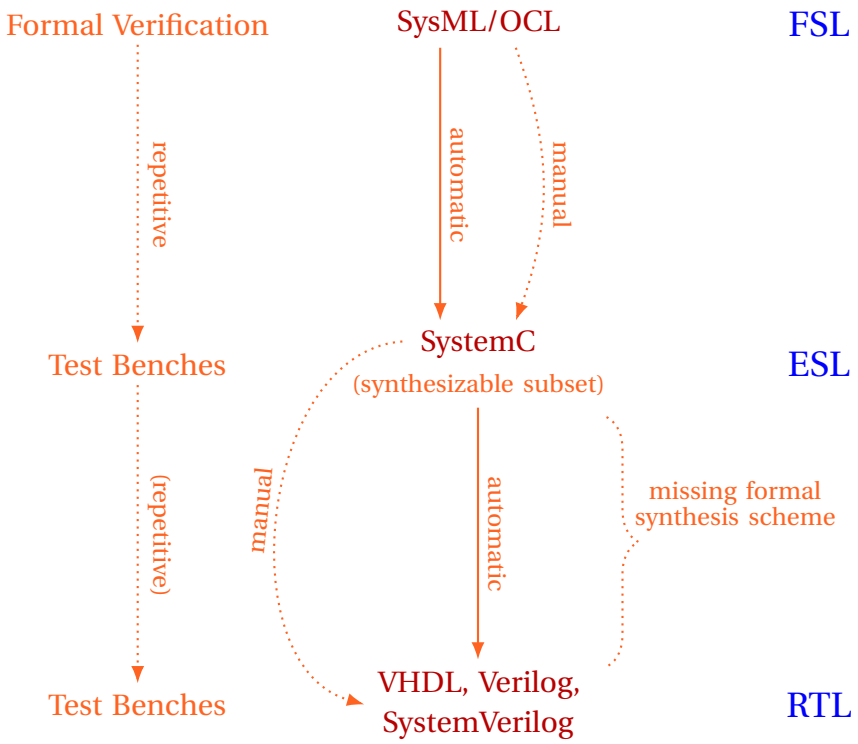


**Figure 3.1 :** *Extract of Figure 1.1 sketching the established correctness-oriented hardware design flow. A formal specification in SysML/OCL is manually translated into a SystemC model. This model is translated manually into an implementation in general. Only a model following the synthesizable subset can be synthesized automatically but this subset lacks a formal synthesis scheme. The manual translations between the models required test benches for the ESL model and the RTL implementation.*

the hardware design, e.g, on an Field-Programmable Gate Array (FPGA)[1] the RTL implementation must be implemented, e.g., in VHDL [CS19]. This step is also manual, in general, as SystemC does not support the synthesis of an arbitrary hardware design but only a design realized in a restricted subset [Inc16, FHT06, SWD13]. Like the SystemC model, the RTL implementation also relies, in general, on the generated test benches quality to ensure the implementation corresponds to the model [CMK12]. The resulting manual implementation and the generation of good-quality test benches make this step time-consuming as well.

[1] An FPGA is an integrated circuit that allows its configuration after manufacturing by the end user. This configuration option allows the adjustment and testing of hardware implementations realized in a HDL without manufacturing a new circuit.

Furthermore, SystemC' synthesizable subset does not define a formal synthesis scheme [EK95, BK13]. Such a scheme formally defines how terms in the implementation represent a term in the model to ensure that both descriptions have a semantic equivalent behavior.

#### 3.2.1.1 SPECIFICATION OF HARDWARE DESIGNS IN SYSML/OCL

This chapter considers a traffic light controller as a running example that controls the lights for the *trams*, the *cars*, and the *pedestrians*. Imagine the traffic controller is part of an intersection where the pedestrian crossing is orthogonal to the car's lane and tram's lane, which are parallel. Figure 3.2 shows the SysML specification of that controller.

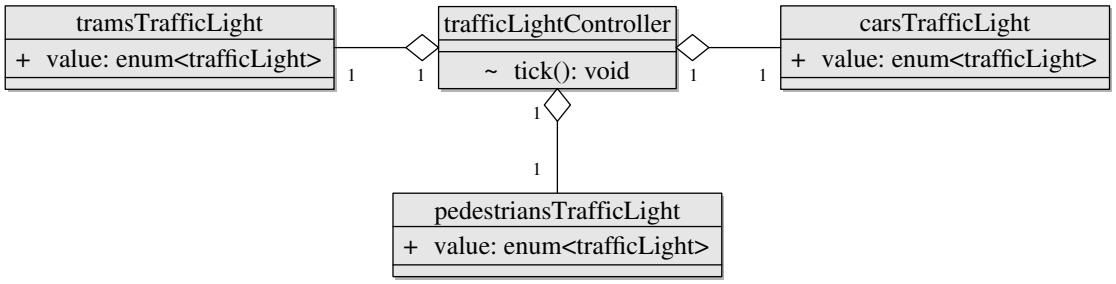


**Figure 3.2 :** *SysML specification of the traffic light controller. The* tick *function describes a Finite State Machine (FSM), which iterates over the individual states and sets the lights according to the current state. The states of this FSM encode the different traffic lights. Each traffic light's* value *attribute contains how the controller switches the individual lights off and on. For instance, if this attribute has the value* Green *for the cars, only the cars' lowest traffic light shines.*

```
1  context trafficLightController
2    inv pedestriansTrafficLightGreen:
3      self.pedestriansTrafficLight.value = trafficLight::Green implies
4      (self.tramsTrafficLight.value <> trafficLight::Green and
5      self.carsTrafficLight.value <> trafficLight::Green)
```

**Listing 3.1 :** *Safety property for the* trafficLightController *specified as OCL invariant. This property states that if the traffic light at the intersection for the pedestrians is green, it is* not *green for the trams and cars.*

As mentioned above, OCL constraints specify the design's functional behavior. They represent conditions, restrictions, or assertions related to one or more elements specified by boolean expressions. These expressions are specified as preconditions, postconditions, invariants. Each of them evaluates to either *true* or *false*. An OCL constraint must be satisfied, i.e., evaluate to *true*, by a correct design. If it evaluates to *false*, the constraint is violated. The invariant *pedestriansTrafficLightGreen* shown in Listing 3.1 specifies such an OCL constraint.

#### 3.2.1.2 REALIZATION OF THE SYSTEMC MODEL

In contrast to the automatic generation of code skeletons, the design's functional behavior , e.g., specified by the *tick* function shown in Listing 3.2, has to be implemented manually. As described above, required test benches ensure the correct behavior [WGHD09].

The extraction of a SystemC model to an implementation at the RTL level is quite challenging. First, these models lack a unique representation and provide unstructured communication that aggravates these model's automatic analysis [FHT06], e.g., communication over shared memory. Second, as SystemC is a C++ class library, constructs like dynamic memory allocation, recursion, loops with variable bounds or pointers are hard to analyze automatically [SWD13, MKM10]. These restrictions resulted in a minimum subset of SystemC for synthesis, also called a synthesizable subset of SystemC [Inc16]. In general, a SystemC model must be rewritten with these guidelines to be synthesizable to an RTL implementation. Due to this restricted subset, the traffic light controller's final implementation is to be realized manually and covered by test benches [CMK12].

### 3.2.2 CONSIDERED PROBLEM

The fundamental problem of the established hardware design flow is the manual and time-consuming translation steps between the individual levels. As a result, the SystemC model and the RTL implementation rely on the generated test benches' quality, in general. Verification results cannot be propagated through the established design flow automatically. These problems cannot be eliminated using SysML/OCL at the FSL level, and SystemC at the ESL level since OCL constraints cannot automatically be translated into executable SystemC code. An arbitrary SystemC model is not synthesizable. Even if the synthesizable subset is used for the model, a formal

```
1  void
2  tick(TrafficLight *states)
3  {
4    Car cars = states->cars;
5    Pedestrians pedestrians = states->pedestrians;
6    Tram trams = states->trams;
7
8    if (cars == Red && pedestrians == Red && trams == Red) {
9      states->cars = Red; states->pedestrians = Red;
10     states->trams = Green;
11   } else if (cars == Red && pedestrians == Red &&
12              trams == Green) {
13     states->cars = RedYellow; states->pedestrians = Red;
14     states->trams = Red;
15   } else if (cars == RedYellow && pedestrians == Red &&
16              tram == Red) {
17     states->cars = Green; states->pedestrians = Red;
18     states->trams = Red;
19   } else if (cars == Green && pedestrians == Red &&
20              tram == Red) {
21     states->cars = Yellow; states->pedestrians = Red;
22     states->trams = Red;
23   } else if (car == Yellow && pedestrians == Red &&
24              tram == Red) {
25     states->cars = Red; states->pedestrians = Green;
26     states->trams = Red;
27   } else {
28     states->cars = Red; states->pedestrians = Red;
29     states->trams = Red;
30   }
31 }
```

**Listing 3.2 :** *Realization of the* tick *function seen in Figure 3.2 for the SystemC model. This listing shows the omitted state transitions of the SysML/OCL specification in detail.*

synthesis scheme is missing. It is unclear whether the model's semantics are correctly represented by the implementation's semantics and how to track verified properties at the *fsl* level in the synthesized implementation.

A design flow that eliminates these problems must first allow the formal specification and verification of hardware designs specifiable by the established one. Second, the flow must propagate verification results from the FSL level to the RTL level automatically. This propagation includes the realization of formal synthesis schemes between the levels. Such a hardware design synthesis flow can be considered as an alternative to the established hardware design flow.

## 3.3 General Idea

This section proposes a new hardware design synthesis flow that addresses the established hardware design flow's problems discussed above. The proposed flow utilizes alternative design flows that have already been evaluated in formal hardware design verification and synthesis. These design flows provide partial solutions to the problems discussed above. After reviewing those flows, a new hardware design synthesis flow is proposed that combines the best of these and eliminates the established hardware design flow's problems.

### 3.3.1 Proof Assistant Based Hardware Design Flows

To address the missing automatic translation between the FSL level and ESL level of the established hardware design flow, we look at the formal specification and verification of programs by proof assistants [Geu09b, BC04, BDN09]. Proof assistants allow the formal specification and verification of mathematical objects like $\mathbb{N}$ or $\mathbb{Z}$ and operations about them using a higher-order logic, as described in Section 2.1. Due to the expressiveness of higher-order logic, a formal and machine-checkable proof is constructed by human-machine collaboration.

Apart from specifying and verifying mathematical objects, some proof assistants like Coq provide an extraction backend [Let08]. This backend allows the automatic extraction of executable code in a functional language, e.g., Haskell or OCaml, from a formal specification. A functional language embedded into the proof assistant's specification language enables the translation of functions and data types into executable code by a straightforward syntactic substitution [Let08]. The functional specification

language describes the program's behavior as a collection of pure and total functions. Pure functions are functions in the mathematical sense and do not contain any internal state. Total functions are required as the proof assistant's type checker must guarantee the function's termination. Non-terminating functions are undecidable, so properties about them cannot be specified. The extraction of executable programs from a formally verified specification is called *certified programming* [Chl13]. A famous example of certified programming is the CompCert project [LBK+16]. This project investigates the verification of C compilers utilizating the proof assistant Coq. The project's proposed compiler comes with a mathematical, machine-checked proof that 5Athe generated binary has the same semantics as the input program[2].

Some work has already been done to combine proof assistants with hardware design verification and synthesis [CVS+17, FSS15]. Figure 3.3 sketches their general design flow. A HDSL is embedded in the proof assistant's specification language. The hardware design is specified as a collection of functions and data types in that HDSL. Combinational circuits are described as pure and total functions, while sequential ones are composed of these functions and combined with a finite state machine, e.g., a Mealy machine [DH89]. This machine describes the clock functionally as a transition from one state to the next. The HDSLs only describe the design's functional behavior without considering dedicated hardware properties, e.g., parallelism or the clock frequency. This description enables the automatic analysis of a specified behavior and the extraction of RTL implementations from a specification [Gam13, BK13].

At first glance, this design synthesis flow addresses the established hardware design flow's problems described in Section 3.2.2. However, the HDSLs focus on dedicated hardware designs, e.g., multi-core CPUs [CVS+17] or low-level hardware designs [FSS15]. The specification language already provided by the proof assistant is restricted by the HDSL, which results in a limited expressivity of hardware designs. For example, the HDSL proposed by the Kami project [CVS+17] is not designed for describing combinational circuits, and the PI-Ware project's HDSL [FSS15] focuses on low-level circuits, e.g., multiplexer or the parallel prefix sum. These HDSLs could be extended to eliminate the restricted expressivity of hardware designs, but this extension is time-consuming.

The proof assistant's ability to extract executable code leads to the following question: *Can arbitrary hardware designs be specified using the proof assistant's specification language and exploiting their existing extraction mechanism?*

[2] The CompCert project's compiler supports a source program in ISO C 99 or ANSI C. This program can be compiled for several architectures such as RISC-V, ARM, and x86. The performance of the generated code is approximately 90% of the performance of the GCC compiler: https://compcert.org/compcert-C.html.

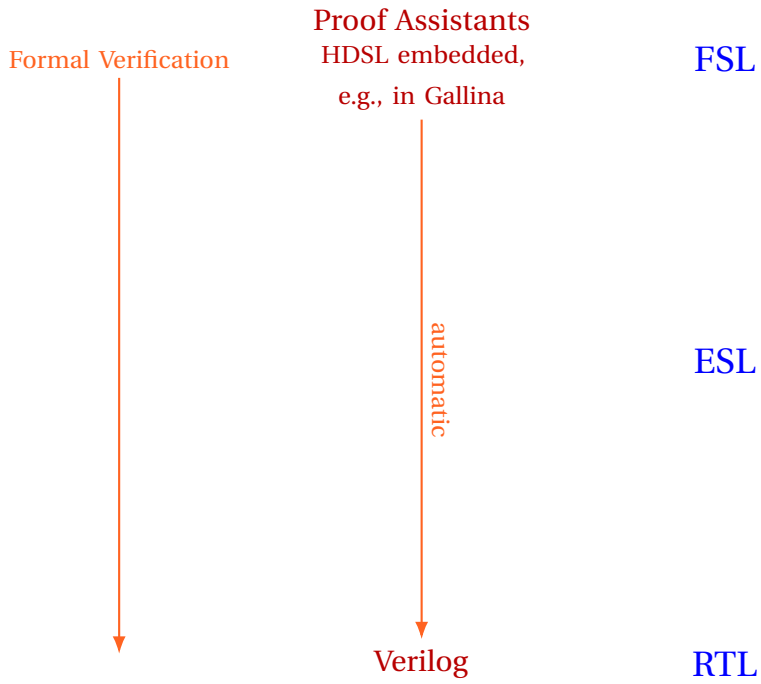


**Figure 3.3 :** *Sketched hardware design synthesis flow using proof assistants. An RTL implementation is extracted automatically from a formal specification in a HDSL. Hence, verification results are propagated automatically down to the implementation.*

### 3.3.2 FUNCTIONAL HARDWARE DESCRIPTION LANGUAGES

We consider Functional HDL (FHDL)s in this section to address the problems of SystemCs synthesizable subset the established hardware design flow has. Analogous to the proof assistant based design flow described above, combinational hardware designs are described as pure functions and data types, as described in Section 2.3.3. Sequential hardware designs are composed of these functions combined with a finite state machine, e.g., the Mealy machine.

The unique representation of such hardware design models and the structured communication between their components, ensured by their underlying type system, enables the automatic analysis and synthesis in an RTL implementation. Figure 3.4 sketches the synthesis of hardware designs by FHDLs.

Replacing SystemC [Gro02] with an FHDL like CλaSH does not solve the established hardware design flow's problems, discussed in Section 3.2. The automatic generation of an executable model in CλaSH from a SysML/OCL specification is impossible. One reason for this is the specification's infinite data types [Obj15].

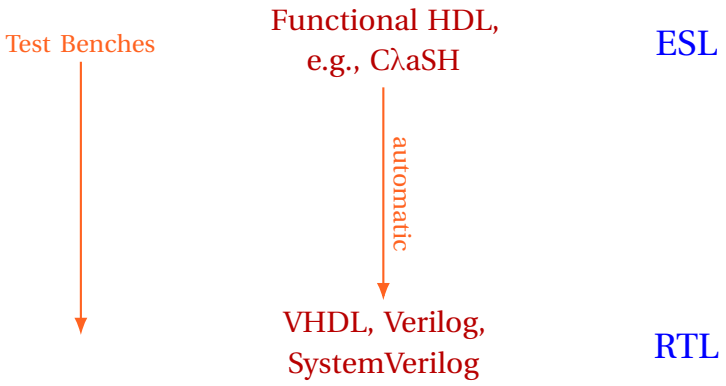


**Figure 3.4 :** *Sketched hardware design synthesis flow using functional HDLs. A model at the ESL level is automatically synthesized into an implementation at the RTL level. Hence, properties verified by test benches also hold for the implementation.*

Since functional HDLs support a larger synthesizable subset than SystemCs one, leading to the synthesis of almost any describable hardware design, the following question arises: *Can proof assistants be combined with functional HDLs for automatic hardware design synthesis?*

### 3.3.3 PROPOSED HARDWARE DESIGN SYNTHESIS FLOW

The proof assistant based hardware design flow is merged with functional hardware description languages to eliminate the established hardware design flow's manual and time-consuming translation steps. The proposed hardware design synthesis flow uses the proof assistant Coq [BC04, Chl13] at the FSL level. This proof assistant provides a formal specification language, called Gallina, based on an expressive formal language called CiC, described in Section 2.1. CiC combines higher-order logic with a richly-typed functional programming language. A dedicated tactic language, called $\mathscr{L}_{tac}$ [Del00], provides the realization of user-defined proof methods to construct formal and machine-checkable proofs by user-machine collaboration. As described in Section 3.3.1, Coq provides the extraction of executable code from a formal specification. The extraction mechanism of Coq was extended by the FHDL CλaSH [BKK+10] to generate an ESL model. In comparison with embedded functional HDLs, like Lava [BCSS98], CλaSH borrows its syntax and semantics from Haskell. This borrowing gives us access to all of Haskell's choice elements, like *case*-expressions and pattern matching [BKK+10]. CλaSH supports combinational and synchronous logic but no

asynchronous logic. In synchronous logic, a global clock signal synchronizes the state changes as that signal is propagated through the entire circuit. Models of synchronous sequential circuits are described either using a *Mealy* or a *Moore* machine. In asynchronous logic, state changes might occur at any time without being triggered by a global clock signal. However, multiple clock domains are generally supported, but a clock signal cannot be generated in one module, which assigns that signal to another. The CλaSH compiler supports the synthesis of a model in different RTL implementations, e.g., in VHDL, Verilog, or SystemVerilog. Figure 3.5 shows the proposed design flow.

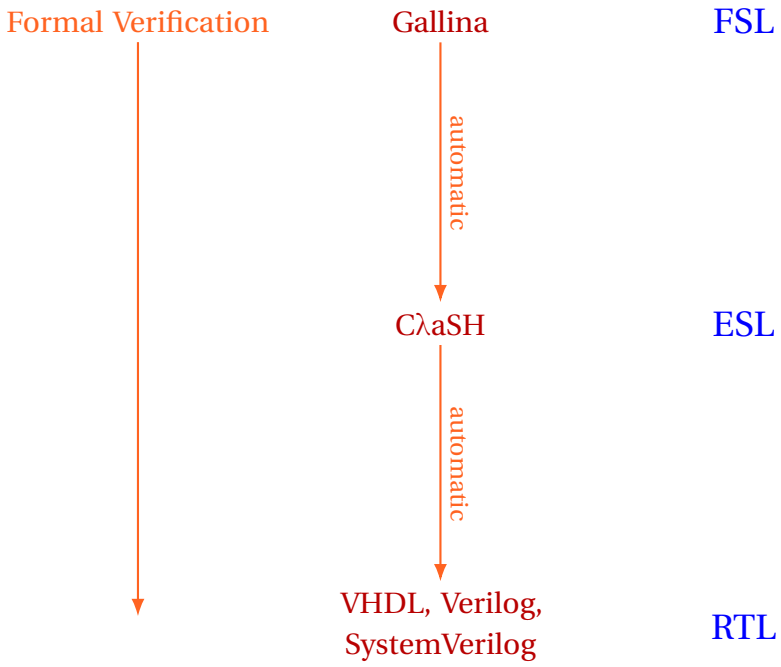


**Figure 3.5 :** *Proposed alternative hardware design synthesis flow to the established hardware design flow. This flow combines proof assistants with functional hardware description languages. A functional specification of a hardware design in Coq's specification language Gallina is formally verified. From this specification an executable model in CλaSH is extracted, which is finally synthesized into an RTL implementation. Hence, verification results are automatically propagated from the formal specification down to the implementation.*

By combining proof assistants with functional hardware description languages, the proposed hardware design synthesis flow does not require manual translation steps as the established design flow, described in Section 3.2.1. The automatic translation steps significantly reduce the implementation effort and thus accelerate the entire hardware

design process. Only a specification at the FSL level is necessary while the model at the ESL level and the implementation at the RTL are generated automatically. The proposed design synthesis flow does also not rely on test bench generation at the ESL and RTL levels as verification results from the FSL level are propagated through the entire flow automatically. Due to the combination of Coq and CλaSH, the proposed hardware design synthesis flow is refered to as the Coq/CλaSH flow for the rest of this dissertation.

## 3.4 SPECIFICATION OF HARDWARE DESIGNS

This section describes how formal specification of a combinational and synchronous sequential hardware designes are formally verified. Furthermore, it shows how the traffic light controller and its safety property, shown in Section 3.2.1, is respecified in Gallina. It is formally verified that the respecified controller satisfies the respecified safety property.

### 3.4.1 FINITE INTEGER TYPES

Proof assistants usually specify infinite types like $\mathbb{N}$ or $\mathbb{Z}$ to prove properties about these mathematical objects. Unlike this, hardware description languages, such as CλaSH or VHDL, describe fixed-sized machine integers or bit-vectors representing the circuit's in- and outputs. Machine integers are integer values represented as bit vectors. Dependent types allow the specification of *finite* types at the specification level, as described in Section 2.2. The CompCert [LBK+16] project provides the user-defined specification of finite integer types as a library for Coq and allows the specification of *signed* and *unsigned* types of arbitrary sizes. Furthermore, it defines arithmetic operations like addition, multiplication, power, and so on, on these types. Utilizing this integer library gives us these operations, including verified properties about them, for free. This is an advantage as merely the integer type of the required size has to be specified and not the arithmetic operations on this type.

### 3.4.2 COMBINATIONAL CIRCUITS

As described in Section 3.3.1, Coq describes behavior as a collection of pure and total functions. These functions exactly represent the behavior of combinational hardware designs because these designs merely map inputs to outputs. The *multiply-accumulate* circuit is a famous example of a combinational hardware design. Listing 3.3 shows this design specified in Gallina.

```
1  Definition mac
2     (acc : Unsigned32.int)
3     (x   : Unsigned32.int)
4     (y   : Unsigned32.int)
5   : Unsigned32.int :=
6   acc + (x * y).
```

**Listing 3.3 :** *Combinational specification in Gallina of an 32-bit unsigned* Multiply-accumulate *circuit. The two input arguments* x *and* y *are multiplied. The result of this multiplication is added to the input accumulator* acc. *The result of this addition is returned by the* mac *function.*

```
1  Theorem macMultipliesAndAccumulates:
2   forall acc x y : Unsigned32.int,
3   mac acc x y = acc + (x * y).
4  Proof.
5    intros.
6    unfold mac.
7    reflexivity.
8  Qed.
```

**Listing 3.4 :** *Theorem in Gallina that denotes that the specification of the* mac *function always adds the accumulator* acc *to the result of* x *times* y.

The integer type *Unsigned32.int* was specified using the CompCert integer library. This type represents the 32-bit unsigned integer type. As mentioned above, different arithmetic operations are already pre-defined for the individual integer types. We can benefit from these operations without redefining them, which is a considerable advantage and simplifies the specification of hardware designs significantly.

After specifying the *mac* circuit's functional behavior in Gallina, we must prove that this behavior satisfies its formal specification, i.e., we formally verify the circuit's functional hardware design. Theorems in Gallina describe such a formal specification, as seen in Listing 3.4.

To prove a theorem, the user must provide a proof script in $\mathscr{L}_{tac}$, as explained in Section 2.1. The theorem's proof is straightforward as the theorem holds by the *mac* function's definition. The proof starts by applying the *intros* tactic to the current subgoal. The *unfold* tactic replaces an identifier (mac) with its definition in the current subgoal:

$$mac\ acc\ x\ y = acc + (x * y)$$

is transformed to:

$$acc + (x * y) = acc + (x * y)$$

by this tactic. The *reflexivity* tactic solves this transformed subgoal and finishes the proof, as there is only one subgoal to prove.

### 3.4.3 SYNCHRONOUS SEQUENTIAL CIRCUITS

As described in Section 3.3.3, the proposed hardware design synthesis flow supports the formal specification of synchronous sequential hardware design. These hardware designs are described by a Moore or a Mealy machine representing the clock in a functional hardware design and allow a time-controlled execution. This section illustrates how these machines are specified in Gallina and what properties were proven about them. Furthermore, this section shows how these state machines are used as a foundation for sequential hardware design specifications.

#### 3.4.3.1 MOORE MACHINE

The *Moore* machine computes its output only by its current state [DH89]. Listing 3.5 shows its specification in Gallina.

The polymorphic *moore* function has four arguments, seen in the Lines 2-5 of Listing 3.5. These arguments are the transfer function *tf*, the output function *of*, the current state *s*, and a list of inputs *l* that should be processed. The output is a list of *o*, as seen in Line 6. This function is defined as a recursive function, indicated by

```
1  Fixpoint moore {S I O:Type}
2      (tf: S -> I -> S)
3      (of: S -> O)
4      (s: S)
5      (l: list(I))
6    : list(O) :=
7    match l with
8     | [] => []
9     | i::is => let s' := tf s i in
10                of s' :: moore tf of s' is
11   end.
```

**Listing 3.5 :** *Specification of the* Moore *machine in Gallina. This machine takes a transfer function* tf *as first argument. This function maps a state (S) and an input (I) to a new state. The second argument is the output function* of *which maps a state to an output (O). An initial state (s) and a list of inputs (list(I)) is also required. The result is a list of outputs (list(O)). The types S, I, and O are inferred at compile time.*

the keyword *Fixpoint*. Coq checks if every recursive function terminates by trying to infer its *decreasing argument*. Recursion, defined by a decreasing argument, is called *structural recursion*. If the inference fails, the designer must specify an additional function argument, which directs Coq to the *decreasing argument*. We omit this argument as Coq can infer that *structural recursion* is performed on the 4th argument: *l*. The two cases for the base case and the recursive step are evaluated by pattern matching. If the decreasing argument is the empty list *[]* – the base case – the recursion is terminated, and the empty list is returned. If the decreasing argument is not the empty list *i::is* – the recursive step – the list is split into its head *i* and tail *is*. The head of a list is its first element, while the tail is the list without the head. If the list is not the empty list, the function *tf* computes the new state *s'* using the current state *s*, together with the head of the list *i*. The function *of* computes the output from the new state. The transfer function, the output function, the new state, and the tail of the list are the new arguments for the *moore* functions' recursive call, as seen in Line 10 of Listing 3.5. The computed output of all recursive calls are appended (::) to the final output list.

We specify two theorems in Gallina depending on the base case and the recursive step. The theorem for the empty list is shown in Listing 3.6.

The theorem *mooreMachineEmptyList* is proved analogously to the theorem *macMultipliesAndAccumulates* seen in Listing 3.4, i.e., the same tactics are applied. If the

```
1  Theorem mooreMachineEmptyList:
2      forall {S I O:Type} (tf : S -> I -> S) (of : S -> O),
3      forall s : S,
4      moore tf of s [] = [].
5  Proof.
6      intros.
7      unfold moore.
8      reflexivity.
9  Qed.
```

**Listing 3.6 :** *Theorem including its proof about the* moore *function specified in Gallina and $\mathcal{L}_{tac}$. It states that for all arguments if the input list of the* Moore *machine is the empty list* [] *its output is the empty list as well.*

decreasing argument is the empty list, the *moore* function returns the empty list as well. The replacement of the identifier *moore* by its definition simplifies the subgoal to show that the empty list is equal to itself.

The second specified theorem is shown in Listing 3.7. This theorem ensures the correct application of the transfer function and the output function the *Moore* machine requires.

```
1  Theorem mooreMachineOneElementList:
2      forall {S I O:Type} (tf : S -> I -> S) (of : S -> O),
3      forall s : S,
4      forall i : I,
5      let s' := tf s i in
6      moore tf of s [i] = [of s'].
7  Proof.
8      intros.
9      unfold moore.
10     reflexivity.
11 Qed.
```

**Listing 3.7 :** *Theorem including its proof about the* moore *function specified in Gallina. The theorem states that if the* moore *function is called with a transfer function* tf*, an output function* of*, a state* s *and a one-element list* [i]*, the output is the result of* of *of* s'*, which is the new state.*

The new state *s'* was specified as the transfer function's call with the state and the input, seen in Line 5 of Listing 3.7. An identifier (s') is bound to a term (tf s i) using the *let* keyword. The resulting statement is part of the subgoal's context. Such statements structure the theorem to make it more clear. Again, the same tactics are applied as for the theorem *mooreMachineEmptyList* to prove this theorem. The replacement of the identifier *moore* results in the simplification of the subgoal, as Coq automatically deduces from the definition that the new state *s'* is the argument for the output function *of*. This simplification leads to proving a subgoal that the *moore* function's output is equal to the = sign's right side.

#### 3.4.3.2 MEALY MACHINE

In contrast to the *Moore* machine, the *Mealy* machine computes the output from its current state and current input [DH89]. Listing 3.8 shows the specification of this state machine in Gallina.

```
1  Fixpoint mealy {S I O:Type}
2     (tf: S -> I -> (S*O))
3     (s: S)
4     (l: list(I))
5   : list(O) :=
6   match l with
7    | [] => []
8    | i::is => let (s', o) := tf s i in
9               o :: mealy tf s' is
10  end.
```

**Listing 3.8 :** *Specification of the* Mealy *machine in Gallina. The machine takes a transfer function* tf *as first argument. This function maps a state (S) and an input (I) to a tuple of a new state and an output (S*O). An initial state (s) and a list of inputs (list(I)) is also required. The result is a list of outputs (list(O)). The types S, I, and O are inferred at compile time.*

The *mealy* machine has three arguments: the transfer function *tf*, the current state *s*, and the input list *l*, as seen in Line 2- Line 4 of Listing 3.8. Its return argument is a list of outputs *O*. Analog the *moore* function definition; the mealy function is specified as a polymorphic recursive function. The *structural recursion* is performed on the 3rd

argument: *l*. The two cases for the base case and the recursive step are evaluated by pattern matching as described above. If the decreasing argument is the empty list, the recursion terminates by returning the empty list. For the recursive step, the transfer function uses the head of the list *i* and the current state *s* to compute a tuple of the new state *s'* and the output *o*. A tuple is required as the transfer function computes both the new state and output. The transfer function, together with the new state and the tail of the list *is*, is applied to the *mealy* function's recursive call. As for the *Moore* machine, the outputs of all recursive calls are appended to the final output list.

We specify two theorems in Gallina depending on the *mealy* function's base case and the recursive step. Listing 3.9 shows the first theorem. Proving theorem *mealyMachineEmptyList* is straightforward since the decreasing argument is the empty list. The same tactics as for the *mooreMachineEmptyList* theorem seen in Listing 3.6 are applied to prove this theorem.

```
1  Theorem mealyMachineEmptyList:
2    forall {S I O:Type} (tf : S -> I -> (S*O)),
3    forall s : S,
4    mealy tf s [] = [].
5  Proof.
6    intros.
7    unfold mealy.
8    reflexivity.
9  Qed.
```

**Listing 3.9 :** *Theorem including its proof about the* mealy *function specified in Gallina. It states that for every transfer function* tf *and for every state* s*, if the* Mealy *machine is called with the empty list* [] *the output is the empty list as well.*

The second theorem is shown in Listing 3.10. This theorem shows the correct application of the transfer function *tf* to the current state and the current input. The same tactics as for the theorem above are applied to prove this theorem, except for the *destruct* tactic. After simplifying the subgoal by replacing the identifier *mealy* with its definition, we need to show that the statement in Line 8 of Listing 3.8 computes the output returned by the *mealy* function. To prove this, the tactic *destruct* is applied to *tf*, as seen in Line 11 of Listing 3.10. If this tactic is applied to an identifier, e.g., *tf*, it erases it if this identifier is not any more dependent in the current subgoal after the application of *destruct*. As a result, this tactic reduces the current subgoal:

$$let(\_,o0) := tf\ s\ i\ in\ (let(\_,o1)\ :=\ tf\ s\ i\ in\ [o1]) = [o0]$$

to the following subgoal:

$$[o0] = [o0]$$

The destruction of *tf* can be seen as an application of *tf* to its arguments. Since the theorem does not consider the new state *s'*, *destruct* reduces the equation's left side to a list containing the output of function *tf*. The final step is to prove that the equation's left side is equal to itself.

```
1  Theorem mealyMachineOneElementList:
2    forall {S I O:Type} (tf : S -> I -> (S*O)),
3      forall s s': S,
4      forall o : O,
5      forall i : I,
6      let (s',o) := tf s i in
7      mealy tf s [i] = [o].
8  Proof.
9      intros.
10     unfold mealy.
11     destruct tf.
12     reflexivity.
13 Qed.
```

**Listing 3.10 :** *Theorem including its proof about the* mealy *function specified in Gallina. The theorem states that if the* mealy *function is called with a transfer functions* tf*, a state* s*, and a one-element input list* [i]*. The output (o) is the second item of the tuple returned by the transfer function.*

Summarized, using the generic types *S*, *I*, and *O* for both state machine specifications make them polymorphic and independent from a specific hardware design. These polymorphic function definitions allow the specification and verification of arbitrary synchronous sequential hardware designs.

#### 3.4.3.3 SPECIFICATION AND VERIFICATION OF HARDWARE DESIGNS

This section describes how sequential hardware designs are specified in Gallina using a Mealy machine specification. To show what such an implementation looks like, we consider the traffic light controller described in Section 3.2.1. This controller

was respecified in Gallina using the *mealy* machine specification introduced in Section 3.4.3.2. First, we look at the machine's function type:

$$\{S\ I\ O : Type\}\ (tf : S \to I \to (S \times O))\ (s : S)\ (l : list(I)) : list(O)$$

This type requires a transfer function *tf* that turns a state of the type $S$ and an input of the type $I$ into a tuple of the new state and an output tuple $S \times O$. Furthermore, an initial state $s$ and a list of inputs $l$ are required. The result is a list of outputs *list(O)*.

Second, the finite integer types for the polymorphic types $S$, $I$, and $O$ have to be determined. To determine these types, we consider the SystemC example seen in Section 3.2.1.2. The type $S$ contains the current traffic lights state, e.g., the pedestrians have green, and the cars and trams have red. The counter is also part of the state. It determines after which period of time the transition function calls the *tick* function. The type $O$ contains the new traffic light state that determines which light is turned on or off. The definition of the input type $I$ is unnecessary for the traffic light controller, as there is no real input that needs to be processed. The state contains all information necessary for the controller. However, the definition of the transition function requires an input, which we define as boolean. The specification of the traffic light controller is shown in Listing 3.11.

The *tick* function, as seen in Line 1 of Listing 3.11, transforms a given state (*s*) into a new state. The state transitions are analog to the ones described in Listing 3.2 and can be found in Appendix A.1. The inductive type *state* implements the different traffic lights for the *cars*, *pedestrians*, and *trams*. As described in Section 2.1, an inductive type defines constructors to create an element of this type. The different traffic lights are implemented as constructors of this inductive type. For example, the inductive type *trafficLightPedestrians* implements the constructors: *pedestriansRed* and *pedestriansGreen*. The specified inductives types can be seen in detail in Appendix A.1. The *clockFrequency* and the *delay* are constant functions. The *transitionFunction* definition implements the transition function for the Mealy machine. The state machine's initial state is a tuple that contains the initial state for the traffic lights *State carsRed pedestriansRed tramsRed* and an initial counter value *unsigned32_zero*, as seen in Line 16 of Listing 3.11.

After specifying the hardware design, we need to prove that this specification corresponds to the safety property described in Listing 3.1. Listing 3.12 shows this safety property respecified in Gallina.

The proof of this theorem is essentially a big case analysis. It checks for every constructor combination of *trafficLightCar*, *trafficLightPedestrians*, and *trafficLightTram* if the following property holds: a green traffic light for the pedestrians implies that the traffic light for the cars and the trams is *not* green. First, the *intros* tactic introduces the quantified variables and the hypothesis to the subgoal's local context. An implication

```
1  Definition tick (s : state) : state
2
3  Definition transitionFunction
4     (data:(state*Unsigned32.int))
5     (dummy:bool)
6   : (state*Unsigned32.int*state) :=
7   match data with
8    | (state, counter) =>
9        if counter = (clockFrequency * delay)
10         then let state' := tick state in
11              ((state', unsigned32_zero), state')
12         else ((state, increment counter), state)
13  end.
14
15 Definition topEntity := mealy transitionFunction
16   (State carsRed pedestriansRed tramsRed, unsigned32_zero).
```

**Listing 3.11 :** *Specification of the* trafficLightController *in Gallina. A counter is incremented by one until it reaches an upper bound (clockFrequency × delay). If that bound reaches the new state (state') is used as the input for the next recursive call and as the output. The counter is reset back to zero (unsigned32_zero). As long as that bound is not reached the state remains unchanged. The* mealy *machine specification, seen in Listing 3.8, is used. Note that Coq's type checker automatically inferres the input list (list(I)).*

```
1  Theorem pedestriansTrafficLightGreen:
2    forall s : state,
3    forall cars : trafficLightCar,
4    forall pedestrians : trafficLightPedestrians,
5    forall trams : trafficLightTram,
6    (State cars pedestrians trams = tick s /\
7    pedestrians = pedestriansGreen)        ->
8    (cars <> carsGreen /\ trams <> tramsGreen).
9  Proof.
10   intros.
11   destruct s as [cars' pedestrians' trams'].
12   destruct cars; destruct pedestrians; destruct trams.
13   all: destruct cars';
14        destruct pedestrians';
15        destruct trams'.
16   all: intuition.
17   all: discriminate.
18 Qed.
```

**Listing 3.12 :** *Theorem including its proof of the transformed safety property, seen in Listing 3.1, specified in Gallina. The theorem denotes that for all states, cars, pedestrians and trams, if the traffic light for the pedestrians is green this implies that it is not green either for the cars or for the trams.*

(->) consists of an hypothesis (left side) and a conclusion (right side). To prove an implication, we must show that under the assumption that its hypothesis holds, its conclusion can be deduced from this hypothesis. To prove the different cases, we first eliminate all quantified variables in the hypothesis and the generated subgoals. If the *destruct* tactic is called with an inductive type like *s*, it generates a subgoal for every constructor of that type. The *all* tactic applies a given tactic to all subgoals in the proof state. This tactic allows the destruction of the remaining quantified variables in all of these subgoals. The *intuition* tactic is a decision procedure that generates a set of subgoals that are equivalent to the original one but much easier to prove. After the destruction of all variables as seen in Line 13 of Listing 3.12, one subgoal is:

$$
\begin{aligned}
&H : State\ carRed\ pedestriansRed\ tramRed = \\
&\quad tick(State\ carRed\ pedestriansRed\ tramRed) \wedge \\
&\quad pedestriansRed = pedestriansGreen \\
&\text{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -} \\
&carRed <> carGreen \wedge tramRed <> tramGreen
\end{aligned}
$$

The dotted line seperates the above context from the below subgoal. By calling the *intuition* tactic on this subgoal, the following context and subgoals are generated:

$$
\begin{aligned}
&H0 : State\ carRed\ pedestriansRed\ tramRed = \\
&\quad tick(State\ carRed\ pedestriansRed\ tramRed) \\
&H1 : pedestriansRed = pedestriansGreen \\
&H : carRed = carGreen \\
&\text{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -} \\
&False \\
&\text{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -} \\
&False
\end{aligned}
$$

The new subgoals are generated from the $\wedge$ statement of the original subgoal. Not (<>) is essentially $a = b -> False$. The *intuition* tactic unfolds *not* automatically and added its hypothesis $H$ to the local context. Furthermore, it splits the $\wedge$ statement into two subgoals. The subgoals generated by the *intuition* tactic are solved by applying the tactic *discriminate*. This tactic solves a subgoal if there is trivial inequality in its context. Applying this tactic to all remaining subgoals solves them and so finishes the proof. Additional properties were proven about the traffic light controller, e.g., that the traffic lights are never green at the same time, as seen in Appendix A.2.

This section showed how the traffic controller, introduced in Section 3.2.1, was re-specified in Gallina using the Mealy machine specification described in Section 3.4.3.2.

It also showed the controller's safety property's respecification, introduced in Listing 3.1, in Gallina, which was subsequently formally verified using Coq's tactic language $\mathscr{L}_{tac}$.

## 3.5 SYNTHESIS OF HARDWARE DESIGN SPECIFICATIONS

This section describes how a hardware design at the FSL level specified in Coq's specification language Gallina is eventually synthesized into an implementation at the RTL level. First, a CλaSH [BKK+10] model is extracted automatically from the specification. This extraction was implemented by extending Coq's extraction backend by CλaSH. Second, the model is synthesized into an implementation. This synthesis is already implemented by CλaSH and is not part of this dissertation.

### 3.5.1 MODEL EXTRACTION FROM SPECIFICATIONS

The hardware design synthesis flow, proposed in Section 3.3.3, uses the functional hardware description language CλaSH to synthesize an extracted model to an implementation from a formal specification. Coq's extraction backend was extended by CλaSH to achieve this. This section focuses on how this extension was implemented and how the extraction process works in detail.

Proof assistants like Coq provide an extraction backend allowing the extraction of executable code from a formally specified and verified specification, as mentioned in Section 3.3.1. The supported target programming languages are functional ones like Haskell or OCaml [Let08, Chl13]. Since Coq's specification language embeds a functional programming language, the extraction of executable code in a functional language is realized by a straightforward syntactical substitution process [Let08]. However, as seen in Section 3.4, a formal specification in Coq represents the functional behavior described by functions and theorems to prove properties about these functions. Now the question arises: *What happens to theorems during the extraction process?*

The convention in Coq is that the type *Set* is meant for computation while the type *Prop* is meant for logical propositions [Chl13]. Thus, functions are of the type *Set*, and theorems are of the type *Prop*. Note that *Type*, seen in Section 3.4.3, is a supertype of these types. Coq's extraction mechanism distinguishes between *Set* and *Prop* to extract everything related to *Set* while it erases everything related to *Prop* during this

process [Chl13]. As a result, only the specified functional behavior is extracted to the target language.

Coq's extraction mechanism supports two different modes. In the first mode, the mechanism recursively extracts everything related to the specified function, i.e., other called functions and used data types. This mode does not consider any intrinsic functions or data types of the target language. The semantics of the specification are automatically translated into the equivalent semantics of the target language. In the second mode, the mechanism substitutes specified functions and data types with those of the target language. In this case, it is up to the user to substitute the specified functions and data types with the ones that have the same semantic behavior in the target language. These substitutions have to be configured by hand. However, the extraction backend provides some of these substitutions, e.g., how to substitute Coq's *option* type with Haskell's *maybe* type. These configured substitutions aim to use semantic equivalent functions and data types of the target language and only extract the underlying functional behavior. Using the target language's intrinsic functions might result in an increased performance or enable the usage of library functions.

```
1  tick :: State0 -> State0
2
3  transitionFunction :: ((,) State0 (Unsigned 32))
4                     -> CLaSH.Prelude.Bool
5                     -> (,) ((,) State0 (Unsigned 32)) State0
6  transitionFunction data0 _ =
7    case data0 of {
8     (,) state counter -> let {state' = tick state} in
9      case (CLaSH.Prelude.==)
10           counter ((CLaSH.Prelude.*) (50*(10^6)) 15) of {
11       CLaSH.Prelude.True ->
12         (,) ((,) state' 0) state';
13       CLaSH.Prelude.False ->
14         (,) ((,) state (increment counter)) state }}
15
16 topEntity = mealy transitionFunction ((,)
17   (State carsRed pedestriansRed tramsRed) 0)
```

**Listing 3.13 :** *Extract of the CλaSH model automatically extracted from the* trafficLightController *specified in Gallina. The specified mealy machine, seen in Listing 3.11, was replaced by the semantically equivalent one, provided by CλaSH.*

With the second mode, the different data type notations of both the specification and the target language can be considered. CλaSH, in particular, relies on special notations for finite integer types. For this reason, the integer types from the CompCert library could not be extracted directly. Coq's integer type *Unsigned32.int* was replaced by CλaSHs integer type *Unsigned 32*. The constant integer values of the functions *clockFrequency* ($50 * 10^6$) and *delay* (15) were replaced as well by their intrinsic notations. The same applies to Gallina's *mealy* machine specification as CλaSH's intrinsic *mealy* function is needed to execute and synthesize the model. An extract of the model extracted from the traffic light controller, shown in Section 3.11, is shown in Listing 3.13.

The extracted traffic light controller specification uses a function called *topEntity*. This function does not have any further meaning in the specification but for the CλaSH model. Every hardware model that should be synthesized requires the realization of this function. It defines a root function that determines which functions and data types are synthesized into an implementation, analog to the *main* function required by programs in C or C++. The type of the *topEntity* function is inferred automatically by the CλaSH compiler.

From the extracted functional model, an implementation was automatically synthesized by CλaSH's compiler. The compiler supports the synthesis of models for the following HDLs: VHDL, Verilog, and SystemVerilog.

### 3.5.2 SYNTHESIS OF IMPLEMENTATIONS

After extracting the functional model from the specification, CλaSH's compiler synthesizes it. In contrast to SystemC, CλaSH's strong and static type system enables a unique representation of hardware designs, which subsequently enables the synthesis of such designs in an implementation. First, CλaSH requires a dedicated root function, called *topEntity*, for every hardware design model that the compiler should synthesize, as explained in Section 3.5.1. This function defines a type that has to be implemented by the hardware design. If the model does not provide the realization of that function, CλaSH's compiler cannot synthesize it. Second, the communication between the individual components is structured. The individual components only communicate by function application. This structured communication contrasts with SystemC, which provides unstructured communication [FHT06], e.g., the communication over shared memory.

CλaSH indeed has some restrictions regarding the synthesis of hardware designs, such as dynamic data-dependent recursion or value recursion, due to their infinitely

deep structure. However, CλaSH's compiler evaluates the model regarding these structures before synthesizing it. If the model contains unsynthesizable structures, it has to be revised.

To ensure that the model's semantics are correctly translated into an implementation, a synthesis scheme needs to be implemented. CλaSH implements such a synthesis scheme in the form of a Term Rewriting System [BK13]. This scheme formally denotes how terms of the implementation represent a subterm of the model.

Summarized, CλaSH's type system ensures that a model can be synthesized. It guarantees a unique representation that is automatically analyzed, resulting in the synthesis of an RTL implementation. The implemented synthesis scheme combined with Coq's extraction mechanism ensures that the specification's semantics are propagated down to the final implementation. As a result, properties that were verified for the specification hold for the model and the implementation.

## 3.6 EVALUATION

This section evaluates different hardware designs that have been specified by the proposed hardware design synthesis flow, described in Section 3.3.3. For evaluation purposes, a set of benchmarks for both combinational and synchronous sequential hardware designs have been specified and synthesized on an FPGA. The final implementations were compared to manual ones, following the established hardware design flow described in Section 3.2.1, in terms of their consumed space in *Adaptive Logic Modules* (ALMs)[3], *registers*, and *maximum clock frequency*. The aim of this evaluation is 1) illustrating the lesser implementation effort the proposed hardware design synthesis flow has and 2) showing the applicability of this flow in general.

The different benchmarks that have been implemented using both flows are described in the following in detail:

- *MAC:* This hardware design describes the combinational *multiply-and-accumulate* (MAC) circuit seen Section 3.4.2.

- *Chaser Light:* This hardware design describes a sequential chaser light that iterates over the LEDs of an FPGA.

[3] The Intel® Quartus® Prime synthesis tool uses the name *Adaptive Logic Modules* (ALMs) for the basic building blocks for hardware designs. A more familiar name might be *Lookup Tables* (LUTs), as these blocks are called by the Xilinx Vivado synthesis tool.

- *Traffic Light:* This design describes a sequential traffic light controller that considers traffic lights for cars, trams, and pedestrians. This design uses the controller, introduced in Section 3.4.3.3, as its foundation but uses bit vectors as output to access the LEDs of an FPGA.
- *Airbag Controller:* This hardware design describes a sequential airbag controller that links sensors with their corresponding airbags and triggers them independently.
- *Ticket Machine:* This hardware design describes a sequential ticket machine for a tram. It offers different kinds of tickets, e.g., for children, groups, or price savings.

### 3.6.1 Implementation Effort

This section compares the established hardware design flow's implementation effort, described in Section 3.2.1, with the implementation effort of the synthesis flow, proposed in Section 3.3.3. Every benchmark described above was implemented in the languages both design flows rely on to make this comparison. The individual implementation's implementation effort ranges from 0.5$h$ (MAC) to 8$h$ (Ticket Machine).

#### 3.6.1.1 Established Hardware Design Flow

The above benchmarks have been implemented in the following languages for the established design flow: SysML/OCL [Obj15, Obj12], SystemC [Gro02], and VHDL [Ash02], which results in three manual realizations for each benchmark. This manual implementation effort can be applied to larger hardware designs since it cannot be eliminated, as described in Section 3.2.2. The verification of the SysML/OCL specification and the generation of test benches for the SystemC model and the VHDL implementation further increase the implementation effort.

#### 3.6.1.2 Proposed Hardware Design Flow

The benchmark set for the proposed hardware design synthesis flow has to be implemented only for Coq [BC04] since the CλaSH [BKK+10] model, and the final

VHDL [Ash02] implementation has been generated automatically. The Coq specification's implementation effort has been the same for the individual implementations required by the established design flow. Even though the specification's formal verification would increase the implementation effort, the reduction remains as the hardware design has to be specified only once instead of three times, as discussed above. Evaluating the implementation effort of both the established hardware design flow and the proposed synthesis flow shows that the second decreases this effort significantly, accelerating the hardware design process. While the established one requires three manual realizations: at the FSL level, at the ESL level, and at RTL level, the one proposed in this chapter only requires one at the FSL level. The model at the ESL level and the final implementation at the RTL level are extracted, respectively, synthesized automatically from the formal specification.

### 3.6.2 Quality of the Results

Automatizing design tasks leads to an automatization overhead, i.e., the resulting designs are often not as efficient/compact as when a user implements them by hand. The final VHDL [Ash02] implementations of the above benchmark set for the established and the proposed design flow have been synthesized on an FPGA to evaluate this trade-off. More precisely, the trade-off was quantified by measuring the consumed space (ALMs/Register) and the maximum clock frequency ($F_{max}$). For synthesizing these implementations, the commercial Intel® Quartus® Prime synthesis tool was used. Table 3.1 shows the results of the evaluation.

This table shows that the proposed design flow is practical as all VHDL implementations have been synthesized on the FPGA. These implementations might consume more space since users can optimize more efficiently by hand than by tools. The *Traffic Light*'s and the *Ticket Machine*'s maximum clock frequency shows that for some implementations, the difference between those generated by the proposed hardware design synthesis flow and those generated by the established design flow is marginal.

Summarized, this evaluation shows that the proposed hardware design synthesis flow can be employed in practice. Our results show that the design synthesis flow proposed in this chapter eliminates the established design flow's problems, described in Section 3.2.2. These results answer the question: *Can proof assistants be combined with functional HDLs for automatic hardware design synthesis?*, asked in Section 3.3.2, affirmatively, for combinational and synchronous sequential hardware designs. The proposed synthesis flow cannot describe asynchronous hardware designs, as explained in Section 3.5.2. This flow automatically synthesizes a formally

**Table 3.1 :** *Evaluation by comparing the implementations of the benchmarks concerning their consumed space in Adaptive Logic Modules (ALMs) and registers, and the maximum clock frequency ($F_{max}$).*

| | **established flow** | | **proposed flow** | |
|---|---|---|---|---|
| Circuit | ALMs / Register | $F_{max}$ | ALMs / Register | $F_{max}$ |
| MAC | 61 / 0 | - | 61 / 0 | - |
| Chaser Light | 78 / 60 | 252.02 MHz | 121 / 69 | 191.46 MHz |
| Traffic Light | 163 / 45 | 231.75 MHz | 759 / 36 | 230.79 MHz |
| Airbag Controller | 128 / 110 | 240.96 MHz | 299 / 125 | 185.98 MHz |
| Ticket Machine | 747 / 196 | 74.48 MHz | 1141 / 146 | 66.36 MHz |

*Note that the* MAC *circuit is combinational. For this reason, it does neither consume registers nor has a maximum clock frequency.*

specified and verified hardware design in an RTL implementation and propagates verification results from the specification down to the final implementation. Test benches at the ESL and RTL level are no longer needed, making the described alternative design flow unsusceptible for implementation errors at these levels as no manual intervention is needed.

## 3.7 SCALABILITY

This section discusses the scalability of the proposed hardware design synthesis flow regarding the specification and the verification of hardware designs. As seen in Section 3.6.2, different hardware designs from small combinational to larger synchronous sequential ones have been specified. Furthermore, extensive specifications ,e.g., a RISC-V processor [CVS+17], have already been specified in Coq and implemented in CλaSH[4]. These hardware designs show the potential of the proposed synthesis flow regarding the formal specification and verification of hardware designs.

In general, finding a formal proof for a property is a challenging task. Automatic theorem provers such as MiniSat [ES03] or Z3 [dMB08] face the problem that solving an SAT or SMT formula is NP-complete [CES+09]. Proof assistants like Coq [BC04] or Isabelle/HOL [NPW02] are no exception. As the user guides the proof assistant through the proof, it can indeed be separated into smaller lemmas, which are easier

[4] https://github.com/adamwalker/clash-riscv

to prove, i.e., applying the *divide-and-conquer* principle. However, this separation often leads to a "lemma explosion", where the proof of a theorem requires the proof of several lemmas first. Furthermore, as the proof process is user-guided, it depends on the user's skills how fast a proof can be constructed. Once the user constructed the proof, the proof assistant can verify that proof very fast by merely executing the tactics and type check the resulting term, as described in Section 2.1. Having a user construct the proof instead of a machine leads to proving more complex theorems, such as the four-color theorem [Xia09], Kepler conjecture [HAB+15], or properties about C compilers [LBK+16]. Proving these properties by an automatic theorem prover would take far too long.

The combination of automatic theorem provers and proof assistants is investigated to help the user finding a formal proof faster in a proof assistant based setup [EMT+17, BBP11, MXHM20, CK18]. Tools that implement such a connection are called *Hammers* [BKPU16]. Hammers translate a proof assistant's subgoal into the automatic theorem solver's input language, e.g., a decidable subset of first-order logic for SMT. The solver's answer is either used as an oracle, meaning that it is blindly trusted, or rechecked by the proof assistant using proof reconstruction [BKPU16]. Proof reconstruction reduces the automatic theorem prover's answer to inferences of the proof assistant's type checker. However, these approaches try to automate finding formal proofs for a specific category, e.g., bit vectors or functional arrays [EMT+17], or finding "relatively simple" proofs using accessible lemmas [CK18]. For example, proving the following lemma: if an element $x$ occurs either in list $l1$ or in list $l2$ it also occurs in the list created by appending $l1$ to $l2$.

Due to the underlying automatic solving technique, Hammers are focused on finding proofs for lemmas or theorems expressed in a decidable subset of first-order logic [CK18]. Hammers cannot find a proof for a theorem or lemma denoted by a proof assistant in general but might be able to support the designer finding a proof in particular.

## 3.8 CONCLUSION

This chapter proposes a hardware design synthesis flow that automatically synthesizes an implementation at the RTL level from a specification at the FSL level. The established hardware design flow uses SysML/OCL [Obj15, Obj12] at the FSL level and SystemC [Gro02] at the ESL level cannot provide such an automatic translation process. It is impossible to generate an executable SystemC model from a SysML/OCL specification or synthesize an arbitrary model afterward in an RTL implementa-

tion [Inc16]. This lack of automation motivates the proposition of an alternative hardware design synthesis flow. The proposed flow addresses this problem by combining the proof assistant Coq [BC04] with the functional hardware description languages CλaSH [BKK+10]. Using Coq at the FSL level allows the formal specification and verification of hardware designs while using CλaSH at the ESL level allows to synthesize hardware design models in RTL implementations. Coq's extraction mechanism was extended to automatically generate an executable CλaSH model from a non-executable but provable specification. CλaSH describes combinational as well as synchronous sequential circuits – either by using a Mealy or a Moore machine. Thus, the proposed hardware design synthesis flow reduces the implementation effort compared to the established design flow. Since the proposed flow propagates verification results from the FSL level down to the RTL level, the generation of test benches at the ESL and RTL levels is no longer necessary. This propagation accelerates the hardware design process significantly.

Different hardware designs were used as benchmarks to evaluate the proposed hardware design synthesis flow and have been synthesized on an FPGA. The synthesis of these benchmarks shows the proposed hardware design synthesis flow's applicability and introduced it as an alternative to the established design flow. This introduction opens the door for further research in this area.

# 4

# Detecting Arithmetic Integer Overflows in Formal Specifications

The chapter's foundation is the work published by Bornebusch et al. [BLWD20a, BLWD20b]. It uses the Coq/CλaSH synthesis flow proposed in Chapter 3 to address low-level problems as arithmetic integer overflows at the FSL level. The detection of these overflows is realized by utilizing the finite integer types provided by CompCert's integer library [LBK+16]. Properties about arithmetic operations that implement these detections are formally verified and automatically propagated by the Coq/CλaSH flow from the specification down to an implementation at the RTL level.

## 4.1 Introduction

As described in Section 3.1, hardware designs are an integral part of our lives, and their complexity increases evermore. These designs are described at different abstraction levels to address their increasing complexity. These levels allow a better design understanding and enable the verification of properties. The established hardware design flow starts with a formal specification in SysML/OCL [Obj15, Obj12, DSW12], as described in Section 3.2.1. This specification is manually translated into a SystemC [Gro02, Arn00] model that is subsequently translated, in general, manually into a low-level implementation, e.g., in VHDL [Ash02].

This established hardware design flow reveals a *semantic gap* between the specification and the model, respectively, the implementation regarding integer types. The specification describes *infinite* integer types, while the model and the implementation describe *finite* ones. This semantic gap leads to verified properties that hold for the specification but not for the model, e.g., detecting arithmetic integer overflows. This semantic gap is one reason that prevents the realization of a SystemC model that is semantically equivalent to a SysML/OCL specification. Finite integer types describe a wrap-around or overflow behavior as they implement a quotient ring [DLRA15, CCF+05, CKK+12, CH13]. As a result, arithmetic operations for infinite

integer types are not semantically equivalent to arithmetic operations for finite integer types. Undetected integer overflows might lead to severe problems in the final hardware design implementation. For instance, the first Ariane-5 rocket self-destructed during its initial flight in 1996 because of an undetected integer overflow in the flight control system. Even these days, the detection of integer overflows is part of current research [DLRA15, MGE17, MMS+18].

The engineer has to detect them manually in the SystemC model due to the lack of tool support for reliable automatic detection of arithmetic overflows. An alternative hardware design flow needs to be considered to address the established hardware design flow's problem. The Coq/CλaSH flow, proposed in Chapter 3, allows the extraction of a semantic equivalent model from a formal specification, closesing the *semantic gap* between both descriptions the established flow has. This proposed flow describes finite integer types at the FSL level utilizing the CompCert integer library's dependent types [BMH07, HD92, LBK+16], allowing the realization of both signed and unsigned finite types of arbitrary sizes. As a result, operations can be specified that detect arithmetic integer overflows at the FSL level. Proven properties about these operations ensure the reliable detection of these overflows.

This chapter is structured as follows: First, it reviews the established hardware design flow to describe its problem regarding the semantic gap, in Section 4.2. Section 4.3 discusses the related work and why it is not appropriate to adequately address the established hardware design flow's semantic gap. Section 4.4 and Section 4.5 describe how the detection of arithmetic overflows are formally specified, and properties about them are formally verified. Section 4.6 describes the proposed generalizable integer overflow detection scheme. Section 4.7 evaluates the proposed approach by comparing basic arithmetic integer operations with their corresponding overflow detecting counterparts, regarding the synthesized hardware implementation's speed and consumed space. Section 4.7.3 discusses the evaluation's results, while Section 4.8 concludes this chapter.

## 4.2 MOTIVATION

This section reviews the established design flow, presented in Section 3.2 but focuses on the *semantic gap* between the SysML/OCL specification and the SystemC model. This semantic gap motivates this chapter, and it demonstrates why the combination of SysML/OCL and SystemC is a problem for the detection of arithmetic integer overflows. A traffic light controller serves as a running example to illustrate the estab-

lished hardware design flow's problem and shows how the Coq/CλaSH synthesis flow proposed in Chapter 3 addresses it.

### 4.2.1 INTRODUCTORY EXAMPLE

As described in Section 3.2.1, the established hardware design flow starts with a formal specification in SysML/OCL [Obj15, Obj12, DSW12], which can later be used to formally verify properties [BW06, CCR08, SWK+10]. The design structure is described by SysML class diagrams, while OCL constraints describe the functional behavior. These constraints are specified as preconditions and postconditions as well as invariants, as described in Section 3.2.1.

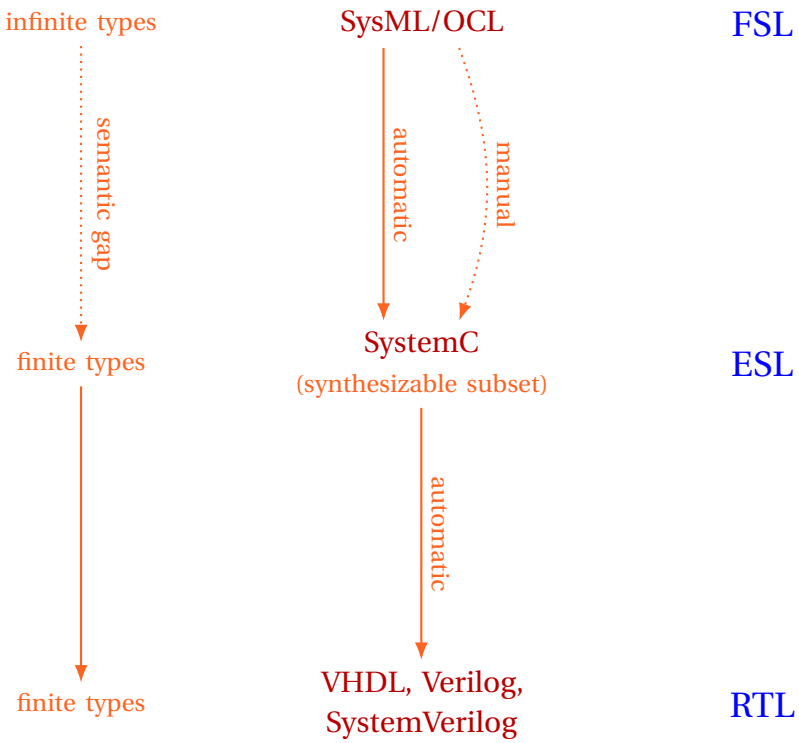


**Figure 4.1 :** *Sketched semantic gap in the established correctness-oriented hardware design. The semantic gap results from the description of* infinite *integer types used in the* SysML/OCL *specification and the description of* finite *integer types used in the SystemC model. This gap leads to properties that hold for the specification but not for the model and the implementation.*

In contrast to the previous chapter, this one investigates the *semantic gap* between the integer types used in the SysML/OCL specification and integer types used in the SystemC model. Figure 4.1 illustrates this semantic gap.

**Example 4.1.** *Figure 4.2 shows the SysML class diagram for the traffic light controller that serves as a running example in this chapter. The controller connects three different traffic lights: for the* trams*, for the* cars*, and for the* pedestrians *like the one seen in Section 3.2.1. This controller's basis is two FSMs, implemented by the* switch *and the* tick *function. The OCL constraints for these state machines can be seen in Listing 4.1.*

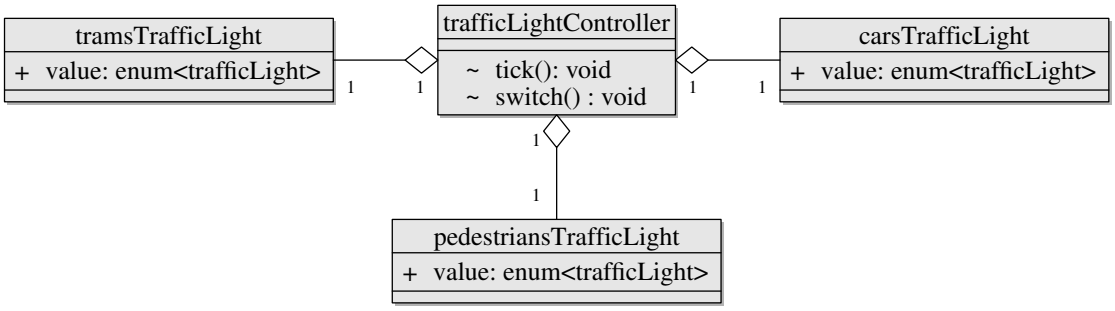


**Figure 4.2 :** *SysML class diagram of the traffic light controller. The controller connects three different lights: for the trams, for the pedestrians, and for the cars. According to a pre-defined state machine, specified by the* switch *function, the* tick *function changes the three lights and a configurable upper bound. This class diagram omits the specification of the* switch *function, as it only implements state transitions.*

*The* tick *function represents the clock in the SysML/OCL specification. As seen in Listing 4.1, it increases a* counter *and resets it to 0 if an upper bound is reached (pre_reset_counter). This* counter *is used to count the nanoseconds until the* tick *function calls the* switch *function.*

*The controller considers traffic situations, such as rush-hour. For this reason, the* delay *can be configured at runtime, which allows the configuration of a dynamic transition time. The transition time is the time the* counter *takes to reach its upper bound. Until that bound is not reached, the* counter *is increased by the* cycleTime *as the OCL constraints* pre_incr_counter *and* incr_counter *denotes. If the upper bound is reached, the* counter *is reset to 0 as indicated by* reset_counter*. In this case, the FSM implemented by the* switch *function moves into a new state where the car's traffic light is no longer green but yellow as indicated by the constraints* pre_switch *and* post_switch*. The* cycleTime *is constant and indicates the cycle time of the hardware in nanoseconds (nsec). For instance, if the transition time is 30 seconds the* delay *must be set to 1.500.000.000 with a* cycleTime *of 20 nsec.*

```
1  context trafficLightController::tick()
2   pre pre_incr_counter : self.counter  <
3                          (self.delay -1) * self.cycleTime
4   post incr_counter : self.counter = self.counter@pre +
5                       self.cycleTime and
6                       self.delay = self.delay@pre and
7                       self.cycleTime = self.cycleTime@pre
8
9  context trafficLightController::tick()
10  pre pre_reset_counter : self.counter >=
11                           (self.delay -1) * self.cycleTime
12  post reset_counter: self.counter = 0 and
13                      self.delay = self.delay@pre and
14                      self.cycleTime = self.cycleTime@pre
15
16 context trafficLightController::switch()
17  pre pre_switch : self.counter >=
18                   (self.delay -1) * self.cycleTime and
19                   self.tramsLight.value = Red and
20                   self.pedestriansLight.value = Red and
21                   self.carsLight.value = Green
22  post post_switch : self.tramsLight.value = Red and
23                     self.pedestriansLight.value = Red and
24                     self.carsLight.value = Yellow
25
26  inv: self.counter > -1
27  inv: self.delay > 0
28  inv: self.cycleTime > 0
```

**Listing 4.1 :** *OCL constraints for the* tick *function and the* switch *function introduced in Figure 4.2. Additionally, the range for the variables,* counter, delay *and* cylcleTime *is restricted by invariants.*

*The* switch *function implements the state transitions for the traffic lights. This state machine determines whether a traffic light is on or off to avoid situations such as the lights for cars and pedestrians are both green at the same time. The different states for the lights are encoded as* Green, Yellow, *and* RedYellow *and* Red. *An exemplary state transition is indicated by the* pre_switch *and the* post_switch *constraints seen in Listing 4.1. Note that the* delay *might not always be necessary for the state transition, e.g., the pedestrians might have a constant transition time while the transition time for the cars relies on the* delay *to consider situations like rush-hour. Since this chapter considers arithmetic integer overflows, the state machine is not described in detail, as no arithmetic operations are involved in the state transitions.*

A SystemC model is described after specifying the traffic light controller's behavior in SysML/OCL and its subsequent formal verification. This step is manual as indeed the SysML structure can be translated automatically into a C++ class structure, as described in Section 3.2.1. However, the functional behavior specified by OCL constraints cannot automatically be translated into executable SystemC code, as described in Section 3.3.2.

**Example 4.2.** *Listing 4.2 shows the realization of the* tick *function for the SystemC model.*

```
1  sc_uint<32> counter, delay, cycleTime;
2
3  void
4  tick()
5  {
6   if ( counter < (delay -1) * cycleTime ) {
7    counter = counter + cycleTime;
8   } else {
9    counter = 0;
10   switch_();
11  }
12 }
```

**Listing 4.2 :** *Realization of the* tick *function for the SystemC model, introduced in Listing 4.1. The state transitions are implemented by the* switch_ *function, which are omitted as they do not implement any overflow behavior.*

*As specified by the OCL constraints in Listing 4.1, the SystemC model increases the counter by the* cycleTime *until the upper bound is reached, as seen in Line 6 of*

*Listing 4.2. Otherwise, the counter is reset to 0, and the* switch *function is called. The specification's integer type* Int *was replaced by the model's integer type* sc_uint<32>. *This type was chosen as it can represent the above exemplary transition time. However, the chosen integer type for the model is independent of the considered problem, as we see in a moment. The* switch *function's realization in the model is omitted since this function only represents state transitions, as mentioned above.*

### 4.2.2 CONSIDERED PROBLEM

To illustrate the problem that motivates this chapter, we look at the safety property derived from the specification seen in Listing 4.1. This safety property holds for the specification but not for the model. This section shows why this is the case, which illustrates the considered problem of this chapter. It also shows why this is a conceptional problem and not dedicated to the chosen finite integer type in the SystemC model.

**Example 4.3.** *Listing 4.3 shows the safety property derived from the OCL constraints, seen in Listing 4.1. This property is specified as an OCL invariant.*

```
1  context trafficLightController
2   inv: self.counter < self.delay * self.cycleTime
```

**Listing 4.3 :** *Safety property derived from the OCL constraints introduced in Listing 4.1. This invariant determines that the* counter *is less than the multiplication of the* delay *and the* cycleTime. *As the SysML type* Integer *is infinite, the property holds for the specification.*

To prove that the safety property holds, we show that if the precondition, invariants, and safety property hold in the pre-state and the postcondition holds in the post-state, then the safety property holds in post-state as well. This proof consists of a case analysis of the OCL constraints for the *tick* function, seen in Listing 4.1. The notation x' is used to denote the value of the variable x in the post-state. The *self* prefix seen in the OCL constraints is also omitted.

**Example 4.4.** *To show that the safety property, seen in Listing 4.3, holds in the above specification, we look at some assumptions derived from the specification seen in Listing 4.1. We assume that the preconditions and the safety property hold in the pre-states and that the postconditions hold in the post-states.*

*We prove the safety property by case analysis. The first case considers the precondition* pre_reset_counter *and the postcondition* reset_counter. *The second case considers the precondition* pre_incr_counter *and the postcondition* incr_counter. *In the first case, the* counter *is reset to 0 in the postcondition. The invariants indicate that the* delay *and the* cycleTime *are greater than 0, so the safety property holds in the post-state. To prove the safety property for the second case, we take a look at the precondition* pre_incr_counter. *Since the monotonicity of addition holds in* $\mathbb{Z}$*, we add* cycleTime *to both sides of the precondition. This addition gives us the postcondition* incr_counter *on the left side. If we dissolve the right side, we see that the safety property holds in the post-state.*

$$
\begin{aligned}
& counter + cycleTime' < ((delay' - 1) * cycleTime') + cycleTime' \\
\Longrightarrow\ & counter + cycleTime' < (delay' * cycleTime' - cycleTime') + cycleTime' \\
\Longrightarrow\ & counter + cycleTime' < delay' * cycleTime' \\
\Longrightarrow\ & counter < delay' * cycleTime'
\end{aligned}
$$

This proof shows that the safety property holds in the SysML/OCL specification's post-states, so why does it not hold for the SystemC model? If we consider the case analysis of the proof for the SystemC model, we see that proof holds for the first case. However, for the second case, the monotonicity of addition does not hold. The SystemC model describes the quotient ring $\mathbb{Z}_{>-1}/2^{32}\mathbb{Z}$, while the specification models $\mathbb{Z}$. This quotient ring describes an integer type of limited size. That is precisely why the safety property does not hold in the SystemC model, as the monotonicity of addition does not hold for quotient rings due to their implemented wrap-around behavior;

$$
\begin{gathered}
2+2=4 \in \mathbb{Z} \\
\text{vs.} \\
2+2=0 \in \mathbb{Z}/4\mathbb{Z}
\end{gathered}
$$

In other words, the multiplication operation in the SysML/OCL specification is not semantically equivalent to the one used in the SystemC model, as in SystemC all integer types describe a quotient ring: $\mathbb{Z}/m\mathbb{Z}$, where $m \in \mathbb{Z}_{>-1}$ (signed integer) or $\mathbb{Z}_{>-1}/m\mathbb{Z}$, where $m \in \mathbb{Z}_{>-1}$ (unsigned integers). This *semantic gap* between SysML's infinite integer type and SystemC's finite integer types motivates this chapter, which utilizes the Coq/CλaSH synthesis flow allowing the description of finite integer types at the FSL level.

**Example 4.5.** *Let us consider again the translation step of the OCL constraints seen in Listing 4.1 for the SystemC model seen in Listing 4.2. The model assumes that the unsigned integer multiplication operation's implementation is the same as in the*

*specification. This assumption is understandable at first glance since the same behavior is apparently described. However, as we have seen above, this is not the case, as the integer type in the specification is infinite, while the one in the model is finite. As a result, the SystemC model violates the safety property, shown in Listing 4.3.*

*This violation bears a direct impact on the change of the configurable delay at runtime and thus on the state machine's transition time, which considers traffic situations such as rush-hour. For instance, a changed delay might lead to unintended behavior as the multiplication operation on the quotient ring* sc_uint<32> *implements a wrap-around behavior. In this case, instead of increasing the transition time, it is decreased. This unintended transition time change is a severe problem. Depending on the decreased transition time's value, the car's green phase can be too small to let even one car pass, resulting in a massive traffic jam or even accidents.*

A look in the C++ standard[1] reveals two different behaviors of integer arithmetic regarding overflows. Unsigned integer arithmetic defines total functions and does not overflow. A result that cannot be interpreted by the resulting data type is reduced by $2^n, n \in \mathbb{Z}_{>-1}$, where $n$ is the number of bits in the value representation, e.g., 32 for *sc_uint<32>*. Through the modulo operation, arithmetic operations on these data types implement a wrap-around behavior. So in the case of unsigned integer arithmetic, the execution of an arithmetic operation might lead to unintended behavior according to the C++ standard. Signed integer arithmetic does overflow and defines either total functions or partial functions, depending on the underlying hardware platform. The functions are total if the platform represents the values in the 2's complement. In this case, the same wrap-around behavior is implemented as for the unsigned integer arithmetic operations. If the platform uses traps[2] to indicate an overflow, the arithmetic function becomes partial, as in this case, the function does not define a return value for a pair of input values. As signed-integer arithmetic behavior is platform-dependent, it is considered undefined by the C++ standard.

The term *arithmetic integer overflow* often refers to both unsigned integer and signed integer arithmetic [DLRA15, CCF+05]. As described above, the distinction according to their behavior becomes blurred. For this reason, this chapter (subsequently) uses that term to address both behaviors.

The fundamental problem of the *semantic gap* between SysML's infinite integer types and SystemC's finite types motivates this chapter. A semantic equivalent finite type is needed at the FSL level to address this problem as hardware descriptions are finite by design and rely on these types. Having such types at the FSL level enables

[1] The current standard for the C++ programming language is specified in ISO/IEC 14882:2017. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/n4659.pdf

[2] A trap is a software interrupt that is triggered due to an instruction execution, e.g., division-by-zero, by the processor.

the clear distinction between the correct result of an arithmetic integer operation and the occurred overflow. This chapter calls this distinction the *detection of arithmetic overflows*. As overflows are inevitable on finite integer types, this chapter proposes an overflow detection scheme using a total function that explicitly distinguishes between the result of an arithmetic integer operation and an occurred overflow.

The following section evaluates the related work and discusses why it is not appropriate to address the problem described above. This discussion eventually leads to addressing the considered problem by the Coq/CλaSH synthesis flow, proposed in Chapter 3.

## 4.3 Related Work

This section evaluates and discusses the related work to show why arithmetic integer overflows cannot appropriately be detected neither in SysML/OCL specifications nor in SystemC models. The possibility to define finite integer types of arbitrary sizes needs to be implemented in the SysML specification to detect integer overflows. However, this is not the case in the current standard [Obj15]. Specified invariants could restrict SysMLs *Integer* type to describe a lower and an upper bound, but these bounds are independent of the integer type used in the SystemC model. For example, after the SystemC class structure's automatic generation from the SysML specification: what should the equivalent type to SysML's *Integer* type be in SystemC? Options include a standard type, like *Integer*, which is always represented as *sc_uint<32>* but prohibits bounds from being changed. Another option would be to extract the model's integer type from the described bounds mentioned above. The problem with the first option is that both types are not semantically equivalent, as described in Section 4.2.2. The second option's problem is that invariants could describe the bounds, but it is impossible to automatically extract these invariants in executable code. Furthermore, these bounds cannot be specified in any way but have to match the minimum and maximum values describable by finite integer types in the SystemC model. If these bounds are translated manually into the SystemC model, they might change during the model's development phase. For example, it was discovered that a different type, e.g., *sc_uint<31>*, is needed, invalidating the specification's integer bounds and, thus, the specification itself. The fundamental problem is that SysML/OCL specifications describe infinite integer types while SystemC models describe finite ones.

The automatic arithmetic overflow detection of C++ programs must be considered to detect integer overflows directly in the SystemC model. The detection of overflows by a C++ compiler is quite challenging because of the low-level nature of C++.

The standard allows the manipulation of bits [DLRA15]. This manipulation makes it very challenging to detect overflows by the compiler reliably, as it is not clear to the compiler whether the engineer intends such manipulation or not. Furthermore, the standard allows undefined behavior semantics that allows optimizations by the compiler [DLRA15]. For this reason, C++ compilers can only detect arithmetic overflows in constant-expression evaluation, but not in general. As a result, C++ compilers are not suitable to detect arithmetic integer overflows automatically.

Since there is no support from compilers, static code analysis tools, such as Astrée [CCF+05] or Frama-C [CKK+12], should be considered. Astrée relies on abstract interpretation [Cou12, FL10] and aims to prove the absence of runtime errors, such as integer overflows, in C programs. Abstract interpretation is used to derive a computational abstract semantic interpretation from a programming language's behavior. The resulting interpretation does not contain the actual values but focuses on dedicated parts of the program. The static analysis scope is determined by these parts and defines what kinds of errors are detected. The limit of abstract interpretation is the analysis of loops, as loops define an infinite number of paths in the interpretation tree. SystemC models are C++ programs, which is not the input language of Astrée. Astrée could, of course, be extended to support C++ programs but SystemC describes hardware designs. Such designs rely on parallel execution and run in infinite loops. As mentioned above, loops create an infinite number of paths in the interpretation tree. For this reason, Astrée is not suitable to detect integer overflows in hardware designs.

Frama-C is another static code analysis tool that relies on *C Intermediate Language* (CIL) [NMRW02] and supports annotations written in *ANSI/ISO C Specification Language* (ACSL) [CKK+12]. It applies different static analysis techniques, such as deductive verification of annotated C programs by automatic provers, e.g., Z3 [CKK+12]. The detection of integer overflows is supported by the *Runtime Error Annotation Generation* (RTE) plugin, which includes the generation of annotations by syntactic constant folding in the form of assertions. RTE seeds these annotations into other plugins, e.g., for generating weakest-preconditions with proof obligations. Like Astrée, the input language for Frama-C is a C program, which could be extended to support C++ programs. However, the static analysis of infinite loops hardware designs rely on is quite challenging. For this reason, Frama-C is not suitable to detect integer overflows in SystemC models.

As discussed in this section, a SysML/OCL specification and a SystemC model are not suitable to detect arithmetic integer overflows. The specification describes infinite types and lacks the description of finite integer types of arbitrary sizes. The model describes finite integer types, and compilers or static analysis tools do not support the reliable detection of arithmetic integer overflows. As a result, the engineer needs to detect overflows pro-actively and explicitly in ESL models.

The problem discussed in Section 4.2.2, combined with the related work, leads to the following question: *Can arithmetic integer overflows in hardware designs be detected at the FSL?*

## 4.4 Specification of Arithmetic Integer Overflow Detection

As described in Section 4.2.2, we need an explicit distinction between the correct result of an arithmetic integer operation, e.g., multiplication, and the ocurred overflow. A dependent type that either represents an operation's application or indicates an overflow is used for this distinction. Dependent types depend on an additional value, as described in Section 2.2. A dependent type that realizes this explicit distinction is *option*. This type has two constructors: *None* and *Some(A)*. The constructor *Some* requires a value of an arbitrary type A as an argument, while the constructor *None* is argument-less, as seen in Listing 4.4.

```
1  Inductive option
2      (A : Type)
3    : Type :=
4    | Some : A -> option A
5    | None : option A.
```

**Listing 4.4 :** *Definition of the* option *type in Gallina provided by the Coq standard library (Coq.Init.Datatypes). This type is dependent to an arbitrary type* A*, which is inferred at compile time.*

We define the semantic of this type as the following: the constructor *Some (A)* contains the operation's result, while the constructor *None* indicates the overflow. Consider again the running example introduced in Section 4.2.1. This example uses the basic multiplication operation that has the type:

$$n \in \mathbb{N} \Rightarrow Unsigned^n \rightarrow Unsigned^n \rightarrow Unsigned^n$$

An operation that implements this function type combines two arguments of the type $Unsigned^n$ to an output value of the same type. The $\Rightarrow$ denotes a constraint for $n$ used for the dependent type $Unsigned^n$. The $\rightarrow$ separates the function's individual parameters and return type. This dependent type describes the integer type *Unsigned*,

which depends on the type *n*, representing its length in bits. This operation cannot distinguish an overflow from the actual result. This chapter refers to arithmetic integer operations that have the above type as *basic arithmetic operations*. The *option* type extends the above function type to enable the distinction:

$$n \in \mathbb{N} \Rightarrow Unsigned^n \rightarrow Unsigned^n \rightarrow option\ Unsigned^n$$

The semantics of functions that implement this type is defined according to the semantics of *option* type: the two input parameters are combined by the arithmetic operation that implements this type. If the result does not overflow, the constructor *Some*($Unsigned^n$) is returned. This constructor contains the result of the operation. If the operation would lead to an overflow, the constructor *None* is returned. Now that we have defined the function's semantics, one question remains: *How are both cases explicitly distinguished?*

We look at the case where an overflow occurs in the unsigned multiplication to answer this question. Note that both *a* and *b* are of the type $Unsigned^n$ and $x \mapsto y$ denotes: x is transformed to y. The function *max* returns the maximum representable value of a given integer type.

$$b > 0 \wedge a * b > max(a) \mapsto b > 0 \wedge a > max(a)/b$$

The left side's condition (x) indicates the intuitive check of an overflow. Suppose the product of *a* and *b* is larger than the maximum value of the operand's integer type (max(a)) than an overflow occurred in the multiplication. However, if we implement this using finite integer types, this condition constantly evaluates to *true*, as by definition, there is no larger value of an integer type than its maximum. For this reason, we need to transform the left side to the right side (y). The right side's condition evaluates only to *true* if the multiplication of *a* and *b* is greater than *max(a)*. If the condition evaluates to *false* both operands can be multiplied safely. Using the *option* type definition described above and the transformed condition, we can implement the safe multiplication operation *safe_mult* that detects an occurred overflow, as seen in Listing 4.5. As for the SystemC model, seen in Listing 4.2, we specified a 32-bit unsigned integer value (Unsigned32.int), which was specified using the CompCert integer library [LBK+16].

Like in the SystemC example, illustrated in Listing 4.2, our multiplication is defined for 32-bit unsigned integer values. According to the semantic of the extended function type, the *safe_mult* function returns *Some(a*b)* if no overflow occurs and *None* otherwise. The *safe_mult* function replaces the basic arithmetic multiplication in our respecified *tick* function, as we see in the next section.

```
1  Definition safe_mult
2     (a b : Unsigned32.int)
3   : option Unsigned32.int :=
4   if (b >? 0%unsigned32) &&
5      (a >? (unsigned32_max / b))
6     then None
7     else Some (a*b).
```

**Listing 4.5 :** *Specified* safe_mult *function in Gallina that detects the overflow in the multiplication operation for 32-bit unsigned integer values.*

## 4.5 Proving Properties about Arithmetic Integer Overflows

To formally verify the *safe_mult* function, we first show that the multiplication defined for 32-bit unsigned integer values maps the multiplication for integer values in $\mathbb{Z}$ modulo $2^{32}$. The resulting theorem, including its proof, is shown in Listing 4.6.

```
1  Theorem multUnsigned32mapsMultZ:
2    forall a b : Z,
3    Unsigned32.mul (Unsigned32.repr a) (Unsigned32.repr b) =
4    Unsigned32.repr (a * b).
5  Proof.
6    intros.
7    apply Unsigned32.eqm_samerepr.
8    apply Unsigned32.eqm_mult.
9    - apply Unsigned32.eqm_sym.
10     apply Unsigned32.eqm_unsigned_repr.
11   - apply Unsigned32.eqm_sym.
12     apply Unsigned32.eqm_unsigned_repr.
13 Qed.
```

**Listing 4.6 :** *Theorem including its proof that the multiplication in* $\mathbb{Z}_{>-1}/2^{32}\mathbb{Z}$ *(Unsigned32) maps the multiplication in* $\mathbb{Z}$*, whose product is represented as a value in* $\mathbb{Z}_{>-1}/2^{32}\mathbb{Z}$*.*

The prefix *Unsigned32* is used as the type *Unsigned32.int* instantiates CompCerts generic type *int*, which uses an uninstantiated word size. Only the type instantiation sets this word size to a fixed value. As a result, proven theorems and lemmas for the generic type also hold for instantiated types.

This theorem denotes that for all $a$ and $b$ in $Z$, the product of their corresponding 32-bit unsigned integer values is equal to the product in $Z$ converted into a 32-bit value. To convert a value of the type $Z$ to a finite integer type, the CompCert integer library provides the function *repr*. This library already provides proofs about this function. For example, one theorem ensures that a value in $Z$ is not changed during its conversion into a finite integer type's value if the value is less than or equal to the finite type's maximum value. This theorem can be seen in detail in Appendix B.1.

To prove the *multUnsigned32mapsMultZ* theorem, proven lemmas from the CompCert library are called into action, seen in Line 7 to Line 12 of Listing 4.6. These lemmas can be seen in Appendix B.1 in detail. The first lemma applied to the subgoal is *eqm_samerepr*. The abbreviation *eqm* means: equality modulo $2^{wordsize}$, i.e., $a = b \mod 2^{wordsize}$. For our defined type Unsigned32.int the wordsize is 32. The *eqm_samerepr* lemma rewrites the subgoal:

$$Unsigned32.repr\ a * Unsigned32.repr\ b = Unsigned32.repr(a * b)$$

to:

$$Unsigned32.eqm\ (Unsigned32.unsigned(Unsigned32.repr\ a) * Unsigned32.unsigned(Unsigned32.repr\ b))\ (a * b)$$

This subgoal denotes the following: according to *eqm*, the converted product (repr) in $Z$ is equal to the product of $a$ and $b$. The function *unsigned*, provided by the CompCert integer library, converts a value of an finite integer type to $Z$. The application of lemma *eqm_mult* results into two subgoals, seen in Line 9 and Line 11 of Listing 4.6. We must show that *Unsigned32.unsigned (Unsigned32.repr a)* is equal to $a$ according to *eqm* to solve the first subgoal. The lemma *eqm_sym* shows the symmetry between both values, so we can use it to swap them. The lemma *eqm_unsigned_repr* solves the first subgoal. This lemma proves that according to *eqm* the conversion of a value in $Z$ to a finite integer type does not change the value. The second subgoal is solved analogously to the first one.

After proving that the multiplication in $Z$, whose product is represented as an unsigned 32-bit value, maps the multiplication in $Unsigned^{32}$, we show that *safe_mult* detects the overflow according to the above-transformed condition. Two theorems

are formulated to prove this detection based on the if condition's branches: The occurred overflow's detection and returning the product of two unsigned 32-bit values if no overflow occurs. The theorem that detects the arithmetic overflow is shown in Listing 4.7.

```
1  Theorem detectOverflow:
2    forall a b : Unsigned32.int,
3    b >? 0 = true /\ (a >? unsigned32_max / b) = true ->
4    safe_mult a b = None.
5  Proof.
6    intros.
7    destruct H.
8    unfold safe_mult.
9    rewrite H.
10   rewrite H0.
11   reflexivity.
12 Qed.
```

**Listing 4.7 :** *Theorem including its proof specified in Gallina and $\mathcal{L}_{tac}$ to verify that the specification of the* safe_mult *function detects the overflow in the 32-bit unsigned integer multiplication.*

The theorem *detect_overflow* denotes the following: if *b* is greater than zero and *a* is greater than *unsigned32_max* divided by *b*, the *safe_mult* function returns *None*. The *>?* sign seen in Line 3 of Listing 4.7 denotes *greater than* returning a boolean. The *>* sign is of type *Prop*. Since the *safe_mult* function uses the boolean operation, the *>?* sign is used in this case. Furthermore, we must determine the *>?* operation's return value (true) so that Coq can use it as a proposition. To prove this theorem, we must show that under the assumption the proposition:

$$b >? 0 = true \wedge a >? unsigned32\_max/b = true$$

holds, the *safe_mult* function's if-condition evaluates to *true*, i.e. *None* is returned. To do this, we first destruct the hypothesis H (the above proposition) into two hypotheses – H and H0. H represents the left side of the ∧ sign, and H0 the right side. The next step is to rewrite the current subgoal with these hypotheses. This rewriting reduces the subgoal to *None* = *None*, which the *reflexivity* tactic solves.

After proving that the integer overflow in the multiplication is detected, we need to prove that no overflow occurs when the two operand's product is returned (Some (a*b)). The corresponding theorem is shown in Listing 4.8.

```
1  Theorem noOverflow:
2    forall a b : Types.Unsigned32.int,
3    a >? unsigned32_max / b = false -> safe_mult a b = Some(a*b).
4  Proof.
5    intros.
6    unfold safe_mult.
7    rewrite H.
8    destruct (b >? 0).
9    - reflexivity.
10   - reflexivity.
11 Qed.
```

**Listing 4.8 :** *Theorem including its proof specified in Gallina and $\mathcal{L}_{tac}$ to verify that the specification of the* safe_mult *function returns the product of the two 32-bit unsigned integer values.*

The theorem *no_overflow* denotes the following: if the proposition

$$a >? unsigned32\_max / b = false$$

holds, the *safe_mult* function returns *Some(a*b)*. This proposition does not consider *b*, as the right side of the if condition's && operation evaluates to *false*. The proof script destructs the condition (b >? 0), resulting in one case for *true* and *false*. Coq simplifies these cases automatically based on hypothesis *H*, representing the above proposition. Through this simplification, both subgoals are reduced, and the *reflexivity* tactic solves the resulting subgoals: *Some (a * b) = Some (a * b)*.

After specifying and verifying an overflow detecting multiplication operation for 32-bit unsigned integer values, we can respecify the *tick* function described in Listing 4.1 in Gallina using this operation. Since this function also uses the addition operation, in which an overflow can occur, we must transform this operation to its overflow detecting counterpart first. We follow the transformation pattern introduced in Section 4.4 to specify a condition that detects an overflow in the 32-bit unsigned integer addition operation:

$$a + b > max(a) \mapsto a > max(a) - b$$

The $\mapsto$ sign's right side describes the condition we use in the specified *safe_add* function, seen in Listing 4.9.

The specified theorems including their proofs for the *safe_add* function can be seen in Appendix B.1. They follow the same pattern used for the *safe_mult* function specification described above.

```
1  Definition safe_add
2      (a b : Unsigned32.int)
3    : option Unsigned32.int :=
4    if a >? (unsigned32_max - b)
5      then None
6      else Some (a+b).
```

**Listing 4.9 :** *Specified* safe_add *function in Gallina that detects the overflow in the addition operation for 32-bit unsigned integer values.*

We use the *safe_mult* and the *safe_add* functions to replace the respecified *tick* function's basic arithmetic operations, as seen in Listing 4.10. Thus, the *tick* function propagates the overflows in the multiplication and addition operation to its calling function. If no overflow occurs, the counter is increased or reset to zero, as defined by the OCL constraints seen in Listing 4.1.

```
1  Definition switch (s : State) : State.
2
3  Definition tick
4     (input:Unsigned32.int*Unsigned32.int*Unsigned32.int*state)
5   : option Unsigned32.int*state :=
6   match input with
7     | (counter, delay, cycleTime, states) =>
8        match (safe_mult (delay -1%unsigned32) cycleTime,
9               safe_add counter cycleTime) with
10       | (Some resMult, Some resAdd) =>
11           if counter <? resMult
12             then (Some(resAdd), states)
13             else (Some(0%unsigned32), switch states)
14       | (_,_)     => (None, states)
15       end
16  end.
```

**Listing 4.10 :** *Respecification of the* tick *function in Gallina, introduced in Section 4.2.1. The respecified function uses the* safe_mult *function introduced in Listing 4.5 and the* safe_add *function introduced in Listing 4.9.*

Note that the *switch* function, seen in Figure 4.2, has a different type in Gallina. Pure functional languages, such as Gallina, do not allow internal states, in contrast to a SysML specification. For this reason, the type of the *switch* function had to be changed. This chapter considers the detection of arithmetic integer overflows, and there are no arithmetic operations involved in the *switch* function's state transitions; this section omits its realization. Since the multiplication and addition operations have semantically changed compared to the operations used in the SysML/OCL specification, one question arises: *How to handle an arithmetic integer overflow?*

The answer to this question highly depends on the environment the traffic light controller runs in, e.g., return to a safe state or ignore the newly configured delay. As this would be out of scope for this chapter, the *tick* function returns an instance of the tuple *option Unsigned32.int*State*. The tuple's first value contains an instance of the type *option Unsigned32.int*, while the tuple's second value defines the new state. This state can either be the old one or the one changed by the *switch* function. The overflow is not handled by this function directly but is propagated to the calling function instead. The state remains unchanged in this case.

The SysML/OCL specification, described in Section 4.2.1, satisfies the derived safety property described in Section 4.2.2. Since this safety property argues about infinite integer types instead of finite ones, we cannot adopt it one-to-one but have to change it first. The problem illustrated in Section 4.2.2 shows that an overflow occurs in the SystemC model but not in the SysML/OCL specification. We take this as an opportunity to formulate a new safety property that considers the overflow directly in the specification. Two theorems express this safety property. The first theorem is shown in Listing 4.11 and describes the case no overflow occurs either in the multiplication or in the addition. The second theorem is shown in Listing 4.12 and describes when an overflow occurs in the multiplication or in the addition. Proof of those two theorems verifies that the *tick* function either changes the *counter* or propagates the detected overflows.

The first theorem denotes: if no overflow occurs in the multiplication of *delay -1* and *cycleTime* and in the addition of *counter* and *cycleTime*, the *tick* function returns the new counter *counter'* and the new state *state'*. The new counter is either *counter* + *cycleTime* or *0*. Note that the new state might be the old state as it depends on the if condition whether the state is changed or not, as seen in Listing 4.10.

We prove this theorem by the if condition's case analysis. The hypothesis conjunction seen in Line 5 of Listing 4.11 is separated into two hypotheses *H1* and *H2*, by the *destruct* tactic. The term seen in Line 7 of Listing 4.11 represents the hypothesis *H0*. Unfolding the *tick* function in *H0* and rewrite it by *H1* and *H2* reduces *H0* to the if condition seen in Line 11 of Listing 4.10. The destruction of this condition results in two subgoals, one for each branch. The first subgoal considers the left part of the

```
1  Theorem tickFunctionNoOverflow:
2    forall counter counter' delay cycleTime : Unsigned32.int,
3    forall state state' : state,
4    let resMult := (delay -1) * cycleTime in
5    safe_mult (delay -1) cycleTime = Some(resMult) /\
6     safe_add counter cycleTime = Some(counter + cycleTime ) ->
7    tick (counter, delay, cycleTime, state) = (Some (counter'), state') ->
8    counter' = counter + cycleTime \/ counter' = 0%unsigned32.
9  Proof.
10   intros.
11   destruct H as [H1 H2].
12   unfold tick in H0.
13   rewrite H1 in H0.
14   rewrite H2 in H0.
15   destruct (counter <? resMult) in H0.
16   - left.
17     inversion H0.
18     reflexivity.
19   - right.
20     inversion H0.
21     reflexivity.
22 Qed.
```

**Listing 4.11 :** *Safety property, specified as a theorem in Gallina, including its proof specified in $\mathscr{L}_{tac}$. It describes the change of the counter if no overflow occurs, neither in the multiplication nor in the addition. The notation x' denotes the changed variable x returned by the* tick *function.*

conclusion's disjunction seen in Line 8 of Listing 4.11. The *inversion* tactic allows the deduction of equalities that must be true based on the given equality between two constructors. For instance, it concludes from the Hypothesis $f\,a = f\,b$ that $a = b$. Using this tactic, we can conclude the new hypothesis: *counter + cycleTime = counter'*. Furthermore, the *inversion* tactic tries to rewrite the current subgoal with this new hypothesis. As a result, the subgoal

$$counter' = counter + cycleTime$$

is rewritten to:

$$counter + cycleTime = counter + cycleTime$$

and the hypothesis *H3 : counter + cycleTime = counter'* is added to the subgoal's context. This rewriting results in showing that *counter + cycleTime* is equal to itself. The second subgoal considers the right part of the conclusion's conjunction seen in Line 8 of Listing 4.11. Proof of this subgoal works similarly to the first one. The *inversion* tactic allows the deduction that *0%unsigned = counter'* which reduces the second subgoal to solve *0 = 0*.

The second theorem denotes: if the product of *delay -1* and *cycleTime* or the sum of *counter* and *cycletime* is *None*, the *tick* function returns *None*. The state remains unchanged in this case.

We prove this theorem by case analysis on the hypothesis *H*, seen in Line 4 of Listing 4.12. The destruction of this hypothesis generates two subgoals – one for the left side of the disjunction and one for the right side. For the first subgoal, hypothesis *H* represents the left side, and for the second subgoal, hypothesis *H* represents the right side. The first subgoal shows that if *safe_mult* returns *None*, the *tick* function returns *None*. The second subgoal shows the same behavior for the *safe_add* function. Rewriting the first subgoal with *H* reduces it to *(None, state0) = (None, state0)*, the *safe_mult* function is firstly evaluated by the *tick* function. Analog to solving the first subgoal, we first rewrite the second one using hypothesis *H* to solve it. Since the *safe_add* function is called second, we need to destruct the *safe_mult* function. This function is no longer part of the subgoal's context, so we cannot use it for rewriting the subgoal. The destruction of *safe_mult* leads to two subgoals based on its return type – one for the *Some(A)* constructor and one for the *None* constructor. Due to the *tick* function's specification – both terms in the *match* statement must use the constructor *Some(A)* to return something other than *None* –, the result for both constructors is *None*, as the hypothesis *H* denotes that *safe_add* returns *None*. As a result, the equation's left side is reduced to match the right side for the two subgoals.

Summarized, in this section, we have seen how to specify a safe multiplication and addition operation using *dependent types* to detect an overflow in these operations for

```
1  Theorem tickPropagateOverflow:
2    forall counter delay cycleTime : Unsigned32.int,
3    forall state : state,
4    safe_mult (delay -1) cycleTime = None \/
5      safe_add counter cycleTime = None ->
6    tick (counter, delay, cycleTime, state) = (None, state).
7  Proof.
8    intros.
9    unfold tick.
10   destruct H.
11   - rewrite H.
12     reflexivity.
13   - rewrite H.
14     destruct safe_mult.
15     reflexivity.
16     reflexivity.
17 Qed.
```

**Listing 4.12 :** *Safety property, specified as a theorem in Gallina, including its proof in* $\mathscr{L}_{tac}$*. It describes the propagation of an overflow occured either in the multiplication or the addition. The* tick *functions returns* None *in this case and the current state is not changed.*

32-bit unsigned values at the FSL level. The *tick* function's specification, shown in Listing 4.1, had to be respecified in Gallina manually. The SysML/OCL's safety property, shown in Listing 4.3, had to be transformed, as this property argument about infinite integer types rather than finite ones. Proving this transformed property verifies that the respecified *tick* function propagates the detected overflows through the hardware design. The *tick* function's respecification and the proved transformed safety property shows that this section successfully addressed the problem of missing overflow detection at the FSL level the established hardware design flow has, as described in Section 5.2.

## 4.6 PROPOSED INTEGER OVERFLOW DETECTION SCHEME

Based on the arithmetic integer overflow detection pattern used in Section 4.4, this section proposes an overflow detection scheme that generalizes that pattern. This scheme defines the semantics of a total function that distinguishes the arithmetic integer operation's result from an overflow by the *option* type described in Section 4.4. The proposed detection scheme is shown in Listing 4.13.

```
1  data option A = None | Some A
2
3  f : A  →  A  →  option A
4  f x y = if p(x, y)
5            then None
6            else Some(op(x,y))
```

**Listing 4.13 :** *Proposed overflow detection scheme. The function* f *distinguishes the arithmetic integer overflow from the performed arithmetic operation by the* option *type.*

The scheme requires the following definitions. First, a data type defining two constructors: *None* and *Some (A)*, where *A* defines the finite integer types: Unsigned$^n$ or Signed$^n$, where $n \in \mathbb{N}$. *op* denotes the arithmetic integer operation function *f* performs. Second, a function *f* that implements the following type:

$$A \rightarrow A \rightarrow option\ A$$

This function takes two arguments of the finite integer type $A$ and returns a value of the previously defined *option* type, seen in Section 4.4. That section defines this type's semantics as follows: the constructor *None* indicates the overflow and the constructor *Some (A)* indicates the result of the performed operation. Last, a predicate $p$ to perform the case analysis on:

$$p(x,y) = \begin{cases} true, & \text{overflow detected for x and y} \\ false, & \text{otherwise} \end{cases}$$

This predicate realizes the overflow detection condition for a particular operation, e.g., the condition defined in Section 4.4 for the multiplication of 32-bit unsigned integer values. If the overflow is detected for a particular operation *true* is returned and *false* otherwise.

Summarized, the proposed arithmetic overflow detection scheme can be used for any arithmetic operation that is applied to finite integer types. It only requires a predicate $p$ called by a function $f$ to distinguish an overflow from the result of an arithmetic integer operation using the *option* type. As a result, function $f$ is used to replace its basic arithmetic counterpart that does not allow this distinction.

### 4.6.1 PROVING PROPERTIES

Two theorems must be proven to verify that the function $f$ distinguishes the overflow from the multiplication's result. One for each constructor of the predicate $p$'s return value; *true* and *false*. Proof of the first theorem, seen in Theorem 1, formally verifies that *None* is returned for each input pair which causes an overflow for the performed arithmetic operation.

**Theorem 1** (Overflow in integer arithmetic operation)**.** $\forall x, y \in A$, *where* $A$ *is a set defined by an arbitrary finite integer type, and* $p$ *is the predicate defined above.*

$$p(x,y) = true \implies f(x,y) = None$$

Proof of the second theorem, seen in Theorem 2, formally verifies that for all inputs that do not cause an overflow for the performed arithmetic operation, the result of this operation is returned.

**Theorem 2** (No overflow in arithmetic integer operation)**.** $\forall x, y \in A$, *where* $A$ *is a set defined by an arbitrary finite integer type and,* $p$ *is the predicate defined above.*

$$p(x,y) = false \implies f(x,y) = Some(A)$$

These two theorems ensure that the specified function $f$ returns the constructor *None* only in the case of an occurred overflow. The constructor *Some (A)* is only returned if no overflow occurs and contains the performed operation's result.

### 4.6.2 EXCEPTIONS

This section indicates cases in which the overflow detection scheme is necessary and when it can be avoided. If we look at the general behavior of operations applied to finite integer types, an overflow might always occur. The result of such an operation can potentially be larger than its finite integer type is able to represent. In general, the overflow detection scheme described above should be applied.

However, there are cases where this scheme can be avoided. This avoidance might be the case when the potential input values are restricted, e.g., by the external environment. It has to be verified that the applied arithmetic operation never causes an overflow having such restrictions. Theorem 3 shows a property that must be proven to ensure the above behavior.

**Theorem 3** (Overflow detection operation can be avoided)**.** $\forall x, y \in A'$, *where* $A' \subset A$ *and* $A$ *is a set defined by an arbitrary finite integer type.*

$$f(x, y) = Some(A)$$

If this theorem holds, then it is unnecessary to replace the basic arithmetic operation with the overflow detecting one defined by function $f$. The proposed scheme's general steps are the following: First, define function $f$, as described above, for the desired finite integer type and operation. Second, proof of Theorem 3 to verify that the chosen subset of input values is never too large to cause an overflow. Proof of this theorem is not the case for every $A$' but might be for a particular one, e.g., if $f$ returns the product of $x$ and $y$:

$$f(x, y) = Some(A), \text{ where } A = \mathbb{Z}/2^{32}\mathbb{Z} \text{ and } A' = \mathbb{Z}/2^{16}\mathbb{Z}$$

Imagine we have proof that the arithmetic operation defined by function $f$ never returns an overflow. In this case, the following question arises: *Can this proof be used to replace function* f *in a specification by its corresponding basic arithmetic operation automatically?*

To answer this question, we look again at Coq's extraction mechanism, described in Section 3.5.1. The extraction process only extracts the functional behavior, written in Gallina, in an executable target language. Theorems and Lemmas, which denote

propositions, are erased in the extraction process. Thus, it is impossible to automatically replace the defined function *f* by its corresponding basic arithmetic operation by providing a proof of theorem 3. Coq's entire extraction mechanism must be changed to allow such an automatic replacement. Furthermore, function *f*'s type would change from $A \rightarrow A \rightarrow option\ A$ to $A \rightarrow A \rightarrow A$, because of this replacement, as described in Section 4.4. This change results in a function that is not semantically equivalent to the original one. As a result, such a replacement would affect the entire specification recursively, including all theorems using function *f*. The designer must change theorems and proof scripts to respond to the new function type. A more suitable way to address this challenge is to propagate the constructor *Some (A)* returned by function *f* through the specification. This propagation avoids the recursive changing of all functions depending on *f* manually as the type of function *f* remains the same, i.e., $A \rightarrow A \rightarrow option\ A$.

Summarized, if Theorem 3 holds, we have proof that the specification of function *f* can be changed to constantly return *Some(op(x,y))* as no overflow occurs.

### 4.6.3 Closure of Functions

Since the overflow detection scheme proposed in this chapter has the function type $A \rightarrow A \rightarrow option\ A$, functions that implement this scheme are no longer closed. A set is called closed under an operation if an operation applied on members of a set always produce a member of the same set. For this reason, it is not possible to cascade these functions, e.g., *safe_mult (safe_mult 3 4) 5*. The *option* monad was specified to address this problem. Monads come from the mathematical field of category theory and model computations [Wad95]. They are used as a design scheme in functional languages and represent a specific form of computation. Analog to monad's realization in other functional languages, e.g., in Ocaml, two functions were implemented, seen in Listing 4.14.

Since these function's cascading might not always be necessary, this monad is proposed instead of changing the proposed scheme, seen in Listing 4.13. The *option* monad allows greater flexibility between both use cases. The functions of this monad are shown in Listing 4.14.

The *option* monad contains two functions: *ret* and *bind*. The *ret* function takes a value *x* of the polymorphic type *A* and transforms it into a value of *option A*. The *bind* function takes a function *f* of the proposed overflow detection scheme's function type and the two input arguments *x* and *y* of type *option A*. If both arguments contain a value of the type *A*, the function *f* is called with these values. Otherwise, [5 *None* is

```
1  Definition ret {A : Type}
2    (x : A)
3  : option A :=
4    Some x.
5
6  Definition bind {A : Type}
7     (f : A -> A -> option A)
8     (x y: option A)
9   : option A :=
10   match (x, y) with
11    | (Some x', Some y') => f x' y'
12    | (_,_) => None
13   end.
```

**Listing 4.14 :** *Specification of the* option *monad in Gallina consisting of two functions. The* ret *function transformes a value of type* A *to the type* option A. *The* bind *function applies the function* f *to the two input arguments* x *and* y.

returned. The *option* monad applies to all functions that turn two arguments of type *A* to a value of type *option A*. The two arguments, *x* and *y*, and the return type must be of type *option A*. Since the *ret* and *bind* functions are polymorphic, they are not restricted to a dedicated type but any finite integer type.

To verify the *bind* function's correct behavior, we prove two theorems – representing the two cases treated by the function. The first theorem, seen in Listing 4.15, denotes that if both arguments *x* and *y* contain values of type *Some(A)*, then the *bind* function calls function *f* with their corresponding values *x'* and *y'*, as seen in Listing 4.15. This theorem ensures that only when both arguments for *f* contain computable values, this function is called. It derives from the *bind* function's first case.

We prove the theorem by applying the hypothesis conjunction's terms, seen in Line 6 of Listing 4.15, to the *bind* function. By definition, if the bind function's arguments *x* and *y* are both of type *Some (A)*, function *f* is called. The proof is accomplished by applying the same tactics already described in Section 4.5.

The second theorem, seen in Listing 4.16, denotes that if either the argument *x* or the argument *y* is *None*, the *bind* function returns *None*, as seen in Listing 4.16. This theorem ensures that function *f* is not called with incomputable values. It derives from the *bind* function's second case.

```
1  Theorem fIfSome:
2    forall (A : Type),
3    forall f : (A -> A -> option A),
4    forall x y : option A,
5    forall x' y' : A,
6    x = Some (x') /\ y = Some (y') ->
7    bind f x y = f x' y'.
8  Proof.
9    intros.
10   unfold bind.
11   destruct H as [H1 H2].
12   rewrite H1.
13   rewrite H2.
14   reflexivity.
15 Qed.
```

**Listing 4.15 :** *Theorem that verifies that the function* f *is only called by the* bind *function if both arguments are of type* A.

We prove this theorem by case analysis on the hypothesis, similar to the proof shown in Listing 4.12. This hypothesis's destruction results in two subgoals; one for the left side of the disjunction and one for the right side. Since *x* is the first argument, the *bind* function performs case analysis on, as seen in Listing 4.14, rewriting the subgoal by the hypothesis *H* reduces the left side of the equation to match the right side. This reduction solves the first subgoal. The second subgoal is reduced first by rewriting it using hypothesis *H*. As this hypothesis only contains the right part of the disjunction (x = None), the destruction of *x* is required. This destruction results in two subgoals – one for each *option* type's constructor. By the *bind* function's definition, the arguments *x* and *y* must be of type *Some (A)* to return something other than *None*. As the hypothesis denotes *y* is *None*, the two subgoals are reduced to show that *None* is equal to itself.

Summarized, the *option* monad closes operations that implement the proposed overflow detection scheme, which allows the cascading of these functions, e.g., *bind safe_mult (bind safe_mult (ret 3) (ret 4)) (ret 5)*. The cascading of operations enables the formulation of more complex operations based on applying overflow detecting arithmetic operations. The *bind* function propagates an occurred overflow through

```
1  Theorem noneIfNone:
2    forall (A : Type),
3    forall f : (A -> A -> option A),
4    forall x y : option A,
5    x = None \/ y = None -> bind f x y = None.
6  Proof.
7    intros.
8    unfold bind.
9    destruct H.
10   - rewrite H.
11     reflexivity.
12   - rewrite H.
13     destruct x.
14     reflexivity.
15     reflexivity.
16 Qed.
```

**Listing 4.16 :** *Theorem that verifies that in the case of invalid arguments for function* f None *is returned by the* bind *function.*

the cascaded operations. At the end of the calculation, it can be evaluated whether the result is correct or an overflow occured in one of the operations.

## 4.7 EVALUATION

The evaluation's foundation presented in this section compares basic arithmetic integer operations with their corresponding overflow detecting operations regarding their impact on the maximum clock frequency and consumed space. The operations were specified for both signed and unsigned arithmetic integer operations and used by the traffic light controller, described in Section 4.5, to determine these values. First, the overflow detecting operations are introduced in this section. Second, the final implementations synthesized on an FPGA using the commercial Intel® Quartus® Prime synthesis tool are evaluated.

### 4.7.1 INTEGER OVERFLOW DETECTION SPECIFICATIONS

In addition to the integer overflow detecting operations, introduced in Section 4.4 and Section 4.5, this section introduces arithmetic signed integer operations used for evaluation. These operations are defined for 32-bit signed integer values and follow the overflow detection scheme introduced in Section 4.6.

There are multiple conditions needed to cover all possible overflow detection cases. Since signed integer values are either negative or positive, both the minimum representable and maximum representable values must be considered. The function *Signed32.min_signed* represents the minimum representable value, and the function *Signed32.max_signed* represents the maximum one for the signed 32-bit integer type *Signed32*.

Listing 4.17 shows the operation's specification that detects an arithmetic overflow in the multiplication of two signed 32-bit values.

Listing 4.18 shows the operation's specification that detects an arithmetic overflow in the addition of two signed 32-bit values.

```
1  Definition safe_mult_signed
2     (a b : Signed32.int)
3   : option Signed32.int :=
4   if (a >? 0%signed32) &&
5      (b >? 0%signed32) &&
6      (a >? (Signed32.max_signed / b))
7      then None
8      else if (a >? 0%signed32) &&
9              (b <? 0) &&
10             (a <? (Signed32.min_signed / b))
11      then None
12      else if (a <? 0%signed32) &&
13              (b >? 0%signed32) &&
14              (a <? (Signed32.min_signed / b))
15     then None
16     else if (a <? 0%signed32) &&
17             (b <? 0%signed32) &&
18             (a >? Signed32.max_signed / b)
19      then None
20      else Some(a*b).
```

**Listing 4.17 :** *Definition of the* safe_mult_signed *function in Coq that detects an overflow in the multiplication operation for signed 32 bit values.*

```
1  Definition safe_add_signed
2     (a b : Signed32.int )
3   : option Signed32.int :=
4   if (a >? 0%signed32) &&
5      (b >? 0%signed32) &&
6      (a >? (Signed32.max_signed - b))
7      then None
8      else if (a <? 0%signed32) &&
9              (b <? 0%signed32) &&
10             (a <? (Signed32.min_signed -b))
11      then None
12      else Some (a+b).
```

**Listing 4.18 :** *Definition of the* safe_add_signed *function in Gallina that detects an overflow on the multiplication operation for signed 32 bit values.*

### 4.7.2 RESULTS

This section compares the different implementations that detect an arithmetic integer overflow proposed in this work regarding their consumed space in *ALMs*, *registers*, and *maximum clock frequency*. This comparison's foundation is the traffic light controller's implementation, shown in Section 4.5. Table 4.1 shows the result of final implementation's comparison.

As seen in Table 4.1, the consumed space in the form of ALMs and registers differs slightly, except for the unsigned multiplication operation, which we discuss below. The maximum clock frequency also slightly differs with a maximum difference of 10 MHz for the signed addition operation.

The overflow detecting unsigned multiplication operation has a significantly larger amount of ALMs. During the synthesis process, this operation's integer division is replaced by the *lpm divide* megafunction. Megafunctions are Programmable Logic Devices (PLD) that describe a specific functionality, e.g., integer multiplication. These functional blocks are ready-made, pre-tested, and augment hardware designs, so the functionality does not have to be implemented again. It seems, for unsigned integer division, the Intel® Quartus® Prime synthesis tool includes the *lpm divide* block automatically, while for signed integer division, it does not. This decision is based

**Table 4.1 :** *Evaluation by comparing the consumed space in ALMs and registers (Regs) and the maximum clock frequency ($F_{max}$) for signed 32 and unsigned 32 integer operations (operation) used by the traffic light controller, described in Section 4.5. The* basic operation *column contains the values for the basic arithmetic operations, while the* overflow detection *column contains the values for the arithmetic operations using the proposed detection scheme.*

| | **basic operation** | | **overflow detection** | |
|---|---|---|---|---|
| operation | ALMs/Regs | $F_{max}$ | ALMs/Regs | $F_{max}$ |
| unsigned mult | 92 / 36 | 72.20 MHz | 670 / 36 | 65.51 MHz |
| unsigned add | 81 / 36 | 111.76 MHz | 112 / 36 | 109.57 MHz |
| signed mult | 112 / 36 | 68.19 MHz | 122 / 36 | 68.84 MHz |
| signed add | 81 / 36 | 119.82 MHz | 148 / 36 | 109.24 MHz |

*Consumed space and maximum clock frequency synthesized for the Cyclone V family using the commercial synthesis tool Intel® Quartus® Prime. Note that the abbreviation* unsigned mult *means unsigned multiplication, the abbreviation* unsigned add *means unsigned addition, the abbreviation* signed mult *means signed multiplication, and the abbreviation* signed add *means signed addition.*

on the analysis of the RTL code. For this reason, the amount of ALMs is significantly higher.

### 4.7.3 DISCUSSION

This section discusses the proposed arithmetic integer overflow detection scheme and the results of the evaluation.

The overflow detection scheme, proposed in Section 4.6, leads to arithmetic operations that are no longer closed since their input type is no longer the output type. This difference in types prevents the operation's cascading, which is possible with their corresponding basic arithmetic operations. The proposed monad for the *option* type, seen in Section 4.6.3, addresses this issue. This monad closes operations realizing the proposed arithmetic overflow detection scheme. The closure of functions enables the cascading of those operations, which results in the description of more complex calculations similar to the cascading of basic arithmetic operations.

According to the evaluation results, the speed and consumed space's impact depends on the replaced basic arithmetic integer operation and the integer type. The

difference between the basic arithmetic operations and their corresponding overflow detecting operations for unsigned addition and signed multiplication is even negligible.

In general, this opens a trade-off between safety-oriented and performance-oriented hardware designs. The added overflow detection has an impact on the consumed space or the maximum clock frequency. However, it depends on the concrete hardware design whether the safety aspect is important enough to accept this impact or not. The acceptance of this impact might not always be the case, and the concrete values regarding the speed and consumed space highly depend on the chosen FPGA. Note that the arithmetic integer operations in larger hardware designs represent only a tiny part of the complete functionality. In this case, the overflow detecting arithmetic operation's impact compared with their corresponding basic arithmetic operations becomes negligible.

This discussion shows that the overflow detection scheme proposed in this chapter is applicable, and the question: *Can arithmetic integer overflows in hardware designs be detected at the FSL level?*, asked in Section 4.3, is answered in the affirmative. The maximum frequency and the consumed space for the overflow detecting arithmetic operations are slightly slower or even negligible. The only exception is the overflow detecting unsigned multiplication operation. The Quartus Prime synthesis tool chooses to use the *lpm divide* megafunction automatically during the synthesis process, which was omitted for the other operations. However, even in this case, the maximum clock frequency is only slightly slower than in the other implementations used for the evaluation.

## 4.8 SUMMARY

This chapter addressed the *semantic gap* between the infinite integer types of a SysML/OCL specification and a SystemC model's finite integer types. This *semantic gap* might lead to arithmetic overflows in the model that are unknown in the specification, as explained in Section 4.2.2. This gap motivates this chapter, and it was addressed by the hardware design synthesis flow proposed in Section 3.3.3.

This flow uses the proof assistant Coq [BC04, Chl13] combined with the CompCert integer library [LBK+16] to address this semantic gap. The CompCert Integer library allows the description of both signed and unsigned finite integer types of arbitrary sizes as *dependent types* [HD92, BMH07]. This library was utilized to describe finite integer types in Coq. This description enables the specification of arithmetic integer operations that provable detect overflows, as described in Section 4.5. These descrip-

tions result in a generalizable scheme for detecting overflows in arithmetic integer operations. Furthermore, this chapter proposes a method to close the functions that implement the proposed detection scheme described in Section 4.6. This method allows the cascading of operations implementing the proposed overflow detection scheme, analog to their corresponding basic arithmetic operations to describe more complex calculations.

This chapter evaluates the proposed overflow detection scheme by comparing basic arithmetic operations with their corresponding overflow detecting operations in terms of their maximum clock frequency and consumed space. These values were gathered from an FPGA synthesis process, explained in Section 5.5. This evaluation shows a trade-off between safety-oriented and performance-oriented hardware designs, as additional safety checks have an impact on the consumed space and maximum clock frequency. However, for some synthesized implementations, this trade-off is even negligible, which shows the proposed overflow detection scheme's applicability for hardware designs that consider performance.

# 5

# Case Study: Performance Aspects of a Formally Specified 32-bit MIPS Processor

This chapter's foundation is the work published by Bornebusch et al. [BLWD21]. It analysis performance aspects of correctness-oriented hardware design synthesis flows. For this reason, a formal 32-bit MIPS processor design specification synthesized by the Coq/CλaSH flow, proposed in Chapter 3, is compared in a case study with the processor implementation synthesized by the state-of-the-art acceleration-oriented HLS framework LegUp. This comparison's results show the Coq/CλaSH flow's potential regarding the synthesis of correctness-oriented hardware implementations, which also consider performance.

## 5.1 Introduction

As discussed in Chapter 3, hardware designs become more and more complex over time. The goal of synthesis flows is either to synthesize accelerated implementations that have a high performance or to synthesize correct ones that guarantee correctness properties. As synthesis flows with an emphasis on acceleration often do not provide the ability to formulate proofs of correctness, these design flows present a severe issue when applied in safety-critical systems such as cars, airplanes, or medical devices. This *trade-off* between performance and correctness leads to the question of whether both goals can be combined to get the best of both worlds.

To address this question, we first take a look at HLS flows with an emphasis on acceleration. Flows like Bambu [PF13], DWARV [NSO+12], or LegUp [CCA+13, CCF+16] evolved to tackle the synthesis of performance-oriented hardware designs. These flows are referred to as *acceleration-oriented synthesis flows* in the rest of this Chapter. They start with a model at the ESL written in a DSL embedded into the C programming language to describe hardware designs. After the model is realized, it is synthesized in

an HDL implementation at the RTL level, e.g., Verilog. During the automatic synthesis process, different optimizations like loop pipelining or functional pipelining [CCA+13, HHL91] are performed to accelerate the final implementation.

One problem with these synthesis flows is the missing definition of a synthesis scheme [EK95, BK13] and the resulting lack of property verification. Analog to the SystemC design HLS flow, mentioned in Section 3.1, it is unclear whether the model's semantics are correctly represented by the implementation's semantics and how to track and verify properties denoted at the FSL level in the RTL implementation.

In contrast, synthesis flows with an emphasis on verification like Kami [CVS+17] or the Coq/CλaSH flow, proposed in Chapter 3, starting with a specification in a formal language. This specification allows the finding of formal proofs for correctness properties about a hardware design. These flows are referred to as *correctness-oriented synthesis flows* in the rest of this chapter. After the specified behavior was formally verified, an implementation is synthesized automatically. This way, these flows guarantee a correct transformation of the specification's semantics to the final implementation and ensure the verified properties hold on all levels.

However, while these approaches can guarantee correctness, it remains unclear how the resulting design's performance compares to the performance of designs obtained by the acceleration-oriented synthesis flows reviewed above. It is intuitive to assume that a focus on verification may harm the performance of the synthesized implementation. Unfortunately, this possible trade-off has not been addressed in detail yet. While anecdotal evidence suggests correctness-oriented flows can be competitive concerning performance, this chapter presents a systematic analysis by comparing a non-trivial hardware design with two representative flows from each camp.

The missing trade-off leaves designers wondering whether they should focus on correctness – motivating the utilization of a correctness-oriented synthesis flow – or on performance – motivating the utilization of an acceleration-oriented synthesis flow.

To this end, this chapter investigates both design flow paradigms – using the LegUp HLS framework and the Coq/CλaSH flow as representatives for acceleration-oriented and correctness-oriented synthesis, respectively. Those flows were chosen as they represent the most efficient (cf. [NSP+16]) and most current flows available thus far, implementing the respective concepts. The foundation of the investigation is a 32-bit MIPS processor [HTHT09], which is synthesizable by LegUp. The processor's functional behavior is formally specified, verified, and synthesized using the Coq/CλaSH synthesis flow.

A quantitative analysis of these two processor implementations gives the first impression to gauge the trade-off between performance and correctness. Even though

there will be cases that justify applying the acceleration-oriented flows, the analysis shows the potential of further research of applying correctness-oriented flows in an industrial setting, even in cases where performance is a critical issue. Moreover, it is easier to increase the implementation's performance synthesized by correctness-oriented flows than to make hardware designs following acceleration-oriented flows correct.

This chapter is structured as follows: First, it motivates the considered problem by describing and discussing the LegUp synthesis flow. Section 5.3 describes the correctness-oriented synthesis flow in detail and how it addresses the considered problem. Section 5.4 describes the processor's specification and how properties on it are proven. Section 5.5 evaluates and discusses the RTL implementations. Finally, Section 5.6 summarizes this chapter.

## 5.2 MOTIVATION

This section analyzes the LegUp HLS framework synthesis flow [CCA+13] as a representative of the state-of-the-art acceleration-oriented synthesis flows [NSP+16, Tak16]. On average, LegUp synthesizes the fastest hardware designs, so it was chosen as a representative [NSP+16]. This flow analysis shows the missing ability to verify correctness properties of models realized for these flows.

An available model of a 32-bit MIPS processor is used as a running example to analyze the LegUp HLS framework's synthesis flow [HTHT09]. The MIPS architecture describes an ISA for a reduced Reduced Instruction Set Computer (RISC) processor design [MT08].

### 5.2.1 THE LEGUP SYNTHESIS FLOW

The LegUp framework's foundation is the Low Level Virtual Machine (LLVM) compiler infrastructure [LA04]. LLVM is a modular compiler infrastructure for optimized code generation. A model is transformed into LLVM's internal intermediate representation, a machine-independent assembly language using the Clang compiler front-end, and later optimized by a series of built-in compiler optimizations. LegUp extends the LLVM backend by generating Verilog code instead of machine code [CCA+13]. It performs different additional optimizations during the optimization process, e.g., loop pipelining [CCA+13] or functional pipelining [HHL91], to accelerate hardware

design implementations. These optimizations aim to identify behavior that can be accelerated. Whether an optimization can be performed and, therefore, result in an implementation that satisfies the required performance properties depends on the described hardware design model.

LegUp defines a Domain-specific language embedded into the C programming language to describe hardware design models and provide two synthesis modes. The first mode is the generation of a hybrid processor/accelerator architecture. The described behavior of a model is compiled and executed on a dedicated processor that profiles its execution. After profiling, LegUp selects segments of the ESL model to be accelerated by RTL implementations. The final part is re-compiling the model into a hybrid hardware/software system [CCF+16]. The second mode is synthesizing a model to a pure hardware implementation, sketched in Figure 5.1. After LegUp synthesized a model in an implementation, it can be synthesized on an FPGA using commercial synthesis tools, e.g., Intel® Quartus® Prime. In contrast to the first mode, constructs like dynamic memory management, recursion, and floating-point arithmetic are not supported [CCA+13].

This chapter focuses on the second mode since the running example is a 32-bit MIPS processor model. This model does not contain unsynthesizable constructs as it is designed for FPGA synthesis.

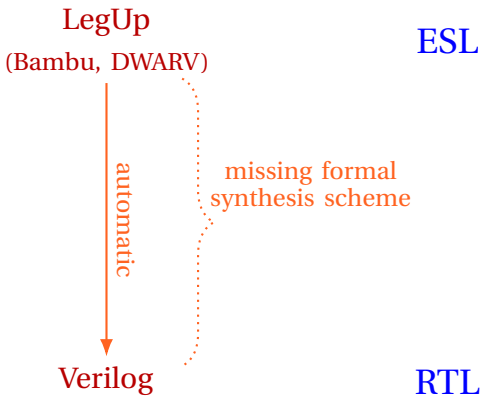


**Figure 5.1 :** *Extract of Figure 1.1 sketching the LegUp synthesis flow for pure hardware designs. This chapter focuses on the missing formal synthesis scheme of acceleration-oriented synthesis flows like LegUp, Bambu, and DWARV. A model in a hardware DSL defined by LegUp is synthesized in an accelerated RTL implementation in Verilog automatically.*

**Example 5.1.** *The model of a 32-bit MIPS processor is considered to analyze the LegUp synthesis flow regarding the verification of properties. This model is already the subject of current research [HTHT09, NSP+16] and is sketched in Listing 5.1. The model*

```
1  ins = imem[IADDR (pc)];
2  op = ins >> 26;
3  switch (op) {
4   case R:
5    funct = ins & 0x3f;
6    shamt = (ins >> 6) & 0x1f;
7    rd = (ins >> 11) & 0x1f;
8    rt = (ins >> 16) & 0x1f;
9    rs = (ins >> 21) & 0x1f;
10   switch (funct) {
11    case ADDU:
12     reg[rd] = reg[rs] + reg[rt];
13     break;
14    case SLL:
15     reg[rd] = reg[rt] << shamt;
16     break;
17 [...]
18   address = ins & 0xffff;
19   rt = (ins >> 16) & 0x1f;
20   rs = (ins >> 21) & 0x1f;
21   case ADDIU:
22    reg[rt] = reg[rs] + address;
23    break;
24 [...]
25   case J:
26    tgtadr = ins & 0x3ffffff;
27    pc = tgtadr << 2;
28    break;
```

**Listing 5.1 :** *Extract from the 32-bit MIPS processor model that contains the ADDU, SLL, ADDIU, and J instruction [HTHT09]. The model is implemented as a state machine that iterates over the instructions. The current instruction is separated into its parts using logical* shift *and logical* and *operations.*

*realizes a subset of the 32-bit MIPS standard instruction set [MT08], which are roughly 30 instructions. It also provides a sample program consisting of roughly 40 bit-vectors that follow the bit order for instructions stated by the 32-bit MIPS instruction specification [MT08]. According to the program counter, the processor reads the current instruction from the instruction array and processes it in one iteration. Each instruction is separated into its parts, e.g., the* operation code, function code, *or* operands. *After separation, the instruction is processed according to its* operation code *or* function code. *The program counter is changed after instruction execution so that the next instruction is read from the instruction array. The model also contains a register file storing 32 entries and a data memory storing 64 entries. The program execution is stopped by a dedicated instruction (syscall 10), which means* exit *and is part of the sample program.*

### 5.2.2 CONSIDERED PROBLEM

As explained above, LegUp implements a new LLVM backend to synthesize hardware designs to Verilog implementations [CCA+13]. legup's input language defines a sequential execution scheme, but hardware designs define a parallel one. To formally describe a sequential scheme's transformation into a parallel one, *synthesis schemes* [EK95, BK13] can be used. According to the LegUp authors [CCA+13], the transformation from LLVM's internal representation language to Verilog does not follow such a synthesis scheme. The same goes for synthesis flows implemented by Bambu [PF13] and DWARV [NSO+12]. As a result, it is unclear how properties formulated at the ESL level relate to the RTL implementation and how to verify them.

Moreover, it is unclear how to formulate properties in the hardware DSL due to its embedding into C. While there are tools to state and verify C programs' properties, such as Frama-C [CKK+12] or Astrée [CCF+05], these tools assume a compiler behaving according to the C standard semantics [CSt99]. These assumptions are not the case for hardware designs as just described.

The missing ability to verify models' properties using the LegUp synthesis flow leads to the following questions: *First, can we synthesize the 32-bit MIPS processor with a different synthesis flow that allows us to prove its correctness? Second, what would be the performance of the final implementation compared to the implementation synthesized by LegUp?*

## 5.3 CORRECTNESS-ORIENTED SYNTHESIS FLOWS

In contrast to the acceleration-oriented synthesis flows just introduced, there are other syntheses flows like the correctness-oriented flows implemented by Kami [CVS+17] or by the Coq/CλaSH flow proposed in Chapter 3. This section discusses and evaluates both flows to address the considered problem of acceleration-oriented synthesis flows.

Kami and the Coq/CλaSH flow rely on the proof assistant Coq [BC04, Chl13] to formally specify and verify hardware designs and automatically synthesize them afterward to an implementation. As described in Section 2.1, Coq uses higher-order logic to specify a behavior formally and subsequently prove properties about it by human-machine collaboration. In contrast to the acceleration-oriented synthesis flows such as the one implemented by LegUp, both of these flows formulate a synthesis scheme describing how the specification's semantics are propagated to the final implementation [BLWD21, CVS+17]. This synthesis scheme ensures that the verified properties at the FSL level also hold for the RTL implementation.

Both flows seem capable of addressing the problem discussed in Section 5.2.2. They allow the specification and verification of the 32-bit MIPS processor and subsequently synthesize the design on an FPGA. For example, Kami has been used to implement a RISC-V multi-core processor as a case study [CVS+17]. The Coq/CλaSH flow is more light-weight and flexible, as CλaSH allows the synthesis of arbitrary combinational and synchronous sequential hardware designs, as illustrated in Section 3.4. Due to its flexibility, this one was chosen as a representative of correctness-oriented hardware synthesis flows.

However, the question remains whether such correctness-oriented synthesis flows will result in less efficient designs concerning the synthesized circuit's performance. After all, correctness-oriented flows emphasize property verification and not the acceleration of implementations. For this reason, one would not be surprised if the implementation synthesized by the Coq/CλaSH flow would be slower than the one synthesized by LegUp. Even then, the question would remain by *how much* the design would be slower.

## 5.4 Formal Specification and Verification of a 32-bit MIPS Processor

This section describes the formal 32-bit MIPS processor's specification in Gallina and how properties about this specification are denoted and proven. Thus, this chapter analyzes the correctness-oriented design flow and a benchmark that, afterward, is used to compare to the acceleration-oriented design flow. The foundation regarding the implemented instructions, register file, and memory is the 32-bit MIPS processor, described in Section 5.2.1.

### 5.4.1 Specification of Sequential Hardware Designs

To represent sequential circuits functionally in the Coq/CλaSH synthesis flow, Mealy or Moore machines are used, as described in Section 3.4. These machines abstract the clock by defining state transitions. These transitions allow a time-controlled execution of a hardware design. Listing 5.2 shows the Mealy machine's function type used to specify the MIPS processor.

```
1  Fixpoint mealy {S I O:Type}
2     (f: S -> I -> (S*O))
3     (s: S)
4     (l: list(I))
5   : list(O)
```

**Listing 5.2 :** *Function type of the* Mealy *finite-state machine specified in Gallina. A transition function processes a list of inputs by mapping the current state and the current input to a new state and an output, starting from an initial state. For a detailed explanation of the* Mealy *machine's semantics, see Section 3.4.3.2*

For the processor specification, the program counter, the register file, and the memory define the state (S). The input (I) is ignored by the specification of function *f*, as the benchmark (the sample program mentioned in Section 5.2.1) is a fixed set of instructions. Since an output (list(O)) is required, the result of an instruction is returned. Listing 5.3 shows the instantiated function type defined by the transfer function *f*. The *registerFileType* and the *memoryType* are fixed-sized vectors with a length of 32 and 64, respectively.

```
1  Definition mips
2     (data : registerFileType*memoryType*
3             Unsigned32.int)
4     (dummy : bool)
5   : (registerFileType*memoryType*Unsigned32.int)*Unsigned32.int
```

**Listing 5.3 :** *Instantiated function type of the* mips *function, required by the* mealy *machine's transition function seen in Listing 5.2, specified in Gallina.*

The *mips* function's first argument is called *data*. It is a tuple of the *RegisterFileType*, the *memoryType*, and the *Unsigned32.int* type. This tuple represents the polymorphic type *S* in the above *mealy* function's type. This tuple's first two arguments, seen in Line 2 of Listing 5.3, are fixed-size vectors that represent the LegUp model's arrays – the sample instruction set and the memory. The tuple's third argument, seen in Line 3 of Listing 5.3, represents the program counter. The *mips* function's second argument is of the type *bool* representing the polymorphic type *I* in the above *mealy* function's type. Since the MIPS processor model defines a constant set of executed instructions (the sample program), there is no actual input to the *mealy* function. For this reason, the second argument is called *dummy*. The return type is a tuple of the *mips* function's first argument's type and a 32-bit unsigned value (Unsigned32.int), representing the tuple *S*O* in the above *mealy* function's type. This value defines the Mealy machine's output, i.e., the result of an executed instruction.

After an instruction was executed, the changed register file, the changed memory, and the new program counter are returned (the new state). If and how the register file or memory is changed depends on the executed instruction.

### 5.4.2 CONSTRUCTION OF INSTRUCTIONS

The instructions, together with their operands, are encoded as 32-bit unsigned integer values [HTHT09]. These instructions are specified in three different formats according to the 32-bit MIPS instruction set [MT08]. In addition to the operation code (op), which they all have in common, they differ in interpreting their bits. The operation code

always consists of the highest six bits. The first format is the R-Format that specifies three registers, one shift amount, and one function code and has the following layout:

$$\underbrace{op(6)\ rs(5)\ rt(5)\ rd(5)\ shamt(5)\ funct(6)}_{31\ ...\ 0\ bits}$$

The three registers state the first register operand (rs - 5 bits), the second register operand (rt - 5 bits), and the register destination (rd - 5 bits). The shift amount (shamt) also has 5 bits, while the function code (funct) has 6 bits. The operation code in the R-Format is always zero.

The second format is the I-Format. In addition to the operation code, this format specifies two registers and one immediate value and has the following layout:

$$\underbrace{op(6)\ rs(5)\ rt(5)\ immediate(16)}_{31\ ...\ 0\ bits}$$

The operation code and the two registers have the same bit sizes as in the R-Format. The immediate value is 16 bit in size.

The third format is the J-Format and states one address value, which results in the following format:

$$\underbrace{op(6)\ address(26)}_{31\ ...\ 0\ bits}$$

The address value is 26 bits in size. These formats enable a unique interpretation of the instruction bits. The specification of the 32-bit MIPS processor, sketched in Listing 5.1, is shown in Listing 5.4.

A couple of auxiliary functions were specified to separate an instruction into its parts. The *ADDU* instruction (R-Format), shown in Listing 5.4, is taken as a reference to illustrate these functions. This instruction adds two unsigned 32-bit values stored in the register file under the addresses *rs* and *rt* and stores the register file's result at the address *rd*.

- **getOpCode**: The operation code is selected by applying *logical shift right* by *26* to the current instruction.
- **getFunct**: The function code is determined by *logical conjunction*, which is applied to the instruction with the hexadecimal value *0x3f*. This value represents a bit vector of 26 0s followed by six 1s from the most significant bit (MSB) to the least significant (LSB) (big-endian).
- **getShamt**: The shift amount is selected by first applying *logical shift right* by *6* to the instruction. Afterward, *logical conjunction* is applied to that value and the hexadecimal value *0x1f*. This value represents a bit vector of 27 0s, followed by five 1s, from MSB to LSB (big-endian).

```
1  let instr := nth instructionMemory pc in
2  let op := getOpCode instr in
3  match toFormat op with
4   | RFormat =>
5    let funct := getFunct instr in
6    let shamt := getShamt instr in
7    let rt := getRT instr in let rs := getRS instr in
8    let rd := getRD instr in
9    match toFunctionCode funct with
10    | ADDU =>
11     let value := add (nth registerFile rs)
12                      (nth registerFile rt) in
13     let registerFile' := replaceAt rd value registerFile in
14     ((registerFile',memory,newPC pc),value)
15    | SLL =>
16     let value := sll (nth registerFile rt) (toZ shamt) in
17     let registerFile' := replaceAt rd value registerFile in
18     let pc' := newPC pc in
19      ((registerFile',memory,pc'),value)
20     [...]
21  | IFormat =>
22    let address := getAddress instr in
23    let rt := getRT instr in
24    let rs := getRS instr in
25    match toOperationCode op with
26     | ADDIU =>
27      let value := add (nth registerFile rs) address in
28      let registerFile' := replaceAt rt value registerFile in
29      ((registerFile',memory,newPC pc),value)
30      [...]
31   | JFormat => match toOperationCode op with
32    | J =>
33     let tgtadr := getTargetAddress instr in
34     let pc' := sll tgtadr z2 in
35     ((registerFile,memory,pc'),pc')
36     [...]
37   end
```

**Listing 5.4 :** *Extract of the 32-bit MIPS processor specified using the Coq/CλaSH synthesis flow. It pattern matches over the instructions to access the parts of an instruction as defined by the format.*

- **getRD**: The destination register is selected by first applying *logical shift right* by 11 to the instruction. Afterward, *logical conjunction* is applied to that value and the hexadecimal value *0x1f*.

- **getRT**: The first register is selected by first applying *logical shift right* by *16* to the instruction. Afterward, *logical conjunction* is applied, as for the destination register.

- **getRS**: The second register is selected by *logical shift right* by *21* to the instruction first. Afterward, *logical conjunction* is applied, as for the destination register.

After separating an instruction into its parts as defined by the format, the actual operation can be performed. As seen in Listing 5.4, the *ADDU* instruction defines two register file addresses (rs and rt). The values for these addresses are selected first; *nth registerFile rs* and *nth registerFile rt*. The function *nth* returns a fixed-size vector (registerFile) for a given index (rt). The addition of these two values is stored in the register file at the index *rd* (replaceAt rd value registerFile). The *replaceAt* function replaces a value (value) at an index (rd) of a fixed-size vector (registerFile). The final step is to replace the old register file (registerFile) with the changed one (registerFile') and increment the program counter for the next instruction. The complete set of the implemented instructions can be found in Appendix C.1.

### 5.4.3 PROVING OF PROPERTIES

This section describes how to prove properties described as theorems in Coq about the 32-bit MIPS processor specification. Custom proof methods using Coq's tactic language $\mathscr{L}_{tac}$ are implemented to prove theorems for the implemented instructions. These methods allow simplifying proofs about implemented instructions following the *R-Format*, *I-Format*, and *J-Format*, described in Section 5.4.2. Since not all instructions have an operation or function code, e.g., the *no operation* instruction does implement a code but is interpreted as a logical shift left operation, theorems about these instructions have to be proven to show the correct behavior of the processor specification.

```
1  Theorem mipsADDU:
2    forall registerFile : registerFileType,
3    forall memory : memoryType,
4    forall pc : Unsigned32.int,
5    forall dummy : bool,
6
7    let instr := nth instructionMemory pc in
8    let op := getOpCode instr in
9    let funct := getFunct instr in
10   let rd := getRD instr in
11   let rt := getRT instr in
12   let rs := getRS instr in
13   let value := add (nth registerFile rs)
14                    (nth registerFile rt) in
15
16   toFormat op = RFormat /\
17   toFunctionCode funct = ADDU ->
18   mips (registerFile, memory, pc) dummy =
19     ((replaceAt rd value registerFile, memory, newPC pc), value).
20 Proof.
21   proveRFormat.
22 Qed.
```

**Listing 5.5 :** *Theorem including its proof specified in Gallina to verify that the* ADDU *instruction adds the two values of the register addresses* rt *and* rs *and stores the result at the register file address* rd.

#### 5.4.3.1 CUSTOM PROOF METHODS

After the 32-bit MIPS processor was specified in Gallina, the designer must prove properties about its specification to ensure its correct behavior. Since the implemented processor instructions, seen above, use either the *R-Format*, the *I-Format*, or the *J-Format*, implemented proof methods demonstrate how instructions following these formats can be verified by applying them. Listing 5.5 shows a property about the specified *ADDU* instruction, which uses the *R-Format*.

The theorem denotes that if the operation code *op* indicates the *RFormat* and the function code *funct* indicates the *ADDU* instruction, then the values of the register

file addresses *rs* and *rt* are added together. The result is stored in the register file at address *rd*. Supplemental statements are defined, starting with the *let* keyword for a better understanding of the theorem. This keyword binds an identifier to a term, as described in Section 3.4.3.1.

$\mathscr{L}_{tac}$ allows the specification of user-defined proof methods [Del00], as described in Section 2.1. The specified proof method *proveRFormat* represents such a user-defined proof method and allows proving properties about instructions, which implement the *R-Format* and follow the theorem structure described above. The tactic is seen in Listing 5.6.

The *proveRFormat* tactic is built from tactics already provided by the Coq proof assistant. These tactics are described in Section 3.4 and Section 4.5 in detail. This proof method proves a theorem following the above format by reducing the left side of the conclusion's equation to the right side, seen in Line 19 of Listing 5.5. Note that Section 2.1 describes how Coq represents a proof and how tactics manipulate subgoals by their local context to finally solve them. The first tactic that is used is the *intros* tactic. This tactic introduces variables, such as *registerFile* and *op*, and the theorem's hypothesis, i.e., the left side of the implication (→) to the local context. The next step is to match the hypothesis – Coq names the hypothesis with *H* by default. For this reason, we search for this hypothesis in the current subgoal's context, as seen in Line 4 of Listing 5.6. Without the *match* construct, the proof script does not have any access to the identifier. The same goes for every identifier the proof scripts needs access to in the local context. Since the hypothesis represents a conjunction, the next step is to destruct this conjunction into two hypotheses (H1 and H2) and replace the identifier *mips* with its definition in the subgoal by applying the *unfold* tactic. The *op* and *instr* identifier are replaced with their terms in the hypothesis *H1*. Rewriting the subgoal by this changed hypothesis reduces it to the second *match* statement *toFunctionCode funct*, seen in Line 9 of Listing 5.4, as *H1* matches the first *match* statement, seen in Line 3 of Listing 5.4. The next step is to replace the *funct* and *instr* identifier in the hypothesis *H2* with their terms. Rewriting the subgoal reduces it to the right side of the equation. The *reflexivity* tactic finally solves this trivial equality..

The second specified proof method considers instructions following the *I-Format*. These instructions contain an immediate value. The instruction's operation code defines how to combine this value with a register value or memory value. The *ADDIU* instruction, shown in Listing 5.7, illustrates such an instruction.

The specification of a theorem for an instruction following the *I-Format* works similarly to the theorem for the *R-Format* described above. First, the instruction must be separated into its parts: *op*, *address*, *rt*, *rs*, and *value*. Supplement statements denote these parts. After the separation, the format must be the *IFormat*, and the operation code must match that format, e.g., the operation code *ADDIU*.

```
1  Ltac proveRFormat :=
2    intros;
3    match goal with
4     | [ H : _ |- _] =>
5       destruct H as [H1 H2 ];
6       unfold mips;
7       match goal with
8        | [op: _ |- _] =>
9          unfold op in H1;
10         match goal with
11          | [instr: _ |- _] =>
12            unfold instr in H1;
13            rewrite H1;
14            match goal with
15             | [funct: _ |- _] =>
16               unfold funct in H2;
17               unfold instr in H2;
18               rewrite H2;
19               reflexivity
20            end
21         end
22      end
23   end.
```

**Listing 5.6 :** proveRFormat *tactic in* $\mathscr{L}_{tac}$*. The tactic allows the proving of theorems about instructions that follow the R-Format, which can be seen as an example in Listing 5.5. This tactic reduces the* mips *function specification in the subgoal, seen in Listing 5.4, according to the given instruction format and function code to prove if the theorem's equation is satisfied.*

```
1  Theorem mipsADDIU:
2    forall registerBank : registerBankType,
3    forall memory : memoryType,
4    forall pc : Types.Unsigned32.int,
5    forall dummy : bool,
6
7    let instr := nth instructionMemory pc in
8    let op := srl instr z26 in
9    let address := and instr 0xffff in
10   let rt := and (srl instr z16) 0x1f in
11   let rs := and (srl instr z21) 0x1f in
12   let value := add (nth registerBank rs) address in
13
14   toFormat op = IFormat /\ toOperationCode op = ADDIU ->
15   mips (registerBank, memory, pc) dummy =
16   ((replaceAt rt value registerBank, memory, newPC pc), value).
17 Proof.
18   proveIFormat.
19 Qed.
```

**Listing 5.7 :** *Theorem including its proof specified in Gallina to verify that the* ADDIU *instruction adds the value of the register addresses* rs *to the immediate value* address *and stores the result at the register file address* rt.

```
1  Ltac proveIFormat :=
2    intros;
3    match goal with
4     | [ H : _ |- _] =>
5       destruct H as [H1 H2 ];
6       unfold mips;
7       match goal with
8        | [op: _ |- _] =>
9          unfold op in H1;
10         match goal with
11           | [instr: _ |- _] =>
12             unfold instr in H1;
13             rewrite H1;
14             unfold op in H2;
15             unfold instr in H2;
16             rewrite H2;
17             reflexivity
18         end
19      end
20   end.
```

**Listing 5.8 :** proveIFormat *tactic in* $\mathscr{L}_{tac}$. *The tactic allows the proving of properties about instructions that follow the I-Format, which can be seen as an example in Listing 5.7. This tactic reduces the* mips *function specification in the subgoal, seen in Listing 5.4, according to the given instruction format and operation code to prove if the theorem's equation is satisfied.*

The proof method *proveIFormat*, shown in Listing 5.8, proves a theorem that follow the above format. It works analogously to the proof method *proveRFormat*, described above.

The *proveIFormat* proof method contains the same tactics as the *proveRFormat* proof method described above in detail until Line 11 of Listing 5.8. The *instr* identifier is replaced by its term in the hypothesis *H1*. Rewriting the changed hypothesis H1 reduces the subgoal to the *mips* function's second *match* statement.

The replacement of the identifier *op* and *instr* changes the hypothesis *H2* to match the function code in the *mips* function's second *match* statement. The rewriting of the hypothesis *H2* reduces the left side of the equals sign to the right side.

```
1  Theorem mipsJ:
2    forall registerBank : registerBankType,
3    forall memory : memoryType,
4    forall pc : Types.Unsigned32.int,
5    forall dummy : bool,
6
7    let instr := nth instructionMemory pc in
8    let op := srl instr z26 in
9    let tgtadr := and instr 0x3ffffff in
10   let pc' := sll tgtadr z2 in
11
12   toFormat op = JFormat /\ toOperationCode op = J ->
13   mips (registerBank, memory, pc) dummy =
14   ((registerBank, memory, pc'), pc').
15 Proof.
16   proveJFormat.
17 Qed.
```

**Listing 5.9 :** *Theorem including its proof specified in Gallina to verify that the* J *instruction jumps to a given memory address by changing the program counter. The notation* pc' *denotes the new program counter.*

The last implemented proof method considers instructions following the *J-Format*. This format implements jump instructions to cause a branch in the program execution, e.g., to implement loops or conditional statements.

The specification of a theorem for an instruction following the *J-Format* works similarly to the theorems described above. First, the instruction must be separated into its parts: the operation code *op* and the target address *tgtadr*. After the separation, the format must be the *JFormat*, and the operation code must match that format, e.g., the operation code *J*.

These proof methods simplify the proving of properties about instructions already implemented and those added in the future. The proofs of all implemented instructions using the above proof methods can be found in Appendix C.2.

```
1  Ltac proveJFormat := proveIFormat.
```

**Listing 5.10 :** proveJFormat *tactic in* $\mathscr{L}_{tac}$*. The tactic allows the proving of properties about instructions that follow the J-Format, which can be seen as an example in Listing 5.9. As this format is the same required for the* I-Format*, seen in Listing 5.8, the* proveJFormat *is an alias for the* proveIFormat *tactic.*

#### 5.4.3.2 PROVING ADDITIONAL PROPERTIES

After verifying that the specified instructions are correct using the proof methods described above, the designer might prove other properties to demonstrate that the processor functionally behaves as expected. One of those properties is that the *mips* function executes the NOP (no operation) instruction specified by the 32-bit MIPS architecture standard [MT08]. For instance, this instruction is used to enforce memory alignment or prevent hazards in the processor's instruction pipeline. The NOP instruction does not change any state but only increments the program counter. The MIPS standard does not specify the NOP instruction with an extra operation code but maps it to the logical shift left operation (SLL), seen in Listing 5.4. The theorem shown in Listing 5.11 specifies this behavior.

The NOP instruction is a 32-bit vector containing only zeros (Ox00000000). This instruction does not change the initial register file's content, but the *SLL* operation returns a new register file, as seen in Line 15 of Listing 5.4. For this reason, the statement *registerFile'* is specified. As the processor interprets the NOP instruction as the *SLL* operation, the *output* statement defines its result. Note that this theorem cannot be proven using the proposed proof methods. It does neither contain logical conjunction in the hypothesis nor define the required supplement statements, e.g., for the operation code *op* or the function code *funct*. The *mips* function needs to be reduced using the *NOP* instruction to show that it is interpreted as an *SLL* operation to prove the *mipsNOP* theorem. This operation only changes the program counter to the next instruction (newPC pc). The *newPC* function increments the program counter by one and resets it back to zero if it reaches the maximum amount of instructions.

After introducing the quantified variables, statements, and the hypothesis to the subgoal's local context, the *mips* identifier is replaced by its specification. Rewriting the subgoal with the hypothesis *H* instantiate the instruction evaluated in *mips* function's *match* statements with the NOP instruction. Since this instruction is a constant bit

```
1  Theorem mipsNOP:
2    forall registerFile : registerFileType,
3    forall memory : memoryType,
4    forall pc output : Types.Unsigned32.int,
5    forall dummy : bool,
6
7    let nop := 0x00000000 in
8
9    let registerFile' := replaceAt
10    (and (srl nop z11) 0x1f)
11    (sll
12     (nth registerFile (and (srl nop z16) 0x1f))
13     (toInt (and (srl nop z6) 0x1f)))
14    registerFile in
15
16   let output := sll
17     (nth registerFile
18     (and (srl nop z16) 0x1f))
19     (toInt (and (srl nop z6) 0x1f)) in
20
21   nth instructionMemory pc = nop ->
22   mips (registerFile,memory,pc) dummy =
23   ((registerFile', memory, newPC pc), output).
24 Proof.
25   intros.
26   unfold mips.
27   rewrite H.
28   simpl.
29   reflexivity.
30 Qed.
```

**Listing 5.11 :** *The NOP instruction is implemented for the MIPS processor as: sll r0 r0 0. The value of register r0 is logically shifted left by 0, and the result is stored in r0. The theorem* mips_nop *ensures this behavior. Note that the register r0 returns the constant zero [MT08].*

vector, the subgoal's equation's left side is reduced to the right side by the *simpl* tactic. The final subgoal is again a trivial equality and solved by the *reflexivity* tactic.

Appendix C.3 includes additional theorems including their proofs to verify the MIPS processor's correct behavior. For example, an unknown function code or an unknown operation code reset the *program counter* to 0 (error). This error behavior comes from the original 32-bit MIPS processor model [HTHT09].

This section showed how to formally specify and verify a 32-bit MIPS processor using the Coq/CλaSH synthesis flow. The verification of properties successfully addresses the LegUp synthesis flow deficiencies described in Section 5.2.2. The processor specification was automatically synthesized into a Verilog implementation to answer how the two implementation's performance – the one synthesized by LegUp and the one synthesized by the Coq/CλaSH flow – compares.

## 5.5 EVALUATION

This section evaluates and discusses the RTL implementation's performance synthesized by an acceleration-oriented and correctness-oriented flow. The foundation is the implementation of the 32-bit MIPS processor synthesized by LegUp and Coq/CλaSH. Both implementations implement the same instructions and execute the sample program, described in Section 5.2.1. In the following, the results obtained by both implementations are summarized first. Afterward, the conclusions that can be drawn from these results are discussed.

### 5.5.1 RESULTS

Table 5.1 shows the performance results of both implementations. This table's entries should be considered as an approximation. They highly depend on the FPGA the hardware design is synthesized for; they indicate rather than precisely quantify the relation between the two synthesized hardware designs.

The individual rows of Table 5.1 are now explained in detail. The first row contains the maximum clock frequency $F_{MAX}$ at which the final circuit can be operated. As we see, the circuit synthesized by the LegUp HLS framework operates at a higher frequency than the one synthesized by Coq/CλaSH.

The second row contains the clock cycles needed by the processor implementations to execute the sample program (clock latency). These values are evaluated by

**Table 5.1 :** *Evaluation of the two 32-bit MIPS processor implementations. The* LegUp *column contains the values based on the implementation synthesized by the LegUp HLS framework. The* Coq/CλaSH *column contains the values of the synthesized design based on the Coq/CλaSH synthesis flow used in this work, which is discussed in Section 5.3.*

| | **LegUp** | **Coq/CλaSH** |
|---|---|---|
| $F_{MAX}$ in [MHz] | 63.36 | 55.86 |
| Cycles | 5035 | 12220 |
| Wall-Clock in $\mu s$ | 79.5 | 218.76 |
| ALMs | 1045 / 56480 (2%) | 1772 / 56480 (3%) |
| Registers | 939 | 1644 |
| DSP Block | 6 / 156 (4%) | 2 / 156 (1%) |
| Total Block Memory Bits | 3072 / 7024640 (< 1%) | 0 / 7024640 (0%) |

*The RTL implementations in Verilog were synthesized for the Cyclone V family using the commercial synthesis tool Intel® Quartus® Prime.*

simulation using Intel® Quartus® Prime's ModelSim. For simulation, a clock cycle of 20 ns was used. Together with the maximum frequency, this results in the time it takes in $\mu s$ to execute the program (Wall-Clock) provided by the model, described in Section 5.2.1. The implementation synthesized by LegUp takes 79.5 $\mu s$ for execution, while the implementation synthesized by Coq/CλaSH takes 218.76 $\mu s$.

The fourth row contains the Adaptive Logic Modules (ALMs), also called Lookup Tables (LUTs), in the Xilinx Vivado synthesis tool. These are the basic building blocks for hardware designs on an FPGA. As seen, the circuit synthesized by LegUp consumes 2% of the available ALMs, while the one synthesized by Coq/CλaSH consumes 3% of the available ALMs. To better classify these values, we take a look at the last row of the table. This row contains the total block memory in bits, which is the block RAM of the FPGA. The memory of an FPGA is separated into distributed RAM (ALMs) and block RAM. LegUp stores each local and global memory in the separated block RAM by default. For larger memories, the block RAM is much faster than distributed RAM. The implementation synthesized by Coq/CλaSH uses no block ram but stores the entire design in distributed RAM.

The fifth row of Table 5.1 contains the consumed registers. To classify these values, we consider the design of the 32-bit MIPS processor. The LegUp processor's model changes the values of an array in place, so only one array, e.g., for the register file, is needed. The Coq specification's functional foundation and thus the CλaSH model requires *single assignment* of variables. For this reason, the underlying Mealy machine

needs the changed register file as part of the new state, as seen in Listing 5.4. The synthesis of this behavior results in the consumption of more registers.

The sixth row contains the amount of used Digital Signal Processing (DSP) blocks. These blocks describe a dedicated functionality, e.g., multipliers, which are provided by the synthesis tool. Intel® Quartus® Prime automatically infer the usage of those DSP blocks by analyzing the RTL code.

The usage of those DSP blocks is automatically inferred by Intel® Quartus® Prime through analyzing the RTL code.

### 5.5.2 DISCUSSION

This section discusses the results of the evaluation described above. Acceleration-oriented synthesis flows such as LegUp define a model in a HDSL embedded into C. This language's low-level nature allows a more acceleration-oriented realization of hardware design implementations but lacks the verification of properties, as described in Section 5.2.2.

On the other hand, correctness-oriented synthesis flows such as the Coq/CλaSH flow define a behavior functionally at a higher level of abstraction, making them easier to understand, and hence less error-prone [Hug89], and susceptible to verification in the first place. However, it impacts performance, resulting in lower clock frequency or a higher amount of clock cycles.

The evaluation presented above shows that although the synthesized implementation using the Coq/CλaSH flow was slower than the one synthesized by LegUp, the Coq/CλaSH flow could synthesize a 32-bit MIPS processor, comparable concerning performance indicators like clock frequency or execution time. The Coq/CλaSH flow's standard toolchain was used to perform this evaluation.

This systematic comparison demonstrates the potential of correctness-oriented synthesis flows, indicating that these flows result in circuits with competitive performance. Research projects like Kami show that the synthesis of verified specifications is a subject of current research. The successful synthesis of a RISC-V processor shows the potential of correctness-oriented flows [CVS+17]. When hardware is used in safety-critical systems, verifying the correct functional behavior becomes essential; the evaluation demonstrates that correctness-oriented flows can achieve this without sacrificing too much performance. Moreover, there is still an unexplored potential for performance gains in correctness-oriented flows, whereas adding verification to an acceleration-oriented flow seems, at first sight, far more challenging.

For these reasons, the quantified analysis presented in the above section suggests that correctness-oriented synthesis flows can be employed when the need for verification arises in a performance-oriented environment.

## 5.6 CONCLUSION

This chapter analyzed the acceleration-oriented hardware design synthesis flows implemented by Bambu [PF13], DWARV [NSO+12], and LegUp [CCA+13] and showed their missing property verification ability. In contrast, correctness-oriented hardware design synthesis flows, as implemented by Kami [CVS+17] or the Coq/CλaSH synthesis flow proposed in Chapter 3, were considered. This chapter addressed the question of a quantitative analysis of the trade-off concerning performance between acceleration-oriented and correctness-oriented flows by comparing a non-trivial electronic circuit designed by two representative flows. The designed circuit was a synthesized RTL implementation of a 32-bit MIPS processor [HTHT09]. LegUp was chosen as a representative of the acceleration-oriented synthesis flows, while the Coq/CλaSH flow was chosen as a representative of the correctness-oriented flows.

The presented evaluation, seen in Table 5.1, allows a quantitative analysis of the trade-off between performance and correctness. This chapter indicates that using a hardware design synthesis flow that allows proof of correctness does not require sacrificing much performance in the implemented system. However, if better performance is needed, one argues that it is easier to increase the performance of hardware designs synthesized by correctness-oriented flows than to add correctness to acceleration-oriented flows. For this reason, this chapter suggests further research to enhance the performance of correctness-oriented flows.

# 6

# Conclusion and Outlook

This chapter concludes this dissertation by referring to the initial research questions asked in Section 1.2 and outlines further research based on the proposed contributions.

## 6.1 Conclusion

Figure 6.1 sketches how this dissertation addressed the individual problems of both the established correctness-oriented hardware design flow and the established HLS flows to answer the initial research questions.

This dissertation first proposed an alternative hardware design flow to the established one, which uses SysML [Obj15] and OCL [Obj12] at the FSL level and SystemC [Gro02, Arn00] at the ESL level. This flow's fundamental problem is the missing propagation of verification results from the specification to the RTL implementation, as described in Chapter 3. The proposed flow automatically synthesizes a hardware design at the FSL level in an implementation at the RTL level. In contrast to the established design flow, the proposed one uses the proof assistant Coq [BC04] at the FSL level and the functional hardware description language CλaSH [BKK+10] at the ESL level. Coq allows the formal specification of functional hardware designs, including their formal verification. Combinational circuits are represented as pure and total functions, and synchronous sequential circuits combine these functions either with a Mealy or Moore machine, as described in Section 3.4. These machines functionally describe the timing-behavior of hardware designs as transitions between individual states. CompCert's integer library [LBK+16] is utilized, as proof assistants like Coq [BC04] usually model infinite data types, such as $\mathbb{Z}$ or $\mathbb{N}$. This library specifies finite *signed* and *unsigned* integer types of arbitrary sizes in Coq's specification language. After *formally verifying* the specification, a model in the hardware description language CλaSH is extracted automatically. The combination of Coq and CλaSH allows the automatic propagation of verification results from the specification down to the implementation. This propagation ensures that properties verified for the specification also hold for the model and the implementation. In contrast to the established

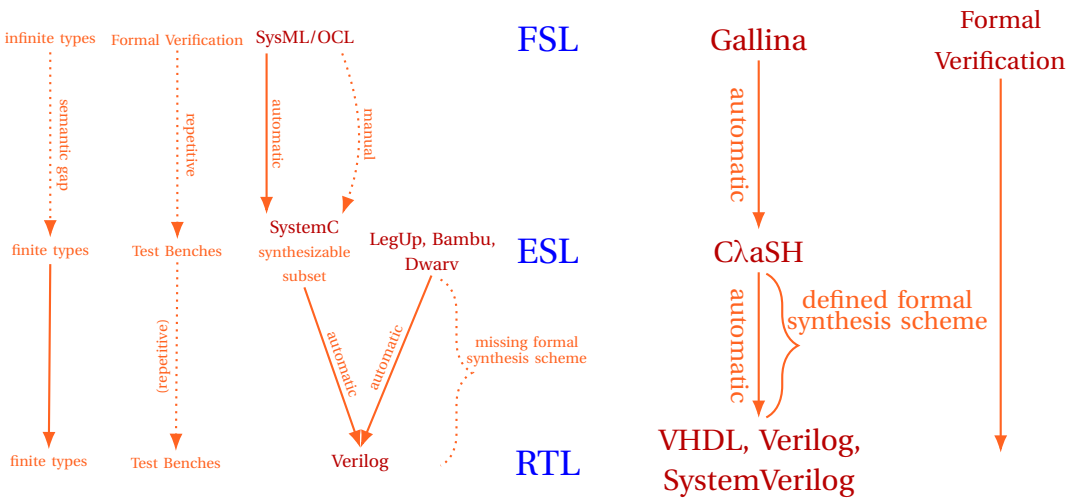


**Figure 6.1 :** *Addressed problems of the established correctness-oriented hardware design flow and of the HLS flows in this dissertation. The left side highlights these problems as dotted lines, illustrated in Figure 1.1. The right side illustrates how these problems are addressed. A formal and functional specification in Gallina is formally verified and a functional CλaSH model at the ESL level is automatically extracted from this specification. Utilizing the CompCert library allows the specification of finite integer types at the FSL level and closes the* semantic gap *regarding integer types the established correctness-oriented hardware design flow has. Since CλaSH defines a formal synthesis scheme, the semantics of this model are propgated correctly to the final implementation at the RTL level. As a result formally verified properties are propagated automatically down to the RTL implementation.*

hardware design flow, which relies on SysML/OCL and SystemC, the proposed design flow does not require test benches for the model and the implementation. Chapter 3 answers the first research question: *Can an alternative hardware design flow to the established one be developed to propagate verification results automatically?*, asked in Section 1.1, in the affirmative. The automatic propagation is advantageous as it allows the satisfiability of correctness properties at the RTL level required by electronic circuits in safety-critical environments.

Based on the proposed hardware design synthesis flow, Chapter 4 proposes an overflow detection scheme. In contrast to the established hardware design flow, which models infinite integer types, the proposed flow utilizes the CompCert integer library, as described above. This utilization allows addressing problems with finite integer types, e.g., detection of arithmetic integer overflows, already at the FSL level. The proposed detection scheme uniquely distinguishes an occurred overflow from

the arithmetic operation's actual result, which is ensured by proven properties. The SysML/OCL specification of the established design flow is not aware of these overflows, as it models infinite integer types. The scheme requires a different function type evolving the *option* monad. This monad provides two constructors: *None* indicates the overflow, and *Some(A)* contains the result of the performed arithmetic operation. Chapter 4 also addresses the issue that operations that implement the required function type are no longer closed. As a result, these operations can no longer be cascaded like their arithmetic counterparts. Furthermore, it might be the case that the input values of a function do not lead to an overflow due to the specification of external boundaries. In this case, proposed theorems allow, if proven, the safe usage of arithmetic operations as no overflow occurs. Chapter 4 answers the second research question: *If the semantic gap can be closed, can the alternative hardware design flow be used to address low-level problems such as arithmetic integer overflows already at the FSL level?*, asked in Section 1.1, in the affirmative. The proposed overflow detection scheme is a further advantage of the proposed synthesis flow, increasing the electronic circuit's safety.

To address the last research question: *Can the alternative design flow be used to gauge the trade-off between performance-oriented and correctness-oriented hardware designs?*, asked in Section 1.1, a 32-bit MIPS processor was formally specified, verified, and subsequently synthesized using the proposed hardware design flow. The foundation of the processor is a benchmark published as part of the CHStone benchmark suite [HTHT09]. This processor contains a sample program to investigate the processor's performance. The CHStone benchmark suite allows the comparison of HLS flows concerning performance and consumed space. Since these benchmarks are described in a HDL embedded in C, the processor was re-specified in Coq's specification language Gallina. This specification was synthesized using the proposed design flow and compared with the processor benchmark synthesized by the state-of-the-art hardware acceleration framework LegUp. Comparing their impact on performance and consumed space showed that the implementation synthesized from the specified processor is slower but comparable with the implementation's performance synthesized by LegUp. However, one can argue that it is easier to increase the performance of an implementation synthesized by a correctness-oriented synthesis flow fast than increasing the safety of an implementation synthesized by an acceleration-oriented synthesis flow. The evaluation gives designers a first impression how to gauge the impact of correctness-oriented synthesis flows on the implementation's performance. It was performed with the proposed hardware design synthesis flow's standard toolchain, allowing room for improving this toolchain regarding performance. In general, there is always a *trade-off* between performance-oriented and correctness-oriented synthesis flows, and they both have their merits. As performance is a crucial aspect, even in

safety-critical environments, it opens the door for further research in this area and shows the proposed synthesis flow's applicability.

## 6.2 OUTLOOK

The proposed contributions presented in this dissertation can be used as the foundation for further work. This future work considers the synthesis of electronic circuits from formally specified and verified hardware designs that satisfy correctness properties and consider performance.

Chapter 4 proposes a generic overflow detection scheme for arithmetic integer operations. However, it remains to be seen how it addresses overflows or even underflows in other arithmetics, e.g., floating-point arithmetic. Answering this question is outside of this dissertation's scope as it investigates arithmetic integer overflows. This investigation, combined with the results of Chapter 4, could lead to the publication of a library of overflow detecting arithmetic operations for arbitrary finite integer and floating-point types. The publication of such a library would lead to more safety-oriented hardware designs. Furthermore, it could draw attention to address other low-level problems, such as out-of-memory access by processor instructions directly at the FSL level.

Chapter 5 uses a 32-bit MIPS processor as the foundation of its analysis. This chapter uses that processor to investigates performance aspects of correctness-oriented hardware design flows, as the CHStone benchmark suite provides a reference model for this processor [HTHT09]. Besides the MIPS ISA, the open RISC-V ISA [RIS20] has received much attention over the last decade. For instance, Kami [CVS+17] provides a verified 32-bit RISC-V processor that implements the RISC-Vinteger instruction set. Unfortunately, the CHStone benchmark suite does not provide a RISC-V reference model that investigates performance aspects comparing correctness-oriented, and performance-oriented hardware design flows. However, it would be interesting to investigate how an implementation synthesized by the proposed Coq/CλaSH hardware design synthesis flow compares to the implementation synthesized by Kami concerning performance and consumed space. This comparison would help to classify the proposed Coq/CλaSH synthesis flow in the field of correctness-oriented hardware design synthesis flows.

Chapter 5 investigates correctness-oriented synthesis flows concerning performance, as described above. Based on this investigation, further hardware design models that are part of the CHStone benchmark suite [HTHT09] should be compared with implementations synthesized from formally verified specifications synthesized

by the Coq/CλaSH hardware, proposed in Chapter 3. Since the CHStone benchmark models are written in a HDL embedded into C, they exploit C's low-level nature to describe hardware designs closer to the final implementation, as described in Section 5.5. An entirely quantitative analysis of correctness-oriented implementations realizing these benchmarks that take performance under consideration is not trivial. Such a complete analysis is beyond this dissertation's scope for the following reasons: First, it remains to be seen if the performance of the remaining benchmark implementations synthesized by the Coq/CλaSH flow is comparable to the implementations synthesized by LegUp. In general, this comparability is probably not always given. Some of these benchmark models, e.g., the JPEG image decompression algorithm, are highly optimized to allow a fast decompression of images. Second, if additional acceleration optimizations like functional pipelining can be added to the CλaSH compiler, its underlying formal synthesis scheme must be adapted to these optimizations. As described in Section 3.5.2, the CλaSH compiler implements such a scheme to ensure that the model's semantics are propagated correctly to the implementation's semantics. However, this dissertation aims to indicate the potential of correctness-oriented hardware design synthesis flows that take performance under consideration due to a missing available quantitative analysis. Even though these first indications look promising, a detailed comparison including the necessary adjustments to the CλaSH compiler and the synthesis scheme is future work.

These promising future work topics could draw more attention to correctness-oriented synthesis flows, taking performance under consideration. As a result, designers might gauge in detail the *trade-off* between performance and correctness of hardware design implementations, bringing us efficient and at the same time safe electronic circuits.

# BIBLIOGRAPHY


[Arn00] Guido Arnout. Systemc standard. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 573–578, 2000.

[Ash02] Peter J. Ashenden. *The designer's guide to VHDL, 2nd Edition*. The Morgan Kaufmann series in systems on silicon. Kaufmann, 2002.

[BBP11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *International Conference on Automated Deduction (CADE)*, volume 6803 of *Lecture Notes in Computer Science (LNCS)*, pages 116–130. Springer, 2011.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BCBO18] Ronan Baduel, Mohammad Chami, Jean-Michel Bruel, and Iulian Ober. Sysml models verification and validation in an industrial context: Challenges and experimentation. In Alfonso Pierantonio and Salvador Trujillo, editors, *Modelling Foundations and Applications, European Conference on Modelling Foundations and Applications (ECMFA)*, volume 10890 of *Lecture Notes in Computer Science (LNCS)*, pages 132–146. Springer, 2018.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *The ACM SIGPLAN International Conference on Functional Programming(ICFP)*, pages 174–184, 1998.

[BDBK09] D. Black, Jack Donovan, Bill Bunton, and Anna Keist. Systemc: From the ground up, second edition. 2009.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - A functional language with dependent types. In *Theorem Proving in Higher Order Logics (TOPHOLS)*, pages 73–78, 2009.

[BG01] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Hand-*

*book of Automated Reasoning (in 2 volumes)*, pages 1149–1238. Elsevier and MIT Press, 2001.

[BK13] Christiaan Baaij and Jan Kuper. Using rewriting to synthesize functional languages to digital circuits. In Jay McCarthy, editor, *Trends in Functional Programming (TFP)*, volume 8322 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2013.

[BKK+10] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. C$\lambda$ash: Structural descriptions of synchronous hardware using haskell. In *Euromicro Conference on Digital System Design (DSD)*, pages 714–721, 2010.

[BKPU16] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[BLWD20a] Fritjof Bornebusch, Christoph Lüth, Robert Wille, and Rolf Drechsler. Integer overflow detection in hardware designs at the specification level. In *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2020.

[BLWD20b] Fritjof Bornebusch, Christoph Lüth, Robert Wille, and Rolf Drechsler. Safety first: About the detection of arithmetic overflows in hardware design specifications. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Revised Selected Papers*, volume 1361 of *Communications in Computer and Information Science*, pages 26–48. Springer, 2020.

[BLWD20c] Fritjof Bornebusch, Christoph Lüth, Robert Wille, and Rolf Drechsler. Towards automatic hardware synthesis from formal specification to implementation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020.

[BLWD21] Fritjof Bornebusch, Christoph Lüth, Robert Wille, and Rolf Drechsler. Performance aspects of correctness-oriented synthesis flows. In *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2021.

[BMH07] Edwin Brady, James McKinna, and Kevin Hammond. Constructing correct circuits: Verification of functional aspects of hardware specifications

with dependent types. In *Trends in Functional Programming (TFP)*, pages 159–176, 2007.

[BW06] Achim D. Brucker and Burkhart Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[BWD17] Fritjof Bornebusch, Robert Wille, and Rolf Drechsler. Towards lightweight satisfiability solvers for self-verification. In *International Symposium on Electronic System Design (ISED)*, pages 1–5. IEEE, 2017.

[CCA+13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz S. Czajkowski, Stephen Dean Brown, and Jason Helge Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems*, 13(2):24:1–24:27, 2013.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *European Symposium on Programming (ESOP)*, pages 21–30, 2005.

[CCF+16] Andrew Canis, Jongsok Choi, Blair Fort, Bain Syrowik, Ruolong Lian, Yu Ting Chen, Hsuan Hsiao, Jeffrey B. Goeders, Stephen Dean Brown, and Jason Helge Anderson. Legup high-level synthesis. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 175–190. Springer, 2016.

[CCR08] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL class diagrams using constraint programming. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 73–80, 2008.

[CES+09] Koen Claessen, Niklas Eén, Mary Sheeran, Niklas Sörensson, Alexey Voronov, and Knut Åkesson. Sat-solving in practice, with a tutorial example from supervisory control. *Discrete Event Dynamic Systems*, 19(4):495–524, 2009.

[CGMT09] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andrés Takach. An introduction to high-level synthesis. *ieeedtc*, 26(4):8–17, 2009.

[CH13] Zack Coker and Munawar Hafiz. Program transformations to fix C integers. In *International Conference on Software Engineering (ICSE)*, pages 792–801, 2013.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[CHM07] J. Castillo, P. Huerta, and J. Martínez. An open-source tool for systemc to verilog automatic translation. *Latin American Applied Research*, 37:53–58, 2007.

[CK18] Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1-4):423–453, 2018.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. pages 233–247, 2012.

[CMK12] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. Automatic RTL test generation from systemc TLM specifications. *ACM Transactions on Embedded Computing Systems*, 11(2):38:1–38:25, 2012.

[Cou12] Patrick Cousot. Formal verification by abstract interpretation. In *NASA Formal Methods Symposium (NFM)*, pages 3–7, 2012.

[CP88] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *Computer*, 21(7):8–19, 1988.

[CS05] IEEE Computer Society. IEEE 1364-2005 - IEEE Standard for Verilog® Hardware Description Language. 2005. https://standards.ieee.org/standard/1364-2005.html.

[CS18] IEEE Computer Society. IEEE 1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. 2018. https://standards.ieee.org/standard/1800-2017.html.

[CS19] IEEE Computer Society. IEEE 1076-2019 - IEEE Standard for VHDL Language Reference Manual. 2019. https://standards.ieee.org/standard/1076-2019.html.

[CSt99] Programming languages — C. ISO/IEC Standard 9899:1999(E), 1999. Second Edition.

[CVS+17] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP):24:1–24:30, 2017.

[Del00] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)*, pages 85–95. Springer, 2000.

[DH89] Doron Drusinsky and David Harel. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(7):798–807, 1989.

[DHJ+10] Mourad Debbabi, Fawzi Hassaïne, Yosr Jarraya, Andrei Soeanu, and Luay Alawneh. *Verification and Validation in Systems Engineering - Assessing UML / SysML Design Models*. Springer, 2010.

[DLRA15] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. Understanding integer overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.*, 25(1):2:1–2:29, 2015.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science (LNCS)*, pages 337–340. Springer, 2008.

[dMB11] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Comm. of the ACM*, 54(9):69–77, 2011.

[DSW12] Rolf Drechsler, Mathias Soeken, and Robert Wille. Formal Specification Level. In *Forum on Specification and Design Languages (FDL)*, pages 37–52, 2012.

[Dut14] Bruno Dutertre. Yices 2.2. In *International Conference on Computer-Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science (LNCS)*, pages 737–744. Springer, July 2014.

[EK95] Dirk Eisenbiegler and Ramayya Kumar. Formally embedding existing high level synthesis algorithms. In Paolo Camurati and Hans Eveking, editors, *Correct Hardware Design and Verification Methods, IFIP WG*

*10.5 Advanced Research Working Conference, (CHARME)*, volume 987 of *Lecture Notes in Computer Science (LNCS)*, pages 71–83. Springer, 1995.

[EMT+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into coq. In Rupak Majumdar and Viktor Kuncak, editors, *International Conference on Computer-Aided Verification (CAV)*, volume 10427 of *Lecture Notes in Computer Science (LNCS)*, pages 126–133. Springer, 2017.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science (LNCS)*, pages 502–518. Springer, 2003.

[FHT06] Joachim Falk, Christian Haubelt, and Jürgen Teich. Efficient representation and simulation of model-based designs. In *Forum on Specification and Design Languages (FDL)*, pages 129–135, 2006.

[FL10] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. pages 10–30, 2010.

[FSS15] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-ware: Hardware description and verification in agda. In *International Conference on Types for Proofs and Programs (TYPES)*, pages 9:1–9:27, 2015.

[Gam13] Peter Gammie. Synchronous digital circuits as functional programs. *ACM, Comp. Surveys*, 46(2):21:1–21:27, 2013.

[GD19] Mehran Goli and Rolf Drechsler. Scalable simulation-based verification of systemc-based virtual prototypes. In *Euromicro Conference on Digital System Design (DSD)*, pages 522–529. IEEE, 2019.

[GdM09] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *International Conference on Computer-Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science (LNCS)*, pages 306–320. Springer, 2009.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin. High – level synthesis: Introduction to chip and system design. 1992.

[Geu09a] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34:3–25, 2009.

[Geu09b] Herman Geuvers. Proof assistants : history, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[GLH18] T. Grimm, D. Lettnin, and Michael Hubner. A survey on formal verification techniques for safety-critical systems-on-chip. *Electronics*, 7:81, 2018.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(4):44–54, 1994.

[Gro02] T. Grotker. *System Design with SystemC*. 2002.

[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, 1992.

[HAB+15] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *Computing Research Repository (CoRR)*, abs/1501.02155, 2015.

[HD92] F. K. Hanna and Neil Daeche. Dependent types and formal synthesis. 1992.

[HGLD18] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. Extensible and configurable RISC-V based virtual prototype. In Hiren Patel, Tom J. Kazmierski, and Sebastian Steinhorst, editors, *Forum on Specification and Design Languages (FDL)*, pages 5–16. IEEE, 2018.

[HGPD20] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. RISC-V based virtual prototype: An extensible and configurable platform for the system-level. *Journal of Systems Architecture*, 109:101756, 2020.

[HGW+20] Vladimir Herdt, Daniel Große, Jonas Wloka, Tim Güneysu, and Rolf Drechsler. Verification of embedded binaries using coverage-guided fuzzing with systemc-based virtual prototypes. In Tinoosh Mohsenin, Weisheng Zhao, Yiran Chen, and Onur Mutlu, editors, *ACM Great Lakes Symposium on VLSI*, pages 101–106. ACM, 2020.

[HHL91] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin. Scheduling for functional pipelining and loop winding. In A. Richard Newton, editor, *Design Automation Conference (DAC)*, pages 764–769. ACM, 1991.

[HLW+20] Lan Huang, Dalin Li, Kangping Wang, Teng Gao, and Adriano Tavares. A survey on performance optimization of high-level synthesis tools. *Journal of Computer Science and Technology*, 35(3):697–720, 2020.

[Hof97] M. Hofmann. Syntax and semantics of dependent types. 1997.

[HT15] O. Hasan and S. Tahar. Formal verification methods. 2015.

[HTHT09] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing (JIP)*, 17:242–254, 2009.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Inc16] Accellera Systems Initiative Inc. Systemc synthesizable subset version 1.4.7. 2016. https://www.accellera.org/images/downloads/standards/systemc/SystemC_Synthesis_Subset_1_4_7.pdf.

[Joh83] S. Johnson. Synthesis of digital designs from recursion equations. In *PhD thesis*, 1983.

[KC11] Alexander S. Kamkin and Mikhail M. Chupilko. Survey of modern technologies of simulation-based verification of hardware. *Programming and Computer Software*, 37(3):147–152, 2011.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[KKN+20] Wilayat Khan, Muhammad Kamran, Syed Rameez Naqvi, Farrukh Aslam Khan, Ahmed S. Alghamdi, and Eesa Alsolami. Formal verification of hardware components in critical systems. *Wireless Communications and Mobile Computing*, 2020:7346763:1–7346763:15, 2020.

[KM96] Matt Kaufmann and J. Strother Moore. Acl2: an industrial strength version of nqthm. 1996.

[KSHY19] Wilayat Khan, David Sanán, Zhe Hou, and Liu Yang. On embedding a hardware description language in isabelle/hol. *Design Automation for Embedded Systems*, 23(3-4):123–151, 2019.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, 2004.

[LBK+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*. SEE, 2016.

[Let08] Pierre Letouzey. Extraction in coq: An overview. In *Computability in Europe (CIE)*, pages 359–369, 2008.

[LFD+19] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *Applications of Computer Systems*, 52(5):100:1–100:41, 2019.

[Mar03] Grant Martin. Systemc: From language to applications, from tools to methodologies. In *Symposium on Integrated Circuits and System Design (SBCCI)*, page 3. IEEE Computer Society, 2003.

[MB00] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *International Conference on Types for Proofs and Programs (TYPES), Selected Papers*, volume 2277 of *Lecture Notes in Computer Science (LNCS)*, pages 181–196. Springer, 2000.

[MBP07] Grant Martin, Brian Bailey, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., 2007.

[MGE17] Paul Muntean, Jens Grossklags, and Claudia Eckert. Practical integer overflow prevention. *Computing Research Repository (CoRR)*, abs/1710.03720, 2017.

[Mil79] Robin Milner. LCF: A way of doing proofs with a machine. In Jirí Becvár, editor, *International Symposium on Mathematical Foundations of Com-*

*puter Science (MFCS)*, volume 74 of *Lecture Notes in Computer Science (LNCS)*, pages 146–159. Springer, 1979.

[MKM10] Kevin Marquet, Bageshri Karkare, and Matthieu Moy. A theoretical and experimental review of systemc front-ends. In Adam Morawiec and Jinnie Hinderscheit, editors, *Forum on Specification and Design Languages (FDL)*, pages 124–129. ECSI, Electronic Chips & Systems design Initiative, 2010.

[MMS+18] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. Intrepair: Informed fixing of integer overflows. *Computing Research Repository (CoRR)*, abs/1807.05092, 2018.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[MT08] Inc. MIPS Technologies. MIPS32® Instruction Set Quick Reference. 2008. https://www2.cs.duke.edu/courses/fall13/compsci250/MIPS32_QRC.pdf.

[MXHM20] Guangshuai Mo, Yan Xiong, Wenchao Huang, and Lu Ma. Automated theorem proving via interacting with proof assistants by dynamic strategies. In *International Conference on Big Data Computing and Communications (BIGCOM)*, pages 71–75. IEEE, 2020.

[NK09] Adam Naumowicz and Artur Kornilowicz. A brief overview of mizar. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TOPHOLS)*, volume 5674 of *Lecture Notes in Computer Science (LNCS)*, pages 67–72. Springer, 2009.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. pages 213–228, 2002.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2002.

[NRJ+07] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI Magazine.*, 28(3):13–30, 2007.

[NSO+12] Razvan Nane, Vlad Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. DWARV 2.0: A cosy-based c-to-vhdl hardware compiler. In Dirk Koch, Satnam Singh, and Jim Tørresen, editors, *International Conference on Field programmable Logic and Applications (FPL)*, pages 619–622. IEEE, 2012.

[NSP+16] Razvan Nane, Vlad Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Dean Brown, Fabrizio Ferrandi, Jason Helge Anderson, and Koen Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[Obj12] Object Management Group. Object Constraint Language. Technical Report formal/2014-02-03, OMG, 2012.

[Obj15] Object Management Group. OMG Systems Modeling Language (OMG SysML). Technical Report formal/2015-06-04, OMG, 2015.

[Obj17] Object Management Group. Unified Modeling Language. Technical Report formal/2017-12-05, OMG, 2017.

[OKB+18] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the intel MPX system stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):28:1–28:30, 2018.

[PF13] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *International Conference on Field programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2013.

[Pol97] R. Pollack. How to believe a machine-checked proof. *BRICS Report Series*, 4, 1997.

[PSC+06] Ingo Pill, Simone Semprini, Roberto Cavada, Marco Roveri, Roderick Bloem, and Alessandro Cimatti. Formal analysis of hardware requirements. In Ellen Sentovich, editor, *Design Automation Conference (DAC)*, pages 821–826. ACM, 2006.

[PWD16] Nils Przigoda, Robert Wille, and Rolf Drechsler. Analyzing inconsistencies in UML/OCL models. *Journal of Circuits, Systems, and Computers*, 25(3), 2016.

[PWPD18] Nils Przigoda, Robert Wille, Judith Przigoda, and Rolf Drechsler. *Automated Validation & Verification of UML/OCL Models Using Satisfiability Solvers*. Springer, 2018.

[RBL+19] Martin Ring, Fritjof Bornebusch, Christoph Lüth, Robert Wille, and Rolf Drechsler. Better late than never : Verification of embedded systems after deployment. In Jürgen Teich and Franco Fummi, editors, *Design, Automation and Test in Europe (DATE)*, pages 890–895. IEEE, 2019.

[RBL+20] Martin Ring, Fritjof Bornebusch, Christoph Lüth, Robert Wille, and Rolf Drechsler. Verification runtime analysis: Get the most out of partial verification. In *Design, Automation and Test in Europe (DATE)*, pages 873–878. IEEE, 2020.

[RIS20] RISC-V. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, version 1.12-draft edition, 2020. https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-privileged.pdf.

[SBF+20] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(POPL):8:1–8:28, 2020.

[SH13] Muhammad Usman Sanwal and Osman Hasan. Formal verification of cyber-physical systems: Coping with continuous elements. In Beniamino Murgante, Sanjay Misra, Maurizio Carlini, Carmelo Maria Torre, Hong-Quang Nguyen, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi, editors, *Computational Science and Its Applications (ICCSA)*, volume 7971 of *Lecture Notes in Computer Science (LNCS)*, pages 358–371. Springer, 2013.

[She84] Mary Sheeran. mufp, A language for VLSI design. In *ACM conference on LISP and functional programming (LFP)*, pages 104–112. ACM, 1984.

[SWD13] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Data extraction from systemc designs using debug symbols and the systemc API. In *isvlsi*, pages 26–31, 2013.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation and Test in Europe (DATE)*, pages 1341–1344, 2010.

[Tak16] Andrés Takach. High-level synthesis: Status, trends, and future directions. *IEEE Design & Test*, 33(3):116–124, 2016.

[Tea] CWE Team. 2020 CWE Top 25 Most Dangerous Software Weaknesses. Website. Online available under: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, accessed: 05.03.2021.

[Tei12] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings IEEE*, 100(Centennial-Issue):1411–1430, 2012.

[TH19] Lenny Truong and Pat Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *Summit oN Advances in Programming Languages (SNAPL)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[Val96] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science (LNCS)*, pages 429–528. Springer, 1996.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science (LNCS)*, pages 24–52, 1995.

[Wad15] Philip Wadler. Propositions as types. *Comm. of the ACM*, 58(12):75–84, 2015.

[Wen02] Markus Wenzel. *Isabelle, Isar - a versatile environment for human readable formal proof documents*. PhD thesis, Technical University Munich, Germany, 2002.

[WGHD09] Robert Wille, Daniel Große, Finn Haedicke, and Rolf Drechsler. Smt-based stimuli generation in the systemc verification library. In *Forum on Specification and Design Languages (FDL)*, pages 1–6, 2009.

[WPK+16] Ralph Weissnegger, Markus Pistauer, Christian Kreiner, Markus Schuß, Kay Römer, and Christian Steger. Automatic testbench generation for simulation-based verification of safety-critical systems in UML. In *International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*, pages 70–75, 2016.

[Xia09] Limin Xiang. A formal proof of the four color theorem. *Computing Research Repository (CoRR)*, abs/0905.3713, 2009.

[YD19] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019.

# A

# Appendix for Chapter 3

## A.1 Traffic Light Controller Specification

```
1  Inductive trafficLightCar:=
2      | carRed
3      | carYellow
4      | carGreen
5      | carRedYellow.
6
7  Inductive trafficLightPedestrians:=
8      | pedestriansRed
9      | pedestriansGreen.
10
11 Inductive trafficLightTram:=
12     | tramRed
13     | tramGreen.
14
15 Inductive state :=
16     | State : trafficLightCar -> trafficLightPedestrians ->
17               trafficLightTram -> state.
18
19 Definition tick (s : state) : state :=
20     match s with
21      | State carRed pedestriansRed tramRed =>
22          State carRed pedestriansRed tramGreen
23      | State carRed pedestriansRed tramGreen =>
24          State carRedYellow pedestriansRed tramRed
25      | State carRedYellow pedestriansRed tramRed =>
26          State carGreen pedestriansRed tramRed
27      | State carGreen pedestriansRed tramRed =>
28          State carYellow pedestriansRed tramRed
29      | State carYellow pedestriansRed tramRed =>
```

```
30            State carRed pedestriansGreen tramRed
31        | State _ _ _ => State carRed pedestriansRed tramRed
32      end.
```

**Listing A.1 :** *The specification of the* tick *function in Gallina.*

## A.2 Theorems about the Traffic Light Controller

```
1   Theorem all_trafficLight_never_green_at_the_same_time:
2       forall s : state,
3       tick s <> State carGreen pedestriansGreen tramGreen.
4   Proof.
5       intro s.
6       destruct s as [car walker tram].
7       all: destruct car; destruct walker; destruct tram.
8       all: simpl.
9       all: discriminate.
10  Qed.
```

**Listing A.2 :** *Theorem including its proof that all traffic lights are never green at the same time.*

# B

# Appendix for Chapter 4

## B.1 Lemmas and Theorems about Overflow Detection

```
1  Theorem unsigned_repr:
2    forall z, 0 <= z <= max_unsigned -> unsigned (repr z) = z.
```

**Listing B.1 :** *Theorem about type conversion from Z to finite integer*

```
1  Lemma eqm_samerepr:
2    forall x y, eqm x y -> repr x = repr y.
3
4  Lemma eqm_mult:
5    forall a b c d, eqm a c -> eqm b d -> eqm (a * b) (c * d).
6
7  Lemma eqm_sym:
8    forall x y, eqm x y -> eqm y x.
9
10 Lemma eqm_unsigned_repr:
11   forall z, eqm z (unsigned (repr z)).
```

**Listing B.2 :** *Lemmas for proving the* multUnsigned32mapsMultZ *theorem.*

```
1  Lemma eqm_add:
2    forall a b c d, eqm a b -> eqm c d -> eqm (a + c) (b + d).
3
4  Theorem multZmapsAddUnsigned32 :
5    forall a b,
```

```
6     Types.Unsigned32.add
7        (Types.Unsigned32.repr a) (Types.Unsigned32.repr b) =
8     Types.Unsigned32.repr (a + b).
9   Proof.
10    intros.
11    apply Types.Unsigned32.eqm_samerepr.
12    apply Types.Unsigned32.eqm_add.
13    - apply Types.Unsigned32.eqm_sym.
14      apply Types.Unsigned32.eqm_unsigned_repr.
15    - apply Types.Unsigned32.eqm_sym.
16      apply Types.Unsigned32.eqm_unsigned_repr.
17  Qed.
18
19
20  Theorem detectOverflowAdd:
21    forall a b : Types.Unsigned32.int,
22    (a >? Types.unsigned32_max - b) = true ->
23    safe_add a b = None.
24  Proof.
25    intros.
26    unfold safe_add.
27    rewrite H.
28    reflexivity.
29  Qed.
30
31  Theorem noOverflowAdd:
32    forall a b : Types.Unsigned32.int,
33    a >? Types.unsigned32_max - b = false ->
34    safe_add a b = Some(a+b).
35  Proof.
36    intros.
37    unfold safe_add.
38    rewrite H.
39   reflexivity.
40  Qed.
```

**Listing B.3 :** *Theorems including proofs for the safe_add function.*

# C

# Appendix for Chapter 5

## C.1 32-bit MIPS Processor Specification

```
Definition mips
  (data : registerBankType*memoryType*Unsigned32.int)
  (dummy : bool)
 : (registerBankType*memoryType*Unsigned32.int)*(Unsigned32.int) :=
  match data with
   | (registerBank, memory, pc) =>
      let instr := nth instructionMemory pc in
      let op := srl instr z26 in
      match toFormat op with
        | RFormat =>
           let funct := and instr 0x3f in
           let shamt := and (srl instr z6) 0x1f in
           let rd := and (srl instr z11) 0x1f in
           let rt := and (srl instr z16) 0x1f in
           let rs := and (srl instr z21) 0x1f in

           match toFunctionCode funct with
             | ADDU => let value := add (nth registerBank rs)
                                        (nth registerBank rt) in
                       let rb := replaceAt rd value registerBank  in
                       let pc' := newPC pc in
                       ((rb, memory, pc'), value)


             | SUBU => let value := sub (nth registerBank rs)
                                        (nth registerBank rt) in
                       let rb := replaceAt rd value registerBank in
                       let pc' := newPC pc in
                       ((rb, memory, pc'), value)
```

```
30
31
32              | MULT => let value := mult (nth registerBank rs)
33                                          (nth registerBank rt) in
34                      let rb := replaceAt rd value registerBank in
35                      ((rb, memory, newPC pc), value)
36
37              | MULTU => let value := multu (nth registerBank rs)
38                                           (nth registerBank rt) in
39                       let rb := replaceAt rd value registerBank in
40                       let pc' := newPC pc in
41                       ((rb, memory, pc'), value)
42
43
44              | AND => let value := and (nth registerBank rs)
45                                        (nth registerBank rt) in
46                       let rb := replaceAt rd value registerBank in
47                       let pc' := newPC pc in
48                       ((rb, memory, pc'), value)
49
50
51              | OR => let value := or (nth registerBank rs)
52                                      (nth registerBank rt) in
53                      let rb := replaceAt rd value registerBank in
54                      let pc' := newPC pc in
55                      ((rb, memory, pc'), value)
56
57
58              | XOR => let value := xor (nth registerBank rs)
59                                        (nth registerBank rt) in
60                       let rb := replaceAt rd value registerBank in
61                       let pc' := newPC pc in
62                       ((rb, memory, pc'), value)
63
64
65              | SLL => let value := sll (nth registerBank rt)
66                                        (toInt shamt) in
67                       let rb := replaceAt rd value registerBank in
68                       let pc' := newPC pc in
69                       ((rb, memory, pc'), value)
70
```

```
71
72              | SRL => let value := srl (nth registerBank rt)
73                                        (toInt shamt) in
74                     let rb := replaceAt rd value registerBank in
75                     let pc' := newPC pc in
76                     ((rb, memory, pc'), value)
77
78              | SLLV => let value := sll (nth registerBank rt)
79                                         (toInt (nth registerBank rs)) in
80                      let rb := replaceAt rd value registerBank in
81                      let pc' := newPC pc in
82                      ((rb, memory, pc'), value)
83
84              | SRLV => let value := srl (nth registerBank rt)
85                                         (toInt (nth registerBank rs)) in
86                      let rb := replaceAt rd value registerBank in
87                      let pc' := newPC pc in
88                      ((rb, memory, pc'), value)
89
90
91              | SLT => let value :=
92                        if (toSigned (nth registerBank rt)) <s
93                           (toSigned (nth registerBank rs))
94                           then one else zero in
95                     let rb := replaceAt rd value registerBank in
96                     ((rb, memory, newPC pc), value)
97
98              | SLTU => let value := if (nth registerBank rt) <
99                                        (nth registerBank rs)
100                                       then one else zero in
101                     let rb := replaceAt rd value registerBank in
102                     let pc' := newPC pc in
103                     ((rb, memory, pc'), value)
104
105             | JR => let pc' := nth registerBank rs in
106                    ((registerBank, memory, pc'), pc')
107
108             | _ => ((registerBank, memory, 0), 0) (* error *)
109           end
110
111         | JFormat => match toOperationCode op with
```

```
112                       | J => let tgtadr := and instr 0x3ffffff in
113                              let pc' := sll tgtadr z2 in
114                              ((registerBank, memory, pc'), pc')
115
116                       | JAL => let tgtadr := and instr 0x3ffffff in
117                              let rb := replaceAt 31%unsigned32
118                                                  pc registerBank in
119                              let pc' := sll tgtadr z2 in
120                              ((rb, memory, pc'), pc')
121
122                       | _ => ((registerBank, memory, 0), 0) (* error *)
123                     end
124
125           | IFormat => let address := and instr 0xffff in
126                     let rt := and (srl instr z16) 0x1f in
127                     let rs := and (srl instr z21) 0x1f in
128
129                     match toOperationCode op with
130                     | ADDIU => let value := add (nth registerBank rs)
131                                                 address in
132                                let rb := replaceAt rt value
133                                                    registerBank in
134                                let pc' := newPC pc in
135                                ((rb, memory, pc'), value)
136
137                     | ANDI => let value := and (nth registerBank rs)
138                                                address in
139                                let rb := replaceAt rt value
140                                                    registerBank in
141                                let pc' := newPC pc in
142                                ((rb, memory, pc'), value)
143
144                     | ORI => let value := or (nth registerBank rs)
145                                              address in
146                              let rb := replaceAt rt value
147                                                  registerBank in
148                              let pc' := newPC pc in
149                              ((rb, memory, pc'), value)
150
151                     | XORI => let value := xor (nth registerBank rs)
152                                                address in
```

```
153                         let rb := replaceAt rt value
154                                                registerBank in
155                         let pc' := newPC pc in
156                         ((rb, memory, pc'), value)
157
158                   | LW => let value := nth memory rs  in
159                         let rb := replaceAt rt value
160                                                registerBank in
161                         let pc' := newPC pc in
162                         ((rb, memory, pc'), value)
163
164                   | SW => let value := nth registerBank rt in
165                         let memory' := replaceAt rs value
166                                                   memory in
167                         let pc' := newPC pc in
168                         ((registerBank, memory', pc'), value)
169
170                   | LUI => let value := sll address z16 in
171                          let rb := replaceAt rt value
172                                                 registerBank in
173                          let pc' := newPC pc in
174                          ((rb, memory, pc'), value)
175
176                   | BEQ => let pc' := if (nth registerBank rs) =?
177                                          (nth registerBank rt)
178                                          then address else pc in
179                          ((registerBank, memory, pc'), pc')
180
181                   | BNE => let value := add (nth registerBank rs)
182                                             address in
183                          let rb := replaceAt rt value
184                                                 registerBank in
185                          ((rb, memory, newPC pc), value)
186
187                   | BGEZ => let value := add (nth registerBank rs)
188                                              address in
189                           let rb := replaceAt rt value
190                                                  registerBank in
191                           ((rb, memory, newPC pc), value)
192
193                   | SLTI => let value :=
```

```
194                                   if (toSigned (nth registerBank rs)) <s
195                                      (toSigned address)
196                                      then 1 else 0 in
197                                   let rb := replaceAt rt value
198                                                       registerBank in
199                                   ((rb, memory, newPC pc), value)
200
201                         | SLTIU => let value := if (nth registerBank rs) <
202                                                 address
203                                                 then 1 else 0 in
204                                   let rb := replaceAt rt value
205                                                       registerBank in
206                                   let pc' := newPC pc in
207                                   ((rb, memory, pc'), value)
208
209                         | _ => ((registerBank, memory, 0), 0) (* error *)
210
211                     end
212           end
213     end.
```

**Listing C.1 :** *Specification of a 32-bit MIPS processor in Gallina.*

## C.2 THEOREMS PROVED BY CUSTOM PROOF METHODS

```
1
2   Theorem mipsSUBU:
3     forall registerBank : registerBankType,
4     forall memory : memoryType,
5     forall pc : Unsigned32.int,
6     forall dummy : bool,
7
8     let instr := nth instructionMemory pc in
9     let op := srl instr z26 in
10    let funct := and instr 0x3f in
11    let rd := and (srl instr z11) 0x1f in
```

```
12    let rt := and (srl instr z16) 0x1f in
13    let rs := and (srl instr z21) 0x1f in
14    let value := sub (nth registerBank rs) (nth registerBank rt) in
15
16    toFormat op = RFormat /\ toFunctionCode funct = SUBU ->
17    mips (registerBank, memory, pc) dummy =
18      ((replaceAt rd value registerBank, memory, newPC pc), value).
19  Proof.
20    proveRFormat.
21  Qed.
22
23
24  Theorem mipsMULT:
25    forall registerBank : registerBankType,
26    forall memory : memoryType,
27    forall pc : Unsigned32.int,
28    forall dummy : bool,
29
30    let instr := nth instructionMemory pc in
31    let op := srl instr z26 in
32    let funct := and instr 0x3f in
33    let rd := and (srl instr z11) 0x1f in
34    let rt := and (srl instr z16) 0x1f in
35    let rs := and (srl instr z21) 0x1f in
36    let value := mult (nth registerBank rs) (nth registerBank rt) in
37
38    toFormat op = RFormat /\ toFunctionCode funct = MULT ->
39    mips (registerBank, memory, pc) dummy =
40      ((replaceAt rd value registerBank, memory, newPC pc), value).
41  Proof.
42    proveRFormat.
43  Qed.
44
45
46  Theorem mipsMULTU:
47    forall registerBank : registerBankType,
48    forall memory : memoryType,
49    forall pc : Unsigned32.int,
50    forall dummy : bool,
51
52    let instr := nth instructionMemory pc in
```

```
53    let op := srl instr z26 in
54    let funct := and instr 0x3f in
55    let rd := and (srl instr z11) 0x1f in
56    let rt := and (srl instr z16) 0x1f in
57    let rs := and (srl instr z21) 0x1f in
58    let value := multu (nth registerBank rs) (nth registerBank rt) in
59
60    toFormat op = RFormat /\ toFunctionCode funct = MULTU ->
61    mips (registerBank, memory, pc) dummy =
62      ((replaceAt rd value registerBank, memory, newPC pc), value).
63  Proof.
64    proveRFormat.
65  Qed.
66
67  Theorem mipsAND:
68    forall registerBank : registerBankType,
69    forall memory : memoryType,
70    forall pc : Unsigned32.int,
71    forall dummy : bool,
72
73    let instr := nth instructionMemory pc in
74    let op := srl instr z26 in
75    let funct := and instr 0x3f in
76    let rd := and (srl instr z11) 0x1f in
77    let rt := and (srl instr z16) 0x1f in
78    let rs := and (srl instr z21) 0x1f in
79    let value := and (nth registerBank rs) (nth registerBank rt) in
80
81    toFormat op = RFormat /\ toFunctionCode funct = AND ->
82    mips (registerBank, memory, pc) dummy =
83      ((replaceAt rd value registerBank, memory, newPC pc), value).
84  Proof.
85    proveRFormat.
86  Qed.
87
88  Theorem mipsOR:
89    forall registerBank : registerBankType,
90    forall memory : memoryType,
91    forall pc : Unsigned32.int,
92    forall dummy : bool,
93
```

```
  let instr := nth instructionMemory pc in
  let op := srl instr z26 in
  let funct := and instr 0x3f in
  let rd := and (srl instr z11) 0x1f in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := or (nth registerBank rs) (nth registerBank rt) in

  toFormat op = RFormat /\ toFunctionCode funct = OR ->
  mips (registerBank, memory, pc) dummy =
    ((replaceAt rd value registerBank, memory, newPC pc), value).
Proof.
  proveRFormat.
Qed.

Theorem mipsXOR:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in
  let op := srl instr z26 in
  let funct := and instr 0x3f in
  let rd := and (srl instr z11) 0x1f in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := xor (nth registerBank rs) (nth registerBank rt) in

  toFormat op = RFormat /\ toFunctionCode funct = XOR ->
  mips (registerBank, memory, pc) dummy =
    ((replaceAt rd value registerBank, memory, newPC pc), value).
Proof.
  proveRFormat.
Qed.

Theorem mipsSLL:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,
```

```
135
136     let instr := nth instructionMemory pc in
137     let op := srl instr z26 in
138     let funct := and instr 0x3f in
139     let rd := and (srl instr z11) 0x1f in
140     let shamt := and (srl instr z6) 0x1f in
141     let rt := and (srl instr z16) 0x1f in
142     let value := sll (nth registerBank rt) (toInt shamt) in
143
144     toFormat op = RFormat /\ toFunctionCode funct = SLL ->
145     mips (registerBank, memory, pc) dummy =
146       ((replaceAt rd value registerBank, memory, newPC pc), value).
147   Proof.
148     proveRFormat.
149   Qed.
150
151   Theorem mipsSRL:
152     forall registerBank : registerBankType,
153     forall memory : memoryType,
154     forall pc : Unsigned32.int,
155     forall dummy : bool,
156
157     let instr := nth instructionMemory pc in
158     let op := srl instr z26 in
159     let funct := and instr 0x3f in
160     let rd := and (srl instr z11) 0x1f in
161     let shamt := and (srl instr z6) 0x1f in
162     let rt := and (srl instr z16) 0x1f in
163     let value := srl (nth registerBank rt) (toInt shamt) in
164
165     toFormat op = RFormat /\ toFunctionCode funct = SRL ->
166     mips (registerBank, memory, pc) dummy =
167       ((replaceAt rd value registerBank, memory, newPC pc), value).
168   Proof.
169     proveRFormat.
170   Qed.
171
172   Theorem mipsSLLV:
173     forall registerBank : registerBankType,
174     forall memory : memoryType,
175     forall pc : Unsigned32.int,
```

```
176     forall dummy : bool,
177
178     let instr := nth instructionMemory pc in
179     let op := srl instr z26 in
180     let funct := and instr 0x3f in
181     let rd := and (srl instr z11) 0x1f in
182     let rt := and (srl instr z16) 0x1f in
183     let rs := and (srl instr z21) 0x1f in
184     let value := sll (nth registerBank rt)
185                      (toInt (nth registerBank rs)) in
186
187     toFormat op = RFormat /\ toFunctionCode funct = SLLV ->
188     mips (registerBank, memory, pc) dummy =
189       ((replaceAt rd value registerBank, memory, newPC pc), value).
190   Proof.
191     proveRFormat.
192   Qed.
193
194   Theorem mipsSRLV:
195     forall registerBank : registerBankType,
196     forall memory : memoryType,
197     forall pc : Unsigned32.int,
198     forall dummy : bool,
199
200     let instr := nth instructionMemory pc in
201     let op := srl instr z26 in
202     let funct := and instr 0x3f in
203     let rd := and (srl instr z11) 0x1f in
204     let rt := and (srl instr z16) 0x1f in
205     let rs := and (srl instr z21) 0x1f in
206     let value := srl (nth registerBank rt)
207                      (toInt (nth registerBank rs)) in
208
209     toFormat op = RFormat /\ toFunctionCode funct = SRLV ->
210     mips (registerBank, memory, pc) dummy =
211       ((replaceAt rd value registerBank, memory, newPC pc), value).
212   Proof.
213     proveRFormat.
214   Qed.
215
216   Theorem mipsSLT:
```

```
217     forall registerBank : registerBankType,
218     forall memory : memoryType,
219     forall pc : Unsigned32.int,
220     forall dummy : bool,
221
222     let instr := nth instructionMemory pc in
223     let op := srl instr z26 in
224     let funct := and instr 0x3f in
225     let rd := and (srl instr z11) 0x1f in
226     let rt := and (srl instr z16) 0x1f in
227     let rs := and (srl instr z21) 0x1f in
228     let value := if (toSigned (nth registerBank rt)) <s
229                     (toSigned (nth registerBank rs))
230                     then one else zero in
231
232     toFormat op = RFormat /\ toFunctionCode funct = SLT ->
233     mips (registerBank, memory, pc) dummy =
234       ((replaceAt rd value registerBank, memory, newPC pc), value).
235   Proof.
236     proveRFormat.
237   Qed.
238
239   Theorem mipsSLTU:
240     forall registerBank : registerBankType,
241     forall memory : memoryType,
242     forall pc : Unsigned32.int,
243     forall dummy : bool,
244
245     let instr := nth instructionMemory pc in
246     let op := srl instr z26 in
247     let funct := and instr 0x3f in
248     let rd := and (srl instr z11) 0x1f in
249     let rt := and (srl instr z16) 0x1f in
250     let rs := and (srl instr z21) 0x1f in
251     let value := if (nth registerBank rt) <
252                     (nth registerBank rs)
253                     then one else zero in
254
255     toFormat op = RFormat /\ toFunctionCode funct = SLTU ->
256     mips (registerBank, memory, pc) dummy =
257       ((replaceAt rd value registerBank, memory, newPC pc), value).
```

```
258  Proof.
259    proveRFormat.
260  Qed.
261
262  Theorem mipsJR:
263    forall registerBank : registerBankType,
264    forall memory : memoryType,
265    forall pc : Unsigned32.int,
266    forall dummy : bool,
267
268    let instr := nth instructionMemory pc in
269    let op := srl instr z26 in
270    let funct := and instr 0x3f in
271    let rs := and (srl instr z21) 0x1f in
272    let pc' := nth registerBank rs in
273
274    toFormat op = RFormat /\ toFunctionCode funct = JR ->
275    mips (registerBank, memory, pc) dummy =
276      ((registerBank, memory, pc'), pc').
277  Proof.
278    proveRFormat.
279  Qed.
280
281  Theorem mipsUnknownFunctionCode:
282    forall registerBank : registerBankType,
283    forall memory : memoryType,
284    forall pc : Unsigned32.int,
285    forall dummy : bool,
286
287    let instr := nth instructionMemory pc in
288    let op := srl instr z26 in
289    let funct := and instr 0x3f in
290
291    toFormat op = RFormat /\ toFunctionCode funct = UNKNOWN ->
292    mips (registerBank, memory, pc) dummy =
293      ((registerBank, memory, 0), 0). (* error *)
294  Proof.
295    prove_RFormat.
296  Qed.
```

**Listing C.2 :** *Theorem including their proofs about the specified 32-bit MIPS processor instructions using the R Format.*

```
Theorem mipsANDI:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in

  let op := srl instr z26 in
  let address := and instr 0xffff in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := and (nth registerBank rs) address in

  toFormat op = IFormat /\ toOperationCode op = ANDI ->
  mips (registerBank, memory, pc) dummy =
    ((replaceAt rt value registerBank, memory, newPC pc), value).
Proof.
  proveIFormat.
Qed.

Theorem mipsORI:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in

  let op := srl instr z26 in
  let address := and instr 0xffff in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := or (nth registerBank rs) address in

  toFormat op = IFormat /\ toOperationCode op = ORI ->
  mips (registerBank, memory, pc) dummy =
    ((replaceAt rt value registerBank, memory, newPC pc), value).
Proof.
  proveIFormat.
Qed.
```

```
42
43  Theorem mipsXORI:
44    forall registerBank : registerBankType,
45    forall memory : memoryType,
46    forall pc : Unsigned32.int,
47    forall dummy : bool,
48
49    let instr := nth instructionMemory pc in
50
51    let op := srl instr z26 in
52    let address := and instr 0xffff in
53    let rt := and (srl instr z16) 0x1f in
54    let rs := and (srl instr z21) 0x1f in
55    let value := xor (nth registerBank rs) address in
56
57    toFormat op = IFormat /\ toOperationCode op = XORI ->
58    mips (registerBank, memory, pc) dummy =
59      ((replaceAt rt value registerBank, memory, newPC pc), value).
60  Proof.
61    proveIFormat.
62  Qed.
63
64  Theorem mipsLW:
65    forall registerBank : registerBankType,
66    forall memory : memoryType,
67    forall pc : Unsigned32.int,
68    forall dummy : bool,
69
70    let instr := nth instructionMemory pc in
71
72    let op := srl instr z26 in
73    let address := and instr 0xffff in
74    let rt := and (srl instr z16) 0x1f in
75    let rs := and (srl instr z21) 0x1f in
76    let value := nth memory rs in
77
78    toFormat op = IFormat /\ toOperationCode op = LW ->
79    mips (registerBank, memory, pc) dummy =
80      ((replaceAt rt value registerBank, memory , newPC pc), value).
81  Proof.
82    proveIFormat.
```

```
Qed.

Theorem mipsSW:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in

  let op := srl instr z26 in
  let address := and instr 0xffff in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := nth registerBank rt in

  toFormat op = IFormat /\ toOperationCode op = SW ->
  mips (registerBank, memory, pc) dummy =
    ((registerBank, replaceAt rs value memory, newPC pc), value).
Proof.
  proveIFormat.
Qed.

Theorem mipsLUI:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in

  let op := srl instr z26 in
  let address := and instr 0xffff in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := sll address z16 in

  toFormat op = IFormat /\ toOperationCode op = LUI ->
  mips (registerBank, memory, pc) dummy =
    ((replaceAt rt value registerBank, memory, newPC pc), value).
Proof.
```

```
124     proveIFormat.
125   Qed.
126
127   Theorem mipsBEQ:
128     forall registerBank : registerBankType,
129     forall memory : memoryType,
130     forall pc : Unsigned32.int,
131     forall dummy : bool,
132
133     let instr := nth instructionMemory pc in
134
135     let op := srl instr z26 in
136     let address := and instr 0xffff in
137     let rt := and (srl instr z16) 0x1f in
138     let rs := and (srl instr z21) 0x1f in
139     let pc' := if (nth registerBank rs) =?
140                   (nth registerBank rt)
141                   then address else pc in
142
143     toFormat op = IFormat /\ toOperationCode op = BEQ ->
144     mips (registerBank, memory, pc) dummy =
145       ((registerBank, memory, pc'), pc').
146   Proof.
147     proveIFormat.
148   Qed.
149
150   Theorem mipsBNE:
151     forall registerBank : registerBankType,
152     forall memory : memoryType,
153     forall pc : Unsigned32.int,
154     forall dummy : bool,
155
156     let instr := nth instructionMemory pc in
157
158     let op := srl instr z26 in
159     let address := and instr 0xffff in
160     let rt := and (srl instr z16) 0x1f in
161     let rs := and (srl instr z21) 0x1f in
162     let value := add (nth registerBank rs) address in
163
164     toFormat op = IFormat /\ toOperationCode op = BNE ->
```

```
    mips (registerBank, memory, pc) dummy =
      ((replaceAt rt value registerBank , memory, newPC pc), value).
Proof.
  proveIFormat.
Qed.

Theorem mipsBGEZ:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in

  let op := srl instr z26 in
  let address := and instr 0xffff in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := add (nth registerBank rs) address in

  toFormat op = IFormat /\ toOperationCode op = BGEZ ->
  mips (registerBank, memory, pc) dummy =
    ((replaceAt rt value registerBank, memory, newPC pc), value).
Proof.
  proveIFormat.
Qed.

Theorem mipsSLTI:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in

  let op := srl instr z26 in
  let address := and instr 0xffff in
  let rt := and (srl instr z16) 0x1f in
  let rs := and (srl instr z21) 0x1f in
  let value := if (toSigned (nth registerBank rs)) <s
                  (toSigned address)
```

```
206                   then 1 else 0 in
207
208     toFormat op = IFormat /\ toOperationCode op = SLTI ->
209     mips (registerBank, memory, pc) dummy =
210       ((replaceAt rt value registerBank, memory, newPC pc), value).
211   Proof.
212     proveIFormat.
213   Qed.
214
215   Theorem mipsSLTU:
216     forall registerBank : registerBankType,
217     forall memory : memoryType,
218     forall pc : Unsigned32.int,
219     forall dummy : bool,
220
221     let instr := nth instructionMemory pc in
222
223     let op := srl instr z26 in
224     let address := and instr 0xffff in
225     let rt := and (srl instr z16) 0x1f in
226     let rs := and (srl instr z21) 0x1f in
227     let value := if (nth registerBank rs) <
228                   address then 1 else 0 in
229
230     toFormat op = IFormat /\ toOperationCode op = SLTIU ->
231     mips (registerBank, memory, pc) dummy =
232       ((replaceAt rt value registerBank, memory, newPC pc), value).
233   Proof.
234     proveIFormat.
235   Qed.
```

**Listing C.3 :** *Theorem including their proofs about the specified 32-bit MIPS processor instructions using the I Format.*

```
1   Theorem mipsJAL:
2     forall registerBank : registerBankType,
3     forall memory : memoryType,
4     forall pc : Unsigned32.int,
5     forall dummy : bool,
```

```
6
7     let instr := nth instructionMemory pc in
8     let op := srl instr z26 in
9     let tgtadr := and instr 0x3ffffff in
10    let pc' := sll tgtadr z2 in
11
12    toFormat op = JFormat /\ toOperationCode op = JAL ->
13    mips (registerBank, memory, pc) dummy =
14      ((replaceAt 31%unsigned32 pc registerBank, memory, pc'), pc').
15  Proof.
16    proveJFormat.
17  Qed.
```

**Listing C.4 :** *Theorem including their proofs about the specified 32-bit MIPS processor instructions using the J Format.*

## C.3 General Theorems

```
1   Theorem mipsNOPisSLL:
2     forall registerBank : registerBankType,
3     forall memory : memoryType,
4     forall pc output : Unsigned32.int,
5     forall dummy : bool,
6
7     let nop := 0x00000000 in
8
9     let registerBank' := replaceAt (and (srl nop z11) 0x1f)
10                         (sll (nth registerBank (and (srl nop z16) 0x1f))
11                         (toInt (and (srl nop z6) 0x1f))) registerBank in
12
13    let output := sll (nth registerBank (and (srl nop z16) 0x1f))
14                  (toInt (and (srl nop z6) 0x1f)) in
15
16    nth instructionMemory pc = nop ->
17    mips (registerBank, memory, pc) dummy =
18      ((registerBank', memory, newPC pc), output).
```

```
Proof.
  intros.
  unfold mips.
  rewrite H.
  simpl.
  reflexivity.
Qed.

Theorem mipsUnknownOperationCode:
  forall registerBank : registerBankType,
  forall memory : memoryType,
  forall pc : Unsigned32.int,
  forall dummy : bool,

  let instr := nth instructionMemory pc in
  let op := srl instr z26 in
  let funct := and instr 0x3f in

  (toFormat op = JFormat \/ toFormat op = IFormat) /\
  toOperationCode op = UNKNOWN ->
  mips (registerBank, memory, pc) dummy =
    ((registerBank, memory, 0), 0). (* error *)
Proof.
  intros.
  destruct H as [H1 H2].
  destruct H1 as [H11 | H12].
  unfold op in H11.
  unfold instr in H11.
  unfold op in H2.
  unfold instr in H2.
  unfold mips.
  rewrite H11.
  rewrite H2.
  reflexivity.
  unfold op in H12.
  unfold instr in H12.
  unfold op in H2.
  unfold instr in H2.
  unfold mips.
  rewrite H12.
  rewrite H2.
```

```
60      reflexivity.
61    Qed.
```

**Listing C.5 :** *Theorems including their proofs about the 32-bit MIPS processor.*