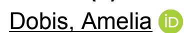
ETH zürich

ETH Library

Formal Verification of Hardware using MLIR

Master Thesis

Author(s):

Dobis, Amelia

Publication date:
2024

Permanent link:
https://doi.org/10.3929/ethz-b-000668906

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Formal Verification of Hardware using MLIR

Master Thesis

Master of Science (MSc) in Computer Science

Amelia Dobis

April 15, 2024

Advisors: Kevin Laeufer[1], Prof. Dr. Zhendong Su[2]

[1]Department of Electrical Engineering and Computer Science, UC Berkeley

[2]Department of Computer Science, ETH Zürich

Acknowledgments

Thank you to Zhendong Su for taking the time to follow my progress throughout the past 6 months.

I also thank the CIRCT developers, particularly Andrew Lenharth, Fabian Schuiki, Mike Urbach, and Leon Hielscher for the detailed reviews and discussions that we had around this work.

Thank you to Morten Borup Petersen for sharing his in-depth knowledge about the CIRCT project with me and helping me acquire the necessary skills to start, debug, and complete this work.

Thank you to the students of the SLICE lab, in particular Tianrui Wei, who shared the valuable expertise he had from working on a similar problem in the past, and Charles Hong, who took the time to help me debug and run my automated testing suite.

Thank you to my friends Marie, Matthieu, and Fred who have supported me along the way and came halfway across the world to California with me to help me move and settle into this foreign place. I would also like to thank my cats Sami and Tiramisu for keeping me sane during the difficult periods of this thesis.

A special thanks is needed for eyes robson who supported me throughout my thesis, during both the difficult and easier times, and took the time to meticulously proofread the final work.

Finally, thank you to Kevin Laeufer for being an amazing advisor, not only for this thesis, but also for leading me through this difficult project, for helping me apply to PhD programs, for helping me get an internship, and for generally being a good person who always took my well-being into account and allowed me to do the best work I could have done during this short thesis. I hope that I will be able to work with you again in the future.

Berkeley, California, USA - April 15th 2024 - Amelia Dobis

Abstract

Modern hardware development is increasingly turning to high-level hardware construction languages to speed up hardware development. CIRCT aims to unify these languages into a single MLIR-based compiler, which allows for various domain-specific intermediate representations (IR) to be used in the same bit of code. While the design capabilities of these new languages surpass those of their predecessors, i.e. SystemVerilog and VHDL, they lag far behind them in terms of verification functionalities. As a result, engineers implementing designs in high-level hardware construction languages find themselves relying on SystemVerilog to verify those designs. In this thesis, I aim to unify verification support for these high-level hardware languages into a single compiler. To do so, I implement a novel formal back end for the CIRCT compiler that supports the creation of formal descriptions of digital hardware designs implemented in any of CIRCT's front ends, so they can be used for Bounded Model Checking (BMC). Additionally, I enable the use of SystemVerilog-like temporal properties through direct lowerings into synthesizable logic, so they can be supported by any tool that understands synthesizable logic, including my formal back end. These two passes are verified using custom automated testing tools. With these contributions to CIRCT, engineers now have access to an open-source, end-to-end formal verification flow that supports temporal specifications, that is entirely integrated into a single compiler.

Contents

Acknowledgments i

Abstract iii

Contents v

1 Introduction 1

2 Background 3
2.1 Hardware Development 3
2.1.1 Hardware Languages 3
2.1.2 Designing Hardware 4
2.1.3 Verifying Hardware 4
2.2 SystemVerilog 5
2.2.1 Overview of the Language 5
2.2.2 Linear Temporal Logic (LTL) 6
2.2.3 SystemVerilog Assertions 8
2.3 Formal Verification 10
2.3.1 Boolean Satisfiability and SMT 10
2.3.2 Bounded Model Checking 10
2.3.3 The BTOR2 Format and BTORMC 12
2.4 CIRCT : Circuit IR Compilers and Tools 14
2.4.1 Hardware Dialects 15

3 Formal Back End for CIRCT 19
3.1 Formal Compilation Flow 19
3.1.1 Simple Example 19
3.1.2 Chisel Elaboration 21
3.1.3 Compiling FIRRTL to the Core Dialects 21
3.1.4 Compiling the Core Dialects to BTOR2 21
3.1.5 Checking the BTOR2 Model 21

3.2 Compiling Hardware to First-Order Logic 23
3.2.1 Combinatorial Circuits 23
3.2.2 Sequential Circuits 26
3.3 Results 28

4 Encoding Temporal Properties for Bounded Model Checking 31
4.1 Encoding Property Assertions 31
4.1.1 The `AssertProperty` Statement 31
4.2 Implementing SVA Properties in CIRCT 33
4.2.1 A Study of SVA Property Usage 33
4.2.2 Implementing Non-Overlapping Implication 35

5 Verifying the Compiler Passes 41
5.1 Verifying the BTOR2 Emission 41
5.1.1 Creating a Miter Circuit 44
5.2 Verifying the SVA Property Implementation 47
5.2.1 Cycle-bound Exhaustive Differential Testing 47
5.2.2 Semantic Equivalence Checking 51

6 Related Work 55
6.1 ChiselTest 55
6.2 Yosys 56
6.3 BlueSpec Verilog 57

7 Conclusion 59
7.1 Results 59
7.2 Future Work 60
7.3 Conclusion 62

Bibliography 65

Chapter 1

Introduction

The ever-increasing demand for performance has lead to a rise in domain-specific accelerators [24]. These accelerators are being design and implemented under ever-shortening time constraints. However, traditional hardware development languages lack the efficiency needed to keep up with this trend. As a result, several high-level hardware languages were born, each promising to increase the efficiency of implementing designs [10, 49]. Many of these languages build on-top of previous infrastructure and add their own custom compiler that allows them to map their high-level semantics to low level circuit descriptions in a common industry standard language like Verilog using a software-embedded domain-specific language (DSL) [40, 64, 9, 62]. This approach led to many different tool-chains being created, each with a relatively low adoption rate. CIRCT [48] is a unified hardware compiler based on MLIR [47] that offers a solution to this problem by using many of these high-level hardware languages as its front ends and by supporting several targets including Verilog [32].

When these high-level languages were developed, the focus was mostly on design functionalities, not verification [9]. This created a gap in the capabilities of these tools. While implementing a design could be done very efficiently, verifying the same design relies on manipulating the compiled Verilog rather than the high-level source design. This reliance on tools for a different language drastically increases the overhead of verifying designs implemented in a high-level hardware language and has limited their adoption outside of academia.

Because of the high costs of fabricating hardware, engineers are often required to give strong guarantees on the correctness of their designs. Formal verification [14], i.e. using static analysis and symbolic techniques to prove the correctness of a design against a specification, has become a popular approach [41]. While it requires specialized knowledge to write correct specifications, checking them using Bounded Model Checking provides stronger

guarantees than many dynamic approaches.

One main issue related to writing specifications for hardware design is the difficulty in expression timing constraints, which are very important in sequential designs, in a concise way that can used in formal verification. An industry standard for this is SystemVerilog Assertion properties and sequences [1] for SystemVerilog [6], and PSL [5] for VHDL [3], which allow the user to express sequences of events and relations between events across multiple clock cycles. These temporal semantics are complex and difficult to implement [18], thus only a small subset of often commercial tools for Verilog (the target for many of these high-level languages) support it, greatly limiting the use of such an important set of semantics.

This leads to the main problem this work is trying to solve. With the existence of a unified compiler for high-level languages [48], there should be also be a unified infrastructure for verification. In this thesis, I solve one of the verification gaps, by introducing an end-to-end compilation flow for formal verification. In addition, I unify the encoding and implementation of temporal logic in the same compiler in order to use it to express timing relations within my verification flow. This new, fully open-source, end-to-end formal verification flow is then verified using custom automated testing techniques. This formal verification flow is fully integrated into the compiler for Chisel [9], a widely used high-level hardware language, called `firtool` through the use of a single flag.

This work introduces two new compiler passes to the CIRCT compiler, a new compilation flow integrated into `firtool`, and two open-source automated testing tools for `btor2` [53] and SystemVerilog. The first compiler pass converts the core design dialects of the CIRCT project into the `btor2` format for bounded model checking [21]. The second pass introduces a lowering for the two most common temporal properties used in practice, non-overlapping implication and concatenation, and outputs an MLIR description in the core dialects. These two passes are combined into the `--btor2` flag of the compiler and allow for an end-to-end compilation flow from any of CIRCT's front ends to a model checking format, while supporting the use of temporal relations in the specification.

The contributions of this work are available as part of the CIRCT project[1]. The formal compilation flow is fully merged into the main project, and the SVA property and sequence lowering has gone through extensive reviews and is expected to be merged into CIRCT in the near future.

[1] https://github.com/llvm/circt

Chapter 2

Background

This chapter covers the necessary background for this thesis, including concepts from hardware development, formal verification, the CIRCT compiler, SystemVerilog, and Linear Temporal Logic (LTL).

2.1 Hardware Development

We begin with a discussion about current hardware development practices, the tools they rely on, and the types of languages they use to express their designs.

2.1.1 Hardware Languages

There many different types of languages that engineers can use to describe the digital hardware designs they are trying to implement. We will group these languages into three broad categories:

- **Hardware Description Languages (HDL):** These languages directly express the hardware we are trying to describe, usually at the Register Transfer Level (RTL) [30]. This includes languages such as Verilog [6] or VHDL [3], which are the industry standard.
- **Hardware Construction Languages:** These languages are often used to express high-level parameterized concepts that can then be used to generate lower-level RTL descriptions. These have become popular with agile hardware development methodologies [49, 10] as they allow for a more efficient description of hardware. Languages such as Chisel [9], Magma [64], or Amaranth [70] can be included in this category.
- **High-Level Synthesis (HLS):** This category encompasses methods that transform a behavioral software program into a hardware description with logically equivalent semantics [22]. This is often used in conjunc-

tion with other high-level hardware languages to simplify the creation of domain-specific hardware accelerators [56].

There are of course other categories, such as rule-based languages like BlueSpec [16] or BlueSpec Verilog [55], or domain-specific languages such as the Processor Description Language [75]. In this work, we focus on expanding the verification-related capabilities of hardware construction languages such as Chisel, Magma, and also newer innovative hardware infrastructures such as Calyx [54].

2.1.2 Designing Hardware

Digital hardware designs can very broadly be separated into two categories:

- **Combinatorial Designs:** These designs do not depend on a clock. The outputs of these designs are purely logical function of the current inputs, so there is no state in the circuit.
- **Sequential Designs:** These depend on a clock which triggers the storage and updating of information inside of stateful elements such as registers or memory. In a sequential design, the output is a function of both current and past input values.

Generally speaking, hardware is often a combination of both combinatorial and sequential logic, and thus any encoding of it should support both types of logic [57].

2.1.3 Verifying Hardware

Unlike software, the final goal of designing hardware is often to produce a physical silicon chip that can be used in real-world systems. This production process is called a "tape-out" and is incredibly expensive to do. Because of this, a lot of effort is put in to verifying that a design is near perfect, or at least matches its specification, before producing the physical chip [41].

This process is often done by relying on design simulation, which is the process of converting a design into a semantically equivalent software model that can be run and interacted with on a computer. The most popular HDL is currently SystemVerilog [6], so many simulators have been implemented for this language, both as open-source and commercial tools.

The way we stimulate our design is by wrapping it in what is called a test bench. This is simply a program, usually described in the same language as the design, that instantiates the Design Under Test (DUT) and feeds it inputs and checks its outputs [57]. This can of course be made arbitrarily complex using methods such as hardware design fuzzing [46, 63, 37] and constrained random testing [28]. In HDLs such as SystemVerilog, test benches can be

written in SystemVerilog. They function by defining the interface for the DUT and then stimulating it with inputs before using this interface to check the value of the outputs. This check is usually done against a software model of the same design, called the golden model. Some tools, such as `cocotb` [35] allow for test benches for HDLs like SystemVerilog or VHDL to be written in Python instead of in the HDL itself.

For Hardware Construction Languages (HCLs), the idea is to utilize the general-purpose programming languages that they are embedded in to write the test benches [9]. This requires testing libraries that allow the user to interact with a DUT during a simulation directly in a software-based test, e.g. ChiselTest [50], which can be used to interact with and simulate Chisel designs from a Scala [68] test bench. This design decision has led to certain gaps in the verification-specific functionalities of HCLs, so oftentimes using the host-language's features isn't enough. This is why certain additional libraries such as ChiselTest and ChiselVerify [28] were created. These add verification and test bench capabilities to Chisel through a small set of custom instrumentation passes and a robust tool set built directly in Scala. This allows for similar functionalities to solutions such as `cocotb` but directly in the source language of the design, thus reducing the overall tooling overhead.

A constant in all of these solutions is that they are small and often incomplete compared to the hyper-engineered commercial tools that are available for the traditional RTL languages.

2.2 SystemVerilog

SystemVerilog is the core language upon which most high-level hardware languages rely [9, 64, 62, 70], and much of the work presented in this thesis will either use or transform elements of this language. This section serves as a brief introduction to SystemVerilog while focusing mainly on its verification functionalities.

2.2.1 Overview of the Language

SystemVerilog is an extension of the HDL Verilog that adds a lot of non-synthesizable elements, i.e. constructs that do not explicitly describe hardware, such as object-oriented programming concepts, several verification functionalities [4], and an entirely new sub-language, known as SystemVerilog Assertions [1] used to express temporal logic [6]. The language consists of over 250 keywords that allow for the expression of incredibly complex constructs across several domains. This amount of complexity in the language has lead to many compilers and tools for this language only supporting a subset of the sub-language — usually the core Verilog language with a few additional verification constructs [20]. Figure 2.1 shows an example of a

```
module Counter(input clock, reset, en);
  logic [31:0] count; // 32-bit register named count
  always @(posedge clock) begin
    if (reset)
      count <= 32'h0; // On reset set count to 0
    else if (count != 32'h16 & en) // if(count is not 22)
      count <= count + 32'h1;
    else if (count == 32'h16 & en)
      count <= 32'h0;
    assert(count != 32'ha); // check that count is not 10
  end // always @(posedge)
endmodule
```

Figure 2.1: Example of a sequential design described in SystemVerilog. Modules are the core abstraction level used in SystemVerilog (similar to a struct in software languages). Every construct in the language is defined using explicit bit-widths. An important distinction from general-purpose programming languages, which mostly focus on sequential execution, is the highly concurrent event-driven style expressed using `@` keywords, e.g. `@(posedge clock)` meaning "in the event of a rising edge of the clock". The `always` keyword is used to signify indefinite repetition, which in practice starts a process that is triggered by its input sensitivity list. Literals are described in hexadecimal using `'h`, decimal using `'d`, or binary using `'b`, and must have a bit-width, e.g. 32'h16 is a 32-bit literal of value 22. This circuit is a direct lowering of the Chisel design described in Figure 2.8 using CIRCT.

simple counter design described in SystemVerilog. SystemVerilog is defined by the IEEE 1800-2023 standard [6].

In this work, I will focus mainly on the verification features of the language. More specifically, I will be focusing on the temporal expressions that can be described using SVA properties and sequences.

2.2.2 Linear Temporal Logic (LTL)

Before describing SVA properties and sequences, it is important to understand a bit of the context in which I am working — and why this particular subset of the language is so difficult to implement in practice. SVA properties and sequences allow us to express a form of Linear Temporal Logic (LTL) [67]. While SVA properties can express more than what is possible with LTL, LTL still remains a core aspect of the language. LTL is a type of higher-order logic [69] that allows us to express relations between parts of our design across different clock cycles. Table 2.1 contains a few examples of LTL expressions that I seek to express in the high-level hardware languages used in this work. These mostly express ideas such as concatenation, i.e. two events consistently happening a few cycles apart, non-overlapping implication, i.e. one event implying another at a later cycle, and variable delays, i.e. events that can happen within a cycle range and not necessarily in a specific cycle. Conceptually, these concepts are easy to express individually, however, they

LTL Expression	Description
`G(A & XB)`	B will always hold one cycle after A holds
`G(A -> XB)`	If A holds then B will hold one cycle later
`G(A -> (B || XB || XXB))`	if A holds and B holds 0 to 2 cycles later
`F(A)`	A will eventually hold
`G(A U B)`	A holds until B holds

Table 2.1: Table illustrating a few example LTL expressions and their associated meaning. This is of course incomplete, but it contains most of the elements that we will be focusing on expressing in our high-level hardware languages throughout this thesis. `G(...)` describes a relation that always holds, `X...` is the next operator, which describes an event that happens with a 1 cycle delay, `F(...)` describes an event which will eventually hold i.e. hold at least once at some point in the trace, and `...U...` describes an event that will only happen until another event holds.

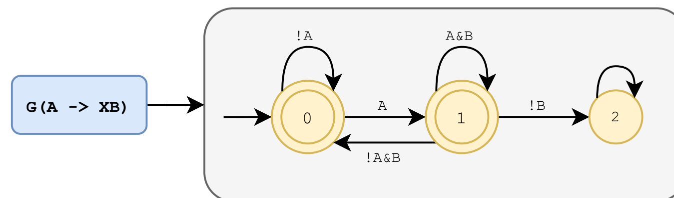


Figure 2.2: Example of a conversion of an LTL expression into a deterministic Büchi automaton. The expression being converted is a simple form of non-overlapping implication (NOI), i.e. "A implies B after one cycle". The resulting automaton has two accepting states (0, 1), i.e. states we need to terminate at in order for our expression to hold, and one dead state (2), i.e. a state which we can not recover from. Note that every arc is only evaluated on a clock tick, and thus moving from one state to the next requires one cycle.

become difficult to convert to synthesizable logic once we start nesting delays and implications. The general idea is to convert an LTL expression into a Büchi automaton [15], which is a type of automata where a state is considered active if it can theoretically be reached an infinite number of times. These automata can then be made deterministic, which enables their implementation using synthesizable hardware. Some tools such as SPOT [29] were designed explicitly for the task of translating LTL expressions into an automaton, however, these are often very powerful and well-engineered tools that have been actively developed for decades. Figure 2.2 shows an example conversion of a common LTL expression pattern, known as non-overlapping implication (NOI) [6] into a deterministic Büchi automaton [31]. This simple case results in a relatively compact automaton, so it can be implemented quite directly. However, more complex expressions which include variable delays or nesting eventually and until statements can quickly lead to state explosion or the need for non-determinism to directly express the property [18].

2.2.3 SystemVerilog Assertions

SystemVerilog implements a set of temporal relations using its SVA sub-language [1]. More specifically, this is implemented using SVA sequences and properties. These allow for the expression of Linear Temporal Logic, as well as other more complex temporal expressions, such as indefinite sequences and variable delays.

Sequences Sequences encode a series of events over a number of cycles. They generally take the form of "A then B after n cycles". The number of cycles between events is called a delay, which can be either an exact number or a range, in which case we call it a variable delay. Sequences must generally be clocked, meaning that they will only be evaluated on the associated clock's ticks. Figure 2.3 shows an example of a sequence that defines a series of

```
sequence s;
    @(posedge clock) a ##[0:2] b ##1 c;
endsequence;
```

Figure 2.3: Example sequence describing the following series of events: "a holds then within 0 and 2 cycles b holds then c holds one cycle later". This sequence is clocked using the standard`@(posedge clock)` event.

events using both exact and variable delays. The operation used to create a sequence, i.e. `##n`, is called the concatenation operator.

Properties Properties encode concurrently checked predicates which can encode relations between elements of a design [6, 1]. These relations can be temporal, i.e. span multiple cycles. The most commonly used property is implication (`a |-> b`). Properties such as implication are defined as an antecedent implying a consequent. The antecedent can be any sequence, but it cannot be another property, while the consequent can be any property. This means that we can't express something like `property |-> property`, but we can express `sequence |-> property`. Properties can also be disabled given a certain condition, meaning that they are only checked if the disable condition doesn't hold. Figure 2.4 shows an example of a property which is defined using a sequence and another property.

```
assert property (@(posedge clock)
    disable iff (reset)
    a ##1 b |=> (c |-> d)
);
```

Figure 2.4: Example of a property assertion that is disabled while the design is being reset. This assertion checks that if b happens one cycle after a, then one cycle later if c happens, d should happen. This can be expressed as `G((a & Xb) -> X(c -> d))` in LTL.

Always vs. Initial Properties SVA properties can be expressed using temporal modifiers [6]. These allow the user to define how and when their property is expected to hold. Two commonly used modifiers are `always`, which is equivalent to the `G(...)` LTL function and expresses that a property should hold in every cycle in order to be valid, and `initial`, which expresses that a property need only hold in the first cycle in which it can terminate to be valid. Figure 2.5 shows an example of a property expressed using the `always`

```
always assert property (@(posedge clock)
    disable iff (reset)
    a |=> b
);
```

Figure 2.5: Example of an `always` property. This will only hold if `a` implies `b` after one cycle for every cycle in our simulation.

modifier. In simulation this need only hold for the duration of the simulation to be correct. In model checking this encodes a safety property [19] for our design. Figure 2.6 shows an example of a property expressed using the

```
initial assert property (@(posedge clock)
    disable iff (reset)
    a |=> b
);
```

Figure 2.6: Example of an `initial` property. This will only hold if `a` implies `b` after one cycle in the first two cycles after our design is reset, i.e. once our property is enabled.

`initial` modifier. This property only needs to hold in the first two valid cycles of our simulation in order for the assertion to hold. By default SVA assumes that we are using always properties [6].

Using a combination of both properties and sequences, one can express complex timing relations in the form of the design's specification. Given the strict timing constraints that generally exist in hardware, these expressions are vital for correctly describing a design's behavioral and structural specification. Unfortunately, the complexity of implementing these temporal semantics in a general form has led to SVA properties and sequences only being supported in a small set of mostly commercial simulators. One of the goals of this work is to expand the accessibility of these vital semantics by lowering them to a form that can be understood by anything that can understand basic Verilog, thus closing the gap in the verification capabilities of both open-source simulators and high-level hardware generators.

2.3 Formal Verification

The verification methods described in section 2.1 are considered to be dynamic, as they rely on simulating the DUT in order to verify it. Another approach one could take would be to statically reason about the design through the use of mathematical modeling and formal methods.

2.3.1 Boolean Satisfiability and SMT

The basis of our formal methodology is the boolean satisfiability problem (SAT) [51]. The problem is defined as follows:

Given a formula $F(x_0, x_1, ..., x_n)$ with $x_i \in 0, 1$, does there exist an assignment to these variables such that F evaluates to 1?

We say that a formula of this sort is satisfiable if we can find such an assignment, otherwise it is said to be unsatisfiable. This problem is known to be NP-Complete [23], however, it can still be solved quickly in many cases using smart heuristics. Tools that solve this problem are called SAT Solvers. We can extend the scope of this problem by including non-boolean theories, e.g. integer or real number theories. These expanded problems are called Satisfiability Modulo Theories (SMT) [13] and are the basis of many formal methods used in practice today. In the particular case of hardware verification, we focus on the bitvector theory, which encodes fixed-width integers. Tools that are capable of solving this problem are called SMT solvers, such as Z3 [25], CVC5 [11], or Boolector [53]. Generally speaking, the type of logic used in these formulas is called First-Order Logic [36]. In brief, this is a form of logic that expresses quantified variables, i.e. variables that can reasoned about using quantifiers such as ”there exists”($\exists$) or ”for all”($\forall$), over non-logical objects, i.e. values within certain theories. This is in contrast to higher-order logics [69], such as temporal logic [31], which require the use of stronger semantics to express their formulae.

2.3.2 Bounded Model Checking

With this tool in hand, we can perform Bounded Model Checking [21], where we encode our designs, along with a specification in the form of assertions and assumptions, as first-order logic which is then given to an SMT (or SAT) solver to check whether or not said specification can be violated.

Verifying Combinatorial Designs Combinatorial logic can be directly mapped to an SMT formula. To do so, we encode our design as the conjunction of all of its constraints, i.e. variable definitions, and the inverse of all its assertions.

We take the inverse of the assertions because we want to detect if any assignments to our inputs can violate the assertions [44]. Figure 2.7 illustrates how a simple design that increments its input can be converted into an SMT formula.

```
class AddOne extends Module {
  val a = IO(Input(UInt(32.W)))
  val b = a + 1.U
  assert(b > a)
}

// Becomes

(and (equal b (add a 1)) // define b
     (not (gt b a))  // Look for a couterexample to this assertion
)
```

Figure 2.7: Example, taken from Kevin Laeufer's guest lecture on formal verification [44], encoding of a basic combinatorial circuit which increments a given input and checks that the incremented version is always bigger than the original input. Given the fixed-width nature of the input, this should fail when $a == 2^{32} - 1$, after which b will be 0 and the assertion will fail. An SMT solver can very easily find this case.

Verifying Sequential Designs Sequential designs require additional work to be converted into an SMT formula, as they represent state can evolve over time. In this case we need some sort of abstraction to model this time-reliant behavior. A common way to do this is to create what is called a state-transition system [21]. The idea is to model the design using states, which represent a set of values for all of the stateful elements in our system, and transitions, which represent how these elements can evolve across a single time-step, which for hardware is a clock cycle. Once this conversion to a state-transition system is complete, we no longer need an explicit clock expression, as clock ticks become implicit in the transition arcs of the system. Figure 2.8 illustrates this abstraction through the conversion of a sequential design into a state-transition system. In practice, obtaining such a state-transition system requires having a bound on the number of cycles we are willing to unroll our design for. The goal of Bounded Model Checking [14] is to find if there exists a set of assignments to our inputs that create a path in the state-transition system which can lead to a bad state, i.e. a state which contradicts an assertion in our design. In practice, this is modeled using symbolic states, which encode the entire system as an initial state I and a transition function $Next$ which represents how a state evolves across a single cycle [44].

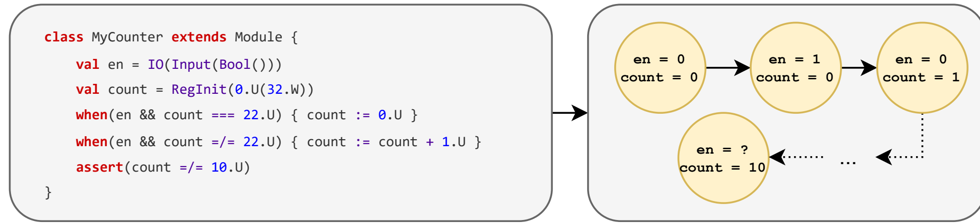


Figure 2.8: Example conversion from a sequential design that implements a counter which is reset when 22 is reached and its associated state-transition system. The goal for a checker here is to find if there exists a set of states which creates a path from our initial state to a state that violates the assertion.

2.3.3 The BTOR2 Format and BTORMC

BTOR2 [53] is a format designed to allow for the quick prototyping of model checkers for use in various Bounded Model Checking competitions. The main use of this format reduce the overhead of creating model checkers for hardware. The format thus supports the explicit definition of state-transition systems, which are needed to encore constructs like memories or registers. A state-transition system represents the model as a group of states containing the possible value combinations that each register can take across multiple cycles. In formats such as SMTLib [12], this involves manually unrolling the circuit across several cycles to simulate the behavior of registers using only combinatorial logic. In `btor2`, however, we can express this simply by declaring a `state` and a `next` arc for each register in our design, thus explicitly modeling the state-transition system.

The format itself is a declarative format that encodes models at the level of SMT logic. Another peculiarity of this format is that it uses ordered unique identifiers, often line numbers, to reference operations. Figure 2.9 shows an example of how this format is used to express a module that takes an input and adds 1 to it. The line numbers on the left side of each line represent the unique identifiers declared for each line. These must be in increasing order but are otherwise arbitrary. These identifiers are then used to reference that operation in a future line.

BTOR2 is supported by many tools, but I mainly use `btormc` [53] in this work, which is a Bounded Model Checker running on the `boolector` SMT solver, which is optimized for solving in the `bitvector` and `array` theories. This format is ideal for encoding hardware models as it allows using solvers specialized for theories that are relevant for hardware model checking, and the format itself allows for a straightforward encoding of concepts such as registers and memories. Table 2.2 shows an overview of the `btor2` format, focusing on the instructions that are supported in the solutions proposed in this work. These leave out any instructions specific to modeling and

Instruction	Description
<lid> sort <type> <width>	Declares a type
<lid> input <sid> <name>	Declares an input
<lid> output <out>	Declares an output
<lid> bad <cond>	Checks the inversion of a condition
<lid> constraint <cond>	Assumes a condition
<lid> zero <sid>	Declares a 0 constant
<lid> one <sid>	Declares a 1 constant
<lid> ones <sid>	Declares a bit-string of 1s
<lid> not <sid> <cond>	Negates a condition
<lid> constd <sid> <val>	Declares a decimal constant
<lid> consth <sid> <val>	Declares a hexadecimal constant
<lid> const <sid> <val>	Declares a binary constant
<lid> state <sid> <name>	Declares a stateful element
<lid> init <sid> <state> <val>	Initializes a state
<lid> next <sid> <state> <next>	Sets the transition logic of a state
<lid> slice <sid> <op> <w> <lb>	Extracts bits [lb:lb+w] from a result
<lid> ite <sid> <cond> <t> <f>	If-then-else expression
<lid> implies <sid> <lhs> <rhs>	Logical implication
<lid> iff <sid> <lhs> <rhs>	If and only if expression
<lid> add/sub/mul <sid> <l> <r>	Binary operation
<lid> {s,u}div <sid> <l> <r>	Signed or unsigned division
<lid> smod <sid> <l> <r>	Signed modulo
<lid> s{l,r}l <sid> <l> <r>	Logical shift left/right
<lid> sra <sid> <l> <r>	Arithmetic shift right
<lid> and/or/xor <sid> <l> <r>	Binary logical operators
<lid> concat <sid> <l> <r>	Concatenate two results
<lid> eq/neq <sid> <l> <r>	Equality comparators
<lid> {s,u}gt <sid> <l> <r>	Signed/Unsigned $l > r$
<lid> {s,u}gte <sid> <l> <r>	Signed/Unsigned $l \geq r$
<lid> {s,u}lt <sid> <l> <r>	Signed/Unsigned $l < r$
<lid> {s,u}lte <sid> <l> <r>	Signed/Unsigned $l \leq r$

Table 2.2: Short overview of the btor2 format [53]. This table contains all of the instructions that are used in my emission pass. <lid> refers to a line identifier, which must be in increasing order but don't have to be consecutive. <sid> refers to a sort's <lid>; these are used to define the resulting type of the instruction and can be either a bitvector, which models fixed-width integers, or an array, which models interacting with memory units. Note that btormc does not support outputs.

```
1 sort bitvector 32 ; declares a 32-bit bitvector type
2 input 1 a         ; declares a 32-bit input named "a"
3 constd 1 1        ; declares a 32-bit constant of value 1
4 sort bitvector 33 ; declares a 33-bit bitvector type
5 add 4 2 1         ; performs the operation a + 1
6 slice 1 5 31 0    ; get rid of overflow bit
7 sort bitvector 1  ; declares a 1-bit bitvector type
8 ugt 7 6 2         ; (a + 1) > a ?
9 bad 8             ; assert (a + 1) > a
```

Figure 2.9: Example `btor2` model that takes an input and adds 1 to it. This is an encoding of the example circuit from Figure 2.7.

interacting with memories, as those are not yet supported in my formal back end.

2.4 CIRCT : Circuit IR Compilers and Tools

This work centers around the CIRCT compiler [48], which is an open-source, MLIR-based [47] project focused on electronic design automation (EDA) tools. More specifically, CIRCT transforms an input source design (in a supported hardware language called a front end) into target languages such as SystemVerilog, Calyx, or even an internal representation for open-source simulation directly in CIRCT. Much like any other MLIR-based tool-set, CIRCT defines a set of dialects, which are domain-specific operations that form their own intermediate representation (IR) that can be used together in a single design. Through its core dialects, CIRCT defines operations that can represent a generalization of hardware. CIRCT also defines front-end dialects that allow for specific elements from each front end to be expressed concisely in their own IR.

Using CIRCT, one can describe hardware in any of its front ends and have that design converted into a generalized representation, which can then benefit from general optimizations before outputting it to a target language like SystemVerilog. Basically, CIRCT attempts to rethink hardware compilers as simply an extension of a software compiler in order to exploit the vast tooling and expertise found in the software compiler domain. MLIR [47] is a well-suited environment to solve this problem, as it allows us to create individual dialects to encode specificities of each front-end language, e.g. Chisel has its own intermediary language called FIRRTL [39], so porting Chisel to CIRCT is done by mapping FIRRTL to a dialect constructed at an identical level of abstraction with the same operations.

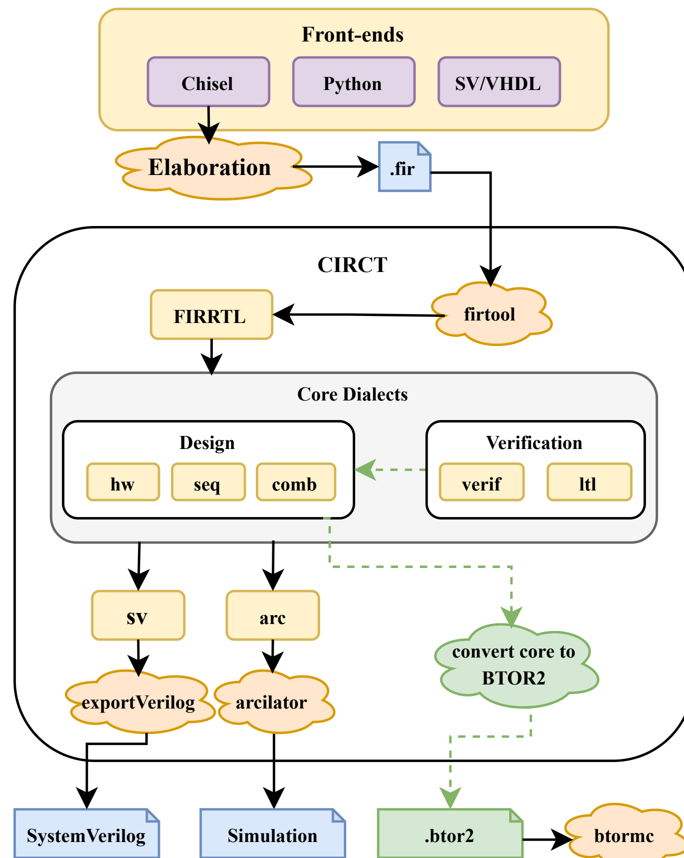


Figure 2.10: Overview of the subset of CIRCT that I am contributing to in this work. The contributions proposed are highlighted in green. I generally focus on the Chisel front end to illustrate my work, so this figure mostly illustrates that flow. Chisel is lowered to a `.fir` file in the FIRRTL IR, before it is handed to `firtool`, which is a tool capable of converting FIRRTL into its MLIR dialect. After that, the `firrtl` dialect is lowered into the core dialects using a mixture of `hw`, `seq`, `comb`, `verif`, and `ltl` dialects. The contributions of this thesis enable the `ltl` and `verif` dialects to be lowered to the core design dialects, thus supporting their broader use in CIRCT's target languages. This can then be exported to many targets including SystemVerilog, arcilator (for simulation), and, with this work, btor2.

2.4.1 Hardware Dialects

CIRCT defines a large set of dialects that are specific to hardware, however, in this work I only focus on a small subset of those dialects, called the core dialects. The core dialects function as a generalized representation of digital hardware, and they can be grouped into design dialects and verification dialects. Figure 2.10 shows an overview of the parts of CIRCT relevant to this thesis.

```
module {
    hw.module @ex(in %a : i32, out inc : i33) {
        %c1_i32 = hw.constant 1 : i32
        %1 = comb.add bin %a, %c1_i32 : i33
        hw.output %1 : i33
    }
}
```

Figure 2.11: Example design that takes an input and increments it by 1 before outputting it. Here, a 32-bit constant of value 1 is created, which is then added to the input a. This yields a 33-bit result, as addition always requires 1 extra bit to avoid overflows. The result of this addition is then set as our output using `hw.output`

Core Design Dialects The main part of the core dialects is the design dialects. These are used to express a digital hardware design in an RTL form. This category loosely contains the following dialects:

- **hw**: This is generally used to define generic hardware elements outside of any sort of logic. This includes defining a hardware module and its interface using the `hw.module` and `hw.output` operations and creating connections within the module itself using the `hw.wire` and `hw.constant` operations. This dialect also defines important types and methods that allow us to gain information about various hardware elements in the compiler, most notably via the `hw::getBitWidth` function.
- **seq**: This is used to defined sequential design elements, such as clocks using the `seq.to_clock` operation, or to declare registers using the `seq.firreg` and `seq.compreg` operations.
- **comb**: This is used to define combinatorial logic elements, such as binary operations like `comb.add` and `comb.and`, comparator operations such as `comb.icmp`, or binary manipulation operations such as `comb.concat` and `comb.replicate`.

Using these dialects, one can model basically any digital hardware design. Figure 2.11 shows an example of a simple incrementing design that is expressed using only elements from the core design dialects. Note that MLIR is generally in Single-Static-Assignment (SSA) form [47], and thus all lines need to be stored as new results in SSA result values.

Core Verification Dialects CIRCT also recently added core dialects to express constructs that are specifically for verification purposes. For this, there two main dialects:

- **verif:** This dialect allows us to express constructs used to interact with a design for the sake of verifying it. This includes defining a specification for the design using `verif.assert`, `verif.assume`, or `verif.cover` operations, as well as to express higher level constructs used to perform other verification tasks such as the `verif.has_been_reset` operation, which models a register that keeps track of whether or not the DUT has been initialized yet.
- **ltl:** This dialect is specifically used to model SystemVerilog Assertion (SVA)-like temporal expressions [6], which are extremely useful for hardware verification, as they allow us to express relations across multiple clock cycles. The capabilities of this dialect aim to encode a small subset of SVA properties, using operations such as `ltl.implication`, `ltl.concat`, or `ltl.delay`. It also allows for additional temporal constraints to be added to assertions, e.g. using the `ltl.clock` or `ltl.disable` operations which associate an assertion to a clock and disable it under certain conditions, respectively.

These two dialects are at the core of the work in this thesis, as they strongly rely on the use of the poorly supported SVA property and sequence sublanguage of SystemVerilog to output their higher-order logic. My goal in this work is to find an encoding for this higher-order logic as something that can be used for Bounded Model Checking.

Chapter 3

Formal Back End for CIRCT

One of the main methods used in Hardware Verification to maximize the reliability of a design is to use formal methods to check the design against a specification. In practice, this is done using Bounded Model Checking (BMC) [14]. However, this is currently under-supported outside of a select number of commercial Electronic Design Automation (EDA) tools; BMC is even more poorly supported for high-level languages. Generally, using this method requires a translation of a design into a format that can be understood by SMT-based BMC tools such as `btormc` [53]. My goal in this work is to enable BMC directly in open-source through the CIRCT compiler [48], which benefits a large amount of front ends simultaneously instead of merely a single language. This is done by introducing a formal back-end into CIRCT, i.e. a new target format that can be used specifically for BMC. The specific target that I chose to target is `btor2` as it is particularly well-suited for expressing state-transition systems. I illustrate the complete workflow using Chisel as an input language, however, the formal back end I developed works with any of CIRCT's front-end languages. This chapter discusses how model checking is enabled through my btor2 back end, which is now part of CIRCT.

3.1 Formal Compilation Flow

We begin by breaking down the general steps needed to go from a Chisel design to a Bounded Model Checking result.

3.1.1 Simple Example

We start with a simple timer circuit written in Chisel, shown in Figure 3.1. This example is of a 32-bit counter that is reset when it reaches 22. This design also contains an assertion which describes a property we expect our design to have, in this case that it never reaches the value 10. With the conversion pass I added to CIRCT, this property will be able to be checked

automatically using BMC [44]. Figure 3.2 shows an overview of how a

```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 22.U) { count := 0.U }
    when(count =/= 22.U) { count := count + 1.U }
    assert(count =/= 10.U)
}
```

Figure 3.1: This design models a 32-bit counter that has a reset value of 0, and can be incremented up until 22 before being reset again. The assertion in this design can be checked automatically using the BMC, thanks to the emission pass I added to CIRCT.

Chisel design maps to `btor2`. In this illustration, the register maps to a state declaration and an initialization. Note that in reality `RegInit` in Chisel sets a reset value and not an initial value for verification, i.e. a *power on* value, this was simplified for illustrative purposes. The remainder of this chapter will detail how I constructed this lowering in the CIRCT compiler and the various steps required to reach the final `btor2` description.

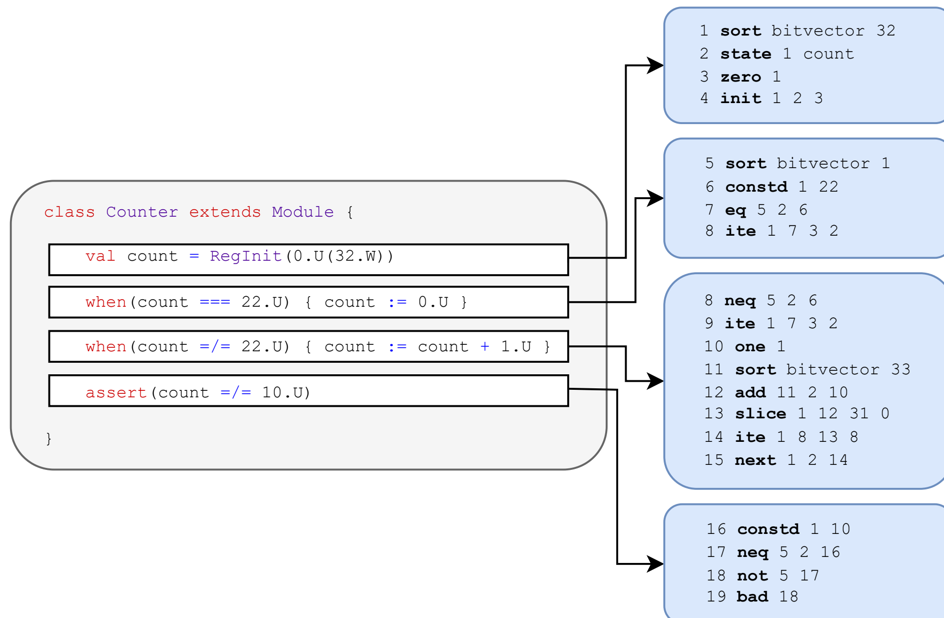


Figure 3.2: Overview of my compilation goal. This illustrates how the various parts of a Chisel design are mapped to `btor2`. Note that the conversion in this figure is a simplification for illustrative purposes and the final conversion contains some additional details.

3.1.2 Chisel Elaboration

The first step in our compilation flow is extracting the CHIRRTL representation [39], a high level form of FIRRTL, from our design. This step is called Chisel elaboration, as Chisel is in reality a library in Scala that is used to generate CHIRRTL using a special emission function called `emitChirrtl`. This can then be mapped to the `firrtl` dialect in the CIRCT compiler [48]. CHIRRTL represents the highest-level form of FIRRTL, one which closely resembles the input Chisel source. This representation can then be converted into MLIR using `firtool` and the `FIRRTL` dialect. The goal of this step is to create an MLIR-based entry point into the CIRCT compiler itself, and to do some Chisel-specific lowerings such as generate implicit clock and reset signals.

3.1.3 Compiling FIRRTL to the Core Dialects

The `firrtl` dialect is then lowered into CIRCT's core dialects. These are used as a generalized representation of hardware, and are a conversion point for many of CIRCT's front ends where general optimizations can take place. Figure 3.3 shows the core dialect representation generated for our example. The MLIR code itself is a lowering of all of Chisel's high-level constructs to a more bare-bones representation of hardware.

3.1.4 Compiling the Core Dialects to BTOR2

Once the core dialect representation has been generated, we can move on to my contribution to this flow — the btor2 emission. This is explored in greater detail in Section 3.2, but the general idea is to convert our circuit into a state-transition system and encode it in the `btor2` format. The core dialects express designs using semantics that closely match the expression semantics of the bitvector logic in SMTLib [12]. The main work is thus in encoding registers as a state-transition system, which can be done by emitting `state` and `next` instructions for each register.

3.1.5 Checking the BTOR2 Model

The ultimate goal of this conversion is to perform Bounded Model Checking on our design, as presented in Section 2.3. This is done using `btormc` [53], which can take our newly obtained `btor2` description as input along with a cycle-bound used to perform Bounded Model Checking. The result will tell us whether our model is satisfiable, i.e. our assertion can be violated, in which case it will give us a counter-example, or unsatisfiable, i.e. our design meets the given specification. Figure 3.4 shows the result of running our example model through `btormc`. The result we obtain is `SAT`, meaning satisfiable. A satisfiable result is accompanied by a counterexample, which

```
module {
    hw.module @Counter(in %clock : !seq.clock, in %reset : i1) {
        %c1_i32 = hw.constant 1 : i32
        %c10_i32 = hw.constant 10 : i32
        %c22_i32 = hw.constant 22 : i32
        %true = hw.constant true
        %c0_i32 = hw.constant 0 : i32
        %0 = seq.from_clock %clock
        %count = seq.firreg %3 clock %clock
            reset sync %reset, %c0_i32 : i32
        %1 = comb.icmp bin eq %count, %c22_i32 : i32
        %2 = comb.add %count, %c1_i32 : i32
        %3 = comb.mux bin %1, %c0_i32, %2 : i32
        %4 = comb.xor bin %reset, %true : i1
        %5 = comb.icmp bin ne %count, %c10_i32 : i32
        %6 = comb.and bin %4, %5 : i1
        sv.always posedge %0 {
            sv.assert %6
        }
    }
}
```

Figure 3.3: Result from compiling the example design from Figure 3.1 into an MLIR representation using the core dialects.

can be used to prove how the assertion can be violated. This counterexample is in the form of a trace and shows us how our counter can reach the illegal value of 10.

```
sat
b0
#0
0 00000000000000000000000000000000 count@0
0 1 reset@0
[...]
0 00000000000000000000000000001010 count@10
0 0 reset@10
.
```

Figure 3.4: Abbreviated result from running our `btor2` model through `btormc`. This result shows us a counterexample where the value of each signal is given for each cycle and the assertion is violated in cycle 10 when the counter is equal to 10.

3.2 Compiling Hardware to First-Order Logic

In this section, we discuss the details of my `btor2` back end and how each individual conversion is performed.

3.2.1 Combinatorial Circuits

The first step to having a complete encoding of hardware in `btor2` is to start with encoding purely combinatorial circuits. The goal in Bounded Model Checking is to find out whether or not the given assertions hold in every possible case. To do so we want to find a set of assignments to variables in a given formula that allows the inverse of the assertion to hold. If this assignments exists, then we say that the formula is satisfiable and return the assignments as a counterexample, otherwise we say that the formula is unsatisfiable. The main idea is to encode a given circuit containing an assertion as the conjunction between all of the constraints, i.e. the definition of wires and components, and the inverse of the assertion as a single SMT formula that can then be handed off to an SMT-solver such as `boolector` [53].

MLIR to BTOR2 Conversions Let us now detail how each type of combinatorial MLIR operation is converted to `btor2`. For combinatorial circuits, the dialects that need to be considered are the `hw`, `comb`, `seq`, and `sv` dialects.

- **Ports:** Inputs and outputs to a module are not explicitly defined using MLIR operations but are rather defined as arguments to the `hw::ModuleOp` operation. These are the first constructs that we need to convert. We thus extract the ports from the module, then use the information stored in the obtained `hw::PortInfo` to retrieve their bit-width, name, and port direction. When it comes to model checking, outputs are omitted as they are unnecessary. The final emission for ports would yield the conversion in Figure 3.5. As explained in Section 2.3, clocks

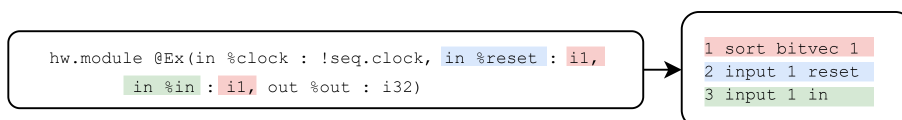


Figure 3.5: Illustration of how modules are lowered to `btor2`.

  are ignored as our system is expressed in terms of states and transitions, and not in terms of when those transitions occur.

- **Constants:** The constant's value and bit-width are extracted from the `hw::ConstantOp`, and used to generate the equivalent constant declaration in `btor2`. A new sort is generated only if a bitvector of that width does has not yet been declared. This yields the conversion in

Figure 3.6. Note that while btor2 supports several constant declaration

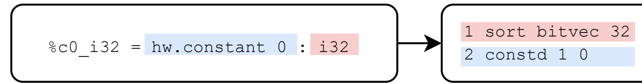


Figure 3.6: Illustration of how constants are lowered to btor2.

operations, we will exclusively use the constd operation as it supports decimal representations and thus avoids unnecessary string formatting.

- **Wires:** Wires represent an intermediate connection in our design and are often used to improve readability. These are thus treated as aliases in our pass and don't explicitly generate any btor2. The only subtlety with these is that they may be nested and require iterative de-aliasing until a non-wire operation is reached. Every time a hw::WireOp, which defines a wire in our design, is reached in our pass, we simply store the operation to operation mapping and use it to find the first operation mapped to a concrete line identifier when being used as an operand in another operation. In practice this means that when we emit btor2 for an operation that uses a he::WireOp as an operand, we need to call getOpAlias(wire) and fetch the identifier of the result and not the identifier of the wire itself.
- **Binary Combinatorial Operations:** These all behave the same way and have one-to-one mappings between the comb operations and btor2 operations. We simply need to extract the width of the result and the name of the operation to generate a valid btor2 operation. Figure 3.7 shows an example of such a lowering.

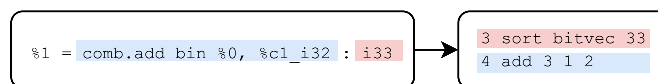


Figure 3.7: Illustration of how an addition is lowered to btor2. Note that all binary combinatorial operations are lowered following the same pattern. Here 1 and 2, in line 4 of the btor2 result, refer to the conversions of %0 and %c1_i32 in the input MLIR.

- **Bit Extraction:** The comb::ExtractOp is equivalent to the btor2 slice operations and models the extraction of a subset of a bitvector's bits. The only difference is that extract only takes a low-bit as argument, so the output slice operation always goes from the low-bit to the bit-width minus 1. Figure 3.8 illustrates this conversion.
- **Comparisons:** These are a one-to-one mapping as the naming of comb::ICmpOp comparators is identical to the ones used in btor2, apart from the exception of non-equality which is written as ne is MLIR but

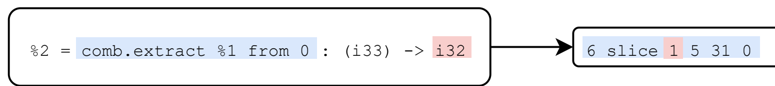


Figure 3.8: Illustration of how extractions are lowered to `btor2`. Note that 5 in the `btor2` result refers to the conversion of %1 from the MLIR input.

as `neq` in `btor2`. This was added as a special case. Figure 3.9 illustrates this lowering.

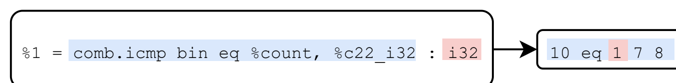


Figure 3.9: Illustration of how comparisons, e.g. equality, are lowered to `btor2`. Note that 7 and 8 in the `btor2` result refer to the conversions of `%count` and %c22_i32 from the MLIR input respectively.

- **Multiplexers:** These map to `ite` operations in `btor2`. This instruction refers to an "if then else" statement, which has similar semantics as those used by the `comb::MuxOp`. We thus only need to extract the condition, true and false results, and reorder them in the `ite` operation. Figure 3.10 illustrates this lowering.

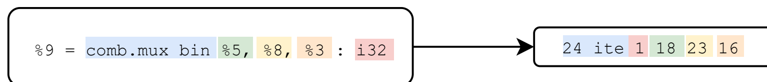


Figure 3.10: Illustration of how multiplexers are lowered to `btor2`. Note that 18, 23, and 16 in the `btor2` result refer to the conversions of %5, %8 and %3 from the MLIR input respectively.

- **Assertions and Assumptions:** These are not explicitly defined in the core design dialects, so basic assertions and assumptions lower to their equivalents expressed in the `sv` dialect. As described in Section 2.3, the goal in BMC is to find a counterexample that violates our assertion (if any). In order to do so, we need to convert our assertion into a `bad` `btor2` instruction which checks for the inverse of our original assertion condition. Assertions are often predicated by an enable signal using an `sv::IfOp`. In that case, the assertion condition is first expressed as the consequent of an implication on the enable condition, after which it is inverted to be used in the `bad` instruction. Assumptions are simply mapped one-to-one to `btor2` `constraint` operations. This conversion is illustrated in Figure 3.11.

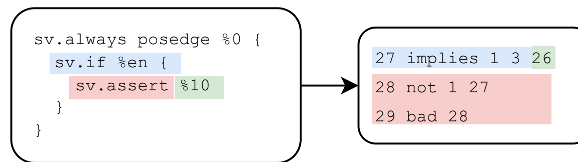


Figure 3.11: Illustration of how assertions are lowered to `btor2`. The `sv.always` operation in MLIR is ignored, as we are working with state-transition systems which encode the clock implicitly.

3.2.2 Sequential Circuits

The main difference between a combinatorial circuit and a sequential one comes from the presence of registers. This adds the necessity to generate states and transition arcs for each register in the design. The difficulty with this is that registers often modify their own value depending on the one they held during the previous cycle. This means that our implementation needs to support two additional emission details:

- Handle cyclical operands, meaning operations such as registers defined using an operation on themselves in their `next` expression e.g. `count := count + 1`. Implemented naively, this would yield null identifiers as the register would not yet be defined when defining its `next` expression.
- Handle back-edges in the IR to allow for a def-after-use pattern.

In order to support cyclical operands, i.e. an operation that uses itself as an operand, we declare all registers at the beginning of the `btor2` model. This is done by pre-walking the Intermediate Representation (IR) to visit all registers before any other operation. Once all registers have been emitted, we can do the second pass of the IR to emit everything else. To support def-after-use patterns, which are often present when working with registers in the core dialects, we need to perform our second walk of the IR in a Depth-First Search (DFS) order, meaning that we need to first emit all of an operation's operands before emitting the the operation itself. This ordering also allows us to only emit operations that are actually used, as operations that are not another's operand will never be visited and thus won't cause any unused `btor2` operations to be emitted.

Register Conversion In practice, registers will be converted in two phases. First we will generate a `state` instruction that is used to declare the register and give it an identifier. This is done in the pre-walk of the IR. Afterwards, the register operation is stored in a map that associates it to its identifier, so that at the end of the DFS walk, we can emit the `next` instructions, which define how the register's value evolves from one cycle to the next. Figure 3.12 shows an example of what a register conversion could look like. One final subtlety that needs to be dealt with is the handling of reset values, which are

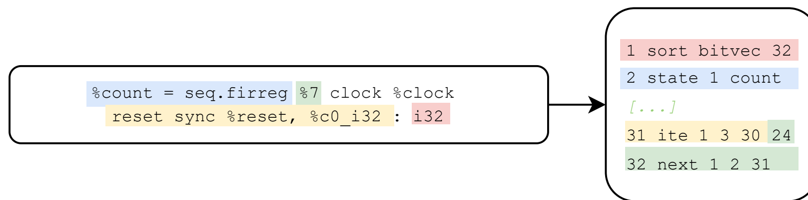


Figure 3.12: Illustration of how registers are lowered to `btor2`. The state declaration contains a sort and a name. The reset and next instructions are then merged into a single next value that implements `next = reset ? resetVal : nextVal`.

the value given to a register when the design's reset signal is raised. Given that most registers are tied to a reset, we need to record a specific identifier for the reset (assuming only one exists) to then use as a condition on the value taken in the register's transition arc. This is done when the module's arguments are visited in the beginning of the DFS pass.

Initial Values There are two types of registers in CIRCT, `firreg`, which is generated by a FIRRTL lowering, and `compreg`, which is the more generic register for CIRCT used by most front ends. The main difference between these two is in how they support initial values, `firreg` uses a type of annotation to encode them, while `compreg` can accept a `powerOn` operand which can directly contain an initial value for the register. This is very important for formal verification, as without an initial values, an SMT solver can simply arbitrarily set the register's value at cycle 0, trivially finding false counterexamples for any design. To solve this we generate initial values for both register types using either the `powerOn` values or the `resetValue` if no initial value was given. We then use these values in an `init` statement in the generated `btor2`. This conversion is illustrated in Figure 3.13.

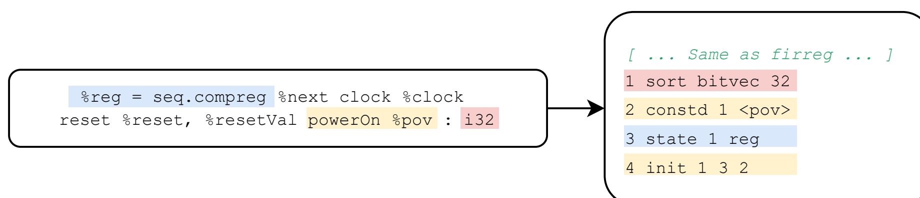


Figure 3.13: Illustration of how registers initialization are lowered to `btor2`. Registers can be defined using `seq.firreg`, which is specific to FIRRTL, or `seq.compreg`. Both register types lower the same way to `btor2`.

Handling Asynchronous Resets The above method is specific to designs that only use synchronous resets. In the case of asynchronous resets, the register can be reset at any moment. The problem with this is that in `btor2`,

a register's value is only "updated" on a state transition, so it can't be reset arbitrarily. One solution is to generate `ite` statements that are conditioned on the reset in order to select either the reset value or the current register value on every read. This way we don't actually need to worry about the register being reset synchronously. For example, we can have a register `%reg` declared with an asynchronous reset signal. When we then use `%reg`, it will be read as `reset ? %resetVal : %reg`. This conversion is illustrated in Figure 3.14.

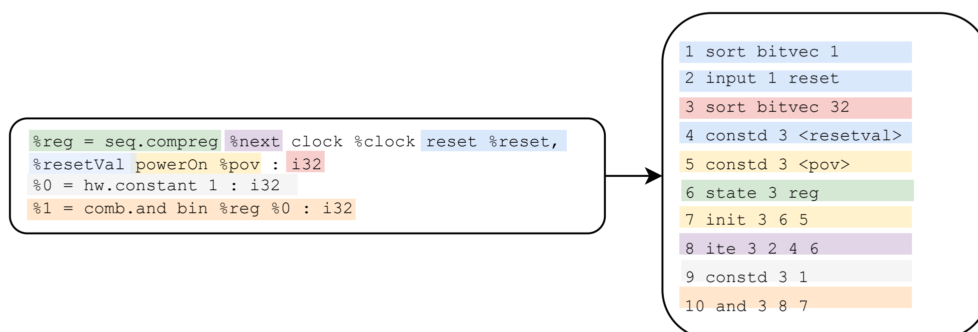


Figure 3.14: Illustration of how asynchronous registers are handled in the `btor2` emission pass. When asynchronous resets are used, the generated `next` instruction, which is conditioned on the reset, is used as the register's value instead of the register itself.

Clocks Clocks are generally ignored in my model, as we assume a single clock design. This is often enough as many designs are globally synchronized. There are also methods called clock-gating which allows a single clocked design to have multiple clock behaviors by performing some logical operation on the clock signal before connecting it to registers [57]. Explicitly handling multiple clocks would have to be done using additional inputs and conditioning transition arcs on the clock that is associated to the current register, but this is generally ignored in my solution as it makes handling temporal specifications more complex, and isn't necessary for the large majority of designs.

This addition now allows us to model arbitrary sequential circuits using our formal back end. We can now represent registers to enable the modeling of designs that evolve over time, unlike combinatorial designs which represent a pure logical transformation of inputs into outputs.

3.3 Results

The output from this work is a `btor2` emission pass that was up-streamed into CIRCT [26] and is integrated into the `firtool` compiler for Chisel. This pass

```
1 sort bitvec 1
2 input 1 reset
4 sort bitvec 28
5 constd 4 0
6 constd 1 0
7 sort bitvec 32
8 constd 7 22
9 constd 1 -1
10 sort bitvec 4
11 constd 10 -6
12 constd 7 0
13 state 7 count    ; count = Reg<32>
14 eq 1 13 8
16 ite 7 14 12 13   ; count == 22 ? 0 : count
17 neq 1 13 8
19 sort bitvec 33
20 concat 19 6 13
21 constd 19 1
22 add 19 20 21     ; count + 1
23 slice 7 22 31 0  ; model behavior of chisel add (can overflow)
24 ite 7 17 23 16   ; count != 22 ? (count+1)[32..0] : `16`
25 constd 7 10
26 neq 1 13 25
28 not 1 27
29 bad 28           ; solve(!(count != 10))
30 zero 7
31 ite 7 2 30 24
32 next 7 13 31     ; count := reset ? resetval : `24`
```

Figure 3.15: Result from running the `convert-hw-to-btor2` pass on the Chisel design expressed in Figure 3.1. The notation 'x' refers to the instruction defined with the identifier x, e.g. '24' refers to `24 ite 7 17 23 16`.

can be run using `circt-opt --convert-hw-to-btor2 <mlir-file>.mlir` and can convert any design expressed in CIRCT's core dialects into a `btor2` model. This model can then be given as an input to `btormc` to perform Bounded Model Checking.

This pass enables a simple version of specification-based verification using non-temporal assertions and assumptions. We can then easily formally verify if certain purely combinatorial assertions hold for any given circuit. For example, using this pass, we can now convert the design from Figure 3.1 into a `btor2` file yielding the result in Figure 3.15.

This automated modeling was not possible in CIRCT prior to this work. While

existing solutions were present in the Scala FIRRTL Compiler (SFC) [43] to perform a similar conversion, this was not adapted to the newer versions of Chisel, which rely on the far faster CIRCT compiler. Additionally, our solution allows for the formal modeling of a design described in any of CIRCT's front ends, not just Chisel, which was the case with the SFC-based solution.

Limitations This pass focuses on creating a model of the hardware design, and it does not support any complex specifications that do not rely on synthesizable logic. More specifically, this solution does not support memories, which rely on `array` sorts in `btor2` and a special set of instructions to interact with those sorts [53]. As mentioned earlier, this also does not support multi-clock designs, as that would add too much complexity, particularly when trying to support temporal logic, and is rarely used in practice.

This formal back-end functions as the backbone for the rest of this work. In the next chapter, I present how the input MLIR can be modified to allow for the expression of temporal logic and other properties directly in our design's specification in a way that can be used for BMC.

Chapter 4

Encoding Temporal Properties for Bounded Model Checking

The previous chapter presents a formal back end that can be used to verify digital hardware designs against a specification written out as a set of assertions. In this chapter, I focus on improving how we can express these specifications. In particular, I introduce two custom lowerings for SVA properties and sequences that enable the use of temporal expressions in an assertion.

4.1 Encoding Property Assertions

As I presented in Section 2.2.3, property assertions are at the core of encoding temporal expression in a specification. As a result, I start this chapter by creating an encoding for the elements required to express property assertions. In order for the encoding to work for Bounded Model Checking, I need a way to guarantee that the SMT solver won't simply select illegal values for all of the registers in the design at cycle 0. To do so, I need to disable the assertions in the design as long as the circuit has not been reset. This essentially allows me to set an undefined state at the beginning of the model checking process.

4.1.1 The `AssertProperty` Statement

In our source language, Chisel, there exists a statement which directly maps to SystemVerilog's `assert property` statement, namely `AssertProperty` [42]. Using this statement in Chisel will cause the CIRCT compiler to generate the MLIR code in Figure 4.1.

The generated MLIR has four operations that are not part of the traditional synthesizable core dialect operations and thus cannot be directly used with

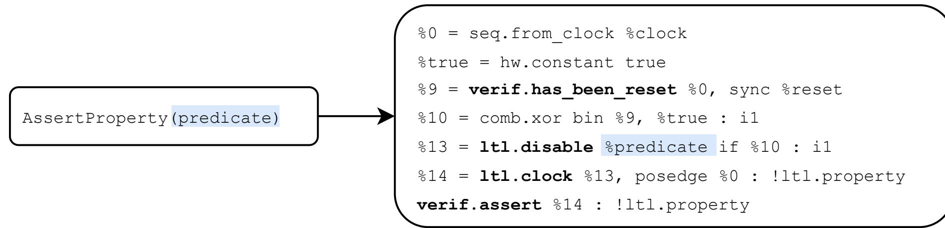


Figure 4.1: Resulting MLIR generated from a single `AssertProperty(predicate)` statement in Chisel. `verif.has_been_reset` keeps track of the reset being raised, `ltl.disable` disables the given predicate if the reset has not yet been raised, and `ltl.clock` associates a clock to our assertion.

our Bounded Model Checking back end, so we need to find a synthesizable encoding for these concepts.

Has_been_reset This operation represents a 1-bit register which is set to 1 the first time the reset is active, and then remains at 1 forever. One small detail that can be found in the `verif` dialect's documentation [59] is that the register must be set to 1 one cycle after the reset is active for the first time. I implement this using a register (`hbr`) whose value is set to `reset | hbr` and is used via `(not (and (not reset) hbr)` in the disable condition. This allows us to keep track of the reset and make sure that we aren't considering the assertion in the same cycle that the circuit is being reset in. Figure 4.2 illustrates this conversion. An additional subtlety about this operation is that

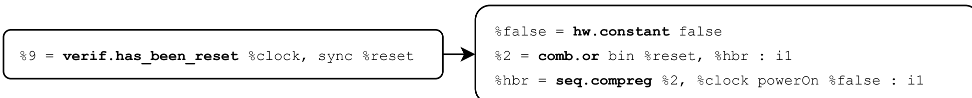


Figure 4.2: Illustration of the lowering of a `has_been_reset` operation to core dialect operations.

it should only be read as active once the reset cycle has ended. The "has been reset" signal is therefore a combination of the `hbr` register and the current value of the reset signal, i.e. `(hbr & !reset)`. This guarantees that we never consider the cycle where the reset process is still ongoing as valid.

Disable This operation disables the input predicate given a disable condition. By default, this is used to disable assertions before the first reset. Logically disabling an input is equivalent to masking its result with a 1 when the disable condition is valid. This is implemented by converting the disable operation into an implication with a negated condition, i.e. `(implies (not cond) input)`, which is equivalent to `(or cond input)`. Figure 4.3 shows how this conversion is performed.

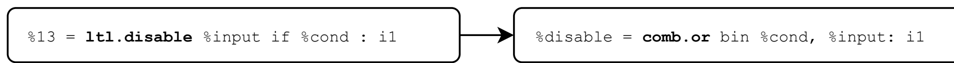


Figure 4.3: Illustration of the lowering of a `disable` operation to core dialect operations.

Clock and Assert These two operations are handled together, as the clock is needed to lower the assertion. Generally, CIRCT uses assertions from the `sv` dialect to encode simple assertions from its front ends. Assertions from the `verif` dialect are currently only used to encode property assertions [59], so they need to be lowered into the `sv` dialect in my pass for these to encode the behavior of a standard assertion. In practice, this conversion is the most complex one to implement in CIRCT, as performing a backwards conversion while keeping strict type safety requirements in check requires engineering finesse — the details are available in the source code [27]. Figure 4.4 illustrates this conversion.

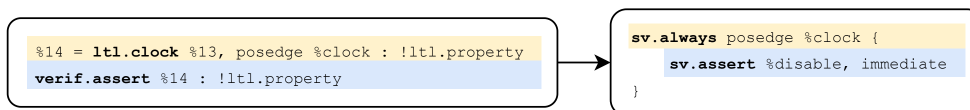


Figure 4.4: Illustration of how a `verif.assert` operation, along with its associated clock, are lowered to core dialect operations.

With this conversion, I can now encode assertions that can be disabled using either an arbitrary condition or the ”has been reset” signal as synthesizable hardware, meaning that it can be used with my Bounded Model Checking backend.

4.2 Implementing SVA Properties in CIRCT

I described how temporal expressions are traditionally encoded as first-order logic through the use of Büchi automata in Section 2.2.2. However, this solution is not easily implementable in a compiler, as it requires numerous complex automata transformations, which require far more work to implement than is possible in the scope of this thesis. Instead, we propose a practical implementation of certain non-nested properties using a direct encoding of the operation’s semantics as synthesizable hardware.

4.2.1 A Study of SVA Property Usage

To figure out what properties need to be focused on in my specialized implementation, I looked at data gathered from a previous survey of SVA property and sequence usage in open-source projects done at UC Berkeley.

Additionally, I received additional SVA property and sequence usage data through discussions with verification engineers at various companies.

Previous Work on Surveying Existing SV Designs In the scope of the original Chisel formal project [45], a group of students looked at 11 different designs across two open-source projects, namely the "axi4 vip" project [66] and the OpenHW group's CV32E40S RISC-V IP [34], and gathered all of the SVA properties that were used to verify them. Figure 4.5 shows the results

Design	#OI	#NOI	#Concat
AXI	32	0	13
Alignment Buffer	10	1	2
FSM Controller	1	0	0
CPU Core	1	7	0
Decoder	1	0	0
If Stage	1	0	0
Load Store Unit	4	0	0
Prefetch Unit	2	0	0
Prefetcher	5	0	0
Sleep Unit	9	0	0
Write Buffer	8	2	0
Total	74	10	15

Figure 4.5: Results from a survey conducted across two projects: a verified implementation of an AXI bus, and a verified implementation of a 32-bit Risc-V core. The results show that the most commonly used properties across these two large projects were overlapping implication (OI), non-overlapping implication (NOI), and delayed concatenation (concat).

from the survey. These results count the number of properties found in two large projects and separate them by type. The figure shows the number of overlapping implications, non-overlapping implications, and concatenations used in the properties found across these two projects. These results show that the most important properties to focus on are overlapping implication and non-overlapping implication. Concatenation seems useful but mostly focused around a single design, so the two first property types should be prioritized.

SVA Property Usage at Intel After presenting this work at the SLICE Lab retreat to a set of industry sponsors, researchers from Intel contacted us to learn more about our research. Through this discussion, I was able to get information about the properties they use in their designs. None of the temporal properties they mentioned are currently supported by the popular open-source Verilog simulator Verilator [60], something which could

be solved by my lowering. Figure 4.6 shows us the most common types of

Property	Description	Verilator/My Support
`a |-> b`	Simple implication	Yes/Yes
`a ##n b`	Constant delay concatenation	No/Yes
`a ##2 b ##[1:5] c`	Variable delay concatenation	No/No
`(disable iff (d)) ...`	Custom disabling of properties	No/Yes

Figure 4.6: Usage information gathered from engineers at Intel. This shows us the most important properties used in their Chisel development team. This table also illustrates how little support these properties receive in the open-source simulator Verilator.

properties used by researchers writing Chisel at Intel Labs. This team focuses on using open-source tools for their development, so they cannot use many of the commercial properties that could be useful to them. This information shows us that implications as well as delayed concatenations are important properties to focus on lowering to a form that any tool that understands basic Verilog could understand. This would allow both open-source simulators and our formal back-end to benefit from SVA properties.

Discussions with Other Groups Discussions with engineers at both SiFive and the FZI research group in Karlsruhe lead to a very similar conclusion. The most commonly used SVA properties at both of these companies were non-overlapping implications and constant delay concatenations. This information, along with the results shown in the two previous paragraphs, shows that we should focus our implementation on those two constructs, i.e. non-overlapping implication and concatenation with a constant delay. Note that overlapping implication is supported using a simply a logical implication, so it is trivial to support.

4.2.2 Implementing Non-Overlapping Implication

Our previous study concluded that the most beneficial lowering to implement is non-overlapping implication (NOI). As presented earlier, NOI, in contrast to overlapping implication, encodes a logical implication over multiple cycles [1]. Figure 4.7 show how this is represented in SVA.

```
input clock, a, b;
integer n;

assert property (@(posedge clock) a ##n true |-> b) // SVA
G(a XX..[n]X1 -> b) // LTL
```

Figure 4.7: Example of a non-overlapping implication property and its equivalent LTL formula.

For a single cycle delay, this can be written in SystemVerilog, as `a |=> b`. In any case `a` is called the *antecedent* of the implication, and `b` would be the *consequent*. In order to encode this directly, I designed a lowering that allows monitoring the property using sythesizable logic, rather than an automaton. Using this lowering, I can encode the property in a form that can be expressed in the core design dialects and is supported by all of CIRCT's targets, including my formal back end. To perform this lowering, I create two sets of registers:

- A register that counts the number of cycles elapsed since the last time the property was enabled.
- A pipeline of registers that delay the antecedent for the number of cycles required by delay, i.e. `n` in the above example.

Once we have these two sets of registers in place, we can simply modify our final assertion to check that we are in one of three cases, either our delay register is below `n`, our final register in the antecedent delay pipeline implies our consequent, or that we are currently in a reset cycle. Figure 4.8 illustrates how this lowering is done in practice. A diagram of the circuit generated by this lowering can be found in Figure 4.9. [ht]

```
a ##n true |-> b

// becomes

reg delay, a_0, ..., a_n;

delay' = disable ? 0 : delay + 1

a_0' = a;
a_1' = a_0;
// ...
a_n' = a_(n-1)

assert (delay < n) || (a_n -> b) || disable
```

Figure 4.8: Pseudo-code illustrating how my lowering implements non-overlapping implication with a delay of `n` cycles.

The three cases in the assertion are justified as follows:

- `delay < n`: if the delay has not yet been reached then the value of the last register in the pipeline is "garbage", i.e. undefined, as it theoretically stores information from before the property was last enabled. We need this condition to make sure that the assertion is disabled as long

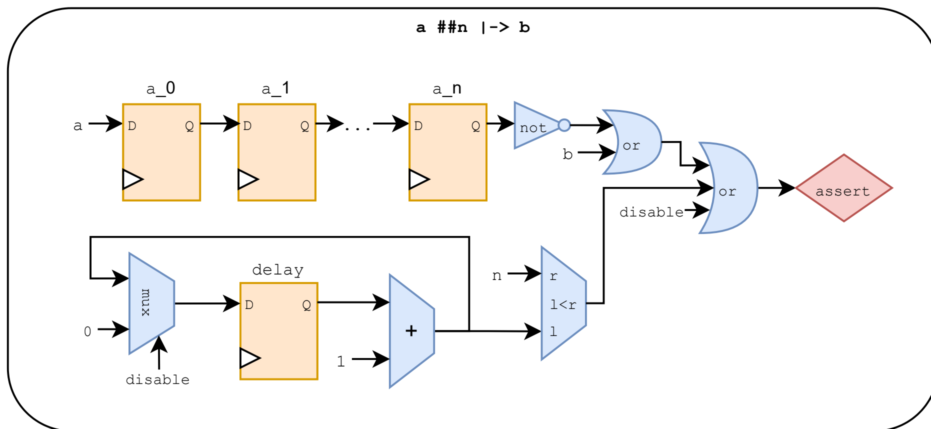


Figure 4.9: Diagram illustrating the circuit generated by lowering non-overlapping implication with a delay of `n` cycles using my solution. The labeled squares represent registers, each of which is clocked using the clock associated to the property using the `ltl.clock`. The antecedent registers do not need to be reset, as they are delayed versions of the antecedent, which is expected to be reset externally. The delay register is reset using the disable condition associated to the property.

as we haven't waited long enough to evaluate it.

- `a_n -> b`: this simply encodes our core condition, i.e. that `a` implies `b` `n` cycles later, so `a` delayed by `n` cycles should imply `b`.
- `disable`: if the disable condition holds, then the assertion should be disabled.

Concrete NOI Lowering Now that the main idea of the implementation was presented, I describe the actual lowering done in MLIR. When an NOI is written in Chisel, it generates a set of particular intrinsics in FIRRTL which then get lowered to three operations in MLIR [42]. This is a result of NOI being encoded as `antecedent ##n 'true' |-> consequent`.

```
a |=> b

// becomes

%true = hw.constant true
%1 = ltl.delay %true, 1, 0 : i1
%2 = ltl.concat %a, %1 : i1, !ltl.sequence
%3 = ltl.implication %2, %b : !ltl.sequence, i1
```

Figure 4.10: Encoding of a non-overlapping implication expression in MLIR. `a |=> b` is encoded as `a ##1 true |-> b`.

<table>
<tr><th>Name</th><th>Pattern</th><th>Inputs</th></tr>
<tr><td>Non-overlapping implication (NOI)</td><td><code>a ##n true |-> b</code></td><td>$a, b \in \{0, 1\}$</td></tr>
<tr><td>Overlapping implication (OI)</td><td><code>a |-> b</code></td><td>$a, b \in \{0, 1\}$</td></tr>
<tr><td>Concatenation</td><td><code>a ##n b ##m c</code></td><td>$a, b, c \in \{0, 1\}$</td></tr>
</table>

Table 4.1: Summary of the patterns supported by my lowering. Concatenation can have an arbitrary number of elements in its sequence.

In order to correctly implement my direct lowering, I need to identify this particular pattern in MLIR. Given that an individual general lowering of each operation may not compose correctly, I opt for a more pattern-specific approach, where the NOI pattern is handled differently than the OI pattern, which differs from the concatenation pattern. Table 4.1 summarizes the types of patterns that are supported in my lowering.

In the NOI case, I extract the information needed from the above three operations, i.e. the `Value` of the antecedent, which must evaluate to a boolean, the `Value` of the consequent, which must also evaluate to a boolean, and the length of the delay, which can be any positive integer. With this information I can implement the method presented above, yielding the conversion in Figure 4.11. As we can see, my lowering generated two new registers, a

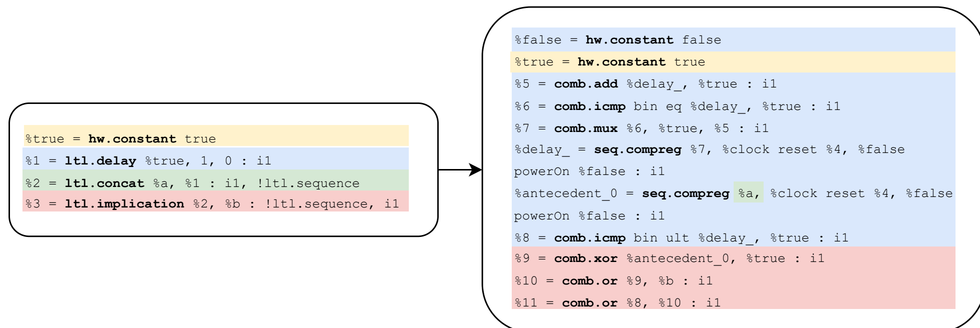


Figure 4.11: Illustration of the lowering of a non-overlapping implication pattern to core dialect operations. Note that in this pattern, the `ltl.concat` operation is only useful for obtaining the antecedent.

`delay` register which counts the number of cycles elapsed since the assertion was last disabled, and a `antecedent_0` register, which delays the antecedent by one cycle. The reset signal used for these new registers is the disable condition of the assertion, as disabling the assertion interrupts the evaluation of a sequence, making any reasoning about it incoherent (and thus our entire evaluation of the property should be reset).

Supporting Concatenation Concatenation is the most basic type of sequence and is used to expresses a series of signals holding across several cycles. Taking a look at the NOI lowering, we can see that concatenation follows a similar implementation pattern. Rather than only delaying the antecedent, I delay each element in the sequence by the number of cycles that follow it in said sequence. This means that for a sequence a ##n_0 a_0 ##n_1 a_1 ... ##n_m a_m, a is lowered to a pipeline of $n_0 + n_1 + ... + n_m$ registers, a_0 is lowered to a pipeline of $n_1 + ... + n_m$ registers, etc... The final value of the concatenation is the conjunction of all of the last registers in the pipelines. The concatenation should be disabled as long as $\sum_{i=0}^{m} n_i$ cycles have not elapsed.

```
a ##n0 a0 ##n1 a1 ... ##nM aM

// becomes

D = sum(i in 0 until M)(ni) // sum of all delays

reg delay;
reg a_0, ..., a_last; // D registers
reg a0_0, ..., a0_last; // (D - M) registers
reg a1_0, ..., a1_last; // (D - M - (M-1)) registers
//...
reg am_minus1_last;

delay' = disable ? 0 : delay + 1;

a_0' = a;
a_1' = a_0;
//...
a_last' = a_(D-1);

a0_0' = a_0;
a0_1' = a0_0;
//...
a0_last' = a0_(D - m);

//...

am_minus1_last' = am-1;

assert (delay < D) || a_last && and(i : 0..m)(ai_last) && am || disable
```

Figure 4.12: Implementation of a general SVA sequence using only exact delays.

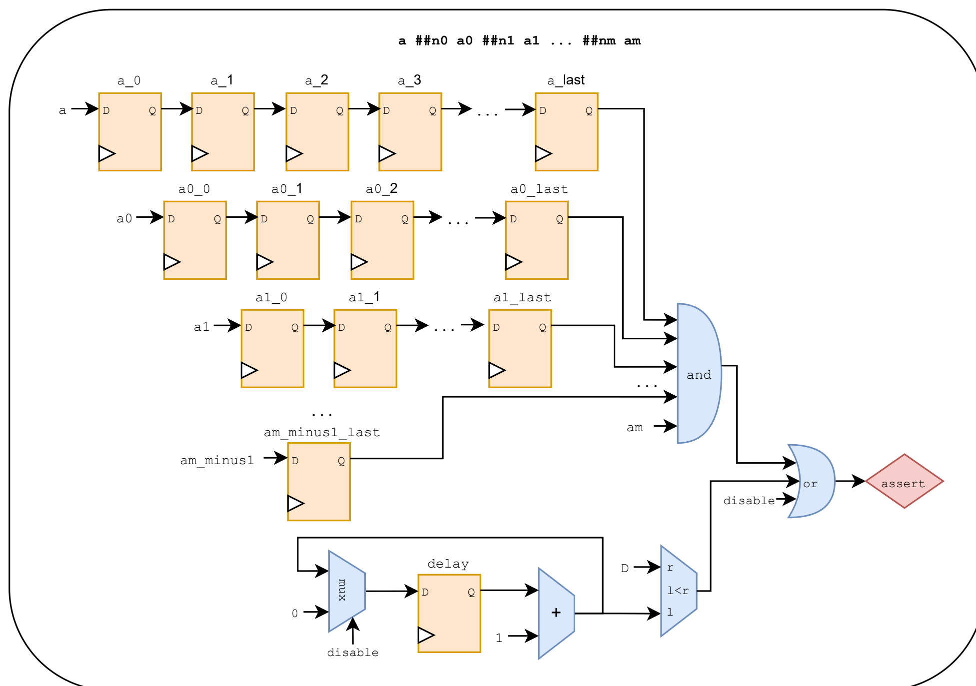


Figure 4.13: Diagram illustrating the circuit generated by lowering a generalized concatenation using my solution. As with the NOI lowering, only the delay register requires being reset using the disable condition associated to the property.

As we can see in Figures 4.12 and 4.13, even simple sequences start to generate a significant amount of hardware, which highlights the complexity of this translation task.

With these two contributions, I now support the most commonly used SVA properties identified in the survey. This pass is integrated into the `firtool --botr2` compilation flow before the `btor2` emission. In general this pass will be the final pass in the core dialects during compilation, as it is only needed to add verification support for CIRCT's target formats. This pass can also be used manually using `circt-opt --lower-ltl-to-core <design>.mlir`.

Chapter 5

Verifying the Compiler Passes

Two new compiler passes are introduced in this work, which need to be verified in order guarantee their correctness. To do so, I implemented a logical equivalence checker to verify the formal back end and a cycle-bound exhaustive tester to verify the SVA property lowerings. This chapter describes both of those testing techniques as well as the results of the test campaign.

5.1 Verifying the BTOR2 Emission

We start by verifying the core of my solution, which is the `btor2` back end in CIRCT. In this case, I am able to compare my solution to the formal back end that was implemented in the Scala FIRRTL Compiler (SFC). This allows us to take a differential approach to our verification. I start by tackling the design of an oracle that can be used to tell us whether a conversion is correct or not. Given that I am comparing two formal descriptions of a circuit, I can use formal equivalence checking [52] as an oracle. To do so, I create what is called a "miter" circuit [17] from the two input designs. A miter circuit is a single design containing both of the input designs concatenated to each other and sharing a single set of identical input and output ports. Then the two outputs are removed and replaced with an equivalence oracle in the form of an assertion that both outputs are equal. Figure 5.1 shows an example of what this process may look like for two equivalent designs. The result is checked using `btormc`, and if `btormc` finds a set of inputs that leads to a disagreement in the outputs we have found a bug in our design. Figure 5.2 shows an overview of the structure of the formal equivalence checker.

For the test-case generation problem, I ran this on a handful of designs that contained only elements supported by both the CIRCT and SFC formal back ends. In doing, I found that the two back ends handle uninitialized registers quite differently [65]. Given that this is considered undefined behavior, neither of the two approaches are technically incorrect. As a

```
1 sort bitvector 32
2 input 1 a
3 sort bitvector 33
4 add 3 2 2
5 output 4

++merge with++

1 sort bitvector 32
2 input 1 a
3 constd 1 2
4 sort bitvector 33
5 mul 4 2 3
6 output 5

==becomes==

1 sort bitvector 32
2 input 1 a
3 sort bitvector 33
4 add 3 2 2

5 sort bitvector 32
6 constd 5 2
7 sort bitvector 33
8 mul 7 2 6

9 sort bitvector 1
10 neq 9 4 8
11 bad 10
```

Figure 5.1: Example of two equivalent circuits being merged into a miter circuit. A condition for creating a miter circuit is that both designs share the exact same interface [17]. Here, I take the two first designs and merge them by removing the second instance of the inputs and converting the outputs into an equality assertion. This can then be given to `btormc` to check for equivalence.

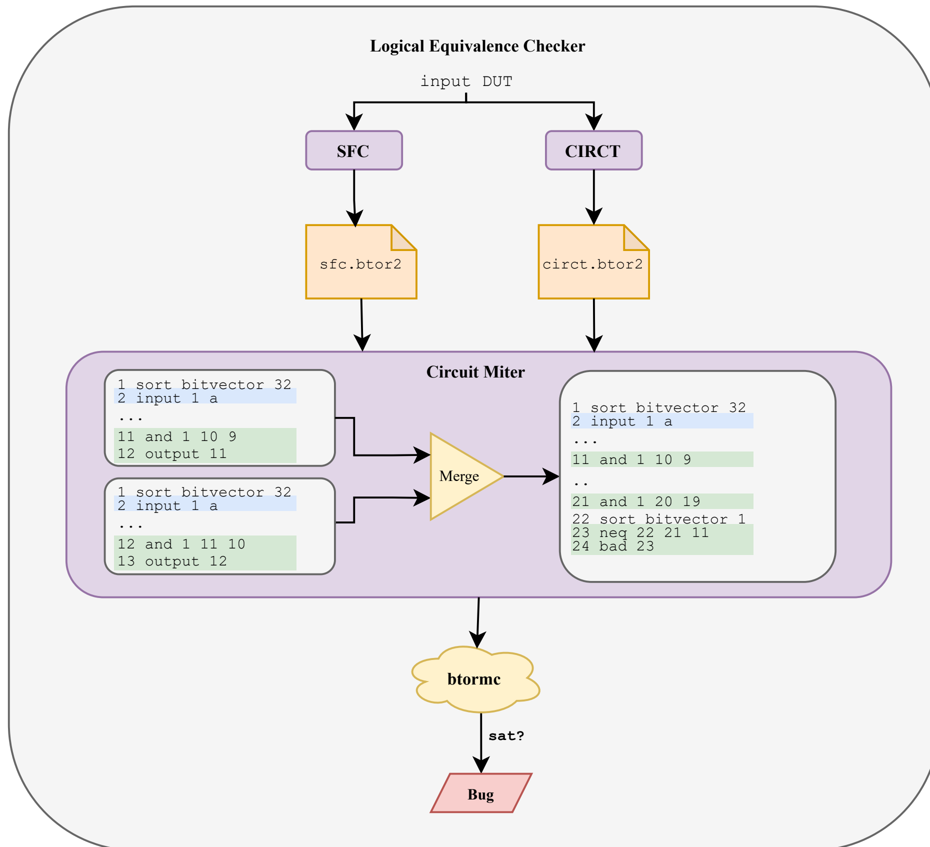


Figure 5.2: Overview of the logical equivalence checker built to verify the formal back end. This functions by taking a certain design as input (usually as a FIRRTL of Chisel file), then converting it into a `btor2` file using both the original Scala FIRRTL Compiler (SFC) and my formal back end in the CIRCT compiler using `firtool --btor2`. The instructions highlighted in blue show how the inputs are merged, and the instructions highlighted in green show how the outputs are handled. The resulting two `btor2` circuits are merged into a single miter circuit where the two outputs are compared in an assertion to verify their equivalence. This final miter `btor2` file is then checked using `btormc`. If the result is satisfiable, then we have found a bug in our design.

workaround, I implemented a post-processing compiler pass that gives all state elements an initial value which is only used for verification purposes and doesn't modify the design itself. This allows me to avoid having to deal with undefined behavior and leads to both back ends behaving equivalently on small sequential and combinatorial designs that do not use memories. Larger designs such as those from the RISC-V mini project[1] were planned to be verified with my solution, however this was not completed due to time constraints and is left for future work.

5.1.1 Creating a Miter Circuit

In order to create a miter circuit, I need to be able to easily manipulate `btor2` descriptions. To do so, we create a basic `btor2` parser that gives us small amounts of control over the program, such as moving a set of instructions up and down the program, thus modifying their identifiers. To avoid dealing with tedious string manipulations, we recreate a syntax tree from our input strings and store them as two separate ordered lists of instructions. The line identifiers of each instruction are associated to the instruction through an internal field. An additional validation pass is used to guarantee the well-ordering of the identifiers. The instructions themselves are represented as a base class which is extended for each instruction type. Figure 5.3 shows the implementation of the `Instruction` base class. As mentioned, each instruction type is its own subclass of the `Instruction` class, where they are extended to contain the specific information about that class, e.g. the `constd` instruction requires an additional field for its constant value, which is a non `Instruction`-typed operand.

The final result basically functions as a `btor2` compiler that allows us to parse, traverse, and perform basic manipulations on any `btor2` design [33]. Using this representation, we can easily merge two designs together using the merge function shown in Figure 5.4.

The resulting miter file is the serialization of the above merged files. The created compiler also support passes built as functions that take a list of `Instructions` as a parameter. These can then be run arbitrarily in the main function, or through command line arguments, to perform basic validation on the `btor2` design before outputting it. For example, this is used to guarantee that the `lids` in the miter circuit are well-ordered. The output of this verification work is an open-source tool called `btor2-opt`[2].

[1] https://github.com/ucb-bar/riscv-mini
[2] https://github.com/Dobios/btor2-opt

```python
class Instruction:
    def __init__(self, lid: int, inst: str,\
            operands: list[Instruction] = []):
        self.lid = lid
        self.inst = inst
        self.operands = operands

    def move_up(self, amount: int):
        self.lid += amount

    def move_down(self, amount: int):
        self.lid -= amount

    def move(self, lid: int):
        self.lid = lid

    def eq(self, inst: Instruction) -> bool:
        return self.operands == inst.operands and\
            self.inst == inst.inst

    def isin(self, p: list[Instruction]) -> bool:
        for inst in p:
            if self.eq(inst):
                return True
        return False

    def serialize(self) -> str:
        return str(self.lid) + " " + self.inst + " " +\
            [str(op.lid) + " " for op in self.operands]
```

Figure 5.3: Implementation of the `Instruction` base class in Python. This simply allows for a unified interface to manipulate and serialize instructions in such a way that it allows for minimal overhead when merging two circuits.

```python
def create_lec_assertion(
    out1: Instruction, out2: Instruction, base_lid: int
) -> list[Instruction]:
    op1 = out1.operands[0]
    op2 = out2.operands[0]
    sort = Sort(base_lid, 1)
    neq = Neq(base_lid + 1, [sort, op1, op2])
    bad = Bad(base_lid + 2, neq)
    return [sort, neq, bad]

def merge(p1: list[Instruction], p2: list[Instruction])\
        -> list[Instruction]:
    # Start by extracting the inputs
    inputs = []
    for op in p1:
        if isinstance(op, Input):
            inputs.append(op)
    # Extract outputs (assume only 1 output per design at end of file)
    out1 = p1[len(p1) - 1]

    # Then reconstruct p2 without inputs and with an offset lid
    new_p2 = []
    cur_lid = len(p1) # don't count the output of p1
    for op in p2:
        if not isinstance(op, Input):
            op.move(cur_lid)
            cur_lid += 1
            new_p2.append(op)
        # Update input lids in operands
        for oper in op.operands:
            if isinstance(oper, Input):
                if oper.isin(inputs):
                    oper = next(inp for inp in inputs if inp.eq(oper))
    out2 = p2[len(new_p2) - 1]

    lec = create_lec_assertion(out1, out2, new_p2[len(new_p2) - 1].lid)

    # Remove outputs
    p1.pop()
    new_p2.pop()

    return p1 + new_p2 + lec # merge everything
```

Figure 5.4: Merge function that creates a miter circuit from two parsed btor2 designs. The final miter circuit has only one set of inputs and no outputs.

5.2 Verifying the SVA Property Implementation

The implementation I introduced for the two SVA properties lowers temporal logic to synthesizable logic by looking at the semantics of the expressions. Given the nature of this approach, I need to design robust methods for verifying the correctness of these lowerings. To do so, I take two approaches:

- **Exhaustive Testing:** Generate all possible non-overlapping implication (NOI) statements within a given bound, then for each one run all possible combinations of values for `a` and `b` over a given number of cycles. Next, I compare the standard SVA lowering to my core lowering running on the generated stimuli.
- **Semantic Equivalence Checking:** Express the comparison as an equivalence problem and use a formal tool to check it.

Each approach allows us to verify different correctness properties of my implementation, which are detailed in the following paragraphs.

5.2.1 Cycle-bound Exhaustive Differential Testing

As with any automated testing task, there are two problems that need to be solved, the test-case generation problem and the oracle problem. In my case, I start by proposing the following solutions:

- **Test-case generation problem:** Unlike with my `btor2` pass, the space of supported SVA properties is very small, so I can take a cycle-bound exhaustive expression generation approach where I create every possible expression of NOI within a given number of simulated cycles, i.e. the length of the simulation, and then generate every possible input vector for the antecedent and the consequent within that number of cycles.
- **Oracle problem:** Use a differential testing approach, where I compare my solution to a well-respected commercial SVA implementation, Synopsis VCS [61].

Figure 5.5 illustrates the structure of our exhaustive tester. My system takes two inputs: a maximum delay length N, and a number of cycles I an simulating for. I start by generating the NOI properties, such that they cover all possible delays up until N. Then, for each generated statement, I generate all possible binary input vectors for the antecedent and consequent values. These vectors store the values used for the antecedent, the consequent, and the reset at each cycle in the simulation. Finally, for each generated triplet I generate a test-bench that stimulates the properties using the given values at each cycle and use said test-bench to simulate the property lowered to SystemVerilog under the two compilation pipelines.

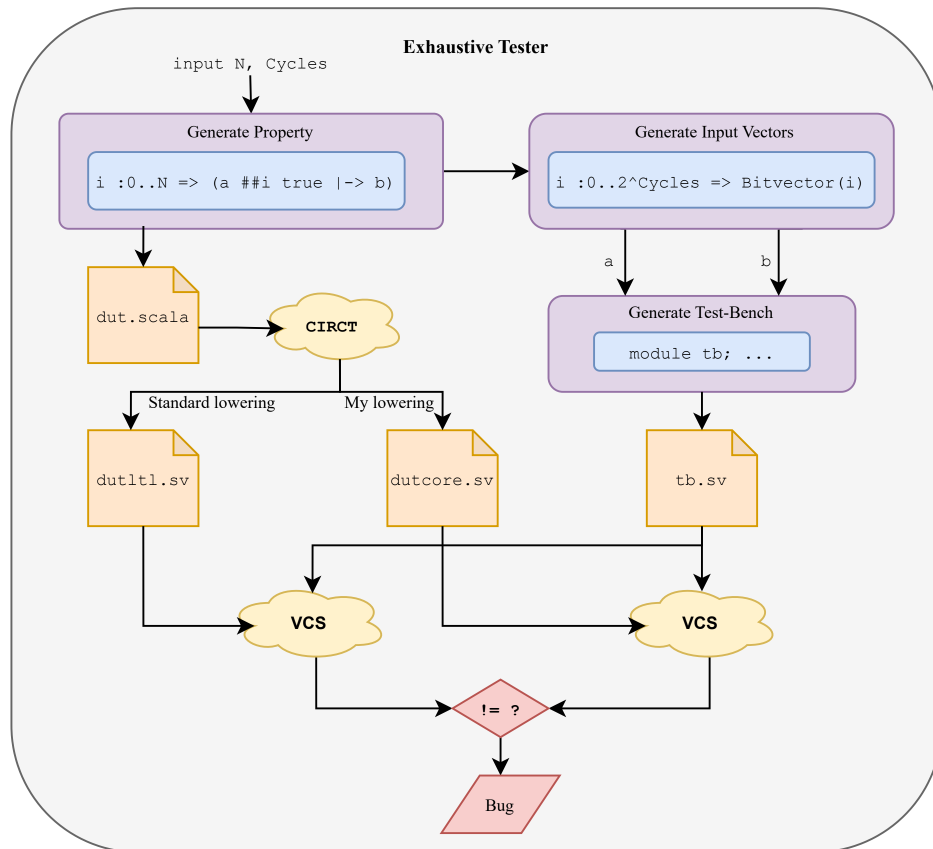


Figure 5.5: Overview of the exhaustive test infrastructure used to stress-test the `lower-ltl-to-core` pass. The testing tool takes two inputs, `N`, which is the longest delay I want to generate, and `Cycles`, which is the number of cycles I want to exhaustively simulate our designs for. The two designs are compared running with the generated test-bench on Synopsys VCS, which is a commercial Verilog simulator that supports certain SVA properties, including NOI.

Generating Properties As mentioned above, I generate every possible NOI property up to a given delay bound. The generated properties are then wrapped in a dummy module which takes the antecedent and consequent as inputs. Figure 5.6 illustrates the design generated by the first part of

```
class NOI extends Module {
    val a, b, t = IO(Input(Bool()))
    AssertProperty(a.delay(i) ### t |-> b)
}
```

Figure 5.6: Code generated by our exhaustive tester. In practice i would be replaced by a concrete value and the input `t` is always given the value 1, which is used for syntactical reasons.

the exhaustive tester. The design is then lowered using both the standard

```
// Generated by CIRCT unknown git version
module NOI(input clock, reset, a, b);
  reg hbr = 1'h0;
  reg delay_ = 1'h0;
  reg antecedent_0 = 1'h0;
  always_ff @(posedge clock)
    hbr <= reset | hbr;
  always_ff @(posedge clock) begin
    if (reset) begin
      delay_ <= 1'h0;
      antecedent_0 <= 1'h0;
    end
    else begin
      delay_ <= delay_ | delay_ - 1'h1;
      antecedent_0 <= a;
    end
  end // always_ff @(posedge)
  always @(posedge clock)
    assert(~(hbr & ~reset) | ~delay_ | ~antecedent_0 | b | reset);
endmodule
```

Figure 5.7: Lowering of the generated design using my custom SVA-free pass pipeline.

CIRCT compilation pipeline, and my custom SVA-free pipeline. The output of those are an SV file containing the design with an SVA property used as the assertion and a second SV file containing the same design with my synthesized version of the property being checked under a standard assertion. In order to verify that both designs are equivalent, I need a simulator that supports SVA properties. Unfortunately, only a handful of Verilog simulators support these constructs for simulation, which is why I resort to using the commercial simulator Synopsis VCS. Figures 5.7 and 5.8 show the lowered designs that will be run through VCS alongside a test bench during exhaustive testing. Both of these are generated by CIRCT using my custom pipeline, presented in Figure 5.7, as well as the standard CIRCT SV pipeline, presented in Figure 5.8.

Generating Test Benches In order to have some context to compare the properties in, I must stimulate the designs using the same test bench. Given the limited scope of our inputs, i.e. 1-bit input vectors, it is reasonable to generate every possible sequence of inputs within a certain number of cycles. Given the total number of cycles, I generate 3 input vectors per test bench: one for the antecedent, one for the consequent, and one for the reset. These vectors are then used to create a standard SV test bench file that stimulates the design with each value of each vector over a number of cycles equal to

```
// Generated by CIRCT unknown git version
module NOI(input clock,        reset, a, b);
  reg hasBeenResetReg;
  initial
    hasBeenResetReg = 1'bx;
  always @(posedge clock) begin
    if (reset)
      hasBeenResetReg <= 1'h1;
  end // always @(posedge)
  assert property (@(posedge clock)
    disable iff (~(hasBeenResetReg === 1'h1 & reset === 1'h0))
    a |=> b
  );
endmodule
```

Figure 5.8: Lowering of the generated design using the standard CIRCT pipeline. Here, the design contains an SVA property, which is not supported in most open-source simulators.

the length of those vectors. Figure 5.9 shows the structure of the test benches that are generated by my exhaustive tester. The idea is to connect the values from the generated input vectors into an instance of the design. Given that designs already have an assertion in them, the goal is to check the outputs from VCS running these test benches on the two designs and ensure that they are identical in both cases.

Revealed Bugs This exhaustive test campaign was run over 20 cycles, taking about a day to execute given the slow startup time of VCS. This simple test campaign was already able to reveal two bugs, one in my implementation, and one in the design of the lowering itself (both of which have since been resolved). The implementation bug was simply a typo, but one difficult to spot as it modified only a single character in the output MLIR. Instead of delaying the antecedent, I was accidentally delaying the `true` signal at the end of the concatenation. This lead to the NOI signal behaving the same regardless of the number of cycles I would delay it by.

The second bug that this test campaign revealed was in my reasoning about the reset signals used for my generated registers. Originally, I interpreted the SVA standard [6] as saying that the property was never reset, so the sequence is considered available once the antecedent's delay has been reached in the simulation. Therefore, I could reset my registers on the module's reset so that my delay counter started when the simulation started. This, however, was wrong — my implementation would yield an incorrect result when the design is reset in the middle of the simulation. This led me to understanding that the correct reset signal should rather be the disable condition given to

```
module tb;
    reg clock; reg reset; reg a; reg b;
    NOI dut (.clock(clock), .reset(reset), .a(a), .b(b));
    always #5 clock = ~clock; // Generate Clock
    initial begin
        reset <= 1; // Reset the design
        {clock, a, b} <= 0;
        repeat(2) @(posedge clock);
        reset <= 0; // End reset
        // Bitvector execution
        a <= a_0; b <= b_0; reset <= reset_0;
        @(posedge clock);
        a <= a_1; b <= b_1; reset <= reset_1;
        @(posedge clock);
        //...
        a <= a_n; b <= b_n; reset <= reset_n;
        repeat(2) @(posedge clock);
        #20 $finish;
    end
endmodule
```

Figure 5.9: Test bench generated to stimulate our designs using the input vectors generated by my exhaustive tester. The antecedent and consequent are used to set the values in the assertion and the reset is used to check that our assertions are correctly disabled.

the property, which was confirmed by rerunning the test campaign. The code for this part of the verification work is fully open-source[3].

5.2.2 Semantic Equivalence Checking

Additionally, a form of semantic equivalence checking was explored to verify the lowerings. While the previous method focused more on the test case generation, here I wanted to see if I could solve the oracle problem differently to reveal a different set of bugs. The idea was to create a type of miter circuit, similar to what is described in Section 5.1, but this time by checking the semantic equivalence of the assertions rather than simply the circuits themselves. To do this, both circuits generated in the first part were merged into a single circuit, where the assertion was rewritten to check the equivalence between the two conditions.

In order to check for equivalence between two assertions, we want to rewrite them such that one is true if and only if the other is, as is illustrated in Figure 5.10.

[3] https://github.com/Dobios/SVExhaustiveTester

```
assert cond1;
assert cond2
// becomes
assert (cond1 implies cond2) and (cond2 implies cond1);
```

Figure 5.10: Example of an equivalence checking assertion.

There are several ways to express implications in SVA [6], all of which allow for differ types to be used on each side of the implication. Given that we need to be able to express a property, such as NOI, on each side of our implication, we can not use a simple OI (`|->`) as we did previously as this only accepts sequences in the antecedent, we instead need to use SVA's `implies` operator, which holds if the antecedent evaluates to false or the consequent evaluates to true. The difference between these two operators is in the timing of when the consequent is evaluated. For the overlapping-implication operator, the consequent starts being evaluated on the last cycle in which the antecedent sequence holds, while for the `implies` operator, both the antecedent and consequent are evaluated concurrently. This allows for properties to be used in the antecedent, as they don't require a strict end cycle. Luckily, SVA offers a syntax that expresses our if and only if condition in the form of `iff`, which encodes `(cond1 implies cond2) and (cond2 implies cond1)`.

For this expression to be valid both sides must be properties and thus my lowering needs to be wrapped in a `assert property` statement that is properly disabled. This is yet another modification that may cause slight semantic changes to my lowering. Figure 5.11 shows the result of this fusion between our two generated circuits for a basic NOI with a delay of 1 cycle. Once this circuit is generated, I can simulate it with the exhaustive test benches presented earlier and check if VCS signals a failure or not. This simplifies the oracle part of the exhaustive tester and removes any potential error that could come from incorrectly parsing VCS's output. Figure 5.12 shows the structure of my test infrastructure using the semantic equivalence check approach for the oracle. I now need only a single instance of VCS to check the correctness of my lowering, making it significantly faster. There is also no need to parse the output of the simulator, as a crash is sufficient to detect whether or not the two assertions are equivalent.

```
// Generated by CIRCT unknown git version
module NOI(input clock, reset, a, b);
  // Lowered SVA-free circuit
  reg hbr = 1'h0;
  reg delay_ = 1'h0;
  reg antecedent_0 = 1'h0;
  always_ff @(posedge clock)
    hbr <= reset | hbr;
  always_ff @(posedge clock) begin
    if (reset) begin
      delay_ <= 1'h0;
      antecedent_0 <= 1'h0;
    end
    else begin
      delay_ <= delay_ | delay_ - 1'h1;
      antecedent_0 <= a;
    end
  end // always_ff @(posedge)
  property core;
    (@(posedge clock)
        disable iff(hbr & ~reset)
        (~delay_ | ~antecedent_0 | b | reset));
  endproperty;

  // SVA property circuit
  reg hasBeenResetReg;
  initial
    hasBeenResetReg = 1'bx;
  always @(posedge clock) begin
    if (reset)
      hasBeenResetReg <= 1'h1;
  end // always @(posedge)
  property ltl;
    (@(posedge clock)
        disable iff (~(hasBeenResetReg === 1'h1 & reset === 1'h0))
        a |=> b);
  endproperty
  // Logical equivalence check
  assert property (@(posedge clock) core iff ltl);
endmodule
```

Figure 5.11: Miter circuit used to check the equivalence between our lowered assertion and the original SVA property assertion.

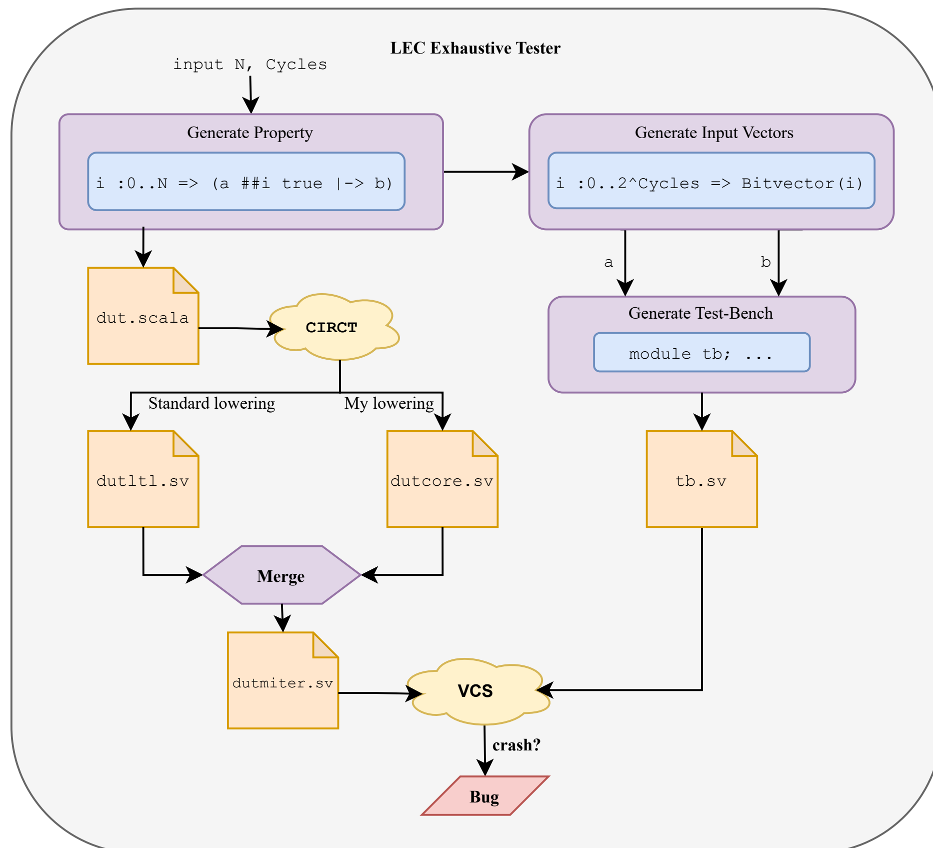


Figure 5.12: Overview of the exhaustive testing infrastructure using a semantic equivalence check for the oracle. The main difference compared to the previous figure is that the two generated designs, `dutltl.sv` and `dutcore.sv`, are merged into a single circuit called `dutmiter.sv`, which is then checked on a single instance of VCS. The benefit of this is that rather than having to intercept and parse the complex output of the simulator, I can simply look for a crash and report a bug if that happens.

Chapter 6

Related Work

The way SystemVerilog's concurrent assertions are supported varies drastically depending on the implementation of the language, and few solutions currently support the full specification. In this thesis, I present a solution to this that functions well for CIRCT and the format that is targeted, `btor2`, but there exists a handful of other solutions to this problem that solve it in a way better suited for other environments. In this section I present several of those solutions and express how they differ to the one proposed here.

6.1 ChiselTest

Previous versions of Chisel relied on a Scala-based compiler for its intermediary language called FIRTTL [39]. this compiler is known as the Scala FIRRTL Compiler (SFC). While FIRRTL is still used as an IR for Chisel, it now goes through CIRCT rather than the SFC. ChiselTest [50] is a testing library for Chisel which enables various verification functionalities directly in test benches written as Scala unit tests [68]. ChiselTest also allows for designs to be converted into a format that can be used for Bounded Model Checking through custom compiler passes added to the SFC. This is done using the `verify` function and is based off of a similar conversion method as the one used in this thesis's solution.

Figure 6.1 shows an example of how ChiselTest can be used to perform Bounded Model Checking. One difference with the solution presented in this thesis is how the model checking is performed. In ChiselTest, the Bounded Model Checking is performed directly in the Scala testing framework by querying the selected backend when necessary [28], while the solution in CIRCT functions as more of a traditional compiler which takes any design and converts it into a format which meant to be handed off to an external solver. Both solutions support `btor2`-based solvers, however the ChiselTest

```
class Add extends Module {
  val in = IO(Input(UInt(8.W)))
  val out = IO(Output(UInt(8.W)))
  out := in + 1.U
  assume(in > 12.U && in < 255.U)
  assert(out > 13.U)
}
verify(new Add, Seq(BoundedCheck(1), DefaultBackend))
```

Figure 6.1: Example of a basic incrementing circuit implemented in Chisel. The `verify` function is used to convert this design into a format specified by the selected back end. By default, the back end will be Z3, so the design will be unrolled and converted into SMTLib using the number of cycles defined by the `BoundedCheck` argument [28].

solution only works on designs implemented Chisel, while my solution works with all of CIRCT's front ends.

ChiselTest also directly supports limited SVA-like temporal expressions, such as the `past` function, which delays the value of a given signal by a cycle. Using `past` in an assertion automatically delays the evaluation of that assertion [43]. This allows one to express things like `a > past(a)`, which states that a is monotonically increasing.

An extension of ChiselTest, named CHA [74], was proposed to support a number of SVA properties and sequences. CHA does so by converting the supported properties into an LTL formula that can then be handed off to SPOT [29]. SPOT then converts this formula into a deterministic automaton, which is then parsed and implemented as synthesizable hardware by CHA. While this initally seems to be a good approach to solve this issue, the implementation is not ideal. CHA can only provably support very basic properties, those of a similar nature to the ones supported by my work, and their implementation relies on string expressions in a custom assertion, which is very error-prone and not ideal as a general language contribution. Additionally, CHA depends on an external tool, SPOT, to support these basic properties, while my solution is entirely implement in CIRCT. A similar approach was taken to integrate SPOT into ChiselTest directly, but this project never passed the prototype phase [45].

6.2 Yosys

Yosys [73, 72] is a popular open-source tool which implements a subset of SVA properties and sequences specifically for Bounded Model Checking. This tool can construct an AST of the described hardware then lower it to a `btor2` description that can be used for Bounded Model Checking. This is currently the most complete solution for Bounded Model Checking, but

it only supports SystemVerilog as a front end, and it relies on having a commercial license for Verific [7] to be able to use many of its features. The actual lowering of SVA properties to synthesizable hardware, however, is completely open-source.

The way the Yosys lowers SVA properties and sequences is based on constructing automata that monitor the behavior of the temporal expressions. This automaton construction is done on a case-by-case basis through a large sequence of special-casing to determine exactly how to intersect different automata. The solution does not exploit many techniques found in automata theory literature to optimize the construction, but rather benefits from specific knowledge of the semantics they are trying to lower to obtain a decently scalable solution.

The implementation of Yosys contains an internal representation of non-deterministic automata, which is used to easily nest sequences and properties by intersecting their respective automata. Yosys makes the generated automata deterministic several times throughout the construction process. This allows it to perform necessary optimizations while constructing the final automaton that is then implemented as synthesizable logic. A benefit of this approach is that is allows for the construction of more complex properties than my direct lowering, however this is only supported in Yosys for designs written in SystemVerilog, while my lowerings are implemented at a higher level in the compilation pipeline and support many different front ends.

6.3 BlueSpec Verilog

Prior work [55] has looked at how SVA properties and sequences can be implemented in a rule-based environment such as BlueSpec Verilog (BSV). BSV is an implementation of the BlueSpec [16] rule-based high-level synthesis semantic model in SystemVerilog. The goal of that work is to enable the expression of SVA semantics in BSV. This implementation is split into two parts, the sequence implementation and the property implementation. The idea behind both is based in the conversion of SVA expressions to FSMs. This is done by interpreting the SVA expressions using the semantics outlined in Appendix F of the SV specification [2], and then converting each primitive into its own state machine following methods similar to those presented in Subsection 2.2.2.

Once the FSMs are obtained from the SVA expressions, we must check them to verify that their expression holds. This is done in two different ways in order to handle both sequences and properties. Sequences are represented by an interface containing two methods, `advance()`, which advances to the next state in the FSM and returns whether or not the sequence matches for this cycle, and `ended()`, which signals that the sequence has ended. Their

implementation guarantees that FSMs return to their initial state after halting, allowing for their immediate reuse. This structure allows for the description of sequences containing elements such as the concatenation operator `##1`, enabling the description of sequences such as `s1 ##1 s2` which holds if `s1.end()` returns `true`, then `s2.end()` returns true.

The implementation of properties in BlueSpec Verilog is more complex. A property expressed in the form of `sequence |-> property` is first converted recursively into two automata, one for the sequence and one for the property. A worst case analysis is then done on the delay of the antecedent sequence to obtain the maximum possible cycle-duration, `max_n`, for that sequence. BSV then creates `max_n` copies of the property automaton. This ensures that the overarching property has enough versions of the consequent automaton in the initial state to be able to check that the property matches on every cycle that the sequence could match in. These generated automata are then used directly to match the described property using the FSM interface we described earlier.

Unlike other approaches, BSV encodes properties as FSMs, which are much easier to compose into larger properties without knowing the entire expression beforehand. This is a very similar approach to how we generate the logic for our NOI implementation, as the combination of our register-delay pipeline and our delay counter can be interpreted as simply monitoring our conditions through copies of an FSM.

Chapter 7

Conclusion

In this work, I have presented two new compiler passes, which add a formal back end for Bounded Model Checking as well as a lowering for certain SVA properties to synthesizable logic to the CIRCT compiler. These two passes were then verified using a custom automated test suite designed specifically for each solution. In this conclusion, I summarize the results and contributions of this thesis as well as propose some directions for future work.

7.1 Results

The formal verification flow, which includes the SVA property and sequence lowering, and the `btor2` emission pass, was integrated directly into `firtool` which is the compiler for Chisel in the CIRCT project. This integration was done by adding a `--btor2` flag that triggers the use of the SVA property lowering pass followed by the `btor2` emission pass to then yield a `btor2` description of the original design using only a single flag.

The SVA property lowering can be used with any of CIRCT's back ends. This enables the use of SVA properties for any tool that can understand basic Verilog. Table 7.1 shows a list of Verilog simulators and compilers and their support for SVA properties. As we can see, very few tools actually support SVA properties for simulation. Yosys supports `btor2` emission with decent SVA property in with its open-source parser, but simulation is only supported using the commercial front end. With the contributions of this thesis one can use SVA properties with entirely open-source simulators, e.g. the popular Verilog simulator Verilator.

Finally, the performance of the proposed formal back end far outperforms that of the Scala-based one. This is a result of the environment in which the CIRCT compiler exists, i.e. MLIR and C++. While the effort it takes

Tool	SVA property support	Open-Source?
Synopsys VCS [61]	Yes	No
Intel Questa [38]	No	No
Yosys (open-source parser) [72]	Yes (not for simulation)	Yes
Yosys (Verific [7] parser) [73]	Yes	No
Icarus Verilog [71]	No	Yes
Verilator [60]	No	Yes

Table 7.1: List of Verilog Simulators and compilers, whether or not they support SVA properties, and whether they are open-source. As this table shows, very few simulators support the use of SVA properties – and all those that do are commercial. Yosys allows for the emission of `btor2` with limited SVA property support using their open-source subset, but it does not support open-source simulation. With the contributions from this work, all of these tools can now support the SVA properties supported in my lowering pass.

to implement such a back end is far greater in MLIR than in Scala, the performance gains are significant enough for this to be worthwhile.

7.2 Future Work

While this thesis introduces important contributions to the CIRCT compiler and verification for high-level hardware languages in general, there is always more one could do to improve this new verification ecosystem. The following is a non-extensive list of ideas and potential directions to extend this work in order to improve the capabilities of the CIRCT compiler, both in terms of designs it can support for formal verification and in the temporal properties the user can express.

Extending the Formal Back End The current `btor2` emission pass supports converting designs which do not rely on the use of memories in any way. While this already covers a large number of designs, adding support for memories would allow for descriptions of entire processors to be verified using Bounded Model Checking.

Memories can be supported by converting them into a particular state element that uses the `array` sort rather than the `bitvector` one. Reading and writing these special states could be done through the use of specific expressions which model that behavior, such as `read` and `write`. Doing so would allow the formal back end to support the same design functionalities as the SFC formal back end.

Add Support for Variable Delays in the LTL to Core Lowering The only delays currently supported in my solutions are fixed delays, i.e. delaying by an exact number of cycles. In practice, as the discussion highlighted in

Figure 4.6 illustrates, the use of variable delays can be very important when describing behaviors such as transactions that are expected to take a variable number of cycles to propagate across a network-on-chip [58].

Variable delays are expressed in the form `a ##[n:m] b` which means that `a` holds between `n` to `m` cycles before `b` holds. Implementing this would require a similar approach to what I described in Paragraph 4.2.2, but with a wider range of cycles that must be checked. The idea would be to create a pipeline of `m` registers to track the longest possible delay that `a` could have. Then the accept condition would check for a range of registers rather than an exact amount. For example, while `a ##n b` could have the following final assertion (following the implementation strategy from Figure 4.12):

```
assert (delay < n) || a_n && b || reset
```

Using a variable delay, of the type `a ##[n:m] b` could yield the following final assertion:

```
assert (n <= delay && delay < m) || (a_n || ... || a_m) && b || reset
```

This would generate the same number of registers as an exact delay of the upper bound of the variable delay.

Arbitrary Concatenation in the Antecedent of an Implication The next step in extending the support for SVA properties proposed in this thesis would be to allow for arbitrary sequences to be expressed in the antecedent of an implication. The current solution only supports a very specific pattern for implication, `a ##n true |-> b`. Using a combination of the two methods presented above, one could create a more generalized lowering that supports a generic version of non-overlapping implication. However, there is some difficulty in checking these types of properties when they find themselves in an `always` property, as this would require creating multiple copies of the antecedent lowering to monitor every element of the general concatenation on every cycle. A direct lowering would be possible, but using an automaton to monitor the property instead could make it easier to check, e.g. using the same method as `BSV` [55].

Extended SVA Property Support through Automata Transformations Relying on direct lowerings of an SVA property to synthesizable hardware drastically limits the scope of what one can support in a compiler. A solution to this limitation would be generating automata which monitor the property.

In order to do so, SPOT [29] could be integrated into a compiler pass in order to facilitate the automata generation and manipulation. This could be done by converting the input SVA property into an LTL formula which could then be given to SPOT through its C++ library. Note that this cannot be done for all SVA properties. Properties that utilize indefinite sequences or infinite

delays are difficult to encode as an LTL formula. This tool would then output a deterministic automaton which monitors the LTL formula it was given in a format called `HOA` [8]. In order to integrate this, we would need to parse the `HOA` output and convert it into the `fsm` dialect in CICRT which could then be lowered into the core dialects. Having this integration would greatly simplify the task of supporting more complex SVA properties, as it would reduce the problem of finding an equivalent implementation to that of finding an equivalent LTL formula.

Better Testing of the Formal Back End As mentioned in Section 5.1, the verification of the formal back end still has room for improvement. In particular, once memories are added to the CIRCT formal back end, one could explore larger designs using the formal equivalence checking tool I constructed to see if the CIRCT formal back end is equivalent to the SFC formal back end in a more realistic setting.

7.3 Conclusion

This work presents a new compilation flow in MLIR from high-level hardware designs to a format suitable for Bounded Model Checking that enables the use of SVA-like temporal properties in the specification. This was done through: first, introducing a formal back end in the form of a `btor2` emission pass in the CIRCT compiler, second, designing and implementing a lowering for the two most common SVA properties and sequences (which enabled the use of these temporal semantics in both open-source simulators and the initial formal back end), and finally, designing an automated test-suite to verify the correctness of these new passes. For the formal back end I built an equivalence checking tool which creates a miter circuit by merging the output of the SFC's formal back end with that of my own back end then checks for disagreements in the outputs of both `btor2` circuits using `btormc`. For the SVA property and sequence lowering I created an exhaustive tester which compares assertions expressed using SVA properties with our lowered versions using VCS. With these two verified passes, one can now express a complex temporal specification directly in any of CIRCT's front ends and verify those formally using Bounded Model Checking entirely in open-source. This was previously only possible through the use of commercial tools using SystemVerilog. The importance of this contribution is highlighted by the interest my work has attracted from various agile hardware development teams that use high-level languages such as Chisel to improve their design efficiency — who were previously stuck using SystemVerilog for their verification. With these contributions more engineers can start adopting high-level languages and open-source tools for both design and verification.

I have learned many things through this work. Firstly, this project has taught me how modern high-performance MLIR-based compilers such as CIRCT are structured, and I have also learned how to contribute to them. While my initial contributions took months to be accepted into the compiler, my final ones took just a few hours thanks to the knowledge about writing MLIR passes I acquired through dozens of reviews on my code. Secondly, this project exposed me to the field of implementing checkers for temporal logic, and, more importantly, how difficult it is to design a general lowering from an LTL formula to an equivalent automaton. Finally, this project allowed me to explore various methods for verifying compiler passes and how one could go about creating automated test suites for that.

This project also triggered a movement towards improving the support for SVA-style temporal expressions in the CIRCT compiler. Since my initial contributions, there have been numerous discussions and follow-up contributions for various verification functionalities in the compiler which I have been able to be a part of. Alongside a research group at the University of Cambridge, I helped design a generalized automata dialect for MLIR, as well as a lowering from the `fsm` dialect to the core dialects, thus enabling potential future implementations of more complex temporal expressions. I have also reviewed and contributed to extensions of the `ltl` dialect to support various complex expressions that go far beyond what the current dialect supports. All of these new contributions and discussions surrounding verification support in these projects show that the future is bright for high-level hardware languages, and that hardware development has exited the commercially focused, HDL-based, status-quo it has been in for so many years.

Bibliography

[1] *Introduction to SVA*, pages 7–88. Springer US, Boston, MA, 2005.

[2] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[3] IEEE Standard for VHDL Language Reference Manual. *IEEE Std. 1076*, 2019.

[4] IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std. 1800.2*, 2020.

[5] IEEE Standard for Property Specification Language (PSL). *IEEE Std. 1850*, 20210.

[6] IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. *IEEE Std. 1800*, 2023.

[7] Verific Design Automation. Verific's Parser Platform. https://www.verific.com/wp-content/uploads/2019/05/brochure-june-2019-web.pdf, 2019.

[8] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejček. The hanoi omega-automata format. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 479–486, Cham, 2015. Springer International Publishing.

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC Design Automation Conference 2012*, 2012.

[10] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[11] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[13] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. 2008.

[14] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded Model Checking. *Advances in computers*, 58(11):117–148, 2003.

[15] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science*, Volume 2, Issue 5, November 2006.

[16] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery.

[17] D. Brand. Verification of large synthesized designs. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 534–537, 1993.

[18] Doron Bustan and John Havlicek. Some complexity results for systemverilog assertions. In Thomas Ball and Robert B. Jones, editors,

Computer Aided Verification, pages 205–218, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[19] Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 143–202, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[20] Chips-alliance. SV Tests. https://chipsalliance.github.io/sv-tests-results, 2023.

[21] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, Jul 2001.

[22] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[23] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[24] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, June 2020.

[25] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[26] Amelia Dobis. BTOR2 Conversion Pass. https://github.com/llvm/circt/pull/6378, 2024.

[27] Amelia Dobis. LowerLTLToCore pass, CIRCT. https://github.com/Dobios/circt/blob/ltltocore/lib/Conversion/LTLToCore/LTLToCore.cpp#L265-L367, 2024.

[28] Amelia Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. Verification of Chisel Hardware Designs with ChiselVerify. *Microprocessors and Microsystems*, 96:104737, 2023.

[29] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. From Spot 2.0 to Spot 2.10: What's new? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, volume 13372 of *Lecture Notes in Computer Science*, pages 174–187. Springer, August 2022.

[30] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog HDL and Its Ancestors and Descendants. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–90, 2020.

[31] Valentin Goranko and Antje Rumberg. Temporal Logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2023 edition, 2023.

[32] David J Greaves. A verilog to c compiler. In *Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000. Shortening the Path from Specification to Prototype (Cat. No. PR00668)*, pages 122–127. IEEE, 2000.

[33] Oleg Grenrus. Do you have a problem? Write a compiler! http://oleg.fi/gists/posts/2019-09-26-write-a-compiler.html, 2019.

[34] OpenHW Group. CORE-V CV32E40S RISC-V IP. https://github.com/openhwgroup/cv32e40s, 2021.

[35] Chris Higgs, Stuart Hodgson, and Eric Wieser. cocotb. https://github.com/cocotb/cocotb, 2021.

[36] Wilfrid Hodges and Thomas Scanlon. First-order Model Theory. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2024 edition, 2024.

[37] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[38] Intel. Questa. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/questa-edition.html, 2024.

[39] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In

Proceedings of the 36th International Conference on Computer-Aided Design, ICCAD '17, 2017.

[40] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. Mamba: Closing the performance gap in productive hardware development frameworks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[41] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing testing with formal verification in intel (c) core (tm) i7 processor execution engine validation. In *International Conference on Computer Aided Verification*, pages 414–429. Springer, 2009.

[42] Jack Koenig. Chisel6 LTL Specification. https://github.com/chipsalliance/chisel/blob/main/src/test/scala/chiselTests/LTLSpec.scala, 2023.

[43] Kevin Laeufer. Chisel formal verification support. https://github.com/ucb-bar/chiseltest/pull/673, 2023.

[44] Kevin Laeufer. Agile Hardware Design: Formal Verification. https://github.com/agile-hw/lectures/blob/main/22-formal/lec22-formal.ipynb, 2024.

[45] Kevin Laeufer and Vighnesh Iyer. Chisel sequences. https://github.com/ekiwi/chisel-sequences, 2022.

[46] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*, 2018.

[47] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[48] Chris Lattner, John Demme, Mike Urbach, Andrew Lenharth, Andrew Young, Schuyler Eldridge, Fabian Schuiki, and Hanchen Ye. CIRCT. https://github.com/llvm/circt, 2019.

[49] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan

Blagojevic, Pi-Feng Chiu, Rimas Avizienis, Brian Richards, Jonathan Bachrach, David Patterson, Elad Alon, Bora Nikolic, and Krste Asanovic. An agile approach to building risc-v microprocessors. *IEEE Micro*, 36(2):8–20, 2016.

[50] Richard Lin and Kevin Laeufer. ChiselTest. https://github.com/ucb-bar/chiseltest, 2024.

[51] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. 2021.

[52] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '06, page 836–843, New York, NY, USA, 2006. Association for Computing Machinery.

[53] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 587–595, Cham, 2018. Springer International Publishing.

[54] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A Compiler Infrastructure for Accelerator Generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[55] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil. Synthesis of synchronous assertions with guarded atomic actions. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, pages 15–24, 2005.

[56] Morten Borup Petersen. A dynamically scheduled hls flow in mlir. page 92, 2022.

[57] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019.

[58] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. S4noc: a minimalistic network-on-chip for real-time multicores. In *Proceedings of the 12th International Workshop on Network on Chip Architectures*, pages 8:1–8:6. ACM, October 2019.

[59] Fabian Schuiki. CIRCT verif dialect. https://circt.llvm.org/docs/Dialects/Verif, 2023.

[60] Wilson Snyder et al. Verilator. https://www.veripool.org/wiki/verilator, 2023.

[61] Synopsys. VCS. https://www.synopsys.com/verification/simulation.html, 2023.

[62] Shinya Takamaeda-Yamazaki. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings 11*, pages 451–460. Springer, 2015.

[63] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. *arXiv preprint arXiv:2102.02308*, 2021.

[64] Lenny Truong and Pat Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, 2019.

[65] Mike Turpin. The dangers of living with an x (bugs hidden in your verilog). In *Synopsys Users Group Meeting*, 2003.

[66] Muneeb Ulla Shariff. axi4 vip. https://github.com/muneebullashariff/axi4_vip, 2020.

[67] Moshe Y. Vardi. *An automata-theoretic approach to linear temporal logic*, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[68] Bill Venners, Lex Spoon, and Martin Odersky. *Programming in Scala, 3rd Edition*. Artima Inc, 2016.

[69] Jouko Väänänen. Second-order and Higher-order Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2021 edition, 2021.

[70] whitequark. amaranth. https://github.com/amaranth-lang/amaranth, 2022.

[71] Stephen Williams et al. Icarus Verilog. https://steveicarus.github.io/iverilog/, 2023.

[72] Claire Wolf. SymbiYosys, 2021.

[73] Claire Wolf and Johann Glaser. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[74] Shizhen Yu, Yifan Dong, Jiuyang Liu, Yong Li, Zhilin Wu, David N. Jansen, and Lijun Zhang. Cha: Supporting sva-like assertions in formal verification of chisel programs (tool paper). In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*, page 324–331, Berlin, Heidelberg, 2022. Springer-Verlag.

[75] Drew Zagieboylo, Charles Sherk, G. Edward Suh, and Andrew C. Myers. Pdl: a high-level hardware design language for pipelined processors. In *43rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2022.