ChiSA: Static Analysis for Lightweight Chisel Verification

JIACAI CUI, QINLIN CHEN, and ZHONGSHENG ZHAN, Nanjing University, China
TIAN TAN* and YUE LI*, Nanjing University, China

The growing demand for productivity in hardware development opens up new opportunities for applying programming language (PL) techniques to hardware description languages (HDLs). Chisel, a leading agile HDL, embraces this shift by leveraging modern PL features to enhance hardware *design* productivity. However, *verification* for Chisel remains a major productivity bottleneck, requiring substantial time and manual effort. To address this issue, we advocate the use of *static analysis*—a technique proven well-suited to agile development workflows in software—for lightweight Chisel verification.

This work establishes a theoretical foundation for Chisel static analysis. At its core is λ_C, a formal core calculus of ChAIR (a Chisel-specific intermediate representation for analysis). λ_C is the first formalism that captures the essence of Chisel while being deliberately minimal to ease rigorous reasoning about static analysis built on λ_C. We prove key properties of λ_C that reflect real hardware characteristics, which in turn offer a form of retrospective validation for its design. On the basis of λ_C, we define and formalize the *hardware value flow analysis* (HVFA) problem, which underpins our static analyses for critical Chisel verification tasks, including bug detection and security analysis. We then propose a synchronized fixed-point solution to the HVFA problem, featuring hardware-specific treatment of the synchronous behavior of clock-driven hardware registers—the essential feature of Chisel programs. We further prove key theorems establishing the guarantees and limitations of our solution.

As a proof of concept, we develop ChiSA (30K+ LoC)—the first Chisel static analyzer that can analyze intricate hardware value flows to enable lightweight analyses for critical Chisel verification tasks such as bug detection and security analysis. To facilitate thorough evaluation of both ChiSA and future work, we provide ChiSABench (11M+ LoC), a comprehensive benchmark for Chisel static analysis.

Our evaluation on ChiSABench demonstrates that ChiSA offers an effective and significantly more lightweight approach for critical Chisel verification tasks, especially on large and complex real-world designs. For example, ChiSA identified 69 violable developer-inserted assertions in large-scale Chisel designs (9.7M+ LoC) in under 200 seconds—eight of which were recognized by developers and scheduled for future fixes—and detected all 60 information-leak vulnerabilities in the well-known TrustHub benchmark (1.1M+ LoC) in just one second—outperforming state-of-the-art Chisel approaches like ChiselTest's bounded model checking and ChiselFlow's secure type system. These results underscore the high promise of static analysis for lightweight Chisel verification. To encourage continued research and innovation, we will fully open-source ChiSA (30K+ LoC) and ChiSABench (11M+ LoC).

CCS Concepts: • **Theory of computation → Program analysis**; • **Hardware → Hardware description languages and compilation**.

Additional Key Words and Phrases: Static Analysis, Hardware, Chisel

*Corresponding author.

Authors' Contact Information: Jiacai Cui, jiacaicui@smail.nju.edu.cn; Qinlin Chen, qinlinchen@smail.nju.edu.cn; Zhongsheng Zhan, yahya_chan@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Tian Tan, tiantan@nju.edu.cn; Yue Li, yueli@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China.

ACM 2475-1421/2026/1-ART18
https://doi.org/10.1145/3776660

ACM Reference Format:
Jiacai Cui, Qinlin Chen, Zhongsheng Zhan, Tian Tan, and Yue Li. 2026. ChiSA: Static Analysis for Lightweight Chisel Verification. *Proc. ACM Program. Lang.* 10, POPL, Article 18 (January 2026), 33 pages. https://doi.org/10.1145/3776660

1 Introduction

Inherent inefficiencies in general-purpose processors are driving a shift toward domain-specific architectures (DSAs), intensifying the need for more productive hardware development workflows [53, 54, 70]. This creates opportunities for programming language (PL) techniques to play a critical role in hardware development [113]. To meet this demand, Chisel [17]—a leading agile hardware description language (HDL)—leverages modern PL features to improve hardware design productivity, and has seen successful adoption in both academia [22, 77, 119] and industry [8, 15].

While Chisel improves hardware *design* productivity, *verification* remains a major bottleneck, typically requiring substantial time and manual effort. Hardware projects often spend over 70% of development time on verification [51], employ more verification engineers than design engineers, and even require designers to devote nearly half of their time to verification tasks [49].

Current approaches to Chisel verification largely inherit heavyweight methodologies from the broader hardware community, which fall into three main categories, each with efficiency limitations:

(1) *Simulation-based testing* [19, 37, 41, 43, 44, 67, 68, 101] remains the predominant hardware verification technique, but it is fundamentally constrained by the fact that simulation is slow—orders of magnitude slower (10^3–$10^6\times$) than real-time hardware execution [45, 55, 78, 116].
(2) *Formal verification* techniques—including bounded model checking [42, 73, 117, 121], which suffer from the well-known state explosion problem [30], and theorem-proving [46], which require labor-intensive construction of formal proofs—are typically only applied to small-scale, critical modules since they do not scale to million-line-scale designs, let alone support rapid development cycles at that scale.
(3) *Secure type systems* [39, 40, 47] extend Chisel's standard type system with security labels to enforce security policies via type checking. However, their practical adoption is hindered by the extensive manual annotation burden [91, 92], which grows proportionally with code size.

To address these limitations, we advocate for the use of (sophisticated) static analysis [112]—a technique proven well-suited to agile development workflows [34], with established success in software bug detection [120] and security analysis [29]—for lightweight Chisel verification.

Note that although Chisel currently relies on Verilog [1] as a backend to maintain compatibility with *commercial* electronic design automation (EDA) toolchains, we *intentionally* base our work natively on Chisel rather than on its generated, flattened, low-level Verilog. This choice preserves high-level Chisel-specific information that is lost during standard translation to Verilog—such as assertions expressing design intent, high-level memory abstractions amenable to specialized treatment, and source locations essential for traceability. As further elaborated and discussed in Section 6, retaining this information facilitates more effective static analysis that is better tailored to Chisel. This Chisel-native approach also aligns with ongoing efforts in the Chisel community to develop *open-source* EDA workflows that increasingly avoid reliance on Verilog, including Chisel-native simulation [19, 68], waveform viewing [80], synthesis [24, 93, 114], and design space exploration (DSE) [48].

In this work, we establish a theoretical foundation for Chisel static analysis. At its core is λ_C, a formal core calculus of ChAIR (a Chisel-specific intermediate representation for analysis). λ_C is the first formalism that captures the essence of Chisel while being deliberately minimal to ease rigorous reasoning about static analysis built on λ_C. Below, we briefly introduce ChAIR and λ_C,

then present the hardware value flow analysis (HVFA) problem defined atop λ_C, which forms the foundation of our analyses for Chisel verification tasks like bug detection and security analysis.

ChAIR. To ground both our theory and practice, we introduce ChAIR, an intermediate representation (IR) for analysis that offers a simple yet expressive abstraction of Chisel hardware designs. In contrast to Firrtl [63]—the official Chisel compiler IR—and its associated CIRCT [74] dialects, which adopt recursive IR structures based on abstract syntax trees (ASTs) or MLIR [75] operations and are primarily designed for transformation and lowering tasks, ChAIR employs a flat, linear three-address code (3AC) structure that prioritizes simplicity to ease the development of static analyses [97]. To further enable efficient sparse analysis, ChAIR adopts static single assignment (SSA) form [122], which also closely aligns with the structural essence of digital circuits, as discussed in Section 2.2. Due to space limits, the full specification of ChAIR, which captures comprehensive Chisel language features, is provided in the supplementary material accompanying this paper.

λ_C. To support rigorous reasoning about static analyses for Chisel, we formalize a core calculus for ChAIR, denoted λ_C (Section 2). Unlike heavyweight formalisms developed for other HDLs such as Verilog [25] and VHDL [60], which aim to comprehensively characterize full language specifications, λ_C is the first formalism that captures the essence of Chisel while being deliberately minimal—making it well-suited for formal reasoning about static analysis [61, 71, 72, 94]. Since Chisel specifically focuses on describing synchronous digital circuits [103], λ_C is tailored to minimally capture the essence of such circuits. Among various aspects of Chisel that λ_C exposes, we highlight the synchronous behavior of hardware registers—driven by clock ticks—as a key semantic distinction from conventional software languages and a central concern for Chisel static analysis. Furthermore, we prove key properties of λ_C that reflect physical realities of synchronous digital circuits, such as the correspondence between combinational loops and circuit instability [33], which in turn offer a form of retrospective validation for λ_C's design.

HVFA. On the basis of λ_C, we define and formalize the HVFA problem (Section 3). Owing to λ_C's ability to capture the essence of Chisel, HVFA underpins our static analyses of critical Chisel verification tasks—including bug detection and security analysis. HVFA draws inspiration from classical data/value flow analysis in software [66], but incorporates hardware-specific customizations to handle the synchronous semantics of clock-driven hardware registers (the essential feature of Chisel programs) —a fundamental departure from conventional software behavior. In particular, we introduce synchronous flow functions to approximate simultaneous register updates synchronized by clock ticks, and define a synchronized fixed-point solution to statically over-approximate dynamic synchronous circuit behavior. We formally characterize these hardware-specific customizations by proving theorems that establish a theoretical foundation for HVFA's guarantees and limitations. The formal study of HVFA also illustrates how λ_C supports rigorous reasoning about Chisel static analyses.

Proof of Concept. We developed ChiSA (30K+ LoC), the first Chisel static analyzer capable of analyzing intricate hardware value flows to enable efficient yet sophisticated analyses for critical Chisel verification tasks, such as bug detection and security analysis. Note that a significant portion of ChiSA's codebase is dedicated to constructing reusable infrastructure to support ongoing research and the development of new Chisel static analyses. This infrastructure includes, but is not limited to: (1) various HVFAs (a set of fundamental analyses for building value flows for hardware), (2) ChAIR, along with a front-end that automatically translates Chisel code to ChAIR, (3) a rich set of useful graph representations built on top of ChAIR, and (4) an analysis manager for orchestrating multiple analyses and integrating new ones.

Evaluation. To support thorough evaluation for ChiSA, we provide ChiSABench (11M+ LoC), a comprehensive Chisel static analysis benchmark suite spanning a broad range of language features, design purposes, and code scales. To enhance ChiSABench's out-of-the-box accessibility and hands-on usability for future research, we invested considerable effort to pre-elaborate all designs (hardware programs) in ChiSABench into standalone Firrtl [63] files, eliminating the need for tedious environment setup or project-specific build steps, including managing multi-language dependencies and resolving fragile toolchain version conflicts.

To investigate whether ChiSA provides an effective and more lightweight approach for critical Chisel verification tasks—particularly for large and complex real-world Chisel designs—and to assess its fundamental ability to analyze hardware value flows, we evaluate ChiSA's representative analyses on ChiSABench. The results are highly promising:

(1) Hardware Bug Detection (ChiSA vs. Bounded Model Checking). ChiSA's static assertion analysis, identified 69 violable embedded (developer-inserted) assertions—eight of which were recognized by developers and scheduled for future fixes—across large-scale real-world Chisel designs (9.7M+ LoC) in just 200 seconds, illustrating its effectiveness and lightweight nature. In contrast, the state-of-the-art Chisel bounded model checking provided by ChiselTest [73] (referred to as ChiselTest-BMC) failed to analyze any of these designs due to its limitations under real-world conditions. For example, the most common failure occurred when encountering external hardware modules whose definitions were inaccessible—a typical scenario in hardware development given the widespread use of intellectual property (IP) cores. Unlike ChiselTest-BMC, which crashes in this scenario, ChiSA handles such cases by supporting incomplete analysis via conservative approximation of external components, underscoring the practical advantages of static analysis.

To further validate ChiSA's lightweight nature, we additionally evaluated both tools on 877 small-scale, simpler Chisel designs (average 256 LoC), where statistics for ChiselTest-BMC could be obtained. ChiSA completed the analysis of all these designs in just 3 seconds, a significant reduction compared to the 2776 seconds required by ChiselTest-BMC.

(2) Hardware Security Analysis (ChiSA vs. Secure Type System). ChiSA's taint analysis detected all 18 vulnerabilities in ChiselFlow's [47] microbenchmark (655 LoC) using only 44 coarse-grained source/sink annotations—a substantial reduction in annotation burden compared to the 228 fine-grained type annotations needed by ChiselFlow, demonstrating more lightweight manual effort while maintaining comparable effectiveness.

On the much larger TrustHub benchmark [102, 104] (1.15M LoC), ChiSA identified all 60 information-leak vulnerabilities using only 85 source/sink annotations guided by TrustHub's documentation. In contrast, ChiselFlow could not be applied to TrustHub due to the impracticality of retrofitting a million-line-scale codebase with an annotation-intensive type system whose manual effort grows proportionally with code size.

In summary, this work makes the following contributions:

- We present λ_C, the formal core calculus of our Chisel-specific intermediate representation for analysis. λ_C is the first formalism that captures the essence of Chisel while being deliberately minimal to ease rigorous reasoning about static analysis built atop it.
- Based on λ_C, we define and formalize the hardware value flow analysis (HVFA) problem, which adapts classical data/value flow analysis from software to hardware by incorporating hardware-specific treatment of synchronous semantics of clock-driven registers—the essential feature of Chisel. We further prove key theorems establishing HVFA's guarantees and limitations.
- As a proof of concept for HVFA, we developed ChiSA (30K+ LoC), the first Chisel static analyzer capable of analyzing intricate hardware value flows to enable efficient yet sophisticated analyses for critical Chisel verification tasks, such as bug detection and security analysis.

- We provide ChiSABench (11M+ LoC), a comprehensive benchmark suite for evaluating Chisel static analyses. Our evaluation demonstrates that ChiSA offers an effective and highly lightweight solution for critical Chisel verification tasks, outperforming state-of-the-art Chisel techniques (such as bounded model checking and secure type system) on large, complex, real-world designs.
- We will submit an artifact to reproduce all experimental results in the paper, and fully open-source both ChiSA (30K+ LoC) and ChiSABench (11M+ LoC) to the community.

Although ChiSA is still in its early stages and under active development for additional client analyses, we believe it demonstrates the potential of static analysis for lightweight hardware verification. We hope this work contributes to future innovations in Chisel static analysis and inspires broader applications of programming language techniques to hardware verification.

2 λ_C: The Core Calculus of ChAIR

To support rigorous reasoning about static analyses for Chisel, we formalize a core calculus for ChAIR, called λ_C. In contrast to prior formalisms for other hardware description languages (HDLs), such as Verilog [25] and VHDL [60], which aim to comprehensively characterize full language specifications and often become heavyweight, λ_C is the first formalism that captures the essence of Chisel while being deliberately minimal. This makes it well-suited for formally defining and reasoning about static analyses [61, 71, 72, 94], as we illustrate in Section 3.

Since Chisel specifically focuses on describing synchronous digital circuits [103]—with its idioms and libraries assuming a single global clock domain by default—we accordingly concentrate our discussion on synchronous circuits[1]. While λ_C targets a minimal core of Chisel to enable clear and tractable formal reasoning, ChAIR itself supports comprehensive Chisel language features, as detailed in our supplementary material.

To clarify the relationship among source-level Chisel, ChAIR, and λ_C: (1) Source-level Chisel designs can be comprehensively compiled into ChAIR through ChiSA's frontend, a process we have thoroughly validated on ChiSABench (11M+ LoC). (2) ChAIR, in turn, can be fully desugared into λ_C, with the only exceptional case being gated clocks [31], a rarely used case in Chisel—cross all real-world projects in ChiSABench, we only observed gated clocks in roughly 1 out of every 1,000,000 lines. We deliberately exclude this rare case to keep λ_C as clean and minimal as we can without compromising its practical adequacy as a theoretical core calculus for ChAIR.

In this section, we begin with an example to informally introduce λ_C and highlight the essence of Chisel, while also providing necessary hardware background (Section 2.1). We then formalize the structural essence of circuits—digital components and their interconnections—via the syntax of λ_C (Section 2.2). Next, we formalize the behavioral essence—namely, reactivity and synchronicity—through the operational semantics of λ_C (Section 2.3). Finally, in Section 2.4, we establish key circuit properties in λ_C that reflect real-world physical characteristics, such as the correspondence between combinational loops and circuit instability [33], which in turn offer a form of retrospective validation for its design.

2.1 λ_C Informal: Understanding the Essence of Chisel Circuits

This section provides an informal introduction to λ_C using the accumulator example in Figure 1 to highlight the essence of Chisel circuits it captures, focusing on both the static *structure* and the dynamic *behavior* of these circuits. Relevant hardware background is included as needed within the section.

From a static structural perspective, a circuit consists of digital *components* and the *connections* between them. In λ_C, components are modeled using operators—$y\ op\ z$ for arithmetic (e.g., adders)

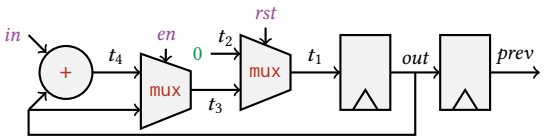
[1]Unless otherwise noted, all circuits discussed in the formal sections (Section 2 and Section 3) are assumed to be synchronous.

```
val out = RegInit(0)
val prev = RegNext(out)
when (en) {
    out := out + in
}
```

(a) Chisel Code

$$
\begin{aligned}
out &\Leftarrow t_1 \\
t_1 &= \mathtt{mux}(rst, t_2, t_3) \\
t_2 &= 0 \\
t_3 &= \mathtt{mux}(en, t_4, out) \\
t_4 &= out + in \\
prev &\Leftarrow out
\end{aligned}
$$

(b) λ_C Representation



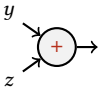
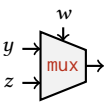
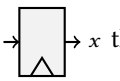
(c) Schematic

λ_C Reduction Rule	External Stimulus	Internal Connection	*in*	*en*	*rst*	*out*	*prev*	t_1	t_2	t_3	t_4
C-Eval	–	$t_4 = out + in$	0	0	0	0	0	0	0	0	0
C-Eval	–	$t_3 = \mathtt{mux}(en, t_4, out)$	0	0	0	0	0	0	0	0	0
C-Eval	–	$t_2 = 0$	0	0	0	0	0	0	0	0	0
C-Eval	–	$t_1 = \mathtt{mux}(rst, t_2, t_3)$	0	0	0	0	0	0	0	0	0
S-Poke	poke *en* 1	–	0	1	0	0	0	0	0	0	0
C-Eval	–	$t_3 = \mathtt{mux}(en, t_4, out)$	0	1	0	0	0	0	0	0	0
S-Poke	poke *in* 2	–	2	1	0	0	0	0	0	0	0
C-Eval	–	$t_4 = out + in$	2	1	0	0	0	0	0	0	2
C-Eval	–	$t_3 = \mathtt{mux}(en, t_4, out)$	2	1	0	0	0	0	0	2	2
C-Eval	–	$t_1 = \mathtt{mux}(rst, t_2, t_3)$	2	1	0	0	0	2	0	2	2
S-Tick	tick	$out \Leftarrow t_1 \parallel prev \Leftarrow out$	2	1	0	2	0	2	0	2	2
C-Eval	–	$t_4 = out + in$	2	1	0	2	0	2	0	2	4
C-Eval	–	$t_3 = \mathtt{mux}(en, t_4, out)$	2	1	0	2	0	2	0	4	4
C-Eval	–	$t_1 = \mathtt{mux}(rst, t_2, t_3)$	2	1	0	2	0	4	0	4	4
S-Tick	tick	$out \Leftarrow t_1 \parallel prev \Leftarrow out$	2	1	0	4	2	4	0	4	4
C-Eval	–	$t_4 = out + in$	2	1	0	4	2	4	0	4	6
C-Eval	–	$t_3 = \mathtt{mux}(en, t_4, out)$	2	1	0	4	2	4	0	6	6
C-Eval	–	$t_1 = \mathtt{mux}(rst, t_2, t_3)$	2	1	0	4	2	6	0	6	6

(d) An execution trace of the accumulator, based on the reduction rules from Definition 2.14 and formally described in Section 2.3. The external stimulus sequence is poke *en* 1; poke *in* 2; `tick`; `tick`, with the relevant stimulus for each step shown in the "External Stimulus" column. The "Internal Connection" column indicates the currently active internal connection during each step. The remaining columns display variable values after each step; values changed in the current step are highlighted in red, while values touched by the reduced connection but unchanged are shown in green.

Fig. 1. An informal introductory example of λ_C. The Chisel code, λ_C representation, and schematic describe the same accumulator design, where *out* accumulates *in* when *en* is high, and resets to 0 when *rst* is high; *prev* records the previous *out*. Unfocused Chisel boilerplate (e.g., module declaration, literal type conversion) is omitted for clarity.

or logical (e.g., shifters) units and $\mathtt{mux}(w, y, z)$ for multiplexers—as well as constants (1 for voltage high/power, 0 for voltage low/ground). Connections are expressed using assignment statements: $x = \cdots$ for *wire connections*, and $x \Leftarrow \cdots$ for *register connections*. For example:

- $\boxed{x = y + z}$ models an adder  that reactively updates x with the sum of y and z.
- $\boxed{x = \mathtt{mux}(w, y, z)}$ models a multiplexer  that reactively sets x to y if w is nonzero, or to z otherwise.
- $\boxed{x \Leftarrow y}$ models a register  that synchronously updates x with the value of y on each global clock tick, in parallel with all other registers.

The correspondence between Figure 1b and Figure 1c illustrates how λ_C uses these simple, composable primitives to describe a real functional hardware design. Comparing Figure 1a and Figure 1b,

we see that high-level Chisel constructs can be desugared into λ_C to explicitly expose structural details. For example, the `RegInit` method encapsulates reset logic; the `when` statement desugars to a multiplexer.

From a dynamic behavior perspective, circuit execution manifests as voltage changes at each circuit location in response to external stimuli. λ_C provides two primitives for external stimuli, corresponding to the two types of external stimuli supported by Chisel's official simulator [14]:

- `poke` x n sets the value of input variable x to n, modeling physical actions like toggling a switch.
- `tick` advances the global clock, triggering synchronous updates for all registers.

Figure 1d illustrates how the accumulator in Figure 1b behaves under the stimulus sequence `poke` *en* 1; `poke` *in* 2; `tick`; `tick`, assuming all variables are zero by default, consistent with the default behavior of Chisel's simulator [14]. Initially, wire connections are active even before any external stimulus is applied, reflecting the physical reality that combinational components respond instantaneously when the circuit is powered up. Then, the execution trace highlights two fundamental principles of circuit behavior:

(1) *Reactivity*: Wire connections (e.g., $t_4 = out + in$) propagate changes reactively—any update to a right-hand side (RHS) expression is immediately reflected in the left-hand side (LHS) variable.
(2) *Synchronicity*: Register connections (e.g., $out \Leftarrow t_1 \parallel prev \Leftarrow out$) are updated synchronously on each clock tick. Here, we use the symbol $\parallel$ to combine register connections, indicating that the combined register connections are updated in synchrony.

As illustrated in Figure 1c, each *variable* in λ_C corresponds to a distinct *location* in the circuit. Thus, value changes of a variable represent voltage changes at a physical location in the design—capturing the essence of circuit behavior from a programming language perspective.

2.2 λ_C Syntax: Circuit (Static) Structure

The formal syntax of λ_C is intuitive in light of the introductory example in Section 2.1.

Definition 2.1 (λ_C Syntax for Circuit Design). A λ_C circuit C is defined as:

$$
\begin{array}{rlcl|rlcl}
C & \in \textbf{Circuit} & := & \mathcal{P}(\textbf{Item}) & \iota & \in \textbf{Item} & := & x = n \\
op & \in \textbf{Op} & := & \{(\text{operators})\} & & & | & x = y\ op\ z \\
w, x, y, z & \in \textbf{Var} & := & \{(\text{symbols})\} & & & | & x = \mathsf{mux}(w, y, z) \\
n & \in \textbf{Value} & = & \mathbb{Z} & & & | & x \Leftarrow y
\end{array}
$$

As explained in Section 2.1, items ι capture the essential structure of circuits: components and their connections. Expressions on the right-hand side of $x = \cdots$ model circuit components. Assignments of the form $x = \cdots$ representing wire connections and $x \Leftarrow \cdots$ representing register connections. We have aimed to keep λ_C as minimal as possible; the current version reflects our best-effort design.

We choose **Value** $= \mathbb{Z}$ (i.e., the set of mathematical integers) purely for ease of understanding and presentation in this paper. This simplification does not impact the generality of our formalization, discussion, or proof sketches. A more rigorous and detailed treatment is provided in the supplementary material.

We now introduce several auxiliary notations. Throughout, we use ":=" to denote "is defined as".

*Definition 2.2 (*Def *&* Use *Selector).* $\mathsf{Def} : \textbf{Item} \to \textbf{Var}$ and $\mathsf{Use} : \textbf{Item} \to \mathcal{P}(\textbf{Var})$ are defined as:

$$
\begin{array}{llll}
\mathsf{Def}(x = n) := x & \mathsf{Def}(x = y\ op\ z) := x & \mathsf{Def}(x = \mathsf{mux}(w, y, z)) := x & \mathsf{Def}(x \Leftarrow y) := x \\
\mathsf{Use}(x = n) := \emptyset & \mathsf{Use}(x = y\ op\ z) := \{y, z\} & \mathsf{Use}(x = \mathsf{mux}(w, y, z)) := \{w, y, z\} & \mathsf{Use}(x \Leftarrow y) := \{y\}
\end{array}
$$

Definition 2.3 (Wire & Register Connection). For a circuit C , $C_{=} := \{\_ = \_ \in C\}$ denotes its wire connections and $C_{\Leftarrow} := \{\_ \Leftarrow \_ \in C\}$ denotes its register connections, where _ is a wildcard.

Definition 2.4 (Input Variable). A variable x is an input of a circuit C if and only if it is used but not defined in the circuit. Formally, $x \in \mathbf{Input}_C$ iff. $\exists\, \iota \in C.\ x \in \mathsf{Use}(\iota) \wedge \neg(\exists\, \iota \in C.\ \mathsf{Def}(\iota) = x)$ where $\mathbf{Input}_C$ denotes the set of all input variables of circuit C.

As introduced in Section 2.1, only input variables can be externally poked.

Definition 2.5 (λ_C Syntax for External Stimuli). The external stimuli (a sequence of external stimulus) S of a circuit C is defined as $S \in \mathbf{Stimulus}_C := (\mathtt{poke}\ x\ n \mid \mathtt{tick})^*$ where $x \in \mathbf{Input}_C$.

Since each physical location in a circuit should only be driven by at most one component—an invariant also enforced in Chisel—λ_C naturally adheres to the single static assignment (SSA) form.

Axiom 2.1 (Single Driver). *λ_C is SSA:* $\forall C \in \mathbf{Circuit}.\forall \iota_1, \iota_2 \in C.\mathsf{Def}(\iota_1) = \mathsf{Def}(\iota_2) \Rightarrow \iota_1 = \iota_2$.

To facilitate the development of static analysis algorithms, we define a *value flow graph* (VFG) representation of the circuit structure.

Definition 2.6 (Value Flow Graph for λ_C Hardware Design). The value flow graph of a circuit C is defined as $G_C = \langle C, E\rangle$, where $E = \{\langle \iota_1, \iota_2\rangle \in C \times C \mid \mathsf{Def}(\iota_1) \in \mathsf{Use}(\iota_2)\}$. We define the predecessor and successor selectors as $\mathsf{Pred}(\iota) := \{\iota' \mid \langle \iota', \iota\rangle \in E\}$ and $\mathsf{Succ}(\iota) := \{\iota' \mid \langle \iota, \iota'\rangle \in E\}$.

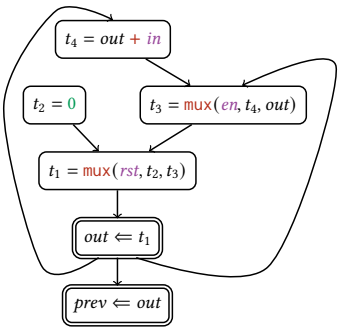


Fig. 2. Value flow graph of the introductory example in Figure 1.

Figure 2 shows the VFG for the accumulator in Figure 1b, which serves as an abstract static-analysis-friendly view of the circuit schematic in Figure 1c. Register connections—whose values update synchronously on each clock tick—are depicted with double-line rectangles to distinguish them from reactive wire connections.

2.3 λ_C Semantics: Circuit (Dynamic) Behavior

This section formalizes the operational behavior of circuits in λ_C, highlighting two core behavioral features of hardware: *reactivity* and *synchronicity*.

We begin by defining the evaluation of expressions within an evaluation environment.

Definition 2.7 (Environment). An environment $E : \mathbf{Var} \to \mathbf{Value}$ maps variables to values. The domain of environments, representing the set of all possible environments, is denoted by **Env**.

Definition 2.8 (Expression). An expression e is any right-hand side appearing in an item; that is:

$$e := n \mid y\ op\ z \mid \mathsf{mux}(w, y, z) \mid y$$

Definition 2.9 (Evaluation). $[\![e]\!]_E$, defined case by case as follows, denotes the value of expression e evaluated in environment E:

$$[\![n]\!]_E := n \quad [\![y\ op\ z]\!]_E := E(y)\ op\ E(z) \quad [\![\mathsf{mux}(w, y, z)]\!]_E := \begin{cases} E(y), & \text{if } E(w) \neq 0 \\ E(z), & \text{if } E(w) = 0 \end{cases} \qquad [\![y]\!]_E := E(y)$$

Items are the core elements of λ_C that govern a circuit's dynamic behavior in response to changes in the environment. Wire connections make the circuit *reactive*: whenever the value of the RHS expression of a wire connection changes, the LHS variable is updated immediately. The following *activation function* (Definition 2.10) specifies, for any change in the environment, which set of wire

connections should update their LHS values. Register connections make the circuit *synchronous*: all left-hand sides of register connections are updated simultaneously at each clock tick. The *synchronization function* (Definition 2.13) describes how the environment is updated on a clock tick. These two utility functions are central to the operational semantics of λ_C.

Definition 2.10 (Activation Function). Given a circuit C, its activation function $\mathcal{A}_C : \mathbf{Env} \times \mathbf{Env} \to \mathcal{P}(C_=)$ is defined as

$$\mathcal{A}_C(E, E') := \{\iota \in C_= \mid \exists x \in \mathsf{Use}(\iota).E'(x) \neq E(x)\}$$

It computes the set of wire connections activated by an environment change from E to E'.

Definition 2.11 (Atomic Effect). The effect of an item ι on the environment is a function $[\![\iota]\!] : \mathbf{Env} \to \mathbf{Env}$, defined as $[\![x(=|\Leftarrow)e]\!](E) := E[x \mapsto [\![e]\!]_E]$, where $(=|\Leftarrow)$ denotes either $=$ or $\Leftarrow$.

Definition 2.12 (Synchronous Effect). For a set of items $I = \{\iota_1, \ldots, \iota_n\}$, where each ι_i has the form $x_i(=|\Leftarrow)e_i$, we define their synchronous effect $[\![I]\!] : \mathbf{Env} \to \mathbf{Env}$ as

$$[\![I]\!](E) := E[x_1 \mapsto [\![e_1]\!]_E, \ldots, x_n \mapsto [\![e_n]\!]_E],$$

In the following discussion, for convenience, we often write $[\![I]\!]$ in decompositional form as $[\![\iota_1 \parallel \ldots \parallel \iota_n]\!]$ to emphasize its synchronous nature.

Definition 2.13 (Synchronization Function). The synchronization function of a circuit C is $[\![C_\Leftarrow]\!] : \mathbf{Env} \to \mathbf{Env}$ (an instance of Definition 2.12). It represents the effect that a clock tick has on the environment.

We now define the full operational semantics of λ_C, orchestrating the above definitions into a simple and unified circuit execution model.

Definition 2.14 (Operational Semantics of λ_C). The semantics of a circuit C is given as a reduction relation over configurations $\mathbf{Config}_C := \mathbf{Stimulus}_C \times \mathbf{Env} \times \mathcal{P}(C_=)$.

A configuration $\langle S, E, I\rangle \in \mathbf{Config}_C$ consists of: (1) S, the remaining external stimulus sequence; (2) E, the current environment; (3) I, the set of pending internal wire connections to take effect.

We write $C \vdash \langle S, E, I\rangle \rightsquigarrow \langle S', E', I'\rangle$ to denote a one-step reduction. The reduction relation $C \vdash \rightsquigarrow$ is defined as follows, where c_0 is the initial configuration under external stimulus S_0.

S-Poke
$$\frac{E' = [\![x = n]\!](E)}{C \vdash \langle \mathsf{poke}\ x\ n; S, E, \emptyset\rangle \rightsquigarrow \langle S, E', \mathcal{A}_C(E, E')\rangle}$$

S-Tick
$$\frac{E' = [\![C_\Leftarrow]\!](E)}{C \vdash \langle \mathtt{tick}; S, E, \emptyset\rangle \rightsquigarrow \langle S, E', \mathcal{A}_C(E, E')\rangle}$$

C-Eval
$$\frac{\iota \in I \qquad E' = [\![\iota]\!](E)}{C \vdash \langle S, E, I\rangle \rightsquigarrow \langle S, E', (I - \{\iota\}) \cup \mathcal{A}_C(E, E')\rangle}$$

$c_0 = \langle S_0, E_{\text{rand}}, C_=\rangle$, where E_{rand} maps each variable to a random value

Intuitively, each reduction step corresponds to either an external stimulus—such as pokes (S-Poke) and clock ticks (S-Tick)—or an internal circuit update (C-Eval). The rules ensure the following: register updates are synchronized with clock ticks (S-Tick); input changes are applied through pokes (S-Poke); and wire-driven updates propagate reactively (C-Eval). Together, these rules define the dynamic behavior of a synchronous hardware design under external stimuli. The execution trace in Figure 1d corresponds to a sequence of such reduction steps beginning from an initial configuration $\langle \mathsf{poke}\ \mathit{en}\ 1; \mathsf{poke}\ \mathit{in}\ 2; \mathtt{tick}; \mathtt{tick}, \{\_ \mapsto 0\}, \{t_2 = 0\}\rangle$, with each row in the table representing a single reduction step and listing the rule applied in the first column.

The reader may notice that the reduction steps can be infinite, as the size of I in the configuration may never decrease. We will soon explain the physical meaning of this phenomenon and demonstrate how it justifies the design of λ_C in Section 2.4.

We set the initial configuration c_0 to $\langle S_0, E_{\text{rand}}, C_{=}\rangle$. Here, S_0 represents the initial sequence of external stimuli in its entirety. The initial environment, E_{rand}, assigns random values to all variables, reflecting the physical reality that initial voltages at circuit locations are unpredictable. Wire connections, denoted as $C_{=}$, are included in the initial configuration c_0, capturing the fact that combinational components respond instantaneously once the circuit is powered up.

2.4 λ_C Properties: Circuit Characteristics

In this section, we formally establish several key properties of λ_C. These properties are chosen to demonstrate that λ_C accurately models the characteristic behaviors of real hardware circuits. Our aim is to justify the design of λ_C and to show that it can effectively support the study of Chisel circuits using programming language techniques.

The theorems discussed in this section concern the reduction sequences of λ_C configurations. As noted in Section 2.3, where we defined the reduction rules of λ_C, the reader may observe that the reduction steps can be infinite, even when the initial configuration contains only a finite sequence of external stimuli. This is because the size of I in the configuration does not necessarily decrease under all reduction rules. In particular, rule C-Eval removes ι from I but simultaneously merges a new set $\mathcal{A}_c(E, E')$ into it. At first glance, this may seem like a flaw in the design of λ_C. However, this behavior precisely models a specific phenomenon in circuits: oscillation, where the voltage levels on wires continue to change indefinitely, even after external stimuli have ceased. Theorem 2.1 expresses this issue both in the language of λ_C and in its corresponding physical interpretation.

Theorem 2.1. *A circuit could oscillate even if only a finite sequence of external stimuli is applied:*

$$\exists C \in \textbf{Circuit}.\exists c \in \textbf{Config}_C.\exists \textit{an infinite reduction sequence from } c.$$

Proof Sketch. We prove by constructing such a circuit: $x = y + 1, y = x + 1$. This circuit exhibits an infinite reduction sequence:

$$\langle \mathsf{poke}\ x\ 1, \{\_ \mapsto 0\}, \emptyset\rangle \rightsquigarrow \langle \epsilon, \{x \mapsto 1, y \mapsto 0\}, \{y = x + 1\}\rangle \rightsquigarrow \langle \epsilon, \{x \mapsto 1, y \mapsto 2\}, \{x = y + 1\}\rangle$$
$$\rightsquigarrow \langle \epsilon, \{x \mapsto 3, y \mapsto 2\}, \{y = x + 1\}\rangle \rightsquigarrow \langle \epsilon, \{x \mapsto 3, y \mapsto 4\}, \{x = y + 1\}\rangle \rightsquigarrow \ldots \qquad \square$$

Moreover, Intel's design guidelines [33] explicitly recommend avoiding such oscillations whenever possible. This raises an important question: how can we formulate a condition on λ_C programs that guarantees the absence of infinite reduction sequences? Theorem 2.2 shows that a circuit whose value flow graph contains no cycles composed entirely of wire connections cannot have an infinite reduction sequence. These cycles correspond exactly to the so-called "combinational loops" in hardware engineering, which should be avoided according to best practices.

Definition 2.15 (Combinational Loop). A combinational loop in circuit C is a cycle composed entirely of wire connections in its value flow graph G_C.

Theorem 2.2. *A circuit without a combinational loop cannot oscillate:*

If C contains no combinational loop, then $\nexists$ infinite reduction sequence from any $c \in \textbf{Config}_C$.

Proof Sketch. Since C has no combinational loop, the subgraph of G_C induced by $C_{=}$—denoted $G_C[C_{=}]$—is a directed acyclic graph (DAG). Let $m = |C_{=}|$ and define a reverse topological ranking $r : C_{=} \to \{1, \ldots, m\}$ such that $r(\iota) = i$ implies that ι is the i-th largest node in the topological ordering of $G_C[C_{=}]$.

Now define a potential function $\varphi \in \mathbf{Config}_C \to \mathbb{N}^{m+1}$ as $\varphi(\langle S, E, I\rangle) = \langle |S|, s_1, \ldots, s_m\rangle$, where the s_i's are the ranks of the elements of I, sorted in descending order and padded with zeros to length m. The lexicographic order over $\mathbb{N}^{m+1}$ is a well-order, and there exists no infinite strictly decreasing sequence in a well-ordered set. Therefore, to prove the absence of infinite reduction sequences, it suffices to show that $C \vdash c \rightsquigarrow c' \Rightarrow \varphi(c) > \varphi(c')$.

The S-Poke and S-Tick rules trivially decrease φ due to $|S|$ decreasing. For the non-trivial case C-Eval: suppose $C \vdash \langle S, E, I\rangle \rightsquigarrow \langle S, E', I'\rangle$ where $E' = [\![\iota]\!](E)$ and $I' = (I - \{\iota\}) \cup \mathcal{A}_C(E, E')$. Since $\mathcal{A}_C(E, E') \subseteq \mathrm{Succ}(\iota) \cap C_=$ (by Definition 2.6 and Definition 2.10) and $\iota' \in \mathrm{Succ}(\iota) \Rightarrow r(\iota') < r(\iota)$ (by definition of r), all newly added elements in I' are ranked strictly lower than ι. Hence $\varphi(c')$ must be lexicographically smaller than $\varphi(c)$, completing the proof. □

To conclude the discussion, we present one more theorem (Theorem 2.3), which can be interpreted as stating that a circuit without combinational loops has a uniquely determined steady state for a given sequence of external stimuli, aligning with our understanding of physical circuits. This further justifies the design of λ_C and its semantics.

Theorem 2.3. *The steady-state reached by a circuit without combinational loops is uniquely determined:*

$$C \text{ has no combinational loop} \wedge C \vdash \langle S, E, C_=\rangle \rightsquigarrow^* \langle \epsilon, E_1, \emptyset\rangle \wedge C \vdash \langle S, E, C_=\rangle \rightsquigarrow^* \langle \epsilon, E_2, \emptyset\rangle \Rightarrow E_1 = E_2.$$

Proof Sketch. In the light of Theorem 2.2 and Newman's Lemma [86], it is sufficient to prove the following proposition: if $C \vdash \langle S, E, C_=\rangle \rightsquigarrow^* \langle S_m, E_m, I_m\rangle \wedge C \vdash \langle S_m, E_m, I_m\rangle \rightsquigarrow \langle S_1, E_1, I_1\rangle \wedge C \vdash \langle S_m, E_m, I_m\rangle \rightsquigarrow \langle S_2, E_2, I_2\rangle$, there is a configuration $\langle S', E', I'\rangle$ such that $C \vdash \langle S_1, E_1, I_1\rangle \rightsquigarrow^* \langle S', E', I'\rangle \wedge C \vdash \langle S_2, E_2, I_2\rangle \rightsquigarrow^* \langle S', E', I'\rangle$.

To prove this, it is necessary to analyze the reduction rules. Among all reduction rules, only C-Eval introduces nondeterminism in subsequent reductions, as it allows pending wire connections to be consumed in an arbitrary order. Therefore, to prove that the final environment reduced from a given configuration is uniquely determined, it suffices to show that the nondeterministic evaluation order of C-Eval does not affect the determinism of its resulting environment. Although the result appears intuitive, its rigorous proof is nontrivial and is thus provided in the supplementary material due to space limitations. □

3 HVFA: Hardware Value Flow Analysis

To demonstrate how λ_C facilitates rigorous reasoning about Chisel static analyses, we define and formally study the *hardware value flow analysis* (HVFA) problem based on λ_C. Leveraging λ_C's ability to capture the essence of Chisel, HVFA enables our lightweight analyses for critical Chisel verification tasks, as illustrated in our evaluation (Section 5).

HVFA draws inspiration from classical data/value flow analysis [66] in the software domain, but incorporates hardware-specific customizations: (1) we introduce *synchronous flow functions* (Definition 3.6) to approximate the synchronous behavior of clock-driven hardware registers; and (2) we present the *synchronized fixed-point solution* (Definition 3.11) to derive sound value flow information (e.g., constants, intervals, taints) at each circuit location (i.e., each variable in the hardware design).

This section first formulates the HVFA problem in Section 3.1. Section 3.2 explains how synchronous flow functions are used to approximate the behavior of clock-driven hardware registers, a key hardware-specific distinction from classic data/value flow analysis for software. In Section 3.3, we define the synchronized fixed-point solution and present an intuitive algorithm for computing it. Section 3.4 proves theorems about the conditions that guarantee the soundness and factors

that affect precision. Finally, Section 3.5 introduces representative application-specific instances of HVFA that support our hardware bug detection and security analysis for Chisel designs.

3.1 HVFA Problem Formulation

We formalize the HVFA problem by specifying its input, the form of its output, and the correctness property the output must satisfy.

Input. The input to an HVFA problem includes a *circuit* $C \in \textbf{Circuit}$, a *lattice* $\langle L, \sqsubseteq \rangle$ encoding application-specific knowledge (e.g., constants, intervals, taints), and a description $\delta : \textbf{Input}_C \to L$ about the external stimulus—hereafter referred to as *stimulus description*.

Output. The output of an HVFA is a *store* $\sigma \in \textbf{Store} := \textbf{Var} \to L$, which assigns to each variable—representing a circuit location as illustrated in Figure 1c—a value from the input lattice.

Correctness. The output store σ is deemed correct if it soundly over-approximates all possible runtime configurations that may arise from any execution of the circuit under external stimuli consistent with the input description δ, as formalized in Definition 3.4.

Definition 3.1 (Abstraction Function). We use an abstraction function $\alpha : \textbf{Value} \to L$ to represent the relationship between static abstractions and dynamic values. For brevity without harming clarity, we simply overload α by defining $\alpha : \textbf{Env} \to \textbf{Store}$ as $\alpha(E) := \{x \mapsto \alpha(n) \mid E(x) = n\}$ and $\alpha : \textbf{Config} \to \textbf{Store}$ as $\alpha(\langle \_, E, \_ \rangle) := \alpha(E)$.

Definition 3.2 (Stimulus-Description Consistency). A stimulus $S \in \textbf{Stimulus}_C$ is consistent with a description $\delta : \textbf{Input}_C \to L$ if $\forall \mathsf{poke}\ x\ n \in S.\alpha(n) \sqsubseteq \delta(x)$.

Definition 3.3 (Store Precision Ordering). For $\sigma_1, \sigma_2 \in \textbf{Store}$, we say σ_1 is more precise than σ_2, denoted as $\sigma_1 \sqsubseteq \sigma_2$, if $\forall x.\sigma_1(x) \sqsubseteq \sigma_2(x)$.

Definition 3.4 (Soundness). A store σ is sound if for any dynamic execution trace $C \vdash c_0 \rightsquigarrow c_1 \ldots$ starting from c_0 (Definition 2.14) where S_0 is consistent with the stimulus description δ (Definition 3.2), we have $\forall i \geq 0.\alpha(c_i) \sqsubseteq \sigma$.

Discussion About Stimulus Description. While the soundness property of a concrete HVFA result is defined over external stimuli consistent with the input stimulus description δ, the general HVFA formulation itself is *not stimulus-specific*. For instance, one can define a trivial stimulus description $\delta = \{\_ \mapsto \top\}$, which is *naturally consistent with all possible external stimuli*, since the corresponding consistency condition $\forall \mathsf{poke}\ x\ n \in S.\alpha(n) \sqsubseteq \top$ (Definition 3.2) is *trivially satisfied*. In this case, the soundness property would quantify over all possible external stimuli without imposing any additional constraints from the stimulus description δ.

The purpose of including the stimulus description δ is to allow flexibility in encoding known constraints on external stimuli to improve precision. For example, if we know in advance that for a specific x, $\forall \mathsf{poke}\ x\ n \in S.\alpha(n) \sqsubseteq l$, where $l \in L$ is non-trivial (i.e., $l \sqsubset \top$), we can specify $\delta(x) = l$ instead of the trivial $\delta(x) = \top$. This enables the analysis to incorporate non-trivial prior knowledge, thus enhancing precision.

3.2 Approximate Synchronous Register Behavior: Synchronous Flow Functions

Inspired by classical data/value flow analysis, we use flow functions $f[\![\_]\!] : \textbf{Store} \to \textbf{Store}$ to approximate the dynamic circuit behavior $[\![\_]\!] : \textbf{Env} \to \textbf{Env}$ formalized in Section 2.3. In particular, we define atomic flow functions $f[\![\iota]\!]$ (Definition 3.5) to approximate atomic effects $[\![\iota]\!]$ (Definition 2.11), and synchronous flow functions $f[\![\iota_1 \parallel \iota_2]\!]$ (Definition 3.6) to approximate synchronous effects $[\![\iota_1 \parallel \iota_2]\!]$ (Definition 2.12). This section primarily focuses on synchronous flow

functions—a hardware-specific customization used in Section 3.3 to approximate the dynamic behavior of clock-driven hardware registers.

3.2.1 Atomic Flow Functions.

Definition 3.5 (Atomic Flow Function). The atomic flow function of a single item ι is denoted $f[\![\iota]\!] : \textbf{Store} \to \textbf{Store}$ and defined as $f[\![x(=|\Leftarrow)e]\!](\sigma) := \sigma[x \mapsto [\![e]\!]_\sigma]$, where $[\![e]\!]_\sigma \in L$ represents the lattice value of expression e evaluated under store σ.

Here, the *evaluation strategy* $[\![e]\!]_\sigma$ in each HVFA instance is problem-specific and defined by its developer, as we will see in Example 3.1, Example 3.2, and Section 3.5.

3.2.2 Synchronous Flow Functions.

Definition 3.6 (Synchronous Flow Function). Let $I = \{\iota_1, \ldots, \iota_n\}$ where each item ι_i has the form $x_i(=|\Leftarrow)e_i$, their synchronous flow function $f[\![I]\!](\sigma) : \textbf{Store} \to \textbf{Store}$ is defined as

$$f[\![I]\!](\sigma) := \sigma[x_1 \mapsto [\![e_1]\!]_\sigma, \ldots, x_n \mapsto [\![e_n]\!]_\sigma].$$

In the following discussion, for convenience, we often write $f[\![I]\!]$ in decompositional form as $f[\![\iota_1 \parallel \ldots \parallel \iota_n]\!]$ to emphasize its synchronous nature.

Below we establish that *monotonicity*, *soundness*, and *precision ordering* are preserved under synchronous composition (Lemmas 3.1, 3.2, and 3.3). Thus, to verify these properties for a synchronous flow function $f[\![\iota_1 \parallel \ldots \parallel \iota_n]\!]$, it suffices to verify them for each atomic flow function $f[\![\iota_1]\!], \ldots, f[\![\iota_n]\!]$ individually.

Definition 3.7 (Flow Monotonicity). A flow function $f : \textbf{Store} \to \textbf{Store}$ is monotonic if $\forall \sigma_1 \sqsubseteq \sigma_2 . f(\sigma_1) \sqsubseteq f(\sigma_2)$.

Lemma 3.1 (Monotonicity Preservation). *If $f[\![\iota_1]\!]$ and $f[\![\iota_2]\!]$ are monotonic, then so is $f[\![\iota_1 \parallel \iota_2]\!]$.*

Proof Sketch. Let ι_1 be $x_1(=|\Leftarrow)e_1$, ι_2 be $x_2(=|\Leftarrow)e_2$. Suppose $\sigma_1 \sqsubseteq \sigma_2$. Monotonicity of $f[\![\iota_1]\!]$ implies $[\![e_1]\!]_{\sigma_1} \sqsubseteq [\![e_1]\!]_{\sigma_2}$. Similarly, $[\![e_2]\!]_{\sigma_1} \sqsubseteq [\![e_2]\!]_{\sigma_2}$. Thus $f[\![\iota_1 \parallel \iota_2]\!](\sigma_1) = \sigma_1[x_1 \mapsto [\![e_1]\!]_{\sigma_1}, x_2 \mapsto [\![e_2]\!]_{\sigma_1}] \sqsubseteq \sigma_2[x_1 \mapsto [\![e_1]\!]_{\sigma_2}, x_2 \mapsto [\![e_2]\!]_{\sigma_2}] = f[\![\iota_1 \parallel \iota_2]\!](\sigma_2)$. □

Definition 3.8 (Flow Soundness). An atomic flow function $f[\![\iota]\!]$ is sound if $\alpha(E) \sqsubseteq \sigma \Rightarrow \alpha([\![\iota]\!](E)) \sqsubseteq f[\![\iota]\!](\sigma)$; a synchronous flow function $f[\![I]\!]$ is sound if $\alpha(E) \sqsubseteq \sigma \Rightarrow \alpha([\![I]\!](E)) \sqsubseteq f[\![I]\!](\sigma)$.

Definition 3.8 formalizes the notion that a flow function $f[\![\_]\!]$ statically over-approximates the dynamic behavior $[\![\_]\!]$. Intuitively, if $f[\![\iota]\!]$ is sound, then whenever the store σ over-approximates an environment E, that is, $\alpha(E) \sqsubseteq \sigma$, the static output $f[\![\iota]\!](\sigma)$ must still over-approximate the dynamic result $[\![\iota]\!](E)$, that is, $\alpha([\![\iota]\!](E)) \sqsubseteq f[\![\iota]\!](\sigma)$.

Lemma 3.2 (Soundness Preservation). *If $f[\![\iota_1]\!]$ and $f[\![\iota_2]\!]$ are sound, then so is $f[\![\iota_1 \parallel \iota_2]\!]$.*

Proof Sketch. Let ι_1 be $x_1(=|\Leftarrow)e_1$, ι_2 be $x_2(=|\Leftarrow)e_2$. Suppose $\alpha(E) \sqsubseteq \sigma$. Soundness of ι_1 implies $\alpha([\![e_1]\!]_E) \sqsubseteq [\![e_1]\!]_\sigma$. Similarly, $\alpha([\![e_2]\!]_E) \sqsubseteq [\![e_2]\!]_\sigma$. Hence, $\alpha([\![\iota_1 \parallel \iota_2]\!](E)) = \alpha(E)[x_1 \mapsto \alpha([\![e_1]\!]_E), x_2 \mapsto \alpha([\![e_2]\!]_E)] \sqsubseteq \sigma[x_1 \mapsto [\![e_1]\!]_\sigma, x_2 \mapsto [\![e_2]\!]_\sigma] = f[\![\iota_1 \parallel \iota_2]\!](\sigma)$. □

To facilitate *unified reasoning* about different suites of atomic and synchronous flow functions, we introduce the notion of a *flow function family*.

Definition 3.9 (Flow Function Family). We write f_ζ to denote the family of flow functions $f_\zeta[\![\_]\!]$ (either atomic or synchronous) derived from a common flow evaluation strategy $[\![\_]\!]^\zeta$. We say that a flow function family f_ζ has a property (e.g., monotonicity, soundness, precision ordering) if all its members share that property. For simplicity, we omit the strategy subscript and superscript ζ in general discussions.

Example 3.1 (A Template Path-Insensitive Flow Function Family). A template path-insensitive flow function family f_π typically follows this evaluation strategy, where path condition w of $\mathsf{mux}(w, y, z)$ is ignored:

$$[\![n]\!]^\pi_\sigma = \alpha_\pi(n) \quad [\![y\ op\ z]\!]^\pi_\sigma = \sigma(y)\ op_\pi\ \sigma(z) \quad [\![\mathsf{mux}(w, y, z)]\!]^\pi_\sigma = \sigma(y) \sqcup_\pi \sigma(z) \quad [\![y]\!]^\pi_\sigma = \sigma(y)$$

Here α_π, op_π , $\sqcup_\pi$ are customizable for different application purposes. This example defines a standard path-insensitive over-approximation for hardware multiplexers (i.e. $\mathsf{mux}(w, y, z)$).

Definition 3.10 (Flow Precision Ordering). An atomic flow function $f_1[\![\iota]\!]$ is more precise than $f_2[\![\iota]\!]$, written as $f_1[\![\iota]\!] \sqsubseteq f_2[\![\iota]\!]$, if $\forall\sigma. f_1[\![\iota]\!](\sigma) \sqsubseteq f_2[\![\iota]\!](\sigma)$. Likewise, for synchronous flow function $f_1[\![I]\!] \sqsubseteq f_2[\![I]\!]$ if $\forall\sigma. f_1[\![I]\!](\sigma) \sqsubseteq f_2[\![I]\!](\sigma)$.

Definition 3.10 formalizes the precision ordering between flow functions: one function is more precise if it yields more informative (i.e., smaller) over-approximations than the other.

Lemma 3.3 (Precision Ordering Preservation). *If $f_1[\![\iota]\!] \sqsubseteq f_2[\![\iota]\!]$ and $f_1[\![\iota']\!] \sqsubseteq f_2[\![\iota']\!]$, then $f_1[\![\iota \parallel \iota']\!] \sqsubseteq f_2[\![\iota \parallel \iota']\!]$.*

Proof Sketch. Let $\iota = x(=|\Leftarrow)e$, $\iota' = x'(=|\Leftarrow)e'$. For any σ, from $f_1[\![\iota]\!] \sqsubseteq f_2[\![\iota]\!]$ we get $[\![e]\!]^1_\sigma \sqsubseteq [\![e]\!]^2_\sigma$. Similarly, $[\![e']\!]^1_\sigma \sqsubseteq [\![e']\!]^2_\sigma$. Thus, $f_1[\![\iota \parallel \iota']\!](\sigma) = \sigma[x \mapsto [\![e]\!]^1_\sigma, x' \mapsto [\![e']\!]^1_\sigma] \sqsubseteq \sigma[x \mapsto [\![e]\!]^2_\sigma, x' \mapsto [\![e']\!]^2_\sigma] = f_2[\![\iota \parallel \iota']\!](\sigma)$. □

The following example illustrates how the above definitions work together in practice.

Example 3.2 (Zero Analysis Flow Function Family). Zero analysis tracks whether a circuit location holds the value zero, which can be used to detect simple divide-by-zero bugs. It is defined as follows:

$$\langle L_Z, \sqsubseteq_Z\rangle = \quad \begin{array}{ccccc} & & \top & & \\ & \nearrow & & \nwarrow & \\ Z & & & & N \\ & \nwarrow & & \nearrow & \\ & & \bot & & \end{array} \qquad \delta_Z(x) = \top \quad \alpha_Z(n) = \begin{cases} Z, \text{ if } n = 0 & [\![y\ op\ z]\!]^Z_\sigma = \top \\ N, \text{ if } n \neq 0 & \text{(Other cases inherit from Example 3.1)} \end{cases}$$

Here, we conservatively approximate all binary operations as $\top$ for simplicity. This flow function family remains atomically monotonic and sound by definition, and Lemmas 3.1 and 3.2 guarantee that synchronous compositions inherit these properties. More precise flow function families are also possible, and their relative precision can be easily compared with the help of Lemma 3.3.

3.3 Synchronized Fixed-Point Solution for HVFA

In ChiSA, executing an HVFA amounts to computing the least *synchronized fixed-point solution* (Definition 3.12). Its properties will be discussed in Section 3.4. This section focuses on its calculation.

Definition 3.11 (Synchronized Fixed-Point Solution). Given a circuit C, a lattice $\langle L, \sqsubseteq\rangle$, a stimulus description δ and a flow function family f, the synchronized fixed-point (SFP) solution is

$$\mathbf{SFP}^{C,\delta}_f := \{\sigma \in \mathbf{Store} \mid (\forall x \in \mathbf{Input}_C. \sigma(x) = \delta(x)) \wedge (\forall \iota \in C_=. f[\![\iota]\!](\sigma) = \sigma) \wedge f[\![C_\Leftarrow]\!](\sigma) = (\sigma)\}.$$

Here, $f[\![C_\Leftarrow]\!]$—an instance of Definition 3.6—over-approximates $[\![C_\Leftarrow]\!]$ (Definition 2.13) used in the S-Tick rule of the λ_C semantics, which defines the synchronous register updates triggered by clock ticks (Definition 2.14). The soundness of this approximation will be established in Theorem 3.5.

Definition 3.12 (Least SFP Solution). $\mathbf{LSFP}^{C,\delta}_f := \{\sigma \in \mathbf{SFP}^{C,\delta}_f \mid \forall\sigma' \in \mathbf{SFP}^{C,\delta}_f. \sigma \sqsubseteq \sigma'\}$.

Algorithm 1 presents an intuitive procedure for computing $\mathbf{LSFP}_f^{C,\delta}$. It maintains two worklists: $WL_=$ for reactive wire connections and $WL_{\Leftarrow}$ for synchronous register connections, reflecting their different hardware semantics discussed in Section 2.3. Initially, all connections are added to their respective worklists, and the store σ is initialized to $\bot$ for each variable, with overrides from the stimulus description δ (Line 1). The algorithm repeatedly applies flow functions to items in the worklists, updating σ as necessary (Line 2-8). Each change to $\sigma(x_i)$ affects all items in $\mathrm{Succ}(\iota_i)$ (Definition 2.6) where $\mathrm{Def}(\iota_i) = x_i$, which are added to the appropriate worklist for further processing (Line 9-10).

Algorithm 1 Synchronized-Worklist Algorithm

Input: an HVFA problem $C, \langle L, \sqsubseteq\rangle, \delta$ and a flow function family f
Output: $\sigma \in \mathbf{LSFP}_f^{C,\delta}$

1: $WL_= = C_=, WL_{\Leftarrow} = C_{\Leftarrow}, \sigma = \delta \sqcup \{\_ \mapsto \bot\}$
2: **while** $WL_= \neq \emptyset$ **or** $WL_{\Leftarrow} \neq \emptyset$ **do**
3: **if** $WL_= \neq \emptyset$ **then**
4: Remove ι from $WL_=$.
5: $\sigma = f[\![\iota]\!](\sigma)$ ▷ see Definition 3.5
6: **else**
7: $\sigma = f[\![WL_{\Leftarrow}]\!](\sigma)$ ▷ see Definition 3.6
8: $WL_{\Leftarrow} = \emptyset$
9: Add affected wire connections to $WL_=$.
10: Add affected register connections to $WL_{\Leftarrow}$.

Theorem 3.4 gives the convergence condition for Algorithm 1.

Theorem 3.4 (Convergence). *Algorithm 1 converges to the least synchronized fixed point* $\mathbf{LSFP}_f^{C,\delta}$ *if* f *is monotonic and* $\langle L, \sqsubseteq\rangle$ *is a complete lattice with finite height.*

Proof Sketch. We construct a function which simulates Algorithm 1, and then prove its least fixed point is equal to any element in $\mathbf{LSFP}_f^{C,\delta}$ by fixed-point theorems [69, 81, 111]. A detailed proof is provided in the supplementary material due to space constraints. □

The time complexity of Algorithm 1 depends on the implementation of lattice operations, flow functions, and worklist management. Assuming these operations all take constant time, we can roughly estimate the complexity based on the number of single-point updates to the store σ. Since each variable in $C \in \mathbf{Circuit}$ can be updated at most h times—the height of the lattice L—the overall convergence time is bounded by $O(|C| \cdot h)$. Section 5 provides detailed runtime statistics from our implementation in ChiSA, which incorporates careful optimization efforts. We omit discussion of these efforts in this paper as they fall outside the scope of our core contributions.

3.4 Soundness and Precision Discussion of HVFA

This section establishes key properties of the $\mathbf{SFP}_f^{C,\delta}$ and $\mathbf{LSFP}_f^{C,\delta}$ solution from Section 3.3, focusing on conditions that guarantee soundness and factors that influence precision.

3.4.1 Soundness Guarantee Conditions. The soundness of the synchronized fixed-point solution $\mathbf{SFP}_f^{C,\delta}$ to HVFA (Definition 3.4) is ensured when the flow function family f is itself sound, and the initial store derived from the stimulus description δ (Line 1 or Algorithm 1) soundly over-approximates the circuit's initial configuration. This guarantee is formally established in Theorem 3.5.

Theorem 3.5 (Soundness). $\sigma \in \mathbf{SFP}_f^{C,\delta}$ *is sound (Definition 3.4) if* f *is sound (Definition 3.8) and initial store defined by external description* δ *over-approximates initial configuration* c_0 *(i.e.* $\alpha(c_0) \sqsubseteq \sigma_0$ *where* $\sigma_0 = \delta \sqcup \{\_ \mapsto \bot\}$*).*

Proof Sketch. We prove that $\forall c_i.\alpha(c_i) \sqsubseteq \sigma$ by induction on the trace $C \vdash c_0 \rightsquigarrow c_1 \rightsquigarrow \ldots$

Base Case: By assumption, $\alpha(c_0) \sqsubseteq \sigma_0$, and since $\sigma_0 \sqsubseteq \sigma$ (Definition 3.11), $\alpha(c_0) \sqsubseteq \sigma$.

Inductive Step: Let $c_i = \langle \_, E_i, \_\rangle$ and $c_{i+1} = \langle \_, E_{i+1}, \_\rangle$, and assume $\alpha(c_i) \sqsubseteq \sigma$, i.e., $\alpha(E_i) \sqsubseteq \sigma$. We prove $\alpha(c_{i+1}) \sqsubseteq \sigma$, i.e., $\alpha(E_{i+1}) \sqsubseteq \sigma$, by case analysis on the applied reduction rule:

(1) Case C-Eval: $E_{i+1} = [\![\iota]\!](E_i)$. By the soundness of f (Definition 3.8): $\alpha(E_i) \sqsubseteq \sigma \Rightarrow \alpha([\![\iota]\!](E_i)) \sqsubseteq f[\![\iota]\!](\sigma)$. Since σ is a synchronized fixed point (Definition 3.11), $f[\![\iota]\!](\sigma) = \sigma$, thus $\alpha(E_{i+1}) \sqsubseteq \sigma$.

(2) Case S-Tick: $E_{i+1} = [\![C_{\Leftarrow}]\!](E_i)$. Similarly, $\alpha(E_i) \sqsubseteq \sigma \Rightarrow \alpha(E_{i+1}) \sqsubseteq f[\![C_{\Leftarrow}]\!](\sigma) = \sigma$.

(3)Case S-Poke: $E_{i+1} = [\![x = n]\!](E_i) = E_i[x \mapsto n]$. By Definition 3.4, we have $\alpha(n) \sqsubseteq \delta(x) = \sigma(x)$, thus $\alpha(E_{i+1}) \sqsubseteq \sigma$. □

The proof of Theorem 3.5 clarifies how the synchronized fixed-point solution soundly over-approximates the dynamic circuit behavior defined by λ_C. Owing to the deliberately minimal design of λ_C, which comprises only three reduction rules (Definition 2.14), the proof requires considering just three corresponding cases.

3.4.2 Precision Affecting Factors. As shown in Algorithm 1, the fixpoint computation itself introduces no additional imprecision beyond that already inherent in the flow function family f. Consequently, the precision of the computed solution $\mathbf{LSFP}_f^{C,\delta}$ is entirely determined by the precision of f. For a fixed circuit C and stimulus description δ, a more precise flow function family yields a more precise solution, as formally established in Theorem 3.6. This implies that the precision of a hardware value flow analysis can be systematically improved by refining the flow functions, as demonstrated in Example 3.3.

Definition 3.13 (Precision Ordering of Flow Function Families). For two flow function families f_1 and f_2, we write $f_1 \sqsubseteq f_2$ if $\forall \iota.\, f_1[\![\iota]\!] \sqsubseteq f_2[\![\iota]\!]$.

Theorem 3.6 (Precision Ordering). *If $\langle L, \sqsubseteq\rangle$ is a complete lattice, $f_1 \sqsubseteq f_2$, $\sigma_1 \in \mathbf{LSFP}_{f_1}^{C,\delta}$, and $\sigma_2 \in \mathbf{LSFP}_{f_2}^{C,\delta}$, then $\sigma_1 \sqsubseteq \sigma_2$.*

Proof Sketch. We prove the theorem by constructing a unified function that emulates the entire flow function family and applying Tarski's fixed-point theorem [111]. A detailed proof is provided in the supplementary material due to space constraints. □

Example 3.3 (A More Precise Flow Function for Zero Analysis). In the zero analysis of Example 3.2, the flow function $f_Z[\![x = \mathtt{mux}(w, y, z)]\!](\sigma) = \sigma[x \mapsto \sigma(y) \sqcup \sigma(z)]$ introduces imprecision by ignoring the path condition w. We present a more precise alternative, $f'_Z[\![x = \mathtt{mux}(w, y, z)]\!]$, that explicitly accounts for the condition:

$$f'_Z[\![x = \mathtt{mux}(w, y, z)]\!](\sigma) = \begin{cases} \sigma[x \mapsto \bot] \text{ if } \sigma(w) = \bot, & \sigma[x \mapsto \sigma(z)] \text{ if } \sigma(w) = Z, \\ \sigma[x \mapsto \sigma(y)] \text{ if } \sigma(w) = N, & \sigma[x \mapsto \sigma(y) \sqcup \sigma(z)] \text{ otherwise .} \end{cases}$$

It is straightforward to verify by case analysis that $f'_Z[\![x = \mathtt{mux}(w, y, z)]\!] \sqsubseteq f_Z[\![x = \mathtt{mux}(w, y, z)]\!]$. By defining all other cases of $f'_Z[\![\iota]\!]$ identically to $f_Z[\![\iota]\!]$, we obtain $f'_Z \sqsubseteq f_Z$. Then, by Theorem 3.6, the corresponding least fixed-point solution $\sigma' \in \mathbf{LSFP}_{f'_Z}^{C,\delta}$ is more precise than $\sigma \in \mathbf{LSFP}_{f_Z}^{C,\delta}$, i.e., $\sigma' \sqsubseteq \sigma$.

3.5 HVFA Instances for Lightweight Chisel Verification

By specifying different problem-specific lattices, stimulus descriptions, and flow function families, various HVFAs can be instantiated to suit distinct application goals. As an illustration, this section presents representative instances that support our lightweight analyses for critical Chisel verification tasks, including bug detection and security analysis, as evaluated in Section 5. Since these instances follow relatively standard formulations, we omit detailed discussion. To save space, we reuse the template path-insensitive flow function family from Example 3.1 and only specify the parts that differ.

3.5.1 Hardware Bug Detection. ChiSA identifies violable assertions (Section 5.1) by leveraging interval analysis (Example 3.4) and constant propagation analysis (Example 3.5) to approximate assertion violation conditions.

Example 3.4. Interval analysis computes value ranges for each circuit location.

- *Lattice*: $\langle L_I, \sqsubseteq_I \rangle$, where $L_I = \textbf{Interval} \cup \{\bot\}$ and: $\textbf{Interval} := \{[m, n] \mid m \in \{-\infty\} \cup \mathbb{Z} \wedge n \in \mathbb{Z} \cup \{+\infty\} \wedge m \leq n\}$ with $\forall x \in \mathbb{Z}. -\infty \leq x \leq +\infty$. The abstraction function is $\alpha_I(n) = [n, n]$. The partial order is defined as: $\bot \sqsubseteq_I l$ for all $l \in L_I$, and $[a, b] \sqsubseteq_I [c, d]$ iff. $c \leq a \wedge b \leq d$.
- *Stimulus Description*: $\delta_I(x) = [-\infty, +\infty]$.
- *Flow Function Family*: $[\![y \ op \ z]\!]^I_\sigma = \sigma(y) \ op_I \ \sigma(z)$, where $[a, b] \ op_I \ [c, d] = [\min\{m \ op \ n \mid m \in [a, b] \wedge n \in [c, d]\}, \max\{m \ op \ n \mid m \in [a, b] \wedge n \in [c, d]\}]$ and $l \ op_I \ \bot = \bot$ (symmetric).

Discussion About the Interval Lattice in Practice. The interval lattice L_I in Example 3.4 has infinite height and thus does not satisfy the convergence condition in Theorem 3.4, meaning that convergence of Algorithm 1 in this case is not theoretically guaranteed by the theorem. However, in practice, variables in Chisel have fixed-width integer types, such as `UInt<3>` for unsigned 3-bit integers and `SInt<4>` for signed 4-bit integers. Consequently, the top lattice element $\top$ for a variable of type `UInt<n>` is $[0, 2^n - 1]$, and for `SInt<n>`, it is $[-2^{n-1}, 2^{n-1} - 1]$—both bounded ranges rather than the unbounded $[-\infty, +\infty]$. Thus, the practical interval lattice has finite height, and the interval analysis in practice will converge as guaranteed by Theorem 3.4. For completeness, note that convergence over unbounded mathematical integers in Example 3.4 can be ensured using a standard *widening operator* [82]. We omit further discussion of widening, as it is a well-established technique, and our Chisel interval analysis already converges in practice without it.

Example 3.5. Constant propagation analysis tracks constants flowing through circuit locations.

- *Lattice*: $\langle L_C, \sqsubseteq_C \rangle$, where $L_C = \mathbb{Z} \cup \{\bot, \top\}$. The abstraction function is $\alpha_C(n) = n$. The partial order is defined as: $\forall l \in L_C. \ \bot \sqsubseteq_C l \sqsubseteq_C \top$.
- *Stimulus Description*: $\delta_C(x) = \top$.
- *Flow Function Family*: $[\![y \ op \ z]\!]^C_\sigma = \sigma(y) \ op_C \ \sigma(z)$, where $m \ op_C \ n = m \ op \ n$ if $m, n \in \mathbb{Z}$, $l \ op_C \ \bot = \bot$ if $l \in L_C$, $l \ op_C \ \top = \top$ if $l \in L_C - \{\bot\}$ (symmetric for both $\bot$ and $\top$).

3.5.2 Hardware Security Analysis. ChiSA detects confidentiality and integrity violations by identifying taint flows (Section 5.2) from secret sources to public sinks or from untrusted sources to trusted sinks with the help of reachability analysis (Example 3.6).

Example 3.6. Reachability analysis computes which circuit locations are reachable from given sources in **Source**.

- *Lattice*: $\langle L_T, \sqsubseteq_T \rangle$, where $L_T = \{\bot, \top\}$ with $\bot \sqsubseteq_T \top$. The abstraction function is $\alpha_T(n) = \bot$. A location with abstract value $\top$ is considered *reachable*.
- *Stimulus Description*: $\delta_T(x) = \top$ if $x \in \textbf{Source}$, otherwise $\delta_T(x) = \bot$.
- *Flow Function Family*: $[\![y \ op \ z]\!]^T_\sigma = \sigma(y) \sqcup_T \sigma(z)$, $[\![\texttt{mux}(w, y, z)]\!]^T_\sigma = \sigma(w) \sqcup_T \sigma(y) \sqcup_T \sigma(z)$.

4 ChiSA: A Proof of Concept

As a proof of concept for our theoretical foundations in Sections 2 and 3, we develop ChiSA (30K+ LoC), the first Chisel static analyzer capable of analyzing intricate hardware value flows to enable sophisticated analyses for critical Chisel verification tasks, such as bug detection and security analysis. To support future research and facilitate the development of new Chisel analyses, a substantial portion of the codebase is dedicated to constructing reusable infrastructure. This section offers a brief overview of how this infrastructure supports ChiSA's end-to-end analysis workflow.

ChiSA's core components—its IR (ChAIR) and fundamental analyses—are designed and implemented based on the theoretical foundations of λ_C (Section 2) and HVFA (Section 3), respectively. Due to space constraints, we do not provide detailed descriptions of ChiSA's individual components; interested readers can refer to our open-source code and documentation for implementation details and engineering specifics.

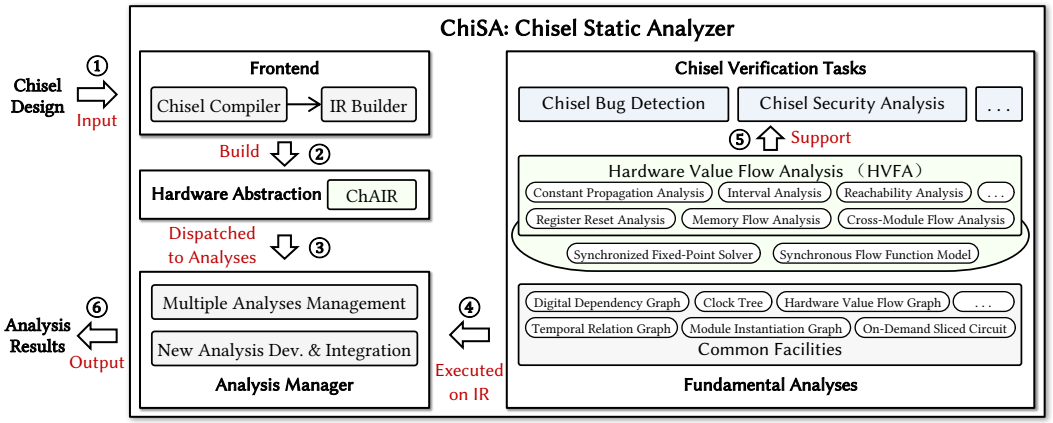


Fig. 3. The architecture and end-to-end workflow of ChiSA.

Figure 3 illustrates ChiSA's architecture and end-to-end workflow, which are inspired by high-quality software analysis frameworks [20, 64, 109]. ChiSA takes a Chisel hardware design (or a Chisel program) as input and produces analysis results such as bug reports or security warnings:

- ① & ② : Given a Chisel design as input, the ChiSA frontend first builds our ChAIR by reusing standard passes from Chisel's official compiler [12, 13] for common compilation tasks, followed by a dedicated IR builder that applies ChiSA's custom transformations to generate ChAIR.
- ③ & ④ : Next, the analysis management system dispatches the IR to the analyses requested by users and orchestrates their executions, managing possible intricate analysis dependencies and configurations. This system also provides mechanisms that facilitate developers to develop and integrate new analyses by reusing existing ones.
- ⑤ : Analyses for Chisel verification tasks require substantial infrastructure *support*. Central to this are various hardware value flow analyses (HVFAs) that provide fundamental information about Chisel hardware designs. Additionally, ChiSA offers common facilities including a rich set of useful graph representations built on top of ChAIR and a circuit slicing tool to help hardware designers narrow down the scope of bug localization during debugging. These components are general-purpose and designed to be reusable for future analyses.
- ⑥ : Finally, the analysis manager outputs the analysis results as bug reports or security warnings, depending on the specific verification tasks.

5 Evaluation

To validate ChiSA's effectiveness for Chisel verification, we address the following research questions:

RQ1. How does ChiSA support hardware bug detection compared to bounded model checking?
RQ2. How does ChiSA perform in hardware security analysis compared to secure type systems?

To our knowledge, no existing static analysis work can detect our target Chisel bugs and security vulnerabilities, as these require sophisticated reasoning about hardware value flows—capabilities beyond current AST-based analyses in the Chisel compiler (see the next paragraph). Consequently, we compare ChiSA against state-of-the-art techniques in respective verification tasks: bounded model checking for bug detection and secure type systems for security analysis.

Comparison with Official Chisel Compiler. The official Chisel compiler provides only basic static analyses (e.g., type checking) operating on the Firrtl AST, with constant propagation as its sole data

flow analysis. To address the potential concern regarding how ChiSA performs against the only available comparable analysis, we compared both implementations on the Chisel compiler's official test suite (877 tests). ChiSA's constant propagation identified 99.8% of the constants detected by the official implementation (20,509 constants total) while achieving superior performance (1.5s vs. 2.1s). This result is significant given that the official compiler's constant propagation is highly optimized and specifically designed to exploit the Firrtl AST information. The results (comparable soundness and precision) indirectly demonstrates that ChiSA's streamlined IR (Section 2) effectively preserves essential semantic information from the Firrtl AST level, while also validating ChiSA's fundamental analytical capabilities in non-verification tasks.

Benchmarks. To support thorough evaluation of both ChiSA and future research on Chisel static analysis, we provide ChiSABench, a comprehensive Chisel static analysis benchmark suite encompassing over eleven million lines of code.

Real-world Chisel projects typically involve a mix of heterogeneous languages—including hardware languages such as Chisel and Verilog, software languages such as Scala, Java, and C, and assembly languages like RISC-V—each with its own build system and dependency management. In practice, only specific version combinations of these toolchains and dependencies are known to work, making it cumbersome to build all benchmarks in ChiSABench from scratch.

To address this, we invested considerable effort to pre-elaborate all designs in ChiSABench into standalone Firrtl [63] files, eliminating the need for tedious environment setup or project-specific build steps. This greatly enhances ChiSABench's out-of-the-box accessibility and hands-on usability for future research. Additionally, although the TrustHub [102, 104] benchmark was originally written in Verilog [1], we incorporate it in ChiSABench due to the absence of authoritative security benchmarks for Chisel. To make it compatible, we convert it to Firrtl using Yosys [11], a widely adopted open-source hardware synthesis tool that supports Verilog-to-Firrtl translation, as recommended by the official Firrtl project itself [12].

As summarized in Table 1, ChiSABench distinguishes itself through the following attributes:

- *Authority*: The hardware designs in ChiSABench are primarily drawn from Chisel's official toolchain [12–14] and from projects endorsed by the Chisel community [32].
- *Diversity*:
  (1) *Language Feature Diversity*: ChiSABench includes official compiler and simulator test cases that exercise a comprehensive spectrum of Chisel language features.
  (2) *Design Purpose Diversity*: ChiSABench includes a wide range of real-world applications, such as system-on-chip (SoC), network-on-chip (NoC), deep neural network (DNN) accelerators, and vector co-processors. Notably, it also incorporates security-relevant benchmarks, including information leak trojans from the widely used TrustHub suite [102, 104] and secure type system tests from the state-of-the-art ChiselFlow project [47], facilitating evaluation of security-oriented analyses.
  (3) *Code Scale Diversity*: ChiSABench spans designs ranging from a few dozen lines to over seven million lines of code, enabling evaluation across projects of vastly different sizes.
- *Real-World Impact*: ChiSABench features numerous popular open-source Chisel hardware projects with significant community adoption—many with thousands of GitHub stars. These projects, categorized as real-world and large-scale in Table 1, reflect practical relevance and ensure that analyses evaluated on ChiSABench are applicable to production-level designs.

Implementation. To balance fine-grained control over analysis efficiency with seamless integration into the Chisel ecosystem (written in Scala), we choose to implement ChiSA in Java. The ChiSA codebase comprises over 30K lines of Java code. To ensure robustness, we have thoroughly

Table 1. Overview of ChiSABench. *Characteristic* denotes the primary attribute of each benchmark category. *Design Purpose* describes the original design intent of each benchmark. *Benchmark* lists the names of all benchmarks. *#Designs* indicates the number of hardware designs in each benchmark. *LoC* reports the total lines of pre-elaborated Chisel hardware designs in Firrtl [63] that enhances accessibility and usability.

Characteristic	Design Purpose	Benchmark	#Designs	LoC
Feature-Diverse	Official Toolchain Tests	Chisel3 [12–14]	877	224,845
Real-World	System-on-Chip	Rocket [16]	2	560,405
		BOOM [23]	1	550,147
		Quasar [6]	1	159,179
		Sodor [100]	5	21,109
		RiscvMini [99]	1	2,971
	Network-on-Chip	IceNet [98]	1	236,506
		Constellation [124]	1	5,389
	Deep Neural Network Accelerator	Gemmini [50]	1	632,327
	Vector Co-Processor	Hwacha [76]	1	553,087
Large-Scale	System on Chip	XiangShan [119]	1	7,176,167
Vulnerable	Information Leak Trojan	TrustHub [102, 104]	25	1,155,854
	Secure Type System Tests	ChiselFlow [47]	18	657
		Total:	935	11,278,643

exercised all analyses in ChiSA across the entire ChiSABench suite (11M+ LoC), with no crashes observed. We will publicly release both ChiSA and ChiSABench to support future research and development in static analysis for Chisel.

Experimental Setup. All experiments were conducted on an Intel Xeon 2.2GHz machine with the JVM heap memory capped at 64GB. We evaluate ChiSA on different portions of ChiSABench (Table 1), selectively chosen to align with the specific goals of each experiment. The rationale for each selection is provided in the corresponding subsections.

5.1 RQ1: Hardware Bug Detection — ChiSA vs. Bounded Model Checking

Hardware bug detection remains a highly challenging task, even in industrial settings. According to Siemens's 2024 global industry study [49], fewer than 15% of hardware projects reported zero bug escapes into production, despite substantial verification investments. Among these bugs, logic errors have consistently been the leading cause over the past decades [49]. A widely adopted paradigm for detecting such errors is assertion-based verification (ABV) [115], which checks for violations of assertions that encode correctness properties. Therefore, we select assertion violation detection as a representative task to evaluate ChiSA's lightweightness and effectiveness in hardware bug detection.

The state-of-the-art technique for automated assertion violation detection in Chisel is bounded model checking (BMC), provided by ChiselTest [73] (hereafter referred to as ChiselTest-BMC). ChiselTest-BMC serves as the BMC backend in most of the recent Chisel verification efforts [42, 105, 121], including the latest one [105]. While BMC can perform well on small-scale designs, it fundamentally struggles to scale to real-world, complex hardware designs with millions of lines of code, due to the well-known state explosion problem [30]. This is exactly the scenario where static

Table 2. Results of ChiSA's static assertion analysis compared to ChiselTest-BMC [73]. For each benchmark, we report the number of violable assertions detected (#Violable), those partially validated by manually writing assertion-triggering module-level testbenches (#P-Validated), the number of designs the tool failed to handle due to crashes (#Crashes), and total runtime (in seconds). ChiselTest-BMC was configured with a 10-cycle bound. It failed on all complex real-world designs, producing various errors (detailed in Section 5.1).

Feature	Benchmark	LoC	ChiSA			ChiselTest-BMC		
			#Violable (#P-Validated)	#Crashes	Time (s)	#Violable (#P-Validated)	#Crashes	Time (s)
Small-Scale	Chisel3 (877 designs)	256 (on average)	25 (24)	0	**3.1**	139 (139)	72	**2776.3**
Real-World	XiangShan	7,176,167	28 (23)	0	145.3	Assumption Errors		
	Gemmini	632,327	8 (7)	0	10.7	Internal Errors		
	Rocket	560,405	13 (13)	0	9.6	Incomplete Errors & Internal Errors		
	Hwacha	553,087	7 (7)	0	10.8	Internal Errors		
	Boom	550,147	7 (3)	0	17.6	Incomplete Errors		
	IceNet	236,506	0 (0)	0	3.8	Incomplete Errors		
	Constellation	5,389	6 (3)	0	0.1	Incomplete Errors		
	Total:	9,714,028	69 (56)	0	197.9	0 (0)	8	—

analysis can serve as a lightweight complement—requiring significantly less time while still being effective.

ChiSA's static assertion analysis detects assertion violations by approximating the conditions under which an assertion might be violated, leveraging two HVFA instances: interval analysis (Example 3.4) and constant propagation (Example 3.5). Although this approximation-based approach is inherently less accurate than BMC's exhaustive state enumeration, it remains effective on large, real-world hardware designs while being significantly more lightweight. To demonstrate this, we applied ChiSA's static assertion analysis to all Chisel designs in ChiSABench that include developer-inserted, real-world embedded assertions.

As shown in Table 2, ChiSA efficiently completed whole-program analysis on real-world designs totaling over 9 million lines of code in under 200 seconds, identifying 69 potentially violable assertions. Due to the large codebase and complexity of these projects, manually verifying all detected assertions through system-level testbenches is infeasible. Instead, we partially validated ChiSA's results by constructing module-level testbenches: 56 of the flagged assertions were successfully triggered, each causing the corresponding assertion to fail. On average, crafting each testbench involved manually inspecting thousands of lines of hardware module code to figure out the assertion violation scenario. This manual validation provides partial confirmation of ChiSA's effectiveness in uncovering risky assertion violations. Notably, eight assertion violations were recognized by the developers and scheduled for future fixes—further showcasing the potential of static analysis to aid in detecting hardware bugs. All testbenches used to trigger these assertion failures will be included in our artifact.

In contrast, ChiselTest-BMC failed to analyze any of these real-world designs, reporting various types of errors stemming from its own limitations:

- *Incomplete Errors:* The most common failure mode was incomplete errors, observed in four designs. These were caused by the presence of external modules whose definitions were inaccessible—a common scenario in hardware development due to the frequent use of intellectual property (IP) cores. ChiSA handles such cases by supporting incomplete analysis through automatic

and conservative approximation of external component behavior—giving it an advantage over ChiselTest-BMC in realistic design settings.

- *Assumption Errors:* ChiselTest-BMC failed to analyze the XiangShan system-on-chip (SoC) [119], reporting an assumption error due to unmet internal expectations, such as requiring exactly one input port named `clock`. This suggests that the assumptions underpinning ChiselTest-BMC are too narrow to accommodate popular and production-ready designs such as XiangShan, which has been successfully taped out [119]. These assumptions are required by ChiselTest-BMC to construct a precise model of the hardware design for property checking. In contrast, ChiSA does not rely on such assumptions, thanks to the over-approximate nature of static analysis—making it more applicable to complex, real-world scenarios.
- *Internal Errors:* The remaining three designs triggered internal errors due to unhandled edge cases or limitations in the tool's implementation, producing error messages like "Internal Error! Please file an issue at our repository." The official ChiselTest-BMC test suite [4] primarily consists of small examples averaging fewer than 100 lines of code, which do not reflect the complexity and scale of real-world Chisel designs. This suggests that, although ChiselTest-BMC represents the current state of the art, it has not been adequately tested against real-world cases.

To further highlight ChiSA's lightweight nature, we additionally evaluated both tools on 877 small-scale, simpler Chisel designs (average 256 LoC), where runtime statistics for ChiselTest-BMC could be obtained. As shown in Table 2, ChiSA completed the analysis of all these designs in just 3 seconds, a significant reduction compared to the 2776 seconds required by ChiselTest-BMC. Even in this small-scale evaluation, ChiselTest-BMC reported 72 crashes, in stark contrast to ChiSA, which completed all analyses without a single failure.

It is also evident from Table 2 that although ChiSA demonstrates promising results on complex real-world designs, it is less capable of uncovering violable assertions than ChiselTest-BMC in cases where BMC works well. This is because ChiSA deliberately trades soundness for precision here to reduce false positives and enhance practical usability. Specifically, a Chisel assertion consists of a `predicate` signal and an `enable` signal, and triggers a failure when `predicate` is `false` while `enable` is `true`. A sound approach would flag all assertions where `predicate` may be `false` and `enable` may be `true`, but in practice, this leads to many false positives. To address this issue, ChiSA flags only those assertions where the `predicate` must be `false` while the `enable` signal may be `true`. This approach significantly reduces false positives while maintaining sufficient soundness to remain useful, as already illustrated via experiments on complex, real-world Chisel designs.

Case Study. To illustrate the violable assertions identified by ChiSA in Table 2, we present a representative case that has been recognized by the developers and scheduled for future fixes. This assertion violation was found in the `DCache` module (6.6K LoC) of the Rocket [16] system-on-chip (SoC), and takes the form (simplified for readability) `assert(clock, release_ack_wait, grantIsVoluntary, "A ReleaseAck was unexpected by the dcache.")` in ChAIR, ChiSA's intermediate representation. This assertion ensures that ReleaseAck messages—a type of acknowledgement in the TileLink protocol [62]—arrive only when the cache expects them. Violating this assertion could signal a protocol mismatch, potentially leading to inconsistent cache states or deadlocks. The assertion fails when the predicate signal `release_ack_wait` is `false` (i.e., the cache is not expecting an acknowledgment), while the `enable` signal `grantIsVoluntary` is `true` (i.e., a ReleaseAck message is present). The `clock` argument is simply the signal used to drive runtime assertion checks and does not affect the assertion logic itself. ChiSA detects this assertion as potentially violable by querying the results of constant propagation and interval analysis. It determines that `release_ack_wait` is statically `false`, while `grantIsVoluntary` may be `true`, thereby flagging the assertion as potentially violated. Our test case triggers this assertion violation

Table 3. Results of ChiSA's taint analysis compared to the secure type system provided by ChiselFlow [47]. #Designs denotes the number of designs and *Function* denotes their hardware functionality. "*" indicates ChiselFlow's developer-crafted micro-benchmark with no specific hardware functionality. For each benchmark, we report the number of detected unintended information flows (#Vulnerabilities), including false positives (#FP) and false negatives (#FN), the number of required annotations (#Annotations)—sources/sinks for ChiSA, type labels for ChiselFlow—and analysis time in seconds. "—" denotes that ChiselFlow could not be applied due to the prohibitive annotation overhead. For the ChiselFlow benchmark, since the ground truth is derived from ChiselFlow's own results, we omit redundant reporting in the ChiselFlow columns.

Benchmark	#Designs × Function	LoC	ChiSA			ChiselFlow	
			#Vulnerabilities (#FP / #FN)	#Annotations (#Sources / #Sinks)	Time (s)	#Annotations (Type Labels)	Time (s)
ChiselFlow	18 × *	655	19 (1 / 0)	44 (25 / 19)	0.006	228	14.475
TrustHub	19 × AES [85]	1,004,180	54 (0 / 0)	73 (19 / 54)	0.175	—	
	3 × ISCAS89 [21]	143,440	3 (0 / 0)	6 (3 / 3)	0.435		
	1 × PIC16F84 [65]	5,932	1 (0 / 0)	2 (1 / 1)	0.017		
	2 × RSA [83]	2,302	2 (0 / 0)	4 (2 / 2)	0.005		
	Total:	1,155,854	60 (0 / 0)	85 (25 / 60)	0.632		

by issuing a voluntary release operation and delivering the corresponding ReleaseAck in the same clock cycle, revealing that the DCache fails to handle a legal scenario in which acknowledgments arrive with minimal latency during valid protocol transactions.

5.2 RQ2: Hardware Security Analysis — ChiSA vs. Secure Type System

Hardware security has become increasingly critical in this new golden age of computer architecture [53, 54]. A fundamental road for uncovering insecure hardware behavior is the detection of unintended information flows [56]. We therefore select this task—detecting unintended information flows—as a representative case to evaluate ChiSA's lightweightness and effectiveness in hardware security analysis. Because evaluating unintended information flow detection requires ground-truth knowledge of design intent, we limit our evaluation to the ChiselFlow [47] and TrustHub [102, 104] benchmarks from the ChiSABench suite, both of which include ground-truth labels for unintended information flows.

The state-of-the-art in hardware security for Chisel—SecChisel [39, 40] and ChiselFlow [47]—relies on extending Chisel's type system with security labels to detect unintended information flows via type annotations. However, this type-system-based approach requires substantial manual annotation effort [91, 92] that scales with code size, placing a significant burden on hardware designers and limiting its adoption in large, real-world projects. For example, as shown in Table 3, ChiselFlow requires as many as 228 type labels to detect only 18 unintended flows in its own benchmark, which contains just 657 lines of code. We do not include SecChisel in this evaluation because it is not open-source and its core methodology closely mirrors that of ChiselFlow.

ChiSA's taint analysis detects unintended information flows—referred to as taint flows—by tracking value flows from private sources to public sinks (confidentiality violations) or from untrusted sources to trusted sinks (integrity violations). It performs this tracking via the reachability analysis (Example 3.6) powered by HVFA. Unlike ChiselFlow, which requires fine-grained type annotations throughout the entire flow path—including all intermediate variables and statements—ChiSA's taint analysis only requires users to annotate sources and sinks via a lightweight configuration file to express their security intent and will automatically check whether there're unintended taint flows between from specified sources to sinks. As a result, the annotation effort required by our

approach scales with the hardware design's security goals expressed in source/sink pairs, rather than with the code size (as required by ChiselFlow's type annotations), drastically reducing the manual burden.

As shown in Table 3, ChiSA's taint analysis detects all 18 taint flows in the ChiselFlow benchmark using just 44 source/sink annotations—a substantial reduction compared to ChiselFlow's 228 type annotations. This demonstrates ChiSA's lightweight annotation burden compared to ChiselFlow, significantly reducing manual effort and making it far more practical for real-world hardware security analysis.

Another important advantage of ChiSA over ChiselFlow is its ability to be retroactively applied to large, existing Chisel codebases that lack security type labels. This is a far more common scenario in practice, because real-world projects typically adopt the standard Chisel type system, which is officially supported and actively maintained, whereas ChiselFlow remains a research prototype without active maintenance. As shown in Table 3, ChiSA efficiently detects all 60 information-leak vulnerabilities in the TrustHub benchmark—which comprises over a million lines of code—using only 85 source/sink annotations provided as analysis configuration, exhibiting a lightweight annotation burden at this scale. In contrast, ChiselFlow is inapplicable to TrustHub due to the prohibitive manual effort required to retrofit millions of lines of existing code with its fine-grained type system, whose annotation burden scales with code size.

The single false positive ChiSA reported on the ChiselFlow benchmark stems from its over-approximate analysis, which may soundly flag infeasible taint flows that cannot occur in actual executions. In contrast, ChiselFlow avoids this false positive by leveraging fine-grained type annotations along the entire information flow path. These annotations can encode path conditions using dependent types, allowing ChiselFlow to distinguish merged flows with higher precision. Nevertheless, we argue that this modest loss of precision is a worthwhile trade-off for the significant reduction in manual annotation effort afforded by ChiSA.

Case Study. To shed light on the vulnerabilities detected by ChiSA, we examine `AES-T100` (1.2K LoC), the first case in TrustHub's information-leak vulnerability suite [102, 104] as an example. This benchmark contains a cryptographic chip running the AES algorithm that has been compromised with a hardware Trojan, which leaks the secret key through a covert channel. Following the security intent described in the benchmark's official documentation, we configure ChiSA's taint analysis by marking the input port `top.key`—which carries the secret AES key—as a secret source, and the output port `top.Capacitance`—a signal observable via physical measurements—as a public sink. Leveraging its underlying hardware value flow analysis, ChiSA automatically identifies an unintended information flow from `top.key` to `top.Capacitance` without further manual effort beyond these two source/sink configurations, thereby revealing a violation of confidentiality and indicating the presence of the Trojan. In contrast, existing secure type systems such as ChiselFlow are not directly applicable, as `AES-T100` was written using the standard type system rather than ChiselFlow's extended version. Although conceptually possible, retrofitting an existing codebase with a new type system is highly impractical in reality: ChiselFlow's own benchmark (655 LoC) requires 228 manual security annotations, highlighting the substantial annotation burden introduced by its type-based approach. This case study underscores the practicality of ChiSA's analysis for identifying security vulnerabilities in unmodified, real-world hardware designs without requiring intrusive code changes or extensive manual effort.

6 Related Work

Chisel Verification. Unlike ChiSA's lightweight static analysis approach, existing Chisel verification techniques largely adopt heavyweight methodologies inherited from the broader hardware

community, with the efficiency limitations already discussed in Section 1. Here, we examine these efforts in greater detail, tracing their methodological roots to traditional hardware verification:

(1) Simulation-Based Testing. ChiselTest [101], integrated within the ScalaTest framework [7], provides robust IDE support and continuous integration capabilities for executing unit-level simulation tests. It runs simulations by interpreting either Chisel-generated Verilog via Verilator [106], or Chisel's compiler intermediate representation, Firrtl [63], via Treadle [14]. ChiselVerify [43] extends ChiselTest by introducing coverage metrics [41], fuzzing capabilities [44], and features specifically tailored for testing approximate hardware designs [37]. DESSERT [67], an advancement over Strober [68], enhances simulation performance by translating Firrtl to FAME1 [110] for FPGA acceleration, and supports efficient differential testing for Chisel designs.

Rather than comparing against dynamic testing approaches, we focus our evaluation on static analysis versus other static techniques. This follows common practice, as static and dynamic methods are generally considered orthogonal, each addressing distinct verification dimensions [2].

(2) Formal Verification. ChiselTest [73] and ChiselVerify [42] provide bounded model checking (BMC) abilities by translating Firrtl designs into transition systems expressed in SMTLib [18] or Btor2 [88], then solving them using Z3 SMT Solver [38] or BtorMC model checker [88]. CHA [121], built atop ChiselTest, extends its assertion support to include SystemVerilog Assertions (SVAs)[5]. ChiselFV [117] also supports SVA but instead compiles Chisel with embedded assertions directly to SystemVerilog and leverages SymbiYosys [10] for SystemVerilog BMC. Due to the well-known state explosion problem [30], bounded model checking (BMC) suffers from poor scalability and is thus typically applied only to small-scale designs [105, 118]. Chicala [46], inspired by V2C [3]—a Verilog to C translator, translates Chisel designs into behaviorally equivalent Scala programs, which are then verified using Stainless [9], a theorem-proving tool for Scala.

(3) Secure Type System. SecChisel [39, 40] and ChiselFlow [47], closely following the design of SecVerilog [123], extend Chisel's type system to express security policies via type annotations. These annotations are translated into type constraints and then statically checked using the Z3 SMT solver [38] to detect violations of confidentiality and integrity properties.

Our evaluation (Section 5) compares ChiSA's static analyses with representative techniques from both *(2) Formal Verification* and *(3) Secure Type Systems*, showing that ChiSA provides an effective and significantly more lightweight solution, particularly for large, real-world Chisel designs.

Static Analysis for Verilog. Despite Chisel's current reliance on Verilog as a backend for compatibility with the existing commercial electronic design automation (EDA) ecosystem, these static analyses designed for regular hand-written Verilog are ill-suited for Chisel-generated Verilog. Here we review mainstream static analysis approaches developed for Verilog to clarify this mismatch, as well as highlighting the need for Chisel-native static analyses that ChiSA provides.

The development of static analysis for Verilog remains far less mature than that for software programming languages. In practice, static analysis for Verilog is overwhelmingly dominated by linters, including open-source tools such as Slang [95], Verible [26], and SVLint [52], as well as industrial-grade solutions like Spyglass Lint [107]. These tools perform syntactic, stylistic, and pattern-based checks over abstract syntax trees (ASTs) to detect common issues in regular hand-written Verilog early in the design cycle. However, linting rules that are effective for regular hand-written Verilog can break down when applied to Chisel-generated Verilog. For example, a widely adopted rule flags variables in combinational `always` blocks that are not assigned along all execution paths—a heuristic commonly used to detect unintended latch inference. Yet this rule becomes totally useless in Chisel-generated Verilog, which, due to Chisel's structural modeling semantics, does not emit combinational `always` blocks at all. This disconnect highlights the limitations of directly

reusing Verilog-oriented static analyses for Chisel workflows and underscores the need for dedicated analyses specifically tailored to Chisel's design idioms.

Other tools, such as Pyverilog [108] and VeriPy [96], construct graph-based representations from Verilog ASTs to perform control-flow and data-flow analyses. These approaches are well-suited to Verilog's behavioral modeling constructs, such as `always` blocks, which encapsulate control and computation logic. In contrast, Chisel eschews behavioral modeling in favor of structural descriptions targeting synthesizable hardware descriptions. As a result, the Verilog emitted by Chisel tends to contain flat, repetitive `always` blocks that encode simple register connections, with minimal embedded control or computation logic. This flattening substantially diminishes the effectiveness of control/data-flow analyses designed for regular hand-written Verilog. Altogether, the structural gap between regular hand-written Verilog and Chisel-generated Verilog highlights the inadequacy of repurposing Verilog-based static analysis tools for Chisel.

Not only are existing Verilog static analysis tools inherently ill-suited for reuse in Chisel contexts, as discussed above, but even static analyses specifically tailored for Chisel-generated Verilog will still face fundamental limitations. The key issue is that important Chisel-specific semantic information is lost during the Verilog generation process, making it infeasible to reconstruct high-level analysis targets. This further underscores the necessity of Chisel-native static analyses that ChiSA provides. We highlight this mismatch with three intuitive examples: *First*, assertions written in Chisel do not survive the standard translation into Verilog, eliminating the possibility of performing static assertion analysis (discussed in Section 5.1) directly on the generated Verilog. *Second*, while Chisel source locations are embedded in the generated Verilog, they are stored only in comments. Because comments are discarded during parsing, this information is inaccessible to Verilog-based analysis tools, undermining traceability of analysis results. For instance, applying circuit slicing—provided by ChiSA to reduce bug localization scope for hardware designers (though not discussed in detail)—would require tough manual effort to trace analysis results back to the original Chisel code, substantially harming usability. *Third*, Chisel treats memories as distinct language constructs, allowing native analyses to handle them differently from register vectors. In Chisel-generated Verilog, however, memories are lowered into register vectors indistinguishable from manually written register vectors. This erasure of semantic distinction prevents specialized treatment of Chisel memories, such as modeling read/write latencies or resolving read-under-write behavior—capabilities that facilitate more Chisel-targeted analysis.

Theoretical Foundations of Analysis for Traditional Hardware Description Languages (HDLs). Although no prior theoretical foundations have been proposed for Chisel static analysis, analysis theories do exist for traditional HDLs—most representatively, abstract interpretation for Verilog [84] and VHDL [57–59]. Additionally, formal semantics for Verilog [25, 27, 79] and VHDL [60] can also support analysis theories. However, these theories target the behavioral modeling paradigm of traditional HDLs (as discussed in the *Static Analysis for Verilog* paragraph above) and formalize heavyweight features specific to that paradigm, making them ill-suited to Chisel's lightweight, structural-oriented nature. To address this gap, we present the first theoretical foundation targeting Chisel, capturing its essence in a minimal way to enable tractable reasoning about static analyses.

Lightweight Formal Methods for HDLs. Besides static analyses—the primary focus of this paper—type systems constitute another representative class of lightweight formal methods that have been extensively developed in the programming languages (PL) community. Advanced type systems can facilitate static analyses by encoding richer semantic information in program and enforcing stronger safety properties through type checking [87]. However, the standard type systems of traditional HDLs (e.g., Verilog [1]) remain relatively basic and lack the expressive power and safety guarantees common in PL research. Recently, several novel HDL type systems have been proposed

to encode richer semantic information in program and enforce stronger reliability-related properties for *safe composition* [28, 89, 90].

For instance, timeline types [89, 90] have been proposed to encode timing constraints, thereby preventing *structural hazards* that harm safe composition through type checking. In contrast, because Chisel's standard type system does not provide this timing information, ChiSA would need to approximate this information if it were to detect timing-related issues such as structural hazards, which introduces additional computational overhead and potential imprecision. Specifically, the analysis would have to approximate the number of clock cycles it takes for a signal starting from an input port to arrive at an output port by counting the register connections along paths in the value flow graph (Definition 2.6). If Chisel were equipped with timeline types in the future, this information would be immediately and precisely available, enabling more efficient and effective static analyses in ChiSA for timing-related issues.

Another example is wire sorts [28], a type system designed for safe composition by enforcing the absence of combinational loops via type checking. Similar to the discussion of timeline types above, future analyses in ChiSA could also benefit from the combinational reachability information encoded in wire sorts, if such a type system were integrated into Chisel.

7 Conclusions

This work establishes a theoretical foundation for Chisel static analysis. We introduced λ_C, a minimal core calculus that captures the essence of Chisel while enabling rigorous reasoning about static analysis. Building on λ_C, we formalized the hardware value flow analysis (HVFA) problem, adapting classical data/value flow analysis from software to hardware by handling the essential feature of Chisel, i.e., synchronous semantics of clock-driven hardware registers. We proved key theorems establishing HVFA's guarantees and limitations. As a proof of concept, we developed ChiSA, the first Chisel static analyzer capable of analyzing intricate hardware value flows for verification tasks such as bug detection and security analysis. To thoroughly evaluate ChiSA's effectiveness, we introduce ChiSABench, a comprehensive benchmark suite for Chisel static analysis. Our evaluation on ChiSABench demonstrates that ChiSA offers an effective and highly lightweight approach, significantly outperforming state-of-the-art techniques on large, real-world Chisel designs. By open-sourcing both ChiSA (30K+ LoC) and ChiSABench (11M+ LoC), we hope to facilitate future research in Chisel static analysis and inspire broader applications of programming language techniques to hardware verification.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by National Key R&D Program of China under Grant No. 2023YFB4503804 and National Natural Science Foundation of China under Grant No. 62402210. Tian Tan, the co-corresponding author, is also supported by Xiaomi Foundation.

Data-Availability Statement

We have provided an artifact [35] that automatically reproduces all experimental results presented in Section 5 and includes the full open-source release of both ChiSA and ChiSABench as promised. The artifact is available at https://doi.org/10.5281/zenodo.17700253. To reproduce the results, please refer to the instructions provided in the accompanying `README.pdf` document within the artifact.

In addition, we have provided supplementary material [36] that presents the full specification of ChAIR and complete proofs of key theoretical results omitted from the paper due to space constraints. The supplementary material is available at https://doi.org/10.5281/zenodo.17623491.

References

[1] 2005. IEEE Standard for Verilog Hardware Description Language. doi:10.1109/IEEESTD.2006.99495 ISBN: 9780738148519.

[2] 2007. Static and Dynamic Analysis: Better Together. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 302–302. doi:10.1007/978-3-540-76637-7_20

[3] 2016. v2c – A Verilog to C Translator. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 580–586. doi:10.1007/978-3-662-49674-9_38 ISSN: 0302-9743, 1611-3349.

[4] 2021. Tests for ChiselTest Bounded Model Checker. https://github.com/ucb-bar/chiseltest/tree/v0.5.6/src/test/scala/chiseltest/formal/examples.

[5] 2023. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. doi:10.1109/ieeestd.2024.10458102 ISBN: 9798855705003.

[6] 2023. Quasar 2.0: Chisel equivalent of SweRV-EL2. https://github.com/Lampro-Mellon/Quasar/tree/2bc0985afd670d0c9b1b6983b53c3c2b340fcb5a.

[7] 2025. ScalaTest: A testing tool for Scala and Java developers. https://github.com/scalatest/scalatest.

[8] 2025. SiFive. https://www.sifive.com/.

[9] 2025. Stainless: Verification Framework and Tool for Higher-order Scala Programs. https://github.com/epfl-lara/stainless.

[10] 2025. SymbiYosys (sby): Front-end for Yosys-based Formal Verification Flows. https://github.com/YosysHQ/sby.

[11] 2025. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys.

[12] Chips Alliance. 2022. Firrtl: Flexible Intermediate Representation for RTL. https://github.com/chipsalliance/firrtl/tree/v1.5.6.

[13] Chips Alliance. 2023. Chisel: A Modern Hardware Design Language. https://github.com/chipsalliance/chisel/tree/v3.5.6.

[14] Chips Alliance. 2023. Treadle: A Chisel/Firrtl Execution Engine. https://github.com/chipsalliance/treadle/tree/v1.5.6.

[15] Krste Asanovic. 2020. Information on Coreplex IP Access. https://forums.sifive.com/t/information-on-coreplex-ip-access/105/8.

[16] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[17] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, San Francisco California, 1216–1225. doi:10.1145/2228360.2228584

[18] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7.* Technical Report. Department of Computer Science, The University of Iowa.

[19] Scott Beamer and David Donofrio. 2020. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, USA, 1–6. doi:10.1109/DAC18072.2020.9218632

[20] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.* 44, 10 (Oct. 2009), 243–262. doi:10.1145/1639949.1640108

[21] F. Brglez, D. Bryan, and K. Kozminski. 1989. Combinational profiles of sequential benchmark circuits. In *IEEE International Symposium on Circuits and Systems*. IEEE, Portland, OR, USA, 1929–1934. doi:10.1109/ISCAS.1989.100747

[22] Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, and Frederic Petrot. 2022. Toward Agile Hardware Designs With Chisel: A Network Use Case. *IEEE Des. Test* 39, 1 (Feb. 2022), 77–84. doi:10.1109/MDAT.2021.3063339

[23] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Technical Report UCB/EECS-2015-167. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[24] Vikas Chauhan, Neel Gala, and V. Kamakoti. 2016. ChADD: An ADD Based Chisel Compiler with Reduced Syntactic Variance. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*. IEEE, Kolkata, India, 499–504. doi:10.1109/VLSID.2016.44

[25] Qinlin Chen, Nairen Zhang, Jinpeng Wang, Tian Tan, Chang Xu, Xiaoxing Ma, and Yue Li. 2023. The Essence of Verilog: A Tractable and Tested Operational Semantics for Verilog. *Proc. ACM Program. Lang.* 7, OOPSLA2 (Oct. 2023), 234–263. doi:10.1145/3622805

[26] Chipsalliance. 2025. Verible: A Suite of SystemVerilog Developer Tools, including a Parser, Style-linter, Formatter and Language Server. https://github.com/chipsalliance/verible.

[27] Joonwon Choi, Jaewoo Kim, and Jeehoon Kang. 2025. Revamping Verilog Semantics for Foundational Verification. *Proc. ACM Program. Lang.* 9, OOPSLA2 (Oct. 2025), 950–977. doi:10.1145/3763084

[28] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. 2021. Wire sorts: a language abstraction for safe hardware composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 175–189. doi:10.1145/3453483.3454037

[29] Cristina Cifuentes, François Gauthier, Behnaz Hassanshahi, Padmanabhan Krishnan, and Davin McCall. 2023. The role of program analysis in security vulnerability detection: Then and now. *Computers & Security* 135 (Dec. 2023), 103463. doi:10.1016/j.cose.2023.103463

[30] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification*, Bertrand Meyer and Martin Nordio (Eds.). Vol. 7682. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–30. doi:10.1007/978-3-642-35746-6_1 Series Title: Lecture Notes in Computer Science.

[31] Chisel Community. 2025. How do I override the implicit clock or reset within a Module? https://www.chisel-lang.org/docs/cookbooks/cookbook#how-do-i-override-the-implicit-clock-or-reset-within-a-module.

[32] Chisel Community. 2025. Projects Using Chisel/FIRRTL. https://www.chisel-lang.org/community.

[33] Intel Corporation. 2025. Intel®Quartus®Prime Standard Edition User Guide. https://www.intel.com/content/www/us/en/docs/programmable/683323/18-1/avoid-combinational-loops.html.

[34] Perforce Corporation. 2025. How Static Analysis Automates Agile Software Development. https://www.perforce.com/resources/kw/static-analysis-automates-agile-software-development.

[35] Jiacai Cui, Qinlin Chen, Zhongsheng Zhan, Tian Tan, and Yue Li. 2025. *ChiSA: Static Analysis for Lightweight Chisel Verification (Artifact)*. doi:10.5281/zenodo.17700253

[36] Jiacai Cui, Qinlin Chen, Zhongsheng Zhan, Tian Tan, and Yue Li. 2025. ChiSA: Static Analysis for Lightweight Chisel Verification (Supplementary Material). doi:10.5281/zenodo.17623491

[37] Hans Jakob Damsgaard, Aleksandr Ometov, and Jari Nurmi. 2023. Verification of Approximate Hardware Designs with ChiselVerify. In *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, Aalborg, Denmark, 1–7. doi:10.1109/NorCAS58970.2023.10305474

[38] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 337–340. https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/

[39] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Y. Serhan Gener, Onur Demir, and Jakub Szefer. 2017. SecChisel: Language and Tool for Practical and Scalable Security Verification of Security-Aware Hardware Architectures. https://eprint.iacr.org/2017/193 Published: Cryptology ePrint Archive, Paper 2017/193.

[40] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y. Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. 2019. SecChisel Framework for Security Verification of Secure Processor Architectures. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, Phoenix AZ USA, 1–8. doi:10.1145/3337167.3337174

[41] Andrew Dobis, Hans Jakob Damsgaard, Enrico Tolotto, Kasper Hesse, Tjark Petersen, and Martin Schoeberl. 2022. Enabling Coverage-Based Verification in Chisel. In *2022 IEEE European Test Symposium (ETS)*. IEEE, Barcelona, Spain, 1–6. doi:10.1109/ETS54262.2022.9810435

[42] Andrew Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. 2023. Verification of Chisel Hardware Designs with ChiselVerify. *Microprocessors and Microsystems* 96 (Feb. 2023), 104737. doi:10.1016/j.micpro.2022.104737

[43] Andrew Dobis, Tjark Petersen, Hans Jakob Damsgaard, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. 2021. ChiselVerify: An Open-Source Hardware Verification Library for Chisel and Scala. In *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, Oslo, Norway, 1–7. doi:10.1109/NorCAS53631.2021.9599869

[44] Amelia Dobis, Tjark Petersen, and Martin Schoeberl. 2021. Towards Functional Coverage-Driven Fuzzing for Chisel Designs. (Nov. 2021). doi:10.3929/ETHZ-B-000539444 Medium: application/pdf,4 p. accepted version Publisher: [object Object].

[45] EETimes. 2016. A Match Made in Chip Verification Heaven: Simulation and Emulation. https://www.eetimes.com/a-match-made-in-chip-verification-heaven-simulation-and-emulation/.

[46] Weizhi Feng, Yicheng Liu, Jiaxiang Liu, David N Jansen, Lijun Zhang, and Zhilin Wu. 2024. Formally Verifying Arithmetic Chisel Designs for All Bit Widths at Once. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. ACM, San Francisco CA USA, 1–6. doi:10.1145/3649329.3657311

[47] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto Canada, 1583–1600. doi:10.1145/3243734.3243743

[48] Bruno Ferres, Olivier Muller, and Frédéric Rousseau. 2023. A Chisel Framework for Flexible Design Space Exploration through a Functional Approach. *ACM Trans. Des. Autom. Electron. Syst.* 28, 4 (July 2023), 1–31. doi:10.1145/3590769

[49] Harry Foster. 2024. IC/ASIC Functional Verification Trend Report - 2024. https://verificationacademy.com/topics/planning-measurement-and-analysis/2024-siemens-eda-and-wilson-research-group-functional-verification-study/.

[50] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, USA, 769–774. doi:10.1109/DAC18074.2021.9586216

[51] Youstina M. Halim, Khaled A. Ismail, Mohamed A. Abd El Ghany, Sameh A. Ibrahim, and Youssef M. Halim. 2022. Reinforcement-Learning Based Method for Accelerating Functional Coverage Closure of Traffic Light Controller Dynamic Digital Design. In *2022 32nd International Conference on Computer Theory and Applications (ICCTA)*. IEEE, Alexandria, Egypt, 44–50. doi:10.1109/ICCTA58027.2022.10206069

[52] Naoya Hatta. 2025. SVLint: SystemVerilog Linter. https://github.com/dalance/svlint.

[53] John L. Hennessy and David A. Patterson. 2018. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 27–29. doi:10.1109/ISCA.2018.00011

[54] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. doi:10.1145/3282307 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[55] Jim Hogan. 2013. The Science of SW Simulators, Acceleration, Prototyping, Emulation. https://www.deepchip.com/items/0522-02.html.

[56] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2022. Hardware Information Flow Tracking. *ACM Comput. Surv.* 54, 4 (May 2022), 1–39. doi:10.1145/3447867

[57] Charles Hymans. 2002. Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation. In *Static Analysis* (Berlin, Heidelberg), Manuel V. Hermenegildo and Germán Puebla (Eds.). Springer, 444–460. doi:10.1007/3-540-45789-5_31

[58] Charles Hymans. 2003. Design and Implementation of an Abstract Interpreter for VHDL. In *Correct Hardware Design and Verification Methods* (Berlin, Heidelberg), Daniel Geist and Enrico Tronci (Eds.). Springer, 263–269. doi:10.1007/978-3-540-39724-3_23

[59] Charles Hymans. 2005. Verification of an Error Correcting Code by Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg), Radhia Cousot (Ed.). Springer, 330–345. doi:10.1007/978-3-540-30579-8_22

[60] Vincent Iampietro. 2022. Formal Semantics of H-VHDL. (May 2022). https://hal.science/hal-03664656

[61] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. doi:10.1145/503502.503505

[62] SiFive Inc. 2025. SiFive TileLink Specification. https://www.sifive.com/documentation/tilelink/tilelink-spec/.

[63] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Irvine, CA, 209–216. doi:10.1109/ICCAD.2017.8203780

[64] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Vol. 5673. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255. doi:10.1007/978-3-642-03237-0_17 Series Title: Lecture Notes in Computer Science.

[65] Sid Katzen. 2001. The PIC16F84 Microcontroller. In *The Quintessential PIC Microcontroller*, A. J. Sammes (Ed.). Springer London, London, 77–104. doi:10.1007/978-1-4471-3704-7_4 Series Title: Computer Communications and Networks.

[66] Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. 2017. *Data Flow Analysis: Theory and Practice* (1 ed.). CRC Press. doi:10.1201/9780849332517

[67] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanovic. 2018. DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Dublin, Ireland, 76–764. doi:10.1109/FPL.2018.00021

[68] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanovicc. 2016. Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Seoul, South Korea, 128–139. doi:10.1109/ISCA.2016.21

[69] Stephen Cole Kleene. 1952. *Introduction to Metamathematics*. North-Holland Publishing Company, Amsterdam.

[70] Anish Krishnakumar, Umit Ogras, Radu Marculescu, Mike Kishinevsky, and Trevor Mudge. 2023. Domain-Specific Architectures: Research Problems and Promising Approaches. *ACM Trans. Embed. Comput. Syst.* 22, 2 (March 2023), 1–26. doi:10.1145/3563946

[71] Shriram Krishnamurthi. 2015. Desugaring in Practice: Opportunities and Challenges. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (Mumbai, India) *(PEPM '15)*. Association for Computing Machinery, New York, NY, USA, 1–2. doi:10.1145/2678015.2678016

[72] Shriram Krishnamurthi, Benjamin S. Lerner, and Liam Elberty. 2019. The Next 700 Semantics: A Research Challenge. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:14. doi:10.4230/LIPIcs.SNAPL.2019.9

[73] Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2021. Open-Source Formal Verification for Chisel. (2021).

[74] Chris Lattner. 2025. CIRCT: Circuit IR Compilers and Tools. https://circt.llvm.org/.

[75] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Seoul, Korea (South), 2–14. doi:10.1109/CGO51591.2021.9370308

[76] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. 2015. *The Hwacha Microarchitecture Manual, Version 3.8.1.* Technical Report UCB/EECS-2015-263. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-263.html

[77] 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. 2016. An Agile Approach to Building RISC-V Microprocessors. *IEEE Micro* 36, 2 (March 2016), 8–20. doi:10.1109/MM.2016.11

[78] Harry Foster Lionel Bening. 2002. *RTL Logic Simulation.* Kluwer Academic Publishers, Boston, 69–101. doi:10.1007/0-306-47631-2_5

[79] Andreas Lööw. 2025. The Simulation Semantics of Synthesisable Verilog. *Proc. ACM Program. Lang.* 9, OOPSLA1 (April 2025), 1295–1320. doi:10.1145/3720484

[80] Raffaele Meloni, H. Peter Hofstee, and Zaid Al-Ars. 2024. Tywaves: A Typed Waveform Viewer for Chisel. In *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, Lund, Sweden, 1–6. doi:10.1109/NorCAS64408.2024.10752465

[81] Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. 47–48 pages. http://cs.au.dk/~amoeller/spa/ Department of Computer Science, Aarhus University.

[82] Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. 79–87 pages. http://cs.au.dk/~amoeller/spa/ Department of Computer Science, Aarhus University.

[83] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. 2016. *PKCS #1: RSA Cryptography Specifications Version 2.2.* Technical Report RFC8017. RFC Editor. RFC8017 pages. doi:10.17487/RFC8017

[84] R Mukherjee. 2018. *Precise abstract interpretation of hardware designs.* PhD Thesis. University of Oxford.

[85] National Institute of Standards and Technology (US). 2023. *Advanced Encryption Standard (AES).* Technical Report NIST FIPS 197-upd1. National Institute of Standards and Technology (U.S.), Washington, D.C. NIST FIPS 197–upd1 pages. doi:10.6028/NIST.FIPS.197-upd1

[86] M. H. A. Newman. 1942. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43, 2 (1942), 223–243. http://www.jstor.org/stable/1968867

[87] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Ernst-Rüdiger Olderog, and Bernhard Steffen (Eds.). Vol. 1710. Springer Berlin Heidelberg, Berlin, Heidelberg, 114–136. doi:10.1007/3-540-48092-7_6 Series Title: Lecture Notes in Computer Science.

[88] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. 2018. Btor2 , BtorMC and Boolector 3.0. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Vol. 10981. Springer International Publishing, Cham, 587–595. doi:10.1007/978-3-319-96145-3_32 Series Title: Lecture Notes in Computer Science.

[89] Rachit Nigam, Pedro Henrique Azevedo De Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI (June 2023), 343–367. doi:10.1145/3591234

[90] Rachit Nigam, Ethan Gabizon, Edmund Lam, and Adrian Sampson. 2024. Correct and Compositional Hardware Generators. doi:10.48550/ARXIV.2401.02570 Version Number: 1.

[91] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2021. An Empirical Study on Type Annotations: Accuracy, Speed, and Suggestion Effectiveness. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (April 2021), 1–29. doi:10.1145/3439775

[92] John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. 2018. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Montpellier France, 190–201. doi:10.1145/3238147.3238173

[93] Yan Pi, Hongji Zou, Tun Li, Wanxia Qu, and Hai Wan. 2023. ESFO: Equality Saturation for FIRRTL Optimization. In *Proceedings of the Great Lakes Symposium on VLSI 2023*. ACM, Knoxville TN USA, 581–586. doi:10.1145/3583781.3590239

[94] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/2384577.2384579

[95] Michael Popoloski. 2025. Slang: SystemVerilog Language Services. https://github.com/MikePopoloski/slang.

[96] Md Imtiaz Rashid and B. Carrion Schaefer. 2024. VeriPy: A Python-Powered Framework for Parsing Verilog HDL and High-Level Behavioral Analysis of Hardware. In *2024 IEEE 17th Dallas Circuits and Systems Conference (DCAS)*. IEEE, Richardson, TX, USA, 1–6. doi:10.1109/DCAS61159.2024.10539889

[97] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. 2020. TACAI: an intermediate representation based on abstract interpretation. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. ACM, London UK, 2–7. doi:10.1145/3394451.3397204

[98] Berkeley Architecture Research. 2023. IceNet: A library of Chisel designs related to networking. https://chipyard.readthedocs.io/en/1.10.0/Generators/IceNet.html.

[99] Berkeley Architecture Research. 2023. RiscvMini: Simple RISC-V 3-stage Pipeline in Chisel. https://github.com/ucb-bar/riscv-mini/tree/3473cfd8c0ebca6593d3c324209b0f5a7e582802.

[100] Berkeley Architecture Research. 2023. Sodor Processor Collection: Educational Microarchitectures for Risc-V ISA. https://github.com/ucb-bar/riscv-sodor/tree/sodor-old.

[101] Berkeley Architecture Research. 2025. ChiselTest: The batteries-included testing and formal verification library for Chisel-based RTL designs. https://github.com/ucb-bar/chiseltest.

[102] Hassan Salmani, Mohammad Tehranipoor, and Ramesh Karri. 2013. On design vulnerability analysis and trust benchmarks development. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, Asheville, NC, USA, 471–474. doi:10.1109/ICCD.2013.6657085

[103] Martin Schoeberl. 2025. *Digital Design with Chisel*. Kindle Direct Publishing. https://github.com/schoeberl/chisel-book

[104] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. 2017. Benchmarking of Hardware Trojans and Maliciously Affected Circuits. *J Hardw Syst Secur* 1, 1 (March 2017), 85–102. doi:10.1007/s41635-017-0001-6

[105] Shidong Shen, Yicheng Liu, Lijun Zhang, Fu Song, and Zhilin Wu. 2025. Formal Verification of RISC-V Processor Chisel Designs. In *Dependable Software Engineering. Theories, Tools, and Applications*, Timothy Bourke, Liqian Chen, and Amir Goharshady (Eds.). Vol. 15469. Springer Nature Singapore, Singapore, 142–160. doi:10.1007/978-981-96-0602-3_8 Series Title: Lecture Notes in Computer Science.

[106] Wilson Snyder. 2025. Verilator. https://veripool.org/verilator.

[107] Synopsys, Inc. 2025. Spyglass Lint. https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-lint.html

[108] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing*, Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz (Eds.). Vol. 9040. Springer International Publishing, Cham, 451–460. doi:10.1007/978-3-319-16214-0_42 Series Title: Lecture Notes in Computer Science.

[109] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle WA USA, 1093–1105. doi:10.1145/3597926.3598120

[110] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. 2010. A case for FAME: FPGA architecture model execution. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 290–301. doi:10.1145/1816038.1815999

[111] Alfred Tarski. 1955. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific J. Math.* 5, 2 (June 1955), 285–309.

[112] Patrick Thomson. 2022. Static analysis. *Commun. ACM* 65, 1 (Jan. 2022), 50–54. doi:10.1145/3486592

[113] Lenny Truong and Pat Hanrahan. 2019. 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) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:21. doi:10.4230/LIPIcs.SNAPL.2019.7 ISSN: 1868-8969.

[114] Sheng-Hong Wang, Hunter James Coffman, Kenneth Mayer, Sakshi Garg, and Jose Renau. 2023. A Multi-threaded Fast Hardware Compiler for HDLs. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. ACM, Montréal QC Canada, 25–36. doi:10.1145/3578360.3580254

[115] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-based Hardware Verification. *ACM Comput. Surv.* 54, 11s (Jan. 2022), 1–33. doi:10.1145/3510578

[116] Remigiusz Wiśniewski, Arkadiusz Bukowiec, and Marek Wegrzyn. 2001. Benefits of Hardware Accelerated Simulation. (June 2001).

[117] Mufan Xiang, Yongjian Li, and Yongxin Zhao. 2023. ChiselFV: A Formal Verification Framework for Chisel. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Antwerp, Belgium, 1–6. doi:10.23919/DATE56975.2023.10137221

[118] Mufan Xiang, Yongjian Li, and Yongxin Zhao. 2023. RVFC: RISC-V Formal in Chisel. In *2023 International Symposium of Electronics Design Automation (ISEDA)*. IEEE, Nanjing, China, 162–167. doi:10.1109/ISEDA59274.2023.10218484

[119] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Chicago, IL, USA, 1178–1199. doi:10.1109/MICRO56248.2022.00080

[120] Jones Yeboah and Saheed Popoola. 2023. Efficacy of Static Analysis Tools for Software Defect Detection on Open-Source Projects. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, Las Vegas, NV, USA, 1588–1593. doi:10.1109/CSCI62032.2023.00262

[121] Shizhen Yu, Yifan Dong, Jiuyang Liu, Yong Li, Zhilin Wu, David N. Jansen, and Lijun Zhang. 2022. CHA: Supporting SVA-Like Assertions in Formal Verification of Chisel Programs (Tool Paper). In *Software Engineering and Formal Methods*, Bernd-Holger Schlingloff and Ming Chai (Eds.). Vol. 13550. Springer International Publishing, Cham, 324–331. doi:10.1007/978-3-031-17108-6_20 Series Title: Lecture Notes in Computer Science.

[122] Bowen Zhang, Wei Chen, Hung-Chun Chiu, and Charles Zhang. 2024. Unveiling the Power of Intermediate Representations for Static Analysis: A Survey. doi:10.48550/ARXIV.2405.12841 Version Number: 1.

[123] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *SIGPLAN Not.* 50, 4 (May 2015), 503–516. doi:10.1145/2775054.2694372

[124] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanovic. 2022. Constellation: An Open-Source SoC-Capable NoC Generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. IEEE, Chicago, IL, USA, 1–7. doi:10.1109/NoCArc57472.2022.9911299

Received 2025-07-10; accepted 2025-11-06