Qihe: A General-Purpose Static Analysis Framework for Verilog

QINLIN CHEN, Nanjing University, China
NAIREN ZHANG, Nanjing University, China
JINPENG WANG, Nanjing University, China
JIACAI CUI, Nanjing University, China
TIAN TAN*, Nanjing University, China
XIAOXING MA, Nanjing University, China
CHANG XU, Nanjing University, China
JIAN LU, Nanjing University, China
YUE LI*, Nanjing University, China

In the past decades, static analysis has thrived in software, facilitating applications in bug detection, security, and program understanding. These advanced analyses are largely underpinned by general-purpose static analysis frameworks, which offer essential infrastructure to streamline their development. The generality of these frameworks is often founded on components such as a front end for program conversion, an intermediate representation (IR), and a suite of fundamental analyses providing key program information. Soot exemplifies such a framework in Java, sparking thousands of analyses and impacting both academia and industry. Conversely, hardware lacks such a framework, which may limit the exploration of diverse static analysis applications in this field. This gap could overshadow the promising opportunities for sophisticated static analysis in hardware, hindering achievements akin to those witnessed in software. To advance static analysis in hardware, we introduce Qihe, the first general-purpose static analysis framework for Verilog—a highly challenging endeavor given the absence of precedents in hardware. Qihe features an analysis-oriented front end, a Verilog-specific IR, and a suite of diverse fundamental analyses that capture essential hardware-specific characteristics—such as bit-vector arithmetic, register synchronization, and digital component concurrency—and enable the examination of intricate hardware data and control flows. These fundamental analyses are specifically designed to support a wide array of analysis clients for hardware. To validate Qihe's utility, we further developed a series of analysis clients spanning bug detection, security, and program understanding, and applied them to real-world hardware projects. Unlike software, real-world hardware projects tend to contain fewer but harder-to-detect bugs, as they typically undergo extensive simulation and rigorous verification to prevent the prohibitive costs of hardware defects. Despite this, our preliminary experimental results are highly promising; for example, Qihe uncovered 9 previously unknown bugs in popular real-world hardware projects (averaging 1.5K+ GitHub stars), all of which were subsequently confirmed by developers; moreover, Qihe successfully identified 18 bugs beyond the capabilities of existing static analyses for Verilog bug detection (i.e., linters), and detected 16 vulnerabilities in real-world hardware programs. These results underscore the transformative potential of static analysis in hardware design. By open-sourcing Qihe[1], which comprises over 100K lines of code, together with its carefully curated and comprehensive documentation[2], we aim to inspire further innovation and applications of sophisticated static analysis for hardware, aspiring to foster a similarly vibrant ecosystem that software analysis enjoys.

---

*Corresponding author.

[1]https://qihe.pascal-lab.net
[2]https://qihe-docs.pascal-lab.net

Authors' Contact Information: Qinlin Chen, qinlinchen@smail.nju.edu.cn, Nanjing University, China; Nairen Zhang, nairen@smail.nju.edu.cn, Nanjing University, China; Jinpeng Wang, jpwang@smail.nju.edu.cn, Nanjing University, China; Jiacai Cui, jiacaicui@smail.nju.edu.cn, Nanjing University, China; Tian Tan, tiantan@nju.edu.cn, Nanjing University, China; Xiaoxing Ma, xxm@nju.edu.cn, Nanjing University, China; Chang Xu, changxu@nju.edu.cn, Nanjing University, China; Jian Lu, lj@nju.edu.cn, Nanjing University, China; Yue Li, yueli@nju.edu.cn, and all authors are also affiliated with the State Key Laboratory for Novel Software Technology, Nanjing University, China.

1   Introduction

In the software domain, static analysis is widely recognized as an effective technique for enhancing software quality. Over the years, numerous sophisticated static analyses have been developed to achieve various application tasks, such as detecting bugs [55], identifying security vulnerabilities [7], and aiding in program understanding [46].

Despite its proven successes in software, static analysis remains underutilized in hardware, particularly with Verilog, the predominant hardware description language (HDL) [29]. The current situation mirrors the early days of software static analysis, where its applications were limited to compiler-integrated optimizations and basic syntax checks (e.g., early linting tools), with more sophisticated software analyses, such as those for bug detection and security, yet to emerge. In hardware, static analysis techniques are primarily adopted by hardware synthesizers (e.g., Yosys [79]) for compiler optimization purposes; linting tools [16, 32, 57, 66] are popularly used to enforce code-style or syntactic checks but lack the sophistication needed to detect deeper design flaws like improper register resets and hardware Trojans.

We argue that one of the main reasons for the stagnation of hardware static analysis is the lack of *a general-purpose framework* that provides the necessary infrastructure to enable the development of sophisticated analyses. In the software domain, such frameworks have proven transformative. For example, Soot [73], a general-purpose static analysis framework for Java, has catalyzed thousands of analysis works in both academia and industry. Research efforts like FlowDroid [7], widely used for detecting security vulnerabilities in Android apps, rely on Soot's fundamental analyses for identifying control and data flows statically. Some companies also leverage FlowDroid and Soot to develop their own analyses, which allows them to identify bugs early in the development cycle, leading to potentially significant cost savings.

Drawing from the designs of existing frameworks for software analysis [24, 69, 73], a *general-purpose* framework typically requires the close cooperation of the following key components: (1) *fundamental analyses* that provide basic program information used across various application-specific analyses, (2) *a front end* and *an IR* that derive an analysis-oriented program abstraction from source code, and (3) *an analysis manager* that facilitates the reuse of results from existing analyses and the integration of new ones.

However, given the absence of a general-purpose framework in hardware, the development of hardware static analysis typically relies on one of the following three kinds of options, each with its own limitations. Linters such as Slang [57], Verible [16], SVLint [32], and Verilator-Lint [66] are confined to syntax- or pattern-based analyses on abstract syntax trees (ASTs). Tools like Pyverilog [68] and VeriPy [58] provide hardware abstractions beyond ASTs but lack the infrastructure to address critical hardware features like synchronization and concurrency, highly limiting their utility for diverse analysis needs. While CIRCT [43], a comprehensive hardware compiler framework primarily used for synthesis, can also be adapted for static analyses, it lacks the fundamental facilities necessary for building sophisticated analyses that require intricate value flows and deep semantic reasoning, resulting in substantial development costs.

In this paper, we present Qihe, the first general-purpose static analysis framework for Verilog, comprising three key components:

- *A Set of Fundamental Analyses* (Section 3): We have developed 22 fundamental analyses to statically construct the hardware program information necessary to support a wide array of analysis clients. This unique contribution stems from our decade-long experience in frontline research and development in foundational static software analysis, sustained communication with industrial hardware practitioners, and an in-depth understanding of Verilog semantics.

- *An Analysis-Oriented IR and Front End* (Section 4): We have designed the IR and front end to work collaboratively, meeting specific analysis needs such as supporting analysis for incomplete hardware programs and retaining additional source information, while still ensuring the IR remains simple for ease of analysis.
- *An Analysis Manager* (Section 5): We have designed an engineering-focused mechanism that enables new developers to conveniently integrate and execute new analyses within Qihe.

To validate Qihe as a general-purpose framework capable of supporting a variety of application-specific analyses, we developed 20 client analyses, which also serve as examples to guide future developers in creating their own. These analyses target common but non-trivial needs identified through surveys on hardware bugs and vulnerabilities [17, 18, 48, 52, 59, 64] along with insights from industrial hardware practitioners, spanning diverse application domains, including bug detection, security analysis, and program understanding.

We conducted experiments by applying these client analyses to a set of real-world Verilog programs, including System-on-Chip (SoC) designs, encryption modules, communication protocols (e.g., AXI implementations), and more [28, 49, 54, 56, 59, 60, 64, 81]. The preliminary experimental results are summarized as follows:

- *Hardware Bug Detection*: Unlike software, hardware tends to contain fewer but harder-to-detect bugs, since it typically undergoes extensive simulation and rigorous verification to avoid the prohibitive costs of defects [6, 8]. Despite this, our analyses uncovered 9 previously unknown bugs in popular real-world hardware projects (averaging over 1.5K GitHub stars), all of which were subsequently confirmed by the developers. Examples include registers not correctly reset in a RISC-V CPU, an unreachable state in a CRC generator, unexpected bit truncation in a SHA512 encryption module, etc. Our analyses also successfully detected another 9 hardware bugs: some were collected from commit histories of real-world programs by previous surveys, while others were summarized by industrial hardware experts and injected into real programs as benchmarks [17, 18, 48]. Note that all 18 bugs, spanning eight categories as summarized in Table 2, cannot be detected by existing static analyses for Verilog bug detection [16, 32, 57, 66].
- *Hardware Security Analysis*: Our analyses detected 15 vulnerabilities related to information leakage, particularly involving the exposure of secret encryption keys via observable output ports like antennas and capacitance. Additionally, a hardware Trojan [41] is also detected that exploits the special X value in Verilog for concealment.
- *Hardware Program Understanding*: Our analysis helps developers understand the semantics of Verilog synthesis by predicting their physical implementation details, such as whether a variable is mapped to a clock signal, latch, or register, with high recall and precision. For example, to minimize latency, clocks should avoid being used in complex logic in the code, but these clock usages are often identified after a time-consuming synthesis process. With static analysis, these issues can be identified earlier, such as during the development phase, thereby reducing potential high engineering costs.

Moreover, from the preliminary experimental results, we highlight three complementary points. Firstly, existing static analysis techniques for hardware struggle to detect the aforementioned bugs and vulnerabilities due to their inability to reason about complex value flows within hardware programs. This underscores the necessity for sophisticated analyses to uncover deep hardware issues, highlighting the role of Qihe as a foundational framework for advancing hardware analysis.

Secondly, the detection of these deep hardware issues has traditionally relied on resource-intensive methods such as model checking and fuzzing, as well as labor-intensive techniques like simulation-based testing [12, 23, 38, 45, 70, 83], which often dominate and slow down the hardware development process. In contrast, static analysis can be deployed at an earlier stage and is often
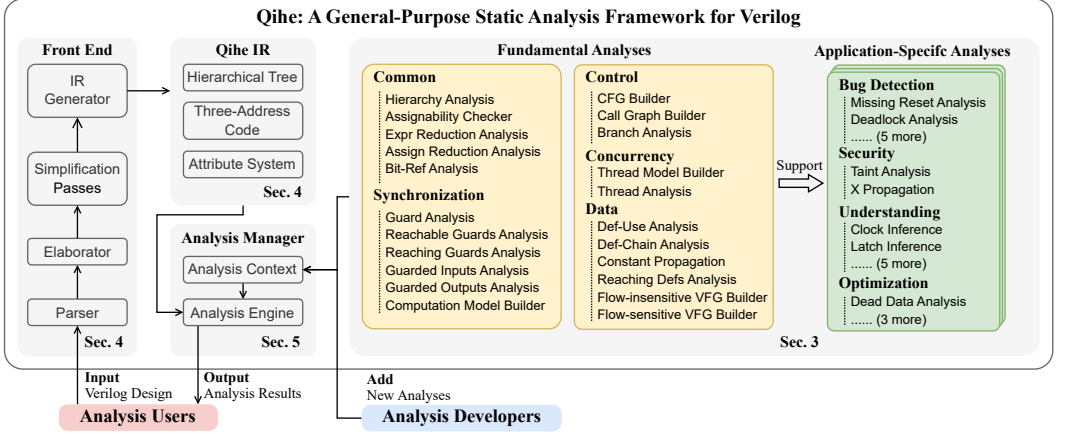
Fig. 1. Workflow of Qihe. It supports both *analysis users* in executing analyses on Verilog designs and *developers* in creating new analyses, enabled by the collaboration of its components.

considerably more lightweight. For instance, on a real-world RISC-V SoC codebase comprising 1.8 million lines, Qihe can complete all 20 client analyses, along with 22 fundamental analyses, in approximately 5 minutes using 40GB of memory.

Finally, the development cost for these client analyses is significantly reduced thanks to the infrastructure provided by the Qihe framework, particularly its extensive suite of fundamental analyses. On average, each client analysis requires only about 300 lines of Java code when implemented leveraging Qihe's infrastructure, whereas developing the same analysis from scratch—without utilizing Qihe's fundamental analyses—would require over 5,000 lines of code on average.

In summary, this paper makes a singular contribution by introducing Qihe, the first general-purpose static analysis framework for Verilog, consisting of over 100K lines of Java code. Qihe is open-source and can be obtained from its website[3]. We also provide comprehensive documentation[4] to support the development of diverse application-specific analyses.

The growth of hardware analysis may not mirror the widespread success of software analysis. Yet, over 25 years ago, it was also hard to foresee that static analysis would help enhance software quality so significantly. Qihe is a bold venture, uncovering the initial potential of sophisticated hardware analysis. We hope it serves as a catalyst for collaboration between software and hardware communities to jointly shape the still-blurry yet promising future of hardware analysis.

2 Overview of Qihe

Qihe, as a general-purpose static analysis framework for Verilog, is designed to address two key use cases: enabling *analysis users* to execute available analyses on their hardware designs and empowering *analysis developers* to create diverse new analyses using the Qihe framework. Figure 1 illustrates the underlying workflow for each use case, supported by four core components: the front end, Qihe IR, diverse analyses, and the analysis manager. Below, we describe this workflow in detail, guided by the figure. Further details about Qihe's components are discussed in later sections.

For analysis users, the process begins by inputting Verilog designs into the *front end*, which converts them into *Qihe IR*, a program abstraction for Verilog. The *analysis manager* then executes
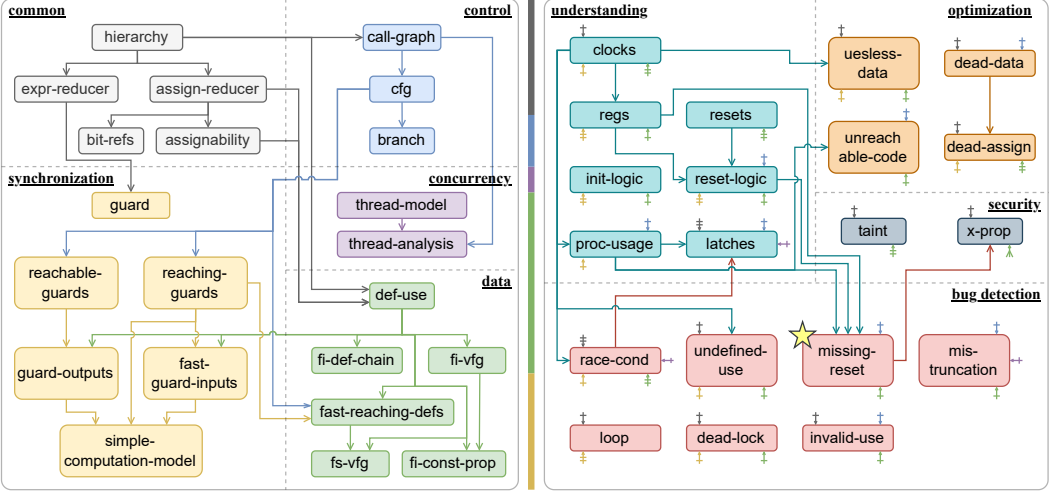

Fig. 2. Analyses and their dependencies in Qihe. Each analysis is represented by its short name. The left section presents fundamental analyses, categorized into five groups, while the right section illustrates application-specific analyses across four domains. Arrows depict dependency relations between these analyses. To maintain clarity, we use short arrows ("⬆", "⬆", and "⬆") to succinctly represent the number of dependencies that an application-specific analysis has on fundamental analyses. These arrows denote dependencies on one, two, or three fundamental analyses within a specific category, with the category identified by the arrow's color (or position in black-white printing). For example, the starred analysis missing-reset has two short arrows "⬆": one top-right and one bottom-right, signifying its dependency on *one* analysis from the *control* category and *one* from the *data* category. We use the central bar to visualize the relative usage of each category by color and length.

the request analyses, as configured by users, on the Qihe IR and delivers the results to users. Specifically, the manager detects available analyses and configurations, registers them into an analysis context, and invokes an analysis engine to apply them to the Qihe IR. Analyses are executed in topological order based on their dependency relations to ensure correctness.

For analysis developers, the *analysis manager* provides an intuitive interface for creating new analyses (whether fundamental or application-specific) based on existing ones. Once new analyses are implemented, the manager automatically detects and registers the new analysis into the analysis context, making it available for users to execute. We encourage developers to leverage Qihe to explore and contribute more potentially useful analyses.

3 Analysis Suite

Qihe supports the development of diverse sophisticated analyses by utilizing the cooperation among various analyses. Figure 2 depicts an overview of all the major analyses in Qihe and their dependencies, illustrating such cooperation. Note that the dependencies between fundamental analyses and client analyses are represented by the special arrows described in the figure caption.

Guided by this figure, Section 3.1 first explains how the analyses are derived and categorized. Section 3.2 then elaborates on missing-reset, a representative bug detection client in Qihe, to demonstrate the sophistication required to detect real-world hardware bugs that elude existing

linter-style static analyses for Verilog bug detection [16, 32, 57, 66] and highlight the fundamental differences between hardware analysis and typical software analysis. Finally, Section 3.3 discusses how existing analyses in our framework collaborate to support this critical hardware bug detection client.

3.1   Overview of All Analyses

Qihe offers a rich suite of fundamental and application-specific analyses. Since no general-purpose static analysis framework for Verilog exists as a reference, we derived these analyses through a three-step process that leverages surveys, research experience, and engineering practices: (1) We identified analysis needs by surveying hardware bugs and security issues [17, 18, 48, 52, 59, 64], examining existing analyses in synthesis and linting tools [32, 43, 57, 79], and directly communicating with industrial hardware practitioners [3, 4]; (2) Building on this survey and leveraging our decade-long research experience in fundamental static analysis for software, we distilled *22 fundamental analyses* providing essential Verilog program information commonly required across diverse application needs; and (3) To demonstrate how to build diverse applications by leveraging fundamental analyses in Qihe, we additionally developed *20 application-specific analyses* to address common but non-trivial needs identified in our survey.

To facilitate a clear understanding of these analyses, we categorize them by their design purpose, as shown in Figure 2. Fundamental analyses (on the left) are grouped into five categories based on their focus on specific aspects of hardware design: *common* language features, *data* and *control* flows, and *concurrency* and *synchronization* characteristics. Application-specific analyses (on the right) are categorized into four popular application domains: hardware *bug detection*, *security* vulnerability identification, program *understanding*, and program *optimization*.

To enhance analysis modularity and reusability, each analysis in Qihe is designed with a single purpose and delegates other responsibilities to upstream analysis as dependencies. This practice naturally forms intricate dependencies, as depicted in Figure 2.

For a deeper understanding of these analyses and their dependencies, we next elaborate on a representative analysis in Qihe (Section 3.2) and then discuss its supporting dependencies (Section 3.3).

3.2   Sophisticated Analysis for Hardware: A Motivating Example

This section presents a motivating example that illustrates three key questions: (1) What do hardware bug-detection problems look like, and how do they differ from their software analogues? (2) How to design sophisticated hardware static analyses that leverage hardware-specific insights to address these problems? (3) What is the challenge in building sophisticated hardware static analysis?

Our motivating example is the `missing-reset` analysis, which detects a class of critical hardware bugs where registers fail to initialize properly during circuit reset, creating unstable system states that attackers could potentially exploit [5, 20, 48, 50]. It serves as a representative Verilog analysis in our analysis suite that not only handles diverse hardware-specific semantics—such as clocks, registers, and resets—that are uncommon in software analysis, but also demonstrates the level of sophistication required to detect real-world hardware bugs (summarized in Table 2) that elude existing linter-style static analyses for Verilog bug detection [16, 32, 57, 66].

Below, we first introduce the *missing-reset* problem in Verilog (Section 3.2.1) and distinguish it from the classical *use-before-initialization* (UBI) problem in software (Section 3.2.2). We then present the core insights underlying our `missing-reset` analysis for addressing this problem (Section 3.2.3). Finally, we use this analysis to discuss the key challenge in developing sophisticated hardware analyses (Section 3.2.4).

3.2.1 The Missing-Reset Problem in Verilog. The *missing-reset* problem occurs when hardware *registers* lack proper *reset*. Below, we first provide the necessary Verilog background on *registers* and *reset*, and then present the *problem formulation*.

Registers. A hardware register is a digital circuit compo-
nent that stores and updates values under the control of
a driving clock signal alternating between 0 and 1; Verilog
uses variables and always blocks to model hardware reg-
isters. As exemplified in the code snippet of Figure 3, acc
is a hardware register whose driving clock is specified in
the always block header (line 1) and whose value update
logic is specified in the block body (lines 2–5). Functionally,
this Verilog program describes an accumulator where reg-
ister acc updates to 0 when reset is 1 (lines 2–3) and to
acc + in otherwise (lines 4–5), whenever the driving clock
signal clock transitions from 0 to 1, known as a positive
clock edge (as specified by the posedge keyword in line 1).
To illustrate this dynamic behavior, the table in Figure 3
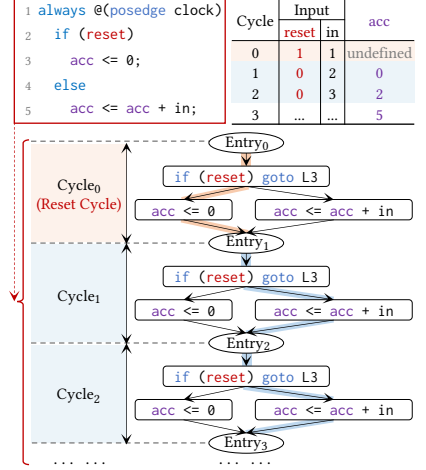shows an example of how variable values evolve over time:

Fig. 3. A properly-reset register example.

- The "Cycle" column enumerates clock cycles, where the
  number i indicates that this row shows the values of
  all variables during the i-th *clock cycle*, denoted Cycle_i
  hereafter. Here, a clock cycle is the time interval between consecutive positive edges of the clock
  signal clock.

- The "Input" columns provide the values of input variables such as reset and in, which are not
  defined within this code snippet but are externally supplied each cycle (e.g., by toggling a switch
  or pressing a button during circuit execution).

- The "acc" column shows the value of register acc over time. For simplicity, we use x_i to denote
  the value of variable x during Cycle_i. Initially, all registers hold undefined values at Cycle_0, i.e.,
  $\text{acc}_0 = \text{undefined}$ in this example. During each cycle, lines 2–5 execute, and the *new register
  value takes effect in the next cycle*, reflecting the special semantics of Verilog's non-blocking
  assignment operator <=, which models clock-driven updates of hardware registers—unlike
  software assignments. For example, (1) during Cycle_0, acc <= 0 (line 3) executes since $\text{reset}_0 = 1$,
  making $\text{acc}_1 = 0$; (2) during Cycle_1, acc <= acc + in (line 5) executes since $\text{reset}_1 = 0$, giving
  $\text{acc}_2 = \text{acc}_1 + \text{in}_1 = 0 + 2 = 2$; (3) during Cycle_2, the same update applies again, yielding
  $\text{acc}_3 = \text{acc}_2 + \text{in}_2 = 2 + 3 = 5$.

For further clarity, the lower portion of Figure 3 presents a cycle-expanded control flow graph
(CFG) that visualizes this dynamic behavior by highlighting execution paths in orange and blue. In
this representation, Entry_i denotes entry to the always block body (lines 2–5) during Cycle_i.

Reset. The *reset signal* represents one of the most fundamental control signals critical for *reliable*
behavior in digital design [76], ensuring circuits begin operation from a known initial state rather
than the default undefined state at power-up. For example, in Cycle_0 of Figure 3 (highlighted
in orange), reset_0 is set to 1, ensuring that normal operation of the accumulator begins with a
determined acc = 0 instead of the undefined acc = undefined. We refer to the cycle in which the
reset signal is active (i.e., reset = 1) as the *reset cycle*. This reset cycle is essential for reliable normal
operation: only after that cycle can we guarantee that during any subsequent non-reset Cycle_i
(where $i \geq 1$ and $\text{reset}_i = 0$), the accumulator *reliably* maintains the invariant $\text{acc}_{i+1} = \sum_{j=1}^{i} \text{in}_j$.

(a) A simple missing-reset example.

(b) The hardware dependency graph of (a).

(c) A real-world missing-reset example [48] containing the first two bugs in our evaluation (Table 2).
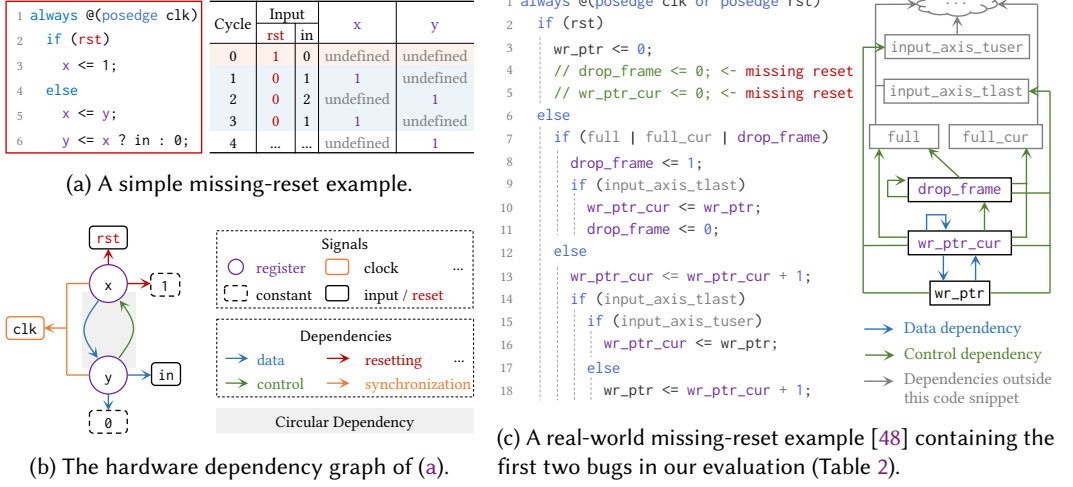
Fig. 4. Missing-reset examples and their hardware dependency graphs.

By convention, any *legal execution* of a circuit with reset capability should begin with a reset cycle to establish an initial state, followed by non-reset cycles for normal operation, as shown in Figure 3. This creates a clear division of responsibility: hardware developers should ensure their Verilog programs function reliably under all legal executions, while hardware users should perform legal executions (i.e., reset before normal operation). Consequently, the primary bugs of interest to developers are those that can occur during legal executions, as these fall within their expected domain of responsibility.

Problem Formulation. The *missing-reset* problem occurs on a register if there exists a *legal execution* path (i.e., a reset cycle followed by non-reset cycles, as illustrated in Figure 3) in which this register's undefined value is *infinitely often* used. "Infinitely often" is a standard term in mathematics [22]. In our context, it means that for a register x, after any given Cycle_N, the undefined value of x will be used again in some later cycle Cycle_n where $n > N$. This problem is particularly dangerous because it implies that a register may unpredictably hold undefined values at any time, even during a legal execution, undermining hardware reliability and potentially leading to fatal functional flaws. In contrast, finite occurrences of undefined values are common and acceptable in hardware (e.g., in pipelined designs [75]), and such transient behavior should not be considered a bug.

Figure 4a illustrates a missing-reset example in which registers x and y both exhibit this problem, as shown in the table on the right. In the reset cycle Cycle_0 (shaded in orange), line 3 executes, updating x from undefined to 1 in Cycle_1, while y remains undefined because it is not assigned in this cycle. In each non-reset cycle Cycle_i ($i \geq 1$, shaded in blue), lines 5–6 execute, resulting in the value updates $x_{i+1} = y_i$ and $y_{i+1} = x_i$? $\text{in}_i : 0$, according to the Verilog non-blocking assignment semantics introduced in the *Registers* paragraph. More concretely, after Cycle_1, we have $x_2 = y_1 = \text{undefined}$ and $y_2 = (x_1 ? \text{in}_1 : 0) = (1 ? 1 : 0) = 1$; after Cycle_2, we have $x_3 = y_2 = 1$ and $y_3 = (x_2 ? \text{in}_2 : 0) = (\text{undefined} ? 2 : 0) = \text{undefined}$; and so on. As a result, the undefined value propagates back and forth between x and y, as reflected by the table in Figure 4a, demonstrating the missing-reset problem where the undefined values of x and y are used infinitely often. To fix this issue, the developer should add a reset for y in Cycle_0, in addition to the existing reset for x

(line 3), ensuring that y_1 is defined and preventing the undefined values of x and y from being used infinitely often.

3.2.2 Missing-Reset vs. UBI. The missing-reset problem in Verilog bears a resemblance to the classical *use-before-initialization* (UBI) problem in software, which is prevalent in large-scale systems such as the Linux kernel and has driven decades of extensive research into static analyses for UBI detection [44, 82]. Despite their shared focus on undefined-value-related bugs, analyses for missing resets are fundamentally distinguished from those for UBIs due to their *hardware-specific nature*:

- *Hardware-Specific Problem Formulation:* The missing-reset problem concerns a register whose undefined value is *infinitely often* used in a *legal* Verilog execution, whereas the UBI problem concerns a variable whose undefined value is *once* used in *any possible* software execution. Consequently, analyses for missing resets must establish hardware-specific insights to handle the "infinitely often" and "legal" constraints that do not arise in the UBI formulation. These insights will be elaborated in Section 3.2.3.
- *Hardware-Specific Information Requirements:* The missing-reset problem is inherently grounded in hardware semantics, including clocks, registers, and resets, beyond the scope of UBI. Therefore, detecting such bugs naturally requires fundamental hardware information that goes beyond the needs of traditional UBI analyses. Notably, obtaining all sorts of fundamental information from Verilog programs demands a series of non-trivial hardware analyses, a challenge further elaborated in Section 3.2.4.

These distinctions highlight the necessity of establishing hardware-specific insights to design hardware bug detection analyses capable of capturing hardware-specific problems and fundamental hardware analyses that can retrieve essential hardware semantic information from Verilog programs.

3.2.3 The Insight of Our Missing-Reset Analysis. The key insight of our `missing-reset` analysis is that only circuits involving *circular dependencies* among registers can have missing-reset problems. To see why, consider the opposite case: if a circuit has no circular dependencies among registers, then a topological order must exist for these registers. Following this order, defined values from constants or inputs can eventually propagate to all registers, ensuring that no register's undefined value is infinitely often used in any legal execution.

Based on this hardware-specific insight, our `missing-reset` analysis identifies registers that lack proper reset by reporting unreset registers involved in circular dependencies within the *hardware dependency graph.* This graph captures the data, control, resetting, and synchronization dependencies among hardware signals, including registers, clocks, resets, inputs, and constants. For instance, Figure 4b shows the hardware dependency graph of the missing-reset example from Figure 4a. We explain this hardware dependency graph as follows.

- The blue edge from x to y represents that x is data-dependent on y (introduced by x <= y in line 5 of Figure 4a), while the green edge from y to x indicates that y is control-dependent on x (introduced by y <= x ? ... in line 6 of Figure 4a). Together, these edges form a circular dependency (highlighted in the gray region) between x and y, underlying the situation where undefined values propagate back and forth between them, as elaborated in the *Problem Formulation* paragraph of Section 3.2.1.
- The orange synchronization edges from x and y to the clock signal clk indicate that x and y are registers synchronized by the clock clk.
- The red resetting edges from x to the reset signal rst and the constant 1 capture x's reset logic—resetting x to 1 under the control of rst.

More concretely, `missing-reset` includes the following four steps: (1) *Non-Synthesizable Code Exclusion*: Verilog codebases typically include non-synthesizable portions written for simulation

purposes, which do not appear in the final synthesized hardware and cannot contain real hardware bugs. Thus, we exclude such code from our analysis. (2) *Hardware Dependency Graph Construction*: Build the hardware dependency graph by resolving all sorts of dependencies in the Verilog program (e.g., as shown in Figure 4b). (3) *Circular Dependency Identification*: Locate cycles in the hardware dependency graph (the gray region in Figure 4b) to identify registers involved in circular dependencies (e.g., x and y in Figure 4b). (4) *Missing Reset Reporting*: Report registers from step (2) that lack proper reset. In Figure 4b, y is reported because it is unreset, whereas x is not since it is reset to 1 by rst (indicated by the red resetting edges in Figure 4b).

To provide a glimpse into real-world missing-reset bugs, Figure 4c presents such an example along with its hardware dependency graph, which retains only data and control dependencies for clarity. The code snippet originates from a FIFO (first-in, first-out) buffer in the Verilog-AXIS project [25], a popular open-source design with over 800 GitHub stars. Our analysis detected two missing-reset bugs in the registers drop_frame and wr_ptr_cur (lines 4–5 in Figure 4c) through the four steps described above. These bugs are *fatal*: the missing reset of drop_frame, a frame-dropping state register, leads to unintended packet drops, while the missing reset of wr_ptr_cur, a write-pointer register, causes writes to incorrect buffer addresses.

3.2.4   The Challenge in Building Sophisticated Hardware Static Analysis. Even with the core insights established, building a sophisticated hardware analysis remains challenging due to the lack of analyses focused on extracting fundamental hardware information—a topic that few prior works have explored.

For example, to detect missing-reset problems involving hardware *registers* driven by *clock* signals and reset by *reset* signals, as guided by the insights established in Section 3.2.3, the analysis requires essential hardware information about whether each variable represents a hardware register, a clock signal, a reset signal, or none of these. However, this is nontrivial because Verilog variables do not explicitly declare such information. This information is typically revealed after lengthy synthesis processes that map Verilog code to physical hardware components based on behavioral semantics. To bridge this gap for both the missing-reset analysis and other clients requiring this hardware information, we developed hardware understanding analyses that reason about such behavioral semantics (e.g., inferring registers, clocks, and reset signals), enabling us to efficiently obtain the necessary information for missing-reset analysis without resorting to heavyweight synthesis.

Moreover, these hardware understanding analyses require more fundamental hardware information, which is also crucial for bug detection clients and future potential clients, such as hardware security analyses. This necessitates diverse analyses, including but not limited to: (1) Verilog module hierarchy analysis for inter-module reasoning; (2) hardware data-flow and control-flow analyses for constructing various graph structures (e.g., the hardware dependency graph used by the missing-reset analysis); (3) analyses for capturing hardware's synchronization and concurrency characteristics, which are essential for reasoning about behavioral semantics in hardware understanding; (4) analyses of bit selects/part selects and bit-vector arithmetic, which are indispensable for achieving practically useful precision in hardware analysis.

Ultimately, building sophisticated hardware static analysis challenges us to uncover diverse fundamental analyses, understand their interdependencies, and enable their interoperability. To this end, we introduce the well-organized analysis suite in Figure 2. Next, we'll discuss the analyses that support the missing-reset analysis.

3.3   Supporting Sophisticated Analyses: A Case Study

Although the four steps of missing-reset analysis are succinctly described in Section 3.2.3, their implementation relies on the collaboration of nearly twenty analyses in our suite, totaling 9,700

lines of code. Figure 5 illustrates all analyses supporting `missing-reset`, arranged in topological order from fundamental analyses (top) to analysis clients (bottom). Below, we use Figure 5 to help readers understand why the `missing-reset` analysis requires its dependencies and how these dependencies contribute to its overall effectiveness, thereby providing insight into the organization of our analysis suite for supporting sophisticated analyses. For readability, we do not discuss algorithmic details here; interested readers can find them in our open-source project.

Interdependencies. To clarify why the `missing-reset` analysis requires its dependencies in Figure 5 while maintaining simplicity and readability, we backtrack the bold dependency arrows in the figure, emphasizing both the functionality of these analyses and the necessity of their interdependencies. To begin, (1) Since missing-reset bugs are confined to registers, `missing-reset` directly depends on the `regs` analysis to infer physical registers, and utilizes `def-use` and `branch` analyses to build a graph containing data and control dependencies for cycle detection, as described in Section 3.2.3; (2) Since registers are synchronized by clock signals, `regs` tightly depends on `clocks` to infer physical clocks; (3) Since clocks are propagated as data across modules in Verilog, the `clocks` analysis directly depends on `fi-def-chain` (where `fi` denotes flow-insensitive analysis), which provides an inter-module graph representing data flow to support such reasoning; (4) Lastly, `fi-def-chain` leverages the `def-use` analysis to obtain define-use relations between

Fig. 5. All analyses that support the `missing-reset` analysis.

variables, including inter-module relations, to build the inter-module graph. The lighter dependency arrows in Figure 5 also support `missing-reset` in a similar way to the bold ones; for brevity, we omit a redundant discussion of these dependencies.

Analysis and Its Downstream Effectiveness. These interdependencies underscore that the effectiveness of each analysis can be critical for downstream analyses. For example, reduced precision in the `clocks` analysis (i.e., misclassifying non-clock variables as clocks) directly affects the precision of `regs`, since registers are synchronized by clock signals; insufficient precision in `regs` can lead to excessive false reports in `missing-reset`, undermining its practical value. Likewise, insufficient recall in `clocks` and `regs` (i.e., reduced soundness that omits some true behaviors) may cause `missing-reset` to overlook genuine missing-reset problems. To address these challenges, key supporting analyses are designed to achieve high effectiveness. For readability, we present our design considerations for two representative analyses: `regs` and `fi-const-prop`. The former shows our approach to designing a hardware analysis, which is not found in software, to achieve high recall and precision. The latter—a flow-insensitive constant propagation analysis—shows that some analyses in our suite, though inspired by software concepts, require Verilog-specific adaptations to achieve effective results on hardware.

`regs` (the register inference analysis) predicts which Verilog variables are synthesized as hardware registers and identifies their driven clocks. To achieve high precision, it cannot simply report every variable declared with Verilog's `reg` keyword, because many such declarations denote intermediate variables rather than physical registers (the keyword is misleading), yielding numerous false positives. Instead, `regs` classifies registers by their semantics. Register updates are synchronized to clock edges, which are expressed in Verilog as non-blocking assignments guarded by event
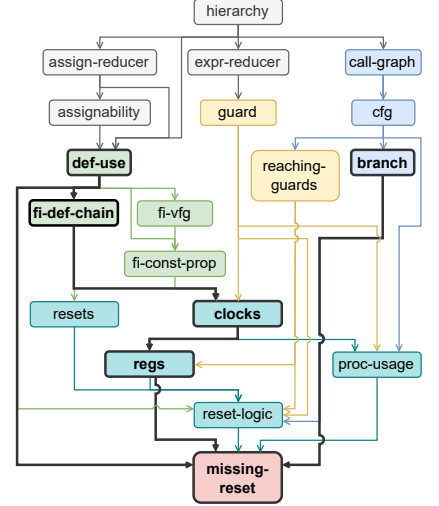
controls that reference clock signals; accordingly, the core step of regs is to locate such non-blocking assignments and report their left-hand-side (LHS) variables as registers. Recall that Figure 3 illustrates an event control (e.g., @(posedge clock)) guarding a non-blocking assignment (e.g., acc <= acc + in), where the LHS variable acc is the register. To achieve high recall, the key is to soundly extract the guarding relationship between non-blocking assignments and event controls. Simply matching the pattern shown in Figure 3 provides unsound guarding-relationship results because event controls and non-blocking assignments can appear in arbitrary orders within an always block, yielding nontrivial guarding relationships. For example, the body of an always block could be @(posedge clock1); x <= 1; @(negedge clock2); y <= 0;, where y <= 0 is guarded by the later @(negedge clock2), not the earlier @(posedge clock1). Thus, we compute the guard relation via a sound data-flow analysis over the control-flow graph using the fundamental reaching-guards analysis, on which regs depends, as shown in Figure 5.

fi-const-prop is widely used in our suite, and its precision is crucial for the effectiveness of downstream analyses. Unlike typical software programs, achieving high precision in Verilog requires bit-level constant reasoning, as Verilog programs frequently operate on individual bits or bit-vectors corresponding to physical wires. Accordingly, fi-const-prop accurately models Verilog's four-valued logic (0/1/X/Z) and bit-vector arithmetic to enable precise bit-level constant propagation. We discuss a bug detected thanks to this design in Section 6.2. Note that we do not adopt the flow-sensitive variant of constant propagation commonly used in software analysis, as we found it provides only marginal precision improvement while notably reducing analysis speed. This is mainly because most Verilog variables are defined in a single location, reflecting the fact that hardware signals typically have only one definition, which makes flow-sensitivity less beneficial. This further underscores the necessity of Verilog-specific adaptation.

Discussion on Soundness. As mentioned above, unsoundness in our hardware understanding analyses (e.g., clocks and regs) can render missing-reset unsound. Although we strive to maintain their high recall in practice, it is important to note that perfect soundness for these analyses is inherently unattainable. This arises because these hardware understanding analyses attempt to predict the circuit structure produced by synthesis tools, while the Verilog standard [1] specifies only circuit behaviors, not their structure. Consequently, different synthesis tools may generate structurally distinct circuits from the same Verilog code, with no formal standard serving as ground truth for sound prediction. Nevertheless, in practice, synthesis tools tend to follow common mapping conventions, allowing our analyses to effectively predict useful synthesis outcomes despite this unavoidable limitation.

Excluding this source of unsoundness, the *reachable closure* of missing-reset's results—the set of registers reachable from the reported registers in the hardware dependency graph—can *soundly* capture all possible missing-reset problems. Registers outside this closure cannot exhibit missing-reset issues, as they do not participate in circular dependencies within the circuit. Based on our insight in Section 3.2.3, their values are ultimately computed from constants, inputs, or reset registers, all of which obtain defined values after the reset cycle in any legal execution.

The collaborative nature of our analyses goes beyond those mentioned above and can be more intricate when developing hardware static analyses that are more sophisticated. We hope the example of missing-reset can serve as a reference for developers to create their own applications by leveraging existing analyses in our analysis suite.

4   Analysis-Oriented IR and Front End

The design of the IR and front end, a classic topic in compilers, requires further care in the context of a hardware analysis framework.

Basic Definitions

(Identifier)	*id, key, label*	::=	symbols
(Hierarchical Identifier)	*hierId*	::=	(*id* .)* *id* . *id*
(Value)	*val*	::=	bit vectors and real numbers
(Type)	τ	::=	bit vector type, real type, and array type
(Attributes)	*attr*	::=	(***** *key* **=** *val* (**,** *key* **=** *val*)* *****)

Hierarchical Structures

(Design)	*design*	::=	*attr*$^?$ *module**
(Module)	*module*	::=	**module** *id attr*$^?$ **{** (**port**$^?$ *net*)* (**port**$^?$ *var*)* *const**
			instModule* *proc** *func** **}**
(Net)	*net*	::=	*netKind id* **:** τ **;** *attr*$^?$
	netKind	::=	**wire** | **wor** | **wand** | **uwire** | $\cdots$
(Variable)	*var*	::=	**var** *id* **:** τ **;** *attr*$^?$
(Constant)	*const*	::=	**const** *id* **=** *val* **;** *attr*$^?$
(Module Instantiation)	*instModule*	::=	**inst** *id* **:** *moduleId* **;** *attr*$^?$
(Process Block)	*proc*	::=	**proc** *attr*$^?$ **{** *stmt** **}**
(Function)	*func*	::=	**func** *id* **(** *args*$^?$ **)** **->** **(** *args*$^?$ **)** *attr*$^?$ **{** *var** *const** *stmt** **}**
	args	::=	*id* (**,** *id*)*

Three-Address Code

(Statement)	*stmt*	::=	*label* **:** | *nonLabelStmt attr*$^?$
	nonLabelStmt	::=	*assignStmt* | *guardStmt* | *ifStmt* | *caseStmt* | *gotoStmt*
			| *invokeStmt* | *receiveStmt* | *syscallStmt* | **return;** | **pass;**
	assignStmt	::=	(*localLoad* | *localStore* | *hierLoad* | *hierStore* | *compute*) **;**
	guardStmt	::=	*timingControl* **;**
	ifStmt	::=	**if** *id* **goto** *label* **;**
	caseStmt	::=	(**case** | **casex** | **casez**) *id* **{** (*expr* **:** **goto** *label* **;**)*
			(**default: goto** *label* **;**)$^?$ **}**
	gotoStmt	::=	**goto** *label* **;**
	invokeStmt	::=	**invoke** (*id* | *hierId*) **(** *params*$^?$ **);**
	receiveStmt	::=	**receive** **(** *params*$^?$ **);**
	syscallStmt	::=	**syscall** *id* **(** *params*$^?$ **)** (**->** **(** *params* **)**)$^?$ **;**
	params	::=	*id* (**,** *id*)*
(Assignment)	*localLoad*	::=	*id* (**=** | **<=** | **<-**) *localAccessExpr*
	localStore	::=	*localAccessExpr* (**=** | **<=** | **<-**) *id*
	hierLoad	::=	*id* (**=** | **<=** | **<-**) *hierAccessExpr*
	hierStore	::=	*hierAccessExpr* (**=** | **<=** | **<-**) *id*
	compute	::=	*id* (**=** | **<=** | **<-**) *computingExpr*
(Expression)	*expr*	::=	*localAccessExpr* | *hierAccessExpr* | *computingExpr*
	localAccessExpr	::=	*id* (**[** *id* **]**)$^?$ *selector*$^?$
	hierAccessExpr	::=	*hierId* (**[** *id* **]**)$^?$ *selector*$^?$
	selector	::=	**[** *m* **:** *n* **]** | **[** *id* (**+** | **-**) **:** *n* **]** (*m, n* ::= non-nagative integers)
	computingExpr	::=	*uop id* | *bop id id* | *id* **?** *id* **:** *id*
			| (**zext** | **sext**) *id* **to** τ | **cast** *id* **to** τ
	uop	::=	**neg** | **not** | **rand** | **ror** | **rxor** | **buf**
	bop	::=	**add** | **sub** | **mul** | **udiv** | **sdiv** | **urem** | **srem** | **pow** | **upow**
			| **ult** | **ugt** | **ule** | **uge** | **slt** | **sgt** | **sle** | **sge**
			| **eq** | **equiv** | **neq** | **nequiv** | **and** | **or** | **xor**
			| **shl** | **ashr** | **lshr** | **concat**
(Timing Control)	*timingControl*	::=	**#** *id* | **@(** *eventExprList*$^?$ **)** | **repeat** **(** *id* **)** **@(** *eventExprList*$^?$ **)**
	eventExprList	::=	*eventExpr* (**or** *eventExpr*)*
	eventExpr	::=	*expr* | **posedge** *expr* | **negedge** *expr*

Fig. 6. The syntax of Qihe IR, which uses `fixed-width bold` fonts for terminals, *italics* for non-terminals, and gray fonts to omit common definitions for brevity. The symbol *pattern*$^?$ means *pattern* is optional. *pattern** means zero or more occurrences of *pattern*. Section 4 provides a detailed example (Figure 7) to highlight the key design features of Qihe IR.
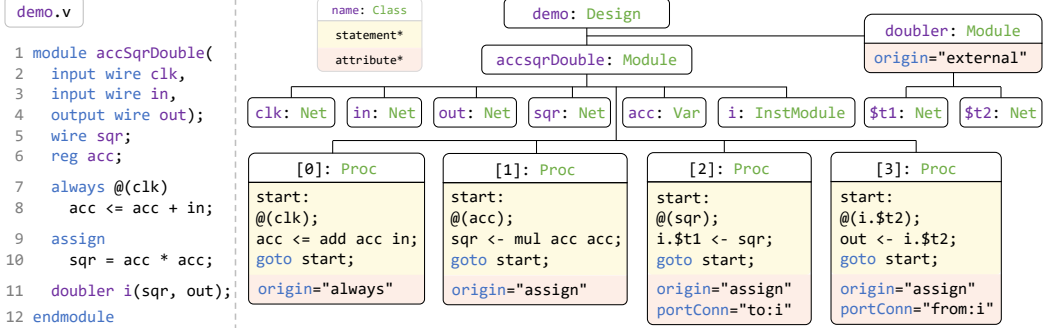
Fig. 7. Example of Qihe IR derived from a Verilog program. Each square on the right represents an IR node modeling Verilog's hierarchical language construct on the left, as elaborated in Section 4.1. Specific nodes (e.g., Proc) contain three-address code statements, detailed in Section 4.2. Each node can have zero or more attributes, whose uses are discussed in Sections 4.3 and 4.4. Nodes with $-prefixed names are generated by the front end.

Regarding the IR, Qihe introduces a hardware program abstraction called Qihe IR, which is designed to capture all Verilog features for describing hardware[5] while remaining simple enough to facilitate the development of analyses. Figure 6 presents the syntax of Qihe IR. Apart from some basic definitions, Qihe IR consists of two main components: hierarchical structures and three-address code (3AC). Instead of elaborating on the full syntax and semantics of Qihe IR, which would be verbose and difficult to follow, we use an example in Figure 7 to highlight the key designs of these two components in Sections 4.1 and 4.2, respectively. Readers can refer to our project documentation for additional details about Qihe IR.

Regarding the front end, beyond its standard task of converting Verilog code into Qihe IR, we find that it must also cooperate with Qihe IR to address specific needs in hardware analysis. We highlight two important aspects: (1) how to satisfy the seemingly conflicting requirements of varied analyses (Section 4.3), and (2) how to handle incomplete programs to enable sophisticated analyses (Section 4.4). Additional design and implementation details of the front end are available in Qihe's open-source repository.

4.1 Intuitive Tree Representation for Verilog's Hierarchical Structure

A Verilog design consists of modules that can include nets, variables, sub-module instantiations, functions, and other constructs, forming a nested hierarchy. Qihe IR's high-level structure is designed to closely mirror this hierarchical organization of hardware designs. As illustrated in the *Hierarchical Structures* portion of Figure 7, the Qihe IR syntax begins with a design that contains modules. These modules, in turn, include nets, variables, module instantiations, process blocks, and functions, directly reflecting Verilog's hierarchical structure. This design enables users familiar with Verilog to quickly understand the IR and navigate it intuitively using only the Verilog source code as a reference when developing analyses.

---

[5]The Verilog language includes synthesizable constructs for describing hardware and non-synthesizable ones for writing testbenches. Non-synthesizable features are not relevant to Qihe's hardware analysis and are thus excluded from this paper. When referring to Verilog in this context, we focus on its synthesizable subset. However, Qihe does support certain commonly used non-synthesizable features to expand its application scope. Additionally, Qihe also supports frequently used features from SystemVerilog [2], an extension of Verilog.

In this section, we use the example in Figure 7 to illustrate how Qihe IR's hierarchical structure corresponds to Verilog's structure, explaining the semantics of Verilog constructs along the way for readers who may be less familiar with Verilog. This example covers the most commonly used constructs in Verilog.

As illustrated in Figure 7, the left side shows a Verilog program named demo.v, while the right side presents its corresponding Qihe IR. The hierarchical structure of Qihe IR is illustrated as a tree rather than in textual syntax, because the tree representation more intuitively conveys the hierarchical relationships among different Verilog constructs and is closer to the in-memory representation of Qihe IR used by analyses. Each node in the tree corresponds to an IR construct and is labeled with a name and a class. The name identifies the node and is derived from the Verilog code when possible. The class indicates which Qihe IR construct the node represents, corresponding to the non-terminals defined in Figure 6. Let's examine the tree from top to bottom.

The root of the IR is a node named demo of class Design, representing the entire hardware design described by the Verilog code in demo.v on the left of Figure 7. The name demo is derived from the Verilog file name demo.v. The Design class corresponds to the *design* non-terminal in Figure 6. We omit detailed explanations of node names and classes in subsequent discussions unless necessary, as they are largely self-explanatory.

The root node has two Module nodes as its children, each representing a Verilog module, reflecting the fact that a Verilog design is composed of a collection of module definitions. In Verilog, a module is a fundamental unit for encapsulating reusable circuit designs, similar to how a function encapsulates reusable computation logic, and can have inputs and outputs known as ports. For example, the left part of Figure 7 defines a Verilog module named accSqrDouble (line 1) with inputs clk and in (lines 2–3) and an output out (line 4). This module encapsulates a circuit that accumulates input values from in into a register acc synchronized by the clock clk (lines 7–8), squares (lines 9–10) and doubles (line 11) the value from acc, and outputs the result to out (line 11). Let's follow the accSqrDouble node to elaborate on these details.

The accSqrDouble node has several subnodes representing different constructs within the module that describe the circuit. To begin with, Net and Var correspond to Verilog's wire (lines 2–5) and reg (line 6), respectively. These constructs are used to describe physical registers and wires. Note that module ports are special wires or regs with additional input or output keywords (lines 2–4). Although Verilog includes additional keywords like wand and wor for declaring special types of wires, Qihe IR uniformly represents all wire-related constructs using the Net class for simplicity. Furthermore, Qihe IR adopts the term Var instead of Reg to represent Verilog's reg declarations. This is because, in Verilog, the reg keyword does not necessarily imply a physical register in hardware; rather, it simply declares a software-like variable that may be mapped to wires, registers, or optimized out entirely based on its usage in the design. To know whether a Var is mapped to a physical register, developers can utilize Qihe's reg analysis, which provides this information, as discussed in Section 3.3.

Next, the accSqrDouble node includes a subnode of class InstModule due to the module instantiation on line 11. Module instantiation is a fundamental mechanism in Verilog for reusing encapsulated circuit logic. In this example, the line doubler i(sqr, out) first creates an instance named i of the module doubler. The definition of doubler resembles the one shown in Figure 8, which encapsulates a circuit that doubles its input value and outputs the result. The instantiation next connects the wire sqr to the input port $t1 of the module instance i and the wire out to the output port

```
module doubler(
  input wire $t1,
  output wire $t2);
  // double $t1 and
  // assign to $t2
endmodule
```

Fig. 8. The inferred signature of module doubler in Figure 7.

$t2, ensuring that out carries twice the value of sqr without requiring the doubling logic to be

redefined. It is important to note that the `InstModule` node represents only the module instance i, while the port connections are modeled separately by `Proc` nodes. This design choice is further elaborated in the following paragraph. Additionally, the definition of a module like `doubler` may be unavailable in many real-world scenarios due to the widespread use of third-party IP cores with intellectual property restrictions. To handle such cases, our front end automatically infers key information about the missing module, such as the types and directions of its ports. The code shown in Figure 8 is actually inferred by our front end when processing the Verilog program in Figure 7 without the definition of `doubler`, and the top-right corner of Figure 7 illustrates a subtree representing this inferred module. This inferred information is critical for enabling sophisticated analyses, and we elaborate on the inference process in Section 4.4.

Finally, the `accSqrDouble` node includes four `Proc` nodes that unify the representation of Verilog `always` blocks (lines 7–8), continuous assignments (lines 9–10), and port connections (line 11). These three constructs are among the most commonly used for describing circuit logic in Verilog. However, they differ significantly in syntax, which complicates the development of analysis logic. Without unification, developers must handle each construct separately, resulting in redundant and complex code. To address this challenge, Qihe IR introduces the `Proc` node (process blocks), which uses three-address code (3AC) to provide a uniform representation of these constructs. This unification is made possible because, under Verilog's simulation semantics [1, 15], all these constructs can be interpreted as concurrent processes that execute statements to emulate the behavior of physical circuits. The term `Proc` is used to reflect this. In Section 4.2, we will elaborate on how these constructs describe circuit logic and how 3AC is designed to model them uniformly.

4.2 Unified Three-Address Code Representation for Verilog's Circuit Logic

In addition to Verilog's hierarchical structure, Qihe IR uses three-address code (3AC) to model circuit logic. The 3AC design simplifies the development of analyses by representing various Verilog constructs in a straightforward and uniform manner. Figure 6 illustrates all the 3AC statements in Qihe IR. For readers familiar with Verilog, many of these statements intuitively correspond to Verilog constructs. As a result, explaining each 3AC statement in detail in this paper would be redundant and tedious, so we have left these specifics to our documentation. Instead, we focus on a key design feature: how 3AC is used to uniformly model different Verilog constructs for describing circuit logic. Using the example in Figure 7, we explain the semantics of the three most frequently used Verilog constructs: `always` blocks, continuous assignments, and port connections, and how the circuit logic described by them is modeled using 3AC in Qihe IR.

Always Blocks. An `always` block uses statements to specify software-like computation logic between variables, which the Verilog synthesizer maps to circuit logic between physical wires and registers. In this example, the body of the `always` block contains the statement `@(clk) acc <= acc + in`, meaning that whenever clk changes (`@clk`), the register acc is updated by adding in to its previous value. Note that Verilog introduces a special non-blocking assignment operator, `<=`, which is unique and essential for distinguishing between register updates and software-like variable assignments within `always` blocks. The behavior of the `always` block is modeled by the 3AC statements within the `[0]:Proc` node[6]. `[0]:Proc` contains an infinite loop that monitors the clock via a statement `@(clk)` and updates acc using `add  acc  in` whenever clk changes, directly corresponding to the behavior of the `always` block. To correctly model register updates in Verilog, we retain the non-blocking assignment operator `<=` as in line 8.

---

[6]Since `always` blocks, continuous assignments, and port connections are unnamed, we use indices (`[0]`, `[1]`, etc.) to refer to them by their sequence of occurrence in the source code.

Continuous Assignments. Although `always` blocks are enough to describe any circuit logic, Verilog also provides continuous assignments as a simpler way to express combinational (i.e., stateless) logic. A continuous assignment establishes a constraint between the wires and variables on the left-hand side and right-hand side of the assignment. For example, `assign sqr = acc * acc` specifies that `sqr` should always hold the value of `acc * acc`. The Verilog synthesizer generates the corresponding circuit logic for this multiplication to enforce this constraint. This behavior is modeled by `[1]:Proc`, which monitors changes in the value of `acc` and updates `sqr` accordingly. It resembles `[0]:Proc`, as continuous assignments can often be replaced by an equivalent `always` block; therefore, we unify them under `Proc`. However, the key difference is the use of the `<-` assignment operator, which is designed in Qihe IR to support Verilog's multiple drivers feature. Because of this feature, continuous assignments cannot always be fully replaced by `always` blocks. In Verilog, it is legal for multiple continuous assignments to target the same wire, such as "`assign u = 1; assign u = 0`". The final value of `u` is determined by merging these assigned values according to predefined rules. For example, if `u` is declared as a `wire`, the merged value is Verilog's special unknown value X; if `u` is declared as a `wand` (wired-AND), the merged value is `0 & 1 = 0`. To model this behavior, we introduce the `<-` operator, which records all assigned values to `u`. When there are multiple `<-` assignments, all recorded values are merged to determine the final value of `u`.

Port Connections. Port connections are implicitly established during module instantiation and share the same semantics as continuous assignments. For example, the instantiation of module `doubler` on line 11 of Figure 7 creates two port connections: one from `sqr` to `$t1` of instance `i` and another from `$t2` of instance `i` to `out`. This means that whenever `sqr` changes, its value is assigned to `$t1`, and whenever `$t2` changes, its value is assigned to `out`. This behavior is modeled by `[2]:Proc` and `[3]:Proc`, in a manner similar to how `[1]:Proc` models the continuous assignment. Note that we use the hierarchical reference `i.$t1` to refer to the `$t1` port of instance `i` without ambiguity in the statement; the same applies to `i.$t2`.

In summary, while Verilog provides convenient constructs that simplify hardware design, these features increase the complexity of circuit logic analysis. By converting such constructs into Qihe's process blocks with 3AC statements, Qihe enables analysis developers to handle them uniformly. During this conversion, Qihe IR also introduces statements and operators uncommon in software analysis IRs, such as guard statements and the `<-` operator, to accurately capture Verilog semantics.

The remaining part of Figure 7 not yet discussed is the attribute system, which enables key-value pairs to be associated with Qihe IR's hierarchical structures and statements. This system works closely with our front end to support additional analysis-oriented features, including those described in the following two sections.

4.3 Extensible IR for Varied Analysis Requirements

In addressing the varied requirements of different analyses, we identified seemingly conflicting requirements: (1) Some analyses require Qihe to simplify programs by reducing complex language constructs to their basic forms, relieving them from handling redundant details in the source code. (2) Others require Qihe to preserve extensive source-level information, which might otherwise be lost during simplification, to help them infer user intentions or speed up analysis.

To satisfy both needs, Qihe IR is designed around a simple core augmented with an extensible attribute system: (1) Its basic constructs form a small core of Verilog that is semantically equivalent to the full language, as discussed in previous sections. This core simplifies Verilog by desugaring syntactic sugar and complex constructs into a uniform, simple representation. As a result, developing analyses on this core becomes much simpler, as it avoids dealing with unnecessary source-level details and enables handling different Verilog constructs using the same IR constructs. (2) Qihe IR

augments the core with an attribute system so that the front end can supplement additional information about the original syntax patterns desugared during IR generation. This allows specialized analyses to leverage these attributes, enhancing precision or improving performance.

For example, Verilog's `always` blocks, port connections, and continuous assignments are all converted to process blocks in Qihe IR, allowing analyses to handle them uniformly with ease. Meanwhile, the attribute system preserves information about their original forms in the source code. Consider the port connection from `sqr` to `i.$t1` in Figure 7 as an example. The connection is first desugared into a continuous assignment and then converted to a process block `[2]:Proc`. The block has the attribute `portConn="to:i"`, which is added by the front end during desugaring, indicating that this is a port connection directed into the instance `i`. It also has the attribute `origin="assign"`, which is added during the conversion to a process block, indicating that it originated from a continuous assignment (itself desugared from the port connection). These attributes can guide the optimization of certain analyses. For example, the `fast-reaching-defs` analysis in Figure 2 utilizes this information to achieve an average 1.7× speedup across the 13 programs in Table 1 (reducing runtime from 44.6 s to 25.6 s on the largest XS program with 1.8M lines).

4.4 Automatic Signature Inference for Incomplete Programs

Incomplete programs are common in hardware design due to the widespread use of IP (intellectual property) cores whose implementations are encrypted or inaccessible. While linters can ignore the missing modules and still perform local pattern-based analysis, to support sophisticated inter-module analyses with better precision and soundness, Qihe needs to properly handle the missing pieces, which typically requires the signatures of the missing modules.

A module's signature includes its name, port types and directions, plus metadata such as compile-time (elaboration-time) parameters. Many of Qihe's more sophisticated analyses rely on this information. For example, taint analysis must track how values propagate within or across modules and determine whether a value is tainted. If a module lacks a signature—specifically, the direction of each port (input or output)—the taint analysis cannot propagate values at the module's ports because it does not know whether the value should flow into or out of the module, unless additional work is performed to infer this signature. However, with our front end automatically inferring this signature, not only taint analysis but also other analyses involving inter-module flow can benefit significantly. Another example is bit-width analysis. It requires the type and metadata information to check implicit type conversions that typically lead to bugs in Verilog. When there are multiple instantiations of a missing module, if a port is connected to wires of different types in different instantiations (e.g., one 8-bit and one 16-bit) and no metadata appears to explain the mismatch, analyses should raise a warning about the implicit type conversion.

Qihe's front end is thus designed to automatically infer the missing module signatures from the available context properly, relieving analysis developers from having to do this repeatedly when implementing their own analyses. We illustrate how Qihe handles a program with a missing module definition.

Consider the undefined module `doubler` in Figure 7: when processing the module instantiation "`doubler i(sqr, out)`", the front end detects that the definition of the instantiated module `doubler` is absent. To proceed, Qihe infers the signature of the module `doubler`, which includes the type and direction of its ports (this module appears to have no metadata). (1) First, it determines the number of ports. In this example, `doubler` has two ports, named `$t1` and `$t2`, corresponding to the two wires `sqr` and `out` connected to it. (2) Next, Qihe infers the port type based on the type of the connected wires. For instance, since `out` (connected to `$t2`) has a one-bit type, Qihe assumes `$t2` also has a one-bit type. Here, the `doubler` is only instantiated once; however, if a module is instantiated multiple times, for the same port, we need to check all the wires connected to it in

different instantiations. If these wires share the same type, that type is used as the result; otherwise, their types are merged to determine the inferred result. (3) Finally, the port direction is inferred by examining the definitions of the connected wires. If a wire connected to the port is defined by other assignments, the port is unlikely to be an output port, as this would cause conflicts. Thus, it is considered an input port. Otherwise, it is treated as an output port. In this example, $t1 is inferred as an input port because the wire sqr connected to it is already defined by lines 9–10, indicating that values flow from sqr into the instantiated module through $t1. In contrast, $t2 is inferred as an output port because the wire out connected to it is an output of the enclosing module (line 4) and is not otherwise driven, indicating that values are supplied by the instantiated module via $t2.

Note that because the generated doubler module cannot be distinguished from a deliberately designed two-port module with an empty implementation, Qihe marks it as external using the attribute system, as shown in Figure 7. This guides subsequent analyses to make sound assumptions about its unknown implementation.

5 Analysis Management

While traditional static analysis frameworks for software like Soot [73] and WALA [24] have gained wide adoption, they primarily focus on implementing standalone analyses rather than offering mechanisms for managing interdependent analyses. These frameworks provide limited support for use cases like (1) easy integration of new analyses, and (2) automatic execution of analyses with intricate dependencies.

We argue that effective management of analyses, particularly the integration and execution of interdependent analyses, is crucial for establishing Qihe as a practical framework. Without such capabilities, maintaining over 40 interdependent analyses, as illustrated in Figure 2, would become a challenging task. The recent software static analysis framework Tai-e [69] has also emphasized the importance of developer-friendly static analysis frameworks, introducing a configuration-file-based mechanism for analysis management, which received positive feedback from its users.

Inspired by Tai-e, we introduce a systematic mechanism for analysis management in Qihe, but with a different design. Instead of requiring developers to write configuration files for analysis management, as in Tai-e, we provide Java annotations to achieve the same functionality (as Qihe is implemented in Java). This approach simplifies the development of new analyses by utilizing Java annotations to automate tasks such as analysis discovery, which would otherwise require explicit registration in configuration files. Moreover, it helps developers avoid careless mistakes by taking advantage of the mature Java type system and associated tools, such as code completion and compile-time validation, which are typically unavailable when using configuration files.

This section highlights the key design aspects of Qihe's analysis manager in terms of analysis integration and execution. Beyond these core facilities, Qihe also provides additional management capabilities, such as analysis options and diagnostics handling. For readers interested in further details, we encourage consulting our project documentation.

5.1 Analysis Integration

To simplify the integration of new analyses, Qihe offers annotation-based facilities to address the following common needs in analysis integration:

- Registering the new analysis for management.
- Accessing the program abstraction to implement the analysis logic.
- Exporting results for use by other analyses.
- Retrieving results from existing analyses, if needed.

Analysis Integration

```
1 @Analysis(name="hello")          17 @Analysis(name="dump")
2 class HelloAnalysis {            18 class DumpAnalysis {
3                                  19
4   private String helloStr;       20   private HelloAnalysis hello;
5                                  21
6   @Analysis.Run                  22   @InjectAnalyses
7   public void run(               23   public Dumper(
8     Design design) {             24     HelloAnalysis hello) {
9     helloStr = "Hello, " +       25     this.hello = hello;
10        design.getName();        26   }
11  }                              27
12                                 28   @Analysis.Run
13  public String result() {       29   public void run() {
14    return helloStr;             30     println(hello.result());
15  }                              31   }
16 }                               32 }
```

Analysis Execution

```
$ qihe run dump -i demo.qh
```

Parse Java Annotations → **Analysis Manager** ← Parse the Command Line

Execute ↓

```
Design design = loadQiheIR("demo.qh");
// Create an analysis context holding
// analysis instances
var hello = new HelloAnalysis();
var dump = new DumpAnalysis(hello);
// Execute analyses by order
hello.run(design);
dump.run(); // "Hello, demo"
```
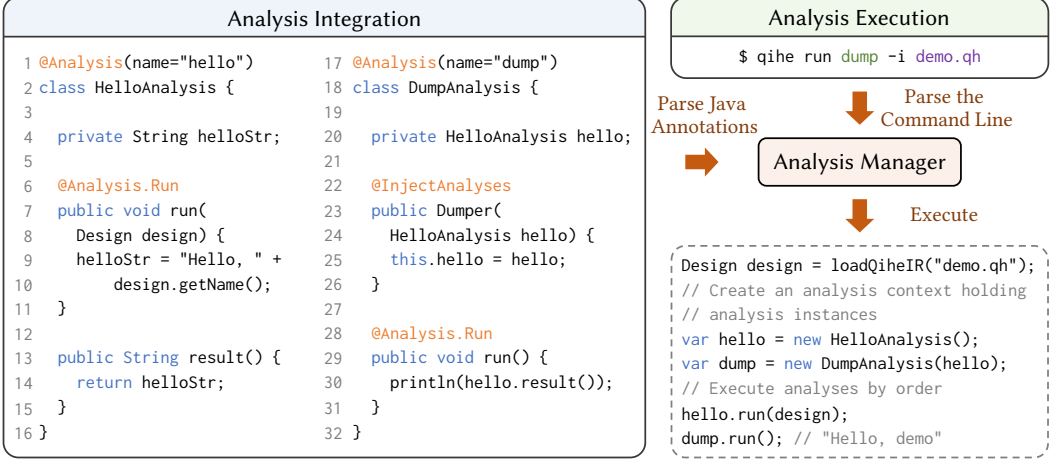
Fig. 9. An illustration of the process for integrating and executing analyses using the analysis manager. Qihe users only need to develop the analysis code (shown in "Analysis Integration") and specify which analyses to run (shown in "Analysis Execution"). The manager automatically executes the analyses in a manner similar to the code shown in the bottom-right of the figure. For simplicity, the analysis context and analysis executor, two core components of the manager depicted in Figure 1, are omitted here, and their logic is represented by the semantically equivalent code.

We will refer to Figure 9 to illustrate how each of these needs is fulfilled. The code shown in Figure 9 is written in Java, as Qihe is implemented in it.

To address the first need (i.e., analysis registration), each analysis in Qihe must be a Java class annotated with @Analysis(name="str") (see lines 1 and 17 in Figure 9). This annotation enables the framework to automatically discover and register analyses. The name attribute in the annotation serves as a brief identifier for the analysis, allowing users to specify which analysis to run via the command line. For instance, Figure 9 defines two analyses named hello and dump, and uses the command "qihe run dump -i demo.qh" to execute the analysis dump on the input Qihe IR demo.qh, as specified by the -i flag. This IR file represents the Verilog design compiled from the Verilog source file demo.v (shown in Figure 7) using the command "qihe compile demo.v -o demo.qh". It is worth noting that, in real-world projects, Verilog designs are typically described across multiple files. The "qihe compile" command can compile these files together into a single Qihe IR, which can then be analyzed using "qihe run".

To access the program abstraction (i.e., Qihe IR) and implement analysis logic, developers must define a Run method for each analysis (see lines 7 and 29). The first parameter of this method is a Design object provided by Qihe, which represents the Qihe IR specified by the -i flag on the command line (e.g., demo.qh in this example). If an analysis does not require direct access to the IR and only needs results from other analyses—for instance, a data-flow analysis that only relies on the control-flow graph provided by another analysis—this parameter can be omitted (as in the dump analysis). The body of the Run method contains the analysis logic. The method must be annotated with @Analysis.Run so that Qihe can automatically identify and invoke it during analysis execution.

To export the results of an analysis, developers simply provide public methods that return the analysis results. For example, in line 13 of Figure 9, the hello analysis defines a public method, result, which returns the "Hello" string computed in its Run method.

To access the results of existing analyses, a new analysis must declare its dependencies using a constructor that takes other analysis instances as parameters (see lines 22–26). This constructor should be annotated with `@InjectAnalyses`, allowing Qihe to identify and invoke it to inject the dependencies. If there are no dependencies, the constructor can be omitted (as in the `hello` analysis). The new analysis can store these dependent analysis objects as fields and use their results in its Run method. For example, the dump analysis retrieves the "Hello" string from the `hello` analysis stored in a field and prints it to standard output.

5.2 Analysis Execution

A usual approach to executing analyses involves manually creating instances of analysis classes and invoking their Run methods in the correct dependency order. For instance, to run the analysis dump, as shown in the bottom-right of Figure 9, one might load the Qihe IR, create instances of `hello` and then dump, and sequentially invoke their Run methods, passing the `Design` object if necessary.

However, as the number of analyses grows, manually managing this process becomes cumbersome and difficult to maintain. To address this issue, Qihe automates the entire process. Users simply need to execute a command like:

```
qihe run <analysis> -i <ir>
```

The framework then performs the following steps, achieving the same effect as generating and executing the code shown in the bottom-right of Figure 9 using Java's reflection mechanism:

(1) Loads the Qihe IR under analysis (i.e., `<ir>`) into a `Design` object.
(2) Scans the Qihe codebase for classes annotated with `@Analysis` to discover all available analyses and parses their annotations to record metadata.
(3) Resolves the dependencies of these analyses, instantiates them, and injects their dependencies in order via constructors annotated with `@InjectAnalyses`. If circular dependencies are detected, an error is reported.
(4) Executes the requested `<analysis>` and its dependencies by invoking their Run methods in the correct dependency order, passing the `Design` object as needed. Each Run method is invoked only once to avoid redundant computation.

With Qihe managing these tasks, developers can focus on utilizing existing analyses to implement their analysis logic, leaving the complexity of runtime management to the framework.

6 Evaluation

The purpose of Qihe is to support and facilitate the development of a wide range of static analysis clients in hardware. To achieve this, Qihe leverages a snowballing process, where fundamental analyses synergistically combine to build increasingly sophisticated analyses, ultimately enabling diverse application-specific analyses. To our knowledge, no prior work has explored these fundamental analyses or investigated their synergistic, snowballing-like composition.

This section begins with a case study (Section 6.1) that illustrates how the snowballing process enables the detection of critical missing-reset bugs by revealing the internal statistics integral to this process. Subsequently, we evaluate Qihe's utility in supporting useful, application-specific analyses across diverse domains: hardware bug detection (Section 6.2), hardware security analysis (Section 6.3), and hardware program understanding (Section 6.4).

6.1 Snowballing Process in Qihe: A Case Study

The growth of analysis capabilities in Qihe resembles the snowballing process, where existing analyses collaboratively contribute to increasingly sophisticated analyses. This process ultimately allows us to detect bugs that were previously unattainable with existing static analyses [16, 32, 57, 66] for

Table 1. Results of `missing-reset` and its representative dependencies, ordered topologically by their dependencies, across 13 real-world Verilog programs. Columns represent programs, rows represent analyses (except the `Meta` row lists lines of code and IR of these programs), and cells show summarized statistics. For `cfg` and `fi-def-chain` that provide graphical program abstraction, complete ground truth is infeasible to collect and thus omitted. For `clocks`, `regs`, and `resets`, results are measured by precision and recall against the ground truth (#Truth), where precision measures the proportion of identified results that are ground truth and recall indicates the proportion of ground truth results that are identified. For `missing-reset`, since no ground truth exists, we present how many reported bugs are confirmed by developers in the format "#Reported (#Confirmed)". If no bugs are confirmed, we do not count them as true bugs and omit "(#Confirmed)", as our goal is to highlight how many reported bugs are actually of concern to developers.

		XS	ziu	biv	FPC	ha3	riv	ax1	sev	pi2	dav	adf	st1	sha
Meta	LoC	1,821,858	22,290	13,601	7,751	7,662	6,832	6,804	3,461	3,049	2,753	2,724	2,553	2,171
	LoIR	8,261,716	26,554	25,435	41,065	16,053	12,309	28,151	5,825	5,747	3,592	7,128	9,429	2,281
cfg	#Node	5,020,594	15,308	14,762	18,641	8,130	7,035	17,159	3,362	3,170	2,013	4,215	5,844	1,846
	#Edge	4,355,660	14,646	13,773	24,735	8,583	6,635	15,812	2,892	3,398	1,799	4,161	5,100	1,932
fi-def-chain	#Node	1,549,502	6,369	5,784	11,280	4,272	3,024	7,753	1,211	1,932	1,024	1,476	2,124	1,656
	#Edge	3,176,863	12,009	10,387	15,147	7,486	4,653	13,575	1,694	9,414	1,628	2,106	3,479	87,631
clocks	#Truth	2,651	29	36	13	10	14	39	13	3	9	23	16	1
	Precision	100%	100%	100%	87%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	Recall	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
regs	#Truth	63,678	367	197	152	97	153	604	50	156	52	66	263	32
	Precision	100%	99%	86%	100%	100%	100%	100%	100%	99%	100%	100%	100%	100%
	Recall	100%	100%	97%	100%	100%	99%	99%	100%	97%	100%	94%	100%	100%
resets	#Truth	2,218	37	36	14	8	14	38	7	4	10	19	6	1
	Precision	98%	97%	100%	92%	80%	100%	100%	100%	100%	50%	100%	100%	100%
	Recall	88%	81%	100%	79%	100%	100%	100%	86%	75%	100%	100%	83%	100%
missing-reset	#Reported	50	9	0	0	0	0	20 (2)	10	5	3 (2)	1 (1)	12	0

Verilog bug detection, as summarized in Table 2. Section 3.3 has discussed an example of this snowballing process, where key analyses interdepend and collaborate to support the `missing-reset` analysis. In this section, we take a deeper look at this process by examining the experimental results associated with `missing-reset`. Specifically, we apply the `missing-reset` analysis and its dependencies to real-world programs, emphasizing three key aspects from the experimental results: (1) each analysis contributes to downstream effectiveness (i.e., analysis soundness and precision), (2) our analysis suite significantly reduces development costs, and (3) the efficiency of fundamental analyses dominates overall system efficiency.

Program Set. To provide practical statistics associated with the `missing-reset` analysis, we select thirteen real-world hardware programs spanning diverse domains, such as CPU design, encryption, AXI protocols, and computer vision. These programs are drawn from popular open-source GitHub projects [28, 39, 53, 60, 71, 72, 77, 78, 80, 81], which average approximately 1.9K stars, as well as from hardware bug benchmarks [17, 49], offering additional bug-related statistics. The top of Table 1 lists these programs by shorthand names for brevity, as well as their metadata, including lines of code (LoC) and lines of IR (LoIR) after compilation. All these programs are available in our project repository. To facilitate future research that develops new analysis clients atop our infrastructure (e.g., Verilog front end and IR) and evaluates them on these popular open-source projects, our repository preprocesses these projects into forms suitable for immediate analysis, such as precompiled IR or single Verilog files. This preprocessing saves developers from repeating

the substantial compilation effort, as Verilog projects lack a standard build system and often require careful understanding of each project's structure and build process to generate the IR.

6.1.1 Representative Analyses and Results. To showcase how analyses in Figure 5 enhance downstream effectiveness in the snowballing process, Table 1 presents representative analyses and their results, organized in topological order. The absence of certain analyses from this table, such as `hierarchy` (for module hierarchy resolution in Verilog) and `reaching-guards` (for synchronization information extraction), does not diminish their importance. They are excluded because their results are either unsuitable for concise presentation in the table or require substantial human effort to collect their ground truth. Nevertheless, their effectiveness is still reflected in the performance metrics of analyses listed in Table 1, as these analyses depend on them.

`cfg` (control-flow graphs) and `fi-def-chain` (flow-insensitive def-use chains) appear first in the table, highlighting their foundational role in enabling subsequent key analyses. For example, the `clocks` analysis, conducted later, utilizes edges from the `fi-def-chain` to propagate clock signal information. Initially, we explored a flow-sensitive implementation (`fs-def-chain`) due to concerns that false positive edges in the `fi-def-chain` might generate infeasible propagation paths, thereby reducing the precision of clock inference in `clocks` analysis. However, we discovered that the flow-insensitive version provides sufficient precision and significant speed benefits in practice, making it our preferred approach. This case also implies that Qihe actually offers a range of fundamental analyses, allowing developers to explore alternatives to make preferred trade-offs in practice.

The next three analyses in the table (`clocks`, `resets`, and `regs`) infer the physical usages of variables. Their losses in precision and recall (the definitions of which are explained in the caption of Table 1) can significantly compromise the effectiveness of their dependent analyses and the practical utility in real-world scenarios, as discussed in Section 3.3. Therefore, we dedicated careful effort for good precision and recall, and the results align with our expectations, as shown in Table 1. Since these analyses are intended for hardware understanding, Section 6.4 provides a detailed explanation of how we obtain the ground truth results and how these analyses achieve high precision and recall.

Finally, `missing-reset` effectively identifies missing-reset bugs in real-world programs that could lead to unintended hardware behavior, leveraging the effectiveness of the preceding analyses. As shown in Table 1, bold values indicate five bugs confirmed by developers. It is worth noting that the number of reported missing-reset bugs remains manageable. For example, even in the million-line XS program, only 50 bug reports are generated, which is acceptable for manual inspection.

Controlling the number of false bug reports is critical in hardware bug detection, as excessive reports can overwhelm developers and obscure true issues. We address this in two ways: (1) By inspecting false positives in real-world programs, we introduce refinements tailored to each client when necessary, reducing spurious bug reports without significantly sacrificing recall. For example, in the `missing-reset` analysis, we skip excessively large cycles when detecting cycles in the dependency graph. Such cycles are typically caused by false positive edges, which are inevitably introduced when building a sound dependency graph, and rarely correspond to genuine bugs. (2) We employ a general strategy for hardware analyses: Verilog programs often use meta-programming to describe repetitive circuit patterns. After elaboration, which expands meta-programming constructs, the same issue—especially a false positive—may be reported multiple times. To address this, we collapse near-duplicate reports and emit a single representative. While this approach may merge multiple instances of a true bug, fixing the representative root cause typically eliminates all related occurrences. As a result of these efforts, the numbers of bugs (i.e., the bugs in Table 2) reported by our detection clients on real-world programs are *all* manageable, similarly to the situation with `missing-reset`, making it easier for users to validate the true bugs among them.

6.1.2   Development Effort. In the snowballing process, developing a new client becomes significantly easier by leveraging Qihe's fundamental analyses. The left axis of Figure 10 shows all analyses contributing to `missing-reset`, arranged from bottom to top in topological order based on their dependencies. The gray bars represent the lines of code (LoC) for implementing each analysis. As shown, implementing `missing-reset` as a client within Qihe requires significantly less development effort (540 LoC) compared to implementing it from scratch without utilizing Qihe's fundamental analyses (9,700 LoC, calculated by summing the lines of code of all the required analyses for `missing-reset`). On average, each bug detection client built on our suite requires about 350 LoC, versus 5,900 LoC from scratch. This substantial reduction in development effort underscores Qihe's ability to facilitate the development of new application-specific analyses.

6.1.3   Analysis Time. Fundamental analyses, which address essential analysis needs with a focus on hardware characteristics, also shoulder the majority of the analysis workload required to support downstream analyses. Figure 10 shows the accumulated runtime cost of each analysis when running `missing-reset` on XS, a large-scale RISC-V SoC from Table 1 with 1.8M lines of code, measured on a machine with an Intel i9-13905H CPU and using 16 GB of memory. In this case, fundamental analyses (from `hierarchy` to `reaching-guards`) contribute 93% of the total time (68.2 s of 73.1 s). Similarly, across the other twelve programs from Table 1, they average 84% of the total time.

Fig. 10. Missing-reset analysis on a large-scale RISC-V SoC (XS from Table 1).

This highlights the dominant role of fundamental analyses in Qihe, further indicating that optimizing fundamental analyses offers a significant opportunity to reduce the overall analysis time. We accelerate these analyses by leveraging Verilog-specific properties—such as the acyclicity of combinational logic, the branchless nature of continuous assignments, and the synchronicity of guard statements—alongside conventional software optimization techniques. Before optimizations, some analyses on XS took up to an hour using 128 GB of memory. After optimization, the *entire* analysis suite (i.e., all 20 application-specific analyses, along with the 22 fundamental analyses in Figure 2) on XS completes in about 5 minutes using 40 GB of memory. For the other twelve programs in Table 1, totaling 82K lines, the entire suite completes in just 6 seconds. Note that running multiple analysis clients together in one process enables the sharing of analysis results and reduces recomputation, but increases peak memory usage by about 1 GB per additional analysis on this million-line XS program. Therefore, running `missing-reset` (involving 19 analyses) on XS takes about 1 minute and 16 GB, whereas executing the entire suite (involving 42 analyses) completes in about 5 minutes using 40 GB.
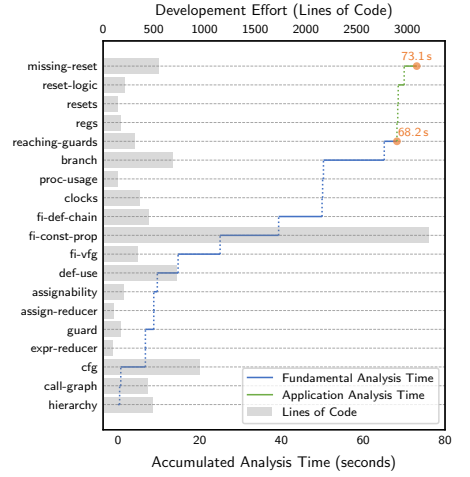
6.2   Hardware Bug Detection

Detecting hardware bugs remains a formidable challenge, even in industrial settings. According to Siemens' comprehensive global industrial study in 2024 [26], merely 14% of IC/ASIC projects achieved first silicon success, and only 13% of FPGA projects reported zero bug escapes into production. Unlike software, where bugs can be patched remotely over the internet at a low cost, late-stage hardware bugs in production can result in exponentially increasing remediation expenses [6, 8].

Static analysis offers a lightweight approach for early-stage bug detection at design time, making it especially cost-effective for hardware. To showcase Qihe's capability as a general-purpose

Table 2. Hardware bugs identified by Qihe in real-world projects. All listed bugs are undetectable by existing linter-style static analyses for Verilog bug detection [16, 32, 57, 66]. Newly discovered bugs are shown in **bold**; they were previously unknown and have been confirmed by developers.

Category	ID	Description	Project	Freshness
Missing Reset	B01	Register `drop_frame` lacks proper reset logic.	Verilog-AXIS [25]	Known from [48].
	B02	Register `wr_ptr_cur` lacks proper reset logic.		
	B03	Register `correct` lacks proper reset logic.	PULPissimo [62]	Known from [50].
	B04	Register `TIMER` lacks proper reset logic.	DarkRISCV [60]	Newly discovered.
	B05	Register `TIMEUS` lacks proper reset logic.		
Unreachable States	B06	Used state `dma_ctrl_reg == CTRL_ABORT` is unreachable.	OpenPiton [9]	Known from [18].
	B07	Used state `count == 2` is unreachable.	32-Verilog-Projects [54]	Newly discovered.
	B08	Used state `count == 8` is unreachable.		
	B09	Used state `count == 32` is unreachable.		
Deadlock	B10	`o_sclk` gets stuck from a deadlock.	SDSPI [27]	Known from [48].
	B11	`rd_addr` gets stuck from a deadlock.	HDL Lib [42]	
Undriven Signals	**B12**	Use of values from the undriven signal `clock_div`.	Basic Verilog [56]	Newly discovered.
	B13	Use of values from the undriven signal `r_cv`.	ZipCPU [28]	
	B14	Use of values from the undriven signal `v`.	32-Verilog-Projects [54]	
Unloaded Signals	B15	Module `aes_1cc` has an unloaded signal `rst`	PULPissimo [62]	Known from [50].
Submodule Misuse	**B16**	Misuse of submodule `invert` due to wrong port order.	32-Verilog-Projects [54]	Newly discovered.
Mis-Truncation	B17	Mis-truncation of `rx_mmio_channel` causes data loss.	Sha512 [14]	Known from [48].
Invalid Use	B18	Use of invalid value from signal `axi_adrv9001_rx_channel`.	HDL Lib [42]	

framework for supporting hardware bug detection clients, as described in Section 3, we developed a set of Verilog analyses, the insights behind which were derived from studying known bugs in the most recent hardware bug survey [48, 49], the renowned annual hardware bug detection hackathon [17–19], and through direct communication with industrial hardware practitioners [3, 4].

As shown in Table 2, the preliminary experimental results are very promising. Our analyses successfully identified 18 bugs across 8 categories—such as missing resets, unreachable states, and deadlocks—in a diverse set of real-world hardware projects, such as system-on-chips (SoCs), communication protocols, and encryption modules. As confirmed by our attempts, none of these bugs could be detected by existing Verilog linters [16, 32, 57, 66], the dominant static analysis tools for hardware bug detection. Their syntax-based approach and lack of fundamental semantic analysis prevent them from identifying bugs that require deeper hardware semantic reasoning. Notably, 9 of the 18 bugs were newly uncovered by our analyses from popular open-source projects [28, 54, 56, 60] (averaging over 1.5K GitHub stars). All of these new bugs have since been confirmed by the respective developers. These results not only highlight the effectiveness of our analyses but also underscore the promise of sophisticated static analysis for practical hardware bug detection.

To further illustrate how our analyses identify additional categories of real-world hardware bugs beyond the missing-reset bugs already discussed in Section 3.2, we present two additional case studies—an unreachable-state bug (B09) and a hardware deadlock bug (B10)—as supplementary representative examples of sophisticated Verilog static analyses for practical hardware bug detection.

6.2.1 An Unreachable-State Bug Case Study. Finite-state machine (FSM) design is a common task for Verilog engineers [51]. An unreachable state is one that the FSM cannot reach from any initial state. Such states often indicate functional issues, such as unintended infeasible state transitions [61].

(a) A real-world unreachable-state bug and its finite-state machine (FSM) diagram.

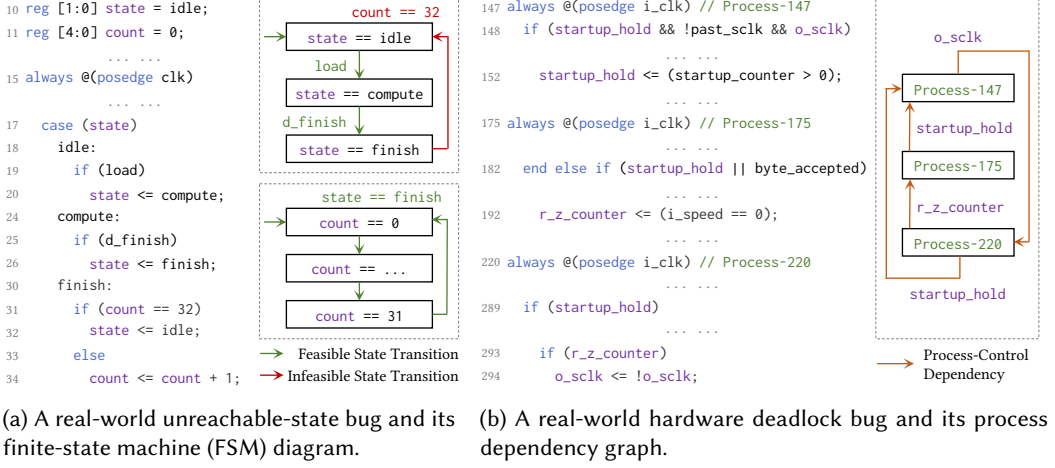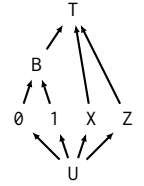(b) A real-world hardware deadlock bug and its process dependency graph.

Fig. 11. Real-world bug case studies: B09 (a) and B10 (b) from Table 2. Line numbers correspond to those in the original source projects.

Figure 11a presents a real-world unreachable-state bug from a CRC-32 serial counter in an open-source Verilog project [54], where idle, compute, and finish are constant enumeration values for state. The diagram on the right visualizes two FSMs defined by the Verilog code on the left: nodes represent states (corresponding to the values of state and count, respectively), edges denote transitions, and expressions on edges indicate transition conditions. Since all transitions in the count FSM (the lower one) share the same condition, we show it only once to avoid redundancy. The bug arises as follows. Lines 30–34 of Figure 11a describe the interaction between the state FSM (the upper one) and the count FSM: when state is finish, count starts incrementing, and when count == 32, state should return to idle. However, the developer failed to notice that count is declared as a 5-bit vector (line 11) that overflows after reaching 31 and wraps back to 0, an easy-to-overlook case in Verilog programming. Consequently, the state where count == 32 is unreachable in the count FSM, breaking the intended interaction between the two FSMs and causing the state FSM to exhibit an infeasible transition from state == finish to state == idle (highlighted in red). This flaw is severe, as it ultimately traps the system in the finish state and prevents it from ever resuming normal operation.

To detect such unreachable states, our analysis performs *Verilog-specific bit-level whole-program constant propagation* to determine all possible values of every variable with bit-level precision. As discussed in Section 3.3, this precision is critical in Verilog, where programs frequently manipulate individual bits, and loss of precision can cause the analysis to miss bugs that would otherwise be detected, such as the one in this case study. For each variable, our constant propagation overapproximates its concrete bit-vector value using a specially designed abstract bit vector, where each bit is drawn from the lattice shown on the right: U represents an undefined bit, 0/1/X/Z represent the fixed Verilog bit values 0/1/X/Z, B (short for *both*) represents a bit that can be either 0 or 1, and T represents the top lattice element encompassing all possible bit values.

In this case study, we focus primarily on the 0/1/B portion; for example, for a three-bit signal x, $x = B01$ implies that x can take only two possible values: 001 or 101. In Figure 11a, the transition condition count == 32 (line 31) is interpreted by our analysis as comparing 0BBBBB—with count zero-extended to 6 bits (an implicit type conversion per the Verilog specification [1])—against 100000, representing the fixed value 32. This equality is impossible because the most significant bit

never matches (0 ≠ 1); therefore, the state where count == 32, used in lines 31–32 of Figure 11a, is unreachable. In contrast, directly using a conventional non-bit-level constant propagation would treat the left-hand side as a non-constant value (denoted NAC), so evaluating NAC == 32 would yield a "possible" (reachable) rather than "impossible" (unreachable) result, causing this unreachable-state bug to be missed. This example underscores that directly applying software-like analyses to hardware is often ineffective, and that Verilog-specific customizations, such as maintaining bit-level precision, are essential. Additional designs and algorithms for Verilog-specific static analysis are available in our open-source framework and documentation.

6.2.2 A Hardware Deadlock Bug Case Study. According to the Verilog language specification [1], a running Verilog program consists of multiple concurrent processes, each typically corresponding to a runtime instance of an always block. A hardware deadlock occurs when circular dependencies among concurrent Verilog processes cause the program to stall infinitely [48], manifesting physically as signals that always fail to update as expected.

Figure 11b presents a real-world deadlock bug from the llsdspi module of the SDSPI project [27], which implements an SD card controller. This controller is expected to generate a driver clock signal named o_sclk. However, o_sclk fails to update due to a deadlock caused by circular dependencies among three processes: Process-147, Process-175, and Process-220, where Process-i denotes the runtime Verilog process corresponding to the always block starting at line i. Specifically, the update of o_sclk (line 294) in Process-220 relies on the update of r_z_counter (line 192) in Process-175, which in turn relies on the update of startup_hold (line 152) in Process-147, which again relies on the update of o_sclk (line 294) in Process-220 through line 148. As a result, all three processes remain infinitely stalled, preventing the SD card controller from generating the active driver clock o_sclk. Without this clock, the SD card becomes completely non-functional, unable to perform any data access operations.

To detect such hardware deadlock bugs, our analysis first constructs a *process dependency graph* that captures whether the value updates in one Verilog process depend on those in another, as exemplified in Figure 11b. In this graph, an edge from Process-i to Process-j ($i \neq j$) labeled with x indicates that Process-i is dependent on Process-j through a shared signal x; that is, x is used in a branch condition within Process-i and defined by an assignment in Process-j. To identify circular dependencies among concurrent Verilog processes, our analysis detects all cycles in this graph. For the example in Figure 11b, the analysis reports a circular process dependency among Process-147, Process-175, and Process-220, pinpointing the deadlock that stalls the system.

6.3 Hardware Security Analysis

Hardware security has become increasingly crucial due to the widespread use of Integrated Circuits (ICs) in systems that are particularly vulnerable to security threats like sensitive information leaks [35] and malicious hardware Trojans [36]. To further demonstrate Qihe's utility as a general framework to facilitate the development of various application-specific analyses, we developed two security analyses, taint and x-prop, as clients on top of Qihe.

Taint Analysis. Qihe's taint analysis is designed to address *information flow security* concerns. In this context, users specify *taint sources* and *sinks* to require no information flows from sources to sinks, as such a flow indicates potential vulnerabilities like unauthorized access or confidential information leaks. Taint sources in Verilog could be given as registers holding confidential data, while taint sinks could be nets or module ports susceptible to exploitation by attackers.

Compared to information flow analysis in software, tracking information flow in hardware introduces additional complexities due to hardware-specific semantics, such as bit-level value propagation and bidirectional port connections. Nevertheless, Qihe's infrastructure effectively

```
1  module trigger (...);              10  module DSP48E1 #(...) (
2    DSP48E1 #(...) signal_x_dsp (    11    ...,
3      .OVERFLOW(signal_x),           12    output OVERFLOW /* overflow indicator bit */
4      ...                            13  );
5    );                               14    buf b_of (OVERFLOW, OF);
6    always @*                        15    generate
7      if (signal_x) activated = 1;   16      if (...) assign OF = 1'bx;
8      else          activated = 0;   17      else ...
9  endmodule                          18    endgenerate
                                      19  endmodule
```

Fig. 12. A simplified excerpt of the hardware Trojan from [41] making use of X-Propagation. The value `1'bx` is propagated through various logic gates and port connections, as shown in red direct edges, ultimately reaching an `if` statement, which serves as the exploit point.

handles and abstracts these hardware features, enabling the successful development of a static information flow analysis based on classical explicit flow modeling.

To evaluate Qihe's taint analysis (i.e., `taint`), we utilized Trust-Hub [59, 64], the de-facto standard benchmark widely used for assessing the effectiveness of hardware security verification techniques. From its collection of Verilog designs, we collected all 25 related to information flow security and excluded 2 VHDL designs (incompatible with Qihe's Verilog-specific frontend) and 4 designs lacking explicitly specified taint sources/sinks. This resulted in 19 Verilog designs for final evaluation. These selected designs focus primarily on cryptographic hardware implementations, reflecting real-world applications where information flow security needs to be addressed.

Our analysis successfully identified 15 out of 19 information leak instances, where secret keys or the contents of internal state registers were leaked via observable output ports. After manual inspection, the 4 cases not identified by `taint` were implicit information flows caused by control dependencies. Currently, implicit covert channels (e.g., control dependencies, delays, and power consumption) are not handled by `taint`, but we believe it serves as a case for future work to feature more powerful analyses on top of Qihe.

X-Propagation Analysis. Qihe's `x-prop` analysis is designed to identify the propagation of X-values (unknown states), which often signify critical design flaws. These flaws can result in simulation failures or, more alarmingly, be exploited by hardware Trojans to conceal malicious behaviors.

Detecting X-propagation issues is non-trivial as it requires careful handling of X's unique semantics only seen in HDLs such as Verilog, and is further complicated when intertwined with hardware behaviors such as reset, memory, and bit addressing. Rather than developing a standalone analysis that accounts for every detail to address such challenges, Qihe's `x-prop` can focus on modeling X-related semantics by leveraging existing fundamental analyses to divide and conquer the intricate hardware behaviors. This approach results in a succinct implementation containing no more than 300 lines of code. In our implementation, Qihe leverages `resets` and `missing-reset` to understand the hardware reset behavior; infers the possible value range of variables for out-of-bounds access detection via constant propagation from `fi-const-prop`; captures bit-level semantics of Verilog by utilizing `bit-refs`.

Figure 12 illustrates a hardware Trojan from [41] identified by Qihe through its `x-prop` analysis. The example has been simplified for ease of understanding. We first explain how the Trojan is concealed by the X value and then describe how `x-prop` detects it.

To conceal the Trojan, the attacker exploits the discrepancy in the interpretation of the unknown value X between simulation and synthesis, rendering the Trojan undetectable during simulation but triggerable in fabricated circuits. In lines 7–8 of Figure 12, the attacker employs specialized circuit logic (detailed in the following paragraph) to ensure that the condition `signal_x` in the `if`

statement always evaluates to X. During simulation, this X value is treated as false as specified by the language standard [1], so activated remains 0. Since activated controls the execution of the Trojan payload—which could perform actions such as leaking secret keys (omitted for brevity in the example)—the payload is never triggered in simulation. However, after synthesis, the unknown X value is resolved to either 0 or 1 in the physical circuit. Thus, the condition signal_x may evaluate to 1, allowing activated to be set to 1 and enabling the Trojan payload.

To ensure that the if statement's condition evaluates to X during simulation and is less likely to be detected, the attacker often hides the source of X in a separate module. As highlighted by the red path in Figure 12, the attacker first configures a third-party DSP module (i.e., DSP48E1) to generate a X value on the variable OF (line 16). This X value is then propagated through a buffer gate (line 14) to the output port OVERFLOW of the DSP48E1 module (line 12). From there, the X value is further propagated into the signal_x variable in the trigger module via the connection to the output port of the instantiated DSP48E1 (lines 2–3). Ultimately, signal_x delivers the X value to the condition of the if statement (line 8). x-prop successfully detects that the variable OF in DSP48E1 may contain a X value during simulation and traces the propagation path from OF to the if statement. As a result, x-prop flags this as a potential X-Propagation exploit and reports it as an error. Through manual analysis, we confirmed the presence of a Trojan leveraging X values for concealment purposes.

6.4 Hardware Program Understanding

Hardware design is a process involving multiple stages, such as functional design, synthesis, verification, etc., ultimately leading to the physical implementation. When utilizing Verilog for functional design, it is crucial for hardware designers to grasp the details of physical implementation. For instance, they need to determine whether their clocks are integrated into complex logic to minimize latency and understand how registers are mapped to on-board resources to optimize resource utilization. Unfortunately, this vital information is often only disclosed after the lengthy synthesis process.

To help developers better understand their Verilog designs earlier, Qihe developed seven analyses for hardware program understanding that predict physical implementation details, with each analysis focusing on a specific understanding task. For instance, the clocks analysis predicts variables that function as part of the physical clock tree and flags improper usages like those used in combinational logic, while the regs analysis infers variables mapped to physical registers.

Notably, Qihe completes these seven analyses in approximately one minute on the real-world program XS from Table 1 comprising 1.8 million lines. In contrast, synthesizing the same codebase using Yosys [79], the most popular open-source synthesizer, exceeds our one-hour time budget.

Below, we examine the effectiveness of Qihe's analyses through two case studies: the clocks analysis and the regs analysis. We reuse the 13 diverse real-world programs introduced in Section 6.1, and the results are already presented in Table 1.

In the table, the ground truth is taken from Yosys's results since our analyses aim to predict the post-synthesis physical implementation. It is important to note that Yosys typically does not provide APIs to access results commonly required by static analysis, as it is not designed for this purpose. For example, it does not provide APIs to dump the commonly needed clock tree in a circuit—a feature supported by our clocks analysis. This limitation thus complicates the process of building ground truth: to obtain the true clock tree, we had to modify Yosys to dump the clock tree leaves (i.e., clock signals directly connected to registers), extract the variables connected to these leaves to approximate the clock tree, and then manually verify their validity as clock tree signals.

As shown in Table 1, the effectiveness of these analyses is measured using precision and recall. These metrics are critical for hardware program understanding, as insufficient performance in

either metric would lead to excessive false positives or false negatives, compromising the tool's practical utility in real-world scenarios. Thus, we devoted careful effort to design algorithms that go beyond simplistic pattern matching to achieve high precision and recall, and the results, as shown in the table, align with our expectations.

For clocks, it achieves excellent precision and recall across all evaluated programs due to our semantic-reasoning approach. The key insight is that hardware clocks form a tree structure whose boundaries with the rest of the circuit can largely be inferred from code semantics. Once these boundaries are identified, the entire tree can be extracted, starting from any tree node.

The regs analysis also achieves high precision and recall, leveraging the effectiveness of the clocks analysis it depends on and its hardware semantic-level reasoning algorithm. Its recall is slightly reduced in some programs because it misses registers updated via blocking assignments, rather than the standard non-blocking assignments. While Yosys accommodates such cases, blocking assignments cannot express the register's "update value in next clock cycle" semantics [15], and it is widely recognized as bad practice to use blocking assignments for register updates. Therefore, this minor loss of recall in such cases does not raise practical concerns for regs.

To summarize, these quick and accurate analyses for program understanding tasks can offer developers effective insights in the early stage of hardware development, and hopefully, more such analyses can be produced to further enhance hardware programming efficiency in the future.

7   Related Work

This work focuses on static analysis for hardware, with the most pertinent studies already discussed in Section 1. As the first attempt to create a general-purpose static analysis framework for Verilog, we briefly elucidate below how our idea was shaped by previous research and highlight their limitations in achieving our objective.

Existing static analysis tools for hardware, such as linters [16, 32, 57], have proven to be effective in production for simple bug detection and code style checking. However, the rapid growth of hardware complexity and diversity has posed new challenges in reliability, security, and maintainability [30, 40, 47, 74], which are far beyond the reach of simple syntax- or pattern-based static analyses, forcing developers to switch back to resource-intensive testing and verification [12, 23, 38, 70].

This coincides with the history of static analysis for software quality: researchers and engineers first adopted testing and verification [21, 33] with only simple static analyses for pattern-based checks [34, 37], and later sophisticated static analyses for bug detection and security, even for million-line code [10, 11, 13, 24, 31, 65, 67, 69, 73]. We believe the vast potential of sophisticated static analysis for addressing emerging challenges in hardware is yet to be explored.

This naturally raises the question: on what foundation can we build such sophisticated static analyses for hardware? An intuitive solution would be to leverage existing tools that already incorporate frontends and IRs. However, none of these tools are suited for developing sophisticated hardware static analyses [16, 32, 43, 57, 58, 63, 68]. This limitation stems from the fact that they are not specifically designed for static analysis and inevitably prioritize other design goals over analysis, let alone their lack of infrastructure, such as a set of necessary fundamental analyses to support diverse sophisticated analysis clients. Therefore, we introduce Qihe—a general-purpose static analysis framework for Verilog that explicitly prioritizes the needs of hardware analysis in the design of all its components.

8   Conclusion

We introduce Qihe, the first general-purpose static analysis framework for Verilog, designed to support a wide variety of application-specific analyses. This is accomplished through the collaboration of a comprehensive suite of fundamental analyses, an analysis-oriented front end and IR,

and a facilitative analysis manager. To showcase Qihe's potential, on top of its infrastructure, we further developed client analyses that successfully detect hardware bugs and vulnerabilities, and facilitate program understanding tasks, all in real-world projects. This work marks a preliminary step toward unlocking the full potential of sophisticated static analysis for hardware. We hope that Qihe, with its over 100K lines of fully open-source code, could inspire researchers and engineers from both the software and hardware communities to work in collaboration in advancing static analysis for hardware in the future.

Acknowledgments

References

[1] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. doi:10.1109/IEEESTD.2006.99495

[2] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), 1–1315. doi:10.1109/IEEESTD.2018.8299595

[3] 2025. HiSilicon. https://www.hisilicon.com/en.

[4] 2025. T-Head. https://www.t-head.cn/?lang=en.

[5] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2024. On Hardware Security Bug Code Fixes by Prompting Large Language Models. *IEEE Transactions on Information Forensics and Security* 19 (2024), 4043–4057. doi:10.1109/TIFS.2024.3374558

[6] Ken Albin. 2016. The Cost of SoC Bugs. https://dvcon-proceedings.org/document/the-cost-of-soc-bugs/.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. doi:10.1145/2666356.2594299

[8] Francine Bacchini, Robert F. Damiano, Bob Bentley, Kurt Baty, Kevin Normoyle, Makoto Ishii, and Einat Yogev. 2004. Verification: what works and what doesn't. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, Sharad Malik, Limor Fix, and Andrew B. Kahng (Eds.). ACM, 274. doi:10.1145/996566.996648

[9] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 217–232. doi:10.1145/2872362.2872414

[10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. doi:10.1145/1646353.1646374

[11] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 196–207. doi:10.1145/781131.781153

[12] Aaron R. Bradley and Zohar Manna. 2007. Checking Safety by Inductive Generalization of Counterexamples to Induction. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD '07)*. IEEE Computer Society, USA, 173–180. doi:10.1109/FAMCAD.2007.15

[13] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. doi:10.1145/1640089.1640108

[14] Ciro Luiz Araujo Ceissler. 2025. An SHA512 Accelerator. https://github.com/efeslab/hardcloud/tree/e28ca96fdbb67904ef909fb04e026cf6dc724198/samples/sha512.

[15] 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, Article 230
     (oct 2023), 30 pages. doi:10.1145/3622805
[16] Chipsalliance. 2025. Verible. https://chipsalliance.github.io/verible/.
[17] HACK@DAC-18 Competition Committee. 2018. Hack@DAC 2018 Buggy SoC. https://github.com/HACK-EVENT/
     hackatdac18.git.
[18] HACK@DAC-21 Competition Committee. 2021. Hack@DAC 2021 Buggy SoC. https://github.com/HACK-EVENT/
     hackatdac21.git.
[19] HACK@DAC Competition Committee. 2024. What is HACK@DAC? https://www.dac.com/Conference/HackDAC.
[20] CWE Community. 2024. CWE-1271: Uninitialized Value on Reset for Registers Holding Security Settings. https:
     //cwe.mitre.org/data/definitions/1271.html.
[21] E. Allen Emerson and Edmund M. Clarke. 1980. Characterizing correctness properties of parallel programs using
     fixpoints. In *Automata, Languages and Programming*, Jaco de Bakker and Jan van Leeuwen (Eds.). Springer Berlin
     Heidelberg, Berlin, Heidelberg, 169–181.
[22] E. Allen Emerson and Joseph Y. Halpern. 1986. "Sometimes" and "not never" revisited: on branching versus linear
     time temporal logic. *J. ACM* 33, 1 (Jan. 1986), 151–178. doi:10.1145/4904.4999
[23] Hongyu Fan and Fei He. 2024. Leveraging Datapath Propagation in IC3 for Hardware Model Checking. *Trans.
     Comp.-Aided Des. Integ. Cir. Sys.* 43, 7 (July 2024), 2215–2228. doi:10.1109/TCAD.2024.3360022
[24] Watson Libraries for Analysis. 2006. WALA. http://wala.sf.net.
[25] Alex Forencich. 2025. Verilog AXI Stream Components. https://github.com/alexforencich/verilog-axis.
[26] 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/.
[27] Dan Gisselquist. 2025. SD-Card controller. https://github.com/ZipCPU/sdspi.
[28] Dan Gisselquist. 2025. ZipCPU: A Small, Lightweight, RISC CPU Soft Core. https://github.com/ZipCPU/zipcpu/.
[29] Steve Golson and Leah Clark. 2016. Language wars in the 21st century: verilog versus vhdl–revisited. *Synopsys Users
     Group (SNUG)* (2016).
[30] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on
     AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. doi:10.1109/SP.2011.22
[31] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of
     lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*
     (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 290–299. doi:10.
     1145/1250734.1250767
[32] Naoya Hatta. 2025. svlint. https://github.com/dalance/svlint/tree/master.
[33] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.
     doi:10.1145/363235.363259
[34] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. doi:10.
     1145/1052883.1052895
[35] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware Information Flow Tracking. *ACM Comput. Surv.*
     54, 4, Article 83 (May 2021), 39 pages. doi:10.1145/3447867
[36] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. 2016. Detecting Hardware Trojans with Gate-Level Information-
     Flow Tracking. *Computer* 49, 8 (2016), 44–52. doi:10.1109/MC.2016.225
[37] Stephen C Johnson. 1977. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill.
[38] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan
     Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-
     Exploitable Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston,
     MA, 3219–3236. https://www.usenix.org/conference/usenixsecurity22/presentation/kande
[39] Olof Kindgren. 2025. SERV - The SErial RISC-V CPU. https://github.com/olofk/serv.
[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan
     Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre attacks: exploiting speculative execution.
     Commun. ACM 63, 7 (June 2020), 93–101. doi:10.1145/3399742
[41] Christian Krieg, Clifford Wolf, Axel Jantsch, and Tanja Zseby. 2017. Toggle MUX: How X-Optimism Can Lead
     to Malicious Hardware *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages.
     doi:10.1145/3061639.3062328
[42] Rejeesh Kutty. 2025. HDL libraries and projects. https://github.com/analogdevicesinc/hdl.
[43] Chris Lattner. 2025. CIRCT: Circuit IR Compilers and Tools. https://circt.llvm.org/.
[44] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-
     Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (April 2024), 26 pages. doi:10.1145/3649828

[45] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 109–120. doi:10.1145/1993498.1993512

[46] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:27. doi:10.4230/LIPIcs.ECOOP.2016.15

[47] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (May 2020), 46–56. doi:10.1145/3357033

[48] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. Debugging in the brave new world of reconfigurable hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 946–962. doi:10.1145/3503222.3507701

[49] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. Debugging in the brave new world of reconfigurable hardware (Bugbase Artifact). https://github.com/efeslab/hardware-bugbase.git.

[50] Xingyu Meng, Shamik Kundu, Arun K. Kanuparthi, and Kanad Basu. 2022. RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2022), 466–477. doi:10.1109/TCAD.2021.3066560

[51] Cody Miller. 2010. State Machine Design Techniques. https://www.edn.com/state-machine-design-techniques/.

[52] MITRE Corporation. 2025. Common Weakness Enumeration. https://cwe.mitre.org/.

[53] Kevin E. Murray. 2025. Verilog to Routing – Open Source CAD Flow for FPGA Research. https://github.com/verilog-to-routing/vtr-verilog-to-routing.

[54] Sudhamshu B N. 2025. Implementing 32 Verilog Mini Projects. https://github.com/sudhamshu091/32-Verilog-Mini-Projects.

[55] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. *SIGPLAN Not.* 41, 6 (June 2006), 308–319. doi:10.1145/1133255.1134018

[56] Konstantin Pavlov. 2025. Must-have Verilog SystemVerilog Modules. https://github.com/pConst/basic_verilog.

[57] Michael Popoloski. 2025. Slang. https://sv-lang.com/.

[58] 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)*. 1–6. doi:10.1109/DCAS61159.2024.10539889

[59] 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)*. 471–474. doi:10.1109/ICCD.2013.6657085

[60] M. Samsoniuk. 2025. DarkRISCV. https://github.com/darklife/darkriscv.

[61] Shaker Sarwary and Michael A Beaver. 2005. A Systematic Approach to Verifying FSMs. https://www.edn.com/a-systematic-approach-to-verifying-fsms/.

[62] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 1–3. doi:10.1109/S3S.2018.8640145

[63] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 258–271. doi:10.1145/3385412.3386024

[64] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. 2017. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security* 1 (2017), 85–102. doi:10.1007/s41635-017-0001-6

[65] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706. doi:10.1145/3192366.3192418

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

[67] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[68] 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.). Springer International Publishing, Cham, 451–460.

[69] 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 (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1093–1105. doi:10.1145/3597926.3598120

[70] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3237–3254. https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[71] ultraembedded. 2025. 32-bit Superscalar RISC-V CPU. https://github.com/ultraembedded/biriscv.

[72] ultraembedded. 2025. RISC-V CPU Core (RV32IM). https://github.com/ultraembedded/riscv.

[73] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., USA, 214–224. doi:10.1145/1925805.1925818

[74] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 991–1008.

[75] VLSIFacts. 2025. Understanding Pipeline Design in Verilog: How to Stage Data Across Clock Cycles for High Performance. https://vlsifacts.com/understanding-pipeline-design-in-verilog-how-to-stage-data-across-clock-cycles-for-high-performance/.

[76] VLSIFacts. 2025. Understanding Reset Signals in Digital Design: Types, Pros & Cons, and Best Practices. https://vlsifacts.com/understanding-reset-signals-in-digital-design-types-pros-cons-and-best-practices/.

[77] Xuan Wang. 2025. An FPGA-based Field Oriented Control (FOC) for driving BLDC/PMSM motor. https://github.com/WangXuan95/FPGA-FOC.

[78] Clifford Wolf. 2025. PicoRV32 - A Size-Optimized RISC-V CPU. https://github.com/YosysHQ/picorv32.

[79] Claire Wolf. 2025. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.

[80] Luke Wren. 2025. Hazard3 - A 3-Stage RV32IMACZb* Processor with Debug. https://github.com/Wren6991/Hazard3.

[81] 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)*. 1178–1199. doi:10.1109/MICRO56248.2022.00080

[82] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 221–232. doi:10.1145/3368089.3409686

[83] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 503–516. doi:10.1145/2694344.2694372