Qihe：面向 Verilog 的通用静态分析框架

QINLIN CHEN，南京大学，中国
NAIREN ZHANG，南京大学，中国
JINPENG WANG，南京大学，中国JIA
CAI CUI，南京大学，中国TIAN TAN*，南
京大学，中国XIAOXING MA，南京大学，中
国CHANG XU，南京大学，中国JIAN LU，
南京大学，中国YUE LI*，南京大学，中国

arXiv:2601.11408v1 [cs.PL] 16 Jan 2026

过去几十年来，静态分析在软件领域蓬勃发展，推动其应用于错误检测、安全性和程序理解。这些先进的分析在很大程度上依托于通用静态分析框架，这类框架提供了必要的基础设施，以简化相关分析的开发。这些框架的通用性通常建立在若干组件之上，例如用于程序转换的前端、中间表示（IR），以及一套能够提供关键程序信息的基础分析。Soot 就是 Java 领域此类框架的典范，催生了数千种分析，并对学术界和工业界产生了深远影响。相比之下，硬件领域缺少这样的框架，这可能限制该领域对多样化静态分析应用的探索。这一缺口可能掩盖静态分析在硬件领域的巨大潜力，阻碍该领域取得类似软件领域的成果。为了推动静态分析在硬件领域的发展，我们推出了 Qihe，这是首个面向 Verilog 的通用静态分析框架——鉴于硬件领域缺乏先例，这是一项极具挑战性的工作。Qihe 具备面向分析的前端、Verilog 专用中间表示，以及一套多样化的基础分析，能够捕捉硬件特有的关键特征，例如位向量算术、寄存器同步和数字组件并发，并支持检查复杂的硬件数据流与控制流。这些基础分析专为支持广泛的硬件分析客户端而设计。为验证 Qihe 的实用性，我们还开发了一系列涵盖错误检测、安全性和程序理解的分析客户端，并将其应用于真实世界硬件项目。与软件不同，真实世界硬件项目通常包含较少但更难检测的漏洞，因为它们一般会经过大量仿真和严格验证，以避免硬件缺陷造成高昂代价。尽管如此，我们的初步实验结果仍十分出色；例如，Qihe 在热门的真实世界硬件项目中发现了 9 个此前未知的漏洞（平均拥有 1.5K+ 个 GitHub 星标），这些漏洞随后均得到开发人员确认；此外，Qihe 成功检测出 18 个现有 Verilog 漏洞检测静态分析（即代码检查器）无法发现的漏洞，并在真实世界硬件程序中检测到 16 个安全漏洞。这些结果凸显了静态分析在硬件设计领域的变革潜力。通过开源 Qihe[1]，（其代码超过 100K 行），以及精心整理、内容全面的文档[2]，我们希望激发更多面向硬件的先进静态分析创新与应用，并致力于培育一个像软件分析领域一样充满活力的生态系统。

*通讯作者。

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

作者联系信息：Qinlin Chen，qinlinchen@smail.nju.edu.cn，南京大学，中国；Nairen Zhang，nairen@smail.nju.edu.cn，南京大学，中国；JinpengWang，jpwang@smail.nju.edu.cn，南京大学，中国；Jiacai Cui，jiacaicui@smail.nju.edu.cn，南京大学，中国；Tian Tan，tiantan@nju.edu.cn，南京大学，中国；Xiaoxing Ma，xxm@nju.edu.cn，南京大学，中国；Chang Xu，changxu@nju.edu.cn，南京大学，中国；Jian Lu，lj@nju.edu.cn，南京大学，中国；Yue Li，yueli@nju.edu.cn，南京大学，中国；所有作者还隶属于南京大学新型软件技术国家重点实验室，中国。

1 引言

在软件领域，静态分析被广泛认为是提升软件质量的有效技术。多年来，人们开发了许多先进的静态分析，以完成各种应用任务，例如检测漏洞 [55], 识别安全漏洞 [7],以及辅助程序理解 [46]。

尽管静态分析在软件领域已取得显著成效，但在硬件领域，尤其是以 Verilog 这一主流硬件描述语言（HDL） [29]为代表的领域中，静态分析仍未得到充分应用。目前的情况类似于软件静态分析发展的早期阶段：当时其应用仅限于编译器集成优化和基础语法检查（例如早期的代码检查工具），而更复杂的软件分析（例如用于错误检测和安全性的分析）尚未出现。在硬件领域，静态分析技术主要由硬件综合器（例如 Yosys [79]）用于编译器优化；代码检查工具 [16, 32, 57, 66] 则广泛用于执行代码风格或语法检查，但缺乏检测更深层设计缺陷所需的复杂能力，例如寄存器复位不当和硬件木马。

我们认为，硬件静态分析停滞不前的主要原因之一，是缺少通用框架，以提供开发复杂分析所需的基础设施。在软件领域，此类框架已被证明具有变革性。例如，面向 Java 的通用静态分析框架 Soot [73], ，推动了学术界和工业界数以千计的分析研究。FlowDroid [7], 等研究成果被广泛用于检测 Android 应用中的安全漏洞，并依赖 Soot 的基础分析来静态识别控制流和数据流。一些公司也利用 FlowDroid 和 Soot 开发自己的分析工具，使其能够在开发周期早期发现漏洞，从而有望显著节省成本。

借鉴现有软件分析框架的设计 [24, 69, 73]，构建通用框架通常需要以下关键组件紧密协作：(1) 基础分析，提供各类特定应用分析共用的基本程序信息；(2) 前端和中间表示，从源代码中导出面向分析的程序抽象；(3) 分析管理器，促进复用现有分析的结果，并集成新的分析。

然而，由于硬件领域缺乏通用框架，硬件静态分析的开发通常依赖以下三类方案之一，而每类方案都有其局限性。Slang [57]、Verible [16]、SVLint [32],和 Verilator-Lint [66] 等代码检查器仅限于对抽象语法树（AST）进行基于语法或模式的分析。Pyverilog [68] 和 VeriPy [58] 等工具提供了超越 AST 的硬件抽象，但缺乏处理同步和并发等关键硬件特性的基础设施，这极大限制了它们满足多样化分析需求的能力。CIRCT [43], 是一个主要用于综合的综合性硬件编译器框架，虽然也可用于静态分析，但缺少构建复杂分析所必需的基础设施，而这类分析需要处理复杂的值流并进行深入的语义推理，因此开发成本高昂。

在这篇论文中，我们介绍了 Qihe，这是首个面向 Verilog 的通用静态分析框架，包含三个关键组件：

- 基础分析集 （第3节）：我们开发了 22 项基础分析，用于静态构建设计一系列分析客户端所需的硬件程序信息。这项独特贡献源于我们在基础静态软件分析领域长达十年的前沿研发经验、与工业界硬件从业者的持续交流，以及对 Verilog 语义的深入理解。

• 面向分析的中间表示与前端 （第 4节）：我们设计了中间表示和前端，使二者能够协同工作，满足特定的分析需求，例如支持对不完整硬件程序的分析并保留额外的源信息，同时确保中间表示保持简单，便于分析。

• 分析管理器 （第 5节）：我们设计了一种面向工程的机制，使新开发人员能够在 Qihe 中便捷地集成和执行新的分析。

为了验证 Qihe 作为通用框架支持各种面向应用的特定分析的能力，我们开发了 20 个客户端分析，同时也将其作为示例，指导未来的开发人员创建自己的分析。这些分析针对通过硬件漏洞和安全漏洞调查所发现的常见但并不简单的需求 [17, 18, 48, 52, 59, 64]，并结合了硬件从业者的行业见解，涵盖包括错误检测、安全分析和程序理解在内的多个应用领域。

我们通过将这些客户端分析应用于一组真实世界的 Verilog 程序开展了实验，其中包括片上系统（SoC）设计、加密模块、通信协议（例如 AXI 实现）等 [28, 49, 54, 56, 59, 60, 64, 81]。初步实验结果总结如下：

• 硬件漏洞检测：与软件不同，硬件往往包含数量较少但更难检测的漏洞，因为硬件通常会经过大量仿真和严格验证，以避免缺陷造成的高昂代价 [6, 8]。尽管如此，我们的分析在热门的真实世界硬件项目（GitHub 星标平均超过 1.5K）中发现了 9 个此前未知的漏洞，随后均得到了开发人员的确认。例如，RISC-V CPU 中的寄存器未能正确复位、CRC 生成器中存在不可达状态、SHA512 加密模块中出现意外的位截断等。我们的分析还成功检测出另外 9 个硬件漏洞：其中一些由以往的调查从真实世界程序的提交历史中收集而来，另一些则由工业硬件专家总结，并作为基准测试注入真实程序中 [17, 18, 48]。请注意，如表 2所总结，这 18 个漏洞涵盖八个类别，现有用于 Verilog 漏洞检测的静态分析均无法检测到它们 [16, 32, 57, 66]。

• 硬件安全分析：我们的分析检测到 15 个与信息泄露相关的安全漏洞，尤其是通过天线和电容等可观察输出端口泄露加密密钥。此外，还检测到一个硬件木马 [41]，它利用 Verilog 中特殊的 X 值进行隐蔽。

• 硬件程序理解：我们的分析通过预测 Verilog 综合的物理实现细节，帮助开发人员理解其语义，例如以较高的召回率和精确率判断变量是否映射为时钟信号、锁存器或寄存器。例如，为了尽量降低延迟，代码中的时钟应避免用于复杂逻辑中，但这类时钟用法往往要经过耗时的综合过程才能发现。借助静态分析，可以更早发现这些问题，例如在开发阶段，从而降低潜在的高昂工程成本。

此外，根据初步实验结果，我们强调三个相辅相成的要点。首先，现有的硬件静态分析技术难以检测上述漏洞和安全漏洞，因为它们无法推理硬件程序中的复杂值流。这凸显了开展复杂分析以发现深层硬件问题的必要性，也彰显了 Qihe 作为推动硬件分析发展的基础框架所发挥的作用。

其次，检测这些深层硬件问题传统上依赖模型检测和模糊测试等资源密集型方法，以及基于仿真的测试等耗费人力的技术 [12, 23, 38, 45, 70, 83],，而这些方法往往占据硬件开发流程的大量时间，拖慢开发进度。相比之下，静态分析可以在更早阶段部署，而且通常

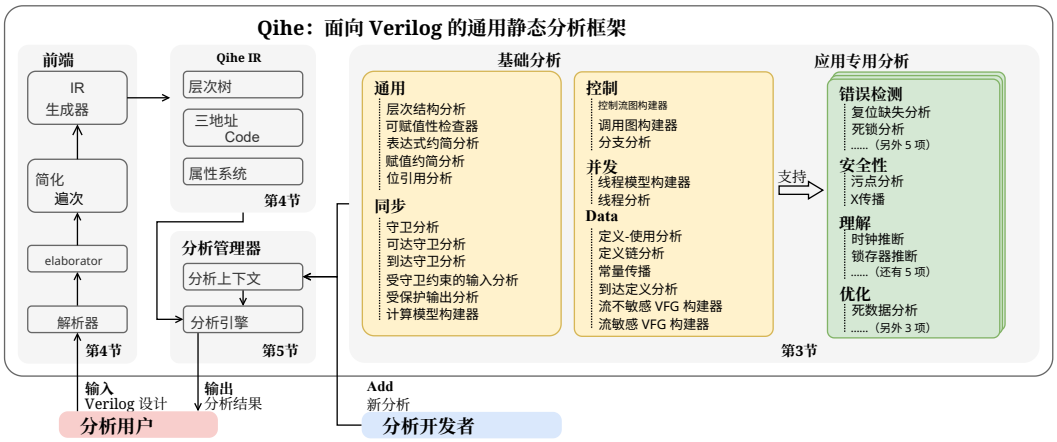


图 1. Qihe 的工作流程。借助各组件的协同作用，它既支持分析用户 执行 Verilog 设计分析，也支持开发人员 创建新的分析。

轻量得多。例如，在一个包含 180 万行代码的真实 RISC-V 片上系统代码库上，Qihe 使用 40 GB 内存，约 5 分钟即可完成全部 20 项客户端分析和 22 项基础分析。

最后，得益于 Qihe框架提供的基础设施，尤其是其丰富的基础分析套件，这些客户端分析的开发成本大幅降低。平均而言，利用 Qihe 的基础设施实现每项客户端分析只需约 300 行 Java 代码；而若从头开发相同的分析，不使用 Qihe 的基础分析，平均则需要超过 5,000 代码行。

总之，本文介绍了 Qihe，这是首个面向 Verilog 的通用静态分析框架，包含超过 10 万行 Java 代码，这构成了本文的独特贡献。Qihe 是开源的，可从其 网站[3]获取。我们还提供了全面的 文档[4]，以支持开发各种特定应用分析。

硬件分析的发展或许无法复制软件分析的广泛成功。然而，25 年前，人们也很难预见静态分析能如此显著地提升软件质量。Qihe 是一项大胆的尝试，旨在挖掘复杂硬件分析的初步潜力。我们希望它能成为软件与硬件社区开展合作的催化剂，共同塑造硬件分析仍然模糊却前景广阔的未来。

2 Qihe 概述

作为一款面向 Verilog 的通用静态分析框架，Qihe 旨在满足两种关键用例：使分析用户能够对其硬件设计执行现有分析，并赋能分析开发者使用 Qihe框架创建多种新分析。图1展示了这两种用例各自的基础工作流程，该流程由四个核心组件提供支持：前端、Qihe IR、多种分析以及分析管理器。下文将结合该图详细介绍这一工作流程。后续章节将进一步讨论 Qihe 的各个组件。

对于分析用户而言，流程始于将 Verilog设计 输入前端，前端会将其转换为 *Qihe IR*，即用于 Verilog 的程序抽象。随后，分析管理器 会执行

[3]https://qihe.pascal-lab.net [4]
https://qihe-docs.pascal-lab.net

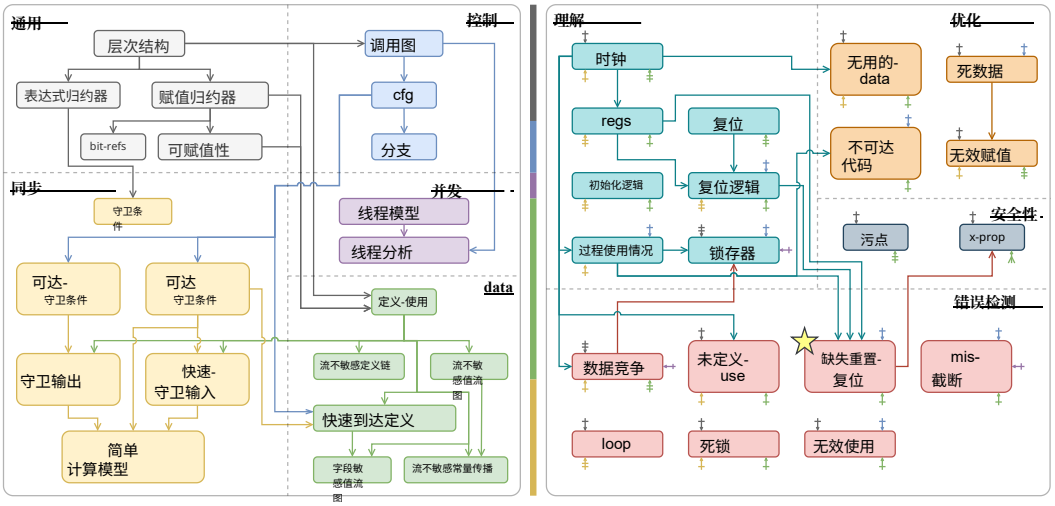


图 2. Qihe 中的分析及其依赖关系。每项分析均以简称表示。左侧展示了分为五组的基础分析，右侧则展示了涵盖四个领域的应用专用分析。这些分析之间的依赖关系以箭头表示。为保持清晰，我们用短箭头（“ ”、“ ”和“ ”）简洁地表示应用专用分析依赖的基础分析数量。这些箭头表示其依赖特定类别中的一项、两项或三项基础分析，类别由箭头颜色（或黑白印刷时的位置）标识。例如，带星号的 missing-reset 分析有两个短箭头“ ”：一个指向右上方，一个指向右下方，表示它依赖于一项控制类别的分析，以及一项数据类别的分析。我们用中间的条形图通过颜色和长度展示各类别的相对使用情况。

用户配置的请求分析，并基于 Qihe IR 将结果交付给用户。具体而言，管理器会检测可用的分析及其配置，将其注册到分析上下文中，并调用分析引擎将其应用于 Qihe IR。分析会根据其依赖关系按拓扑顺序执行，以确保 correct。

对于分析开发者，分析管理器提供了一个直观的界面，可基于现有分析创建新的分析（无论是基础分析还是应用专用分析）。新分析实现后，管理器会自动检测并将其注册到分析上下文中，供用户执行。我们鼓励开发人员利用 Qihe 探索并贡献更多潜在有用的分析。

3 分析套件

Qihe 通过利用各种分析之间的协作，支持开发多种复杂的分析。图 2 展示了 Qihe 中所有主要分析及其依赖关系的概述，体现了这种协作。请注意，基础分析与客户端分析之间的依赖关系以图注中所述的特殊箭头表示。

在此图的指引下，第3.1 节首先说明如何推导和分类这些分析。第3.2 节随后详细介绍缺失复位，这是 Qihe 中一个具有代表性的错误检测客户端，用以展示检测现有方法无法发现的现实世界硬件错误所需的精密技术

用于 Verilog 漏洞检测的代码检查器式静态分析 [16, 32, 57, 66]，并着重指出硬件分析与典型软件分析之间的根本差异。最后，第3.3 节将讨论我们框架中现有的分析如何协同工作，以支持这一关键的硬件缺陷检测客户端。

3.1 所有分析概述

Qihe 提供了丰富的基础分析和特定应用分析。由于目前没有可供参考的 Verilog 通用静态分析框架，我们通过一个结合调研、研究经验和工程实践的三步流程推导出这些分析：(1) 通过调研硬件漏洞和安全问题 [17, 18, 48, 52, 59, 64], 考察综合和代码检查工具中的现有分析 [32, 43, 57, 79], 并直接与硬件从业者交流 [3, 4]，确定分析需求；(2) 在此调研的基础上，并结合我们十年来在软件基础静态分析方面的研究经验，提炼出*22* 项基础分析，为满足不同应用需求提供常用的 Verilog 程序基本信息；(3) 为展示如何在 Qihe 中利用基础分析构建多种应用，我们还开发了*20* 项特定应用分析，以满足调研中发现的常见但并非简单的需求。

为清晰理解这些分析，我们按照其设计目的对其进行分类，如图 2所示。基础分析（左侧）根据其关注的硬件设计具体方面分为五类：通用语言特性、数据与控制流，以及并发与同步特性。特定应用分析（右侧）则分为四个热门应用领域：硬件错误检测、安全性漏洞识别、程序理解和程序优化。

为增强分析的模块化和可复用性，Qihe 中的每项分析都围绕单一目的设计，并将其他职责作为依赖关系交由上游分析处理。这种做法自然形成了复杂的依赖关系，如图 2所示。

为深入理解这些分析及其依赖关系，接下来我们将详细介绍 Qihe 中一项具有代表性的分析（第3.2 节），然后讨论其支持性依赖关系（第3.3 节）。

3.2 硬件复杂分析：一个启发性示例

本节通过一个启发性示例阐释三个关键问题：(1) 硬件错误检测问题是什么样的？它们与软件中的类似问题有何不同？(2) 如何设计复杂的硬件静态分析，利用硬件特定洞见来解决这些问题？(3) 构建复杂硬件静态分析面临什么挑战？

我们的动机示例是缺失复位分析，它用于检测一类关键硬件漏洞：寄存器在电路复位期间未能正确初始化，从而产生不稳定的系统状态，攻击者可能利用这些状态 [5, 20, 48, 50]。它是我们分析套件中一个具有代表性的 Verilog 分析，不仅能够处理软件分析中并不常见的各种硬件特定语义——例如时钟、寄存器和复位，还展示了检测现实世界硬件漏洞所需的复杂程度（见表2）。这些漏洞是现有用于 Verilog 错误检测的代码检查器式静态分析难以发现的 [16, 32, 57, 66]。

下面，我们首先介绍 Verilog 中的缺失复位 问题（第 3.2.1 节），并将其与软件中的经典使用前未初始化 （UBI）问题（第 3.2.2 节）区分开来。然后，我们介绍用于解决该问题的缺失复位分析所依据的核心见解（第 3.2.3 节）。最后，我们利用这一分析讨论开发复杂硬件分析时面临的关键挑战（第 3.2.4 节）。

3.2.1 Verilog 中的缺失复位问题。 当硬件寄存器 缺少适当的复位时，就会出现缺失复位 问题。下面，我们首先介绍有关寄存器和复位的必要 Verilog 背景知识，然后给出问题定义。

寄存器。 硬件寄存器是一种数字电路组件，在 0 和 1 之间交替变化的驱动时钟信号控制下存储并更新值；Verilog 使用变量和 始终 块对硬件寄存器进行建模。如图 3 所示的代码片段中，累加器是一个硬件寄存器，其驱动时钟在 始终 块头部（第 1 行）中指定，其值更新逻辑则在块体（第 2–5 行）中指定。功能上，这段 Verilog 程序描述了一个累加器，其中寄存器 累加器 在0 时更新为 复位 为 1 （第 2–3 行），否则（第 4–5 行）更新为累加器 +输入，每当驱动时钟信号 时钟 从 0 变为 1 时，即发生时钟上升沿（如第 1 行的 上升沿 关键字所指定）。为说明这一动态行为，图 3中的表格展示了变量值随时间演变的示例：

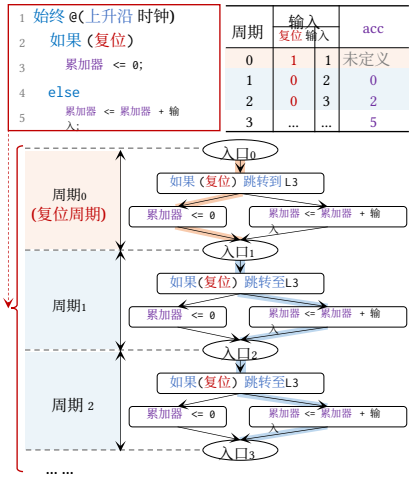


图 3. 正确复位的寄存器示例。

- “周期”列枚举时钟周期，其中数字 i 表示该行显示所有变量在第 i个时钟周期期间的值，记作 Cycle$_i$

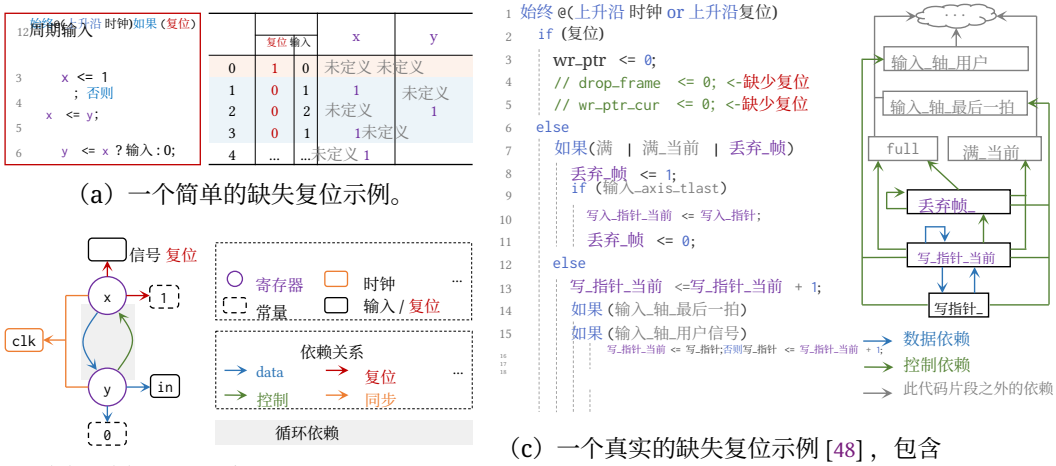
下文中的周期均指此定义。这里，时钟周期是时钟信号 clock 两个连续上升边之间的时间间隔。

- “输入”列提供输入变量（如 reset 和 in）的值。这些变量未在此代码片段中定义，而是由外部在每个周期提供（例如，在电路执行期间拨动开关或按下按钮）。

- “acc”列显示寄存器 acc 随时间变化的值。为简便起见，我们使用 x_i 表示变量 x 在 Cycle$_i$期间的值。在 Cycle$_0$，所有寄存器最初都持有未定义值，即在此示例中 acc_0 =未定义。在每个周期中，第 2–5 行会执行，并且新的寄存器值会在下一个周期生效，这体现了 Verilog 非阻塞赋值运算符 <=的特殊语义：该运算符用于模拟时钟驱动的硬件寄存器更新，不同于软件赋值。例如：(1) 在 Cycle$_0$期间，acc <= 0 （第 3 行）执行，因为 $\text{reset}_0 = 1$，使得 $\text{acc}_1 = 0$；(2) 在 Cycle$_1$期间，acc <= acc +in（第 5 行）执行，因为 $\text{reset}_1 = 0$，使得 $\text{acc}_2 = \text{acc}_1 + \text{in}_1 = 0 + 2 = 2$；(3) 在 Cycle$_2$期间，相同的更新再次生效，得到 $\text{acc}_3 = \text{acc}_2 + \text{in}_2 = 2 + 3 = 5$。

为进一步说明，图3 的下半部分展示了一个周期展开控制流图（控制流图），通过以橙色和蓝色突出显示执行路径，直观呈现这种动态行为。在该表示中，入口$_i$ 表示在始终块体（第 2–5 行）期间进入i 周期。

复位。复位信号是数字设计中最基本且对可靠运行至关重要的控制信号之一， [76], 它确保电路从已知的初始状态开始运行，而非从上电时默认的未定义状态开始。例如，在图3的周期$_0$（橙色高亮部分）中，复位$_0$ 被设为 1，确保累加器正常运行时以确定的 acc = 0 开始，而不是以未定义的 acc = 未定义状态开始。我们将复位信号处于有效状态（即复位 = 1）的周期称为复位周期。该复位周期对于可靠正常运行至关重要：只有经过该周期，我们才能保证在此后任何非复位周期$_i$（其中 $i \geq 1$ 且复位$_i = 0$）中，累加器都能可靠地保持不变量 $\text{acc}_{i+1} = \sum_{j=1}^{i}$ 输入$_j$。



（a）一个简单的缺失复位示例。

（b）（a）的硬件依赖图。

（c）一个真实的缺失复位示例 [48]，包含我们在评估中发现的前两个漏洞（表2）。

图 4. 缺失复位示例及其硬件依赖关系图。

按照惯例，任何具有复位能力的电路的合法执行都应从一个复位周期开始以建立初始状态，随后进入非复位周期以正常运行，如图3所示。这明确划分了职责：硬件开发者应确保其 Verilog 程序在所有合法执行中都能可靠运行，而硬件用户应执行合法操作（即在正常运行前先复位）。因此，开发人员关注的主要漏洞是在合法执行期间可能出现的漏洞，因为这些问题属于他们预期的职责范围。

问题定义。如果存在一条缺失复位问题执行路径（即一个复位周期后接若干非复位周期，如图*3*所示），在该路径上，寄存器的未定义值无限次被使用，那么该寄存器就存在缺失复位问题。“无限次”是数学中的标准术语 [22]。在我们的语境中，它表示对于寄存器 x，在任意给定的周期N之后，x 的未定义值都会在某个更晚的周期n 再次被使用，其中 $n > N$。这个问题尤其危险，因为它意味着寄存器可能在任何时候不可预测地持有未定义值，即使在合法执行期间也是如此，从而损害硬件可靠性，并可能导致致命的功能缺陷。相比之下，未定义值有限次出现的情况在硬件中很常见且可以接受（例如，在流水线设计中 [75]），这种瞬态行为不应被视为漏洞。

图 4a 展示了一个缺失复位示例，其中寄存器 x 和 y 都存在这一问题，如右侧表格所示。在复位周期 Cycle$_0$ （以橙色阴影标示）中，第 3 行执行，使 x 在 Cycle$_1$，中从未定义更新为 1，而 y 由于在该周期中未被赋值，仍保持未定义。在每个非复位周期 Cycle$_i$ （$i \geq 1$，以蓝色阴影标示）中，第 5–6 行执行，根据 Registers 段落中介绍的 Verilog 非阻塞赋值语义，导致 $x_{i+1} = y_i$ 和 $y_{i+1} = x_i$？在 $_i$ 中的值更新为：0。更具体地说，Cycle$_1$，之后，我们有 $x_2 = y_1 =$ 未定义，以及 $y_2 =$（x_1？在 $_1$ 中：0）=（1 ? 1 : 0）= 1；Cycle$_2$，之后，我们有 $x_3 = y_2 = 1$ 和 $y_3 =$（x_2？在 $_2$ 中：0）=（未定义 ? 2 : 0）= 未定义；以此类推。因此，未定义值在 x 和 y 之间来回传播，如图 4a 的表格所示，展示了缺失复位问题：x 和 y 的未定义值被无限次使用。为解决这一问题，开发者应在 Cycle$_0$，中为 y 添加复位，此外还要保留现有的 x 复位

（第 3 行），从而确保 y_1 已定义，并防止 x 和 y 的未定义值被无限次使用。

3.2.2 缺失复位与 *UBI*。Verilog 中的缺失复位问题与经典的使用前未初始化（UBI）问题相似。该问题在 Linux 内核等大型系统中十分普遍，并推动了数十年来针对 UBI 检测的静态分析研究 [44, 82]。尽管二者都关注未定义值相关错误，但缺失复位分析与 UBI 分析有着根本区别，因为它具有硬件特定性质：

• 硬件特定的问题定义：缺失复位问题关注的是这样一个寄存器：其未定义值在一次合法的 *Verilog* 执行中被使用无穷多次；而 UBI 问题关注的是这样一个变量：其未定义值在任意一次软件执行中被使用。因而，缺失复位分析必须建立硬件特定洞见，以处理“无穷多次”和“合法”这两个 *UBI* 问题定义中不存在的约束。第3.2.3 节将进一步阐述这些洞见。

• 硬件特定信息要求： 缺失复位问题的根源在于硬件语义，包括时钟、寄存器和复位，这些内容超出了 UBI 的范围。因此，检测此类漏洞自然需要超出传统 UBI 分析需求的基础硬件信息。值得注意的是，从 Verilog 程序中获取各种基础信息需要一系列并不简单的硬件分析，这一挑战将在第 3.2.4 节中进一步阐述。

这些区别凸显了建立硬件特定洞见的必要性：一方面，需要设计能够捕获硬件特定问题的硬件错误检测分析；另一方面，需要能够从 Verilog 程序中提取基本硬件语义信息的基础硬件分析。

3.2.3 我们的缺失复位分析的洞见。 我们的缺失复位分析的关键洞见是：只有寄存器之间存在循环依赖 的电路才可能出现缺失复位问题。为了解释这一点，考虑相反的情况：如果一个电路的寄存器之间不存在循环依赖，那么这些寄存器必然存在一个拓扑顺序。按照该顺序，来自常量或输入的已定义值最终可以传播到所有寄存器，从而确保在任何合法执行中，都不会有寄存器的未定义值被无限次使用。

基于这一硬件特有的洞见，我们的缺失复位分析通过报告硬件依赖图中涉及循环依赖的未复位寄存器，识别缺少适当复位的寄存器。硬件依赖图。该图捕捉硬件信号之间的数据、控制、复位和同步依赖关系，这些信号包括寄存器、时钟、复位、输入和常量。例如，图4b 展示了图4a中缺失复位示例的硬件依赖图。下面对该硬件依赖图进行说明。

• 从 x 到 y 的蓝色边表示 x 对 y 存在数据依赖（由图4a <= 第 5 行中的xy引入），而从 y 到 x 的绿色边表示 y 对 x 存在控制依赖（由图4a第 6 行中的yx ? ...引入）。这些边共同构成了 x 与 y之间的循环依赖（灰色区域所示），这也是未定义值在二者之间来回传播的根源，详见第*3.2.1*节“问题描述”段。

• 从 x和 y到时钟信号 clk 的橙色同步边表明， x和 y是由时钟 clk 同步的寄存器。

• 从 x 指向复位信号 rst 和常量 1 的红色复位边，描述了 x 的复位逻辑——在 rst 的控制下将 x 复位为 1。

更具体地说，缺失复位分析包括以下四个步骤：(1)不可综合代码排除：Verilog代码库通常包含为仿真编写的不可综合代码

，这些代码不会出现在最终的综合硬件中，也不可能包含真正的硬件漏洞。因此，我们会将此类代码排除在分析之外。(2)硬件依赖图构建：通过解析Verilog程序中的各种依赖关系，构建硬件依赖图（例如图4b所示）。(3) 循环依赖识别：在硬件依赖图中定位周期（图4b中的灰色区域），以识别参与循环依赖的寄存器（例如图 x 和 y 所示）。(4) 缺失复位报告：报告步骤 (2) 中缺少适当复位的寄存器。在图4b中， y 因未复位而被报告，而 x 则不会，因为它通过1 被复位复位到4b）。

为了展示现实世界中的缺失复位错误，图 4c 给出了这样一个示例及其硬件依赖图。为清晰起见，该图仅保留数据依赖和控制依赖。代码片段源自 Verilog-AXIS 项目中的一个先进先出（FIFO）缓冲区 [25],，这是一个广受欢迎的开源设计，拥有超过 800 个 GitHub 星标。我们的分析通过上述四个步骤检测出寄存器 `drop_frame` 和 `wr_ptr_cur` （图 4c中的第 4–5 行）存在缺失复位错误。这些错误是致命的：由于 `drop_frame`（帧丢弃状态寄存器）缺少复位，会导致数据包被意外丢弃；而 `wr_ptr_cur`（写指针寄存器）缺少复位，则会导致数据被写入错误的缓冲区地址。

3.2.4 构建复杂硬件静态分析所面临的挑战。 即使已经确立了核心洞见，构建复杂的硬件分析仍然具有挑战性，因为缺乏专注于提取基础硬件信息的分析方法，而这一主题鲜有先前研究进行探索。

例如，为了检测涉及硬件寄存器的缺失复位问题——这些寄存器由时钟信号驱动，并由复位信号复位——根据第*3.2.3*节总结的见解，分析需要掌握关键硬件信息，以确定每个变量是否表示硬件寄存器、时钟信号、复位信号，或以上皆非。然而，这并非易事，因为 Verilog 变量并未显式声明此类信息。通常，只有经过漫长的综合过程，根据行为语义将 Verilog 代码映射到物理硬件组件后，才能揭示这些信息。为弥合这一差距，满足缺失复位分析及其他需要此类硬件信息的客户端的需求，我们开发了硬件理解分析，以推理此类行为语义（例如，推断寄存器、时钟和复位信号），使我们能够在无需采用重量级综合的情况下，高效获取缺失复位分析所需的信息。

此外，这些硬件理解分析还需要更基础的硬件信息，这些信息对于错误检测客户端和未来可能出现的客户端（例如硬件安全分析）也至关重要。因此，需要进行多种分析，包括但不限于：(1) Verilog 模块层次结构分析，以支持模块间推理；(2) 硬件数据流和控制流分析，用于构建各种图结构（例如，缺失复位分析所使用的硬件依赖图）；(3) 用于捕获硬件同步与并发特征的分析，这对于推理硬件理解中的行为语义至关重要；(4) 位选取/部分选取和位向量算术分析，这些分析对于在硬件分析中实现实用的精确率不可或缺。

最终，构建复杂的硬件静态分析要求我们发掘多样化的基础分析，理解它们之间的相互依赖关系，并实现它们之间的互操作性。为此，我们在图2中介绍了组织良好的分析套件。接下来，我们将讨论支持缺失复位分析的分析。

3.3 支持复杂分析：案例研究

尽管缺失复位分析的四个步骤已在第3.2.3节中简要介绍，但其实现依赖于分析套件中近二十项分析的协同，总计 9,700

行代码。图5 展示了支持缺失复位的所有分析，并按拓扑顺序排列，从基础分析（顶部）到分析客户端（底部）。下文将借助图5帮助读者理解为何缺失复位分析需要其依赖关系，以及这些依赖关系如何提升该分析的整体效能，从而深入了解我们的分析套件如何组织起来以支持复杂分析。为便于阅读，此处不讨论算法细节；感兴趣的读者可在我们的开源项目中查阅。

相互依赖。 为阐明 缺失复位分析为何需要图 5 中的依赖关系，同时保持简洁易读，我们回溯了图中的粗体依赖箭头，以强调这些分析的功能及其相互依赖的必要性。首先，(1) 由于缺失复位错误仅涉及寄存器，缺失复位分析直接依赖寄存器分析来推断物理寄存器，并利用定义-使用和分支分析构建包含数据依赖和控制依赖的图以检测周期，如第 3.2.3 节所述；(2) 由于寄存器由时钟信号同步，寄存器分析紧密依赖时钟分析来推断物理时钟；(3) 由于时钟在 Verilog 的各模块间作为数据传播，时钟分析直接依赖流不敏感定义链(其中 `fi`表示流不敏感分析)，该分析提供表示数据流的模块间图，以支持此类推理；(4) 最后，流不敏感定义链利用定义-使用分析来获取

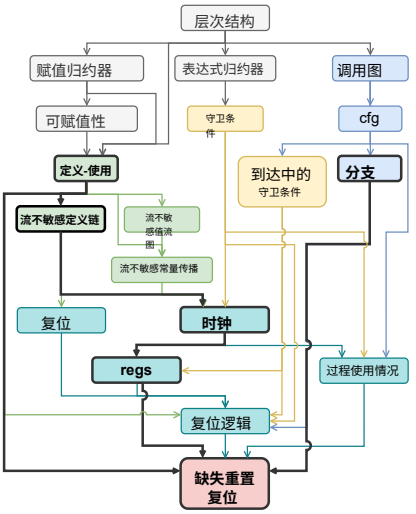


图 5. 支持缺失复位分析的所有分析。

变量，包括模块间关系，用于构建模块间图。图 5 中较浅的依赖箭头 也以类似粗体箭头的方式表示 缺失复位；为简洁起见，我们不再赘述这些依赖关系。

分析及其下游有效性。 这些相互依赖关系凸显了每项分析的有效性对下游分析至关重要。例如，时钟分析的精确率降低（即将非时钟变量误分类为时钟）会直接影响寄存器分析的精确率，因为寄存器由时钟信号同步；寄存器分析的精确率不足可能导致缺失复位分析产生过多误报，从而削弱其实用价值。同样，时钟分析和寄存器分析的召回率不足（即可靠性降低，未能涵盖某些真实行为）可能导致缺失复位分析漏掉实际存在的缺失复位问题。为应对这些挑战，我们对关键的支持性分析进行了设计，以确保其具有较高的有效性。为便于阅读，我们介绍两种代表性分析的设计考量：寄存器分析和流不敏感常量传播。前者展示了我们如何设计一种软件领域中没有的硬件分析，以实现较高的召回率和精确率。后者——一种流不敏感常量传播分析——则表明，我们分析套件中的某些分析虽然借鉴了软件概念，但仍需针对 Verilog 进行适配，才能在硬件上取得有效结果。

regs（寄存器推断分析）可预测哪些 Verilog 变量会被综合为硬件寄存器，并识别其驱动时钟。为了达到较高的精确率，它不能简单地将所有声明为 Verilog 的 reg 关键字的变量都报告为寄存器，因为许多此类声明表示的是中间变量，而非物理寄存器（该关键字容易造成误导），这会产生大量假阳性。regs 不同于此，而是根据寄存器的语义对其进行分类。寄存器更新与时钟沿同步，在 Verilog 中表现为由事件

控制保护的非阻塞赋值，而这些控制会引用时钟信号；因此，regs 的核心步骤是定位此类非阻塞赋值，并将其左侧（LHS）变量报告为寄存器。回想一下，图3展示了一个事件控制（例如，@(posedge clock)）保护一个非阻塞赋值（例如，acc<= acc +in）的情况，其中左侧变量 acc 就是寄存器。要达到较高的召回率，关键在于可靠地提取非阻塞赋值与事件控制之间的保护关系。仅匹配图3所示的模式无法可靠地得出保护关系，因为事件控制和非阻塞赋值可以在 always 块中以任意顺序出现，从而形成复杂的保护关系。例如，always 块的主体可以是 @(posedge clock1); x <= 1; @(negedge clock2); y <= 0;，其中 y <= 0受后面的 @(negedge clock2) 保护，而非前面的 @(posedge clock1) 保护。因此，我们通过在控制流图上执行可靠的数据流分析来计算守卫关系，所用的是基础的到达保护分析。regs 依赖于该分析，如图5所示。

流不敏感常量传播 在我们的分析套件中得到广泛应用，其精确率对于下游分析的有效性至关重要。与典型的软件程序不同，要在 Verilog 中实现高精确率，需要进行位级常量推理，因为 Verilog 程序经常对物理导线对应的单个位或位向量进行操作。因此，流不敏感常量传播 准确地建模 Verilog 的四值逻辑（0/1/X/Z）和位向量算术，从而实现精确的位级常量传播。我们在第 6.2 节讨论一个得益于这一设计而检测到的错误。请注意，我们没有采用软件分析中常用的常量传播流敏感变体，因为我们发现它只能略微提高精确率，却会显著降低分析速度。这主要是因为大多数 Verilog 变量只在单一位置定义，这反映了硬件信号通常只有一个定义，因此流敏感性带来的益处较小。这进一步凸显了针对 Verilog 的适配的必要性。

关于可靠性的讨论。 如上所述，我们的硬件理解分析（例如时钟和 reg）若不可靠，可能会导致缺失复位分析也不可靠。尽管我们努力在实践中保持较高的召回率，但必须注意，这些分析不可能达到完美可靠性。这是因为这些硬件理解分析试图预测综合工具生成的电路结构，而 Verilog 标准 [1] 仅规定电路行为，并未规定其结构。因此，不同的综合工具可能会根据相同的 Verilog 代码生成结构不同的电路，而且没有正式标准可作为可靠预测的真实依据。尽管存在这一不可避免的局限，综合工具在实践中往往遵循通用的映射惯例，因此我们的分析仍能有效预测有用的综合结果。

排除这一不可靠性的来源后，可达闭包（即缺失复位结果所报告寄存器在硬件依赖图中可达的寄存器集合）可以可靠地捕获所有可能的缺失复位问题。闭包之外的寄存器不会出现缺失复位问题，因为它们不参与电路中的循环依赖。根据我们在第 3.2.3 节的见解，这些寄存器的值最终由常量、输入或复位寄存器计算得出，而在任何合法执行中，这些值在复位周期之后都有明确定义。

我们的分析具有协作性，这一点不仅体现在上述内容中；在开发更复杂的硬件静态分析时，协作也会变得更加复杂。我们希望缺失复位的示例能为开发人员提供参考，让他们利用分析套件中已有的分析来创建自己的应用。

4 面向分析的中间表示与前端

在硬件分析框架的背景下，中间表示和前端的设计——这是编译器中的经典课题——需要格外谨慎。

基本定义

（标识符）id, key,标签：= 符号（层次标识符）hierId：= (id.)*id.id（值）
val：= 位向量和实数（类型）τ：= 位向量类型、实数类型和数组类型（属性）
attr：= (* key =val (, key =val)* *)

层次结构

（设计）*design*：= *attr*?*module**（模块）*module*：= *moduleid attr*? *{ (port*? *net)* (port*? *var)** *const***instModule** *proc** *func** *}*（网络）*net*：= *netKind id*： τ*; attr*? *netKind*：= *wire* | *wor* | *wand* | *uwire* | · · ·（变量）*var*：= *var id*： τ*; attr*?（常量）*const*：= *constid* =*val ;attr*?（模块实例化）*instModule*：= *instid*：*moduleId ;attr*?（过程块）*proc*：= *procattr*? *{stmt** *}*（函数）*func*：= *funcid (args*? *)*->*(args*? *)attr*? *{var** *const** *stmt** *} args*：= *id (,id)**

三地址码

（语句）*stmt*：= 标签： | 非标签语句 *attr*?非标签语句：= 赋值语句 |守卫条件语句 | *if*语句 | *case*语句 |*goto*语句| 调用语句 | 接收语句 | 系统调用语句 | *return*； |*pass*；赋值语句：= （局部加载 | 局部存储 | 层次加载 | 层次存储 | 计算）；守卫条件语句：= 时序控制；*if*语句：= *if id goto* 标签；*case*语句：= （*case* | *casex* | *casez*）*id {*（*expr*：*goto* 标签；）*（*default*：*goto* 标签；）?*}* *goto*语句：= *goto* 标签；调用语句：= *invoke*（*id* | 层次*id*）（参数?）；接收语句：= *receive*（参数?）；系统调用语句：= *syscall id*（参数?）（-> （参数））?；参数：= *id*（, *id*）*（赋值）局部加载：= *id*（= | <= | <-）局部访问表达式 局部存储：= 局部访问表达式（= | <= | <-）*id* 层次加载：= *id*（= | <= | <-）层次访问表达式 层次存储：= 层次访问表达式（= | <= | <-）*id* 计算：= *id*（= | <= | <-）计算表达式（表达式）*expr*：= 局部访问表达式 |层次访问表达式 |计算表达式 局部访问表达式：= *id*（[*id*]）?选择器?层次访问表达式：= 层次*id*（[*id*]）?选择器?选择器：= [*m*:*n*] | [*id*（+ |-）：*n*]（*m*, *n*：= 非负整数）计算表达式：= 一元运算符 *id* | 二元运算符 *id id* | *id*? *id*：*id*|（*zext* | *sext*）*id to* τ | *cast id to* τ一元运算符：= *neg* | *not* | *rand* | *ror* | *rxor* |*buf* 二元运算符：= 加法 | 减法 | 乘法 | 无符号除法 | 有符号除法 |无符号余数 | 有符号余数 |幂 | 无符号幂| 无符号小于 | 无符号大于 | 无符号小于等于 | 无符号大于等于 | 有符号小于 |有符号大于 | 有符号小于等于 | 有符号大于等于| 等于 |等价 | 不等于 |不等价 | 与 | 或 | 异或| 左移 | 算术右移 | 逻辑右移 |拼接（时序控制）时序控制：= # *id* | @ （事件表达式列表?） | *repeat* （*id*） @ （事件表达式列表?）事件表达式列表：= 事件表达式（或 事件表达式）*事件表达式：= 表达式 | 上升沿 表达式 | 下降沿 表达式

图 6. QiheIR 的语法，其中使用等宽粗体字体表示终结符，斜体表示非终结符，并使用灰色字体省略通用定义以简化内容。符号*pattern*? 表示*pattern* 是可选项。*pattern**表示零次或多次出现*pattern*。第4节提供了一个详细示例（图7），以突出 Qihe IR 的关键设计特性。



图 7. 从 Verilog 程序导出的 Qihe IR 示例。右侧的每个方框都表示一个 IR 节点，用于建模左侧的 Verilog 层次语言结构，相关展开内容见第4.1节。特定节点（例如 Proc）包含三地址码语句，详见第4.2节。每个节点可以有零个或多个属性，其用途将在第 4.3 节和第 4.4节中讨论。名称以 $ 为前缀的节点由前端生成。

关于中间表示，Qihe 引入了一种称为 Qihe IR 的硬件程序抽象，旨在捕获用于描述硬件的所有 Verilog 特性[5] 同时保持足够的简洁性，以便于开展分析。图 6 展示了 Qihe IR 的语法。除了一些基本定义外，Qihe IR 由两个主要部分组成：层次结构和三地址码（3AC）。我们不对 Qihe IR 的完整语法和语义进行详尽阐述，因为这会使内容冗长且难以理解；相反，我们在第 7图中通过示例突出介绍这两个组成部分的关键设计，分别见第 4.1 和 4.2节。读者可参阅我们的项目文档，以获取有关 Qihe IR 的更多详细信息。

关于前端，除了将 Verilog 代码转换为 Qihe IR 这一标准任务外，我们发现它还必须与 Qihe IR 协同工作，以满足硬件分析中的特定需求。我们重点介绍两个重要方面：(1) 如何满足不同分析看似相互冲突的需求（第 4.3 节），以及 (2) 如何处理不完整的程序以支持复杂分析（第 4.4 节）。有关前端的其他设计和实现细节，请参阅 Qihe 的开源代码库。

4.1 Verilog 层次结构的直观树表示

Verilog 设计由模块构成，模块可以包含网络、变量、子模块实例化、函数及其他构造，从而形成嵌套的层次结构。Qihe IR 的高层结构旨在紧密对应硬件设计的这一层次组织。如图层次结构部分所示，图7中 Qihe IR 的语法从包含模块的设计开始。这些模块又包含网络、变量、模块实例化、过程块和函数，直接反映了 Verilog 的层次结构。这种设计使熟悉 Verilog 的用户能够快速理解 IR，并在开发分析时仅凭 Verilog 源代码作为参考，直观地浏览其中的内容。

[5]Verilog 语言包含用于描述硬件的可综合结构，以及用于编写测试平台的不可综合结构。不可综合特性与 Qihe 的硬件分析无关，因此本文不予讨论。在此语境下提及 Verilog 时，我们关注的是其可综合子集。不过，Qihe 确实支持某些常用的不可综合特性，以拓展其应用范围。此外，Qihe 还支持 SystemVerilog 的常用特性 [2], , SystemVerilog 是 Verilog 的扩展。

本节将以图7 中的示例说明 Qihe IR 的层次结构与 Verilog 结构之间的对应关系，并在此过程中解释 Verilog 构造的语义，帮助可能不太熟悉 Verilog 的读者理解。该示例涵盖了 Verilog 中最常用的构造。

如图7所示，左侧展示了一个名为demo.v的 Verilog 程序，右侧则展示了对应的 Qihe IR。Qihe IR 的层次结构以树形而非文本语法的形式呈现，因为树形表示能更直观地传达不同 Verilog 结构之间的层次关系，也更接近分析所使用的 Qihe IR 内存表示。树中的每个节点都对应一个 IR 结构，并标有名称和类。名称用于标识节点，并尽可能从 Verilog 代码中提取。类表示该节点所代表的 Qihe IR 结构，对应于图6中定义的非终结符。下面我们从上到下查看这棵树。

中间表示的根节点是一个名为demo 、类为设计的节点，表示左侧图7中的 Verilog 代码demo.v 所描述的整个硬件设计。名称demo 源自 Verilog 文件名demo.v。设计 类对应于图6中定义的设计非终结符。除非必要，后续讨论中我们不再详细解释节点名称和类，因为它们大多不言自明。

根节点有两个模块节点作为其子节点，每个节点代表一个 Verilog 模块，这反映了 Verilog 设计由一组模块定义组成这一事实。在 Verilog 中，模块是封装可复用电路设计的基本单元，类似于函数封装可复用的计算逻辑，并且可以有称为端口的输入和输出。例如，图7 左侧定义了一个名为 accSqrDouble 的 Verilog 模块（第 1 行），其输入为时钟和输入（第 2–3 行），输出为输出（第 4 行）。该模块封装了一个电路：它将来自输入的输入值累加到寄存器累加器中，并由时钟时钟同步（第 7–8 行）；对累加器中的值进行平方（第 9–10 行）和加倍（第 11 行），并将结果输出到输出（第 11 行）。下面我们沿着accSqrDouble 节点进一步说明这些细节。

该accSqrDouble 节点包含若干子节点，分别表示模块中描述电路的不同结构。首先，网络和Var 分别对应 Verilog 的wire （第 2–5 行）和reg （第 6 行）。这些结构用于描述物理寄存器和连线。请注意，模块端口是带有额外wire或reg关键字的特殊输入或输出（第 2–4 行）。尽管 Verilog 还包含用于声明特殊类型连线的其他关键字，例如 wand 和 wor ，Qihe IR 为简化起见，使用网络类统一表示所有与连线相关的结构。此外，Qihe IR 使用 Var 而非Reg 来表示 Verilog 的reg声明。这是因为在 Verilog 中，reg 关键字并不一定意味着硬件中存在物理寄存器；它仅声明一个类似软件变量的变量，该变量根据其在设计中的用法，可能映射为连线或寄存器，也可能被完全优化掉。若要确定某个Var是否映射为物理寄存器，开发人员可以使用 Qihe 的reg 分析功能获取相关信息，详见第3.3节。

接下来，accSqrDouble 节点由于第 11 行的模块实例化，包含一个类为InstModule 的子节点。模块实例化是 Verilog 中复用封装电路逻辑的基本机制。在此示例中，行 doubler i(sqr,out) 首先创建一个名为 i 的doubler模块实例doubler 的定义类似于图 8所示的定义，其中封装了一个将输入值加倍并输出结果的电路。接着，实例化操作将 wire sqr 连接到模块实例$t1 i 的输入端口，并将 wire out 连接到输出端口 $t2，确保 使输出携带的值是 sqr 的两倍，而无需重新定义实现倍增逻辑的模块

```
模块 doubler(输入线 $t1, 输出线 $t2);// 将 $t1 加倍并 // 赋值给 $t2endmodule
```

图 8. 图 doubler中模块的推断签名，见图 7。

。需要注意的是，`InstModule` 节点仅表示模块实例 `i`，而端口连接则由 `Proc` 节点单独建模。我们将在下一段进一步阐述这一设计选择。此外，在许多现实场景中，由于第三方 IP 核广泛受到知识产权限制，类似 `doubler` 这样的模块定义可能无法获取。为处理此类情况，我们的前端会自动推断缺失模块的关键信息，例如其端口的类型和方向。图 8 所示的代码实际上是我们的前端在处理图 7中的 Verilog 程序时，在缺少 `doubler`定义的情况下推断得到的；图 7 右上角展示了表示这一推断模块的子树。这些推断信息对于实现复杂分析至关重要，我们将在第 4.4 节详细介绍推断过程。

最后，`accSqrDouble` 节点包含四个`Proc` 节点，统一表示 Verilog 的始终块（第 7–8 行）、连续赋值（第 9–10 行）和端口连接（第 11 行）。这三种结构是 Verilog 中描述电路逻辑最常用的结构。然而，它们在语法上存在显著差异，这使分析逻辑的开发变得复杂。若不进行统一，开发人员就必须分别处理每种结构，导致代码冗余且复杂。为应对这一挑战，Qihe IR 引入了`Proc` 节点（过程块），它使用三地址码（3AC）对这些结构进行统一表示。这种统一之所以可行，是因为根据 Verilog 的仿真语义， [1, 15], 所有这些结构都可以解释为并发进程，通过执行语句来模拟物理电路的行为。使用`Proc` 这一术语正是为了体现这一点。在第4.2节中，我们将详细阐述这些结构如何描述电路逻辑，以及三地址码如何设计以对它们进行统一建模。

4.2 Verilog 电路逻辑的统一三地址码表示

除了 Verilog 的层次结构之外，Qihe IR 还使用三地址码（3AC）对电路逻辑建模。三地址码设计通过以直观且统一的方式表示各种 Verilog 结构，简化了分析的开发。图 6 展示了 Qihe IR 中的所有三地址代码语句。对于熟悉 Verilog 的读者而言，其中许多语句都直观地对应于 Verilog 结构。因此，在本文中详细解释每条三地址代码语句既多余又繁琐，所以我们将这些具体内容留给文档介绍。相反，我们重点介绍一个关键设计特点：如何使用三地址码统一表示不同的 Verilog 结构，以描述电路逻辑。我们以图 7为例，解释最常用的三种 Verilog 结构的语义：始终 块、连续赋值和端口连接，以及如何在 Qihe IR 中使用三地址码对它们所描述的电路逻辑建模。

始终块。 always 块使用语句来描述变量之间类似软件的计算逻辑，Verilog 综合器会将其映射为物理导线和寄存器之间的电路逻辑。在此示例中，always 块的主体包含语句 `@(clk) acc <=acc + in`，这表示每当 clk 发生变化（@clk）时，寄存器 acc 都会在其先前值的基础上加上 in。请注意，Verilog 引入了一种特殊的非阻塞赋值运算符 `<=`，它具有独特且重要的作用，可用于区分 always 块中的寄存器更新与类似软件的变量赋值。always 块的行为由 `[0]`:Proc 节点中的三地址代码语句建模[6]。 `[0]`:Proc 包含一个无限循环，通过语句 @(clk) 监视时钟，并在时钟发生变化时使用 add acc in 更新 acc，这与 always 块的行为直接对应。为了正确建模 Verilog 中的寄存器更新，我们保留非阻塞赋值运算符 `<=`，如第 8 行所示。

[6]由于 always 块、连续赋值和端口连接均未命名，因此我们使用索引（例如 `[0]`, `[1]`,）根据它们在源代码中的出现顺序来引用它们。

连续赋值。 虽然 always 块足以描述任何电路逻辑，但 Verilog 还提供了连续赋值，以便更简单地表达组合逻辑（即无状态逻辑）。连续赋值会在赋值运算符左侧和右侧的连线与变量之间建立约束。例如，assign sqr = acc * acc 指定 sqr 应始终保持 acc * acc 的值。Verilog 综合器会生成相应的乘法电路逻辑，以满足这一约束。此行为由 [1]:Proc 建模，它会监视 acc 值的变化，并相应地更新 sqr。它与 [0]:Proc 类似，因为连续赋值通常可以替换为等效的 always 块；因此，我们将二者统一归入 Proc。不过，关键区别在于使用了 <- 赋值运算符，该运算符在 Qihe IR 中专为支持 Verilog 的多驱动特性而设计。由于这一特性，连续赋值并不总能完全替换为 always 块。在 Verilog 中，允许多个连续赋值指向同一根 wire，例如“assign u = 1; assign u = 0”。u 的最终值由预定义规则合并这些赋值来确定。例如，如果 u 声明为 wire，合并后的值就是 Verilog 的特殊未知值 X；如果 u 声明为 wand（线与），合并后的值就是 0 & 1 = 0。为了对此行为建模，我们引入 <- 运算符，用于记录所有赋给 u 的值。当存在多个 <- 赋值时，会合并所有记录的值，以确定 u 的最终值。

端口连接。端口连接在模块实例化期间隐式建立，其语义与连续赋值相同。例如，图 7 第 11 行对模块 doubler 的实例化创建了两个端口连接：一个从 sqr 连接到实例 i 的 $t1，另一个从实例 i 的 $t2 连接到 out。这意味着，每当 sqr 发生变化，其值就会赋给 $t1；每当 $t2 发生变化，其值就会赋给 out。此行为通过[2]:Proc 和 [3]:Proc 建模，方式类似于 [1]:Proc 对连续赋值的建模。请注意，我们使用层次化引用 i.$t1 来指代实例 i 的 $t1 端口，从而避免语句中出现歧义；i.$t2 也同样如此。

总而言之，尽管 Verilog 提供了便于简化硬件设计的构造，但这些特性增加了电路逻辑分析的复杂性。通过将这类构造转换为包含三地址代码语句的 Qihe 过程块，Qihe 使分析开发者能够以统一的方式处理它们。在此转换过程中，Qihe IR 还引入了软件分析中间表示中不常见的语句和运算符，例如守卫语句和 <- 运算符，以准确捕获 Verilog 语义。

图 7 中尚未讨论的其余部分是属性系统，它支持将键值对关联到 Qihe IR 的层次结构和语句。该系统与我们的前端紧密协作，以支持其他面向分析的功能，包括以下两节所述的功能。

4.3 面向多样化分析需求的可扩展中间表示

在应对不同分析的多样化需求时，我们发现了一些看似相互冲突的要求：(1) 某些分析要求 Qihe 通过将复杂的语言构造简化为基本形式来简化程序，从而免于处理源代码中的冗余细节。(2) 另一些分析则要求 Qihe 保留大量源代码级信息，因为这些信息在简化过程中可能会丢失，从而帮助它们推断用户意图或加快分析速度。

为满足这两方面的需求，Qihe IR 以一个简单的核心为基础，并辅以可扩展的属性系统：(1) 如前文所述，其基本构造构成了 Verilog 的一个小型核心，在语义上等同于完整语言。该核心通过将语法糖和复杂构造进行语法糖消解，简化 Verilog 并将其转换为统一、简单的表示。因此，基于该核心开发分析会容易得多，因为这样既无需处理不必要的源代码层级细节，也能使用相同的 IR 构造处理不同的 Verilog 构造。(2) Qihe IR

通过属性系统扩展这一核心，使前端能够补充 IR 生成过程中被消解的原始语法模式的额外信息。专用分析可以利用这些属性，从而提高精确率或改善性能。

例如，Verilog 的始终 块、端口连接和连续赋值都会在 Qihe IR 中转换为过程块，使分析能够轻松地以统一方式处理它们。同时，属性系统会保留它们在源代码中的原始形式信息。以图 `sqr` 到 `i.$t1` 的端口连接为7 中的示例。该连接首先经语法糖消解转换为连续赋值，然后再转换为过程块 `[2]:Proc`。该块具有属性 `portConn="to:i"`，该属性由前端在语法糖消解期间添加，表示这是一个指向实例`i`的端口连接。它还具有属性 `origin="assign"`，该属性在转换为过程块时添加，表示它源自连续赋值（而该连续赋值本身由端口连接经语法糖消解而来）。这些属性可以指导某些分析的优化。例如，图 2中的快速到达定义分析利用了这一信息，在表 1.7× 1列出的 13 个程序中实现了平均加速（对于最大的 XS 程序，即 180 万行代码，运行时间从 44.6 秒降至 25.6 秒）。

4.4 不完整程序的自动签名推断

由于加密或无法访问其实现的 IP（知识产权）核被广泛使用，硬件设计中经常会出现不完整程序。Lint 工具可以忽略缺失的模块，继续进行局部的基于模式的分析；但若要支持精确率和可靠性更高的复杂模块间分析，Qihe 就需要妥善处理缺失部分，这通常需要获取缺失模块的签名。

模块的签名包括其名称、端口类型和方向，以及编译时（展开时）参数等元数据。Qihe 的许多复杂分析都依赖这些信息。例如，污点分析必须跟踪值在模块内或模块间的传播，并确定某个值是否受污点污染。如果模块缺少签名——具体而言，缺少各端口的方向（输入或输出）——污点分析就无法在模块端口传播值，因为它不知道值应流入还是流出模块，除非另行推断该签名。不过，由于我们的前端能够自动推断签名，不仅污点分析，其他涉及模块间数据流的分析也能从中显著受益。另一个例子是位宽分析。它需要类型和元数据信息，以检查通常会导致 Verilog 漏洞的隐式类型转换。如果同一缺失模块有多个实例化，且某个端口在不同实例化中连接到不同类型的连线（例如一个为 8 位，另一个为 16 位），同时又没有元数据可以解释这种不匹配，分析就应针对隐式类型转换发出警告。

因此，Qihe 的前端旨在根据现有上下文自动正确推断缺失的模块签名，从而免去分析开发者在实现自己的分析时反复执行这一操作。下面以一个缺少模块定义的程序为例，说明 Qihe 如何处理。

以图 7 中的未定义模块 doubler 为例：处理模块实例化“`doubler i(sqr, out)`”时，前端检测到实例化模块`doubler` 的定义缺失。为继续处理，Qihe 推断模块 `doubler`的签名，其中包括其端口的类型和方向（该模块似乎没有元数据）。（1）首先，确定端口数量。在此示例中，`doubler` 有两个端口，分别命名为 `$t1` 和 `$t2`，对应于连接到它的两条连线 `sqr` 和 `out`。（2）接下来，Qihe 根据已连接连线的类型推断端口类型。例如，由于 `out`（连接到 `$t2`）是一位类型，Qihe 假设`$t2` 也具有一位类型。此处，`doubler` 只实例化了一次；然而，如果一个模块被多次实例化，对于同一个端口，我们需要检查在

不同实例化中与其连接的所有连线。如果这些连线类型相同，则采用该类型作为结果；否则，合并这些类型以确定推断结果。（3）最后，通过检查已连接连线的定义来推断端口方向。如果连接到端口的连线由其他赋值定义，那么该端口不太可能是输出端口，因为这会导致冲突。因此，将其视为输入端口。否则，将其视为输出端口。在此示例中，$t1 被推断为输入端口，因为与其连接的连线 sqr 已在第 9–10 行定义，这表明值通过 $t1 从 sqr 流入实例化模块。相比之下，$t2 被推断为输出端口，因为与其连接的连线 out 是外围模块的输出（第 4 行），且没有其他驱动源，这表明值由实例化模块通过 $t2 提供。

请注意，由于生成的 `doubler` 模块无法与有意设计的空实现双端口模块区分开来，Qihe 会使用属性系统将其标记为 外部，如图 7所示。这有助于后续分析针对其未知实现作出可靠假设。

5 分析管理

尽管 Soot [73] 和 WALA [24] 等面向软件的传统静态分析框架已得到广泛应用，但它们主要侧重于实现独立分析，而非提供管理相互依赖分析的机制。这些框架对以下用例的支持有限：(1) 轻松集成新的分析；(2) 自动执行具有复杂依赖关系的分析。

我们认为，有效管理分析，尤其是集成和执行相互依赖的分析，对于使 Qihe 成为实用框架至关重要。若缺少此类能力，管理图 2所示的 40 多项相互依赖分析将是一项艰巨的任务。近期的软件静态分析框架 Tai-e [69] 也强调了开发者友好型静态分析框架的重要性，并引入了基于配置文件的分析管理机制，获得了用户的积极反馈。

受 Tai-e 启发，我们在 Qihe 中引入了一套系统化的分析管理机制，但采用了不同的设计。与 Tai-e 要求开发人员编写用于分析管理的配置文件不同，我们提供了 Java 注解来实现相同的功能（因为 Qihe 是用 Java 实现的）。这种方式利用 Java 注解自动完成分析发现等任务，从而简化了新分析的开发；如果采用配置文件，这些任务通常需要显式注册。此外，借助成熟的 Java 类型系统及其相关工具（如代码补全和编译时验证），这种方式还能帮助开发人员避免粗心错误，而这些功能在使用配置文件时通常无法获得。

本节将从分析集成和执行两方面，介绍 Qihe 分析管理器的关键设计。除这些核心设施外，Qihe 还提供了其他管理功能，例如分析选项和诊断处理。对于希望了解更多细节的读者，我们建议查阅项目文档。

5.1 分析集成

为了简化新分析的集成，Qihe 提供了基于注解的设施，以满足分析集成中的以下通用需求：

- 注册新的分析，以便进行管理。
- 访问程序抽象以实现分析逻辑。
- 导出结果，供其他分析使用。
- 如有需要，从现有分析中获取结果。

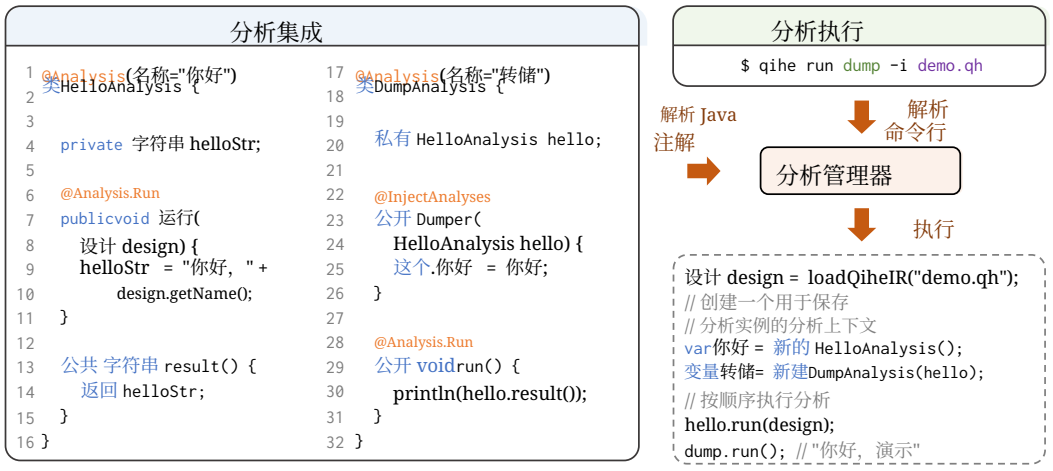


图 9. 使用分析管理器集成和执行分析的流程示意图。Qihe 用户只需开发分析代码（如“分析集成”部分所示），并指定要运行的分析（如“分析执行”部分所示）。管理器会自动执行这些分析，执行方式与图右下方所示代码类似。为简洁起见，这里省略了图 1中所示的管理器的两个核心组件——分析上下文和分析执行器，其逻辑由语义等效的代码表示。

我们将参考图9 来说明如何满足这些需求。图 9中所示的代码使用 Java 编写，因为 Qihe 是用 Java 实现的。

为满足第一个需求（即分析注册），Qihe 中的每项分析都必须是一个带有 `@Analysis(name="str")` 注解的 Java 类（参见图 9中的第 1 行和第 17 行）。此注解使框架能够自动发现并注册分析。注解中的 `name` 属性是该分析的简短标识符，用户可通过命令行指定要运行的分析。例如，图 9 定义了两个名为 `hello` 和 `dump`的分析，并使用命令“`qihe run dump -i demo.qh`”对输入 Qihe 中间表示demo.qh执行分析 dump；其中，`-i` 标志用于指定输入。该中间表示文件是使用命令“qihe compile demo.v -o demo.qh”从 Verilog 源文件 demo.v 编译得到的 Verilog 设计（如图 7所示）。值得注意的是，在实际项目中，Verilog 设计通常分布在多个文件中。“`qihe compile`”命令可以将这些文件一并编译为单个 Qihe 中间表示，随后即可使用“`qihe run`”进行分析。

要访问程序抽象（即 Qihe IR）并实现分析逻辑，开发人员必须为每项分析定义一个 Run 方法（见第 7 行和第 29 行）。该方法的第一个参数是 Qihe 提供的 Design 对象，它表示命令行中由 -i 标志指定的 Qihe IR（例如本例中的 demo.qh）。如果某项分析不需要直接访问 IR，只需使用其他分析的结果——例如，仅依赖其他分析提供的控制流图的数据流分析——则可以省略此参数（如 dump 分析所示）。Run 方法的主体包含分析逻辑。该方法必须使用 @Analysis.Run 注解，以便 Qihe 能在分析执行期间自动识别并调用它。

要导出分析结果，开发人员只需提供返回分析结果的公共方法。例如，在图 9 的第 13 行中，`hello` 分析定义了一个公共方法`result`，该方法返回其`Run` 方法中计算出的“Hello”字符串。

若要访问已有分析的结果，新分析必须通过一个以其他分析实例为参数的构造函数声明其依赖关系（参见第 22–26 行）。此构造函数应使用 @InjectAnalyses 注解，以便 Qihe 识别并调用它来注入依赖项。如果没有依赖项，则可以省略构造函数（如 hello 分析所示）。新分析可以将这些依赖分析对象存储为字段，并在其 Run 方法中使用它们的结果。例如，dump 分析会从存储在字段中的 hello 分析获取“Hello”字符串，并将其打印到标准输出。

5.2 分析执行

执行分析的一种常见方法是手动创建分析类的实例，并按正确的依赖顺序调用它们的 `Run` 方法。例如，要运行`dump`分析（如图 9右下角所示），可以加载 Qihe IR，创建 `hello`和`dump`的实例，然后依次调用它们的 `Run` 方法，并在需要时传入 `Design` 对象。

然而，随着分析数量的增加，手动管理这一过程会变得繁琐且难以维护。为解决这一问题，Qihe 将整个过程自动化。用户只需执行如下命令：

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

随后，框架会执行以下步骤，其效果等同于使用 Java 的反射机制生成并执行图 9右下角所示的代码：

(1) 将待分析的 Qihe IR（即 <ir>）加载到 Design 对象中。(2) 扫描 Qihe 代码库，查找带有 @Analysis 注解的类，以发现所有可用的分析，并解析其注解以记录元数据。(3) 解析这些分析的依赖关系，实例化它们，并通过带有 @InjectAnalyses 注解的构造函数按顺序注入依赖项。如果检测到循环依赖，则报告错误。(4) 按正确的依赖顺序调用 Run 方法，执行请求的 <analysis> 及其依赖项，并根据需要传入 Design 对象。每个 Run 方法只调用一次，以避免重复计算。

Qihe 负责管理这些任务后，开发人员便可以专注于利用现有分析来实现自己的分析逻辑，而将运行时管理的复杂性留给框架处理。

6 评估

Qihe 旨在支持并促进硬件领域中各类静态分析客户端的开发。为此，Qihe 采用滚雪球过程：基础分析协同组合，逐步构建出日益复杂的分析，最终实现多种特定应用分析。据我们所知，先前的工作尚未探索这些基础分析，也未研究它们以协同、类似滚雪球的方式进行组合。

本节首先通过案例研究（第 6.1 节）说明滚雪球过程如何揭示该过程所依赖的内部统计信息，从而检测出关键的缺失复位漏洞。随后，我们评估 Qihe 在支持不同领域中实用的特定应用分析方面的效用：硬件缺陷检测（第6.2 节）、硬件安全分析（第 6.3 节）以及硬件程序理解（第6.4 节）。

6.1 Qihe 中的滚雪球过程：案例研究

Qihe 分析能力的增长类似于滚雪球过程：现有分析相互协作，逐步促成更复杂的分析。最终，这一过程使我们能够检测到以往借助现有静态分析无法发现的漏洞 [16, 32, 57, 66] 用于

表1. 在13个真实世界 Verilog 程序中，缺失复位及其代表性依赖关系的结果，按依赖关系进行拓扑排序。列表示程序，行表示分析（Meta 行除外，该行列出这些程序的代码行数和中间表示），单元格显示汇总统计数据。对于提供图形化程序抽象的控制流图和流不敏感定义链，收集完整的真实值不可行，因此省略。对于时钟、寄存器和复位，结果通过与真实值（#真相）比较精确率和召回率来衡量；精确率表示已识别结果中真实值所占的比例，召回率则表示真实值中已被识别的比例。对于缺失复位，由于不存在真实值，我们以“#已报告（#已确认）”的格式展示开发人员确认的报告漏洞数量。如果没有漏洞得到确认，我们不将其计为真实漏洞，也省略“（#已确认）”，因为我们的目标是突出显示开发人员实际关注的已报告漏洞数量。

		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	#节点	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
流不敏感定义链	#节点	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
时钟	#真相	2,651	29	36	13	10	14	39	13	3	9	23	16	1
	精确率	100%	100%	100%	87%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	召回率	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
regs	#真相	63,678	367	197	152	97	153	604	50	156	52	66	263	32
	精确率	100%	99%	86%	100%	100%	100%	100%	100%	99%	100%	100%	100%	100%
	召回率	100%	100%	97%	100%	100%	99%	99%	100%	97%	100%	94%	100%	100%
复位	#真相	2,218	37	36	14	8	14	38	7	4	10	19	6	1
	精确率	98%	97%	100%	92%	80%	100%	100%	100%	100%	50%	100%	100%	100%
	召回率	88%	81%	100%	79%	100%	100%	100%	86%	75%	100%	100%	83%	100%
缺失复位	#已报告	50	9	0	0	0	0	20（2）	10	5	3（2）	1（1）	12	0

Verilog漏洞检测，如表 2所总结。第 3.3 节讨论了这一滚雪球过程的一个示例，其中关键分析相互依赖、协同工作，以支持 缺失复位分析。本节通过考察与 缺失复位相关的实验结果，深入探讨这一过程。具体而言，我们将 缺失复位 分析及其依赖关系应用于真实世界程序，并重点关注实验结果的三个关键方面：（1）每项分析都能提升下游效果（即分析可靠性和精确率）；（2）我们的分析套件显著降低了开发成本；（3）基础分析的效率主导着整体系统效率。

程序集。 为了提供与缺失复位分析相关的实用统计数据，我们选取了十三个来自不同领域的真实硬件程序，涵盖 CPU设计、加密、AXI协议和计算机视觉等。这些程序来自颇受欢迎的开源GitHub项目 [28, 39, 53, 60, 71, 72, 77, 78, 80, 81],，这些项目平均约有 1.9K 个星标；此外，我们还选取了硬件漏洞基准 [17, 49], 以提供更多与漏洞相关的统计数据。表1 上半部分为简洁起见列出了这些程序的简称，以及其元数据，包括编译后的代码行数（LoC）和 IR 行数（LoIR）。所有这些程序都可在我们的项目仓库中获取。为便于未来研究在我们的基础设施（例如 Verilog前端和中间表示）之上开发新的分析客户端，并在这些热门开源项目上对其进行评估，我们的仓库已将这些项目预处理为可立即分析的形式，例如预编译IR或单个 Verilog 文件。此预处理可免去开发人员重复

大量编译工作的麻烦，因为 Verilog 项目缺少标准构建系统，而且通常需要仔细理解各个项目的结构和构建流程，才能生成 IR。

6.1.1 代表性分析及其结果。为展示图 5 中的分析如何提升滚雪球过程中下游工作的有效性，表 1 按拓扑顺序列出了代表性分析及其结果。表中未列出某些分析，例如层次结构 （用于 Verilog 中的模块层次结构解析）和 可达保护条件 （用于提取同步信息），并不意味着它们不重要。之所以未列出，是因为它们的结果不适合在表格中简洁呈现，或者收集其真实值需要投入大量人工。尽管如此，表 1 中所列分析的性能指标仍体现了它们的有效性，因为这些分析依赖于它们。

控制流图（控制流图）和流不敏感定义链（流不敏感定义-使用链）首先出现在表中，凸显了它们在支持后续关键分析方面的基础作用。例如，稍后进行的时钟分析会利用流不敏感定义链中的边来传播时钟信号信息。起初，考虑到流不敏感定义链中的假阳性边可能会产生不可行的传播路径，从而降低时钟分析中时钟推断的精确率，我们尝试采用流敏感实现（流敏感定义链）。然而，我们发现，实践中流不敏感版本已能提供足够的精确率，并显著提升速度，因此我们更倾向于采用这种方法。这一案例也表明，Qihe 实际上提供了多种基础分析，使开发人员能够探索不同方案，从而在实践中做出更合适的权衡。

表中的接下来三项分析（时钟、复位和寄存器）用于推断变量的物理用途。它们在精确率和召回率方面的损失（相关定义见表1的说明）可能会显著降低依赖分析的有效性，以及这些分析在现实场景中的实用价值，正如第3.3节所述。因此，我们投入了大量心力，以确保精确率和召回率良好，结果也符合预期，如表1所示。由于这些分析旨在帮助理解硬件，第6.4节详细说明了我们如何获得真实值结果，以及这些分析如何实现较高的精确率和召回率。

最后，缺失复位能够有效识别真实世界程序中的缺失复位错误，这些错误可能导致非预期的硬件行为；其识别能力得益于前述分析。如表1所示，粗体值表示开发人员确认的五个错误。值得注意的是，报告的缺失复位错误数量仍在可控范围内。例如，即使对于一个有百万行代码的 XS 程序，也只会生成 50 条错误报告，人工检查完全可以接受。

在硬件缺陷检测中，控制错误报告的数量至关重要，因为报告过多会让开发人员应接不暇，并掩盖真正的问题。我们通过两种方式解决这一问题：(1) 检查真实世界程序中的假阳性，并在必要时针对各个客户端进行改进，从而减少虚假的错误报告，同时不会显著牺牲召回率。例如，在缺失复位分析中，我们在依赖图中检测周期时跳过过大的周期。这类周期通常由假阳性边导致；构建可靠依赖图时不可避免地会引入这些边，而且它们很少对应真正的漏洞。(2) 我们采用一种适用于硬件分析的通用策略：Verilog 程序通常使用元编程来描述重复的电路模式。展开会扩展元编程结构；展开后，同一个问题（尤其是假阳性）可能会被多次报告。为此，我们合并近似重复的报告，并仅输出一条代表性报告。虽然这种方法可能会合并真实漏洞的多个实例，但修复代表性报告所指出的根本原因，通常就能消除所有相关问题。因此，我们的检测客户端在真实世界程序中报告的漏洞数量（即表2中的漏洞）都处于可控范围内，情况与缺失复位分析类似，从而使用户更容易确认其中真正的漏洞。

6.1.2 开发工作量。在滚雪球过程中，借助 Qihe 的基础分析，开发新的客户端会容易得多。图10 左侧纵轴展示了缺失复位所依赖的所有分析，并按依赖关系以拓扑顺序自下而上排列。灰色条形表示实现每项分析所需的代码行数（LoC）。如图所示，在 Qihe 中将缺失复位实现为客户端所需的开发工作量（540 LoC），远少于不使用 Qihe 基础分析、从头实现的工作量（9,700 LoC；通过汇总实现缺失复位所需的所有分析的代码行数计算得出）。平均而言，基于我们的分析套件构建的每个错误检测客户端约需 350 LoC，而从头构建则需 5,900 LoC。开发工作量的大幅减少凸显了 Qihe 在促进开发新的特定应用分析方面的能力。

6.1.3 分析时间。 基础分析着眼于硬件特性，满足基本的分析需求，同时也承担了支持下游分析所需的大部分分析工作量。图 10展示了在 XS（表1中的一款大规模 RISC-V 片上系统）上运行 missing-reset 时各项分析累计的运行时间开销。该系统包含 180 万代码行数；测量使用配备 Intel i9-13905H CPU 和 16 GB 内存的机器完成。在此情况下，基础分析（从 层次结构 到 可达保护条件）占总耗时的 93%（68.2秒，共 73.1秒）。同样，在表1中的其他十二个程序上，它们平均占总耗时的 84%。

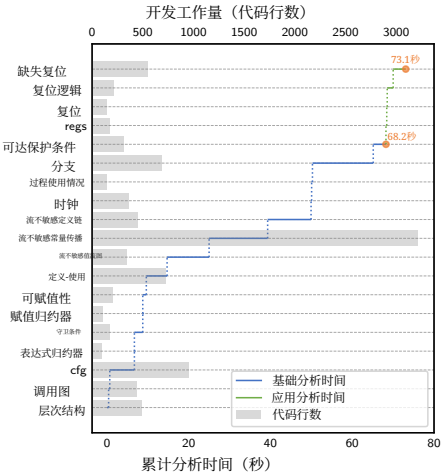


图 10. 大规模 RISC-V 片上系统上的缺失复位分析（表 1中的 XS）。

这凸显了基础分析在 Qihe 中的主导作用，也进一步表明，优化基础分析是缩短整体分析时间的重要途径。我们加速了这些分析时利用了 Verilog 的特有属性——例如组合逻辑的无环性、连续赋值不含分支的特性，以及守卫语句的同步性——并结合传统的软件优化技术。在优化之前，对 XS 执行某些分析使用 128 GB 内存时最长需要一小时。优化之后，整个分析套件（即全部 20 项特定应用分析，以及图2中的 22 项基础分析）在 XS 上使用 40GB 内存约 5 分钟即可完成。对于表1中的其他十二个程序（共 82K 行），整个套件仅需 6 秒即可完成。请注意，在一个进程中同时运行多个分析客户端，可以共享分析结果并减少重新计算，但对于这个百万行的 XS 程序，每增加一项分析，峰值内存使用量就会增加约 1 GB。因此，在 XS 上运行缺失复位（涉及 19 项分析）需要约 1 分钟和 16 GB 内存，而执行整个套件（涉及 42 项分析）则使用 40 GB 内存，约 5 分钟即可完成。

6.2 硬件缺陷检测

即使在工业环境中，检测硬件漏洞仍是一项艰巨的挑战。根据西门子的全球工业研究，在 2024 [26], 仅有14%的IC/ASIC项目实现了首次流片成功，只有13%的FPGA项目报告称没有漏洞逃逸到生产环节。与软件不同，软件漏洞可以通过互联网以较低成本远程修复，而生产阶段后期的硬件漏洞可能导致修复费用呈指数级增长 [6, 8]。

静态分析为设计阶段的早期缺陷检测提供了一种轻量级方法，因此对硬件而言尤其经济高效。为了展示 Qihe 作为通用型

表 2. Qihe 在实际项目中识别出的硬件漏洞。所列漏洞均无法通过现有的、用于 Verilog漏洞检测 的代码检查器式静态分析检测 [16, 32, 57, 66]。新发现的漏洞以**粗体**显示；这些漏洞此前无人知晓，且已得到开发人员确认。

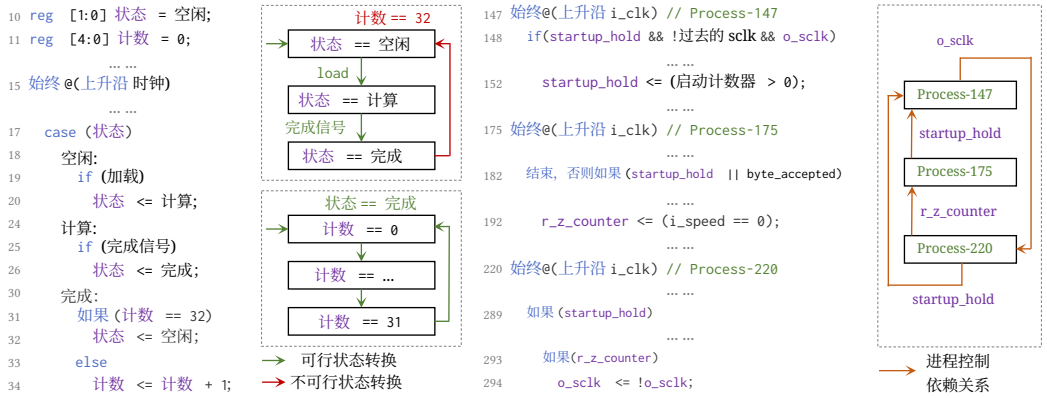
类别	ID	描述	项目	新颖程度
缺少复位	B01 B02	寄存器 drop_frame 缺少适当的复位逻辑。 寄存器 wr_ptr_cur 缺少适当的复位逻辑。	Verilog-AXIS [25]	已知来源： [48]。
	B03	寄存器 correct 缺少适当的复位逻辑。	PULPissimo [62]	已从 [50] 中知悉。
	B04 **B05**	TIMER 寄存器缺少适当的复位逻辑。 TIMEUS 寄存器缺少适当的复位逻辑。	DarkRISCV [60]	新发现。
不可达状态	B06	使用的状态 dma_ctrl_reg == CTRL_ABORT 不可达。	OpenPiton [9]	已知于 [18]。
	B07 **B08** **B09**	使用的状态计数 == 2不可达。 使用的状态计数 == 8不可达。 使用的状态计数 == 32不可达。	32-Verilog-Projects [54]	新发现。
死锁	B10	o_s时钟因死锁而卡住。	SDSPI [27]	已知于 [48]。
	B11	rd_addr 因死锁而卡住。	HDL Lib [42]	
未驱动信号	**B12**	使用来自未驱动信号 clock_div 的值。	基础 Verilog [56]	新发现。
	B13	使用未驱动信号 r_cv 的值。	ZipCPU [28]	
	B14	使用未驱动信号 v 的值。	32-Verilog-Projects [54]	
未加载信号	B15	模块 aes_ˈcc 存在未加载的信号 rst	PULPissimo [62]	已知信息来自 [50]。
子模块误用	**B16**	由于端口顺序错误而误用子模块 invert。	32-Verilog-Projects [54]	新发现。
错误截断	B17	rx_mmio_channel 截断错误导致数据丢失。	Sha512 [14]	已知来自 [48]。
无效使用	B18	使用了信号 axi_adrv9001_rx_channel 中的无效值。	HDL Lib [42]	

框架在支持硬件缺陷检测客户端方面的能力（如第 3 节所述），我们开发了一组 Verilog 分析，其背后的见解源自对最新硬件错误调查 [48, 49], 这一知名年度硬件错误检测黑客马拉松 [17–19],以及与硬件行业从业者的直接交流 [3, 4]。

如表 2所示，初步实验结果非常令人鼓舞。我们的分析成功地在一系列不同的真实世界硬件项目中发现了 18 个、分属 8 个类别的漏洞，例如复位缺失、不可达状态和死锁。这些项目包括片上系统（SoC）、通信协议和加密模块。正如我们的尝试所证实的，这些漏洞均无法被现有的 Verilog 代码检查工具检测到。 [16, 32, 57, 66], 这些工具是硬件缺陷检测领域的主流静态分析工具。它们基于语法的方法以及缺乏基础语义分析的局限，使其无法识别那些需要深入硬件语义推理才能发现的漏洞。值得注意的是，我们的分析从广受欢迎的开源项目中发现了 18 个漏洞中的 9 个新漏洞 [28, 54, 56, 60]（这些项目的 GitHub 星标平均超过 1.5K）。此后，相关开发人员均已确认这些新漏洞。这些结果不仅凸显了我们分析方法的有效性，也彰显了高级静态分析在实用硬件缺陷检测中的潜力。

为了进一步说明我们的分析如何识别现实世界硬件错误中的其他类别，超越第 3.2 节中已经讨论的缺失复位错误，我们再介绍两个额外的案例研究——一个不可达状态错误（B09）和一个硬件死锁错误（B10）——作为面向实际硬件错误检测的高级 Verilog 静态分析的补充性代表示例。

6.2.1 不可达状态错误案例研究。有限状态机（FSM）设计是 Verilog 工程师的一项常见工作 [51]。不可达状态是指 FSM 无法从任何初始状态到达的状态。这类状态通常意味着存在功能问题，例如意外的不可行状态转换 [61]。

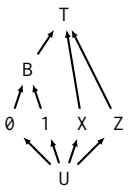


(a) 一个真实的不可达状态错误及其有限状态机（FSM）图。 （b）真实硬件死锁错误及其进程依赖图。

图 11. 真实世界中的错误案例研究：表 2中的 B09（a）和 B10（b）。行号对应原始项目源代码中的行号。

图 11a 展示了一个来自某开源 Verilog 项目的 CRC-32 串行计数器中的真实世界不可达状态错误 [54], ，其中 idle、compute 和 finish 是状态的常量枚举值。右侧的图示将左侧 Verilog 代码定义的两个有限状态机可视化：节点表示状态（分别对应 state 和 count 的值），边表示转换，边上的表达式表示转换条件。由于计数有限状态机（下方那个）中的所有转换共享同一个条件，为避免重复，我们只展示一次。该错误的产生过程如下。图 11a 的第 30–34 行描述了状态有限状态机（上方那个）与计数有限状态机之间的交互：当 state 为 finish 时，count 开始递增；当 count == 32 时，state 应返回 idle。然而，开发者没有注意到 count 被声明为一个 5 位向量（第 11 行），达到 31 后会溢出并回绕至 0。这是在 Verilog 编程中容易忽略的情况。因此，在计数有限状态机中，count == 32 所对应的状态不可达，破坏了两个有限状态机之间预期的交互，并导致状态有限状态机出现从 state == finish 到 state ==idle 的不可行转换（以红色突出显示）。这一缺陷十分严重，最终会使系统陷入 finish 状态，无法恢复正常运行。

为了检测此类不可达状态，我们的分析会执行针对 *Verilog* 的位级全程序常量传播，以位级精度确定每个变量的所有可能值。如第3.3节所述，这种精确率在 Verilog 中至关重要，因为程序经常操作单个位，精确率的损失可能导致分析漏掉原本能够检测到的漏洞，例如本案例研究中的漏洞。对于每个变量，我们的常量传播



使用专门设计的抽象位向量对其具体位向量值进行过度近似，其中每一位都取自右侧所示的格：U 表示未定义位，0/1/X/Z 表示固定的 Verilog 位值 0/1/X/Z，B（both 的缩写）表示该位可以是 0 或 1，而 T 表示包含所有可能位值的格顶元素。

在本案例研究中，我们主要关注0/1/B 部分；例如，对于一个三位信号 x，x = B01 意味着 x 只能取两个可能的值：001 或101。在图11a中，转移条件count == 32 （第 31 行）在我们的分析中被解释为将0BBBBB（将count按 Verilog规范进行零扩展至 6 位（隐式类型转换 [1]））与100000进行比较，该值代表固定值 32。此相等关系不可能成立，因为最高有效位

永远不会匹配（$0 \neq 1$）；因此，`count == 32`所在的状态（图11a第 31–32 行使用）不可达。相比之下，如果直接使用传统的非位级常量传播，左侧会被视为非常量值（记为NAC），因此求值NAC == 32 会得出“可能”（可达）而非“不可能”（不可达）的结果，导致无法发现这个不可达状态错误。这个例子表明，直接将类似软件的分析应用于硬件往往效果不佳，而 Verilog 专用静态分析定制（例如保持位级精度）至关重要。更多针对 Verilog 专用静态分析的设计和算法可在我们的开源框架及文档中找到。

6.2.2 硬件死锁错误案例研究。根据 Verilog 语言规范 [1], ，一个正在运行的 Verilog 程序由多个并发进程组成，每个进程通常对应一个运行时 always 块实例。当并发 Verilog 进程之间的循环依赖导致程序无限期停滞时，就会发生硬件死锁 [48], ；其物理表现为信号始终无法按预期更新。

图 11b 展示了 SDSPI 项目的 llsdspi 模块中的一个真实死锁错误 [27],，该模块实现了一个 SD 卡控制器。该控制器应生成名为 o_sclk 的驱动时钟信号。然而，由于三个进程之间存在循环依赖，o_sclk 未能更新：Process-147、Process-175 和 Process-220。其中，Process-i 表示与从第 i 行开始的 always 块相对应的运行时 Verilog 进程。具体而言，Process-220 中 o_sclk（第 294 行）的更新依赖于 Process-175 中 r_z_counter（第 192 行）的更新；后者又依赖于 Process-147 中 startup_hold（第 152 行）的更新，而后者又通过第 148 行依赖于 Process-220 中 o_sclk（第 294 行）的更新。因此，这三个进程都无限期停滞，导致 SD 卡控制器无法生成有效的驱动时钟 o_sclk。没有这个时钟，SD 卡将完全无法工作，也无法执行任何数据访问操作。

为检测此类硬件死锁错误，我们的分析首先构建一个进程依赖图，用于捕捉一个 Verilog 进程中的值更新是否依赖于另一个进程中的值更新，如图11b所示。在该图中，从Process-i 指向Process-j ($i \neq j$)且标记为 x 的边表示，Process-i 通过共享信号 x依赖于Process-j；也就是说， x在Process-i 的分支条件中使用，并由Process-j中的赋值定义。为了识别并发 Verilog 进程之间的循环依赖，我们的分析会检测该图中的所有周期。对于图11b中的示例，分析报告了以下进程之间存在循环进程依赖：Process-147、Process-175和Process-220，从而精准定位导致系统停滞的死锁。

6.3 硬件安全性分析

随着集成电路（IC）在系统中的广泛应用，硬件安全变得愈发重要；这类系统尤其容易受到敏感信息泄露 [35] 和恶意硬件木马 [36]等安全威胁的影响。为进一步展示 Qihe 作为通用框架在支持开发各种特定应用分析方面的实用性，我们在 Qihe 之上开发了两种安全分析：污点分析和 x-prop。

污点分析。 Qihe 的污点分析旨在解决信息流安全 问题。在此情境下，用户指定污点源 和汇 ，要求信息不得从源流向汇，因为此类信息流可能意味着存在未授权访问或机密信息泄露等安全漏洞。在 Verilog 中，污点源可以是存有机密数据的寄存器，而污点汇可以是易受攻击者利用的网络或模块端口。

与软件中的信息流分析相比，由于硬件特定语义（例如位级值传播和双向端口连接），跟踪硬件中的信息流会引入额外的复杂性。尽管如此，Qihe 的基础设施能够有效地

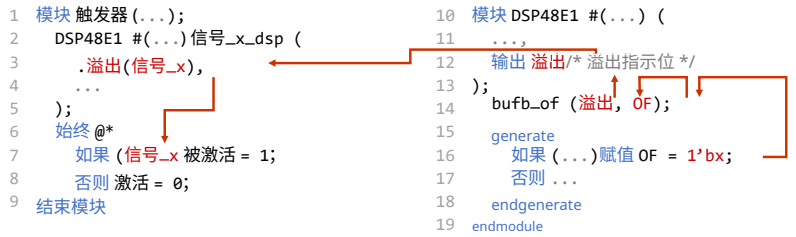


图 12. 利用 [41] X 传播的硬件木马简化摘录。值 1'bx沿着各个逻辑门和端口连接传播，如红色有向边所示，最终到达一个 if语句，该语句构成利用点。

处理并抽象这些硬件特性，从而成功开发出一种基于经典显式信息流建模的静态信息流分析。

为评估 Qihe 的污点分析（即 污点），我们采用了 Trust-Hub [59, 64], 这一评估硬件安全验证技术有效性的事实标准基准。从其 Verilog设计集合中，我们收集了全部 25 个与信息流安全相关的设计，并排除了 2 个 VHDL 设计（与 Qihe 专用的 Verilog 前端不兼容）以及 4 个未明确指定污点源/汇的设计。最终，我们选取了 19 个 Verilog 设计进行评估。这些设计主要聚焦于密码硬件实现，反映了信息流安全需要得到保障的现实应用场景。

我们的分析成功识别出 19 个信息泄漏实例中的 15 个，其中密钥或内部状态寄存器的内容通过可观察输出端口泄漏。经人工检查，污点分析未识别出的 4 个案例均为控制依赖导致的隐式信息流。目前，污点分析尚未处理隐式隐蔽信道（例如控制依赖、延迟和功耗），但我们认为，这为未来基于 Qihe 开发更强大的分析提供了一个研究方向。

*X 传播分析。*Qihe 的 x-prop 分析旨在识别 X 值（未知状态）的传播，这些值往往意味着存在严重的设计缺陷。这些缺陷可能导致仿真失败，甚至更严重的是，可能被硬件木马利用来掩盖恶意行为。

检测 X 传播问题并非易事，因为这需要谨慎处理 X 特有的语义，而这种语义只存在于 Verilog 等硬件描述语言中；当它与复位、内存和位寻址等硬件行为交织在一起时，问题就更加复杂。面对这些挑战，与其开发一个需要顾及所有细节的独立分析，不如让 Qihe 的 x-prop 利用现有的基础分析，对与 X 相关的语义进行建模，从而分而治之，处理复杂的硬件行为。这种方法使实现简洁明了，代码不超过 300 行。在我们的实现中，Qihe 利用复位和缺失复位来理解硬件复位行为；通过流不敏感常量传播中的常量传播，推断变量可能的取值范围，以检测越界访问；并利用 bit-refs 捕获 Verilog 的位级语义。

图 12 展示了一个由 Qihe 通过其 [41] x-prop 分析识别出的硬件木马。为便于理解，我们对该示例进行了简化。首先解释该木马如何利用 `x` 值隐藏自身，然后说明`x-prop` 如何检测它。

为隐藏木马，攻击者利用仿真与综合对未知值的解释差异 `x`，使木马在仿真期间无法被检测，却能在制造出的电路中被触发。在图12的第 7–8 行中，攻击者采用专门的电路逻辑（详见下一段），以确保条件 信号_x 在`if`

语句始终求值为X。仿真期间，根据语言标准，x 值被视为假，因此 [1], 所以 激活 保持为0。由于 激活 控制木马载荷的执行——该载荷可能会执行泄露密钥等操作（为简洁起见，示例中省略了相关内容）——因此载荷在仿真中绝不会触发。然而，综合后，未知x值会在物理电路中解析为0 或 1 。因此，条件 信号_x 可能求值为1，从而允许 激活 被设为1 并启用木马载荷。

为了确保 if 语句的条件在仿真期间求值为 X，同时降低被检测到的可能性，攻击者通常会将 X 的来源隐藏在单独的模块中。如图 12 中红色路径所示，攻击者首先配置第三方 DSP 模块（即 DSP48E1），使变量 OF 生成一个 X 值（第 16 行）。随后，这个 X 值通过一个缓冲门（第 14 行）传播至 DSP48E1 模块的输出端口溢出（第 12 行）。接着，X 值通过与实例化的 DSP48E1 输出端口的连接，进一步传播到触发模块中的信号_x 变量（第 2–3 行）。最终，信号_x 将 X 值传递到 if 语句的条件中（第 8 行）。x-prop 成功检测到 DSP48E1 中的变量 OF 在仿真期间可能包含 X 值，并追踪了从 OF 到 if 语句的传播路径。因此，x-prop 将其标记为潜在的 X 传播漏洞，并报告为错误。经过人工分析，我们确认存在一个利用 X 值进行隐蔽的木马。

6.4 硬件程序理解

硬件设计是一个涉及多个阶段的过程，例如功能设计、综合、验证等，最终实现物理实现。使用 Verilog 进行功能设计时，硬件设计人员必须掌握物理实现细节。例如，他们需要确定时钟是否集成到复杂逻辑中以降低延迟，并了解寄存器如何映射到板载资源，从而优化资源利用率。遗憾的是，这些重要信息往往要等到漫长的综合过程结束后才会披露。

为了帮助开发人员尽早更好地理解其 Verilog 设计，Qihe 开发了七种用于硬件程序理解的分析方法，用于预测物理实现细节，每种分析方法都专注于一项特定的理解任务。例如，时钟分析会预测作为物理时钟树一部分的变量，并标记不当用法，例如将时钟用于组合逻辑；而寄存器分析则会推断映射到物理寄存器的变量。

值得注意的是，对于表 1 中包含 180 万行代码的真实程序 XS，Qihe 完成这七种分析只需约一分钟。相比之下，使用 Yosys 对同一代码库进行综合 [79]，（Yosys 是最受欢迎的开源综合器）所需时间超过了我们的一个小时预算。

下面，我们通过两个案例研究来考察 Qihe 的分析效果：时钟 分析和 寄存器 分析。我们重新使用第 6.1 节介绍的 13 个多样化真实世界程序，结果已列于表 1 中。

表中的真实值取自 Yosys 的结果，因为我们的分析旨在预测综合后物理实现。需要注意的是，Yosys 通常不提供用于访问静态分析通常所需结果的 API，因为它并非为此目的而设计。例如，它不提供用于导出电路中通常所需时钟树的 API——而我们的时钟分析支持这一功能。因此，这一限制使真实值的构建过程更加复杂：为了获取真实的时钟树，我们必须修改 Yosys 以导出时钟树叶节点（即直接连接到寄存器的时钟信号），提取与这些叶节点相连的变量，以近似得到时钟树，然后手动验证这些变量作为时钟树信号的有效性。

如表 1所示，这些分析的有效性通过精确率和召回率进行衡量。对于硬件程序理解而言，这些指标至关重要，因为其中任一指标的性能不足都会导致

过多的假阳性或假阴性，从而损害该工具在现实场景中的实用性。因此，我们投入了大量精力设计超越简单模式匹配的算法，以实现较高的精确率和召回率；如表中所示，结果符合我们的预期。

对于时钟，由于我们采用了语义推理方法，该方法在所有评估程序上都实现了出色的精确率和召回率。关键在于，硬件时钟构成一个树结构，其与电路其余部分的边界大多可以根据代码语义推断出来。一旦确定这些边界，就可以从任意树节点开始提取整棵树。

寄存器 分析也实现了较高的精确率和召回率，这得益于其所依赖的时钟分析的有效性及其硬件语义级推理算法。在某些程序中，由于未能识别通过阻塞赋值而非标准的非阻塞赋值更新的寄存器，其召回率略有下降。虽然 Yosys 能够处理这类情况，但阻塞赋值无法表达寄存器“在下一个时钟周期更新值”的语义 [15], ，而且普遍认为使用阻塞赋值进行寄存器更新是不良做法。因此，这种情况下召回率的轻微损失不会引发实际问题。

总而言之，这些用于程序理解任务的快速、准确的分析，能够在硬件开发早期为开发人员提供有效的见解；希望未来能涌现更多此类分析，进一步提高硬件编程效率。

7 相关工作

本文聚焦于硬件静态分析，最相关的研究已在第 1 节讨论。作为首个针对 Verilog 构建通用静态分析框架的尝试，我们将在下文简要阐述这一构想如何受到以往研究的启发，并指出这些研究在实现我们的目标方面存在的局限。

现有的硬件静态分析工具（如代码检查器） [16, 32, 57], 已证明，在生产环境中进行简单的错误检测和代码风格检查十分有效。然而，硬件复杂性和多样性的迅速增长给可靠性、安全性和可维护性带来了新的挑战 [30,40, 47, 74], ，而这些挑战远非简单的语法或基于模式的静态分析所能应对，迫使开发人员重新依赖资源密集型的测试与验证 [12, 23, 38, 70]。

这与软件质量静态分析的发展历程相吻合：研究人员和工程师最初采用测试和验证 [21, 33] ，仅使用简单的静态分析进行基于模式的检查 [34, 37], ；后来则采用复杂的静态分析来进行错误检测和安全性分析，甚至处理百万行代码 [10, 11, 13, 24, 31, 65, 67, 69, 73]。我们相信，复杂静态分析在应对硬件领域新兴挑战方面的巨大潜力仍有待发掘。

这自然引出了一个问题：我们可以基于什么来构建此类复杂的硬件静态分析？一种直观的解决方案是利用已有的、包含前端和中间表示的工具。然而，这些工具都不适合用于开发复杂的硬件静态分析 [16, 32, 43, 57, 58, 63, 68]。这是因为它们并非专为静态分析而设计，因此不可避免地会将其他设计目标置于分析之上，更不用说它们缺少必要的基础设施，例如一套用于支持各种复杂分析客户端的基础分析。因此，我们推出了 Qihe——一个面向 Verilog 的通用静态分析框架，在所有组件的设计中明确优先考虑硬件分析的需求。

8 结论

我们介绍了 Qihe，这是首个面向 Verilog 的通用静态分析框架，旨在支持广泛的特定应用分析。这通过综合基础分析套件、面向分析的前端和中间表示（IR）

以及便于使用的分析管理器的协同配合得以实现。为展示 Qihe 的潜力，我们在其基础设施之上进一步开发了客户端分析，这些分析能够在实际项目中成功检测硬件漏洞和安全漏洞，并支持程序理解任务。这项工作是释放复杂静态分析在硬件领域全部潜力的初步尝试。我们希望，拥有超过 10 万行完全开源代码的 Qihe，能够激励软件与硬件社区的研究人员和工程师开展合作，共同推动未来面向硬件的静态分析发展。

致谢

本项目源于好奇心和探索热情。我们衷心感谢来自海思（华为）的 Xuanlin Li 向我们介绍静态分析在硬件领域的重要性。此外，我们感谢 Yufei Liang、Teng Zhang、Menglong Chen 和 Zhiwei Zhang 提出的宝贵建议，这些建议有助于改进本文。我们还感谢 Zhongyi Cai 和 Sitao Lin 早期使用我们的工具并提供建设性反馈。

参考文献

[1] 2006. IEEE Verilog硬件描述语言标准。IEEE Std 1364-2005（修订 IEEE Std 1364-2001）（ 2006 1 590 10 1109 2006 99495 ），–。doi：。/IEEESTD. . . IEEE SystemVerilog–统一硬件设计、规范与验证语言标准。IEEE Std 1800 2017 1800 2012 2018 1 1315 10 1109 2018 8299595 -（修订 IEEE Std -） （），–。doi：。/IEEESTD. . . 海思。https://www.hisilicon.com/en。[4] 2025。平头哥。https://www.t-head.cn/?lang=en。[5] Baleegh Ahmad、Shailja Thakur、Benjamin Tan、Ramesh Karri 和 Hammond Pearce。2024年。通过提示大语言模型修复硬件安全漏洞代码。IEEE信息取证与安全汇刊 19（ 2024 4043 4057 10 1109 2024 3374558 ），–。doi：。/TIFS. . Ken Albin。. 片上系统漏洞的代价。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 和 Patrick McDaniel。2014年。FlowDroid：针对 Android 应用的精确上下文、流、字段、对象敏感且具备生命周期感知的污点分析。SIGPLAN Not. 49, 6 2014 259 269 10 1145 2666356 2594299 （六月），–。doi：。/ . Francine Bacchini、Robert F. Damiano、Bob Bentley、Kurt Baty、Kevin Normoyle、Makoto Ishii 和 Einat Yogev。2004年。验证：哪些方法有效，哪些无效。发表于第41届设计自动化会议论文集，DAC 2004，美国加利福尼亚州圣迭戈，2004年6月7日至11日，Sharad Malik、Limor Fix 和 Andrew B. Kahng（编）。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 和 David Wentzlaff。2016年。OpenPiton：一个开源众核研究框架。发表于第二十一届国际编程语言与操作系统体系结构支持会议论文集（美国佐治亚州亚特兰大）（ ASPLOS '16）。计算机协会，美国纽约州纽约， 217 232 10 1145 2872362 2872414 –。doi：。/ . Al Bessey、Ken Block、Ben Chelf、Andy Chou、Bryan Fulton、Seth Hallem、Charles Henri-Gros、Asya Kamsky、Scott McPeak 和 Dawson Engler。2010年。数十亿行代码之后：运用静态分析在真实世界中查找错误。Commun. ACM 53, 2（2010年2月），66–75。doi:10.1145/1646353.1646374[11] Bruno Blanchet、Patrick Cousot、Radhia Cousot、Jérome Feret、Laurent Mauborgne、Antoine Miné、David Monniaux 和 Xavier Rival。2003年。面向大型安全关键软件的静态分析器。发表于 ACM SIGPLAN 2003 编程语言设计与实现会议论文集（美国加利福尼亚州圣迭戈）（PLDI '03）。计算机协会，美国纽约州纽约， 196 207 10 1145 781131 781153 –。doi：。/ . Aaron R. Bradley 和 Zohar Manna。. 通过归纳反例的归纳泛化检查安全性。发表于计算机辅助设计形式化方法会议论文集（FMCAD '07）。IEEE计算机学会，美国，173–180。doi:10.1109/FAMCAD.2007.15[13] Martin Bravenboer 和 Yannis Smaragdakis。2009年。对复杂指向分析进行严格的声明式规范。发表于第24届 ACM SIGPLAN面向对象编程系统、语言与应用会议论文集（美国佛罗里达州奥兰多）（OOPSLA '09）。计算机协会，美国纽约州纽约，243–262。doi:10. 1145/1640089.1640108[14] Ciro Luiz Araujo Ceissler。2025年。一个 SHA512加速器。https://github.com/efeslab/hardcloud/tree/e28ca96fdbb67904ef909fb04e026cf6dc724198/samples/sha512.

[15] Qinlin Chen、Nairen Zhang、Jinpeng Wang、Tian Tan、Chang Xu、Xiaoxing Ma 和 Yue Li。2023年。Verilog 的本质：一种可处理且经过测试的 Verilog 操作语义。Proc. ACM Program. Lang. 7, OOPSLA2, 文章 230（十月 2023 30 10 1145 3622805），页码。doi：。/ Chipsalliance. . Verible。[16] 2025 https://chipsalliance.github.io/verible/。[17] HACK@DAC-18 竞赛委员会。2018年。Hack@DAC 2018 有缺陷的片上系统。https://github.com/HACK-EVENT/ hackatdac18.git。[18] HACK@DAC-21 竞赛委员会。2021年。Hack@DAC 2021 有缺陷的片上系统。https://github.com/HACK-EVENT/ hackatdac21.git。[19] HACK@DAC 竞赛委员会。2024年。什么是 HACK@DAC? https://www.dac.com/Conference/HackDAC。[20] CWE 社区。2024年。CWE-1271：保存安全设置的寄存器在复位时未初始化。https: //cwe.mitre.org/data/definitions/1271.html。[21] E. Allen Emerson 和 Edmund M. Clarke。1980年。使用不动点刻画并行程序的正确性属性。发表于《自动机、语言与编程》，Jaco de Bakker 和 Jan van Leeuwen（编）。Springer Berlin Heidelberg，柏林，海德堡，169–181。[22] E. Allen Emerson 和 Joseph Y. Halpern。1986年。“有时”和“并非从不”再探：分支时间与线性时间时序逻辑之比较。J. ACM 33, 1 1986 151 178 10 1145 4904 4999（一月），–。doi：。/ . Hongyu Fan 和 Fei [23] 2024 3 He。. 利用集成电路中的数据通路传播进行硬件模型检测。Trans. Comp.-Aided Des. Integ. Cir. Sys. 43, 7 2024 2215 2228 10 1109 2024 3360022（七月），–。doi：。/TCAD. . [24] 2006 Watson 分析库。. WALA。http://wala.sf.net。[25] Alex Forencich。2025年。Verilog AXI 流组件。https://github.com/alexforencich/verilog-axis。[26] Harry Foster。2024年。IC/ASIC 功能验证趋势报告 - 2024年。https://verificationacademy.com/topics/planning-measurement-and-analysis/2024-siemens-eda-and-wilson-research-group-functional-verification-study/。[27] Dan Gisselquist。2025年。SD 卡控制器。https://github.com/ZipCPU/sdspi。[28] Dan Gisselquist。2025年。ZipCPU：小型轻量级 RISC CPU 软核。https://github.com/ZipCPU/zipcpu/。[29] Steve Golson 和 Leah Clark。2016年。21世纪的语言之争：Verilog 与 VHDL——再探。Synopsys 用户组（SNUG）（2016年）。[30] David Gullasch、Endre Bangerter 和 Stephan Krenn。2011年。缓存游戏——让针对高级加密标准的基于访问的缓存攻击成为现实。发表于 *2011* 490 505 10 1109 2011 22 IEEE安全与隐私研讨会。–。[31] 2007 doi：。/SP. . Ben Hardekopf 和 Calvin Lin。. 蚂蚁与蚱蜢：面向数百万行代码的快速精准指针分析。发表于第28届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国加利福尼亚州圣迭戈）（PLDI '07）。计算机协会，美国纽约州纽约，290–299。doi:10. 1145/1250734.1250767[32] Naoya Hatta。2025年。svlint。https://github.com/dalance/svlint/tree/master。[33] C. A. R. Hoare。1969年。计算机编程的公理基础。Commun. ACM 12, 10（1969年10月），576–580。doi:10.1145/363235.363259[34] David Hovemeyer 和 William Pugh。2004年。查找错误很容易。SIGPLAN Not. 39, 12（2004年12月），92–106。doi:10. 1145/1052883.1052895[35] Wei Hu、Armaiti Ardeshiricham 和 Ryan Kastner。2021年。硬件信息流跟踪。ACM Comput. Surv. 54, 4, 83 2021 39 10 1145 3447867 文章（五月），页码。doi：。/ Wei Hu、Baolei Mao、Jason [36] 2016 Oberg 和 Ryan Kastner。. 利用门级信息流跟踪检测硬件木马。Computer 49, 8 2016 44 52 10 1109 2016 225（），–。doi：。/MC. . Stephen C Johnson。. Lint, [37] 1977 一种 C 程序检查器。贝尔电话实验室，默里希尔。[38] Rahul Kande、Addison Crump、Garrett Persyn、Patrick Jauernig、Ahmad-Reza Sadeghi、Aakash Tyagi 和 Jeyavijayan Rajendran。2022年。TheHuzz：使用黄金参考模型对处理器进行指令模糊测试，以发现可被软件利用的安全漏洞。发表于第31届 USENIX 安全研讨会（USENIX Security 22）。USENIX 协会，美国波士顿，3219–3236。https://www.usenix.org/conference/usenixsecurity22/presentation/kande [39] Olof Kindgren。2025年。SERV——串行 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 和 Yuval Yarom。2020年。幽灵攻击：利用推测执行。Commun. ACM 63, 7 2020 93 101 10 1145 3399742（六月），–。doi：。/ Christian Krieg、Clifford [41] Wolf、Axel Jantsch 和 Tanja Zseby。2017年。切换多路复用器：X 乐观性如何导致恶意硬件（DAC '17）。计算机协会，美国纽约州纽约，文章 7，6 页。doi:10.1145/3061639.3062328[42] Rejeesh Kutty。2025年。HDL 库与项目。https://github.com/analogdevicesinc/hdl。[43] Chris Lattner。2025年。CIRCT：电路 IR 编译器与工具。https://circt.llvm.org/。[44] Haonan Li、Yu Hao、Yizhuo Zhai 和 Zhiyun Qian。2024年。增强实用错误检测的静态分析：一种集成大语言模型的方法。Proc. ACM Program. Lang. 8, OOPSLA1, 文章 111（2024年4月），26页。doi:10.1145/3649828

[45] Xun Li、Mohit Tiwari、Jason K. Oberg、Vineeth Kashyap、Frederic T. Chong、Timothy Sherwood 和 Ben Hardekopf。2011年。Caisson：一种用于安全信息流的硬件描述语言。载于第32届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国加利福尼亚州圣何塞）（PLDI '11）。计算机协会，美国纽约州纽约，109 120 10 1145 1993498 1993512 –。doi: 。/。Yue Li、Tian Tan、Yifei Zhang 和 Jingling Xue。[46] 2016。程序定制：按顺序准则进行切片。载于第30届欧洲面向对象编程会议（ECOOP 2016）（莱布尼茨国际信息学论文集（LIPIcs），第56卷），Shriram Krishnamurthi 和 Benjamin S. Lerner（编）。Schloss Dagstuhl – Leibniz-Zentrum für Informatik，德国 Dagstuhl，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 和 Raoul Strackx。2020年。熔毁漏洞：从用户空间读取内核内存。《Commun. ACM》63, 6 2020 46 56 10 1145 3357033（五月），–。doi: 。/ Jiacheng Ma、Gefei Zuo、Kevin Loughlin、Haoyang Zhang、Andrew Quinn 和 Baris Kasikci。[48] 2022年。可重构硬件新时代中的调试。载于第27届 ACM 编程语言与操作系统体系结构支持国际会议论文集（瑞士洛桑）（ASPLOS '22）。计算机协会，美国纽约州纽约，946 962 10 1145 3503222 3507701 –。doi: 。/。Jiacheng Ma、Gefei Zuo、Kevin Loughlin、Haoyang Zhang、Andrew Quinn 和 Baris Kasikci。[49] 2022年。可重构硬件新时代中的调试（Bugbase工件）。https://github.com/efeslab/hardware-bugbase.git。[50] Xingyu Meng、Shamik Kundu、Arun K. Kanuparthi 和 Kanad Basu。2022年。RTL-ConTest：用于检测安全漏洞的 RTL 符号具体测试。《IEEE集成电路与系统计算机辅助设计汇刊》41，3 2022 466 477 10 1109 2021 3066560（），–。doi: 。/TCAD。 。[51] Cody Miller。2010年。状态机设计技术。https://www.edn.com/state-machine-design-techniques/。[52] MITRE公司。2025年。通用弱点枚举。https://cwe.mitre.org/。[53] Kevin E. Murray。2025年。Verilog to Routing——用于 FPGA研究的开源CAD流程。https://github.com/verilog- to-routing/vtr-verilog-to-routing。[54] Sudhamshu B N。2025年。实现32个 Verilog迷你项目。https://github.com/sudhamshu091/32-Verilog-Mini-Projects。[55] Mayur Naik、Alex Aiken 和 John Whaley。2006年。Java 的有效静态数据竞争检测。《SIGPLAN Not.》41，6（六月 2006 308 319 10 1145 1133255 1134018），–。doi: 。/。Konstantin Pavlov。 。必备 Verilog SystemVerilog 模块。[56] 2025。https://github.com/pConst/basic_verilog。[57] Michael Popoloski。2025年。Slang。https://sv-lang.com/。[58] Md Imtiaz Rashid 和 B. Carrion Schaefer。2024年。VeriPy：一个由 Python 驱动的用于解析 Verilog 硬件描述语言和进行高层行为分析的硬件框架。载于2024年 IEEE 第17届达拉斯电路与系统会议（DCAS）。1–6。doi:10.1109/DCAS61159.2024.10539889[59] Hassan Salmani、Mohammad Tehranipoor 和 Ramesh Karri。2013年。设计漏洞分析与可信基准开发。载于2013年 IEEE 第31届国际计算机设计会议（ICCD）。471–474。doi:10.1109/ ICCD.2013.6657085[60] M. Samsoniuk。2025年。DarkRISCV。https://github.com/darklife/darkriscv。[61] Shaker Sarwary 和 Michael A Beaver。2005年。验证有限状态机的系统方法。https://www.edn.com/a- systematic-approach-to-verifying-fsms/。[62] Pasquale Davide Schiavone、Davide Rossi、Antonio Pullini、Alfio Di Mauro、Francesco Conti 和 Luca Benini。2018年。Quentin：采用22纳米FDX的超低功耗 PULPissimo片上系统。载于2018年 IEEE SOI-3D-亚阈值微电子技术联合会议（S *3* 1 3 10 1109 3 2018 8640145 S）。–。doi: 。/S S。 。Fabian Schuiki、Andreas Kurth、Tobias Grosser 和 Luca Benini。[63] 2020。LLHD：一种面向硬件描述语言的多级中间表示。载于第41届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI 2020）。计算机协会，美国纽约州纽约，258–271。doi:10.1145/3385412.3386024[64] Bicky Shakya、Tony He、Hassan Salmani、Domenic Forte、Swarup Bhunia 和 Mark Tehranipoor。2017年。硬件木马和恶意篡改电路的基准测试。《硬件与系统安全期刊》1（2017），85–102。doi:10.1007/s41635-017-0001-6[65] Qingkai Shi、Xiao Xiao、Rongxin Wu、Jinguo Zhou、Gang Fan 和 Charles Zhang。2018年。Pinpoint：面向百万代码行的快速精准稀疏值流分析。载于第39届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国宾夕法尼亚州费城）（PLDI 2018）。计算机协会，美国纽约州纽约，693–706。doi:10.1145/3192366.3192418[66] Wilson Snyder。2025年。Verilator。https://veripool.org/verilator。[67] Yulei Sui 和 Jingling Xue。2016年。LLVM 中的过程间静态值流分析。载于第25届国际编译器构造会议（西班牙巴塞罗那）（CC '16）。计算机协会，美国纽约州纽约，265–266。doi:10.1145/2892208.2892235

[68] Shinya Takamaeda-Yamazaki。2015年。Pyverilog：基于 Python 的 Verilog 硬件设计处理工具包。载于《应用可重构计算》，Kentaro Sano、Dimitrios Soudris、Michael Hübner 和 Pedro C. Diniz（编）。Springer International Publishing，Cham，451–460。[69] Tian Tan 和 Yue Li。2023年。Tai-e：通过借鉴经典设计优点构建的开发者友好型 Java 静态分析框架。载于第32届 ACM SIGSOFT 国际软件测试与分析研讨会论文集（ISSTA *2023* 1093 1105 10 1145 3597926 3598120 ）。计算机协会，美国纽约州纽约，–。doi: 。/。[70] Timothy Trippel、Kang G. Shin、Alex Chernyakhovsky、Garret Kelly、Dominic Rizzo 和 Matthew Hicks。 。像软件一样模糊测试硬件。载于第31届 USENIX 安全研讨会（USENIX Security 22）。USENIX 协会，美国波士顿，3237–3254。https://www.usenix.org/conference/usenixsecurity22/presentation/trippel [71] ultraembedded。2025年。32 位超标量 RISC-V CPU。https://github.com/ultraembedded/biriscv。[72] ultraembedded。2025年。RISC-V CPU 内核（RV32IM）。https://github.com/ultraembedded/riscv。[73] Raja Vallée-Rai、Phong Co、Etienne Gagnon、Laurie Hendren、Patrick Lam 和 Vijay Sundaresan。2010年。Soot：Java 字节码优化框架。载于 CASCON 前十年高影响力论文（CASCON '10）。IBM Corp.，美国，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 和 Raoul Strackx。2018年。Foreshadow：利用瞬态乱序执行提取英特尔 SGX 王国的密钥。载于第27届 USENIX 安全研讨会论文集（美国马里兰州巴尔的摩）（SEC'18）。USENIX 协会，美国，991–1008。[75] VLSIFacts。2025年。理解 Verilog 中的流水线设计：如何跨时钟周期分阶段传输数据以实现高性能。https://vlsifacts.co
m/understanding-pipeline-design-in-verilog-how-to-stage-data-across-clock-cycles- for-high-performance/。[76] VLSIFacts。2025年。理解数字设计中的复位信号：类型、优缺点与最佳实践。https: //
vlsifacts.com/understanding-reset-signals-in-digital-design-types-pros-cons-and-best-practices/。[77] Xuan Wang。2025年。用于驱动无刷直流电机/永磁同步电机的基于 FPGA 的磁场定向控制（FOC）。https://github.com/
WangXuan95/FPGA-FOC。[78] Clifford Wolf。2025年。PicoRV32——尺寸优化的 RISC-V CPU。
https://github.com/YosysHQ/picorv32。[79] Claire Wolf。2025年。Yosys 开源综合套件。https://yosyshq.net/yosys/。[80] Luke Wren。2025年。Hazard3——带调试功能的三阶段 RV32IMACZb* 处理器。
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 和 Yungang Bao。2022年。使用敏捷方法开发高性能 RISC-V 处理器。载于2022年第55届 IEEE/ACM 微体系结构国际研讨会（MICRO）。 1178 1199 10 1109 56248 2022 00080 –。doi: 。/MICRO 。 。[82] Yizhuo Zhai、Yu Hao、Hang Zhang、Daimeng Wang、Chengyu Song、Zhiyun Qian、Mohsen Lesani、Srikanth V. Krishna- murthy 和 Paul Yu。2020年。UBITect：一种用于检测 Linux 内核中使用前未初始化错误的精确且可扩展方法。载于第28届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集（美国虚拟会议）（ESEC/FSE 2020）。计算机协会，美国纽约州纽约，221–232。doi:10.1145/3368089.3409686[83] Danfeng Zhang、Yao Wang、G. Edward Suh 和 Andrew C. Myers。2015年。一种面向时序敏感信息流安全的硬件设计语言。载于第二十届国际编程语言与操作系统体系结构支持会议论文集（土耳其伊斯坦布尔）（ASPLOS '15）。计算机协会，美国纽约州纽约，503–516。doi:10.1145/2694344.2694372