计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202331003
61(6): 1436−1449, 2024

面向处理器功能验证的硬件化 SystemVerilog 断言设计

张子卿[1,2] 石 侃[1,2] 徐烁翔[3] 王梁辉[4] 包云岗[1,2]

[1](处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

[2](中国科学院大学计算机科学与技术学院 北京 100049)

[3](上海科技大学信息科学与技术学院 上海 201210)

[4](中国科学技术大学 计算机科学与技术学院 合肥 230027)

(zhangziqing23z@ict.ac.cn)

Design of SystemVerilog Assertions Hardware Towards Efficient Processor Functional Verification

Zhang Ziqing[1,2], Shi Kan[1,2], Xu Shuoxiang[3], Wang Lianghui[4], and Bao Yungang[1,2]

[1](*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[2](*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 100049)

[3](*School of Information Science and Technology*, *Shanghaitech University*, *Shanghai* 201210)

[4](*School of Computer Science and Technology*, *University of Science and Technology of China*, *Hefei* 230027)

Abstract Processor verification occupies more than 70% of the time in the processor development flow, so it is necessary to optimize the efficiency of the processor verification process. Traditional verification methods such as software simulation provide various verification mechanisms including assertions to improve the fine-grained visibility and self-checking capability of verification, but software simulation runs slowly and lacks in efficiency. FPGA-based hardware simulation acceleration methods can greatly improve the verification performance, but debugging ability is weak, and it is difficult to locate the specific location and cause of vulnerabilities. In order to solve the above problems of verification efficiency and effectiveness, we propose a method to automatically convert non-synthesizable SystemVerilog Assertion (SVA) into logically equivalent but synthesizable RTL circuits, focusing on assertions, which is a type of non-global modeling of the design, and vertically penetrates through the various levels of abstraction, and complements the verification capability of the global ISA-based model, which can be used to verify the design. Our method complements the global ISA model-based verification capability. At the same time, combined with the advantages of FPGA fine-grained parallelization and high scalability, the verification process of the processor is hardware-accelerated, which improves the development efficiency of the processor. In this paper, we implement an end-to-end hardware assertion platform, integrate a complete toolchain for hardware-enabling SVAs, and count the triggering and coverage of hardware-enabled assertions running on FPGAs. Experiments show that the proposed method achieves more than 20 000 times verification efficiency improvement compared with software simulation.

Key words assertion; processor verification; hardware simulation; FPGA; prototyping

收稿日期: 2023-12-14; **修回日期**: 2024-03-22

基金项目: 国家重点研发计划项目(2023YFB4405105); 中国科学院计算技术研究所创新项目(E261100); 国家自然科学基金重大项目(62090023)

This work was supported by the National Key Research and development Program of China (2023YFB4405105), the Innovation Project of Institute of Computing Technology, Chinese Academy of Sciences (E261100), and the Major Program of the National Natural Science Foundation of China (62090023).

通信作者: 石侃(shikan@ict.ac.cn)

摘　要　功能验证在处理器芯片开发流程中所占用的时间超过 70%，因此优化提升功能验证环节的效率非常必要. 软件仿真等传统验证方法提供了包括断言等多种验证机制，以提升验证的细粒度可见性和自检查能力，但是软件仿真运行速度较慢，在高效性方面有明显不足. 基于 FPGA 的硬件原型验证方法能极大地加速验证性能，但其调试能力较弱，虽能快速发现漏洞，但难以定位漏洞出现的具体位置和根本原因，存在有效性不足难题. 为同时解决上述功能验证有效性与高效性的问题，提出一种将不可综合的断言语言 SVA（SystemVerilog Assertion）自动转换成逻辑等效但可综合的 RTL 电路的方法，聚焦于断言这一类对设计进行非全局建模、纵向贯穿各抽象层级的验证方式，对基于全局指令集架构（instruction set architecture, ISA）模型的验证能力进行补足. 同时，结合 FPGA 细粒度并行化、高度可扩展的优势，对处理器的验证过程进行硬件加速，提升了处理器的开发效率. 实现了一个端到端的硬件断言平台，集成对 SVA 进行硬件化的完整工具链，并统计运行在 FPGA 上的硬件化断言的触发和覆盖率情况. 实验表明，和软件仿真相比，所提方法能取得超过 2 万倍的验证效率提升.

关键词　断言；处理器验证；硬件仿真；现场可编程逻辑门阵列；原型验证

中图法分类号　TP391

处理器开发阶段可以分为设计阶段和验证阶段. 当前，处理器的验证已经成为了处理器开发流程中的关键路径[1-4]. 面对不断增长的算力需求和不断多样化的应用类型，处理器的开发过程变得愈加复杂. 一方面，多样的应用需求和高算力需求驱动着处理器开发者们不断设计出更为复杂的处理器微结构以适应这一趋势，处理器状态空间不断膨胀. 另一方面，面对多样化的算力需求和摩尔定律接近终结的现状，发展领域专用架构（domain specific architecture, DSA）成为新的趋势，这要求处理器开发走向敏捷，快速迭代以适应 DSA 发展的需求. 面对这些趋势，高抽象层次的设计语言和面向对象的开发方法的应用对处理器的设计效率带来了巨大的提升. 然而，处理器的验证效率并未相应地提升. 面对不断膨胀的处理器状态空间和正在涌现的处理器敏捷开发需求，处理器验证正面临着前所未有的挑战. 验证已经成为了处理器开发中耗时最长的阶段，通常占据芯片开发周期的 50%~70%[3]. 因此，提出全新的敏捷验证方法迫在眉睫.

以现代处理器当前的庞大规模而言，几乎不可能仅通过软件仿真的方式实现对处理器的充分验证. 从时间成本与经济成本的综合考虑的角度出发，FPGA 原型验证，作为一种价格相对低廉、运行速度极快的验证方法，成为工业界的处理器开发流程中不可缺少的一环. 然而，FPGA 的信号可见性差、可调试能力非常有限，因此很难对 FPGA 上检查到的处理器错误进行定位和解决. 基于断言的验证（assertion based verification, ABV）方法也是一种被广泛使用的方法[1,3]. 断言，尤其是基于时序逻辑的断言，能够高效地对处理器的局部功能行为进行检查，其贯穿于处理器设计的多个抽象层级，提升内部信号的可见性. 因此，断言可以很好地对 FPGA 原型验证的弱调试能力、弱信号可见性等不足进行针对性补充. 但是，在传统的 FPGA 原型验证当中，断言，尤其是支持时序逻辑的断言，是一种无法被 FPGA 的 EDA 工具综合的语法特性[5]，这一方面是由于 EDA 厂商长期只关注基础的调试工具，另一方面是由于基于时序逻辑的断言本身的复杂性（很多软件仿真器也未能完整实现对这些特性的支持）. 因此，断言方法被限制在部分软件仿真平台上，无法与 FPGA 原型验证相互配合使用.

为了解决以上问题，本文围绕着断言与 FPGA 原型验证，面向处理器的功能验证作出了 3 点贡献：

1）面向处理器功能验证需求，提出并实现了一种断言语言[6-7]SVA（SystemVerilog Assertion）的自动化编译方法，以及一套端到端的工具流和硬件平台，能将不可综合的断言转化为逻辑等效但可综合的 RTL 电路.

2）针对 FPGA 平台的断言提出了一种获取和利用断言结果的方法，从而对验证过程的进度进行指示，对设计运行状态进行检查.

3）通过 FPGA 加速的断言平台，取得了相对于软件仿真 2 万倍的验证加速能力，以及比传统硬件调试方法更小的资源占用和更全面的调试能力.

1　传统处理器功能验证方法及其特点

处理器验证首要的任务是保证处理器的功能正

确性, 即功能验证. 功能验证一般通过输入一定的测试激励, 动态地运行待测设计, 从而检查处理器运行过程中是否存在错误, 这与形式化验证方法有所不同. 处理器的功能验证基本上通过动态地与参考模型行为对比来实现: 将待验证设计(design under verification, DUV)与验证人员预期的设计行为进行对比. 高效的功能验证涉及到 2 个方面: 1)对于验证人员所预期的处理器行为进行建模的方式; 2)处理器的验证平台. 以下从这 2 个方面对传统的处理器验证方法进行介绍.

1.1 处理器行为建模方法

对处理器行为进行建模的方式, 决定了验证过程中使用的错误检查机制. 设计建模的层级、被建模状态与设计核心功能的关联程度, 都影响着验证的有效性与高效性. 一般来说, 建模的抽象层级越高, 就越难定位到错误根因; 被建模状态与核心功能关联越远, 在调试过程中就越需要更多额外回溯.

处理器验证中使用的错误检查机制大致可以被归类为 3 类: 基于 I/O 对比、基于全局指令集模型(instruction set simulator, ISS)对比以及基于局部模型对比.

基于 I/O 对比的方式通过观察 DUV 的输出结果是否与预期一致来对设计进行验证. 此种方式建模的抽象层级较高且与处理器的核心逻辑关联较弱, 因此当出现错误时, 往往需要大量回溯进行错误定位.

基于全局指令集模型对比的方式通过在处理器系统的某一个抽象层级, 如指令集架构(instruction set architecture, ISA)层级、微架构(micro-architecture, μARCH)层级, 建立 ISS 作为参考模型, 并暴露待测设计的关键信号与之进行对比, 从而对处理器进行验证. 全局模型对处理器的功能进行全面的建模, 然而模型所在抽象层次之下的一些错误相对来说难以发现, 如 ISA 级的 ISS 对比不会检查设计的分支预测实现, 而进一步提升建模的精度会造成建模成本的上升和模型可迁移性的下降.

基于局部模型对比的方式通过对设计的一部分功能进行建模来检查设计功能行为. ABV 就属于这一类型. ABV 方法是指验证人员在待测设计中加入断言来描述处理器局部的组合与时序行为以验证设计正确性的方法. 在功能验证中, 支持断言功能的平台会动态地对这些功能属性进行监测和检查, 判断是否出现违例. 对电路行为进行描述的断言语言有很多, 包括 SVA[6-7] 和属性规范语言(property specification language, PSL)等. 其中, SVA 可以内嵌于 SystemVerilog 代码当中, 而 SystemVerilog 语言[7] 则是当前使用最为广泛的验证语言之一[3]. SVA 相对于命题逻辑断言增加了对时序信息的描述, 可以描述一组信号在一段时间内复杂的组合与时序行为. SVA 中提供了多种描述时序逻辑(temporal logic)的操作符, 包括采样函数*$past* 和*$rose*, 延迟操作符##等. 相对于命题逻辑断言, SVA 表达能力更强、更贴近电路当中的硬件行为. SVA 可以从 2 方面有效地增强验证过程当中的信号可见性: 一方面是空间上可见性提升, 即局部关键信号的暴露; 另一方面是时间上可见性提升, 即断言时序逻辑描述能让对电路行为的检查不局限于某个周期, 而是对一段时间内的信号行为进行监测. 断言在实际中有着广泛的应用, 在处理器设计中, 断言可以应用于总线协议检查、仲裁机制合理性判断、状态机状态转移过程跟踪, 也可以在大型设计中各个系统模块集成时对各类系统级接口行为进行检测[8-9].

1.2 处理器验证平台

主流的处理器验证平台可以分为 3 类: 软件仿真器、硬件仿真处理器和 FPGA.

软件仿真器包括各种开源或商业的仿真器, 例如 Modelsim, VCS, Verilator, Icarus Verilog 等. 软件仿真方式拥有几乎全视的信号可见性, 能够记录大量的运行时信息, 方便对设计进行回溯. 同时, 极高的信号可见性带来了多种高效的验证方法, 可以动态地检查断言或在线与参考模型进行对比[2,4]. 软件仿真器最大的缺点是其极低的仿真速率. 软件仿真器周期精确的仿真方式使得软件仿真很难利用多处理器并行地对设计进行仿真, 这也使得软件仿真往往受限于处理器的单核频率[10]. 在实际应用中, 对于处理器规模的设计, 仿真器的运行速率一般在 kHz 级别, 因此, 软件仿真的运行效率极低, 现实中几乎难以仅依靠软件仿真对一个处理器进行充分地验证.

硬件仿真处理器是一种专门用于芯片验证的产品, 例如 Cadence 的 palladium、Synopsis 的 Zebu. 这类设备有着相对较好的可调试能力, 速度介于 FPGA 与软件仿真器之间. 其最大缺点在于价格极其高昂, 其成本每台达到几百万美元, 对于个人和中小规模的开发团体来说, 几乎没有使用的可能.

通过 FPGA 作为验证平台, 构建与处理器功能一致的硬件原型系统, 通过对该原型系统进行测试的方式被称为仿真或者原型验证[11]. FPGA 平台有着 3 类平台中最高的运行效率, 并且有着相对更低的价格. FPGA 原型验证方式充分利用了 FPGA 高并行度

的计算能力，相对于软件仿真器，可以对仿真速度起到质的提升. 由于 FPGA 仿真效率更高，可以使验证流程尽快收敛. 然而，传统的 FPGA 原型验证受限于自身弱调试能力，弱信号可见性的不足，往往被认为是一种高速运行以发现错误的工具而非定位错误的调试工具.

1.3 小 结

基于上述背景，断言与 FPGA 的结合对于基于断言的验证方法和 FPGA 验证平台都是很大的补充. 一方面，断言作为一种高效的验证手段，能够对 DUV 内部逻辑进行监测和检查，将设计内部的关键信号暴露出来，提升了验证过程中的信号可见性，对 FPGA 平台的验证方法有着很好的补充作用；另一方面，断言这种原本只能在软件仿真器上运行的机制，可以借助 FPGA 的高效运行速度更快更精准地定位错误，可以通过 FPGA 平台更快地完成验证. 正因如此，将 FPGA 与断言相结合成为了必然的方向.

2 基于 FPGA 加速的验证方法

综合时间成本与经济成本 2 方面的因素，FPGA 成为了充分验证处理器的较好选择. 本节主要针对基于 FPGA 的验证方法进行综述，重点围绕错误检查机制和可调试能力 2 个方面展开.

2.1 传统的 FPGA 原型验证方法

传统的 FPGA 原型验证方法使用 FPGA 厂商提供的 EDA 工具，将待测设计进行综合、布局布线、生成比特流等步骤映射到 FPGA 硬件电路上运行，并将运行结果和参考模型的结果进行比对，从而实现对待测设计的验证过程. 特别是对于处理器验证来说，基于 FPGA 的验证方法构建了一个系统级原型，可有效测试顶层软件和底层外设模型等软件仿真很难实际建模的场景，且运行速度远高于软件仿真，便于快速检测设计中是否存在错误.

然而，基于传统 FPGA 原型的验证方法在错误检查机制和可调试能力 2 方面都很有限，这主要受限于 2 个原因：传输带宽有限导致验证过程中只能对少量的信号进行监测；片上存储资源有限则限制了片上可容纳的运行时信息.

从错误检查机制来说，传统的 FPGA 原型验证方法往往只能使用简单的基于 I/O 的建模方式进行错误检查，通过 FPGA 的一些接口如串口的输出来检验设计的正确性. 从调试方法来说，FPGA 开发者一般是通过扫描链和 FPGA 生产商提供的信号记录缓冲(trace buffer)机制[12]来进行片上调试. 例如 Xilinx 公司提供的 ILA 机制和 Intel 的 SignalTap 都属于信号记录缓冲方式. 这些方式通过消耗片上的额外存储资源来提供一个对内部信号观察的窗口. 但是由于错误出现的时刻以极其离散的方式分布在验证过程当中，有限的观察窗口内对有限信号的观察难以支持验证的错误检查与定位.

针对于 FPGA 的这一短板，一些工作针对于如何提升 FPGA 调试能力的问题展开了研究. 例如，文献 [13] 提出了将软件仿真中常用 if-printf 语句组合进行硬件化，从而方便地设置信号记录缓冲的内容与触发条件，但这些工作实际上并未提升 FPGA 的验证能力，仍然未突破 FPGA 原型在调试上的 2 方面不足.

2.2 敏捷 FPGA 原型验证方法

伯克利大学的 DESSERT[14] 工作对于 FPGA 的错误检查机制和可调试能力都进行了一定的扩展. DESSERT 实现了简单的基于命题逻辑的断言和基于指令提交记录与软件仿真的对比，同时通过局部区域的输入输出记录实现了一定程度的错误重放.

多伦多大学的工作利用部分高端 FPGA 所支持的回读功能实现了一种软硬件协同的仿真框架 StateMover[15]. StateMover 是一种基于检查点的 FPGA 调试框架，该框架利用对电路某一时刻的状态进行快照，从而允许用户在仿真器和 FPGA 之间转移状态，实现了 FPGA 原型与软件仿真器之间的无缝衔接切换. StateMover 通过软件仿真器将 FPGA 上一瞬间的全局信号可见，拓展到检查点之后无限时间的全局信号可见. 然而 StateMover 要消耗秒级回读回写时间，相对于以大约 100 MHz 运行的 FPGA 原型系统来说，回读回写的开销显得过于高昂. 因此需要给出与出错现场接近的快照时间点，才能有效提升验证效率.

敏捷验证框架 ENCORE[4] 实现了一种软硬件协同仿真对比检查错误的机制. ENCORE 在 FPGA 的行为上进行了扩展. ENCORE 将 FPGA 上的处理器原型与通用处理器上的指令集模拟器进行对比，在线对比处理器 ISA 级的关键信号. ENCORE 同时结合了前文提到的快照机制，以原型系统和仿真的不一致作为快照触发条件. 相对于 StateMover, ENCORE 将出错点作为设计快照点，很好地解决了何时快照的问题. 但是，ISA 级的模型隐藏了底层的微架构细节，导致部分错误无法被即时捕捉，需要大量回溯. 而进一步提升模型的细节则会增大总线带宽的压力，降低软件端运行效率，导致整体系统效率进一步下降.

断言在 FPGA 原型系统中有 2 大优势：1)局部任

意粒度的行为建模；2)片上在线自检查. 正如前文所述，DESSERT 实现了基于命题逻辑的简单断言，但是此种断言的表达能力相对于 SVA 断言来说更弱. 此前也存在其他的一些基于 SVA 的硬件化断言方法[16-18]，但是这些工作往往只关注运算符的转换，而忽略了在实际设计中如何从设计中提取断言、如何将硬件断言与 DUV 进行连接，缺少对断言结果的访问方法和使用手段，需要一个全面完整的硬件化 SVA 的方法.

3 硬件化断言平台的设计

断言是一种有效的辅助调试手段和错误检查机制，对于 SVA 的硬件化的困难来源于 3 个方面：1)SVA 常内嵌于电路设计的代码之中，为了保证断言可以监测到正确的电路功能区域，需要分析断言与设计之间的连接关系，也需要考虑设计本身的嵌套关系. 2)SVA 中包含大量的不可综合运算符，例如非交叠蕴含(non-overlapped implication)操作符、延迟操作符等，需要提供这些运算符的对应可综合实现，才能将整个断言表达式转化为硬件，嵌入实际的硬件之中. 3)此前的工作鲜少涉及到断言结果的使用，断言结果包括断言触发对验证进度的指示和断言失败对错误的电路或断言状态的暴露. 本文针对这一问题进行研究并提出一套结果收集方案.

硬件化断言平台的整体工作流程与结构关系如图 1 所示. 本节后续内容按照图 1 中划分的三大部分进行介绍，分别是：硬件化断言编译器、硬件化断言算子库、硬件化断言收集器.

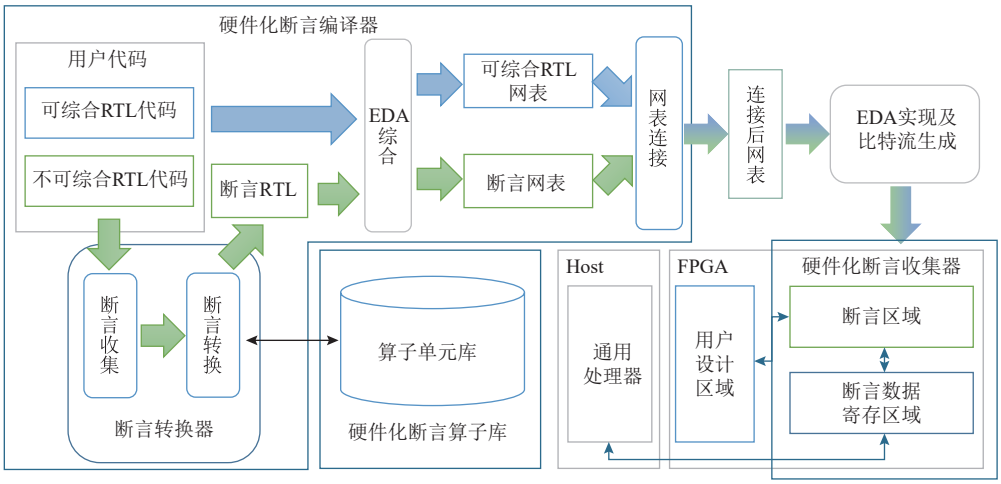


Fig. 1 Workflow and platform structure of the hardware assertion automatic complier

图 1 硬件化断言自动化编译器工作流与平台结构

3.1 硬件化断言编译器

硬件化断言编译器负责将设计中的断言进行提取、转换，同时保留必要的信息进行设计连接. 内嵌于设计代码之中的 SystemVerilog 断言与设计代码紧密关联，其中蕴涵了两大类的信息，即断言外部环境信息和断言内部结构信息. 外部环境信息指的是断言模块与待监测设计信号的连接关系，本质上是待监测设计信号与断言输入之间的映射关系. 内部结构信息则是断言的表达式结构，该结构描述了断言的各个子表达式之间的联系和数据流向. 硬件化断言编译器分 2 个步骤将这些信息进行提取和转化. 其具体结构如图 2 所示.

SystemVerilog 是一种基于上下无关文法的硬件描述语言. 为了充分表达断言的外部环境信息与内

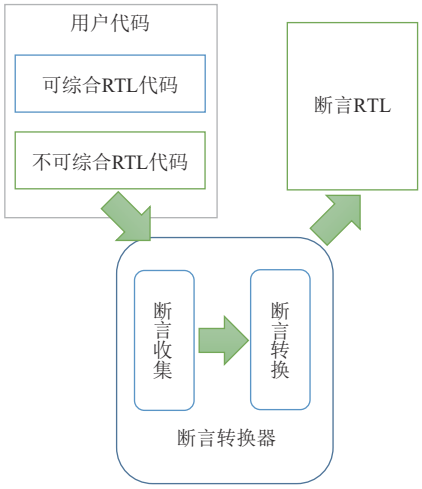


Fig. 2 Structure of the hardware assertion complier

图 2 硬件化断言编译器结构

部结构信息，本文选择在 SystemVerilog 的抽象语法树（abstract syntax tree, AST）上对断言代码进行提取和转化. 本文使用 Slang[19] 作为前端解析工具，对 SystemVerilog 代码进行词法分析和语法分析，从而获得其 AST. 后续的分析都在设计代码对应的 AST 或 AST 子树上进行.

为了提取断言外部信息，同时也为了去除 AST 其余部分对分析过程的干扰，只保留断言本身的内部结构，本文设计了断言提取工具来完成这一步骤. 断言提取工具首先将用户指定的设计文件输入到 Slang 当中并进行解析，得到其对应的 AST. 通过自上而下的方式遍历语法树，收集语法树上所有的断言子树. 值得注意的是，由于 SystemVerilog 存在层级嵌套关系，一个模块当中可能存在多个被实例化的子模块，一个含有断言的子模块也可能被多个模块所实例化. 此外，不同层级之间也可能存在重名等问题. 因此，为了维护断言与待监测信号之间的关系，需要对当前断言例化的作用域和断言进行记录，以便后续维护断言所需监测的信号层次化名称. 通过此种方式，维护断言所在的作用域内的名称，从而保证连接关系的正确性. 具体伪代码如算法 1 所示.

算法 1. 基于 AST 解析的断言收集算法.

输入：RTL 代码的 AST 根结点 *ASTnode* 与根路径 *$root*;

输出：所有断言表达式的子树根结点与其所在路径构成的二元组的集合 *S*.

① *GatherAssertions*(*ASTnode, path*);
② *S*←{};
③ if *ASTnode* is *Module* then
④ foreach *subNode* of *ASTnode* do
⑤ *S*←*S*∪*GatherAssertions*(*subNode, path.join*(*subNode.instanceName*));
⑥ end for
⑦ else if *ASTnode* is *Assertion* then
⑧ *S.insert*(<*ASTnode, path*>);
⑨ end if
⑩ return *S*.

算法 1 以 AST 根结点和 SystemVerilog 语言中定义的根路径为输入. 根路径代表了一个设计当中的顶层模块. 断言提取工具将 AST 上所有的断言子树进行提取构成一个断言集合，集合中的每一个元素称为一个断言实例，代表一个被设计代码例化的断言表达式. 每个表达式都排除了其与环境的关联，只包含表达式内部的结构. 以图 3 所示的 SystemVeilog

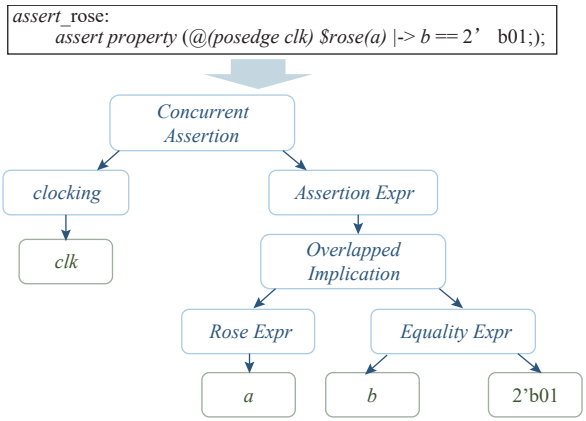


Fig. 3 An example of assertion expression syntax tree

图 3 断言表达式语法树示例

断言表达式为例. 对于这个表达式来说，断言形式会被表示为图 3 中的树形结构，即一个表达式子树.

对每个断言实例分别进行硬件化的任务由断言转化工具完成. 断言转化工具主要负责维护节点信息与连接关系. 由于各个节点之间存在较大的差异：节点本身的结构不同，节点本身代表的断言语义也不同，因此，每种类型的节点生成都有专门的处理方式. 按照处理方式的不同，断言节点大致可以被划分为 2 个大类，分别是代表表达式中间结果的非叶子节点 *InternalNode* 和代表待监测信号和常量的叶子节点 *LeafNode*.

对于叶子节点，断言转换工具提取该节点对应的变量位宽，对于待监测信号则记录其信号名称. 常量在更高层节点的处理当中会进行简单的常量传播优化，将一些不必要的子表达式进行剪枝，降低硬件的开销.

非叶子节点的处理相对较为复杂. 在断言子树中，一个非叶子节点可能代表 SystemVerilog 中的多种运算符，包括 4 类：采样函数（sample function）、序列（sequence）、属性（property）相关的操作符，以及 SystemVerilog 中的基本运算符，如逻辑运算、移位运算、关系运算和四则运算. 从节点对应的运算符是否可被综合，可以将所有的非叶子节点归类为可综合节点和不可综合节点. 对于可综合节点，算法仅需要将对应的子树进行扁平化即可；而对于不可综合节点，则需要将对应的子树替换为一个等价的 RTL 模块.

算法 2. 单个断言实例的硬件化算法.

输入：一个断言实例对应的表达式子树的根节点；

输出：实例化的断言与断言顶层信号.

① *EvalAssertions*(*ExprNode*);
② if *ExprNode* is *LeafNode* then

③ *NodeInfo*← *GenLeafNode*(*ExprNode*);
④ return *NodeInfo*;
⑤ else if *ExprNode* is *InternalNode* then
⑥ *SubNodeInfoSet*←{};
⑦ foreach *subNode* of *ExprNode* do
⑧ subNodeInfo←*EvalAssertion*(*subNode*);
⑨ *subNodeInfoSet.insert(subNodeInfo)*;
⑩ end foreach
⑪ *NodeInfo* ← *GenInteralNode*(*ExprNode*, *subNodeInfoSet*);
⑫ return *NodeInfo*;
⑬ end if

以图 4 中的断言实例对应的语法树为例, 对其中的*$rose* 表达式的操作过程介绍如下: 断言转化模块会自上而下递归访问整个表达式子树, 自下而上地构建表达式, 对于叶子结点 *a*, 这是一个待监测信号, 转化工具会记录路径, 为后面连接工作做准备, 程序会返回 *a* 的信号名和位宽信息到 *RoseExpr* 节点的转化程序当中, 该节点的转化程序会调用硬件库当中的*$rose* 函数对应的 RTL 模块, 并对该 RTL 模块进行实例化, 指定其连线与参数, 例化中间模块时需要利用子节点返回的信息隐式推断出当前模块的参数 *RoseExpr*, 例化完成后会以类似的形式将该模块的结果输出到上一层中代表交叠蕴含(overlapped implication)运算的 *Overlapped Implication* 进行处理, 过程类似, 不再赘述.

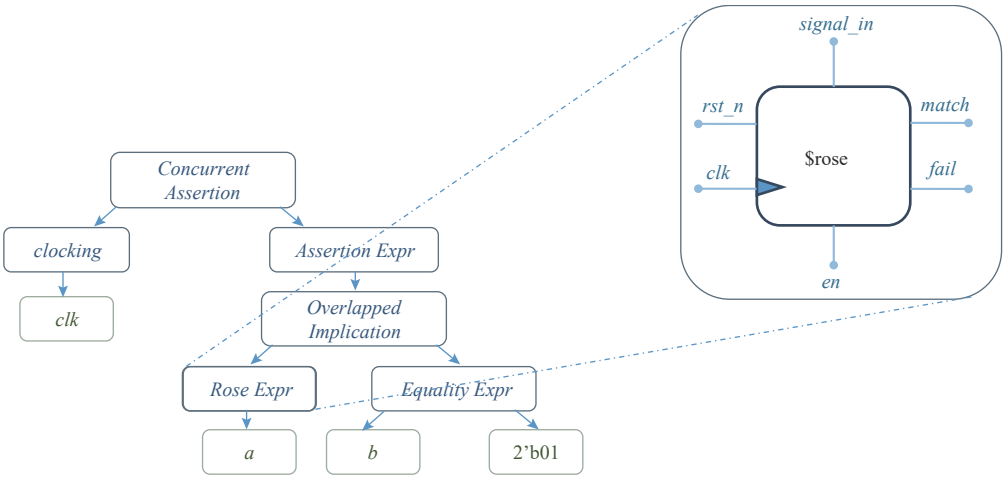


Fig. 4 An example of inner node instantiation

图 4 内部结点例化示例

每一个断言实例都会被转化为一个逻辑等效的硬件模块, 这样一个模块被称为一个硬件断言单元, 包含着表达式需要监测信号的所有接口, 记录当前断言状态的信号匹配(*match*)与失败(*fail*)以及必要的时钟与复位信号, *match* 和 *fail* 展示了当前的断言状态, 后续将会由断言数据收集平台进行记录供验证人员访问.

3.2 硬件化断言算子库

硬件化断言的核心步骤是对不可综合的运算符进行自动识别、替换, 并将其转换成语义等价的可综合 RTL 实现. 这一系列的可综合实现被称为硬件断言算子. SystemVerilog 断言中支持的序列操作符和属性操作符种类繁多, 同时还包括供形式化验证工具使用的运算符. 因此, 本文面向处理器的功能验证需求, 针对 4 类的操作符, 实现了其对应的操作符:

1)采样函数. 包括*$rose*, *$past*, *$fell* 等; 这些运算符主要用来对设计中的一些信号进行采样和行为检测. 其中*$past* 是其中最为基础的采样函数, 原则上其他的采样函数都可以由*$past* 导出, 采样函数提供了方便的接口去采样一个信号过去未来的值, 对于处理器设计中的信号变化、状态转移, 都可以通过采样函数进行描述.

2)序列操作符(sequence operator). 包括 *and*, *or*, *intersect* 等, 这些运算符主要对 2 个时序序列进行与或操作. 与简单的布尔变量的与或运算不同, 2 个序列的评估并非 1 个瞬时可以完成, 往往需要多个周期、序列操作符.

3)延时与重复. 包括延时操作符##和重复操作符 [*n]. 延时操作符在 SVA 标准中被称为序列连接(sequence concatenation)操作符, 其后的操作数指定了前后 2 个序列的评估时序关系, ## 1 表示其后的序列在其前面的序列结束后 1 周期开始评估, ##[1:2]

则指明了多种可能的序列. 重复(repetition)操作符则表示一个序列会重复若干次. 延迟与重复在 SVA 标准中都属于序列操作符, 但由于延迟与重复的操作在基于 SVA 的验证中使用太过广泛, 因此, 本文将其独立出来作为单独的一类. 延时与重复操作符以一种类似正则表达式的形式将若干信号的时序行为表示出来. 例如 *a* ##1 *b*[*5] 描述了信号 *a* 使能后, 一周期后信号 *b* 会使能 5 个周期.

4)属性操作符. 包括非交叠蕴含|->和交叠蕴含|=>. 蕴含操作符描述了一种条件关系, 即蕴含操作符的起因序列(antecedent sequence)匹配时, 会对其结果序列(consequent sequence)进行判断, 只有当起因序列匹配且结果序列不匹配时, 蕴含才判定为假. 例如 *a* |-> *b* 这样一个断言, *a* 是其中的起因序列, *b* 是其中的结果序列, 当信号 *a* 为真时, 信号 *b* 也应当为真. 交叠蕴含与非交叠蕴含的区别在于前者的结果序列评估开始于起因序列评估结束后的一个周期, 而后者的结果序列评估与起因序列评估的结束同时发生. 蕴含操作符同样在处理器设计的验证中应用非常广泛, 因为验证者很多情况下会关注某一条件下的处理器行为, 这类行为很容易通过蕴含操作符表示.

硬件算子繁多而复杂, 但是不同的硬件算子满足着相同的基本设计规律. 以下主要讨论硬件断言算子的基本结构, 以及断言输入输出的形成.

从实现的思路上来说, 不同的硬件断言算子都可以被视为一个有限状态机. 每一个运算符会维护其运算状态, 用于记录正在等待子节点所代表的断言算子结果或正节点中处于计算当中的待运算任务以及判断最终的计算结果. 不同的运算符状态机通过使能与反馈机制将各个运算符联合成一个有机整体共同实现完整的断言功能.

从输出结构来说, 大多数硬件断言算子采用了 *match*, *fail* 信号组作为输出, 用以描述断言表达式的结果. 这一组输出被称为断言标准输出. 由于 SVA 断言是一种基于时序逻辑的断言描述方式, 其描述的断言往往不能在当前周期就完成断言表达式结构的计算, 甚至计算周期不是固定的, 因此, 断言表达式的结果并不是一个非对即错的二值逻辑, 还存在处于计算当中的中间态. 因此, *match* 信号标志着一个表达式或子表达式出现了匹配, 即一次对断言的评估尝试结束并且与断言所描述的行为相符. 而 *fail* 信号则相反, 若断言的一次评估尝试结束但被监测信号的行为与硬件预期不相符, 则会出现 *fail*. 一个完整的断言表达式同样采用标准输出, 一个断言表达式的结果就是最顶层硬件化断言算子的输出.

断言的输入通过采样函数来完成. 采样函数是硬件算子库中一类特殊的操作符, 负责对输入信号或输入信号组成的组合逻辑信号进行采样并进行一系列基本的运算. 例如, *$rose*, *$past* 函数通过对输入信号进行采样, 计算其当前值相对于前一周期是否发生上升或下降, 从而生成相应的 *match* 和 *fail* 信号. 采样函数中还有另一类纯粹的采样函数, 这类函数只负责对当前的信号值进行采样, 而不进行额外的逻辑运算, 如 *$past*, *$sample*, 其中 *$past* 函数会采样某个信号若干周期前的值, 这一类采样函数则不采用标准输出, 而是此前将某周期的信号值进行输出, 并将其作为其他算子的输入进行下一步运算. 除了 SVA 中支持的标准采样运算符之外, 本文还给出了一类自定义的特殊采样函数 *signal_check*, 主要用于将一个组合逻辑表达式转化为标准输出. 对于一个只包含组合逻辑表达式的断言, 如@(*posedge clk*) *a &b*; *signal_check* 能保证该断言与其他断言的结构同一性. 对于一个时序逻辑断言的组合逻辑部分, 也需要通过 *signal_check* 将其进行转化, 以使得该断言表达式的结果可以作为其他断言操作符的输入.

为进一步阐述断言算子的内部结构及外界互联关系, 以“|->”运算符为例进行具体阐述. 从端口上看, 该断言包含 3 个部分, 分别是连接前因序列的标准输出、连接结果序列的标准输出和使能, 以及该模块的标准输出. 当起因序列的评估结束后, 对断言的评估转移至“|->”运算符, 该运算符将会使能, 若结果为 *match* 信号使能, 则该运算符会使能后继运算符进行评估, 在后继节点完成计算后, 输出本节点的评估结果. 由于一个序列的评估可能耗费多个周期, 而在后继序列进行评估的过程中, 也可能存在前因序列因为一次不同的断言评估请求而产生新的评估操作, 运算符内支持对于一定时间内已发出的对后继断言的评估操作进行记录, 并内部维护这些评估操作的时序关系, 以便正确地产生结果.

从方法学的角度, 验证可分为基于动态方法的功能仿真验证和基于静态方法的形式化验证. 这 2 种方法都会使用断言的方式进行验证实现, 但二者常使用的运算符不同. 本文是基于 FPGA 加速的仿真方法, 因而针对开源项目中基于仿真的功能验证需求进行调研, 归纳得到表 1 中的 13 种运算符. 对于不在表 1 中的断言操作符, 本文工具尽管不能直接将其转化为对应的硬件电路, 但仍支持对于该部分的断

Table 1 Supported Hardware Assertion Operators

表 1 已支持的硬件化断言算子

硬件算子类型	硬件算子名称	算子使用示例
采样函数	*$rose*	*$rose(a)*
	$fell	*$fell(a)*
	$stable	*$stable(a)*
	$past	*$past(a)*
序列操作符	*and*	*a and b*
	or	*a or b*
	intersect	*a intersect b*
	first_match	*first_match(a)*
属性操作符	*|=>*	*a |=> b*
	|->	*a |-> b*
	if … else	*if (a) b else c*
延迟与重复	*##*	*a ## 1 b / a ##*[1:2]*b*
	[*]	*a*[*2]/*a*[*1:2]*b*

言符号识别，当识别到该类运算符时会对用户进行报告. 本文最终支持共计包括隐式调用的 *signal_check* 在内的 14 种操作符，可以覆盖处理器功能验证的基本需求.

3.3 硬件化断言方法

断言是一种重要的测试手段，而断言是否触发，则是一种重要的测试指标. 以往的工作往往忽略了断言触发结果的获取，未将其作为一个指导验证过程的指标.

如 3.1 节所述，每一个硬件断言单元会给出断言的触发与失败状态，断言的触发，即 *match* 信号使能，标志着当前的断言所描述的行为是设计中实际会出现的行为，断言的触发与否同样是一种测试充分性的指标，称为断言覆盖率. 本文扩展了基础的断言覆盖率，从断言覆盖与否转为断言覆盖的次数：通过将一个断言单元的 *match* 信号与一个计数器相连，通过计数器数值反映测试进展. 各个断言对应的计数器构成断言计数器组，而当断言失败，即 *fail* 信号使能，则表示断言所描述的行为与实际设计运行行为不符，需要检查设计或断言当中是否存在错误. 一个断言单元的 *fail* 信号连接到上位机的中断控制器上，以及时地向上位机反馈当前断言的运行状况. 所有的断言会被编号，而触发失败的断言会被记录在相应的寄存器当中供上位机的中断处理函数读取访问，如图 5 所示.

为了向上位机提供简单的访问界面，本文提供了一个根据地址索引访问断言数据的简单协议，在断言数据区域的地址空间范围内，上位机可通过地址索引对该模块进行访问. 断言数据平台的数据流结构如图 6 所示. 断言数据区域除了断言计数器组和断言失败记录，还有性能计数器用于生成断言触发

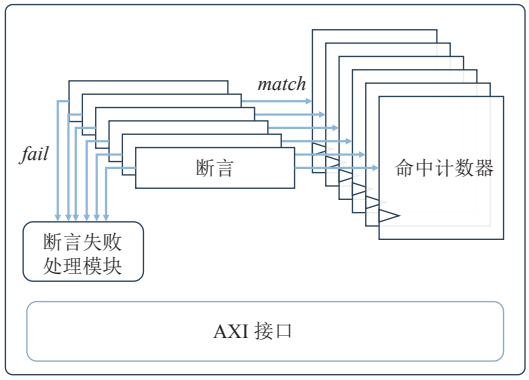


Fig. 5 Structure of assertion data region

图 5 断言数据区域结构

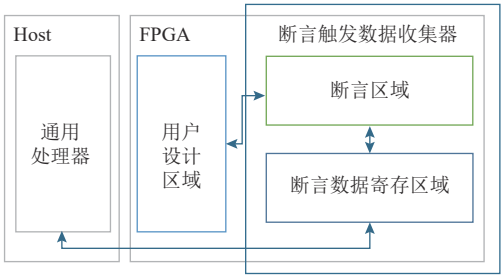


Fig. 6 Dataflow structure of assertion data platform

图 6 断言数据平台数据流结构

的时间戳以及进行性能方面的比较. 具体的地址映射关系如表 2 所示.

Table 2 Address Mapping of Assertion Data

表 2 断言数据地址分配

地址	地址段功能	寄存器宽度
0x0000-0x0fff	触发计数器	16
0x1000	断言失败记录	32
0x1004	性能计数器低位	32
0x1006	性能计数器高位	32

本文设计基于软硬件协同仿真，将 DUT 映射到 FPGA 上，并通过 FPGA 与上位机的数据交互通路实现硬件仿真过程、测试覆盖程度、断言触发情况的监测. 由于对断言状态的访问数据量小，本文设计对于上位机的性能、架构、数据通路协议、带宽没有特定要求. 本文的具体实验环境将在第 5.1 节进行了详细介绍.

本文实现了一个基于 AMD ZYNQ SoC(System on Chip)的具体用例，设计了一个端到端的硬件断言平台. 一个 AMD ZYNQ 片上系统由 2 部分组成，其中的处理系统(processing system, PS)充当上位机，与 FPGA 部分，即可编程逻辑(programmable logic, PL)通过 AXI 接口进行交互，用户可以通过相应的软件读取到硬件上的断言数据信息.

4 基于断言的 RISC-V 处理器核功能验证

4.1 实例分析：开源 RISC-V 处理器核的功能验证

为了对提出的硬件化断言的有效性和高效性进行全面评估，本文选取了一款开源 RISC-V 处理器核：果壳处理器（Nutshell）[20] 作为待验证设计和实验对象. Nutshell 是一个基于 RISC-V 架构的 64 位处理器，使用 Chisel[21] 语言进行开发，支持 RISC-V 的 I, M, A, C, Zicsr 与 Zifencei 的指令扩展和 M, S, U 特权级. Nutshell 支持分支预测、多级缓存，可以运行 Linux/Debian 系统.

4.2 针对特定微架构处理器的断言集合

本文设计的断言分为两大类：一类断言是基于 RISC-V 架构或者通用总线标准的断言，这类断言和待验证设计的微架构细节无关，也并非针对 Nutshell 设计，并且可以被快速迁移到其他采用相同指令集或总线协议的设备上. 另一类断言则聚焦于与处理器核微架构相关的行为，意在展示断言针对处理器微架构的验证能力.

断言覆盖的内容包括处理器的缓存、分支预测器、总线协议、特权寄存器行为等，具体如表 3 所示. 这些断言展示了针对标准断言的可扩展性，也展示了断言本身所具有的贯通各个抽象层级的验证能力.

Table 3 Type and Amount of Assertions

表 3 断言类型与数量

断言类型	断言子类型	数量	属性来源
AXI Assertion	KEEP STABLE	29	Standard
	INVALID WHEN RESET	10	Standard
	TIME_OUT	1	Standard
	RESERVE SIGNAL	2	Standard
CSR Assertion	TRAP	5	Standard
	RET	6	Standard
	EXCEPTION PIRORITY	4	Standard
BPU Assertion		4	Design Spec
Cache Assertion		3	Design Spec
Regfile		1	Standard

4.3 断言嵌入方法

为了解决 Chisel 转化后的 Verilog 文件可读性较差、难以进行验证的问题，本文采用了 Chisel 中的黑盒（blackbox）机制，通过黑盒可以在 Chisel 当中以模块的形式调用 SystemVerilog 设计的断言. 基于黑盒的方法可以在 Chisel 层面连接断言与设计，以保证断言能够正确地监视信号.

为了对比 Nutshell 在加入和去除断言后在硬件上的性能差异，以及基于硬件化断言的验证方法与 Xilinx 的 ILA 这样一种传统 FPGA 调试方式的资源占用对比，本文采用了参数化的设计方法，通过调整传入到 Chisel 的编译器参数，设计可以在无断言模式、断言模式、ILA 模式间直接进行切换. 其中 ILA 模式通过 Chisel 内置的 BoringUtil 和参数化端口将处理器内部的信号飞线至顶层模块，主要用于与传统的 ILA 模式进行对比.

5 硬件化断言平台上的 RISC-V 处理器核功能验证

本节主要对于面向 RISC-V 处理器核 Nutshell 的硬件化断言平台进行评估. 本文通过硬件化断言平台的工具支持，将第 4 节中提到的断言集合进行硬件化，并与 Nutshell 设计相连，构建了完整的面向 RISC-V 处理器核的硬件化断言平台，并通过一系列的实验完成 3 个问题的回答：1）整个测试在何种平台上完成？ 2）将断言与 RISC-V 处理器核相连后，是否能够有效地对 Nutshell 进行验证；3）硬件化断言平台在资源占用、运行效率方面表现如何？

5.1 实验平台

本文的软件测试运行在一台拥有 11th Gen Intel Core i9-11 700 CPU, 32 GB 3200MT/s 的内存的机器上. 操作系统为 Ubuntu 22.04, 运行 Modelsim 2 020.4 作为仿真后端.

本文的硬件部分将 Nutshell 部署在 Fidus Sidewinder 开发板上，其上的 SoC 芯片是 AMD ZYNQ UltraScale+ XCZU19EG SoC，该芯片包括一块拥有 114.3 万逻辑单元的大容量 FPGA（PL 端）和 4 核 ARM Cortex-A53 处理器（PS 端）.

5.2 软件平台上的 RISC-V 处理器核功能验证

为了对比硬件化断言在验证效率上的提升，本文以软件仿真器为基线进行评估. 首先在软件平台上运行 Nutshell，并提供一致的断言集合.

为了能够运行 SystemVerilog 断言，本文选取了 Modelsim 作为软件测试平台. 相对于不支持断言或只支持简单断言的开源仿真器 Verilator 和 Icarus Verilog，Modelsim 支持更为完整的断言功能，因此本文选取 Modelsim 作为仿真器比较的基线. 由于 Modelsim 在运行时本身不会记录实际的仿真时间，本文额外使

用 VPI 机制记录仿真开始和结束的时间戳, 以精准地测量仿真的时间.

另外, 为了能够在一定程度上确认断言是否能找到处理器内核中的错误, 本文在 Nutshell 的代码中植入了若干人为设计漏洞, 如针对于寄存器堆和分支预测器的逻辑错误等. 这些人为植入的漏洞可以通过参数化选项在编译时选择开启或关闭. 实验中; 为了对 RISC-V 处理器核进行功能验证, 本文选取了一系列常用的通用基准测试(Benchmark)程序作为处理器的工作负载, 包括 Coremark[22], Dhrystone[23], Microbench[24], 其中, Microbench 包括 10 个子测试程序, 如 qsort, bf, lzip, md5 等. 本文实验选取了每个基准测试的前 1 000 000 条指令执行.

从实验结果来看, 在功能方面, 人为插入的断言均能被硬件化断言检测到; 而在性能方面, 在软件 Modelsim 当中以不同的基准测试作为测试负载, Nutshell 的仿真时间平均为 456.16 s. 实验过程中记录了仿真过程中的周期数, 通过将周期数与总共花费的时间做比值, 可以得到软件仿真的仿真频率, Nutshell 的仿真频率为 4~7 kHz 左右. 具体的结果如表 4 所示.

Table 4 Testing Results of Each Benchmark Run by Software Simulation Nutshell

表 4 软件仿真 Nutshell 运行各基准测试结果

基准测试	运行时间/s	周期数	仿真频率/kHz
coremark	371.42	1 858 853	5.005
dhrystone	437.63	2 143 378	4.898
microbench-qsort	463.21	2 369 069	5.114
microbench-queen	360.46	1 821 639	5.054
microbench-bf	448.01	1 967 435	4.392
microbench-fib	390.40	2 639 506	6.761
microbench-sieve	485.60	3 174 664	6.538
microbench-15pz	457.99	2 585 612	5.646
microbench-dinic	539.19	3 412 524	6.329
microbench-lzip	510.03	3 796 330	7.443
microbench-ssort	495.73	3 109 954	6.273
microbench-md5	514.30	4 066 747	7.907

实际上, 1 000 000 条指令相较于动态指令数的规模而言只占很小的一部分. 以 coremark 为例, 完整运行 coremark 程序需要总共运行 353 976 300 条指令, 是目前运行的测试集指令数的 354 倍. 以现有的仿真频率运行, 大约需要一天半的时间. microbench 的 md5 测试程序的指令数高达 6 576 597 538 条, 在现有仿真频率下需要 39 天才能完整运行完. 另外, 在运行仿真的过程中, Modelsim 的波形输出被全部关闭, 如果发生错误, Modelsim 没有任何对错误的回溯能力. 如果打开波形输出的开关, 仿真器的性能还会进一步降低.

5.3 硬件化断言平台上的 RISC-V 内核的功能验证

为了验证硬件化断言平台的验证效率, 本文将 5.2 节中软件运行的断言集合进行硬件化, 与 Nutshell 一起部署在 FPGA 平台上, 部署结果如图 7 所示. 硬件 FPGA 将 Nutshell 默认配置在 100 MHz 的情况下进行, 对其运行时间进行了测试, 运行时间通过记录的运行周期数除以时钟频率得到.

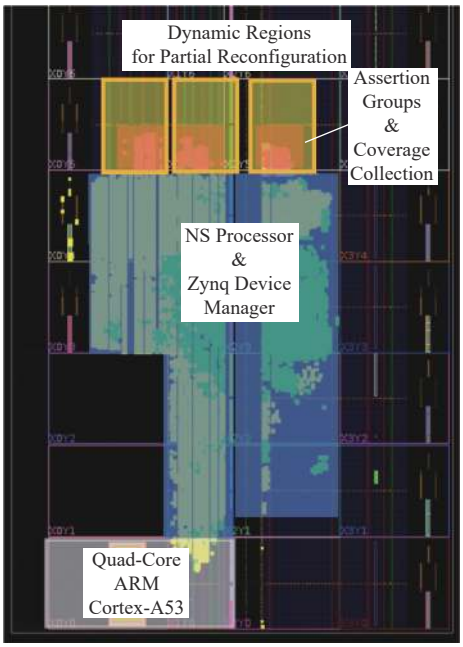


Fig. 7 Floorplan of hardware assertion platform

图 7 硬件化断言平台版图

本文对比了硬件化断言平台相对于软件仿真运行相同基准测试的时间, 通过计算相同的指令数量下, 硬件与软件的运行时间占比来计算加速比, 具体结果如图 8 所示. 不同的基准测试的加速比不同, 其

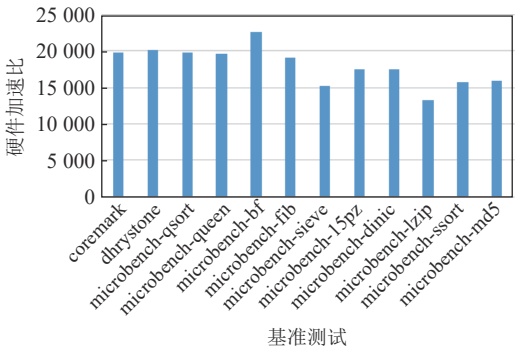


Fig. 8 Acceleration ratio of FPGA simulation compared with software simulation

图 8 FPGA 仿真相对于软件仿真加速比

范围是 13 424~22 589, 几何平均数为 17 858. 即, 软件需要运行几天甚至几周的基准测试程序, 硬件只需要跑几分钟就可以得到结果. 如果为软件仿真器提供基本的调试能力, 如波形导出, 其速度还将进一步下降, 这一加速比还将继续提升.

传统的验证调试主要采用 ILA 等基于信号记录缓冲的方法, 为了与传统的 ILA 的方式进行对比, 比较硬件化断言的额外资源占用, 本文同时测试了基于 ILA 的调试方式的面积资源占用.

在工程实践当中, 常常会面临通过 ILA 设计内部行为逻辑检查的需求, 因而本文设置了实验对断言方式的资源占用进行有效说明: 通过 ILA 方式与断言对相同功能进行检查. 为此, 本文通过 Vivado 工具链将所有原本被断言监视的信号与 ILA 相连. ILA 通过接出的信号进行离线检查, 而断言进行在线检查. 本文配置了不同的 ILA 深度, 包括 1 024 和 65 536 这 2 种配置, 具体的资源占用如表 5 所示. 为了对比, 表 5 中同时列举了 Nutshell 部署在 FPGA 上本身所消耗的资源数量作为一个基准参考.

Table 5 Resource Usage and Resource Usage Ratio of Hardware Assertions and ILA

表 5 硬件断言与 ILA 的资源占用量和资源占用比

资源类型	硬件断言	ILA（配置 1）	ILA(配置 2)	果壳处理器
LUTs	1 223 (0.23%)	2 266 (0.43%)	3 749 (0.72%)	41 537 (7.95%)
Registers	933 (0.09%)	3 833 (0.37%)	4 243 (0.41%)	53 273 (5.1%)
BRAMs	0 (0%)	8 (0.81%)	512 (52.03%)	56 (5.69%)
DSPs	0 (0%)	0 (0%)	0 (0%)	16 (0.81%)

注：括号内部的资源占用比表示该类型资源用量与 FPGA 片上资源总量的比值.

可以看到, 硬件断言相对传统的 ILA 方式, 占用的资源更小. 其中, 硬件化的断言几乎不耗费 BRAM 资源, 这是由于硬件化断言并不需要保存过去运行状态的全部过程, 只需要捕捉关键信号和关键逻辑的出错现场即可. 反观 ILA, 它需要通过信号记录缓冲机制保存运行时信息, 即使这些信息对于调试并无帮助, 甚至会成为系统的资源瓶颈. 比如, 深度为 65 536 的 ILA 仅能监测 6 万个周期的设计状态, 使用的 BRAM 资源却超过了片上该资源总量的一半, 这一数量甚至远远超过 Nutshell 本身所使用的 BRAM 资源数量.

传统的基于 ILA 调试方法中, 设计者往往会在可能出错的点位附近设置 ILA 的记录触发器, 然后观察可能出错点位前后的信号状态以确定错误. 然而, 真正导致设计错误的只有很短时间内的异常行为, 因此, ILA 的记录中含有大量冗余信息. 与之对比, 断言可以更精准地定位错误的位置, 并且内嵌了对于设计行为的描述, 所以断言不需要额外的记录, 因此硬件断言相对于传统的 ILA 方式消耗的资源用量更小.

相对于传统的 FPGA 原型验证方式, ILA 和断言由于在设计上插入了新的硬件单元, 对于 FPGA 编译过程中的综合与布局布线等步骤会产生一定的影响, 进而影响系统时序的收敛. 但如果新加入的硬件单元不在系统的关键路径上, 则对时序的影响较小. 为验证此问题, 本文利用 Vivado 工具对实现后的设计时序进行分析, 对处理器所在时钟域下的时序裕量进行测量, 并计算其理论最大频率 *fmax*. 实验结果表明, ILA 的 *fmax* 值为 169.43 MHz 而断言的 *fmax* 值是 172.08 MHz, 因此可以认为二者对于时序的影响差异较小. 此外通过检查时序报告发现, 整个设计的关键路径并不在处理器或者断言实现, 而在待测处理器之外用于模拟外界环境的外设, 因此 ILA 和断言对于完整工程的时序都无明显影响.

6 其他硬件化断言工作

在本文之前, 国际上也有一些学者对断言的硬件化做了研究, 然而, 这些工作既没有开放其源代码也没有开放其设计的硬件化断言工具, 无法定量地进行横向对比. 因此, 本节根据这些工作的描述, 主要讨论本文与此前工作在设计思路、技术选型上的不同.

Pellauer 等人[16] 提出了一种将 SVA 转换成 Bluespec SystemVerilog 的方法, 支持了较为完整的 SystemVerilog 断言运算符, 然而, 该工作对于常用的采样函数如 *$rose*, *$past* 等没有支持. 从实现上来说, 该工作以一个 Sequence/Property 表达式为建立状态机的基本单元, 对于 SVA 中的多种可能的执行路径采用对 Property 状态机复制的方式来实现.

Das 等人[17] 的工作支持的运算符相对更少. 实现上来说, 该工作以延迟/重复/基本布尔逻辑为基础模块, 在此基础上构建出复杂的断言实现. 对于每个 Sequence 操作符, 只提供一个 *match* 信号用于标注断言匹配, 对于断言失败的情况, 则是通过对 Sequence 表达式进行逻辑取反等价变换, 对变换后的表达式生成一个额外的断言硬件模块进行检测. 对于推测路径, Das 等人也采用了复制所有执行路径的状态机的方法.

Kastelan 等人[18] 的工作在支持运算符上同样比较少. 实现上来看, 他们的工作同样需要额外的逻辑

用于计算断言失败的情况. 另外, 对于推测路径的处理没有太多涉及.

本文工作面向功能验证的需求, 实现了足够多的断言运算符. 从实现上说, 本文工作以运算符作为基本单元, 每个基本单元存储了必要的信息来维护正在进行中的断言计算以解决推测执行问题. 文献 [16-18] 的工作除了断言本身实现与本文工作不同外, 这些工作都仅仅聚焦在断言单元的实例化. 对于断言提取、连接、数据收集都没有关注, 不能形成真正完整的断言工具链.

7 结 论

本文聚焦于处理器验证中的实际问题, 针对果壳处理器(Nutshell)这一个具体的处理器实例, 根据待验证处理器的具体功能设计了一组断言进行验证.

断言是一种贯穿设计各层级的非全局建模方法. 相对于传统的 ILA 等方式, 硬件化断言提升了 FPGA 原型验证中的信号可见性, 大大提升了验证的有效性与高效性, 能够帮助开发者在验证过程中更好地发现错误、定位错误. 通过断言的方式, 本文针对一个 RISC-V 内核的 ISA 级和 μARCH 级的行为进行了验证, 确认断言方法能够有效发现和定位处理器当中的错误.

面向处理器功能验证的需求, 本文提供了一个 SystemVerilog 断言子集的硬件化方法, 将原有的软件断言以更高效的形式运行在硬件上. 同时提供了硬件断言的断言触发情况收集平台, 让硬件验证人员可以有效地对测试进度与设计异常进行监测.

通过对于硬件化断言平台的实验可以看到, 本文相对于软件断言有着高达 2 万倍的加速比, 极大提升了验证效率. 断言提供了细粒度的可见性和自检查能力, 很好缓解了 FPGA 本身信号可见性不足的问题. 相对于 ILA 等传统 FPGA 调试方式, 本文提出的方法也有更高的调试效率.

当然, 目前的硬件断言平台与特定 RISC-V 内核的验证需求息息相关, 如果未来要向更为实际的应用场景中迁移, 还需要进一步地丰富硬件化断言的转化工具链. 另外, 在功能性验证方面, 单纯依靠断言很难做到全面, 仍然依赖于基于模型检查的方法进行补充.

作者贡献声明: 张子卿完成了设计的实现和论文的撰写; 石侃对论文的实现与撰写进行了指导; 徐烁翔协助完成实验平台的搭建和部分实验数据的测量; 王梁辉协助完成了硬件断言算子库的设计; 包云岗提出指导意见. 徐烁翔和王梁辉在中国科学院计算技术研究所实习期间完成本文相关工作.

参 考 文 献

[1] Witharana H, Lyu Y, Charles S, et al. A survey on assertion-based hardware verification[J]. ACM Computing Surveys, 2022, 54(11s): 1−33

[2] Xu Yin'an, Yu Zihao, Tang Dan, et al. Towards developing high performance RISC-V processors using Agile methodology[C]//Proc of the 55th IEEE/ACM Int Symp on Microarchitecture (MICRO). Piscataway, NJ: IEEE, 2022: 1178−1199

[3] GRAPHICS M. The 2020 Wilson research group functional verification study[EB/OL]. (2020-10-10)[2023-12-08]. https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/

[4] Shi Kan, Xu Shuaxiang, Diao Yuhan, et al. ENCORE: Efficient architecture verification framework with FPGA acceleration[C]//Proc of the ACM/SIGDA Int Symp on Field Programmable Gate Arrays. New York: ACM, 2023: 209–219

[5] Xilinx. Vivado Design Suite User Guide: Synthesis (UG901)[EB/OL]. [2023-04-17]. https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis

[6] Cerny E, Dudani S, Havlicek J, et al. SVA: The Power of Assertions in SystemVerilog[M]. Berlin: Springer, 2015

[7] Design Automation Standards Committee. IEEE1800-2017 IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language Standard IEEE 1800[S].[2023-12-03]. http://www.edastds.org/sv/

[8] Zhou Zhili, Xie Zheng, Wang Xin'an. Development of verification envioronment for SPI master interface using SystemVerilog[C]//Proc of the 11th Int Conf on Signal Processing. Piscataway, NJ: IEEE, 2012: 2188−2192

[9] Gurha P, Khandelwal R R. Systemverilog assertion based verification of amba-ahb[C]//Proc of 2016 Int Conf on Micro-Electronics and Telecommunication Engineering (ICMETE). Piscataway, NJ: IEEE, 2016: 641−645

[10] Tan Zhangxi, Andrew W, Henry C, et al. A case for FAME: FPGA architecture model execution[C]//Proc of the 37th Annual Int Symp on Computer Architecture. New York: ACM, 2010: 290–301

[11] Krupnova H, Saucier G. FPGA-based emulation: Industrial and custom prototyping solutions[C]//Proc of Int Workshop on Field Programmable Logic and Applications. Berlin: Springer, 2000: 68–77

[12] Veneris A, Keng B, Safarpour S. From RTL to silicon: The case for automated debug[C]//Proc of the 16th Asia and South Pacific Design Automation Conf (ASP-DAC 2011). Piscataway, NJ: IEEE, 2011: 306−310

[13] Ma Jiacheng, Zuo Gefei, Loughlin K, et al. Debugging in the brave new world of reconfigurable hardware[C]//Proc of the 27th ACM Int Conf on Architectural Support for Programming Languages and

Operating Systems. New York: ACM, 2022: 946−962

[14] Kim D, Celio C, Karandikar S, et al. DESSERT: Debugging RTL effectively with state snapshotting for error replays across trillions of cycles[C]//Proc of the 28th Int Conf on Field Programmable Logic and Applications (FPL). Piscataway, NJ: IEEE, 2018: 76−764

[15] Attia S, Betz V. StateMover: Combining simulation and hardware execution for efficient FPGA debugging[C]//Proc of the 2020 ACM/SIGDA Int Symp on Field-Programmable Gate Arrays. New York: ACM, 2020: 175−185

[16] Pellauer M, Lis M, Baltus D, et al. Synthesis of synchronous assertions with guarded atomic actions[C]//Proc of the 2nd ACM and IEEE Int Conf on Formal Methods and Models for Co-Design (MEMOCODE'05). Piscataway, NJ: IEEE, 2005: 15−24

[17] Das S, Mohanty R, Dasgupta P, et al. Synthesis of system verilog assertions[C]//Proc of the Design Automation & Test in Europe Conf. Piscataway, NJ: IEEE, 2006, 2: 1−6

[18] Kastelan I, Krajacevic Z. Synthesizable SystemVerilog assertions as a methodology for SoC[C]//Proc of the 1st IEEE Eastern European Conf on the Engineering of Computer Based Systems. Piscataway, NJ: IEEE, 2009: 120−127

[19] Slang-SystemVerilog Language Services [EB/OL]. (2015) [2023-12-01]. https://github.com /MikePopoloski/slang

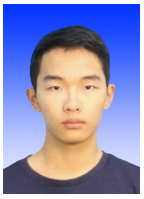
[20] OSCPU. Nutshell processor core[EB/OL]. [2023-04-17]. https://oscpu. gitbook.io/nutshell/jie-shao/introduction (in chinese)
(OSCPU Nutshell(果壳)处理器核 [EB/OL]. [2023-04-17]. https://oscpu. gitbook.io/nutshell/jie-shao/introduction)

[21] Bachrach J, Vo H, Richards B, et al. Chisel: Constructing hardware in a scala embedded language[C]//Proc of the 49th Annual Design Automation Conf. San Francisco, CA: DAC, 2012: 1216−1225

[22] Gal-on S, Levy M. Exploring coremark a benchmark maximizing simplicity and efficacy[R]. The Embedded Microprocessor Benchmark Consortium.2012[2023-12-01].https://www.eembc.org/techlit/articles/coremark—whitepaper.pdf

[23] Weicker R P. Dhrystone: A synthetic systems programming benchmark[J]. Communications of the ACM, 1984, 27(10): 1013−1030

[24] Yu Zihao, Jiang Yanyan. Benchmarks for CPU correctness and performance testing[EB/OL]. 2021 [2023-12-01]. https://github.com/NJU-ProjectN/am-kernels/tree/master /benchmarks/ microbench



Zhang Ziqing, born in 2002. PhD candidate. His main research interests include FPGA, chip verification, and computer architecture.

张子卿, 2002 年生. 博士研究生. 主要研究方向为 FPGA、芯片验证、计算机体系结构.

Shi Kan, born in 1988. PhD, associate professor. His main research interests include agile chip design and verification, FPGA, and computer architecture.

石　侃, 1988 年生, 博士, 副研究员, 主要研究方向为处理器芯片敏捷设计与验证, FPGA、计算机体系结构.

Xu Shuoxiang, born in 1999. Master candidate. His main research interests include FPGA, and agile chip design and verification.

徐烁翔, 1999 年生. 硕士研究生. 主要研究方向为 FPGA、敏捷芯片设计与验证.

Wang Lianghui, born in 2000. Master candidate. His main research interests include FPGA, and chip design and verification.

王梁辉, 2000 年生. 硕士研究生. 主要研究方向为 FPGA、芯片设计与验证.

Bao Yungang, born in 1980. PhD, professor. His main research interests include data-center architecture, agile design methodology of processor chips, and ecosystem of open-source processor chips.

包云岗, 1980 年生. 博士, 研究员. 主要研究方向为数据中心体系结构、处理器芯片敏捷设计方法论、开源处理器芯片生态.