Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation

摘要

半导体行业的验证趋势研究表明，随着设计复杂度的增加，越来越少的公司能够在第一次流片中取得成功，通常需要多次迭代才能投入生产。为了应对这一挑战，公司正在雇佣更多的验证工程师，而硬件设计周期中有 53% 的时间用于设计验证。流片重制的成本非常高，超过 40% 的重制原因是流片后发现了功能性错误。研究还表明，验证工程师有 65% 的时间用于调试、测试创建和仿真。本论文提出了一组用于 RISC-V 处理器验证工程师的工具，这些工具可以在生产前发现更多错误，同时提高调试、测试创建和仿真工作的效率。论文提出了 Logic Fuzzer (LF) [1]，这是一种通过无需创建额外验证测试即可扩展验证空间的新型工具。Logic Fuzzer 随机化设计中不影响功能的状态或控制信号，将处理器的执行引导到正常流程之外，从而增加测试所能覆盖的微架构状态数量。论文还介绍了 Dromajo，这是一种针对 RISC-V 内核的最先进的处理器验证框架。Dromajo 是一个专为协同仿真目的设计的 RV64GC 模拟器，它可以引导 Linux 系统，并能够实时处理外部事件，例如中断和调试请求。同时，它可以以最小的修改量集成到现有的测试平台中。论文将这些工具应用于三个 RISC-V 核心：CVA6、BlackParrot 和 BOOM。Dromajo 本身发现了九个错误，通过与 Logic Fuzzer 的结合，进一步发现了四个错误，且无需创建额外的验证测试。

关键词：微处理器验证；联合仿真；RISC-V

1 引言

当今半导体行业中的验证趋势研究表明，设计复杂性正在增加，成功实现首次硅片的公司越来越少，并且在生产前需要更多的重新设计 (respin)。而为了应对这种挑战，公司开始雇佣更多的验证工程师，硬件设计周期中有 53% 的时间被用于设计验证。重新设计的成本很高，其中超过 40% 的原因是后期发现的功能性错误。在芯片验证过程中，随机指令生成器 (RIG) 和参考模型比较作为最常见的验证工具，已被广泛应用于工业界和学术界。而随着芯片设计的日益复杂，传统的验证方法逐渐暴露出局限性，这催生了更为先进的验证技术和框架，比如协同仿真和形式化验证技术。以下将详细介绍几种常见的验证方法及其应用：

- **随机指令生成器 (RIG):**
  随机指令生成器（RIG）是一种用于生成随机汇编指令流的软件工具，能够广泛覆盖微处理器的功能。RIG 通过指定配置参数来生成测试程序，能够创建复杂的测试用例，这些用例通常很难由工程师手动编写。然而，由于完全随机性，生成的测试用例很难深入到复杂交互的细节，因此可能错过一些深层次的 BUG。

- **参考模型比较:**
  参考模型比较是一种验证技术，通过将设计与一个高层次的软件模型进行对比来检查功能是否正确。参考模型通常是简化的、快速的、与实现细节无关的模型。该方法主要用于比较指令集架构状态在设计和参考模型中的一致性。常见的参考模型都会使用协同仿真的技术：当指令提交时，RTL 模型会向参考模型发送一个提交指令的信号，并比较相关的体系结构状态。如果比较失败，执行会立即停止，并报告导致失败的刺激。这种方法简化了调试工作，因为工程师从最接近分歧点的地方开始调查。

- **形式化验证:**
  形式化验证技术显示出很大的潜力，这种验证方法会穷尽地检查所有可能的执行路径，严格地试图证明或反驳设计模型的正确性。然而，由于可扩展性问题，形式方法通常仅适用于模块化程度较高或设计复杂度适中的设计。对于具有高复杂度的系统，例如现代微处理器，工业界仍然主要依赖动态验证技术或基于仿真的验证方法。

研究显示，由于复杂性增加，成功实现首次流片的公司变得越来越少，而流片失败会导致高昂的重新设计成本（respin）。尽管进行了大量的协同仿真并达成了高覆盖率，但是导致 respin 的 40

- **Logic Fuzzer** 技术通过使处理器的执行流程脱离其正常路径，从而提高在仿真阶段发现边界错误的机会。其关键在于开发能够将处理器带入错误的微架构状态的代码序列，这种状态会导致与黄金模型的体系结构状态不一致。例如，即使重排序缓冲区（ROB）处于未满状态，也可以随机激活其“满”或“阻塞”信号。

- **Dromajo** 为 RISC-V 核心构建了最先进协同仿真框架。能够启动 Linux，处理外部刺激（如中断和调试请求），并能轻松集成到现有测试平台中。同时，Dromajo 也是唯一支持检查点的 RISC-V 验证框架，支持让工程师从模型与 DUT 的分歧点开始检查，从而简化了调试工作。

2 相关工作

在处理器验证领域，有多种工具和方法被提出以提高验证效率和覆盖率。以下总结了与 Dromajo 和 Logic Fuzzer 相关的主要工作：

2.1 Dromajo 的相关工作

- **Imperas 软件平台:**
  Imperas Software Ltd. 是一家商业公司，开发支持多种指令集架构（包括 RISC-V）的虚拟平台。其工具支持逐步对比（step-and-compare）仿真功能。Imperas 的 RISC-V 核心模型在 Apache 2.0 许可下发布，但模拟器通过 OVP 固定平台套件（Fixed Platform Kit）进行授权。与 Dromajo 的主要区别在于，Dromajo 支持检查点功能，并且完全在 Apache 2.0 许可下开放。

- **Herdt 等人提出的工具:**
  lowRISC 为其 32 位 RISC-V 核心 Ibex 提出了验证流。该基础设施在 RTL 实现和参考模型上独立运行目标二进制代码，通过比较执行后的指令流来验证正确性。然而，参考模型与 RTL 之间完全解耦，导致在外部刺激（如中断和调试请求）存在时无法实现逐条指令的实时对比。

- **形式化验证:**
  Herdt 等人的工作提出了一种包含协同仿真的测试基础设施。该方法的创新之处在于将指令流生成器与协同仿真集成到一个包中，能够在运行时生成指令并持续进行协同仿真。然而，与 Ibex 的验证流类似，此方法也无法处理异步刺激。

2.2 Logic Fuzzer 相关工作

- **输入刺激模糊 (Input-stimuli fuzzing):**
  受到软件验证技术的启发，一些工作将模糊测试应用于硬件验证。例如，RFUZZ 将 American Fuzzy Lop 的概念转移到硬件领域; Trippel 等人则将设计转化为软件模型，应用成熟的软件模糊测试技术。PyMTL 框架中的方法采用了假设测试 (Hypothesis Testing)，这种基于属性的测试要求构造必须始终成立的断言，寻找最小的破坏断言的示例。然而，这些方法均采用“外部输入驱动”（outside-in）的方法，而 Logic Fuzzer 采用“内部逻辑扰动”（inside-out）的方式，通过随机化 RTL 逻辑来进行模糊测试。

- **错误注入 (Fault Injection):**
  表面上看，Logic Fuzzer 与故障注入（Fault Injection）方法存在一定相似性。故障注入通过在系统中引入故障并观察行为来分析系统的健壮性。然而，Logic Fuzzer 的核心区别在于其确保插入的逻辑不会对功能产生任何副作用。故障注入旨在分析故障导致的系统崩溃并提出防范措施，而 Logic Fuzzer 则将系统崩溃视为潜在的功能性错误并进行标记。

3 本文方法

3.1 Logic Fuzzer 基本原理

最简单的 Logic Fuzzer 类型是“Congestor”（拥塞器）。如图 1 所示，一个简单的拥塞器示例是将一个与 FIFO 模块的 full 信号相连接的“或”门（OR gate）插入其中。即使 FIFO 的条件尚未满足，full 信号也会被激活。然后，拥塞器会随机激活，导致人工的背压。插入的逻辑会在运行代码时打乱执行顺序。通过插入拥塞器逻辑并运行相同的测试，我们可以证明设计表现出不同的行为，并观察到新的活动。

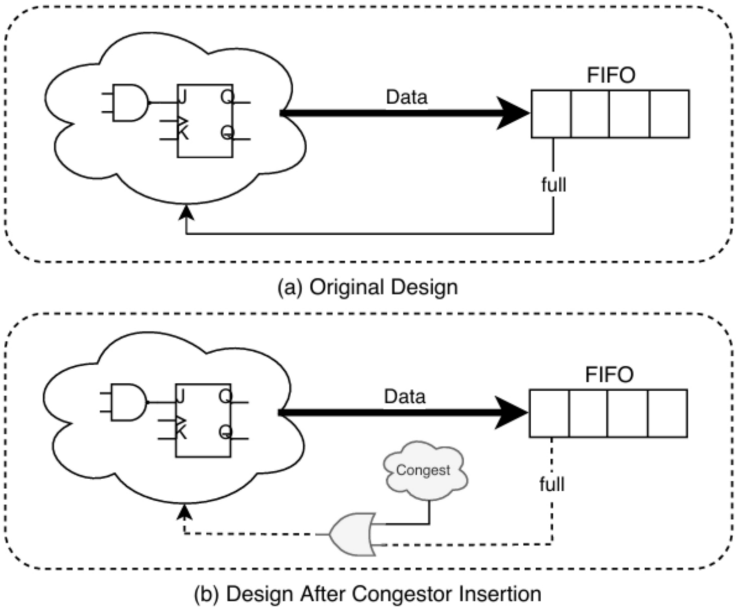

图 1. Congestor

由此，论文将 LogicFuzzer 的概念推广到现代处理器当中，并通过下面三个例子来说明 LogicFuzzer 是如何帮助处理器的验证过程：

3.1.1 Table Mutatators

以 CVA6 核为例，图 2 展示了当运行超过 50 个通过 Google riscv-dv 工具生成的随机测试时，L1 缓存组（way）利用率（仅包含存储操作）。我们进行了三次运行。第一轮（a 行）显示了常规运行，即关闭表变异器的情况。可以看出，基于程序请求的内存位置，CVA6 的方式选择逻辑偏向于选择方式 0。在验证阶段，工程师可能会注意到这一点，并决定通过更改内存请求来增加未充分利用的缓存方式。传统上，工程师会重新生成二进制文件，通过配置随机指令生成器来提供特定的内存请求。但是，这个过程通常非常耗时间，并且需要深入研究缓存替换策略的细节。

然而，加入 Logic Fuzzer 之后，能保证所有缓存中的组都能被覆盖到，如 b 行和 c 行所示。具体做法只需要在 RTL 代码中修改五行代码，并通过 DPI 访问 Table Mutator 类，进而刺激所需的缓存组。

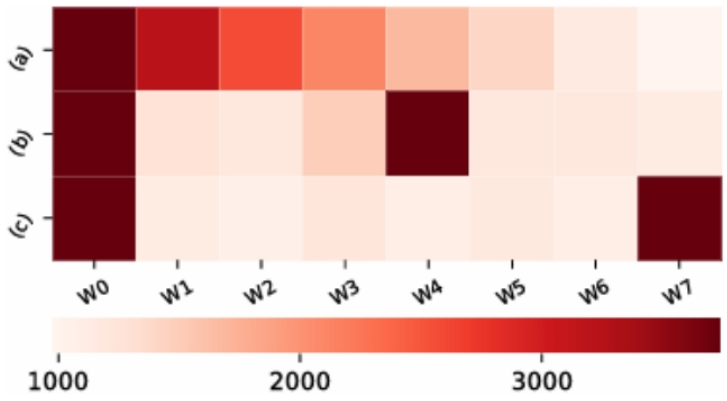

图 2. Table Mutatators

3.1.2 Stressing mispredicted path

现代处理器分支预测技术的进步已经使预测准确度超过了 95

还是以 CVA6 中的误预测路径的指令覆盖率为例，如图三所示，经过超过 200 次测试，分支预测错误路径上的指令覆盖率甚至没有达到 60

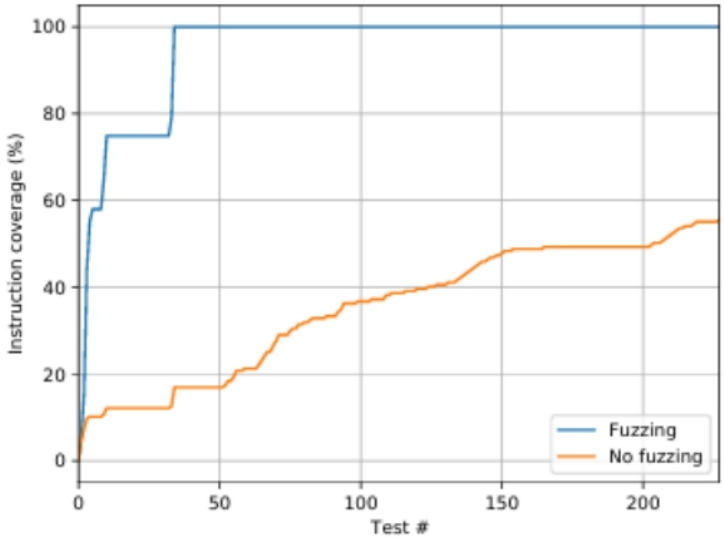

图 3. Stressing mispredicted path

3.1.3 Instruction address ranges in BTB

图 4 是一个散点图，每个数据点表示在给定测试中，分支目标缓冲区（BTB）对程序计数器（PC）地址的预测。红色标记是没有启用模糊测试时的 PC 预测。我们可以看到，预测的地址集中在一个狭窄的范围内。根据设计，BTB 应该基于已解析的分支目标地址历史来提供预测，因此它受限于 ELF 文件中.text 段的地址范围。另一方面，处理器必须足够健壮，能够处理非典型的情况。蓝色圆圈标记是启用了模糊测试后的 BTB 预测结果。我们可以看到，模糊测试能够修改 BTB 条目，提供更广泛的预测地址，甚至在运行时生成随机地址。这些场景可能会导致误预测路径中的 iTLB 页面错误。相同的技术也可以应用于返回地址栈（Return Address Stack）。

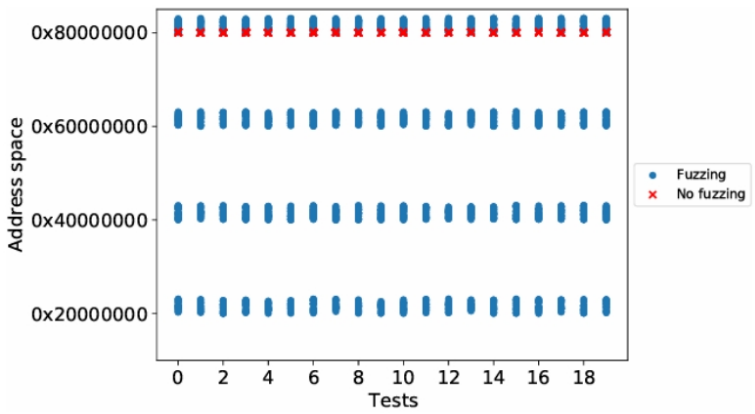

图 4. Instruction address ranges in BTB

3.2 Dromajor 基本原理

Dromajo，一种用于与 RTL 处理器进行协同仿真的仿真器，支持实现 RISC-V RV64GC 指令集的处理器下图 5 展示了使用 Dromajo 实现 RISC-V 核心验证流程的五个步骤。整个流程可以分为两部分：检查点生成（步骤 1、2、3）和协同仿真（步骤 4、5）

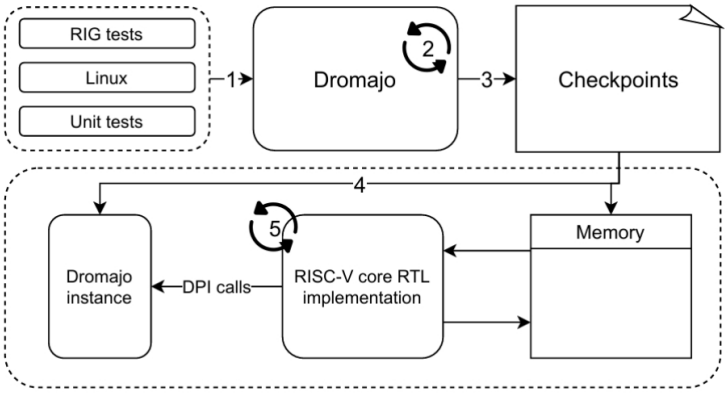

图 5. Test Bench

- **检查点生成:**
  首先，Dromajo 接受任意的 RISC-V ELF 二进制文件（步骤 1）。该流程在使用随机生成的测试时非常高效。然后，我们运行 Dromajo 独立运行（步骤 2），并在一定时间后将整个架构状态保存到检查点（步骤 3）。当仿真开始时，检查点会被加载到两个模型的主内存中（步骤 4）。加入检查点能够允许将长时间运行的程序保存为检查点，以便在仿真中捕获重要阶段，同时也能够支持并行运行，从而加快仿真速度。

- **协同仿真:**
  Dromajo 被编译成一个共享库。将该库链接到仿真器中，并通过 Verilog 的 DPI 调用与 Dromajo 交互。

图 6 描述了 RTL 和 Dromajo 在仿真期间的交互流程。DPI 函数 cosim_init() 在 Verilog 的初始块中被调用。cosim_init() 调用 Dromajo 的初始化 API 函数，传递配置文件路径作为参数，并初始化 Dromajo。该函数返回指向已初始化的 Dromajo RISC-V 参考模型的指针。配置文件包含检查点的路径以及特定核心的信息（如内存映射、处理器参数等）。

DPI 函数 step() 将程序计数器、指令和存储数据传递给 Dromajo。当有效指令提交时，应该调用此函数（步骤 5）。例如，在将 Dromajo 集成到 BOOM 基础设施时，可以通过在 Reorder Buffer 模块中实现简单的监控逻辑来调用 DPI。调用时，Dromajo 会在其一侧提交一条指令，并进行数据对比。如果发生不匹配，函数将返回一个非零代码，仿真会中止。

由于协同仿真是一个同步过程，但中断不是同步的，因此我们需要一种方法来记录核心接收到中断，并强制 Dromajo 内部的控制流执行相同操作。这使我们能够协同仿真中断陷阱处理程序。DPI 包装函数 raise_interrupt() 就是用来做这个的，它传递中断原因并设置 Dromajo 中的陷阱向量。

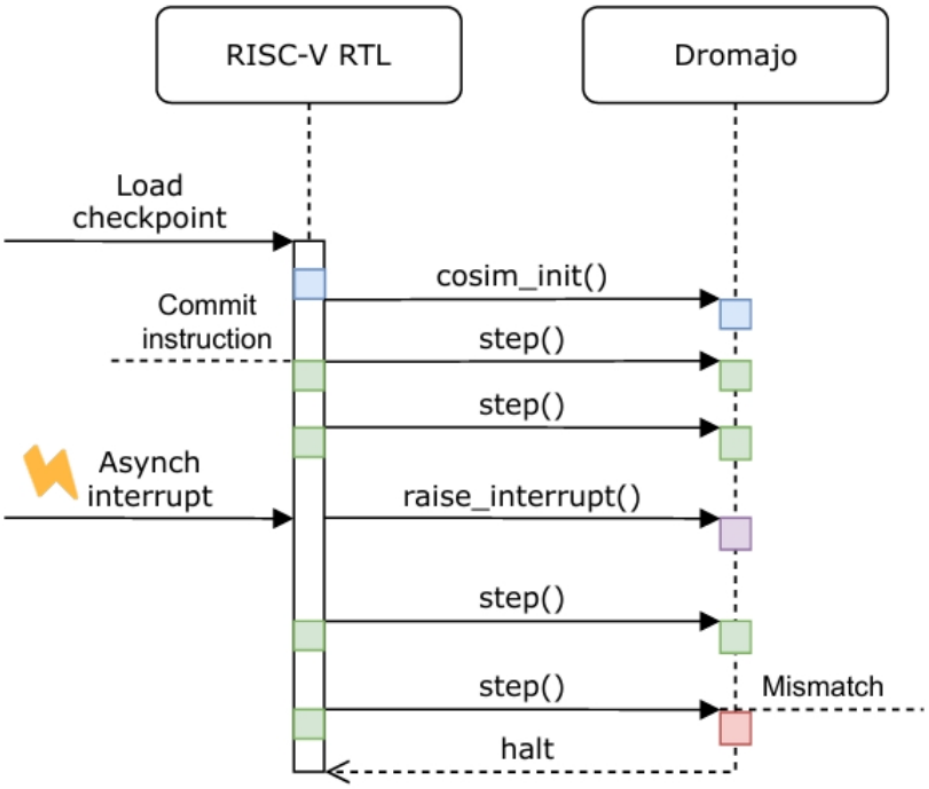

图 6. Co-simulation

4 复现细节

4.1 与已有开源代码对比

原有代码中使用https://github.com/chipsalliance/dromajo作为 co-simulation 框架，使用 chiffre 作为 chisel 代码插装工具，使用单独的 Domajor 环境和添加 Logic Fuzzer 之后的 Domajor 环境作为对照，在三个开源 RISC-V 核中发现的 bug 数量作为最终的评估效果：

- **CVA6:**
  https://github.com/openhwgroup/cva6
- **BlackParrot:**
  https://github.com/black-parrot/black-parrot
- **BOOM:**
  https://github.com/riscv-boom/riscv-boom

复现工作在以上三个 RISC-V 核的基础之上，新增了 RISCV-MINI 核https://github.com/ucb-bar/riscv-mini作为实验的评估目标，并对原有的 RISCV-MINI 核进行手动改动以达到 Logic Fuzzer 的效果，以评估 Dromajo 对 Bug 的查找能力。

4.2 实验环境搭建

使用 CVA6, BlackParrot, BOOM 和 RISCV-MINI 作为被测设计 (DUT), Dromajo 作为参考模型 (golden model), 比较 DUT 和 golden model 再执行每一条指令之后的寄存器和内存状态。每个处理器核的参数信息如下表所示：

Execution	in-order	in-order	out-of-order	in-order
Issue width	1	1	2	1
Extensions	RV64GC	RV64G	RV64GC	RV64G
Priv.modes	M,S,U	M,S,U	M,S,U	M,U
Virt.memory	SV39	SV39	SV39	NULL

Dromajo 是一个包含模拟器和协同仿真库（libdromajo_cosim.a 和对应的头文件 dromajo_cosim.h）的工具。要在 RTL 仿真中使用 Dromajo，首先编译生成静态库和模拟器。然后，通过 Verilator 将 RTL 转换为 C++ 仿真模型，并在测试平台中引入 Dromajo 的协同仿真接口。仿真时，RTL 模型和 Dromajo 模拟器同步执行，通过调用 dromajo_cosim_step 比较两者的寄存器和内存状态，发现潜在的功能性错误。

实验平台如下所示：

操作系统	Ubuntu 20.04.6 LTS
CPU	12th Gen Intel i7-12700
内存	32GB
gcc version	11.5

5 实验结果分析

如下图 6 所示，单独使用 Dromajo 共发现了 9 个 bug。而通过 Logic Fuzzer 增强后的 Dromajo，暴露的 bug 数量增加到了 13 个。需要注意的是，在启用 Logic Fuzzer 时，并未创建额外的测试，而是使用了表 2 中列出的相同测试集来暴露更多的 bug。

Bug ID	Core	Dr*	Dr+LF**	Short description	Reported	Fixed
B1	CVA6	✓		incorrect update of *prv* bits in *dcsr* register	✓	✓
B2	CVA6	✓		incorrect integer division	✓	
B3	CVA6	✓		stval CSR is written on ecall	✓	
B4	CVA6	✓		mtval CSR is written on ecall	✓	
B5	CVA6		✓	incorrect trap cause	✓	
B6	CVA6		✓	arbiter locks with gnt 0	✓	
B7	BlackParrot	✓		integer divide, incorrect handling of sign-extension	✓	✓
B8	BlackParrot	✓		no exception handling on some illegal instructions	✓	✓
B9	BlackParrot	✓		least-significant-bit not cleared on *jalr* instruction	✓	✓
B10	BlackParrot	✓		speculative long latency instructions commit	✓	✓
B11	BlackParrot		✓	core hangs on access to irregular memory region	✓	✓
B12	BlackParrot		✓	backend backpressure breaks instruction ordering	✓	✓
B13	BOOM	✓		incorrect mtval CSR value on traps	✓	✓

图 7. Bugs

6 总结与展望

论文中介绍的 Logic Fuzzer 是一种通过在微架构层面随机化状态，使处理器执行路径超出常规流程的技术。这种方法通过构造非典型场景发现潜在的设计缺陷，而无需生成额外测试。在实际应用中，Logic Fuzzer 的几个变体可以模拟出更多的极端场景，从而帮助设计者在早期阶段发现问题。此外，论文还介绍了 Dromajo，一种先进的 RISC-V 内核验证框架。Dromajo 通过引入检查点机制解决了模拟生产力的瓶颈，能够更高效地验证 RISC-V 处理器的设计。通过对 CVA6、BlackParrot 和 BOOM 的验证，Dromajo 曝露了九个处理器设计错误。结合 Logic Fuzzer 的增强功能，Dromajo 又发现了 CVA6 中的两个额外错误，以及 BlackParrot 中的两个额外错误。不过，在 BOOM 中，Logic Fuzzer 并未发现更多错误，这可能表明 BOOM 的某些设计在错误防范上具有更高的健壮性。

参考文献

[1] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 667–678, 2021.