是主要关心的问题，因为他们不能简单地报告一个错误，让用户通过 Internet 下载代码补丁。即使是简单的逻辑设计错误都可能有很严重的后果，特别是随着微处理器越来越多地用于对我们的生命和健康至关重要的系统的运行中，例如汽车防抱死制动系统、心脏起搏器以及航空控制系统。

简单地模拟设计，运行一些“典型的”程序，不足以用来测试一个系统。相反，全面的测试需要设计一些方法，系统地产生许多测试尽可能多地使用不同指令和指令组合。在创建 Y86-64 处理器的过程中，我们还设计了很多测试脚本，每个脚本都产生出很多不同的测试，运行处理器模拟，并且比较得到的寄存器和内存值和我们 YIS 指令集模拟器产生的值。以下是这些脚本的简要介绍：

optest：运行 49 个不同的 Y86-64 指令测试，具有不同的源和目的寄存器。

jtest：运行 64 个不同的跳转和函数调用指令的测试，具有不同的是否选择分支的组合。

cmtest：运行 28 个不同的条件传送指令的测试，具有不同的控制组合。

htest：运行 600 个不同的数据冒险可能性的测试，具有不同的源和目的的指令的组合，在这些指令对之间有不同数量的 nop 指令。

ctest：测试 22 个不同的控制组合，基于类似 4.5.8 节中我们做的那样的分析。

etest：测试 12 种不同的导致异常的指令和跟在后面可能改变程序员可见状态的指令组合。

这种测试方法的关键思想是我们想要尽量的系统化，生成的测试会创建出不同的可能导致流水线错误的条件。

旁注 形式化地验证我们的设计

即使一个设计通过了广泛的测试，我们也不能保证对于所有可能的程序，它都能正确运行。即使只考虑由短的代码段组成的测试，可以测试的可能的程序的数量也大得难以想象。不过，形式化验证(formal verification)的新方法能够保证有工具能够严格地考虑一个系统所有可能的行为，并确定是否有设计错误。

我们能够形式化验证 Y86-64 处理器较早的一个版本[13]。建立一个框架，比较流水线化的设计 PIPE 和非流水线化的版本 SEQ。也就是，它能够证明对于任意 Y86-64 程序，两个处理器对程序员可见的状态有完全一样的影响。当然，我们的验证器不可能真的运行所有可能的程序，因为这样的程序的数量是无穷大的。相反，它使用了归纳法来证明，表明两个处理器之间在一个周期到一个周期的基础上都是一致的。进行这种分析要求用符号方法(symbolic methods)来推导硬件，在符号方法中，我们认为所有的程序值都是任意的整数，将 ALU 抽象成某种“黑盒子”，根据它的参数计算某个未指定的函数。我们只假设 SEQ 和 PIPE 的 ALU 计算相同的函数。

用控制逻辑的 HCL 描述来产生符号处理器模型的控制逻辑，因此我们能发现 HCL 代码中的问题。能够证明 SEQ 和 PIPE 是完全相同的，也不能保证它们忠实地实现了 Y86-64 指令集体系结构。不过，它能够发现任何由于不正确的流水线设计导致的错误，这是设计错误的主要来源。

在实验中，我们不仅验证了在本章中考虑的 PIPE 版本，还验证了作为家庭作业的几个变种，其中，我们增加了更多的指令，修改了硬件的能力，或是使用了不同的分支预测策略。有趣的是，在所有的设计中，只发现了一个错误，涉及家庭作业 4.58 中描述的变种的答案中的控制组合 B(在 4.5.8 节中讲述的)。这暴露出测试体制中的一个弱点，