CPU的调试

理解本文档需要的相关知识：mips工具链的使用，mips延迟槽的相关知识，spim的使用，对流水线的理解（尤其是冒险），插桩，编译原理的基本块相关知识，程序的运行时环境相关知识。

# CPU的测试

因为cpu的测试需要覆盖大量的指令，而手写汇编指令能覆盖的情况十分的有限，所以在基本指令测试通过之后便应该用工具链生成可执行文件用于测试。

测试方案有2种，一种是通过IDE的仿真测试进行，另一种是直接在开发板上进行测试。我个人认为后者对基本指令的正确性要求很高，如果不能保证bug足够少，用这种测试方法比较困难，因为每一次对程序的修改都要经过综合、实现、产生二进制流等过程，时间消耗大且能获取的信息少。所以我采用了仿真测试的方法进行调试。

在关于工具链的部分中，我们介绍了如何用mips的toolchain生成仿真用的数据（即将二进制程序转化得到的仿真程序可用的文本），测试时只需要用将仿真文件读入ram中即可（仿真时由IDE完成）。

# 错误发现与调试

## 如何发现错误

在暂时没有实现中断和io的情况下，cpu的测试只能通过trap来完成，即编写特定的函数：good trap和bad trap，如果cpu执行正确，那么跳转到good trap，否则跳转到bad trap。

在x86中，trap可以通过内联汇编hlt实现，而在我们的实现中，trap是通过无穷递归的函数实现的，目的就是使程序的pc在hit trap后保持在一定的区间内，方便确认程序是否正确执行。在c程序中写2个无穷递归的函数分别作为good trap和bad trap（在开启了-O2优化选项后循环很容易被优化掉，所以不能使用死循环），然后在正常的代码中插入这两个trap：在程序错误时才可能被执行的分支上插入bad trap，在程序完成执行处插入good trap。

通过工具链得到该程序的汇编代码和仿真程序可以读入的文本。在汇编代码中可以看到2个trap（无穷递归的函数）的PC，然后在仿真的结果中查看pc的值，如果pc的值最终停留在了good trap的范围内则说明该测试样例通过，否则测试样例未通过。

## 如何定位错误

在2.1中讲了如何发现错误，那么出现错误之后如何定位呢？使用我们平时写程序的调试方法：插桩，在原有的程序的各个部分中插入trap，以确认在程序执行的哪一个位置出错，进而找到错误的指令或是引发错误的指令序列。

但是上述方法不总是有效的。CPU的bug调试不同于高级语言程序的调试，在PA中实现过断点，我们是通过将程序指令的第一个字节替换为断点指令实现的，这样做的好处是可以最小程度的影响程序的执行流，在单线程程序的调试中，此方法实现的断点几乎不会影响原程序的执行。但是在cpu的测试中，我们只能通过trap检查程序执行的正确性，而trap本身是通过改变程序的执行流实现的：在一般情况下，我们会检查程序的某个变量的值是否符合预期，如果不符合则跳转到bad trap。然而我们实现的是流水线的cpu，错误的来源可能不是单条指令，而是特定的指令序列造成的冒险。这时，插入trap会导致原本的执行流被改变，使特定的指令序列被打乱，bug无法重现。

（PA中也存在插了判断或Log就屏蔽BUG的情况）

在实验过程中，我们在测试快速排序时遇到了这样的问题。在不插入bad trap的情况下，我们的快速排序会无穷递归；在某个位置插入bad trap之后，bug便不会重新，快速排序能正确执行。这时候bug的定位十分困难：不可能用蛮力去查看每一条指令的信号是否正确；错误没有导致程序挂掉，而是导致程序死循环，难以定位错误发生的位置。

这种情况下我想到了通过比对正常的执行流和异常的执行流来发现bug，让程序正确执行方法是用qemu或者spim执行程序。如果将程序链接到特定的地址上，是可以用qemu运行的，但是目前我没有找到方便的从qemu中获取正常执行流的方法，因为qemu没有提供自动打印pc和指令的命令（类似于NEMU的）。另一个选择则是spim，spim中的step [i]指令与PA1中实现的si [i]类似，可以自动执行一定量的指令，并且输出pc和指令内容。可以将这些输出复制到一个文件中，作为**正确的执行流。**我选择的是使用spim获得正确的执行流，缺点是spim只能执行特定格式的汇编代码，这导致反汇编的代码无法直接在spim上执行，不过这个问题可以解决：我写了一个脚本用于修改gcc生成的汇编代码使其可以在spim上执行。

然后是获得错误的执行流（我们的cpu的执行流），在此之前需要考虑应该比较什么？指令的内容还是PC？由于我们的cpu采用了分支预测技术，所以IF、ID、EX段的指令都可能是错的，而指令的内容只在ID段存在，但是只有执行到MEM段的指令才一定是正确的，所以我们在不修改流水线的情况下无法获得一个正确的指令内容序列。另一个选择是比较PC，这也很麻烦，因为spim使用的地址空间是虚拟地址空间，而我们的cpu使用的是实地址空间，二者的pc不是一一对应的。有一个解决方案是修改存储相关部件，通过简单映射的方式使用虚拟地址空间，但是这个方法的工作量不小，而且由于spim执行的是汇编代码，而不是二进制文件，且spim对延迟槽的处理方法和我们的cpu的处理方法还是不一样，所以我没有选择这个方法。

最后我发现同一个程序，不管在什么情况下，总是相同的一点是：正确的控制流总是会经过同样的基本块序列，即使地址空间是不同的，即使有延迟槽等干扰因素。所以我实现了一个脚本cmp.py，他可以从我们之前得到的spim执行指令的序列和我们从仿真结果中得到的指令序列中得到2者基本块的长度序列。获得的方法比较简单：pc如果不是顺序递增的，那么一定是遇到了分支、跳转，我们认为这是一个基本块的结束；如果遇到一个nop且nop指令的前后两条指令的pc不是相差8，那么认为这是一个基本块的结束。基本块的长度序列有什么用呢？我们知道，不同的基本块的长度可能是相同的，但是长度不同的基本块一定是不同基本块，我们可以利用这一点来定位错误。得到了基本块的长度序列之后，将二者进行比较，找出第一个长度不同的基本块，认为这是cpu的错误导致的控制流的错误，从错误的分支/跳转指令开始往前找。一般的经过优化的指令的基本块都不会太长，这样查找的工作量就被极大的缩小了。

最后我查出我们的快速排序的bug是因为jr指令的load-use冒险没有考虑周全，我是按照上面提到的思路手动找到的。错误发生在第200条指令附近，如果用蛮力去查找该bug是难以想象的。

我上面提到的两个脚本gen\_spim.py和cmp.py在工程的tools目录下，实现的细节涉及到python的相关知识，这里就不便展开了，但是用不同的语言都可以实现，算法是相同的。

# 总结

总的来说，查bug是一件痛苦的事情，在工程设计之初如果可以用软件工程的方法尽可能减少bug是最好的。但是在调试中bug也是不可避免的，这时候应该结合具体情况，应用好自己已经掌握的知识，仔细分析问题，定位问题。在必要的时候学会使用程序工具帮助自己定位问题。