实验报告-PA2

甘晨 181240014

2019 年 11 月 20 日

此次实验未能按时完成，迟于 DDL 提交。

1 实验进程

1.1 PA2.1

反复阅读 PA 讲义，并且 RTFSC，在 PA2.1 阶段大致理解了一条指令的执行过程，以及如何完善一条指令，在完成了几个基本指令后，实现了 dummy 样例的 Hit Good Trap（然而，此时的指令并未完全实现，比如减法指令对标志寄存器的影响，还未完成）。

1.2 PA2.2

经过反复尝试，最终实现了 cputest 测试样例需要的所有指令和库函数（然而，此时有的指令的实现还存在着一些问题未被发现和解决）。

1.3 PA2.3

经过艰难地调试和 debug 过程，终于实现了 printf 的功能，最终实现了各设备功能。

2 必答题

2.1 PA2.1

一条指令的执行过程:

第一步是取指令：由 instr_fetch() 函数取指令，这里主要是取操作码（opcode），通常是取第一个字节以确定指令的操作码。但还有两种方式可以拓展操作码字节：（a）若指令以 0x0f 开头，则需再读取一个字节来确定具体指令；（b）利用 opcode 下字节的

ModR/M, 将其中的 reg/opcode 域读作 opcode 的拓展码以确定具体指令。第二步是指令译码：根据取得的 opcode 作为索引，取得 opcode_table 中的译码辅助函数（包括操作数译码辅助函数）和执行辅助函数，译码的过程实际上是根据指令取操作数，译码过程包括读取内存操作数、寄存器操作数、立即数或跳转地址的。这些获得的译码信息将被保存在结构体 decinfo 之中，decinfo 中会记录操作码（opcode），源操作数（src，src2），目的操作数（dest）以及跳转地址（jmp_pc），这些信息将提供给执行辅助函数使用。第三步是执行指令：执行过程会调用执行辅助函数，根据相应的指令读写做读写内存、寄存器以及跳转等操作，大部分指令执行过程包括对标志寄存器的更新。第四步是更新 pc 的值：原 pc 的值加上指令的长度，则 pc 会指向这一条指令的下一条指令，接下来会重复上面的过程。

2.2 PA2.3

编译与链接:

情形一：同时去掉 *static inline:*

出现了函数重复定义的错误，我认为应该是由于原来的 inline 在展开时，每个调用处都有这样的一段代码，有 static 限定时，其作用域均为所在文件，因而不会发生冲突，但若去掉了 static，就相当于全局定义了多个同样的函数，因而会造成重复定义的错误。

情形二：只去掉 *static:*

没有造成影响，依然可以通过所有 cputest 样例。static 限定了函数的作用域，原本是文件作用域，去掉了 static 相当于函数的作用域变大了，但这并没有给程序执行造成错误，只不过有更多的程序可以调用这一函数了。

情形三：只去掉 *inline:*

出现了函数定义但未使用的 warning，经过查阅资料，我发现内联函数实际是在调用处填上函数体的代码，有点类似于宏定义的展开，利用牺牲空间的方式来减少程序执行过程中不断跳转而带来的时间开销。也就是说在在程序执行过程中并没有按照常规的跳转方式去执行该函数。当我尝试把 inline 去掉再次执行 nemu 时，nemu 没有重新编译，只是重新链接了，因此我认为上一次编译时储产生的被调用函数的副本并没有消失，因而调用处执行的实际上还是被调用函数的函数体，而现在被调用函数本身不再是内联函数，没有跳转到该函数执行的过程从而会报没有使用的错误。

验证想法：

我认为可以另外编写一个小型程序，在几个文件中做一些简单的内联函数调用，按照同样的方法测试上述情况，从而验证想法。

编译与链接:

1: 通过使用 grep 和 wc 命令-> **grep "dummy" -rn |wc -l** 发现有 41 个。

2: 有 42 个，在 common.h 中没加时，有 40 个，加了后有 41 个，在 debug.h 中添加后增加到 42 个。

3: 出现了重定义错误，根据链接过程符号解析多重定义符号的处理规则，未初始化的全局变量名是弱符号（就是 1、2 题中的状态），而若有多个弱符号定义，则在符号解析时会任取其中一个，而函数名和初始化的全局变量名位强符号，强符号只能定义一次。

Makefile:

Makefile 工作方式: 首先利用 make 的-n 选项可以将 make 中需要执行的命令打印出来。同过 RTFM 和回忆蒋炎岩老师习题课的讲解，可以发现，通过 make 可以将一个比较大的项目的编译和链接过程通过一条 make 指令直接完成，但前提是需要在 Makefile 文件中指明待编译文件的关系，还需要指明需要执行的编译指令。结合蒋炎岩老师习题课的讲解，Makefile 文件就是在模拟我们在 shell 中输入指令的过程，通过预先写在 Makefile 中的内容在 Shell 中做输入，再读取执行结果，填到 Makefile 文件相应的位置。然后每次执行 make 指令，都会都会检测哪些源文件发生了变化，继而对发生变化的文件重新编译链接，生成新的可执行文件。

编译链接过程：编译过程: 编译过程是对源程序做词法分析，语法分析，语义分析和代码优化以及存储分配，最终得到的是汇编代码文件 (.s)

链接过程：链接过程是对汇编成的可重定位目标文件 (.o) 进行链接，包括符号解析和重定位两个阶段，其中符号解析是确定符号引用（函数调用和变量使用）之间的关系，重定位是将相关的可重定位文件合并，从而确定函数和变量的地址，再将符号定义的地址填到符号引用处，这就实现了链接过程，按照理论课上所讲的内容，总结起来就是——符号解析——> 同节合并——> 确定地址——> 修改引用。

3 实验心得

3.1 PA2.1

PA2.1 的 PA 讲义真的需要反复阅读，一开始读的一两遍基本上什么也没理解，后来参考了讲义中的建议，开始做了一些笔记，渐渐地才对一条指令的执行过程有了一定的了解。但是，最开始的那几条指令：push，sub 这些着实费了我好大功夫，主要原因主要是以下几点：a）框架代码熟悉成度不够，开始的时候对于一些已有的 rtl 指令还比较陌生，不懂在合适的地方使用这些指令，另外对于给出的译码辅助函数功能也比较陌生，根据指令 i386 手册上的指令描述选择译码辅助函数还比较费劲，其中的缩写和操作数宽度这些概念，到了 PA2.2 实现更多指令时才搞得比较清楚。b）一开始还不懂使用 PA1 中的简易 debug 工具，同时一开始对 gdb 调试工具也不是很熟悉，但是经过 PA2.1 的训练，逐渐掌握了 gdb 工具的基本用法。c）对出错的可能原因没有一个大致的概念，刚开始做 opcode_table 也不太会填，rtl 指令和 make_EHelper 函数的实现也有点一知半解，有时候直接遇到 Hit Bad Trap，或者 pc 跳转到一个明明不是指令起始地址的地方，完全感到一脸懵逼。

3.2 PA2.2

PA2.2 真的是一个噩梦般地过程，实现指令地过程中不断给自己挖坑，然后在之后的过程中不断踩坑。关于 opcode_table 的填写方法，我一开始时在 i386 手册上看具体的指令所在页，后来发现 i386 手册后面有一个指令表，指令表中基本上已经给出可所有指令的填表方式，而且许多相类似地指令在 opcode_table 相对集中的位置，因此在实现了一个指令之后，往往只需要修改一下宽度参数就可以实现一套指令。在一开始，我并不会用 PA2 中的小型调试设施，而 gdb 又没办法查看内存，所以调试起来异常困难，找不到 bug 到底出在哪一步，但由于我做这一阶段时已经落后与整体的进度了，因此，有一位做完 PA2.2 的大佬提醒我可以先去实现 diff-test 来帮助调试，幸好有这一提醒，在加上有大佬指导了如何进入 PA1 中的基础设施，这些东西给我后面的调试提供了不少帮助，至少在找到 bug 所在的地方方便多了，但 debug 的过程依然有很多艰辛。在实现库函数时，由于 string.c 在 C++ 课上基本都练习过，所以实现起来并不难，但在 mem*** 指令实现起来就遇到了一些问题，就是由于 memset 实现的错误导致我过不了相关的一个测试样例。这里我想分享下我是怎么找到是错在 memset 里的，我当时去找一个正确的 string 实现，然后全都 copy 到我的代码下，把我原来的代码都注释掉，然后对每一个函数，利用控制变量法，每次只有一个函数是我的，其他都使用正确的实现，最终找到了这个 bug。后来在习题课上，蒋炎岩老师也给出了类似于这样的调试方法，看来我的想法是对的，这实际上就是 diff-test 的想法，找一个正确的实现来对比。

3.3 PA2.3

PA2.3 最折磨我的地方在于 printf 库函数的实现，而实现这一过程遇到了很多坑，尤其实在实现时钟和跑分测试那些地方，由于一开始实现的 vsprintf 有 bug，而 printf 是调用 vsprintf 的，所以打印过程总是出问题，有的时候调整好了时钟，结果跑分结果莫名其妙的总会在下一个测试的名称之前把上一个测试的得分再打印一遍，但忽略这个 bug，跑分测试都能跑起来，可再回头去看时钟，却发现到了第 83s 之后，忽然变成了 30000+s，然后 pc 跳到了一个不是指令起始地址的地方，这时候调试就出现了一些麻烦，因为此前已经执行了很多循环了，我用单步执行调试工作量太大，无法迅速定位到问题出现的指令周围。后来，在一位大佬的指点下，了解到每次 nemu 执行完都会输出一个执行的指令的条数，我只需要把那一指令条数减去个几十，再执行计算出来的指令条数，就能去定位造成错误的指令。遇到的问题时某一个寄存器的值不对，应该是 0xffffffff，但实际上是 0xff，后来发现这一数据是从栈中取出来的，然后我又去找了 push 这个数据的指令，却发现指令真的是 push 0xff，而且 i386 手册上也写了是 push imm8，也就是说我填的 opcode_table 应该是没问题的，忽然我想到了 PA 手册上好像讲过一个 x86 的坑，赶紧去翻看手册，上面果然写道 push imm8 实际上是需要符号拓展的，于是把译码函数换成 push_SI 就解决了这一问题。这正是说明了认真阅读 PA 讲义的重要啊，当时填 opcode_table 时居然没有在意讲义的提醒。这个问题解决之后就要去处理 printf 的 bug 了，因为在 microbench 跑分会出现多输出一遍上一个样例得分的情况，于是我试着在 native 上跑了一下，结果时 Segmentati fault，我当时感觉问题应该出在 vsprintf，

于是就把原来的代码注释掉了，按照原思路重写了一遍，但并没有解决之一问题，后来又试着重写了 vsprintf 某些部分的代码，但还是没有解决。耽搁了几天后，在和一位大佬讨论的过程中，大佬给我指明了一条相当好的调试方法（前方高能!!!）：首先，新建一个.c 文件，把输入输出库函数包含进来，再把自己写的库函数 copy 进来，用自己写的库函数和标准库函数输出同样的内容，作对比。一开始我也不知道这个调试方法效果究竟怎样，甚至感觉挺麻烦的（狗头保命），但还是试了一下。我发现正常输出%d，%s，%c，这些都没有问题，但要输出限定输出宽度和填充字符就有问题了，我先把 vsprintf 中相关的自定义的函数功能用输出调试法测了一遍，发现没有问题，但自己写的 printf 输出总会时随机字符。经过不断地输出调试，定位到问题在于我储存 out 字符串的数组有问题，这里详细说一下。一开始我初始化字符数组写的时这样：**char *whole = "";** 我认为这样初始化可以生成一个只包含'\0' 的字符数组，然而就是这一段代码导致了 Segmentation fault，于是我将其改为：**char whole[128] = ""** 再次执行，问题就解决了。这应该表示第一种方法没有实现初始化生成'\0'，而我后面调用 strcat 需要循环直到出现'\0'，因而会一直循环下去，而第二种初始化方法就生成了'\0' 符。结果显示，这种对比调试法的效果确实十分显著，使用这种办法我找到 bug 只花了不到 30min，我觉得这也是 diff-test 的思想，真是太厉害了！此外，我在实现 vga 的功能时也遇到了巨大的障碍，一方面是 FPS 之前本该输出数字（upt）的地方总会出现随机字符，但之前曾经有一段时间能输出数字的，我尝试这把 git checkout 回到之前状态下去检查还是没能找到问题，另一方面是屏幕上只能输出当前屏幕的颜色信息而无法输出动画。最后是在大佬的指导下，在 nemu-timer.c 里的时间初始化函数里添加了一段读取启动时间的内容，然后把从端口里读出的内容减去启动时间再赋值给 uptime->lo，这样才正确输出了 FPS。然后在 _DEVREG_VIDEO_FBCTL 里，我本来的代码思路时参考 native 代码，利用 memcpy 的方法把 pixels 里的内容复制到对应位置，但经过测试发现没有动画输出，经过反复失败之后，也是在大佬的指导下，改变为不再利用 memcpy，而是对数组直接赋值的方法，才成功输出了动画效果。

4 鸣谢

由衷感谢给予我帮助和指点的几位大佬，在此表示深深感谢。

5 跑分测试

测试 NEMU 性能：

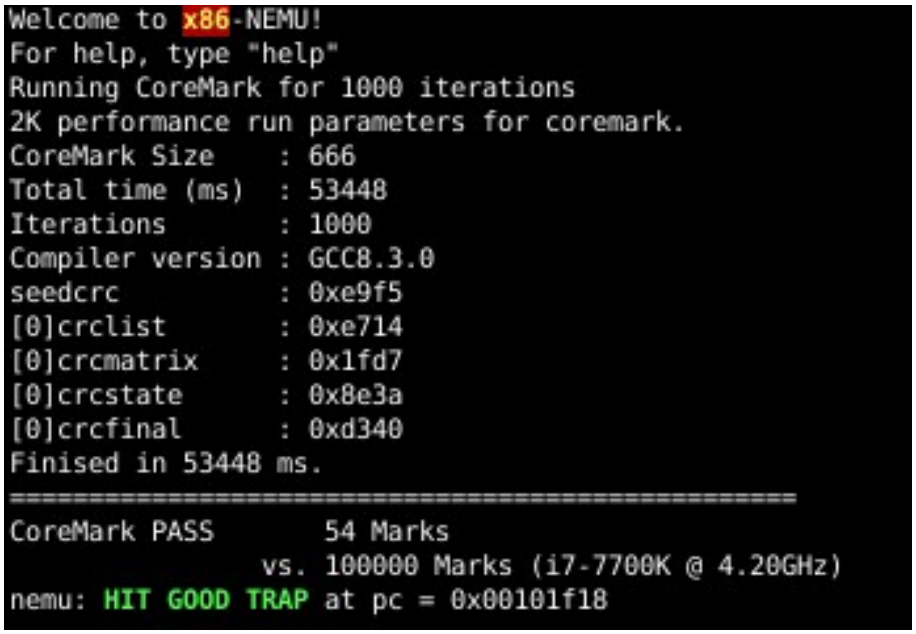

图 1: coremark 测试跑分

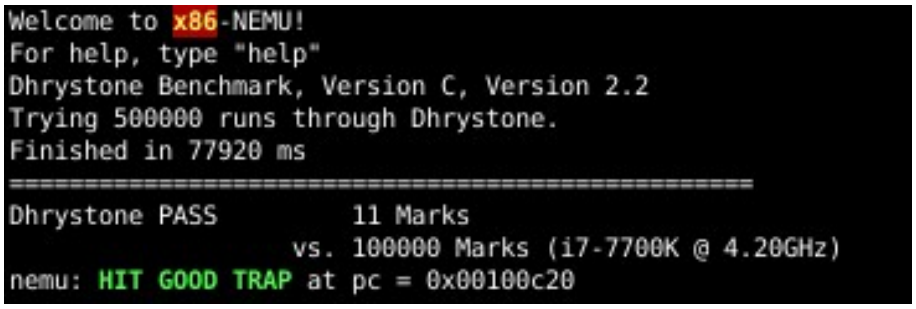

图 2: dhrystone 测试跑分

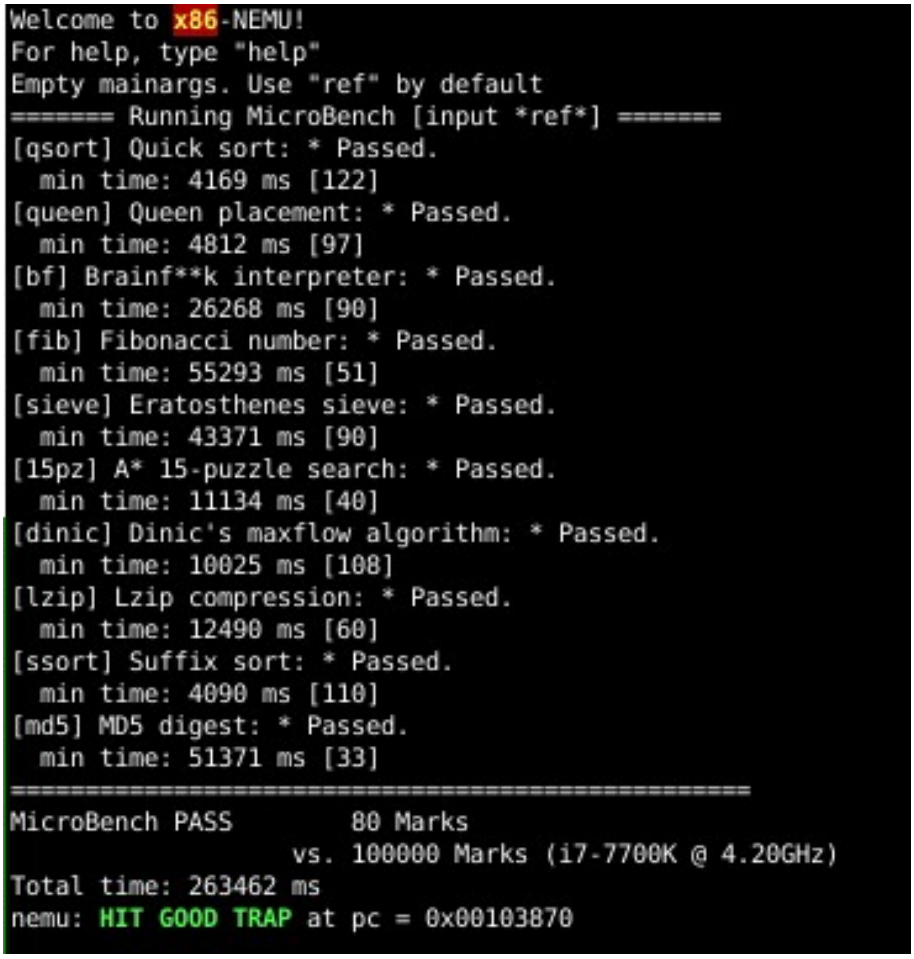

图 3: microbench 测试跑分