实验报告-PA2

甘晨 181240014

2019 年 11 月 19 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，再经过 RTFSC，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。

实现 dummy:

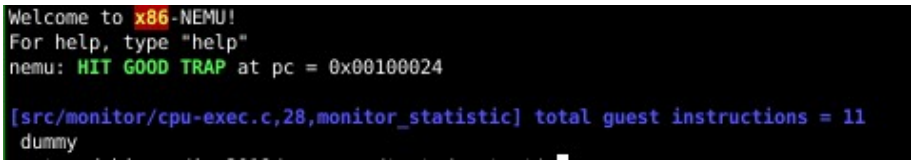

图 1: dummy 样例

2.2 PA2.2

实现更多指令和库函数:

2.3 PA2.3

运行 Hello World:

实现 printf:

实现 IOE:

测试 NEMU 性能:

实现 IOE（2）:

实现 IOE（3）:

目录定位:

Selector 的概念位于 i386 手册的 Chapter5 Memory Management 的 5.1Segment Translation 的 5.1.3Selectors

送分题:

我选择的 ISA 是 x86

理解基础设施:

基于讲义中的假设，450 次调试需要花费 9.375 天从 GDB 中获取并分析信息，另一方面，如果市县实现了调试器，450 次调试需要花费 3.125 天从简易调试器中获取并分析信，也就是说简易调试器可以节约 2/3 的时间。

查阅手册:

CF 位首先表示 CARRY FLAG，是一种状态标志（Status Flag），这一标志受算数指令的影响，在执行算术指令之前，会改变 CF 的值；根据 Apendix C 中的描述，CF 指示高位的借位或进位，有借位或进位位 1（set），否则位 0（clear）。

ModR/M 会出现在操作码（opcode）后面，来确定获得操作数的寻址方式（specify the addressing form to be used)——内存或寄存器. 对于取内存操作数,还通过 ModR/M 来确定地址的计算方式。在 80386 指令集中，以 ModR/M 作为第二个字节的指令很常见。

根据 i386 手册，这里的 MOV 指令时 Intel 格式，不同于 AT&T 格式，其中 MOV A B 中，B 表示源操作数，A 表示目的操作数，与 AT&T 格式恰好相反。其中，源操作数可以使 8/16/32 位寄存器或内存、立即数、段基址和偏移地址做代表的内存中的内容，目的操作数也可以来自上述内容，但不能使立即数。

Shell 指令:

共有 6304 行代码，使用的命令是:（在 nemu/目录下）”find . -name “*.c”|xargs cat |wc -l” 和” find . -name ”*.h”|xargs cat |wc -l “，最后调用之前实现的 p 指令做一下表达式求值即可。

除去空行，共有 5143 行代码，使用的命令是:（在 nemu/目录下）”find . -name “*.c”|xargs cat|grep -v $|wc -l” 和” find . -name ”*.h”|xargs cat|grep -v $|wc -l “.

使用 man:

-Wall 和-Werror 的作用是在编译程序时显示所有警告，并且把 warning 也当做 error 显示出来，正如 PA 讲义中所说的:” 调试是从 failure 回溯 fault 的过程 “，因而尽早的观测到 failure 有可能节省调试的时间，所以说，把 warning 出来的可能出错的地

方在编译时警告显示出来，有可能能避免当发现 failure 时已经很难在回溯的 fault 的情况发生。

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，在加上有大佬知道了如何进入 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，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！

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。


[add-longlong] PASS!
[add] PASS!
[bit] PASS!
[bubble-sort] PASS!
[div] PASS!
[dummy] PASS!
[fact] PASS!
[fib] PASS!
[goldbach] PASS!
[hello-str] PASS!
[if-else] PASS!
[leap-year] PASS!
[load-store] PASS!
[matrix-mul] PASS!
[max] PASS!
[min3] PASS!
[mov-c] PASS!
[movsx] PASS!
[mul-longlong] PASS!
[pascal] PASS!
[prime] PASS!
[quick-sort] PASS!
[recursion] PASS!
[select-sort] PASS!
[shift] PASS!
[shuixianhua] PASS!
[string] PASS!
[sub-longlong] PASS!
[sum] PASS!
[switch] PASS!
[to-lower-case] PASS!
[unalign] PASS!
[wanshu] PASS!
nector@debian:~/ics2019/nemu$

图 2: 一键回归测试

图 3: 实现了 in，out 指令后运行 Hello World

图 4: 实现了 printf 的输出结果

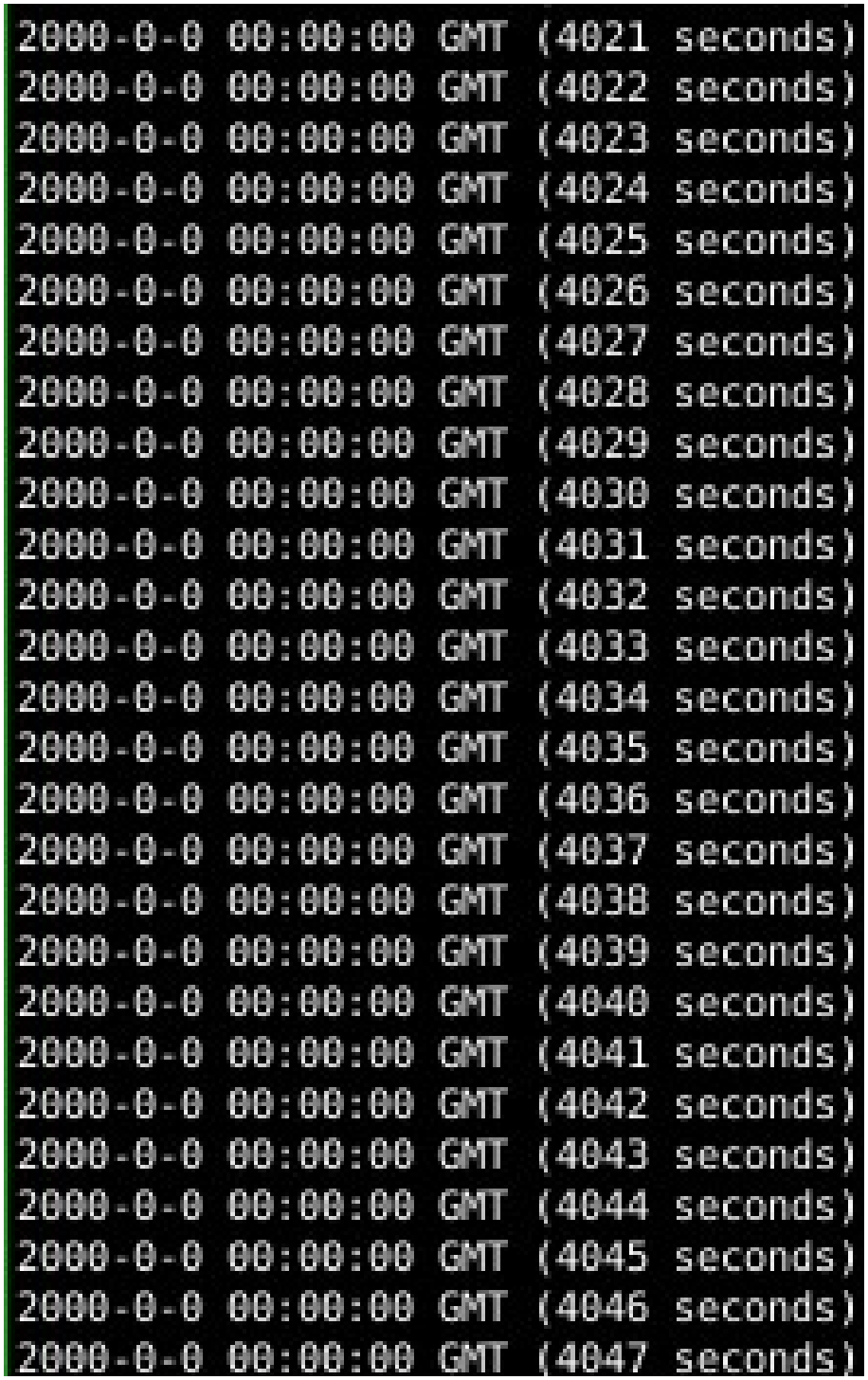
2000-0-0 00:00:00 GMT (4021 seconds)
2000-0-0 00:00:00 GMT (4022 seconds)
2000-0-0 00:00:00 GMT (4023 seconds)
2000-0-0 00:00:00 GMT (4024 seconds)
2000-0-0 00:00:00 GMT (4025 seconds)
2000-0-0 00:00:00 GMT (4026 seconds)
2000-0-0 00:00:00 GMT (4027 seconds)
2000-0-0 00:00:00 GMT (4028 seconds)
2000-0-0 00:00:00 GMT (4029 seconds)
2000-0-0 00:00:00 GMT (4030 seconds)
2000-0-0 00:00:00 GMT (4031 seconds)
2000-0-0 00:00:00 GMT (4032 seconds)
2000-0-0 00:00:00 GMT (4033 seconds)
2000-0-0 00:00:00 GMT (4034 seconds)
2000-0-0 00:00:00 GMT (4035 seconds)
2000-0-0 00:00:00 GMT (4036 seconds)
2000-0-0 00:00:00 GMT (4037 seconds)
2000-0-0 00:00:00 GMT (4038 seconds)
2000-0-0 00:00:00 GMT (4039 seconds)
2000-0-0 00:00:00 GMT (4040 seconds)
2000-0-0 00:00:00 GMT (4041 seconds)
2000-0-0 00:00:00 GMT (4042 seconds)
2000-0-0 00:00:00 GMT (4043 seconds)
2000-0-0 00:00:00 GMT (4044 seconds)
2000-0-0 00:00:00 GMT (4045 seconds)
2000-0-0 00:00:00 GMT (4046 seconds)
2000-0-0 00:00:00 GMT (4047 seconds)

图 5: 实现了时钟设备

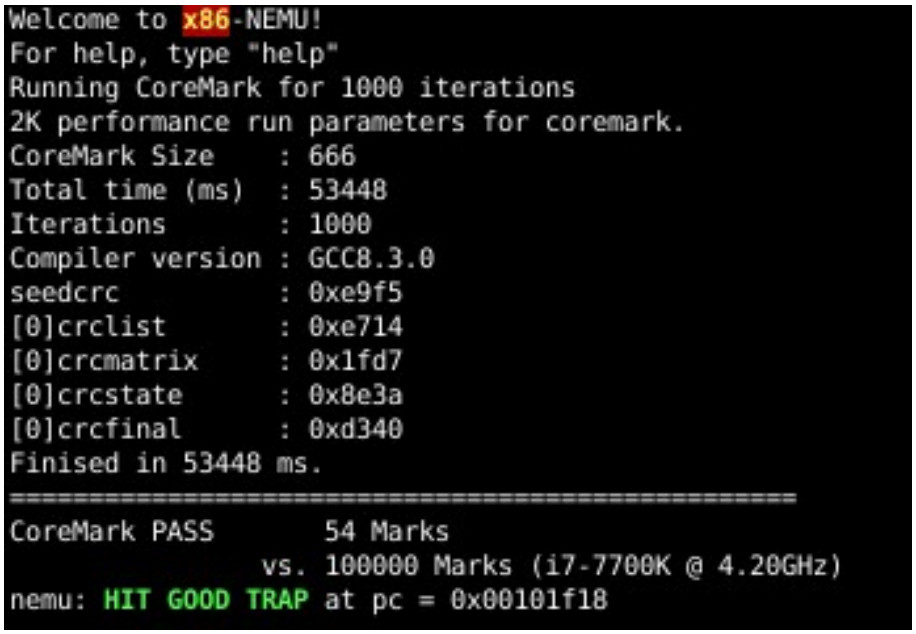

图 6: coremark 测试跑分

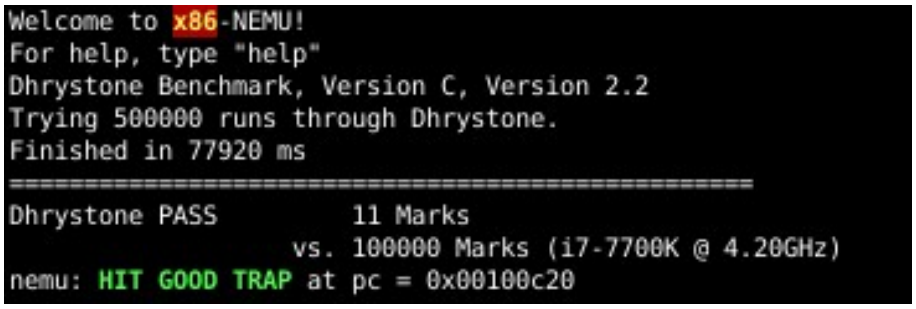

图 7: dhrystone 测试跑分

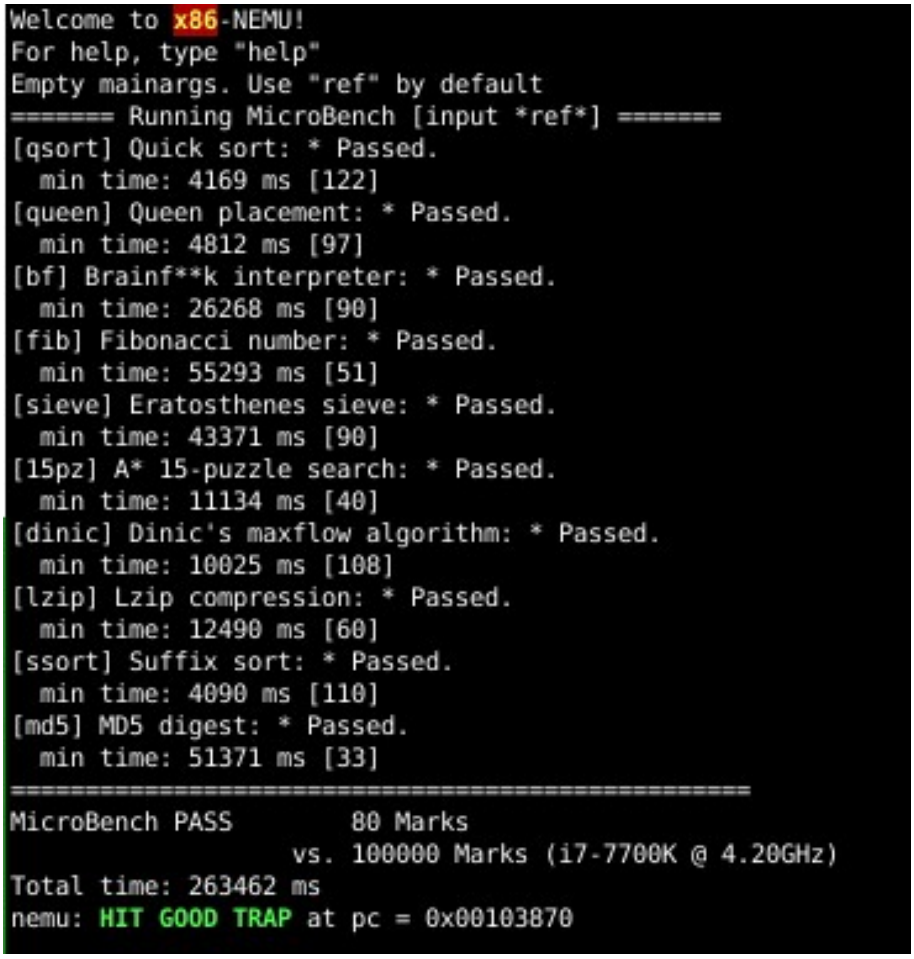

图 8: microbench 测试跑分

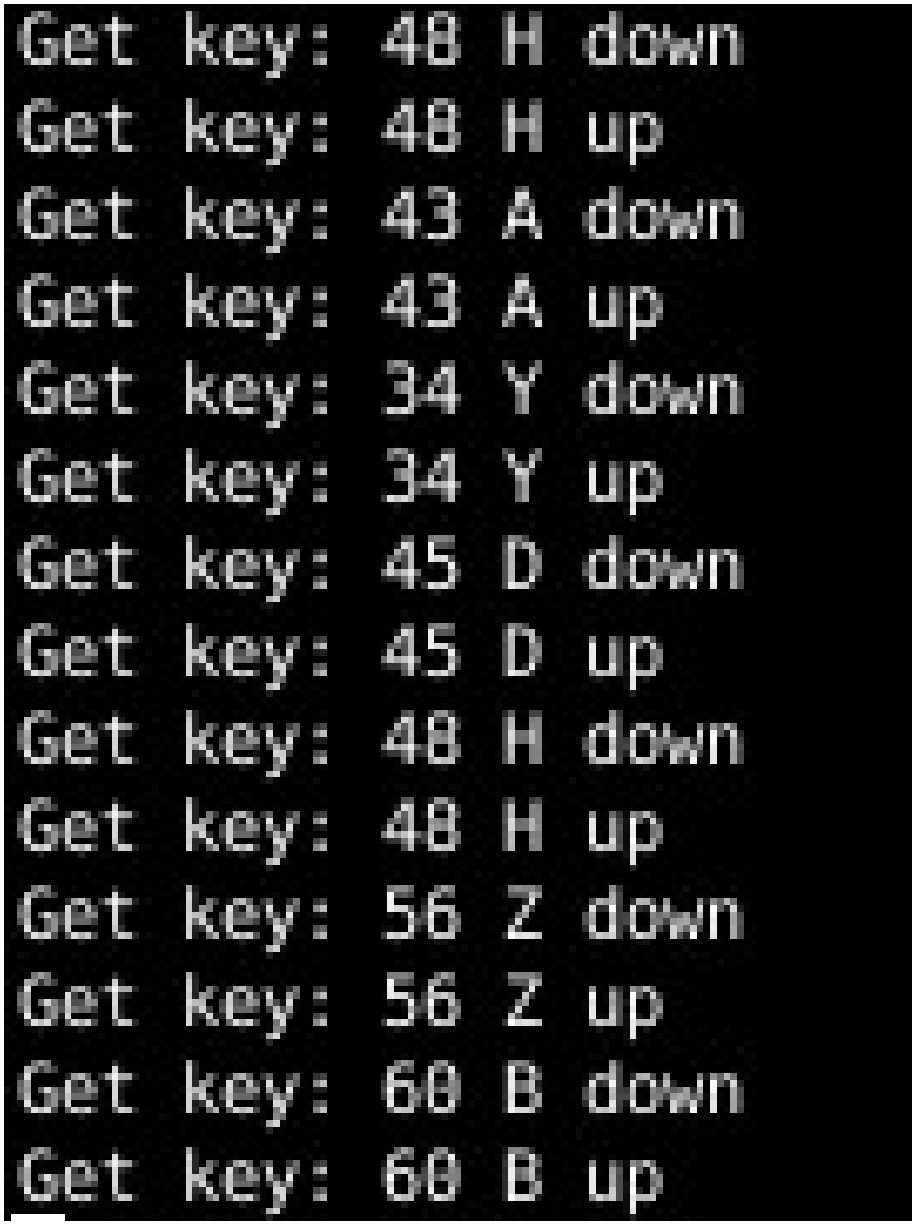

图 9: 实现了键盘设备之后屏幕输出按键信息

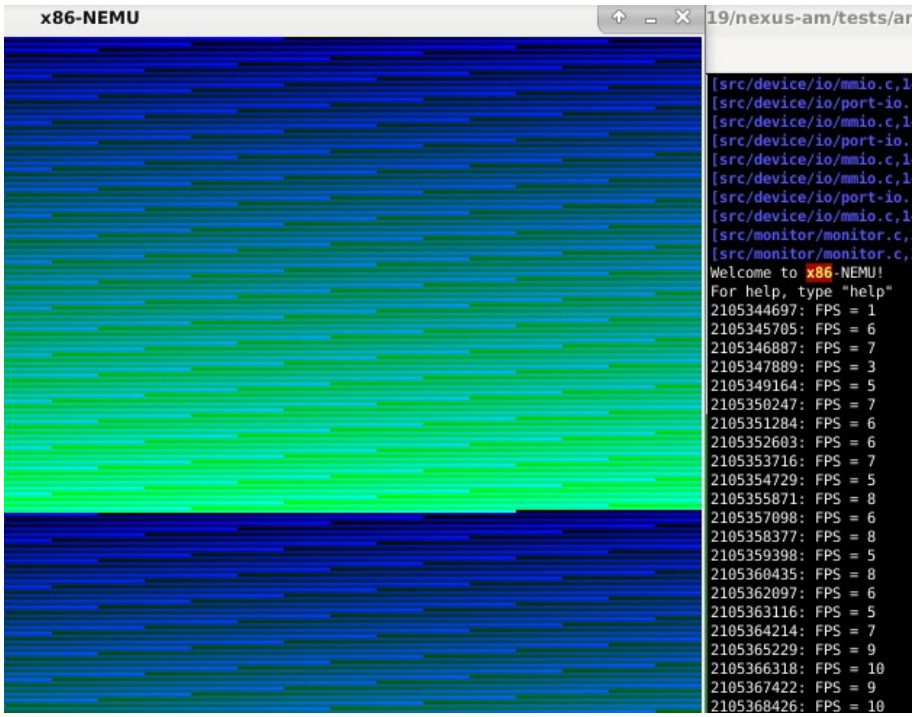

图 10: 实现了屏幕大小和同步寄存器功能和，屏幕输出全屏的颜色信息