PA1 实验报告

韩加瑞

2023 年 11 月 25 日

1 必做题

1.1 理解 YEMU 的执行过程

画出在 YEMU 上执行的加法程序的状态机 由于程序运行过程中只有 PC 和寄存器 R0、R1、内存 M(7) 和 halt 发生变化，因此用五元组 $(PC, R0, R1, M(7), halt)$ 来表示程序的状态。

```
(0,x,x,0,0)       |  R[0] <- M[y]
(1,33,x,0,0)      |  R[1] <- R[0]
(2,33,33,0,0)     |  R[0] <- M[x]
(3,16,33,0,0)     |  R[0] <- R[0] + R[1]
(4,49,33,0,0)     |  M[z] <- R[0]
(5,49,33,49,0)    |  halt=1
(5,49,33,49,1)    |  the end
```

通过 RTFSC 理解 YEMU 如何执行一条指令 YEMU 首先根据操作码判断指令的种类，再根据指令种类对指令进行解码得到相应的地址和寄存器编号，最后对解码得到的内存或寄存器进行指令对应的操作。

1.2 整理一条指令在 NEMU 中的执行过程

在 NEMU 中执行一条指令：在 $exec_once$ 中首先调用 isa_exec_once，在其中调用 $inst_fetch$ 根据 pc 地址和指令长度从内存中取出相应的指令，然后通过 $decode_exec$ 对指令进行解码，得到指令种类，进而得到寄存器编号、

内存地址以及立即数等信息，再根据指令种类对这些内存、寄存器进行相应的操作。

1.3 理解打字小游戏如何运行

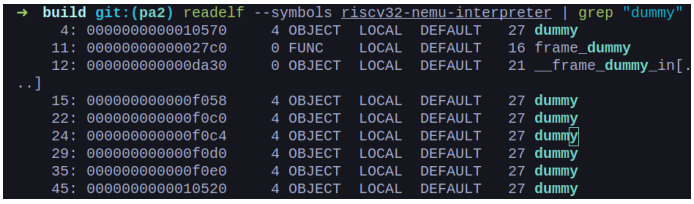
当你按下一个字母并命中的时候，整个计算机系统 (NEMU，ISA，AM，运行时环境，程序) 是如何协同工作，从而让打字小游戏实现出”命中”的游戏效果？ 当我按下一个字母时，首先是 NEMU 中的键盘模块得到我按下的键盘码，之后键盘模块根据键盘码将相应的字符码写入对应的设备寄存器，之后 AM 从对应的设备寄存器中读取字符码并将其发送给打字小程序，打字小程序利用运行时环境提供的函数进行处理，根据这个字符码在当前维护的字符集中寻找高度最低的一个字符并将其速度设为负数。在之后绘制屏幕时，程序根据字符是否命中来决定字符的颜色，并向调用 AM 中的 vgadraw 将之前的字符删去，绘制新的字符，AM 于是向 vga mem 寄存器中更新相应的颜色数据，并在写完后将 vga sync 寄存器设为 1,NEMU 以固定的频率查看 vga sync 寄存器，若为 1 则根据 vga mem 中的数据刷新屏幕，显示命中效果。

1.4 编译与链接 1

在 nemu/include/cpu/ifetch.h 中，你会看到由 static inline 开头定义的 $inst\_fetch()$ 函数．分别尝试去掉 static，去掉 inline 或去掉两者，然后重新进行编译，你可能会看到发生错误．请分别解释为什么这些错误会发生/不发生？你有办法证明你的想法吗？ 结果: 删去 static 或者 inline ，编译都不会发生错误，但如果两个都删去，则会发生错误。

错误原因是由于多个文件声明了 $inst\_fetch$ 函数，这是因为额未加 static 和 inline 时，ifetch 头文件中的代码被多个文件包含，并且每个文件都定义了一个全局的 $inst\_fetch$ 函数，因此导致了重复定义错误。加入 static 后，函数变为局部函数，外部文件无法使用这个函数，因此不会导致重复定义。加入 inline 后，$inst\_fetch$ 为内联函数，编译器会在其调用处将其汇编码展开编译而不为这个函数生成独立的汇编码，不会定义这个函数，因此不会导致重复定义。

1.5 编译与链接 2



在 nemu/include/common.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问重新编译后的 NEMU 含有多少个 dummy 变量的实体? 你是如何得到这个结果的? 重新编译后含有 35 个 dummy 变量实体。

起初我通过 readelf –symbols build/riscv32-nemu-interperter | grep ”dummy” | wc -l 来统计 dummy 的数目，一共是 37 个，但是后来我使用 find build -name ”*.o” | xargs readelf –symbols | grep ”dummy” | wc -l 时发现结果只有 35 个。这是因为有两个 symbol 被误加入了上面的统计中，分别是 $frame_dummy$ 和 $\_\_frame\_dummy\_in$，这两个 symbol 与帧有关，但于 dummy 变量无关。

添加上题中的代码后, 再在 nemu/include/debug.h 中添加一行 volatile static int dummy; 然后重新编译 NEMU. 请问此时的 NEMU 含有多少个 dummy 变量的实体? 与上题中 dummy 变量实体数目进行比较, 并解释本题的结果. 变量实体数目不变，仍为 35 个。

这是因为 common.h 中 include 了 dubug.h, 未初始化的相同名字的局部变量在符号表中只有一个实体。

修改添加的代码, 为两处 dummy 变量进行初始化:volatile static int dummy = 0; 然后重新编译 NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.) 对变量进行初始化之后再编译，会出现 redefinition of dummy 错误。这是因为赋了初值的 static 变量会被认为是一个强符号。C 语言中不允许有两个相同的强符号被定义。而没有赋初值的变量是弱符号，语法上是可以重复定义的。

1.6 了解 Makefile

请描述你在 am-kernels/kernels/hello/目录下敲入 make ARCH=$ISA-nemu 后，make 程序如何组织.c 和.h 文件，最终生成可执行文件 am-kernels/kernels/hello/build/hello-$ISA-nemu.elf. hello 中的 Makefile 首先设置了 SRC 和 NAME，将 hello 目录下所需的源文件包含进来。之后将 AM 中的 Makefile 包含进来，首先将 AM 中所需的源文件加入 SRC 中,并根据隐含规则将 SRC 中的文件编译成.o 文件放进目录 DST_DIR(build) 中，并向目标 OBJ 和 LIB 中添加文件，之后根据 ARCH 将 script 文件夹中的 mk 文件 (如 riscv32-nemu.mk) 包含进来，设置编译和链接参数，然后把我们自己写的库函数 (klib) 打包成 archive，将 SRC 中的源文件编译为可链接文件，之后再链接为 elf 可执行文件。之后通过 *nemu.mk* 文件生成 txt 指令文件和 bin 镜像文件，并设置–image 参数将镜像文件加载到 nemu 的内存中，启动 nemu 执行指令。