PA2实验报告

张运吉 211300063

实验进度

我已完成**pa2全部必做内容**，部分选作内容。

必答题

1. 理解YEMU如何执行程序?

执行加法程序的状态机：(PC, R[0], R[1], R[2], R[3], MEM[0], ..., MEM[15]) -> (PC, R[0] = 0b00100001, R[1], R[2], R[3], MEM[0], ..., MEM[15]) -> (PC, R[0] = 0b00100001, R[1] = 0b00100001, R[2], R[3], MEM[0], ..., MEM[15]) -> (PC, R[0] = 0b00010000, R[1] = 0b00100001, R[2], R[3], MEM[0], ..., MEM[15]) -> (PC, R[0] = 0b00110001, R[1] = 0b00100001, R[2], R[3], MEM[0], ..., MEM[15]) -> (PC, R[0] = 0b00110001, R[1] = 0b00100001, R[2], R[3], MEM[0], ... MEN[7] = 0b00110001, ... MEM[15])
如何执行：首先从内存取出PC指向地址的指令，然后根据前四比特位对操作码译码，再根据译码的结果进行操作数译码，然后执行指令对应的功能，最后pc++更新pc的值，当取出的指令操作码译码不成功时，设置halt为1,并退出指令执行的循环，具体地，在YEMU中执行到pc=6时，将会退出循环。

2. 请整理一条指令在NEMU中的执行过程.

在NEMU中，一条指令的执行被封装在 `isa_exec_once()` 中

取指： `isa_exec_once()` 首先调用 `inst_fetch()` 取指令，并把指令存在s->isa.inst.val中，并更新s->snpc的值。

译码和执行：接下来进入 `decode_exec()` 函数，使用模式匹配的方式对指令进行译码，具体地，先调用 `pattern_decode()` 函数来获取指令的类别，然后根据指令类型调用 `decode_operand()` 函数，这个函数将会根据指令类型的不同从指令中抽取出相应的操作数(进行重排列和拓展)，然后执行相应的功能。

执行：译码阶段结束之后, 代码将会执行模式匹配规则中指定的指令执行操作, 这部分操作会用到译码的结果, 并通过C代码来模拟指令执行的真正行为。

更新pc：把s->dnpc赋值cpu.pc。

3.理解打字小游戏如何运行

`main()` 函数中，先调用 `ioe_init()` 初始化ioe，然后调用 `video_init()` 初始化屏幕，设置背景颜色为紫色。在while游戏主循环中，先根据设置的fps值得到刷新频率，之后调用 `game_logic_update()` 对现存的字母位置进行刷新，刷新的新位置由当前位置和速度计算得到，对于超出的屏幕的字符，使其停留一段时间，停留时间已到的字符，令其消失；之后来到一个while循环获取用户键入的按键，由 `check_hit()` 函数检查键入的按键是否正确，若正确成功数量增加，使字符反向返回。最后调用 `render()` 对屏幕进行重新渲染，打印当前的正确核错误信息。

打字小游戏在riscv32-nemu上运行，由nemu计算出屏幕上每个位置的颜色，然后使用am的软件接口把数据输入到端口，然后nemu从接口获取信息将输出反馈在屏幕上。

4.编译与链接1

static inline都去掉：编译报错，原因在于重复定义，因为在hostcall.c和inst.c同时includ了ifetch.h, 会使得同一个函数被多次定义。
去掉static编译正常，因为加了static后限制了inst_fetch的作用域为本文件，所以不会产生重复定义的问题。
去掉inline编译正常，原因在于加上inline之后，编译器会将函数解释成內联函数，在调用这个函数的时候把函数体的代码拷贝到相应位置，而不会生成inst_fetch函数对象，也不存在函数重复定义的问题。

5.编译与链接2

1）含有36个dummy变量的实体，使用shell命令 `find -name "*.o" | xargs nm -A 2>/dev/null | grep "dummy" -c` 查找编译生存的.o文件中"dummy"字符串的出现次数。
2) 同样是36个，debug.h中include了common.h，common.h中也include了debug.h，未初始化的dummy是弱符号，在编译时同时出现2次的话只会编译一次，所以dummy实体的变量个数没有增加。
3) 编译会报错，初始化后的dummy是强符号，链接的时候不允许不同文件存在同名的强符号，所以触发编译错误。

6. 了解Makefile

Makefile的基本规则是：
target... (目标文件): prerequisites ...（依赖文件）
    command（执行指令）

```
$(DST_DIR)/%.o: %.c
	@mkdir -p $(dir $@) && echo + CC $<
	@$(CC) -std=gnu11 $(CFLAGS) -c -o $@ $(realpath $<)
```

这部分是使用gcc编译的基本规则，其中的%匹配符表示的是相同的文件名，编译的时候先创建一个文件夹，然后将生成的文件放在这个文件夹下，其中一些符号的意思是：$@表示目标文件，$<表示第一个依赖文件。

$(CFLAGS)是一些编译选项，定义如下

```
CFLAGS   += -O2 -MMD -Wall -Werror $(INCFLAGS) \
            -D__ISA__=\"$(ISA)\" -D__ISA_$(shell echo $(ISA) | tr a-z A-Z)__ \
            -D__ARCH__=$(ARCH) -D__ARCH_$(shell echo $(ARCH) | tr a-z A-Z | tr - _) \
            -D__PLATFORM__=$(PLATFORM) -D__PLATFORM_$(shell echo $(PLATFORM) | tr a-
            -DARCH_H=\"arch/$(ARCH).h\" \
            -fno-asynchronous-unwind-tables -fno-builtin -fno-stack-protector \
            -Wno-main -U_FORTIFY_SOURCE
```

其中的$(INCFLAGS)指出.h文件的位置。可以看出，一个.o文件的生成由对应的.c文件及CFLAGS中的编译选项完成。

然后是链接，链接是把所有的文件都链接起来，最终生成.elf文件。

```
$(IMAGE).elf: $(OBJS) am $(LIBS)
	@echo + LD "->" $(IMAGE_REL).elf
	@$(LD) $(LDFLAGS) -o $(IMAGE).elf --start-group $(LINKAGE) --end-group
```

总之，在使用make编译的时候，make程序会根据依赖关系寻找最终生成的.elf文件的依赖文件，具体地，会先寻找$(OBJS)和$(LIBS)中的文件，如果没有就编译生成，最后把这些文件链接一起生成最终的可执行文件。

实验体会

1. pa2.1要求在nemu/src/isa/$ISA/inst.c中添加指令的正确的模式匹配规则，这一部分的难点我觉得在于结合讲义RTFSC的过程，因为涉及到很多的宏定义，看的时候真的会被各种嵌套的宏给绕晕，不过宏的本质类似于粘贴，所以只要运用讲义上讲的如何阅读宏的一些方法，慢慢阅读源代码还是能看的懂的；添加模式匹配规则的时候主要是要RTFM，根据官方手册的讲解理解每一条指令的功能之后，剩下的就是重复性的工作了。
2. 基础设施：基础设施部分最难的部分莫过于ftrace了，看到这个必做内容的时候真的是一头雾水，只能上网搜索elf文件格式，感觉懂了之后开始写相关代码，但写完之后还是bug重重，最后发现是因为我使用malloc给局部指针类型变量分配空间但是没有free，成功读出elf文件后，在实现ftrace的时候还遇到一个问题就是如何在jal和jalr中识别出return和call指令，一开始我的想法就是在执行每一条指令的时候根据模式识别的结果判断是不是jal或jalr指令，是的话再判断

dnpc是不是某个函数的起始地址，若是的话就是一个call指令，否则判断dnpc是不是在某个函数内，若是则是return指令，但是由于尾递归调用的指令反汇编是jr(是一种伪指令，是用jal或jalr实现的)，我先前的判断会把jr指令也算成一种call指令，导致输出的结果和讲义的不一样，后来结合反汇编代码才发现这个问题，然后才顺利解决。

3. pa2.3部分整体做完给我的感觉就是代码量不多，但是需要阅读的源代码很多，读懂了才知道该往哪里添加什么样的代码，还要读相关的makefile文件才知道怎么进行一些必要的测试。时钟相关的代码貌似就只要添加一行，但是要注意大小端的问题，然后困扰我的就是跑分环节，尽管我关掉了各种trace之后还是跑得很慢(只有10分),经过各种控制变量法的排除之后，无可奈何我让我舍友在他的电脑上跑我的nemu发现是正常的(300多分)，然后在我的电脑上跑我舍友正常的nemu也跑的很慢，由此我感觉可能和电脑的处理器有关(我另一个舍友电脑和我是同一种处理器的也遇到这个问题)。(感觉很奇怪，想不明白为什么会这样？)
4. 实现键盘：在实现键盘功能的时候，我借鉴了native的代码，其中关键的是这两行代码:
   kbd->keydown = (key_code & KEYDOWN_MASK ? true : false);
   kbd->keycode = key_code & ~KEYDOWN_MASK;
5. VGA：实现vga困扰我比较旧的是实现 `__am_gpu_fbdraw()` 函数的时候，一开始我不知道如何向相关寄存器中写入绘图信息，后来通过STFW后明白了像素的工作原理，然后结合讲义才弄明白了这一部分该如何写。
6. 总结：整个pa2给我的感觉是比前两次pa都难了很多，不仅代码量变多，而且要求我们对框架代码熟悉并理解，如此一来才能完成pa2。在pa2中，我也学到了很多东西，比如一条指令在计算机中究竟如何执行，以及cpu如何实现io硬件的功能，总而言之，对整个计算机系统的硬件层有了更加深刻的认识。