PA2 实验报告

赵超懿 匡亚明学院 191870271

实验进度

我已完成PA2 **全部必做内容** 以及选做malloc实现并运行超级玛丽

必答题：

1.理解YEMU如何让执行程序

执行加法程序的状态机：

在执行加法指令时：不会改变内存的状态，只改变pc和寄存器的值。

(pc,R[0],R[1],R[2],R[3],内存) --> (pc+1, R[0]+R[1],R[1],R[2],R[3],内存)

pc加1指pc移动到下一个字节。初始R[0],R[1],R[2],R[3]的四个位置分别代表寄存器0，1，2，3，图中假设加法指令rt=0,rs= 1,则R[0]位置上的指变为R[0]+R[1]。若使用rs核rt来表示，即rt位置上寄存器的值变为R[rt]+R[rs]。加法指令不会改变内存上的值，故内存不会变化。

YEMU执行指令的过程：首先从存储的程序中取得指令，然后根据前4为来判断指令类型，然后对操作数进行译码，执行对应的操作，同时在译码过程中，pc指向下一条指令。

将程序看作状态机，其下一个状态既和当前的状态有关，也和输入即指令有关。加法指令是YEMU执行过程中一个特例。

比如mov指令 MOV reg32, imm32 ，其操作码为0xb8+rd，后跟有+rd，代表后面指令的译码结果是32位的寄存器,并根据rd得到目标寄存器，然后会有4字节的立即数。在nemu中，在exec_once中首先获取opcode位0xb8，switch语句定位到IDEX(0xb8+rd,ov_I2r,mov)，然后根据宏定义，先设置位宽位4字节（没有0x66开头的情况下），然后调用def_DHelper(mov_I2r)进行译码，mov_I2r的译码先调用decode_op_r译码得到寄存器的编号，然后处理好指针ddest指向目标寄存器。然后调用decode_op_I译码得到4位立即数，即第二个操作数，处理好指针idsrc1，然后为执行部分，调用def_EHelper（mov）实现指令。最后返回更新pc，进入下一次循环。

3.程序如何运行：理解打字小游戏如何运行。

答：main开始先进行ioe和video的初始化。

ioe_init检测设备是否实现，video_init对屏幕初始化，背景设为紫色，对字母设置颜色。在循环中，根据fps确定刷新频率，game_logic_updata(current)对现存的所有字符的位置进行刷新，刷新的新位置由当前位置核速度计算得到，对于超出的屏幕的字符，统计错的字符数量，使其停留一段时间。对于时间已到的字符，令其消失。再有一个while循换获取键盘的输入，判断是否退出游戏还是不改变还是使字符消失，check_hit函数，判断是否输入正确，若正确成功数量增加，使字符反向返回。最后的render对屏幕先刷新，再对字符的现在的位置进行染色，输出到端口，打印当前的正确核错误信息。

在整个运行过程中，打字小游戏在以x86为架构的nemu运行，使用x86架构的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.在`nemu/include/rtl/rtl.h`中, 你会看到由`static inline`开头定义的各种RTL指令函数. 选择其中一个函数, 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?

static inline都去掉：编译报错，原因在于重复定义，因为在exec.h中使用include包含了rtl.h,会使得同一个函数被多次定义（既在exec.h中，也在rtl.h中）

去掉static产生的结果也是重复定义函数，原因和上面的相同。

只去掉inline 编译器会发出警告，提示function defined but not used，但是由于Makefile中-Wall 和-Werror，会报错。在decode.c编译时会定义而不被使用。原因在于inline会把函数放到调用者内，而不是把函数放置等待调用，故不会出现函数被定义而不被使用的情况。

编译与链接

1. 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?

   27个，在编译后得到的文件中使用grep搜索dummy的.o文件，共27个；

   shell grep -r -e 'dummy' |wc -l，这样得到是文件中含有dummy的文件的数量，如果在grep后加上-c，可以得到总数为54,但是由于dummy这样的定义为弱符号，每个文件中最后会随机挑选一个编译，故最终数量仍然为27个。

2. 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.

   27个，和第一问结果相同。原因在于common.h中包含了debug.h，所以当common.h被include时，debug.h也被include。

3. 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.)

   报错：重复定义。原因在于dug.h包含在common.h，这样会对dummy进行两次定义（初始化后即为强符号），所以报错。

了解Makefile 请描述你在`nemu/`目录下敲入`make` 后, `make`程序如何组织.c和.h文件, 最终生成可执行文件`nemu/build/$ISA-nemu`.

答：格式 file : file 1,file 2,... file 3表示文件file由后面的文件构成,**省略file文件本身**

然后省略了common.h（不直接include的）, 以及common.h include的debug.h 和macro.h.

在执行编译时,起作用的主要是这里

```
$(OBJ_DIR)/%.o: src/%.c

  @echo + CC $<

  @mkdir -p $(dir $@)

  @$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
```

一个.o文件由对应的.c文件及CFLAGS中的编译选项完成

```
INCLUDES  = $(addprefix -I, $(INC_DIR))

CFLAGS   += -O2 -MMD -Wall -Werror -ggdb3 $(INCLUDES) \

      -D__ENGINE_$(ENGINE)__ \

      -D__ISA__=$(ISA) -D__ISA_$(ISA)__ -D_ISA_H_=\"isa/$(ISA).h\"
```

可以看到 $(INCLUDES)给出了要生成.o文件需要的头文件的路径（标准库中的不需要写出），这样是使用gcc的过程中，编译器就可以找到对应的头文件，

`gcc g-O2 -MMD -Wall -Werror -ggdb3 -I./include -I./src/engine/interpreter g-gdgd -c -o build/obj-x86-interpreter/device/alarm.o */*.c`这样的格式，便可以生成.o文件，最后是文件的链接；

```
gcc -O2 -rdynamic  -o build/x86-nemu-interpreter

build/obj-x86-interpreter/device/alarm.o

build/obj-x86-interpreter/device/audio.o

.............(省略)

build/obj-x86-interpreter/engine/interpreter/init.o -lSDL2 -lreadline -ldl
```

链接过程将所有文件链接起来。

makefile中还提供gdb，和run操作，分别对应shell中的gdb和执行二进制文件

整个过程中，gcc -I -I./include -I./src/engine/interpreter,从这两个路径搜寻.h文件，以下是详细包含（直接include的，间接包含的会在-I后的路径去寻找）的文件

alarm.c : common.h time.h signal.h

audio.c: 未实现的情况下为 common.h

serial.c ： map.h

keyboard.c: map.h monitor.h SDL.h

intr.c: isa.h

device.c: common.h

vga.c: comon.h

map.c: isa.h paddr.h vaddr.h map.h

mmio.c: map.h

port-io.c: map.h

monitor.c: isa.h paddr.h monitor.h getopt.h stdlib.h

log.c: common.h stdarg.h

ui.c: isa.h expr.h watchpoint.h stdlib.h readline.h history.h monitor.h paddr.h

expr.c: isa.h regex.h paddr.h

watchpoint.c: watchpoint.h expr.h malloc.h

cpu-exec.c: isa.h monitor.h difftest.h stdlib.h time.h watchpoint.h

dut.c: dlfcn.h isa.h paddr.h monitor.h

paddr.c: isa.h paddr.h vaddr.h map.h stdlib.h time.h

main.c: stdio.h stdlib.h string.h

reg.c: isa.h stdlib.h time.h reg.h

decode.c: exec.h rtl.h reg.h decode.h

logo.c: 无

special.c: exec.h monitor.h difftest.h

exec.c : exec.h decode.h

intr.c: exec.h rtl.h

mmu.c ； isa.g vaddr.h paddr.h

实验感受

PA2的这部分也是写完了才返回来写的。

首先是刚刚进入PA2时的感受，我整个人都不好了，看了一天的框架代码，也没完全搞懂整个运作机制（我太难了），只能继续看了，然后学习就这样痛苦了半周，才差不多弄明白nemu的运行。然后发现其实页不是很高深，就是自己的看框架代码的能力太弱了。

好了，可以通过cpu-test来写nemu的代码了，殊不知真正的痛苦才刚刚开始。（天真.jpg）在写代码的过程中，我逐渐体会到不严格按照手册的后果，一会挂这个一会挂另一个，这个人都傻了。（在写完test之前还没有看difftest，看到后是真的后悔没先看一遍PA2的讲义）

第一个调了很长时间才调好的是CF和OF，通过强制类型转换，我才解决了这个问题(之前写代码基本没用过，在PA中很快就学会了)。然后啪的一下，很快cpu test都写完了，但是有一两个错了，这便是第二个问题，nemu中的规约使用出问题了，这个问题比较隐蔽，是另一个rtl函数调用了sext函数（也是rtl），然后两个函数都使用了t0寄存器，直接挂掉，总之，PA2中的问题都可以归结为RTFM，确实不好好按照规约来写害人不浅。

在写完指令后我就到了快了的环节，首先是快乐的跑分环节，当然在跑分过程中我借助difftest找到很多bug，比如指令位宽等等，不过解决起来还是很快的（体会到了真机的好处，可以跑KVM并且性能极佳，difftest可以到50W条指令每秒，比qemu好太多）然后就是没事跑一跑分，实现了malloc打一打超级马里奥。然后就到了PA2.3

2.3中时钟就卡了好久，最后**感谢贾林杰同学**的提示，实现时钟，后面的就很好解决了，不过中间还有一个错误提醒了我这是一个小端机，在以后的pa中我时刻不敢忘（这个错误出现在VGA读取屏幕长和宽时，由于小端机刚好读反，最后通过打印调试法才解决，在之后的PA3和PA4涉及需要自己读取内存时，时刻注意到这是一个小端机）然后PA2就顺利结束了。

总结

PA2中在写指令时，我深陷bug之中，那是真的感到绝望，不过很庆幸，最后我凭借自己走了出来，在这其中，我深刻体会手册的重要性，也十分佩服编写手册的人，居然可以让如此复杂的计算机精密运转而不出错（tql）。

然后是在PA2中学到的知识，首先明白了指令执行的整个过程，然后学会了基本的调试方法，对整个计算机系统的硬件层有了更加深刻的认识。