PA1 第一阶段 实验报告

实验进度：
完成 PA1 第一阶段的实验：实现 make run; 单步打印命令 si n; 读取寄存器内容 info r; 初步的扫描内存 x N epxr

必答题：

1 阅读代码框架

ui_mainloop()
此函数负责处理用户每次输入的命令。

```
char *str = rl_gets(); // 获取用户的输入命令

char *cmd = strtok(str, " "); // 将 char * 命令中的命令符提取出来
if(cmd == NULL) { continue; }

/* treat the remaining string as the arguments,
* which may need further parsing
*/
char *args = cmd + strlen(cmd) + 1; // 将命令中的参数 args 提取
if(args >= str_end) {
    args = NULL;
}
for(i = 0; i < NR_CMD; i ++) { // 按照命令的符号去查询、执行命令
    if(strcmp(cmd, cmd_table[i].name) == 0) {
        if(cmd_table[i].handler(args) < 0) { return; }
        break;
    }
}
// 异常处理
if(i == NR_CMD) { printf("Unknown command '%s'\n", cmd); }
```

cpu_exec()

```
cpu_exec(-1); // 结束执行
cpu_exec(1); // 单步执行
```

```
int instr_len = exec(cpu.eip);
// exec(cpu.eip) 从 eip 寄存器储存的地址开始执行一步
cpu.eip += instr_len; // 切换到下一步要执行命令的地址
#ifdef DEBUG
print_bin_instr(eip_temp, instr_len);
// 根据寄存器 eip 储存的地址
// 调用 instr_fetch(eip + i, 1)读取相关指令的内容
// instr_fetch 调用了函数 swaddr_read()访问模拟内存的虚拟地址
strcat(asm_buf, assembly);
Log_write("%s\n", asm_buf);
if(n_temp < MAX_INSTR_TO_PRINT) {
  printf("%s\n", asm_buf);
}
```

2 重新组织寄存器结构体

根据 nemu\src\cpu\reg.c 的代码补完 nemu\include\cpu\reg.h 的代码

注意测试代码中的赋值过程：

```
rand()
    → sample[i]
        → reg_l(i)
          reg_l(i)宏定义为 cpu.gpr[check_reg_index(index)]._32
```

因此，实际被赋值的变量只有 cpu.gpr[]._32

但是，在 assert()测试时，有如下判断：

```
assert(sample[R_EAX] == cpu.eax);
```

可以看到，表面上看 cpu.eax 是没有被赋值的，但实际上却可以进行值的比较。因此，cpu.eax 必然通过 union 方法与 cpu.gpr[]共用了一段储存空间。

根据以上推理，以及实验手册的提示，可以通过 union 共用空间如下：

```
typedef struct {
    union{
        union{ uint32_t _32; uint16_t _16; uint8_t _8[2]; } gpr[8];
        /* Do NOT change the order of the GPRs' definitions. */
                struct{
```

```
                    uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
                };
        };
        swaddr_t eip;
} CPU_state;
```

这样，实现的寄存器结构体组织如下：

共用首地址的元素，以前三组寄存器位例

gpr[0]._32	gpr[1]._32	gpr[2]._32	……
gpr[0]._16	gpr[1]._16	gpr[2]._16	……
gpr[0]._8[0]	gpr[1]._8[0]	gpr[2]._8[0]	……
cpu.eax	cpu.ecx	cpu.edx	……

需要注意的是，

```
uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
```

如果没有采用 **struct{};** 按序分配空间，那么实现的结构将为：

共用首地址的元素，以前三组寄存器位例

gpr[0]._32	gpr[1]._32	gpr[2]._32	……
gpr[0]._16	gpr[1]._16	gpr[2]._16	……
gpr[0]._8[0]	gpr[1]._8[0]	gpr[2]._8[0]	……
cpu.eax			
cpu.ecx			
……			

3 为简易调试器添加功能

单步执行 si

直接循环调用 cpu_exec(1)函数即可

打印寄存器 info r

直接调用 cpu 结构体将其中寄存器的值打印出来即可

要注意的是，在打印指针寄存器 esp ebp 时，十进制数值仍然采用 16 进制打印

扫描内存 x

在单步执行 si 指令中，调用了 cpu_exec()函数，能够打印 16 进制的地址，说明在 cpu_exec()内部有打印内存的代码。

检查 cpu_exec()代码可以发现，该函数打印的过程应该通过调用以下函数实现

```
print_bin_instr(eip_temp, instr_len);
```

这个函数中打印内存的语句是：

```
l += sprintf(asm_buf + l, "%02x ", instr_fetch(eip + i, 1));
```

可以发现同样调用了 instr_fetch(eip + i, 1)过程，找到这个过程的实现，其中调用了函数：

```
uint32_t swaddr_read(swaddr_t, size_t);
```

并且，手册 RTFSC 中有提到过这个函数是打印内存的。所以，cpu_exec()能打印内存，依赖于 swaddr_read()。

因此，扫描内存也调用 swaddr_read()即可。

另外需要注意的其他事情是，扫描内存是按 little endian 实现的。

此外，读取 char * 0x100000 的方法是 sscanf。这个方法是从 stackoverflow 上面找到的。