PA1 实验报告

第一部分

reg_b()

根据 reg.c 可以分析知 reg_b(0)即 cpu.gpr[0]._8[0]，reg_b(4)即 cpu.gpr[0]._8[1]，因此需要把前一个参数 i 和 i+4 都映射到 i，可以用 index&0x3 实现；后一个参数 i 映射到 0，i+4 映射到 1，可以用 index>3?1:0 实现

CPU_state

寄存器数组由 8 个 32 位通用寄存器和 1 个 32 位指令寄存器 pc 组成，为了能够访问 32 位寄存器的低位，需要把 gpr 声明为 union；为了使用 eax 等直接访问 gpr[i]._32，需要在 gpr 外面再套一个 union，然后把 eax，ecx 等放入外层的 union，同时为了保证 eax，ecx 等地址连续以和 gpr 对应，需要把 eax，ecx 等声明为 struct，为了能直接使用 cpu.eax 访问 cpu.gpr[0]._32，eax，ecx 等所在 struct 应该为匿名 struct，最后指令寄存器 pc 放在整个 unoin 外面即可

cmd_c()函数给 cpu_exec()传入参数-1 的思考

cpu_exec()的参数 n 代表需要循环执行的指令条数，传入-1 表示直接结束 cpu_exec()函数并正常退出

第二部分

token 集合的定义

在 expr.c 的 enum 中添加对应集合名称即可，考虑到下面的表达式求值部分，我按照优先级顺序依次添加了*、/、+、-、==运算符对应名称，同时使得同一优先级的两个运算符第一个值为偶数，第二个为奇数，使得他们对应值除 2 后相等

rules[]数组的正则表达式

首先由于\不能直接表示，所有需要\的地方都应该为\\

数字类型：[0-9]+可以匹配连续的数字，为了避免匹配如 123xyz456 之类的字符串，应该在正则表达式首尾加上\\b，从而只匹配纯数字

十六进制数字：标志符号用 0[xX]匹配，值部分用[0-9a-fA-F]+匹配，首尾加上\\b

加、乘、左右括号：它们都不能直接表示，应用\\加上自身符号表示

减、除、相等：直接用自身符号表示

空格：用“ +”匹配，表示多个连续空格

字母：同数字，用\\b[a-zA-Z]+\\b 匹配

make_token()函数

e 代表需要匹配的完整字符串，position 为 e 中当前开始匹配的位置，pmatch 结构中的 rm_so、rm_eo 代表一个匹配好的 token 首尾位置，nr_token 记录当前匹配的 token 的

数目。while 循环每次从 position 开始匹配，在 for 循环中用每一条 rule 依次对第一个串匹配，当匹配成功时，需要存储匹配好的串，同时更新位置信息以便对后一个串匹配。首先判断匹配到的 rule 是不是空格，是空格则不必存储，直接开始下一个串的匹配；不是空格，则根据子串的头指针和长度将其存入 tokens 数组，然后在 tokens 存好的串末尾加上’\0’否则不是一个字符串，同时把匹配到的 rule 对应 type 也存入 tokens，最后 nr_token 自增，结束 rule 的匹配。若 for 循环中没有匹配项，for 循环的计数变量 i 会等于 rule 数目即 NR_REGEX，此时函数直接返回 false 结束

expr()函数

首先如果 make_token()返回了 false，那就不必求值了，把 success 设为 false 再结束即可。对于求值，定义表达式求值函数 eval(int start, int end)，start 和 end 为 tokens 数组的序号，表示对 start 到 end 之间的表达式求值。根据给出的 eval 函数模板，首先需要定义字符串转换成数字的函数 str2int(str)和检查表达式函数 check_parentheses(start, end)。str2int 函数可以直接用 sscanf 实现，原理和 scanf 函数基本相同，根据 type 区别数字是十进制还是十六进制，设置 sscanf 的读入格式即可。check_parentheses 函数的第一个功能检查表达式的左右括号是否匹配，可以遍历一遍表达式检查左右括号个数是否相等即可，第二个功能检查表达式是否被一对括号包围，可以在遍历的时候记录第一次左右括号数量相等的位置，如果表达式被一对括号包围，则左右括号数量第一次相等的位置一定在表达式末尾处。在 eval 函数中，首先对于特殊情况判断，对于一般情况，首先是找到主运算符，然后表达式就可以表示为主运算符左边表达式值 val1 和右边表达式的值 val2 进行计算，这个过程可以用 switch 实现，根据不同运算符设置 case 返回 val1 和 val2 进行对应运算符计算结果。val1 和 val2 的值可以递归计算，设 pos 是主运算符在 tokens 中的序号，则 val1=eval(start,pos-1)，val2=eval(pos+1,end)。对于找主运算符，通过 for 循环遍历一遍 tokens 即可实现，用 op 记录 tokens 的 type，pos 记录 tokens 的位置，遇到左括号则用一个 while 循环跳过括号内的表达式；由于我们要找到优先级最低的运算符，根据之前 token 集合的定义，乘法对应 type 值最小，相等对应 type 值最大，因此如果 tokens[i].type/2 >= op 就可以认为找到了一个优先级更低的运算符，从而更新 op 和 pos，通过除 2 实现了乘和除同优先级，加和减同优先级，>=则实现了能找到同优先级中最右侧的运算符。

第三部分

首先对相关指令函数声明，并补全 cmd_table 中的信息
在 ui_mainloop()函数中已经对 args 进行了处理，传入指令函数的 args 已经去除了的第一个指令字符串。

cmd_si()

执行 N 步可以通过调用 cpu_exec(N)实现，没有给定 N 即 args==NULL，此时给 N 赋值 1 即可

cmd_ls()

若没有指定路径，参考 cmd_pwd()函数用 getcwd()读取当前路径。读取路径下的文件用 dirent.h 中定义的相关函数即可，

cmd_info()

在 isa_reg_display()打印所有寄存器值，用 cmd_info 调用即可，对于 pc 则直接打印 cpu.pc 即可

cmd_p()

expr()函数有个 success 参数，其功能是判断是否计算成功，expr()的返回值为计算结果，success 作为引用传递保证即使计算不成功也能传递运算是否成功的判断结果，根据 success 的值决定输出的信息

cmd_x()

首先参考 ui_mainloop()用 strtok()对 args 分离第一个字符串，以及得到余下的子串，使用 sscanf()把第一个字符串作为参数读入 N 中，对余下字符串调用 expr()函数计算得到扫描开始地址 addr，之后调用 memory.c 中定义的 paddr_read()函数，在 for 循环中每次连续读取 4 个字节的数据，设置相应 printf()的输出格式输出即可