VUE-backend Reading

运行 server

request: https://www.runoob.com/python3/python-requests.html

Json: runoob.com/python3/python3-json.html

中央处理器模拟 cpu

全局类memory & register

run：

1. 定义CC：ZF = T, SF = F, OF = F
2. 定义stat：SAOK（运行中）
3. 写回 -> 访存 -> 执行 -> 解码 -> 取指
4. info字典储存五个阶段的信息 + 寄存器 + 内存 + cc + control
5. 当前clock的信息为info（字典套字典）

内存模拟 memory

1. 本身是个大字符串，初始化为00 * MEMSIZE
2. 读取：
    1. Robutness：判断读取的索引是否超出范围，否则抛出异常
    2. 否则将目标地址区间内的字符串进行连接：join函数 https://www.runoob.com/python/att-string-join.html
3. 写入：
    1. Robustness：同上
    2. 否则将写入内容进行字符串拆解：自定义了split2chunks函数
4. 加载：
    1. 提取子串：使用findall https://zhuanlan.zhihu.com/p/139596371

        根据输入文件格式设置子串的正则表达式 https://blog.csdn.net/xuemoyao/article/details/8033138?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216679826811678241254575%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=166798268116782412546575&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-2-8033138-null-null.142^v63^control,201^v3^control_2,213^v2^t3_control2&utm_term=%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F&spm=1018.2226.3001.4187

        ```
        pos_tags = re.findall('(0x[0-9a-fA-F]+)', line)
        # 0x 接 0-9 a-f A-F(重复出现一次或更多次)
        val_tags = re.findall('0x[0-9a-fA-F]+:[ ]*([0-9a-fA-F]+)', line)
        # 0x 接 0-9 a-f A-F(重复出现一次或更多次) 接 冒号 接 空格(重复零次或多次) 接 0-9 a-f A-F(重复出现一次或更多次)
        ```

    2. pos_tags：指令位置
    3. val_tags：指令位置：指令序列
    4. python进制转换：https://www.runoob.com/python/python-func-int.html
    5. 如果地址长度不为0、指令序列不为0：那么将指令序列写入内存
    6. 出现异常：抛出异常
5. info：输出内存信息
    1. 用列表形式返回
    2. 4个储存单位（每单位2bits）为单位读取，用for循环
    3. 用join连接字符串

```
if __name__ == "__main__":
```

什么意思……?

取指 fetch

1. import * : https://stackoverflow.com/questions/2360724/what-exactly-does-import-import
2. 每条指令分别讨论，输出对应的状态更新提示（注意地址大小端切换）
3. try...except：分离icode和ifun，如果失败就设置item_error
4. 计算新pc（valP）：需要判断本条指令的长度
    1. Instr_valid：指令是否是我们做的内容当中的一条（是否合法）
    2. Need_regid：应该是表示需要用寄存器的指令，但是他并没有做cmovXX
    3. Need_valC：需要取出额外8字节常数的指令

$$valP = oldPC + (len(icode) + len(ifun)) + len(needRegid) + len(needValC) \tag{2}$$

5. 设置valC：
    1. 先统一设置为VNONE（const中定义的常量）
    2. 需要valC的指令：读取内存中该指令最后4位（？）
    3. 未成功读取就抛出异常imem_error
6. 设置rA rB：
    1. 先统一设置为RNONE
    2. 需要则从内存中读取：指令第二位
    3. 未成功读取就抛出异常imem_error
7. 异常处理：
    1. imem_error：非法内存访问。程序状态码设为ADR（遇到非法地址）
    2. instr_valid：非法指令，程序状态码设置为INS
    3. icode = HALT：halt指令，停止程序；程序状态码设置为HLT
8. jump和call指令：（条件判断正确时）程序跳转的指令就是valC。注意更新PC需要判断他是否为跳转指令，如果是，则新PC为valC，否则为valP
9. 程序状态码为AOK/HLT（程序正常操作或遇到halt指令）时，需要输出pc信息（注意大小端转换）
10. 更新下一个状态（用字典的update函数 https://www.runoob.com/python/att-dictionary-update.html）

译码 decode

1. 每个寄存器有两个读端口、一个写端口，srcA、srcB为读端口的地址输入，valA、valB为写端口的数据输出。
2. 各指令valA和valB对照（csapp上）：valA、valB其实就是寄存器对srcA、srcB的读入

指令	**valA**	**valB**
rrmov	rA	valE
irmov	/	/
mrmov	/	rB
rmmov	rA	rB
OP	rA	rB
jXX	/	/
callXX	/	%rsp
ret	%rsp	%rsp
push	rA	%rsp
pop	%rsp	%rsp 用于+8
cmovXX	**rA**	**/**

3. 各指令srcA与srcB对照（代码）：

指令	**src A**	**srcB**
rrmov	rA	
rmmov	rA	rB
OP	rA	rB
push	rA	%rsp
pop	%rsp	%rsp
ret	%rsp	
mrmov		rB
call		%rsp

4. 各指令valE对照：

指令	valE
irmov	rB
rrmov	rB
OP	rB
push	%rsp
pop	%rsp
call	%rsp
ret	%rsp

5. 各指令valM对照：

指令	valM
mrmov	rA
pop	rA

6. 特殊情况

指令	情况
call和jmp	不用从寄存器中读取信息，将valA设置为valP

7. 我猜他decode.py中line38-92是流水线几个阶段相关的内容
8. 没有srcA和srcB的时候就没有valA和valB
9. 更新信息

访存 memory

1. 向内存写入数据或读出数据，读出的数据成为valM
2. 分为read和write两个变量，分别指向内存写入的数据和从内存中读出的数据
3. dmem_error：内存操作失败的标志变量，如果失败就输出失败信息
4. 更新信息

写回 writeback

1. 仅当程序状态码为AOK（正常运行）时才进行写入，否则根据运行码进行相应操作
2. 写入寄存器
3. 当dstE和dstM有值就输出对应值
4. 更新状态：operation = op

执行 execute

1. ALU运算：要用二进制写，将带符号整数转换为二进制（注意转换大小端，换成16进制），再将二进制转换成二进制串，按位进行计算，设置ConditionalCode。最后结果二进制串要转换为十六进制带符号整数，大小端转换。加法、减法、与、异或运算分开写。
    1. 加法：
        1. ZF：全0时为T，其余为F
        2. SF：全1时为T，其余为F
        3. OF：res与两加数不同号时为T，其余为F
    2. 减法：
        1. ZF、SF同上
        2. OF：

        ```
        cc.OF = True if (valA > 0 and valB < 0 and res < 0) else False \
        or (valA > 0 and valB < 0 and res < 0) else False
        ```

    3. 与：
        1. ZF、SF同上
        1. OF = False
    4. 异或：
        1. ZF、SF同上
        2. OF = False
2. 需要用到ALU的指令：

指令	具体阶段
OP	valE <- valB OP valA
pop	valE <- valB + 8
push	valE <- valB + (-8)
call	valE <- valB + (-8)
irmov	valE <- 0 + valC
mrmov	valE <- valB + valC
rmmov	valE <- valB + valC
ret	valE <- valB + 8
cmovXX	valE <- 0 + valA

3. 根据以上表格：
    1. 两个跳转语句所用的常数：8和-8，可设置为常量
    2. 除了OP语句以外其余都用加法，将OP另作特判

```
aluA = ZERO
```

```
if cur.E.icode in [IRRMOVL, IOPL]:
    aluA = cur.E.valA
elif cur.E.icode in [IIRMOVL, IRMMOVL, IMRMOVL]:
    aluA = cur.E.valC
elif cur.E.icode in [ICALL, IPUSHL]:
    aluA = NEGFOUR
elif cur.E.icode in [IRET, IPOPL]:
    aluA = FOUR

op.append('Set aluA to {0}'.format(swichEndian(aluA)))

aluB = ZERO
if cur.E.icode in [IRMMOVL, IMRMOVL, IOPL, ICALL, IPUSHL, IRET, IPOPL]:
    aluB = cur.E.valB
```

aluA指上表第二个加数，aluB指上表第二个加数。

设置完需要输出信息。

4. 对于jXX指令的判断：
    1. Cnd作为判断结果变量，为布尔值

指令	Cnd
jmp	True
jle	(SF ^ OF) | ZF
jl	SF ^ OF
je	ZF
jge	not ZF
jg	(SF ^ OF) | ZF

寄存器模拟 register

1. 成员变量为包含了8个"0000"的列表，可看作是32bits（4个一分割）
2. read成员函数：
    1. 读取srcA、srcB的值，并将其对应的valA、valB对应到相应寄存器。
    2. 0xf的情况作为特判，第十六个寄存器不存在（从0开始计数）。
    3. 返回valA和valB
3. write成员函数：
    1. 读取dstE、dstM的值，分别赋给valE和valM
    2. 注意0xf的特判
4. info成员函数：输出所有寄存器的信息，用ret字典存放