COA2021-programming12

Good luck and have fun!

1 实验要求

能够正确解析和执行单条指令，需要实现以下 opcode 对应的指令的解析和执行:

1. opcode=0x05 ADD EAX,imm32
2. opcode=0x2D SUB EAX,imm32
3. opcode=0xA3 MOV moffs32,EAX
4. opcede=0xE9 JMP rel32

2 实验攻略

2.1 实验概述

本次实验为 CPU 的指令执行模拟。作为《计算机组织结构》课程实验的尾声，大家所模拟的 CPU 会指挥先前所制造的所有计算机部件，包括 ALU、MMU、内存单元等等来执行程序，也就是执行一条条的指令。因此，制造 CPU 是整个课程实验的最后一块拼图，大家一个学期的努力终于可以连贯起来。

本次作业为 CPU 模拟的第一部分，x86 指令集的实现。由于 x86 指令集支持的指令有上百条，因此我们选出了几条有代表性的指令让大家实现。本次作业也只需要大家解析和执行单条指令，连续执行多条指令的任务我们放到下次作业。

2.2 代码导读

2.2.1 代码结构

```
.
│ .gitignore
│ pom.xml
│ README.md
│
└─src
  ├─main
  │ └─java
  │   ├─cpu
  │   │ │ CPU.java # CPU 类，需要阅读
  │   │ │ CPU_State.java # CPU 的寄存器列表
  │   │ │
  │   │ ├─alu
  │   │ │
  │   │ ├─decode # 新增文件夹，取指令相关的类，需要阅读
  │   │ │
  │   │ ├─fpu
  │   │ │
  │   │ ├─instr # 需要在这个文件夹里面添加指令的实现类
  │   │ │   Instruction.java # 指令接口，需要阅读
  │   │ │   Mov.java # Mov 指令实现，需要修改
  │   │ │
  │   │ ├─mmu
  │   │ │
  │   │ ├─nbcdu
  │   │ │
  │   │ └─registers # 新增文件夹，寄存器类的定义
  │   │
  │   ├─memory
  │   │
  │   └─util
  │
  └─test
    └─java
      └─cpu
        └─instr
            AddTest.java
            InstrBuilder.java
```

```
        JmpTest.java
        MovTest.java
        SubTest.java
```

2.2.2 框架代码执行流

由于本次作业只要求能够对单条指令进行译码，测试用例会将一条指令写入磁盘起始处，并初始化 eip 的值为 0，然后将所需要的数据存入相应的地方（寄存器或者磁盘）。之后，测试用例会调用 CPU 类中的 execInstr()方法来执行一条指令。

CPU 类的 execInstr()方法，即本次作业大家需要重点理解的方法如下：

```
public int execInstr() {
  String eip = CPU_State.eip.read();
  int opcode = instrFetch(eip, 1);
  Instruction instruction = InstrFactory.getInstr(opcode);
  assert instruction != null;
  return instruction.exec(opcode);
}
```

这段代码了四件事情（分别对应第 1、2、3、5 行）：

1. CPU 读取 eip，即指令指针寄存器的值，eip 指向的地方即为当前需要执行的指令地址。
2. CPU 根据 eip 的值，调用 MMU 去读取当前指令的 opcode。
3. InstrFactory 根据 opcode 查询 decode.Opcode.java 中的表格，构建对应的指令类（需要自己在 instr 包下创建，实现 Instruction 接口，类名首字母大写，其余字母小写）。这里用到了工厂+反射的设计模式，这个设计模式超出了本课程的范围，在此不详细介绍，有兴趣的同学可以阅读 InstrFactory 类源码。大家只需要知道在这一行代码执行结束后，instruction 字段保存了一个指令类的引用。

4. CPU 调用指令类的 exec 接口，指令类根据自身的 opcode 确定指令长度（注意同一条指令可能对应多个 opcode，指令长度和字段含义也有所不同），调用 mmu.read 读取指令的剩余部分并执行。指令类执行完毕需要返回执行的指令长度(字节)。

在框架代码中，需要大家实现的只有最后一行，即各个指令类的 exec()方法。

2.3 实现指导

2.3.1 指令描述

完整的指令描述应该是通过查阅 i386 手册获得。为了降低难度，本次作业直接给出大家需要实现的指令描述如下：

- opcode=0x05 ADD EAX,imm32
    - 指令结构：1 字节 opcode + 4 字节 imm 立即数
    - 功能：DEST ← DEST + SRC;
    - 目的操作数 DEST：EAX 寄存器
    - 源操作数 SRC：imm 立即数
- opcode=0x2D SUB EAX,imm32
    - 指令结构：1 字节 opcode + 4 字节 imm 立即数
    - 功能：DEST ← DEST - SRC;
    - 目的操作数 DEST：EAX 寄存器
    - 源操作数 SRC：imm 立即数
- opcode=0xA3 MOV moffs32,EAX

    - 指令结构：1 字节 opcode + 4 字节 moffs 偏移量
    - 功能：DEST ← SRC;
    - 目的操作数 DEST：一个内存单元，地址为数据段段基址+moffs 偏移量，即(seg:moffs)
    - 源操作数 SRC：EAX 寄存器
- opcede=0xE9 JMP rel32
    - 指令结构：1 字节 opcode + 4 字节 rel 偏移量
    - 功能：EIP ← EIP + rel32;
    - 目的操作数 DEST：EIP 寄存器
    - 源操作数 SRC：rel 偏移量

2.3.2 实现参考

为了方便大家实现具体的指令类，我们已经实现好了一个 Mov 指令供大家进行参考。已经实现好的指令描述如下：

- opcode=0xA1 MOV EAX,moffs32
    - 指令结构：1 字节 opcode + 4 字节 moffs 偏移量
    - 功能：DEST ← SRC;
    - 目的操作数 DEST：EAX 寄存器
    - 源操作数 SRC：一个内存单元，地址为数据段段基址+moffs 偏移量，即(seg:moffs)

测试用例里有关于 opcode=0xA1 的测试代码，分别是 MovTest 的 test1 和 test2。大家在编码之前，可以先结合测试用例，理解整个指令执行的流程，以便更好地进行编码。

2.3.3 数据格式

由于我们模拟的内存中，数据是以 byte 流存储的，内存单元都是以 char 数组进行模拟，每一个 char 表示 8 位。而在我们的模拟的寄存器、alu、fpu 等结构中，数据是以 bit 流进行存储的，每一个 char 表示一位"0"或者"1"。因此，MMU 进行访存的时候，需要对 bit 流和 byte 流进行转换。

我们在 MMU 中已经编写好了两个数据格式转换方法如下：

```
public static String ToBitStream(String data)
public static char[] ToByteStream(String data)
```

由于在 IA-32 结构中，数据是小端存储的，因此我们在这两个函数中也已经处理好了小端存储的情况，大家在合适的地方直接进行调用即可。

3 参考资料

英特尔 80386 程序员参考手册(i386)intel：

https://css.csail.mit.edu/6.858/2014/readings/i386.pdf