Lab4 目标代码生成

史子凡 151180115 vivianszf9@gmail.com

一、 实验进度。

完成了实验四要求的全部内容，生成在 spim simulator 上运行的目标代码，支持多于 4 个的参数传递。

二、 编译方式。

根目录下写有 Makefile 文件，提供以下编译方式：

1. make lab4: 即编译所有与 lab4 相关的文件并生成所需要的 parser 可执行文件；
2. make parser: 对 main.c, syntax.tab.c, treeop.c, symbol_table.c，semantic.c，intercode.c, translate.c 还有 objectcode.c 进行联合编译，生成可执行文件 parser；
3. make clean: 删除 parser 文件；
4. make testcN：N 用 1 到 2 的数字替代，是对 test/compulsory/下的第 N 个必做样例进行测试，并输出文件 cN.s 到 output 文件夹下。比如，make testc2 表示对 test/compulsory/2.cmm 进行测试，输出文件路径为 output/c2.s。
5. make testmN: N 用 1 到 2 的数字替代，测试自己写的 test/my/N.cmm 这个测试样例，并输出文件 output/mN.s。
6. make testall: 对所有测试样例（2 个必做+2 个自己的测试样例）进行测试，生成的文件都在 output 文件夹里。
7. ./parser input output: 读入 input 测试文件，生成 output 文件。

三、 实现细节。

1. 总体思路

对 lab 3 生成的链式中间代码进行逐句翻译，生成目标代码。

2. 寄存器分配

寄存器分配采用了朴素寄存器分配算法，每翻译一条中间代码之前都将需要用到的变量和临时变量从内存中取出来放到寄存器中，在计算完毕后将计算结果写回到内存中去。所以每次在翻译一条中间代码之前，都对所有寄存器进行清零操作，表示未使用。在寄存器的结构中我采用 useornot 位来记录当前时刻此寄存

器有没有被使用。在选择寄存器的时候，遍历寄存器的 useornot 位，找到第一个未被使用的寄存器，使用该寄存器并将 useornot 位设置为 1。采用这种方法，其实在几乎所有情况下，除去一些特殊寄存器，真正被经常使用到的寄存器只有 t0 和 t1 两个。

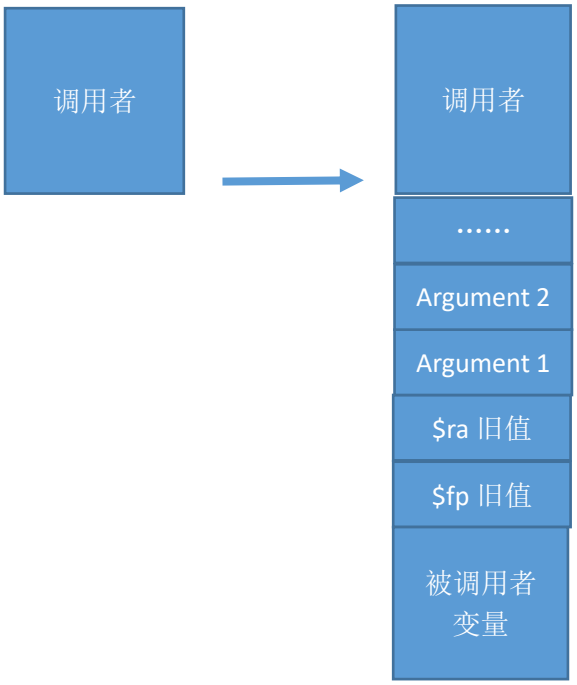
3. 栈中地址分配

由于采用了朴素寄存器分配方法，对于变量和临时变量，都要在栈中进行相应的地址分配。我是在目标代码生成之前先对中间代码中出现的变量和临时变量进行栈中地址计算，这样做便于后面涉及到函数调用的目标代码生成。这个地址是以偏移量的形式显示。栈从高地址往低地址生长，$fp 记录栈底位置，$sp 记录栈顶，在每一个函数内部进行操作，$fp 的值不会发生改变，所以变量和临时变量的位置使用“$fp+偏移量”的形式即可找到。

具体的计算步骤就是在生成目标代码前，遍历中间代码，记录每一个函数中每一个变量/临时变量的偏移量，并将所需空间累加起来最终得到这个函数所需要的整个栈空间的大小。这样在后面生成目标代码的时候就可以直接知道函数所需要的栈空间从而对$sp 进行更改完成栈分配操作。

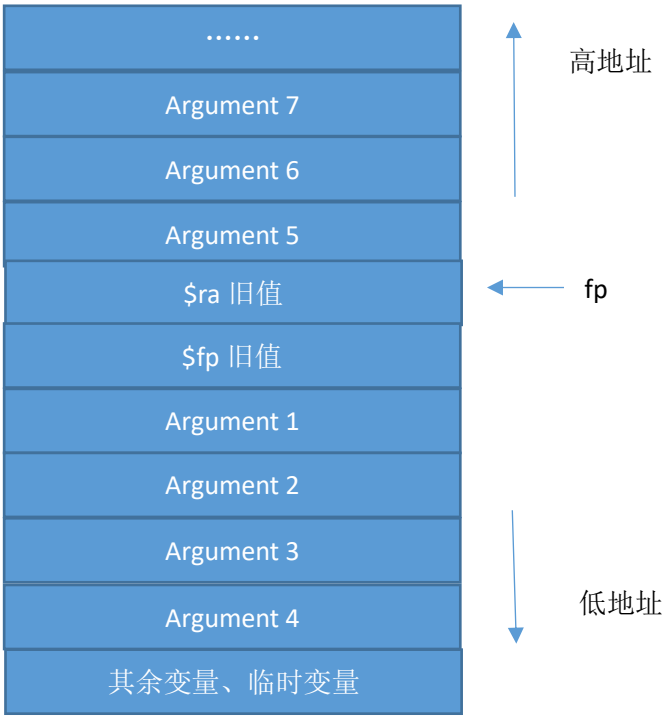
4. 函数调用

在调用一个新函数的时候，将栈底$fp 和返回地址$ra 放入栈中，在被调用函数返回后将$ra 再弹出来。由调用者负责恢复$ra 的旧值，由被调用者负责恢复$fp 和$sp 的值。被调用者栈中内存被舍弃。调用过程可画为：



5. 参数传递

参数个数小于等于 4 个的时候直接利用$a0~$a3 寄存器处理。大于 4 个的时候前 4 个仍然是直接利用$a0~$a3 寄存器，后面的就采用压栈的方式解决。因为采用的是朴素寄存器分配方法，所以前四个参数其实在被调用者的栈上是有位置的。整个栈中的参数布局如下图：



需要注意的是，在之前翻译中间代码的时候，argument 与 parameter 的顺序是相反的，而 argument 是逆序压入的，在一开始就没办法知道最后一个参数事实上是排在第几个，所以可以在遇到 argument 的时候先进行计数，在遇到 call 指令的时候再对之前出现的所有 argument 进行相关操作，这个时候就已知参数的总个数了。

四、 实验总结

复习了调用者和被调用者之间的关系，重新熟悉了汇编代码，回想起了计算机系统基础课上的不少内容。不过由于时间紧张，就没有尝试各种高大上的寄存器算法....