编译原理Lab4实验报告

陈泰霖

概述

本实验的关键部位有二：寄存器分配策略和栈管理。以下按此两部分分节阐述。

朴素寄存器分配

本实验采用每条IR指令所涉及数据即读即写的寄存器分配策略，简单，不易错。在此部分中，最终的是两个函数`LReg`和`SReg`，分别对应寄存器的读写。其接受三个参数：需要加载的寄存器，一定是"$t0", "$t1", "$t2"三者其中之一；需要加载的变量名，通常是"t%d"或"v%d"两类；文件描述符，用于写入加载寄存器和保存寄存器的代码。这两个函数中，最重要的逻辑是，如何找到变量在内存中的位置，这与本实验的管理策略紧密相关。

基于$fp的栈管理

由于变量在栈中的位置不可能用绝对位置寻址，只能采用相对位置寻址，我们首先要选择的是道标。那么可供考虑的道标有两个：$fp，$sp。本人最终选择的是$fp。

为什么不选择$sp？理由很简单，由于本人的实现并不像先遍历整个函数预先为所有变量分配栈空间，而是随着线性翻译，惰性的进行分配。那么很自然，在程序运行过程中，栈就是动态增长的，$sp也是随时变化的，那么每个变量的相对位置也会随之变化。因而引入一个固定位置的栈指针$fp就是最符合要求的选择。

当我们翻译一个函数时，在处理函数的开头`FUNC f%d`时，首先要把调用函数的$fp保存在栈中，然后把新的$fp移入寄存器，然后在程序中维护一个描述变量偏移的变量，以便后续安排变量的相对位置。这样，当函数结束返回值时，在返回前，我们会将栈缩小到$fp处，并把原$fp移回寄存器。而在翻译别的语句时，当我们遇到一个新的变量，就会根据维护的信息对其栈位置进行分配，然后更新信息，从而完成我们的栈管理。

在其中遇到过一个问题：由于在最初实现时，没有考虑到分支对该策略的影响，我们会一边分配变量位置，一边更新栈大小。这就导致，如果被翻译的程序存在分支，栈的大小实际上会比已分配的空间小，这会在函数调用时产生严重的负面影响，因为我选择的传参策略是压栈。为了解决这个问题，我引入了一个同步信号，当程序第一次翻译到`ARG`指令时，我们会对$sp进行一次同步维护，以确保栈寄存器所保存的大小和已经分配的栈的大小保持一致。当然加入了这个机制之后，原先一边分配一边扩张栈的动作其实就是一种无用的浪费了，但是我懒得改了。

其余值得提到的细节

对于手册中已经给到的指令，大多数都忠实地实现了。小部分则由于根据栈分配策略可以稍微做一些优化，但都差别不大。涉及函数调用和返回的指令，则增加了对栈管理的指令的翻译。

对于没有给到的指令翻译。前面提到过`ARG`的翻译时将函数压栈。那么`PARAM`的翻译则是为参数分配位置了，其实也就是根据其出现顺序，分配其相对$fp往上的参数位置。栈模型如下所示。

| ... ... |

| 函数参数 |

| 返回地址 |

| 保存的$fp |<---fp

| 函数变量 |

| ... ... |

对于`DEC`指令，由于在实验三时，我对`DEC x [size]`的实现策略是先用临时变量申请size大小的空间，而x保存其地址。其中实现需要注意一个小问题是，x保存的应该是size空间的底部地址（靠近$sp的那一端）。实现最初保存的是其头部（靠近$fp的那一端）地址，这导致一些错误。

总结

感觉相比前三次，这次实验相对简单，当然这与我采用简单暴力的策略有关。因而最终整体代码行数也不多。算是工作量最小的一次实验。在选择了比较质朴的寄存器分配策略后，唯一的难点就落到了涉及栈策略上，其中在函数调用和返回上的细节也需要一些推敲。当然由于有x86实现经验，这一部分自然有相当多可以借鉴的地方。

虽然最终采用的是最简单的策略，但不得不说的是，其他的策略在我看来十分优美，或许实现起来也是一件美事，可惜已临近期末，便就此搁置吧。