编译原理 Lab4 报告

171860659 吴紫航 联系邮箱：401986905@qq.com

实验目的

在词法分析、语法分析、语义分析和中间代码生成程序的基础上，将 C-源代码翻译为 MIPS32 指令序列，并可以在 SPIM Simulator 上运行。

实验要求

1）寄存器的使用和栈的管理可不遵循 MIPS32 的约定
2）程序可以接收一个输入文件名和输出文件，把 c—转化为汇编文件
3）输出的汇编代码可以被 SPIM Simulator 正确运行并得到预期结果

整体思路

本次实验的所有代码，都放在 asmcode.c 和 asmcode.h 中，整体思路如下

1）在程序最开始进行**中间代码分块**的铺垫工作。即把中间代码分成基本块，基本块内的代码不会出现跳转，将顺序执行。分块的方法是，程序第一条语句是一个块入口，程序的跳转语句的目标，是一个块入口，跳转语句的后一条语句也是一个块入口。一个块入口到下一个块入口前的代码序列，就是一个基本块。
2）然后程序框架会按顺序对每个中间代码行进行翻译。刚开始根据教程的**指令选择**表进行初步的翻译即可。对于寄存器选择、栈的空间分配、函数调用的参数和返回值的传递等在当前步骤可以先不考虑
3）然后介绍**寄存器分配**是如何考虑的。本次实验用到的寄存器有：$v0-v1,$a0-a3,$t0-t9,$s0-s7,$sp,$ra。具体的分配函数代码为 getReg（）

a) $v0 在按约定，在参数返回时使用存储返回值
b) $v1 用作临时存储中间值的寄存器（**例如数组变量的冒泡排序，栈中变量赋值给栈中变量的时候就需要一个中间寄存器**），使用前压栈保存，使用后恢复原本值。$a0-a3 用来保存前 4 个函数参数，ARG 指令中使用前压栈保存，CALL 指令返回后出栈恢复即可
c) $t0-t9 用来保存栈中的局部变量、常量、数组变量在栈中的首地址，作为调用者保存寄存器，每次在基本块将要结束、进行跳转之前，都要把寄存器中的变量溢出到栈中。**这里有个小坑，就是中间代码 call 在内部就进行了函数调用的跳转，在这个跳转前也是需要进行寄存器的溢出。当然 call 作为基本块的末尾，在调用返回以后还要进行一次变量溢出。**总之，指令顺序执行的末尾、跳转之前都需要把调用者保存寄存器溢出，在基本块的入口，$t0-t9 都是空闲的可使用的状态!
d) $v0-v7，用来保存函数块中的前 8 个变量。作为被调用者保存寄存器，在函数的开头，要对这 8 个寄存器进行压栈保存，在 ret 返回以前，要恢复现场。至于函数第 8 之后的变量，前面介绍过了，存放在栈中，需要使用的时候就加载到$t0-t9 即可
e) $sp 按约定作为栈顶指针，对栈内容的访问，就是通过 sp 再计算偏移量即可。
f) $ra 按约定存储了返回地址

4）寄存器的分配完成以后，现在介绍栈中具体的布局安排。
考虑最一般的情况，假设当前函数有 k（k>4）个参数。

a) 因为函数调用需要用到$a0-a3,因此要先保存 $a0-a3 寄存器的旧值
b) 如果参数多于 4 个，将被保存在栈中
c) 然后原返回地址要保存到栈中否则调用者无法返回正确的位置
d) 函数调用跳转后，在函数开头保存 s0-s7
e) 然后函数根据局部变量需要的空间生成栈帧空间

$a0-a3 寄存器的旧值
第 5-k 个参数
$ra 的旧值
$s0-$s7 的旧值
局部变量

栈的布局分配情况

数据结构

1) RegDescriptor
寄存器描述符，用来记录了$t 寄存器是否空闲、是否已被当前中间代码行的变量使用、被哪个变量使用、基本块下一次使用在什么位置等。用这个结构可以管理和调度$t0-t9 寄存器的使用。
2）FunctionBlockDescriptor
函数块描述符，用来存储函数的各个变量的描述符，以及局部变量的占用空间有多大。
3）ValueDescriptor
变量描述符，用来描述变量是否是数组、是否是参数、变量占用多大空间、变量名是什么，作为函数块描述符的内容，以链表的形式被函数块描述符保存。

关键步骤

为了精简报告，这里仅介绍在实验过程中比较关键的、或有体会的细节或难点。

1）在寄存器调度的时候，遇到了一个难点，关于常数占用寄存器的问题。如果发现寄存器被常数所占用，那调度的时候是否能替换掉呢？如果替换了，有可能当前指令是 x:=#k+y，#k 刚分配到一个寄存器，y 在分配的时候就把他挤出去了，从而错误。如果不替换，那常数占用寄存器后就一直不释放，显然也不对。因此在寄存器描述符中，**设置了锁成员标识 isLock。**如果寄存器被锁住，说明这个寄存器被当前正在翻译的中间代码所使用，因此如果是常数锁住该寄存器，那么其将不能被替换。要想使用，除非申请者就是锁住这个寄存器的变量，例如 x:=x+1,第二次 x 可以申请到分配好的已经锁住的寄存器。**每行翻译结束后，所有锁都需要释放。**最后，如果寄存器被常数占用，但没有锁住，则该寄存器可以认为是空闲的，可分配的。问题得到了解决。

2）每次到基本块的末尾，都需要对寄存器进行溢出到栈。这是教程的提示，但是一开始实现的时候产生了误区。本以为只需要在每个基本块最后一行中间代码翻译完后添加上溢出的翻译部分即可，后来发现不行。例如 Goto 的翻译，如果在 j 指令的后面再添加上溢出代码，实际上局部变量将根本得不到溢出，这是由于对教程描述的理解出现了偏差。经过思考，**正确的理解是：在绝对顺序执行的指令的末尾进行溢出。**例如，j x，作为基本块的末尾，溢出行为应该发生在 j x 之前，而如果例如是 x:=y，则需要判断下一条指令是不是基本块的入口，如果是，则说明 x:=y 是绝对顺序执行的指令的末尾，此时需要溢出，因为下一条语句可能是 LABEL 等基本块开头，$t 的状态都将是空闲的。

3）在修复了 2）中提到的溢出位置的 bug 以后，程序依然无法满足预期，经过调试，发现了更隐蔽的地方，就是之前提到的 call 语句。X:=call f，这个代码翻译的时候要分成两个部分，一个跳转，另一个是把返回值赋值给 X。按之前理解，需要在绝对顺序执行的指令的末尾进行溢出，**那么 X：=call f 在翻译的时候需要溢出 2 次！**一次是在 jal f 之前，这是因为会发生跳转。此外，x:=call f 的下一句也有可能是基本块的开头！需要进行判断，这是第二次溢出的原因。在实现的时候设置了全局标识 isBlockEnd，记录当前语句是不是基本块的末尾。

4）紧接着遇到的难点是**把调用者的函数参数作为被调用者的函数参数的情况**。例如，当前 ARG 的目标就是一个参数，对这个参数的寄存器分配可能是 a0-a3,而 a0-a3 同时可能是 ARG 要修改的目标，这将出现混乱。解决方案是，在 a0 和 a3 压栈保存以后，所有对旧 a0-a3 的访问都从栈中保存的位置获取！此时又有一个新的问题，就是偏移量的计算，由于旧的 a0-a3 已经压栈，并且有可能存在新的调用的参数压在栈中（详见之前的栈空间分配表），因此偏移量在计算的时候，要考虑这些变化。无论是旧 a0-a3 的位置，还是调用者局部变量的位置都需要加上一个由于 arg 行为导致栈的偏移量的改变。实际实现时引入了 stackOffsetForArg 全局变量，用来修复 arg 行为带来栈的变化的影响。

5）还有一个难点是关于数组的。每当对数组变量进行访问的时候，中间代码的实现是取数组变量的地址，然后再用偏移量访问具体位置的变量。因此当翻译到例如 t0:=&v0 的中间代码的时候，对于数组变量 v0 的寄存器加载，**只需要把数组在栈中的首地址加载到被分配的寄存器中即可（即 sp+数组的偏移量）。**这里注意，数组作为局部变量，一定是分配在栈中的，而具体在栈中的哪个位置，可以根据最开始初始化的函数块描述符进行查询。

6）最后一个值得注意的地方是变量可能是*t 的形式，t 是某个地址，那么在运算指令中，对于所有变量在分配好寄存器，都要分两种情况进行分别翻译，一种是变量 v，另一种是寄存器表示的地址的变量（往往是数组的某元素）。这有一个难点，我们以*t0:=*t1+*t2 为例，我们分配好了三个寄存器，其中存了栈 3 个地方的地址，但是实际的运算目标指令的操作数不能全是在栈中，因此这个时候需要临时寄存器存储中间结果。那么难点就是，我们如何获取足够的临时寄存器？前面提到，$v1 在本次实验是作为临时寄存器使用的，但是 v1 只有一个显然不够。怎么办呢？此时显然不能重新修改寄存器的分配安排了，那样要改的代码太多。**后来经过思索，我想到一个绝妙的解决方法，就是以常量的名义去 getReg 获取寄存器分配，**这样在同一行翻译时既不会分配冲突，在下一行后又能及时释放！

结果测试

测试的方式是直接用助教提供的 Makefile(没有任何修改)

输入 make clean　　输入 make parse

输入 ./parser [cmm 路径] [目标代码名]　　即可生成目标代码

输入 spim -file [目标代码]　　即可对目标代码的正确性进行测试