LAB4-Report

191220013 陈奕诺

一、实验要求

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（可以包含伪指令），并在SPIM Simulator上运行。

二、实验内容

将中间代码经过与具体体系结构相关的指令选择、寄存器选择以及栈管理之后，转换为MIPS32汇编代码。

1.代码准备

为了适应生成目标代码的需要，我首先对于实验三中生成的中间代码结构进行了必要的修改，将原先的直接打印`t3=*t1+t2`等中间代码，添加一步专门的取值转换`t=*t1`，以方便后续转化中对于数组的操作。

2.指令选择

由于我们使用的是线形IR，这里为了避免麻烦，选择了最简单的指令选择方式，也就是是逐条将中间代码对应到目标代码，进行相应的转化。对于每一行的代码，我们依据表格，将所有的中间代码进行转化，值得注意的是，大部分的代码操作需要通过寄存器来实现，这里我们用一个`ensure`函数来将相应的寄存器分配给对应的变量（或常量）。简单的，以`add`操作为例，我们首先将三个变量（常量）存入到相应的寄存器中，再对相应的寄存器做相应的`addi`指令操作，从而实现代码的转化，同时注意一些有关常数的操作可以提前经过判断后进行化简，通过`li`等指令来去除不必要的寄存器分配工作，从而优化目标代码。

3.寄存器分配

在指令选择中，我们调用了`ensure`函数来分配寄存器，这就是我们接下来要讨论的部分。为了保证一定的代码效率，同时简化转化过程，我们这里采取**局部寄存器分配算法**来进行寄存器的分配。在介绍具体的寄存器分配方式之前，我们首先了解一下，一些辅助的数据结构：

```
struct Register{
    int isfree; //1:free 0:used
    int age; //记录年龄，年龄最大的最可能被释放
    char *name; //寄存器名称 便于打印
};
struct VarDesc
{
    int offset;//栈中偏移量
    int reg_no;//寄存器编号
    Operand op;//变量对象
    struct VarDesc* next;
};
```

主要是一个代表寄存器的结构和一个符号表的结构。寄存器结构用来记录寄存器是否空闲和年龄，从而在寄存器分配时提供帮助，而寄存器的名称则主要是便于生成相应的目标代码。

现在我们来简单介绍一下实验中采用的寄存器分配方法。为了便于判断，每一次进行`ensure`操作时，我们首先对所有的寄存器，进行年龄的增长，而调用到相应的寄存器时对年龄置零。在进行寄存器分配时，我们为了简化管理，同时保证一定的效率，将8到25号寄存器作为通用寄存器，来分配给传入的对象。首先，我们先遍历所有存在寄存器中的变量，若发现变量已经在寄存器中，直接返回该寄存器，否则通过`allocate`函数分配寄存器。在`allocate`函数中，首先，遍历所有的寄存器，寻找是否有空闲的寄存器，若有，将对象存入寄存器，返回寄存器。若没有空闲的寄存器，找到所有寄存器中年龄最大的寄存器，将寄存器中的对象存入到栈中，在将目标对象存入到寄存器中。注意这里只将存有变量的寄存器中的对象存入到栈中，而不对常量进行处理，且我们需要维护一个`var_head`的寄存器存储对象的链表，便于查找和判断。至此，寄存器的分配操作已经基本完成。

了避免不必要的麻烦，我们在完成中间代码生成的工作后，遍历所有的中间代码，完成代码块的划分工作，主要依据条件转移、函数声明、标签等进行代码块的划分，并对每一条中间代码，存储相应的代码块编号，便于后续的处理。在基本块结束时，需要将本块中所有修改过的变量都写回内存中，对于本实验而言，就是把所有的寄存器中的变量写回栈中，并将所有通用寄存器设为空闲状态。

栈管理

现在我们已经有了代码选择和寄存器分配的基本方法，接下来说明栈的管理方法。由于代码运行过程中调用的具体代码的不可预知性，我们这里更多的是从静态的角度进行相应的思考。我们以一个函数为单位进行栈管理，在进入一个新的函数时，我们对寄存器和内存进行初始化，也就是进行置空操作，然后遍历函数中的所有中间代码，给所有的变量（包括所有临时变量）分配栈空间，也就是设置其在栈中的偏移量，这里用与`$fp`的偏移量进行表示，便于后续的访问，并存储在内存中，这里注意对存储栈中对象的链表`var_head_mem`的维护，并对`$sp`进行操作，实现压栈效果。同时在这一阶段要，将栈中存储的传入的参数取出，存储到对应的寄存器和函数自己开辟的栈空间中。

接下来，我们再进行具体的操作，为了减少不必要的麻烦，我们在分配寄存器时，额外进行一些操作。当寄存器不空闲时，要注意把寄存器中原来的对象保存到栈中的对应位置，这里包括对于`var_head`和`var_head_mem`的维护和目标代码的添加。当变量不在寄存器中时，我们默认他存在于栈中，于是在符号表中查找变量，获得偏移量从而添加将栈中对应的变量加载到寄存器中的目标代码，再返回寄存器。

由于函数调用的一些特殊需要，我们对于CALL语句进行特别的处理。为了实现对于`$fp`和`$sp`的维护，同时维护函数调用前后的寄存器状态和内存状态，我们进行了以下操作：

1.保存$ra至栈中

2.保存调用者函数形参至栈，也就是把调用者**使用的**参数寄存器（$a0~$a3）压入栈（这里实际不是压入栈，而是保存到原来开好的栈空间中）。

3.把$fp压入栈中

4.传入参数，将参数存入寄存器（前四个）或者压入栈（超出四个的部分）中

5.将所有通用寄存器压入栈中

6.`move $fp, $sp` 设置$fp为下一个函数操作做准备

7.调用函数 jar

8.`move $sp, $fp` 恢复栈指针

9.将所有通用寄存器弹出栈

10.将所有参数弹出栈

11.恢复$fp, 从栈中弹出得到原来的$fp

12.恢复调用者的参数寄存器

13.恢复$ra,从栈中弹出得到原来的$ra

这就是函数调用阶段进行的操作。而相应的由于我们压入寄存器的时机与规定不一致，要特别注意在处理 `function` 代码时，在从栈中获取相应参数时的偏移量与规定相比，会有72的误差。

至此，关于栈的关键操作已经得到了实现。

数组管理

除了上述基本管理说明以外，在本次实验中，我们还需要特别关注的是数组操作，由于将数组压入栈中会给相应的管理带来一定的困难，基于本实验中的前提条件，我们不妨将所有数组视为全局变量，存储在 .data段，在进行具体的代码转化之前，先遍历所有的代码，为所有的数组在.data段开辟相应的空间，于是对于数组的操作，就变成了.data段中数据的读取和写入。具体的实现，见如下代码：

```
void handle_get_addr(struct InterCodes* x, FILE* fp){
    int left_no = ensure(fp, x->code.u.two.left);
    fprintf(fp, "  la %s, v%d\n", reg[left_no].name, x->code.u.two.right-
>u.var_no);
}
```

```
void handle_into_addr_right(struct InterCodes* x, FILE* fp)
{
    Operand left = x->code.u.two.left;
    Operand right = x->code.u.two.right;
    int left_num = ensure(fp, left);
    int right_num = ensure(fp, right);
    fprintf(fp, "  lw %s, 0(%s)\n", reg[left_num].name, reg[right_num].name);
}
void handle_into_addr_left(struct InterCodes* x, FILE* fp)
{
    Operand left = x->code.u.two.left;
    Operand right = x->code.u.two.right;
    int left_num = ensure(fp, left);
    int right_num = ensure(fp, right);
    fprintf(fp, "  sw %s, 0(%s)\n", reg[right_num].name, reg[left_num].name);
}
```

三、实验总结

基本完成了实验的既定目标，实现了中间代码到目标代码的转化，采用实验3中的测试用例进行了一定的测试，暂未发现问题。实验过程中走了很多弯路，比如静态栈一开始设计成了动态栈，给实验带来了很多困难，在实验前应该进行更加细致的构思。