编译原理四实验报告

211220127 顾嘉宇

实现功能

将上个阶段生成的中间代码转为对应的Mips32指令，使得其能够在SPIM Simulator上运行并得到正确的结果。

变量&寄存器

构建变量和寄存器两个struct类型，分别用来存储函数体中对应变量的偏移位置以及寄存器的名字和存储的对应变量。

```
typedef struct _Register Register;
typedef struct _VarStructure* VarStructure;
#define Register_Num 32
struct _VarStructure{
    Operand op;//变量名
    int offset;    //变量在内存中的存储位置
    VarStructure next; //变量链表;
    // int reg 变量存放的寄存器，由于采用朴素寄存器分配法，每次调用后都会写回内存，所以该变量无效
};

struct _Register{
    int is_used;           // 是否使用
    char *name;          // 寄存器名
    VarStructure variable; //关联的变量VarStructure var
};

Register registers[Register_Num]; //32个寄存器
int local_offset; //被调用函数内存区对应偏移量
VarStructure var_head; //变量链表
int local_argnums; //调用函数前参数数目统计，调用后清零
```

核心思想

采用朴素寄存器分配方法，每次需要对变量进行操作时，则将变量存储到空闲寄存器中，在对应操作实现完后再将其写回内存。

此外，每次遇见函数体定义时，会遍历接下来的中间代码直至下一个函数体。将参数全部压栈，并将遇见的变量全部加入变量链表，方便接下来生成指令时查找变量对应的偏移位置。

```
void generate_function_objcode(CodeList codelists, FILE *file){
    fprintf(file,"\n%s: #函数初始化\n",codelists->code->u.op->u.func_name);
    ....
    local_offset = 0;//偏移量重新置0
    int param_num = 0; //计算参数数量，以方便计算其对应偏移量
    CodeList cur_codelist = codelists->next;
    while(cur_codelist->code->kind == IC_PARAM){ //遍历参数列表，将所有参数压栈
        ......
    }
```

```
    while(cur_codelist != intercodes && cur_codelist->code->kind != IC_FUNCTION)
{ //遇到一个新函数体时，遍历完整个函数内部的中间代码，将变量加载到内存区，方便后续写回内存
        switch (cur_codelist->code->kind)
        {
            ......
        }
        cur_codelist = cur_codelist->next;
    }
    fprintf(file,"\taddi $sp, $sp, -%d #移动帧指针\n",local_offset);
    for(int i = 8; i < 16; ++i)//所有被调用者寄存器均空闲
        registers[i].is_used = 0;
}
```

编译&测试方式

进入`Code`文件夹所在路径，执行`make`命令，即可获得`parser`可执行文件。 执行`./parser test.cmm`命令，即可自动生成output.s文件，使用SPIM Simulator可进行测试。

实验感悟

个人感觉是四次实验最简单的一次，需要理解的点主要就是如何管理好进入函数体和退出函数体时如何生成指令，以及对于存在于等式左右的指针类型Operand需要分别做不同的判断。