编译原理 lab4 实验报告

221220070 刁涟承

一、功能实现

- 成功将实验三中得到的中间代码经过指令选择、寄存器分配以及栈管理等操作，转换为能在 SPIM Simulator 上正确运行的 MIPS32 汇编代码。实现了对不同类型中间代码（如算术运算、逻辑判断、函数调用等）的准确翻译，确保生成的汇编代码逻辑正确且执行结果符合预期。
- 针对函数调用，能妥善处理参数传递、返回值存储以及寄存器保存与恢复等关键环节，保证函数调用过程在汇编层面的稳定执行。

二、实现思路

指令翻译

本次采用策略为在实验三的输出基础上，根据早已建立的打印中间代码的输出，转为根据中间代码树，来输出机器码。因此仿照`IR_out.h`文件结构，针对每个中间代码的节点，生成不同的机器指令。

```
// assemble.h
// 将IR中的操作数由内存加载到寄存器中，同时输出相应的机器指令
char* compile_Operand(Operand opd, int reg);
// 实现IR中基本运算和赋值语句的翻译
void compile_InterCode(InterCode icode);
// 对每个中间代码节点进行翻译，递归翻译下一级节点和语句、操作数
void compile_IRNode(IRNode* node);
```

寄存器分配算法

为了简化实现难度，本次实验采用了朴素寄存器分配。每个运算操作时都先将内存中的变量加载到相应的寄存器中，同时对指针的取地址、解地址做了分情况处理。特别要注意的是，在为数组变量分配栈空间时，需要根据数组的整个长度分配足够的空间份额，不然会造成内存的冲突。

栈管理

本实验中采用了模仿x86架构的栈管理策略，使用一个`$fp`作为活动空间的底部，`$sp`作为栈顶，每次传入参数时递增`$sp`；

同时在函数调用时，先将调用者对活动空间存储在栈中：将`$fp`、`$ra`压栈，更新`$fp`为指向旧值，将`$sp`向下推进，以为被调用者的活动空间分配足够的空间大小；

函数返回时，根据`$fp`指向的旧值恢复`$fp`、`$ra`，同时将`$sp`恢复为旧的`$sp`值，存储返回值，之后继续调用者函数的活动。

```
case _CALL:
    ...
// 保存活动空间现场
printf("\taddi  $sp, $sp, -4\n");
printf("\tsw  $ra, 0($sp)\n");
```

```
printf("\taddi  $sp, $sp, -4\n");
printf("\tsw  $fp, 0($sp)\n");
// 调用者调用
printf("\tjal  %s\n", op1_);
// 被调用者返回，恢复活动空间
printf("\tlw  $ra, 4($fp)\n");
printf("\tlw  $fp, 0($fp)\n");
printf("\taddi  $sp, $fp, -%d\n", FRAME_SIZE);
printf("\tmove  %s, $v0\n", lvar_);
```

三、编译方式

可以使用`Code/Makefile`结合命令行进行编译：

```
make clean: # 清除所有生成文件
make t-%: # 用某个测试样例测试
make io-%: # 用某个样例测试并输出到out.txt
make c-%: #用某个样例测试并输出到out.txt，与预期输出文件比较
```

四、编译环境

操作系统：GNU Linux Release: Ubuntu 22.04 LTS, Kernel version 5.15.153.1-2；
编译器：GCC version 7.5.0；
词法分析工具：GNU Flex version 2.6.4；
语法分析工具：GNU Bison version 3.0.4。