编译原理实验报告四

161220070 李鑫烨

完成的功能点

实验四在语法检查与分析、中间代码生成与优化的基础上，完成指令选择、寄存器分配与栈管理的任务，实现由中间代码生成 MIPS 汇编代码，并通过划分基本块、窥孔优化等手段进行局部优化，保证正确性前提下减少了冗余的读写内存操作，提高了代码效率。

实现思路

实验四将中间代码翻译为 MIPS 汇编代码，主要工作为

- 为一条或多条中间代码选择等价的汇编指令。
- 设计寄存器指派策略，为运算分量分配寄存器。
- 设计栈结构管理局部变量，支持函数调用。

并通过窥孔优化等手段合并指令、消除冗余读写内存操作，提高代码运行效率。

具体实现

相对于实验三为表示中间代码和运算分量设计较多数据结构，用于表示汇编代码的数据结构相对简单。主要内容在于为寄存器分配策略与栈管理设计数据结构，保存局部块内变量使用信息与寄存器状态、栈状态相关信息。

指令选择

指令选择采用非常朴素的一一对应翻译的方法。在此过程中可能产生冗余指令，如

```
addi, r0, r1, r2
lw, t0, 0(r0)
```

可以优化为指令

```
lw, t0, r1(r2)
```

但由于在指令翻译过程中难以收集到变量使用与其他信息，优化程度有限，因此放弃一趟式生成-优化方法，将优化工作留到所有汇编代码生成之后。

寄存器指派

进行全局的寄存器分配策略需要进行复杂的数据流分析，同时在较为简单测试样例中优化效果并不显著，而逐条指令进行寄存器指派的朴素寄存器分配算法将导致过多冗余读写内存操作，因此采用局部块寄存器分配算法，基于局部块内变量使用信息进行分配，实现难度较低同时优化效果接近图染色算法。

局部块寄存器分配算法以基本块为单位，以双向映射的数据机构存储局部块内变量使用信息与寄存器使用信息，封装为 ensure, allocate, free 与 spill 等操作，消除了大量局部块内部的读写内存操作。

栈结构

栈结构与 x86 架构无异，由高地址向低地址增长，同一作用域局部变量分布于同一栈帧，通过偏移量进行访问。

首先问题为如何计算栈帧大小与局部变量相对偏移量。考虑局部变量存放在栈上同时可能有编译器生成的临时变量溢出到栈上，不能在最初即确定栈帧大小与变量偏移量。因此采用遇到声明形式的中间代码即压栈计算偏移量的方法，虽然产生冗余压栈操作，但较为自然且易于调试，且能在后续过程中进行优化。

由于栈帧大小与偏移量在最初不能确定，因此采用 $\$sp$ 与 $\$fp$ 相配合的方法标记栈帧的上界与下界。

编译运行方法

在 lab4/Code 中输入以下指令

- make parser: 生成可执行文件 parser 并替换上级目录中 parser。
- make run: 已生成可执行文件 parser 条件下，测试 lab4/Test 中三个样例。
- make clean: 删除可执行文件以中间文件

若需测试 lab4/Test 中 filename.cmm，则在 lab4 目录下输入如下指令

./parser Test/filename.cmm > Out/filename.s

运行示例

用快速排序对长度为 5 的数组进行排序

```
int main() {
    int k = 0, data[5];
    data[0] = 5; data[1] = 4;
    data[2] = 2; data[3] = 1;
    data[4] = 3;
    quicksort(data, 0, 4);
    while (k < 5) {
        write(data[k]);
        k = k + 1;
    }
    return 0;
}
```

其中 *quicksort* 为快速排序函数具体实现。在排序前后用户栈区数组中内容与运行结果如下

```
User Stack [7ffff68c]..[80000000]
[7ffff68c]    00400018
[7ffff690]    7ffff6a0  00000000  00000004  7ffff6a0
[7ffff6a0]    00000005  00000004  00000002  00000001
[7ffff6b0]    00000003  00000000  00000000  00000001
[7ffff6c0]    7ffff7c8  00000000  7fffff6d  7ffffe62
[7ffff6d0]    7fffff50  7ffffe3a  7ffffe07  7ffffef7
[7ffff6e0]    7ffffed4  7ffffea7  7ffffe9a  7ffffe87
```

Figure 1: 排序前

```
User Stack [7ffff68c]..[80000000]
[7ffff68c]    00400018
[7ffff690]    7ffff6a0  00000000  00000004  7ffff6a0
[7ffff6a0]    00000001  00000002  00000003  00000004
[7ffff6b0]    00000005  00000005  00000000  00000001
[7ffff6c0]    7ffff7c8  00000000  7fffff6d  7ffffe62
[7ffff6d0]    7fffff50  7ffffe3a  7ffffe07  7ffffef7
[7ffff6e0]    7ffffed4  7ffffea7  7ffffe9a  7ffffe87
```

Figure 2: 排序后

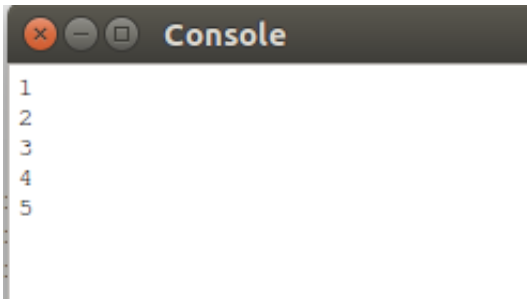

Figure 3: 运行结果