实验四

Chenning Yu 151242062 匡亚明学院 151242062@smail.nju.edu.cn

程序应该如何被编译?

- `make all` 会生成与 Makefile 处于同一文件夹下的 parser。
- `make debug` 是为了协助我 debug 语法生成过程的命令。它与 `make all` 生成 parser 的过程一致，不过这里的 parser 会开启诊断模式，即：运行时会输出状态机的转移过程。
- `make test4_1`，`make test4_2` 分别对应于实验指南中的样例 1，2。它们分别生成与 Makefile 处于同一文件夹下的 `testN.s`，其中 `N` 对应于样例的序号.

对于助教测试，建议先敲入 `make all` 重新生成一遍 parser。

如需测试已有样例，则使用上述的 `make test4_N` 命令； `N` 为对应的样例编号。

如需测试新的样例，请使用 `./parser path/to/testfile path/to/output.s` 命令； `path/to/testfile` 对应新的样例的相对路径， `path/to/output.s` 对应于输出文件的相对路径。比如： `./parser test1 test1.s` 。*需要注意*，为了保持与前面实验的兼容性，生成汇编代码的文件名后缀必须是 `.s` ；如果后缀为 `.ir` ，则会生成中间代码格式的文件；对于其他的文件后缀，则不会进行输出。

本次实验的主要代码位于 `Mips` 文件夹中。

实验实现了哪些功能？有哪些亮点?

我使用**全局寄存器分配方法**，完成了实验四的所有必做内容，比起局部寄存器分配方法来说，大大地缩减了目标代码的长度。接下来围绕一些我在编程中遇到的问题进行说明。以下问题解答和粗体字均为本代码亮点。

容我先简洁复述一下，讲义中全局寄存器分配方法的流程：使用活跃变量分析，也就是数据流方程的方法，计算出每个中间代码的使用到的变量集合 (in)，以及在该代码之后仍然活跃的变量集合 (out)。其次，使用数据流分析得到的集合，生成一张干涉图。干涉图中每个顶点意味着一个变量，顶点和顶点之间的边意味着这两个变量不能分配同样的寄存器，否则由于它们同时活跃，会导致多余的换入换出操作。接着我使用启发式的 Kempe 图染色方法，给干涉图染色，其中相邻点不能染相同颜色。每种颜色对应一个寄存器。最后，我们得到一个全局的寄存器分配机制。

使用位串实现控制流分析的局限性

为了高效地进行集合运算，我使用了位串来表示数据流中的 in 和 out 集合。对于 `out-def` 这种表示，使用位操作来表达，其实就是 `out&(~def)` 。位串在 C 语言中，最方便的表示方法其实就是 int 啦。但是，由于 int 只有 32 位，且每一位代表了一个变量，这就带来了一个问题：程序至多只能有 32 个变量或临时变量。当然如果我们使用 long long，那就可以有 64 个变量，但实际问题并没有得到解决。

在本次实验中，我使用了 int 来保存集合，因为至多支持中间代码中有 32 个变量。这是我代码的缺陷，我必须承认。但所幸在第三阶段时，我花了大量的时间进行中间代码的优化，因此变量数目得到了大幅度的缩减，在测试讲义所给代码时，并没有出现问题。

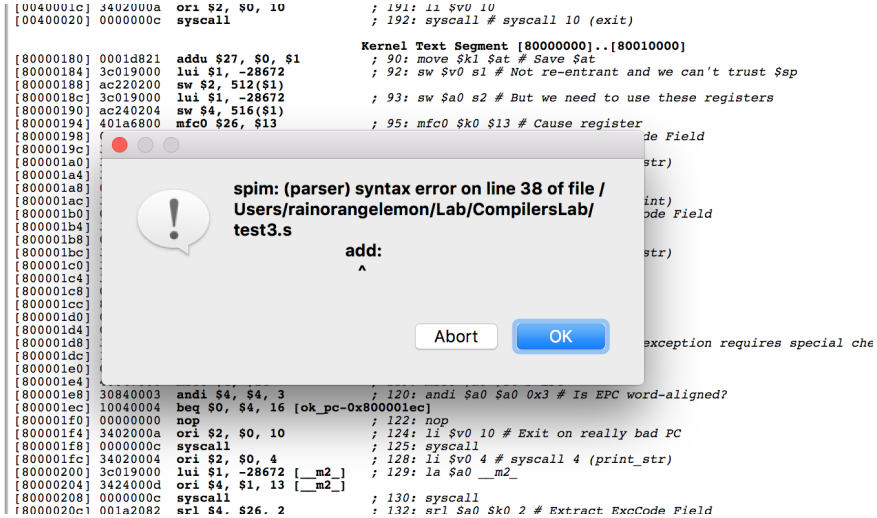
这就是使用位串，来实现控制流分析的局限性：位串的位的长度限制了控制流的集合使用位串的方法。要解决这一方法，我们可以尝试自己造轮子，用 C 语言实现一种能够支持任意大小集合的位串，其底层是通过将多个 int 拼接而实现的。当然，由于时间有限，且本人肝力不足，这一伟大的梦想就留给后人去发掘吧。

用户自定义函数与汇编指令重名引起的问题

在中间代码生成的时候，我们的函数名称是可以任意取的，除了被保留的 write，read，以及必须得有一个 main 的要求。但是，如果我们在原测试文件中写一段如下的程序：

```
int add(int x, int y){
    return x + y;
}
int main(){
    return add(0, 1);
}
```

接着，我们生成其对应的汇编代码。在 `QtSpim` 中测试该代码，会跳出如下的错误：



于是，我猜想，这里产生错误，是因为 `add` 在 Mips 中是保留字，与汇编指令重名，因此产生冲突。于是我将 `add` 改为 `Add`，导入 `QtSpim`，发现没有产生冲突，证明了我的猜想。

这单个汇编文件的语义问题是解决了，怎么把这个解决办法推广呢?

可以发现，汇编指令都是小写字母，那我们需要做的，就是把函数名换做至少包含一个大写字母的函数。最简单的方法，就是将首字母改为大写字母，这也就是我一开始的方法。但是这样做的问题是：原本可能有两个函数，一个叫做 `sub`，一个叫做 `Sub`，如此一变换，两个函数不就会重名了吗？因此，对于初始程序，有一个新的要求：**不同函数名的大写形式不得相同**。当然，虽然这个要求很弱，很容易遵守，但我得写清楚，以便用户查找错误所在。

多少个&哪种寄存器来存放溢出变量?

一开始编程时，我受讲义的暗示，以为仅需使用一个寄存器即可。但仔细想想，发现不是这样：

比如计算 `add reg(x), reg(y), reg(z)` ，且 `x, y, z` 均为溢出变量；那么至少需要两个寄存器来存放：首先我们可以证明不能只有一个寄存器，否则在 x 尚未被计算时，y 便已被 z 代替。其次，假设有两个寄存器，那么使 `reg(y)` 和 `reg(z)` 为两个不同的寄存器，而作为结果 `reg(x)` ，它可以放在其中的任意一个寄存器中，因为 `y, z` 必在内存中有最新的值。

使用以上的证明方法，一一根据目标代码的类型讨论，可以发现**最终其实需要两个寄存器，来存放溢出变量**。那么新问题是：使用哪种寄存器，来存放溢出变量呢?

经过思考，可以发现使用 s 类、而非 t 类的寄存器是最稳妥、最安全的。原因改进全局寄存器分配方法后， t 类的寄存器中保存的内容会被 被调用者 的刷新而丢失，同时，被调用者一定保证在调用前和调用结束返回时的 s 类寄存器中的内容不会更改，因此，无论溢出变量的生命周期有多长，使用 s 类的寄存器永远能保证，其不会出现尚未保存，便被刷新走的现象。

帧大小如何统计?

我们可以先看看，帧由哪些部分组成？答：返回地址，原帧寄存器内容（ fp ），活跃的 s 寄存器（t 寄存器由调用者负责保存，此外我实现的，也就是改良过的全局寄存器分配算法中，t 寄存器所存储的变量是不会在调用前和调用后保持活跃的），以及被声明的数组/结构体空间，最后还有给被溢出的变量保留的空间。

在以上部分中，不确定占了帧多少空间的部分有哪些？答：活跃的 s 寄存器、被声明的数组/结构体空间，还有给被溢出的变量保留的空间。如果要完成一次性扫描中间代码生成帧大小，并且完成目标代码生成的任务，个人认为是不可能的，因为在目标代码需要帧大小的时候（比如在函数的起始位置时，需要将 sp 往顶部生成一个帧大小的空间），计算帧大小的信息尚未完全读入，因而如果不想实现复杂的回填技术的话，目标代码需要等到一次性扫描完一遍后再重新生成，则无法实现一次性扫描生成目标代码的远大理想。

但说到底，为什么我们要计算出精确的帧大小空间呢？**我认为其实是没有必要的。其实，我们只需要给每个帧分配足够大的空间，且知道空间中的每个位置保留了哪个变量（或者没有变量）便足够了。**因此，在函数一开始，我们可以给函数分配与在整个程序中被用到的 s 寄存器的数目相对应的大小，加上 8 （留给返回地址与 fp）即可。这么做的原因，是当前活跃的 s 寄存器的数目肯定不会超过在整个程序中被用到的 s 寄存器的数目。之后，在溢出变量和被声明的数组/结构体申请空间的时，动态地跟踪并更新 sp 和帧大小，最后函数返回时的帧大小，必然是正确且足够容纳所需存放的空间的帧大小了。

如此一来，编译器既保留了程序的一次扫描生成的高效，也让我能够简化编程的难度，可以说是一举两得。

实验总结

- 这次实验大爆肝，我写了**一千两百行**左右的代码。虽然我一开始打算只使用上课讲的 getReg 的局部寄存器分配方法，但是在阅读完整个实验讲义之后，我发现全局寄存器分配算法异常地有趣！其既包含了上课讲过、但我没有实现过的数据流方程分析，也有设计得极其智慧的启发式图染色算法，可以说是趣味性极高的挑战。最终做下来，我发现虽然全局寄存器分配算法确实花费了我不少的脑力，但在此外的实现过程中，也有值得与之相提并论的、对于编程者细心的要求：对于每个中间代码一一细心的枚举，此外还要看情况申请、计算、访问溢出变量相对于 fp 的地址，还有非零常量的引入时机……这一系列的要求，都是我实现这一千两百多行代码的难关，可以说，每写一段代码，难度就多好多，因为写的时候既要考虑之前写过代码的边界情况，也要想到之后要实现哪些函数，这确实让我体会到了做编译器的困难，也让完成了这次实验的我成就感翻了好几番 (๑>◡<๑) 。
- 总得来说，通过这次实验，我接触并独立掌握了上课未涉及的全局寄存器分配算法，并且通过自己思考完成了一些代码的改良、优化，实现了较为复杂的数据流分析，了解了Mips的概念，复习了栈帧的结构。非常感谢这门课的老师大大、助教大大给我的这个机会，我很喜欢这套实验流程，让我独立地实现、优化了之前以为非常复杂的编译器，也锻炼了我对于 C 语言、解 bug 的熟练程度，极大地增加了我的信心，让我了解、掌握编译器的许许多多知识点。