编译原理实验 4 实验报告

鄢振宇（基础班）　　171240518

2020 年 6 月 8 日

I　代码框架

```
/Lab
    /Code #存放代码的文件夹
        Makefile      #用于编译的文件
        common.h      #常用的宏/函数的声明/定义
        error.c       #用于报错的相关函数的定义
        error.h       #用于报错的相关函数的声明
        handlers.c    #包含若干接受node*的函数的定义
        handlers.h    #包含若干接受node*的函数的声明
        ir.c          #包含中间表示、label、operand相关函数的定义
        ir.h          #包含中间表示、label、operand相关函数的声明
        lexical.l     #词法处理工具flex的源代码
        list.h        #类似于<sys/queue.h>，提供可以用于定义链表的宏
        main.c        #包含main函数
        offset.c      #生成IR变量$\rightarrow$偏移量的函数的定义
        offset.h      #生成IR变量$\rightarrow$偏移量的函数的声明
        operand_kinds.def#用X macro定义operand的各种类型和对应的函数原型
        optimizaiton.h    #定义各种优化的等级。高于parser等级的优化在运行时会被忽
        option.c      #用于处理命令行参数
        reg.c         #包含若干operand与寄存器关系的函数的定义
        reg.h         #包含若干operand与寄存器关系的函数的声明
        syntax.y      #语法处理工具bison的源代码
        table.c       #符号表相关函数的定义
        table.h       #符号表相关函数的声明
        test.c        #单元测试相关函数的定义
```

```
    type.c      #类型相关函数的定义
    type.h      #类型相关函数的声明
README #框架包含的一个自述文件
/Test #存放测试文件的文件夹，.cmm后缀表示C—源代码
    ...
parser      #预先编译好的二进制文件
report.pdf #报告
```

II 我的程序应该如何被编译

使用给定的 Makefile

III 我的程序实现了哪些功能

按照讲义要求实现了中间代码到 MIPS 伪指令的翻译，使用的是 naive 的翻译方案。（受益于 Lab3 的优化，我 Lab4 即使是 naive 翻译，也可以在不爆 spim 内存的情况下通过 L3 Advanced 测试）

有很多地方虽然能利用测试样例的特性进行简化，但是有的简化仅仅只能通过测试，而会与 real-world 的编译器背道而驰，我基本都没有采取，尽量使自己的项目保持一个较好的可读性、可扩展性。

IV 实现简述

将原本每个 ir 结构体配备的函数指针（虚函数）改为了函数指针结构体（虚函数表），一个用于打印 IR 格式的中间代码，另一个用于打印 MIPS 伪指令

由于实验 4 的特点，其实可以直接删去打印 IR 格式的相关代码。但考虑到 real-world 的编译器，可能涉及不同语言的联编，将中间代码交给其他程序进行优化等等操作。所以，我选择了采用虚函数表的方式，在不影响打印 IR 代码的情况下支持新的打印方式。

这样也可以让我的程序在增加别的输出格式时更加方便（只需扩展虚函数表）

V 记录各个变量在栈上的偏移

我遇到的第一个问题，就是记录每个变量在栈上的偏移量。

由于我的 IR 代码生成和语义分析是同一趟 (PASS)，所以 IR 代码优化完毕时，符号表已经不复存在了。所以，需要根据中间代码，重新构造一个“符号表”（形式化地说，是 IR 变量到栈上偏移量的映射）。

考虑到新构造的符号表在功能上和旧的符号表非常接近，为了避免 copy-and-paste,我将所谓的 OFFSET 当做两种“类型”(Type 的 kind),OFFSET_BASIC 和 OFFSET_COMP，用于指示在将其读入寄存器时，读入其值 (lw) 还是地址 (addi)。这样就可以完全不必修改 table.c 部分代码，直接支持用符号表记录各个变量在栈上的偏移量

因为在中间代码中，一次 LOAD, STORE 的大小都是一个字长 (word)，所以可以把它们都当做 INT，进而不需要用 Type 记录任何关于类型的信息。（即使需要，也可以通过在新加的 pseudo type 中添加对应的域）

那么，我只需要在进入每个函数前，对其扫描一遍，得到这个函数 IR 代码的“符号表”。翻译完一个函数之后，再将符号表重置即可。

（由于实验做了很多假设，所以完全可以把符号表改成不带删除的，但是那样会和 real-world 的编译器不同，所以我没有采取。）

VI Kipp It Simple and Stupid(KISS 法则)

我一开始写代码的时候，过多考虑了指令集，以 IR 指令的 a := b 为例：

b 为立即数时，要用 li，其他情况，可能要从栈上使用 lw，也可能可以直接从别的寄存器 move 而形如 a := b op c 的其它指令还要进行更多的 if-else 判断，代码可读性极差。

于是，我在 reg.c 中声明了 reg 函数，它接受一个 operand，并返回一个 int，表示将对应 operand 读到了几号寄存器。

这么一来，所有操作就都简化成了寄存器上的操作（虽然会产生很多冗余指令），只需要 reg 函数（准确地说是最底层真正执行操作数读入的函数）中进行充分的 if-else，在 ir.c 中完全不需要考虑，根据 reg 函数的语义，就可以保证操作数读入了寄存器

VII 利用解耦合留下优化空间

在我的代码中，每个 ir 翻译完之后，会对对应的寄存器调用 reg_free 来释放被占用的寄存器, 而 reg_free 又会去调用 spill。这个中间层看似是冗余，其实是为了解耦合，方便未来的优化。

在 naive 的实现中，每个指令执行完之后，便马上把所有 operand spill 回到内存。但是如果要考虑优化，则 reg_free 则可以改成只将寄存器表为“可以

复用”而不是马上 spill，这样，后面遇到的时候就可以复用，节省一存一取。而这个其实是 reg.c 的内部实现，reg_free 本身的语义仅仅是“将某个寄存器标记为可以复用”，而不包含“写回”。这样一来，在修改寄存器分配策略时，这样解耦合的模块化可以让优化更好实现。

与此相似的，还有 reg.c 中的 reg, reg_noload, tmp_reg, reg_use 等函数，它们都有明确的语义，ir.c 通过自己的需要去调用适当的函数，而在 reg.c 的内部，则可以通过较复杂的程序逻辑来优化这些函数的实际行为（只要不违背语义）

此外，虽然目前 mips_printer 中是直接打印出 IR 翻译后的结果，但是可以将打印操作，改为将翻译的结果用链表串起来，然后再在 mips 伪指令的链表上做类似于 L3 的优化。

不过，由于我 L3 优化了比较多，L4 即使采用 naive，也可以在可接受的时间范围内跑出 L3 的 Advanced 测试。因此，我就没有做过多优化了

VIII 其它考量

- 我查阅资料发现，glibc 允许用户在 printf 中注册自己的格式打印函数。于是，我注册了一个打印 Operand 的函数，打印 IR 格式中间代码就变得非常好写

```
make_ir_printer(assign) {
    output("%O := %O", i->res, i->op1);
}
```

- 注意到，如果用户定义的函数名字叫 add，或者其它与 mips 伪指令同名，会导致 spim 运行错误。所以，我在翻译时，统一在函数名字前加上了”f_”。如 add 会变成 f_add, main 变成 f_main。然后，只需手动写一个 main，只包含一条”jr f_main” 即可。

- 定义宏 ir_output 和 mips_output，这样可以很方便的修改 ir 和 mips 输出的格式（比如将 ir 代码的输出全部表上注释符号，这样在.s 中就可以同时看到 IR 和 mips 伪指令，降低阅读难度~~还可以考虑把 cmm 代码一起打印出来~~）

```
#define ir_output(fmt, ...) \
    output("#" fmt, ##__VA_ARGS__)

#define mips_output(fmt, ...) \
    output("  " fmt, ## __VA_ARGS__)
```