编译原理报告——实验4

刘志刚
匡亚明学院
141242022
njuallen@foxmail.com

实验进度

- [x] 基本要求
- [ ] 函数调用参数个数太多（>4个）时的压栈处理

编译运行

进入Code目录下，输入`make`即可编译，生成的可执行文件为`parser`。
执行方式：`./parser test.cmm test.s`。
由于我这次实验使用了C++，因此应安装有g++编译器。

实现说明

寄存器分配算法

我采用的是局部寄存器分配算法（基本块启发式原则），基本流程与讲义上描述的类似。即先将整段代码分拆成一个个基本块，在基本块内进行寄存器分配。在基本块结束后，将修改过的寄存器的值存入内存中。
算法大致框架与讲义上介绍的类似。但在原来的算法框架中，有两处涉及到策略，即所谓的可以应用启发式的地方。分别是：

```
for each operation z = x op y
    rx = Ensure(x)
    ry = Ensure(y)
    // 第一处，确定是否要释放当前寄存器
    if (x is not needed after the current operation)
        Free(rx)
    if (y is not needed after the current operation)
        Free(ry)
    rz = Allocate(z)
    emit MIPS32 code for rz = rx op ry

Allocate(x):
    if (there exists a register r that currently has not been assigned to any variable)
        result = r
    else
        // 第二处，选择spill哪个寄存器
        result = the register that contains a value whose next use is farthest in the future
        spill result
    return result
```

讲义上建议是使用数据流的方式来处理这两处。但我为了简便，在第一处，我的策略是不释放寄存器。在第二处，我的策略是将已使用的寄存器组织成先进先出队列的形式，我们每次spill的都是链表头部的寄存器，新使用的寄存器加入到链表尾部，这样子，有很大的可能，我们取出的寄存器就是最近不会要使用的（只是猜测）。

下面介绍一下相应的数据结构。

```
// 描述内存地址
// mips32仅提供了一种访存模式：基址 + 偏移量
struct mips32_address {
    char *base_register;
    int bias;
};

struct mips32_register_descriptor;

struct mips32_variable_descriptor {
    char *name;
    // 这个变量spill之后的内存地址
    struct mips32_address *mem_addr;
```

```
    struct mips32_register_descriptor *reg;
};

struct mips32_register_descriptor {
    // 这个寄存器的内容是否被修改
    // 寄存器的内容是否与变量在内存中的值一致
    int modified;
    char *name;
    struct mips32_variable_descriptor *variable;
};
```

上面是变量及寄存器描述符的定义，可以看到，这两种描述符是互相指向对方的。我们在实现时，对变量描述符，我们要求能根据变量名快速查找，因此变量描述符是存放在一个map中的，键值为变量名。而对于寄存器描述符，我们在分配、回收寄存器时，一般不关心名字，只关注其是否被分配，因此我将寄存器描述符存放在两个list中，分别表示已使用的和未使用的。
如下所示：

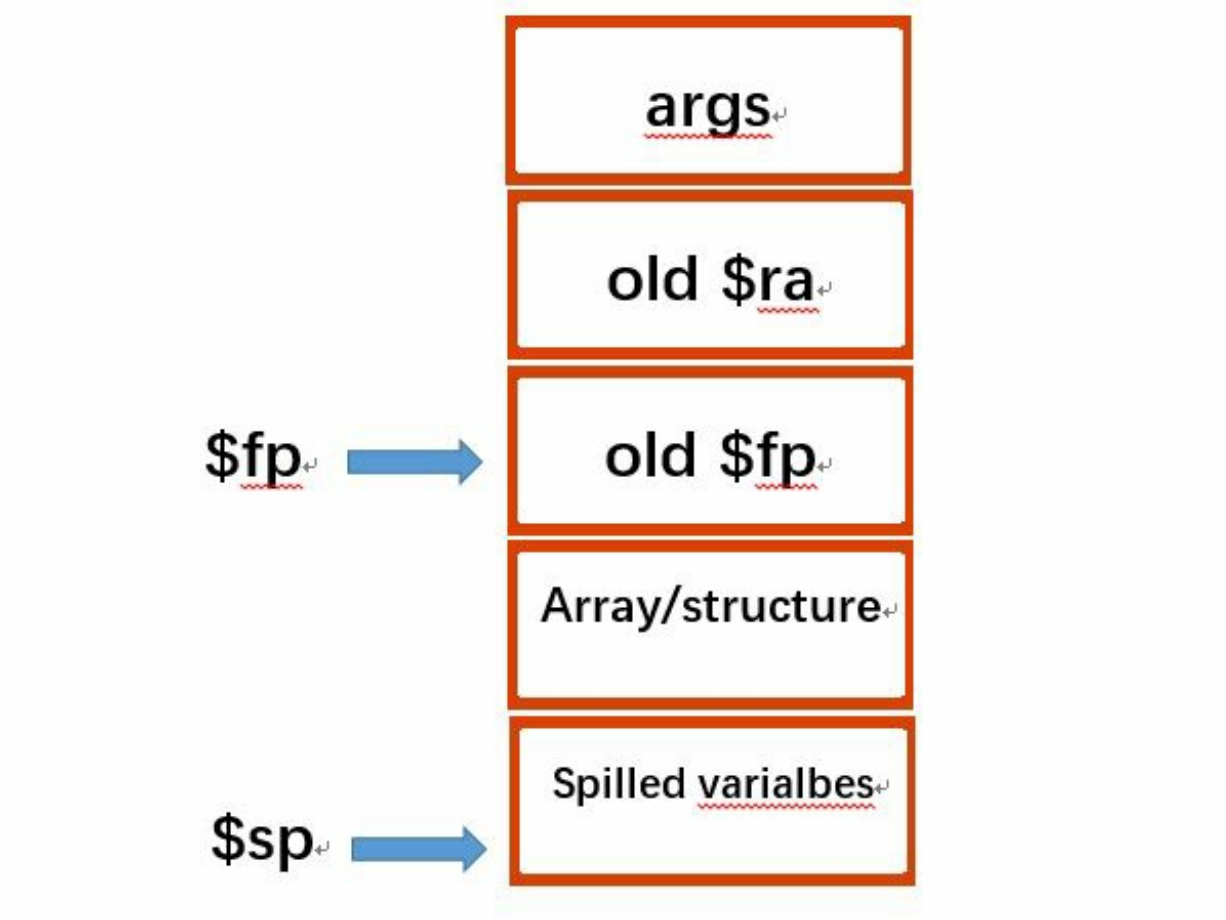
```
list<struct mips32_register_descriptor *> used_registers;
list<struct mips32_register_descriptor *> unused_registers;
map<string, struct mips32_variable_descriptor *> variables;
```

所有变量在栈上并没有实现确定好位置（数组、结构体除外），如果寄存器足够多，则变量不会被spill，则它不会存储在内存中。我们在将一个变量spill时，如果它的描述符里的内存地址为空，则我们临时将其push到栈上，并更新它的描述符中的内存地址。但由于我没有做活跃变量分析，所以会有许多临时变量，它们在使用完之后，其实就可以直接丢弃了，根本不必spill，但是我这个方法还会把它们存放到栈上，浪费栈空间。

calling convention
为了简化处理，我将所有用户可以随便用的寄存器都设置为callee saved。这样在处理时，在函数一开始时，我们将所有寄存器设置为used，等到使用寄存器时，自然地就会按照我们上面描述的寄存器分配流程，将它们spill到栈上，我们只要在函数返回之前再将这些寄存器恢复即可。

栈桢结构

栈桢结构如下图所示：



对应的，生成的prologue是这样子的：

```
main:
    # 保存$ra
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    # 保存$fp
    addi $sp, $sp, -4
    sw $fp, 0($sp)

    # 设置新的$fp
    move $fp, $sp

    # 为数组、结构体等在栈上分配空间，大小为400B
    addi $sp, $sp, 400
```

语言

这次在实现时，我使用的是C++，混用C和C++最主要就是一个链接的问题。C++为了支持重载，会对变量及函数名进行name mangling，为了能让它与C语言的代码正常链接，只要使用extern "C"就行了。

处理函数名

我们在实现时，一般都是把函数名原封不动地变成汇编代码中的一个label。但这样子有一个问题是，如果用户的函数名比较奇怪，正好与mips32的指令名冲突了（例如叫add），则会导致我们的汇编代码运行失败。因此，我对除main函数意外的函数都进行换名，将它们的名字加上func前缀，防止与指令名的冲突。之所以不对main进行换名，主要是因为没有指令叫main，同时main是运行spim所必须的一个label。

Bug

使用此算法之后，提供的寄存器数量越多，产生的寄存器的spill就越多，就越容易暴露出bug。我现在的实现仍然有bug，我编写了18个测试用例，当只提供三个寄存器时，会有两个样例失败。但如果提供18个寄存器（t0-t9以及s0-s7）时，就能全部通过。