编译原理实验4实验报告

功能及实现方式

核心逻辑：由IR生成汇编代码

我们使用了**逐行翻译**的模式: 即每次只考虑一行中间代码, 然后寻找合适的汇编语句翻译它.

代码的核心逻辑位于 `asm.c` 的 `ir2asm()` 函数中, 其框架逻辑如下:

- 函数间的寄存器分配是**独立**的, 因此我们逐函数地翻译中间代码
- 对于一个函数内部的所有中间代码, 可能需要扫描**多趟**, 以分别得到**变量信息**, **寄存器信息**, **参数信息**等
- 变量/参数信息包括: 变量/参数名、变量/参数大小、变量/参数在栈上的位置
- 寄存器信息包括: 变量/参数存到了哪个寄存器, 有哪些寄存器被占用

寄存器分配

对于寄存器的分配, 我们使用了**立即写回**的方式即:

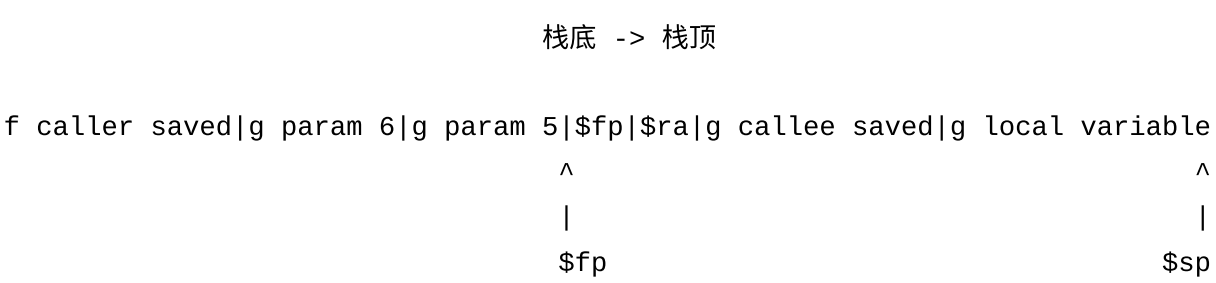
- 把中间代码中出现的所有**变量**都看作是函数的**局部变量**, 存储在函数的栈上
- 为当前要翻译的中间代码中出现的每个**变量**都分配一个空闲的**寄存器**, 将变量从内存中**加载**
- 对于中间代码中被修改的变量, 如 `t1 := a + b` 中 `t1` 获得新值, 我们在寄存器中计算出 `a + b` 的结果后将**立刻**把 `t1` 的新值**写入内存**, 后续用到 `t1` 时将**重新从内存加载**

相关实现代码如下:

```
// 为操作数1分配寄存器, asm_alloc_reg4var()使用了朴素的分配方法, 分配后立即从内存中加载
src1_reg = asm_alloc_reg4var(asm_query_var_idx(inside->op1->name), 1);
...
// 添加语句add reg(dest), reg(src1), reg(src2)
asm_append_code(asm_new_code_add(dest_reg, src1_reg, src2_reg));
// 归还reg(dest), 并同时写回
asm_free_reg(dest_reg, 1);
// 归还reg(src1_reg)和reg(src2_reg), 不用写回, 因为在此语句中src1和src2并没有改变
asm_free_reg(src1_reg, 0);
asm_free_reg(src2_reg, 0);
```

栈管理

我们按照MIPS32调用惯例, 使用了如下栈结构, 假设 `f` 正调用 `g` , `g` 有6个参数:



为了使翻译出来的汇编代码按照这样的栈结构执行, 我们实现了如下逻辑:

- 函数序言与尾声

上述栈结构中, `f` 将负责 `$fp` 至栈底这部分元素的压入, 而 `g` 将负责 `$fp` 至 `$sp` 这部分元素的压入. 因此, 函数的**序言**将压入 `$fp` 至 `$sp` 这些元素, **尾声**将弹出这些元素. 注意到, callee saved的寄存器(`$s0` ~ `$s7`)需要在**整个函数**寄存器分配方案确定后才能得知, 因此我们在**翻译完**整个函数后, 再把函数的序言**插入**回函数前部.

```
/* 1. 保存返回地址 $ra 至栈 */
top = asm_insert_code(top, asm_new_code_sw(reg_ra, -(ra_size+fp_size), reg_sp));
/* 2. 保存栈帧指针 $fp 至栈 */
top = asm_insert_code(top, asm_new_code_sw(reg_fp, -fp_size, reg_sp));
/* 3. 设置新的 $fp 为旧的 $sp */
top = asm_insert_code(top, asm_new_code_move(reg_fp, reg_sp));
/* 4. 按所需栈空间增长栈 */
/* 5. 保存callee saved寄存器至栈 */
...
```

函数的**尾声**与之对应的:

```
/* 1. 回收栈空间 */
/* 2. 恢复callee saved寄存器 */
...
/* 3. 恢复 $fp */
asm_append_code(asm_new_code_lw(reg_fp, -fp_size, reg_sp));
/* 4. 恢复 $ra */
asm_append_code(asm_new_code_lw(reg_ra, -(ra_size+fp_size), reg_sp));
/* 5. 返回 */
asm_append_code(asm_new_code_jr(reg_ra));
```

> 考虑到函数只有一个入口, 却可能有多个出口, 我们为函数设定了一个统一的出口 `funname_exit` . 翻译 `RETURN x` 语句时, 我们将 `reg(x)` 存入返回值寄存器 `$v0` , 随后立即跳转 `funname_exit` , 这样函数的尾声就只需要放在 `funname_exit` 这一个位置.

- 调用序言与尾声

调用者需要保存caller saved寄存器以及传递参数. 遵照MIPS32调用惯例, 前4个参数以寄存器(`$a0` ~ `$a3`)形式传递. 注意到, `$a0` ~ `$a3` 可能原本就保存着**调用者自己的**参数, 如果后续调用者还需要用到这些参数, 这个传递的过程将是**破坏性**的. 我们的解决方案是: 将 `$a0` ~ `$a3` 也视为caller saved的寄存器, 在序言保存, 在尾声恢复.

调用序言与尾声和函数序言与尾声是类似的, 最终将生成如下形式的汇编代码:

1. 增长栈空间
2. 保存caller saved寄存器
3. 压入参数(寄存器形式/压栈形式)
4. 调用函数
5. 回收栈空间
6. 恢复caller saved寄存器
7. 保存返回值

编译相关

环境要求

- ubuntu 20.04
- flex 2.6.4
- bison 3.5.1
- build-essential

编译指令

```
cd Code
make parser
```

最后生成的可执行文件为 `parser` ， 可以通过 `./parser /path/to/input_file /path/to/output_file` 运行