编译原理实验——目标代码生成

实验报告

171860667 袁想　　171860664 谢鹏飞　　yxnanda@sina.com

一、实现功能

将中间代码翻译为 MIPS32 指令（可包含伪指令）并在 SPIM Simulator 上运行。

二、主要数据结构说明

（1）描述寄存器的结构 RegisterDiscriptor，定义在 objectcode.h 中：

```
typedef struct RegisterDescripter
{
    int register_id;
    char register_name[10];
    int occupied;
    struct VariableDescripter *vardescripter;

} RegisterDescripter;

RegisterDescripter regs[32];
```

结构体中分别记录了寄存器的编号（0 到 31 号），寄存器的名字（对应实验指导中给出的 32 个寄存器别名，用于输出汇编指令），occupied 指示该寄存器是否被占用，为 0 表示该寄存器空闲，最后一个指针指向寄存器存储的数据信息（如果被占用）。

（2）描述寄存器存储的数据信息结构 VariableDiscriptor，定义在 objectcode.h 中：

```
typedef struct VariableDescripter
{
    struct Operand *operand;
    int register_id;
    int offset;
    struct VariableDescripter *next;

} VariableDescripter;
```

结构体中分别记录了指向操作数的指针，存放该数据的寄存器编号，该数据（如果是变量、地址或者指针）存放在栈的函数活动记录中的位置，即相对于函数栈帧栈底指针 fp 的偏移，以及指向下一个数据信息描述结构的指针（用于处理函数体生成汇编指令使用）。

三、实验思路与实现方法

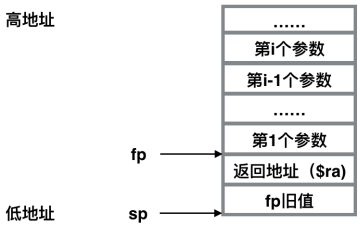
首先参考实验指导向输出文件中输出包括 read 和 write 函数体汇编指令等等基本信息，之后初始化 32 个寄存器并赋予对应名称且设为空闲。之后从中间代码双向链表的头开始向后遍历，读到一个中间代码指令就调用对应函数进行相应处理输出汇编指令。

对于寄存器分配，我们采用的是朴素寄存器分配方法，即每次处理完一条指令就会把结果写回到内存相应的位置。这样分配的好处在于不需要考虑寄存器冲突的情况，一条指令处理完毕后该指令临时用到的寄存器将恢复空闲。我们设定$t0-$t9 以及$s0-$s8 可以被自由使用。以处理 ASSIGN 中间代码为例说明寄存器分配方法。当当前遍历到的中间代码是 ASSIGN 时，将调用处理函数 void assign_objectcode(InterCode *p, FILE *fp)。由于我们实验三的设计，这里可以保证赋值语句左边操作数只可能是 TEMP_VARIABLE、VARIABLE（变量）或者 STAR（指针）类型。如果左边为指针，则用自定义 assign_register 函数分配左右两边所使用的寄存器，该函数即遍历我们之前约定的可以自由分配的寄存器，找到一个空闲的就将其设置为被占用，同时返回寄存器编号。由于右边操作数类型

多样，使用自定义 read_from_memory 函数将等号右边操作数内容读入分配好的寄存器。该函数中判断传入操作数类型，如果是常量或者常变量，直接利用指令“li 分配寄存器名字, 常量值”将值放入寄存器。如果是地址，那么我们通过遍历函数内部操作数信息链表（后面会讲到），找到这个存放这个需要取地址的变量在这个函数栈帧中相对于 fp 的偏移，然后通过“addi 寄存器名字, $fp, 偏移量”将 fp 加上这个偏移就是地址的值，将这个值存入寄存器即获取了地址。如果是 STAR，表明要取一个指针指向的值，那么首先获取操作数在栈帧中的偏移，通过“lw 分配寄存器名字, 偏移量($fp)”获取这个地址的值，然后利用“lw 分配寄存器名字, 0(分配寄存器名字)”取地址存放的值。其他情况，操作数只能是变量，那么利用“lw 分配寄存器名字, 偏移量($fp)”将变量的值读入寄存器。之后利用 lw 指令将左操作数的地址读入分配的寄存器，之后利用 sw 指令将存放右值寄存器的内容存放到左操作数地址指向内存中。如果左边是变量，先分配两个寄存器，调用 read_from_memory 读取两边的值进入寄存器，然后利用 move 指令用右值寄存器覆盖左值寄存器内容，之后将左值内容利用“sw 左值寄存器名字,偏移量($fp)”将内容写回内存。最后释放存放左右值的寄存器，设为空闲。

对于函数栈管理，由于我们采用的是每次对一条指令计算完毕就立即将结果写回内存，因此不需要在函数调用时进行整体的寄存器保存和恢复。此外为了方便对于函数栈进行统一的管理，我们将所有函数的参数都压入栈中而没有用$a0-$a3。因此，每处理一个 ARG 指令，栈开辟 4 字节空间（即 sp 值减 4），通过 read_from_memory 读取参数变量值进入寄存器，然后将其存放在当前 sp 指向的内存位置上。因为在函数调用结束后需要恢复 sp 的值，因此设定一个全局变量 argnum，每次读到一个 ARG 就加 1，因为函数参数传递都是连续的 ARG 最后一个 CALL，因此之后在处理 CALL 指令时，首先通过 jal 跳转到目标函数，由于源程序可能有函数名跟 MIPS 的指令冲突（例如 add），因此我们将在汇编输出的时候所有除了 main 函数之外的函数名前面加一个前缀 func_。在函数返回后将继续回到 jal 汇编指令下一句执行，因此输出 jal 指令之后为 CALL 指令赋值号左边的操作数分配寄存器并将内容读入寄存器，之后利用 move 指令将存放返回值的$v0 的值复制到分配寄存器中，之后利用 sw 指令将分配寄存器内容写回左边的操作数在内存中的位置（通过相对于 fp 偏移确定）。最后恢复函数调用整个过程前 sp 的位置，即处理 ARG 之前的位置，将 sp 加上 argnum * 4 恢复，最后将 argnum 清零。

由于我们需要在函数体内部指令处理前就要确定函数体内所有变量（类型是实验三中的（临时）变量，（临时）地址，指针）在函数的活动记录中的位置，因此我们将在处理 FUNCTION 指令时将这一个函数的函数体遍历一遍，为其中每一个变量分配在函数栈帧中的位置（即确定相对于 fp 的偏移）并且根据此为函数开辟一定大小的栈空间。首先先输出函数名字，之后清空函数内部操作数信息链表。之后进行栈的相关操作：



在处理FUNCTION中间代码一开始，先将**sp**减**4**，用于存放返回地址**$ra**，之后再将**sp**减**4**，用于存放调用这个函数的函数的**fp**（**fp**的旧值），之后将**fp**值设为**sp**，将**fp**加**8**，成为这个函数栈帧的栈底。处理完毕后栈结构如左图所示。

栈从高地址向低地址生长

之后遍历这个函数内的所有 PARAM 指令（是连续的，之后不需再处理），由于中间代码中传入的实参是按照形参定义逆序传入，因此在确定形参的值在栈中位置的时候，第一个形参的值相对于 fp 的偏移就是 0，第二个是 4，依次类推，创建 VariableDiscriptor 记录形参操作数类型以及相对于 fp 的偏移，将其插入到函数内部操作数信息链表中。之后继续遍历整个涉及函数的指令（一直到遇到下一个 FUNCTION 或者中间代码结束）进行预处理，将涉及变量的指令中的变量分配在栈中的位置，将其插入函数内部操作数信息链表。例如对于中间代码中的 ADD，就要将等号左边的操作数，右边的两个操作数分别调用 insert_operand 处理。该函数首先判断操作数是否是常量，如果是则不需插入。否则在已有的函数内部操作数信息链表查询此操作数，如果查询不到，那么当前偏移值更新为原来值减 4（初始化为 0），之后再将当前偏移值减 8 得到真正的偏移（因为偏移值目前是相对于 sp 的计算，而 fp 的值是 sp 加 8，因此相对于 fp 的偏移要在相对于当前 sp 偏移上再减去 8），将这些信息插入到函数内部操作数信息链表中。需要特殊处理的是 DEC（之后不需再处理）。我们获取 DEC 定义的 size，然后当前偏移值更新为原来值减去 size，之后操作与前面讲述相同。这一遍遍历函数体中 PARAM 之后的中间代码只是预处理，后面会调用相关函数真正处理。在预处理遍历结束后，可以确定累计的偏移量，利用指令“addi $sp, $sp, 累计偏移量”开辟函数所需要的栈空间。在处理 RETURN 指令时，首先利用 lw $ra, -4($fp)将返回地址放回$ra，之后 move $sp, $fp 恢复原来 sp 的值，之后分配寄存器读取返回指令需要返回的操作数的内容，之后 lw $fp, -8($fp)恢复 fp 旧值，之后利用 move $v0, 分配寄存器名字将返回值放入$v0，通过 jr $ra 返回。

加减乘除处理：使用前面讲过的函数为表达式左值和右边两个运算数分配寄存器并将右边两个运算数的值存入分配好的寄存器，之后分别使用 add/sub/mul/div 指令运算，最后将左值寄存器的值用 sw 指令存入栈中即可。最后释放之前分配的三个寄存器。

对于 READ 中间代码处理：将栈顶指针 sp 减 4 并将返回地址 ra 存入栈中，之后 jal 跳转，处理结束后，使用 lw 读出返回地址并将 sp 加 4 还原栈，之后分配存放读取结果操作数的寄存器并将内容加载入寄存器中，使用 move 指令将$v0 内容复制到结果寄存器中，最后使用 sw 指令将结果寄存器的值存入栈。处理结束后释放分配的结果寄存器。对于 WRITE 中间代码处理：将要显示的值存入 a0 寄存器，将栈顶指针 sp 减 4 并将返回地址 ra 存入栈中，之后 jal 跳转，处理结束后使用 lw 读出返回地址并将 sp 加 4 还原。

四、编译说明：利用给定的 Makefile 文件，置于与源码相同路径下，在终端执行 make。