编译原理实验报告

实验五 目标代码生成

学号：202011998088　　姓名：臧祝利　　日期：2023年1月8日

实验要求

在词法分析、语法分析、语义分析和中间代码生成程序的基础上，将 `C--` 源代码翻译为 `MIPS32` 指令序列（可以包含伪指令），并在 `SPIM Simulator` 上运行。

基本要求

将中间代码**正确**转化为 `MIPS32` 汇编代码。OK

附加要求

①对寄存器分配进行优化，如使用局部寄存器分配办法等。

探索失败了QAQ

②对指令进行优化

实验分工

小组内的实验分工如下：

> 臧祝利：代码框架+ `RETURN` 语句的翻译
>
> 陈金利：函数部分翻译
>
> 孙泽林： `LABEL, GOTO, WRITE, READ, ARG, ASSIGN` 语句的翻译
>
> 赵建锟： `CALL, ADD, SUB, MUL, DIV, IFGOTO` 语句的翻译

实验环境

- Linux：Ubuntu 20.04 LTS
- Flex： V2.6.4
- GCC： V9.3.0
- Bison：V3.5.1

实验设计

新增 `Simulater.h` 和 `Simulater.c` 文件

使用 朴素寄存器分配 和 线性IR 逐条翻译中间代码

Simulater.h

设计了两个数据结构——寄存器和栈;

寄存器用于存储目前的状态以及寄存器的名称;

```
struct Register
{
    int state;  // 0 is empty, 1 is used
    char *name; // 寄存器的名称
};
```

栈存储结点的类型、编号以及偏移量，用于分配内存大小；

```
struct StackNode
{               // 栈指针
    int offset; // 偏移量
    int kind;   // 类型
    int varno;  // 编号
    struct StackNode *next;
};
```

Simulater.c

逐句翻译中间代码，修改了一下输出顺序，优先把`main`函数的部分输出，然后再输出其他函数的目标代码；

关键的几个函数：

void Load_Reg(Operand op, int reg_, FILE *fp);

函数作用：把值加载到寄存器里；

```
void Load_Reg(Operand op, int reg_, FILE *fp)
{
    switch (op->kind)
    {
    case OP_VARIABLE: // 变量类型
    {
        int offset = findoperand_offset(op);
        if (op->ifaddress == OP_ADDRESS) // 将地址加载到目标寄存器
        {
            fprintf(fp, "  la %s, %d($fp)\n", reg[reg_].name, -offset);
        }
        else // 将值加载到目标寄存器
        {
            fprintf(fp, "  lw %s, %d($fp)\n", reg[reg_].name, -offset);
        }
        break;
    }
    case OP_CONSTANT: // 如果是常数，用li赋值给寄存器
    {
        fprintf(fp, "  li %s, %d\n", reg[reg_].name, op->value);
        break;
    }
    case OP_TEMPVAR: // 临时变量类型
    {
        int offset = findoperand_offset(op);
        if (op->ifaddress == OP_ADDRESS) // 如果是地址类型
        {
            // 赋给寄存器目标地址，然后再得到值
            fprintf(fp, "  lw %s, %d($fp)\n", reg[14].name, -offset);
            // t6
            fprintf(fp, "  lw %s, 0(%s)\n", reg[reg_].name, reg[14].name);
        }
        else // 将值加载到目标寄存器
        {
```

```
            fprintf(fp, "  lw %s, %d($fp)\n", reg[reg_].name, -offset);
        }
        break;
    }
    case OP_FUNCTION:
    {
        // 赋给寄存器函数的地址
        fprintf(fp, "  la %s, %s\n", reg[reg_].name, op->funcname);
        break;
    }
    case OP_LABEL:
    {
        fprintf(fp, "  la %s, label%d\n", reg[reg_].name, op->varno);
        break;
    }
    }
}
```

void Save_Reg(Operand op, int reg_, FILE *fp);

函数作用：把值从寄存器写回内存中

```
void Save_Reg(Operand op, int reg_, FILE *fp)
{
    switch (op->kind)
    {
    case OP_VARIABLE:
    {
        int offset = findoperand_offset(op);
        fprintf(fp, "  sw %s, %d($fp)\n", reg[reg_].name, -offset);
        break;
    }
    case OP_TEMPVAR:
    {
        int offset = findoperand_offset(op);
        if (op->ifaddress == OP_ADDRESS)
        {
            // t6存取目标地址
            fprintf(fp, "  lw %s, %d($fp)\n", reg[14].name, -offset);
            // 存到目标地址中
            fprintf(fp, "  sw %s, 0(%s)\n", reg[reg_].name, reg[14].name);
        }
        else
        {
            fprintf(fp, "  sw %s, %d($fp)\n", reg[reg_].name, -offset);
        }
        break;
    }
    }
}
```

void TransLateCode(FILE *fp, struct InterCodes *cur);

函数作用：对每句中间代码进行翻译，根据中间代码的类型，选择对应的 `mips` 语句进行输出；

这里只说明 `return` 语句的翻译过程。

其余详细内容见以下三人的实验报告；

陈金利：函数部分翻译；

孙泽林：`LABEL, GOTO, WRITE, READ, ARG, ASSIGN`语句的翻译；

赵建锟：`CALL, ADD, SUB, MUL, DIV, IFGOTO`语句的翻译；

RETURN语句翻译

分为了两类：返回变量类型还是返回常数类型，在返回常数类型时又额外判断是否返回0，如果是0的话可以直接用`$zero($0)`寄存器，就不需要再将0这个立即数额外放到一个寄存器中；

```
case IN_RETURN:
    {
        if (cur->code.u.one.op0->kind == OP_CONSTANT) // return的结果是常数
        {
            int value = cur->code.u.one.op0->value;
            int tmpreg = 8;
            if (cur->code.u.one.op0->value == 0) // 如果return的0，直接使用$zero返回即可
            {
                fprintf(fp, "  move $v0, %s\n", reg[0].name);
            }
            else // 需要用一个寄存器存取立即数
            {
                Load_Reg(cur->code.u.one.op0, tmpreg, fp);
                fprintf(fp, "  move $v0, %s\n", reg[tmpreg].name);
            }
        }
        else
        {
            int returnreg = 8;
            Load_Reg(cur->code.u.one.op0, returnreg, fp);
            fprintf(fp, "  move $v0, %s\n", reg[returnreg].name);
        }
        // 结果保存,修改返回地址和fp;
        fprintf(fp, "  move $sp, $fp\n");
        fprintf(fp, "  lw $ra, -4($fp)\n");
        fprintf(fp, "  lw $fp, -8($fp)\n");
        fprintf(fp, "  jr $ra\n");
        break;
    }
```

unimportant

尝试探索的一小部分内容(一堆废话，可忽略)：

对于寄存器的替换，在基础代码中，`$t0,$t1,$t2`为最常使用的寄存器；

想使用局部寄存器分配方法，需要将代码划分成基本块，考虑到个人能力等原因，以类似的思想，使用`朴素分配` + `局部更换`的方式——认为整个代码为`一个`基本块，对于某一个需要寄存器的运算对象，采用以下方法：

> 检查其是否已经占据寄存器，如果有，查找到相关寄存器；(认为只使用$t0 ~ $t9)
>
> 如果没有占据寄存器，进行寄存器的分配，将空寄存器分配；
>
> 如果寄存器已满，则选择将“最近最少使用的寄存器“清空，并使用这个寄存器；

为了使得寄存器能够满足以上条件，寄存器的数据结构修改为:

```
struct Register{
    int state;  // 0 is empty, 1 is used
    char *name;  //寄存器的名词
    int frequecy;
    int opno;
    int opkind;
};
```

其中 `frequecy` 用于记录频率，即调用一次寻找寄存器函数 `get_reg()` ，就会使得其使用频率增加1；

`opno` 和 `opkind` 用于记录运算对象的编号和种类，用于判断是否存在；

但是后续涉及到一些问题，由于时间原因荒废此方案，可作为 改进点 ；

实验结果

编译语句

`sh cmd.sh`

运行指令：

```
./parser Test/test1.cmm out1.s
```

`test1.cmm->out1.s`

```
.data
_prompt: .asciiz "Enter an integer:"
_ret: .asciiz "\n"
.globl main
.text
read:
  li $v0, 4
  la $a0, _prompt
  syscall
  li $v0, 5
  syscall
  jr $ra

write:
  li $v0, 1
  syscall
  li $v0, 4
  la $a0, _ret
  syscall
  move $v0, $0
  jr $ra

main:
  addi $sp, $sp, -44
  sw $ra, 40($sp)
  sw $fp, 36($sp)
  addi $fp, $sp, 44
  lw $t0, -12($fp)
  li $t1, 0
  move $t0, $t1
  sw $t0, -12($fp)
  lw $t0, -16($fp)
  li $t1, 1
  move $t0, $t1
  sw $t0, -16($fp)
  lw $t0, -20($fp)
  li $t1, 0
  move $t0, $t1
  sw $t0, -20($fp)
  jal read
  move $t0, $v0
  sw $t0, -24($fp)
  lw $t0, -28($fp)
  lw $t1, -24($fp)
```

```
  move $t0, $t1
  sw $t0, -28($fp)
label1:
  lw $t0, -20($fp)
  lw $t1, -28($fp)
  bge $t0, $t1, label2
  lw $t1, -12($fp)
  lw $t2, -16($fp)
  add $t0, $t1, $t2
  sw $t0, -32($fp)
  lw $t0, -36($fp)
  lw $t1, -32($fp)
  move $t0, $t1
  sw $t0, -36($fp)
  lw $t0, -16($fp)
  move $a0, $t0
  jal write
  lw $t0, -40($fp)
  li $t1, 0
  move $t0, $t1
  sw $t0, -40($fp)
  lw $t0, -12($fp)
  lw $t1, -16($fp)
  move $t0, $t1
  sw $t0, -12($fp)
  lw $t0, -16($fp)
  lw $t1, -36($fp)
  move $t0, $t1
  sw $t0, -16($fp)
  lw $t1, -20($fp)
  addi $t0, $t1, 1
  sw $t0, -44($fp)
  lw $t0, -20($fp)
  lw $t1, -44($fp)
  move $t0, $t1
  sw $t0, -20($fp)
  j label1
label2:
  move $v0, $zero
  move $sp, $fp
  lw $ra, -4($fp)
  lw $fp, -8($fp)
  jr $ra
```

`SPIM Simulator`运行结果：

```
zangzhuli@ubuntu:~/Desktop/byyl/5-mbdm/Our$ spim -file out1.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter an integer:9
1
1
2
3
5
8
13
21
34
```

`test2.cmm->out2.s`

```
.data
_prompt: .asciiz "Enter an integer:"
_ret: .asciiz "\n"
```

```
.globl main
.text
read:
  li $v0, 4
  la $a0, _prompt
  syscall
  li $v0, 5
  syscall
  jr $ra

write:
  li $v0, 1
  syscall
  li $v0, 4
  la $a0, _ret
  syscall
  move $v0, $0
  jr $ra

main:
  addi $sp, $sp, -28
  sw $ra, 24($sp)
  sw $fp, 20($sp)
  addi $fp, $sp, 28
  jal read
  move $t0, $v0
  sw $t0, -12($fp)
  lw $t0, -16($fp)
  lw $t1, -12($fp)
  move $t0, $t1
  sw $t0, -16($fp)
  lw $t0, -16($fp)
  li $t1, 1
  ble $t0, $t1, label3
  addi $sp,$sp,-4
  lw $s0, -16($fp)
  sw $s0, 0($sp)
  jal fact
  sw $v0, -20($fp)
  lw $t0, -24($fp)
  lw $t1, -20($fp)
  move $t0, $t1
  sw $t0, -24($fp)
  j label4
label3:
  lw $t0, -24($fp)
  li $t1, 1
  move $t0, $t1
  sw $t0, -24($fp)
label4:
  lw $t0, -24($fp)
  move $a0, $t0
  jal write
  lw $t0, -28($fp)
  li $t1, 0
  move $t0, $t1
  sw $t0, -28($fp)
  move $v0, $zero
  move $sp, $fp
  lw $ra, -4($fp)
```

```
  lw $fp, -8($fp)
  jr $ra

fact:
  addi $sp, $sp, -20
  sw $ra, 16($sp)
  sw $fp, 12($sp)
  addi $fp, $sp, 20
  lw $t0, 0($fp)
  li $t1, 1
  bne $t0, $t1, label1
  lw $t0, 0($fp)
  move $v0, $t0
  move $sp, $fp
  lw $ra, -4($fp)
  lw $fp, -8($fp)
  jr $ra
  j label2
label1:
  lw $t1, 0($fp)
  li $t2, 1
  sub $t0, $t1, $t2
  sw $t0, -12($fp)
  addi $sp,$sp,-4
  lw $s0, -12($fp)
  sw $s0, 0($sp)
  jal fact
  sw $v0, -16($fp)
  lw $t1, 0($fp)
  lw $t2, -16($fp)
  mul $t0, $t1, $t2
  sw $t0, -20($fp)
  lw $t0, -20($fp)
  move $v0, $t0
  move $sp, $fp
  lw $ra, -4($fp)
  lw $fp, -8($fp)
  jr $ra
label2:
```

`SPIM Simulator`运行结果：

```
zangzhuli@ubuntu:~/Desktop/byyl/5-mbdm/Our$ spim -file out2.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter an integer:10
3628800
```

实验反思

最后的代码有些冗余，尝试进行修改，但是由于能力问题加时间问题，最后只能做到无函数的时候出来比较好的结果，在函数的地方碰到了问题，一个是函数参数问题，由于`PARAM`和`ARG`对应形成的中间代码的变量名不同，个人认为涉及一个形参和实参对应的问题，尝试时使用存取`ARG`，用`PARAM`去寻找对应参数寄存器进行对应替换，由于中间代码中实参是深搜得到但是形参是循环得到，因此还涉及一个参数顺序的问题，新加一个栈用来寻找...最后实在是有些复杂，只好作罢，继续使用原来的代码。

对于样例以及自编样例而言，原来的代码虽然冗余加简单，但是可用性比较好。

实验终于结束！没有完成进一步的修改感到遗憾，但是这一个过程中也学习到了不少东西，代码能力也提高了一些。