编译原理实验报告

实验二 [二代编译器]

学号： 202111081075 | 姓名： 施懿航 | 日期： 2024.04.14

一、实验要求

二代编译器将一种语法类似 C 语言的语句序列翻译为等价的汇编程序，所输出的汇编程序符合 X86 或 MIPS 汇编语言格式要求。与 lab 1 不同的是，lab 1 提供了一个汇编的框架，用于管理程序执行入口点、函数栈帧等，这次二代编译器需要自行生成相关汇编代码。

二代编译器能够处理的文法如下所示：

关键字： int, return, main

标识符： 符合 C89 标准的标识符（[A-Za-z_][0-9A-Za-z_]*）

常量： 十进制整型，如 1、223、10 等

赋值操作符： =

运算符： + - * / % < <= > >= == != & | ^

标点符号： ; { } ( )

语句：

- 变量声明（单变量且无初始化） int a;
- 简单表达式赋值语句 a = b&1;
- 复杂表达式赋值语句（仅限等级二）a = (d+b&1)/(e!=3^b/c&&d);
- return 语句 return 0;
- 函数调用（只需支持预置函数） println_int(a);

函数定义：只需支持定义 main 函数即可

- 不带参数 int main(){...}
- 带参数 int main(int argc, int argv){...}

预置函数：只需支持对预置函数的调用即可。println_int(int a)与 C 语言中 printf("%d\n", a)有相同输出。

二、实验分工

[施懿航]——[生成代码部分] — [实现新增运算符和正确实现优先级的复杂表达式赋值语句]

[曾永丹]——[生成代码部分] — [实现新增运算符和正确实现优先级的复杂表达式赋值语句]

[刘柯妤]——[生成代码部分] — [函数定义部分和预制函数以及汇编代码框架的生成]

[马可圆]——[词法分析部分] — [使用flex进行词法分析]

三、实验设计

实现功能

本次实验在初代编译器的基础上，新增了以下功能：

1. 主函数定义：识别main，并正确编译函数入口
2. 标识符：符合 C89 标准的标识符（[A-Za-z_][0-9A-Za-z_]*）
3. 运算符：新增% < <= > >= == != & | ^，且能处理复杂表达式中区分运算符优先级，实现正确运算
4. 标点符号：{ } ( )
5. 预置函数： println_int(a);

如何实现

我主要负责的工作是新增运算符的处理和复杂表达式的计算：思路是先将读入的表达式转换为后缀表达式，再对后缀表达式进行处理求值。

一：分析从中缀表达式转为后缀表达式的过程：

中缀表达式转后缀表达式步骤：

1. 初始化两个栈：
    - 运算符栈：s1
    - 中间结果栈：s2
2. 从左到右扫描中缀表达式
3. 遇到操作数时，将其压入 s2
4. 遇到运算符时

    比较 它 与 s1 栈顶运算符的优先级：

    1. 如果 s1 为空，或则栈顶运算符号为 `(` ，则将其压入符号栈 s1

2. 如果：优先级比栈顶运算符 **高**，也将其压入符号栈 s1
3. 如果：优先级比栈顶运算符 **低 或 相等**，将 s1 栈顶的运算符 **弹出**，并压入到 s2 中

再重复第 4.1 步骤，与新的栈顶运算符比较（因为 4.3 将 s1 栈顶运算符弹出了）

这里重复的步骤在实现的时候有点难以理解，下面进行解说：

1. 如果 **s1 栈顶符号** 优先级比 **当前符号 高或则等于**，那么就将其 **弹出**，压入 s2 中（循环做，是只要 s1 不为空）

   如果栈顶符号为 `(`，优先级是 -1，就不会弹出，就跳出循环了
2. 跳出循环后，则将当前符号压入 s1

5. 遇到括号时：
   1. 如果是左括号 `(` ：则直接压入 s1
   2. 如果是右括号 `)`：

      则依次弹出 s1 栈顶的运算符，并压入 s2，直到遇到 **左括号** 为止，此时将这一对括号 **丢弃**
6. 重复步骤 2 到 5，直到表达式最右端
7. 将 s1 中的运算符依次弹出并压入 s2
8. 依次弹出 s2 中的元素并输出，结果的 **逆序** 即为：中缀表达式转后缀表达式

二：分析后缀表达式求值的过程：

1. 从 **左到右** 扫描表达式
2. 遇到 **数字** 时，将数字压入堆栈
3. 遇到 **运算符** 时

   弹出栈顶的两个数（栈顶和次顶），用运算符对它们做相应的计算，并将结果入栈。

   计算顺序是：**后** 弹出来的 (运算符) **先** 弹出来的

然后重复以上步骤，直到表达式的最右端，最后运算出的值则是表达式的值。

> 比如：`(3+4)*5-6` 对应的后缀表达式 `3 4 + 5 * 6 -`
>
> 1. 从左到右扫描，将 3、4 压入堆栈
> 2. 扫描到 `+` 运算符时
>
>   将弹出 4 和 3，计算 `3 + 4 = 7`，将 7 压入栈
> 3. 将 5 入栈
> 4. 扫描到 `*` 运算符时
>
>   将弹出 5 和 7 ，计算 `7 x 5 = 35`，将 35 入栈
> 5. 将 6 入栈
> 6. 扫描到 `-` 运算符时
>
>   将弹出 6 和 35，计算 `35 - 6 = 29`，将 29 压入栈
> 7. 扫描表达式结束，29 是表达式的值

可以得出生成汇编代码的思路：从左至右每读入一个表达式字符，首先判断是否为标识符，如果是标识符则替换为标识符已经赋值的立即数，再判断是数字还是运算符，如果是数字，则将其压入内存上开辟的sp栈空间，如果是运算符，则弹出栈中两个操作数，进行计算后，将结果压入栈，直至读完所有字符，最后弹出栈顶的值，即为表达式运算结果，将其传入寄存器，赋值。

代码解释

代码中定义以下函数：

```
// 判断是否为运算符，如果是，返回true
bool isOperator(TokenType type) {
   return  type == TokenType::LESS_THAN || type == TokenType::LESS_THAN_EQUAL ||
           type == TokenType::GREATER_THAN || type == TokenType::GREATER_THAN_EQUAL ||
           type == TokenType::EQUAL || type == TokenType::NOT_EQUAL||
           type == TokenType::PLUS_OP || type == TokenType::MINUS_OP ||
           type == TokenType::MULTIPLY_OP || type == TokenType::DIVIDE_OP ||
           type == TokenType::MODULO_OP || type == TokenType::AND_OP ||
           type == TokenType::OR_OP || type == TokenType::XOR_OP ;

}


// 根据c语言中运算优先级定义运算符优先级，数字越大优先级越高
int getPrecedence(TokenType type) {
    switch (type) {
        case TokenType::MULTIPLY_OP:
        case TokenType::DIVIDE_OP:
        case TokenType::MODULO_OP:
            return 7;
        case TokenType::PLUS_OP:
        case TokenType::MINUS_OP:
            return 6;
        case TokenType::LESS_THAN:
        case TokenType::LESS_THAN_EQUAL:
        case TokenType::GREATER_THAN:
        case TokenType::GREATER_THAN_EQUAL:
            return 5;
        case TokenType::EQUAL:
        case TokenType::NOT_EQUAL:
            return 4;
        case TokenType::AND_OP:
            return 3;
        case TokenType::XOR_OP:
            return 2;
        case TokenType::OR_OP:
            return 1;
        default:
            return 0; // 对于未知类型或不适用的类型，返回最低优先级
```

```
    }
}
```

以上两个函数是为处理表达式做准备。

```
// 将中缀表达式转换为后缀表达式
vector<Token> infixToPostfix(const vector<Token>& infix) {
    stack<Token> opStack;
    vector<Token> postfix;
    for (const Token& token : infix) {
        if (token.type == TokenType::IDENTIFIER || token.type ==
TokenType::INTEGER_LITERAL) {
            postfix.push_back(token);
        } else if (token.type == TokenType::LEFT_PAREN) {
            opStack.push(token);
        } else if (token.type == TokenType::RIGHT_PAREN) {
            while (!opStack.empty() && opStack.top().type != TokenType::LEFT_PAREN) {
                postfix.push_back(opStack.top());
                opStack.pop();
            }
            if (!opStack.empty()) opStack.pop(); // 弹出左括号
        } else if (isOperator(token.type)) {
            while (!opStack.empty() && getPrecedence(opStack.top().type) >=
getPrecedence(token.type)) {
                postfix.push_back(opStack.top());
                opStack.pop();
            }
            opStack.push(token);
        }
    }
    while (!opStack.empty()) {
        postfix.push_back(opStack.top());
        opStack.pop();
    }
        cout << "# Postfix Expression: ";
    for (const Token& token : postfix) {
        cout << token.value << " ";
    }
    cout << endl;
    return postfix;
}
```

这段代码实现了将中缀表达式转换为后缀表达式的功能，并在转换过程中输出了转换后的后缀表达式。

1. 首先，定义了一个用于存放运算符的栈`opStack`和一个用于存放后缀表达式的向量`postfix`。
2. 接着，对输入的中缀表达式进行遍历。对于每个token（操作数、运算符、括号等），根据其类型进行相应的处理：
    - 若为操作数（标识符或整数），直接将其加入到后缀表达式中。

- 若为左括号，将其压入运算符栈中。
- 若为右括号，则将栈顶的运算符依次弹出并加入到后缀表达式中，直到遇到左括号，将左括号弹出丢弃。
- 若为运算符，将栈中优先级大于等于当前运算符的运算符依次弹出并加入到后缀表达式中，然后将当前运算符压入栈中。

3. 最后，将栈中剩余的运算符依次弹出并加入到后缀表达式中，即可得到转换后的后缀表达式。同时，利用输出语句输出后缀表达式，便于调试和理解转换过程。

```
// 根据后缀表达式生成MIPS代码
void generateMIPSFromPostfix(const vector<Token>& postfix, const map<string, int>&
varMap, const string& resultVar) {

    for (const Token& token : postfix) {
        if (token.type == TokenType::IDENTIFIER) {
            cout << "    lw $t0, " << varMap.at(token.value) << "($fp)" << endl; // 加载
变量到$t0寄存器
            cout << "    sw $t0, 0($sp)" << endl; // 将$t0寄存器中的值存入内存栈
            cout << "addiu $sp, $sp, -4" << endl; // 栈指针下移4字节
        } else if (token.type == TokenType::INTEGER_LITERAL) {
            cout << "    li $t0, " << token.value << endl; // 将立即数加载到$t0寄存器
            cout << "    sw $t0, 0($sp)" << endl; // 将$t0寄存器中的值存入内存栈
            cout << "addiu $sp, $sp, -4" << endl; // 栈指针下移4字节
        } else if (isOperator(token.type)) {
            // 从内存栈中弹出两个操作数
            cout << "    lw $t1, 4($sp)" << endl;
            cout << "    lw $t2, 8($sp)" << endl;

            // 根据操作符生成相应的MIPS指令
            switch (token.type) {
                case TokenType::MULTIPLY_OP:
                    cout << "    mul $t0, $t1, $t2" << endl;
                    break;
                case TokenType::DIVIDE_OP:
                    cout << "    div $t1, $t2" << endl;
                    cout << "    mflo $t0" << endl;
                    break;
                case TokenType::MODULO_OP:
                    cout << "    div $t1, $t2" << endl;
                    cout << "    mfhi $t0" << endl;
                    break;
                case TokenType::PLUS_OP:
                    cout << "    add $t0, $t1, $t2" << endl;
                    break;
                case TokenType::MINUS_OP:
                    cout << "    sub $t0, $t1, $t2" << endl;
                    break;
                case TokenType::LESS_THAN:
```

```
                    cout << "    slt $t0, $t1, $t2" << endl;
                    break;
                case TokenType::LESS_THAN_EQUAL:
                    cout << "    slt $t0, $t2, $t1" << endl;
                    cout << "    xori $t0, $t0, 1" << endl;
                    break;
                case TokenType::GREATER_THAN:
                    cout << "    slt $t0, $t2, $t1" << endl;
                    break;
                case TokenType::GREATER_THAN_EQUAL:
                    cout << "    slt $t0, $t1, $t2" << endl;
                    cout << "    xori $t0, $t0, 1" << endl;
                    break;
                case TokenType::EQUAL:
                    cout << "    seq $t0, $t1, $t2" << endl;
                    break;
                case TokenType::NOT_EQUAL:
                    cout << "    sne $t0, $t1, $t2" << endl;
                    break;
                case TokenType::AND_OP:
                    cout << "    and $t0, $t1, $t2" << endl;
                    break;
                case TokenType::XOR_OP:
                    cout << "    xor $t0, $t1, $t2" << endl;
                    break;
                case TokenType::OR_OP:
                    cout << "    or $t0, $t1, $t2" << endl;
                    break;
                default:
                    // 操作符未实现
                    break;
            }
            // 将计算结果存入内存栈
            cout << "    sw $t0, 8($sp)" << endl;
            cout << "addiu $sp, $sp, 4" << endl; // 栈指针下移4字节
        }
    }

    // 将最终结果存储到目标变量
    cout << "    lw $t0, 4($sp)" << endl;
    cout << "    sw $t0, " << varMap.at(resultVar) << "($fp)" << endl;
    cout << "    addiu $sp, $sp, 4" << endl; // 恢复栈指针
}
```

这段代码实现了根据后缀表达式生成MIPS代码的功能。它遍历了后缀表达式中的每个token，并根据token的类型进行不同的处理：

1. 对于标识符和整数字面量，将其加载到$t0寄存器，并将寄存器中的值存入内存栈，然后栈指针下移4字节。

2. 对于运算符，从内存栈中弹出两个操作数（$t1$和t2寄存器），根据操作符生成相应的MIPS指令，并将计算结果存入内存栈，然后栈指针下移4字节。
3. 最后，将最终结果存储到目标变量中，并恢复栈指针。

这段代码主要使用了以下寄存器和栈操作：

1. 寄存器使用：
    - `$t0` 寄存器用于临时存储标识符或整数字面量的值，以及计算结果。
    - `$t1` 和 `$t2` 寄存器用于临时存储从栈中弹出的操作数的值。
2. 栈操作：
    - 对于标识符和整数字面量，先将值加载到 `$t0` 寄存器，然后将 `$t0` 寄存器中的值存入内存栈中。
    - 对于运算符，先从内存栈中弹出两个操作数（分别存储在 `$t1` 和 `$t2` 寄存器中），然后根据操作符生成相应的 MIPS 指令，计算结果存入 `$t0` 寄存器，最后将 `$t0` 寄存器中的值存入内存栈中。

具体的栈操作如下：

- 对于标识符或整数字面量的存储：
    1. `lw $t0, ...` 从内存加载值到 `$t0` 寄存器。
    2. `sw $t0, ...` 将 `$t0` 寄存器中的值存入内存栈中。
    3. `addiu $sp, $sp, -4` 栈指针下移 4 字节。
- 对于运算符的计算和结果存储：
    1. `lw $t1, 4($sp)` 从内存加载第一个操作数到 `$t1` 寄存器。
    2. `lw $t2, 8($sp)` 从内存加载第二个操作数到 `$t2` 寄存器。
    3. 根据不同的运算符生成相应的 MIPS 指令，计算结果存入 `$t0` 寄存器。
    4. `sw $t0, 8($sp)` 将 `$t0` 寄存器中的值存入内存栈中。
    5. `addiu $sp, $sp, 4` 栈指针上移 4 字节。

最后，将最终的计算结果存入目标变量中。

遇到的问题和解决过程

1. 第一次汇总代码后提交测评为8分，经调试检查，是编译器读到“==”比较符号，会先识别为赋值符号“=”，因此造成混淆。修改正则表达式部分代码后，达到9分。
2. 测试超过10个操作数的复杂表达式时发现，由于之前的代码中只使用10个寄存器保存计算中临时变量，因此会造成变量覆盖，导致结果错误。修改，将临时变量保存至内存。达到满分。

四、实验结果

测试样例1:

```
int
main
(

)
 {
    int
     _0893127890
     ;
    int
    hgbahjkf
    ;
    int
    or
    ;
    int
    and
    ;
    int
    xor
    ;

    _0893127890
    =
    114
    ;
    hgbahjkf
    =
    514
    ;
    or
    =
    _0893127890
    |
    hgbahjkf
    ;
    and
    =
    _0893127890
    &
    hgbahjkf
    ;
    xor
     =
     _0893127890
```

```
     ^
     hgbahjkf
     ;
    println_int
    (
        or
        )
        ;
    println_int
    (
        and
        )
        ;
    println_int
    (
        xor
        )
        ;
    return
    0
    ;
}
```

输出结果：

```
.data
newline: .asciiz "\n"
.text
.globl main
main:
move $fp, $sp # 设置帧指针
addiu $sp, $sp, -0x100 # 为局部变量分配栈空间
    sw $zero, -4($fp) # int _0893127890
    sw $zero, -8($fp) # int hgbahjkf
    sw $zero, -12($fp) # int or
    sw $zero, -16($fp) # int and
    sw $zero, -20($fp) # int xor
    # 开始处理表达式 _0893127890
    li $t0, 114
    sw $t0, -4($fp)
    # 开始处理表达式 hgbahjkf
    li $t0, 514
    sw $t0, -8($fp)
    # 开始处理表达式 or
# Postfix Expression: _0893127890 hgbahjkf |
    lw $t0, -4($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -8($fp)
```

```
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    or $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, 4($sp)
    sw $t0, -12($fp)
    addiu $sp, $sp, 4
    # 开始处理表达式 and
# Postfix Expression: _0893127890 hgbahjkf &
    lw $t0, -4($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -8($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    and $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, 4($sp)
    sw $t0, -16($fp)
    addiu $sp, $sp, 4
    # 开始处理表达式 xor
# Postfix Expression: _0893127890 hgbahjkf ^
    lw $t0, -4($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -8($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    xor $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, 4($sp)
    sw $t0, -20($fp)
    addiu $sp, $sp, 4
    lw $a0, -12($fp)
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, newline
    syscall
    lw $a0, -16($fp)
```

```
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, newline
    syscall
    lw $a0, -20($fp)
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, newline
    syscall
 li $v0, 0
move $v1, $v0
li $v0, 10
syscall
```

运行结果：

```
vboxuser@ubantu22:~/Documents/compiler_lab/lab2/project/cases$ spim -file mips.t
xt
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
626
2
624
```

测试样例2:

```
int main() {
  int a;
  int b;
  int c;
  int d;
  int e;
  int ans;
  a=1;
  b=2;
  c=3;
  d=4;
  e=5;
  ans = (f - g) * (h / i) % j | (k & l) ^ (m < n) <= (o >= p) & (q != r) == (s == t);
  println_int(ans);
  return ans;
}
```

输出结果：

```
.data
newline: .asciiz "\n"
.text
.globl main
main:
move $fp, $sp # 设置帧指针
addiu $sp, $sp, -0x100 # 为局部变量分配栈空间
    sw $zero, -4($fp) # int a
    sw $zero, -8($fp) # int b
    sw $zero, -12($fp) # int c
    sw $zero, -16($fp) # int d
    sw $zero, -20($fp) # int e
    sw $zero, -24($fp) # int f
    sw $zero, -28($fp) # int g
    sw $zero, -32($fp) # int h
    sw $zero, -36($fp) # int i
    sw $zero, -40($fp) # int j
    sw $zero, -44($fp) # int k
    sw $zero, -48($fp) # int l
    sw $zero, -52($fp) # int m
    sw $zero, -56($fp) # int n
    sw $zero, -60($fp) # int o
    sw $zero, -64($fp) # int p
    sw $zero, -68($fp) # int q
    sw $zero, -72($fp) # int r
    sw $zero, -76($fp) # int s
    sw $zero, -80($fp) # int t
    sw $zero, -84($fp) # int ans
    # 开始处理表达式 a
    li $t0, 1
    sw $t0, -4($fp)
    # 开始处理表达式 b
    li $t0, 2
    sw $t0, -8($fp)
    # 开始处理表达式 c
    li $t0, 3
    sw $t0, -12($fp)
    # 开始处理表达式 d
    li $t0, 4
    sw $t0, -16($fp)
    # 开始处理表达式 e
    li $t0, 5
    sw $t0, -20($fp)
    # 开始处理表达式 f
    li $t0, 6
    sw $t0, -24($fp)
    # 开始处理表达式 g
    li $t0, 7
    sw $t0, -28($fp)
```

```
    # 开始处理表达式 h
    li $t0, 8
    sw $t0, -32($fp)
    # 开始处理表达式 i
    li $t0, 9
    sw $t0, -36($fp)
    # 开始处理表达式 j
    li $t0, 10
    sw $t0, -40($fp)
    # 开始处理表达式 k
    li $t0, 11
    sw $t0, -44($fp)
    # 开始处理表达式 l
    li $t0, 12
    sw $t0, -48($fp)
    # 开始处理表达式 m
    li $t0, 13
    sw $t0, -52($fp)
    # 开始处理表达式 n
    li $t0, 14
    sw $t0, -56($fp)
    # 开始处理表达式 o
    li $t0, 15
    sw $t0, -60($fp)
    # 开始处理表达式 p
    li $t0, 16
    sw $t0, -64($fp)
    # 开始处理表达式 q
    li $t0, 17
    sw $t0, -68($fp)
    # 开始处理表达式 r
    li $t0, 18
    sw $t0, -72($fp)
    # 开始处理表达式 s
    li $t0, 19
    sw $t0, -76($fp)
    # 开始处理表达式 t
    li $t0, 20
    sw $t0, -80($fp)
    # 开始处理表达式 ans
# Postfix Expression: f g - h i / * j % k l & m n < o p >= <= q r != s t == == & ^ |
    lw $t0, -24($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -28($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
```

```
    sub $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, -32($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -36($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    div $t1, $t2
    mflo $t0
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    mul $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, -40($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    div $t1, $t2
    mfhi $t0
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, -44($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -48($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    and $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, -52($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -56($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    slt $t0, $t1, $t2
```

```
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, -60($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -64($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    slt $t0, $t1, $t2
    xori $t0, $t0, 1
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    slt $t0, $t2, $t1
    xori $t0, $t0, 1
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, -68($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -72($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    sne $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, -76($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t0, -80($fp)
    sw $t0, 0($sp)
addiu $sp, $sp, -4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    seq $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    seq $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
```

```
    and $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    xor $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    or $t0, $t1, $t2
    sw $t0, 8($sp)
addiu $sp, $sp, 4
    lw $t0, 4($sp)
    sw $t0, -84($fp)
    addiu $sp, $sp, 4
    lw $a0, -84($fp)
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, newline
    syscall
    lw $v0, -84($fp)
```

运行结果：

```
vboxuser@ubantu22:~/Documents/compiler_lab/lab2/lab2-9$ spim -file mips.txt
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
8
```

运行结果正确

五、实验总结与反思

[实验中的一些思考点以及个人总结]

在处理表达式转换和生成 MIPS 代码的过程中，关键的思考点和个人总结如下：

1. **中缀表达式转换为后缀表达式**：在转换过程中，需要考虑操作符的优先级和结合性。使用栈来保存操作符，并根据优先级决定何时弹出操作符到后缀表达式中。
2. **生成 MIPS 代码**：在生成代码时，需要考虑如何利用寄存器和内存栈来处理变量和运算。合理利用寄存器可以减少内存访问次数，提高运行效率。
3. **栈溢出问题**：在使用内存栈时，需要小心栈溢出问题。及时调整栈指针以保证栈空间足够。
4. **错误处理**：在处理表达式和生成代码时，需要考虑如何处理错误情况，例如非法操作符、缺少操作数等。
5. **调试和测试**：对于复杂的代码逻辑，需要进行调试和测试，可以通过输出中间结果或使用调试工具来验证代码的正确性。

6. **优化**：在生成代码时，可以考虑一些优化策略，如减少内存访问次数、利用寄存器重用等，以提高代码效率。