编译原理实验报告

实验一 [初代编译器]

学号： 202111081075 姓名： 施懿航 日期： 2 0 2 4 年3月17 日

1. 实验要求

初代编译器将 C 语言顺序语句序列翻译为等价的汇编程序，所输出的汇编程序符合MIPS 汇编语言格式要求，能够被后续的汇编器翻译为可执行程序运行。目标平台为 MIPS，生成的汇编程序可以在 MIPS 模拟器中运行和验证结果。

初代编译器能够处理的文法如下所示：

关键字：int, return

标识符：单个英文字母，如 a、b 等

常量：十进制整型，如 1、223、10 等

操作符：=、+、-、\*、/

分隔符： ；

语句：表达式语句、赋值语句

二、实验分工

[马可圆]——[tokens和main部分]

[刘柯妤]——[生成代码部分]

[施懿航]——[生成代码部分]

[曾永丹]——[生成代码部分]

三、实验设计

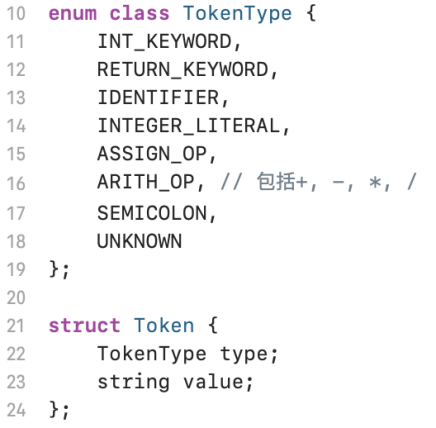
###### 实现功能

本次实验实现了MIPS初代编译器，功能包括：

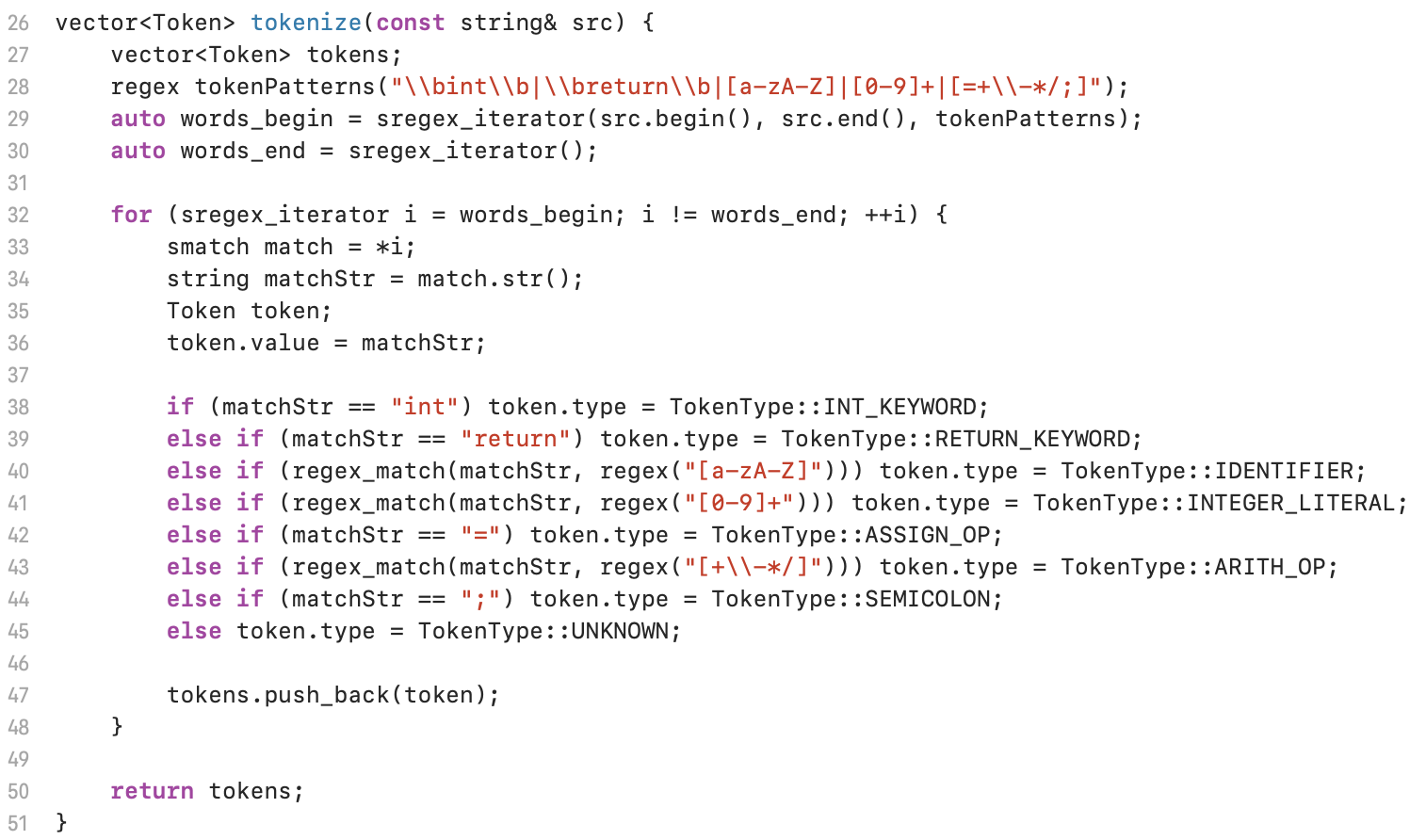
1. 识别关键字：int和return，从而正确编译整型变量声明语句
2. 识别标识符：单个英文字母，可以区分大小写，a-z，A-Z
3. 识别常量：十进制整型
4. 识别操作符：“=”和加减乘除运算符，但不区分运算符优先级，能够实现正确编译运算表达式语句和赋值语句
5. 识别分隔符：；

###### 代码解释（如何实现）

首先，词法分析器部分定义了结构体Token，用于存储源代码中单词对应的类别：



传入的源代码作为字符串经过tokenize函数后，返回包含所有解析出的 token 的 tokens 容器：



接下来进入生成代码部分，也就是generateMIPS函数部分：

void generateMIPS(const vector<Token>& tokens) {

int variableOffset = 0; // 初始化变量偏移量为 0

map<string, int> varMap; // 创建一个映射，用于存储变量名和其在栈中的偏移量

// 遍历 tokens 向量

for (size\_t i = 0; i < tokens.size(); ++i)

代码主要思路是通过读入的token序列判断源语句类型（声明语句、赋值语句、表达式语句），再转换为对应的汇编代码逻辑，判断分支如下：

**1. INT\_KEYWORD 分支：**

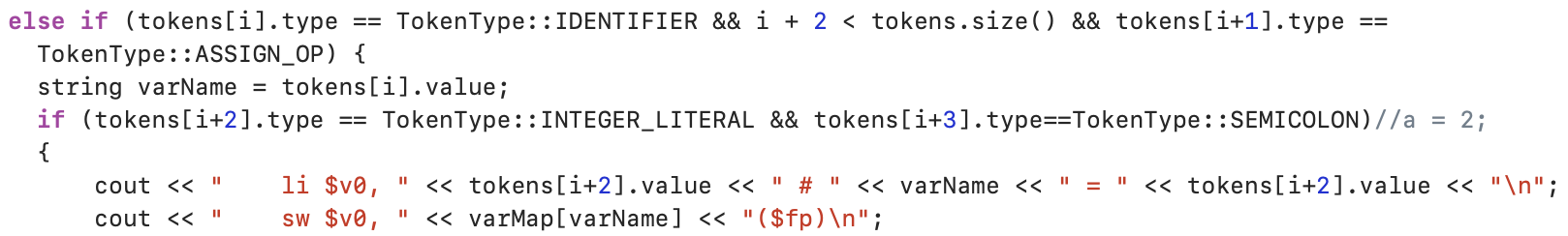
- 当遇到 `int` 关键字时，表示声明变量。通过减去 `variableOffset` 来为每个变量分配栈空间，并将变量名及其偏移量存储在 `varMap` 中。然后输出对应的 MIPS 汇编代码，将零值存储在对应的偏移量处。

- `i` 增加 2 是因为需要跳过声明语句中的标识符和分号。

**2. IDENTIFIER 分支：**

- 当遇到标识符时，可能是变量赋值语句或者表达式的一部分。

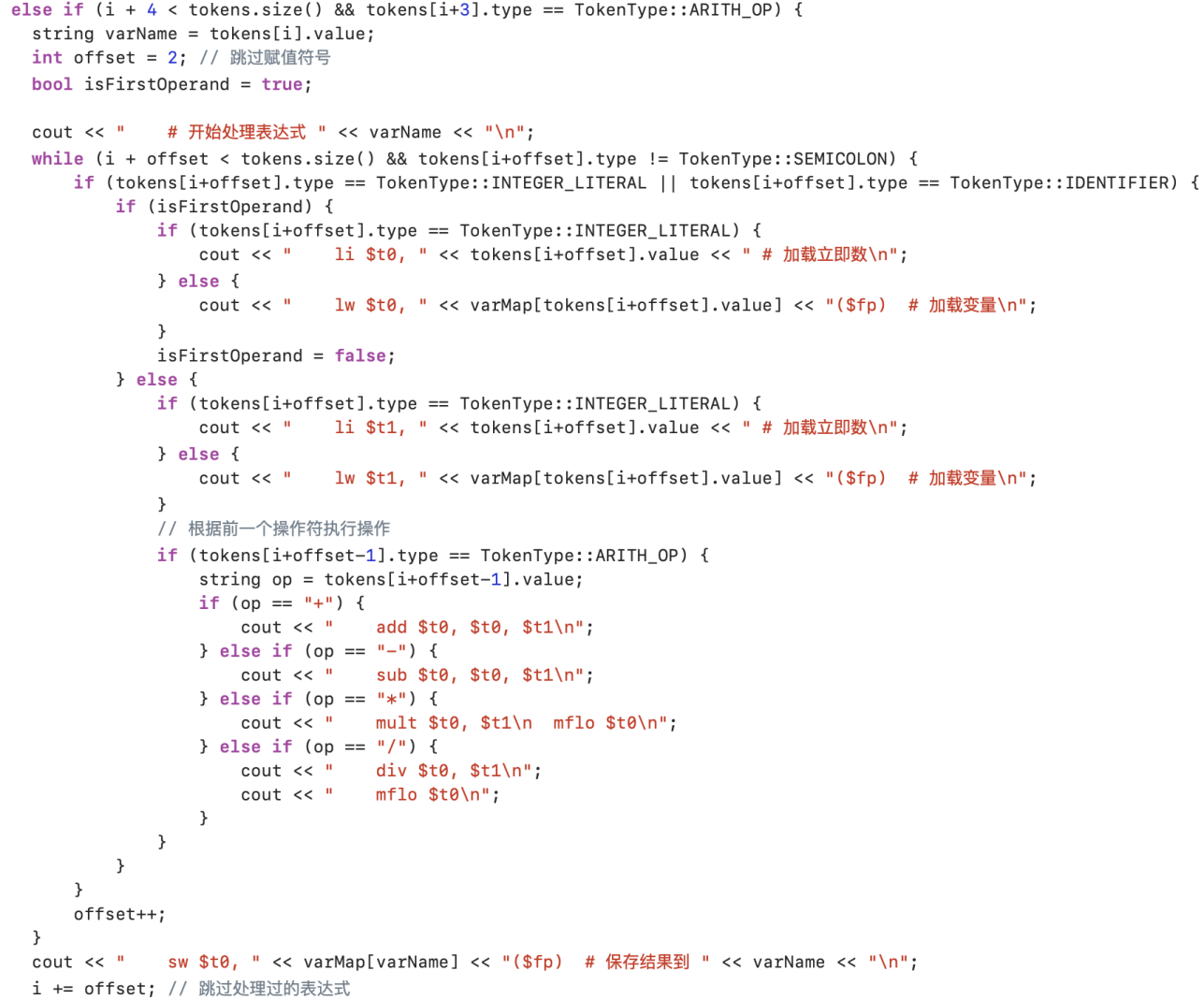
- 如果标识符后跟着赋值符号和整数字面量以及分号，说明是**简单的赋值语句**。生成相应的 MIPS 汇编代码，将整数值存储在变量的偏移量处：



- 如果标识符后跟着赋值符号和算术表达式，则需要进行复杂的**表达式计算**。根据表达式的不同部分，逐步生成 MIPS 汇编代码，最后将结果存储在变量的fp偏移量处。

这一部分我们组进行了三次迭代，刘柯妤同学使用sp栈实现了两个操作数的运算功能，在此基础上我没有使用sp栈，只利用寄存器，实现了多个操作数运算，最后曾永丹同学增加了分支条件判断，实现只有一个操作数以及变量声明时的其他情况。

以下是实现多个操作数运算的代码部分：



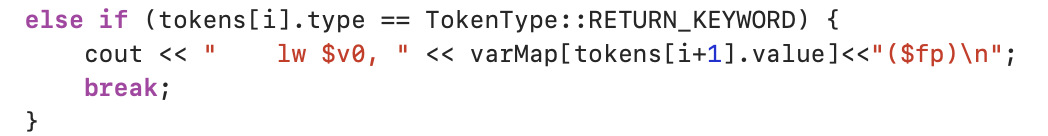
当判断标识符后三位是一个运算符，说明这是一个表达式运算语句，将该标识符存入varName，跳过赋值符号，并设置一个bool变量用于判断是第几个操作数。

使用while循环，直到读到分号结束，每读入一个token，判断是常量还是已存储的变量，如果是常量则加载立即数，如果是变量则先加载其存储在fp对应偏移量处的值，再进行对应运算操作。t0存储第一个操作数，t1存储第二个操作数。运算后结果从t0保存至对应偏移量。

值得注意的是，乘法和除法的实现需要先mult/div $t0,$t1，再mflo $t0

- 如果标识符后跟着赋值符号，但后续不是整数字面量或算术表达式，则说明赋值语句不合法。

**3. RETURN\_KEYWORD 分支：**

 - 当遇到 `return` 关键字时，表示函数返回。输出相应的 MIPS 汇编代码，将返回值从栈中加载到 `$v0` 寄存器中，并终止循环。

generateMIPS函数通过检查不同的 token 类型及其后续的 token，实现了对不同类型语句的识别和处理，并生成相应的 MIPS 汇编代码。

###### 遇到的问题和解决过程

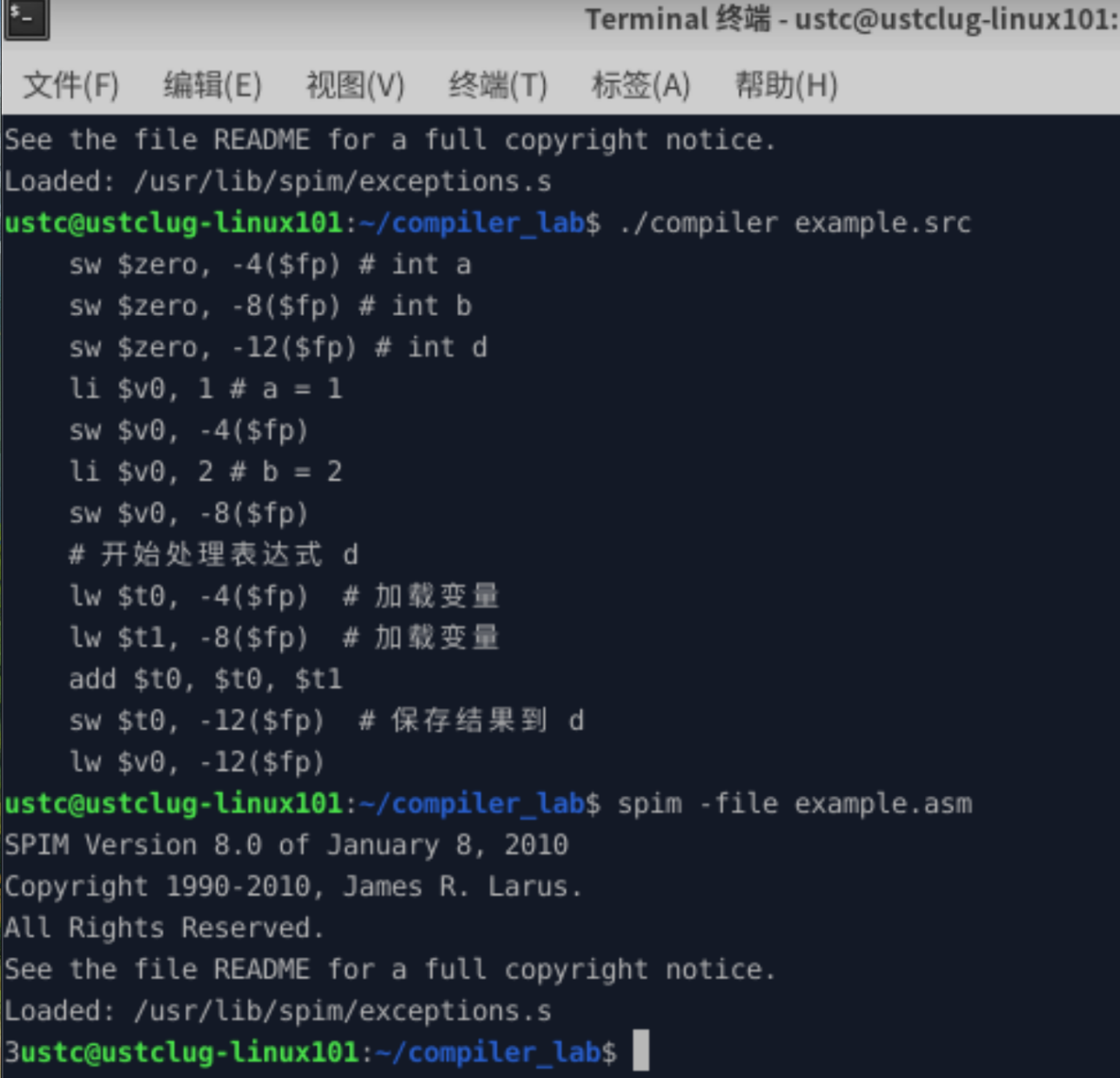
1. 一开始我们组想使用sp栈进行运算操作，一次压入两个操作数入栈，但是由于没有理解清晰逻辑，因此只能进行两个操作数的运算，其实是因为没有将运算结果保存回变量内存，造成丢失，但是我们组没有在栈的基础上改，而是选择只使用寄存器进行运算操作，通过使用while循环可以读取多个操作数，但根本上还是只进行两个数的运算，只不过每次累加，虽然这种方法解决了这次实验，但由于没有使用栈在之后的实验需要考虑运算符优先级等情况就无法解决，下次再进行改进。
2. 还有一些语法bug，例如乘法和除法，要添加mflo语句，以及空格的形式，都在调试中才发现错误并解决。

四、实验结果

[给定样例的测试结果]

测试样例1:

int a ; int b ; int d ; a = 1 ; b = 2 ; d = a + b ; return d ;

输出结果：

测试样例2:

int a ;

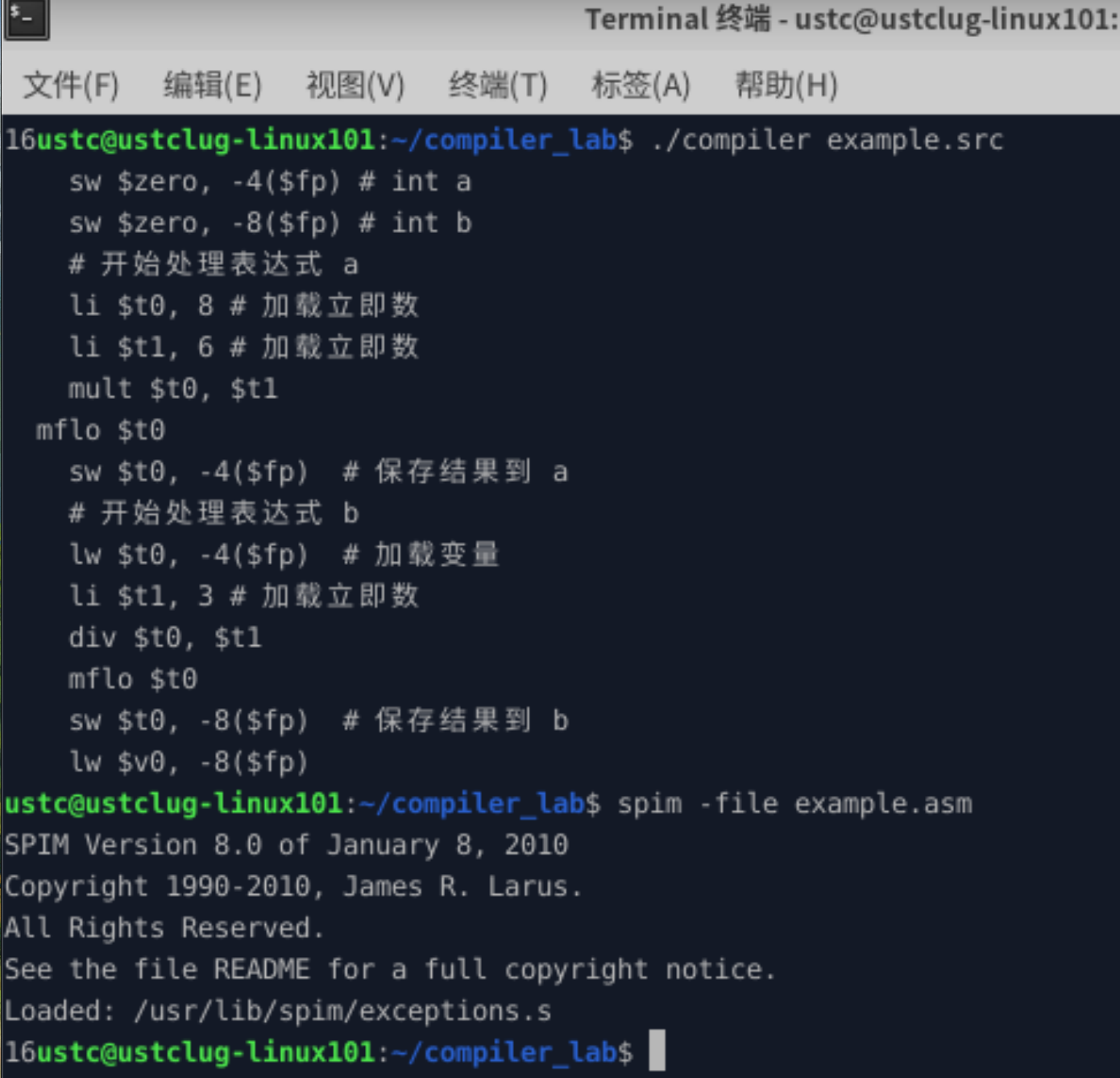
int b ;

a = 8 \* 6 ;

b = a / 3 ;

return b ;

输出结果：



五、实验总结与反思

[实验中的一些思考点以及个人总结]

1. 对于不考虑优先级的运算，只需要寄存器内存之间互相加载就可以实现，但如果考虑优先级则需要使用栈。
2. 只有一个INT类型的关键字，因此每次偏移量-4，但如果更多类型则需要使用数组设置对应偏移量。
3. 本次实验框架中

main:

move $fp, $sp # 设置帧指针

addiu $sp, $sp, -0x100 # 为局部变量分配栈空间

设置了256字节的空间，注意栈空间是从高地址到低地址。

1. sp和fp指针：

栈指针 `$sp`：

- 栈指针指向当前栈顶，它随着栈的操作而变化。当新的数据被压入栈时，栈指针向下移动；当数据从栈中弹出时，栈指针向上移动。在 MIPS 中，栈是向下生长的，即栈顶地址减小。

- 栈指针的初始值由操作系统设置。在函数调用时，通常会将栈指针移动到未使用的内存位置，以便为该函数的局部变量和临时数据分配空间。

帧指针 `$fp`：

- 帧指针通常用于访问函数的参数和局部变量。它指向当前函数的栈帧（Stack Frame）的底部。栈帧包含了函数调用时保存的重要信息，如旧的帧指针、返回地址以及函数的局部变量。

- 在 MIPS 中，帧指针的值通常在函数入口处被保存，然后在函数退出时被恢复。它提供了一个固定的参考点，使得访问局部变量和参数的地址更为简单。

栈帧设置的作用：

- 当一个函数被调用时，通常会在栈上为其分配一个栈帧，其中包含了该函数的局部变量、参数以及其他相关信息。

- 在函数入口处，会将当前的 `$fp` 保存到栈上，并将栈指针 `$sp` 移动到一个新的位置，以为该函数的栈帧分配空间。然后，将 `$fp` 设置为新的栈顶，用于访问该函数的局部变量和参数。

局部变量和参数的访问：

- 通过帧指针 `$fp` 可以访问当前函数的局部变量和参数。局部变量通常位于帧指针的负偏移量处，而参数通常位于帧指针的正偏移量处。

- 访问局部变量时，可以通过将帧指针 `$fp` 加上相应的偏移量来获取局部变量的地址。

- 访问函数参数时，同样可以通过帧指针 `$fp` 加上相应的偏移量来获取参数的地址。在 MIPS 汇编中，参数通常通过寄存器 `$a0` - `$a3` 传递，以及栈上传递。

栈空间的管理：

- 在函数调用过程中，栈指针 `$sp` 会动态地移动，以分配和释放栈空间。当函数被调用时，栈指针会向下移动以为新的栈帧分配空间；当函数返回时，栈指针会向上移动以释放之前分配的空间。