**注意：本实验代码所用编码为GB2312**

1. **源语言的描述**

基于C语法设计的源语言LittleC

（1）结构和语义说明：

a. 该语言的一个程序由且仅由一个块结构组成，该块中包含两部分：可选的声明语句和必须的执行语句。声明语句用于声明变量，执行语句用于表达计算过程，过程中使用这些变量来存储内部数据和计算结果。

b. 该语言支持两种数据类型：整型和布尔型。整型数据占4个字节，布尔型占1个字节。整型数据的常量是整常数，布尔型数据的常量是 true 和 false 。

c. 整型数据支持 +、-、\*、/ 四种算术运算，运算结果为整型值。整型数据可与>、>=...等六个关系运算符结合，运算结果为布尔型值。

d. 布尔型数据支持 || 、&& 、！三种逻辑运算，运算结果为布尔常量。

e. 整型的变量和常量可以用算术运算符组合成算术表达式，其值为整常数；布尔型的变量和常量可以用逻辑运算符组成成布尔表达式，其值为布尔常量；算术表达式可以加上关系运算符构造出布尔表达式。

f. 整常数和算术表达式可以用于整型变量赋值，布尔常量和布尔表达式可以用于布尔型变量赋值；两者不兼容（文法中使用不同的赋值运算符以示区别）。

g. 各种运算符的优先级和结合规则参照C语言语法理解。

h. 控制流语句中的控制条件限定为1个布尔型变量，该量应该在引用之前定值。

i. 该语言不支持数组、结构体、指针等复杂数据类型。

j. 该语言不含有子程序结构，也不支持过程调用。

k. 该语言有简单的I/O功能：使用read指令能够从键盘读入1个整常数，使用write指令能向屏幕打印输出1个整常数。

（2）文法定义：

PROG   → { DECLS STMTS  }

DECLS  → DECLS DECL | empty

DECL   → int  NAMES  ; | bool  NAMES  ;

NAMES  → NAMES , NAME  | NAME

NAME   → id

STMTS  → STMTS STMT   | STMT

STMT   → id  = EXPR ; | id: = BOOL;

STMT   → if id  then STMT

STMT   → if  id  then STMT  else STMT

STMT   → while  id  do STMT

STMT   → { STMTS  STMT   }

STMT   → read id ;

STMT   → write id ;

EXPR   → EXPR ADD TERM   | TERM

ADD    → + | -

TERM   → TERM MULNEGA  | NEGA

MUL    → \* |/

NEGA   → FACTOR  | -FACTOR

FACTOR → (EXPR ) | id  | number

BOOL  → BOOL  || JOIN  | JOIN

JOIN   → JOIN  && NOT   | NOT

NOT    → REL  | !REL

REL    → EXPR  ROP EXPR

ROP    → > | >= | < | <= | == |  !=。

在改写成LL1文法之后：  
PROG → DECLS STMTS

DECLS → DECL DECLS' | ε

DECLS' → DECL DECLS' | ε

DECL → int NAMES ; | bool NAMES ;

NAMES → NAME NAMES'

NAMES' → , NAME NAMES' | ε

NAME → id

STMTS → STMT STMTS' | ε

STMTS' → STMT STMTS' | ε

STMT → id STMT' | = EXPR; | := BOOL; | if id then STMT ELSESTMT | while id do STMT | { STMTS } | read id ; | write id ;

ELSESTMT → else STMT | ε

EXPR → TERM ADD'

ADD' → ADD TERM ADD' | ε

ADD → + | -

TERM → NEGA MUL'

MUL' → MUL NEGA MUL' | ε

MUL → \* | /

NEGA → FACTOR | - FACTOR

FACTOR → ( EXPR ) | id | number

BOOL → JOIN BOOL'

BOOL' → || JOIN BOOL' | ε

JOIN → NOT JOIN'

JOIN' → && NOT JOIN' | ε

NOT → REL | ! REL

REL → EXPR ROP EXPR

ROP → > | >= | < | <= | == | !=

1. **目标语言——mips汇编描述**

MIPS是一种基于RISC（Reduced Instruction Set Computing）理念设计的处理器架构，它的汇编语言具有简洁、统一的特点。

MIPS汇编语言有以下主要特性：

简单的指令集：MIPS汇编语言的指令集相对较小，大部分指令都是三地址格式，即两个源操作数和一个目的操作数。这使得MIPS指令集结构清晰、简单，便于编程和理解。

寄存器：MIPS架构具有32个通用寄存器，每个寄存器都是32位。这些寄存器在程序中被用来存储中间结果、变量值等数据。

支持的数据类型：MIPS汇编语言支持整数和浮点数数据类型。由于LittleC源语言只支持整数和布尔类型，我们将布尔值转换为整数0和1来处理。

支持的操作：MIPS汇编语言支持各种算术和逻辑操作，包括加法、减法、乘法、除法、比较等。这些操作可以直接对应到LittleC源语言的算术运算和布尔运算。

控制流语句：MIPS汇编语言支持跳转和分支指令，可以实现LittleC源语言的if-else、while等控制流语句。

I/O操作：MIPS汇编语言提供了系统调用的机制来实现I/O操作。使用系统调用可以实现整数、字符串的读取输入和输出打印等功能。

以下是一部分mips语法:  
注释：在 MIPS 中，"#" 开头的内容都是注释，例如 # This is a comment.

指令：MIPS 的指令通常有三个部分，操作符和操作数。操作符说明了要执行的操作，而操作数是操作的对象。例如，add $t0, $t1, $t2 指示将 $t1 和 $t2 中的值相加，然后将结果存储在 $t0 中。

标签：标签是程序中的一个标记，用于在跳转和分支指令中引用特定的位置。标签在 MIPS 中以冒号结束，例如 loop:。

系统调用：syscall 指令用于进行系统调用。在此之前，需要在 $v0 寄存器中设置适当的系统调用号，并在其他寄存器中设置所需的参数。例如，li $v0, 10; syscall 将执行系统调用号 10，即结束程序。

伪指令：伪指令不是真正的 MIPS 指令，而是由汇编器提供的，可以让编程更加方便。例如，li $t0, 10 是一个伪指令，它将把立即数 10 加载到 $t0 寄存器中，但在实际的 MIPS 指令集中，并没有这样的指令，汇编器会把它转换为一个或多个真正的 MIPS 指令。

数据声明：你可以在 MIPS 汇编程序中声明数据。例如，.data 指令标记了数据段的开始，.word 指令用于声明一个或多个 32 位的整数，.asciiz 用于声明一个以 null 结尾的字符串等。

在本次实验中，我们可以使用跳转指令实现条件语句与循环语句，数组与标识符来实现栈，用栈来实现数据的自底向上运算。

1. **主要流程与总体结构**

在本次实验中，中间代码生成的过程相比于之前的过程相对简单。

我采用了递归下降的方式来编写中间代码生成这个程序，为每一个非终结符都定义了一个处理函数。  
 void processPROG(TreeNode\* node);

void processPROG(TreeNode\* node);

void processDECLS(TreeNode\* node);

void processDECLS\_(TreeNode\* node);

void processDECL(TreeNode\* node);

vector<string> processNAMES(TreeNode\* node);

vector<string> processNAMES\_(TreeNode\* node);

void processSTMTS(TreeNode\* node);

void processSTMTS\_(TreeNode\* node);

void processSTMT(TreeNode\* node);

void processSTMT\_(TreeNode\* root, string varName);

void processIfStmt(TreeNode\* node);

void processELSESTMT(TreeNode\* node);

void processWhileStmt(TreeNode\* node);

void processReadStmt(TreeNode\* node);

void processWriteStmt(TreeNode\* node);

void processEXPR(TreeNode\* node);

void processADD\_(TreeNode\* node);

void processTERM(TreeNode\* node);

void processMUL\_(TreeNode\* node);

void processNEGA(TreeNode\* node);

void processFACTOR(TreeNode\* node);

void processBOOL(TreeNode\* node);

void processBOOL\_(TreeNode\* node);

void processJOIN(TreeNode\* node);

void processJOIN\_(TreeNode\* node);

void processNOT(TreeNode\* node);

void processREL(TreeNode\* node);

我定义了符号表，处理变量声明和类型检查（未写完）

vector<pair<pair<string,string>,int>>symboltable;

三个值分别存放名字，类型和初始值。如果声明时未初始化则设为-1

在递归下降结束之后，遍历符号表，将符号表中的所有变量对应的声明写在目标汇编语言的.data部分

然后由于代码生成时汇编代码部分很繁琐且重复。我为要重复用到的mips汇编代码写了多个生成函数

void int\_stack\_init() {

emit("#int栈初始化,t8为int栈指针");

emit("la $t8, int\_stack");

emit("addiu $t8, $t8, 508");

}

//t1入栈

void int\_stack\_push() {

emit("#t1入栈");

emit("addiu $t8, $t8, -4");

emit("sw $t1, 0($t8)");

}

//int出栈

void int\_stack\_pop\_1() {

emit("#出栈到t1");

emit("lw $t1, 0($t8)");

emit("addiu $t8, $t8, 4");

}

void int\_stack\_pop\_1\_2() {

emit("#出栈到t1与t2");

emit("lw $t1, 0($t8)");

emit("addiu $t8, $t8, 4");

emit("lw $t2, 0($t8)");

emit("addiu $t8, $t8, 4");

}

void bool\_stack\_init() {

emit("#bool栈初始化");

emit("la $t9, bool\_stack");

emit("addiu $t9, $t9, 127");

}

void bool\_stack\_push\_1() {

emit("#t1入栈");

emit("addiu $t9, $t9, -1");

emit("sb $t1, 0($t9)");

}

void bool\_stack\_pop\_1() {

emit("#出栈到t1");

emit("lb $t1, 0($t9)");

emit("addiu $t9, $t9, 1");

}

void bool\_stack\_pop\_1\_2() {

emit("#出栈到t1和t2");

emit("lb $t1, 0($t9)");

emit("addiu $t9, $t9, 1");

emit("lb $t2, 0($t9)");

emit("addiu $t9, $t9, 1");

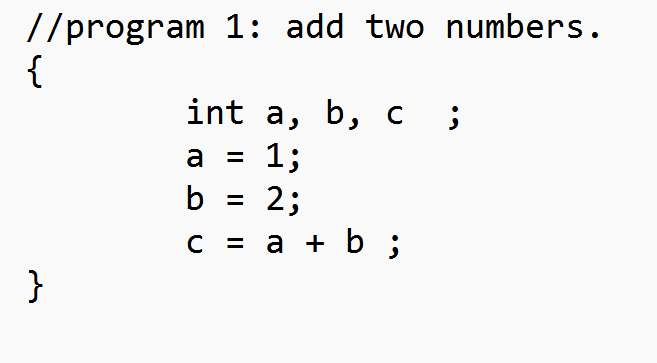
}

1. **程序测试**

在程序生成的代码会存放在final\_result.txt文件中。对汇编语言的检测，我才用的是mars4\_5这个软件。这个软件提供mips汇编的调试与运行环境

1. sourceProgram1测试

源代码：



生成的汇编：

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

a: .word 0

b: .word 0

c: .word 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, a

sw $t1, 0($t2)

li $t1, 2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, b

sw $t1, 0($t2)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

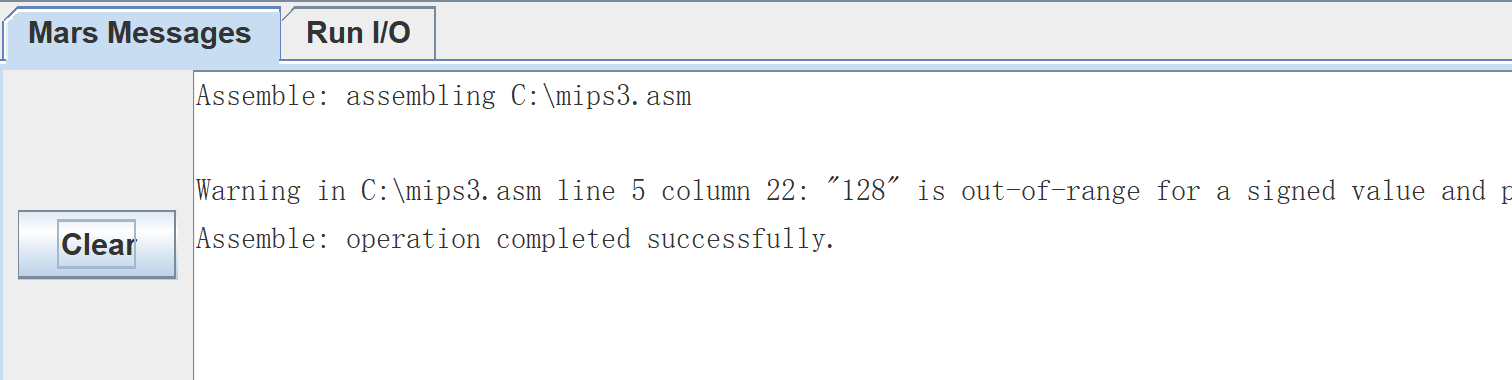
addiu $t8, $t8, 4

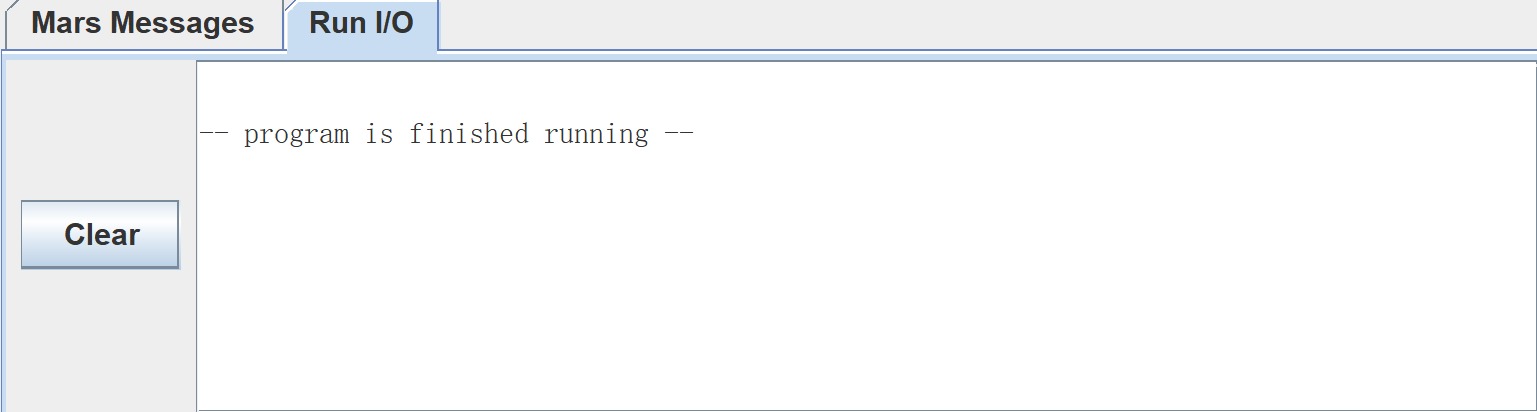
la $t2, c

sw $t1, 0($t2)

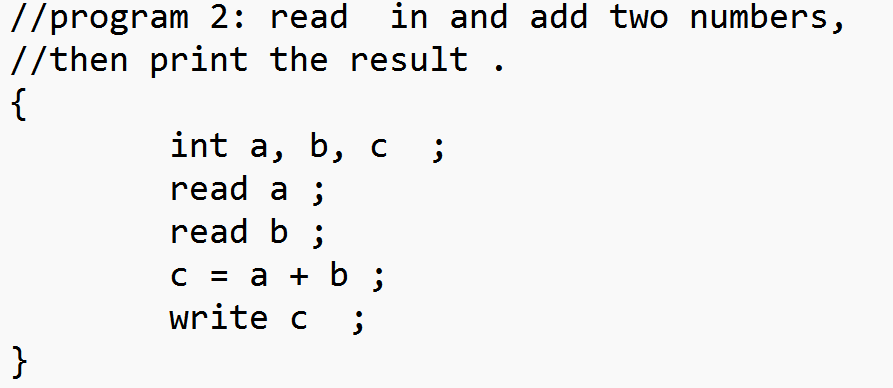
li $v0, 10

syscall

Mars4\_5运行结果:  




1. sourceProgram2

源代码：  


生成的汇编代码：

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

a: .word 0

b: .word 0

c: .word 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, a

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, b

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, c

sw $t1, 0($t2)

# 打印

lw $t0, c

li $v0, 1

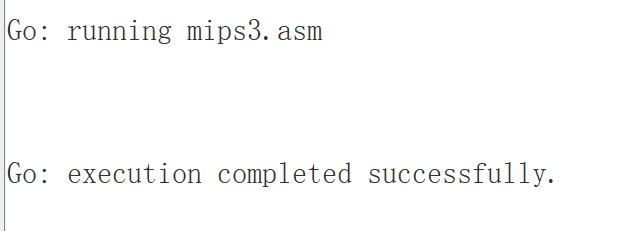
move $a0, $t0

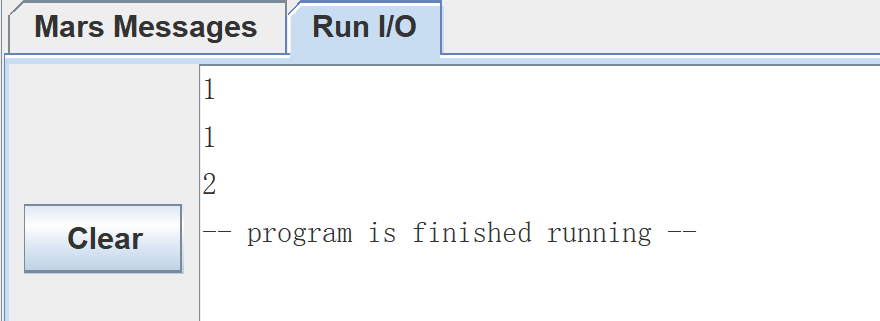
syscall

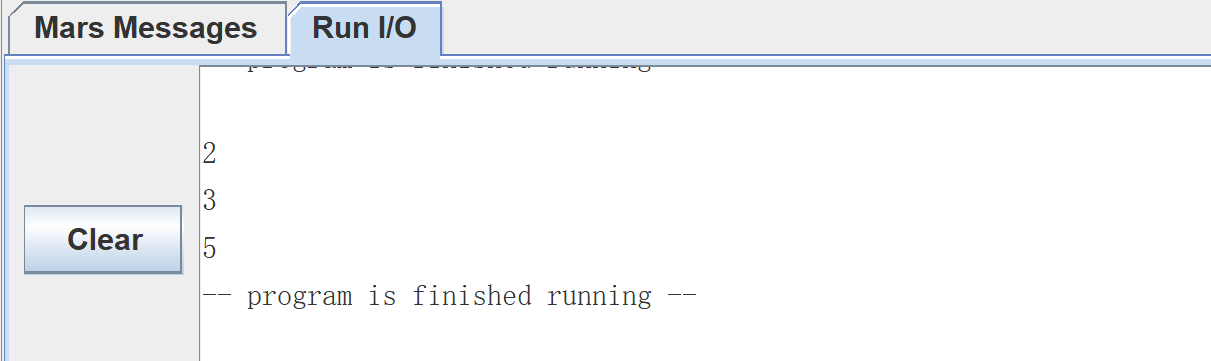
li $v0, 10

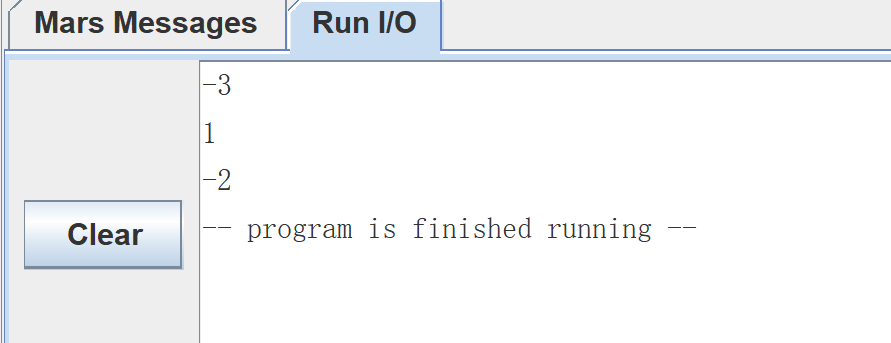
syscall

运行结果：



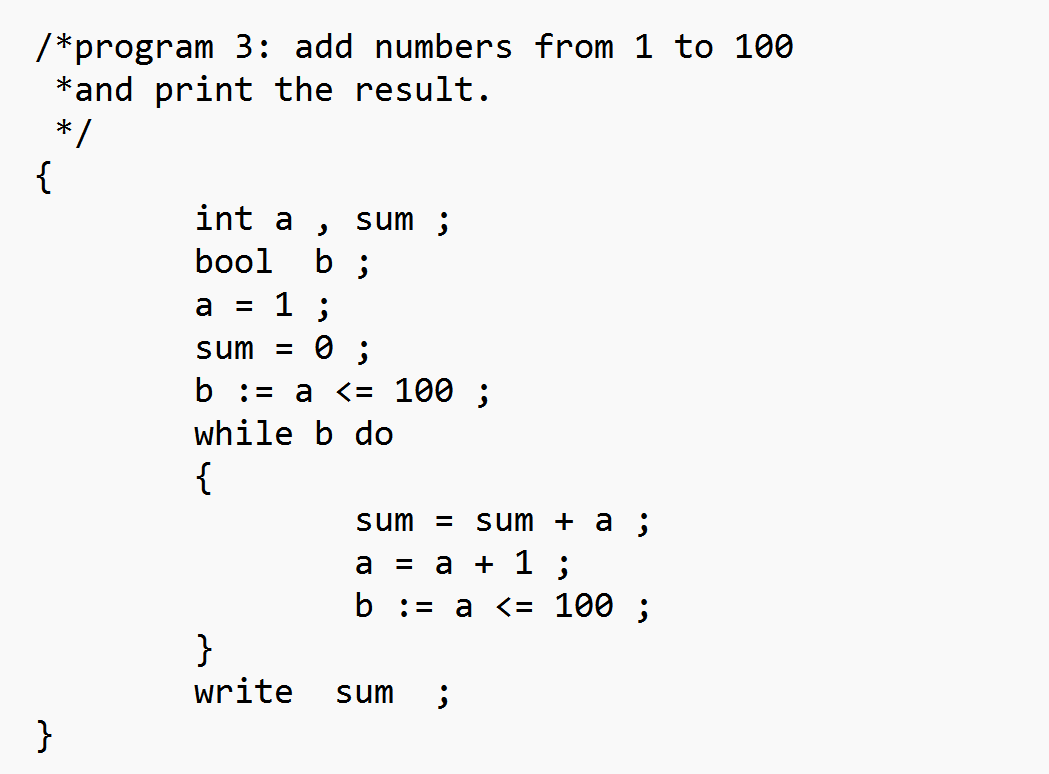






1. sourceProgram3

源代码：

  
生成的汇编代码：

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

a: .word 0

sum: .word 0

b: .byte 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, a

sw $t1, 0($t2)

li $t1, 0

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, sum

sw $t1, 0($t2)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 100

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sle $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, b

sb $t1, 0($t2)

loop0:

lb $t0, b

beqz $t0, endloop0

lw $t1, sum

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, sum

sw $t1, 0($t2)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, a

sw $t1, 0($t2)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 100

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sle $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, b

sb $t1, 0($t2)

j loop0

endloop0:

# 打印

lw $t0, sum

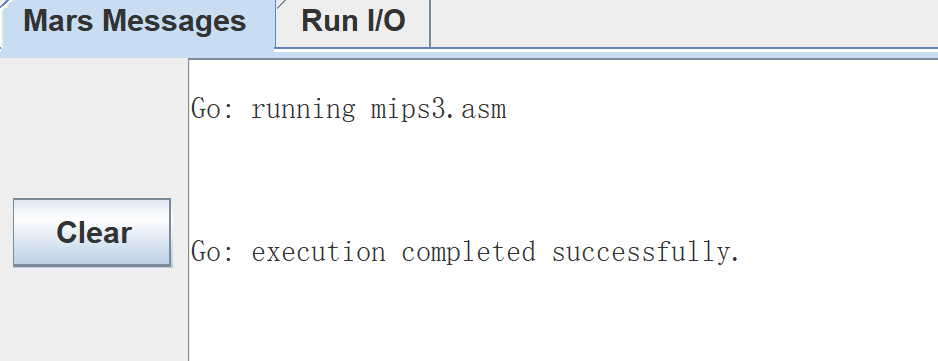
li $v0, 1

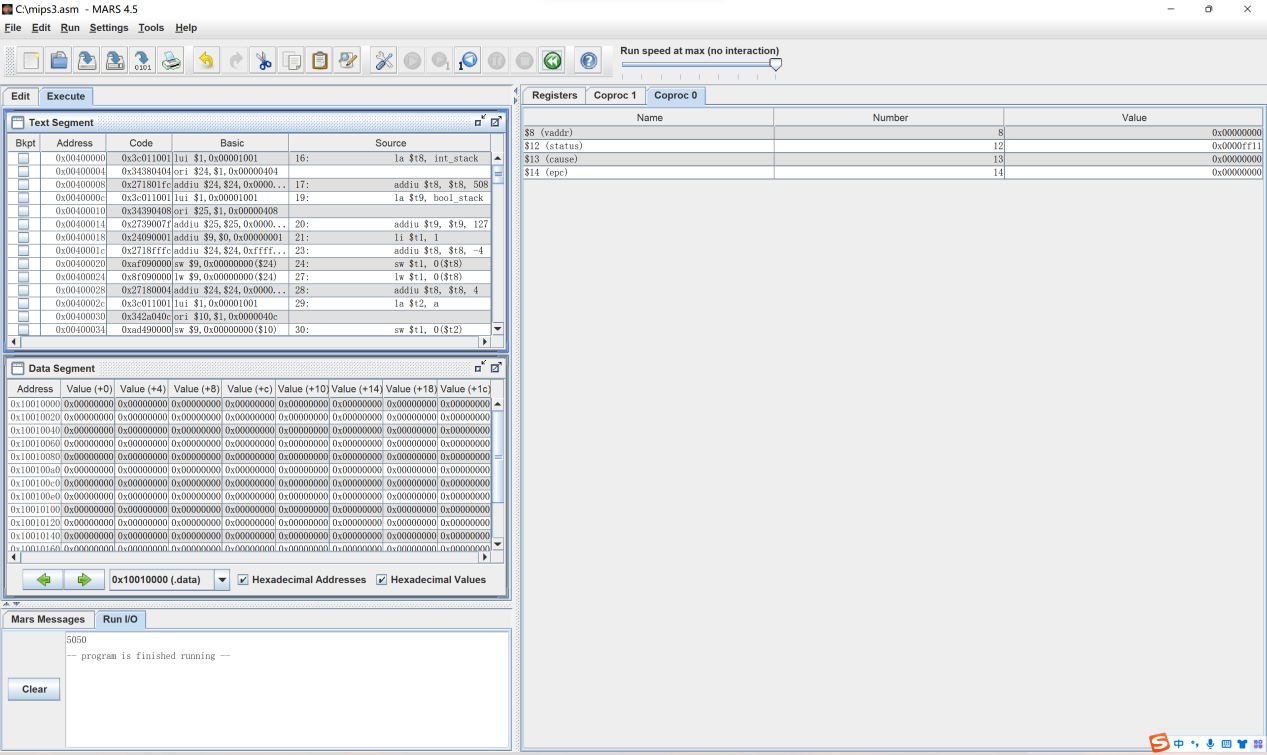
move $a0, $t0

syscall

li $v0, 10

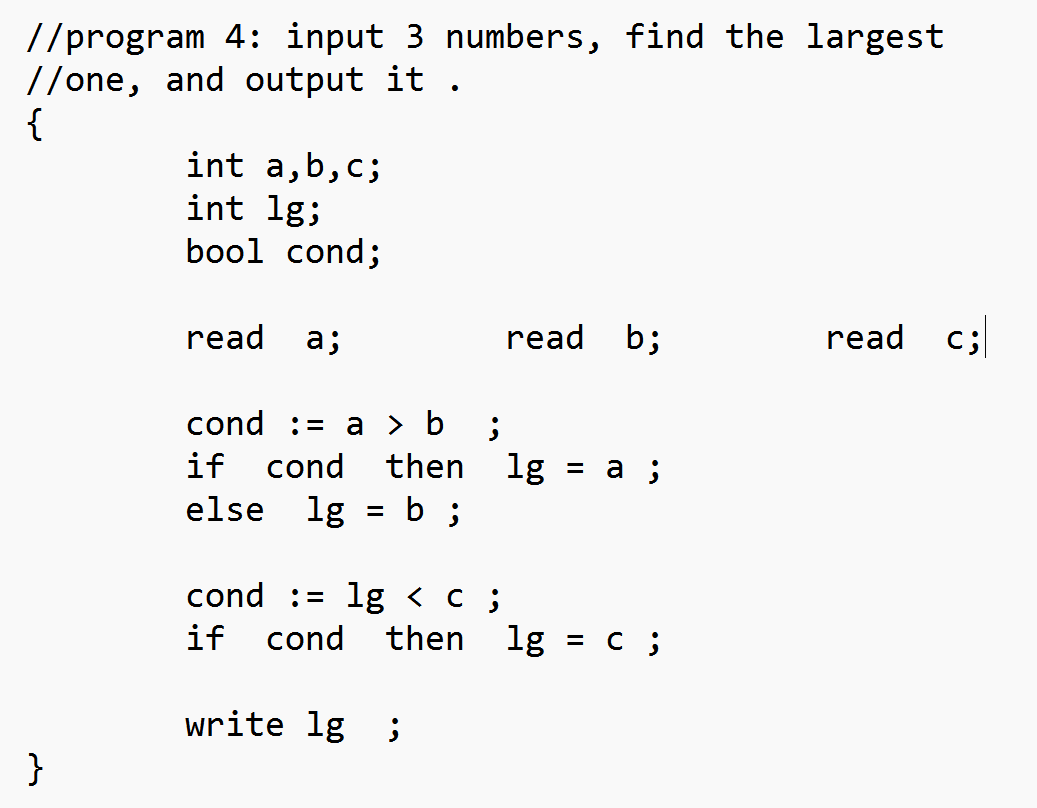
syscall

mars上运行结果：  




1. sourceProgram4

源代码如下：



生成的mips汇编如下:

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

a: .word 0

b: .word 0

c: .word 0

lg: .word 0

cond: .byte 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, a

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, b

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, c

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sgt $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond

sb $t1, 0($t2)

lb $t0, cond

beqz $t0, else\_0

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, lg

sw $t1, 0($t2)

j endif\_0

else\_0:

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, lg

sw $t1, 0($t2)

endif\_0:

lw $t1, lg

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, c

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

slt $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond

sb $t1, 0($t2)

lb $t0, cond

beqz $t0, else\_1

lw $t1, c

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, lg

sw $t1, 0($t2)

j endif\_1

else\_1:

endif\_1:

# 打印

lw $t0, lg

li $v0, 1

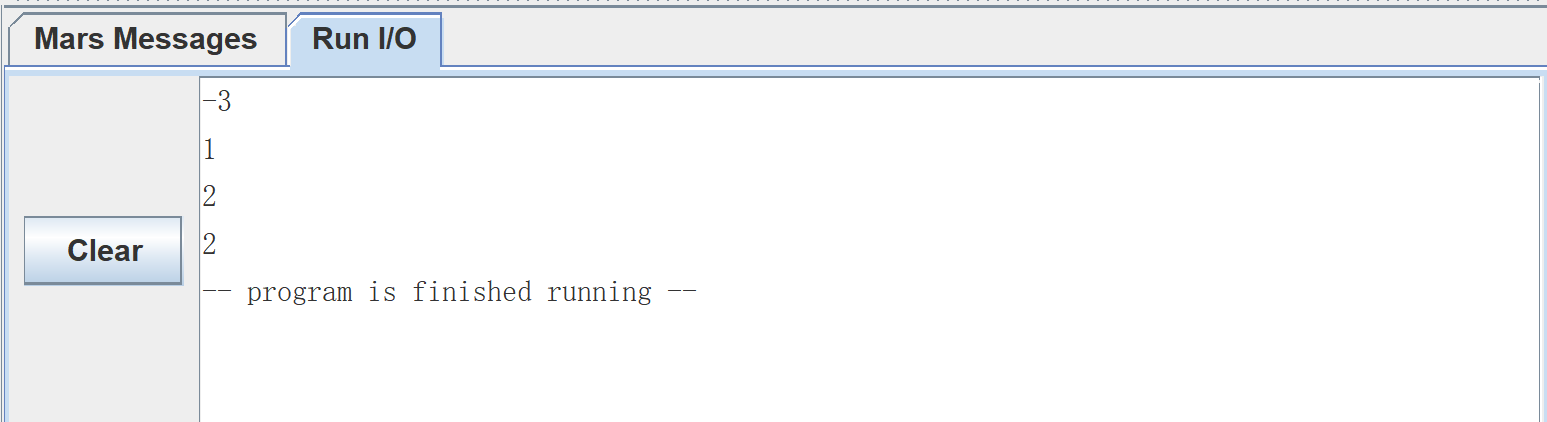
move $a0, $t0

syscall

li $v0, 10

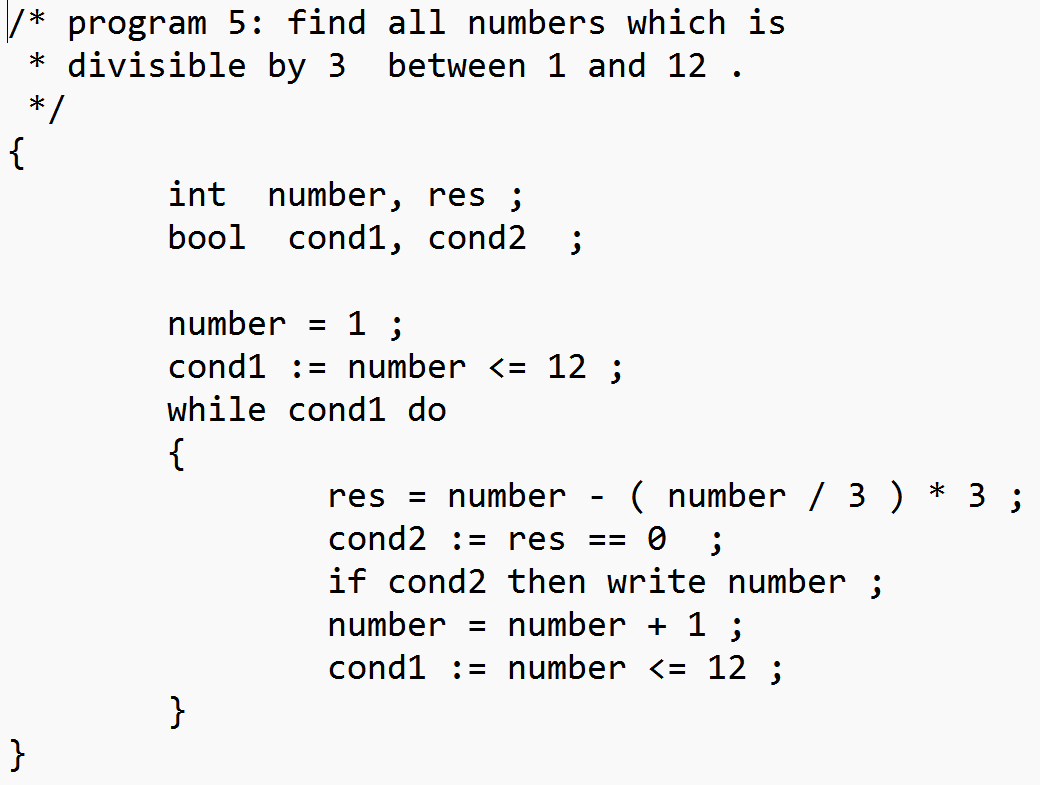
Syscall

运行结果如下：



1. sourceProgram5

源代码：

  
生成的汇编代码：

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

number: .word 0

res: .word 0

cond1: .byte 0

cond2: .byte 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, number

sw $t1, 0($t2)

lw $t1, number

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 12

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sle $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond1

sb $t1, 0($t2)

loop0:

lb $t0, cond1

beqz $t0, endloop0

lw $t1, number

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, number

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 3

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 除法

div $t1, $t2, $t1

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 3

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 乘法

mul $t1, $t1, $t2

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 减法

subu $t1, $t2, $t1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, res

sw $t1, 0($t2)

lw $t1, res

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 0

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

seq $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond2

sb $t1, 0($t2)

lb $t0, cond2

beqz $t0, else\_1

# 打印

lw $t0, number

li $v0, 1

move $a0, $t0

syscall

j endif\_1

else\_1:

endif\_1:

lw $t1, number

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, number

sw $t1, 0($t2)

lw $t1, number

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 12

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sle $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond1

sb $t1, 0($t2)

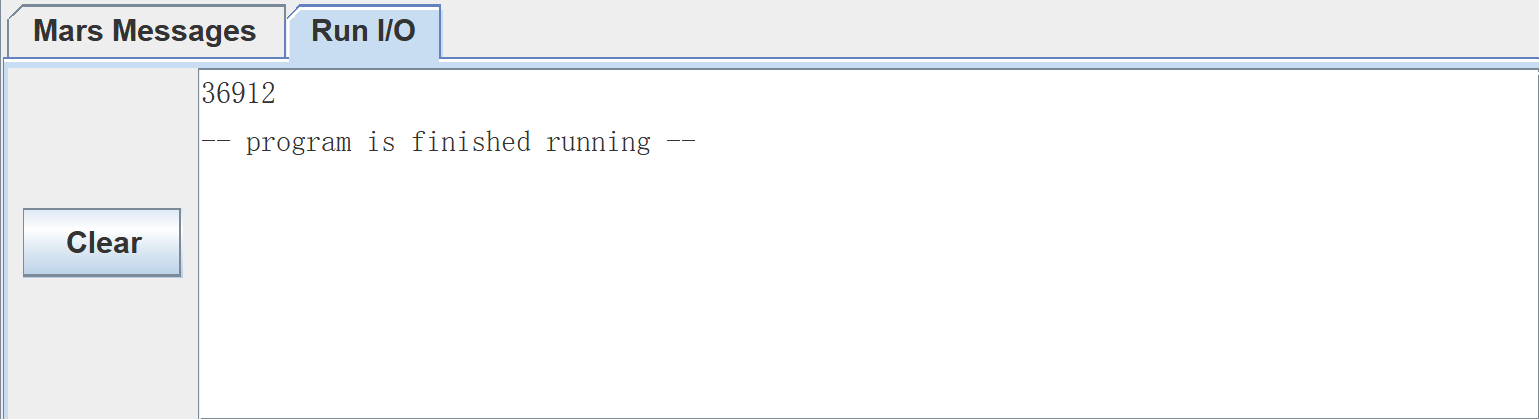
j loop0

endloop0:

li $v0, 10

syscall

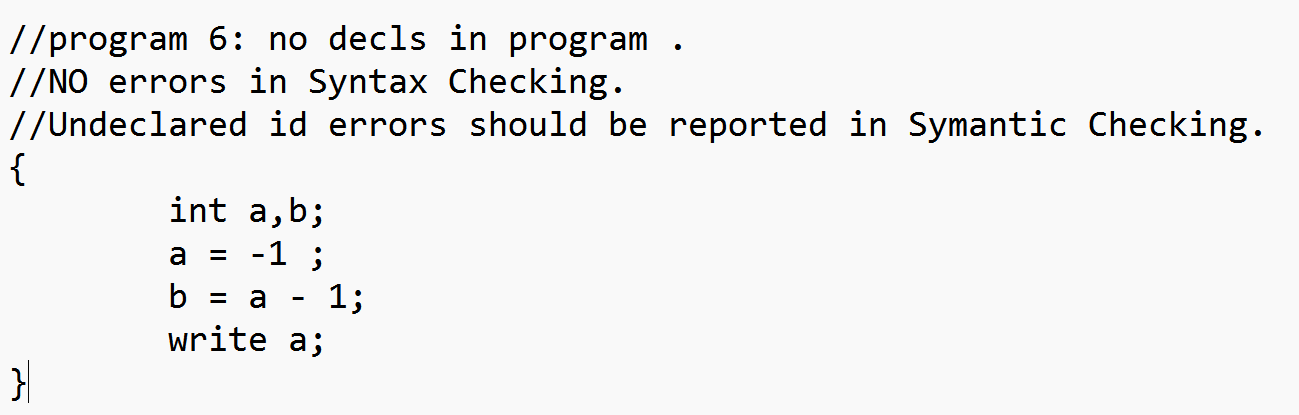
运行结果：



12以内3的倍数有3 6 9 12 源代码中没有间隔，所有生成的目标语言中也没有间隔

1. sourceProgram6

源语言：

  
Mips汇编：

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

a: .word 0

b: .word 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

# 取反

negu $t1, $t1

# 写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, a

sw $t1, 0($t2)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 减法

subu $t1, $t2, $t1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, b

sw $t1, 0($t2)

# 打印

lw $t0, a

li $v0, 1

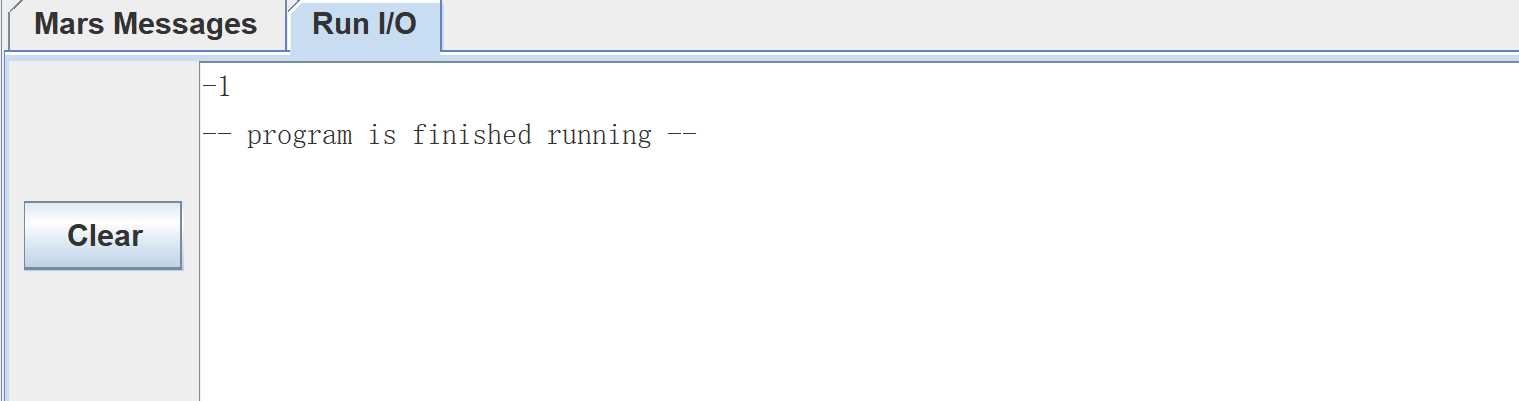
move $a0, $t0

syscall

li $v0, 10

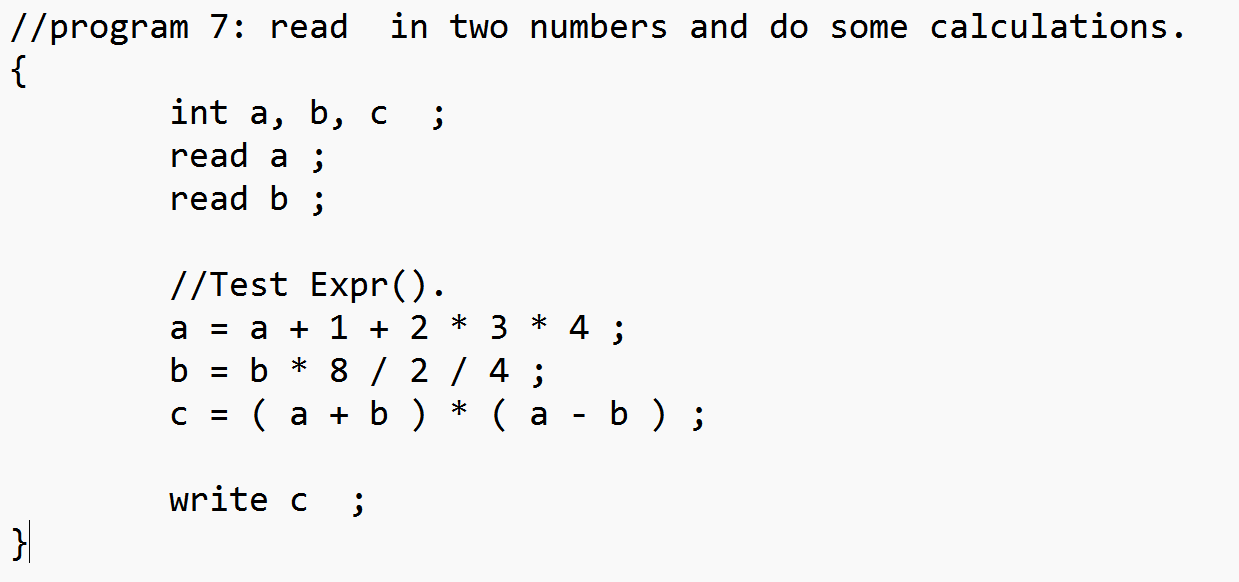
syscall

运行结果：



1. sourceProgram7

源语言：



mips汇编:

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

a: .word 0

b: .word 0

c: .word 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, a

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, b

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 3

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 乘法

mul $t1, $t1, $t2

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 4

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 乘法

mul $t1, $t1, $t2

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, a

sw $t1, 0($t2)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 8

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 乘法

mul $t1, $t1, $t2

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 除法

div $t1, $t2, $t1

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

li $t1, 4

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 除法

div $t1, $t2, $t1

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, b

sw $t1, 0($t2)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 加法

addu $t1, $t1, $t2

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 运算取两个栈顶

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 减法

subu $t1, $t2, $t1

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

# 乘法

mul $t1, $t1, $t2

# 将栈顶元素写回

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

# 将栈顶元素写回

#出栈到t1

lw $t1, 0($t8)

addiu $t8, $t8, 4

la $t2, c

sw $t1, 0($t2)

# 打印

lw $t0, c

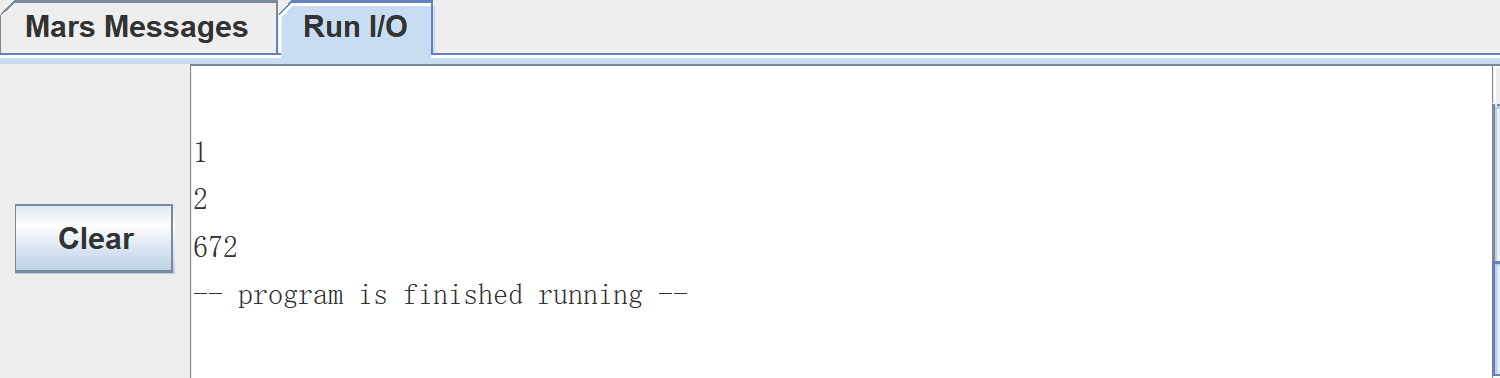
li $v0, 1

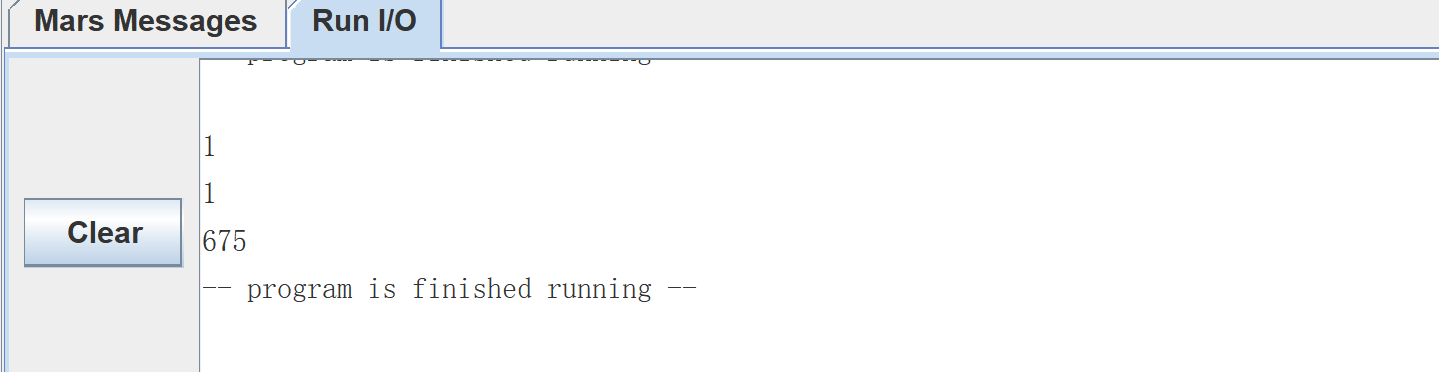
move $a0, $t0

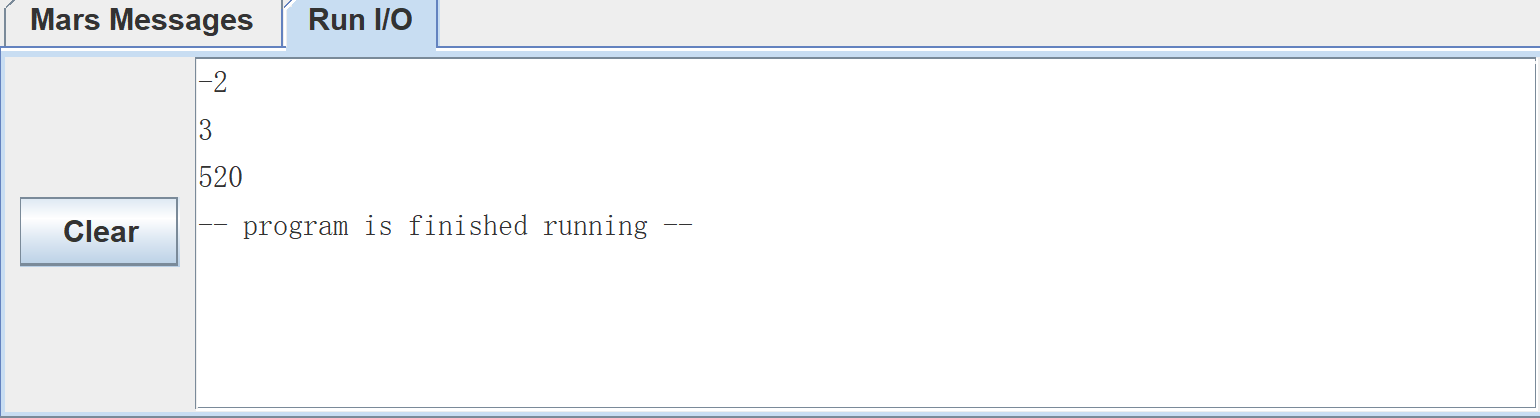
syscall

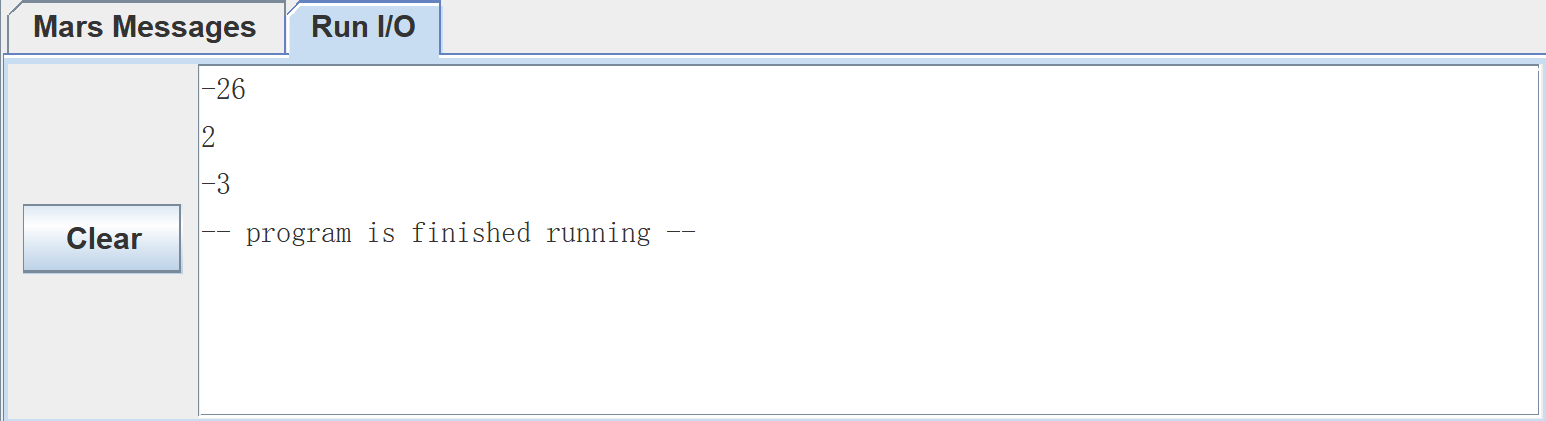
li $v0, 10

syscall

运行结果：  


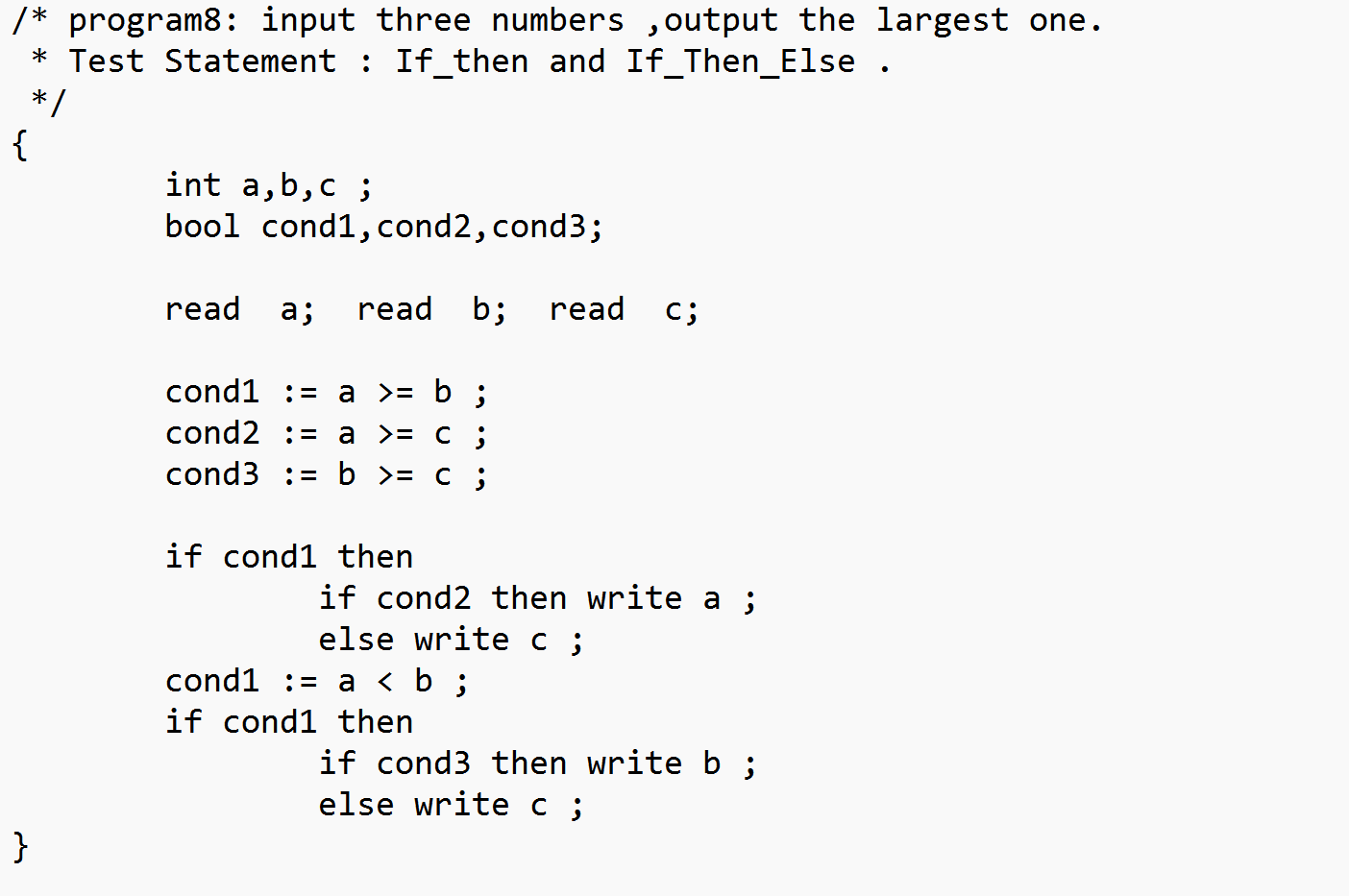






1. sourceProgram8

源代码：

  
mips汇编代码：

.data

input\_stack:.space 1024

input\_buffer: .word 0

int\_stack: .word 128

bool\_stack: .byte 128

a: .word 0

b: .word 0

c: .word 0

cond1: .byte 0

cond2: .byte 0

cond3: .byte 0

.text

.globl main

main:

#初始化栈

#int栈初始化,t8为int栈指针

la $t8, int\_stack

addiu $t8, $t8, 508

#bool栈初始化

la $t9, bool\_stack

addiu $t9, $t9, 127

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, a

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, b

# 读入

la $t0, input\_stack

addiu $t0, $t0, 1024

sw $t0, input\_buffer

li $v0, 5

syscall

sw $v0, c

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sge $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond1

sb $t1, 0($t2)

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, c

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sge $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond2

sb $t1, 0($t2)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, c

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

sge $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond3

sb $t1, 0($t2)

lb $t0, cond1

beqz $t0, else\_0

lb $t0, cond2

beqz $t0, else\_1

# 打印

lw $t0, a

li $v0, 1

move $a0, $t0

syscall

j endif\_1

else\_1:

# 打印

lw $t0, c

li $v0, 1

move $a0, $t0

syscall

endif\_1:

j endif\_0

else\_0:

endif\_0:

lw $t1, a

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

lw $t1, b

#t1入栈

addiu $t8, $t8, -4

sw $t1, 0($t8)

#出栈到t1与t2

lw $t1, 0($t8)

addiu $t8, $t8, 4

lw $t2, 0($t8)

addiu $t8, $t8, 4

slt $t1, $t2, $t1

#t1入栈

addiu $t9, $t9, -1

sb $t1, 0($t9)

# 将栈顶元素写回

#出栈到t1

lb $t1, 0($t9)

addiu $t9, $t9, 1

la $t2, cond1

sb $t1, 0($t2)

lb $t0, cond1

beqz $t0, else\_2

lb $t0, cond3

beqz $t0, else\_3

# 打印

lw $t0, b

li $v0, 1

move $a0, $t0

syscall

j endif\_3

else\_3:

# 打印

lw $t0, c

li $v0, 1

move $a0, $t0

syscall

endif\_3:

j endif\_2

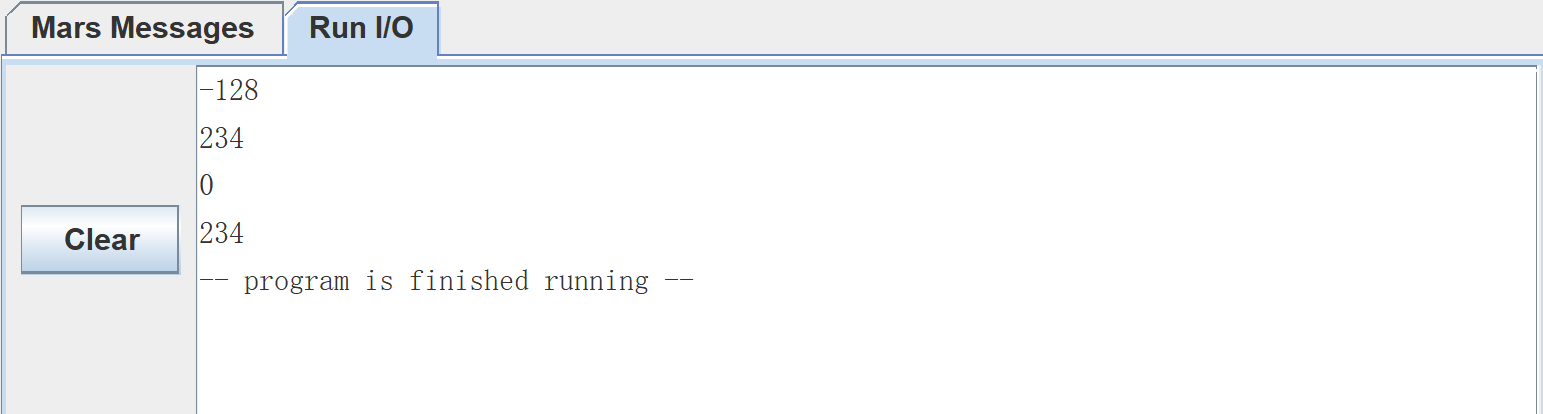
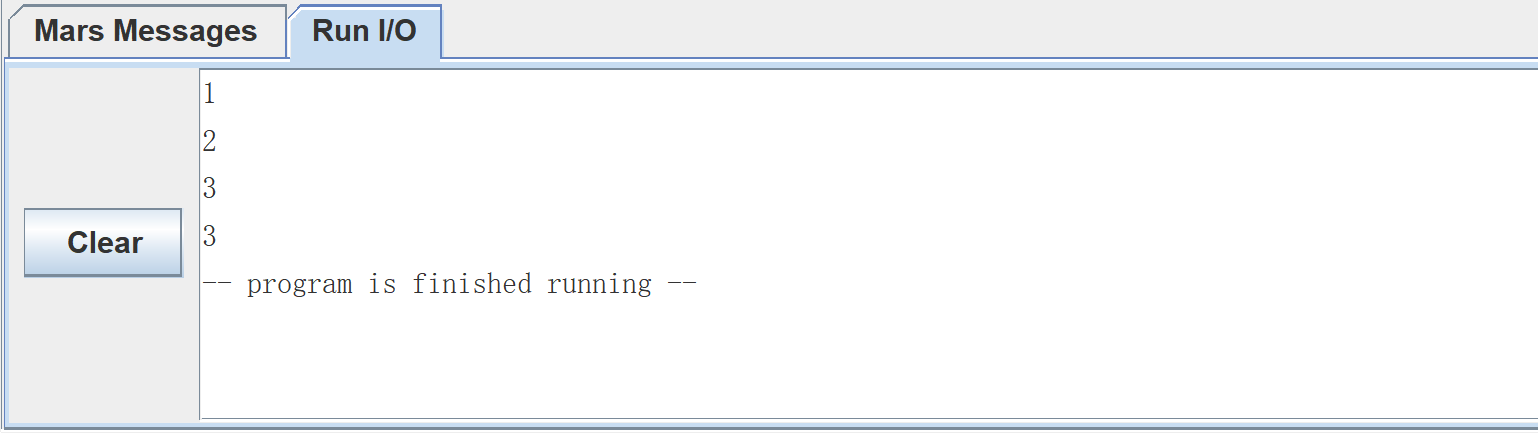
else\_2:

endif\_2:

li $v0, 10

syscall

运行结果：



1. **心得体会**

原本此次的目标语言是8086汇编语言。但是8086汇编语言的整型输入和bool输入有点繁琐，我就选择了mips汇编.

在写mips汇编的过程中最主要的还是栈的各种功能的mips实现。这一部分我采用了用一个特定的通用寄存器来作为栈顶指针。这样可以避免了各种写回，大大节省了工作量

在编写此次实验代码的过程中，我也发现了我以前在学习mips汇编和代码生成这一块的不足，在经过动手编写后，收获良多，对这部分知识也有了更充分的了解。