|  |
| --- |
| 编译原理期末作业 |
| 编译器构造实验 |

|  |
| --- |
| 19335015 陈恩婷  2022-7-7 |

目录

[1. 概述 2](#_Toc108299169)

[2. 编译器设计 2](#_Toc108299170)

[1. 文法设计 2](#_Toc108299171)

[2. 编译器整体架构 3](#_Toc108299172)

[3. 词法分析（扫描器） 3](#_Toc108299173)

[4. 语法分析 5](#_Toc108299174)

[5. 语义分析与中间代码生成 8](#_Toc108299175)

[6. 生成MIPS汇编代码 10](#_Toc108299176)

[7. 符号表系统 12](#_Toc108299177)

[8. 寄存器和变量的分配 14](#_Toc108299178)

[9. 汇编代码的执行 14](#_Toc108299179)

[3. 实验结果与分析 15](#_Toc108299180)

[1. 词法分析器 15](#_Toc108299181)

[2. 语法分析器 16](#_Toc108299182)

[3. 语义分析器 17](#_Toc108299183)

[4. 代码生成 17](#_Toc108299184)

[5. 目标代码的运行成果 19](#_Toc108299185)

[4. 实验总结 19](#_Toc108299186)

[5. 参考链接 20](#_Toc108299187)

# 概述

本实验通过用C++实现一个简单的编译器，包括编译器的词法分析、语法分析、语义分析、目标代码生成等部分，复习与巩固关于编译器的各个子系统的知识，进一步加深对编译器构造的理解。

# 编译器设计

## 文法设计

本实验的程序设计语言文法和C语言的基础文法基本保持一致，包括变量说明语句、算术运算表达式、赋值语句，以及逻辑运算表达式、If语句、While语句等，具体文法如下：

program -> int main ( ) { declarations stmt-sequence }

declarations -> declaration ; declarations | \epsilon

declaration -> int identifiers | \epsilon

identifiers -> ID identifiers'

identifiers' -> , identifiers | \epsilon

stmt-sequence -> statement stmt-sequence | \epsilon

statement -> if-stmt | assign-stmt | while-stmt

if-stmt -> if ( or-exp ) { stmt-sequence } else-stmt

else-stmt -> else { stmt-sequence } | \epsilon

while-stmt -> while ( or-exp ) { stmt-sequence }

assign-stmt -> ID = or-exp ;

or-exp -> and-exp or'-exp

or'-exp -> || or-exp | \epsilon

and-exp -> comparison-exp and'-exp

and'-exp -> && and-exp | \epsilon

comparison-exp -> add-sub-exp comparison'-exp

comparison'-exp -> < comparison-exp | > comparison-exp | <= comparison-exp | >= comparison-exp | == comparison-exp | != comparison-exp | \epsilon

add-sub-exp -> mul-div-exp add-sub'-exp

add-sub'-exp -> + add-sub-exp | - add-sub-exp | \epsilon

mul-div-exp -> factor mul-div'-exp

mul-div'-exp -> \* mul-div-exp | / mul-div-exp | \epsilon

factor -> CONSTANT | ID | ( or-exp ) | ! factor

## 编译器整体架构

编译器的实质是一个程序，其核心功能是将源代码翻译成目标代码。本实验实现的编译器主要分为以下几个模块：

其中前三个模块属于编译器的前端部分，最后一个模块属于编译器的后端部分。各模块的设计会在接下来的小节详细介绍。

## 词法分析（扫描器）

词法分析器是编译的第一阶段，功能是输入源程序，输出单词符号和一些相关信息。单词符号是一个程序语言的基本语法符号。程序语言的单词符号一般可以分为关键字、标识符、常数、运算符、界符等。单词种别通常用整数编码。

本实验支持的单词符号包括：

1. 关键字

本实验支持的关键字包括：

const set<string> Keywords

    = {"else", "if", "int", "main", "while"};

1. 标识符

支持的标识符为字母开头的含有数字和字母的字符串, 但不能以字母’t’开头。

1. 常数

支持的常数包括非负整数。

1. 运算符和界符

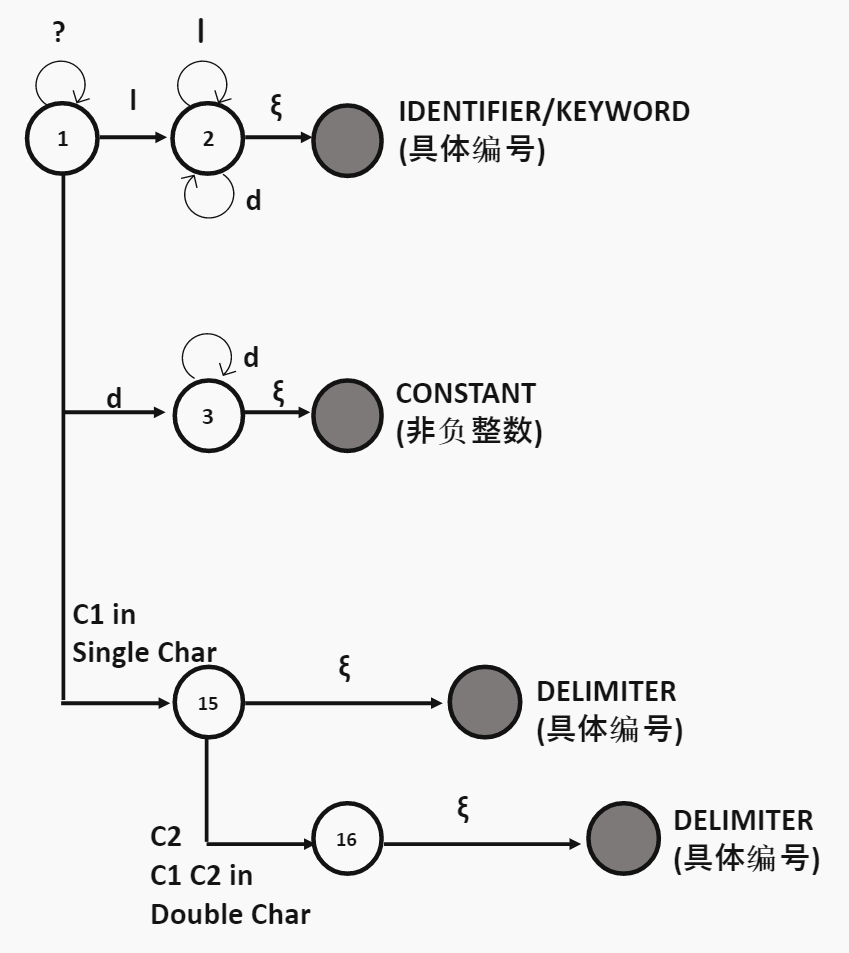
本实验支持的界符包括：

const set<string> Delimiters

    = {"(", ")", ";", ",", "=", "+", "-", "\*", "/", "{", "}",

    "||", "&&", "!", "==", "!=", "<", ">", "<=", ">="};

本实验的词法分析自动机如下：



本实验的词法分析主要输出TOKEN序列、关键字表、界符表、符号表、常数表等。举例如下：

int main(){

int a, b, c;

a = 2;

if ( a > 0 ){

c = 2+1;

}

while ( a > 0 ){

a = a-1;

}

}

生成的各表格信息如下：

关键字K表和界符P表

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 关键字 | 编号 | 界符 |
| 1 | int | 1 | ( |
| 2 | main | 2 | ) |
| 3 | If | 3 | { |
| 4 | while | 4 | , |
|  |  | 5 | ; |
|  |  | 6 | = |
|  |  | 7 | > |
|  |  | 8 | + |
|  |  | 9 | } |
|  |  | 10 | - |

符号表I表

|  |  |  |  |
| --- | --- | --- | --- |
| NAME | TYPE | CAT | ADDR |
| a |  |  |  |
| b |  |  |  |
| c |  |  |  |

这里符号表的后三列是空的，待语义分析时再填入。

常数表C表

|  |
| --- |
| 2 |
| 0 |
| 1 |

TOKEN序列如下：

(l, 1), (l, 2), (p, 1), (p, 2), (p, 3), (l, 1), (i, 1), (p, 4), (i, 2), (p, 4), (i, 3), (p, 5), (i, 1), (p, 6), (c, 1), (p, 5), (l, 3), (p, 1), (i, 1), (p, 7), (c, 2), (p, 2), (p, 3), (i, 3), (p, 6), (c, 1), (p, 8), (c, 3), (p, 5), (p, 9), (l, 4), (p, 1), (i, 1), (p, 7), (c, 2), (p, 2), (p, 3), (i, 1), (p, 6), (i, 1), (p, 10), (c, 3), (p, 5), (p, 9), (p, 9)

以上生成的表格存储在in.txt.la1中，TOKEN序列存储在in.txt.la2中。

## 语法分析

语法分析是编译的第二阶段；其任务是识别和处理比单词更大的语法单位，如：程序设计语言中的表达式、各种说明和语句乃至全部源程序，指出其中的语法错误；必要时，可生成内部形式，便于下一阶段处理。

在本实验中，词法分析器的功能是输入一个段源程序代码，输出是否符合文法的判断结果和错误原因。

**LL(1) 分析法**

本实验采用LL(1)分析法进行语法分析，其主要步骤如下：

1. 读入一个满足LL(1)文法要求的程序设计语言文法

2. 对于每个非终结符，计算其first集

3. 对于每个非终结符，计算其follow集

4. 对于每个产生式，计算其select集

5. 利用select集，生成LL(1)分析表

6. 利用分析表，对输入的源代码进行解析

相应的分析函数如下：

int Parser::parse( vector<pair<TOKEN\_TYPE, string>> & token\_sequence ){

    get\_grammar( "grammar1.txt" );

    get\_first\_set();

    get\_follow\_set();

    get\_select\_set();

    get\_table();

    return ll1\_parsing( token\_sequence );

}

接下来逐步描述以上过程。

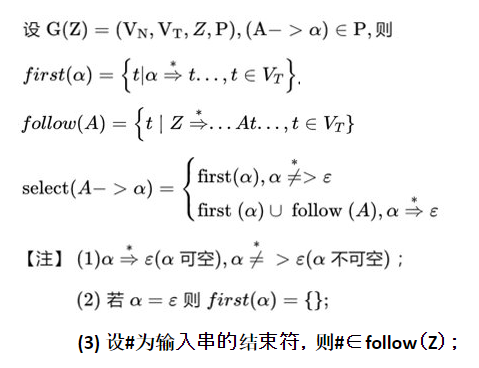
1. **读入文法**

该步骤主要是将文法的每条产生式从文件读入并存储在相应的数据结构中，本实验主要采用的数据结构为：

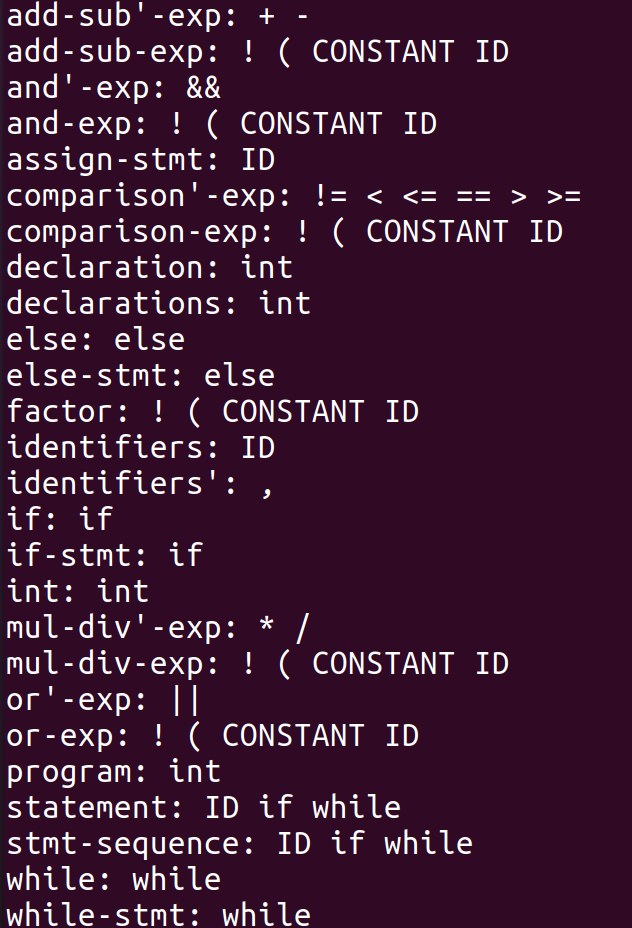
map<string, vector<vector<string>> > productions;

1. **计算first集、follow集和select集**

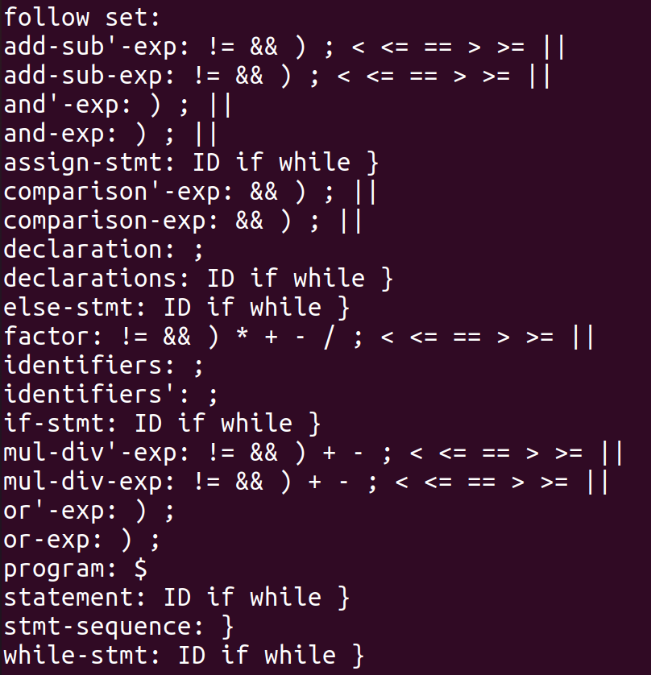
First集、follow集和select集的计算方法如下：



其中first集的计算结果如下：

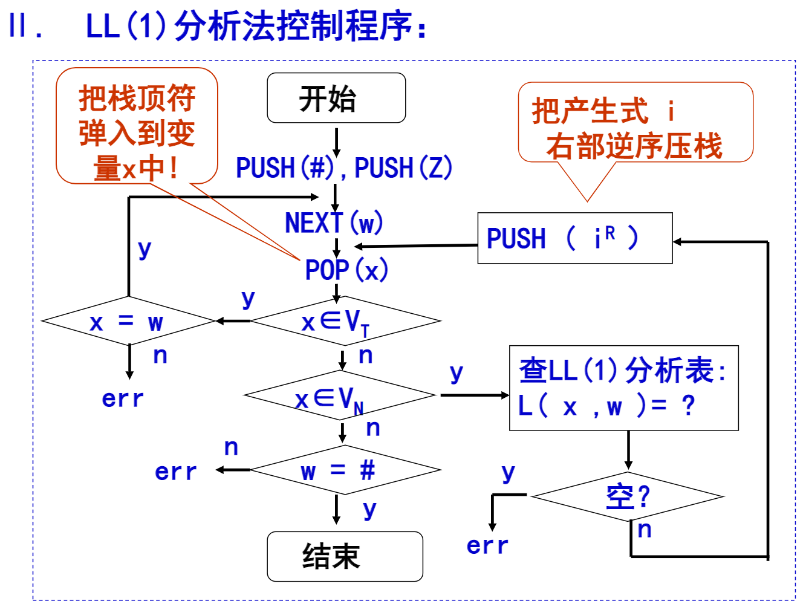


follow集的计算结果如下：



1. **利用分析表，对输入的源代码进行解析**

生成LL(1)分析表之后，就可以对输入的源代码进行解析了，具体流程如下：



## 语义分析与中间代码生成

语法分析是编译的第三阶段，其任务是对结构上正确的源程序进行上下文有关性质的审查，进行类型审查，并生成中间代码。

在本实验中，语义分析器的功能是输入一段源程序，输出四元式格式的中间语言，以及完整的符号表。

其中翻译文法如下：

program -> int main ( ) { declarations stmt-sequence }

declarations -> declaration ; declarations | \epsilon

declaration -> int identifiers | \epsilon

identifiers -> "SYMBOL-TABLE" ID identifiers'

identifiers' -> , identifiers | \epsilon

stmt-sequence -> statement stmt-sequence | \epsilon

statement -> if-stmt | assign-stmt | while-stmt

if-stmt -> if ( or-exp ) "IF" { stmt-sequence } else-stmt "IF-END"

else-stmt -> else "ELSE" { stmt-sequence } | \epsilon

while-stmt -> while "WHILE" ( or-exp ) "DO" { stmt-sequence } "WHILE-END"

assign-stmt -> "PUSH" ID = or-exp "ASSIGN" ;

or-exp -> and-exp or'-exp

or'-exp -> || or-exp "GEQ(||)" | \epsilon

and-exp -> comparison-exp and'-exp

and'-exp -> && and-exp "GEQ(&&)" | \epsilon

comparison-exp -> add-sub-exp comparison'-exp

comparison'-exp -> < comparison-exp "GEQ(<)" | > comparison-exp "GEQ(>)" | <= comparison-exp "GEQ(<=)" | >= comparison-exp "GEQ(>=)" | == comparison-exp "GEQ(==)" | != comparison-exp "GEQ(!=)" | \epsilon

add-sub-exp -> mul-div-exp add-sub'-exp

add-sub'-exp -> + add-sub-exp "GEQ(+)" | - add-sub-exp "GEQ(-)" | \epsilon

mul-div-exp -> factor mul-div'-exp

mul-div'-exp -> \* mul-div-exp "GEQ(\*)" | / mul-div-exp "GEQ(/)" | \epsilon

factor -> "PUSH" CONSTANT | "PUSH" ID | ( or-exp ) | ! factor

如上所示，产生式中被引号括起来的是相应的文法操作。每个操作的解释如下：

1. “SYMBOL-TABLE”

该操作为填充符号表中的信息，包括TYPE、CAT以及ADDR等。

1. “IF”、”ELSE” 和 “IF-END”

这三个操作主要处理if语句的相关四元式生成。对于IF操作，翻译器将需要判断的表达式结果t添加到if四元式中，即输出if t \_ \_；对于ELSE和IF-END操作，翻译器需要输出else \_ \_ \_和if-end \_ \_ \_ 。

1. “WHILE”、“DO” 和 “WHILE-END”

这三个操作主要处理while语句的相关四元式生成，具体操作和if语句类似，但不同的是需要判断的表达式结果t会附在do四元式中，即do t \_ \_，而while与while-end四元式不需要附带表达式结果t。

1. “GEQ(…)” 和 “PUSH”

这两个操作主要处理四则运算和逻辑运算相关的四元式的生成，PUSH表示把当前的终结符压入存放表达式符号的栈sem中，这里的终结符包括常数、变量以及临时变量等，而GEQ表示把当前栈顶的两个终结符弹出，生成四元式，并将最终的计算结果的临时变量放入栈顶。

1. “ASSIGN”

这个操作主要处理赋值语句的相关四元式生成，具体操作就是把当前sem栈顶的临时变量/变量/常数的值赋给在栈顶元素之下的变量。

## 生成MIPS汇编代码

编译的最后一个阶段是后端，也就是汇编代码的生成。本实验中我选用的汇编指令集为MIPS。**MIPS（Microprocessor without Interlocked Pipeline Stages）**是一种采取精简指令集（RISC）的指令集架构（ISA）。本实验采用的是32位的MIPS指令集，具体的介绍可以参考《计算机组成与设计（硬件/软件接口MIPS版 亚洲版）》，实验中主要用到的指令如下表所示：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **助记符** | **指 令 格 式** | | | | | | **示 例** | **示例含义** | **操作及解释** |
| BIT # | 31..26 | 25..21 | 20..16 | 15..11 | 10..6 | 5..0 |  |  |  |
| R-类型 | op | rs | rt | rd | shamt | func |  |  |  |
| add | 000000 | rs | rt | rd | 00000 | 100000 | add $1,$2,$3 | $1=$2+S3 | (rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1 |
| sub | 000000 | rs | rt | rd | 00000 | 100010 | sub $1,$2,$3 | $1=$2-S3 | (rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1 |
| and | 000000 | rs | rt | rd | 00000 | 100100 | and $1,$2,$3 | $1=$2&S3 | (rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1 |
| or | 000000 | rs | rt | rd | 00000 | 100101 | or $1,$2,$3 | $1=$2|S3 | (rd)←(rs) | (rt); rs=$2,rt=$3,rd=$1 |
| xor | 000000 | rs | rt | rd | 00000 | 100110 | xor $1,$2,$3 | $1=$2^S3 | (rd)←(rs)^(rt); rs=$2,rt=$3,rd=$1 |
| nor | 000000 | rs | rt | rd | 00000 | 100111 | nor $1,$2,$3 | $1= ~($2 | S3) | (rd)←~((rs) | (rt)); rs=$2,rt=$3,rd=$1 |
| slt | 000000 | rs | rt | rd | 00000 | 101010 | slt $1,$2,$3 | if($2<$3)  $1=1 else  $1=0 | if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1 |
| sll | 000000 | 00000 | rt | rd | shamt | 000000 | sll $1,$2,10 | $1=$2<<10 | (rd)←(rt)<<shamt,rt=$2,rd=$1,shamt=10 |
| jr | 000000 | rs | 00000 | 00000 | 00000 | 001000 | jr $31 | goto $31 | (PC)←(rs) |
| I-类型 | op | rs | rt | immediate | | |  | | |
| lw | 100011 | rs | rt | offset | | | lw $1,10($2) | $1=Memory[  $2+10] | (rt)←Memory[(rs)+(sign\_extend)offset],  rt=$1,rs=$2 |
| sw | 101011 | rs | rt | offset | | | sw $1,10($2) | Memory[  $2+10] =$1 | Memory[(rs)+(sign\_extend)offset]←(rt),  rt=$1,rs=$2 |
| beq | 000100 | rs | rt | offset | | | beq $1,$2,40 | if($1=$2)  goto PC+4+40 | if ((rt)=(rs)) then (PC)←(PC)+4+( (Sign-Extend) offset<<2), rs=$1, rt=$2 |
| bne | 000101 | rs | rt | offset | | | bne $1,$2,40 | if($1≠$2)  goto PC+4+40 | if ((rt)≠(rs)) then (PC)←(PC)+4+(  (Sign-Extend) offset<<2) , rs=$1, rt=$2 |
| slti | 001010 | rs | rt | immediate | | | slti $1,$2,10 | if($2<10)    $1=1 else    $1=0 | if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1 |
| J-类型 | op | address | | | | |  | | |
| j | 000010 | address | | | | | j 10000 | goto 10000 | (PC)←( (Zero-Extend) address<<2),  address=10000/4 |

指令详细结构

生成MIPS的过程主要是由符号表生成.data部分以及由四元式区生成.text部分。具体需要处理的符号表项以及四元式主要有如下几种：

1. 符号表项

本实验的符号表系统主要支持的是INT类型的操作与计算，而临时变量直接采用相同名称的寄存器进行存储，所以生成代码时只需要输出变量名以及.word即可。

    outfile << ".data" << endl;

    for ( auto data : symbol\_table ){

        outfile << data.name << ": .word " << endl;

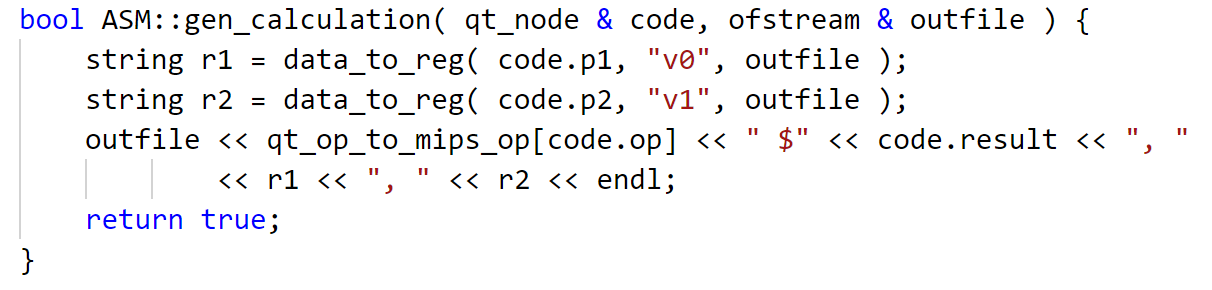
    }

1. 四则运算与逻辑运算类四元式（包括赋值）

对于运算类四元式，需要判断四元式中参与计算的两个操作数是变量、临时变量还是常数：

1. 临时变量：存储在相应的寄存器中，直接在生成MIPS汇编代码时使用寄存器名作为操作数即可。
2. 变量：变量存储在内存中，需要使用lw指令先取出放到寄存器中。
3. 常数：本实验采用了先用li取出数据放到寄存器，再生成寄存器指令的方法。

最后根据当前四元式所做的操作，生成汇编指令即可。

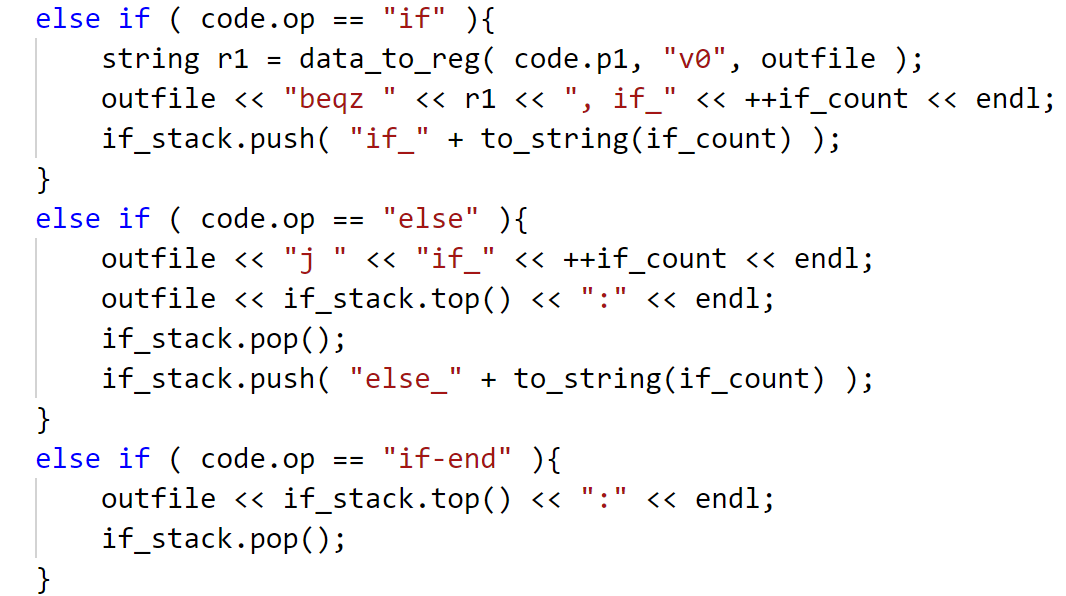


1. If语句

if语句的汇编代码生成主要分为几步：

1. 首先要计算出需要判断的表达式的值是否为零，这个在处理if四元式前就以及完成了；
2. 对于if t \_ \_ 四元式，生成beqz汇编指令，如果t=0，则跳转到else或者if-end并给所跳转到的label进行命名，并存储该label；
3. 遇到else \_ \_ \_四元式时，需要在满足if条件时所执行的代码末尾添加一个j指令，跳转到if-end处，并给else处的汇编代码用之前在处理if四元式时被存储的label进行标记；
4. 遇到if-end四元式时，需要给if-end处的汇编代码用之前在处理if或者else四元式时被存储的label进行标记；。

代码如下：



1. While语句

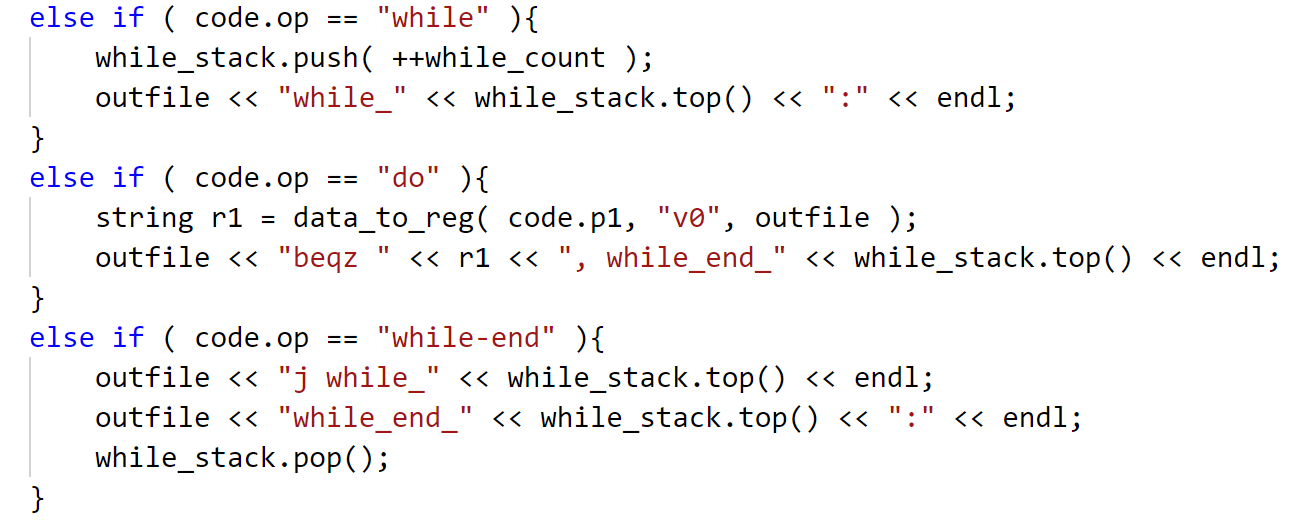
While语句的汇编代码生成主要分为几步：

(1) 对于while \_ \_ \_ 四元式，需在此处标记一个新label，并存储该label的名字；

(2) 对于do t \_ \_ 四元式，生成beqz汇编指令，如果t=0，则跳转到while-end处并给所跳转到的label进行命名，并存储该label；

1. 对于 while-end \_ \_ \_ 四元式，需要在满足if条件时所执行的代码末尾添加一个j指令，跳转到while处，并给while-end处的汇编代码用之前在处理do四元式时被存储的label进行标记；

具体代码如下：



## 符号表系统

由于本实验的编译器只支持INT型变量的计算，所以符号表系统的设计相对简单，主要结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| NAME | TYPE | CAT | ADDR |
| a | INT | V | 0 |
| b | INT | V | 4 |
| c | INT | V | 8 |

如表所示，每个变量在表中有四部分信息：name, type, cat, addr。其中name部分存储变量的名称，type存储变量的数据类型，cat存储变量的类别，addr为变量的相对地址。表格的具体数据结构如下：

enum SYMBOL\_TYPE{

    NULL\_TYPE, // uninitialized

    INT

};

enum SYMBOL\_CAT{

    NULL\_CAT, // uninitialized

    VAR

};

map<TOKEN\_TYPE, string> token\_type\_string = {

    {KEYWORD, "KEYWORD"},

    {DELIMITER, "DELIMITER"},

    {ID, "ID"},

    {CONSTANT, "CONSTANT"},

    {EOF\_, "EOF\_"}

};

map<SYMBOL\_TYPE, string> symbol\_type\_string = {

    {NULL\_TYPE, "NULL\_TYPE"},

    {INT, "INT"}

};

map<SYMBOL\_CAT, string> symbol\_cat\_string = {

    {NULL\_CAT, "NULL\_CAT"},

    {VAR, "VAR"}

};

struct symbol{

    string name;

    SYMBOL\_TYPE type;

    SYMBOL\_CAT cat;

    int addr;

};

## 寄存器和变量的分配

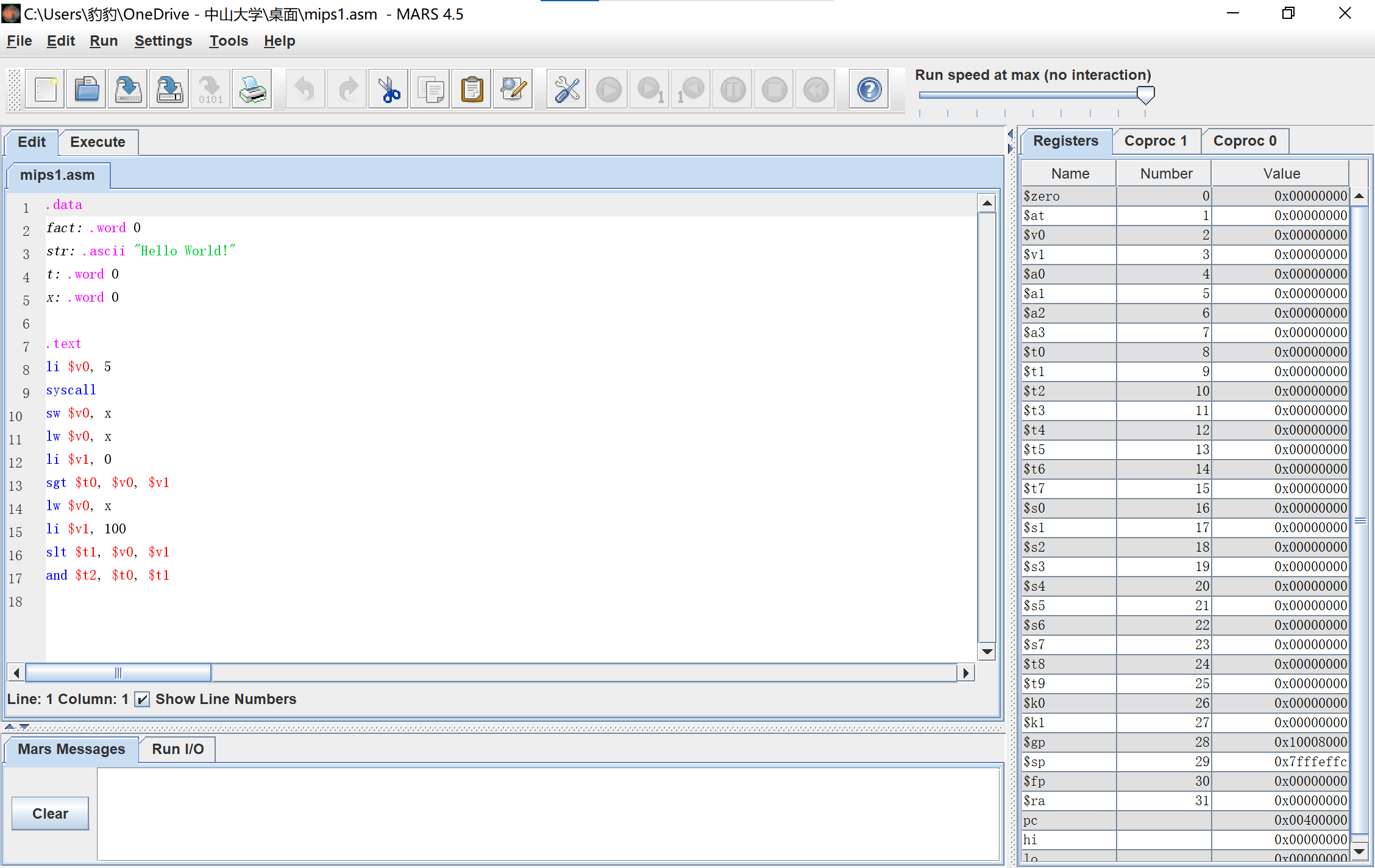
本实验采用的是简易的寄存器和变量分配的方法，对于源程序中出现的变量名，本实验在将汇编代码中以.word形式列于.data部分，也就是将变量存于数据段；而对于临时变量，本实验直接按照他们的名称(t0~t9)将他们存于寄存器中，这样也就意味着程序只能最多使用10个临时变量的限制。

## 汇编代码的执行

前面的几部分已经基本完成了一个编译器的设计，最后一步就是要将编译好的MIPS汇编代码运行起来，这里我选用的是MARS mips simulator模拟器来执行MIPS，它的主页网址如下：

<http://courses.missouristate.edu/kenvollmar/mars/>

它的界面如下图所示：



可以在Run菜单中运行当前的MIPS代码，下方的console可以进行输入输出等操作，右侧的寄存器一栏用于跟踪寄存器所存的值的实时状态。

至此，一个由C++实现的C语言简易编译器就设计好了。

# 实验结果与分析

本章节以下程序为例，分析编译器功能的正确性：

int main(){

int a, b, c;

a = 2;

if ( a < 0 ){

b = 1\*0;

c = a && b;

c = a || b;

}

else{

c = 2/2;

}

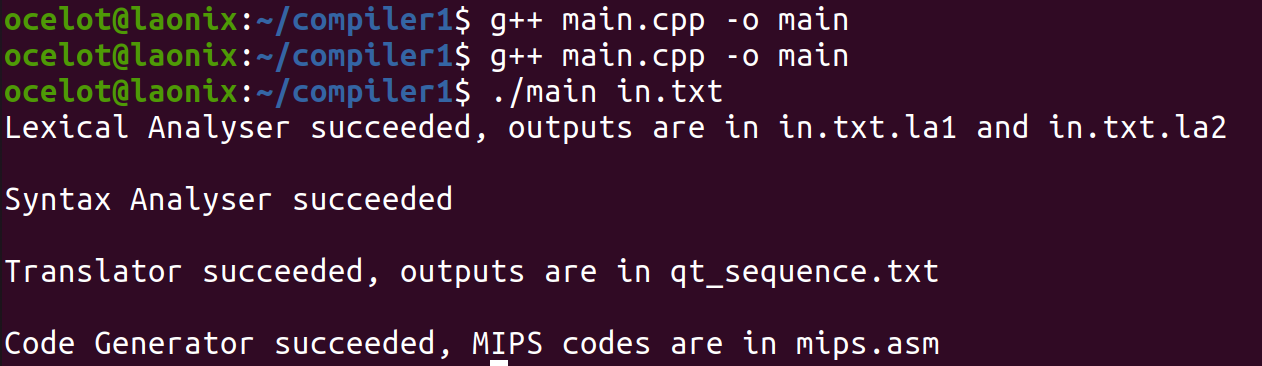
while ( a > 0 ){

a = a-1;

}

}

对于以上程序，编译器成功将它编译成了MIPS汇编代码



## 词法分析器

词法分析器的结果主要是符号表（不完整）、关键字表、界符表、常数表等。对于以上源程序，输出结果如下：

Keyword Table:

1: int

2: main

3: if

4: else

5: while

Delimiter Table:

1: (

2: )

3: {

4: ,

5: ;

6: =

7: <

8: \*

9: &&

10: ||

11: }

12: /

13: >

14: -

Symbol Table:

1: a

2: b

3: c

Constant Table:

1: 2

2: 0

3: 1

输出的TOKEN序列如下：

(l, 1), (l, 2), (p, 1), (p, 2), (p, 3), (l, 1), (i, 1), (p, 4), (i, 2), (p, 4), (i, 3), (p, 5), (i, 1), (p, 6), (c, 1), (p, 5), (l, 3), (p, 1), (i, 1), (p, 7), (c, 2), (p, 2), (p, 3), (i, 2), (p, 6), (c, 3), (p, 8), (c, 2), (p, 5), (i, 3), (p, 6), (i, 1), (p, 9), (i, 2), (p, 5), (i, 3), (p, 6), (i, 1), (p, 10), (i, 2), (p, 5), (p, 11), (l, 4), (p, 3), (i, 3), (p, 6), (c, 1), (p, 12), (c, 1), (p, 5), (p, 11), (l, 5), (p, 1), (i, 1), (p, 13), (c, 2), (p, 2), (p, 3), (i, 1), (p, 6), (i, 1), (p, 14), (c, 3), (p, 5), (p, 11), (p, 11)

如上所示，可见生成的TOKEN序列是正确的。

## 语法分析器

根据前面截图中的结果，可见程序的语法分析没有产生错误。

## 语义分析器

语义分析器主要负责符号表的完整构建以及中间代码生成，对于例子中的程序，完整的符号表如下：

a INT VAR 0

b INT VAR 4

c INT VAR 8

生成的中间代码如下：

= 2 \_ a

< a 0 t0

if t0 \_ \_

\* 1 0 t1

= t1 \_ b

& a b t2

= t2 \_ c

| a b t3

= t3 \_ c

else \_ \_ \_

/ 2 2 t4

= t4 \_ c

if-end \_ \_ \_

while \_ \_ \_

> a 0 t5

do t5 \_ \_

- a 1 t6

= t6 \_ a

while-end \_ \_ \_

如上所示，可见正确地完成了中间代码生成的任务。

## 代码生成

代码生成部分主要是由符号表的信息和四元式代码，生成MIPS汇编程序。对于实例中的程序，生成的代码如下：

.data

a: .word

b: .word

c: .word

.text

li $v0, 2

sw $v0, a

lw $v0, a

li $v1, 0

slt $t0, $v0, $v1

beqz $t0, if\_1

li $v0, 1

li $v1, 0

mul $t1, $v0, $v1

sw $t1, b

sw $t2, c

sw $t3, c

j else\_2

if\_1:

li $v0, 2

li $v1, 2

div $t4, $v0, $v1

sw $t4, c

else\_2:

while\_1:

lw $v0, a

li $v1, 0

sgt $t5, $v0, $v1

beqz $t5, while\_end\_1

lw $v0, a

li $v1, 1

sub $t6, $v0, $v1

sw $t6, a

j while\_1

while\_end\_1:

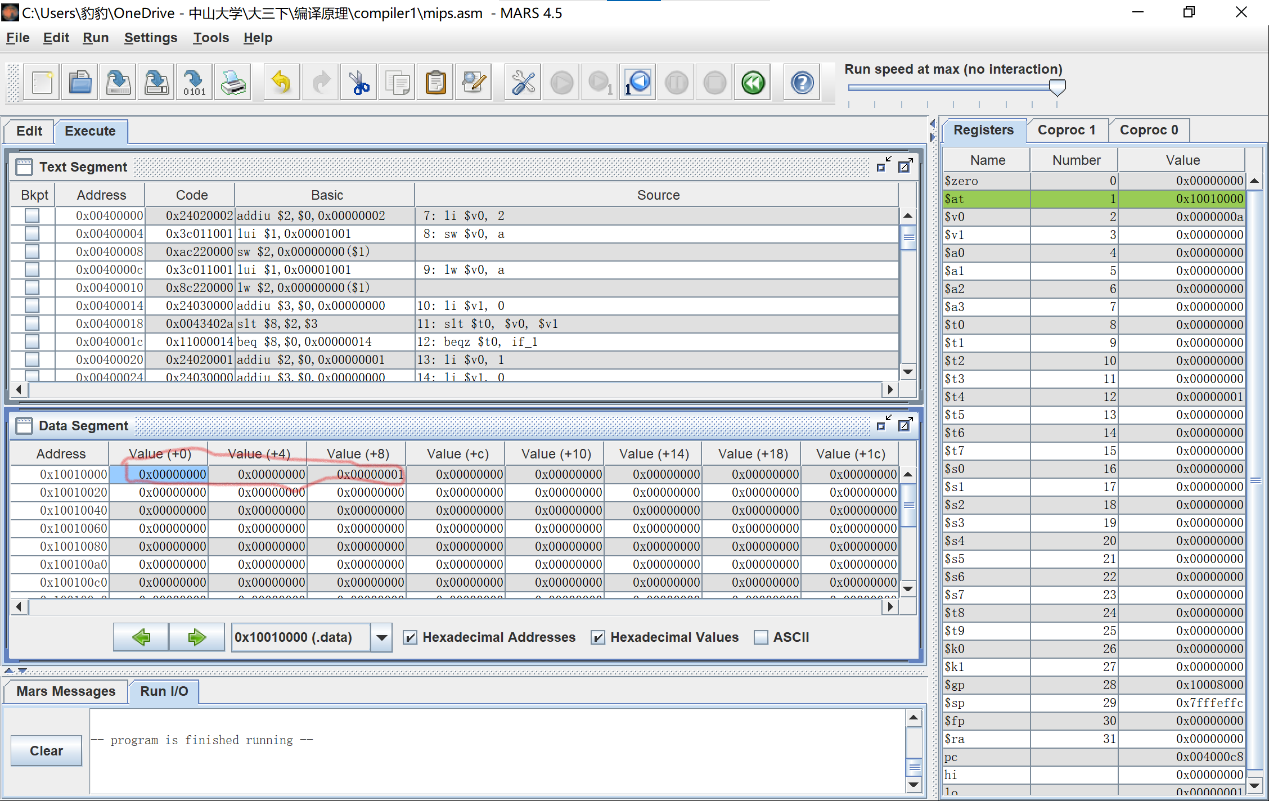
li $v0, 10

syscall

如上所示，可见正确地完成了汇编代码生成的任务。

## 目标代码的运行成果

如图所示，使用MARS模拟器模拟执行MIPS代码，执行结果如下：



如图所示，根据源程序的语义可以推测，a, b, c三个变量的最终值分别为0, 0, 1，这与MARS模拟器的执行结果相符。

至此，源程序的编译就成功了。

# 实验总结

本次实验是编译原理的期末项目，在完成实验的过程中我系统地复习了自动机、词法分析、语法分析、语义分析、中间代码生成以及目标代码生成等知识，并进行了实际练习，在整合前面几次实验，实现完整编译器的同时回顾了编译器的整体设计思想与方法，温故而知新，体会到了前人的智慧与努力。同时我也锻炼了排查和解决问题，并且分析实验结果的能力，学会了如何与他人沟通，向他人请教等等。

在编写代码和运行的过程中，我还体会到了细节的重要性。尽管通常我们都是因为一些小的地方出现的问题而导致实验难以推进，但究其原因很多也是在大的方向上没有设计好，例如解决问题的整个思路和每一步具体需要完成什么目标等等。一旦大体的结构设计清晰，中途进行大改的可能性就会降低，细节处的问题很多也就迎刃而解了。

非常感谢老师和助教不厌其烦地回答我在实验中遇到的问题，希望自己在以后能够再接再厉，在编译原理和其他领域学到更多知识，解决更多问题。

# 参考链接

1. [Compilers: Principles, Techniques, and Tools](http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text%20Books/Compiler%20Design/Alfred%20V.%20Aho,%20Monica%20S.%20Lam,%20Ravi%20Sethi,%20Jeffrey%20D.%20Ullman-Compilers%20-%20Principles,%20Techniques,%20and%20Tools-Pearson_Addison%20Wesley%20(2006).pdf)
2. [编译器构造实验](https://docs.qq.com/slide/DZGtlS05TZ0N3UFB4)
3. [本实验的GitHub链接](https://github.com/Lao-lao/Compiler/tree/main/compiler)