

# 虚拟机实验

## ▼ 虚拟机实验

### ▼ 引入

- 目标功能
- 实现背景

### ▼ 中间代码

- 中间代码生成思路
- ▼ 中间代码实现方法
  - 中间代码的表示

### ▼ 虚拟机

- 执行代码实现思路
- ▼ 执行代码实现方法
  - 分配寄存器
  - 执行中间代码
  - 执行单条中间代码
- 测试用例及运行结果
- 总结

## 引入

## 目标功能

本次虚拟机的目标功能如下所示：

1. 全局变量实现
2. 数组的定义
3. 变量的连续声明及初始化
4. 常量给变量赋值以及变量给变量赋值
5. 无参数和返回值的函数调用
6. 无参数、有返回值的函数调用
7. 带参数、无返回值的函数调用
8. 带参数和返回值的函数调用
9. 允许不同函数中出现同一变量
10. 输出词法分析分析结果、语法分析的结果

11. 能够生成中间代码，可处理的类型包括：赋值语句、算术运算操作（加减乘除）、跳转语句、分支与while循环语句、函数定义（参数声明、参数声明、函数体）、函数调用（函数传参、函数返回）
12. 可处理的类型包括：read函数、write函数、main函数（程序入口）、赋值语句、算术运算操作（加减乘除）、跳转语句、分支与while循环语句、函数返回

## 实现背景

本次虚拟机实验我们使用了flex/bison为主体，通过调用C文件的方式实现了中间代码的转换，并且读入相关中间代码让虚拟机可以正常运行。该实验是基于已经实现词法分析、语法分析、语义分析的程序的背景下进行的。

## 中间代码

### 中间代码生成思路

在进行语义分析的时候，由于bison的特性，使得其可以不用输出中间代码也能完成相关的要求。但是等到进行虚拟机相关操作时，生成中间代码有利于虚拟机更加简单的实现。同时，在bison中调用函数较为麻烦并且容易出错，而编写C文件调用其中的函数更加便捷准确。因此，在多方考虑之后我们小组还是决定将相关语句转化为中间代码之后输出，并利用中间代码实现虚拟机。

### 中间代码实现方法

整个C文件包括了较多之前用来实现语法分析语法树和语义分析判断的代码，因此我在本篇文章中就不再重复叙述之前出现过的代码了。

### 中间代码的表示

中间代码的相关表示与存储使用的是双链表的方式，这样的好处是到时候需要调用时只需传回链表的表头即可，避免出现当传回数据较多时需要编写繁重的语句。在C语言调用的头文件中，数据结构定义如下：单条中间代码可以将代码类型和操作数分别进行存储。

```

typedef struct _OperandStru // 操作数
{
    enum
    {
        VARIABLE, // 变量 x
        TEMPVAR,   // 临时变量 t1
        LABEL,     // 标签 lable1
        CONSTANT,  // 常数 #1
        ADDRESS,   // 取地址 &x
        VALUE,     // 读取地址的值 *x
        FUNC,      // 函数
    } kind;
    union {
        int tempvar; // 临时变量
        int lable;   // 标签
        int value;   // 常数的值
        char *name;  // 语义值, 变量名称、函数名称
    } operand;
    int value;
} OperandStru, *Operand;

```

又因为操作数在储存时，需要区分相关类型；对于不同类型的中间代码，其操作数的数量也是不同的，因此使用union进行存储。同时为了实现对函数声明、函数调用语句的翻译，定义了函数参数列表的数据结构，在递归调用的翻译过程中存储所有的参数。

```

typedef struct _InterCodeStru // 中间代码
{
    // 代码类型
    enum
    {
        _LABEL,    // 定义标号
        _FUNCTION, // 定义函数
        _ASSIGN,    // =
        _ADD,       // +
        _SUB,       // -
        _MUL,       // *
        _DIV,       // /
        _GOTO,      // 无条件跳转
        _IFGOTO,    // 判断跳转
        _RETURN,    // 函数返回
        _ARG,       // 传实参
        _CALL,      // 函数调用
        _PARAM,     // 函数参数声明
        _READ,      // 从控制台读取x
        _WRITE,     // 向控制台打印x
        _NULL       // 空的
    } kind;
    // 操作数
    union {
        struct
        { // 赋值 取地址 函数调用等
            Operand left, right;
        } assign;
        struct
        { // 双目运算 + = * /
            Operand result, op1, op2;
        } binop;
        struct
        { // GOTO 和 IF...GOTO
            Operand lable, op1, op2;
            char *relop;
        } jump;
        // 函数声明、参数声明、标签、传实参、函数返回、读取x、打印x
        Operand var;
    } operands;
    struct _InterCodeStru *prev, *next;
} InterCodeStru, *InterCode;

// 函数参数列表
typedef struct _ArgListStru
{
    int num;
    Operand list[10];
} ArgListStru, *ArgList;

```

```
InterCode CodesHead, CodesTail; // 全局变量，线性IR双链表的首尾
```

整体中间代码是通过语法树生成的，程序通过遍历语法树得到转化过后的非终结符与终结符，再将其转化为对应的中代码。本次遍历语法树使用的是自顶向下遍历。同时为了减少对临时变量的重复声明，在翻译基本表达式、条件表达式以及一些语句时，进行变量表的遍历，对于存储相同值的临时变量直接使用即可，不再重复生成：

```

// 当Exp的翻译模式为INT、ID、MINUS Exp时, 可以获取已经申明过的操作数
Operand get_Operand(tnode Exp)
{
    // INT
    if (Exp->ncld == 1 && !strcmp((Exp->cld)[0]->name, "INT"))
    {
        return find_Const((int)((Exp->cld)[0]->value));
    }
    // ID
    else if (Exp->ncld == 1 && !strcmp((Exp->cld)[0]->name, "ID"))
    {
        Operand variable = new_Variable((Exp->cld)[0]->content);
        return variable;
    }
    // MINUS Exp (Exp:INT)
    else if (Exp->ncld == 2 && !strcmp((Exp->cld)[0]->name, "MINUS"))
    {
        if (!strcmp(((Exp->cld)[1]->cld)[0]->name, "INT"))
        {
            int value = -(int)(((Exp->cld)[1]->cld)[0]->value);
            Operand result = find_Const(value);
            if (result == NULL)
                return new_Const(value);
            else
                return result;
        }
    }
    return NULL;
}

// 查看是否已经声明过同一个常数值的临时变量
Operand find_Const(int value)
{
    int i;
    for (i = 0; i < 100; i++)
    {
        if (tempvar[i] == -1)
            break;
        if (temp_Operands[i]->kind == TEMPVAR && temp_Operands[i]->value == value)
            return temp_Operands[i];
    }
    return NULL;
}

```

语句的翻译模式如下:

// 语句的翻译模式

InterCode translate Stmt(tnode Stmt)

```
{
    // Exp SEMI
    if (Stmt->ncld == 2 && !strcmp((Stmt->cld)[1]->name, "SEMI"))
    {
        return translate_Exp((Stmt->cld)[0], NULL);
    }
    // Compst
    else if (Stmt->ncld == 1 && !strcmp((Stmt->cld)[0]->name, "Compst"))
    {
        // Preorder((Stmt->cld)[0],0);
        return translate_CompSt((Stmt->cld)[0]);
    }
    // RETURN Exp SEMI
    else if (Stmt->ncld == 3 && !strcmp((Stmt->cld)[0]->name, "RETURN"))
    {
        // 中间代码优化
        Operand existOp = get_Operand((Stmt->cld)[1]);
        if (existOp == NULL)
        {
            Operand t1 = new_tempvar();
            InterCode code1 = translate_Exp((Stmt->cld)[1], t1);
            InterCode code2 = new_Code();
            code2->kind = _RETURN;
            code2->operands.var = t1;
            return add_Codes(2, code1, code2);
        }
        else
        {
            InterCode code1 = new_Code();
            code1->kind = _RETURN;
            code1->operands.var = existOp;
            return code1;
        }
    }
    // IF LP Exp RP Stmt
    else if (Stmt->ncld == 5 && !strcmp((Stmt->cld)[0]->name, "IF"))
    {
        Operand lable1 = new_lable();
        Operand lable2 = new_lable();
        InterCode code1 = translate_Cond((Stmt->cld)[2], lable1, lable2);
        InterCode code2 = translate_Stmt((Stmt->cld)[4]);
        return add_Codes(4, code1, new_lable_Code(lable1), code2, new_lable_Code(lable2));
    }
    // IF LP Exp RP Stmt ELSE Stmt
    else if (Stmt->ncld == 7 && !strcmp((Stmt->cld)[0]->name, "IF"))
    {
        Operand lable1 = new_lable();
        Operand lable2 = new_lable();
```

```

Operand lable3 = new_lable();
InterCode code1 = translate_Cond((Stmt->cld)[2], lable1, lable2);
// print_Codes(code1);
InterCode code2 = translate_Stmt((Stmt->cld)[4]);
InterCode code3 = translate_Stmt((Stmt->cld)[6]);
return add_Codes(7, code1, new_lable_Code(lable1), code2, new_goto_Code(lable3), new_lat
}
// WHILE LP Exp RP Stmt
else if (Stmt->ncld == 5 && !strcmp((Stmt->cld)[0]->name, "WHILE"))
{
    Operand lable1 = new_lable();
    Operand lable2 = new_lable();
    Operand lable3 = new_lable();
    InterCode code1 = translate_Cond((Stmt->cld)[2], lable2, lable3);
    InterCode code2 = translate_Stmt((Stmt->cld)[4]);
    return add_Codes(6, new_lable_Code(lable1), code1, new_lable_Code(lable2), code2, new_gc
}
return new_Code();
}

```

对于基本表达式的翻译模式参考下图实现：

在进行基本表达式的翻译时可以实现中间代码的优化，对于变量类型、常数类型的操作数不再申请临时变量对其进行存储，直接输出即可，对于临时变量，查看临时变量表中是否有存有同样值的临时变量，如果有直接使用即可。



// 基本表达式的翻译模式

InterCode translate\_Exp(tnode Exp, Operand place)

```
{
    int isCond = 0;
    // INT
    if (Exp->ncld == 1 && !strcmp((Exp->cld)[0]->name, "INT"))
    {
        Operand value = new_Const((Exp->cld)[0]->value);
        InterCode code = new_assign_Code(place, value);
        return code;
    }
    // ID
    else if (Exp->ncld == 1 && !strcmp((Exp->cld)[0]->name, "ID"))
    {
        Operand variable = new_Variable((Exp->cld)[0]->content);
        InterCode code = new_assign_Code(place, variable);
        return code;
    }
    // Exp1 ASSIGNOP Exp2
    else if (Exp->ncld == 3 && !strcmp((Exp->cld)[1]->name, "ASSIGNOP"))
    {
        // Exp1 -> ID
        if ((Exp->cld)[0]->ncld == 1 && !strcmp(((Exp->cld)[0]->cld)[0]->name, "ID"))
        {
            Operand variable = new_Variable(((Exp->cld)[0]->cld)[0]->content);
            Operand existOp = get_Operand((Exp->cld)[2]);
            // 中间代码优化
            if (existOp == NULL)
            {
                Operand t1 = new_tempvar();
                InterCode code1 = translate_Exp((Exp->cld)[2], t1);
                InterCode code2 = new_assign_Code(variable, t1);
                if (place == NULL)
                    return add_Codes(2, code1, code2);
                else
                {
                    InterCode code3 = new_assign_Code(place, variable);
                    return add_Codes(3, code1, code2, code3);
                }
            }
            else
            {
                return new_assign_Code(variable, existOp);
            }
        }
    }
}
```

## 目标代码的生成

此次目标代码生成我们使用的并非是堆栈的模拟，而是使用寄存器的方式存储对应中间代码中的变量、临时变量、常量，再从寄存器中进行调用。此处分配了一共20个寄存器供此虚拟机使用。下面是分配寄存器的操作，返回的字符串是 t0-t20 以及zero，对于常数或者临时变量值为0的直接返回zero，对于常数或者临时变量值为0的直接返回zero，否则查找寄存器中是否已经为该操作数进行了分配，如果已经分配就返回所在寄存器，如果没有就为其分配一个：

```

// 分配寄存器
char *allocate_reg(Operand op)
{
    int i;
    char *regnumber = (char *)malloc(sizeof(char) * 10);
    char *regname = (char *)malloc(sizeof(char) * 10);
    strcat(regname, "$t");
    // 常数0 寄存器
    if (op->kind == CONSTANT && op->operand.value == 0)
        return "$zero";
    else if (op->kind == TEMPVAR && op->value == 0)
        return "$zero";
    // 寻找存储该操作数的寄存器
    int find = 0;
    for (i = 0; i < reg_num; i++)
    {
        if (regs[i] == NULL || regs[i]->kind != op->kind)
            continue;
        if (regs[i]->kind == CONSTANT && regs[i]->operand.value == op->operand.value)
        {
            find = 1;
            break;
        }
        else if (regs[i]->kind == TEMPVAR && regs[i]->operand.tempvar == op->operand.tempvar)
        {
            find = 1;
            break;
        }
        else if (regs[i]->kind == VARIABLE && !strcmp(regs[i]->operand.name, op->operand.name))
        {
            find = 1;
            break;
        }
    }
    if (find)
    {
        Int2String(i, regnumber);
        strcat(regname, regnumber);
        return regname;
    }
    else
    {
        Int2String(reg_num, regnumber);
        strcat(regname, regnumber);
        regs[reg_num] = op;
        reg_num++;
        return regname;
    }
}

```

# 虚拟机

## 执行代码实现思路

定义虚拟机的指令集，包括操作码、操作数类型和操作数值等信息。例如，可以定义一个双向链表节点的操作码为ADD，操作数类型为INT，操作数值为2，表示将链表中前两个节点的值相加，并将结果存储到一个新的节点中。

定义一个双向链表节点的数据结构，包括前驱节点指针prev、后继节点指针next和节点值value等信息。

定义一个虚拟机的状态结构体，包括指令指针ip、双向链表的头指针head、指针sp等信息。

实现flex文件，定义虚拟机的词法分析器。根据指令集的定义，将每个操作码、操作数类型和操作数值映射到对应的token，例如ADD\_INT\_2表示操作码为ADD，操作数类型为INT，操作数值为2。同时，对于每个操作数，将其类型和值存储到一个全局变量中。

实现bison文件，定义虚拟机的语法分析器。根据指令集的定义，定义不同的语法规则

实现虚拟机的执行函数，根据当前指令指针ip和双向链表的头指针head，执行不同的操作。如果当前指令为ADD\_INT\_2，则从寄存器中取出两个整数，将它们相加得到结果，然后创建一个新的节点，将结果存储到该节点中，并将该节点插入到双向链表的头部。

实现一个虚拟机的main函数，读入源程序文件，调用词法分析器和语法分析器，生成虚拟机的指令序列，并将指令序列传递给执行函数执行。

## 执行代码实现方法

这里是建立在中间代码已经实现且已经建立了一个中间代码链表的基础下

### 分配寄存器

函数 `allocate_reg`，用于在一组寄存器中分配一个寄存器，以存储一个操作数。

函数的输入参数是一个操作数 `op`，它的类型是一个自定义的结构体 `Operand`。该结构体包含了操作数的类型 `kind` 和值 `value`，其中 `kind` 可以是常数、临时变量、变量等类型。

函数首先判断操作数是否是常数0或者临时变量且值为0，若是则返回寄存器0。接着，函数遍历所有已经分配的寄存器，查找是否有已经存储该操作数的寄存器。如果找到，则返回该寄存器的编号；否则，将该操作数存储在下一个可用的寄存器中，并返回该寄存器的编号。

注意，函数中的 `regs` 和 `reg_num` 是全局变量，用于保存所有分配的寄存器和已经分配的寄存器数量。这段代码的作用是实现寄存器分配的逻辑，以便于将程序中的变量和临时值存储到寄存器中，从而提高

程序的运行效率。

```
int allocate_reg(Operand op)
{
    int i;
    // 常数0 寄存器
    if (op->kind == CONSTANT && op->operand.value == 0)
        return 0;
    else if (op->kind == TEMPVAR && op->value == 0)
        return 0;
    // 寻找存储该操作数的寄存器
    int find = 0;
    for (i = 0; i < reg_num; i++)
    {
        if (regs[i] == NULL || regs[i]->kind != op->kind)
            continue;
        if (regs[i]->kind == CONSTANT && regs[i]->operand.value == op->operand.value)
        {
            find = 1;
            break;
        }
        else if (regs[i]->kind == TEMPVAR && regs[i]->operand.tempvar == op->operand.tempvar)
        {
            find = 1;
            break;
        }
        else if (regs[i]->kind == VARIABLE && !strcmp(regs[i]->operand.name, op->operand.name))
        {
            find = 1;
            break;
        }
    }
    if (find)
    {
        return i;
    }
    else
    {
        regs[reg_num] = op;
        reg_num++;
        return reg_num-1;
    }
}
```

## 执行中间代码

函数 `execute_Codes`，用于执行一系列的中间代码 `codes`。该函数遍历每个中间代码，并根据代码的类型分别执行相应的操作。

具体来说，该函数首先将中间代码 `codes` 赋值给一个临时变量 `temp`，然后进入一个循环，遍历所有的中间代码。对于每个中间代码，分为以下三种情况：

如果该中间代码是跳转指令 `_GOTO`，则执行跳转操作。具体来说，该代码会获取跳转目标的标签 `lable`，然后从当前指令开始向前遍历中间代码，直到找到标签为 `lable` 的位置，然后跳转到该位置执行。

如果该中间代码是条件跳转指令 `_IFGOTO`，则根据条件进行跳转操作。具体来说，该代码会获取跳转目标的标签 `lable`、跳转条件的运算符 `op`，以及两个操作数 `op1` 和 `op2`。然后根据运算符和操作数的值来判断是否满足跳转条件，如果满足则跳转到标签为 `lable` 的位置执行。

如果该中间代码不是跳转指令，则直接执行该代码。

在执行每个中间代码之前，该函数会根据操作数分配寄存器，以便于提高程序的运行效率。具体来说，函数会调用 `allocate_reg` 函数分配寄存器，并将操作数存储到寄存器中。

该函数的最后会打印一条消息，表示中间代码已经执行完毕。该函数的作用是将中间代码翻译成目标机器代码，并执行该代码。

```

void execute_Codes(InterCode codes)
{
    InterCode temp = new_Code();
    temp = codes;
    while (temp != NULL)
    {
        if(temp->kind==_GOTO){
            Operand lable = temp->operands.jump.lable;
            InterCode ttemp=new_Code();
            ttemp=get_Tail(temp);
            while (1){
                if(ttemp != NULL && ttemp->kind==_LABEL && (int)(ttemp->operands.var)==(int)lable)
                    temp=ttemp;
                break;
            }
            ttemp=ttemp->prev;
        }
        }else if(temp->kind==_IFGOTO){
            char *op = temp->operands.jump.relop;
            Operand lable = temp->operands.jump.lable;
            Operand op1 = temp->operands.jump.op1;
            Operand op2 = temp->operands.jump.op2;
            if (!strcmp(op, "=")){
                if( regs[allocate_reg(op1)]->value == regs[allocate_reg(op2)]->value){
                    Operand lable = temp->operands.jump.lable;
                    InterCode ttemp=new_Code();
                    ttemp=get_Tail(temp);
                    while (1){
                        if(ttemp != NULL && ttemp->kind==_LABEL && (int)(ttemp->operands.var)==(int)lable)
                            temp=ttemp;
                        break;
                    }
                    ttemp=ttemp->prev;
                }
            }
            }else if (!strcmp(op, "!=")){
                if(regs[allocate_reg(op1)]->value != regs[allocate_reg(op2)]->value){
                    Operand lable = temp->operands.jump.lable;
                    InterCode ttemp=new_Code();
                    ttemp=get_Tail(temp);
                    while (1){
                        if(ttemp != NULL && ttemp->kind==_LABEL && (int)(ttemp->operands.var)==(int)lable)
                            temp=ttemp;
                        break;
                    }
                    ttemp=ttemp->prev;
                }
            }
            }else if (!strcmp(op, ">")){
                if(regs[allocate_reg(op1)]->value > regs[allocate_reg(op2)]->value){

```

```

        Operand lable = temp->operands.jump.lable;
        InterCode ttemp=new_Code();
        ttemp=get_Tail(temp);
        while (1){
            if(ttemp != NULL && ttemp->kind==_LABEL && (int)(ttemp->operands.var)==(
                temp=ttemp;
                break;
            )
            }
            ttemp=ttemp->prev;
        }
    }
}
}else if (!strcmp(op, "<")){
    if((regs[allocate_reg(op1)]->value < regs[allocate_reg(op2)]->value)){
        Operand lable = temp->operands.jump.lable;
        InterCode ttemp=new_Code();
        ttemp=get_Tail(temp);
        while (1){
            if(ttemp != NULL && ttemp->kind==_LABEL && (int)(ttemp->operands.var)==(
                temp=ttemp;
                break;
            )
            }
            ttemp=ttemp->prev;
        }
    }
}
}else if (!strcmp(op, ">=")){
    if(regs[allocate_reg(op1)]->value >= regs[allocate_reg(op2)]->value){
        Operand lable = temp->operands.jump.lable;
        InterCode ttemp=new_Code();
        ttemp=get_Tail(temp);
        while (1){
            if(ttemp != NULL && ttemp->kind==_LABEL && (int)(ttemp->operands.var)==(
                temp=ttemp;
                break;
            )
            }
            ttemp=ttemp->prev;
        }
    }
}
}else if (!strcmp(op, "<=")){
    if(regs[allocate_reg(op1)]->value <= regs[allocate_reg(op2)]->value){
        Operand lable = temp->operands.jump.lable;
        InterCode ttemp=new_Code();
        ttemp=get_Tail(temp);
        while (1){
            if(ttemp != NULL && ttemp->kind==_LABEL && (int)(ttemp->operands.var)==(
                temp=ttemp;
                break;
            )
            }
            ttemp=ttemp->prev;
        }
    }
}}
}else{};
}else execute_Code(temp);

```



```
        temp=temp->next;
    }
    printf("打印完毕\n");
}
```

## 执行单条中间代码

函数execute\_Code，用于执行单个中间代码。该函数接受一个中间代码InterCode类型的参数code，然后根据code的类型执行相应的操作。

首先，该函数会检查code是否为空。如果code为空，则输出错误信息并返回。如果code不为空，则使用switch语句判断code的类型。根据不同的类型，执行相应的操作。

如果code的类型是\_NULL，什么也不做，直接break。

如果code的类型是\_LABEL，什么也不做，直接break。

如果code的类型是\_FUNCTION，什么也不做，直接break。

如果code的类型是\_ASSIGN，进行赋值操作。该操作会获取code中左右两边的操作数left和right。如果right是常量，则将其值存入左边的寄存器中。否则，将右边寄存器的值存入左边寄存器中。

如果code的类型是\_ADD，进行加法操作。该操作会获取code中三个操作数result、op1和op2。如果op2是常量，则将op1寄存器中的值加上op2的值，存入result寄存器中。否则，将op1寄存器中的值加上op2寄存器中的值，存入result寄存器中。

如果code的类型是\_SUB，进行减法操作。该操作会获取code中三个操作数result、op1和op2。如果op2是常量，则将op1寄存器中的值减去op2的值，存入result寄存器中。否则，将op1寄存器中的值减去op2寄存器中的值，存入result寄存器中。

如果code的类型是\_MUL，进行乘法操作。该操作会获取code中三个操作数result、op1和op2，将op1寄存器中的值乘上op2寄存器中的值，存入result寄存器中。

如果code的类型是\_DIV，进行除法操作。该操作会获取code中三个操作数result、op1和op2，将op1寄存器中的值除以op2寄存器中的值，存入result寄存器中。

如果code的类型是\_CALL，什么也不做，直接break。

如果code的类型是\_READ，进行读操作。该操作会获取code中的操作数op，然后从标准输入读入一个整数temp，将temp存入op寄存器中。

如果code的类型是\_WRITE，进行写操作。该操作会获取code中的操作数op，然后输出op寄存器中的值。

如果code的类型是\_RETURN，什么也不做，直接break。

在执行每个操作之前，该函数会调用allocate\_reg函数来分配寄存器，以便提高代码的运行效率。具体来说，该函数会将操作数存储到寄存器中，并返回该寄存器的编号。

总的来说，该函数实现了针对中间代码的基本操作，包括赋值、加减乘除、读写操作等。通过寄存器分配，可以提高代码的运行效率。

```

void execute_Code(InterCode code)
{
    if (code == NULL)
    {
        printf("Error, MIPS is NULL\n");
        return;
    }
    switch (code->kind)
    {
        case _NULL:
            break;
        case _LABEL:
        {
            break;
        }
        case _FUNCTION:
        {
            break;
        }
        case _ASSIGN:
        {
            Operand left = code->operands.assign.left;
            Operand right = code->operands.assign.right;
            if (right->kind == CONSTANT)
            {
                if (left->kind == TEMPVAR && right->value == 0)
                    break;
                else{
                    regs[allocate_reg(left)]->value = right->operand.value;
                }
            }
            else
            {
                regs[allocate_reg(left)]->value = regs[allocate_reg(right)]->value;
            }
            break;
        }
        case _ADD:
        {
            Operand result = code->operands.binop.result;
            Operand op1 = code->operands.binop.op1;
            Operand op2 = code->operands.binop.op2;
            if (op2->kind == CONSTANT)
            {
                regs[allocate_reg(result)]->value = regs[allocate_reg(op1)]->value + op2->value;
            }
            else
            {
                regs[allocate_reg(result)]->value = regs[allocate_reg(op1)]->value + op2->value;
            }
        }
    }
}

```

```

        break;
    }
    case _SUB:
    {
        Operand result = code->operands.binop.result;
        Operand op1 = code->operands.binop.op1;
        Operand op2 = code->operands.binop.op2;
        if (op2->kind == CONSTANT)
        {
            regs[allocate_reg(result)]->value = regs[allocate_reg(op1)]->value - op2 -> value;
        }
        else
        {
            regs[allocate_reg(result)]->value = regs[allocate_reg(op1)]->value - regs[allocate_r

        }
        break;
    }
    case _MUL:
    {
        Operand result = code->operands.binop.result;
        Operand op1 = code->operands.binop.op1;
        Operand op2 = code->operands.binop.op2;
        regs[allocate_reg(result)]->value= regs[allocate_reg(op1)]->value * regs[allocate_reg(op2)]->value;
        break;
    }
    case _DIV:
    {
        Operand result = code->operands.binop.result;
        Operand op1 = code->operands.binop.op1;
        Operand op2 = code->operands.binop.op2;
        regs[allocate_reg(result)]->value=regs[allocate_reg(op1)]->value / regs[allocate_reg(op2)]->value;
        break;
    }
    case _CALL:
    {
        break;
    }
    case _READ:
    {
        Operand op = code->operands.var;
        int temp;
        while(!scanf("\n%d",&temp));
        regs[allocate_reg(op)]->value=temp;
        break;
    }
    case _WRITE:
    {
        Operand op = code->operands.var;
        int temp=regs[allocate_reg(op)]->value;
        printf("%d\n",temp);
    }

```

```

        break;
    }
    case _RETURN:
    {
        Operand res = code->operands.var;
        break;
    }
    default:
        break;
    }
}

```

## 测试用例及运行结果

### 用例一


测试变量的连续声明及初始化、常量给变量赋值以及变量给变量赋值、write函数

```

int main()
{
    int a,b,c;
    a = 3+4;
    write(a);
    b = 3*4;
    write(b);
    c=3-4;
    write(c);
    while(c<a)
        c=c+1;
    write(c);
    return 0;
}

```

结果

 微信图片\_202306032229072

### 用例二

测试变量的连续声明及初始化、常量给变量赋值以及变量给变量赋值、write函数、带参数和返回值的函数调用

```
int max(int x,int y)
{
    int temp;
    if(x>y) temp=x;
    else temp=y;
    return temp;
}
int main()
{
    int a,b,c;
    a = 3+4;
    write(a);
    b = 3*4;
    write(b);

    max(a,b);
    return 0;
}
```

结果

 微信图片\_202306032229071

## 用例三

测试变量的连续声明及初始化、常量给变量赋值以及变量给变量赋值、write函数、带参数和返回值的函数调用、测试多个函数定义

```

int max(int x,int y)
{
int temp;
if(x>y) temp=x;
else temp=y;
return temp;
}
int min(int x,int y)
{
int temp;
if(x<y) temp=x;
else temp=y;
return temp;
}

int main()
{
    int a,b,c;
    a = 3+4;
    write(a);
    b = 3*4;
    write(b);

    max(a,b);

    min(a,b);

    return 0;
}

```

结果

 微信图片\_20230603222907

## 总结

本次实验是一个有关微型虚拟机的实现，目标功能十分全面，包括了变量、数组、函数等多方面的实现。在实验过程中，我深刻地认识到了编程的重要性，同时也学到了很多编程技巧和知识。在实验的过程中，我首先需要了解虚拟机的基本原理和实现方法，同时也需要学习相关的编程语言和工具。我花费了很多时间阅读相关文献和教程，并且反复尝试和调试代码，最终才成功地实现了虚拟机的各项功能。在实验中，我深刻地认识到了编程的重要性。编程可以让我们更好地理解计算机的工作原理，同时也可以让我们更好地解决实际问题。通过编程，我不仅学到了技术知识，也提高了自己的逻辑思维和解决问题的能力。在实验中，我也学到了很多编程技巧和知识。例如，我学会了如何使用数组和函数，如何实现变量的声明和初始化，如何进行算术运算和跳转操作等等。这些技巧和知识对于我今后的编程工作都将非常有用。总的来说，本次实验是一次十分有收获的经历。通过实验，我不仅学到了很多编程技巧和

知识，也提高了自己的编程能力和解决问题的能力。我相信这些经验和技能对我今后的学习和工作都将非常有帮助。