编译原理与设计实验报告

姓名/学号： 鲍吴迪/1120173588

班级： 07111704

实验名称

目标代码生成实验

实验目的

(1)了解编译器指令选择和寄存器分配的基本算法;

(2)掌握目标代码生成的相关技术和方法，设计并实现针对 x86/MIPS/RISC-V 的目标代码生成模块;

(3)掌握编译器从前端到后端各个模块的工作原理，目标代码生成模块与 其他模块之间的交互过程。

实验内容

基于 BIT-MiniCC 构建目标代码生成模块，该模块能够基于中间代码选择合 适的目标指令，进行寄存器分配，并生成相应平台汇编代码。

如果生成的是 MIPS 或者 RISC-V 汇编，则要求汇编代码能够在 BIT-MiniCC 集成的 MIPS 或者 RISC-V 模拟器中运行。需要注意的是，config.xml 的最后一 个阶段“ncgen”的“skip”属性配置为“false”,"target"属性设置为“mips”、 “x86”或者“riscv”中的一个。

如果生成的是 X86 汇编，则要求使用 X86 汇编器生成 exe 文件并运行。

实验环境

Macbook pro
2.6Hz 双核 Intel Core i5 内存 8GB

Mac OS Catalina 10.15.1
Intellij IDEA 2018.3.1
JDK 1.8.0

实验设计与实现

本次实验采用中间代码生成实验中产生的四元式序列，对其进行改进和优化，并以其为基础生成目标代码 MIPS 汇编语言。

1. 对生成四元式的改进

在中间代码生成实验的四元式中，做了以下几点改进：

1. 函数调用四元式，原来是直接 jmp 指令跳转到函数对应的行，但这样无法传递函数参数等信息，改为 call+函数名+参数的形式，如下：

```
22:    (Call,Mars_PrintStr,"Please input a number:\n",)
23:    (Call,Mars_GetInt,,)
24:    (=,,,n)
25:    (Call,fibonacci,,)
26:    (=,,,res)
27:    (Call,Mars_PrintStr,"This number's fibonacci value is :\n",)
28:    (Call,Mars_PrintInt,res,)
```

2. 函数参数为二元表达式时，其中函数参数信息保存在符号表中，call 指令生成目标代码时，则在符号表中根据函数名找到参数信息，将需要的参数出栈。

```
12:    (-,num,1,%3)
13:    (Call,fibonacci,,)
14:    (-,num,2,%4)
15:    (Call,fibonacci,,)
```

3. 当 if-else 语句嵌套使用时，使用简单的 if-else 四元式时，由于没有设置 endif 标识，导致跳转的逻辑不正确，因此为每个 if 添加 endif 标识，并对 endif 进行嵌套的跳转：

```
25:    (J,,,endif1)
26:_endif1:
27:    (J,,,18)
28:    (==,flag,1,%6)
29:    (Jt,,,31)
30:    (J,,,33)
31:    (++,sum,,sum)
32:    (Call,Mars_PrintInt,i,)
33:    (J,,,endif2)
34:_endif2:
35:    (J,,,10)
36:    (return,sum,,)
```

4. 原本的四元式没有实现乘除法，添加了相应实现。
5. 原本的四元式没有实现多变量的初始化列表，如 int i,j,flag = 1;只能生成 flag=1 的对应四元式，现将 flag 对应的初始化表达式添加到 i，j 的初始化表达式中进行实现。

```
3:····(=,1,,i)
4:····(=,1,,j)
5:····(=,1,,flag)
```

6. 访问函数调用节点时，在符号表中添加常量字符串，设置其类型为 StringConstant,便于目标代码 data 字段的生成。

```
if(id.value.equals("Mars_PrintStr")){
    if(funcCall.argList.get(0) instanceof ASTStringConstant){
        ASTStringConstant sc =(ASTStringConstant)funcCall.argList.get(0);
        item.Name =  sc.value;
        ASTIdentifier id3 = new ASTIdentifier();
        id3.value = sc.value;
        opnd2 = id3;
    }
    item.Kind = "StringConstant";
    table_0.add(item);

}else if(id.value.equals("Mars_PrintInt")){
    if(funcCall.argList.get(0) instanceof ASTIdentifier){
        ASTIdentifier id2 =(ASTIdentifier) funcCall.argList.get(0);
        item.Name =  id2.value;
        opnd2 = id2;
    }
    item.Kind = "IntegerConstant";
    table_0.add(item);
```

2. 寄存器分配

由于 MIPS 汇编中通用寄存器的数量比较多，进行寄存器分配时，使用 regcount 变量对其进行计数，每次使用后更新 regcount（循环更新）。

对于符号表中已经记录的变量，如函数参数、函数过程中定义的变量等，在生成目标代码的开始阶段为其分配一个寄存器，并写入（寄存器<->标识符）的 list 中。

```
//分配寄存器
for(int i=1;i<table_g.size();i++){
    for(int j=1;j<table_g.get(i).size();j++){
        TableItem item= table_g.get(i).get(j);
        if(item.Kind.equals("VariableDeclarator")){
            addvalue(regcount,item.Name,table_g.get(i).get(0).Name);
            regcount = renew(regcount);
        }
    }
}
```

3. 常量字符串保存

遍历符号表，如果找到类型为 StringConstant 的表项，则为其建立 data 项，data 项数据结构如下：

```
public class DataItem {
    public String data_name;
    public String data_value;
```

name 为字符串常量名称，如_1sc,_2sc,在传入参数时，根据调用函数中的四元式：

```
22:    (Call,Mars_PrintStr,"Please input a number:\n",)
```

查找 dataitem 项，并将其名字写入需要的寄存器

4. 函数调用

判断是系统函数还是自定义函数，是否需要传入参数、是否需要接收返回值、是否需要通过堆栈保护现场。
如输出字符串不需要接收返回值，而需要传入函数参数，在调用 Mars_PrintStr 前，已经将其字符串值保存在寄存器中，并将该寄存器压入栈 regstack，此时只需要将其弹出并存入子函数调用的参数寄存器$4 中即可。

```
if(str_a[1].equals("Mars_PrintStr")){//输出字符串，不需要获取函数返回结果
    for(int j=0;j<dataItemList.size();j++){
        if(str_a[2].equals(dataItemList.get(j).data_value)){
            this_code.add("\tla $"+regcount+", "+dataItemList.get(j).data_name);
            regstack.push( item: "$"+regcount);
            regcount=renew(regcount);
        }
    }
    int back_reg=0;
    this_code.add("\tsw $"+regcount+", -4($fp)");
    back_reg = regcount;
    regcount = renew(regcount);
    this_code.add("\tsubu $sp, $sp, 4");
    this_code.add("\tsw $fp, ($sp)");
    this_code.add("\tmove $fp, $sp");
    this_code.add("\tsw $31, 20($sp)");
    this_code.add( "\tmove $4, "+regstack.pop());//传入函数参数
    tmp= ("\tjal "+str_a[1]);
    this_code.add(tmp);
    this_code.add("\tlw $31, 20($sp)");
    this_code.add("\tlw $fp, ($sp)");
    this_code.add("\taddu $sp, $sp, 4");
```

而对于 Mars_GetInt 这类不需要传入参数，而需要接收函数返回值的函数，需要申请一个新寄存器存储函数返回的值（存在$2 中）

```
}else if(str_a[1].equals("Mars_GetInt")){//接收输入，不需要传入参数
    this_code.add("\tsubu $sp, $sp, 4");
    this_code.add("\tsw $fp, ($sp)");
    this_code.add("\tmove $fp, $sp");
    this_code.add("\tsw $31, 20($sp)");
    tmp= ("\tjal "+str_a[1]);
    this_code.add(tmp);
    this_code.add("\tlw $31, 20($sp)");
    this_code.add("\tlw $fp, ($sp)");
    this_code.add("\taddu $sp, $sp, 4");
    this_code.add("\tmove $"+regcount+", $2");//保存子函数的返回结果
    regstack.push( item: "$"+regcount);
    regcount = renew(regcount);
```

如果调用用户自定义的函数，则需要查找符号表中关于根据函数名找到该函数的参数个数和参数名称，根据参数名称获取到存储参数的寄存器后，进行

sw 和 lw 操作。

```
for(int k=1;k<table_g.size();k++){
    if(table_g.get(k).get(0).Name.equals(str_a[1])){
        para_name = table_g.get(k).get(1).Name;
    }
}
```

```
                this_code.add("\tsw "+get(para_name)+", -4($fp)");
                this_code.add("\tsubu $sp, $sp, 4");
                this_code.add("\tsw $fp, ($sp)");
                this_code.add("\tmove $fp, $sp");
                this_code.add("\tsw $31, 20($sp)");
                if(str_a.length<3){
                    this_code.add( "\tmove $4, "+regstack.pop());//传入函数参数
                }else{
                    this_code.add( "\tmove $4, "+get(str_a[2]));
                }

            }
        }
        tmp= ("\tjal "+str_a[1]);
        this_code.add(tmp);
        this_code.add("\tlw $31, 20($sp)");
        this_code.add("\tlw $fp, ($sp)");
        this_code.add("\taddu $sp, $sp, 4");
        this_code.add("\tlw "+get(para_name)+", -4($fp)");
        //this_code.add("\tlw $"+back_reg+", -4($fp)");
        this_code.add("\tmove $"+back_reg+", $2");//保存子函数的返回结果
        regstack.push( item: "$"+back_reg);
        regcount = renew(regcount);
    }
```

5. 代码段标识

一开始是根据原本四元式中每一行的标号，将跳转语句需要的标号加入 segItemList 列表，在扫描每一行时，判断其行号有没有出现在 segItemList 中，如果有，则为他设置标识符，标识符为 L1，L2，L3……

但这样做的 bug 是，如果出现后面的语句跳转到前面的语句时，不能再重新为前面的语句添加标识，

因此，改进的做法是，对中间代码生成的四元式文件进行两遍扫描，第一次扫描所有的跳转语句，并将需要跳转的行数加入 SegItemList 中，并为其分配代码段的标识符。

```
for(int i=0;i<srcLines.size();i++){//先扫描一遍跳转语句，将标识和行数加入segItemList
    String four = srcLines.get(i);
    String str = "";
    for(int j=0;j<four.length();j++){
        if(four.charAt(j)=='('){
            for(int k=j+1;four.charAt(k)!=')';k++){
                str+=four.charAt(k);
            }
        }
    }
    String str_a[] = str.split( regex: ",");
    if(str_a[0].equals("J")||str_a[0].equals("Jt")||str_a[0].equals("Jf")){
        SegItem segItem = new SegItem( name: "L"+segcount,str_a[3]);
        segItemList.add(segItem);
        segcount+=1;
    }
```

第二次扫描是，对于每行，判断其行数，如果在 list 中则设置对应的标识：

```
for(int i=0;i<srcLines.size();i++){
    ArrayList<String> this_code=new ArrayList<String>();
    String four = srcLines.get(i);
    //判断是否需要添加代码段的标识
    String line = "";
    for(int j=0;four.charAt(j)!=':';j++){
        line+=four.charAt(j);
    }
    for(int j=0;j<segItemList.size();j++){
        if(segItemList.get(j).linecount.equals(line)){
            this_code.add(segItemList.get(j).Segname+":");
        }
    }
```

6. 选择语句

使用原来的 Jt 和 J 指令实现的 if-then-else 语句存在这样的弊端：then 语句执行完后没有跳转，没有 endif 标识，不知道选择语句进行分支后在哪里汇合，尤其是存在多个 if-else 嵌套时逻辑更加混乱。

因此在四元式的生成中，每执行完一个 then 分支，则添加一个(J,,,endif)语句，并为不同的 endif 进行编号：

```
if(endflag){
    String endif = "_endif"+(endif_count-2);
    Quat quat_endif = new Quat(cursor++,endif);
    quats.add(quat_endif);
    endflag = false;
    opnd1 = null;
    opnd2 = null;
    res = new ASTIdentifier();
    ((ASTIdentifier) res).value = "endif"+(endif_count-1);
    Quat quat_jumpif = new Quat(cursor++, op: "J",res,opnd1,opnd2);
    quats.add(quat_jumpif);
}else{
    String endif = "_endif"+(endif_count-1);
    Quat quat_endif = new Quat(cursor++,endif);
    quats.add(quat_endif);

}
```

在目标代码生成阶段，为代码段添加相应的 endif 标识。

```
}else{
    if(func_name.equals("")){//endif标志
        String tmp = "";
        for(int j=0;j<four.length();j++){
            if(four.charAt(j)==':'){
                for(int k=j+1;k<four.length();k++){
                    tmp+=four.charAt(k);
                }
                break;
            }
        }
        this_code.add(tmp);
```

代码生成结果：

```
L4:
    li $22, 1
    j _endif2
L5:
    sw $19, -4($fp)
    li $22, 1
    sub $21, $19, $22
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    move $4, $21
    jal fibonacci
    lw $31, 20($sp)
    lw $fp, ($sp)
    addu $sp, $sp, 4
    lw $19, -4($fp)
    move $20, $2
    sw $19, -4($fp)
    li $18, 2
    sub $17, $19, $18
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    move $4, $17
    jal fibonacci
    lw $31, 20($sp)
    lw $fp, ($sp)
    addu $sp, $sp, 4
    lw $19, -4($fp)
    move $16, $2
    add $22, $16, $20
_endif1:
    j _endif2
_endif2:
    move $2, $22
    move $sp, $fp
    jr $31
```

7. 返回语句

通过 isdigit（）函数判断返回值是变量还是常量：

```
public boolean isdigit(String s){
    boolean digit = true;
    for(int k=0;k<s.length();k++){//判断字符串内是数字还是标识符
        if(!Character.isDigit(s.charAt(k))){
            digit = false;
        }
    }
    return digit;
}
```

如果需要返回常量，则直接申请一个临时寄存器用于储存常量的值，并返回，如果是变量，则需要通过 get（）函数获取该变量所在的寄存器，并将寄存器中的内容返回。

```
if(isdigit){
    this_code.add("\tli $"+tmpregcount+", "+str_a[1]);
    this_code.add("\tmove $2, $"+tmpregcount);
    this_code.add("\tmove $sp, $fp");
    this_code.add("\tjr $31");
    tmpregcount=renew_tmp(tmpregcount);
}else{//需要返回的是变量
    this_code.add("\tmove $2, "+get(str_a[1]));
    this_code.add("\tmove $sp, $fp");
    this_code.add("\tjr $31");
}
```

8. 一元/二元运算

一元运算和二元运算按照操作数的个数进行判断，如果缺少操作数，则从寄存器栈中弹出相应的操作数，对于计算结果，如果存在需要存储的变量，则将结果存入该变量所在的寄存器中，否则将计算结果入栈。

```
}else if(str_a[0] .equals( "-")){
    String tmp = "\tsub ";
    for(int j=1;j<3;j++){
        boolean isdigit = true;
        for(int k=0;k<str_a[j].length();k++){//判断字符串内是数字还是标识符
            if(!Character.isDigit(str_a[j].charAt(k))){
                isdigit = false;
            }
        }
        if(isdigit){
            this_code.add("\tli $"+regcount+", "+str_a[j]);//是数字就先存到寄存器中
            regstack.push( item: "$"+regcount);
            regcount = renew(regcount);
        }else{
            regstack.push(get(str_a[j]));//是变量就将变量所在的寄存器入栈
            this_code.add("\tsw "+get(str_a[j])+", -4($fp)");
        }
    }
    tmp+="$"+regcount;
    regcount = renew(regcount);
    String reverse = regstack.pop();
    tmp+= ", "+regstack.pop();
    tmp+=", "+reverse;

    this_code.add(tmp);
    regstack.push( item: "$"+(regcount+1));//将减法结果入栈
```

对于比较运算如<=、<比较，主要使用了 MIPS 中的 sle/slt 组合 beq/bne 指令进行实现，

```
if(str_a[0].equals("<")|| str_a[0].equals("<=")){
    String tmp = "";
    String tmp_reg1;
    if(isdigit(str_a[2])){
        if(str_a[0].equals("<")){
            int test = Integer.parseInt((str_a[2]));
            tmp = "li $"+regcount+", "+test;
        }
        else {
            int test = Integer.parseInt((str_a[2]));
            test +=1;
            tmp = "li $" + regcount + ", " + test;
        }
        tmp_reg1="$"+regcount;
        regcount = renew(regcount);
        this_code.add("\t"+tmp);
    }else {
        tmp_reg1 = get(str_a[2]);
    }
    tmp="";
    if(str_a[0].equals("<"))
        tmp+= ("slt "+"$"+regcount+", ");
    else if(str_a[0].equals("<="))
        tmp+= ("slt "+"$"+regcount+", ");
    regstack.push( item: "$"+regcount);
    regcount = renew(regcount);
    String tmp_reg2;
    if (str_a[1].charAt(0) == '%') {
        tmp_reg2 = regstack.pop();
    }else
        tmp_reg2 = get(str_a[1]);
    tmp+=tmp_reg2+", "+tmp_reg1;
    this_code.add("\t"+tmp);
```

对于相等比较，本来可以直接用 beq/bne 实现，但由于自己实现的四元式中使用了跳转指令 J，这样在生成目标代码时会造成重复跳转，因此使用了 sub 指令，相减后再与 0 进行比较。

```
}else if(str_a[0].equals("==")){
    String tmp = "";
    tmp = "li $"+regcount+", "+str_a[2];
    String tmp_reg1="$"+regcount;
    regcount = renew(regcount);
    this_code.add("\t"+tmp);

    String tmp_reg2 ;
    if(str_a[1].charAt(0)=='%')
        tmp_reg2 = regstack.pop();
    else
        tmp_reg2= get(str_a[1]);
    String aim_reg = "$"+regcount;
    this_code.add("\tsub "+aim_reg+", "+tmp_reg1+", "+tmp_reg2);
    regstack.push(aim_reg);
    regcount = renew(regcount);
```

9. 整体测试与总结

对于测试代码 1_Fibonacci.c，生成的汇编代码为：

```
        .data
```

```
blank : .asciiz " "

_1sc : .asciiz "Please input a number:\n"

_2sc : .asciiz "This number's fibonacci value is :\n"

.text

__init:

    lui $sp, 0x8000

    addi $sp, $sp, 0x0000

    move $fp, $sp

    add $gp, $gp, 0x8000

    jal main

    li $v0, 10

    syscall

Mars_PrintInt:

    li $v0, 1

    syscall

    li $v0, 4

    move $v1, $a0

    la $a0, blank

    syscall

    move $a0, $v1

    jr $ra

Mars_GetInt:

    li $v0, 5

    syscall

    jr $ra

Mars_PrintStr:
```

```
    li $v0, 4
    syscall
    jr $ra
fibonacci:
    subu $sp, $sp, 32
    move $25, $4
    move $19, $25
    li $18, 1
    slt $17, $19, $18
    bne $17, $0, L1
    j L2
L1:
    li $22, 0
    j _endif1
L2:
    li $16, 3
    slt $23, $19, $16
    bne $23, $0, L4
    j L5
L4:
    li $22, 1
    j _endif2
L5:
    sw $19, -4($fp)
    li $22, 1
    sub $21, $19, $22
```

```
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    move $4, $21
    jal fibonacci
    lw $31, 20($sp)
    lw $fp, ($sp)
    addu $sp, $sp, 4
    lw $19, -4($fp)
    move $20, $2
    sw $19, -4($fp)
    li $18, 2
    sub $17, $19, $18
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    move $4, $17
    jal fibonacci
    lw $31, 20($sp)
    lw $fp, ($sp)
    addu $sp, $sp, 4
    lw $19, -4($fp)
    move $16, $2
    add $22, $16, $20
```

```
_endif1:
    j _endif2
_endif2:
    move $2, $22
    move $sp, $fp
    jr $31
main:
    subu $sp, $sp, 28
    la $22, _1sc
    sw $21, -4($fp)
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    move $4, $22
    jal Mars_PrintStr
    lw $31, 20($sp)
    lw $fp, ($sp)
    addu $sp, $sp, 4
    lw $21, -4($fp)
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    jal Mars_GetInt
    lw $31, 20($sp)
```

```
    lw $fp, ($sp)
    addu $sp, $sp, 4
    move $20, $2
    move $21, $20
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    move $4, $21
    jal fibonacci
    lw $31, 20($sp)
    lw $fp, ($sp)
    addu $sp, $sp, 4
    lw $19, -4($fp)
    move $19, $2
    move $20, $19
    la $17, _2sc
    sw $16, -4($fp)
    subu $sp, $sp, 4
    sw $fp, ($sp)
    move $fp, $sp
    sw $31, 20($sp)
    move $4, $17
    jal Mars_PrintStr
    lw $31, 20($sp)
    lw $fp, ($sp)
```

```
addu $sp, $sp, 4

lw $16, -4($fp)

subu $sp, $sp, 4

sw $fp, ($sp)

move $fp, $sp

sw $31, 20($sp)

move $4, $20

jal Mars_PrintInt

lw $31, 20($sp)

lw $fp, ($sp)

addu $sp, $sp, 4

li $25, 0

move $2, $25

move $sp, $fp

jr $31
```

除了寄存器的分配和使用，其他语句所用指令、生成的代码与内置代码生成器基本相同，输出结果为：

```
7. Target code generation finished!
8. Simulating
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Please input a number:
0
This number's fibonacci value is :
0
```

```
7. Target code generation finished!
8. Simulating
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Please input a number:
1
This number's fibonacci value is :
1
```

```
7. Target code generation finished!
8. Simulating
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Please input a number:
2
This number's fibonacci value is :
1
```

```
7. Target code generation finished!
8. Simulating
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Please input a number:
3
This number's fibonacci value is :
2
```

```
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Please input a number:
4
This number's fibonacci value is :
3
```

对于 fibonacci 的每个分支都能输出正确的结果，递归调用一次（计算 3 和 4 的 fibonacci 数）时也能运行正确。但在后续测试过程中，对于需要多次函数递归实现的数（如 5）出现了错误。

根据 mars 对生成的 mips 代码进行调试发现，是由于堆栈操作存在的不知名错误导致递归调用函数时寄存器变量没有正确的保存和恢复造成的，但由于时间原因，没能把代码修改正确。

另外选用了 2_Prime.c 的测试用例，也只能达到生成汇编代码并编译通过的程度，不能输出正确结果。

实验心得体会

整个实验实现难度比较大，自己也没能实现老师的要求，感到有点挫败，但依然学会了很多，且明白自己在编译原理的学习道路上任重道远。

一开始从 ICGen 传入自己在上个实验生成的符号表就花费了不小的功夫，最后改了 CodeGen 的 run 函数和 MiniCCompiler 的对应方法勉强实现了。

另外一个卡了比较久的 bug 是在调用函数前后，对保存结果的通用寄存器的值没有保存，导致在计算 3 的 fibonacci 数时，对于 num-1 计算后再计算 num-2 的时候，num 的值被改变，不能传入正确的参数。

通过这次实验，首先是对 MIPS 汇编有了初步的了解，但代码生成时，多数操作比如什么时候存入寄存器，什么时候操作堆栈等主要还是照着 bitminicc 内置生成器生成的 MIPS 汇编代码依葫芦画瓢，所以出现了堆栈操作方面的错误也没有及时的改正，后续还有很多需要学习的地方。

另外通过最后的一次实验对前面的代码进行了改善，明白了自己生成的中间代码哪些信息不完全，对其进行重新调整和补全，对一整个学期的编译原理实验进行了收尾和总结，收获还是非常大的。