编译原理 Lab8: 目标代码生成实验

郑子帆 1120200822

北京理工大学 计算机学院 07112002 班

日期：2023 年 6 月 9 日

摘　　要

本文为北京理工大学《编译原理与设计 2023》课程的 Lab8 实验报告。在本次实验我们设计对于 Lab7 中所设计的四元式进行了向目标代码的转化，生成了 masm32 代码并通过了 nc_test 中前 6 个 c 语言程序。

1　实验简介 [1]

1.1　实验目的

1. 了解编译器指令生成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/ARM 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。

1.2　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。

2 实验过程

2.1 四元式代码修改

在本次实验中，我发现了自己在 lab7 的四元式实现中存在设计缺陷和代码实现上的 bug，故在这次实验中进行了逐步地完善。

首先我们在 lab7 编写程序的时候，只考虑到了 Array Access，也就是数组访问，但是没有考虑到变量声明时使用数组值进行赋值，于是先加入了该部分代码，如下：

```
// 单目运算 或者 函数调用返回值赋值 或者数组值
else if (expr instanceof ASTPostfixExpression
        || expr instanceof ASTUnaryExpression
        || expr instanceof ASTFunctionCall
        || expr instanceof  ASTArrayAccess) {
    this.visit(expr);
    opnd1 = map.get(expr);
}
Quat quat1 = new Quat(op, declarator, opnd1, opnd2);
quats.add(quat1);
map.put(initDeclarator, declarator);
```

其次，在 visit ArrayAccess 时也有一些生成四元式上的错误，比如在原来的代码里，我们回先判断 arrayAccess.arrayName 是否是 ASTArrayAccess 类节点，如果不是则不会进入循环，这就导致了我们无法对一维数组的访问生成正确的四元式，修改后则先取出 arrayAccess.elements.get(0)，也就是 ASTArrayAccess 节点的下标节点，代码如下：

```
// 循环得到每一维数组下标
LinkedList<ASTNode> index = new LinkedList<>();

this.visit(acsIndex);
ASTNode rest = map.get(acsIndex);
index.addFirst(rest);

while (compoundArrayName instanceof ASTArrayAccess) {
    acsIndex = ((ASTArrayAccess)
        compoundArrayName).elements.get(0);
    compoundArrayName = ((ASTArrayAccess)
        compoundArrayName).arrayName;
```

```
    this.visit(acsIndex);
    rest = map.get(acsIndex);
    index.addFirst(rest);
}
```

另外，在 FunctionCall 部分增添了对于 Mars_GetInt 的特判，因为调用这个函数时我们需要先将要读入的数放入一个临时变量中，代码如下：

```
else if (functionName.equals("Mars_GetInt")) {
    // 需要将读入的数存到一个临时变量中
    ASTNode tmp = new TemporaryValue(++ tmpId);
    Quat quat = new Quat("Call", tmp, funcCall.funcname, null);
    quats.add(quat);
    map.put(funcCall, tmp);
}
```

最后，在调试过程中我还进行了一些 bug 修复，在此不再展示，详见 icgen 目录下的代码。

2.2 MyCodeGen 代码编写

在本次实验中，因为 masm32 的环境已经在上学期的《汇编语言与接口》课程中配置完成，故本实验将四元式代码转化成 masm32 汇编代码，并在 masm32 环境中运行测试。

2.2.1 准备工作

首先，在我们建立了一个 StringBuilder 对象 asmCode，使用其 append 接口完成目标代码的加入，并最终将 asmCode 所包含的内容输出到.code.s 文件中。

其次，我们定义了 MyQuat 类，是 icgen 中的 quat 类的“翻版”，即我将所有类中成员的类型从 ASTNode 改成了 String，这样可以省去在目标代码生成中反复在 ASTNode 和 String 中进行转化。

另外，在方法 run 中，我们进行了.ic.txt 文件的读入和四元式的处理，因为在如 3.PerfectNumber.c 程序中会出现如 c 这样在 masm32 中不认可的变量名，所以我们需要额外开设一个 map 用于将这些变量名映射到在 masm32 中合法的变量名。

最后，由于所有四元式代码中的临时变量都是形如%<id> 且在 masm32 中不被认可，所以我们扫描过程中将其自动改成形如 tmp<id> 的临时变量名。代码如下：

```
// 这里我们使用的是 x86 汇编，所以这里只实现了 x86 部分
else if (cfg.target.equals("x86")) {
    // 先读入.ic.txt 文件
    ArrayList<String> quatList = MiniCCUtil.readFile(iFile);
    for (String quat0 : quatList) {// 删除四元式中的左、右括号
        String[] elem = quat0.split(",");// 根据"," 分隔开四元式的 4 个元素
        elem[0] = elem[0].substring(1, elem[0].length());
        elem[3] = elem[3].substring(0, elem[3].length()-1);

        // 同时将四元式中的%<id> 换成 tmp<id>
        for(int jj = 0; jj < 4; jj ++) {
            if(elem[jj].length() > 1 && elem[jj].charAt(0) == '%') {
                elem[jj] = "tmp" + elem[jj].substring(1, elem[jj].length(
            }
            if(invertIdf.get(elem[jj]) != null) {
                elem[jj] = invertIdf.get(elem[jj]);
            }
        }
        MyQuat qquat = new MyQuat(elem[0], elem[3], elem[1], elem[2]);
        quatStr.add(qquat);
    }

    genMASMCode();
    try {
        FileWriter fileWriter = new FileWriter(new File(oFile));
        fileWriter.write(asmCode.toString());// 创建.code.s 的输出文件并将生
        fileWriter.close();
    } catch (IOException e) {
        e.printStackTrace();
    }

    System.out.println("7. Target code generation finished!");
    return oFile;
}
```

2.2.2 目标代码生成

对于代码生成，我们将其分成了三个模块，分别为引入头文件 (genHeader)、数据区 (genData) 和代码区 (genCode)。对于 genHeader，我们直接将固定的内容 append 到 asmCode 即可。注意我们还需要引入 scanf 和 printf 用于输入和输出。

对于 genData，我们需要在.data 区域定义一些全局变量，在本实验中我将 Mars_PrintStr 要打印的内容作为全局变量。另外，还需要添加形如%d 的字符串，所以我们扫描整个四元式代码，如果有作为 Mars_PrintStr 的 Arg 则提出来。代码如下：

```
public void genData() {
    asmCode.append("\n");
    asmCode.append("""
.data
scanFmt^^I    db    '%d', 0
printIntFmt db    '%d ', 0
printStrFmt db    '%s', 0
    """);

    int nowIdx = 0;// 记录当前扫到的四元式的 idx
    int Mars_cnt = 0;// 记录输出的字符串作为全局变量的数量
    MyQuat nowQuat = quatStr.get(nowIdx);
    while(nowIdx < quatStr.size()) {
        String op = nowQuat.getOp();
        String res = nowQuat.getRes();
        String opnd1 = nowQuat.getOpnd1();
        String opnd2 = nowQuat.getOpnd2();

        if(op.equals("Arg") && opnd1.equals("Mars_PrintStr")) {^^I
            // 扫到用 Mar_PrintStr 输出字符串，存成全局变
            if(res.equals("\"\\n\"")) {// 只有换行
                asmCode.append("Mars_PrintStr" +
                    Mars_cnt).append(" db ").append("0ah, 0\n");
            }
            else if(res.contains("\\n")) {// 包含换行
                res = res.replace("\\n", "");
                asmCode.append("Mars_PrintStr" + Mars_cnt)
                .append(" db ").append(res).append(", 0ah, 0\n");
```

```
                }
                else {// 输出内容没有换行
                    asmCode.append("Mars_PrintStr" + Mars_cnt)
                    .append(" db ").append(res).append(", 0\n");
                }
                Mars_cnt ++;
            }
            nowIdx ++;
            if(nowIdx >= quatStr.size()) break;
            nowQuat = quatStr.get(nowIdx);
        }
        asmCode.append("\n");^^I// 换行
}
```

genCode 则是重点部分，在这里我们按每个函数分别进行扫描和代码生成。我们同时做了一些化简，比如将所有临时变量都设成 local 局部变量；然后对于多维数组，结合其行存储的性质，我们可以只定义成一维数组进行存取。

第一步我们需要建立声明函数语句，第二步则是扫描有关该函数定义的所有四元式中的局部变量与临时变量，都设置成 local，代码如下：

```
// 先添加 <funcname> proc <param1> ... 的函数声明语句
MyQuat quat0 = quatStr.get(scanIdx);
String funcName = quat0.getOpnd2();
asmCode.append(funcName).append(" proc");

scanIdx ++; if(scanIdx >= quatStr.size()) return;
MyQuat quat1 = quatStr.get(scanIdx);
while(quat1.getOp().equals("Param")) {
    String paramRes = quat1.getRes();
    asmCode.append(" ").append(paramRes).append(":dword");
    locVar.add(quat1.getRes());

    scanIdx ++; if(scanIdx >= quatStr.size()) break;
    quat1 = quatStr.get(scanIdx);
}
asmCode.append("\n");
System.out.println("Function Declaration Finished.");
```

```
// 将所有形如%id 的临时变量 或 局部变量定义成 local 的布局变量
quat1 = quatStr.get(scanIdx);
int tmpIdx = scanIdx;
while(tmpIdx < quatStr.size()) {
    String op = quat1.getOp(); String res = quat1.getRes();
    String opnd1 = quat1.getOpnd1(); String opnd2 = quat1.getOpnd2();

    if(op.equals("Endp")) {// 函数结束，退出
        break;
    }
    else if(res.equals("")) {// 没有新的临时变量，就继续扫
        tmpIdx ++;
        if(tmpIdx >= quatStr.size()) break;
        quat1 = quatStr.get(tmpIdx);
        continue;
    }
    else if(op.equals("Arr")) {
        if(locVar.search(res) == -1) {// 数组没有被定义
            int totSize = calcArrSize(opnd1);
            String arrVar = res + "[" + totSize + "]";
            asmCode.append("local ").append(arrVar).append(":dword\n");
            locVar.add(arrVar);
        }
    }
    else if(op.equals("Var")
            || (op.equals("Arg") && !opnd1.equals("Mars_PrintStr"))
            || (res.length() >= 3 && res.substring(0, 3).equals("tmp")))
        if(locVar.search(res) == -1) {// 还要特判不是数，如果是数不建 local
            if(!Character.isDigit(res.charAt(0))) {
                asmCode.append("local ").append(res).append(":dword\n");
                locVar.add(res);
            }
            else {// 是数字
                String invertCon = "Con" + res;
                asmCode.append("local ").append(invertCon).append(":dword
                locVar.add(invertCon);
```

```
                invertIdf.put(res, invertCon);
            }
        }
    }

    tmpIdx ++;
    if(tmpIdx >= quatStr.size()) break;
    quat1 = quatStr.get(tmpIdx);
}
System.out.println("Finished localing.");
```

最后一步就是重头扫描这个函数的所有四元式，对于运算部分逐一生成目标代码。基本上分为四类，分别是跳转类、算术运算类、逻辑运算类和函数调用类。

- 跳转类。主要难点在于 Jnt 的条件判断，我们设置变量 typeJnt，在逻辑运算中被设置，用于指定是哪种比较，如 >= 对应的 typeJnt 是 jl。
- 算术运算类。这里我们全部借助 eax 寄存器完成相关操作，如 + 运算，则是先将 opnd1 存到 eax，然后 opnd2 加到 eax，最后 eax 的值存入 res。值得注意的是取模运算和除法运算，需要先将 edx 寄存器置 0。另外，访问数组和赋值给数组元素时我们需要用到 esi 寄存器，即将计算好的地址存入 esi 进行数组访问。
- 逻辑运算类。主要需要根据对应的符号设置 typeJnt，在上文已经说明。
- 函数调用类，先特判 Mars_GetInt, Mars_PrintInt, Mars_PrintStr。对于其他普通函数调用，维护一个 argLists 存储参数相关值，然后生成对应的 invoke 语句。

由于这部分代码篇幅较长，故在附录A进行展示。

3 实验结果

3.1 配置 *config.xml*

配置 *config.xml*，具体内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<config name="config.xml">
    <phases>
        <phase>
          <phase skip="true" type="java" path="" name="preprocess" />
          <phase skip="false" type="java" path="" name="scan" />
          <phase skip="false" type="java" path="" name="parse" />
          <phase skip="false" type="java" path="" name="semantic" />
```

```
            <phase skip="false" type="java"
              path="bit.minisys.minicc.icgen.IMiniCCICGen" name="icgen" />
            <phase skip="false" type="java"
              path="bit.minisys.minicc.ncgen.IMiniCCNCGen" name="icgen" />

            ...
        </phase>
    </phases>
</config>
```

3.2 运行截图

本次实验我实现的程序可以通过 0_BubbleSort.c, 1_Fibonacci.c, 2_Prime.c, 3_PerfectNumber.c, 4_CounterClockwiseRotationArray4_4.c, 5_YangHuiTriangle.c 这 6 个测试程序。运行截图分别如下。

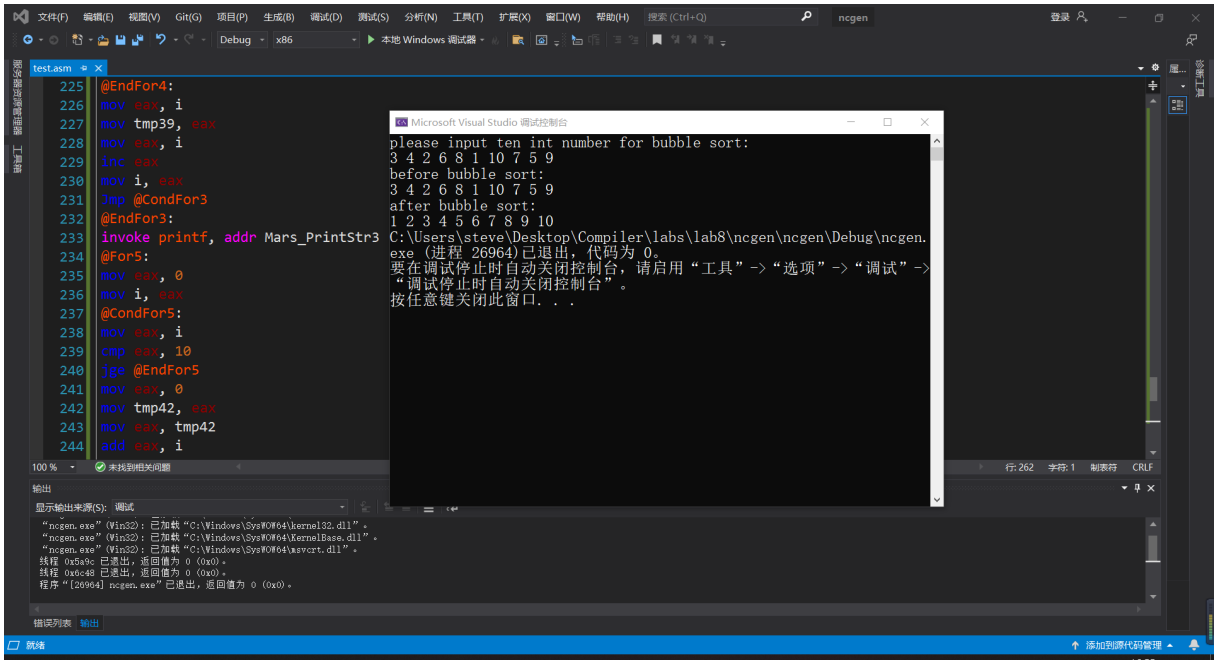

图 1: 0_BubbleSort.c 运行截图

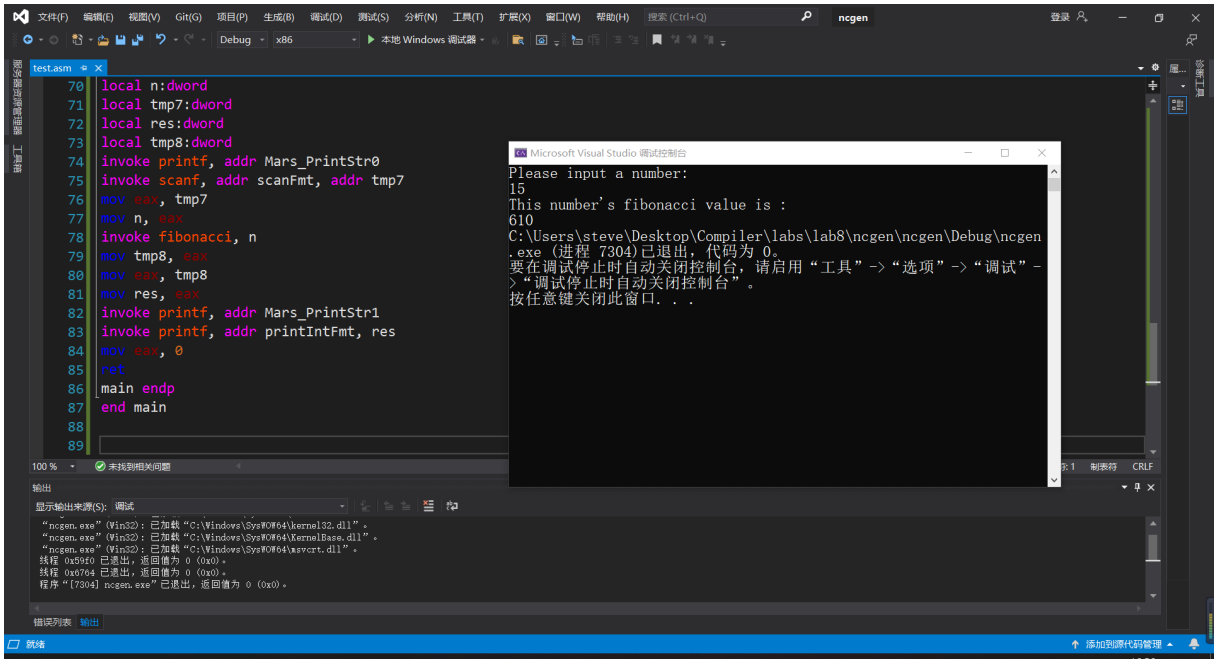

图 2: 1_Fibonacci.c 运行截图

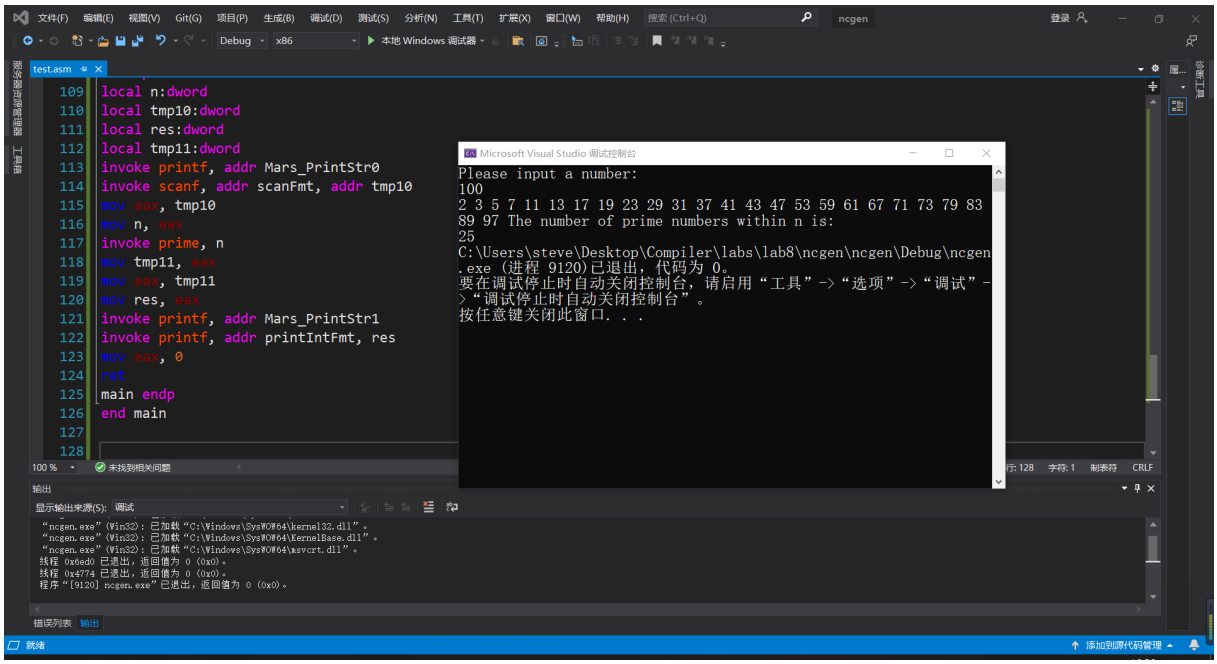

图 3: 2_Prime.c 运行截图

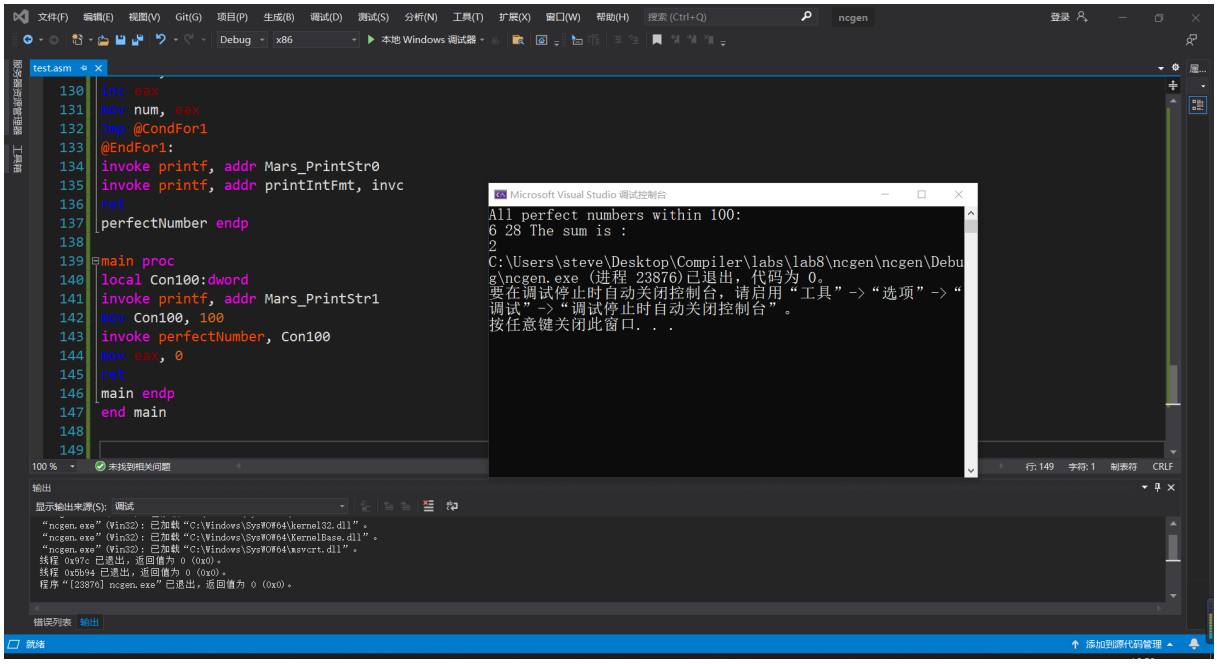

图 4: 3_PerfectNumber.c 运行截图

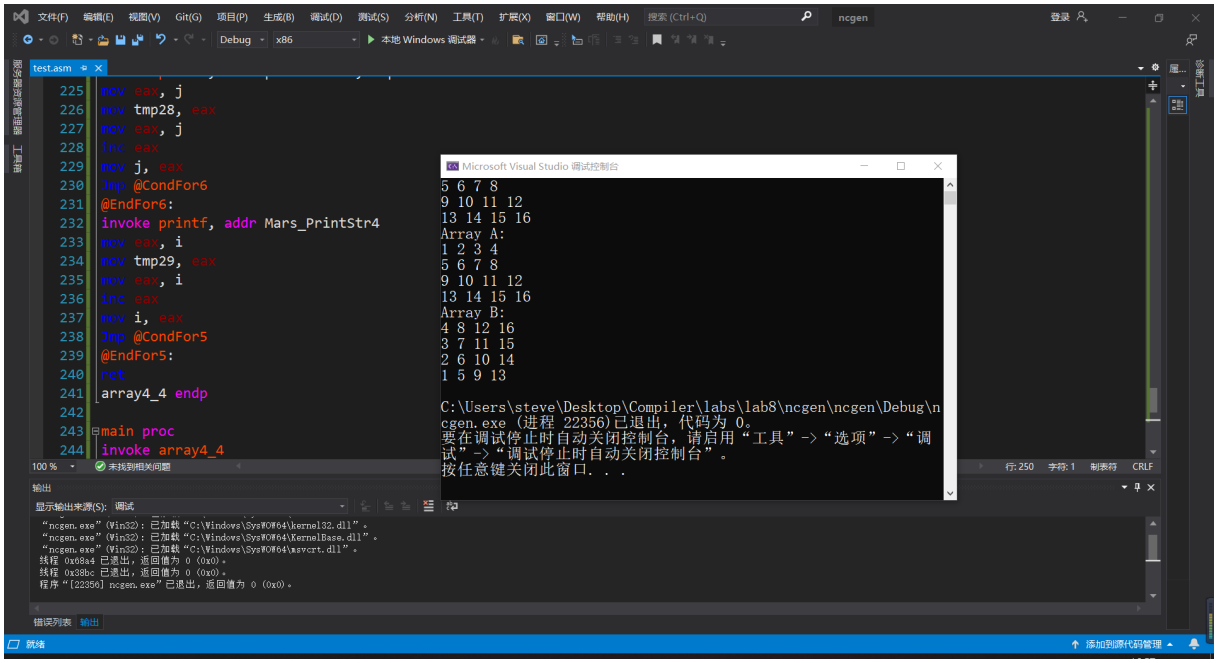

图 5: 4_CounterClockwiseRotationArray4_4.c 运行截图

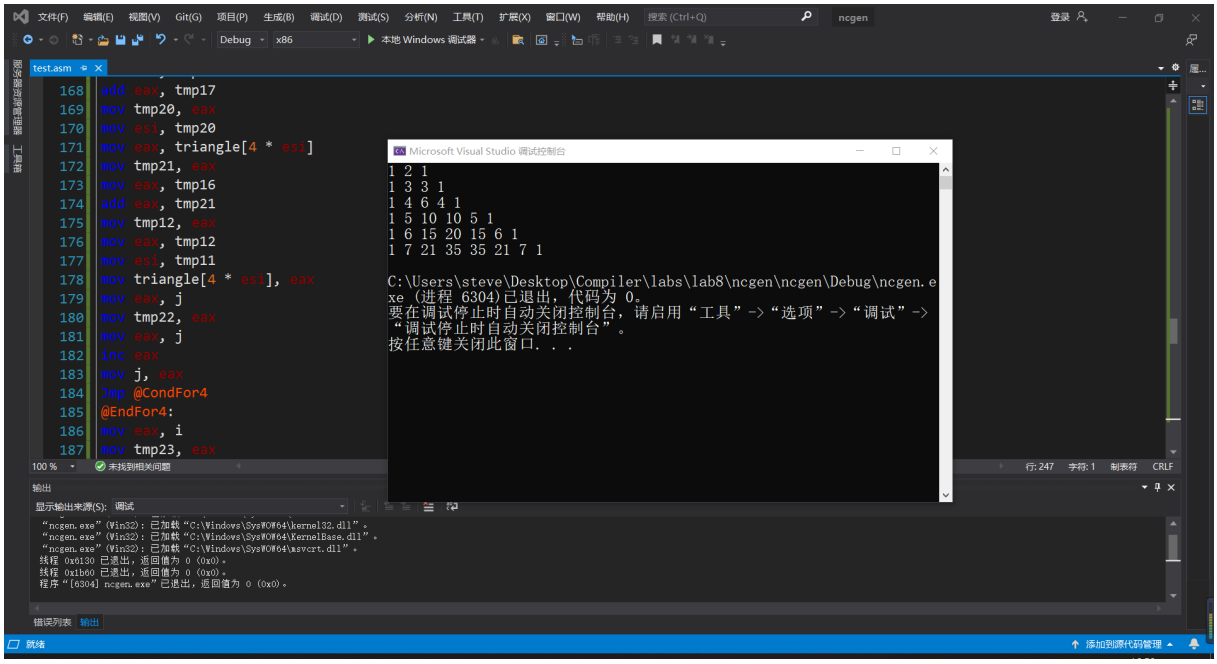

图 6: 5_YangHuiTriangle.c 运行截图

4 实验心得与体会

本次实验我通过修改 lab7 的代码并在此基础上生成了 masm32 汇编代码。在这个过程中，我对于代码生成有了一个初步的认识和实践。做实验的过程中我也遇到了一些问题，比如调用函数时参数可能是常数，需要新建一个 local 变量，将它的值赋过去；一些变量名如 c 在 masm32 中是语法错误，需要进行修改，统一修改成 inv< 原名字 > 等等。在这个过程中我的代码编写能力也得到了很大的长进。

至此，本学期《编译原理与设计》课程的所有实验都已结束，在一学期的学习中，宏观上说我对编译器的前、后端架构的功能分配等有了更深的认识，微观上说我对于词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成都有了一个低层逻辑原理的了解和实践。最终能实现一个简单的 c 语言编译器也令人十分兴奋和有成就感

最后也感谢老师们一学期以来的辛苦教学和指导!

参考文献

[1] *Lab 7 中间代码生成说明及要求.pdf*. zh. 2023.

A genCode 部分代码

```
// 生成当前 Func 其他语句的汇编代码
quat1 = quatStr.get(scanIdx);
while(scanIdx < quatStr.size()) {
    String op = quat1.getOp(); String res = quat1.getRes();
    String opnd1 = quat1.getOpnd1(); String opnd2 = quat1.getOpnd2();

    if(op.equals("Endp")) {// 函数结束
        break;
    }

    if(op.equals("Label")) {
        asmCode.append(res).append(": \n");
    }
    else if(op.equals("Jmp")) {
        asmCode.append("Jmp ").append(res).append("\n");
    }
    else if(op.equals("Jnt")) {
        asmCode.append(typeJnt).append(" ").append(res).append("\n");
    }
    // 对于下列这些运算符，把 opnd1 移到 eax，然后比较
    else if(op.equals(">") || op.equals("<")
            || op.equals(">=") || op.equals("<=")
            || op.equals("==") || op.equals("!=")) {
        asmCode.append("mov eax, ").append(opnd1).append("\n");
        asmCode.append("cmp eax, ").append(opnd2).append("\n");

        if(op.equals(">")) typeJnt = "jle";
        else if(op.equals("<")) typeJnt = "jge";
        else if(op.equals(">=")) typeJnt = "jl";
        else if(op.equals("<=")) typeJnt = "jg";
        else if(op.equals("==")) typeJnt = "jne";
        else if(op.equals("!=")) typeJnt = "je";
    }
    else if(op.equals("ret")) {
```

```
        if(!res.equals("")) {// 如果没有返回值则不用生成代码
            asmCode.append("mov eax, ").append(res).append("\n");
        }
    }
    else if(op.equals("+") || op.equals("-")
            || op.equals("*") || op.equals("+=")
            || op.equals("-=") || op.equals("*=")) {
        asmCode.append("mov eax, ").append(opnd1).append("\n");

        if(op.equals("+") || op.equals("+=")) asmCode.append("add eax, ")
        else if(op.equals("-") || op.equals("-=")) asmCode.append("sub ea
        else if(op.equals("*") ||op.equals("*=")) asmCode.append("imul ea

        asmCode.append("mov ").append(res).append(", eax").append("\n");
    }
    else if(op.equals("/") || op.equals("%")
            || op.equals("/=") || op.equals("%=")) {
        asmCode.append("mov edx, 0\n");// 清零
        asmCode.append("mov eax, ").append(opnd1).append("\n");
        asmCode.append("mov ebx, ").append(opnd2).append("\n");
        asmCode.append("div ebx\n");

        if(op.equals("/") || op.equals("/=")) {
            asmCode.append("mov ").append(res).append(", eax\n");
        }
        else {
            asmCode.append("mov ").append(res).append(", edx\n");
        }
    }
    else if(op.equals("++") || op.equals("--")) {
        asmCode.append("mov eax, ").append(res).append("\n");
        if(op.equals("++")) asmCode.append("inc eax\n");
        else if(op.equals("--")) asmCode.append("dec eax\n");
        asmCode.append("mov ").append(res).append(", eax\n");
    }
    else if(op.equals("=[]")) {
```

```
        asmCode.append("mov esi, ").append(opnd1).append("\n");
        asmCode.append("mov eax, ").append(opnd2)
            .append("[4 * esi]").append("\n");
        asmCode.append("mov ").append(res).append(", eax\n");
    }
    else if(op.equals("[]=")) {
        asmCode.append("mov eax, ").append(opnd1).append("\n");
        asmCode.append("mov esi, ").append(opnd2).append("\n");
        asmCode.append("mov ").append(res).append("[4 * esi], eax\n");
    }
    else if(op.equals("=")) {
        MyQuat lastquat = quatStr.get(scanIdx - 1);
        String Lop = lastquat.getOp(), Lopnd1 = lastquat.getOpnd1();
        if(Lop.equals("Call") && Lopnd1.equals("Mars_GetInt")) { // 需要先
            asmCode.append("mov eax, ").append(opnd1).append("\n");
            asmCode.append("mov ").append(res).append(", eax\n");
        }
        else {
            asmCode.append("mov eax, ").append(opnd1).append("\n");
            asmCode.append("mov ").append(res).append(", eax\n");
        }
    }
    else if(op.equals("Arg") && !opnd1.equals("Mars_PrintStr")) {// 非 Pr
        if(!Character.isDigit(res.charAt(0))) {// 说明参数是一个标识符
            argLists.add(res);
        }
        else { // 说明参数是数字，用之前需要先把值装进去
            asmCode.append("mov ").append(invertIdf.get(res)).append(", "
            argLists.add(invertIdf.get(res));
        }
    }
    else if(op.equals("Call")) {// 先特判输入输出函数，再处理一般函数
        if(opnd1.equals("Mars_GetInt")) {
            asmCode.append("invoke scanf, addr scanFmt, addr ").append(re
        }
        else if(opnd1.equals("Mars_PrintInt")) {
```

```
                String number = argLists.pop();
                asmCode.append("invoke printf, addr printIntFmt, ").append(nu
            }
            else if(opnd1.equals("Mars_PrintStr")) {
                asmCode.append("invoke printf, addr Mars_PrintStr" + nowMars_
                nowMars_cnt ++;
            }
            else {
                asmCode.append("invoke ").append(opnd1);
                while (!argLists.empty()) {
                    String argu = argLists.pop();
                    asmCode.append(", ").append(argu);
                }
                asmCode.append("\n");

                if (!res.equals("")) {// 将函数得到的结果赋值过来
                    asmCode.append("mov ").append(res).append(", eax\n");
                }
            }
        }
        // 取下一个四元式
        scanIdx ++;
        if(scanIdx >= quatStr.size()) break;
        quat1 = quatStr.get(scanIdx);
    }
```