编译原理与设计实验报告

姓名：卜梦煜　　学号：1120192419　　班级：07111905

1. 实验名称

目标代码生成实验

2. 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。

3. 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。

4. 实验环境

Intellij IDEA Community 2021.3.2、masm32

5. 实验过程与步骤

本实验的目标是基于中间代码生成模块生成的四元式，编写编译器目标代码生成模块，生成的目标代码格式选择 x86 汇编语言。

本实验主要步骤如下：首先编写 x86 代码框架，生成引用头文件部分、数据部分、代码部分；之后对中间代码四元式按类进行翻译，编写对应的目标代码；最后修改配置文件，将测试用例生成的 x86 汇编代码在 masm32 汇编器上编译成.exe 文件并运行。

5.1 编写 x86 代码框架

x86 汇编语言代码框架主要由引用头文件部分、数据部分、代码部分组成，编写的四元式大多对应代码部分，因此在进行四元式翻译前，需要先生成 x86 代码框架。

引用头文件部分包括.386 框架伪指令、头文件引用、C 函数调用。框架伪指令、头文件引用对一般的汇编代码是固定的，直接加入生成的汇编代码即可。本次实验提供的测试用例中输入输出函数较为特殊，包括输入函数 Mars_PrintStr、Mars_PrintInt 和输出函数 Mars_GetInt，分别对应于 C 语言的 printf、scanf 函数，声明需要写在 C 函数调用部分。引用部分代码如图。

```
// include
public void generateX86Include() {
    x86Code.append("""
            .386
            .model flat, stdcall
            option casemap:none
            include windows.inc
            include user32.inc
            includelib user32.lib
            include kernel32.inc
            includelib kernel32.lib
            include msvcrt.inc
            includelib msvcrt.lib
            includelib ucrt.lib
            scanf proto c :dword, :vararg
            printf proto c :dword, :vararg

            """);
}
```

数据部分包括全局变量，需要定义为全局变量的包括 scanf 数字输入格式和 printf 数字输出格式、以及 Mars_PrintStr 函数中字符串。Mars_PrintStr 函数中字符串需要定义为全局变量的原因是，如果不定义为全局变量则需要声明全局的临时字符串空间，遇到字符串常量时需要进行字符串拷贝，导致代码较复杂，因此直接将字符串声明成全局变量会更方便。

```
public void generateX86Data() {
    x86Code.append("""
            .data
            digitPrintFmt db "%d ", 0
            digitGetFmt db "%d", 0
            """);

    // 遍历整个func，将所有Mars_PrintInt, Mars_PrintStr作为全局变量定义
    int tempIdx = 0;
    int marsCnt = 0;
    QuatStr quat1 = quats.get(tempIdx);
    while (tempIdx < quats.size()) {
        String op = quat1.getOp();
        String res = quat1.getRes();
        String opnd1 = quat1.getOpnd1();
        String opnd2 = quat1.getOpnd2();
        if (Objects.equals(res, b: "")) {
            quat1 = quats.get(++tempIdx);
            continue;
        }
        if (op.equals("arg") && opnd1.equals("Mars_PrintStr")) {
            if (res.equals("\"\\n\"")) {
                x86Code.append("Mars_PrintStr" + marsCnt).append(" db ").append("0ah, 0\n");
            } else if (res.contains("\\n")) {
                res = res.replace( target: "\\n",  replacement: "");
                x86Code.append("Mars_PrintStr" + marsCnt).append(" db ").append(res).append(", 0ah, 0\n");
            } else {
                x86Code.append("Mars_PrintStr" + marsCnt).append(" db ").append(res).append(", 0\n");
            }
            marsCnt++;
        }
        tempIdx++;
        if (tempIdx < quats.size()) {
            quat1 = quats.get(tempIdx);
        } else {
            break;
        }
    }
    x86Code.append("\n");
}
```

代码部分包括定义的各个函数。函数包括函数声明语句、参数声明语句、局部变量声明语句、函数体代码、函数定义结束语句，由四元式形式的中间代码得到。对于函数体中定义的临时变量，为避免寄存器分配带来的困难，本实验选择将寄存器分配工作交给汇编器，将所有临时变量声明成局部变量，这样保证了在访问四元式时，遇到的所有变量都已经预先声明为局部变量或函数参数。

5.2 编写四元式翻译为 x86 汇编代码的方法

四元式翻译为 x86 汇编代码是本次实验的核心，基本方法是根据四元式类型和内容，编写对应的汇编代码生成格式。四元式类型由四元式中 op 字段确定。本实验需要对四元式组按照函数进行翻译，每个函数做两次遍历，第一次遍历定义局部变量，第二次遍历翻译其他四元式的代码。

函数范围界定相关的四元式 op 包括 param、func、funcEnd。每个函数以 func 为开始，funcEnd 为结束，对应翻译为汇编代码“函数名 proc”、“函数名 endp”，main 函数结尾还需

要语句“end main”。函数形参由 param 指出，对应翻译为汇编代码“参数名 :参数类型。

局部变量定义需要紧跟在函数声明语句后，范围包括：以@开头、在当前函数内未定义的变量，四元式 op 为 var、arr 的变量，四元式 op 为 arg 且非 Mars_PrintStr 的变量。对应翻译为汇编代码“local 局部变量名 :变量类型”。

跳转相关的四元式 op 包括 label、jmp、jf、ret，用于控制程序的执行顺序。Label、jmp、jf 的标签名存放在四元式的 result 中。jf 四元式需要根据条件判断的类型，生成条件为假时的跳转指令。ret 四元式需要根据是否有返回值编写，有返回值时将函数返回值保存在 eax 寄存器中。生成汇编指令的代码如下。

```
// 跳转类
if (op.equals("label")) {
    x86Code.append(res).append(":\n");
} else if (op.equals("jmp")) {
    x86Code.append("jmp ").append(res).append("\n");
} else if (op.equals("jf")) {
    x86Code.append(jfType).append(" ").append(res).append("\n");
} else if (op.equals("ret")) {
    if (!res.equals("")) {
        x86Code.append("mov eax, ").append(res).append("\n");
    }
}
```

逻辑运算相关的四元式 op 包括>=、<=、>、<、==、!=。由于逻辑运算往往用于控制程序跳转，而程序跳转指令根据标志寄存器判断，因此逻辑运算需要使用 cmp 指令完成，对于每个 op，保存条件为假时对应的跳转语句供跳转语句使用。生成汇编指令代码如下。

```
// 逻辑运算类
else if (op.equals(">=") || op.equals("<=") || op.equals(">") ||
        op.equals("<") || op.equals("==") || op.equals("!=")) {
    x86Code.append("mov eax, ").append(opnd1).append("\n");
    x86Code.append("cmp eax, ").append(opnd2).append("\n");
    switch (op) {
        case ">=" -> jfType = "jl";
        case "<=" -> jfType = "jg";
        case ">" -> jfType = "jle";
        case "<" -> jfType = "jge";
        case "==" -> jfType = "jnz";
        case "!=" -> jfType = "jz";
    }
}
```

算术运算类相关的四元式 op 包括+、-、*、/、%、+=、-=、*=、/=、%=。对于加、减、

乘，生成汇编指令的结构相似，只是运算指令不同。对于除、取模指令，由于汇编指令特性，32 位寄存器的源操作数为“edx:eax”，商保存在 eax 中，余数保存在 edx 中，据此编写汇编指令即可。生成汇编指令如下。

```
// 算术运算类
else if (op.equals("+") || op.equals("-") || op.equals("*") ||
        op.equals("+=") || op.equals("-=") || op.equals("*=")) {
    x86Code.append("mov eax, ").append(opnd1).append("\n");
    switch (op) {
        case "+" -> x86Code.append("add eax, ").append(opnd2).append("\n");
        case "+=" -> x86Code.append("add eax, ").append(opnd2).append("\n");
        case "-" -> x86Code.append("sub eax, ").append(opnd2).append("\n");
        case "-=" -> x86Code.append("sub eax, ").append(opnd2).append("\n");
        case "*" -> x86Code.append("imul eax, ").append(opnd2).append("\n");
        case "*=" -> x86Code.append("imul eax, ").append(opnd2).append("\n");
    }
    x86Code.append("mov ").append(res).append(", eax\n");
} else if (op.equals("/") || op.equals("%") || op.equals("/=") || op.equals("%=")) {
    x86Code.append("xor edx, edx\n");
    x86Code.append("mov eax, ").append(opnd1).append("\n");
    x86Code.append("mov ebx, ").append(opnd2).append("\n");
    x86Code.append("div ebx\n");
    if (op.equals("/") || op.equals("/=")) {
        x86Code.append("mov ").append(res).append(", eax\n");
    } else {
        x86Code.append("mov ").append(res).append(", edx\n");
    }
}
```

单目运算类相关的四元式 op 包括++、--。对应生成自增、自减指令即可。生成汇编指令如下。

```
// 单目运算类
else if (op.equals("++") || op.equals("--")) {
    x86Code.append("mov eax, ").append(res).append("\n");
    if (op.equals("++")) {
        x86Code.append("inc eax\n");
    } else {
        x86Code.append("dec eax\n");
    }
    x86Code.append("mov ").append(res).append(", eax\n");
}
```

赋值运算类相关的四元式 op 包括=、[]=、=[]。普通赋值=只需将源操作数移入寄存器、寄存器移入目的操作数即可。[]=含义是取数组值，换算后的地址保存在 opnd1 中，取出的值保存在 opnd2 中。=[]含义是数组赋值，赋值保存在 opnd1 中，换算的地址保存在 opnd2 中。生成汇编指令如下。

```
// 赋值运算类
else if (op.equals("=")) {
    x86Code.append("mov eax, ").append(opnd1).append("\n");
    x86Code.append("mov ").append(res).append(", eax\n");
} else if (op.equals("=[]")) {
    x86Code.append("mov esi, ").append(opnd1).append("\n");
    x86Code.append("mov eax, ").append(opnd2).append("[esi*4]\n");
    x86Code.append("mov ").append(res).append(", eax\n");
} else if (op.equals("[]=")) {
    x86Code.append("mov eax, ").append(opnd1).append("\n");
    x86Code.append("mov esi, ").append(opnd2).append("\n");
    x86Code.append("mov ").append(res).append("[4*esi], eax\n");
}
```

函数调用类相关的四元式 op 包括 arg、call。对函数参数 arg，按从右向左顺序压入一个 argStack 栈中。对函数调用语句 call 首先特判生成 Mars_PrintStr、Mars_PrintInt、Mars_GetInt 对应的汇编指令。对其他函数调用，用 invoke 伪指令生成，存在参数时 argStack 逆序出栈至栈空，依次拼接到 invoke 伪指令后即可。对有返回值的函数，还需要将返回值从 eax 中取出，保存在 result 中。编写代码如下。

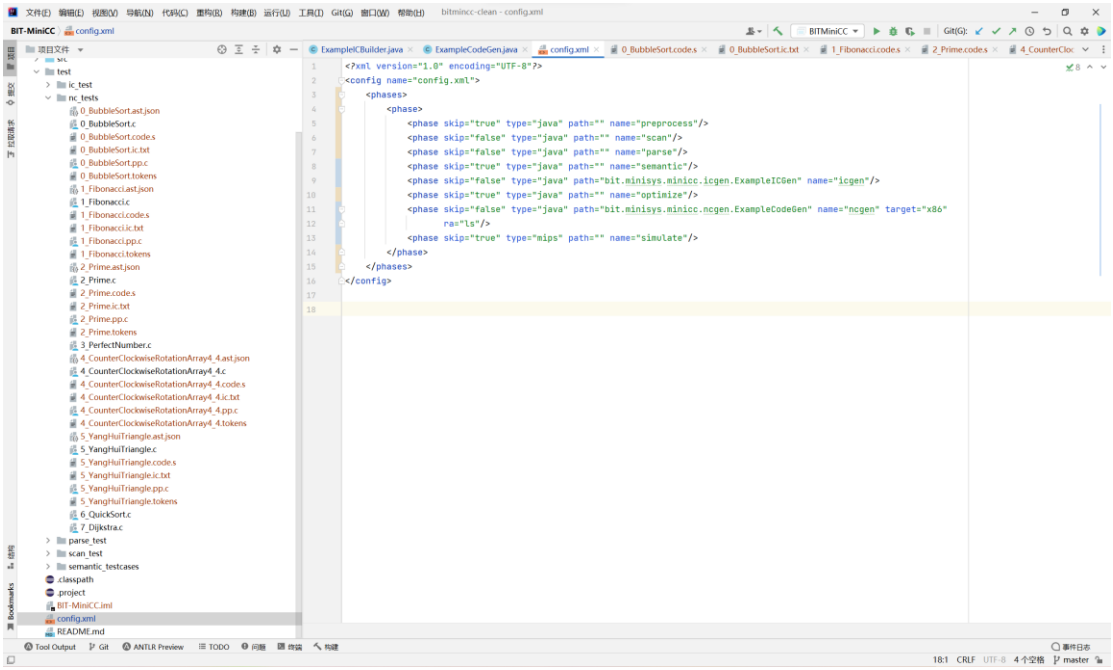
```
// 函数调用类
else if (op.equals("arg") && !opnd1.equals("Mars_PrintStr")) {
    argStack.push(res);
} else if (op.equals("call")) {
    if (opnd1.equals("Mars_PrintStr")) {
        x86Code.append("invoke printf, addr Mars_PrintStr" + marsStrNum).append("\n");
        marsStrNum++;
    } else if (opnd1.equals("Mars_PrintInt")) {
        String src = argStack.pop();
        x86Code.append("invoke printf, addr digitPrintFmt, ").append(src).append("\n");
    } else if (opnd1.equals("Mars_GetInt")) {
        x86Code.append("invoke scanf, addr digitGetFmt, addr ").append(res).append("\n");
    } else {
        x86Code.append("invoke ").append(opnd1);
        while (!argStack.empty()) {
            x86Code.append(", ").append(argStack.pop());
        }
        x86Code.append("\n");
    }

    if (!res.equals("") && !opnd1.equals("Mars_GetInt")) {
        x86Code.append("mov ").append(res).append(", eax\n");
    }
}
```

5.3 修改配置文件，生成汇编代码

修改配置文件中 ncgen 的 skip=false、path 为编写的目标代码生成文件、target=x86，

配置文件如下图。

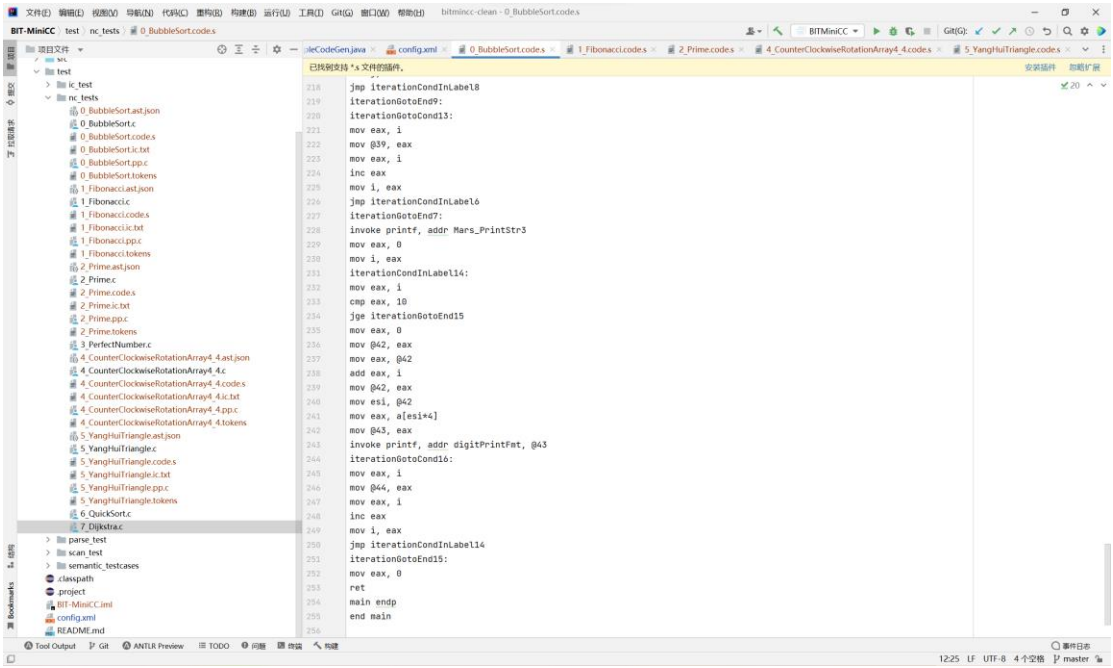


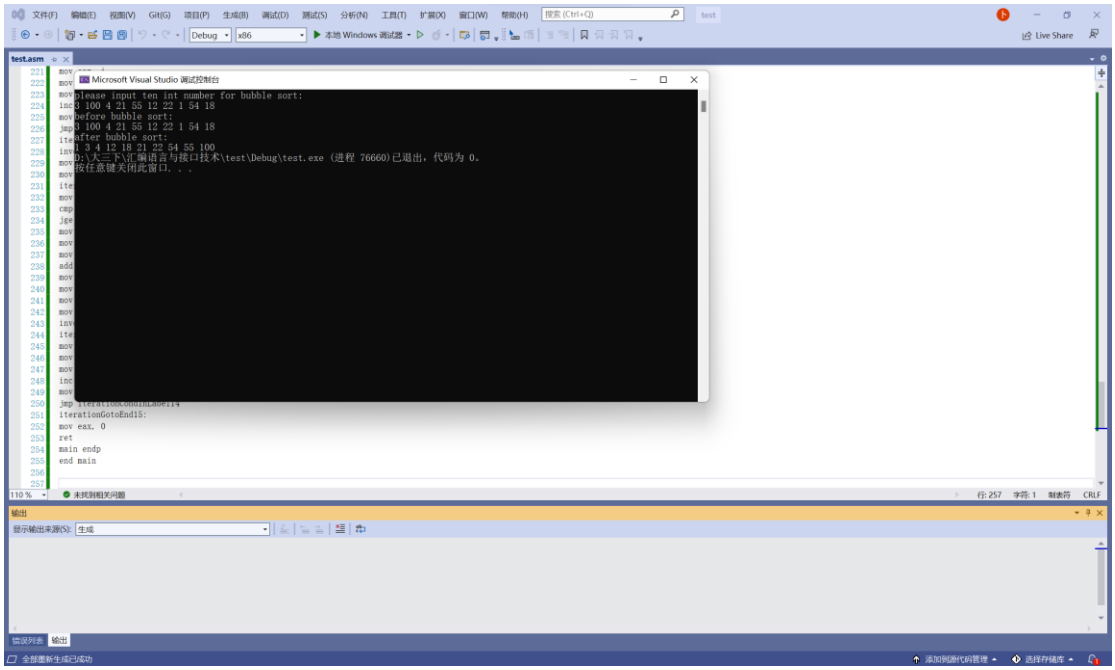
对需要测试的.c 文件，将文件路径写入编译配置中即可。

6. 实验结果与分析

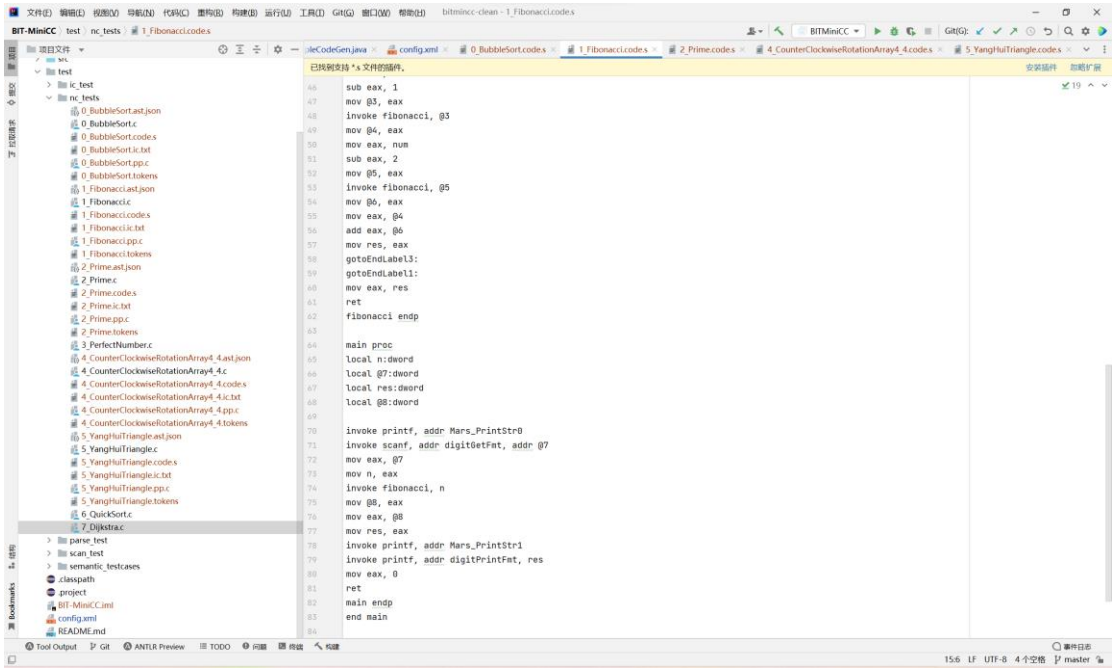
本实验已通过的测试用例有 5 个，分别是 0_BubbleSort.c、1_Fibonacci.c、2_Prime.c、4_CounterClockwiseRotationArray4_4.c 、5_YangHuiTriangle.c 。对未通过的测试用例，3_PerfectNumber.c 因为存在局部变量时汇编关键字的问题，需要额外编写函数名转换的方法。

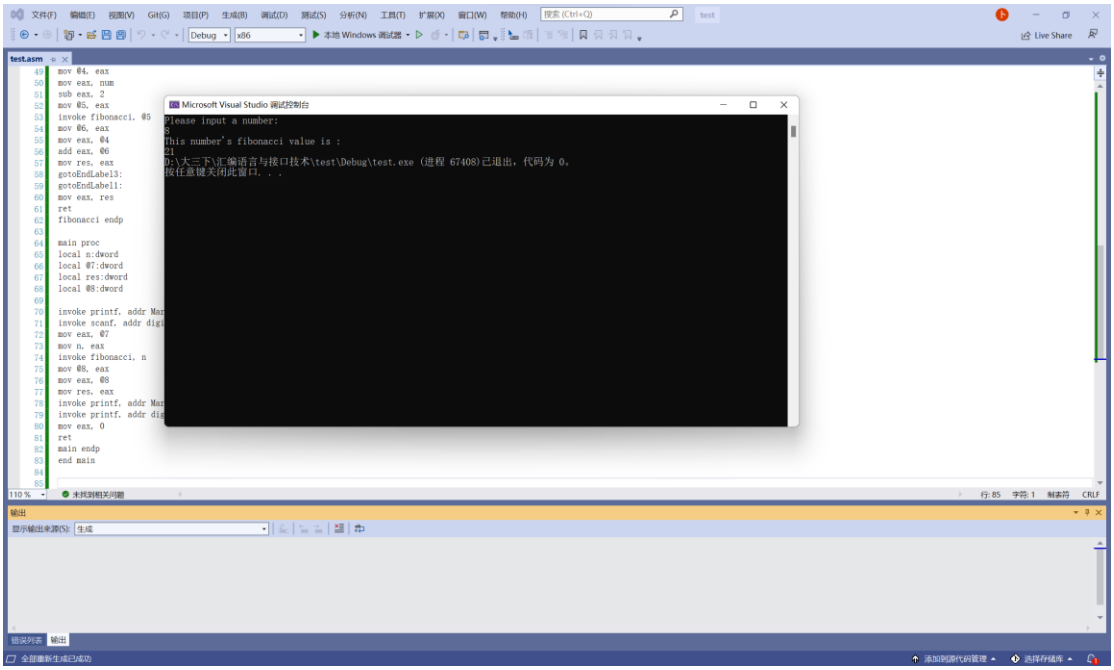
0_BubbleSort.c 生成的汇编代码部分、在 masm32 上的运行结果如下。



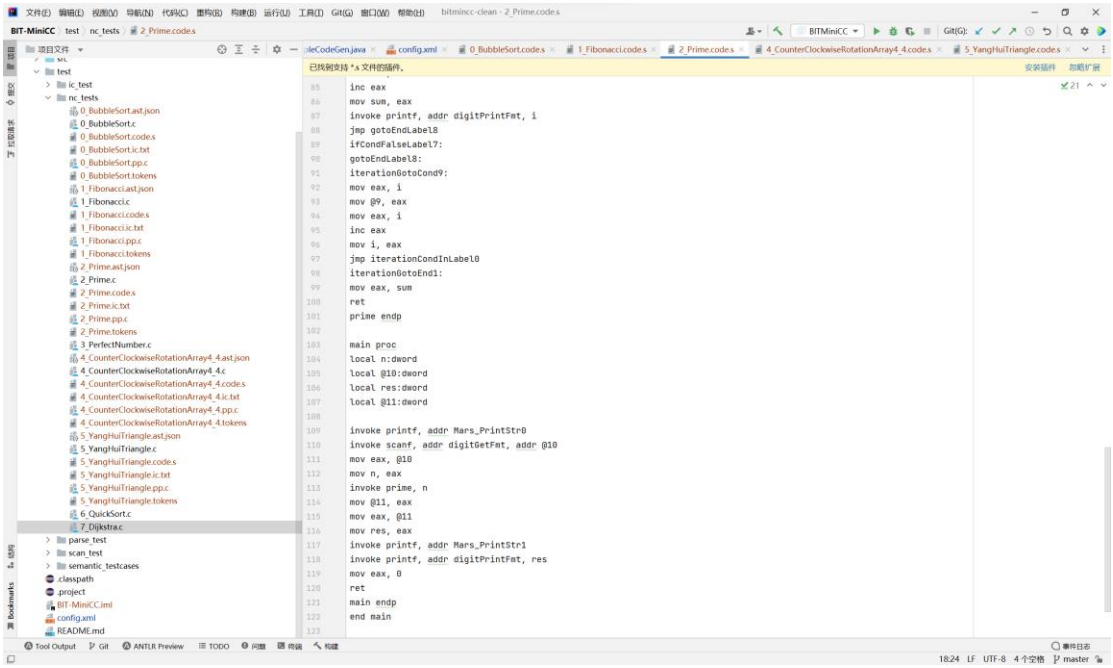


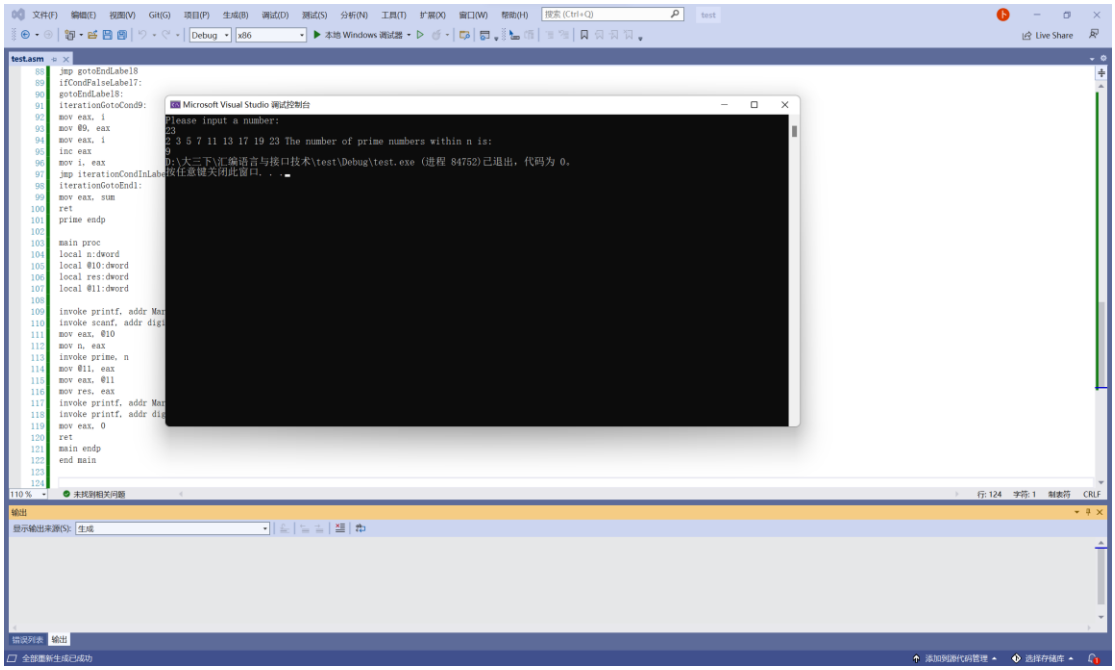
1_Fibonacci.c 生成的汇编代码部分、在 masm32 上的运行结果如下。



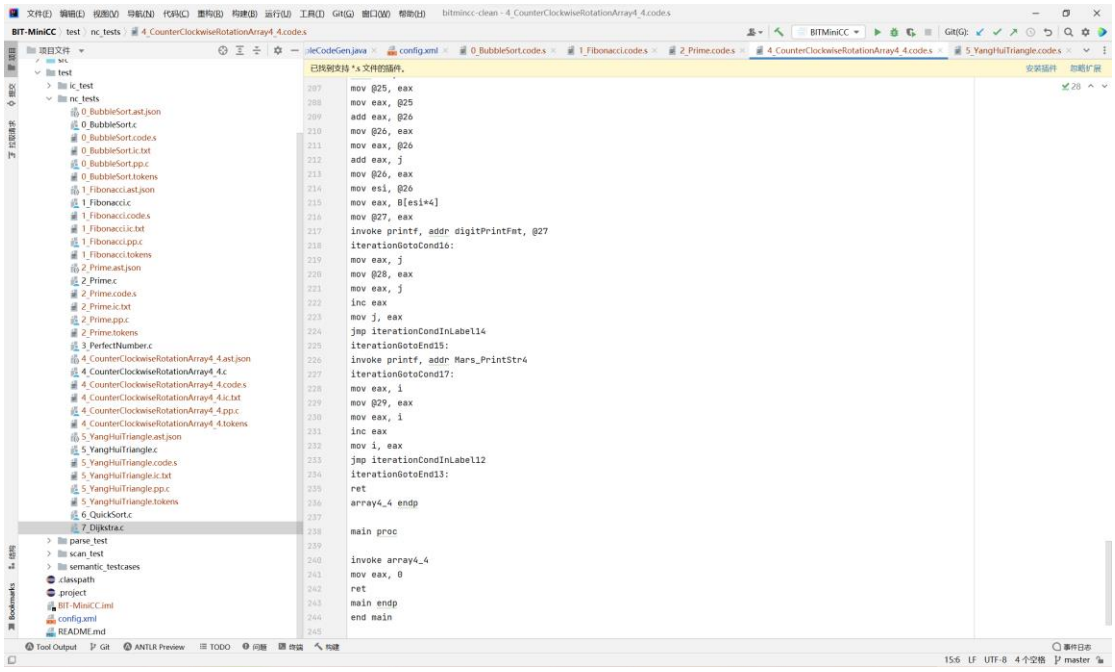


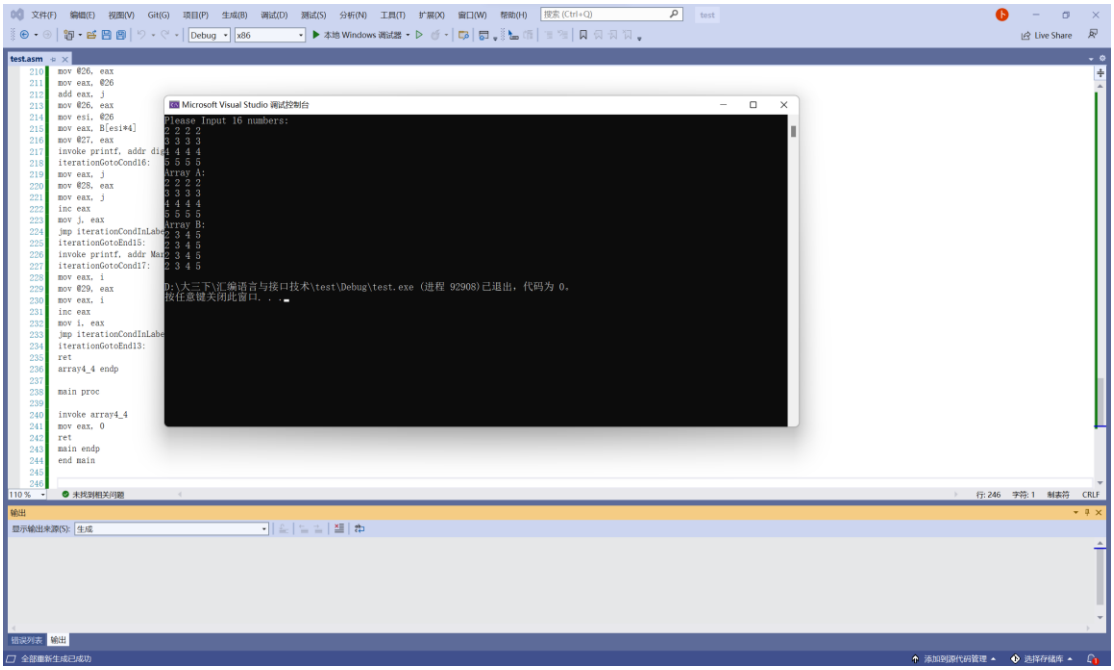
2_Prime.c 生成的汇编代码部分、在 masm32 上的运行结果如下。



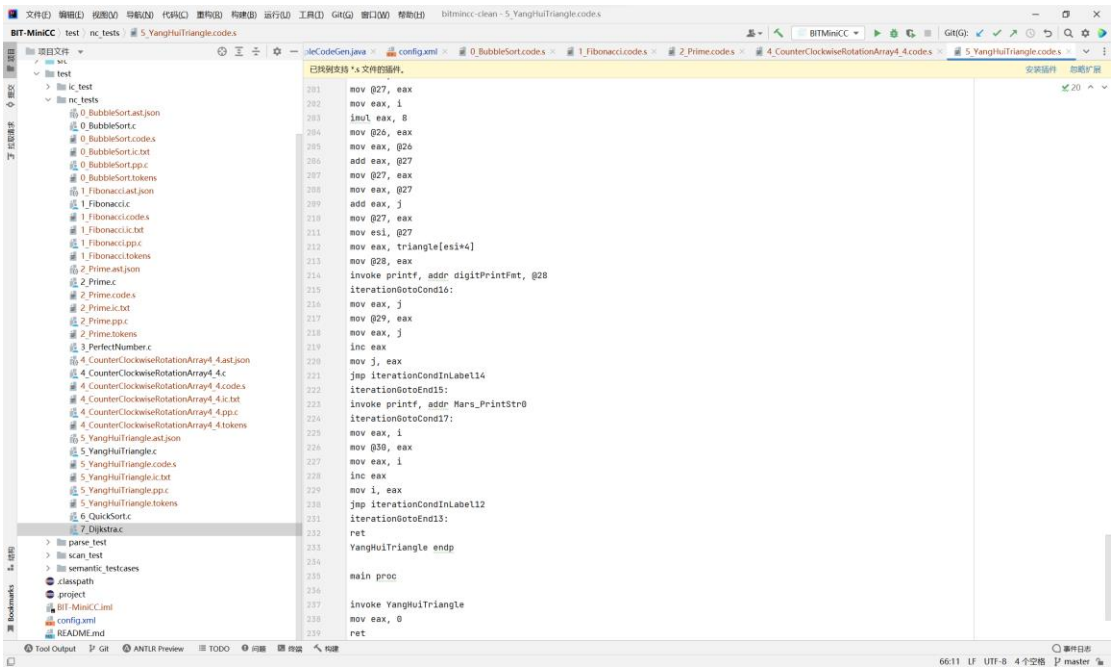


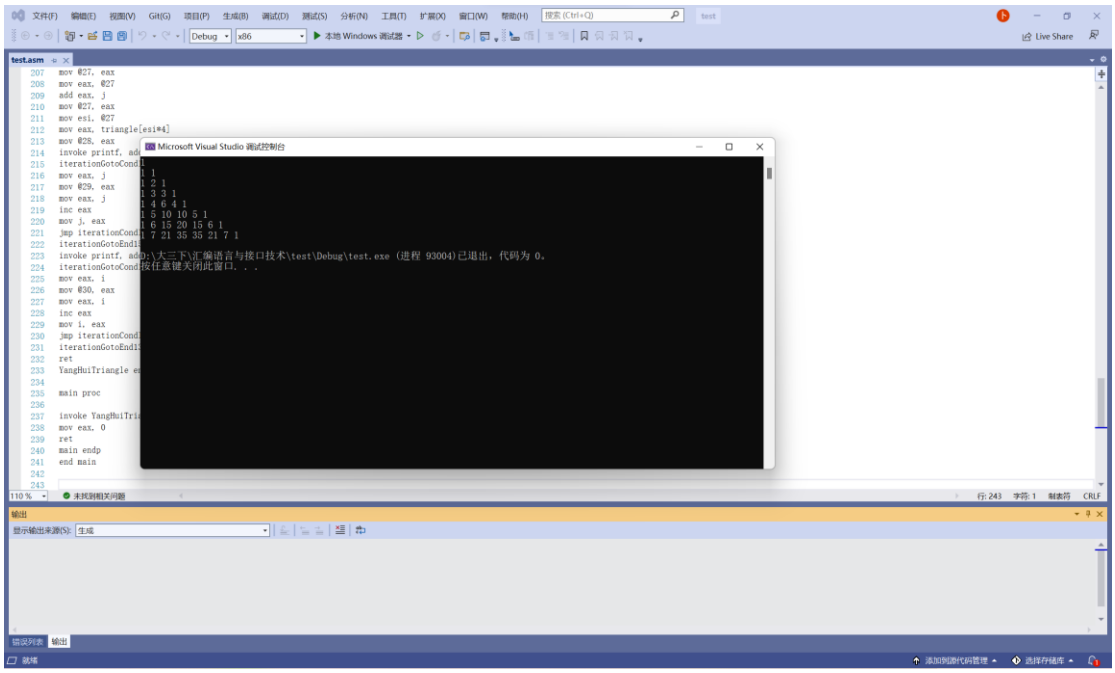
4_CounterClockwiseRotationArray4_4.c 生成的汇编代码部分、在 masm32 上的运行结果如下。





5_YangHuiTriangle.c 生成的汇编代码部分、在 masm32 上的运行结果如下。





通过以上图片可知，生成的汇编代码能够运行并得到正确结果。

7. 个人心得体会

本次实验是编译原理的最后一个实验，相比前几个实验，难度上基本持平。本实验难点在于编写四元式转换成汇编语言的代码，这需要结合汇编语言所学的知识。同时，中间代码的质量也决定了本次实验的任务量，在编写目标代码生成时，我也多次反工修改中间代码生成的部分内容，以使的目标代码生成更合理、简单。本次实验我的收获主要有以下几点。

（1）掌握了目标代码生成的方法，熟悉了多种寄存器分配算法，能够编写目标代码生成器，并将之嵌入到 BITMiniCC 中，在四元式的基础上生成能正确执行的 x86 汇编代码。

（2）复习了汇编语言相关知识，能够熟练掌握四元式与汇编语言之间的转换。