成绩

**编译原理实验报告**

**第一章编译过程理解一**

学院： 计算机学院

班级： 10012003

姓名： 徐金海

学号： 2020302703

日期： 2023.05.01

目录

**[1 实验情况概述 1](#_Toc1774)**

[1.1实验目的 1](#_Toc25319)

[1.2实验任务 1](#_Toc4175)

**[2 实验内容（实验教材中的示例） 1](#_Toc27157)**

[2.1实验原理 1](#_Toc1273)

[2.2 GCC生成可执行程序（单一命令） 2](#_Toc4376)

[2.3 GCC生成可执行程序（多个命令） 5](#_Toc5640)

[2.4 gcc -c命令的细化 6](#_Toc3141)

**[3 编译过程中中间语言IR、汇编语言与源程序对应分析 11](#_Toc22419)**

[3.1分析与处理功能 13](#_Toc29699)

[3.2分析词法功能 14](#_Toc19030)

[3.3分析语法产生出的抽象语法树 14](#_Toc4198)

[3.4中间代码生成 16](#_Toc16203)

[3.5汇编语言生成 17](#_Toc23247)

[3.6可执行程序生成 1](#_Toc19800)8

[3.7反汇编test程序 1](#_Toc11630)8

**[4 函数调用与返回 1](#_Toc4427)9**

**[5 课后作业1 2](#_Toc6093)1**

[5.1作业要求 21](#_Toc29353)

[5.2程序编写 21](#_Toc18649)

[5.3预处理 22](#_Toc15404)

[5.4编译 23](#_Toc16894)

[5.5汇编 24](#_Toc3288)

[5.6链接 25](#_Toc25678)

**[6 课后作业2 26](#_Toc3855)**

[6.1作业要求 26](#_Toc12245)

[6.2 程序编写 26](#_Toc3352)

[6.3汇编语言的生成及分析 27](#_Toc3711)

**[7 课后作业3 2](#_Toc30441)9**

[7.1作业要求 2](#_Toc3901)9

[7.2 程序编写 2](#_Toc19256)9

[7.3 分析（带优化-O2） 30](#_Toc23377)

[7.4 分析（不带优化） 31](#_Toc9758)

**[8 课后作业4 32](#_Toc16094)**

[8.1作业要求 32](#_Toc2517)

[8.2 程序编写 32](#_Toc10045)

[8.3 分析 32](#_Toc23122)

**[9 课后作业5 33](#_Toc19612)**

[9.1作业要求 33](#_Toc2766)

[9.2 程序编写 33](#_Toc10741)

[9.3 验证结果（make命令） 33](#_Toc13435)

[9.4 验证结果（cmake命令） 37](#_Toc15713)

**[10 实验总结 3](#_Toc18424)9**

1 实验情况概述

1.1实验目的

理解 clang 或 GCC 的编译过程，加深对编译器的前端和后端的理解。

1.2实验任务

利用 clang 或者 GCC 逐步编译，分析得出结果的正确性。 重在理解分析，熟悉编译过程后需完成课后题。

2 实验内容（实验教材中的示例）

2.1实验原理

GCC构建（Build）可执行程序的过程通常可以理解为两步，首先是编译（Compile）过程，此步骤可以将源代码转化成目标文件，然后是链接（Link）过程，目的是将多个目标文件转成一个可执行程序。

现假定一个项目（Project）包含有三个文件，头文件 add.h，源文件 add.c 和 main.c，实现一个对两个数进行加法运算。

add.h:

#ifndef  \_\_ADD\_H\_\_

#define  \_\_ADD\_H\_\_

int add(int a, int b);

#define ADD(a,b) ((a)+(b))

#endif

add.c:

#include "add.h"

int add(int a,int b)

{

    return a + b;

}

main.c:

#include "add.h"

int g1 = 10;

int g2;

int main()

{

    int a = 10;

    int c;

    c = add(a, g1);

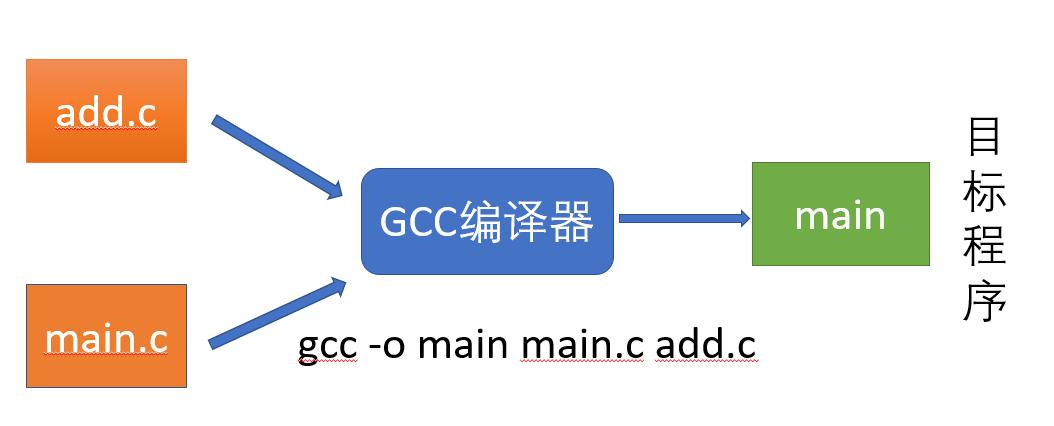
    c = ADD(c, g2);

    return c;

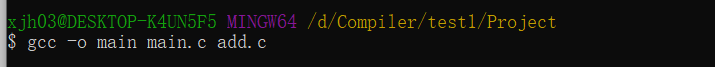
}

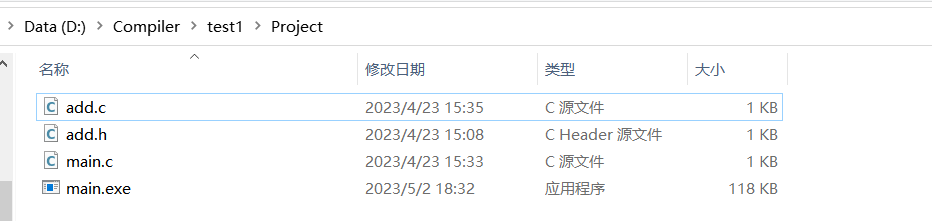
2.2 GCC生成可执行程序（单一命令）

命令：gcc -o main main.c add.c，其示意图如下：



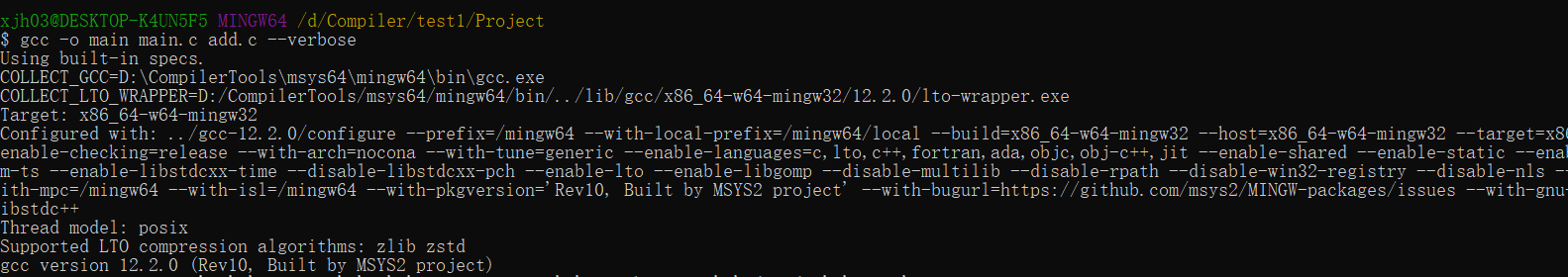
在此过程中，该命令其实包含了两步操作，即编译和链接。



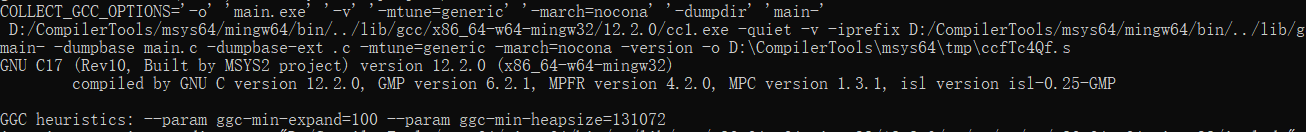


可以看出，在Project目录下生成了main.exe可执行程序文件。

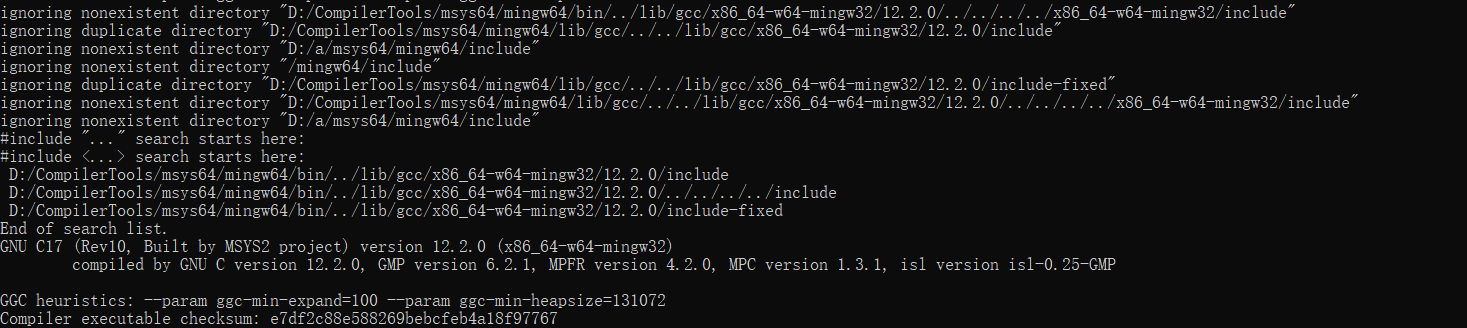
上述过程我们无法显式地查看编译的过程，但可以使用命令gcc -o main main.c add.c --verbose查看编译的过程。具体我们可以查看下面的内容：



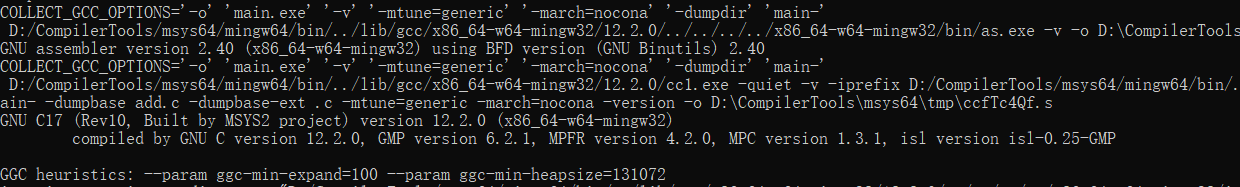
首先可以看到，以上信息包含了关于GCC编译器的版本、配置选项、目标平台以及线程模型等信息。具体来说，它说明了：（1）GCC编译器的版本号为12.2.0，是由MSYS2项目构建的。（2）GCC编译器的目标平台为x86\_64-w64-mingw32，这是一个64位Windows平台，使用了MinGW-w64工具链。（3）GCC编译器的配置选项包括了安装路径、目标架构、构建系统以及支持的语言和工具等设置。GCC编译器使用了POSIX线程模型来支持多线程，包含了Graphite框架来进行循环优化，同时还使用了各种库和包，比如GMP、MPFR和ISL等，用于数值计算和代码生成。



接下来这段信息提供了关于GCC编译器的编译选项和具体编译过程的信息。具体来说，它包含以下信息：（1）编译选项包括了输出文件名为"main.exe"，以及使用"-v"选项输出详细信息，同时使用"-mtune=generic"和"-march=nocona"选项设置编译器优化选项，最后使用"-dumpdir"选项指定编译结果的输出目录为"main-"。（2）GCC编译器使用cc1.exe作为前端编译器，对main.c进行编译。其中，"-iprefix"选项指定了头文件的查找路径。"-dumpbase"选项指定了源文件的基本名称，"-dumpbase-ext"选项指定了源文件的扩展名。 "-quiet"选项指定了编译器的输出信息的详细程度。"-version"选项指定了编译器的版本信息。



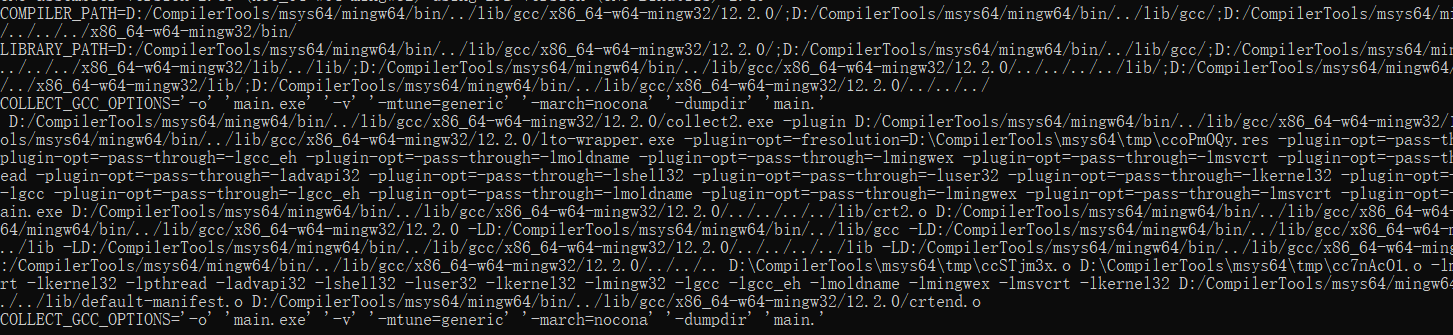
然后这段内容指出了这段信息提供了GCC编译器在查找头文件时遇到的路径以及最终查找的结果。具体来说，编译器忽略了一些不存在的路径，同时列出了实际存在的路径，并指出了头文件搜索的开始位置和结束位置。



接下来的这段信息也说明了GCC编译器的汇编过程和具体编译过程的信息。这里GCC编译器使用as.exe作为汇编器，对前面编译生成的汇编代码文件"ccfTc4Qf.s"进行汇编，生成目标文件"ccSTjm3x.o"。

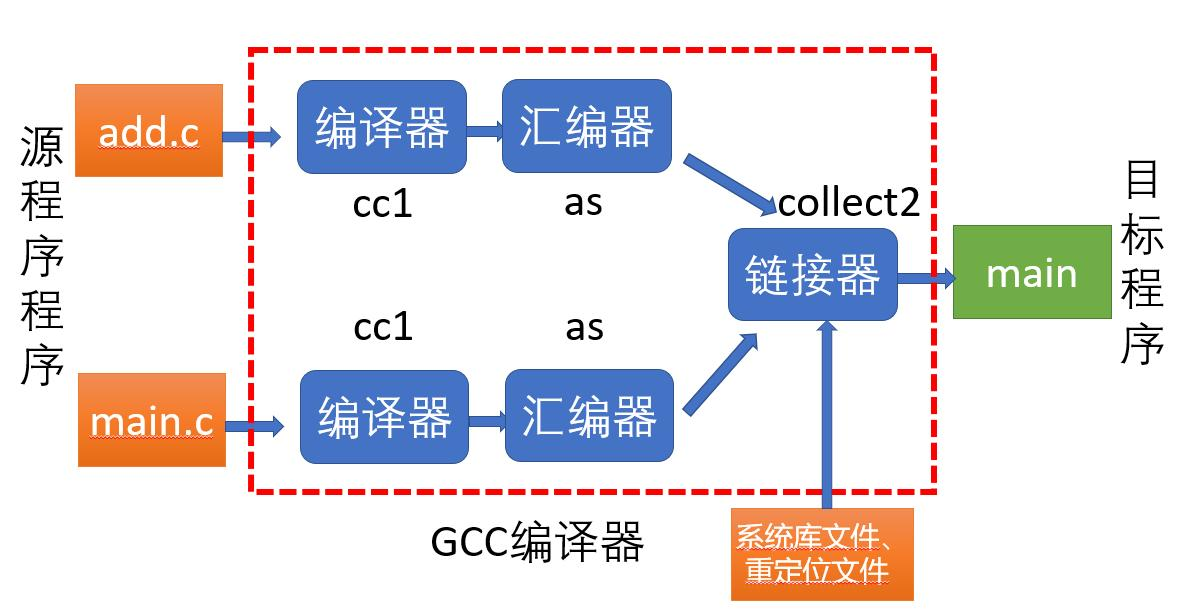
至此，编译器完成了main.c程序的初步编译，然后是add.c程序的初步编译过程，与上述类似，此处便不再过多赘述。

接下来我们可以看一下链接操作部分的具体输出信息：



上面这段信息说明了这段信息提供了GCC编译器的链接过程以及具体的链接选项。具体来说，它包含以下信息：（1）COMPILER\_PATH和LIBRARY\_PATH变量指定了编译器和库的查找路径。（2）COLLECT\_GCC\_OPTIONS包含了链接选项，包括输出文件名为"main.exe"，使用"-v"选项输出详细信息，使用"-mtune=generic"和"-march=nocona"选项设置编译器优化选项，最后使用"-dumpdir"选项指定编译结果的输出目录为"main."。（3）collect2.exe是GNU链接器的前端，它使用liblto\_plugin.dll插件进行链接。"-fresolution"选项指定了在链接器解析符号时使用的文件名。（4）"-pass-through"选项将一些库的名称传递给链接器，这些库包括了mingw32、gcc、gcc\_eh、lmoldname、mingwex、msvcrt、kernel32、pthread、advapi32、shell32和user32。（5）"-m i386pep"选项指定了链接生成的二进制文件的目标架构，这里指定为i386pep。（6）"-Bdynamic"选项指定了生成动态链接库。（7）"-o"选项指定了输出文件名为"main.exe"。（8）接下来的一系列参数指定了要链接的目标文件和库的路径。（9）最后，链接器生成了一个可执行文件"main.exe"。

至此，main.exe生成完毕，其过程如实验教程中的示意图所示：



从上图及终端输出信息可见，main.exe生成的过程需要经过main.c和add.c分别编译和汇编，分别得到两个.o文件，在经过链接器的链接操作得到最终的目标程序文件main.exe。

2.3 GCC生成可执行程序（多个命令）

上述报告中介绍的都是用一条命令生成可执行程序main.exe文件，事实上也可以用以下三行命令完成整个过程：

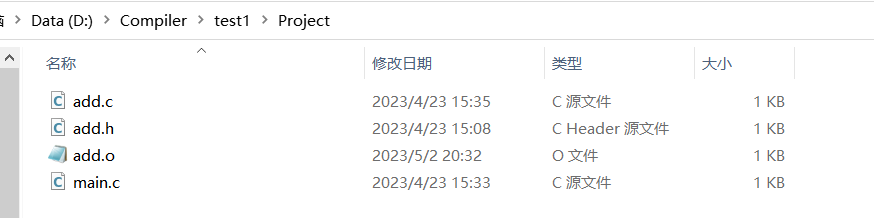
gcc -c -o add.o add.c

gcc -c -o main.o main.c

gcc -o main add.o main.o

gcc -c命令可以奖源代码文件转化成目标文件。如下图所示：





执行gcc -c -o add.o add.c命令后会在对应目录下生成add.c源代码文件的目标文件add.o。同理，执行gcc -c -o main.o main.c命令后会在Project目录下生成main.c的目标文件main.o文件。





随后通过执行gcc -o main add.o main.o将两个目标文件add.o和main.o链接成一个可执行程序文件main.exe，相关截图如下：





至此，上述三条指令便完成了main.exe生成的整个过程。

将编译程序分为三条指令完成可以让源代码在编译成目标代码时可以被并行处理，这种技术主要用于make或cmake等make系统中。

2.4 gcc -c命令的细化

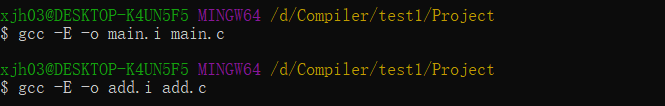
以下是gcc-c命令细化后的示意图，包括预处理器（gcc -E），编译（gcc -S）和汇编器（gcc -c）三个阶段。

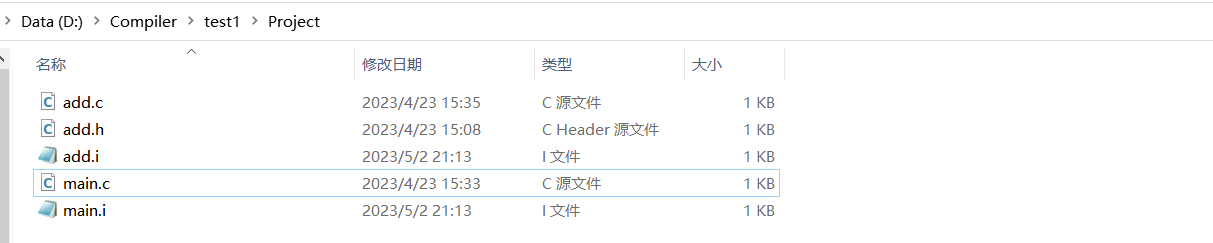
①gcc -E:

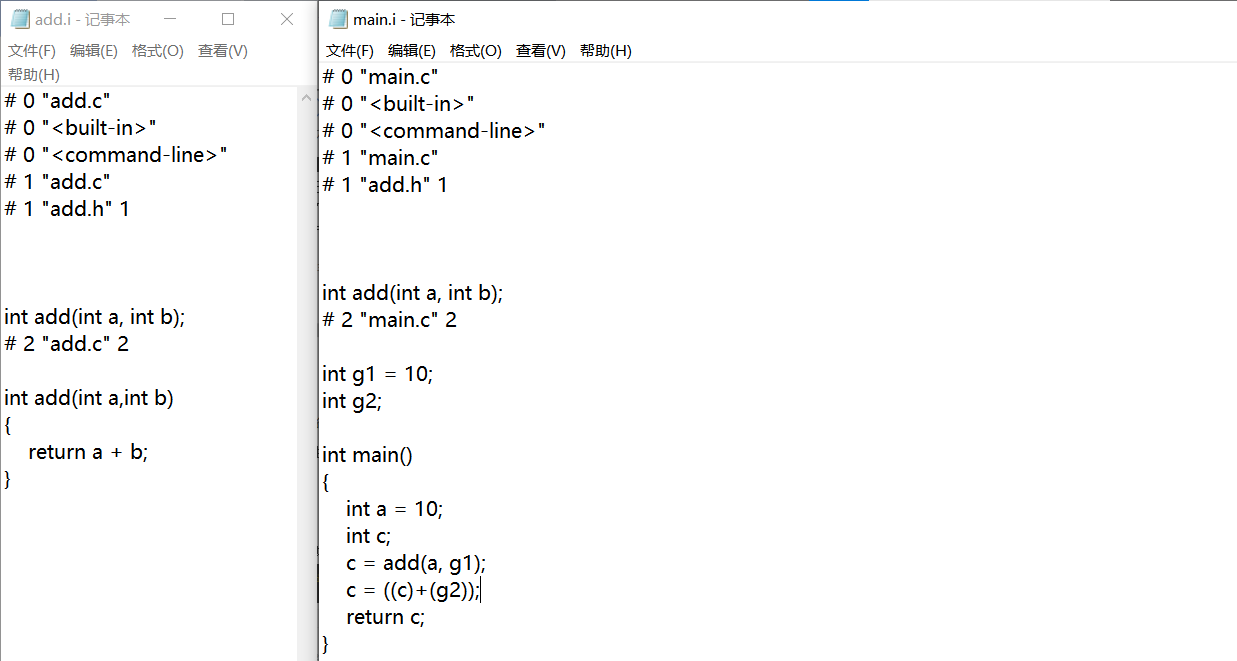
分别执行以下两条命令：

gcc -E -o main.i main.c

gcc -E -o add.i add.c







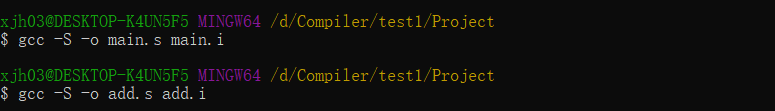
从上面展示的结果和具体内容可以看到，预处理器会根据源代码中的预处理指令，如#include、#define等，将源代码转换成一种更加适合编译器处理的形式。这个过程中，预处理器会将所有的预处理指令展开，并将宏定义替换成它们的实际值，生成一个对应的.i类型的文件，即预处理后的源代码文件。

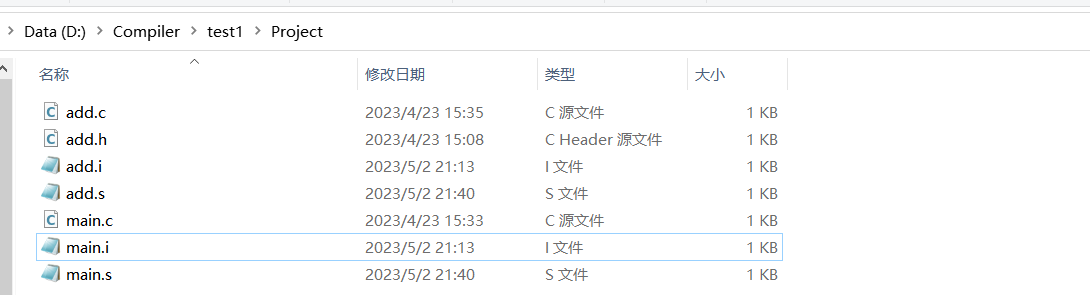
②gcc -S:

gcc -S是GCC编译器的一个选项，它的作用是将源代码编译成汇编语言代码，生成汇编语言代码文件。汇编语言是一种低级语言，它是机器语言的文本表示形式，可以直接被计算机执行。将源代码编译成汇编语言代码可以帮助开发人员深入理解程序的实现细节，同时也可以进行代码优化。我们可以分别执行以下两条命令查看对应内容：

gcc -S -o main.s mian.i

gcc -S -o add.s add.i







以主函数生成的汇编代码信息main.s为例，所生成的汇编代码是一个x86-64架构下的汇编代码，可以分为以下几部分：

（1）文件信息部分：包括".file"指令，用于指定源文件名；".ident"指令，用于在目标文件中插入标识符，这里显示了使用的GCC版本信息。

（2）数据段和代码段：包括".data"和".bss"指令用于定义全局数据区和未初始化数据区，".text"指令用于定义代码段。

（3）符号表信息：包括".def"指令，用于定义函数的符号属性，包括函数名、作用域、类型等信息。

（4）SEH过程信息：包括".seh\_proc"和".seh\_endproc"指令，用于定义SEH过程的开始和结束。

（5）主函数信息：包括"main"函数的定义、入栈、出栈和返回等信息。

（6）函数调用信息：包括对函数"\_\_main"和"add"的调用，包括参数传递、返回值存储和栈的调整等信息。

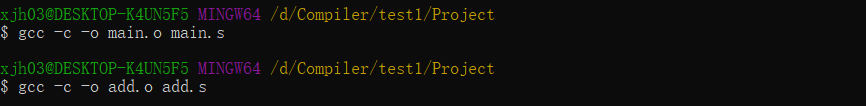
③gcc -c:

gcc -c命令是使用GCC编译器将汇编代码文件main.s编译成目标文件main.o的命令。其中，"-c"参数表示只编译生成目标文件，不进行链接操作，生成的目标文件可以作为静态库或动态库的组成部分，或者和其他目标文件一起链接成可执行程序，即生成的文件是一个可重定位的目标文件。

我们可以执行下面两条命令分别得到对应的可重定位文件：

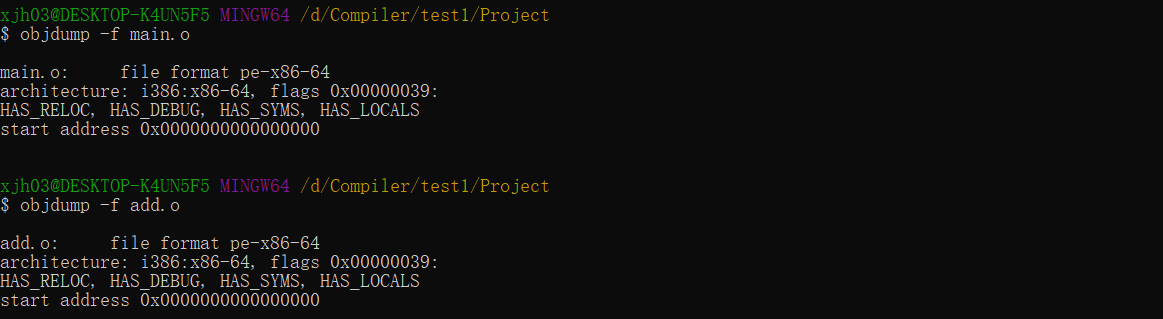
gcc -c -o main.o main.s

gcc -c -o add.o add.s





我们无法通过记事本直接查看.o类型的文件，只能通过objdump命令分析其内容。例如，我们可以通过命令objdump -f main.o和objdump -f add.o分别查看main.o和add.o文件头信息。



以上输出结果这个输出显示了目标文件main.o和add.o的格式、架构和标志位等信息。

（1）文件格式：file format pe-x86-64，表示这是一个Windows平台下的PE格式目标文件，采用x86-64架构。

（2）架构信息：architecture: i386:x86-64，表示该目标文件采用i386:x86-64的架构。

（3）标志位信息：flags 0x00000039: HAS\_RELOC, HAS\_DEBUG, HAS\_SYMS, HAS\_LOCALS，其中：

- HAS\_RELOC表示该目标文件包含重定位表，即该文件中有需要进行地址重定位的代码或数据。

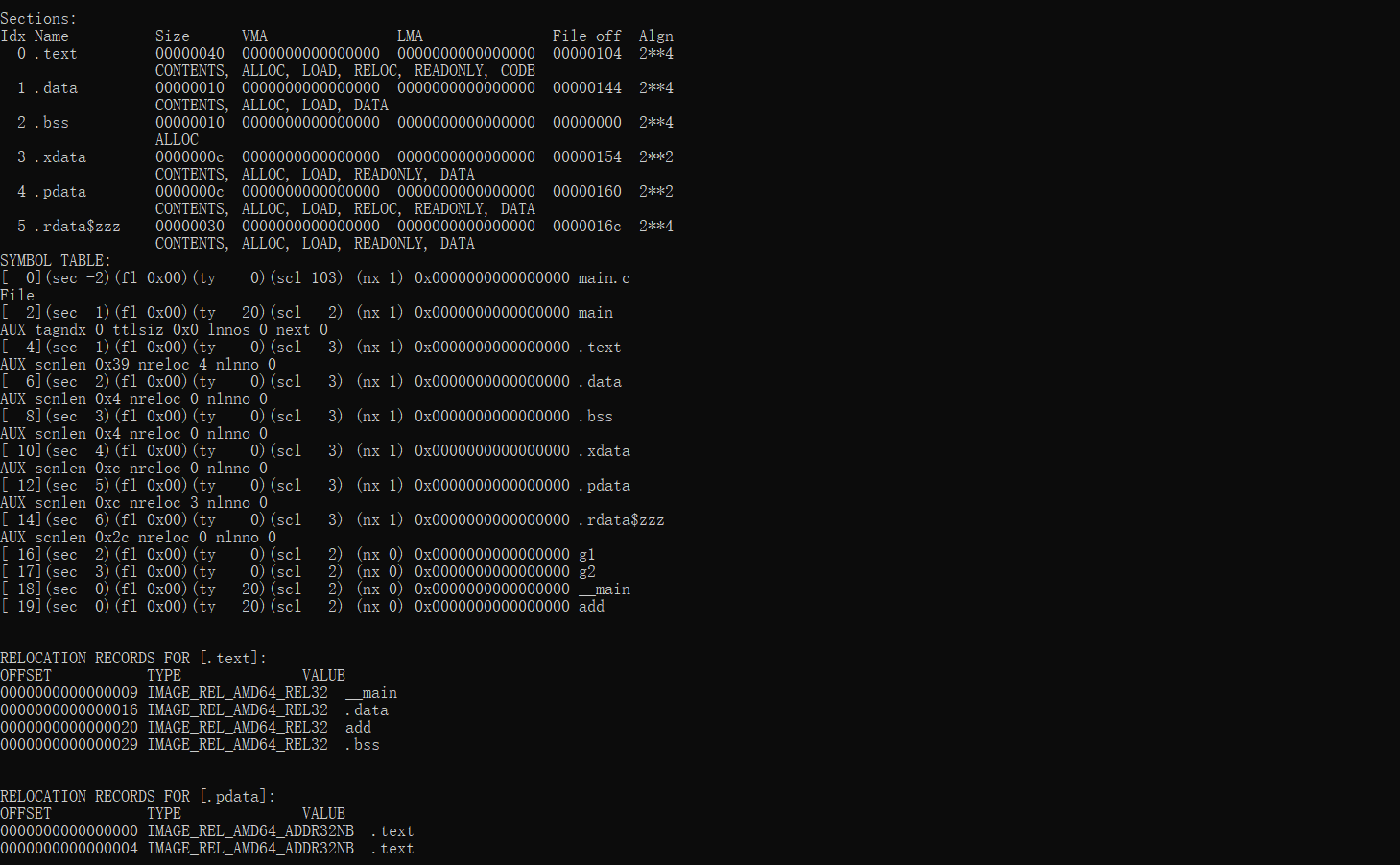
- HAS\_DEBUG表示该目标文件包含调试信息。

- HAS\_SYMS表示该目标文件包含符号表信息。

- HAS\_LOCALS表示该目标文件包含局部符号表信息。

1. 入口地址：start address 0x0000000000000000，表示该目标文件的入口地址为0x0000000000000000，即没有指定入口地址，需要通过链接操作后才能执行。

我们也可以使用objdump -x main.o和objdump -x add.o命令查看目标文件中的Section信息：



上图以main.o的Section信息为例，这个输出展示了目标文件main.o中各个段的信息，以及符号表和重定位表等信息。

Sections部分展示了目标文件中的各个段的信息，包括段名、大小(Size)、虚拟内存地址(VMA)、加载内存地址(LMA)、文件偏移(File off)和对齐方式(Algn)等。这个输出中包含了以下几个段：

.text：代码段，包含可执行代码。

.data：数据段，包含已初始化的全局变量和静态变量。

.bss：数据段，包含未初始化的全局变量和静态变量。

.xdata：扩展数据段，包含异常处理相关的信息。

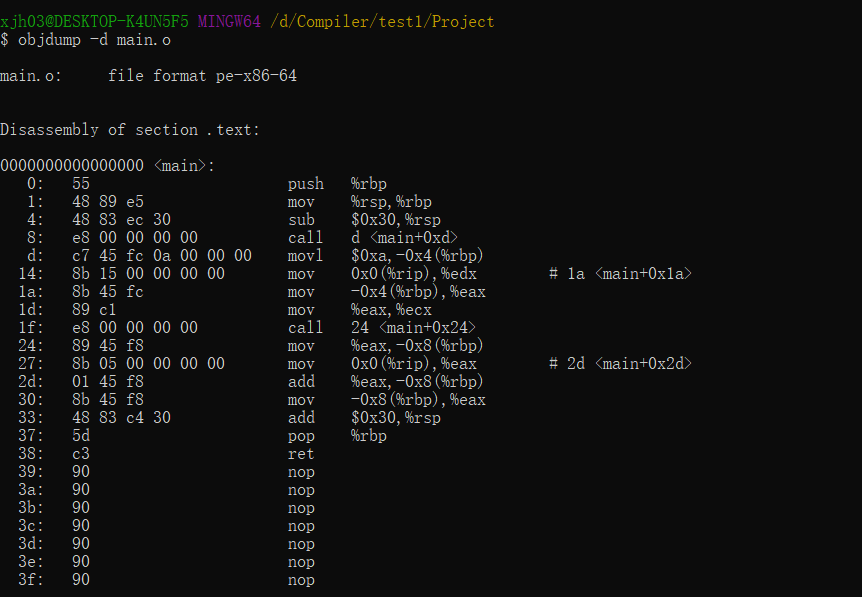
.pdata：过程数据段，包含函数的调试和异常处理信息。

.rdata$zzz：只读数据段，包含只读的数据变量。

SYMBOL TABLE部分展示了目标文件的符号表信息，包括符号的编号、段信息、类型、作用域、值等。

RELOCATION RECORDS部分展示了目标文件的重定位表信息，包括偏移量、重定位类型和重定位目标等。

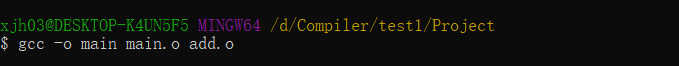
接着我们可以通过命令objdump -d main.o文件中代码段的反汇编信息如下：



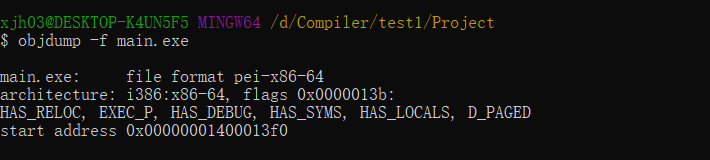
由于此时main.o还没有经过重定位操作，故上面框出的四字节的0都属于待定的地址，在链接时需要重定位。

④链接过程

上面我们已经将main.c和add.c文件生成了对应的目标文件，在生成main.exe可执行程序文件之前，还需要经过链接器的链接操作。此处可通过命令gcc -o main mian.o add.o来实现对两个目标文件的链接工作。

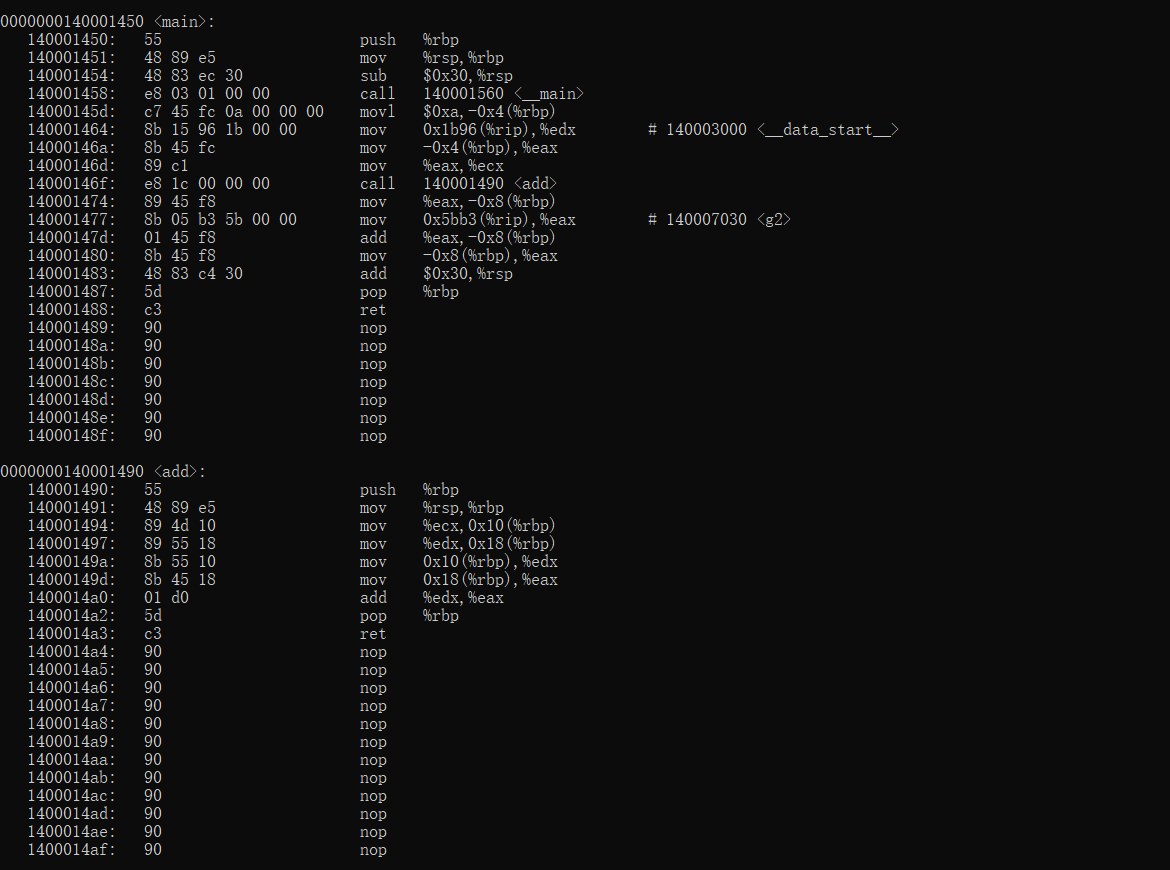


接着我们通过objdump -f main命令来查看main.exe的头文件信息，结果如下：



不难看出，与objdump -f main.o的输出结果相比，此处头文件信息中多了D\_PAGED属性，这表示该可执行文件使用分页内存管理模式。同时，start address的地址也不是全0了，这是因为在链接器的处理过程中，会将各个目标文件中的符号进行合并，并根据它们在程序中的实际使用情况来确定它们在内存中的最终地址。因此，在链接器的处理过程中，程序的入口地址可能会发生变化。

然后，我们可以通过objdump -d main.exe命令查看main.exe的指令信息：



与objdump -d main.o的输出相比，上述红框中圈出的指令中的地址不再是四字节的0，而是重定位完成后的地址。

3 编译过程中中间语言IR、汇编语言与源程序对应分析

请编写含有两个循环、两个循环不嵌套的 C 语言源程序，假定为test.c。要求源代码中宏定义与使用、变量定义、算术表达式混合运算、注释、一个循环中循环条件有短路求值、另外一个循环中要含有 break。

add.h:

同之前

add.c:

同之前

fibonacci.h:

#ifndef \_\_FIBONACCI\_H\_\_

#define \_\_FIBONACCI\_H\_\_

int fibonacci(int a);

#endif

fibonacci.c:

#include "fibonacci.h"

int fibonacci(int n) {

    if (n <= 1)

        return n;

    else

        return fibonacci(n-1) + fibonacci(n-2);

}

test.c：

#include "add.h"

#include "fibonacci.h"

#define g1 10

int g2;

int main ()

{

    int a = 10;

    int c, d;

    int mul;

    int f;

    int i = 1;

    int j = 1;

    for(j=1; j > 5 && j < g1; j++)

    {

        f = fibonacci(j);

    }

    while(1){

        mul= i \* add(a, g1);

        i++;

        if (mul > 100)   break;

    }

    d = fibonacci(a);

    c = add(a, g1);

    c = ADD(c, g2);

    return c;

}

在test.c中，程序通过包含头文件"add.h"和"fibonacci.h"来调用一个名为add()的函数和一个名为fibonacci()的函数。该程序还定义了常量g1和变量g2。

主函数中声明了多个整型变量a、c、d、mul、f、i和j，并给它们初始化。

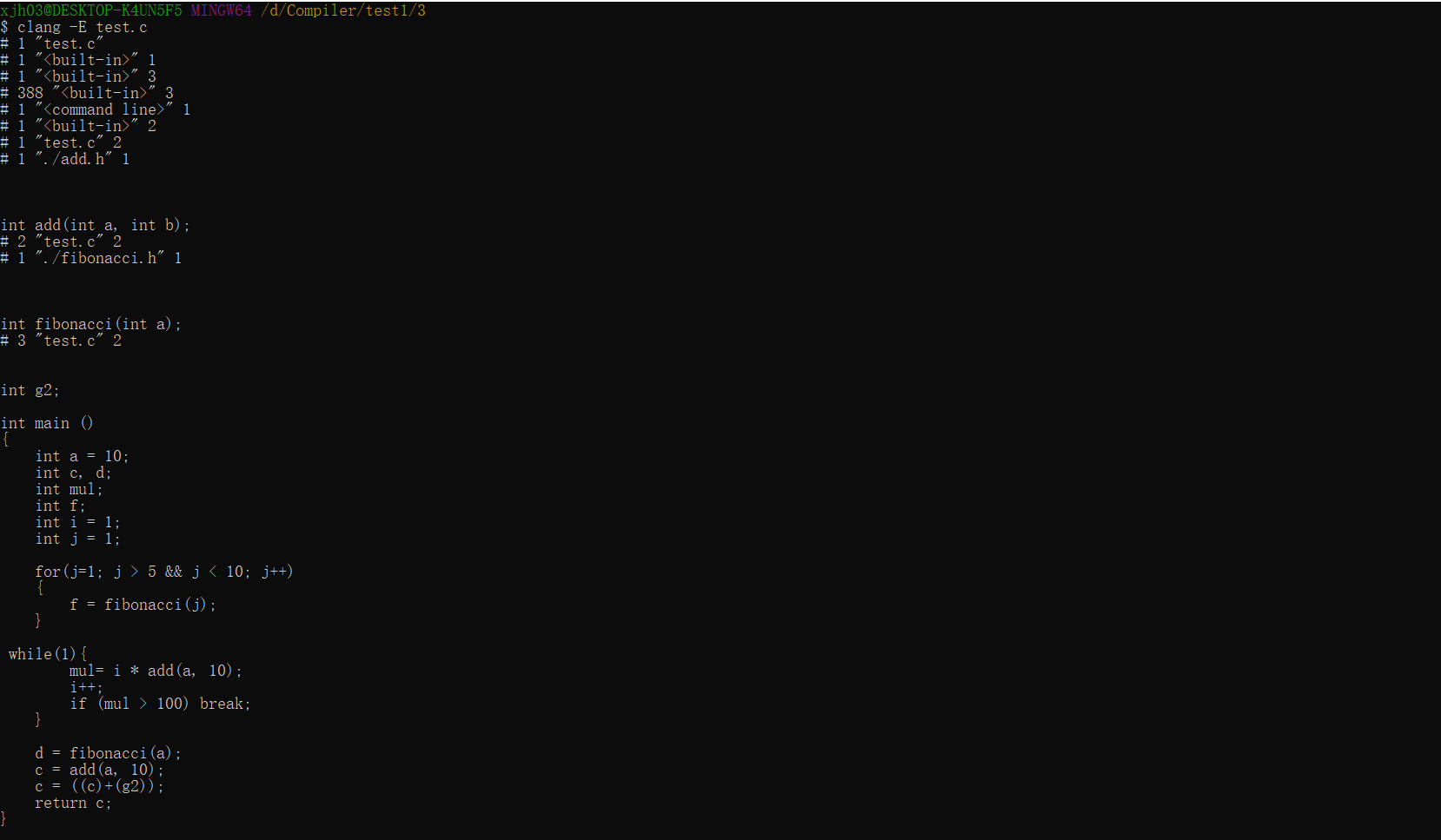
然后，程序使用for循环来计算前5项斐波那契数列，并将结果赋值给变量f。该循环并不执行，因为循环条件j > 5 && j < g1初始值为1，而1既小于5又小于g1的值10。因此，该循环没有被执行，变量f也没有被赋值。

接着，程序使用while循环来计算i与(add(a,g1))的积，并将其与100进行比较。当积超过100时，循环结束。循环体中每次i都递增一次，直到积超过100。

最后，程序通过调用fibonacci()函数计算变量a的斐波那契数列值，并将结果赋值给变量d。程序调用add()函数并将结果存储在变量c中，然后使用ADD()将c与g2相加并返回。

3.1分析与处理功能

命令：clang -E test.c

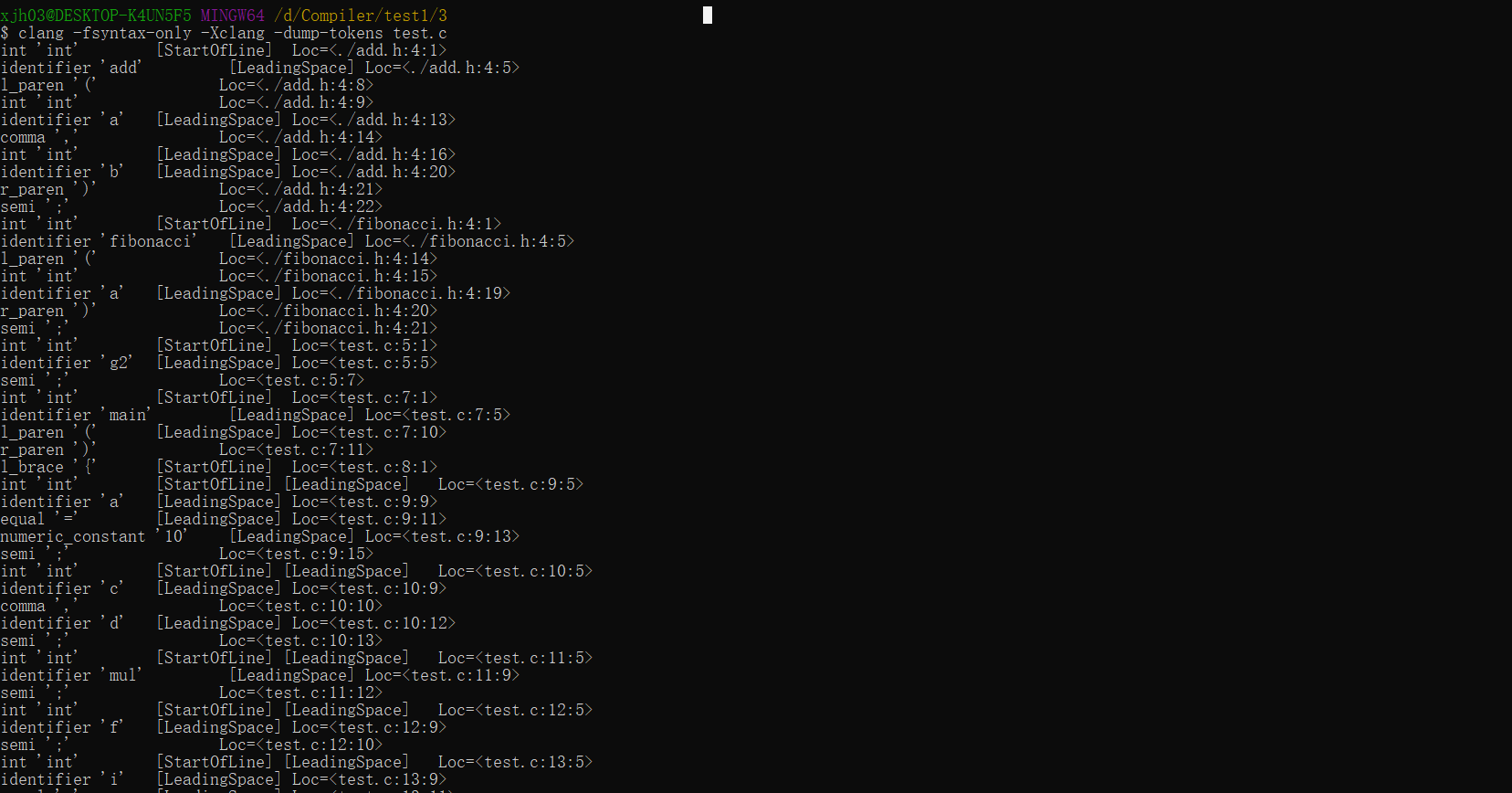


这段代码是经过预处理器处理后的结果，预处理器解析了代码文件test.c中包含的头文件"add.h"和"fibonacci.h"，并将它们的内容插入到这段代码中。

在这段预处理后的代码中，程序的头文件声明部分已经被展开，包括add.h和fibonacci.h文件中的函数声明。

3.2分析词法功能

命令：clang -fsyntax-only -Xclang -dump-tokens test.c



上面截图展示了部分词法分析输出，输出结果中的每一行都对应着代码中的一个词法单元，例如关键字、标识符、运算符、括号等。这些词法单元是编译器在分析和解释代码时所使用的最基本的语法单元，在这个过程中会被被打印出来。在这个命令中，输出的词法单元按照如下格式展示：

词法单元类型 标记值 [标记类型] 位置信息

其中：

词法单元类型：表示该单元所属的词法类型，例如标识符、关键字、运算符等。

标记值：表示该单元实际在代码中的内容，在这里显示为字符串形式。

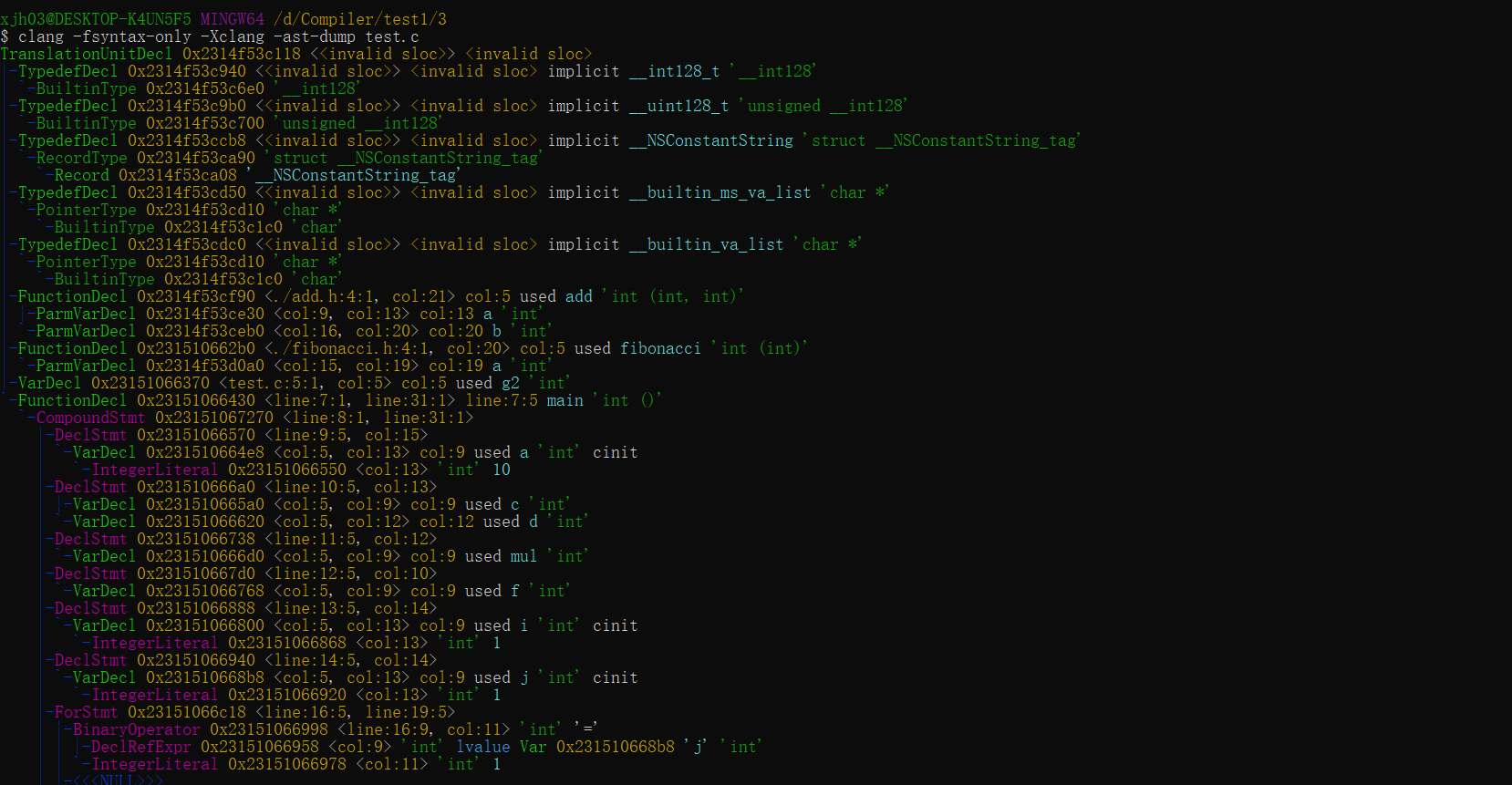
标记类型：表示该单元的详细类型信息，例如 LeadingSpace 表示该单元前面有空白字符。

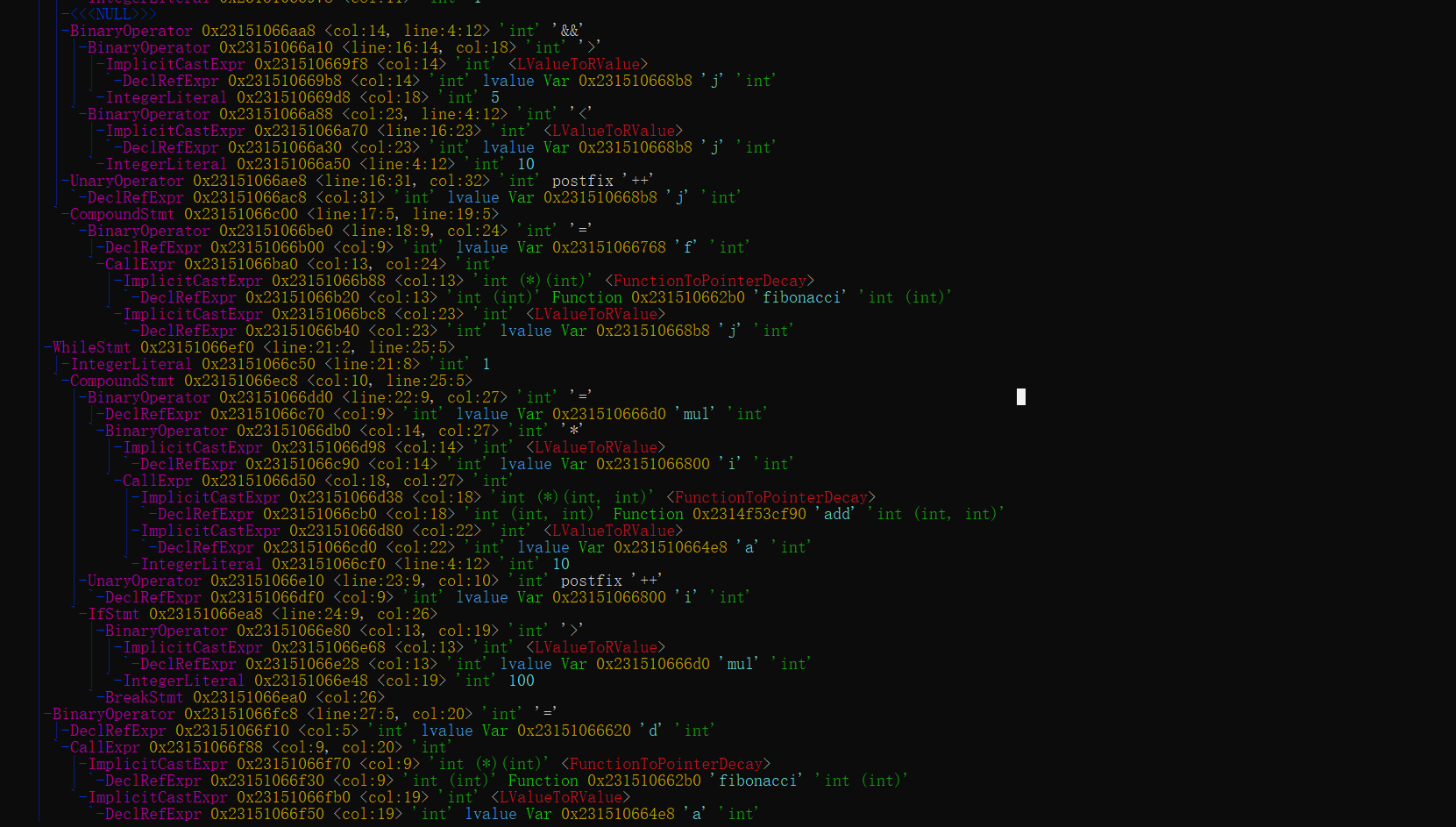
位置信息：表示该单元在代码文件中的位置信息，包括所在文件、行号和列号。

通过这些输出的词法单元，我们可以更好地理解代码的结构和构成，从而更加深入地了解代码的语法规则和实现细节。同时，词法单元也是编译器中非常基础的概念，对于了解编译过程和编译器实现原理都有着非常重要的作用。

3.3分析语法产生出的抽象语法树

命令：clang -fsyntax-only -Xclang -ast-dump test.c







首先，根节点是一个 TranslationUnitDecl 节点，表示整个源文件的翻译单元声明。它包含了所有其他节点。

其中， TypedefDecl 节点表示了三种类型的定义声明: \_\_int128\_t、 unsigned \_\_int128\_t 和 \_\_builtin\_va\_list。

接下来是 FunctionDecl 节点，表示主函数。它包含了一个 CompoundStmt 节点，表示函数体，其中包含了所有其他的节点。

函数体首先包含了 DeclStmt 节点，表示变量 a 的声明和初始化，和变量 c 和 d 的声明。它们都是整型变量，变量 a 的初始值为 10。

接下来是一个 BinaryOperator 节点，表示变量 c 的赋值。它是 add(a, g1) 和 g2 的和， add(a, g1) 调用了函数 add，这个函数的调用通过一个 CallExpr 节点表示。变量 d 是通过 fibonacci(a) 的调用计算得到的，这个调用也使用了 CallExpr 节点。

然后是一个 DeclStmt 节点，表示变量 mul 的声明和初始化。这个变量的计算涉及到 i、 a 和 g1，因此使用了一个 BinaryOperator 节点和两个 ImplicitCastExpr 节点计算了它的值。

接下来是一个 ForStmt 节点，表示循环语句。它的条件是 j > 5 && j < g1，但这个条件在代码中是错误的，因为 j 被初始化为 1，而 g1 的值为 10。循环体中计算了 f 的值，f 是通过 fibonacci 函数调用计算得到的。

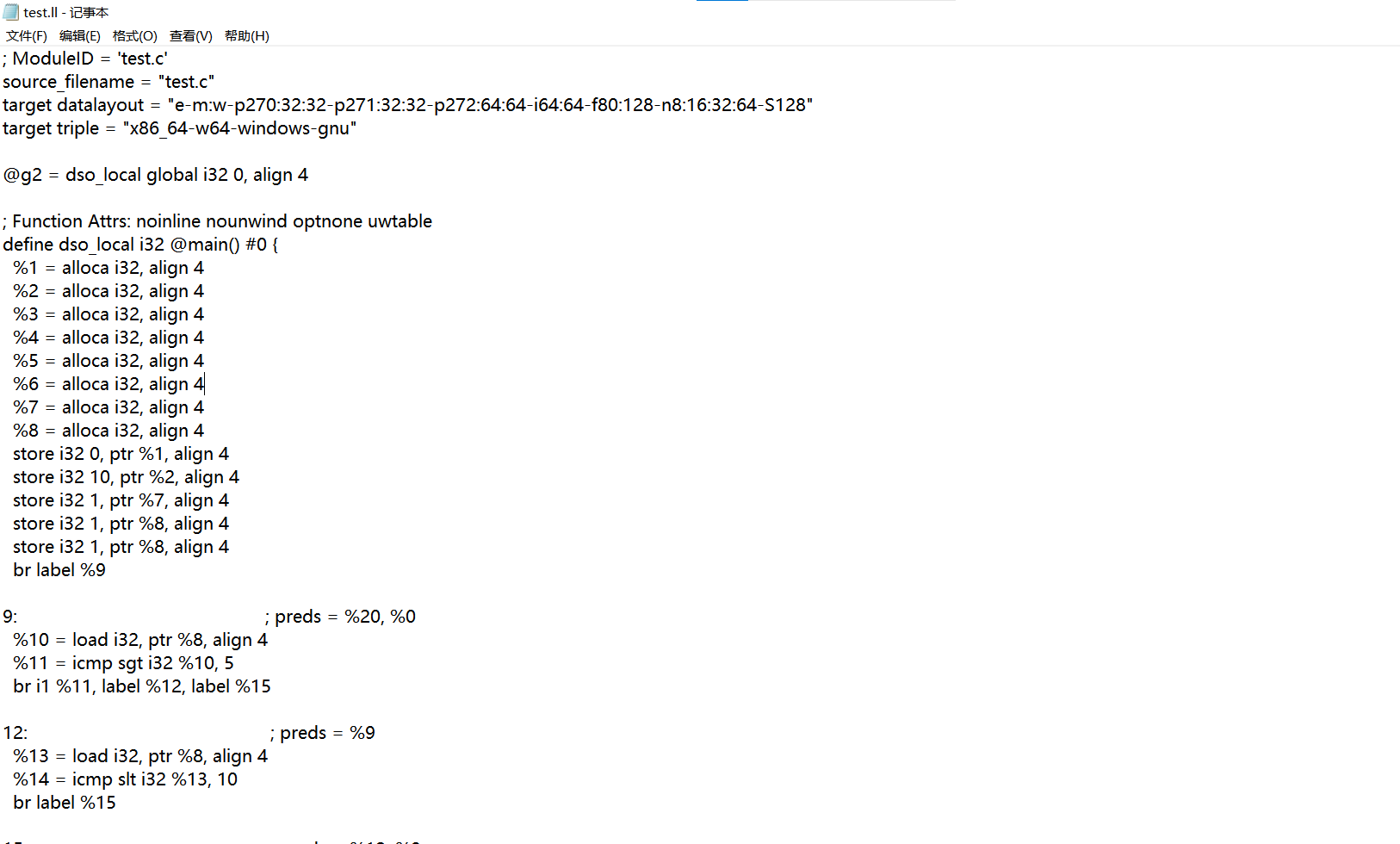
最后是一个 WhileStmt 节点，也是循环语句。它的循环条件是 1，因此会一直循环下去，直到 mul 的值大于 100。循环体中计算了 mul 的值和变量 i 的值。这个循环使用了 BreakStmt 跳出。

最后是一个 ReturnStmt 节点，表示函数的返回语句。它包含了一个 DeclRefExpr 节点，表示变量 c 的引用，这个变量即为函数的返回值。

总体来说，这个抽象语法树展现了 C 语言源代码的结构和语义信息，但并不包含代码的具体执行细节。

3.4中间代码生成

命令：clang -S -emit-llvm -o test.ll test.c



上面截图展示了部分test.ll的内容。test.ll文件的内容是源程序编译过程中的中间表示代码（IR），它是源程序到机器代码之间的一个中间产物。与源程序的对应关系如下：

（1）在test.ll文件中会有全局变量声明g2，定义了为i32类型的全局变量g2，它对应源代码中的int g2。

（2）在test.ll文件的main函数中，将声明并定义了int类型的变量a，c，d，mul，f，i和j，可以看出这些本地变量在LLVM中，它们会被声明为在堆栈上分配的变量。这些变量的赋值和条件语句等操作将会在LLVM IR中映射成对应的操作码。

（3）在test.ll文件中的循环结构代表的是源程序中的for循环，其中调用了函数‘fibonacci’，执行了赋值操作。

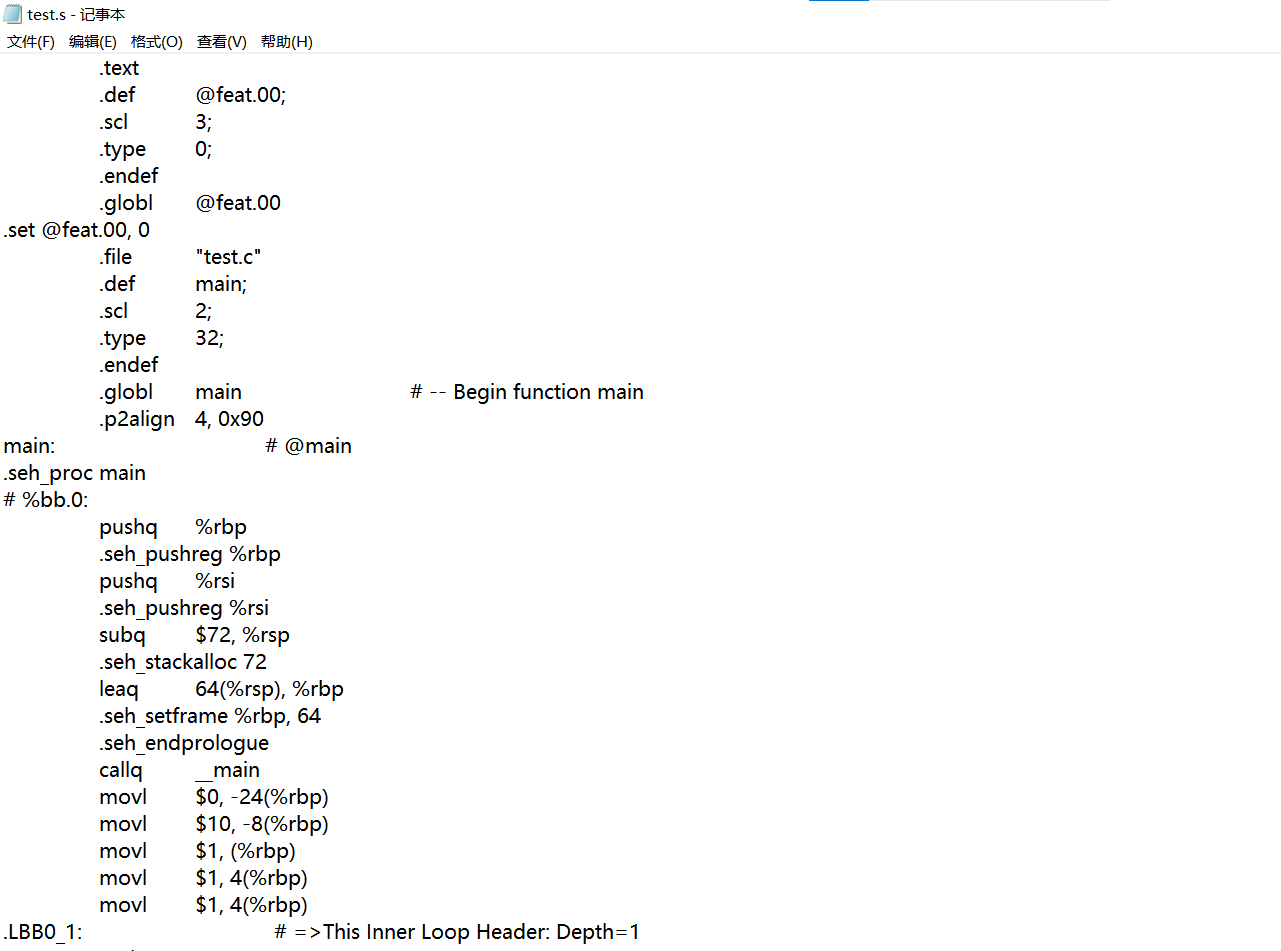
（4）其中while循环的部分对应源代码中的 while 循环结构。

（5）最后一部分相关函数的调用和变量赋值对应着源代码中的函数调用和结果输出的语句。

需要注意的是，由于LLVM IR采用的是抽象语法树（AST），它们可能会在行和列的位置上有所变化。但总体来说，在test.ll文件中的LLVM IR代码与源程序中的语义和行为是一一对应的，而且它们维护了相同的执行逻辑和结构。

3.5汇编语言生成

命令：llc test.ll -o test.s



生成的汇编语言文件test.s部分如上图所示。

3.6可执行程序生成

命令：clang test.s add.s fibonacci.s -o test



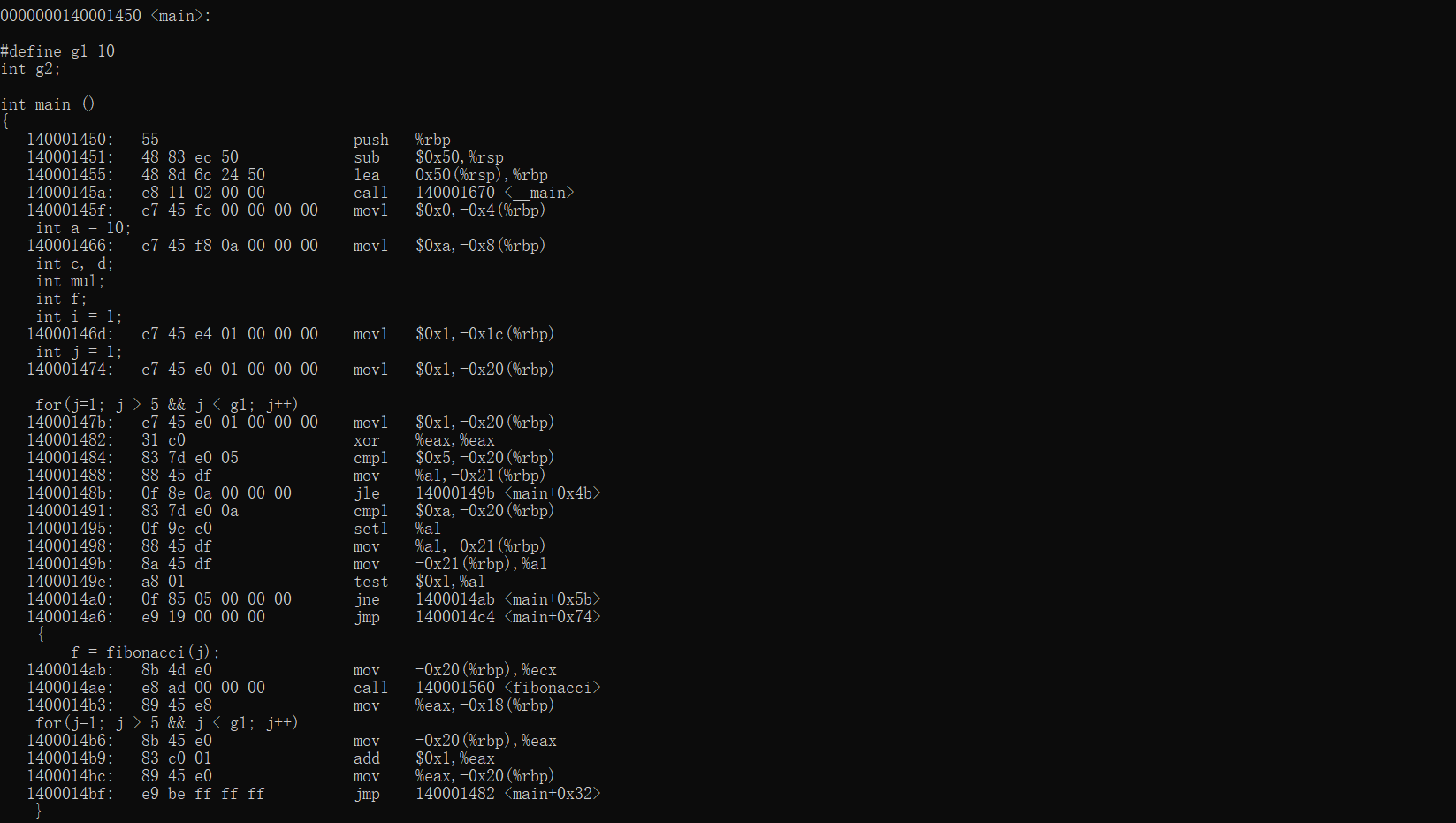


可执行程序生成完毕。

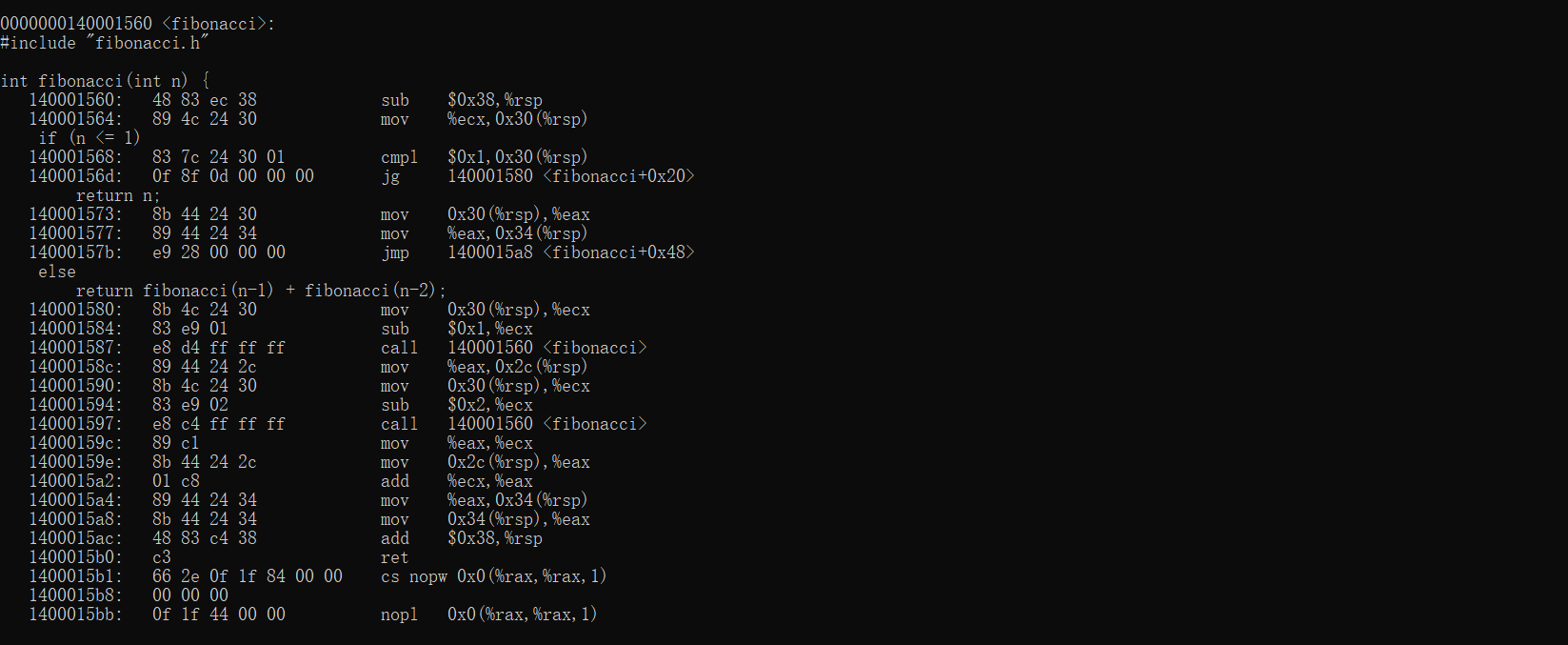
3.7反汇编test程序

命令：objdump -dS test

命令：clang -g -o test test.c后通过objdump查看反汇编更合适







上面展示了部分反汇编结果，我们可以发现其中很多地址被重定位了。这是因为中间IR是一种抽象的代码表示方式，与具体的机器指令无关，所以它的地址是相对稳定的。而在将中间IR转化为机器指令时，由于目标机器的不同和指令长度的不同，会导致机器指令的存储位置发生变化，从而导致反汇编代码中的地址需要重新计算。

4 函数调用与返回

请编写含有函数调用（要有参数传递，参数个数至少6个）的C程序理解编译器在函数调用时所做的动作，并通过汇编试图去分析，验证函数的动态执行过程(含返回)。

程序编写test4.c**：**

#include <stdio.h>

int add (int a, int b, int c, int d, int e, int f, int g, int h)

{

    return a + b + c + d + e + f + g + h;

}

int main()

{

    int sum = add (1, 2, 3, 4, 5, 6, 7, 8);

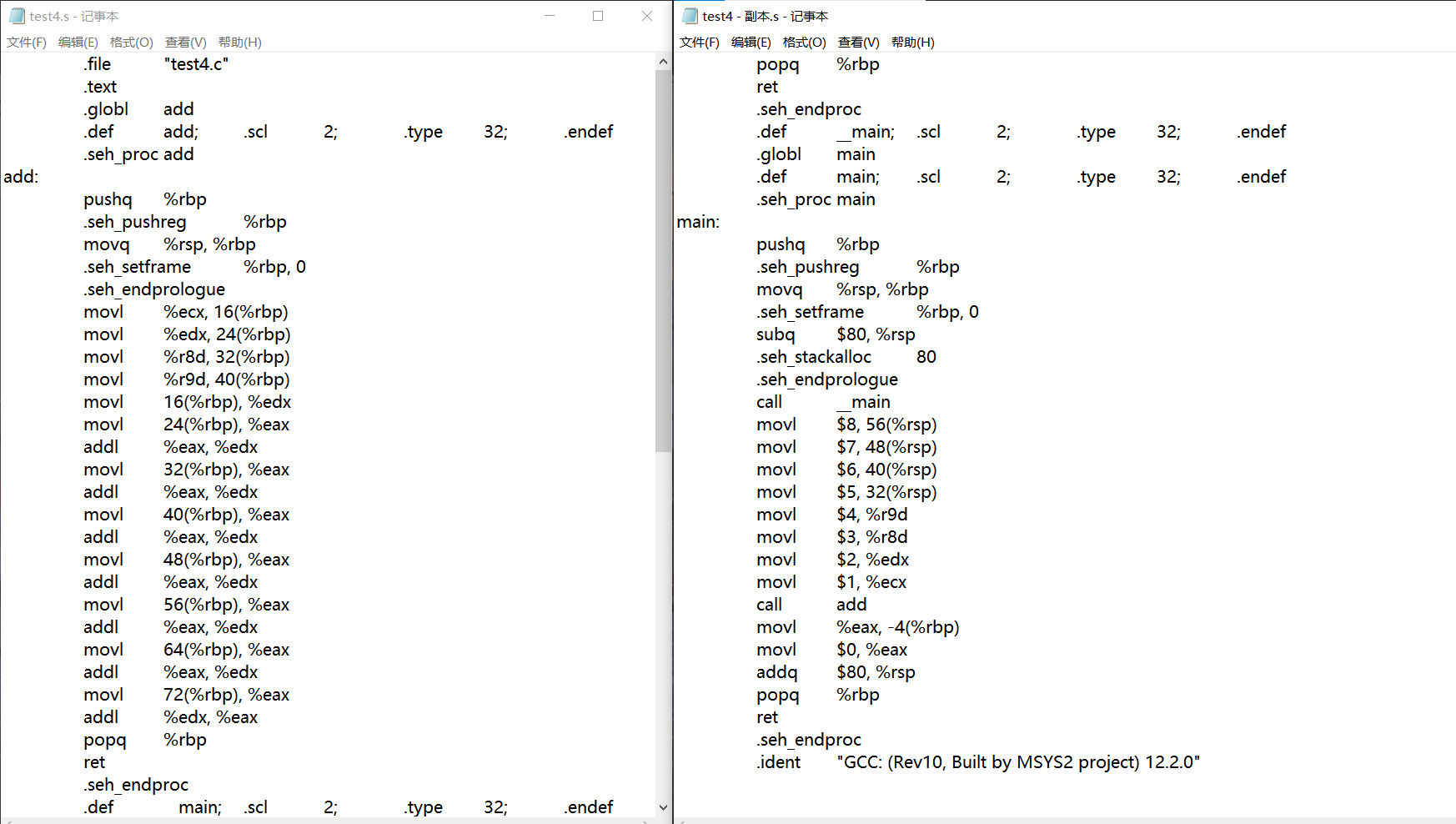
    return 0;

}

上述程序中，主函数调用了上面定义的add()函数，并向其中传递了8个参数。

请优先利用64位的编译器，若没有可使用32位的编译器。64位编译器默认前几个参数通过寄存器传值，后面的采用栈内存传值。请分析C语言的函数调用栈的内容并分析函数如何动态分配栈、参数的传递方式、实参的压栈次序等。

通过$ gcc -E -o test4.i test4.c，$ gcc -S -o test4.s test4.i，$ gcc -c -o test4.o test4.s，$ gcc -o test4 test4.o几条命令将上述程序文件编译链接完成后，发现程序没有错误，我们便可以查看test4.s文件中的汇编代码。相关截图如下：



在上述示例汇编代码中，函数add有8个整型参数，其中前四个(%ecx, %edx, %r8d, %r9d)通过寄存器传递，后四个则通过将其压入栈中来传递。

在调用add函数的时候，main函数将8个值分别压入栈中，栈的内容顺序与实参的次序相反，即先将最后一个实参压栈。然后通过将六个寄存器分别对应到对应的参数位置，加上栈中８个参数，调用栈中包含了函数调用的所有参数。当调用add函数时，将先将寄存器和栈中的参数从左到右压入栈中，然后在函数内部进行处理。

在函数add内部，首先通过pushq指令将基指针入栈为了给之后返回时恢复。立即之后，函数后续使用movq和seh\_setframe指令将帧指针rbp设置为0，并将当前的栈指针赋给基指针rbp。之后在函数体内使用指令movl将从rbp + 16到rbp + 40的8个参数取出并存储到局部变量中，最后使用指令addl对局部变量进行累加。在处理完毕后，将结果存储到eax寄存器中并通过popq指令恢复基指针的值，最后使用ret指令将控制权返回到main函数。

值得注意的是，经过查询资料，我了解到不同的操作系统和编译器可能有不同的函数调用约定。例如，Windows和Linux使用的函数调用约定就有所不同。在64位x86架构下，Windows使用的是Microsoft x64 calling convention，其中前四个整型参数使用%rcx、%rdx、%r8和%r9的寄存器进行传递，其他整型参数和浮点参数都通过栈传递。而在Linux下，使用的是System V AMD64 calling convention，其中前六个整型参数使用%rdi、%rsi、%rdx、%rcx、%r8和%r9的寄存器进行传递，其他参数仍然通过栈传递。因此，不同的操作系统和编译器所使用的函数调用约定可能会影响到函数调用时参数的传递方式、寄存器的使用等细节，进而可能会影响到代码的汇编结果。

5 课后作业1

5.1作业要求

编写的一个简单C语言例子（至少包括3个C文件、至少一个头文件；含有初始化全局变量、未初始化全局变量、函数内静态变量与局部变量赋值语句、if 语句等），尽量不要包含第三方的头文件，按照预处理->编译->汇编->链接的操作过程，截图分析，理解并分析源程序与可执行程序Section 的对应关系，特别是数据与指令，理解编译器的功能。

5.2程序编写

此处使用本报告中“3 编译过程中中间语言IR、汇编语言与源程序对应分析”部分的代码，在其中加上以下文件：

factorial.c:

#include "factorial.h"

int factorial(int a)

{

    // 函数内部静态变量，用于统计函数被调用的次数

    static int count = 0;

    count++;

    int result = 1;

    if (a == 0)

        return 1;

    else if (a < 0)

        return 0;

    else while (a || 0){

        result \*= a;

        a -= 1;

    }

    return result;

}

factorial.h:

#ifndef \_\_FACTORIAL\_H\_\_

#define \_\_FACTORIAL\_H\_\_

int factorial(int a);

#endif

同时将test.c中第一个循环中的f = fibonacci(j);改为f = factorial(j);初始化全局变量、未初始化全局变量及局部变量之前均有定义，现在factorial()函数中定义一个函数内部静态变量count，用于统计函数被调用的次数。接下来便可以开始分析。



5.3预处理

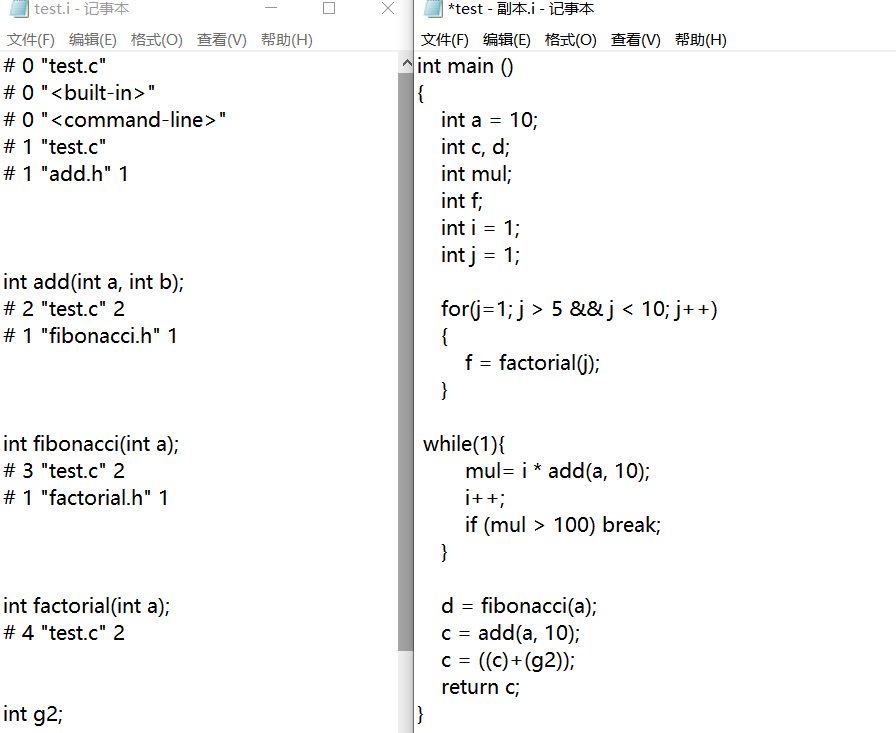
使用以下4条命令完成所有源文件的预处理操作：

gcc -E -o add.i add.c

gcc -E -o fibonacci.i fibonacci.c

gcc -E -o factorial.i factorial.c

gcc -E -o test.i test.c



上述4条命令执行成功后，分别生成了add.i，fibonacci.i，factorial.i和test.i4个预处理文件，以上图中展示的add.i文件为例，我们可以看到：#include 部分被去掉了，并将包含的头文件替换了进来。#define 宏定义被展开成了代码。最终生成了一个干净的 C 代码文件，预处理器处理后的代码更完整，可以被编译器编译使用。

5.4编译

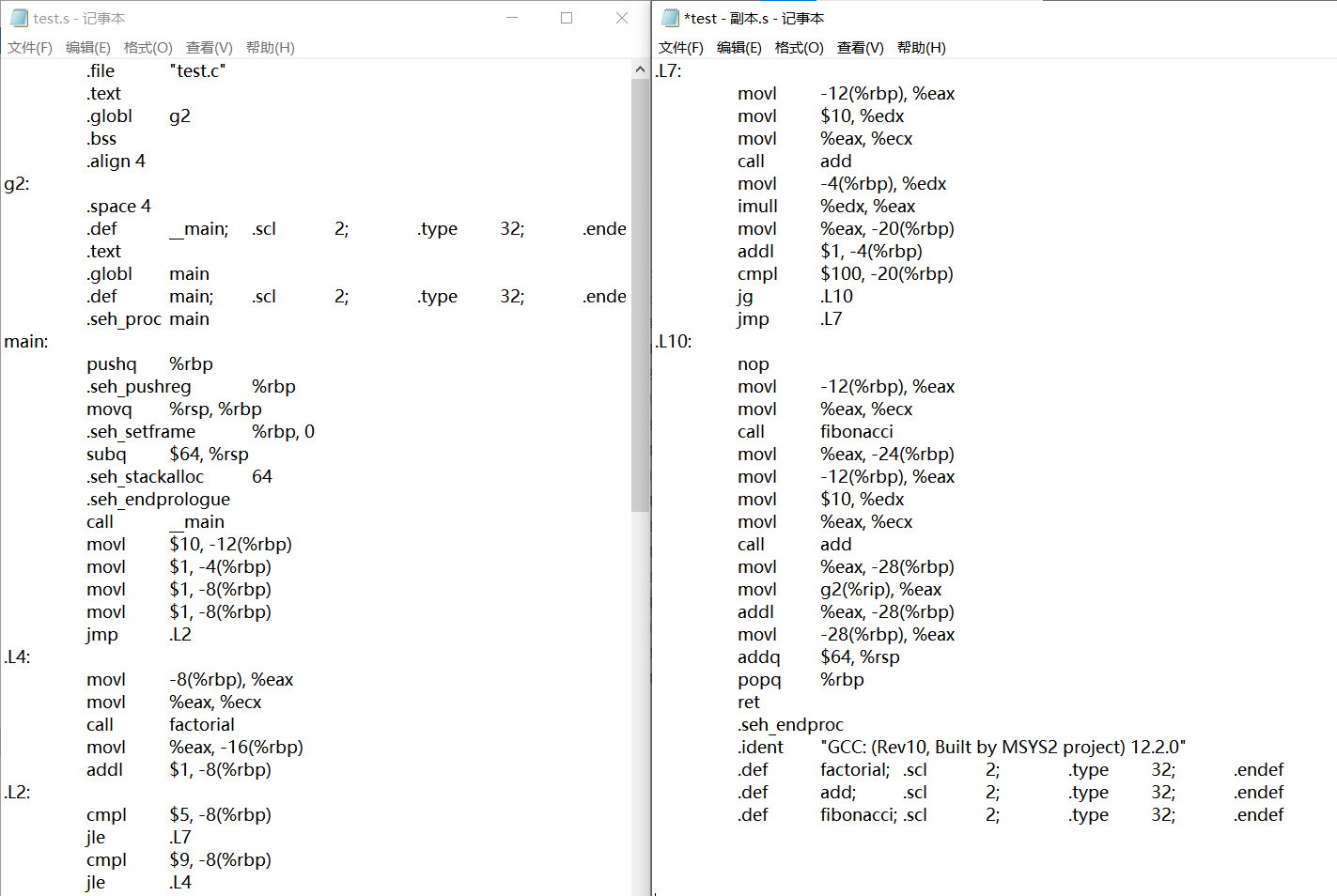
使用以下4条命令完成所有预处理文件的编译操作：

gcc -S -o add.s add.i

gcc -S -o fibonacci.s fibonacci.i

gcc -S -o factorial.s factorial.i

gcc -S -o test.s test.i



上述4条命令执行成功后，分别生成了add.s，fibonacci.s，factorial.s和test.s4个预处理文件，以上图中展示的add.s文件为例，经过预处理和编译的过程生成了汇编代码。

接下来分析test.s文件中的一些指令：第一行 .file "test.c" 表示这个文件来源于 test.c 文件。.globl g2 表示 g2 是一个全局的变量。.bss 段用来定义未初始化的全局变量 g2 占用 4 个字节的存储空间。.def \_\_main ... 用于定义一个符号 \_\_main。.globl main 表示 main 函数是程序入口。.seh\_proc main 和 .seh\_endproc 表示 Windows 的异常处理机制，不用深究。subq 指令用于在栈上分配空间。jmp 指令按照指定的标签无条件跳转到指定的地址执行相应的代码。cmp 和 jle 指令是比较和跳转指令，用于实现类似 if 或 while 语句。.L4: 和 .L10: 是标签。call 指令用于调用其他函数。movl 指令是将数据从内存或寄存器传送到另一个寄存器。addl 指令是将两个数据相加。imull 指令是将两个数据相乘。movq 指令是将数据从内存或寄存器传送到另一个寄存器，不同于 movl，它可以传递 64 位整型数据。ret 指令是从函数栈中返回结果。

5.5汇编

使用以下4条命令完成所有汇编源程序文件的汇编操作：

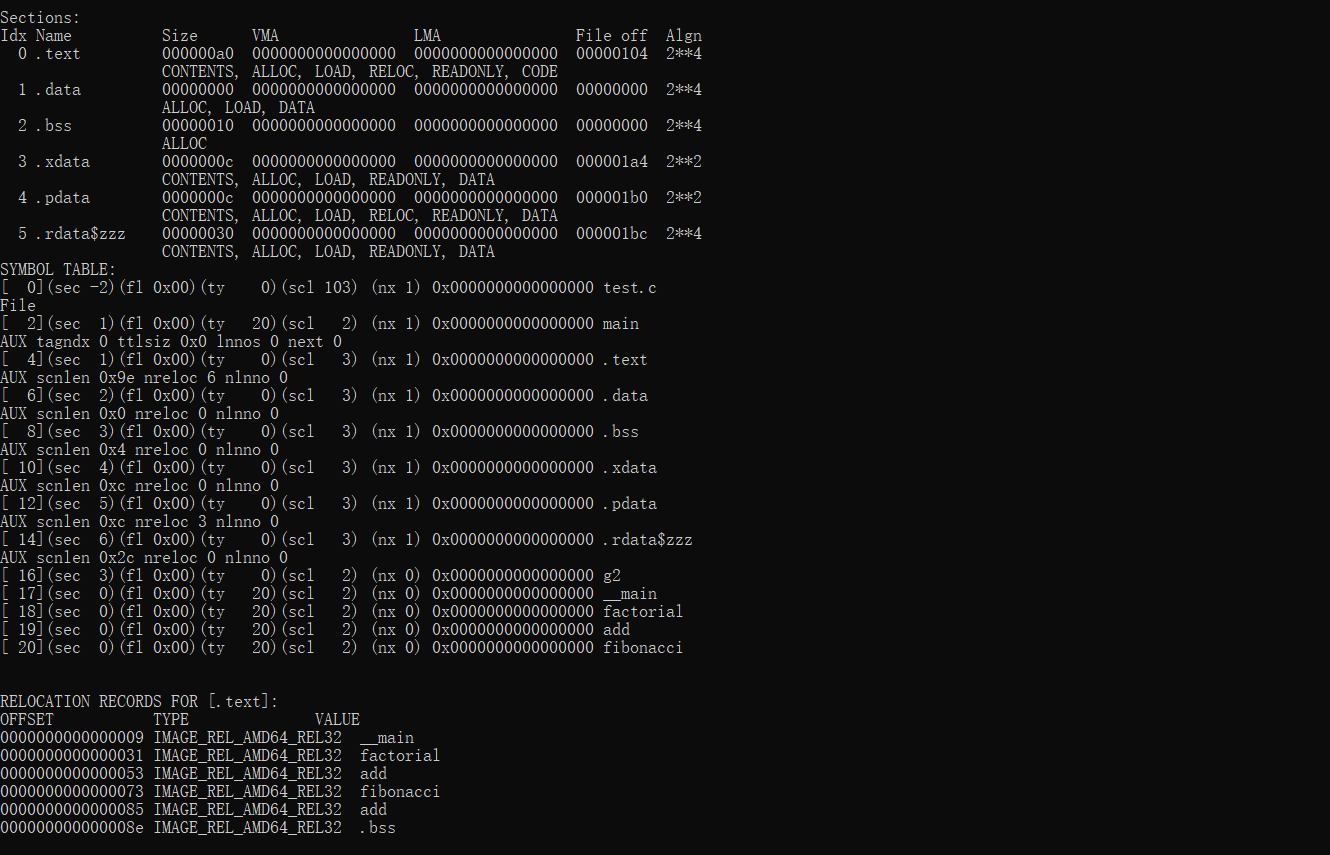
gcc -c -o add.o add.s

gcc -c -o fibonacci.o fibonacci.s

gcc -c -o factorial.o factorial.s

gcc -c -o test.o test.s

上述4条命令执行成功后，分别生成了add.o，fibonacci.o，factorial.o和test.o4个可重定位程序文件，接着我们使用objdump -x test.o命令来查看test.o文件中的Section信息：



我们可以结合源代码对其进行分析：可重定位程序的每个Section与源代码中的变量和函数之间的关系。

.text：该节包含程序的机器代码，包括main函数以及调用的add、factorial和fibonacci函数。这些调用是通过给定的头文件(add.h，factorial.h和fibonacci.h)来实现的。可重定位程序中的代码段(.text)的大小为0x000000a0。

.data：该节包含程序中初始化并具有初始值的变量g2的值，该变量在main函数之前定义。由于其被定义且初始化，.data节的大小为0x00。

.bss：该节包含程序中未初始化的全局变量g2。这是在main函数之前通过上述#define g1 10语句定义的。 注意，它的大小为0x10，即使它只是由一个int变量定义。

.xdata：该节包含只读数据部分，可能包含异常处理期间使用的指针或信息，大小为0x0000000c。它不直接与源代码中的任何函数或变量相关，但可能包含用于处理异常的信息。

.pdata：该节包含程序代码执行期间可能引发异常的指令和相关的异常处理程序的地址。它也不直接与源代码中的任何函数或变量相关，但它提供了对异常处理期间使用的地址的跟踪。它的大小为0x0000000c。

.rdata$zzz：与main函数调用的add、factorial和fibonacci函数以及它们在源代码中的头文件add.h、factorial.h和fibonacci.h相关。

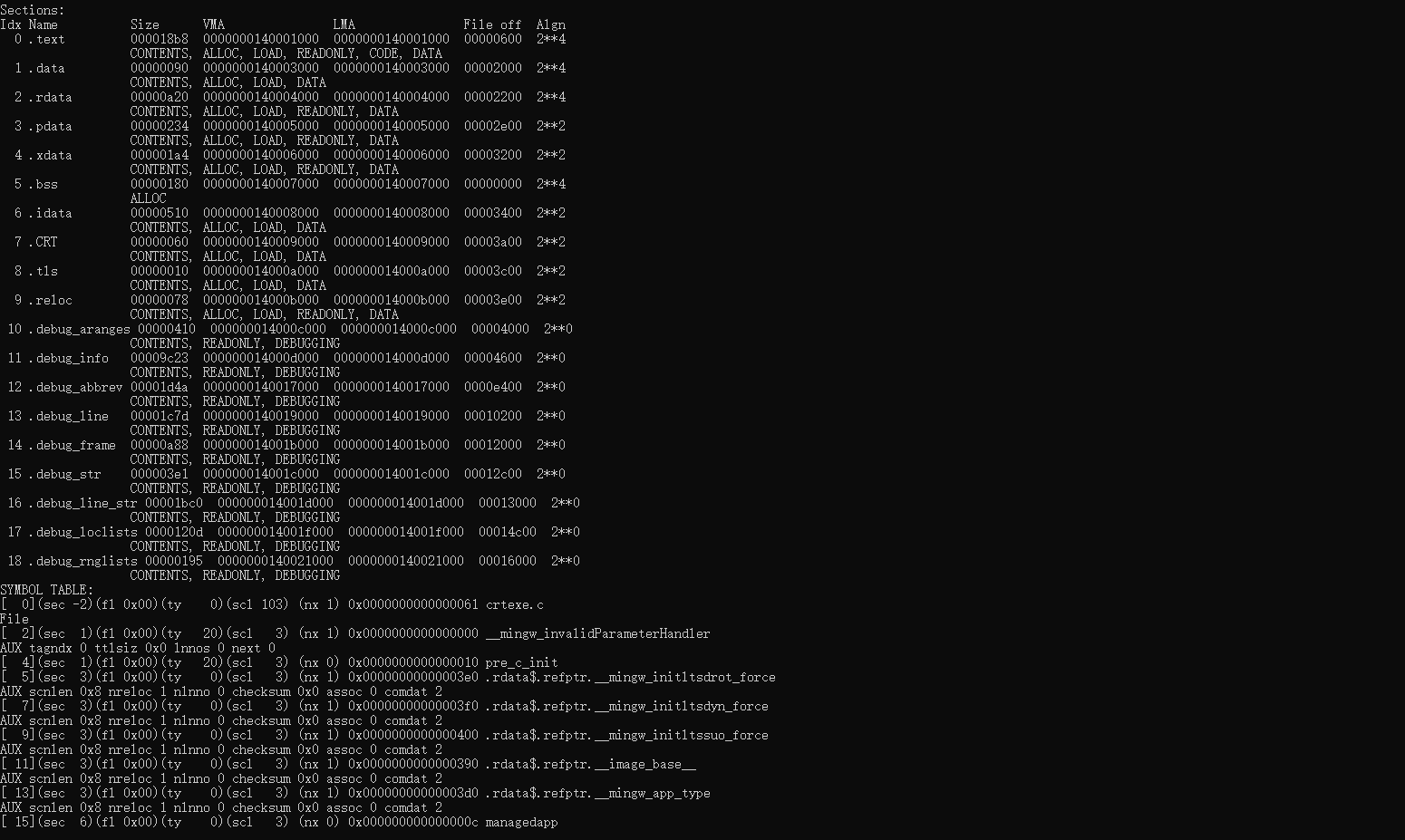
.data和.bss：与源代码中的g2变量相关。

.xdata和.pdata：与异常处理相关，提供了对程序代码执行期间可能引发的异常的更多细节。

5.6链接

接着我们可以使用gcc -o test add.o fibonacci,o factorial.o test.o将上一步得到的4个可重定位程序文件链接起来。

执行完成后便可以生成一个可执行程序test.exe，再通过objdump -x test.exe即可查看可执行程序的Section部分。其结果如下：



首先，我们知道源代码包含三个header文件add.h、fibonacci.h和factorial.h，这些文件包含函数原型声明。这些声明仅告诉编译器如何调用这些函数，但实际的函数定义在不同的源文件中。当生成可重定位程序时，这些函数的实现和主函数将被编译成单独的对象文件。当生成可执行文件时，这些对象文件已经连接在一起，生成单个的可执行文件。

通过阅读可重定位程序和可执行文件的Section表，可以看到可执行文件中有更多的Section，如 .idata（包含导入信息），.CRT和.TLS，而这些在可重定位程序文件中并不存在。这些额外的Section是编译器或链接器为支持可执行文件而添加的。

在.text部分，可重定位程序文件中只有 main 函数的机器指令，而在可执行文件中包含了所有函数的机器指令。.rdata部分包含只读数据，可重定位程序文件和可执行程序文件中都会有，但对于可执行程序，数据量可能更大。

总的来说，在源代码中包含多个函数的情况下，可重定位程序文件和可执行程序文件之间的主要区别在于链接器在连接所有对象文件时使用了更复杂的算法来合并多个函数和数据段，同时添加了一些链接信息。

6 课后作业2

6.1作业要求

编写含有while循环、for循环的C程序，分析C语言的每个句子与汇编语言的对应关系，试图理解编译器如何实现的语义翻译功能。

6.2 程序编写

forWhile.c:

int main() {

    int i = 1;

    int sum = 0;

    while (i <= 10) {

        sum += i \* i;

        i++;

    }

    int j;

    int sum2 = 0;

    for (j = 1; j <= 10; j++) {

        sum2 += j \* j;

    }

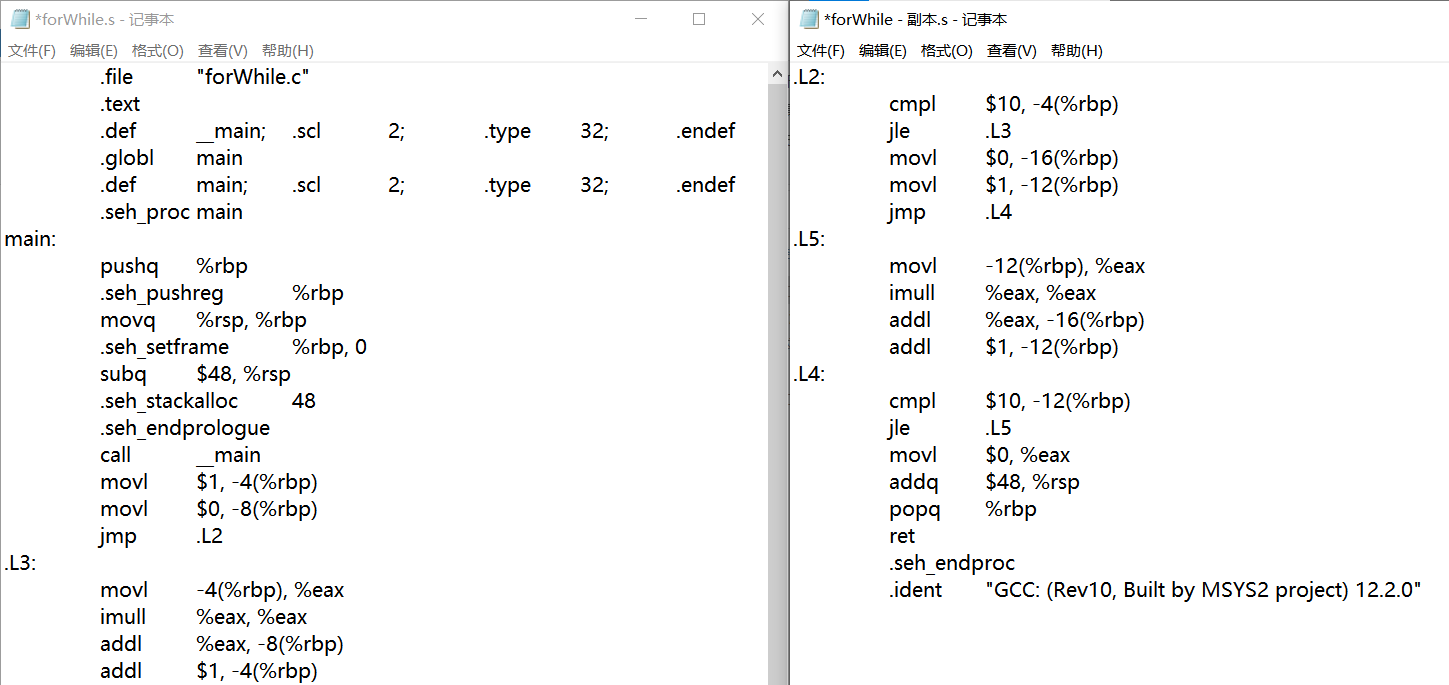
    return 0;

}

以上程序含有while和for循环的C程序，可以求出1~10的平方和。

6.3汇编语言的生成及分析

代码编写完成后，可以通过$文件gcc -E -o forWhile.i forWhile.c和$ gcc -S -o forWhile.s forWhile.i即可生成汇编代码forWhile.s。



根据上述汇编代码，接下来我们对其进行详细分析：

（1）pushq %rbp：在汇编语言中，程序执行采用栈式结构，所以需要先将基址指针寄存器rbp的值压入栈中，保存函数调用过程中的“现场”，以便函数返回时恢复上层函数的现场。此指令的作用就是将函数调用现场压入栈中。

（2）movq %rsp, %rbp：此指令将栈指针寄存器rsp的值赋给基址指针寄存器rbp，以便指向当前栈帧的底部。此时，rbp指向当前函数在栈中的基地址。

（3）subq $48, %rsp：分配48字节的栈空间，称为局部变量空间。这个指令将栈指针向下移动，并分配适当的字节以存储局部变量和函数调用参数。

（4）call \_\_main：调用系统初始化函数\_\_main，对程序进行初始化器操作。

在接下来的指令中，对应每一行C代码做如下解释：

（1）movl $1, -4(%rbp)：将十进制值1存储在地址偏移量为-4(%rbp)的内存单元中，初始化while循环的变量i。

（2）movl $0, -8(%rbp)：将十进制值0存储在地址偏移量为-8(%rbp)的内存单元中，初始化while循环的变量sum。

（3）jmp .L2：无条件跳转到标签.L2处，准备开始while循环。

（4）movl -4(%rbp), %eax：将地址偏移量为-4(%rbp)的内存单元中的值，也就是i，读入累加器eax中，用来计算当前遍历数的平方。

（5）imull %eax, %eax：将eax中的值乘以自身，并将结果存回eax中。这条指令可以得到i的平方。

（6）addl %eax, -8(%rbp)：将eax中存储的数值加到地址偏移量为-8(%rbp)的内存单元中，也就是sum中，此时sum存储的就是每个循环迭代中i的平方的累加和。

（7）addl $1, -4(%rbp)：将地址偏移量为-4(%rbp)的内存单元中值（也就是i），增加一个单位（即加一），同时将结果保存到回原内存地址中，更新变量i。

（8）cmpl $10, -4(%rbp)：将地址偏移量为-4(%rbp)的内存单元中值（即i），与数字10进行比较，同时将结果保存在标志寄存器中，以便下一次处理，判断是否需要再次循环。

（9）jle .L3：如果标志寄存器表明i<=10，则跳转到标签.L3处，继续while循环。

（10）movl $0, -16(%rbp)：将十进制值0存储在地址偏移量为-16(%rbp)的内存单元中，初始化for循环的变量sum2。

（11）movl $1, -12(%rbp)：将十进制值1存储在地址偏移量为-12(%rbp)的内存单元中，初始化for循环的变量j。

（12）jmp .L4：无条件跳转到标签.L4处，准备开始for循环。

（13）movl -12(%rbp), %eax：将地址偏移量为-12(%rbp)的内存单元中的值，也就是j，读入累加器eax中，用来计算当前遍历数的平方。

（14）imull %eax, %eax：将eax中的值乘以自身，并将结果存回eax中。这条指令可以得到j的平方。

（15）addl %eax, -16(%rbp)：将eax中存储的数值加到地址偏移量为-16(%rbp)的内存单元中，也就是sum2中，此时sum2存储的就是每个循环迭代中j的平方的累加和。

（16）addl $1, -12(%rbp)：将地址偏移量为-12(%rbp)的内存单元中值（也就是j），增加一个单位（即加一），同时将结果保存到回原内存地址中，更新变量j。

（17）cmpl $10, -12(%rbp)：将地址偏移量为-12(%rbp)的内存单元中值（即j），与数字10进行比较，同时将结果保存在标志寄存器中，以便下一次处理，判断是否需要再次循环。

（18）jle .L5：如果标志寄存器表明j<=10，则跳转到标签.L5处，继续for循环。

（19）movl $0, %eax：将十进制值0存储在累加器eax中，表示本函数的执行结果为0。

（20）addq $48, %rsp：释放48字节的栈空间，恢复栈指针。

（21）popq %rbp：弹出基地址指针，恢复上层函数过程的调用现场。

（22）ret：从该函数中返回。

编译器将高级语言（如C语言）翻译成汇编语言的过程叫做语义翻译。在语义翻译中，编译器会按照语法规则将C语言代码分析成各种语法成分，包括关键词、变量、函数等，并逐一进行语义检查。如果出现错误，编译器会提示错误信息并中止编译过程。如果没有错误，则编译器将C语言代码转换成中间代码。接着，编译器将中间代码转换成目标汇编代码，也就是计算机可以直接执行的代码。

在刚刚所写的程序中，编译器会先将程序中的变量i、sum、sum2、j等转换成相应的内存地址偏移量，在汇编代码中使用基址指针进行访问，以实现对变量的操作。编译器还会对C语言中的循环、条件分支、函数调用等进行转换，生成相应的汇编代码。例如，在该示例程序中，while循环被转换为汇编代码的跳转指令，for循环被转换为汇编代码的循环结构。编译器可以通过代码优化等手段，使生成的汇编代码尽可能地简短和高效。这样，计算机可以快速地执行编译后的代码。

总之，编译器是将高级语言代码转换成计算机可以执行的汇编代码的工具，实现了高级语言和底层机器语言的语义转换。

7 课后作业3

7.1作业要求

请编写含有函数调用（要有参数传递，参数个数至少要6个）的 C程序，理解编译器如何实现函数调用的翻译，并通过汇编试图去分析，验证函数的函数调用中的如何参数传递、调用与返回等，请注意一种带优化-O2，一种不带优化。

7.2 程序编写

test7.c:

int add (int a, int b, int c, int d, int e, int f, int g, int h)

{

    return a + b + c + d + e + f + g + h;

}

int main()

{

    int sum = add (1, 2, 3, 4, 5, 6, 7, 8);

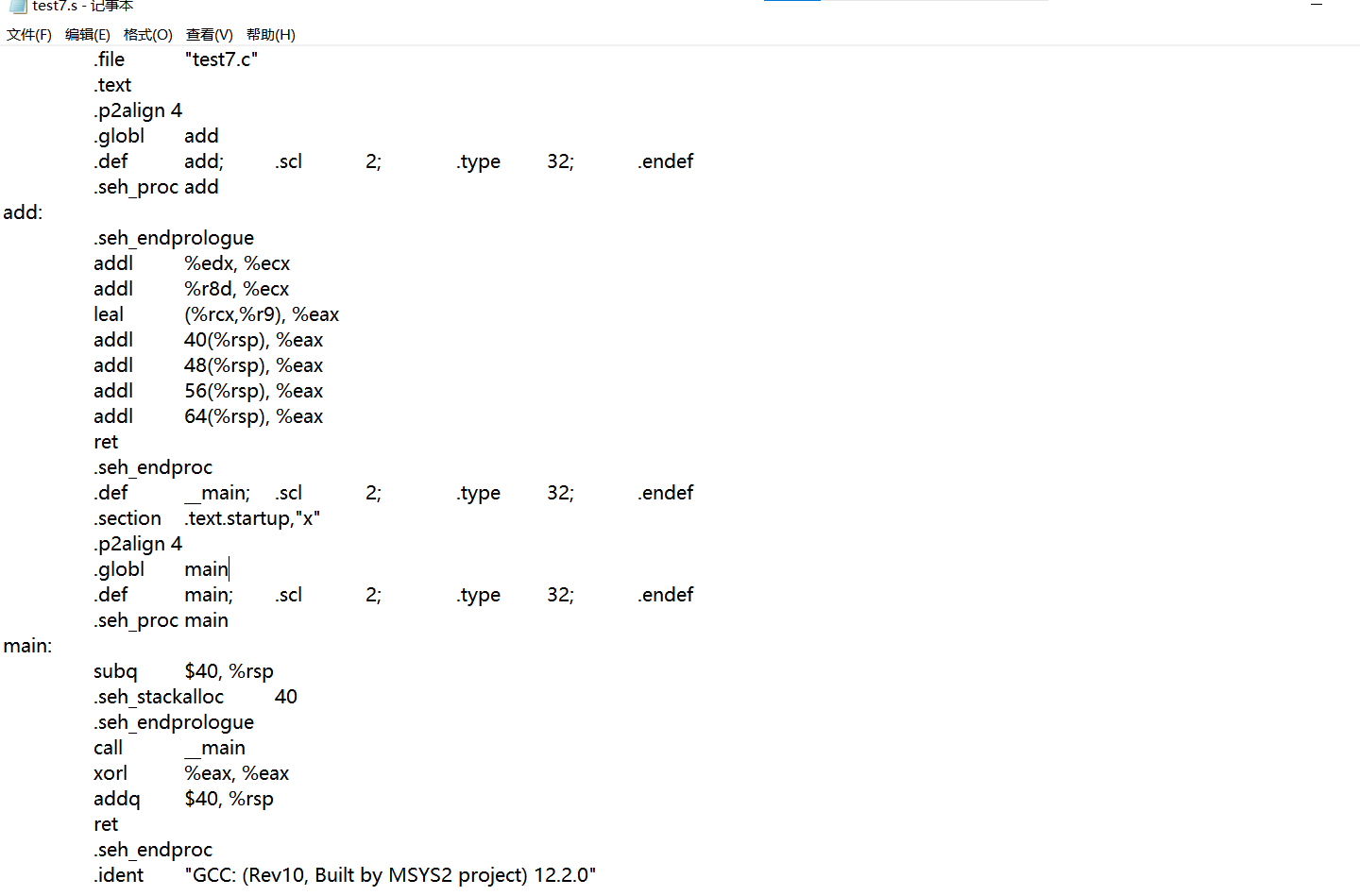
    return 0;

}

上述代码在主函数中实现了对add()函数的调用，同时含8个参数的传递。下面对其进行分析。

7.3 分析（带优化-O2）

可以通过$ gcc -E -o test7.i test7.c和$ gcc -S -O2 -o test7.s test7.i两条命令来实现带-O2优化的汇编。生成的test7.s文件的详细内容如下：



通过上述内容可以看出，编译器在翻译函数调用时，通常将函数参数压入栈中或者存储在寄存器中，然后使用默认协定(cdecl)调用函数。在add函数中，前四个参数分别存储在寄存器$ECX$、$EDX$、$R8D$和$R9$中，后四个参数则被依次压入栈中。

接着我们来分析一下add函数的调用过程。在main函数内部，将1到8八个整数作为参数依次压入栈中（被依次存储在栈空间的偏移量$8$至$32$中），随后使用call指令调用add函数，将add函数的地址压入栈中并跳转到此地址，此时函数调用的参数已经准备完毕。在add函数执行完毕后，使用ret指令返回到main函数，并将add函数的返回值$36$存储在$EAX$寄存器中，随后将 $EAX$寄存器的值赋给变量sum。

下面是 main 函数的汇编代码：

main:

subq $40, %rsp

call \_\_main

xorl %eax, %eax

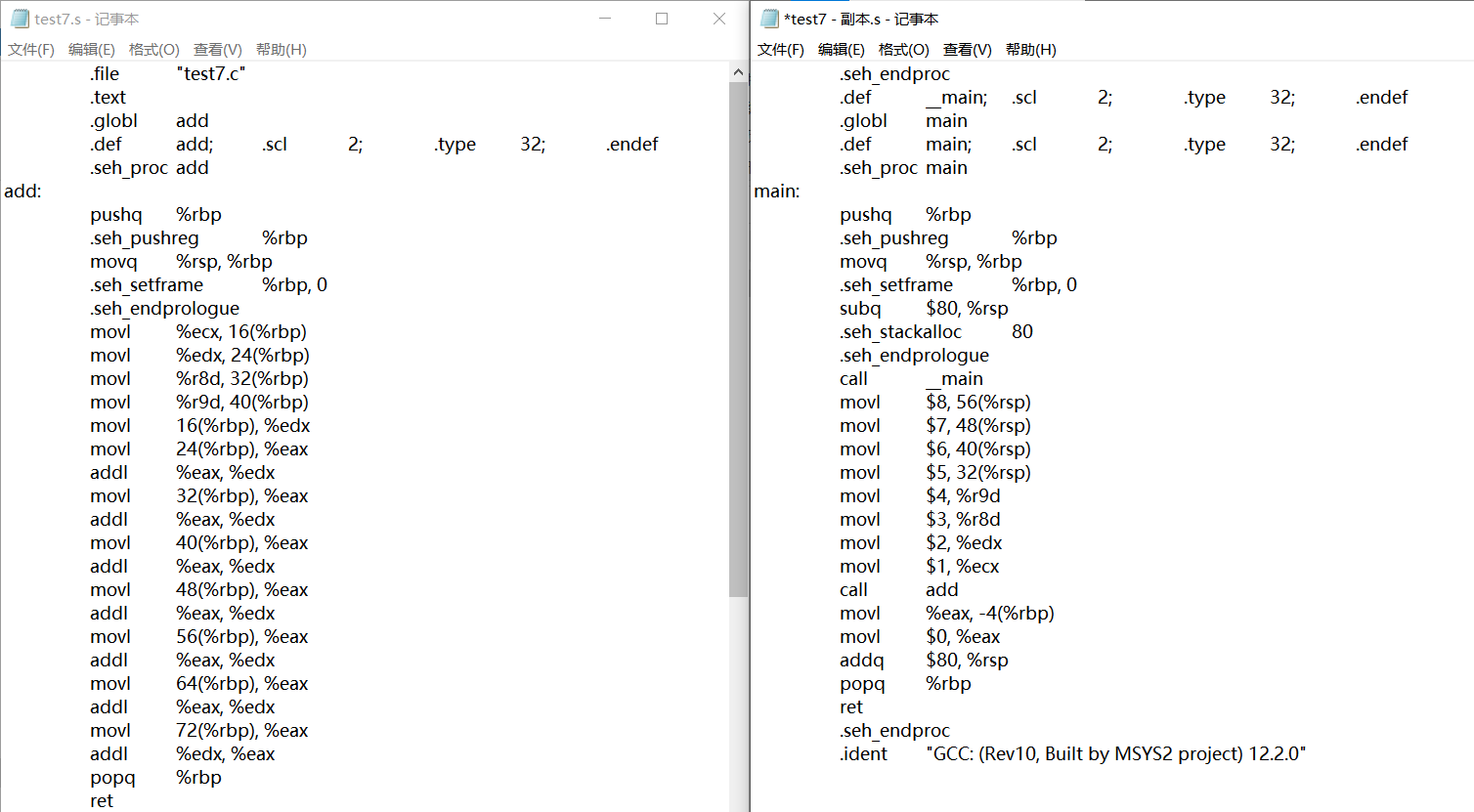
addq $40, %rsp

ret

在这段代码中，首先通过sub指令申请了$40$字节的栈空间，接着调用了C++ 运行时库中的\_\_main函数进行初始化。在add函数执行完毕后，使用ret指令返回到main函数，跳转到main函数指令执行的下一条指令，即将0存储在 $EAX$寄存器中。随后通过addq指令将栈指针调整回原来的位置，最后使用ret指令返回函数执行结果，将$EAX$寄存器中的值作为返回值。

7.4 分析（不带优化）

可以通过$ gcc -E -o test7.i test7.c和$ gcc -S -o test7.s test7.i两条命令来实现不带优化的汇编。生成的test7.s文件的详细内容如下：



很明显，我们可以直观地看出，不含优化的汇编结果比含-O2优化的汇编结果长了很多。相较于不含优化的汇编代码，含-O2优化的汇编代码使用了更多的寄存器，减少了栈空间的使用，并且通过重排指令和删除一些冗余代码来减少程序运行时间。

在add函数中，优化后的代码使用了$ECX, EDX, R8D, R9D$四个寄存器分别存储参数，将栈空间占用减少到了$0$，并且使用leal指令将$ECX+R8D$的值存在$EAX$中，而不是先将$ECX$加到$R8D$上再赋值。另外，虽然优化后的代码有对应的SEH指令，但是SEH指令的作用实际上并不影响程序执行结果，因此在这里可以忽略。

在main函数中，优化后的代码通过使用mov指令和寄存器操作来将参数传递给add函数，避免了将参数保存到堆栈中并使用mov指令传递参数的过程。这样可以减少栈操作，提高函数调用的效率。除此之外，优化后的代码还通过重排代码、删除一些无用指令，以及使用条件码来进行一些优化，来进一步减少程序的运行时间。

8 课后作业4

8.1作业要求

对于全局变量（含初始化变量和未初始化变量），通过汇编分析编译器如何翻译?对于函数的形式参数、局部变量，编译器是如何栈内动态局部变量或形参?（提示:全局变量或静态变量保存在静态存储空间内，函数内的变量一般保存在栈动态存储空间内，malloc分配的内存在堆动态存储中间中。）

8.2 程序编写

test8.c：

int global\_var\_init = 10;   // 已初始化的全局变量

int global\_var\_uninit;      // 未初始化的全局变量

int add (int a, int b)

{

    return a + b;

}

int main()

{

    int sum = add (global\_var\_init, global\_var\_uninit);

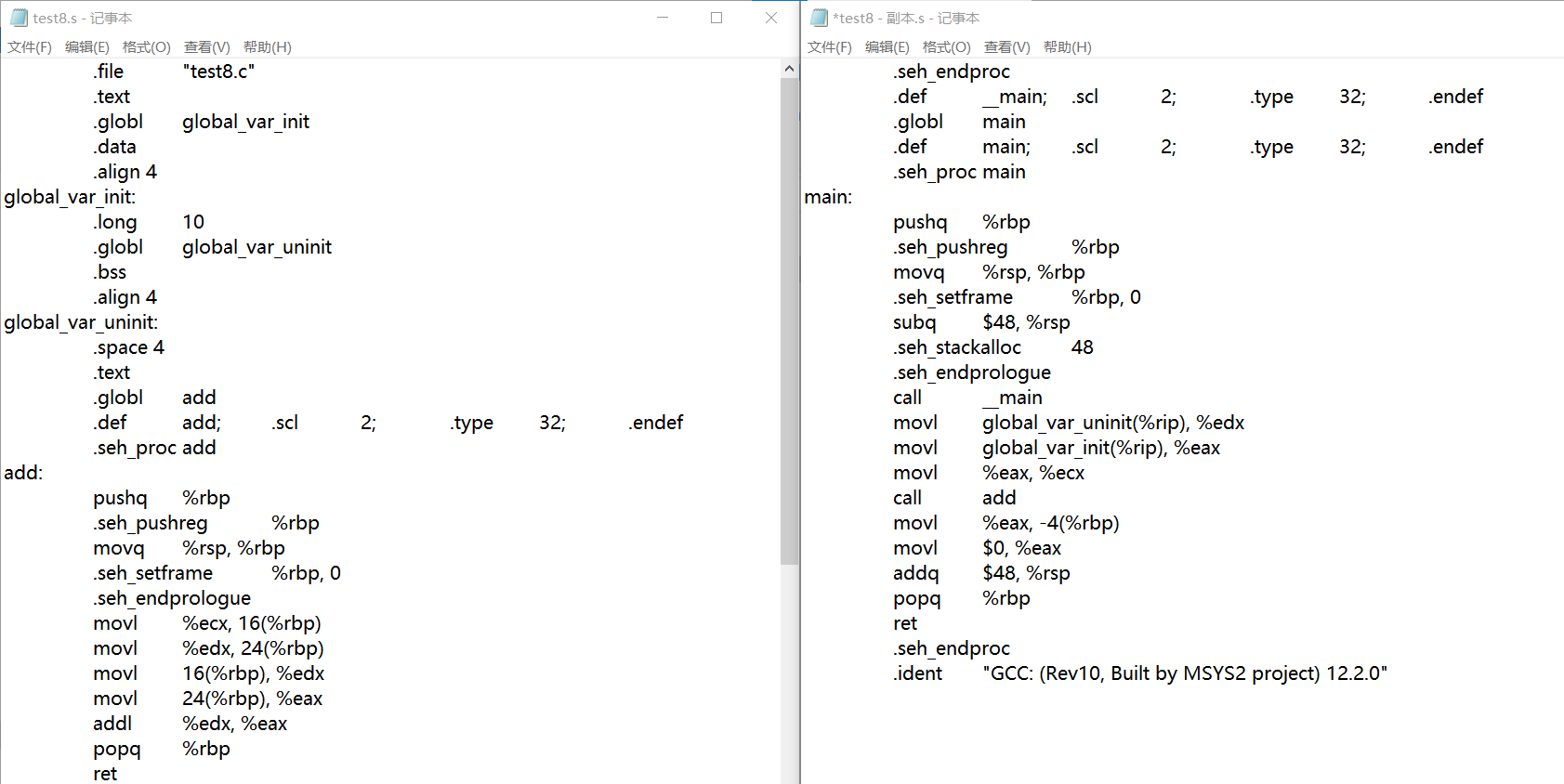
    return 0;

}

从上述代码可以看出，程序中定义两个全局变量，一个已初始化全局变量global\_var\_init，一个未初始化全局变量global\_var\_uninit，add()函数有两个整型形参a和b，在主函数中使用局部变量sum接收add()函数的返回值。

8.3 分析

使用$ gcc -E -o test8.i test8.c和$ gcc -S -o test8.s test8.i两条命令生成test8.s汇编文件，其详细内容如下面的截图所示：



根据上述汇编代码和源代码，可以分析得到以下结论：

（1）对于已初始化的全局变量global\_var\_init，编译器将其定义在程序的.data段中，并使用.long指令将值初始化为10。

（2）对于未初始化的全局变量global\_var\_uninit，编译器将其定义在程序的 .bss段中，并使用.space指令分配4个字节的存储空间。

（3）在函数add的实现中，编译器使用pushq和popq指令对函数栈帧进行操作，通过movl指令将形式参数a和b的值分别移动到函数栈帧的16(%rbp)和24(%rbp)位置上，然后使用addl指令对它们进行相加，将结果存储到EAX寄存器，并通过ret指令返回结果。

（4）在函数main中，编译器使用了一系列指令来设置函数栈帧、调用\_\_main函数、传递形式参数、调用add函数，并将结果存储在栈帧的-4(%rbp)位置上，最后清理函数栈并使用ret指令将0作为函数的返回值。

（5）对于函数的形式参数和局部变量，编译器在函数栈帧中分配内存空间，并使用相应的指令来访问它们。在本例中，形式参数被移动到函数栈帧的16(%rbp)和24(%rbp)位置上，局部变量被分配在栈帧上。需要注意的是，在本例中并没有使用动态分配内存来创建动态数组等数据结构，因此我们无法观察到编译器和操作系统支持的堆分配内存的情况。

9 课后作业5

9.1作业要求

使用make或者cmake实现对多文件项目（至少4个源文件、源文件长度要尽可能长）的编译和链接，并验证加速编译。若在MINGW环境下不明显，请在Linux系统上进行。建议使用cmake进行。（cmake-S源代码位置-B cmake-build-debug、cmake --build cmake-build-debug --parallel）

9.2 程序编写

上述“5课后作业1”中的程序满足本作业要求，此处可沿用该程序。

9.3 验证结果（make命令）

在linux系统中，若要用make命令要实现一次性对多文件项目的编译和链接。需要编写makefile文件，本次makefile文件如下：

test: test.o add.o factorial.o fibonacci.o

    gcc -o test test.o add.o factorial.o fibonacci.o

test.o: test.s

    gcc -c -o test.o test.s

add.o: add.s

    gcc -c -o add.o add.s

factorial.o: factorial.s

    gcc -c -o factorial.o factorial.s

fibonacci.o: fibonacci.s

    gcc -c -o fibonacci.o fibonacci.s

test.s: test.i

    gcc -S -o test.s test.i

add.s: add.i

    gcc -S -o add.s add.i

factorial.s: factorial.i

    gcc -S -o factorial.s factorial.i

fibonacci.s: fibonacci.i

    gcc -S -o fibonacci.s fibonacci.i

test.i: test.c

    gcc -E -o test.i test.c

add.i: add.c

    gcc -E -o add.i add.c

factorial.i: factorial.c

    gcc -E -o factorial.i factorial.c

fibonacci.i: fibonacci.c

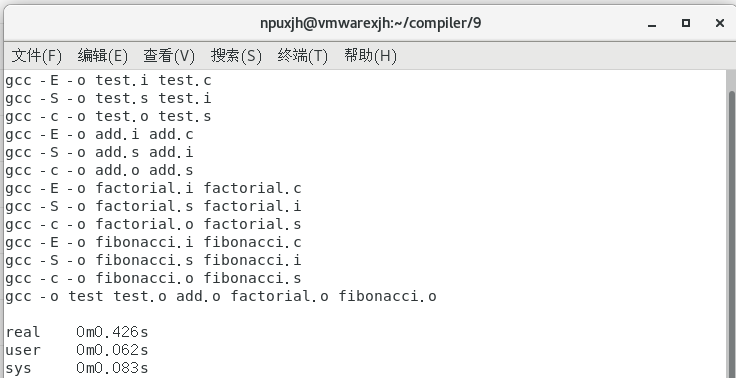
    gcc -E -o fibonacci.i fibonacci.c

clean:

    rm \*.o \*.i \*.s test

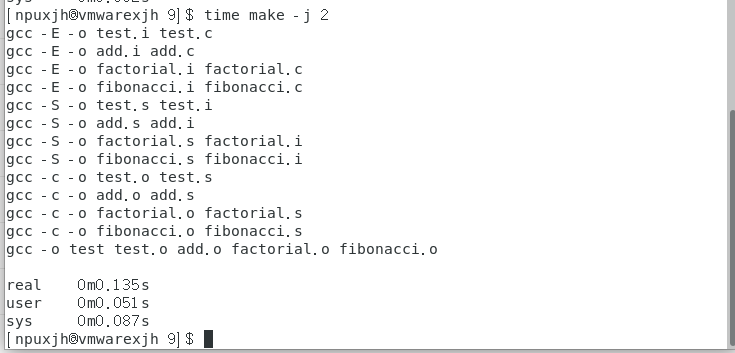
可通过在通过命令行中，使用make来编译并链接源文件，例如：time make。要验证加速编译，可以通过使用time make -j命令来利用多线程进行并行编译，在编译时使用多个线程，例如：time make -j 3。

不加速：



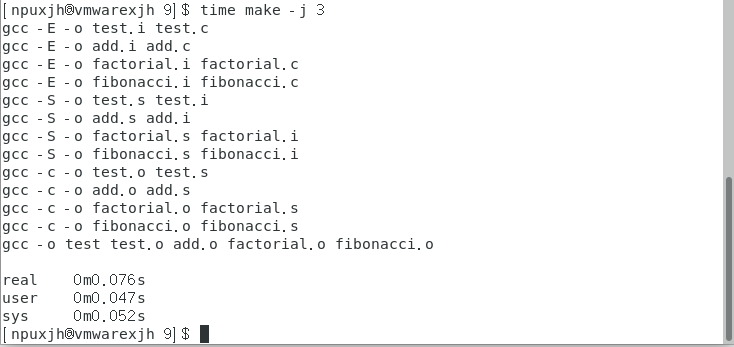
不加速编译整个过程用时0.426s。

2个线程同时进行加速编译：



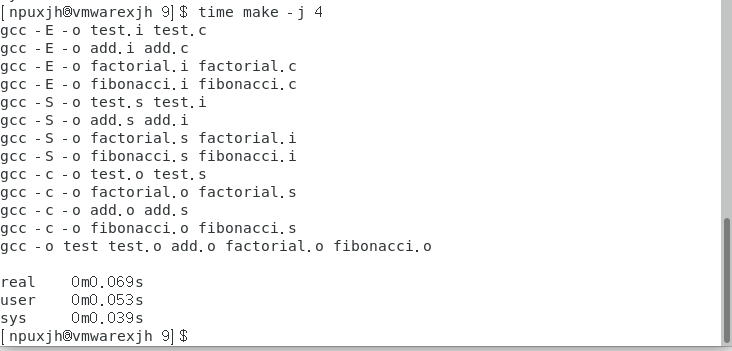
2个线程加速编译整个过程用时0.135s。

3个线程同时进行加速编译：



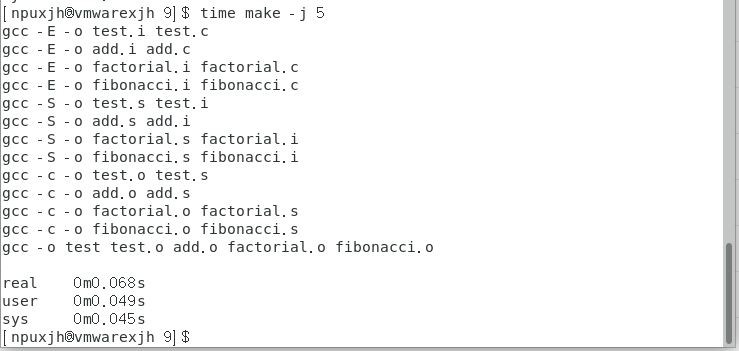
3个线程加速编译整个过程用时0.076s。

4个线程同时进行加速编译：



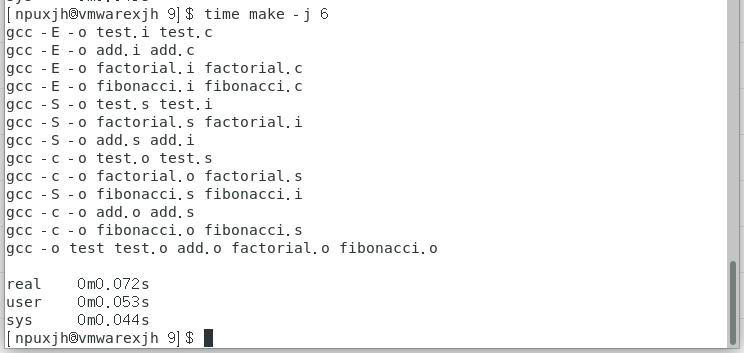
4个线程加速编译整个过程用时0.069s。

5个线程同时进行加速编译：

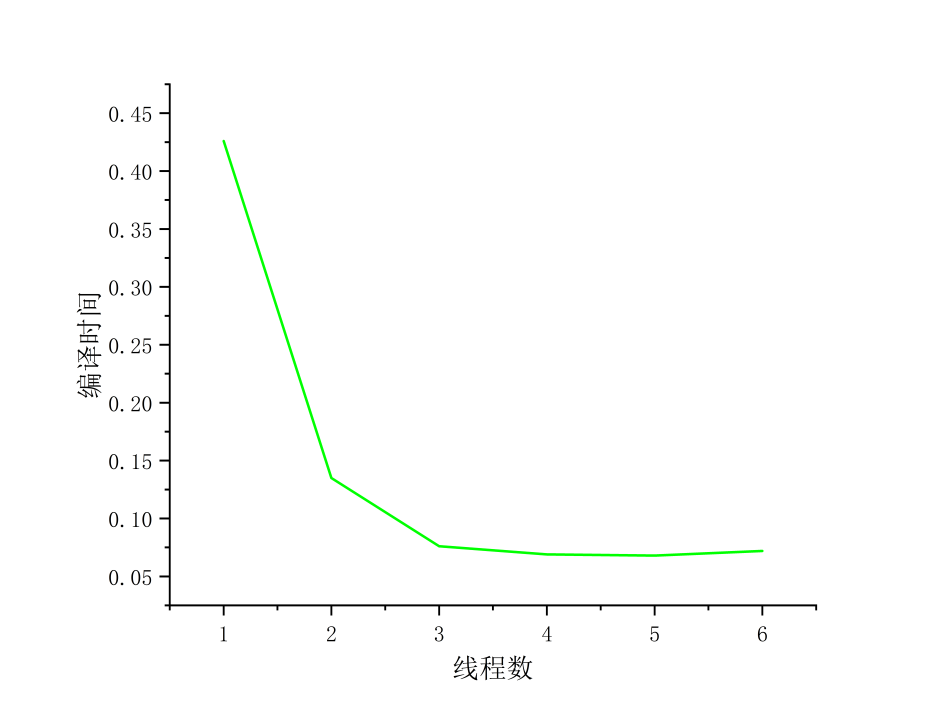


5个线程加速编译整个过程用时0.068s。

6个线程同时进行加速编译：



5个线程加速编译整个过程用时0.072s。



将上述结果总结到下面表格中：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 线程数 | 不加速（1） | 2 | 3 | 4 | 5 | 6 |
| 时间 | 0.426 | 0.135 | 0.076 | 0.069 | 0.068 | 0.072 |

无论是从曲线图还是表格，我们发现使用分步编译后，使用更多的线程同时编译可以明显地加快编译速度，当然，当线程数达到一定的值时，便无法再加速了。

9.4 验证结果（cmake命令）

在windows的MinGW环境下，可以使用cmake命令进行编译，在使用make命令之前，需要按规定格式编写一个CMakeLists.txt文件。

CMakeLists.txt：

cmake\_minimum\_required(VERSION 3.26)

set(CMAKE\_C\_COMPILER "gcc")

set(CMAKE\_CXX\_COMPILER "g++")

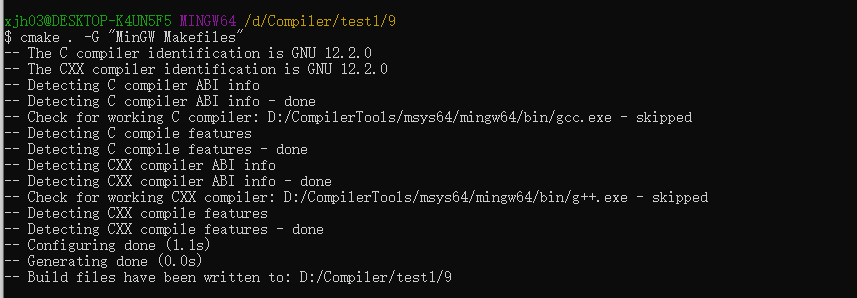
# set the project name

project(test)

# add the executable

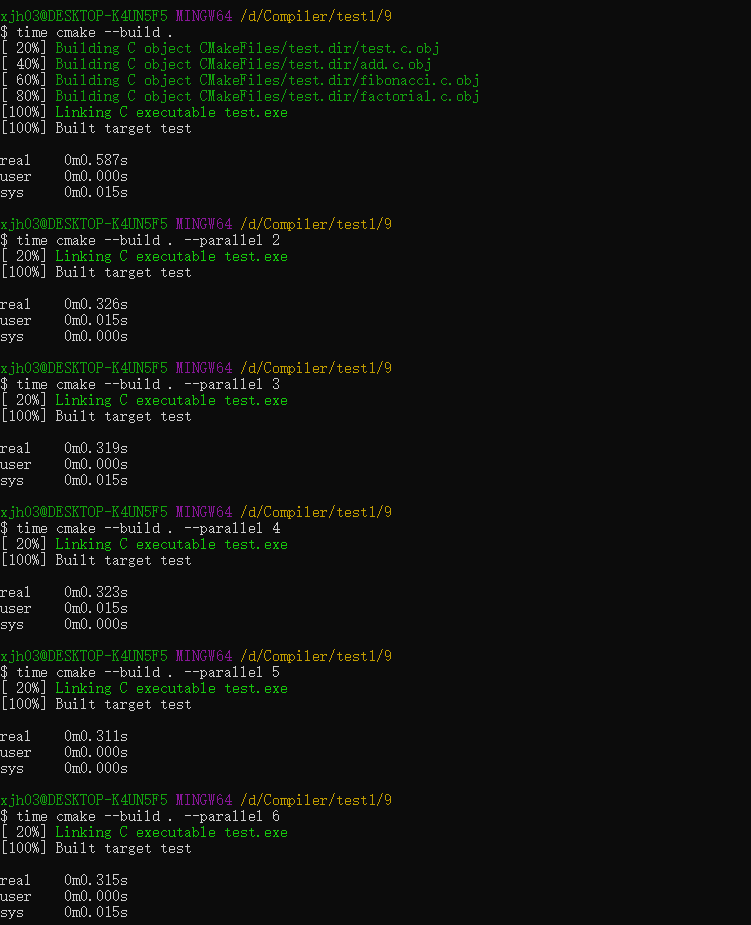
add\_executable(test test.c add.c fibonacci.c factorial.c)

然后在终端进入到源文件所在的目录下，输入命令cmake . -G “MinGW Makefiles”



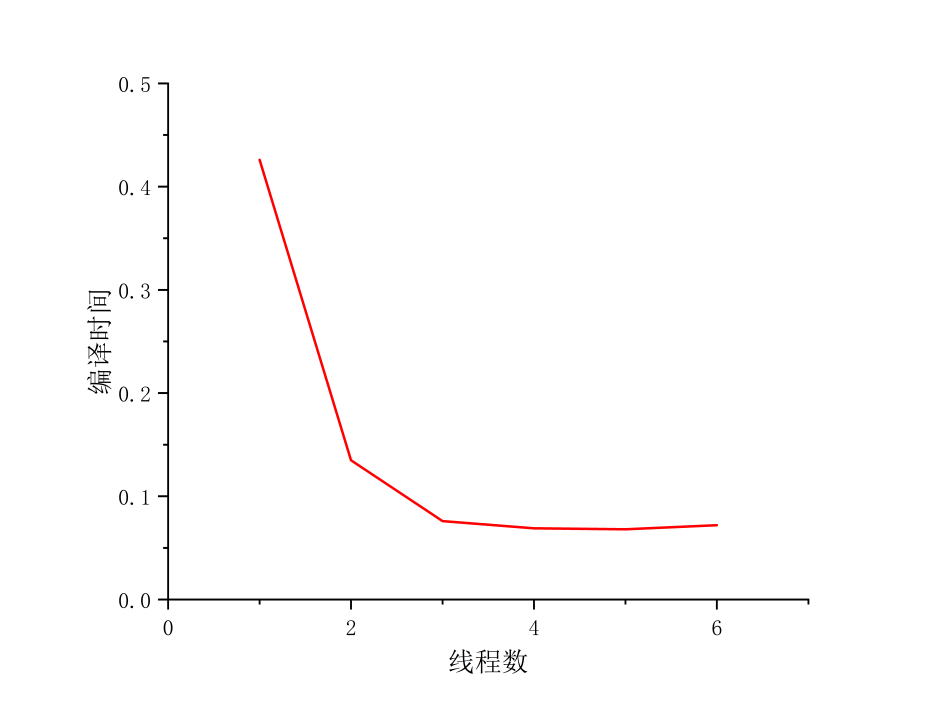


执行完成后，会生成上图中我框出的文件，然后就可以输入命令$ time cmake --build .实现不加速编译并查看编译时间，随后可以使用命令$ time cmake --build . --parallel 2实现双线程加速编译。



将运行结果总结到下面表格中，不加速编译用时0.426s，双线程加速编译用时0.135s，3线程加速编译用时0.076s，4线程加速编译用时0.069s，5线程加速编译用时0.068s，6线程加速编译用时0.072s。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 线程数 | 不加速（1） | 2 | 3 | 4 | 5 | 6 |
| 时间 | 0.426 | 0.135 | 0.076 | 0.069 | 0.068 | 0.072 |



无论是从曲线图还是表格，我们发现使用cmake编译命令时，也使用更多的线程同时编译可以明显地加快编译速度，当然与make命令一样的是，当线程数达到一定的值时，便无法再加速了。

10 实验总结

通过本次实验，我对gcc编译器的工作流程和命令有了一定的了解。同时也熟悉了编译器的工作，以及分布编译过程中的一些细节。同时，我对x86汇编指令的阅读能力也有所提升，能了解C语言程序与x86汇编程序的对应关系。也学习了clang编译命令的一些用法，并以此为工具分析编译过程。在实验过程中，我还尝试分析了编译过程中中间语言IR、汇编语言与源程序文件的对应分析，同时对C语言的函数调用即返回进行了分析。最后了解了用make和cmake命令编译C语言源程序方法，学会了编写makefile文件和CMakeLists.txt文件，并成功地验证了编译加速。

由于make命令实现编译过程在上学期操作系统实验课上有所提及，故在本次实验中没有难到我，但cmake实现编译的流程我并不是很了解，但查阅了相关资料之后还是顺利地完成了实验。相关链接为:http://t.csdn.cn/OrLM5。