

**杭州电子科技大学**

**《编译原理课程实践》**

**实验报告**

题 目：认识编译器-GCC相关操作练习

学 院： 计算机学院

专 业： 计算机科学与技术

班 级： 22052312

学 号： 22050201

姓 名： 黄江晔

完成日期： 2024.10.24

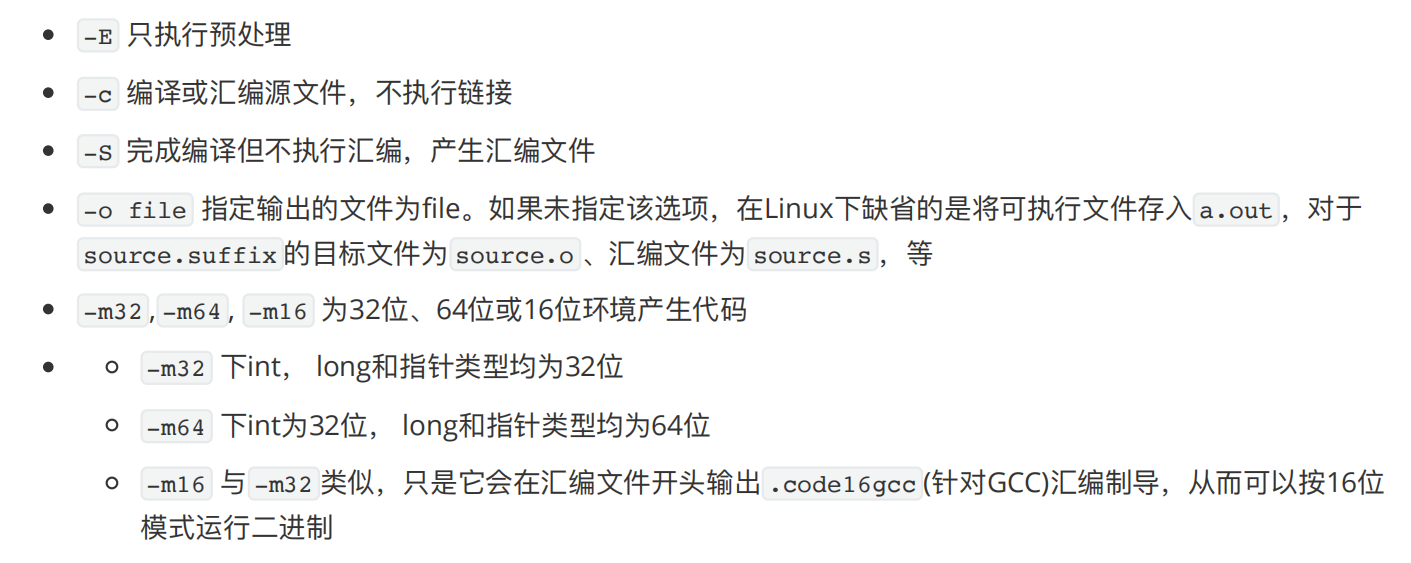
1. **实验目的**

本实验的目的是了解工业界常用的编译器 GCC 和 CLANG，熟悉编译器的安装和使用过程，观察编译器工作过程中生成的中间文件的格式和内容，了解编译器的优化效果，为编译器的学习和构造奠定基础。

1. **实验内容与实验要求**

通过对⼀个简单的 C 程序示例 sample.c ,使⽤不同编译选项进⾏编译，得到程序的不同表示形式，尝试理解这些形式之间的对应关系，进⽽理解编译的主要阶段：预处理、编译、汇编、链接。通过实际操作，回答相关问题，将答案整理在answer.pdf的⽂件中并提交作业⽹站。

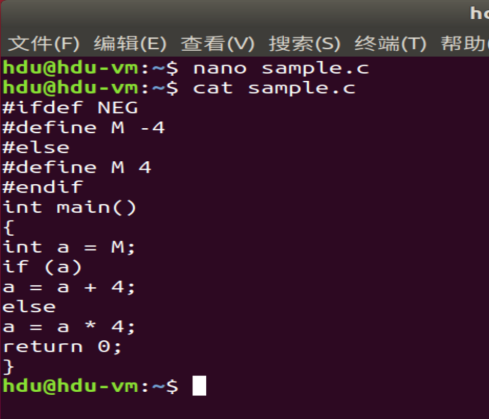
查阅GCC相关教程资料，尝试安装gcc环境，或者直接在⽹络平台https://www.godbolt.org ，GCC部分编译选项摘录如下：



1. **设计方案与算法描述**

**（一）源程序**

sample.c文件内容如下：



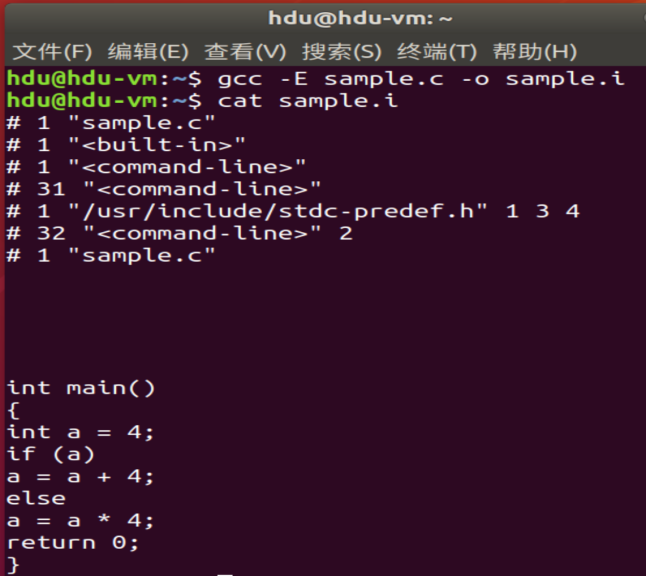
该程序涉及的主要语⾔特征有：

（1）条件编译(1-5⾏)：根据是否定义宏NEG,定义不同的M

（2）宏定义（第2、4⾏）以及宏引⽤（第8⾏）

**（二）预处理**

在命令⾏窗⼝输⼊ gcc -E sample.c -o sample.i ，该命令也等同于cpp sample.c -o sample.i ，将对 sample.c 进⾏预处理，⽣成 sample.i ,其内容如下：

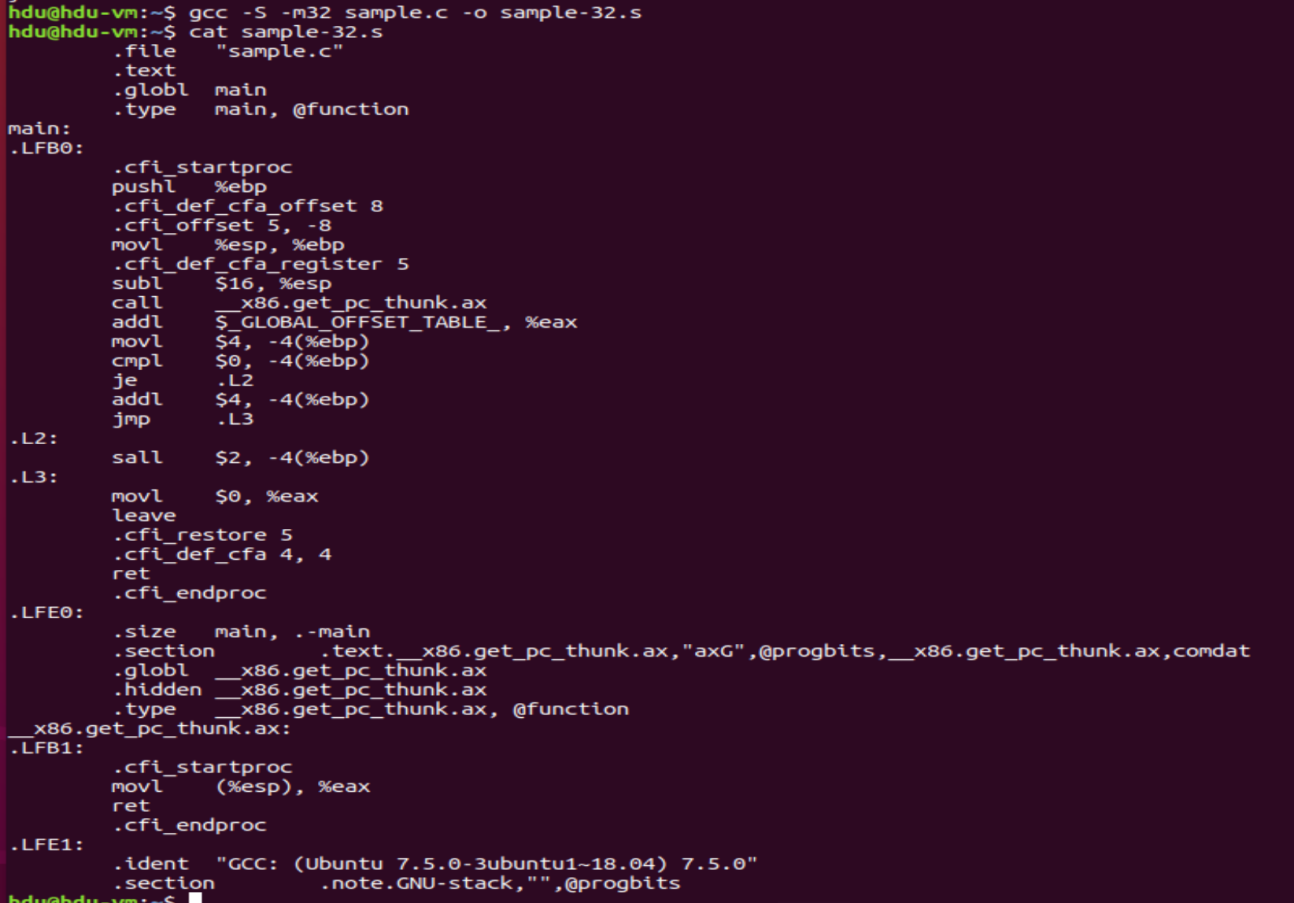


预处理后的程序⽂件没有条件编译了，已经根据没有定义 NEG 选择了 M 定义为4；没有宏定义了，所有的宏引⽤均已经展开，⽐如第14⾏原先对宏 M 的引⽤已展开成4。

**（三）编译得到汇编文件**

（1）32位汇编

gcc -S -m32 sample.c -o sample-32.s

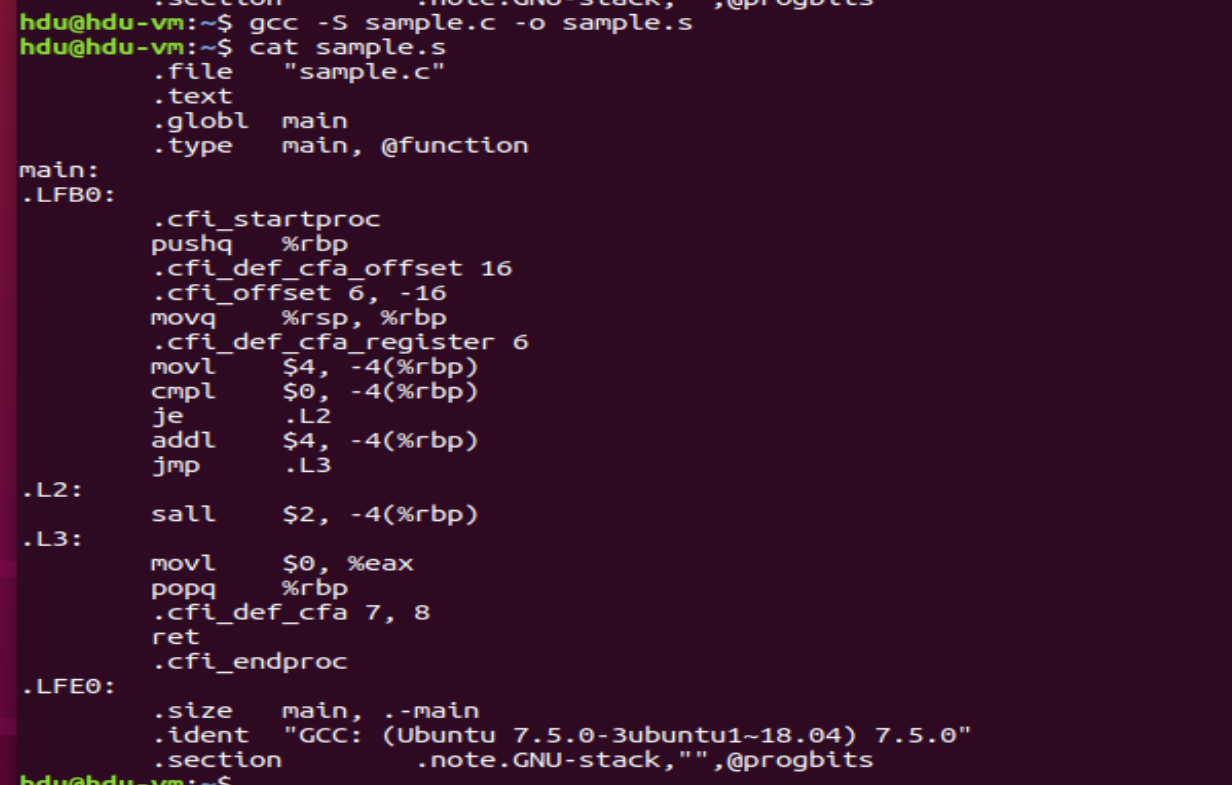


核心代码：



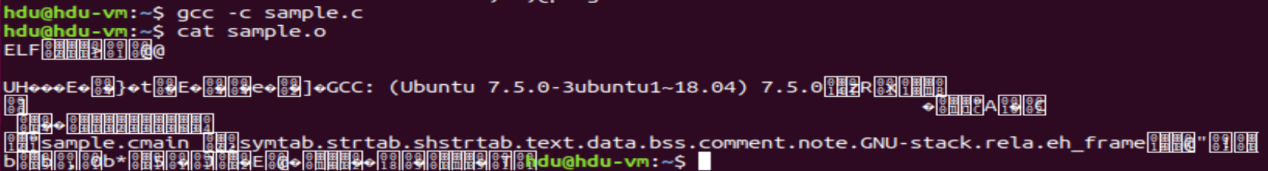
（2）64位汇编

gcc -S sample.c -o sample.s



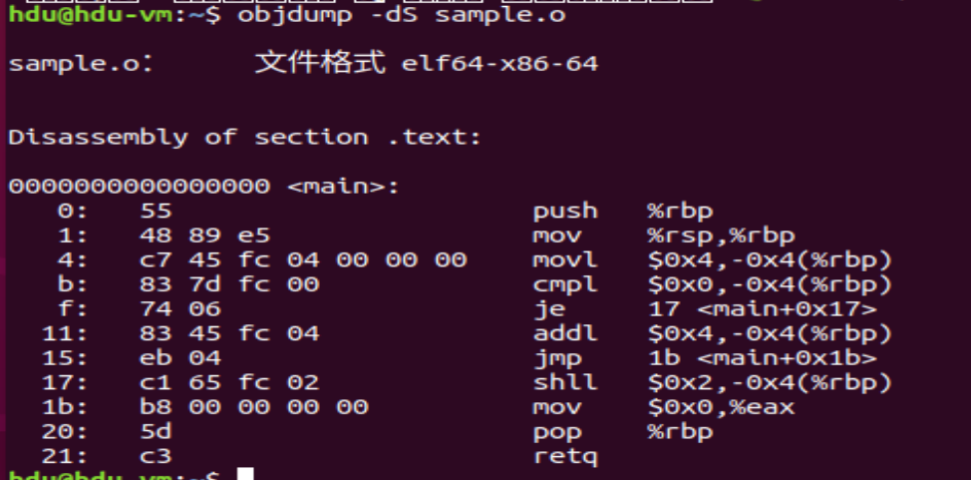
（二）生成目标文件

gcc -c sample.c



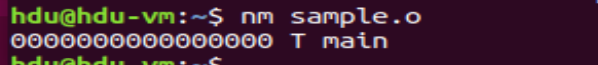
反汇编：

objdump -dS sample.o



全局/外部符号

执⾏ nm sample.o 可以输出该⽬标⽂件的全局符号。



**（四）生成可执行文件**

执行gcc sample.c -o sample可得到可执行文件，由 sample.o 得到可执⾏⽂件是通过调⽤链接器 ld 得到的，但是直接执⾏ ld sample.o -o sample 会产⽣如下警告，主要原因是因为没有链接上需要的crt⽂件。

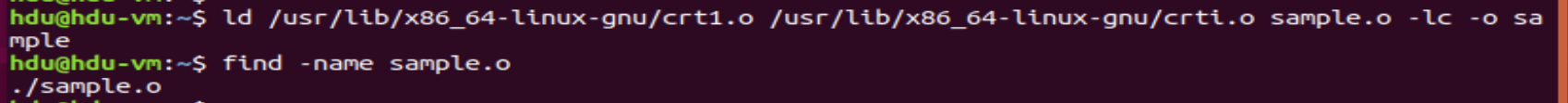
e7ce6690760a0583e41833eb5be17b4

在 /usr/lib/x86\_64-linux-gnu/ 下包含如下⼏个 crt\*.o ⽂件：crt1.o 包含程序的⼊⼝函数 \_start ，它负责调⽤ \_\_libc\_start\_main 初始化 libc 并且调⽤ main 函数进⼊真正的程序主体，crti.o 包含 \_init() 函数，该函数在 main 函数前运⾏，1crtn.o 包含 \_finit() 函数，该函数在 main 函数后运⾏，可以显式地将⽬标⽂件与这些crt⽂件链接，来得到可执⾏⽂件，即执⾏： ld /usr/lib/x86\_64-linux gnu/crt1.o /usr/lib/x86\_64-linux-gnu/crti.o sample.o -lc -o sample 则可以产⽣可执⾏程序，其中 -lc 表示链接C标准库, 其中提供 ：

\_\_libc\_start\_main (main,\_\_libc\_csu\_init,\_\_libc\_csu\_fini)

\_\_libc\_csu\_init （负责调⽤ \_init() ）

\_\_libc\_csu\_fini (负责调⽤ \_finit() )



1. **测试结果**

**问题1-1： 如果在命令行下执行 gcc -DNEG -E sample.c -o sample.i 生成的 sample.i 与之前的有何区别？**



由于 NEG 被定义，预处理器会执行 #ifdef NEG 的分支，并将 M 定义为 -4。

**问题1-2 请对比 sample-32.s 和 sample.s ，找出它们的区别，并上网检索给出产生这些区别的原因。**

（1）寄存器使用：

在32位代码中，使用的是32位寄存器，比如 %ebp 和 %eax。

在64位代码中，使用的是64位寄存器，比如 %rbp 和 %rax。

（2）栈指针操作：

32位代码中，使用 pushl %ebp 和 movl %esp, %ebp 来保存栈帧和移动栈指针。

64位代码中，使用 pushq %rbp 和 movq %rsp, %rbp，因为64位体系下的栈指针和栈基址都是64位寄存器。

（3）地址操作：

32位系统使用 subl $16, %esp 调整栈空间。

64位系统使用 subq $16, %rsp 做类似操作，但针对64位寄存器。

（4）跳转指令和比较：

在32位代码中，比较操作和跳转使用的是 cmpl（32位比较），而在64位代码中，使用的是 cmpq（64位比较）。

（5）返回地址和栈恢复：

在32位代码中，使用 leave 和 ret 指令来返回。

在64位代码中，使用 popq %rbp 和 ret 完成类似的操作，但处理的是64位栈。

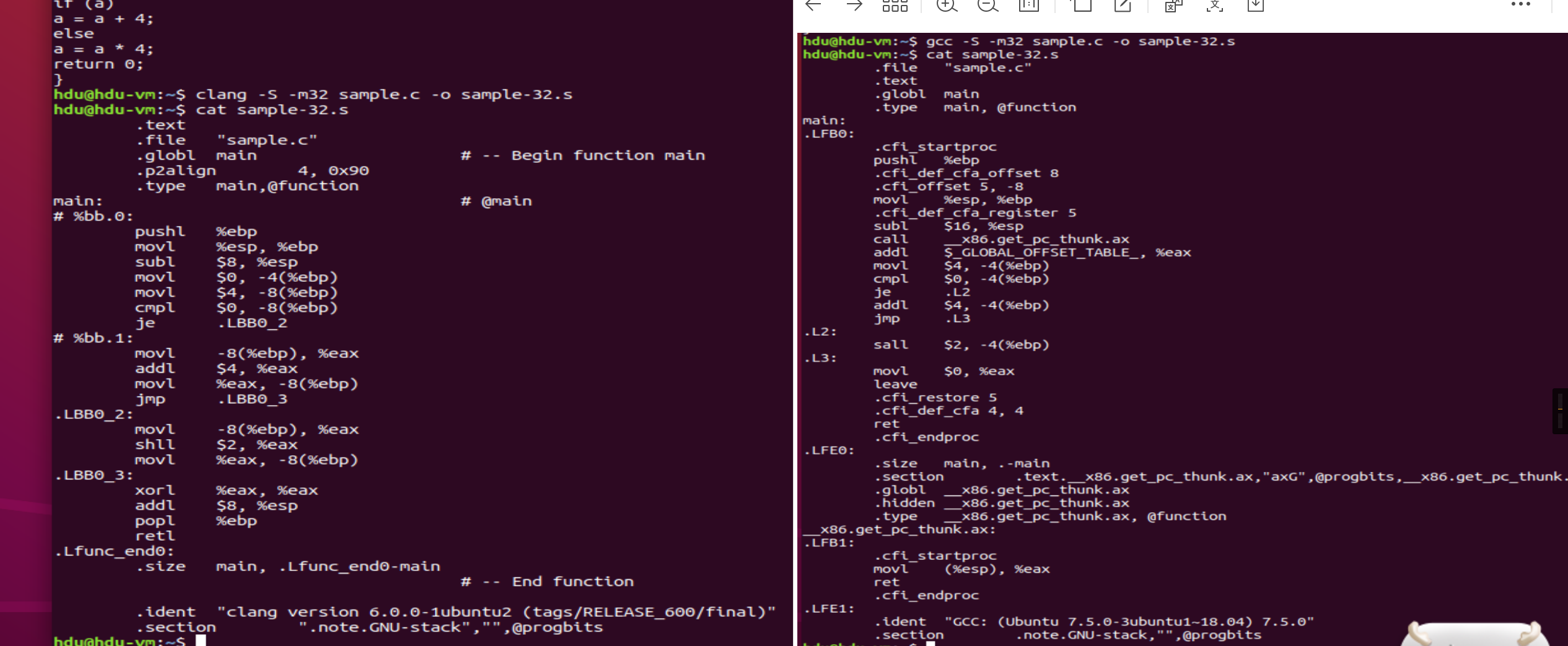
**产生这些区别的原因：**

主要是因为编译目标架构不同。32位程序和64位程序在处理器上操作的寄存器宽度不同（32位和64位），导致生成的汇编代码中使用的寄存器、指令集、栈操作等都有差异。

32位架构（如x86）只能处理32位的寄存器和地址，所以使用 %ebp、%esp 这样的32位寄存器。

64位架构（如x86-64）能够处理64位宽的寄存器，因此使用了 %rbp、%rsp 这样的64位寄存器。

**问题1-3 你可以用clang 替换 gcc ，重复上面的各步，⽐较使⽤ clang 和 gcc 分别输出的结果有何异同。**



clang生成的汇编代码更简洁，带有更多的优化注释，而 gcc 生成的汇编代码则包含更多细节和调试信息。

clang 和 gcc 对于局部变量和临时值的处理有所不同。clang 更频繁地使用寄存器存储临时值，避免不必要的栈操作，而 gcc 会为更多的局部变量分配栈空间。

1. **源代码**

#ifdef NEG

#define M -4

#else

#define M 4

#endif

int main()

{

int a = M;

if (a)

a = a + 4;

else

a = a \* 4;

return 0;

}