**成绩**



**本科生《编译原理》课程实践报告**

**题　　目： 编译器认知实验**

**学 院： 徐特立学院**

**专业名称： 计算机科学与技术**

**姓 名： 陈照欣-1120191086**

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 LLVM，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。

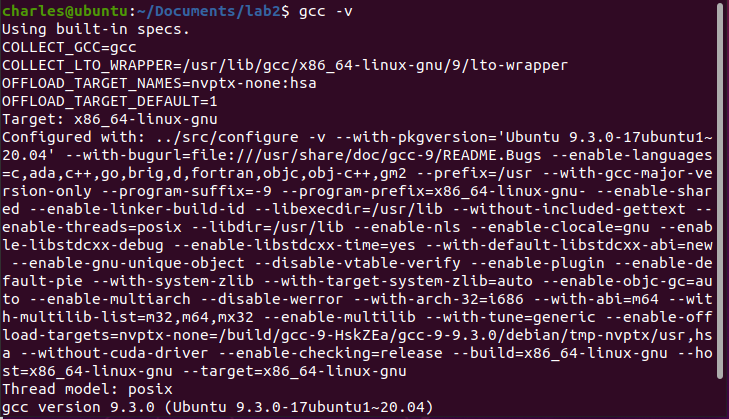
1. 实验内容

本实验主要的内容为在 Linux 平台上安装和运行工业界常用的编译器 GCC 和LLVM，如果系统中没有安装，则需要首先安装编译器，安装完成后编写简单的测试程序，使用编译器编译，并观察中间输出结果。

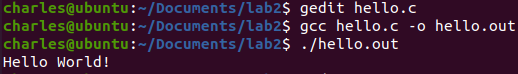
于不同的编译器 (GCC 和 LLVM)，分别完成查看编译器版本内容，程序编译的中间结果，查看汇编代码等任务，编写测试程序，运行编译器并进行观测。分别运行 GCC 和 LLVM 编译器，首先学习编译器的基本使用方法，其次通过编译选项查看分析编译器的中间结果，及其与输入源码之间的对应关系，最后使用-O0、-O1、-O2 和-O3 分别使用两个编译器对输入程序进行优化编译，并对比GCC和LLVM优化后程序的运行效率。

1. 具体实验步骤和操作
   1. GCC
      1. 查看GCC版本

使用命令gcc-V



* + 1. 使用编译器编译单个文件



hello.c代码如下：

#include<stdio.h>

int main()

{

    printf("Hello World!\n");

    return 0;

}

* + 1. 使用编译器链接多个文件

Step1. 建立test.h、test.c、main.c

test.h：

#ifndef test\_h\_\_

#define test\_h\_\_

extern void test(void);

#endif

test.c：

#include<stdio.h>

void test(void)

{

    printf("This is a test library.");

    return;

}

main.c：

#include <stdio.h>

#include "test.h"

int main(void)

{

    puts("This is a shared library test...");

    test();

    return 0;

}

test.h定义了一个接口连接我们的库，一个简单的函数，test()。test.c包含了这个函数的实现，main.c是一个用到我们库的驱动程序

Step2：编译无约束位代码

$ gcc -c -Wall -Werror -fpic test.c

Step3：从一个对象文件创建共享库

$ gcc -shared -o libtest.so test.o

Step4：连接共享库

$ gcc -Wall -o test main.c -ltest

/usr/bin/ld: cannot find -ltest

collect2: error: ld returned 1 exit status

出现报错，GCC有一个默认的搜索列表，但我们的目录并不在那个列表当中。我们需要告诉GCC去哪里找到libfoo.so。这就要用到-L选项。在本例中，我们将使用当前目录/home/charles/Documents/lab2：

$ gcc -L/home/charles/Documents/lab2 -Wall -o Test main.c -ltest

Step5: 运行时使用库

$ ./Test

./Test: error while loading shared libraries: libtest.so: cannot open shared object file: No such file or directory

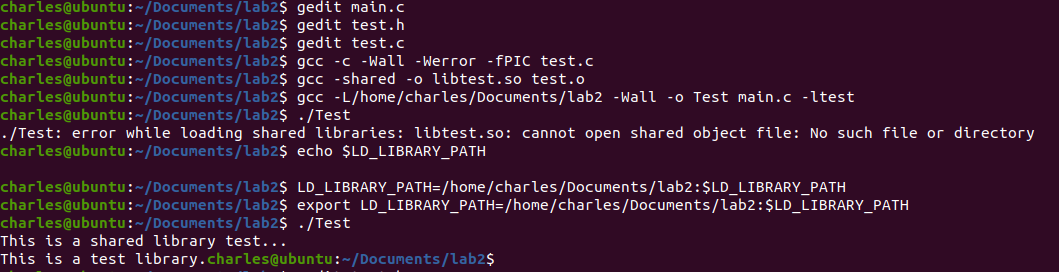
出现报错：加载器不能找到共享库。我们没有将它安装到标准位置，使用环境变量LD\_LIBRARY\_PATH

$ echo $LD\_LIBRARY\_PATH

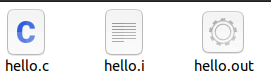
$ LD\_LIBRARY\_PATH=/home/username/foo:$LD\_LIBRARY\_PATH

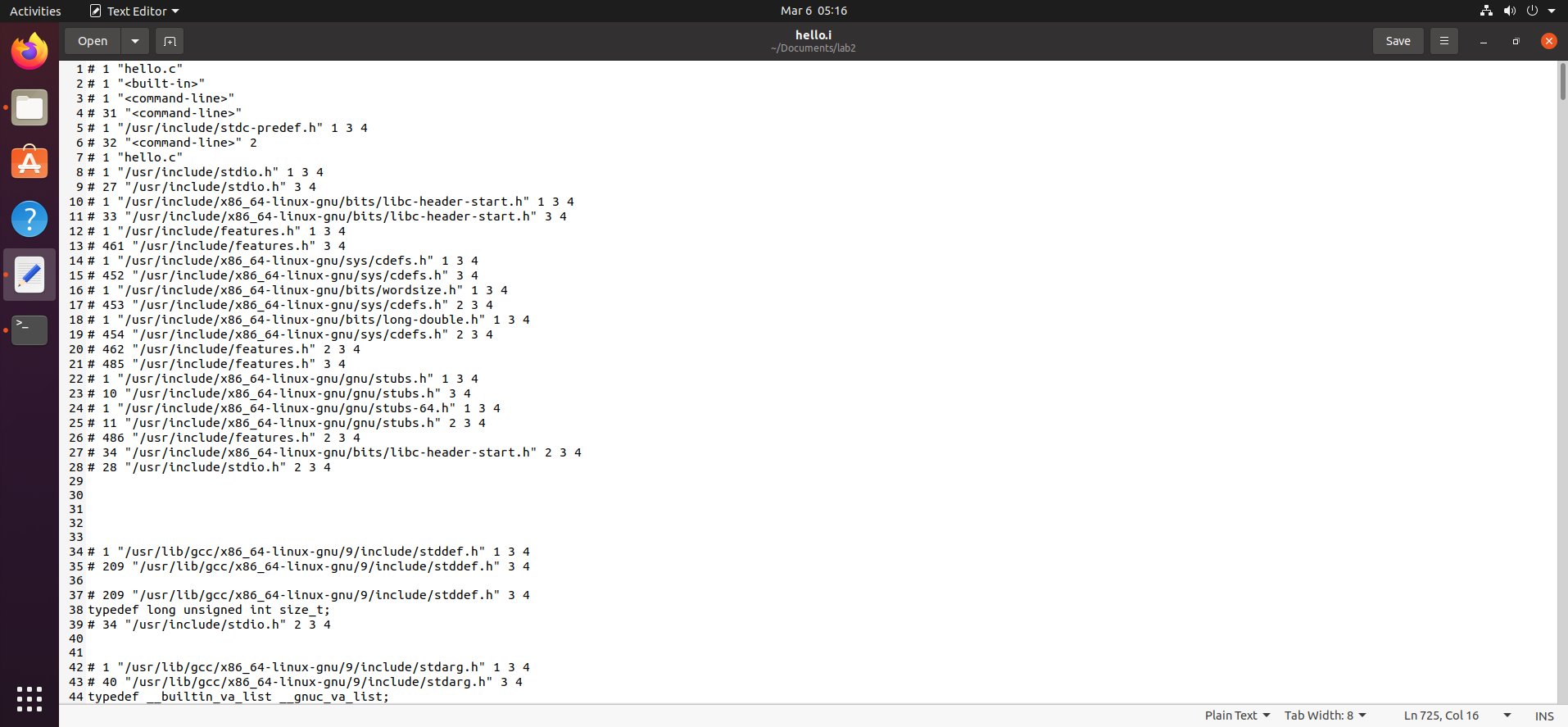
$ export LD\_LIBRARY\_PATH=/home/username/foo:$LD\_LIBRARY\_PATH

$ ./Test



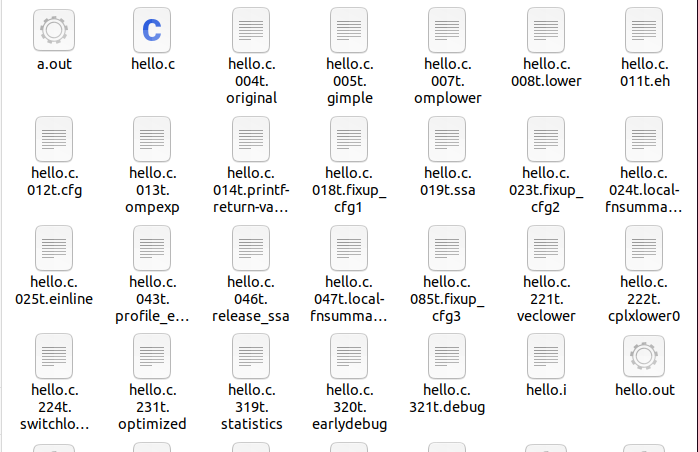
* + 1. 查看预处理结果：gcc -E hello.c -o hello.i



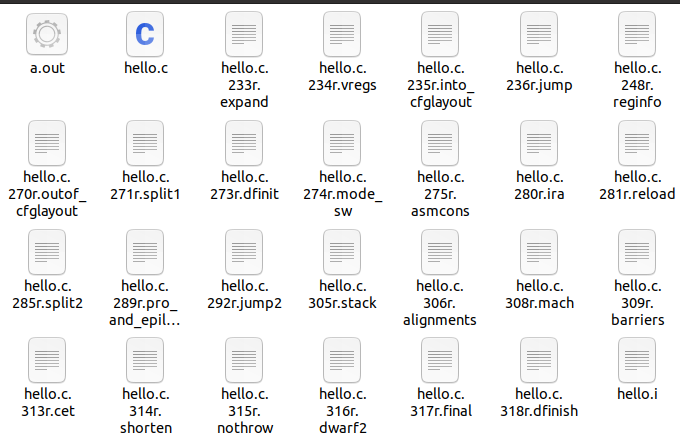
生成hello.i

内容如上

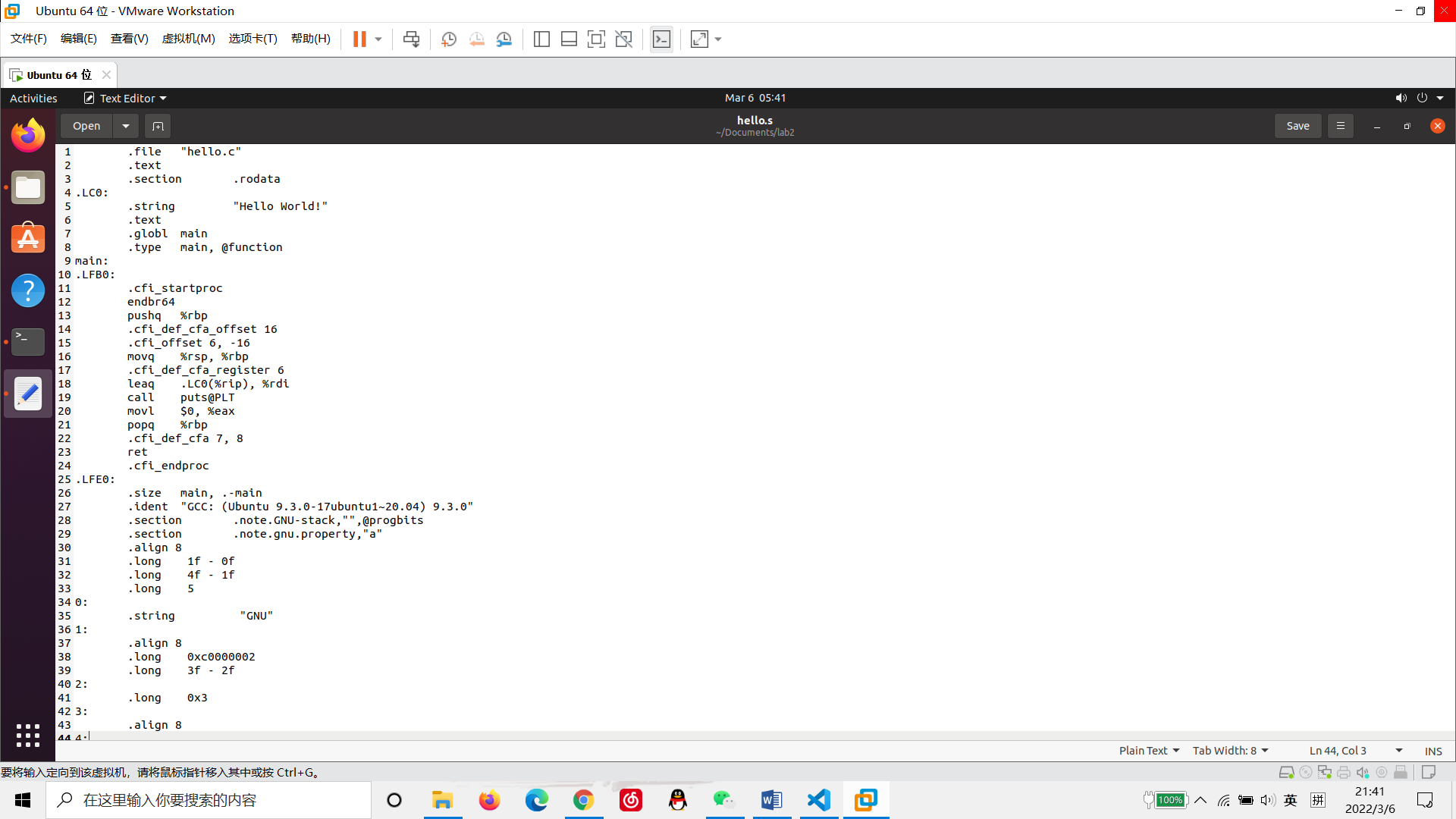
* + 1. 查看语法分析树：gcc -fdump-tree-all hello.c



* + 1. 查看中间代码生成结果：gcc -fdump-rtl-all hello.c



* + 1. 查看生成的目标代码（汇编代码）：gcc –S hello.c –o hello.s

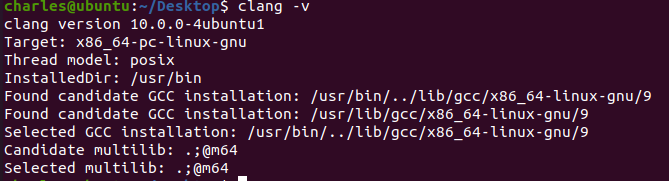


* 1. LLVM
     1. 查看编译器版本

输入命令

clang -v

得到clang版本号：

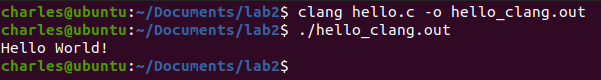


* + 1. 使用编译器编译单个文件

输入命令：

clang hello.c -o hello\_clang.out

得到hello\_clang.out，执行文件得到输出：



* + 1. 使用编译器编译链接多个文件

此处采用与gcc实验中不同的方法实现：

hello.c

#include<stdio.h>

void test();

int main()

{

    printf("Hello World!\n");

    test();

    return 0;

}

test\_clang.c

void test()

{

    printf("Test\n");

    return;

}

输入命令

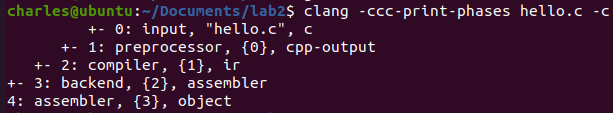
clang -o a.out test\_clang.c hello.c

./a.out

得到输出



* + 1. 查看编译流程和阶段：clang -ccc-print-phases hello.c -c

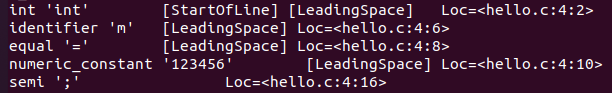


* + 1. 查看词法分析结果：clang hello.c -Xclang -dump-tokens

以如下语句为例：

int m = 123456;

词法分析结果如下：

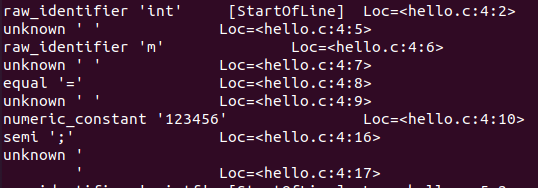


* + 1. 查看词法分析结果2：clang hello.c -Xclang -dump-raw-tokens

以如下语句为例：

int m = 123456;

词法分析结果如下：

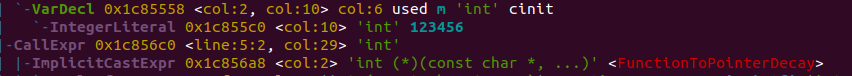


* + 1. 查看语法分析结果：clang hello.c -Xclang -ast-dump

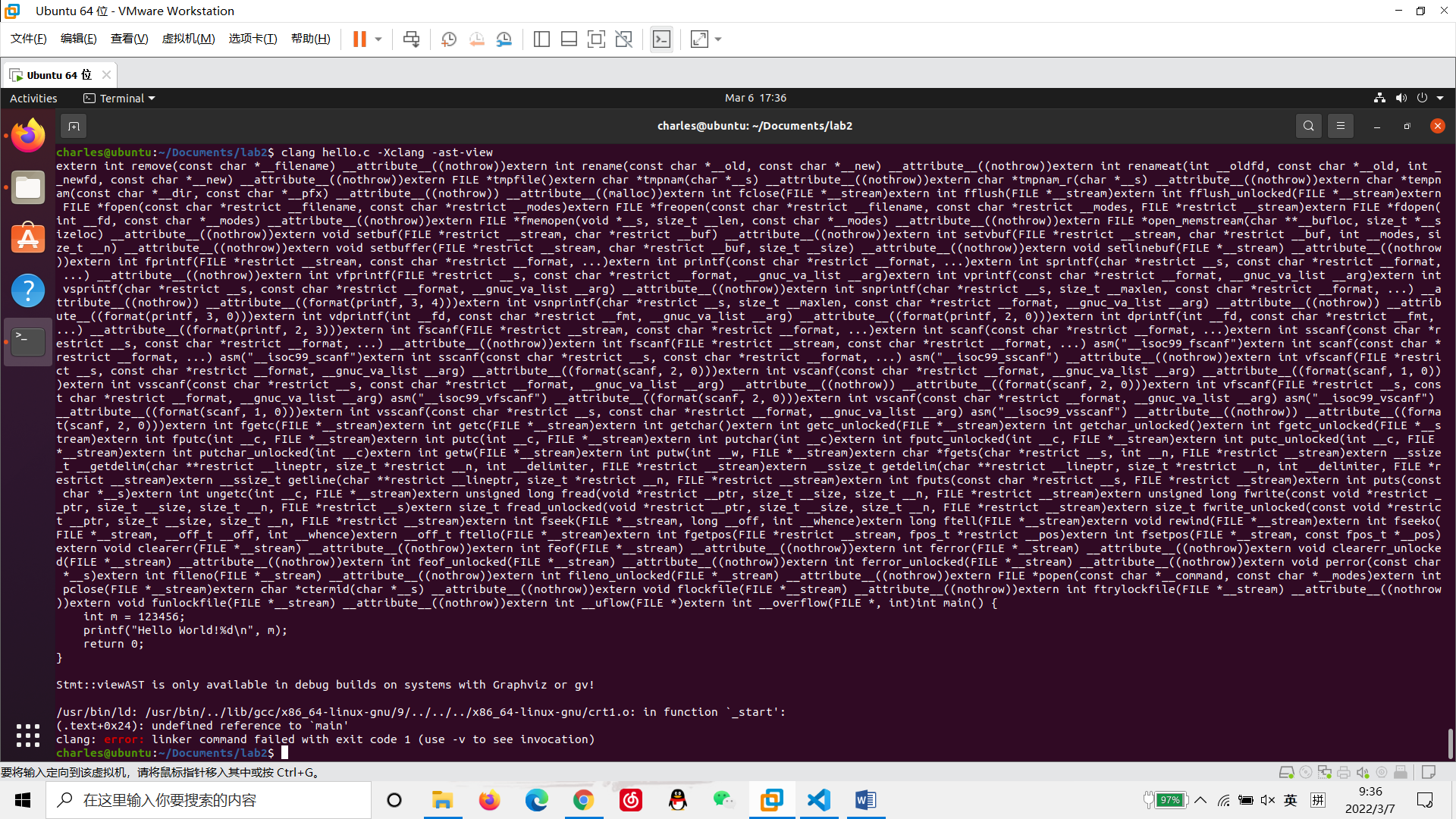
以如下语句为例：

int m = 123456;

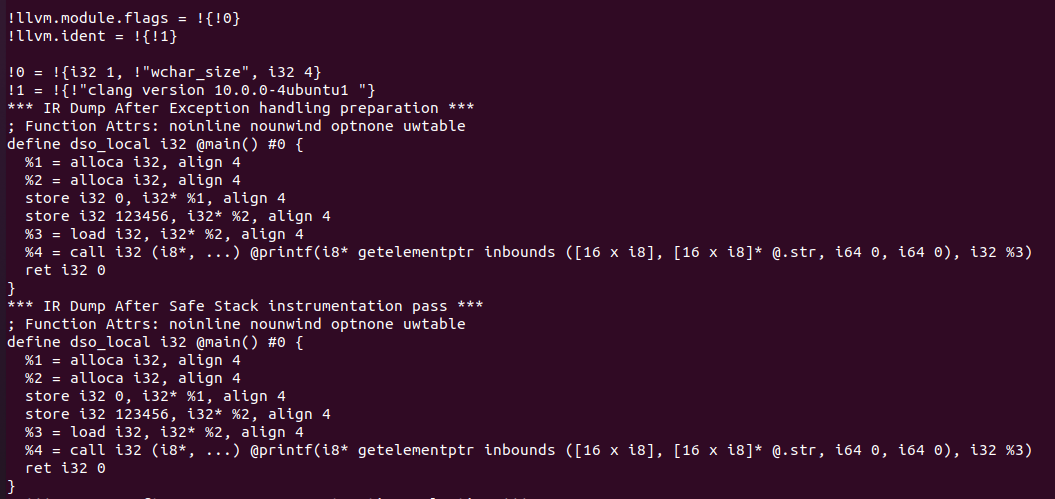
词法分析结果如下：



* + 1. 查看语法分析结果2：clang hello.c -Xclang -ast-view



* + 1. 查看编译优化的结果：clang hello.c -S -mllvm -print-after-all



对应代码

int m = 123456;

将整型常量123456存入一个寄存器中

* + 1. 查看生成的目标代码结果：clang hello.c –S

以语句

int m = 123456;

为例，对应的汇编语句为：



即将常量 123456 移动到堆栈基指针 (rbp) 偏移一个整型常量 (-8) 的寄存器中

1. 具体实验步骤和操作

对于同一个程序 (矩阵乘法), 在相同的配置（Linux）下，经过不同的编译器的不同优化，执行时间如表所示：可以从下表看到，编译器不同级别的优化会对运行时间产生显著的影响。

|  |  |  |
| --- | --- | --- |
| 编译器 | 优化 | 时间/s |
| GCC | -O0 | 2.55 |
| GCC | -O1 | 1.852 |
| GCC | -O2 | 1.788 |
| GCC | -O3 | 1.186 |
| LLVM | -O0 | 2.135 |
| LLVM | -O1 | 1.912 |
| LLVM | -O2 | 1.890 |
| LLVM | -O3 | 1.518 |

1. 具体实验步骤和操作

本次实验让我体会到了高级语言“抽象”的优点，底层代码实在过于复杂，不利于程序员直接实现自己的逻辑与想法。同时也让我认识到了不同编译器之间的区别，以及不同的编译优化对函数运行的时间等有很大的影响。最后，也体会到了编译器设计的不容易，在看生成的中间代码的时候，几乎是所有语句一开始都摸不着头脑，但大致上对GCC 的编译过程有了一个较为“理性”的认知，希望在后续的课程中能够继续学习相关内容。