**实验一 汇编语言程序的上机过程**

**一、实验目的**

* 1. 掌握DOSBox与EMU8086环境下的汇编语言程序的上机步骤

(1 )用字处理程序编辑汇编语言源程序ASM文件

(2 )用MASM程序产生OBJ文件

(3 )用LINK程序产生EXE文件

* 1. 掌握程序的调试方法

**二、实验环境与设备**

计算机一台，安装Windows操作系统和DOSBox与EMU8086模拟器。

**三、预备知识**

* 1. 8086 指令系统中常用的数据传送类指令

数据传送类指令用于在寄存器之间、寄存器与内存之间、以及内存与立即数之间移动数据。这些指令不改变数据的值，只是将其从一个位置复制到另一个位置，不影响标志位。

**1. MOV（Move）**

**功能**：将一个操作数的内容复制到另一个操作数中。

**格式**：

- `MOV destination, source`

- 源操作数和目的操作数不能同时为内存地址。

**示例**：

MOV AX, BX ; 将BX寄存器的内容复制到AX寄存器

MOV AX, 1234h ; 将立即数1234h复制到AX寄存器

MOV AX, [BX] ; 将内存地址[BX]中的内容复制到AX寄存器

MOV [SI], AX ; 将AX寄存器的内容复制到内存地址[SI]

**注意事项：**

- 源操作数和目的操作数的大小必须相同（字节或字）。

- 不能直接在两个内存地址之间移动数据，需要通过寄存器中转。

**2. XCHG（Exchange）**

**功能：**交换两个操作数的内容。

**格式：**

- `XCHG destination, source`

- 两个操作数可以是寄存器或内存地址，但不能同时为内存地址。

**示例：**

XCHG AX, BX ; 交换AX和BX寄存器的内容

XCHG AX, [SI] ; 交换AX寄存器和内存地址[SI]中的内容

**注意事项：**

- 两个操作数的大小必须相同（字节或字）。

- 不能直接在两个内存地址之间交换数据，需要通过寄存器中转。

**3. PUSH（Push）**

**功能：**将一个操作数的内容压入堆栈。

**格式：**

- `PUSH source`

- 源操作数可以是寄存器或内存地址。

**示例：**

PUSH AX ; 将AX寄存器的内容压入堆栈

PUSH [BX] ; 将内存地址[BX]中的内容压入堆栈

**注意事项：**

- 堆栈指针（SP）会自动减小，以指向新的堆栈顶部。

- 操作数必须是16位的字类型。

**4. POP（Pop）**

**功能：**从堆栈中弹出一个操作数的内容。

**格式：**

- `POP destination`

- 目的操作数可以是寄存器或内存地址。

**示例：**

POP AX ; 从堆栈中弹出内容到AX寄存器

POP [BX] ; 从堆栈中弹出内容到内存地址[BX]

**注意事项：**

- 堆栈指针（SP）会自动增加，以指向新的堆栈顶部。

- 操作数必须是16位的字类型。

**5. LEA（Load Effective Address）**

**功能：**将一个内存地址的有效地址加载到寄存器中。

**格式：**

- `LEA destination, source`

- 目的操作数必须是16位寄存器，源操作数必须是内存地址。

**示例：**

LEA AX, [BX+SI] ; 将内存地址[BX+SI]的有效地址加载到AX寄存器

**注意事项：**

- 不会从内存中读取数据，只是计算地址并加载到寄存器。

- 适用于间接寻址和基址加变址寻址。

* 1. 8086指令系统中常用的算术运算类指令

使用这些指令时，需要注意操作数的大小匹配、溢出问题以及标志位的设置。

**1. ADD（Addition）**

**功能：**将两个操作数相加，结果存储在目的操作数中。

**格式：**

- ADD destination, source

- 目的操作数和源操作数可以是寄存器或内存地址，但不能同时为内存地址。

**示例：**

ADD AX, BX ; AX = AX + BX

ADD AX, 1234h ; AX = AX + 1234h

ADD [BX], AX ; [BX] = [BX] + AX

**注意事项：**

- 溢出问题：当结果超出操作数的范围时，会产生溢出。对于无符号数，溢出标志（OF）会被置1；对于有符号数，进位标志（CF）会被置1。

- 操作数大小必须一致（字节或字）。

**2. SUB（Subtraction）**

**功能：**将源操作数从目的操作数中减去，结果存储在目的操作数中。

**格式：**

- SUB destination, source

- 目的操作数和源操作数可以是寄存器或内存地址，但不能同时为内存地址。

**示例：**

SUB AX, BX ; AX = AX - BX

SUB AX, 1234h ; AX = AX - 1234h

SUB [BX], AX ; [BX] = [BX] - AX

**注意事项：**

- 溢出问题：当结果超出操作数的范围时，会产生溢出。对于无符号数，进位标志（CF）会被置1；对于有符号数，溢出标志（OF）会被置1。

- 操作数大小必须一致（字节或字）。

**3. MUL（Multiplication）与IMUL（Integer Multiplication）**

**功能：**将两个操作数相乘，结果存储在AX（字节乘法）或DX:AX（字乘法）中。

**格式：**

- MUL source / IMUL source MUL无符号整数乘法，IMUL有符号整数乘法

- 源操作数可以是寄存器或内存地址。

**示例：**

MUL BL ; AX = AL \* BL（结果存储在AX中）

MUL WORD PTR [SI] ;DX : AX = AX \* [SI]（结果存储在DX:AX中）

**操作数规则：**

8位乘法：被乘数默认放在AL中，乘数放在8位寄存器或内存字节单元中，由指令显式给出。16位乘积默认放在AX中。

16位乘法：被乘数默认放在AX中，乘数放在16位寄存器或内存字单元中，由指令显式给出。32位乘积的高16位放在DX中，低16位放在AX中。

**注意事项：**

- 两个相乘的数必须是同为8位或同为16位。

- MUL对标志位的影响：如果乘积的高半部分（对于8位乘法是AH，对于16位乘法是DX）为零，则CF与OF被清零（CF=0，OF=0），否则CF与OF被置1（CF=1，OF=1）。

- IMUL对标志位的影响：如果乘积的高半部分（对于8位乘法是AH，对于16位乘法是DX）为低半部分的符号位扩展，则CF与OF被清零（CF=0，OF=0），否则CF与OF被置1（CF=1，OF=1）。

- 乘法指令执行后，如果CF与OF被清零（CF=0，OF=0），则只用乘积的低半部分（对于8位乘法是AL，对于16位乘法是AX）即可表示乘积。

**4. DIV（Division）与IDIV（Integer Division）**

**功能：**字节类型除数时，将AX除以源操作数，商存储在AL中，余数存储在AH中；字类型除数时，商存储在AX中，余数存储在DX中。

**格式：**

- DIV source / IDIV source DIV 无符号整数除法，IDIV 有符号整数除法

- 源操作数可以是寄存器或内存地址。

**示例：**

DIV BL ; AL = AX / BL，余数存储在AH中

DIV WORD PTR [SI] ; AX =DX:AX / [SI]，余数存储在DX中

**注意事项：**

- 被除数的位数必须是除数的两倍。

- 除数是16位的字类型时，DX:AX是被除数，商存储在AX中，余数存储在DX中。

- 除数是8位的字节类型时，AX是被除数，商存储在AL中，余数存储在AH中。

- 溢出问题：不用标志位表示溢出，但如果除数为零或商超出目标寄存器的范围，会引发除法错误异常。

**5. INC（Increment）**

**功能：**将目的操作数加1。

**格式：**

- INC destination

- 目的操作数可以是寄存器或内存地址。

**示例：**

INC AX ; AX = AX + 1

INC BYTE PTR [BX] ; [BX] = [BX] + 1

**注意事项：**

- 不影响进位标志（CF）。

**6. DEC（Decrement）**

**功能：**将目的操作数减1。

**格式：**

- DEC destination

- 目的操作数可以是寄存器或内存地址。

**示例：**

DEC AX ; AX = AX - 1

DEC BYTE PTR [BX] ; [BX] = [BX] - 1

**注意事项：**

- 不影响进位标志（CF）。

**7. NEG（Negate）**

**功能：**将操作数取反加1（即取相反数）。

**格式：**

- NEG destination

- 目的操作数可以是寄存器或内存地址。

**示例：**

NEG AX ; AX = -AX

NEG WORD PTR [BX] ; [BX] = -[BX]

**注意事项：**

- 如果操作数为0，结果仍为0。

**8. CMP（Compare）**

**功能：**比较两个操作数的大小，设置相应的标志位，但不改变操作数的值。

**格式：**

- CMP destination, source

- 目的操作数和源操作数可以是寄存器或内存地址，但不能同时为内存地址。

**示例：**

CMP AX, BX ; 比较AX和BX

CMP AX, 1234h ; 比较AX和立即数1234h

CMP [BX], AX ; 比较[BX]和AX

**注意事项：**

- 比较操作实际上是执行减法操作，但不保存结果，只设置标志位。

- 操作数大小必须一致（字节或字）。

* 1. 指令系统中的条件转移与无条件转移指令

**1. 无条件转移指令（JMP）**

**功能：**无条件地将程序控制转移到指定的目标地址。

**格式：**

- JMP label

**示例：**

JMP start ; 跳转到标签start

**注意事项：**

- 短跳转：目标地址在当前指令的-128到+127字节范围内。

- 近跳转：目标地址在当前代码段内。

- 远跳转：目标地址可以跨段，需要指定段地址和偏移量。

- 跳转目标：目标地址可以是标签、偏移量或段地址:偏移量。

**2. 条件转移指令**

条件转移指令根据标志位的状态决定是否跳转。以下是一些常用的条件转移指令：

**功能：**目标地址在当前指令地址的-128到+127字节范围内。

**格式：**

- Jcc label：`cc` 是条件码，`label` 指明转移的目标地址。

**常用条件转移指令：**

|  |  |  |
| --- | --- | --- |
| 指令 | 条件 | 描述 |
| JE/JZ | ZF=1 | 相等/为零 |
| JNE/JNZ | ZF=0 | 不相等/不为零 |
| JG/JNLE | ZF=0且SF=OF | 大于/不小于等于 |
| JL/JNGE | SF≠OF | 小于/不大于等于 |
| JGE/JNL | SF=OF | 大于等于/不小于 |
| JLE/JNG | ZF=1或SF≠OF | 小于等于/不大于 |
| JS | SF=1 | 符号位为1（负数） |
| JNS | SF=0 | 符号位为0（非负数） |
| JO | OF=1 | 溢出 |
| JNO | OF=0 | 未溢出 |
| JC/JB/JNAE | CF=1 | 进位/低于/不高于等于 |
| JNC/JAE/JNB | CF=0 | 无进位/高于等于/不低于 |

**示例：**

JE equal ; 如果ZF=1（相等），跳转到标签equal

JNE not\_equal ; 如果ZF=0（不相等），跳转到标签not\_equal

**应用示例：**

MOV AL, 5

CMP AL, 10

JE equal ; 如果AL等于10，跳转到标签equal

JL less ; 如果AL小于10，跳转到标签less

JG greater ; 如果AL大于10，跳转到标签greater

equal:

; 处理相等的情况

JMP done

less:

; 处理小于的情况

JMP done

greater:

; 处理大于的情况

done:

; 结束

**3. 循环指令（LOOP）**

**功能：**根据条件重复执行代码块。

**格式：**

- LOOP label`：将 `CX` 寄存器减1，如果 `CX` 不为零，则跳转到目标地址。

- LOOPE`/`LOOPZ`：将 `CX` 寄存器减1，如果 `CX` 不为零且 `ZF=1`，则跳转到目标地址。

- LOOPNE`/`LOOPNZ`：将 `CX` 寄存器减1，如果 `CX` 不为零且 `ZF=0`，则跳转到目标地址。

**示例：**

MOV CX, 5

start\_loop:

; 循环体

LOOP start\_loop ; 如果CX不为零，跳转到start\_loop

**使用注意事项**

1. 跳转范围：

-短跳转的目标地址必须在当前指令的-128到+127字节范围内。

-近跳转的目标地址必须在当前代码段内。

-远跳转可以跨段，但需要指定段地址和偏移量。

2. 条件判断：

-条件转移指令依赖于标志位的状态，因此在使用条件转移指令之前，通常需要执行比较（`CMP`）、测试（`TEST`）或其他设置标志位的指令。

3. 循环控制：

-循环指令（如 `LOOP`）依赖于 `CX` 寄存器的值，因此在使用循环指令之前，需要正确初始化 `CX` 寄存器。

4. 代码结构：

-使用跳转指令时，确保代码结构清晰，避免复杂的跳转逻辑，以免影响代码的可读性和维护性。

**四、实验内容**

1、程序样例如下：

; 程序功能是显示字符串 "hello world."

data SEGMENT ; 定义数据段

; 定义变量 message和pkey

message DB "hello world.",0dh,0ah,"$"

pkey DB "press any key...$"

data ENDS

stack SEGMENT stack ; 定义堆栈段

DW 128 dup(0)

stack ENDS

code SEGMENT ; 定义代码段

assume CS:code,DS:data,SS:stack ;关联段寄存器和段名

start:

; data段名表示该段的段地址，赋值给数据段寄存器DS

MOV AX, data

MOV DS, AX

; 利用21H中断的9号功能，输出字符串message

LEA DX, message

MOV AH, 9

INT 21h

; 利用21H中断的9号功能，输出字符串pkey

LEA DX, pkey

MOV AH, 9

INT 21h

; 利用21H中断的1号功能，从键盘读取一个字符

MOV AH, 1

INT 21h

; 利用21H中断的4CH号功能，结束程序运行并返回操作系统

MOV AL, 0

MOV AH, 4ch

INT 21h

code ENDS

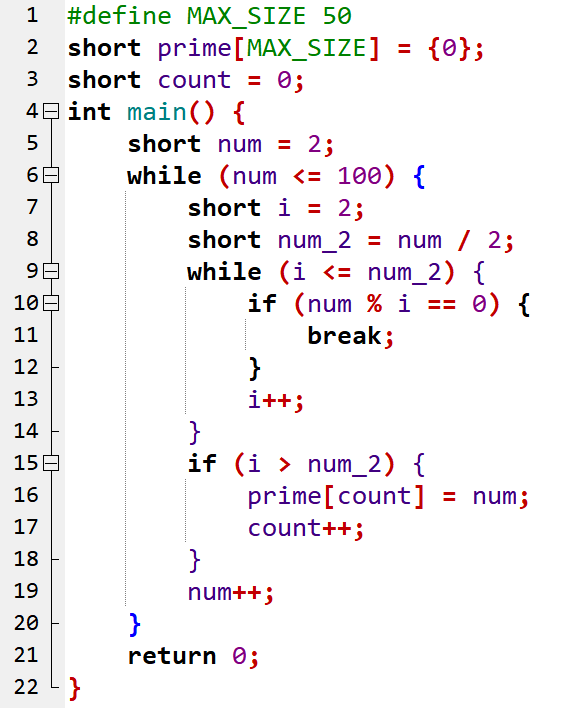
END start ; END伪指令指明此为源程序末尾，并指定程序入口是start标号

分别使用DOSBOX和EMU8086两个模拟器，完成该程序的编辑、汇编、链接和调试，掌握MASM、LINK、DEBUG的用法，熟悉EMU8086的用法。

2、编写程序，在EMU8086模拟器中汇编、链接，并调试运行。

程序功能要求如下：把100内的所有素数放入数组PRIME中。

**备注：**程序的C语言源代码如下图所示，用汇编语言实现该程序。



**五、实验要求**

1、上机前要作好充分准备，包括程序框图、源程序清单、调试步骤、测试方法、对运行结果的分析等。

2、要熟悉与实验有关的系统软件(如编辑程序、汇编程序、连接程序和调试程序等)的使用方法。在程序的调试过程中，有意识地学习及掌握debug程序的各种操作命令，以便掌握程序的调试方法及技巧。

3、程序调试完后，须由实验辅导教师在机器上检查运行结果，经教师认可后的源程序方可通过打印输出。每个实验完成后，应写出实验报告。实验报告的要求如下：

(1)实验目的：对本次实验的目的加以说明。

(2)实验内容：本次实验你所完成的具体内容加以说明。

(3)设计思想：用程序流程图描述算法。

(4)程序代码：经调试并运行确认后的汇编语言源程序清单。

(5)结果分析：包括调试情况，如上机时遇到的问题及解决办法、观察到的现象及其分析．对程序设计技巧的总结及分析；程序的输出结果（屏幕截图）及对结果的分析；实验的心得体会等。

实验报告

班级： 23计科班 姓名： 学号：

1. 实验目的：

1.掌握DOSBox与EMU8086环境下的汇编语言程序的上机步骤

(1 )用字处理程序编辑汇编语言源程序ASM文件

(2 )用MASM程序产生OBJ文件

(3 )用LINK程序产生EXE文件

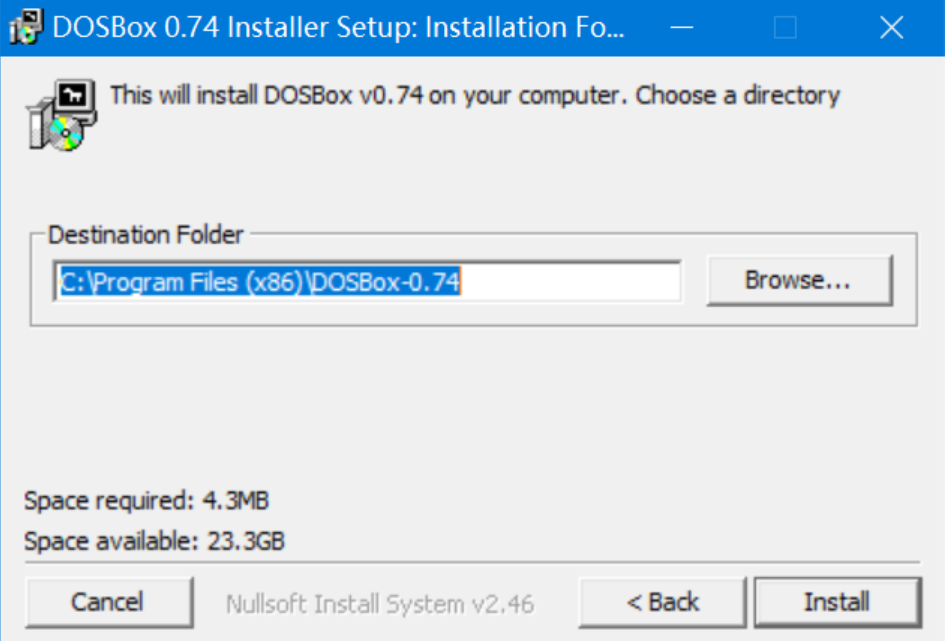
2.掌握程序的调试方法

(2)实验内容：

①.分别使用DOSBOX和EMU8086两个模拟器，完成该程序的编辑、汇编、链接和调试，掌握MASM、LINK、DEBUG的用法，熟悉EMU8086的用法。

DOSBOX用法：

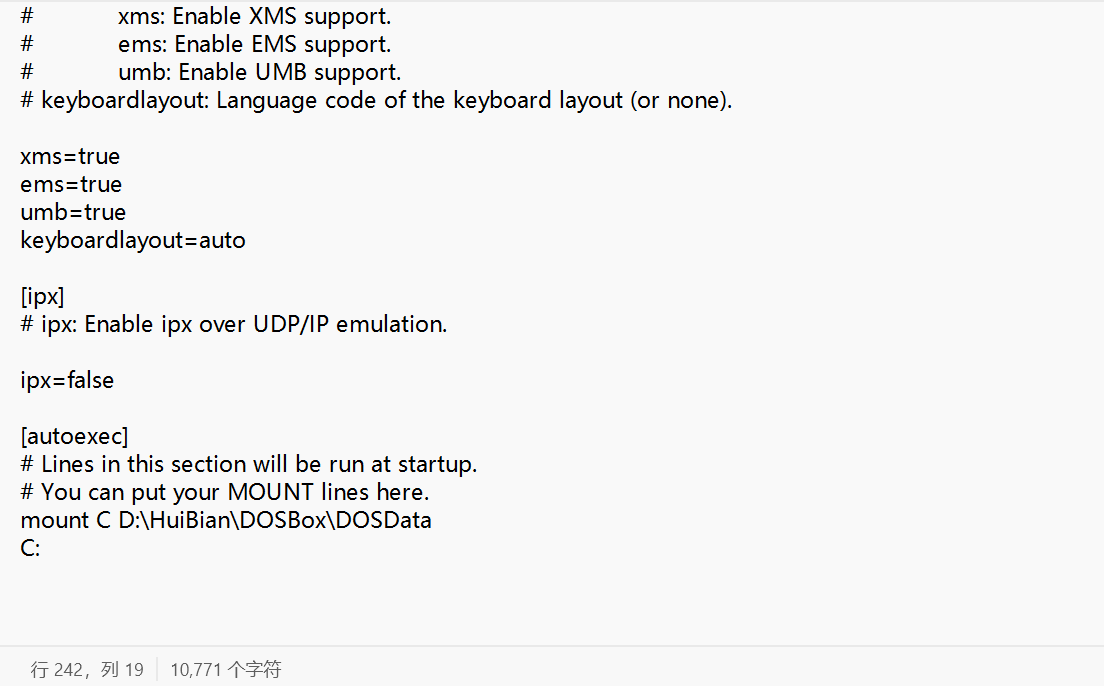
1. 首先安装DOSBOX，我安装路径是D:\HuiBian\DOSBox



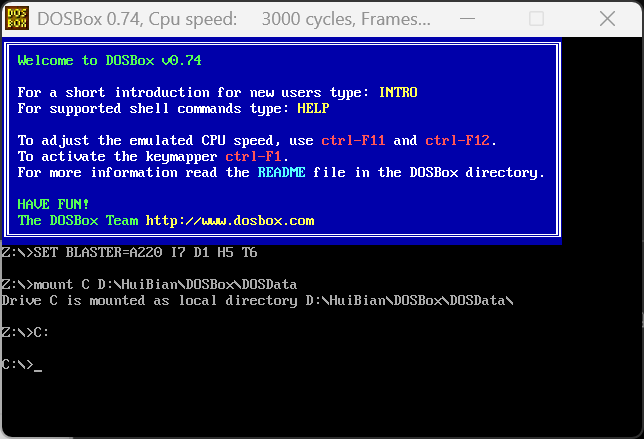
1. 将MASH工具剪切在D盘，我的路径是D:\HuiBian\DOSBox\DOSData，该文件夹会存放你写的指令代码



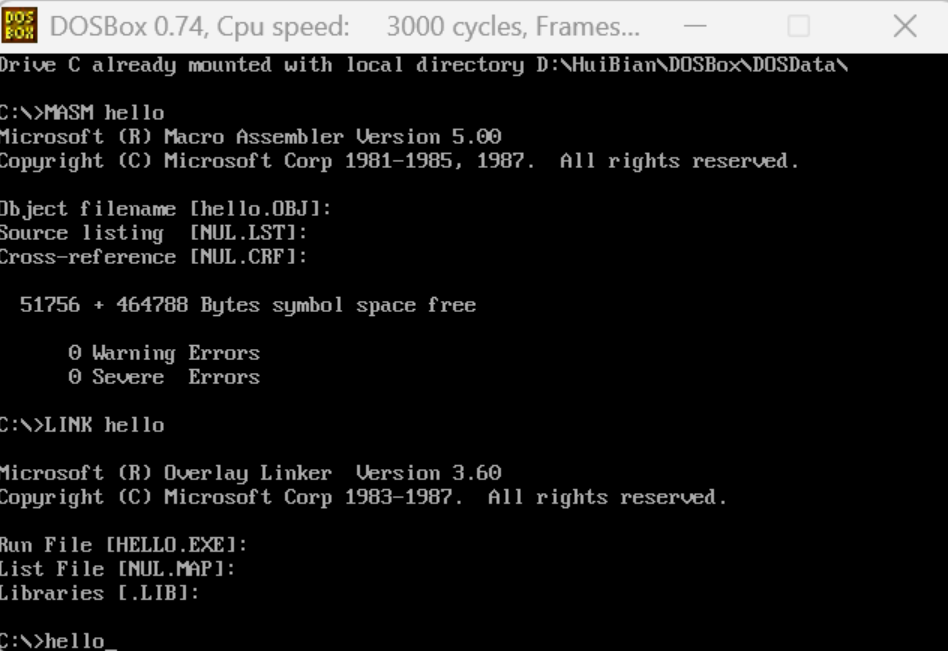
1. 打开安装好的DOSBox的文件夹，打开DOSBox 0.74 Options.bat最末尾在里面添加你第二步的文件夹路径，如下：



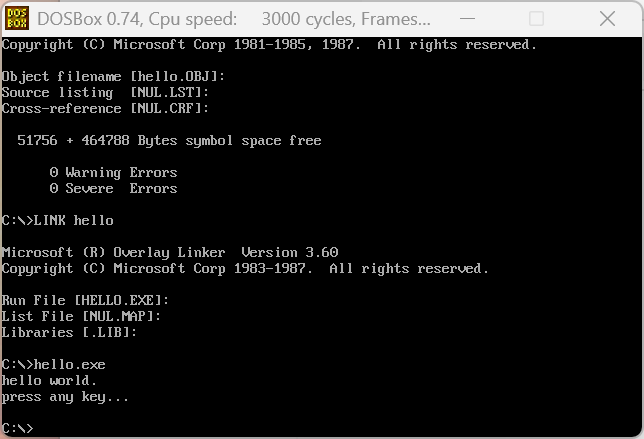
1. 打开DosBox.exe，先切换成虚拟C盘和你要存指令代码的路径下，然后开始实验



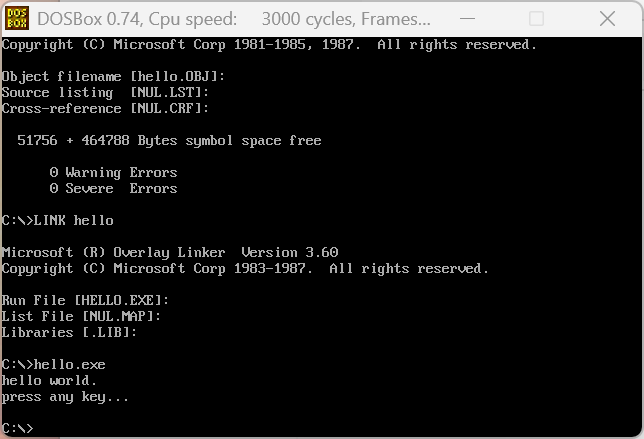
1. 先将样例代码保存在存指令代码的路径下，我将文件命名为hello.ASM,然后输入指令MASM hello,连按三次Enter键如图所示，也可以最后面加上分号;跳过中间生成可执行文件



6.然后输入指令link hello作连接，连按三次Enter键如图所示，还是一样的，也可以最后面加上分号;跳过中间生成可执行文件

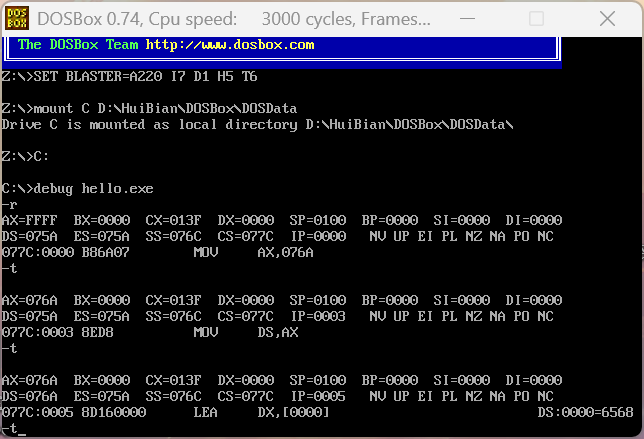


7.输入指令hello.exe后如图所示，返回保存文件夹目录下会发现产生了多了两个文件，分别是hello.OBJ和hello.exe



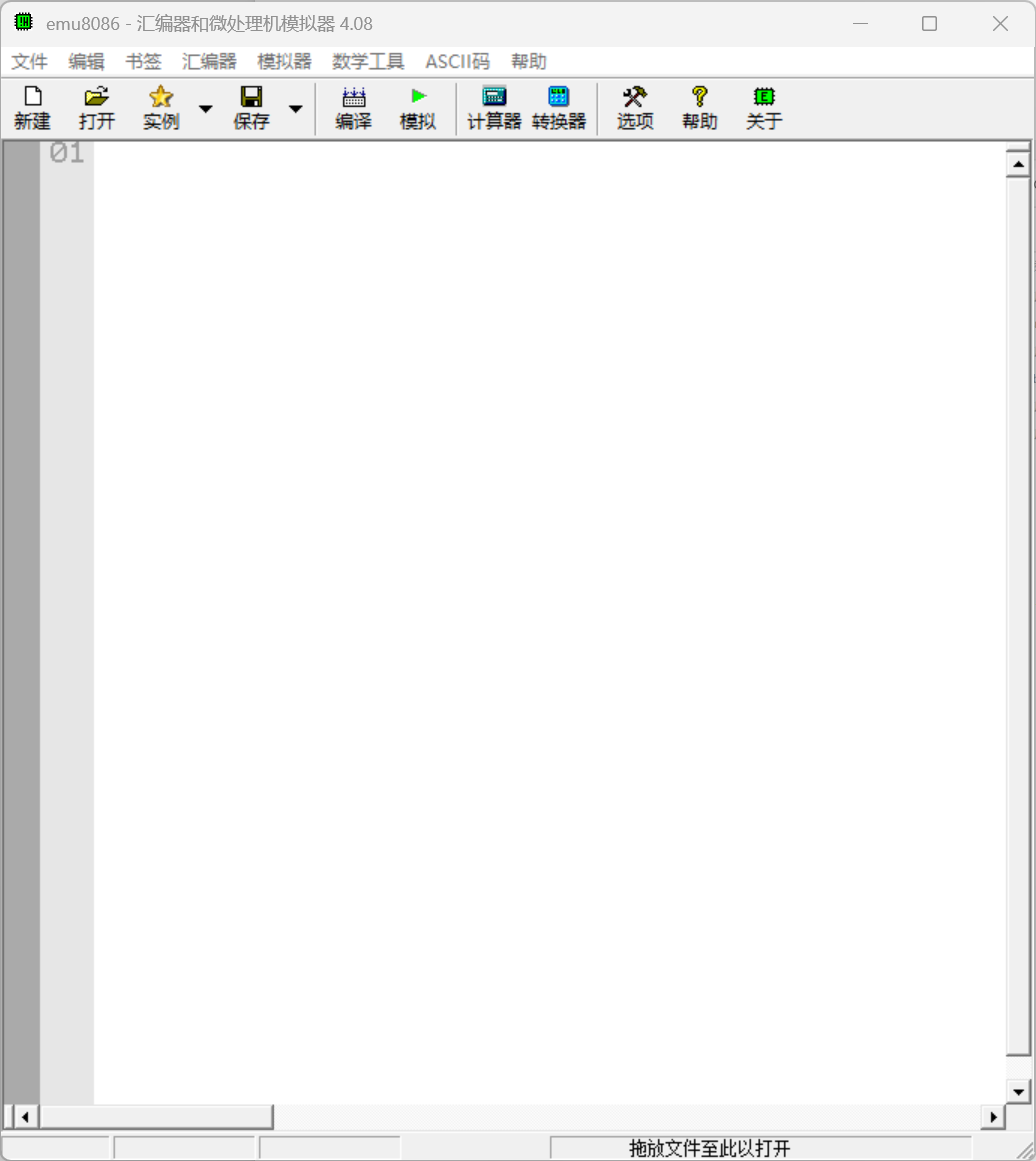


8.输入debug hello.exe指令如图所示,exe后缀这里不能省略

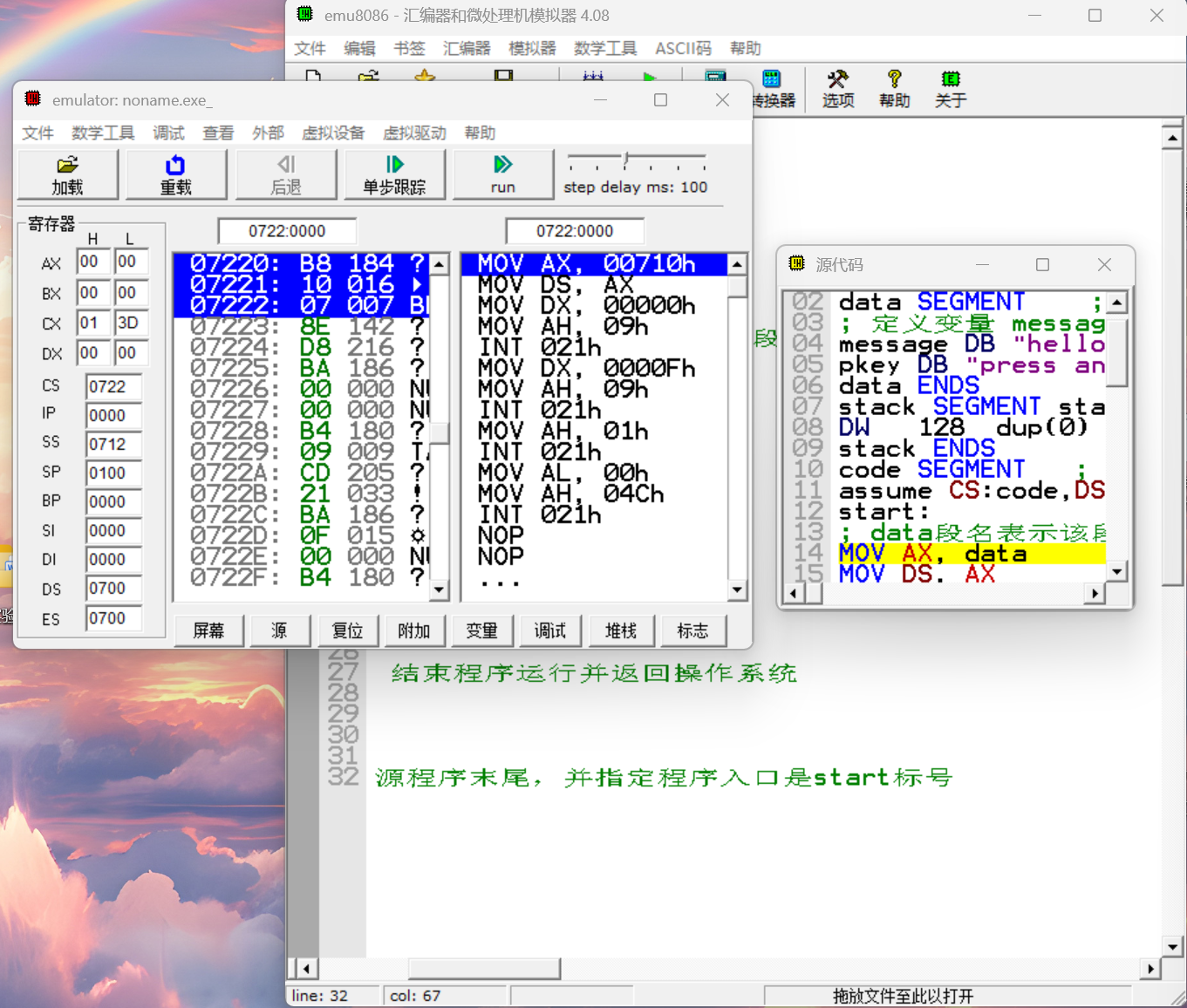


EMU8086用法：

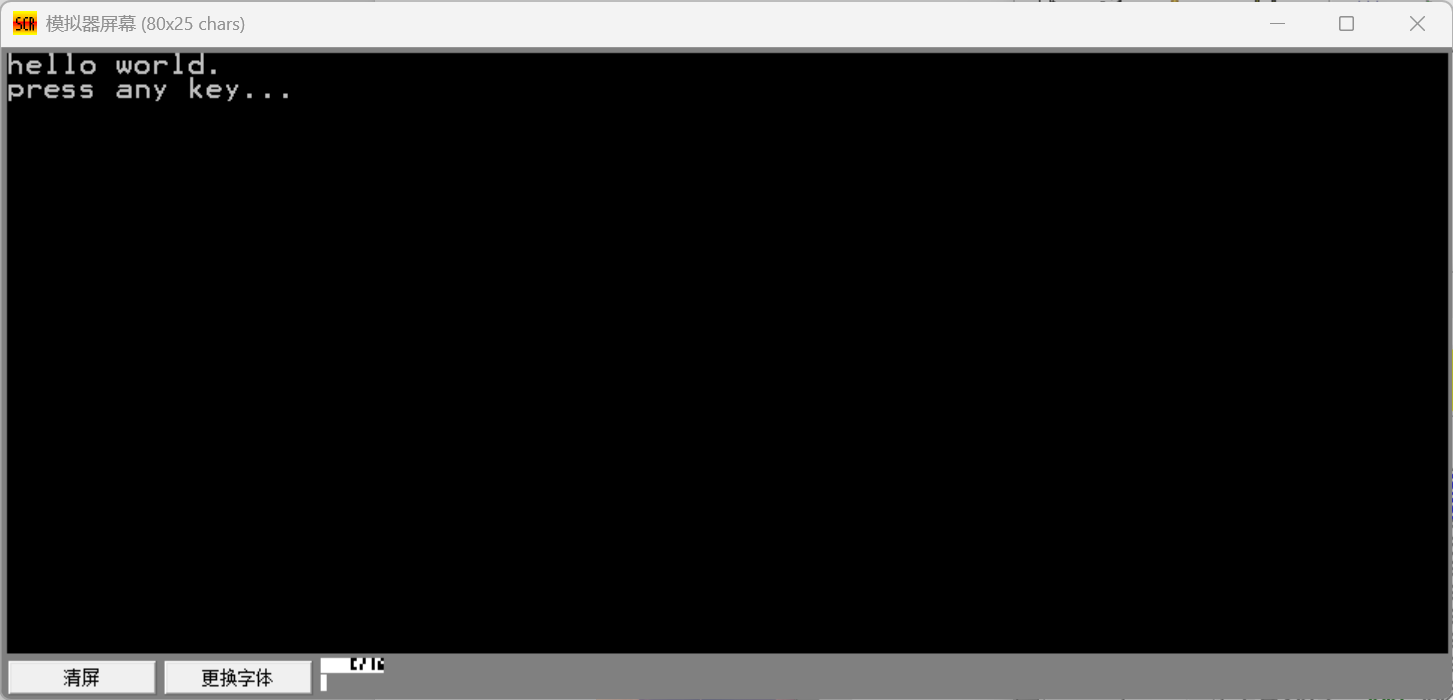
1. 我在CSDN安装了汉化版EMU8086并激活了注册码，如图所示



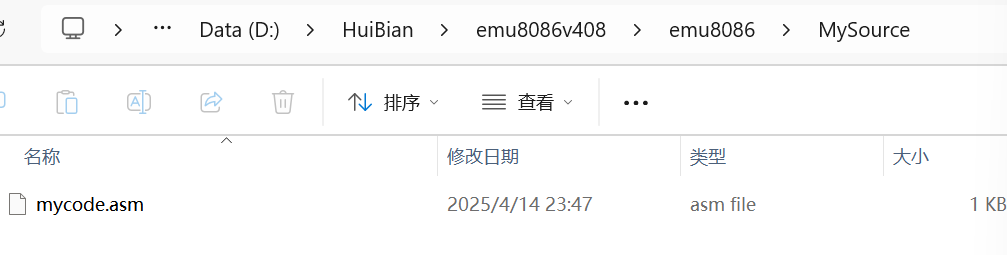
1. 复制样例指令代码然后点击模拟按钮如图所示



1. 在emulator.noname.exe\_里点击run，运行结果如下



1. 运行之后会在MySource里面生成.ASM文件，以及在MyBuild里面生成exe和中间部分的可执行文件还有debug文件，如图所示

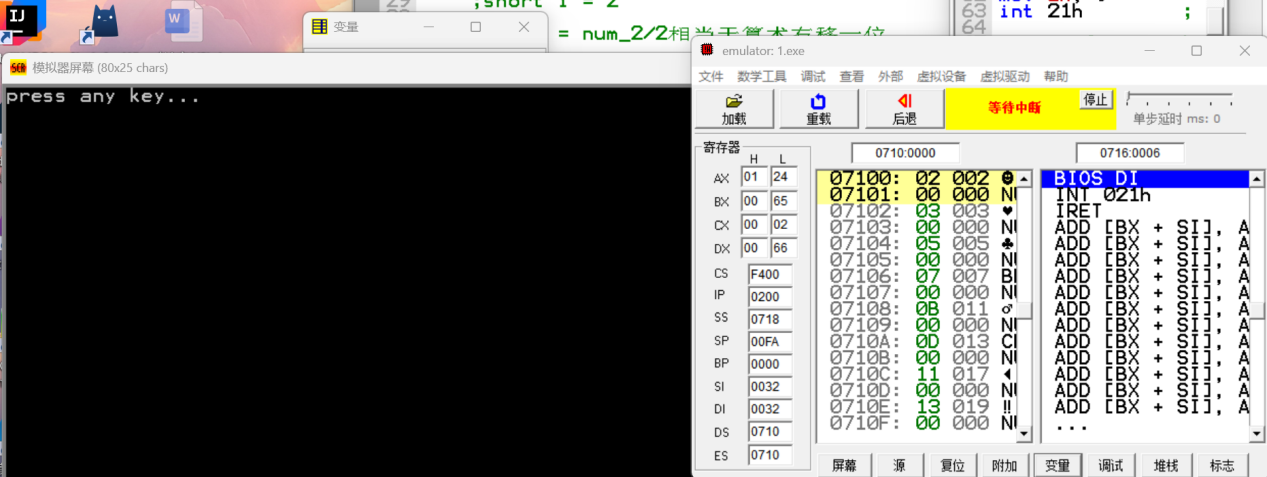




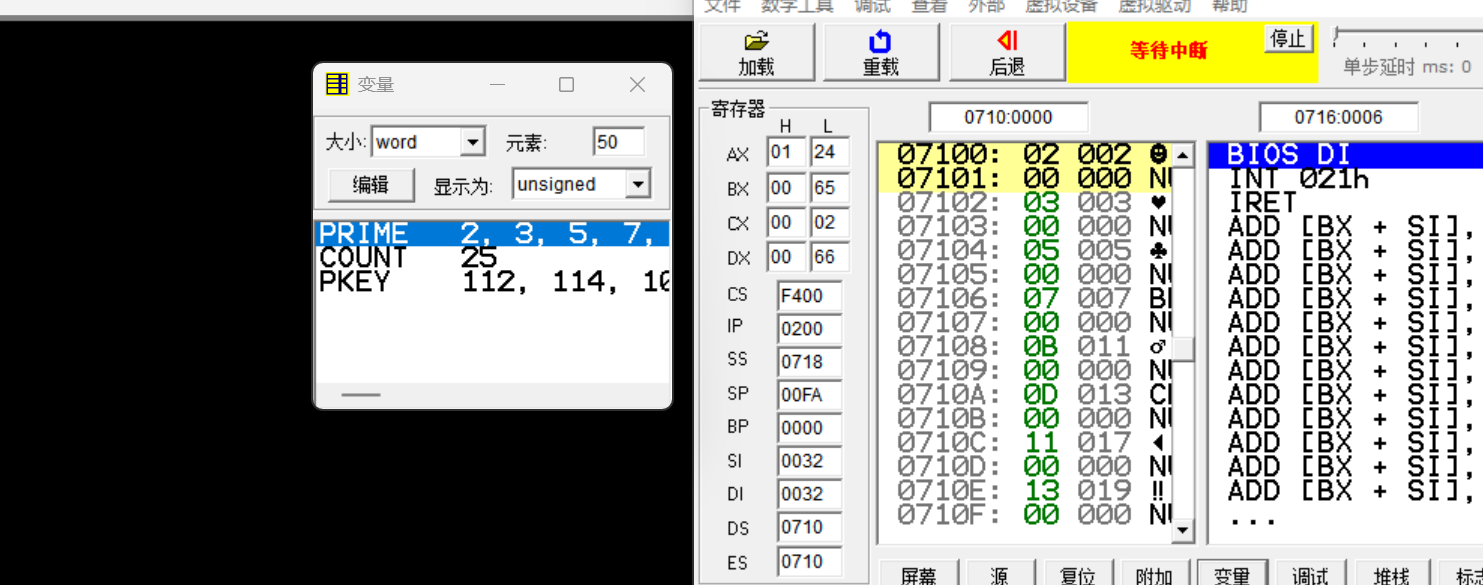
②．编写程序，在EMU8086模拟器中汇编、链接，并调试运行。

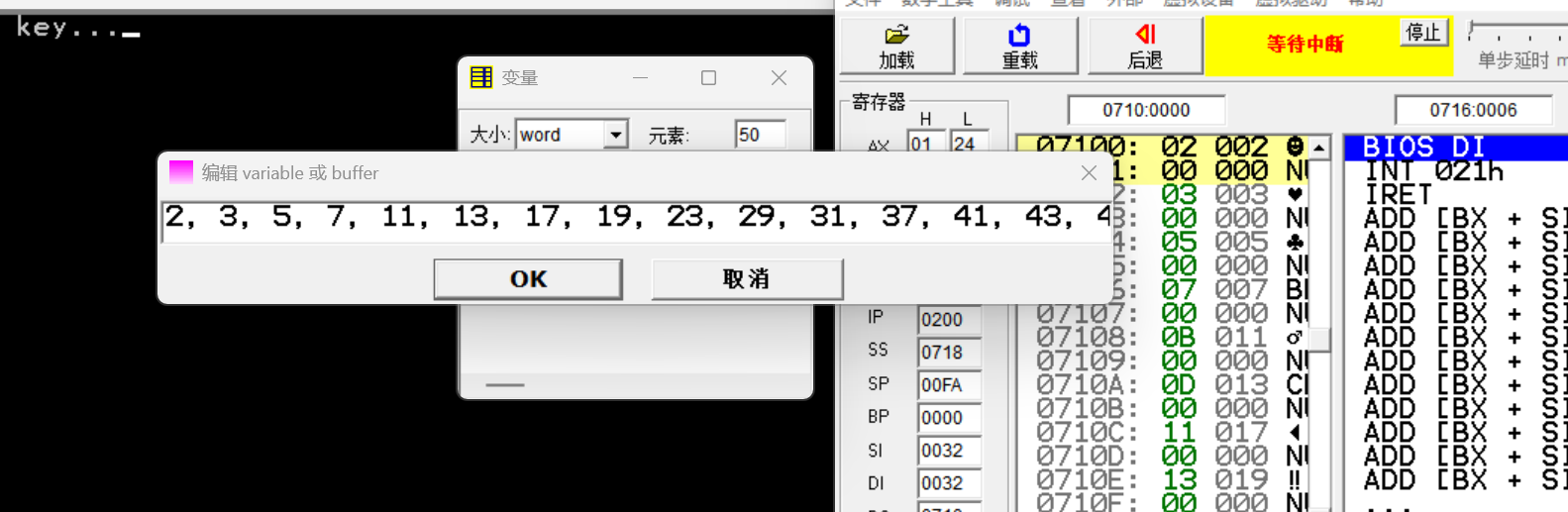
程序功能要求如下：把100内的所有素数放入数组PRIME中。

我将老师在课上教我们的代码运行一下，编译结果如图

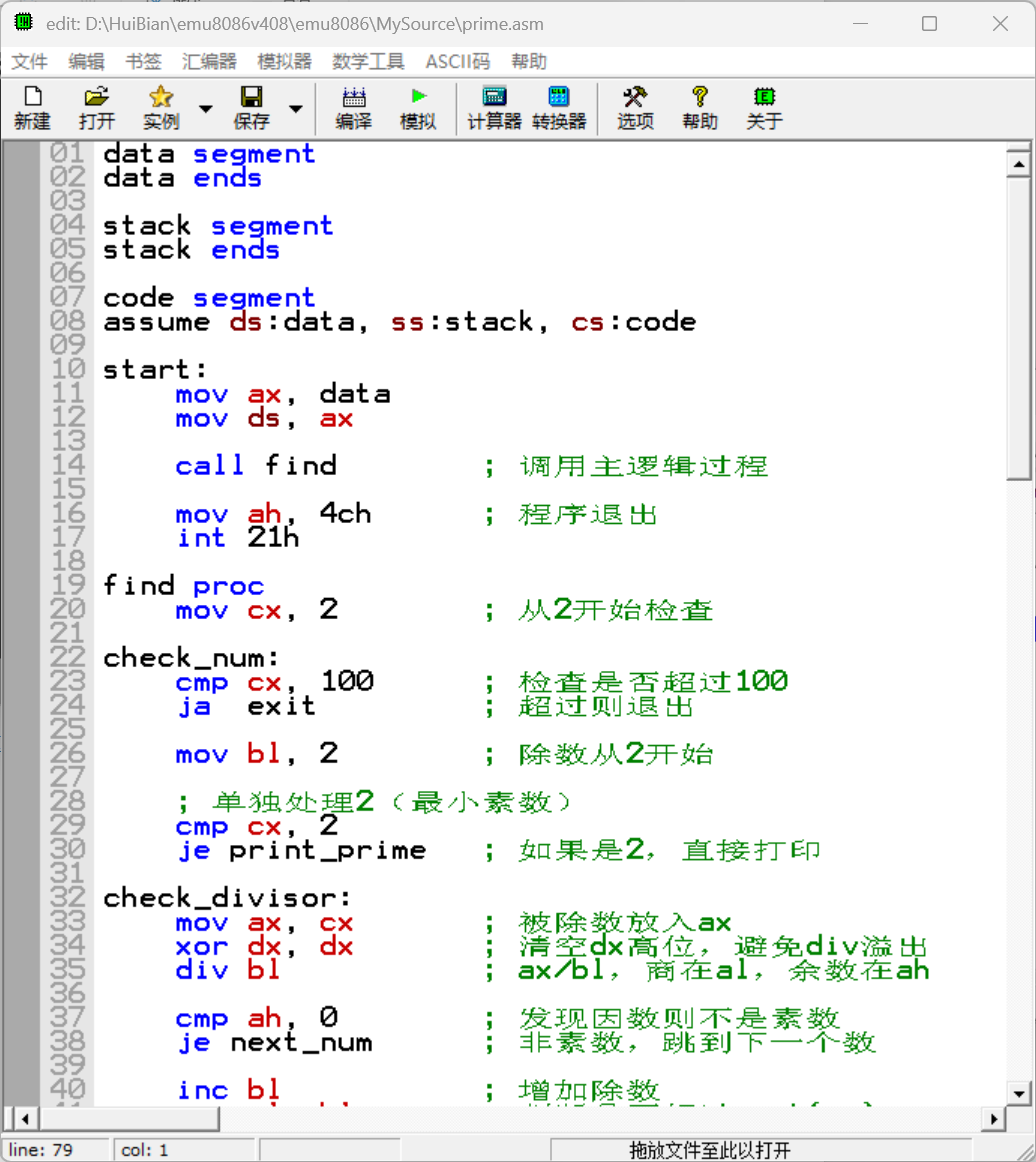


在程序执行完成后点开emulator页面右下角变量即var查看执行过程中变量内容，然后将size改为word,将PRIME和COUNT的显示为就是show as改为unsigned，并将PRIME的变量名改为50，结果如图所示。

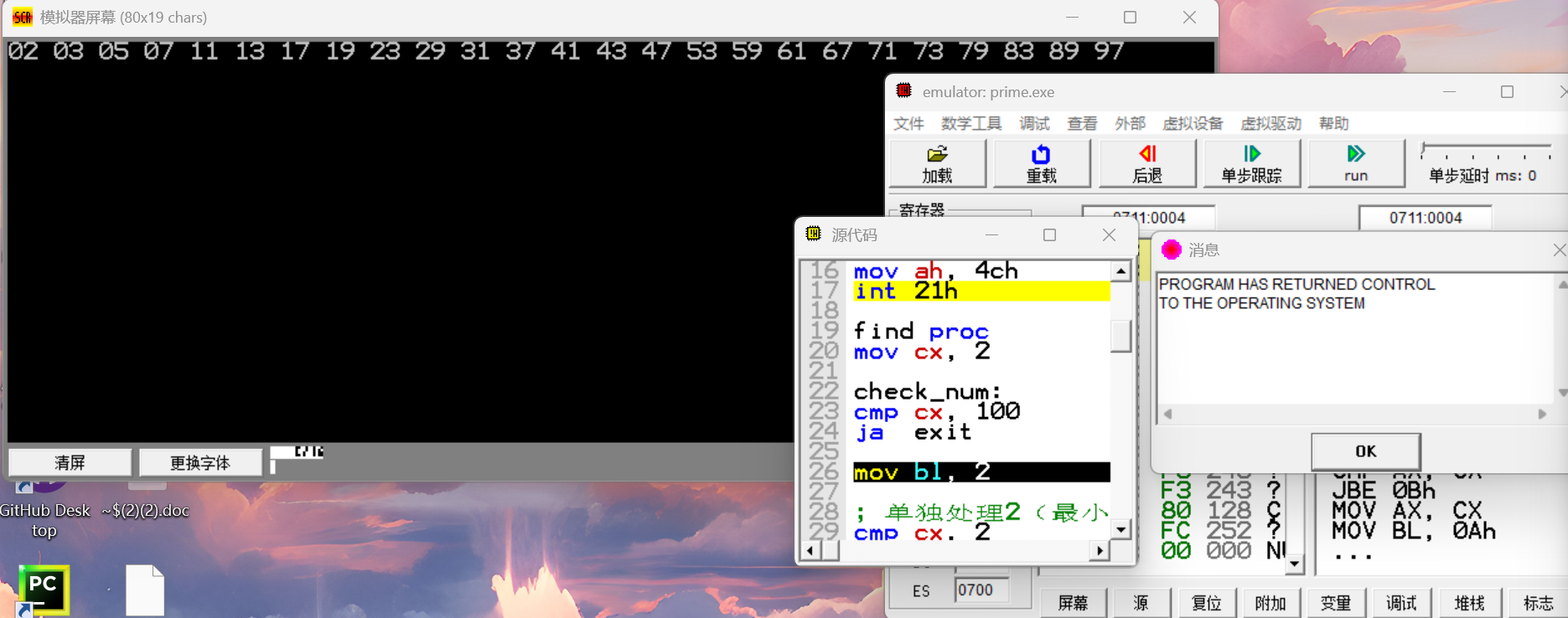




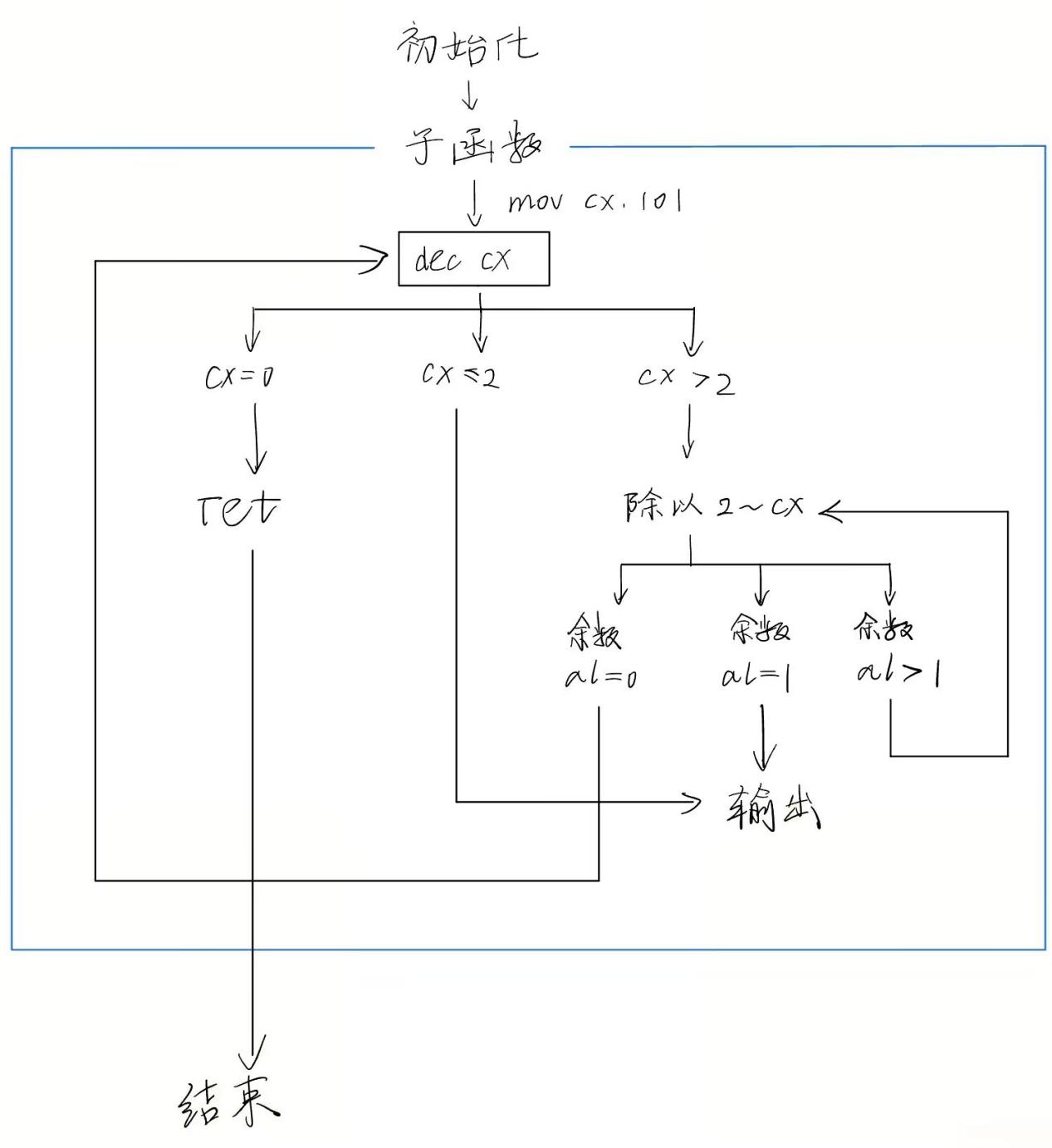
另外，我在CSDN找的可以在exe上面输出代码存入prime.asm，然后点击运行编译，如图所示



再在弹出的小框中点击run，运行结果如图所示



(3)设计思想：用程序流程图描述算法。



算法流程图如上图所示，算法描述：

1.初始化：

设置 CX 为 101，从 2 到 100 进行循环。

设置 DX 为 0，用于存储余数。

2.主循环：

初始化 CX 为 101，进入循环。

每次循环中，从 2 开始，到 CX 减 1 的数进行除法检查。

3.内循环：

对于大于 2 的数，进行除法检查。

用 CX 作为除数，从 2 开始逐步增加，检查 CX 是否为当前数的因数。

如果是因数，则说明该数不是素数，跳过该数。

如果 CX 大于当前数的平方根，则说明该数是素数。

4.输出素数：

如果当前数是素数，则进行输出。

输出格式为两位数，例如 02, 03, 05, 07, 等等。

设计思想总结：

1.采用从 2 开始逐个检查的循环结构，确保不会遗漏任何数。

2.通过计算余数来判断当前数是否为素数，避免使用复杂的算法。

3.将素数输出为两位数，便于阅读和理解。

(4)程序代码：经调试并运行确认后的汇编语言源程序清单。

任务①有样例，且（2）中已实现了，故不赘述了

任务②代码：

data segment

data ends

stack segment

stack ends

code segment

assume ds:data, ss:stack, cs:code

start:

    mov ax, data

    mov ds, ax

    call find        ; 调用主逻辑过程

    mov ah, 4ch      ; 程序退出

    int 21h

find proc

    mov cx, 2        ; 从2开始检查

check\_num:

    cmp cx, 100      ; 检查是否超过100

    ja  exit         ; 超过则退出

    mov bl, 2        ; 除数从2开始

    ; 单独处理2（最小素数）

    cmp cx, 2

    je print\_prime   ; 如果是2，直接打印

check\_divisor:

    mov ax, cx       ; 被除数放入ax

    xor dx, dx       ; 清空dx高位，避免div溢出

    div bl           ; ax/bl，商在al，余数在ah

    cmp ah, 0        ; 发现因数则不是素数

    je next\_num      ; 非素数，跳到下一个数

    inc bl           ; 增加除数

    mov al, bl       ; 判断是否超过sqrt(cx)

    mul al           ; AX = BL\*BL

    cmp ax, cx       ; 只需检查到sqrt(n)

    jbe check\_divisor

print\_prime:

    ; 十位和个位转换

    mov ax, cx       ; 将当前素数加载到ax

    mov bl, 10       ; 除以10

    div bl           ; 商在al，余数在ah

    add al, 30h      ; 十位转ASCII

    add ah, 30h      ; 个位转ASCII

    mov dh, ah       ; 保存个位到DH

    ; 输出十位

    mov dl, al       ; 十位存入DL

    mov ah, 02h      ; DOS中断输出字符

    int 21h          ; 输出十位

    ; 输出个位

    mov dl, dh       ; 从DH恢复个位

    int 21h          ; 输出个位

    ; 输出空格分隔

    mov dl, ' '      ; 空格字符

    int 21h          ; 输出空格

next\_num:

    inc cx           ; 检查下一个数

    jmp check\_num

exit:

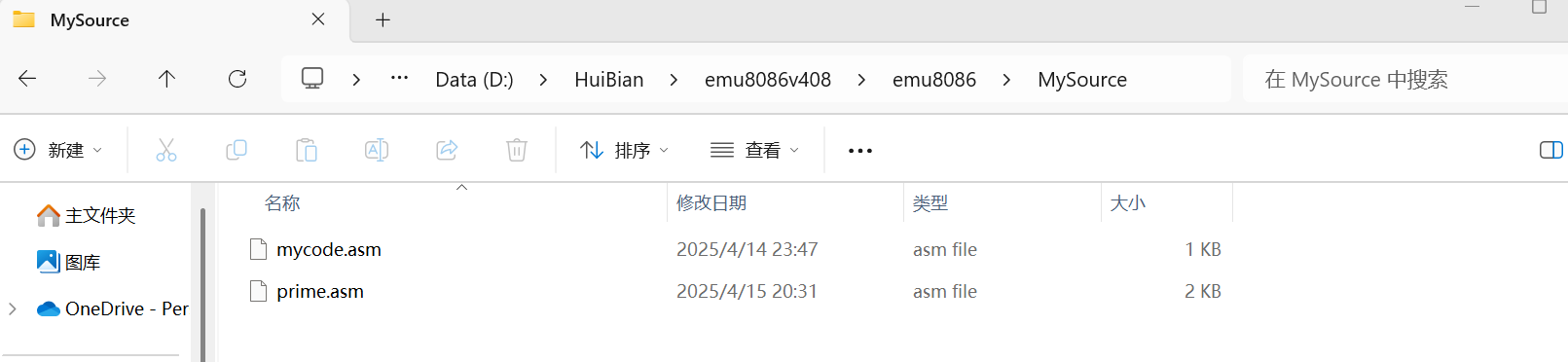
    ret              ; 返回

find endp

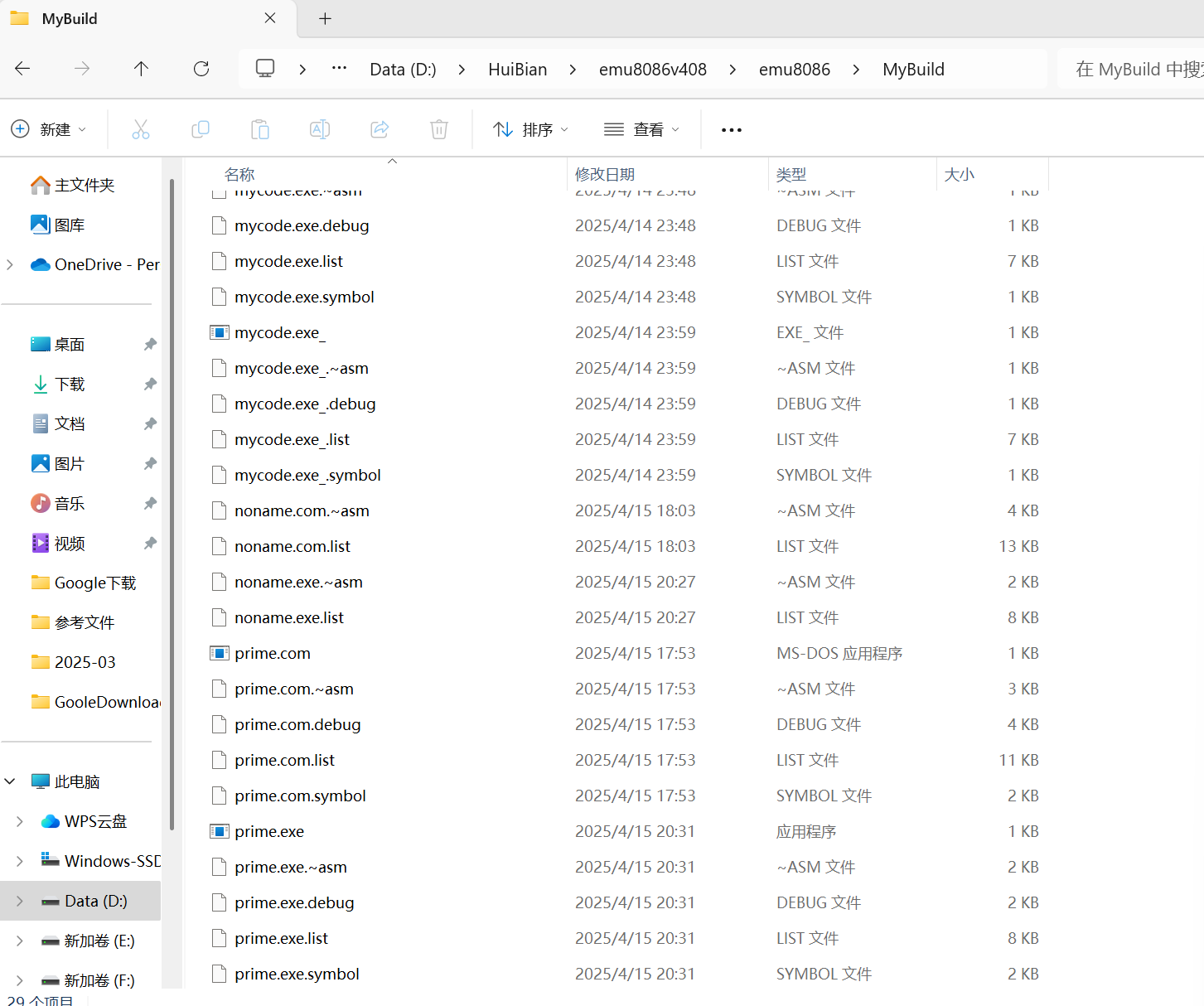
code ends

end start

Prime.asm如图所示



Prime.exe和prime.debug及中间产生的可执行文件如图所示



(5)结果分析：包括调试情况，如上机时遇到的问题及解决办法、观察到的现象及其分析．对程序设计技巧的总结及分析；程序的输出结果（屏幕截图）及对结果的分析；实验的心得体会等。

1. 调试情况

在实验过程中，调试是确保程序正确运行的重要环节。以下是具体的调试情况及解决方案：

数字 2 被遗漏

现象：程序最初没有输出素数 2，导致结果不完整。

原因分析：素数检查逻辑中，默认从 2 开始检查，但未单独处理 2 的特殊情况。

解决办法：在进入素数检查逻辑之前，添加了对 2 的单独判断，并直接将其作为素数输出。

寄存器覆盖问题

现象：在输出两位数素数时，个位数显示错误，如 03 输出为 0H。

原因分析：使用 int 21h 中断输出字符时，AH 寄存器被覆盖，导致个位数的值丢失。

解决办法：在调用中断前，使用栈或临时寄存器，如 DH保存 AH 的余数值，避免被覆盖。

2. 观察到的现象及其分析

程序输出格式整齐，程序成功地将所有素数以两位数的形式输出，且每个素数之间用空格分隔。例如：02 03 05 07 ... 97。我通过在输出前将商和余数分别转换为 ASCII 字符，并补零处理，确保了格式统一。

算法效率较高，程序仅检查到每个数的平方根即可判断是否为素数，而不是逐一检查所有可能的因数。这种优化减少了不必要的计算，提高了程序的运行效率。

3. 程序设计技巧总结及分析

模块化设计：程序采用了清晰的子过程结构，如 find proc，使代码逻辑更加简洁易读。

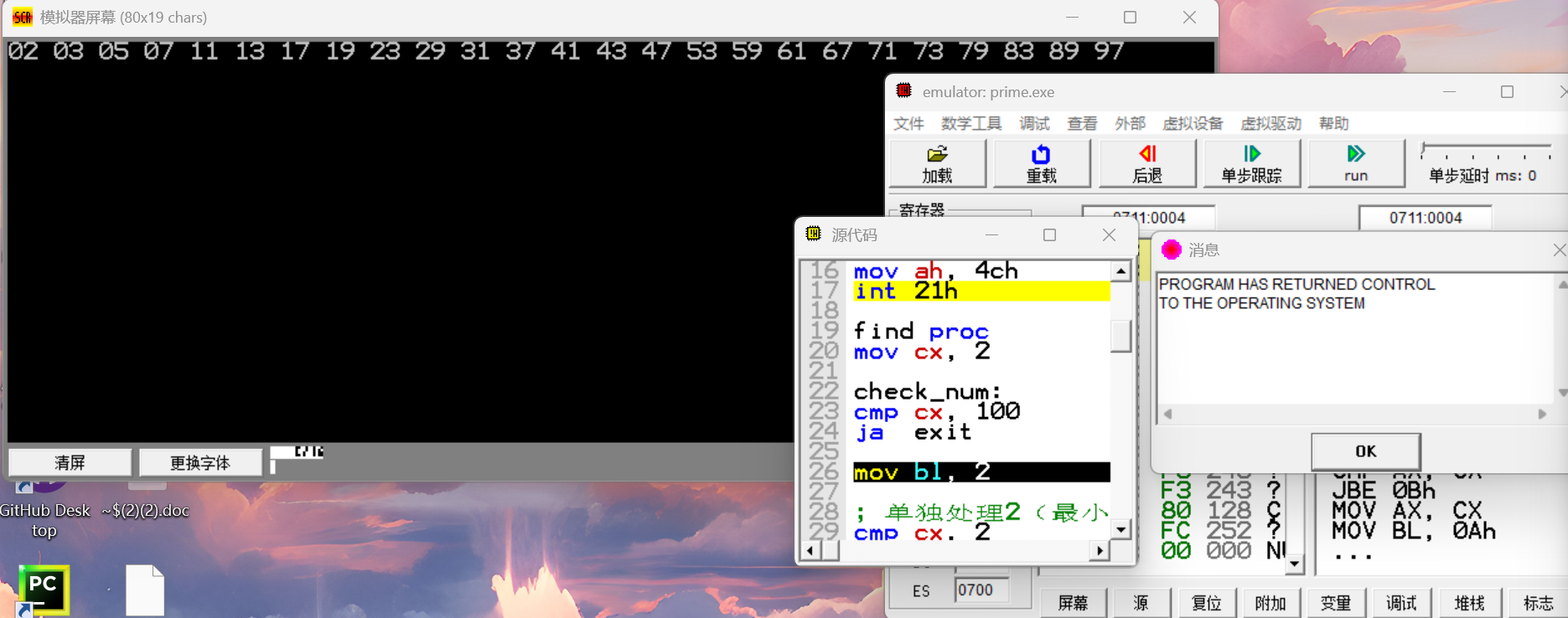
每个模块负责特定的功能，如素数判断、格式化输出，便于调试和扩展。

寄存器管理：在汇编语言中，寄存器是非常有限的资源。通过合理分配和保存寄存器的值，如使用栈保存 AH，避免了数据冲突和覆盖问题。

算法优化：通过限制除数范围，显著减少了循环次数，这是素数判断中的经典优化方法。

格式化输出：使用简单的数学运算如 div bl将数字拆分为十位和个位，并通过 ASCII 转换实现了统一的两位数输出格式。

4. 程序的输出结果及分析



程序正确地输出了 100 以内的所有素数，从左往右数，共计 25 个。输出格式整齐，符合预期的两位数表示要求。素数按从小到大的顺序排列。

5. 实验的心得体会：

（1）汇编语言对硬件的操作更加直接，但也因此要求程序员对寄存器、内存等底层概念有深入理解。这次实验让我体会到寄存器管理和指令选择的重要性。

（2）在调试过程中，我学会了如何利用单步执行和查看寄存器状态来定位问题。这种方法对我未来编写更复杂的汇编程序非常有帮助。

（3）在实验过程中，我通过查阅资料，如汇编语言第3版和CSDN，解决了一些棘手的问题。这让我意识到参考资料的重要性。

（4）汉化版EMU8086打开.asm文件时会经常产生中文乱码，导致编译失败，我合理怀疑为汉化版的bug,只有打开.asm文本复制里面的内容，并在EMU8086里新建一个空的exe文件然后粘贴内容在里面并保存覆盖原来的.asm文件，再点击编译方可正常运行。