**Homework1**

**1）传统方式**

下图为具体操作

文本

AI 生成的内容可能不正确。

**代码解释如下：**

**1. 数据段定义**

asm

DATA SEGMENT

PRINT DB "Hello World!", 0AH, 0DH, '$'

DATA ENDS

DATA SEGMENT 定义数据段开始。

PRINT DB ... 定义字节数据：

"Hello World!" 是要显示的字符串。

0AH 是换行符（LF）。

0DH 是回车符（CR）。

'$' 是 DOS 中断 21H 功能 09H 要求的字符串结束符。

DATA ENDS 结束数据段。

**2. 堆栈段定义**

asm

STACK SEGMENT STACK

DW 20 DUP(0)

STACK ENDS

STACK SEGMENT STACK 定义堆栈段，并标记为 STACK（某些汇编器需要）。

DW 20 DUP(0) 分配 20 个字（word，每个 2 字节），初始化为 0，即 40 字节的堆栈空间。

STACK ENDS 结束堆栈段。

**3. 代码段与段寄存器假设**

asm

CODE SEGMENT

ASSUME CS:CODE, DS:DATA, SS:STACK

CODE SEGMENT 定义代码段。

ASSUME 告诉汇编器各段寄存器对应的段：

CS 指向 CODE

DS 指向 DATA

SS 指向 STACK

**4. 程序主体**

asm

START:

MOV AX, DATA

MOV DS, AX

将数据段 DATA 的地址通过 AX 赋给 DS，因为不能直接 MOV DS, DATA。

这样 DS 就指向数据段，可以访问其中的变量 PRINT。

asm

MOV DX, OFFSET PRINT

MOV AH, 09

INT 21H

MOV DX, OFFSET PRINT 将字符串 PRINT 的偏移地址存入 DX。

MOV AH, 09 设置 DOS 功能号 09H：显示 DS:DX 所指向的以 '$' 结尾的字符串。

INT 21H 调用 DOS 中断，执行显示。

asm

MOV AH, 4CH

INT 21H

MOV AH, 4CH 设置 DOS 功能号 4CH：程序退出。

INT 21H 调用中断，结束程序，控制返回 DOS。

asm

CODE ENDS

END START

CODE ENDS 结束代码段。

END START 告诉汇编器程序入口点是 START 标号。

总结

这段代码在 DOS 环境下运行，功能是：

在数据段定义带有换行和回车的 "Hello World!" 字符串。

设置堆栈段。

在代码段中初始化 DS，调用 DOS 中断显示字符串，然后退出程序。

如果你在 DOSBox 或类似的模拟器中运行，会看到输出：

text

Hello World!

（并换到下一行开头）

**作业的一些思考：**

**这个4CH和4C00H有区别吗**

在 DOS 中断 21H 的功能 4CH 中，**MOV AH, 4CH 和 MOV AX, 4C00H** 在功能上是等效的，但有一些细微差别：

代码对比

asm

**; 方式1：分两次赋值**

**MOV AH, 4CH ; 功能号 4CH**

**INT 21H**

**; 方式2：一次赋值**

**MOV AX, 4C00H ; AH=4CH, AL=00H**

**INT 21H**

功能说明

功能 4CH：程序终止并返回代码

AL 寄存器：包含返回码（exit code）

**在方式1中，AL 的值是未定义的（可能是任何值）**

**在方式2中，明确设置了 AL=00H，表示返回码为 0**

实际效果

在大多数情况下，没有实际区别，因为：

DOS 通常忽略返回码 - 在标准的 DOS 环境中，父进程很少检查子程序的返回码

AL 的初始值通常是 0 - 在程序开始时，寄存器往往被初始化为 0

推荐写法

虽然功能相同，但为了代码的清晰性和可移植性，建议：

asm

; 明确设置返回码为0

MOV AX, 4C00H

INT 21H

; 或者更明确地写为：

MOV AH, 4CH

MOV AL, 00H ; 明确设置返回码

INT 21H

总结

4CH 和 4C00H 在功能调用层面没有区别，但后者更清晰地表达了意图，避免了依赖 AL 的初始值，是更好的编程实践。

**为什么要有offset**

OFFSET 在汇编语言中是一个很重要的操作符，它的存在有几个关键原因：

**1. 区分立即数和地址**

asm

; 没有 OFFSET - 容易混淆

MOV DX, PRINT ; 是把 PRINT 的值（字符 'H'）给 DX？

MOV DX, OFFSET PRINT ; 明确是要 PRINT 的地址

2. 地址与数据的本质区别

asm

DATA SEGMENT

PRINT DB "Hello World!", '$' ; PRINT 标签代表的是字符串数据本身

NUM DW 1234h ; NUM 标签代表的是数值数据本身

DATA ENDS

; 使用 OFFSET 明确获取地址

MOV DX, OFFSET PRINT ; 获取字符串的地址 → DX=PRINT的偏移地址

MOV DX, OFFSET NUM ; 获取变量的地址 → DX=NUM的偏移地址

; 如果不使用 OFFSET

MOV DX, PRINT ; 错误！这会把字符串的第一个字符 'H' (48h) 给 DX

MOV DX, NUM ; 错误！这会把变量值 1234h 给 DX

3. 段地址 + 偏移地址的寻址方式

x86 实模式使用 段:偏移 的寻址方式：

DS:OFFSET PRINT 才能正确指向字符串

DOS 功能 09H 要求 DS:DX 指向字符串

asm

MOV AX, DATA ; 设置 DS 段地址

MOV DS, AX

MOV DX, OFFSET PRINT ; 设置 DX 偏移地址

; 现在 DS:DX 正确指向 "Hello World!" 字符串

4. 实际内存布局示例

假设数据段从 1000h 开始，PRINT 在段内偏移 0000h：

text

内存布局：

DS:0000 → 'H' 'e' 'l' 'l' 'o' ... ; PRINT 字符串

寄存器：

DS = 1000h (段地址)

OFFSET PRINT = 0000h (偏移地址)

实际物理地址 = DS × 16 + OFFSET = 10000h

5. 如果没有 OFFSET 会怎样？

asm

; 错误写法：

MOV DX, PRINT ; 相当于 MOV DX, [PRINT的第一个字节]

; DX = 4848h? ('H' 是 48h，但实际取决于汇编器)

; 正确写法：

MOV DX, OFFSET PRINT ; 明确获取地址偏移量

总结

OFFSET 的存在是为了：

明确区分：数据值 vs 数据地址

类型安全：避免意外地把数据当作地址使用

清晰表达：让代码意图更明确

符合架构：适应 x86 的段式内存管理

在高级语言中，这种区别类似于：

c

char \*p = &str; // 相当于 OFFSET - 取地址

char c = str[0]; // 相当于不用 OFFSET - 取值

**反汇编代码分析：**

文本

AI 生成的内容可能不正确。

**分析这个反汇编的代码**

1. 原始代码整理

地址 机器码 汇编指令

9770:0000 B8 6C 07 MOV AX,076C

9770:0003 8E D8 MOV DS,AX

9770:0005 BA 00 00 MOV DX,0000

9770:0008 B4 09 MOV AH,09

9770:000A CD 21 INT 21

9770:000C B4 4C MOV AH,4C

9770:000E CD 21 INT 21

9770:0010 25 FF BC AND AX,BCFF

9770:0013 A3 78 13 MOV [1378],AX

9770:0016 8C C0 MOV AX,ES

9770:0018 87 46 04 XCHG AX,[BP+04]

9770:001B 5D POP BP

9770:001C 2D D3 12 SUB AX,12D3

9770:001F 51 PUSH CX

2. 程序逻辑分析（前一部分是用户代码）

9770:0000 MOV AX,076C

将 076Ch 送入 AX，这通常是数据段地址。

9770:0003 MOV DS,AX

把 AX 的值赋给 DS，设置数据段。

9770:0005 MOV DX,0000

DX=0000，这是字符串偏移地址（在数据段内）。

9770:0008 MOV AH,09

准备调用 DOS 21H 中断的 09 号功能：显示字符串（DS:DX 指向 '$' 结尾的字符串）。

9770:000A INT 21

执行显示。

9770:000C MOV AH,4C

准备调用 DOS 21H 中断的 4C 号功能：程序退出。

9770:000E INT 21

退出程序，控制权交回 DOS。

注意：从 9770:0010 开始，这些指令并不是原程序的有效代码，而是内存中的残留数据被误反汇编的结果。

因为程序在 000E 处已经 INT 21/4C 退出，后面的字节可能是其他数据或未初始化的内容。

**2）另类方式使用debug-e**

数据段不变输出hello world：

文本

AI 生成的内容可能不正确。

使用-e改写成hello：

图片包含 文本

AI 生成的内容可能不正确。