

# 本科生实验报告

实验课程: 操作系统原理实验

实验名称: 从实模式到保护模式

专业名称: 计算机科学与技术

学生姓名: 数据删除

学生学号: 数据删除

实验成绩:

报告时间: 2025年3月20日

## 实验要求

### Assignment 1

1. 复现Example 1，说说你是怎么做的并提供结果截图，也可以参考Ucore、Xv6等系统源码，实现自己的LBA方式的磁盘访问。
2. 在Example1中，我们使用了LBA28的方式来读取硬盘。此时，我们只要给出逻辑扇区号即可，但需要手动去读取I/O端口。然而，BIOS提供了实模式下读取硬盘的中断，其不需要关心具体的I/O端口，只需要给出逻辑扇区号对应的磁头（Heads）、扇区（Sectors）和柱面（Cylinder）即可，又被称为CHS模式。现在，同学们需要将LBA28读取硬盘的方式换成CHS读取，同时给出逻辑扇区号向CHS的转换公式。最后说说你是怎么做的并提供结果截图。

### Assignment 2

复现Example 2，使用gdb或其他debug工具在进入保护模式的4个重要步骤上设置断点，并结合代码、寄存器的内容等来分析这4个步骤，最后附上结果截图。gdb的使用可以参考appendix的“debug with gdb and qemu”部份。

### Assignment 3

改造“Lab2-Assignment 4”为32位代码，即在保护模式后执行自定义的汇编程序。

## 实验过程

### Assignment 1

#### 1.1

使用下发的Example代码，为了方便运行，编写如下所示的Makefile文件：

HD\_IMG := ../../../../hd/hd.img

BUILD\_DIR := ../../../../build

run:

    @qemu-system-i386 -hda $(HD\_IMG) -serial null -parallel stdio

build:

    @nasm -f bin mbr.asm -o $(BUILD\_DIR)/mbr.bin

    @nasm -f bin bootloader.asm -o $(BUILD\_DIR)/bootloader.bin

    @dd if=$(BUILD\_DIR)/mbr.bin of=$(HD\_IMG) bs=512 count=1 seek=0 conv=notrunc

    @dd if=$(BUILD\_DIR)/bootloader.bin of=$(HD\_IMG) bs=512 count=5 seek=1 conv=notrunc

clean:

    @rm $(BUILD\_DIR)/\*.bin

在代码目录运行：

make build

make run

qemu会运行并输出预期结果。

#### 1.2

计算可得CHS访问磁盘bootloader的磁头号为0，柱面号为0，扇区号为2~6，因为bootloader没有跨柱面或磁头，因此可以一次性读5个扇区。

为了方便调试，将Makefile文件修改如下：

HD\_IMG := ../../../../hd/hd.img

BUILD\_DIR := ../../../../build

MBR\_O := $(BUILD\_DIR)/mbr.o

BL\_O := $(BUILD\_DIR)/bootloader.o

MBR\_SYMBOL := $(BUILD\_DIR)/mbr.symbol

BL\_SYMBOL := $(BUILD\_DIR)/bootloader.symbol

MBR\_BIN := $(BUILD\_DIR)/mbr.bin

BL\_BIN := $(BUILD\_DIR)/bootloader.bin

run:

    @qemu-system-i386 -hda $(HD\_IMG) -serial null -parallel stdio

debug:

    @gnome-terminal -e "qemu-system-i386 -s -S -hda $(HD\_IMG) -serial null -parallel stdio"

build:

    @nasm -g -f elf32 mbr.asm -o $(MBR\_O)

    @ld -o $(MBR\_SYMBOL) -melf\_i386 -N $(MBR\_O) -Ttext 0x7c00

    @ld -o $(MBR\_BIN) -melf\_i386 -N $(MBR\_O) -Ttext 0x7c00 --oformat binary

    @nasm -g -f elf32 bootloader.asm -o $(BL\_O)

    @ld -o $(BL\_SYMBOL) -melf\_i386 -N $(BL\_O) -Ttext 0x7e00

    @ld -o $(BL\_BIN) -melf\_i386 -N $(BL\_O) -Ttext 0x7e00 --oformat binary

    @dd if=$(MBR\_BIN) of=$(HD\_IMG) bs=512 count=1 seek=0 conv=notrunc

    @dd if=$(BL\_BIN) of=$(HD\_IMG) bs=512 count=5 seek=1 conv=notrunc

clean:

    @rm $(BUILD\_DIR)/\*.bin $(BUILD\_DIR)/\*.o $(BUILD\_DIR)/\*.symbol

我使用vscode+ssh远程调试，因此编写以下launch.json文件：

{

"version": "0.2.0",

"configurations": [

{

"name": "Debug",

"type": "gdb",

"request": "attach",

"target": ":1234",

"cwd": "${fileDirname}",

"remote": true,

"valuesFormatting": "parseText",

"autorun": [

"set output-radix 16",

"set disassembly-flavor intel",

"add-symbol-file /path/to/mbr.symbol 0x7c00",

"add-symbol-file /path/to/bootloader.symbol 0x7e00",

],

"preLaunchTask": "make build & debug here"

}

]

}

cwd修改为${fileDirname}而不是${workspaceRoot}，因为每个assignment的代码并不是在根目录存放，而经过多次尝试，得到结论：调试时的工作目录必须和代码目录一致（也可能是执行make build）时的目录一致。

对应的preLaunchTask如下所示：

{

"label": "make build & debug here",

"type": "shell",

"command": "cd ${fileDirname} && make build && make debug",

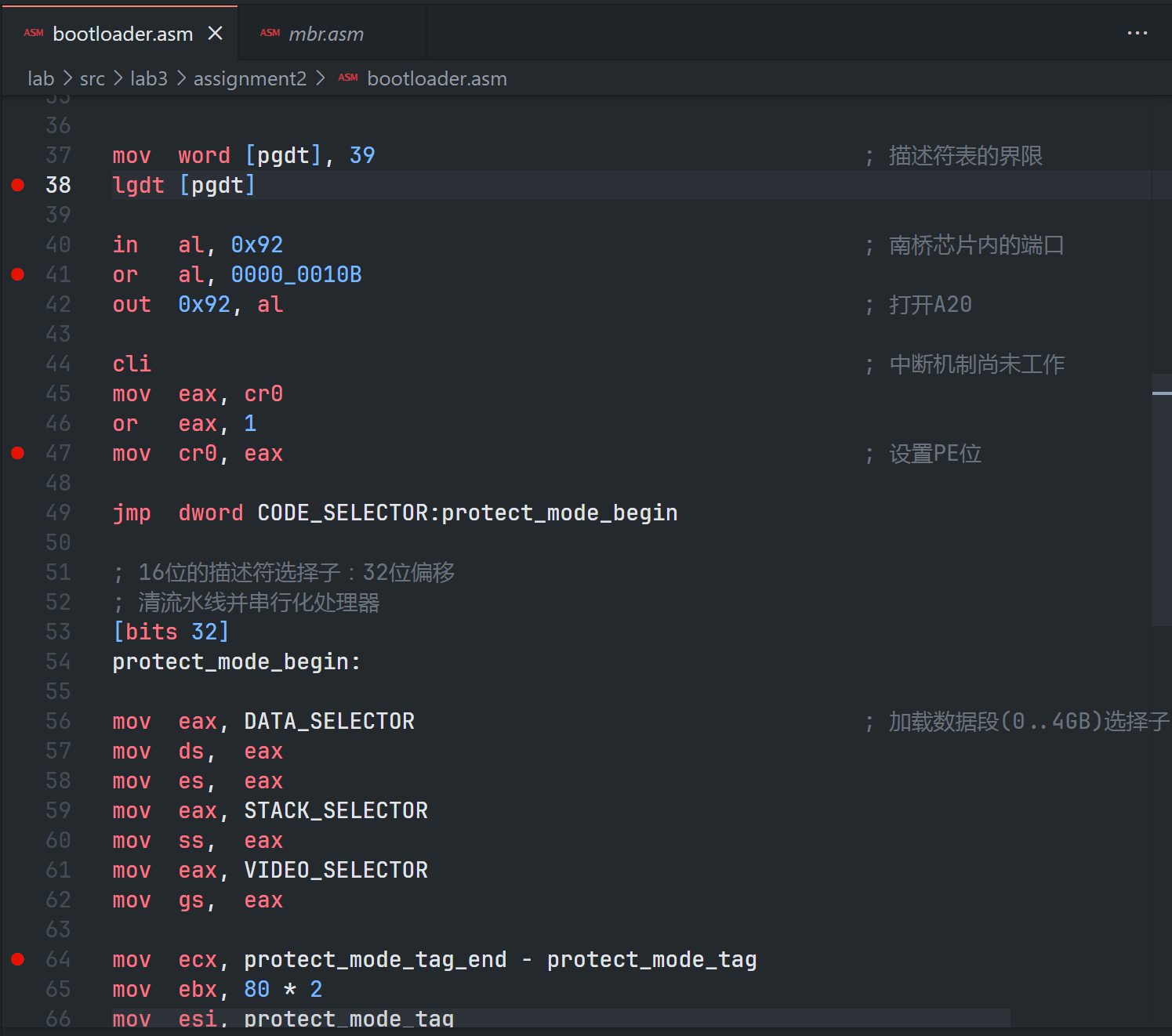
"problemMatcher": []

}

然后就可以按f5一键编译并启动调试。

### Assignment 2

使用下发的Example代码，在vscode在进入保护模式的4个重要步骤上设置断点，如图：



开启调试，程序成功在停止在第一个断点。

然而lgdt的运行结果不知道怎么看，所以略过。

尝试打开A20，发现在本机qemu下该地址线是默认打开的。

找了下发现个相关的文章：

OS boot 的时候为什么要 enable A20？ - 北极的回答 - 知乎

<https://www.zhihu.com/question/29375534/answer/44144613>

打开cr0保护模式标记位，从监视窗口可以发现cr0多出一个PF标志位，表示保护模式标志位已置位。

远跳转进入保护模式，加载段选择子。从监视窗口可以看到段寄存器的内容变成了指定的段选择子，程序正常工作。

### Assignment 3

修改Lab 2 Assignment 4最大的挑战在于，32位保护模式无法调用BIOS中断来完成输出/延时等功能，因此需要自己实现。

考虑将功能封装成函数。引用上次实验报告中的实现思路：

程序要实现的功能分为两部分：渲染反弹的数字、渲染标题。从示例视频看标题不会被反弹的数字覆盖，所以为了方便起见，每次都完整渲染一次标题。

渲染反弹的数字在反弹逻辑设计上如果直接分四个状态做会比较困难。不妨拆分成水平方向速度和垂直方向速度，独立地检测水平和垂直是否即将要出界，如果是，则将对应方向的速度取负数即可。

剩下的部分比较简单：要渲染变化的数字，直接用字符串“1357924680”加上一个循环的下标即可；渲染变化的颜色直接用一个byte，每次加一即可。

稍加分析可知，需要拆解出的函数有：清屏、延时、输出标题、在指定位置输出字符、更新下一次要输出字符的位置和方向向量。因为延时无法用中断，因此采用循环计数实现；除此之外将寄存器改为32位即可。

## 关键代码

### Assignment 1

#### 1.2

; 读取磁盘

mov   ah, 0x02            ; 功能号

mov   al, 5               ; 扇区数

mov   ch, 0               ; 柱面

mov   cl, 2               ; 扇区

mov   dh, 0               ; 磁头

mov   dl, 0x80            ; 驱动器

mov   bx, 0x7e00          ; 缓冲区地址

int   0x13

jc    load\_error          ; 检测 CF 是否不为 0

jmp   0x0000:0x7e00       ; 跳转到bootloader

load\_error:

jmp   $                   ; 死循环

### Assignment 2

%include"boot.inc"

[bits 16]

mov ax,  0xb800

mov gs,  ax

mov ah,  0x03                                            ; 青色

mov ecx, bootloader\_tag\_end - bootloader\_tag

xor ebx, ebx

mov esi, bootloader\_tag

output\_bootloader\_tag:

    mov  al, [esi]

    mov  word[gs:bx], ax

    inc  esi

    add  ebx, 2

    loop output\_bootloader\_tag

mov  dword [GDT\_START\_ADDRESS+0x00], 0x00

mov  dword [GDT\_START\_ADDRESS+0x04], 0x00

; 创建描述符，这是一个数据段，对应0~4GB的线性地址空间

mov  dword [GDT\_START\_ADDRESS+0x08], 0x0000ffff          ; 基地址为0，段界限为0xFFFFF

mov  dword [GDT\_START\_ADDRESS+0x0c], 0x00cf9200          ; 粒度为4KB，存储器段描述符

                                                         ; 建立保护模式下的堆栈段描述符

mov  dword [GDT\_START\_ADDRESS+0x10], 0x00000000          ; 基地址为0x00000000，界限0x0

mov  dword [GDT\_START\_ADDRESS+0x14], 0x00409600          ; 粒度为1个字节

                                                         ; 建立保护模式下的显存描述符

mov  dword [GDT\_START\_ADDRESS+0x18], 0x80007fff          ; 基地址为0x000B8000，界限0x07FFF

mov  dword [GDT\_START\_ADDRESS+0x1c], 0x0040920b          ; 粒度为字节

                                                         ; 创建保护模式下平坦模式代码段描述符

mov  dword [GDT\_START\_ADDRESS+0x20], 0x0000ffff          ; 基地址为0，段界限为0xFFFFF

mov  dword [GDT\_START\_ADDRESS+0x24], 0x00cf9800          ; 粒度为4kb，代码段描述符

mov  word [pgdt], 39                                     ; 描述符表的界限

lgdt [pgdt]

in   al, 0x92                                            ; 南桥芯片内的端口

or   al, 0000\_0010B

out  0x92, al                                            ; 打开A20

cli                                                      ; 中断机制尚未工作

mov  eax, cr0

or   eax, 1

mov  cr0, eax                                            ; 设置PE位

jmp  dword CODE\_SELECTOR:protect\_mode\_begin

; 16位的描述符选择子：32位偏移

; 清流水线并串行化处理器

[bits 32]

protect\_mode\_begin:

mov  eax, DATA\_SELECTOR                                  ; 加载数据段(0..4GB)选择子

mov  ds,  eax

mov  es,  eax

mov  eax, STACK\_SELECTOR

mov  ss,  eax

mov  eax, VIDEO\_SELECTOR

mov  gs,  eax

mov  ecx, protect\_mode\_tag\_end - protect\_mode\_tag

mov  ebx, 80 \* 2

mov  esi, protect\_mode\_tag

mov  ah,  0x3

output\_protect\_mode\_tag:

    mov  al,           [esi]

    mov  word[gs:ebx], ax

    add  ebx,          2

    inc  esi

    loop output\_protect\_mode\_tag

jmp $                                                    ; 死循环

bootloader\_tag db 'run bootloader'

bootloader\_tag\_end:

pgdt           dw 0

               dd GDT\_START\_ADDRESS

protect\_mode\_tag db 'enter protect mode'

protect\_mode\_tag\_end:

### Assignment 3

call clear\_screen

; 用清屏函数

while:

    ; 输出第一个字符

    movzx eax, byte [pos\_1]

    movzx ebx, byte [pos\_1 + 1]

    movzx ecx, byte [string\_index\_1]

    movzx edx, byte [color\_1]

    call display\_char

    ; 更新颜色

    inc byte [color\_1]

    ; 更新字符索引

    inc byte [string\_index\_1]

    movzx eax, byte [string\_index\_1]

    mov bl, byte [string + eax]

    cmp bl, 0

    jne reset\_index\_end\_1

    mov byte [string\_index\_1], 0

    reset\_index\_end\_1:

    ; 更新第一个字符的位置

    call update\_position\_1

    ; 输出第二个字符

    movzx eax, byte [pos\_2]

    movzx ebx, byte [pos\_2 + 1]

    movzx ecx, byte [string\_index\_2]

    movzx edx, byte [color\_2]

    call display\_char

    ; 更新颜色

    inc byte [color\_2]

    ; 更新字符索引

    inc byte [string\_index\_2]

    movzx eax, byte [string\_index\_2]

    mov bl, byte [string + eax]

    cmp bl, 0

    jne reset\_index\_end\_2

    mov byte [string\_index\_2], 0

    reset\_index\_end\_2:

    ; 更新第二个字符的位置

    call update\_position\_2

    ; 显示标题

    call display\_title

    ; 延时

    mov ecx, 0x500000

    delay\_loop:

        loop delay\_loop

    jmp while

clear\_screen:

    push eax

    push ecx

    push edi

    mov eax, 0x0720

    mov ecx, 80\*25

    mov edi, 0

    clear\_loop:

        mov word [gs:edi], ax

        add edi, 2

        loop clear\_loop

    pop edi

    pop ecx

    pop eax

    ret

display\_char:

    ; eax: 行号

    ; ebx: 列号

    ; ecx: 字符索引

    ; edx: 颜色

    push eax

    push ebx

    push ecx

    push edx

    push edi

    ; 计算显存偏移

    imul eax, 80

    add eax, ebx

    shl eax, 1

    mov edi, eax

    ; 获取字符

    mov al, byte [string + ecx]

    mov ah, dl

    ; 写入显存

    mov word [gs:edi], ax

    pop edi

    pop edx

    pop ecx

    pop ebx

    pop eax

    ret

update\_position\_1:

    push eax

    push ebx

    ; 计算新位置

    mov al, byte [pos\_1]

    mov bl, byte [pos\_1 + 1]

    add al, byte [velocity\_1]

    add bl, byte [velocity\_1 + 1]

    ; 检查行边界

    cmp al, 0

    jl reverse\_velocity\_x\_1

    cmp al, 24

    jg reverse\_velocity\_x\_1

    jmp reverse\_velocity\_x\_end\_1

    reverse\_velocity\_x\_1:

        neg byte [velocity\_1]

        mov al, byte [pos\_1]

        add al, byte [velocity\_1]

    reverse\_velocity\_x\_end\_1:

    ; 检查列边界

    cmp bl, 0

    jl reverse\_velocity\_y\_1

    cmp bl, 79

    jg reverse\_velocity\_y\_1

    jmp reverse\_velocity\_y\_end\_1

    reverse\_velocity\_y\_1:

        neg byte [velocity\_1 + 1]

        mov bl, byte [pos\_1 + 1]

        add bl, byte [velocity\_1 + 1]

    reverse\_velocity\_y\_end\_1:

    ; 保存新位置

    mov byte [pos\_1], al

    mov byte [pos\_1 + 1], bl

    pop ebx

    pop eax

    ret

update\_position\_2:

    push eax

    push ebx

    ; 计算新位置

    mov al, byte [pos\_2]

    mov bl, byte [pos\_2 + 1]

    add al, byte [velocity\_2]

    add bl, byte [velocity\_2 + 1]

    ; 检查行边界

    cmp al, 0

    jl reverse\_velocity\_x\_2

    cmp al, 24

    jg reverse\_velocity\_x\_2

    jmp reverse\_velocity\_x\_end\_2

    reverse\_velocity\_x\_2:

        neg byte [velocity\_2]

        mov al, byte [pos\_2]

        add al, byte [velocity\_2]

    reverse\_velocity\_x\_end\_2:

    ; 检查列边界

    cmp bl, 0

    jl reverse\_velocity\_y\_2

    cmp bl, 79

    jg reverse\_velocity\_y\_2

    jmp reverse\_velocity\_y\_end\_2

    reverse\_velocity\_y\_2:

        neg byte [velocity\_2 + 1]

        mov bl, byte [pos\_2 + 1]

        add bl, byte [velocity\_2 + 1]

    reverse\_velocity\_y\_end\_2:

    ; 保存新位置

    mov byte [pos\_2], al

    mov byte [pos\_2 + 1], bl

    pop ebx

    pop eax

    ret

display\_title:

    push eax

    push esi

    push edi

    mov eax, 0

    mov esi, 0

    mov edi, 64

    display\_title\_loop:

        mov cl, byte [title + esi]

        cmp cl, 0

        je display\_title\_end

        ; 写入字符到显存

        mov al, cl

        mov ah, 2

        mov word [gs:edi], ax

        inc esi

        add edi, 2

        jmp display\_title\_loop

    display\_title\_end:

    pop edi

    pop esi

    pop eax

    ret

jmp $

title db 'test 00000000',0

string db '1357924680',0

string\_index\_1 dw 0

color\_1 db 0

pos\_1 db 2,0

velocity\_1 db 1,1

string\_index\_2 dw 0

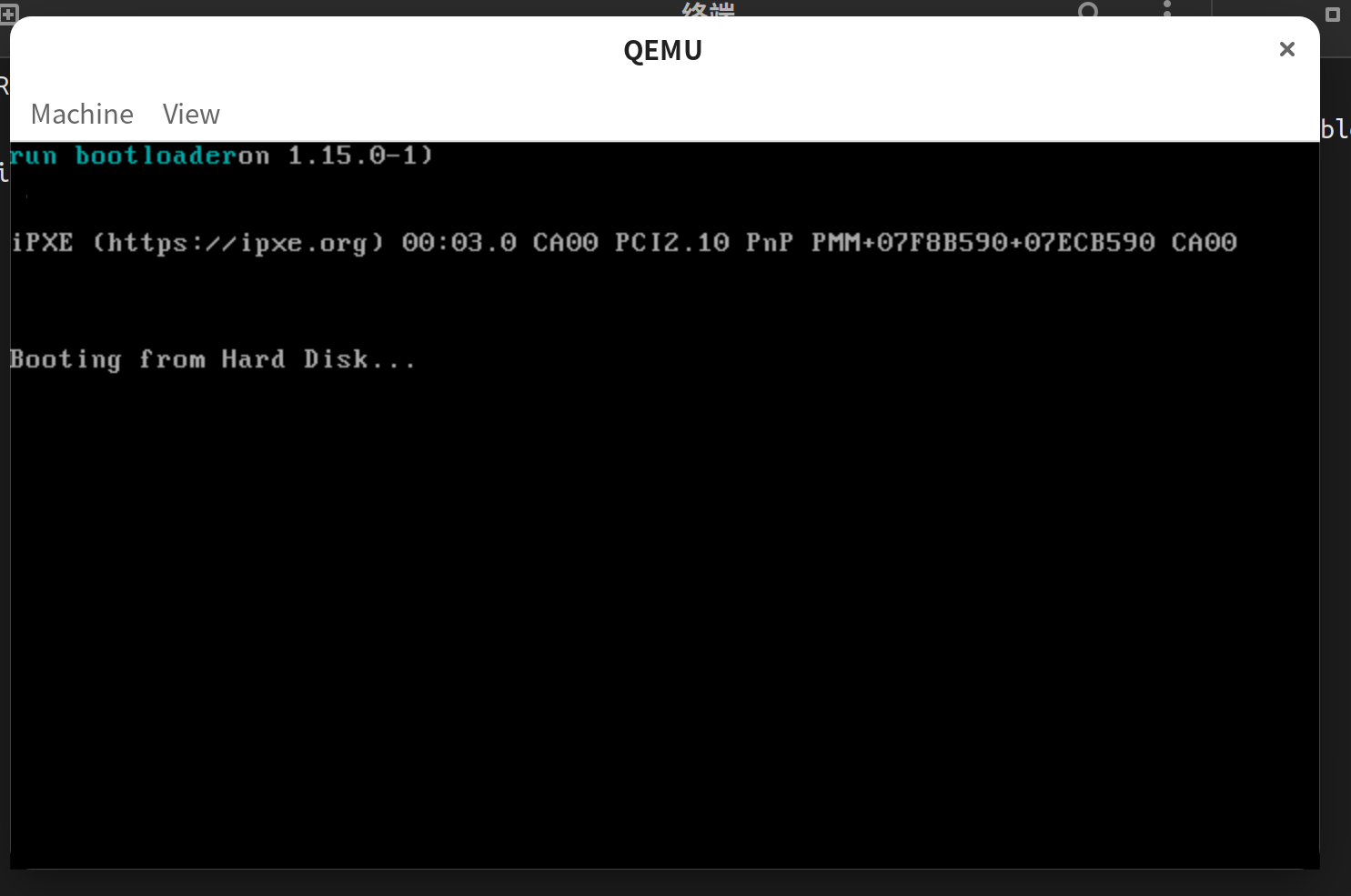
color\_2 db 0x21

pos\_2 db 3,79

velocity\_2 db -1,-1

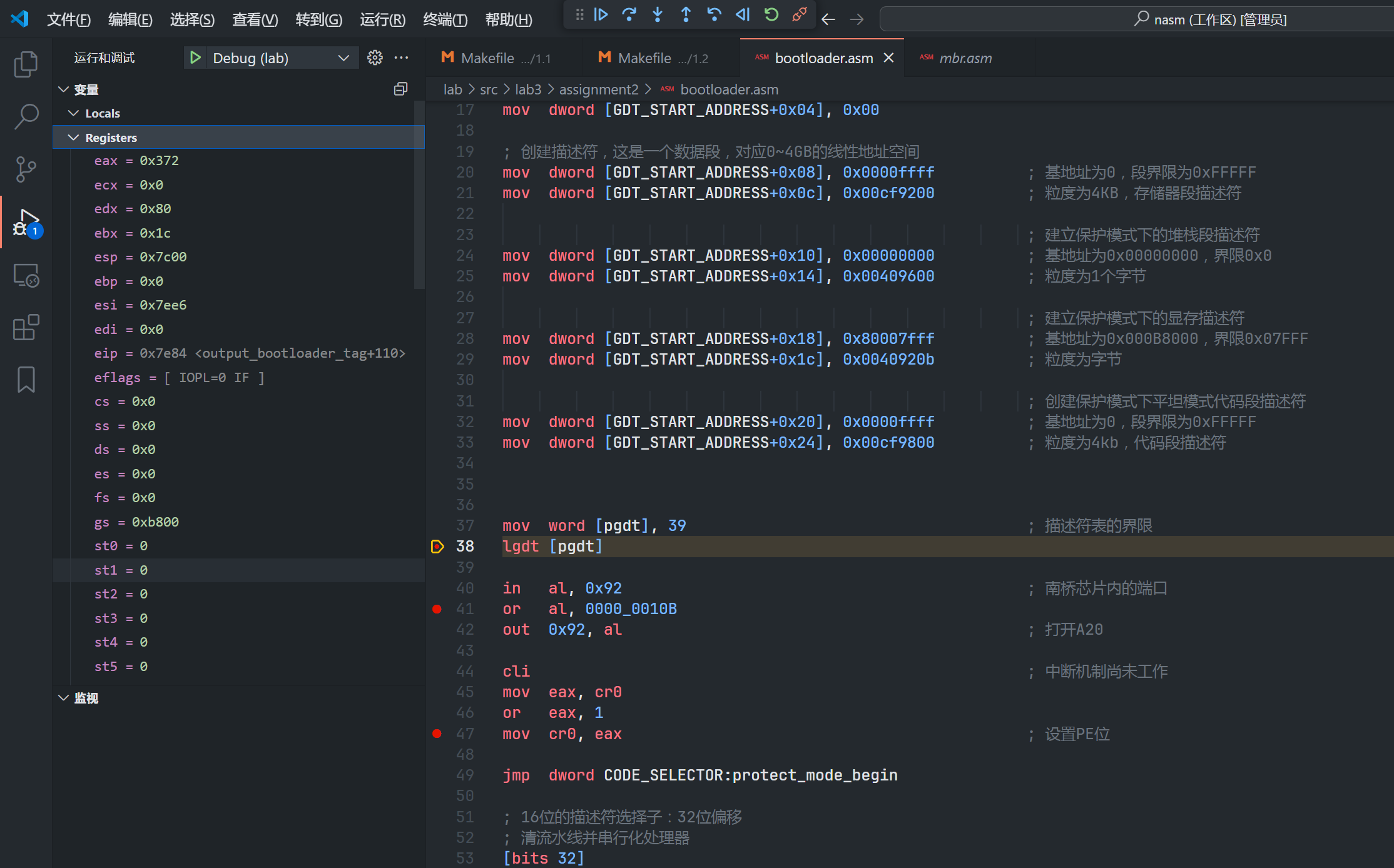
## 实验结果

### Assignment 1

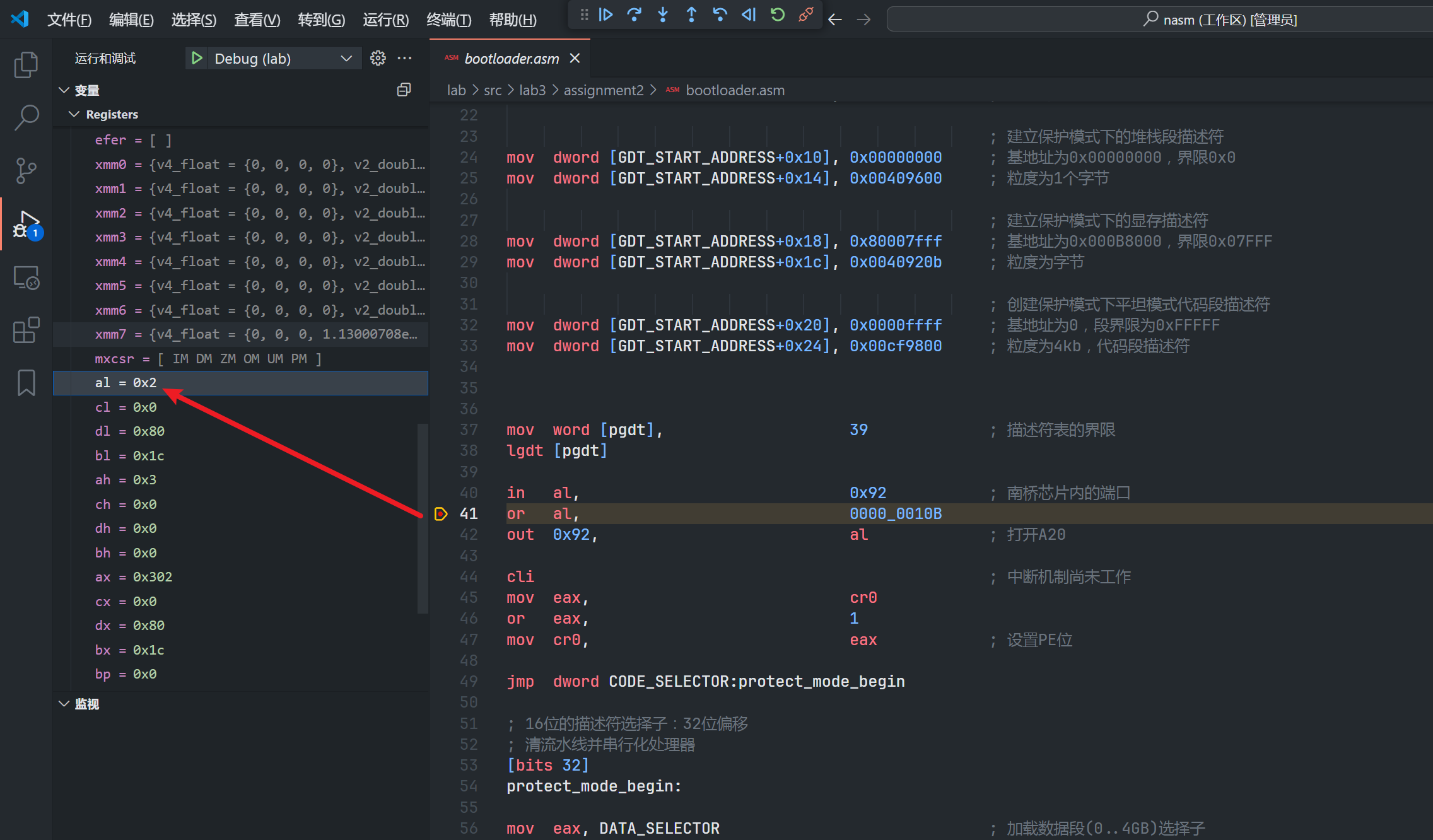


### Assignment 2

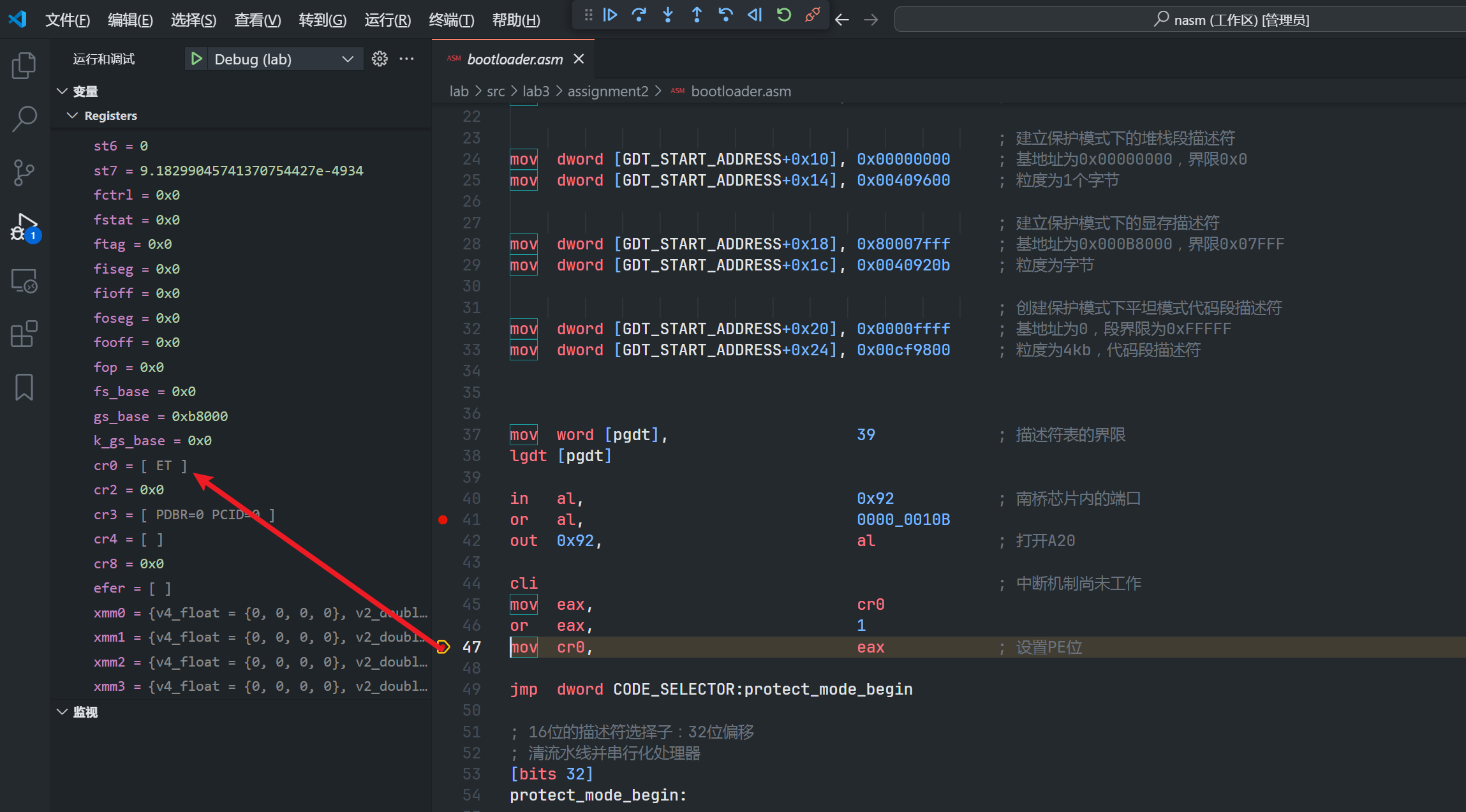
加载gdt：

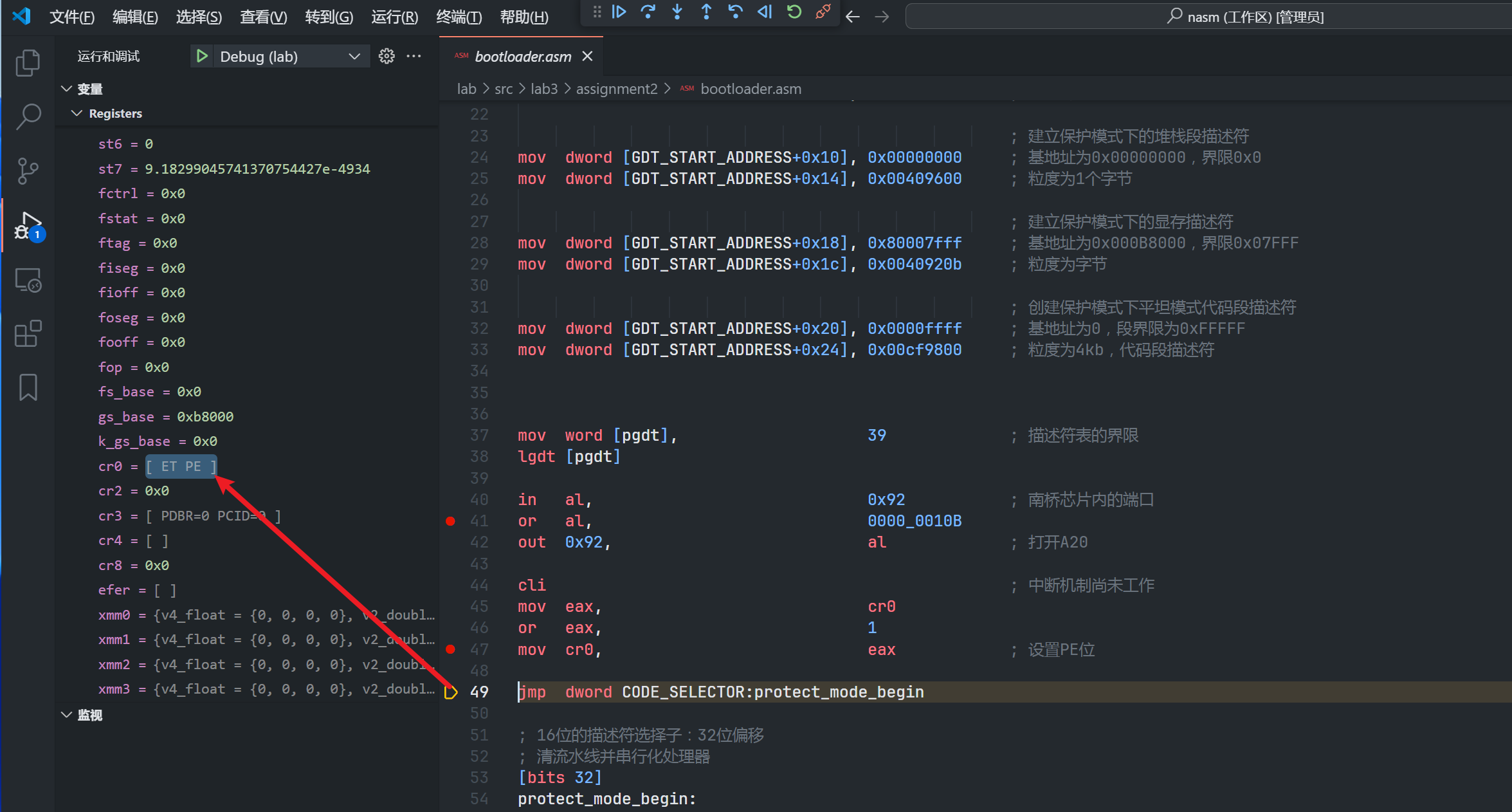


尝试打开A20：

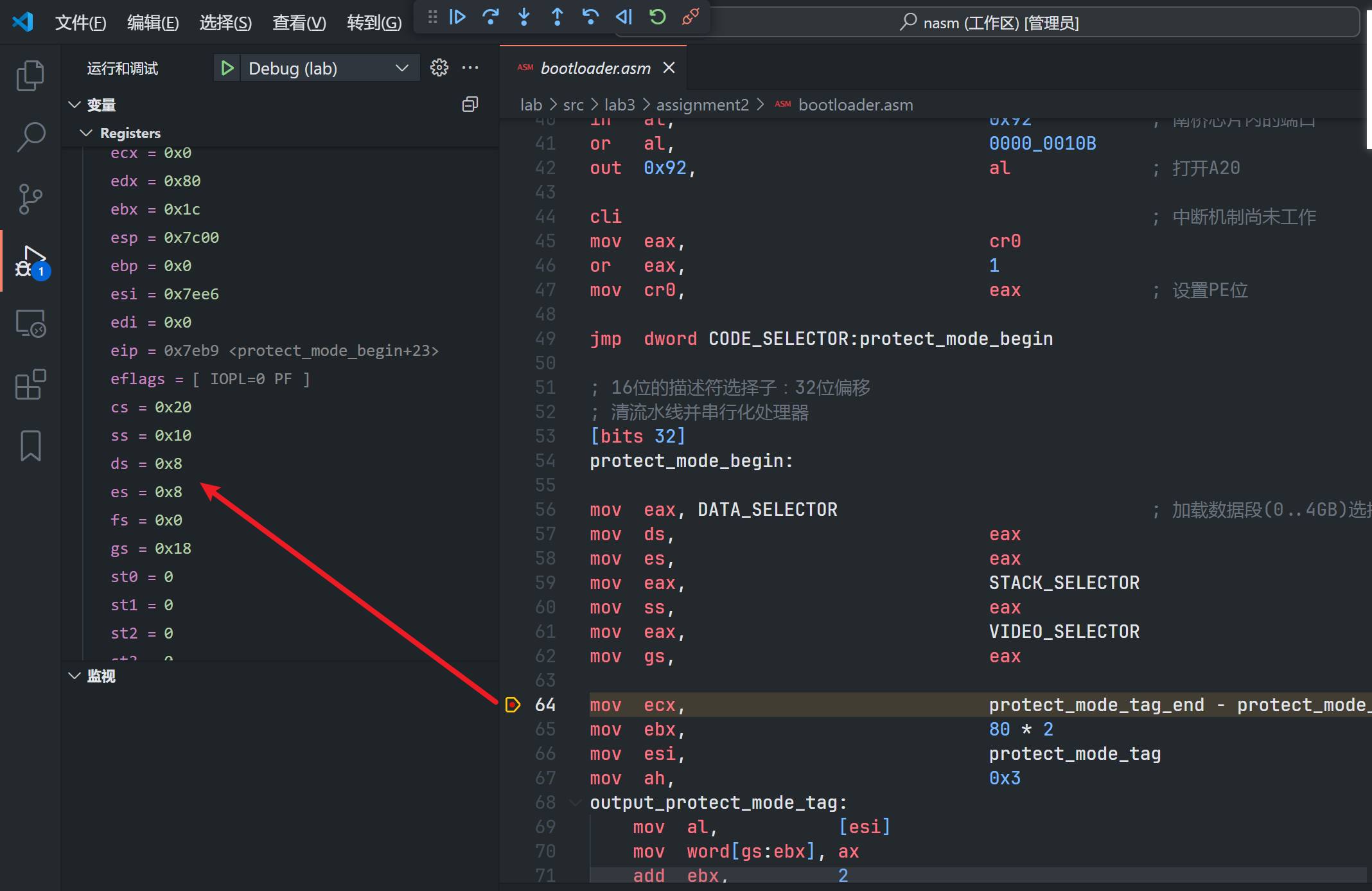


打开cr0保护模式标记位：

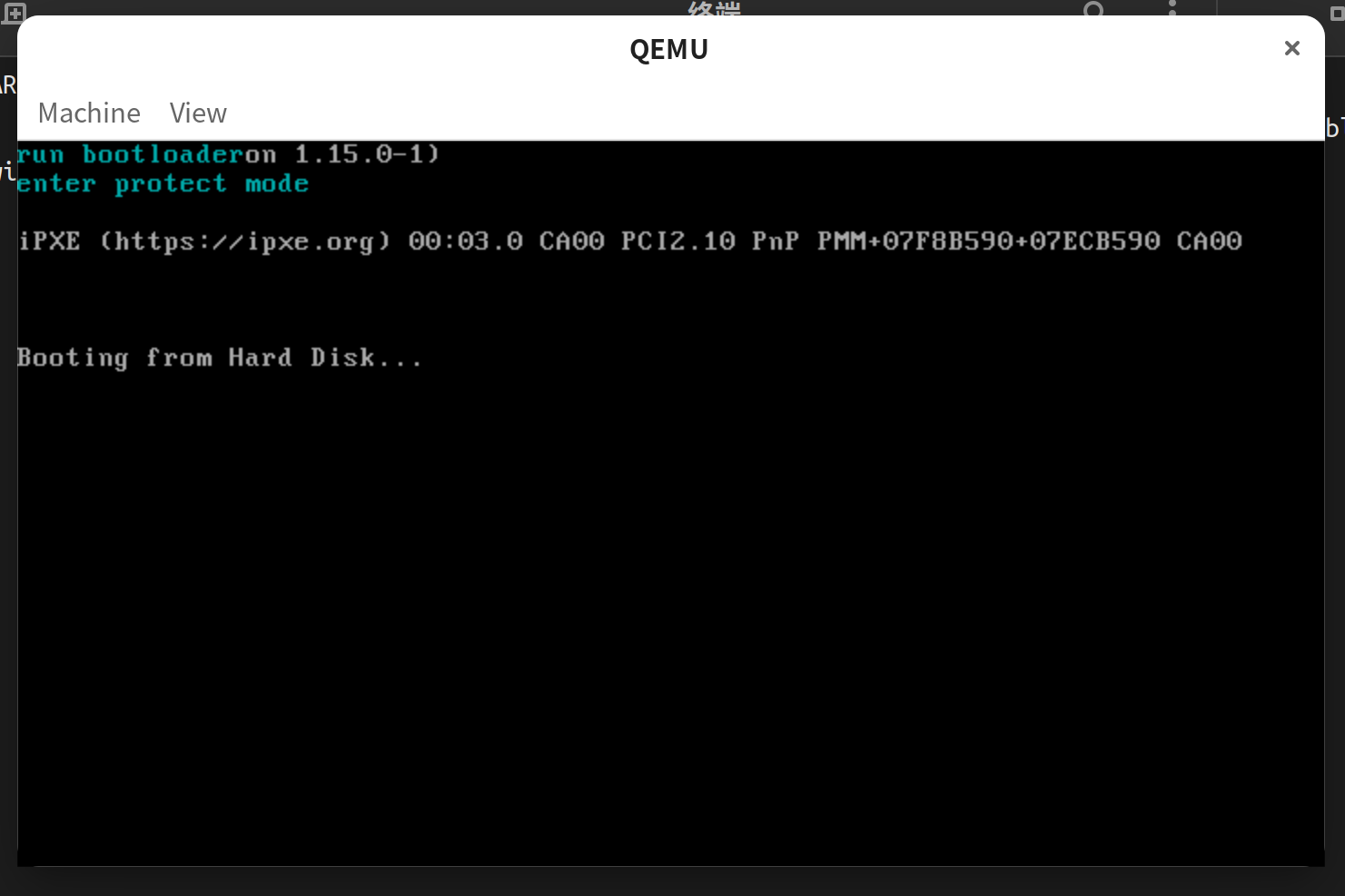




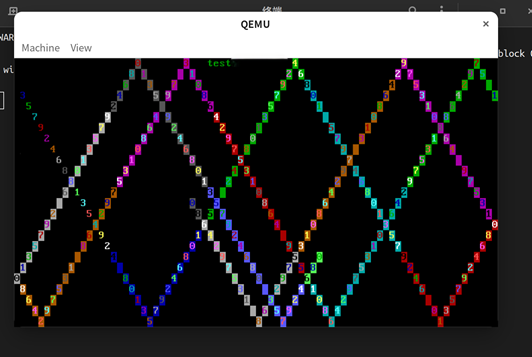
远跳转进入保护模式，加载段选择子：



运行结果：



### Assignment 3



## 总结

这次实验学习了如何从16位实模式跳转到32位保护模式，也初步学会了如何与IO设备交互。实验过程中最大的挑战是如何配置远程调试，配置过程中一度出现配置与指导书几乎完全相同但是跑不通的情况，最后发现是磁盘文件使用的是上一节的大小（512字节，仅能容纳mbr程序），除此之外还遇到了调试器工作目录没配置好导致无法找到源代码等问题，这些问题看似微小，却耗费了大量时间进行排查。希望在接下来的实验中能够对配置的细节进行细致的遵守。