一、 实验目的

1、熟练掌握调用系统功能的几个函数，特别是输出字符，输出字符串，输入字符等。

2、熟练掌握串的相关运算，对 ASCII 码有一定的了解，特别是 A(0x41),a(0x61),1(0x30),CR(0x0D),LF(0x0A)有熟练的记忆，可以通过系统功能输出相应的字符，能够熟练掌握大小写字符的判断和转换。

3、熟练掌握分支、循环结构程序的设计。能够使用 REP 前缀来简化循环结构程序的设计。

4、熟练掌握数字与 10 进制、16 进制字符串的转换。

5、为了更好的实现模块化的设计，使用了 call,ret 命令进行子模块的设计。此时将会使用 push 和 pop 命令来实现“保存现场”的功能。

实验设备

操作设备：个人 PC Window10 系统

模拟环境：Emu8086

二、 实验内容和要求

1、将从 BUF 单元开始的 10 个字节存储单元的小写字符改为大写字符，并统计小写字符的个数，并显示在屏幕上。

2、给定三个无符号数（字或字节），将其中的最大值存入 MAX 单元并在屏幕上显示。

3、给定三个有符号数（字或字节），将其中的最大值存入 MAX 单元并在屏幕上显示。

三、 实验步骤

实验 1:

实验内容

将从 BUF 单元开始的 10 个字节存储单元的小写字符改为大写字符，并统计小写字符的个数，并显示在屏幕上。

设计思路：

使用 SI 和 DI 循环 LAD 和 STO 字符，每次循环时，进行计数和字符转换操作。

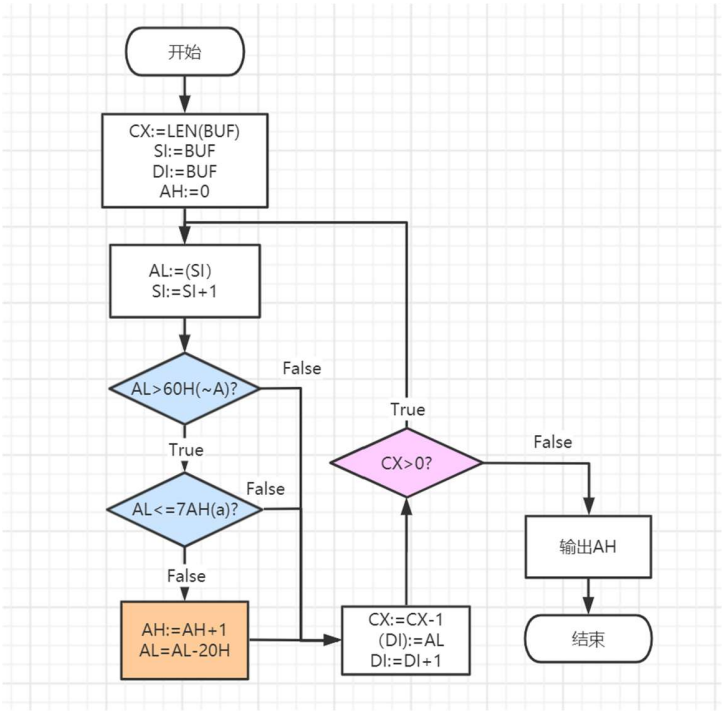

图 5-1 实验 1 的流程图

上面程色为关键的代码，蓝色标识循环内部的分支判断，粉红色标识循环。

代码：

```
data segment
    ; add your data here!
    pkey db "press any key...$"
    buf db "HeLloWorLd"
    buf_len equ $-buf
    numbers db "0123456789"
ends

stack segment
    dw   128  dup(0)
ends

code segment
start:
; set segment registers:
    mov ax, data
    mov ds, ax
    mov es, ax
```

```
    ; 初始化
    mov cx, buf_len
    lea si, buf
    lea di, buf
    mov ah, 0

; 计数循环体
cal_character:
    lodsb ; 读取一个字符
    cmp al,61h
    jb sto_buf ; 与 61H(a)比较，若<61H(a)，则跳到 sto_buf 处。
    cmp al,7ah
    ja sto_buf ; 与 7AH(z)比较，若>7AH(z)，则跳到 sto_buf 处。
    ; 执行小写转大写代码，并计数
    sub al,20h
    inc ah
sto_buf:
    stosb ; 写入一个字符
    loop cal_character

    ; 输出 ah
    ; 最大值为 10，当 ah=10 时输出 10，ah<10 时，输出"30H+al"(转化为字符串)
    cmp ah, 10
    jb print_branch
    mov dl, 31h ;输出字符 1
    mov ah, 2
    int 21h
    mov dl, 30h
    int 21h
    jmp out_print
print_branch:
    add ah, 30h
    mov dl, ah
    mov ah, 2
    int 21h
out_print:

    lea dx, pkey
    mov ah, 9
    int 21h        ; output string at ds:dx

    ; wait for any key....
    mov ah, 1
    int 21h
```

```
    mov ax, 4c00h ; exit to operating system.
    int 21h
ends

end start ; set entry point and stop the assembler.
```

实验结果：

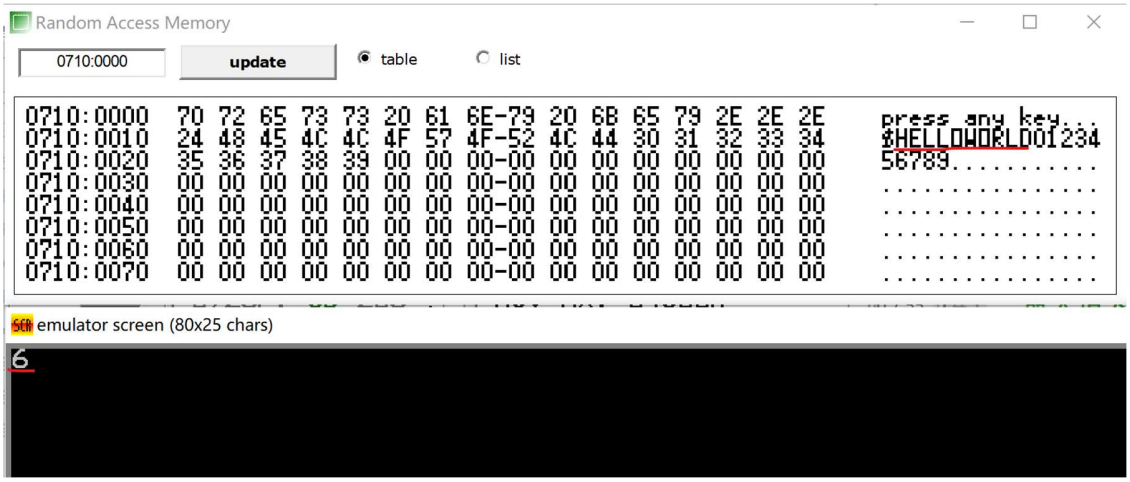

图 5-2 BUF="HelloWorLd"的实验结果。实验结果正确

@Module(print_number)（无符号数以 10 进制显示）：

因为在后面的几个实验中，都需要用到显示数字的功能，因此在这里将会对数字转化为 10 进制字符串的代码做一个补充。这里显示无符号字转 10 进制字符串的代码。

实验思路：整体的思路为除 10 取余法。一开始 CX=0（存储数字的长度），AX（存储数字），在堆栈中存储每一位数字（虽然可能有点浪费空间）。

输入输出：Input(AX=需要进行十进制输出的数字)，Output(Console)

流程图：

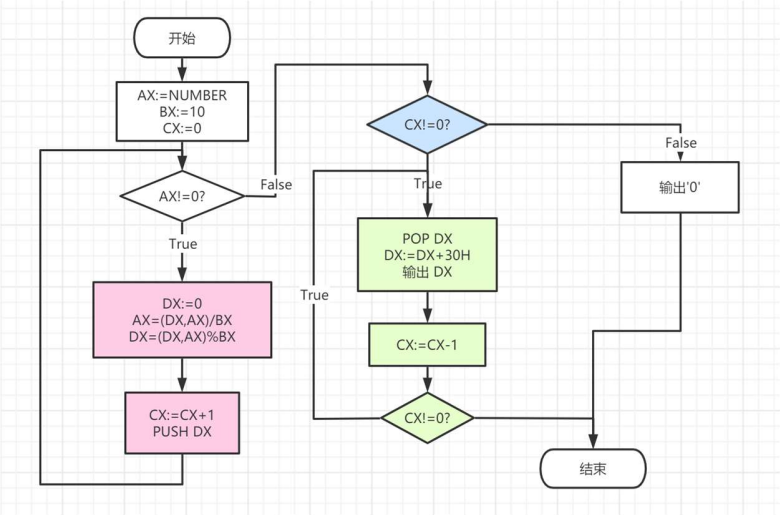

图 5-3 输出无符号字（10 进制）的流程图

实验代码：

```
; multi-segment executable file template.

data segment
    ; add your data here!
    pkey db "press any key...$"
ends

stack segment
    dw   128  dup(0)
ends

code segment
start:
; set segment registers:
    mov ax, data
    mov ds, ax
    mov es, ax

    ; 用于测试
    mov ax, 1323
    call print_number

    lea dx, pkey
    mov ah, 9
    int 21h        ; output string at ds:dx

    ; wait for any key....
```

```
    mov ah, 1
    int 21h

    mov ax, 4c00h ; exit to operating system.
    int 21h

; 子模块，输入(AX)，输出为屏幕。
print_number:
    ; 保存数据
    push bx
    push cx
    push dx

    mov cx, 0
    mov bx, 10

loop_div_number:
    cmp ax, 0
    jz branch_show_number ; 如果 ax!=0 继续执行取数

    mov dx, 0 ; 扩展无符号数
    div bx ; 除 10
    inc cx
    push dx ; 将中间的数字压入堆栈
    jmp loop_div_number ; 无条件循环，必须使用 jmp

branch_show_number:
    cmp cx, 0
    jz print_number_0

loop_print_number:
    pop dx
    add dl, 30h
    mov ah, 2
    int 21h ; 输出堆栈中栈顶的数字
    loop loop_print_number
    jmp print_number_out
print_number_0:
    mov dl, 30H
    mov ah, 2
    int 21h ; 输出字符'0'

print_number_out:
    ; 恢复数据
    pop dx
```

```
    pop cx
    pop bx
    ret
ends

end start ; set entry point and stop the assembler.
```

实验结果：

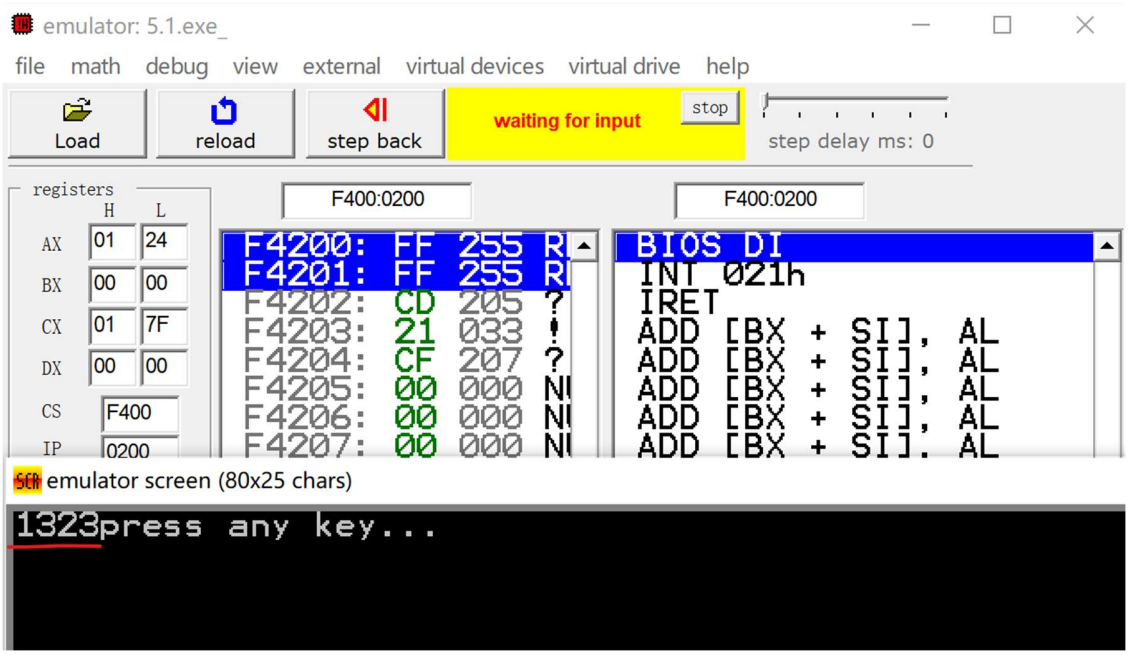

图 5-4 输出无符号字(10 进制)的实验结果，以 1323 为输入样例。

实验 2:

@Using(Module(print_number))，此实验将会使用输出无符号字（10 进制）模块。

实验内容：给定三个无符号数（字或字节），将其中的最大值存入 MAX 单元并在屏幕上显示。

实验思路：本实验中以字进行实验，实验思路为先假定第一个数为最大值，从第二个开始，直到最后一个，如果此数比第一个数大，则将此数替换最大值。如此往复，在最后一定会得到一个最大值。

流程图：

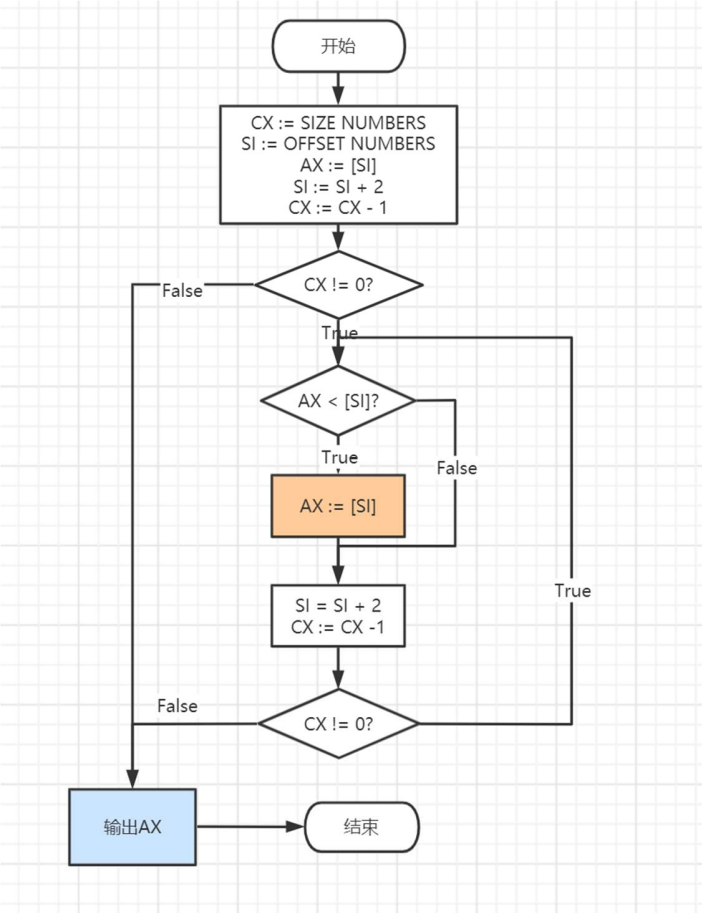

图 5-5 实验 2 计算 3 个无符号数的最大值并显示的流程图

其中程色会关键代码，浅蓝色部分将会调用输出十进制数的模块。

实验代码：

```
; multi-segment executable file template.

data segment
    ; add your data here!
    pkey db "press any key...$"
    numbers dw 1324,2875,1230
    numbers_len db ($-numbers)/2
    number_width dw 2
ends

stack segment
    dw   128  dup(0)
ends

code segment
```

```
start:
; set segment registers:
    mov ax, data
    mov ds, ax
    mov es, ax

    mov ch, 0
    mov cl, numbers_len ; cx 为数字的个数
    lea si, numbers ; si 执行第一个元素
    lodsw
    dec cx ; 移动第一个元素到 ax
    jcxz print_ax ; 如果只有一个数，则输出
loop_cmp_number:
    cmp ax, [si]
    jnb branch_cmp_number_out
    mov ax, [si] ; 当 ax > [si] 进行比较
branch_cmp_number_out:
    add si, number_width  ; si 指向下一个元素
    loop loop_cmp_number

print_ax:
    call print_number

    lea dx, pkey
    mov ah, 9
    int 21h        ; output string at ds:dx

    ; wait for any key....
    mov ah, 1
    int 21h

    mov ax, 4c00h ; exit to operating system.
    int 21h

; 子模块，输入(AX)，输出为屏幕。
print_number:
    ;;; 输出无符号数（10 进制的代码）请见模块 print_number
ends

end start ; set entry point and stop the assembler.
```

实验结果：

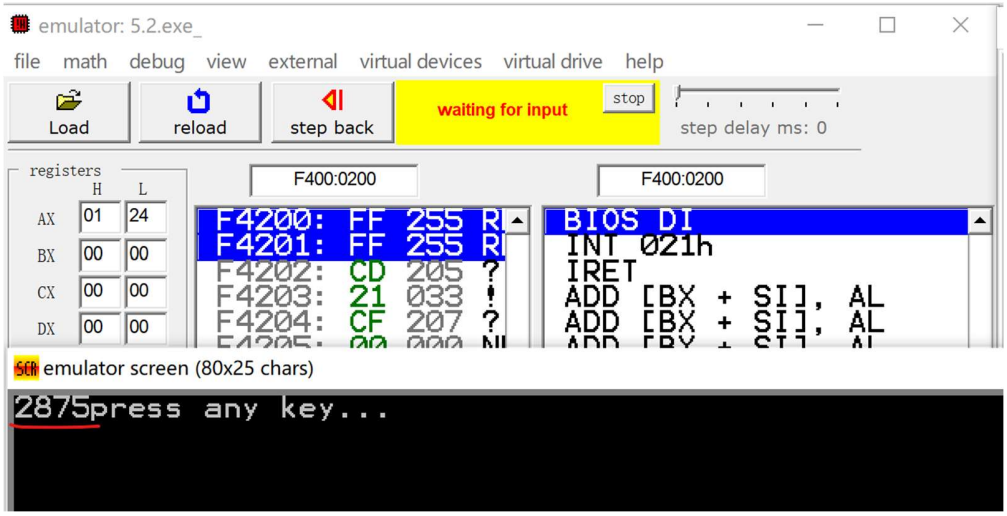

图 5-6 实验 2 的实验结果，实验数据为(1324, 2875, 1230)

@Module(print_number_signal)（有符号数以 10 进制显示）：

此模块依赖于 Module(print_number)

有符号数的十进制数显示可以通过无符号数来实现，若该数不小于 0，则可以直接调用 print_number。若该数小于，则各位取反+1 并调用 print_number 来输出。

实验代码：

```
; multi-segment executable file template.

data segment
    ; add your data here!
    pkey db "press any key...$"
ends

stack segment
    dw   128  dup(0)
ends

code segment
start:
; set segment registers:
    mov ax, data
    mov ds, ax
    mov es, ax

    mov ax, -1324
    call print_number_flag
```

```
    lea dx, pkey
    mov ah, 9
    int 21h        ; output string at ds:dx

    ; wait for any key....
    mov ah, 1
    int 21h

    mov ax, 4c00h ; exit to operating system.
    int 21h

; 子模块，有符号数以 10 进制显示，输入(AX,数字)，输出(Console)。
print_number_flag:
    push dx ; 暂存 dx
    cmp ax, 0
    jge call_print_number

    push ax ; 暂存 ax
    mov dl, 2dh
    mov ah, 2
    int 21h ; 输出一个'-'(0x2d)字符
    ; 转化为相应的正数
    pop ax ; 恢复 ax
    xor ax, 0ffffh ; 所有位取反
    inc ax ; +1
call_print_number:
    call print_number
    pop dx
    ret

; 子模块，无符号数以 10 进制显示，输入(AX,数字)，输出(Console)。
print_number:
    ;;; 输出无符号数（10 进制的代码）请见模块 print_number
ends

end start ; set entry point and stop the assembler.
```

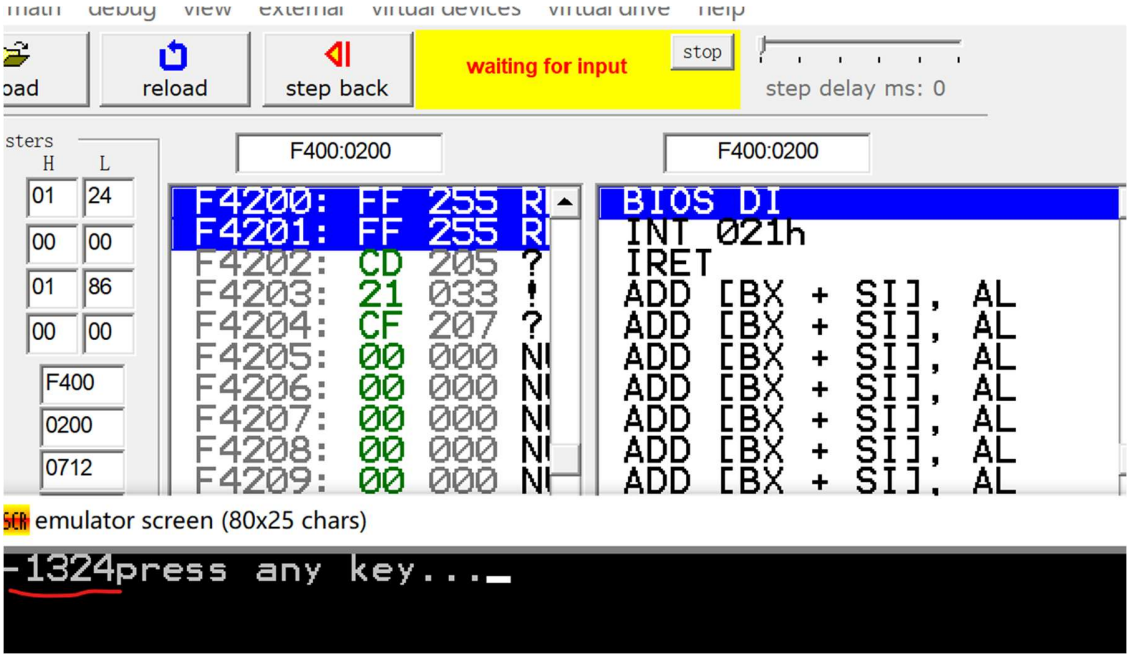

图 5-7 模块 2 的实验结果，以-1324 为实验样例。

实验 3:

实验思路：

实验 3 和实验 2 的区别只有有符号和无符号的区别，所以两者的代码十分相似。因此，在本实验中，不再进行实验思路和流程图的说明，将会直接给出相应的关键代码和实验结果。

实验代码：

```
; multi-segment executable file template.

data segment
    ; add your data here!
    pkey db "press any key...$"
    numbers dw -1324,-2875,-1230
    numbers_len db ($-numbers)/2
    number_width dw 2
ends

stack segment
    dw   128  dup(0)
ends

code segment
start:
```

```
; set segment registers:
    mov ax, data
    mov ds, ax
    mov es, ax

    mov ch, 0
    mov cl, numbers_len ; cx 为数字的个数
    lea si, numbers ; si 执行第一个元素
    lodsw
    dec cx ; 移动第一个元素到 ax
    jcxz print_ax ; 如果只有一个数，则输出
loop_cmp_number:
    cmp ax, [si]
    jnl branch_cmp_number_out ; jnl 是有符号数的比较方式。
    mov ax, [si] ; 当 ax > [si] 进行比较
branch_cmp_number_out:
    add si, number_width  ; si 指向下一个元素
    loop loop_cmp_number

print_ax:
    call print_number_flag

    lea dx, pkey
    mov ah, 9
    int 21h        ; output string at ds:dx

    ; wait for any key....
    mov ah, 1
    int 21h

    mov ax, 4c00h ; exit to operating system.
    int 21h

; 子模块，有符号数以 10 进制显示，输入(AX,数字)，输出(Console)。
print_number_flag:
    ;;; 代码请见 Module(print_number_flag);
ends

end start ; set entry point and stop the assembler.
```

实验结果：

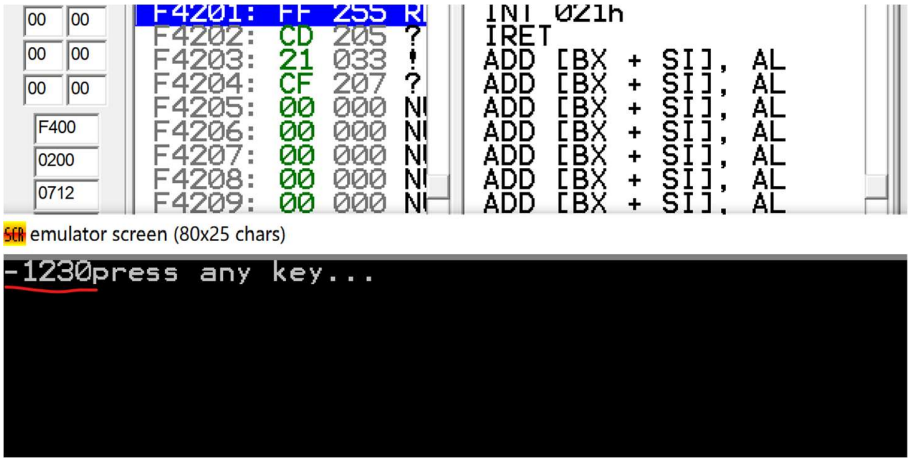

图 5-8 实验 3 的实验结果，以（-1324,-2875,-1230）为实验样例。

四、 实验结果分析

实验结果分析已经并入实验步骤中。

五、 结果讨论

1、后续的两个实验中，由于将无符号数和有符号数以 10 进制的形式进行输出的代码已经可以通过一个函数的概念来进行描绘。在通过搜索引擎查看汇编中的 call、ret，发现可以用来实现类似的功能。

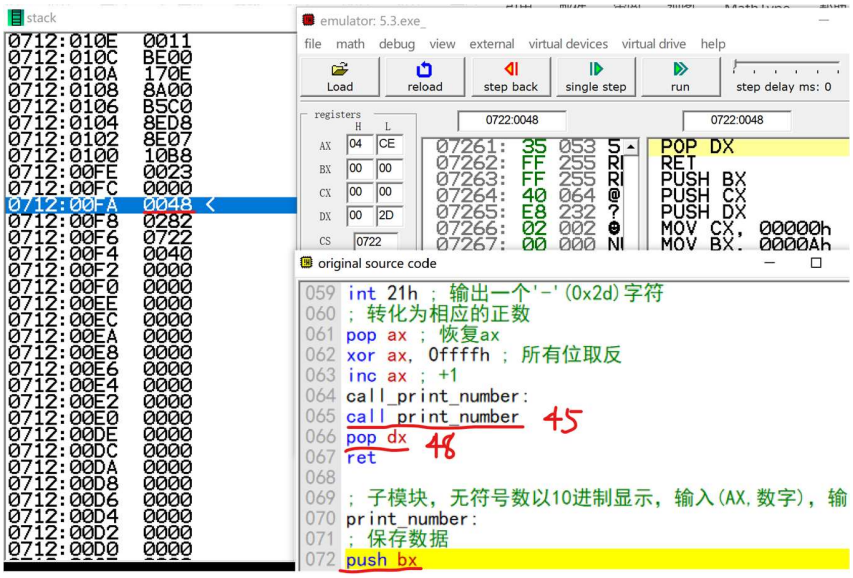

图 5-9 调用 call 命令之后的程序运行状况

程序执行到 `call print_number` 时，此时 `IP=0045H`，该命令行占了 3 个字节。调用后堆栈顶的数据为 `0048H`，刚好是子函数执行完之后返回的位置。在被调函数内

部，使用 ret 将会回到 0048H，即执行完子函数之后下一跳命令的位置，这也能说明 call 函数的功能。

2、有时候，为了提高效率，有时候需要尽可能地使用寄存器来进行运算。有时候也是遇到寄存器数量不够的问题（例如存在多个计数器，但是只有一个 CX 的情况），这个时候就需要将 CX 的数据暂存起来，以便子代码段执行时破坏 CX 数据后能够进行恢复。这在子模块（子函数）的设计过程中是十分关键的，在子函数定义了所需要用到的变量（主要是使用的寄存器）之后，在子模块中，其他寄存器的写入操作都可能破坏中间的数据。这是就需要使用 push~pop 配对来实现变量的暂存。在本实验中多次使用了这个技巧。这里以 print_number 子模块为例。需要注意的是，push 和 pop 需要满足嵌套原则。否则可能出现数据恢复异常和 ret 跳转异常。

```
; 子模块，输入(AX)，输出为屏幕。
print_number:
    ; 保存数据
    push bx
    push cx
    push dx

    ;;;中间是执行的代码
    ; 恢复数据
    pop dx
    pop cx
    pop bx
    ret
```

3、多模块的设计可以充分发挥程序的复用性。特别是常见的场景（例如输入一行字符串，将数字进行打印等），应当尽可能的使用到子模块的设计方式，特别是多次调用此函数时，这种方式相对于复制粘贴代码来说可以减少工作量的同时保证代码的执行效率。