一、 实验目的

1、熟练掌握循环和分支程序设计。特别是分支判断的相关用法要熟练掌握。
2、熟练掌握使用 call 和 ret 来实现子模块的设计，能够使用 push 和 pop 在保存中间的数据。
3、熟练掌握系统中断程序的调用。
4、注意字节类型和子类型的区别，在程序设计时应充分考虑数据类型对程序设计的影响。

二、 实验设备

操作系统：个人 PC Windows10 系统
模拟器环境：emu 8086

三、 实验内容和要求

1、在内存 score 中存放有 10 个学生的成绩数据，成绩为无符号字节数。假设学生成绩在[90，100]区间为优秀，在[80，89]区间为良好，在[70，79]区间为中等，在[60，69]区间为及格，低于 60 为不及格。要求统计出不及格、及格、中等、良好、优秀的人数，分别送入 Notpassed, Passed, Good, Better, Best 字节单元。 确认，各区间的人数之和是否是 10。

2、数据段定义如下：

```
DATAX  DW  2316H

DATAY  DW  0237H

DATA   DW  ?
```

键盘输入 A-C 之间的一个字符，

a) 当输入字符为 A，则计算 X+Y，并存入 DATA 单元
b) 当输入字符为 B，则计算|X-Y|，并存入 DATA 单元
c) 当输入字符为 C，则计算 X*Y，并存入 DATA 单元
d) 当输入其他字符，则显示字符串“您的输入有误，请重新输入！”

四、 实验步骤

实验 1

实验内容：

在内存 score 中存放有 10 个学生的成绩数据，成绩为无符号字节数。假设学生成绩在[90，100]区间为优秀，在[80，89]区间为良好，在[70，79]区间为中等，在[60，69]区间为及格，低于 60 为不及格。要求统计出不及格、及格、中等、良好、优秀的人数，分别送入 Notpassed, Passed, Good, Better, Best 字节单元。 确认，各区间的人数之和是否是 10。

设计思路：在 score 的第一个单元存放学生的个数，后面几个单元存放数据。然后依次取数据，先调用模块 1，判断出应当送到的位置，再调用模块 2，将数据存放到相应的位置。模块 1 的代码主要是多层次的分支判断语句，模块 2 的代码主要是写入相应位置的语句，两个模块并不复杂，但是再整个执行过程中，调用的次数非常多，因此使用了模块化的设计方式。

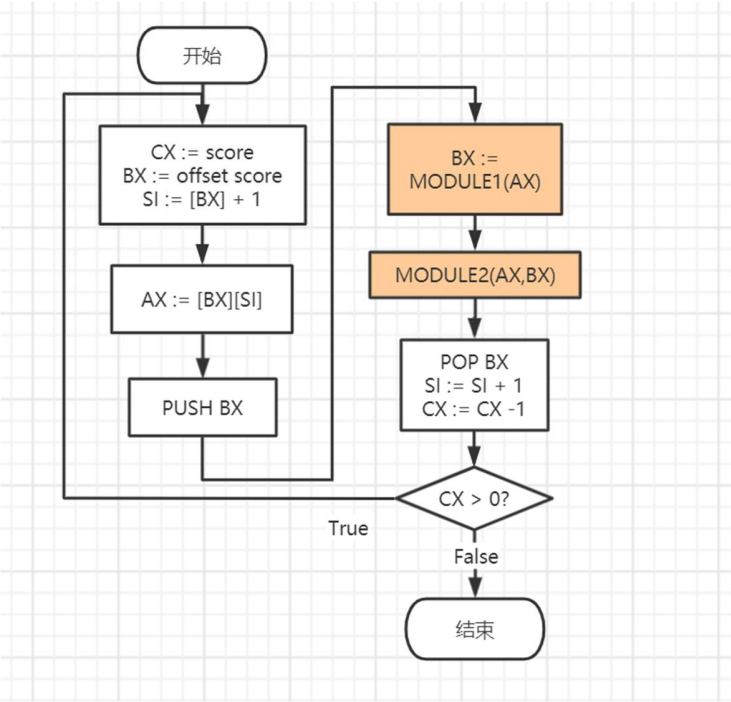

图 7-1 实验 1 的流程图

实验代码：

```
data segment
    pkey db "press any key...$"
    score db 10,87,92,47,88,69,72,58,92,100,84 ; 成绩的数据
    notpassed db 11 dup(0)
    passed db 11 dup(0)
    good db 11 dup(0)
    better db 11 dup(0)
    best db 11 dup(0)
ends
```

```
stack segment
    dw   128  dup(0)
ends

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

    ; add your code here
    mov cl, score
    mov ch, 0 ; cx=学生的个数
    lea bx, score ; bx=score 的首地址
    mov si, 1 ;第一个元素的地址 ; si=当前的成绩的变址
loop_grade:
    mov al,[bx][si] ; al=当前的成绩
    push bx
    call judge_grade
    call push_grade
    pop bx
    inc si ; 跳到下一个成绩
    loop loop_grade

    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
; 判断，输入 al(成绩),输出 bx(对应的存储段的基地址)
judge_grade:
    pushf
    cmp al, 80
    jb judge_cmp_70
    ; 80~100
    cmp al, 90
    jb judge_cmp_1
    ; 90~100
```

```
    lea bx, best
    jmp judge_out
judge_cmp_1:
    ; 80~89
    lea bx, better
    jmp judge_out
judge_cmp_70:
    ; 0~79
    cmp al, 70
    jb judge_cmp_60
    ; 70~79
    lea bx, good
    jmp judge_out
judge_cmp_60:
    ; 0~69
    cmp al, 60
    jb judge_cmp_2
    lea bx, passed
    jmp judge_out
judge_cmp_2:
    ; 0~59
    lea bx, notpassed
judge_out:
    popf
    ret
; 存储成绩，输入 al(成绩),bx(对应的存储段的基地址)
push_grade:
    push si
    push cx
    mov cl, [bx]
    mov ch, 0 ; cl=存储区内存储的个数
    mov si, cx
    inc si ; si=指向待存储的位置
    mov [bx][si], al ; 存入数据
    inc cl
    mov [bx],cl ; cl+1 后写入存储区内存储的个数
    pop cx
    pop si
    ret
ends

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

实验 2

实验题目：

DATAX DW 2316H

DATAY DW 0237H

DATA DW ?

键盘输入 A-C 之间的一个字符，

e) 当输入字符为 A，则计算 X+Y，并存入 DATA 单元
f) 当输入字符为 B，则计算|X-Y|，并存入 DATA 单元
g) 当输入字符为 C，则计算 X*Y，并存入 DATA 单元
h) 当输入其他字符，则显示字符串“您的输入有误，请重新输入！”

实验思路：
可以再主程序中进行判断，然后将 e~h 的各个操作写道子程序里面，分别调用。整个代码相对来说并不是很复杂。
实验代码：

```
data segment
    ; add your data here!
    pkey db "press any key...$"
    datax dw 2316h
    datay dw 0237h
    dataz dw ?
    error_msg db "你的输入有误，请重新输入"
ends

stack segment
    dw   128  dup(0)
ends

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

loop_input:
    mov ah, 1
    int 21h ; 输入一个字符
```

```
    cmp al, 'A'
    jne judge_b
    call func_add ; 执行 add 操作
    jmp judge_out
judge_b:
    cmp al, 'B'
    jne judge_c
    call func_sub ; 执行 sub 操作
    jmp judge_out
judge_c:
    cmp al, 'C'
    jne judge_other
    call func_mul ; 执行 mul 操作
    jmp judge_out
judge_other:
    call func_other ; 执行其他操作
    jmp loop_input
judge_out:

    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
func_add:
    push ax
    mov ax, datax
    add ax, datay
    mov dataz, ax
    pop ax
    ret
func_sub:
    push ax
    mov ax, datax
    sub ax, datay
    cmp ax, 0
    jge re
    xor ax, 0ffffh
    inc ax ; ax 取反+1
re:
```

```
    mov dataz, ax
    pop ax
    ret
func_mul:
    push ax
    push dx
    mov ax, datax
    mov dx, datay
    mul dx
    mov dataz, ax
    pop dx
    pop ax
    ret
func_other:
    lea dx, error_msg
    mov ah, 9
    int 21h
    ret
ends

end start
```

五、 实验结果分析

实验 1：

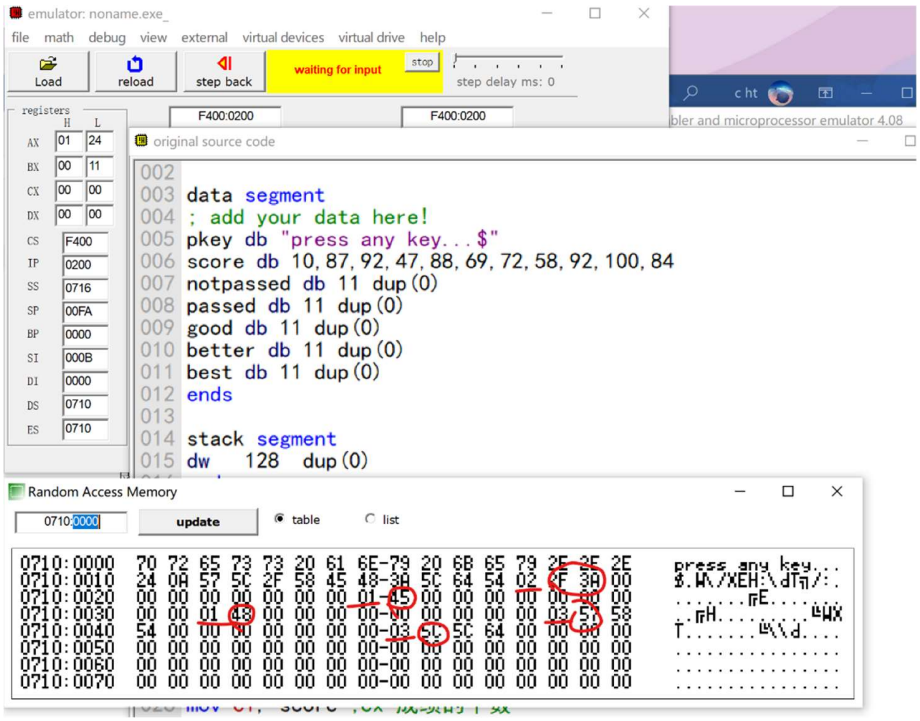

图 7-2 实验 1 的实验结果

从实验结果发现，原来的成绩分成了 5 组，分别为 2，1，1，3，3 个，和为 10，实验结果正确。

实验 2：

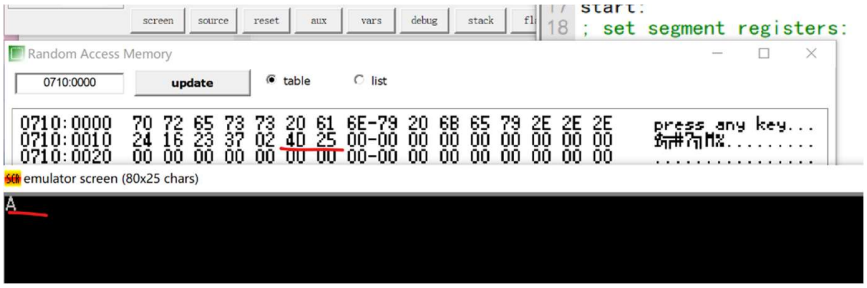

图 7-3 实验 2 的实验结果 1

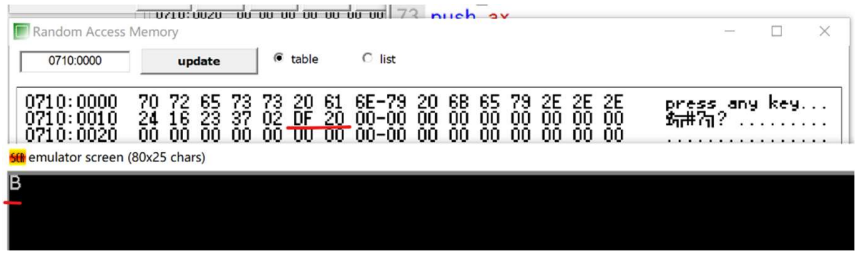

图 7-4 实验 2 的实验结果 2

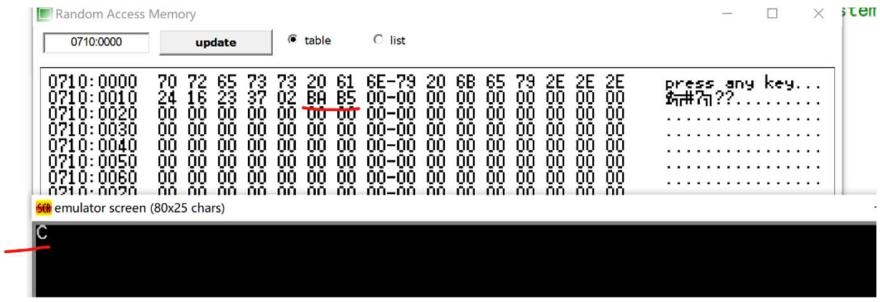

图 7-5 实验 2 的实验结果 3

经过计算器的运算，三组实验结果均正确。

六、 结果讨论

1、此次实验的重点在于思考如何设计子程序，是否有设计子程序的必要，以及如何设计子程序让子程序的适应性（或者）通用性更强。

2、子程序的传参方式一般借助于寄存器和堆栈，也可以使用地址表来传递，但是因为地址表来传递参数会影响子程序的扩展能力，因此并不是很推荐。

3、在子程序中，经常出现 push 和 pop 的操作来存储中间的数据，注意 push 和 pop 必须是成对且嵌套的，一般来说，如果涉及到 psw 的寄存器的更改也会使用 pushf 和 popf 来存储中间的操作。