山东大学

信息科学与工程学院

2020－2021 学年第二学期

实 验 报 告

课程名称： 微处理器原理与应用

实验名称： 子程序汇编 实验 学习和提高

专 业 班 级 通信工程 二班

学 生 学 号 201922121209

学 生 姓 名 陈泽宇

实 验 时 间 2021 年 3 月 27 日

实验报告

【实验目的】

1. 子程序汇编实验学习和提高

【实验要求】

1. 理解汇编语言中的 ASSUME 伪指令和标准的汇编程序
2. 复习 Debug 的使用，提高调试的能力
3. 编写、阅读若干汇编程序，提高编程能力

【实验具体内容】

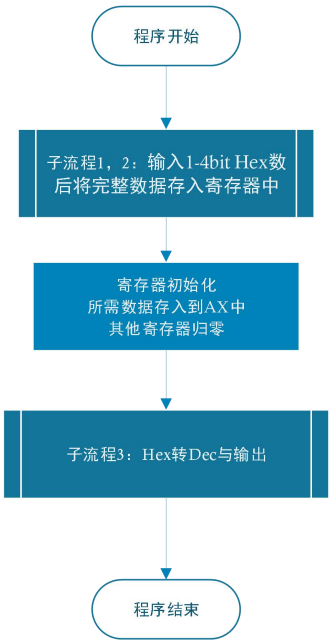
1. 复习一下 Debug -P 和-G
2. 将键盘上输入的十六进制数转换成十进制数，并在屏幕上显示。
   （编写程序，详细注释并画程序流程图）
3. 较为复杂的汇编实例学习：判断该年是否为闰年
   (通过注释重点学习并理解程序，画出程序的流程图，在理解的基础上，如果觉得程序的写法需要修改也可以自行修改)
4. 汇编实例学习与改进：两位数加法

【第一个实验：十六进制转十进制】

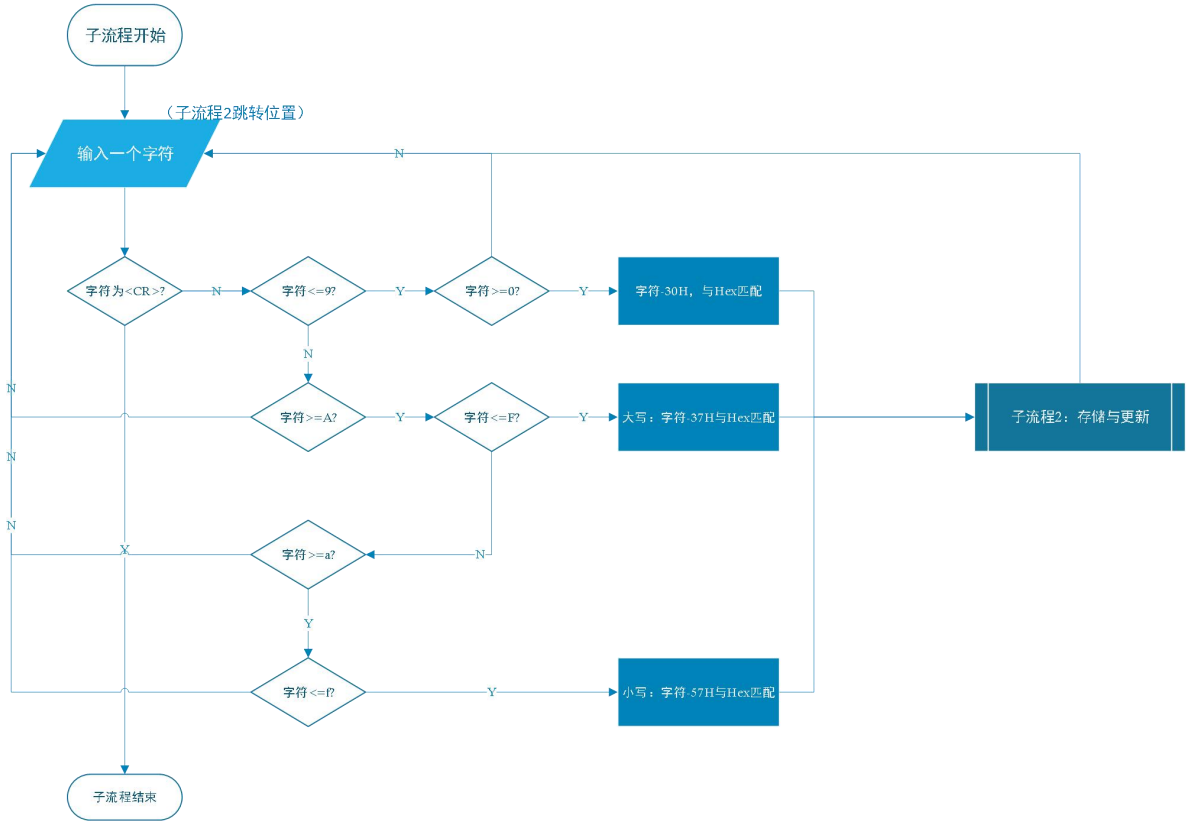
（1） 实验流程图

主流程：

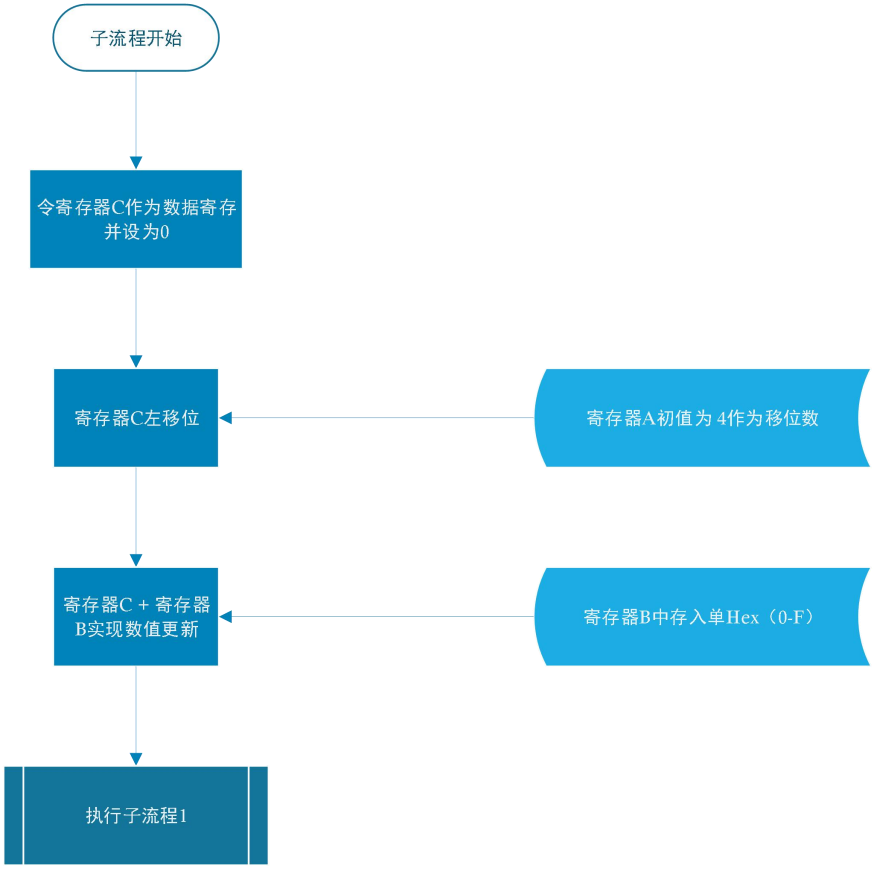
程序功能：
1.实现4位以下的十六进制转十进制并输出到屏幕
2.无效字符自动忽略（不读入）



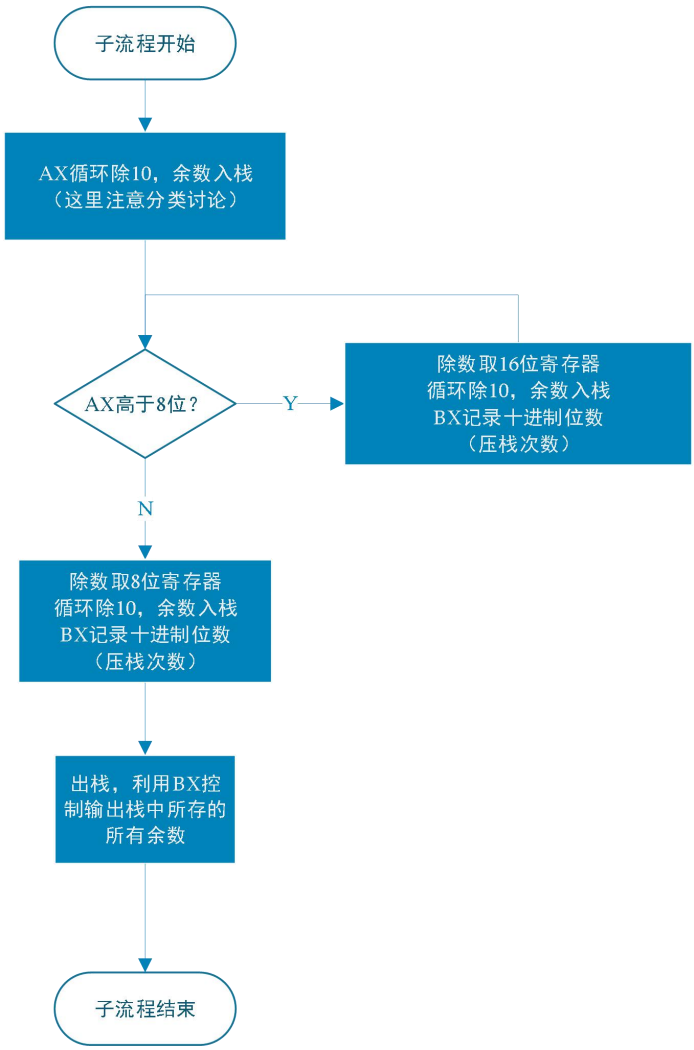
子流程 1：输入判定



子流程 2：存储与更新



子流程 3：数据处理与输出



（2）实验源代码（粘贴源代码）

```
; 程序输入无效字符不读入，例如 1jc=1c，只输入非法字符则输出 0
; 支持 4 位转化
; 借用了 2.2.1&2.2.2 的写法进行字母判断
; upd1：修复了程序没有考虑中间 0 的问题
; upd2：勉强支持部分 4 位 Hex 的进制转换，但还是有大量的 bug,逻辑不清晰（不上传）
; upd3：重构程序的逻辑，去掉了无用的的位数判断使得程序更为清晰
; upd4（Final）：细节优化，比如 FFFF 不显示的问题；完善程序注释
DATA SEGMENT
    STRING DB 13,10,"Please input a Hex number(Up to 4 bits):",'$'
DATA ENDS
STACKS SEGMENT
    DW 200 DUP(0)
STACKS ENDS
```

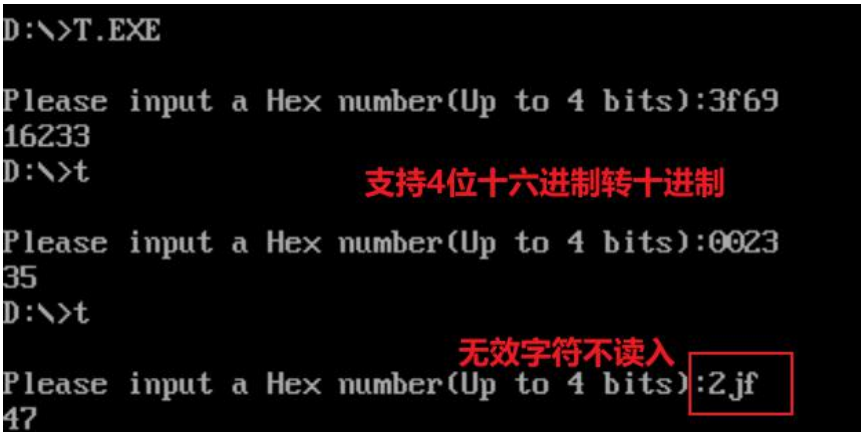
```
CODES SEGMENT
    ASSUME CS:CODES, DS:DATA, SS:STACKS
START:
WELCOME: ;输入提示信息
    MOV AX, DATA
    MOV BX, 0
    MOV DS, AX
    LEA DX, STRING
    MOV AH, 9H
    INT 21H
MAIN_INPUT:
    MOV DX,0
    MOV AH,1
    INT 21H                          ;系统等待输入一个字符，键入一个字符之后会自动转为 ASCII 值存入 AL 中
    MOV DL,AL                        ;向 DL 中写入 AL,DL 作为每次存储时新的一位 Hex
    CMP AL,0DH                       ;如果输入字符为回车则跳到标识符 Init 处执行，进行寄存器初始化
    JE Init
    CMP AL,39H
    JBE NUMBER                       ;如果=<9 则跳到标识符 NUMBER 处执行
    CMP AL,41H                       ;（>9 成立）如果>=A 则跳到 WORD_处执行
    JAE WORD_                        ;如上解析
    JMP MAIN_INPUT                   ;（<A 成立）继续输入字符
STORING_:
    MOV CL, 4                        ;向 CL 中写入 4，作为二进制下的逻辑右移位数，实现了 BX 十六进制角度上整体左移一位，起到了取高位的作用
    SHL BX, CL                       ;逻辑左移指令，实现了 BX 十六进制角度上整体左移一位
    ADD BX, DX                       ;BX 更新存储的十六进制数
    JMP MAIN_INPUT
NUMBER:                              ;字符 0-9
    CMP AL, 30H                      ;判断是否>=0，匹配成功则进一步执行，否则必然是除回车外的其他字符，进行返回字符重新输入
    JAE NUM_PROCING
    JMP MAIN_INPUT
NUM_PROCING:
```

```
    SUB DL, 30H
    JMP STORING_
WORD_:
    CMP AL, 46H                         ;大于 A 的情况下与 F 进行比较
    JBE WORD_PROCING_1                  ;<=F 的情况
    CMP AL, 61H                         ;(通过比较，大于 Z 的情况下)与 a 进行比
较
    JB MAIN_INPUT                       ;小于 a 的情况：其他字符，跳转重新输出
    ;大于等于 a 的情况
    CMP AL, 66H                         ;与 f 进行比较
    JBE WORD_PROCING_2                  ;<=f 成立则跳转
    JMP MAIN_INPUT                      ;不成立(>f)则说明是其他字符，重新输入
WORD_PROCING_1:
    SUB DL, 37H;大写字母转十六进制字
    JMP STORING_
WORD_PROCING_2:
    SUB DL, 57H;小写字母转十六进制字
    JMP STORING_
Init:
; 寄存器初始化，便于后续处理，AX 作为原始数据寄存器，BX 作为压栈次数寄存器，CX 和 DX
全部置零，为除法做准备
    MOV CX, 0
    MOV AX, BX
    MOV BX, DX
    MOV DX, 0
    MOV BX, 0
; 以上实现了输入 1-4bit Hex 数后将数据存入 AX 寄存器中，并初始化寄存器，BX 用于记录
十进制位数便于后续输出

; 下面进行连续/10 入栈运算处理进行分类讨论，难点在于对 DIV 的理解
    CMP AX, 0FFH
    JBE SIMPLE_PROCESS
    CMP AX, 0FFFFH
    JA ENDING
GENERAL:;一般的处理流程，针对除数为 16 位的情况，也是转化到 simple_process 中
    MOV CX, 0AH
    DIV CX
    PUSH DX;余数入栈
```

```
    ADD BX, 1;压栈次数记录，便于后续输出存在栈中的所有数字组成一个完整的十进制数
    MOV DX, 0;这是考虑到后续循环而采用的措施，DX 必须置零，否则会出现错误的结果
    CMP AX, 0FFH;这里也要进行二次/多次判断是否除数还是 16 位，因为两种情况的处理逻辑是不同的
    JAE GENERAL
SIMPLE_PROCESS:;简单 8 位处理情况
; 每次 DL 取 AH 中存放的余数后需要将 AH 置零
    MOV CX, 0
    MOV CL, 0AH
    DIV CL
    MOV DL, AH
    MOV AH, 0
    PUSH DX
    ADD BX, 1;压栈次数记录，便于后续输出存在栈中的所有数字组成一个完整的十进制数
    CMP AL, 0
    JNE SIMPLE_PROCESS
; 以上完成了转化为十进制数并入栈的工作
; 出栈输出操作，与前面的 BX 位数相联系进行输出即可，比较简单
Decimal_Disp:
    POP DX
    ADD DL, 30H;这里是为了正常输出数字，转换成 ASCII 的 Hex 形式
    MOV AH, 02H
    INT 21H
    SUB BX,1;压栈次数减一，类似起到循环控制的作用
    CMP BX,0;BX=0 则表明输出完成，否则继续输出
    JNE Decimal_Disp
ENDING:
    MOV AH, 4CH
    INT 21H
CODES ENDS
END START
```

（3）实验代码、过程、相应结果（截图）并对实验进行说明和分析：

代码运行结果如下所示



程序分析过程如下：（以 3f69 为例）

如流程图所示，整个程序可以大致分为以下几个阶段

1. 循环输入直到输入中止字符，输入结束后同时完成如下过程
   a) 无效字符不读入
   b) 数字、大小写字母统一转化为对应十六进制数存入寄存器内，
2. 寄存器初始化
3. 循环除 10 压栈，余数存入栈中，并记录十进制的位数便于输出十进制内容
4. 出栈输出十进制数

如下所示，在文本编辑器编辑完成后 Debug 装载程序

装载后如下图所示，利用-u 命令查看对应内容，可见命令已被写入对应的内存中

```
D:\>if exist T.exe c:\masm\debug T.exe
-u
0788:0000 B86C07        MOV     AX,076C
0788:0003 BB0000        MOV     BX,0000
0788:0006 8ED8          MOV     DS,AX
0788:0008 8D160000      LEA     DX,[0000]
0788:000C B409          MOV     AH,09
0788:000E CD21          INT     21
0788:0010 BA0000        MOV     DX,0000
0788:0013 B401          MOV     AH,01
0788:0015 CD21          INT     21
0788:0017 8AD0          MOV     DL,AL
0788:0019 3C0D          CMP     AL,0D
0788:001B 7435          JZ      0052
```

由于子流程中的某些过程分析已经在实验 2.2 中完成，因此后续会利用-g 命令进行跳转，仅对程序关键部分的调试分析以便于说明问题。

【输入判断、变换、存储过程分析】

与实验 2.2 中的思路相仿，这里简单叙述，以输入 f 为例，如下图所示，输入 f 后 AL=66H，为 f 的 ASCII Hex 形式

```
-g 17
跳转到输入部分执行
Please input a Hex number(Up to 4 bits):f
AX=0166  BX=0000  CX=0261  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076B  CS=0788  IP=0017   NV UP EI PL NZ NA PO NC
0788:0017 8AD0          MOV     DL,AL
```

下面进行判断流程，判断为合法后跳转到存储部分，这里不再赘述，内容与 2.2 判断逻辑完全相同，如下图所示，判断合法，并且转到了字母转化的部分

```
WORD_:
    CMP AL, 46H
    JBE WORD_PROCING_1
    CMP AL, 61H
    JB MAIN_INPUT
    ;大于等于a的情况
    CMP AL, 66H
    JBE WORD_PROCING_2
    JMP MAIN_INPUT
WORD_PROCING_1:
    SUB DL, 37H;大写字母转十六进
    JMP STORING_
WORD_PROCING_2:
    SUB DL, 57H;小写字母转十六进
    JMP STORING_
```

```
AX=0166  BX=0000  CX=0261  DX=0066  SP=0000  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076B  CS=0788  IP=0042   NV UP EI PL NZ NA PE NC
0788:0042 3C66          CMP     AL,66
-t

AX=0166  BX=0000  CX=0261  DX=0066  SP=0000  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076B  CS=0788  IP=0044   NV UP EI PL ZR NA PE NC
0788:0044 7607          JBE     004D
-t

AX=0166  BX=0000  CX=0261  DX=0066  SP=0000  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076B  CS=0788  IP=004D   NV UP EI PL ZR NA PE NC
0788:004D 80EA57        SUB     DL,57
-▲
可见跳转成功
```

根据代码内容，小写字母十六进制-57H 会转换成为等价十六进制存储，如下所示

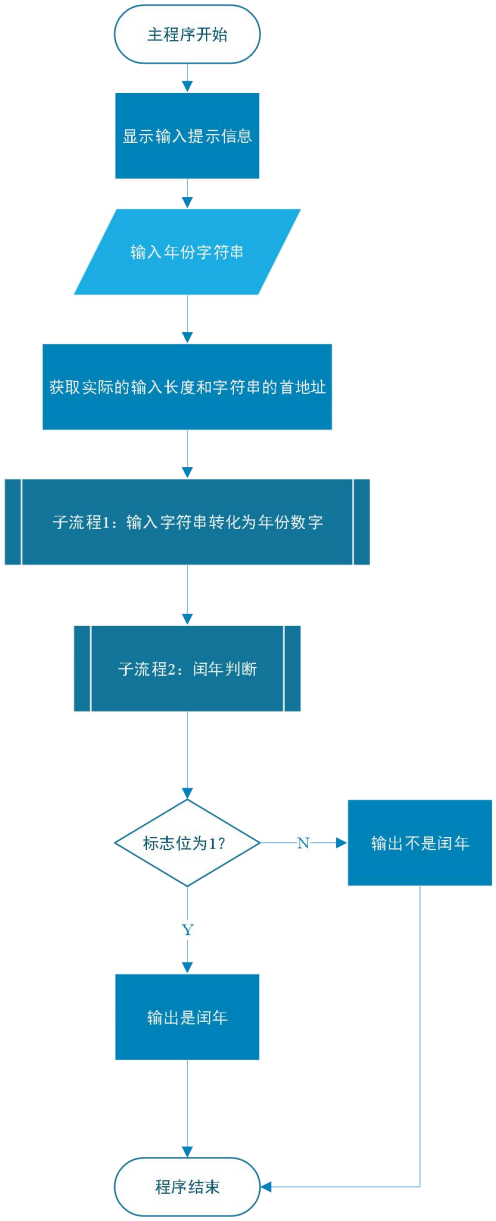
```
AX=0166  BX=0000  CX=0261  DX=0066  SP=0000  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076B  CS=0788  IP=004D   NV UP EI PL ZR NA PE NC
0788:004D 80EA57        SUB     DL,57
-t                                  f(literal)->F(Hex)

AX=0166  BX=0000  CX=0261  DX=000F  SP=0000  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076B  CS=0788  IP=0050   NV UP EI PL NZ AC PE NC
0788:0050 EBD5          JMP     0027
```

【存储】

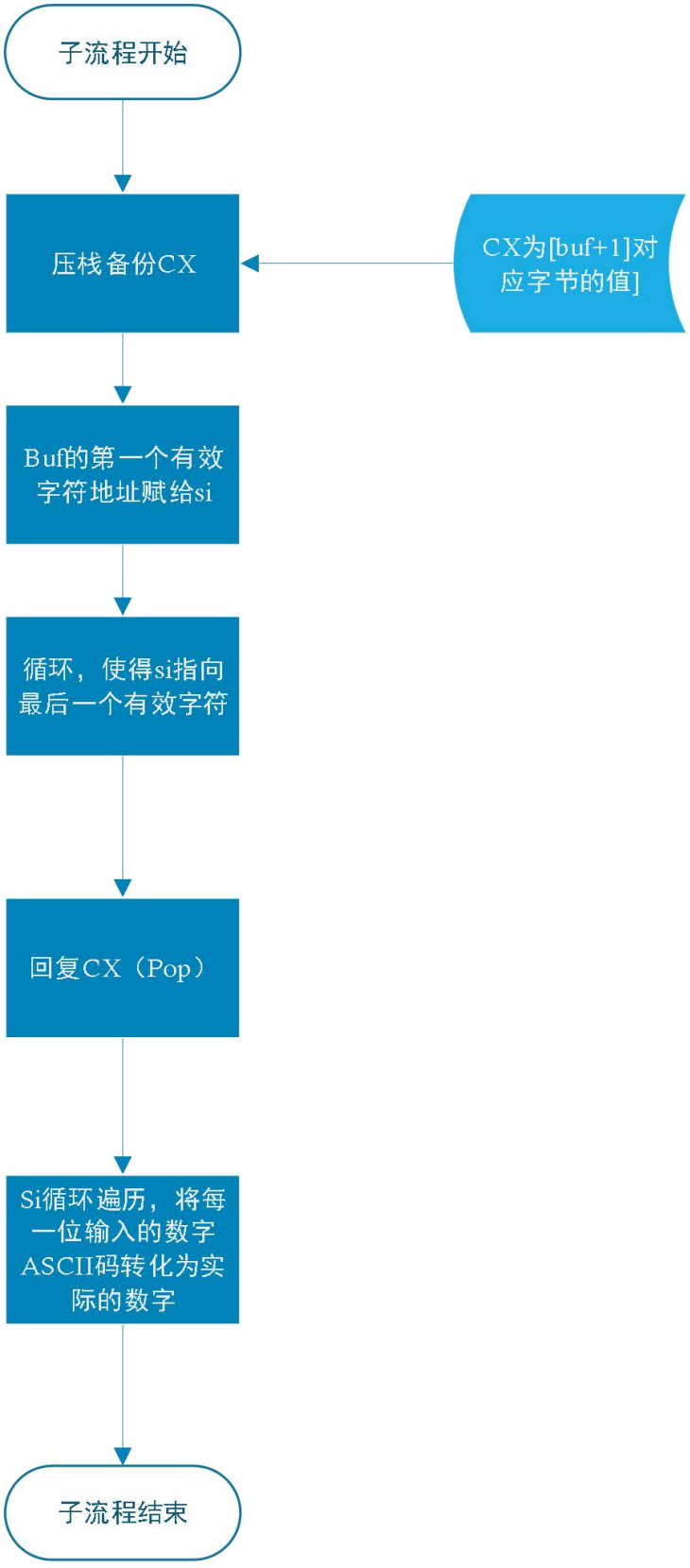
转化完成后，可见跳转到了存储流程

【第二个实验：闰年程序分析画流程图】

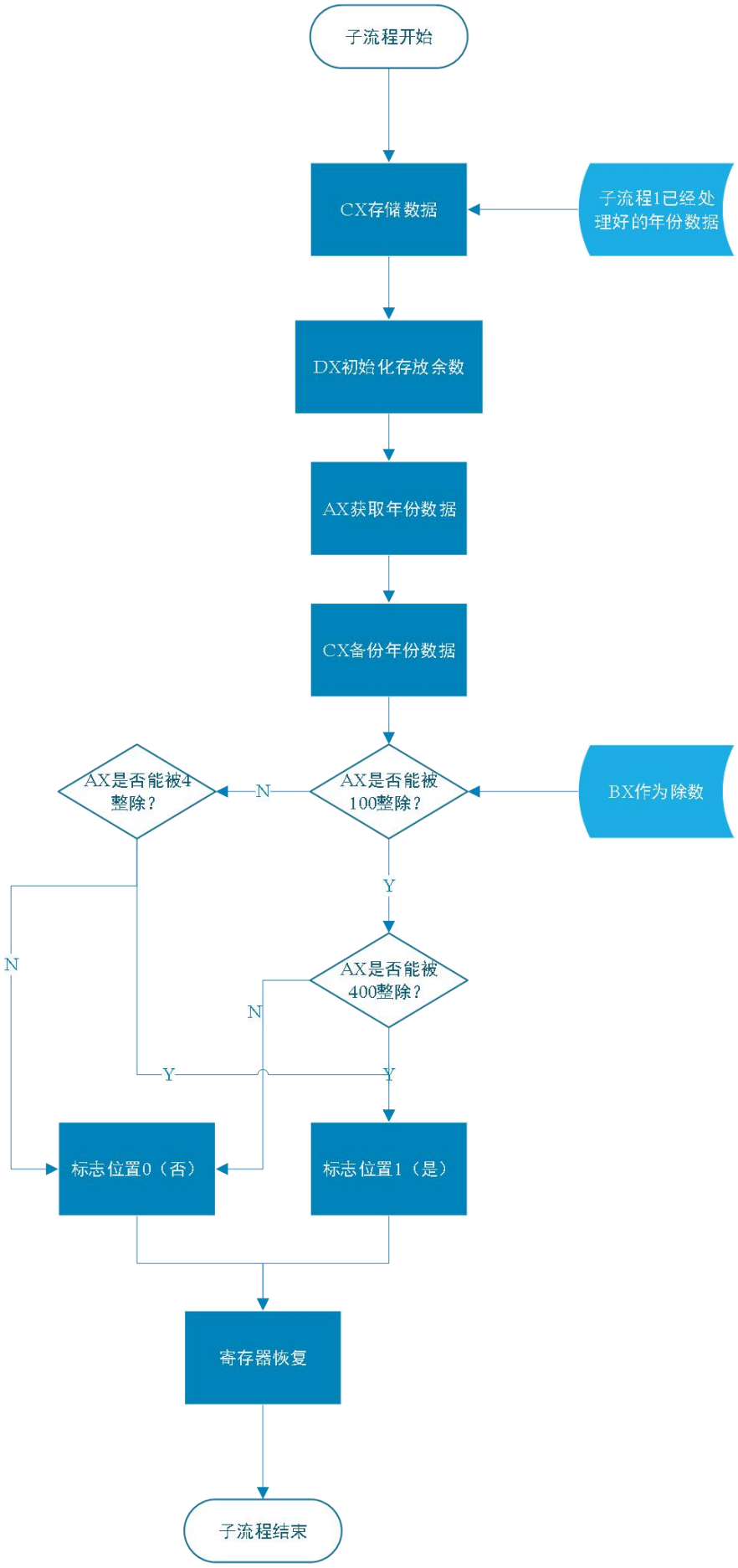
主流程图



子流程 1：数据处理



子流程 2：闰年判断


子流程开始
CX存储数据
子流程1已经处理好的年份数据
DX初始化存放余数
AX获取年份数据
CX备份年份数据
AX是否能被100整除？
BX作为除数
N
AX是否能被4整除？
Y
N
AX是否能被400整除？
N
Y
Y
标志位置0（否）
标志位置1（是）
寄存器恢复
子流程结束

附：闰年程序的改进

```
 修改部分：
; datacate 字符转数字的代码流程，仅用循环进行替代，简化了程序的思路
; 只需要遍历所有的内存单元，依次减去 30H 即可
data segment
      infon db 0dh,0ah,'please input a year: $'
      Y db 0dh,0ah,'This is a leap year! $'
      N db 0dh,0ah,'This is not a leap year! $'
      w dw 0
      buf db 8
          db ?
          db 8 dup(?)
data ends
stack segment stack
     db 200 dup(0)
stack ends
code segment
    assume ds:data,ss:stack,cs:code
start:mov ax,data
      mov ds,ax
      lea dx,infon
      mov ah,9
      int 21h
      lea dx,buf
      mov ah,10
      int 21h
      mov cl,[buf+1]
      mov ch,0
      lea di,buf+2
      call datacate
      call ifyears
      jc a1
      lea dx,n
      mov ah,9
      int 21h
      jmp exit
a1:   lea dx,y
      mov ah,9
```

```
          int 21h
          jmp exit
exit:     mov ah,4ch
          int 21h
datacate proc near
    NUMBER_CONVERT:
       SUB BYTE PTR [DI], 30H
       MOV BL, [DI]
       INC DI
       MOV BYTE PTR [SI], BL
       INC SI
    LOOP NUMBER_CONVERT
       RET
datacate endp
ifyears proc near
    push bx
    push cx
    push dx
    mov ax,[w]
    mov cx,ax
    mov dx,0
    mov bx,100
    div bx
    cmp dx,0
    jnz lab1
    mov ax,cx
    mov bx,400
    div bx
    cmp dx,0
    jz lab2
    clc
    jmp lab3
    lab1:mov ax,cx
         mov dx,0
         mov bx,4
         div bx
         cmp dx,0
         jz lab2
```

```
          clc
          jmp lab3
     lab2:stc
     lab3:pop dx
          pop cx
          pop bx
          ret
ifyears endp
code ends
   end start
```

【第三个实验：汇编实例学习与改进：两位数加法】

【编程逻辑】

1. 建立缓冲区，将输入的数字存入缓冲区内部，随后再存入预设好的数据段内
2. 利用内存单元遍历操作将其倒序存放，最后一个内存单元放最低位，前面的首零内存单元留空，作为预进位单元
3. 二次存入数据后，两个数据相加，覆盖 ADD2
4. 进行遍历做十进制基础下的位数修正
5. 遍历处理成 ASCII 可输出形式
6. 遍历进行输出

这样就可以实现小于 15 位的任意数加法运算

【程序运行示例】

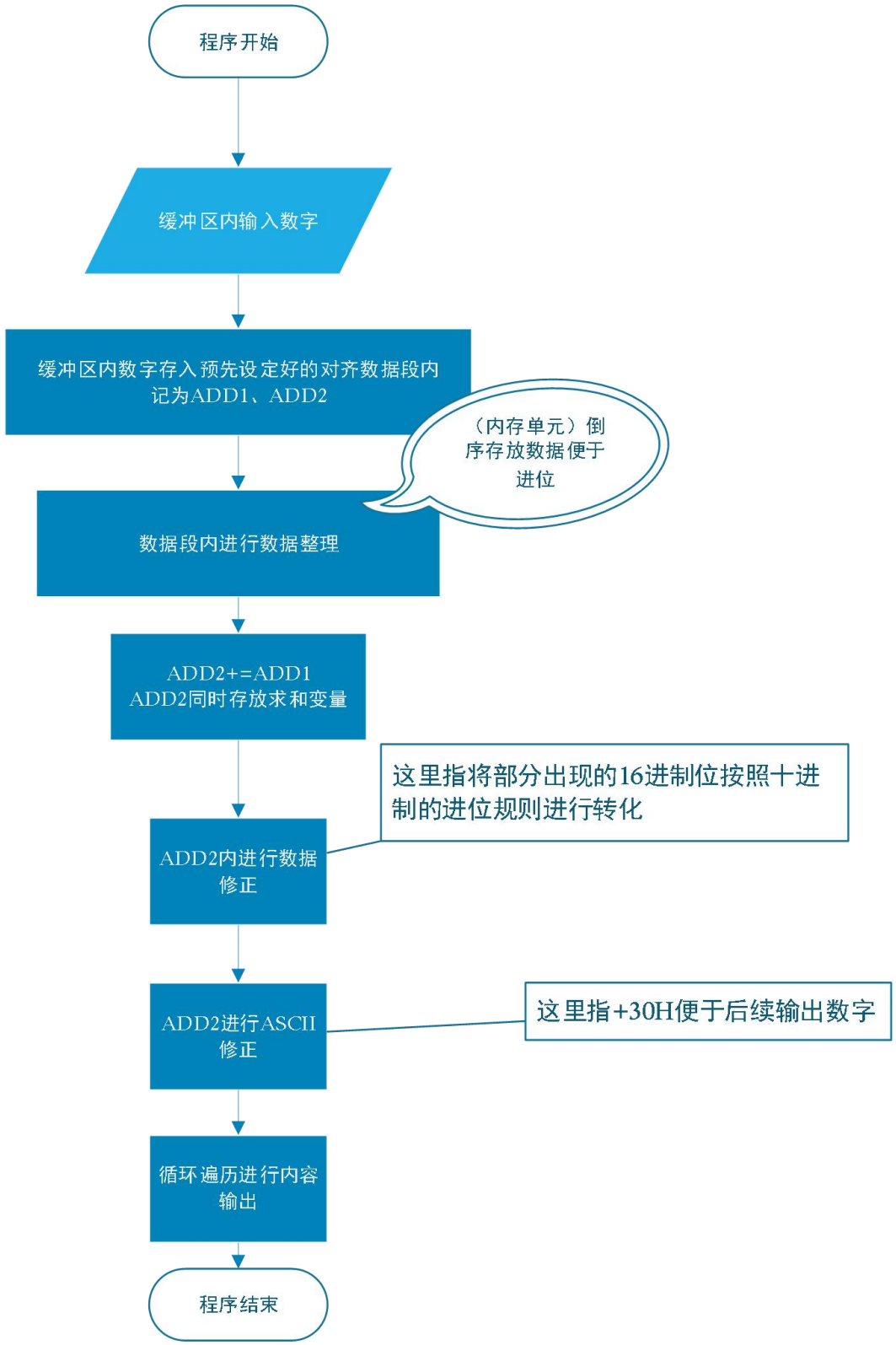
15 位+15 位

```
Please input an integer number: 1233659878963

Please input an integer number: 5869896633259 87
The result is:7103556512229 50
```

小于 15 位但高位有进位

```
Please input an integer number: 45698

Please input an integer number: 99686
The result is:145384
```

【程序流程图】



【实验代码】

```
; 程序版本：3.0
; 编程逻辑：缓冲区、内存
; 实现的功能：n 位+m 位相同位数的 10 进制加法运算（n<=m 且 m,n<15）
```

```
; 修复了最高位进位的 bug
; 另一个 bug，n>m 时计算失效，没有建立好的输入异常机制（Exception）

DATA SEGMENT
    INFON DB 0AH, 0AH, 'Please input an integer number: $',0AH
          DB 12 DUP(?)
    ADD1 DB 16 DUP(?)
    ADD2 DB 16 DUP(?)
    ; 存放输入端十五位数字，同时 ADD2 存放后续求和的数字
    ADD_BUFFER DB 16
        DB ?
        DB 16 DUP(?)
    OUTPUTINFO DB 0AH,'The result is:$',0AH
    ; 缓冲区作为加数输入端
DATA ENDS

STACKS SEGMENT
    DB 200 DUP(0)
STACKS ENDS

CODES SEGMENT
    ASSUME CS:CODES, DS:DATA, SS:STACKS
START:
    MOV AX, DATA
    MOV DS, AX
    LEA SI, ADD1
    ADD SI, 15
    ; 指向最后一位，便于倒序处理
    ; 这里倒序处理的目的是将没有存入数据的内存单元放在前面
    ; 例如默认存入为 09 02 03 04 00 00 00 00，则倒序后
是 00 00 00 00 09 02 03 04，这样是为了处理高位进位时的情况
    CALL NUMBER_PROCESS
    LEA SI, ADD2
    ADD SI, 15
    CALL NUMBER_PROCESS
    CALL ADDING
    ; 进行数据处理，遍历每一位，对大于十的进行进位
    CALL BIT_PROCESS
```

```
    CALL NUM_DISP
    MOV AH, 4CH
    INT 21H
NUMBER_PROCESS PROC NEAR
        LEA DX, INFON
        MOV AH, 9H
        INT 21H
        LEA DX, ADD_BUFFER
        MOV AH, 0AH
        INT 21H
        MOV CL, [ADD_BUFFER + 1]
        MOV CH, 0
        ; CX 记录缓冲区实际数字的数量，为后续循环做准备
        LEA DI, ADD_BUFFER + 2
        ADD DI, CX
        DEC DI
    NUMBER_CONVERT:
        SUB BYTE PTR [DI], 30H
        MOV BL, [DI]
        ; BL 寄存器起到中间变量的作用
        DEC DI
        ; DI 自减 1，整个循环过程类似于 for 循环（i--）
        MOV BYTE PTR [SI], BL
        DEC SI
    LOOP NUMBER_CONVERT
        RET
NUMBER_PROCESS ENDP
ADDING PROC NEAR
    LEA SI, ADD1
    ADD SI, 15
    LEA DI, ADD2
    ADD DI, 15
     ; 指向最后一位，便于倒序处理
    MOV CL, [ADD_BUFFER+1]
    MOV CH, 0
    MOV BX, 0
    ADDING_LOOP:
        MOV BYTE PTR BL,[SI]
```

```
        DEC SI
        ADD [DI], BL
        DEC DI
    LOOP ADDING_LOOP
    RET
ADDING ENDP
BIT_PROCESS PROC NEAR
    LEA DI, ADD2
    MOV CL, [ADD_BUFFER+1]
    MOV CH, 0
    ADD DI, 15
    BIT_LOOP:
        MOV BYTE PTR AL, [DI]
        CMP AL, 0AH
        JB LOOP_ENDING
        ;大于十的情况
        SUB AL, 0AH
        MOV BYTE PTR [DI], AL;位调整
        DEC DI
        ADD [DI],1;前一位加一
        JMP FORCE_END;因为上边已经减了一次做进位处理，这里是为了防止 DI 被
多减 1 次
    LOOP_ENDING:
        DEC DI
    FORCE_END:
    LOOP BIT_LOOP
    RET
BIT_PROCESS ENDP
NUM_DISP PROC NEAR
    LEA DX, OUTPUTINFO
    MOV AH, 9H
    INT 21H
    LEA DI, ADD2
    MOV SI, 0;作为一个“标志性”寄存器存在，当碰到第一个非零数时更改状态
    MOV CX, 16
    NUM_DISP_LOOP:
        MOV BYTE PTR DL, [DI]
        CMP SI, 1
```

```
        JE GENERAL_PROCESS
        CMP DL, 0
        JE  LOOP_END
        MOV SI, 1
    GENERAL_PROCESS:
        ADD DL, 30H
        MOV AH, 2H
        INT 21H
    LOOP_END:
        INC DI
    LOOP NUM_DISP_LOOP
    RET
NUM_DISP ENDP
CODES ENDS
END START
```

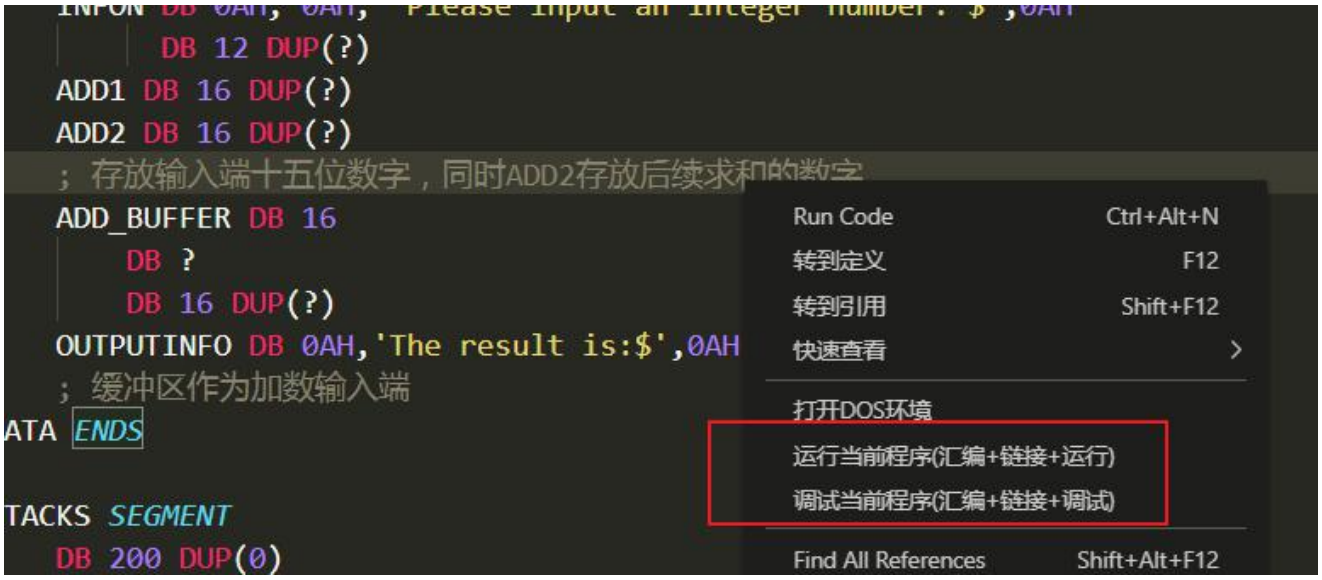
【实验过程分析】

要完成多位可输入型的十进制加法计算，首先要明确无论是缓冲区内输入后存储还是寄存器内存储，都是以 16 位 ASCII 的形式完成的。

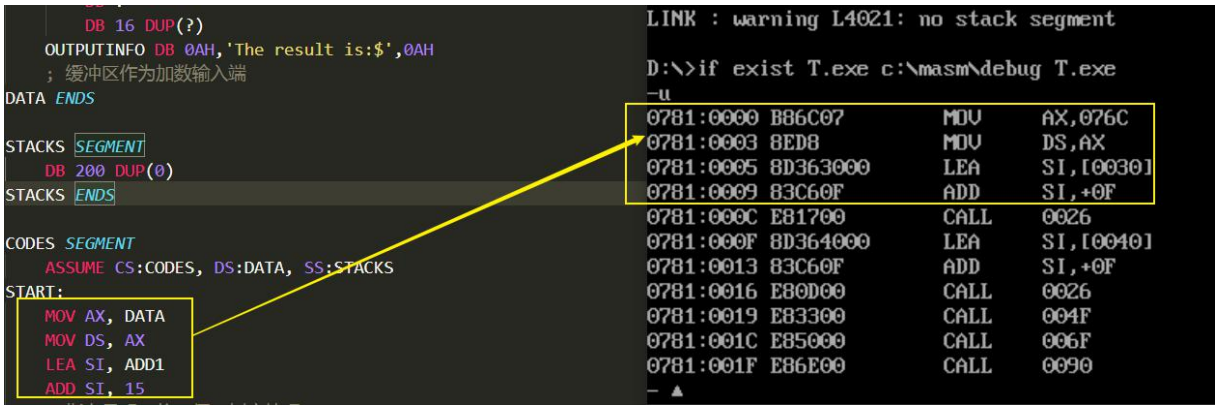
在本次实验中，由于涉及的量都是数字 0-9，没有涉及字母，所以不需要进行 3.1 中 16 进制转 10 进制的判断，这里只需要统一处理数字，也就是遍历-30H 变成十进制数意义上的 0-9 存储在预先定义好的数据段就可以了。

根据这个思路，先分析数据存储和相加的部分

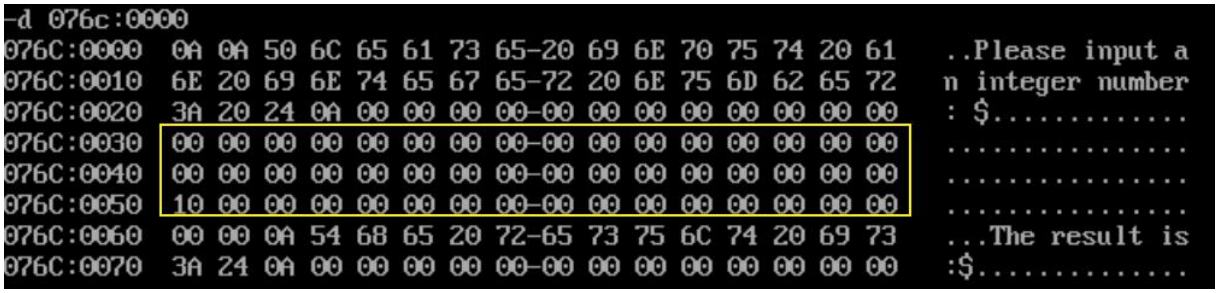
在文本编辑器 VScode 中完成程序编辑，并利用插件功能进行调试



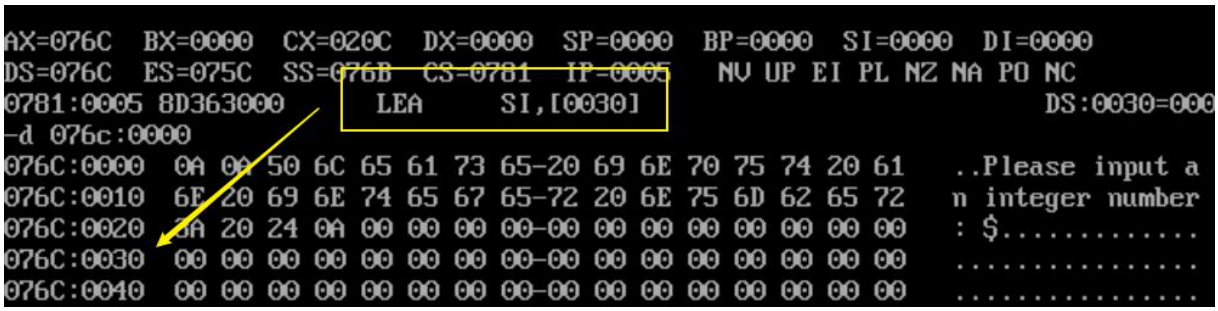
导入结果如下所示，利用-u 命令进行查看，可知导入成功



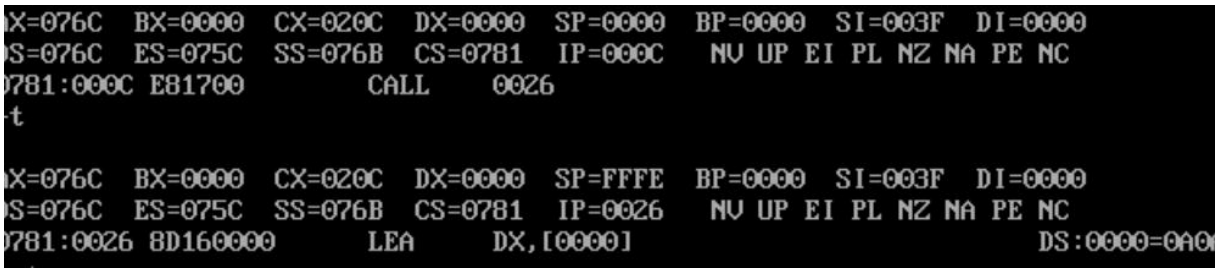
首先查看 DATA 的内存布局，如下图所示，这样是为了将后续要处理的数字进行对齐



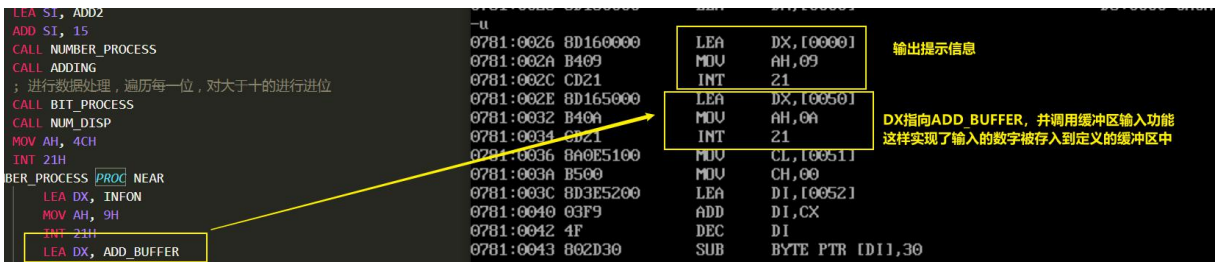
预留的存储单元区域分别为 ADD1、ADD2、ADD_BUFFER



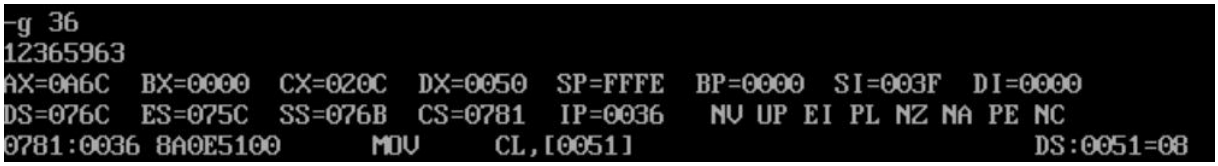
程序首先指向 ADD1，并+15，表示指向最末端，便于后续倒序处理



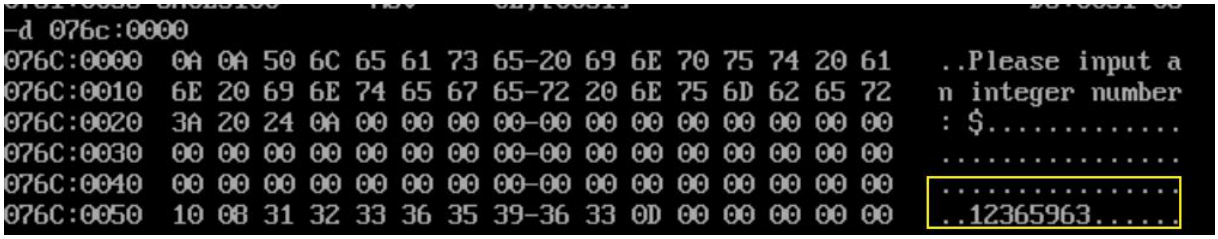
随后程序调用第一个子程序，数字的输入



程序开头的代码段注释如上图所示，利用-g 36 将其执行完毕，如下图所示



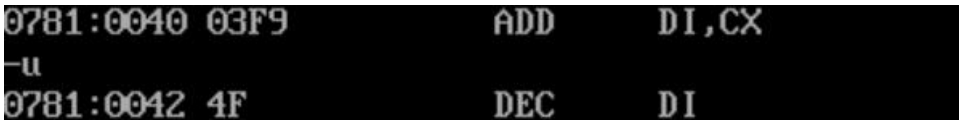
利用-d 命令查看内存



可见输入的数字以 ASCII 的形式被存入到了预先设定好的内存区域中，前面的 **08 表示数字个数，后面会用到**

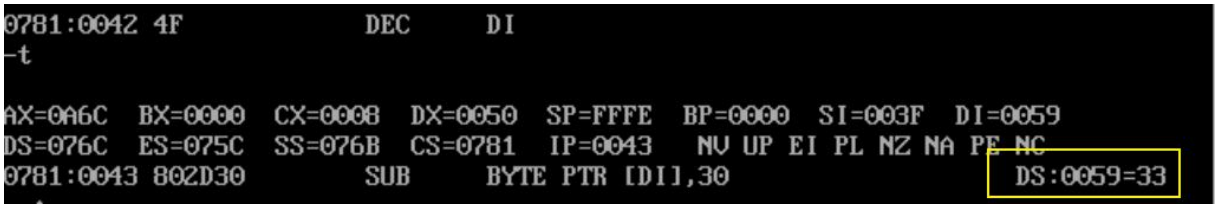
下面进行数字的存储部分，分析如下

该段程序实现了对输入数字个数的相关记录，CX 作为 Loop 过程中的计数器，同时使得 DI 寄存器指向了缓冲区的数据段，为后续各位的数字变换做准备。



上图程序段的作用表示将 DI 指向缓冲区内存的末端

执行完毕后如下所示

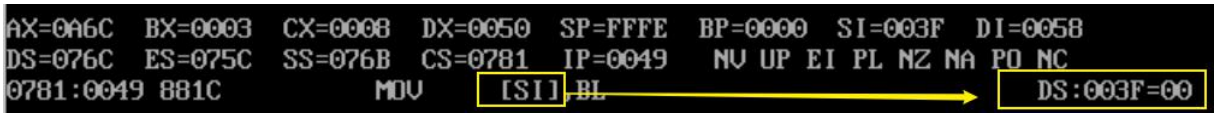


可见[DI]指向了最后一个数字 ASCII 量’33’

下面进行循环的数字处理部分，前面已经将 SI 指向了内存空间的最后一个单元程序段如下所示，仅以单次循环进行说明

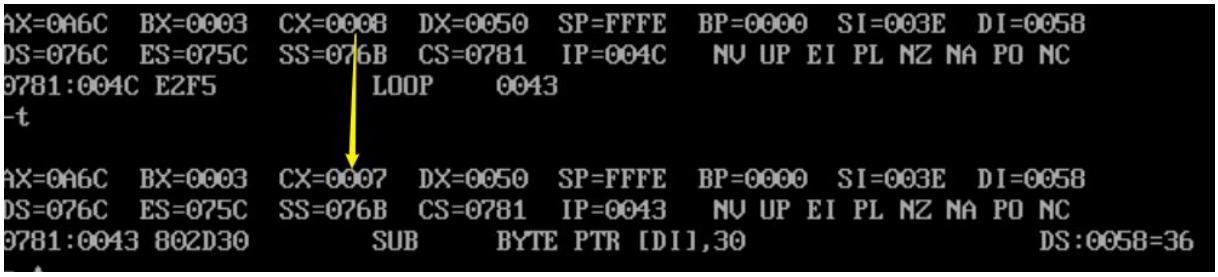
先执行末尾单元数字（最低位）数字转换，如下图所示，可见转换成功，转换为数字 03

再将 03 存入中间量寄存器 BL 中，并使 DI 自减 1，指向高位



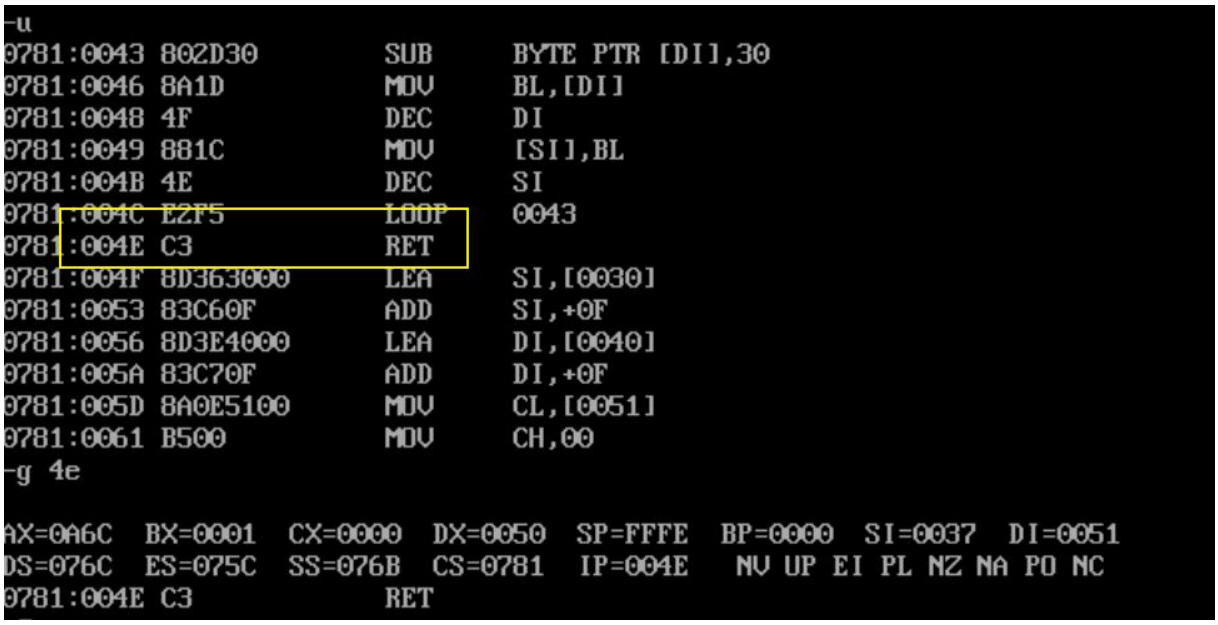
下面将 BL 的值存入[SI]中，第一次会存入到末尾内存单元中，如上图所示，再将 SI 自减一，指向高位，循环结束

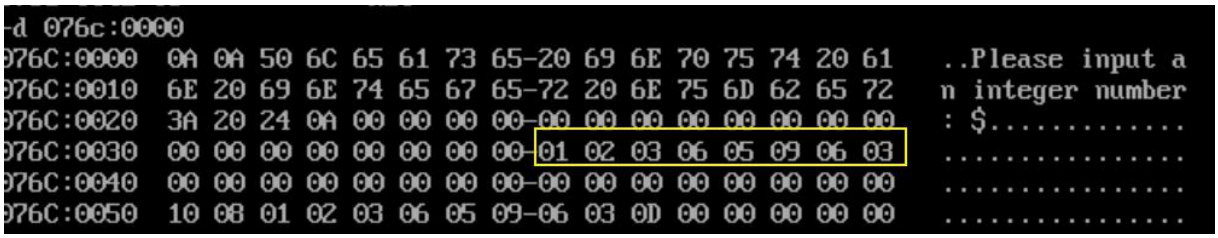
循环执行完毕后-d 如下图所示



CX 自减 1，CX 初始表示输入了 8 位数字，从而控制循环次数

-g 执行所有的循环，执行完毕后子程序结束，利用-d 查看如下图所示

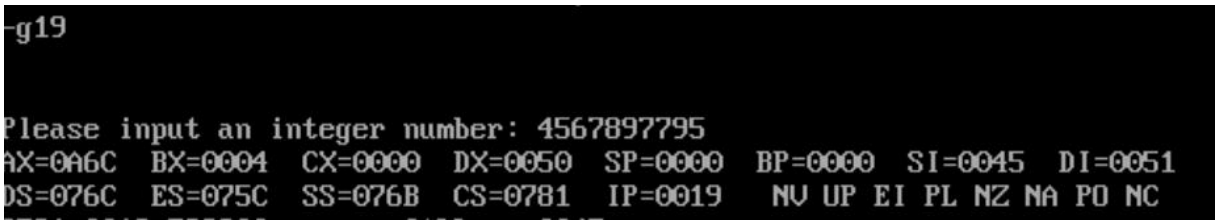


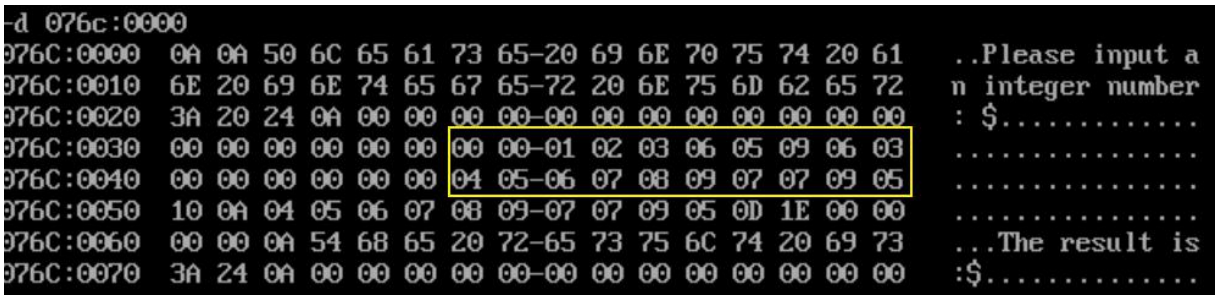


可见数字处理成功

同理下面进行第二个加数的输入，如下图所示

-g19 执行，-d 查看，如下所示





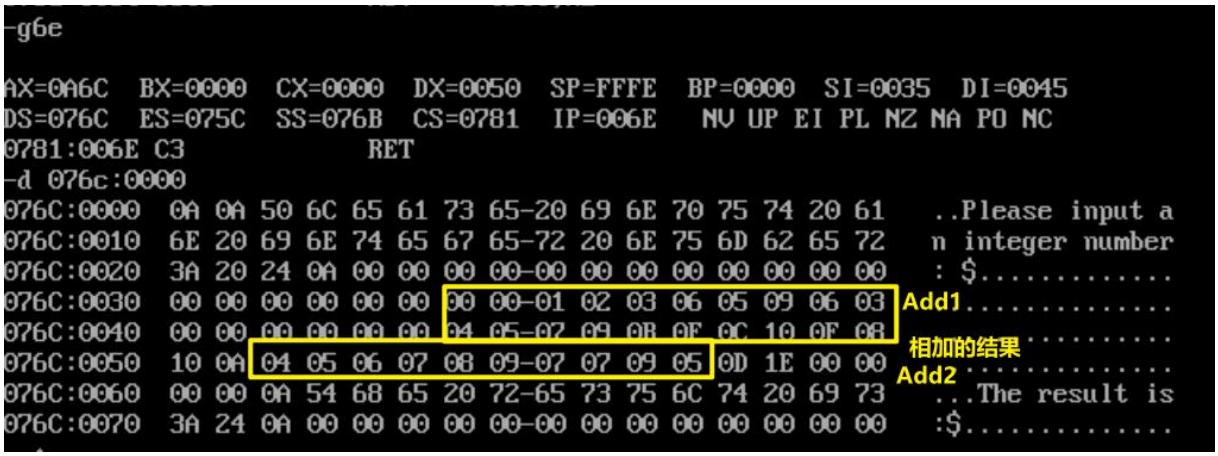
数字同样被成功存入

下面进行各位的数字相加过程

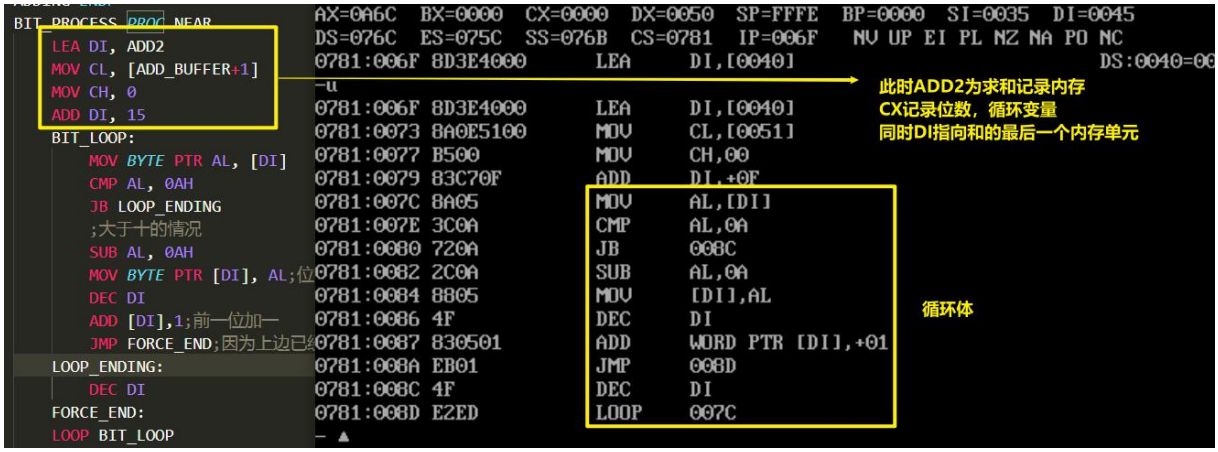
Bug 说明：这里仅利用 ADD2 的位数记录作诸位相加过程的循环变量，忽略了 ADD1 的位数大于 ADD2 的情况，如果不进行比较，仅记录小数字的位数的进行循环，则很明显会出现加法错误。

下面进行循环体的相关分析，-g 进入循环体，这里 CX=A 表示数字位数为 10

循环体的思路仍为倒序遍历相加，过程与数字输入的实现逻辑类似，这里不再赘述，利用-g 完成循环体，查看结果如下所示

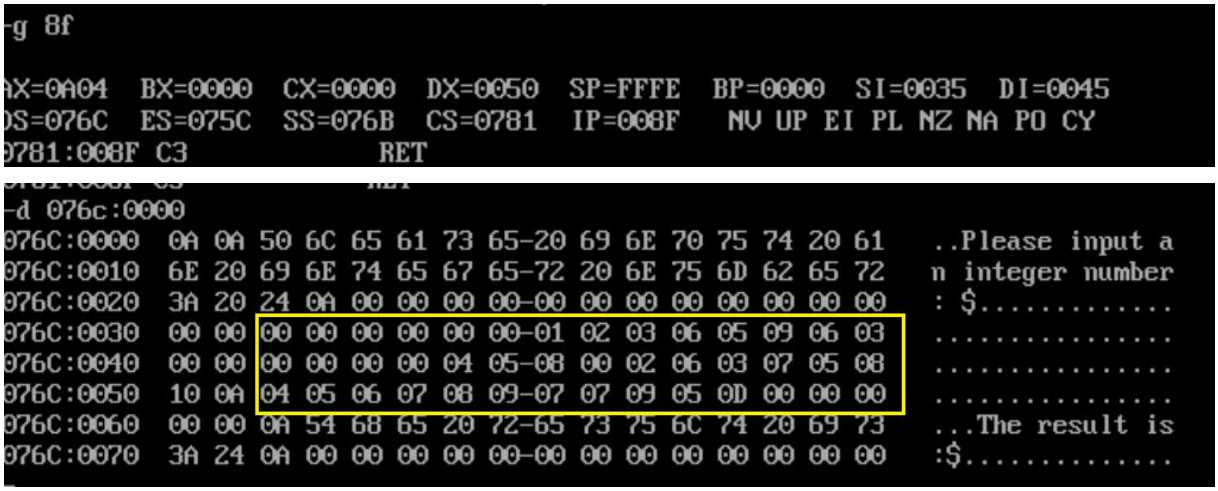


可见相加的结果出现了十六进制的形式，也就是部分不进位的问题，考虑到每一位最大进位数不可能大于 1，因而只需要考虑进一位的过程，下面对 BIT_PROCESS 的循环体进行分析，如下图所示



首先 DI 指向最后一个，AL 暂存数据，并与 0AH 比较，如果大于 0AH（也就是大于 10）则使得前一位加一，然后终止循环，否则不做处理，循环结束后 DI 都要向前移动一位。

循环体执行完毕后，-d 查看内存如下图所示



可见已经完成了对应的进位过程，符合十进制的运算和进位规则

随后要进行每一位的 ASCII 匹配过程，从尾部遍历求和变量的内存区域，每一位+30H，直到遍历到零元素结束循环过程

```
-u
0781:0090 8D166200      LEA     DX,[0062]
0781:0094 B409          MOV     AH,09
0781:0096 CD21          INT     21
0781:0098 8D3E4000      LEA     DI,[0040]
0781:009C BE0000        MOV     SI,0000
0781:009F B91000        MOV     CX,0010
0781:00A2 8A15          MOV     DL,[DI]
0781:00A4 83FE01        CMP     SI,+01
0781:00A7 7408          JZ      00B1
0781:00A9 80FA00        CMP     DL,00
0781:00AC 740A          JZ      00B8
0781:00AE BE0100        MOV     SI,0001
```

输出提示信息

其中 DI 指向存放和的内存区域，SI 作为一个“标志性”寄存器存在，当碰到第一个非零数时更改状态 0-1，这是为了实现程序的位处理，避免输出多余的 0。循环体在判断 SI 基础上实现了每一位都+30H，并把结果存放到 DL 中，随后利用中断调用，立即输出对应的数字字符，这样循环下去，拼接式地完成结果的输出。该段结束后如下图所示

```
-gbb

The result is:4580263758
AX=0238  BX=0000  CX=0000  DX=0038  SP=FFFE  BP=0000  SI=0001  DI=0050
DS=076C  ES=075C  SS=076B  CS=0781  IP=00BB   NV UP EI PL NZ AC PE NC
0781:00BB C3            RET
```

至此程序分析结束，可见求和输出是正确的

【实验心得】

本次实验相对前两次更具有综合性，对汇编程序的编写能力和对计算机程序底层运行逻辑又是一定程度的提升。

首先是 16 进制转 10 进制的处理流程，接续了实验 2.2 的循环输入判断思路，比如复用了 2.2.1&2.2.2 的写法进行字母判断，随后进行除法的运算。实验过程中由于对初始化的重要性与除法的机制了解不足，DX 没有进行初始化就进行了 16 位除法，致使得不到正确的结果，也浪费了很多的时间。最后问题终归得到了解决，程序经过 4 次优化后逻辑也更为清晰，虽然耗费了较多的时间，但是这块实验也让我对汇编语言编程了解进一步加深，同时也初步建立起了模块化编程的思想。

其次是闰年程序的分析，程序的改进是在第三个实验完成后完成的。闰年判断的思路较为直接，主要把握好数字转化和逻辑判断就行了，逻辑判断在 C/C++中已经学过，算法实现起来也不算太陌生。该程序也为第三个程序提供了思路，借用闰年缓冲区和内存的思路可以大大简化 3.3 的分析过程。

最后是多位的十进制加法计算器编写，起初我打算接续第一个程序的思路，利用寄

存器循环输入、压栈、判断，最后发现不仅会有压栈的双字节（字）存储问题，进位溢出的问题也很难解决，于是借鉴闰年的缓冲区与数据存入内存的思路，很快改出了第一版能够正常运行结果的程序。

但是由于没有考虑到高位进位溢出的问题，在存储的时候没有“倒序”存储，致使首位始终没有留空，发生“溢出”错误（例如 55+55=(1)10 的计算错误）因此这里考虑了倒序存放数据，也就是内存单元组最后一个单元存放低位，依次向上存放高位，其余留空作为预置高位，这样就解决了数据溢出的问题。另外，由于涉及到很多内存单元的操作，第三次实验很多篇幅采用了寄存器寻址的方式，编程逻辑也与 C/C++中的数组相仿，这一点在编程时深有体会。个人认为这也可以算作高级语言和汇编语言的某种联系。

总之，这次实验耗费的时间和精力都要高于前两次实验，但获得的收获比前两次实验都要多，这与大量的 Debug 实践是分不开的。本次实验很多改进都是通过 Debug 实现的。程序开发过程很多都是调试（Debug）实现的，本次实验之后，我发现自己在高级语言的程序编写中利用调试的次数都没有汇编多，都是简单的运行与单点查错，很少通过调试进行整个程序结构的优化。这次实验给了我这次体验，使得我对后续高级编程语言的学习更具信心。