

# 嵌入式计算机系统与实验 | 作业 2

Shuiyuan@Noroshi

## 1 汇编代码试运行

我们尝试运行下段代码

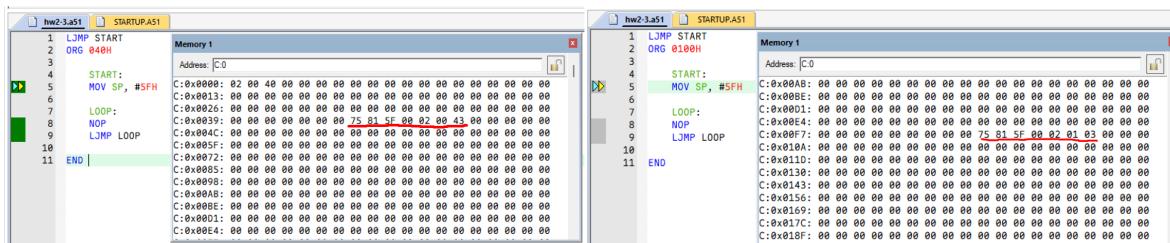
```
ORG 0000H
LJMP START
ORG 040H

START:
    MOV SP, #5FH ;设堆栈

LOOP:
    NOP
    LJMP LOOP ;循环

END ;结束
```

分别将第三行设为 ORG 040H 和 ORG 100H, 推测其会将程序分别存储在程序存储器的 0x0040 和 0x0100 处在编译后观察 C:0 的内存情况



**Figure 1:** 可以注意到操作码是完全一样的, 但位置发生了改变, 这与我们的预期是相同的

指令操作码分别为:

- **MOV SP, #5FH**, 分别为 75(MOV) 81(SP 所在地址) 5F(立即数), 位于 ROM 的 0x0040~0x0042 处(以上段代码内容为准)
- **NOP** 操作码为 00, 位于 0x0043
- **LJMP LOOP** 操作码为 02(LJMP) 00 43(LJMP 的目标地址), 位于 0x0044~0x0046

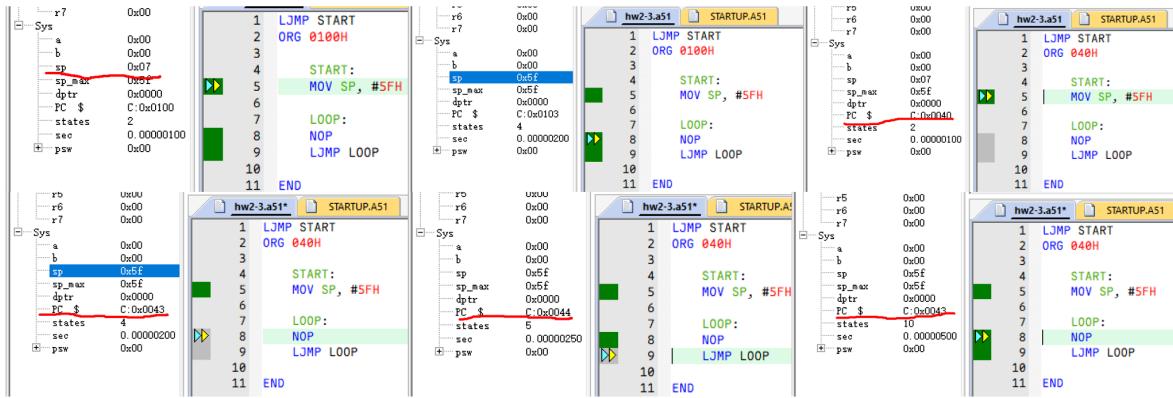


Figure 2: sp 变化与 PC 轨迹

如上图我们可见, 在执行了 `MOV` 指令后 `sp` 发生了变化; 而 `SP` 指针是执行一次增加一起, 其增加的距离就是指令的长度, 例如上面我们看见 `MOV` 指令的长度为 3, 所以在执行完之后就从 40 跳到了 43, 而 `NOP` 就仅+1, 而在 `LJMP` 之后程序指针又指回了标签开始处

`START` 和 `LOOP` 则为程序标签, 本身是什么名字都无所谓, 在这个程序里, `START` 是程序开始的标志, 和 c 语言里的 `main` 函数比较类似, 而 `LOOP` 则是一个死循环, 防止程序运行到不该运行的地方

## 2 四则运算对 PSW 相关比特的影响

### 2.1 error 65: access violation at : no 'execute/read' permission

在我想手写一段验证四则运算的代码的时候, 弹出了标题这样的报错信息

```
ORG 0000H
LJMP START
```

```
START:
    MOV A, #01H
    MOV B, #02H
    ADD A, B
END
```

我感到很疑惑, 因为我认为这段代码是没有什么任何问题的. `END` 也是正常写了的. 但查询资料后发现, 这里的 `END` 是一个伪指令, 其并不会被编译, 当然也不会实际上对代码执行任何影响. 代码会一直按顺序执行, 当执行到未写入程序的部分时就会报错, 从某种意义上除非在程序中设定死循环不然最终都会以无权限结束? 不过 `END` 会对编译的内容有影响, 在 `END` 之后的代码都不会被编译, 如下片代码与上的编译后的内容是完全一样的.

```
ORG 0000H
LJMP START
```

```
START:
    MOV A, #01H
    MOV B, #02H
    ADD A, B
END

FOO:
    WHAT DOES THIS LINE MEAN?
    NO ONE KNOWS AND NO ONE CARES!
```

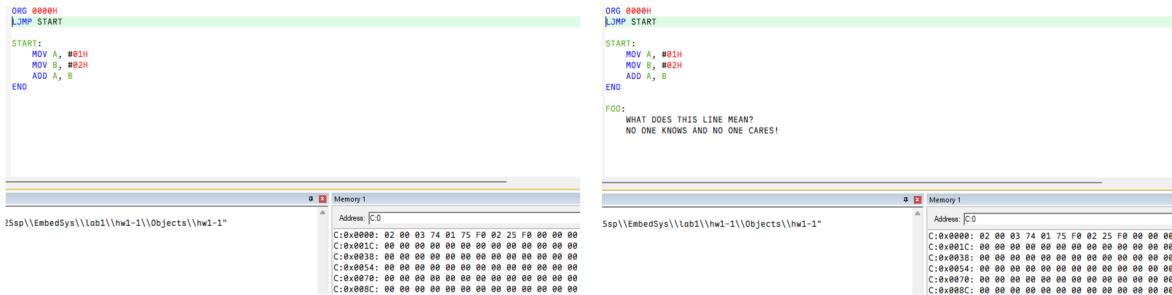


Figure 3: 如图可见, 两段代码在编译后存入内存的机器码是完全一样的

回归正题, 我们在这个实验中重点关注的是在四则运算中 PSW 寄存器特别是其中的 P 位, AC 位, CY 位, OP 位的行为

## 2.2 加法运算

### 2.2.1 CY 位与 OP 位

在.a51 汇编程序中, 似乎并没有像 C 语言中的 `unsigned int` 与 `int` 这种区别, 例如 `11111111B` 可以表示-1(有符号, 补码形式), 也可以表示 255(无符号, 正常二进制表示). 具体是那个似乎只能程序员自己清楚? 二者在实际的寄存器表示中没有区别. 而 OV 主要是为了有符号数的加减而设计的, CY 主要是为了无符号数的加减而设计的, 比如下面两个例子:

```

ADD_1_1:
MOV A, #01000000B ; 有符号数+64/+127, 无符号数64/255
MOV B, #01000000B ; 有符号数+64/+127, 无符号数64/255
ADD A, B           ; 有符号则为+128/+127溢出, 无符号为128/255正常

ADD_1_2:
MOV A, #01000000B ; 有符号数+64/+127, 无符号数64/255
MOV B, #11000000B ; 有符号数-64/-128, 无符号数192/255
ADD A, B           ; 有符号为0/+127正常, 无符号为256/255溢出

```

那么我们可以看出, 在程序段中, 如果把 A, B 视为有符号数则会发生溢出, 而视为无符号数则可以正常完成加法; 而在程序段中, 如果把 A, B 视为有符号数则可以正常完成加法; 而视为无符号数则会发生溢出, 那么观察实际调试时的寄存器结果, 也确实如此.

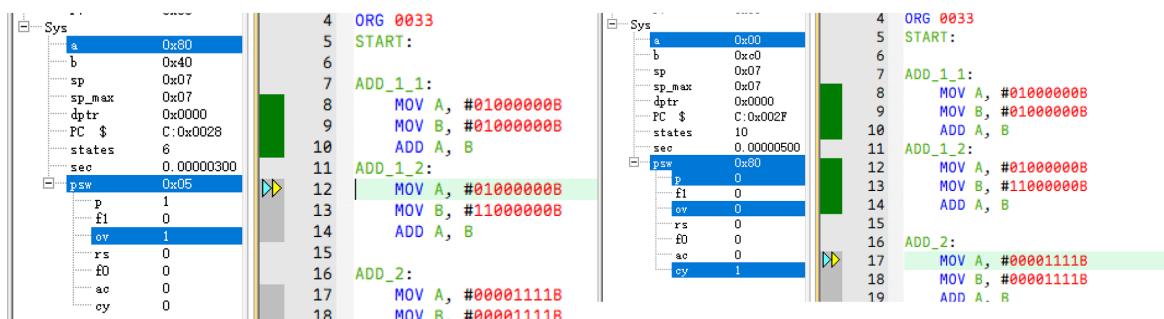


Figure 4: 如图可见, 左边为第一次加法运算后, 可以注意到 OV 变为 1, 而 CY 置零, 说明在有符号情况下出现了溢出而无符号正常.

同理右边为完成第二次加法运算后, 可以注意到 OV 置零, 而 CY 变为 1, 即说明在有符号情况下正常, 而无符号情况下发生了溢出.

### 2.2.2 P 位与 AC 位

P 位为校验位, 用于检测寄存器 A 的数(二进制状态)下 1 的个数, 若为奇数则为 1, 偶数则为 0. 而 AC 位是半进位, 即第四位是否发生了进位, 若发生了则为 1 否则为 0. 为了验证其实际表现, 我们考虑以下例子

**ADD\_2:**

```
MOV A, #00001001B
MOV B, #00001101B
ADD A, B
```

从结果我们可以看出,本来为 0 的 P 位(A 原先有 2 个 1, 现在变为了 3)所以变为了 1, 而第四位发生了进位, AC=1

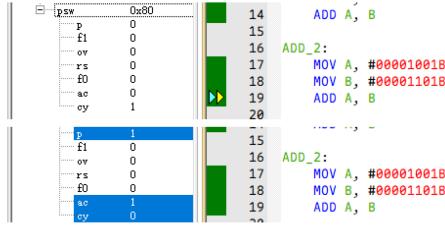


Figure 5: 如图, 执行后的 p, ac 均发生了变化

### 2.2.3 ADDC

ADDC 的作用即是考虑 CY 位的进位状况, 如下片代码

**ADD\_3:**

```
SETB CY
CLR OV
MOV A, #01111110B
MOV B, #00000001B
ADDC A, B
```

如果只考虑 A, B 的话, 那么既不会触发 OV 也不会触发 CY, 但在 CY 为 1 时使用 ADDC 指令, 我们发现 CY 的进位被加在了 A, B 之上, 在这片代码里, 也就是触发了 OV=1.

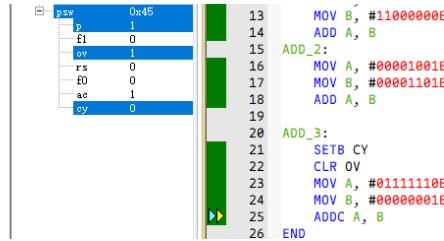


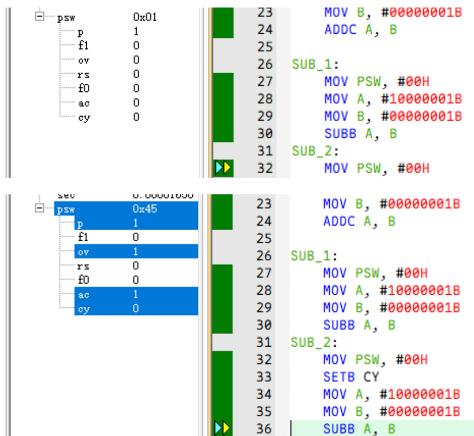
Figure 6: 如图, 发现执行指令之后 P 位为 1, 说明了 CY 的 1 位也计入了运算

### 2.3 减法运算

SUBB, .a51 中的减法同样带有借位功能, 具体来说, 其将减去 CY 位的值, 其他位的行为与加法相似, 考虑如下代码片段:

```
SUBB_1:
MOV PSW, #00H
MOV A, #10000001B
MOV B, #00000001B
SUBB A, B

SUBB_2:
MOV PSW, #00H
SETB CY
MOV A, #10000001B
MOV B, #00000001B
SUBB A, B
```



**Figure 7:** 如图我们可以发现, 在第一段代码并没有中 CY 位为 0, 所以实际减法为-127-1=-128, 可以正常执行不触发一处操作, 而第二段代码由于 CY 位被设为了 1, 实际减法位-127-1-1=-129 < -128, 超过了 8 位补码的表示范围, 因此触发了 OV 溢出提示

## 2.4 乘法运算

乘法的行为与加减稍有不同, 首先是从指令方面 **MUL AB**, AB 是连一起的没有分开, 且乘法默认是无符号数相乘, 其次在结果方面, 由于乘法运算大概率大概率结果>255, 所以此处会把 A, B 都作为储存结果的寄存器, 其中 A 储存低八位, B 储存高八位, 若结果>255 则会将 OV 设为 1, CY 在乘法运算中不会发生变化, 我们以以下代码为例:

```

MUL_1:
  MOV PSW, #00H
  MOV A, #00000001B
  MOV B, #00000001B
  MUL AB

MUL_2:
  MOV PSW, #00H
  MOV A, #10000000B
  MOV B, #00000010B
  MUL AB
      
```

第一个为 1\*1, 即最后得到 1, 未溢出到高八位; 第二个为 128\*2, 发生了溢出, 同时也可以观察到高八位存入了数值

```

    Sys
      a 0x01
      b 0x00
      sp 0x07
      sp_max 0x07
      dptr 0x0000
      PC $ C:0x0060
      states 42
      sec 0.00002100
      psw 0x01
      p 1
      f1 0
      ov 0
      rs 0
      f0 0
      ac 0
      cy 0
  27 MOV PSW, #00H
  28 MOV A, #10000001B
  29 MOV B, #00000001B
  30 SUBB A, B
  31 SUB_2:
  32 MOV PSW, #00H
  33 SETB CY
  34 MOV A, #10000001B
  35 MOV B, #00000001B
  36 SUBB A, B
  37 MUL_1:
  38 MOV PSW, #00H
  39 MOV A, #00000001B
  40 MOV B, #00000001B
  41 MUL AB
  42 MUL_2:
  43 MOV PSW, #00H
  44 MOV A, #10000000B
  45 MOV B, #00000001B
  46 MUL AB
  47 MUL AB
  48 END

```

```

    Sys
      a 0x00
      b 0x01
      sp 0x07
      sp_max 0x07
      dptr 0x0000
      PC $ C:0x0069
      states 51
      sec 0.00002550
      psw 0x04
      p 0
      f1 0
      ov 1
      rs 0
      f0 0
      ac 0
      cy 0
  27 MOV PSW, #00H
  28 MOV A, #10000001B
  29 MOV B, #00000001B
  30 SUBB A, B
  31 SUB_2:
  32 MOV PSW, #00H
  33 SETB CY
  34 MOV A, #10000001B
  35 MOV B, #00000001B
  36 SUBB A, B
  37 MUL_1:
  38 MOV PSW, #00H
  39 MOV A, #00000001B
  40 MOV B, #00000001B
  41 MUL AB
  42 MUL_2:
  43 MOV PSW, #00H
  44 MOV A, #10000000B
  45 MOV B, #00000001B
  46 MUL AB
  47 MUL AB
  48 END
  49

```

**Figure 8:** 第一个乘法运算后结果为 0x0001, 亦没有发生 OV 溢出的情况;  
而第二次结果为 0x0100, 可见高八位(B寄存器)的数值发生了改变, 同时 OV 也变  
为了 1

## 2.5 除法运算

除法的操作逻辑与乘法类似, 不过其 A 寄存器存商, B 寄存器存余数, 若除数为 0 则触发 OV, 例如以下代码

```

DIV_1:
  MOV PSW, #00H
  MOV A, #10000001B
  MOV B, #00000010B
  DIV AB
DIV_2:
  MOV PSW, #00H
  MOV A, #11111111B
  MOV B, #00000000B
  DIV AB

```

第一个应该是  $129/2=64$  余 1, 第二个则会触发 OV

```

    Sys
      a 0x40
      b 0x01
      sp 0x07
      sp_max 0x07
      dptr 0x0000
      PC $ C:0x0072
      states 60
      sec 0.00003000
      psw 0x01
      p 1
      f1 0
      ov 0
      rs 0
      f0 0
      ac 0
      cy 0
  38 MUL_1:
  39 MOV PSW, #00H
  40 MOV A, #00000001B
  41 MOV B, #00000001B
  42 MUL AB
  43 MUL_2:
  44 MOV PSW, #00H
  45 MOV A, #10000000B
  46 MOV B, #00000001B
  47 MUL AB
  48 DIV_1:
  49 MOV PSW, #00H
  50 MOV A, #10000001B
  51 MOV B, #00000010B
  52 DIV AB
  53 DIV_2:
  54 MOV PSW, #00H
  55 MOV A, #11111111B
  56 MOV B, #00000000B
  57 DIV AB
  58 DIV AB
  59 END

```

```

    Sys
      a 0xff
      b 0x00
      sp 0x07
      sp_max 0x07
      dptr 0x0000
      PC $ C:0x007B
      states 69
      sec 0.00003450
      psw 0x04
      p 0
      f1 0
      ov 1
      rs 0
      f0 0
      ac 0
      cy 0
  38 MUL_1:
  39 MOV PSW, #00H
  40 MOV A, #00000001B
  41 MOV B, #00000001B
  42 MUL AB
  43 MUL_2:
  44 MOV PSW, #00H
  45 MOV A, #10000000B
  46 MOV B, #00000001B
  47 MUL AB
  48 DIV_1:
  49 MOV PSW, #00H
  50 MOV A, #10000001B
  51 MOV B, #00000010B
  52 DIV AB
  53 DIV_2:
  54 MOV PSW, #00H
  55 MOV A, #11111111B
  56 MOV B, #00000000B
  57 DIV AB
  58 DIV AB
  59 END
  60

```

**Figure 9:** 可以观察第一次运算后 A 寄存器为 0x40(十进制 64), B 寄存器位 0x01,  
分别为商和余数  
第二个除法运算由于除数为0, 所以触发了OV

### 3 指针与堆栈

#### 3.1 函数

堆栈最常用到的地方就是在函数调用的时候, 所以首先我们应当对.a51 中的函数有一个基本的概念, 当然这里的函数其实和标签基本是一个东西, 一个简单例子如下

```
ORG 0000H
LJMP MAIN

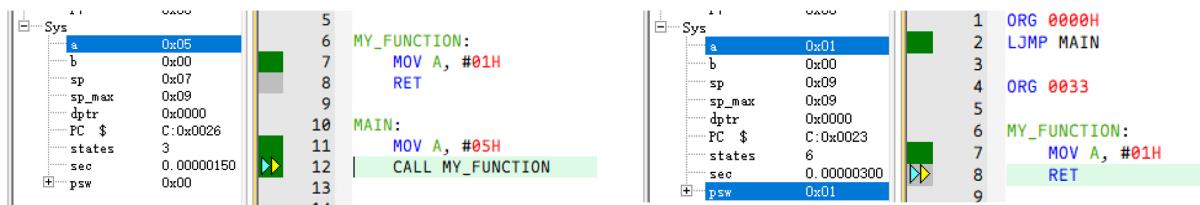
ORG 0033

MY_FUNCTION:
    MOV A, #01H
    RET

MAIN:
    MOV A, #05H
    CALL MY_FUNCTION
```

当执行到 `CALL MY_FUNCTION` 时, 其会跳转至 `MY_FUNCTION` 处, 然后执行对应语句, 直到这里都是与 `LJMP` 到标签是一样的, 不过如果有 `RET` 语句, 则其会直接返回到原 `CALL MY_FUNCTION` 的位置继续向后执行.

.a51 中的函数并没有像 c 语言等高级语言中复杂的传参机制与类型, 其就是简单粗暴地覆盖寄存器中的原数值, 因此堆栈等机制就能发挥上用场了.

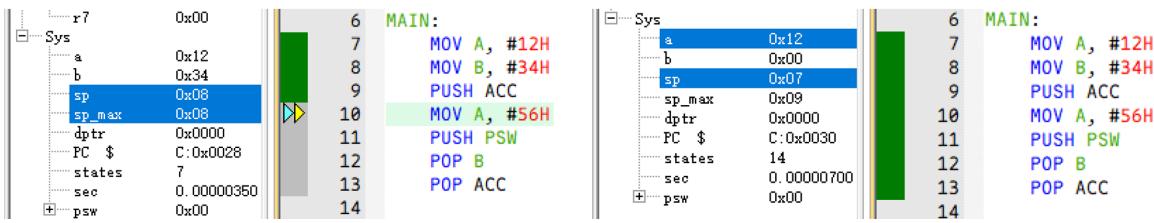


**Figure 10:** 如图, 本来在 MAIN 部分对 A 寄存器已有赋值, 而该赋值在调用 MY\_FUNCTION 的时候被直接覆盖了

#### 3.2 PUSH 与 POP

栈以及其通用的 push 和 pop 操作在数据结构课程中早已有了一些基本的了解, 在 a51 汇编语言中, PUSH/POP 的格式是 `PUSH/POP <ADDR>`, 需要强调的点就是这里的操作的是直接地址, 例如说, A 是累加器的缩写, 但是说 `PUSH A` 是不行的, 必须要用 A 的全称(ACC)或者其直接地址 0E0H 才行. 此外, 此处的寻址是数据存储器中的, 不是程序存储器中的. 另外 PUSH 和 POP 的对象不一定是相同的, 很可能(处于特别目的或者只是不小心), 你把本来是 ACC 的 PUSH 进去之后 POP 到 B 里去了. 下面这段代码就简单体现了 PUSH, POP 的特性

```
MOV A, #12H
MOV B, #34H
PUSH ACC
MOV A, #56H
PUSH PSW
POP B
POP ACC
```



**Figure 11:** 如图可以看见, 在第一次 PUSH 之后, sp 自增; 而在两次 POP 之后, sp 回到自己最初的位置, 而 b 被 PSW 覆盖(PUSH, POP 位置不同), 而 A 本来是 0x12, 后来被 MOV 覆盖为了 0x56H, 在 POP 之后又变回了 0

x12

我们考虑一个包含函数的情况:

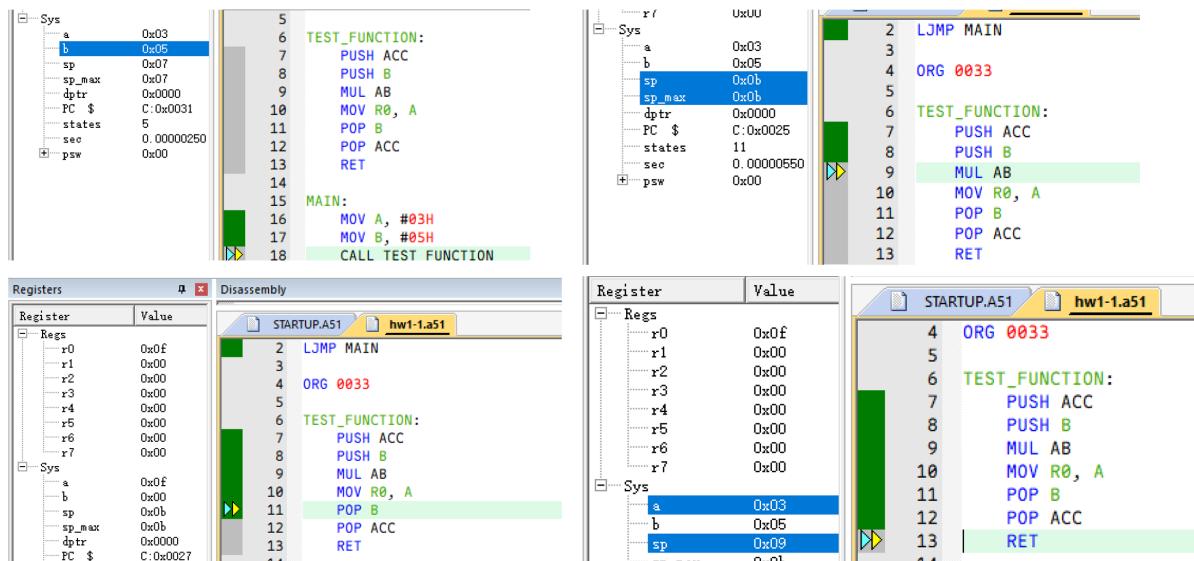
**TEST\_FUNCTION:**

```
PUSH ACC
PUSH B
MUL AB
MOV R0, A
POP B
POP ACC
RET
```

**MAIN:**

```
MOV A, #03H
MOV B, #05H
CALL TEST_FUNCTION
```

我们知道乘法运算中 A, B 都会用于存储结果, 而这段代码利用函数机制和堆栈让结果存在一般寄存器里, 保留了 A, B 中的原操作数



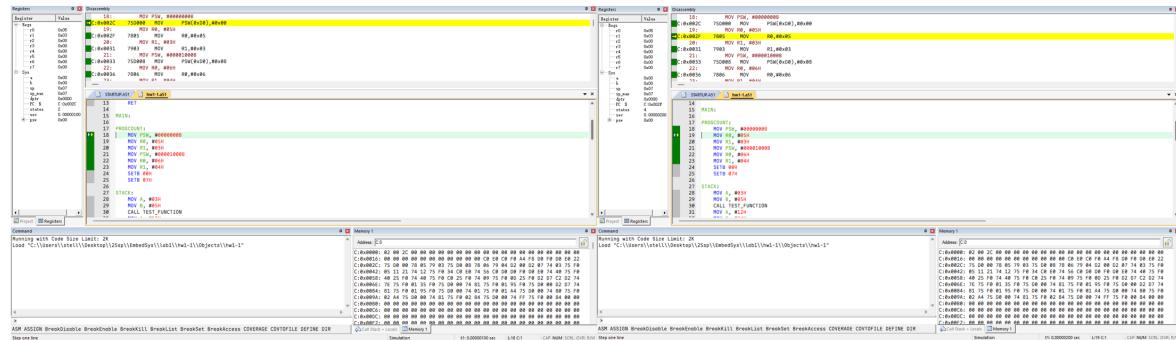
**Figure 12:** 从左上开始, 我们可以看见操作为  $3 * 5$ , 在函数体内, 我们先将 A, B 的数值存入栈中, 然后进行乘法(右上), 再把结果转移至 r0(左下), 然后再将保存在栈中的原操作数出栈(右下)

### 3.3 程序指针与存储器结构

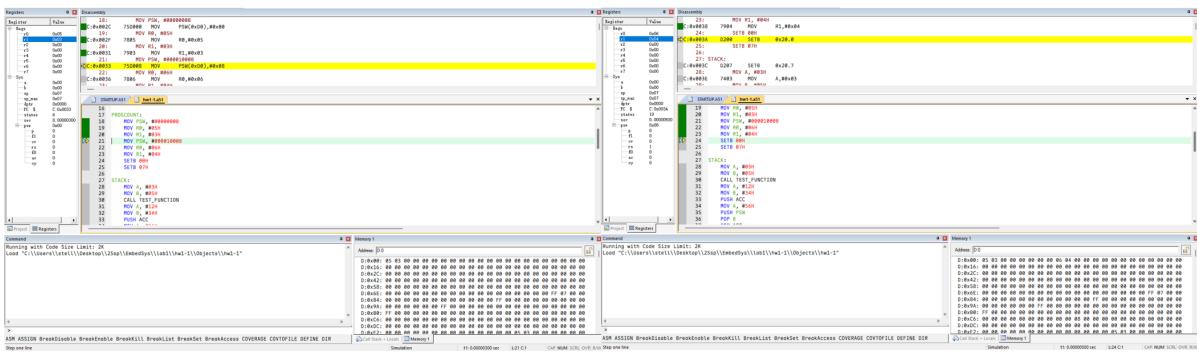
程序指针(PC)是单片机中比较特殊的一类寄存器, 其指向当前正在执行的命令对应的操作数的存储地址(程序存储器内), 且其一般不能被手动改动. 数据存储器中存储寄存器等存储部分的数值, 其中 00H~1FH 的共 32 个地址包含了四组每组 8 个寄存器, 具体使用那组由 PSW 寄存器中的第 4, 5 位决定, 之后 20H 到 2FH 位的 16 个地址为位地址区, 这里的每个地址可以被视作八个二进制地址单独操作(这部分地址实际上被写作 00H~7FH), 之后为通用 RAM 区和堆栈区. 考虑下段代码:

PROGCOUNT:

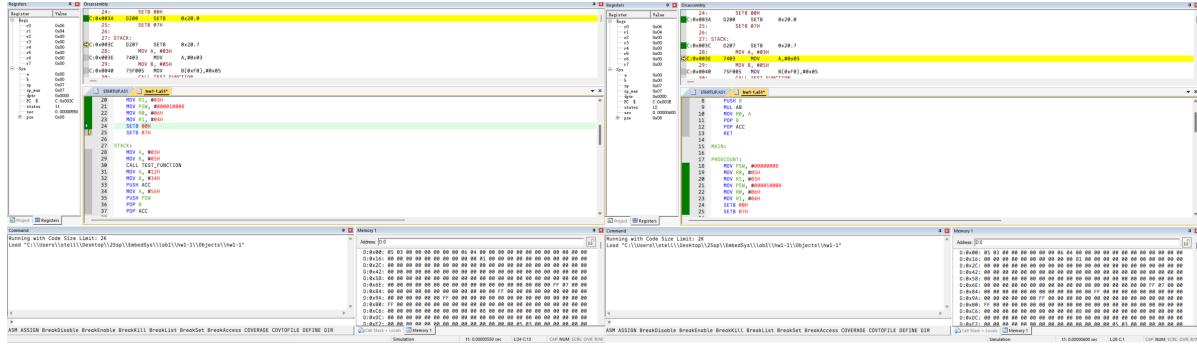
```
MOV PSW, #0000000B
MOV R0, #05H
MOV R1, #03H
MOV PSW, #00001000B
MOV R0, #06H
MOV R1, #04H
SETB 00H
SETB 07H
```



**Figure 13:** 注意左图程序指针的数值 0x002C, 而通过反汇编我们可以看见此时即将执行的 MOV 指令就是处在 0x002C 的地方, 并且此指令占了 3 个地址, 因此在完成执行后 PC 指针直接向后跳到了 0x002F



**Figure 14:** 理论上左右执行的两段代码都是向 R0, R1 存入了数据, 但是我们可以看见此处我们中途更改了一次 PSW 值, 故实际上是存入了两组寄存器中. 这在右下角的数据存储器中也可以看出来, 05, 03; 06, 04 分别存入了 0x00, 0x01, 0x08, 0x09 中, 并没有发生覆盖



**Figure 15:** 此处如果我们是使用的 SETB 这种针对一位的操作的话, 其会自动将后面的地址理解为位地址中的对应位(例如 00H 实际上是对应的 20H 的第一位), 即此处我们可以看见我们在 SETB 00H, SETB 07H 后得到的结果是 20H 位为 81, 也就是 10000001B.

### 3.4 更多例子

ORG 0000H

LJMP START

ORG 040H

START:

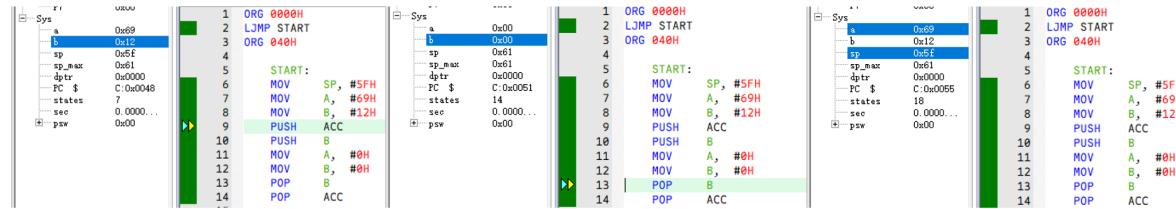
```
MOV SP, #5FH
MOV A, #69H
MOV B, #12H
PUSH ACC
PUSH B
MOV A, #0H
MOV B, #0H
POP B
POP ACC
```

LOOP:

NOP

LJMP LOOP

END



**Figure 16:** 运行结果, 可见其很好地体现了在堆栈对寄存器中数值的保护

## 4 数字键盘控制的 7 段 LED 显示

### 4.1 原理

7 段 LED 灯, 顾名思义就是由 7 段 LED 灯管组成的, 可以显示 0~9 数字的 LED 灯管, 其可以用 7 个 I/O 端口控制, 不同组合(即不同的接口电位的高低状态组合)则会使它显示不同的数字

矩阵键盘, 同样用 7 个接口与单片机连接(矩阵键盘规格为 3\*4, 即三个接口接列, 四个接口接行), 我们会首先对所有的 I/O 口加上高电平, 然后我们会对某一行/列接口加上低电平, 若该行/列有按钮被按下, 则对应所在列/行的出口接口同样会为低电平

## 4.2 代码与电路

```

ORG 0000H
AJMP START

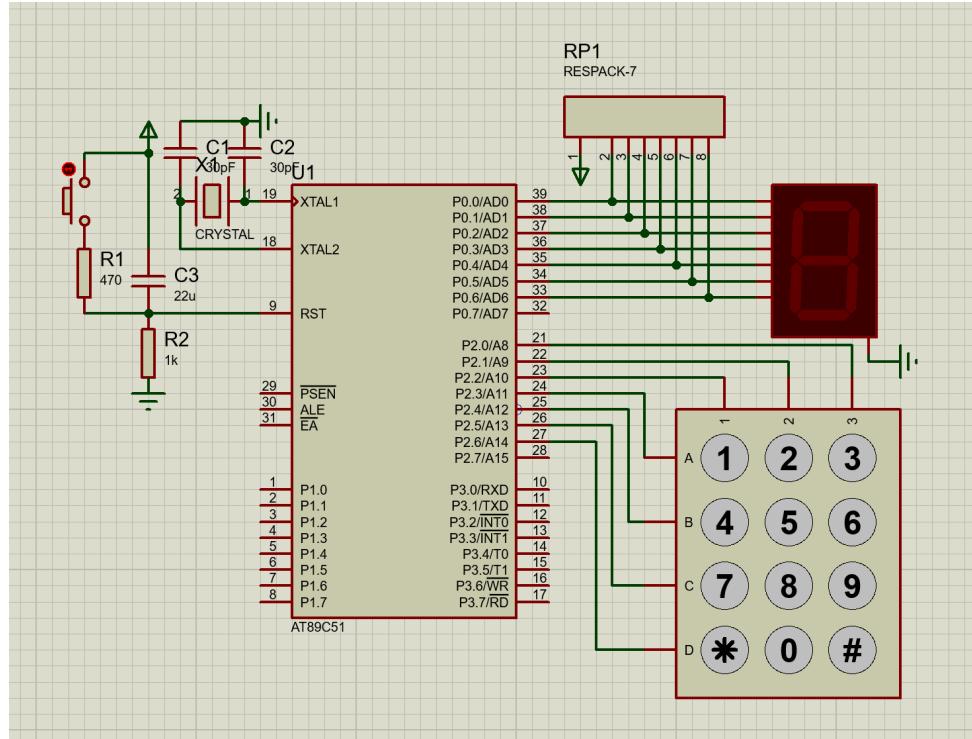
ORG 0040H
START:
    MOV P0, #00H
TEST:
    MOV P2, #0FFH

    CLR P2.3
    JNB P2.2, NUM1
    JNB P2.1, NUM2
    JNB P2.0, NUM3
    SETB P2.3
    CLR P2.4
    JNB P2.2, NUM4
        JNB P2.1, NUM5
        JNB P2.0, NUM6
    SETB P2.4
    CLR P2.5
        JNB P2.2, NUM7
        JNB P2.1, NUM8
        JNB P2.0, NUM9
    SETB P2.5
    CLR P2.6
        JNB P2.1, NUM0
    SETB P2.6
    SJMP TEST

NUM0:
    MOV P0,#3FH
    SJMP TEST
NUM1:
    MOV P0,#06H
    SJMP TEST
NUM2:
    MOV P0,#5BH
    SJMP TEST
NUM3:
    MOV P0,#4FH
    SJMP TEST
NUM4:
    MOV P0,#66H
    SJMP TEST
NUM5:
    MOV P0,#6DH
    SJMP TEST
NUM6:
    MOV P0,#7DH
    SJMP TEST
NUM7:
    MOV P0,#07H
    SJMP TEST
NUM8:
    MOV P0,#7FH
    SJMP TEST
NUM9:
    MOV P0,#6FH

```

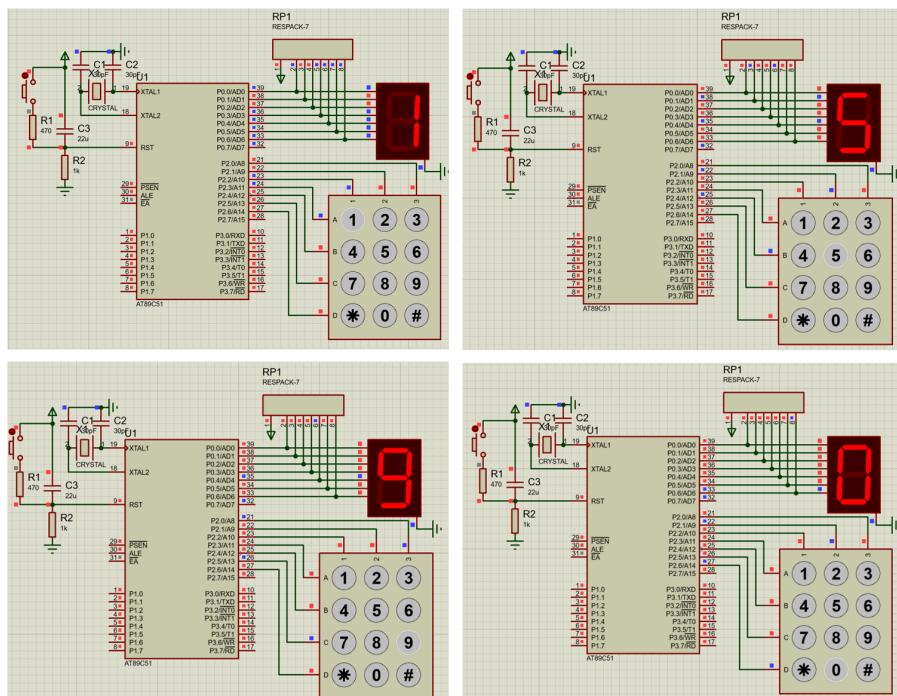
SJMP TEST  
END



**Figure 17:** 7 段 LED 与矩阵键盘的电路

## 4.3 运行效果

我们把代码产生的 hex 文件载入电路模拟中, 可以发现其有按照我们的目标运行



**Figure 18:** 通过观察电位指示与 LED 显示, 可以认为电路行为符合我们预期