作 者：道哥，10+年的嵌入式开发老兵。

公众号：【IOT物联网小镇】，专注于：C/C++、Linux操作系统、应用程序设计、物联网、单片机和嵌入式开发等领域。 公众号回复【书籍】，获取 Linux、嵌入式领域经典书籍。

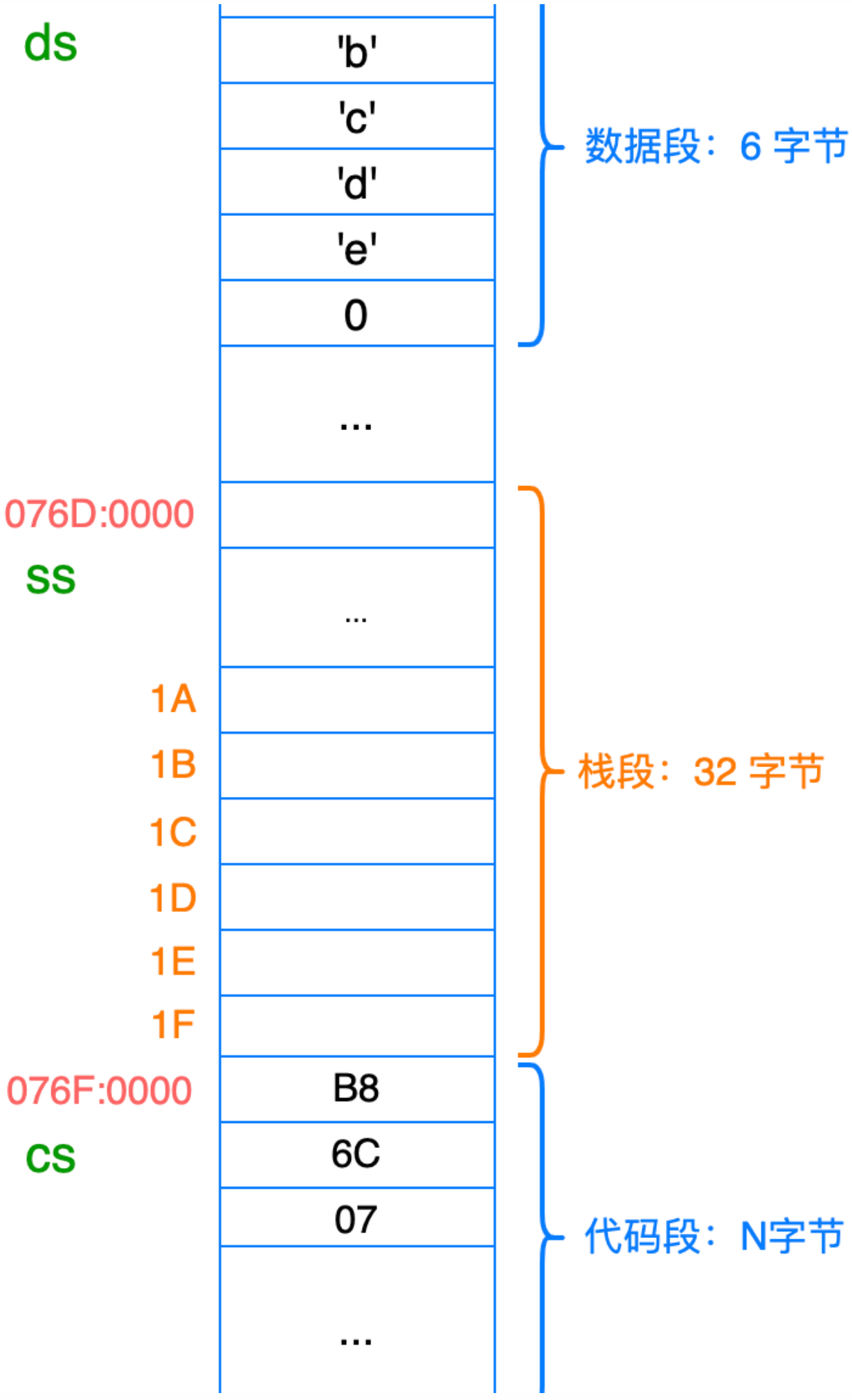
在任何一门编译型语言中，栈操作都是非常重要的。

利用栈的后进先出特性，可以很方便的解决一些棘手的问题，以至于 CPU 单独分配了 push 和 pop 这两个命令来专门操作栈，当然了，还有其他一些辅助的栈操作指令。

对于一些解释型的脚本语言，比如：Javascript、Lua 等，它们与宿主语言之间的参数传递也都是通过栈来操作的。

因此，理解了栈操作的基本原理，对于学习、理解高级语言是非常有帮助的。

这篇文章，我们继续从最底层的指令码入手，通过一个子程序调用(即：函数调用)，来学习栈空间是如何操作的，也就是下面这张图：


ds
'b'
'c'
'd'
'e'
0
...
数据段：6 字节
076D:0000
SS
...
1A
1B
1C
1D
1E
1F
栈段：32 字节
076F:0000
CS
B8
6C
07
...
代码段：N字节

示例虽然是汇编代码，但是指令码一共不超过10个，而且每一句都有注释，相信你阅读一定没有问题！

> 再次重申：我们不是在学习汇编语言，只是利用汇编代码，去繁存简，用最简单的实例来理解栈的操作。

示例代码说明

代码的功能是：

> 主程序：设置数据段、栈段、栈顶这 3 个寄存器，然后调用子程序(函数调用);
>
> 子程序：从寄存器 si 中获取字符串开始地址，然后计算字符串的长度，最后通过寄存器 ax 返回给主程序;

主程序在调用子程序的时候，就涉及到返回地址的入栈、出栈操作。

子程序在计算字符串长度的时候，为了保护一些使用到的寄存器不被破坏，也涉及到入栈和出栈操作。

我们的主要目标就是来研究以上这2部分操作时，栈空间里的数据变化情况。

具体的代码说明如下：

```
assume cs:code, ss:stack, ds:data

data segment
    db 'abcde',0
data ends

stack segment
    db 32 dup (0)
stack ends

code segment
start:
    mov ax, data
    mov ds, ax
    mov ax, stack
    mov ss, ax
    mov sp, 20h

    mov si, 0
    call getlen

    mov ax,4c00h
    int 21h

getlen:
    ; 这里是子程序代码，右侧部分
ret

code ends
end start
```

定义数据段(6 字节)

定义栈段(32 字节)

定义代码段开始地址
程序入口地址

设置数据段寄存器 ds

设置栈段寄存器 ss

设置栈顶寄存器

传递参数: 字符串首地址

调用子程序

程序退出

定义代码结束

通知编译器：程序入口地址

```
getlen:
    push bx
    push cx
    mov bx, 0
s:
    mov cl, [si]
    mov ch, 0
    jcxz over
    inc bx
    inc si
    loop s

over:
    mov ax, bx
    pop cx
    pop bx
    ret
```

bx 内容入栈

cx 内容入栈

bx 内容清零

循环入口

读取一个字符

检查字符是否为0

bx 递增，表示字符串长度

si 递增，获取下一个字符

跳入循环标号：s

子程序返回结果

cx 内容出栈

bx 内容出栈

子程序返回

执行主程序

以下演示的截图，是通过debug.exe这个工具来调试的。

在调试的过程中，主要关心的就是栈空间中的数据，以及几个寄存器的值：

> 代码相关： cs, ip
> 栈相关：ss, sp

初始状态

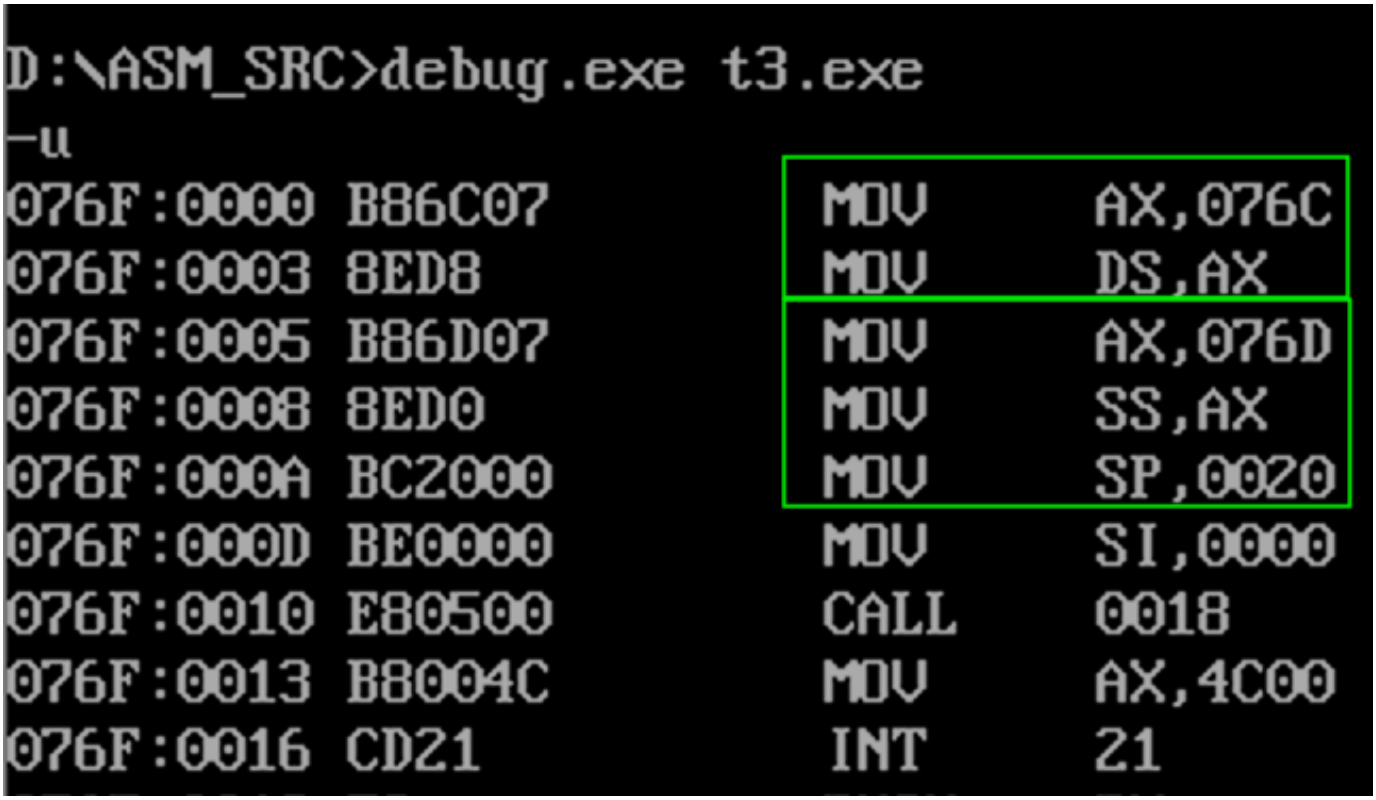
在执行第一条指令之前，首先看一下所有寄存器中的值：

```
D:\ASM_SRC>debug.exe t3.exe
-r
AX=FFFF  BX=0000  CX=005C  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075C  ES=075C  SS=076B  CS=076F  IP=0000   NV UP EI PL NZ NA PO NC
076F:0000 B86C07        MOV     AX,076C
-
```

此时，我们还没有为数据段寄存器ds、栈段寄存器ss赋值，因此里面的值是没有意义的。

只有 cs:ip 寄存器的值是有意义的，此时它们为 076F:0000，指向第一条代码处。

再来看一下指令码：



两个绿框内的指令，就是用来设置数据段寄存器 ds、栈段寄存器 ss 和 栈顶寄存器 sp。

这部分内容在上一篇文章中都已经详细描述过了，这里就不重复了。

执行代码前 5 句

```
mov ax, data
mov ds, ax

mov ax, stack
mov ss, ax
mov sp, 20h
```

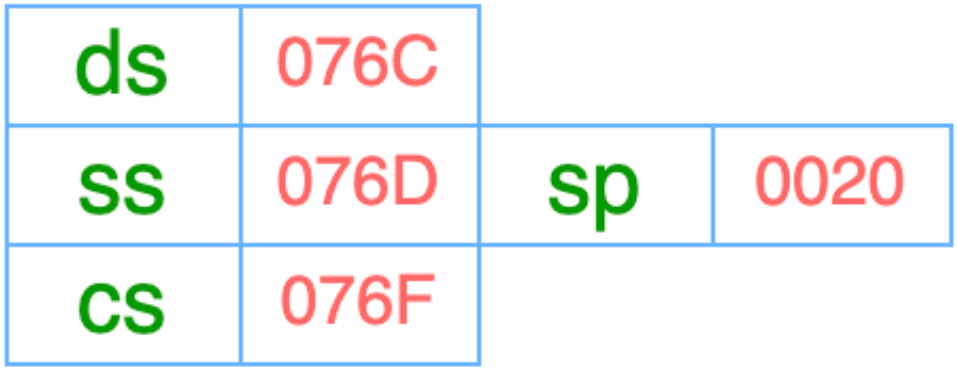
这 5 行代码的功能就是：设置 ds、 ss 和 sp。

执行完这 5 行代码后，寄存器中的值为：

```
AX=076D  BX=0000  CX=005C  DX=0000  SP=0020  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076D  CS=076F  IP=000D   NV UP EI PL NZ NA PO NC
```

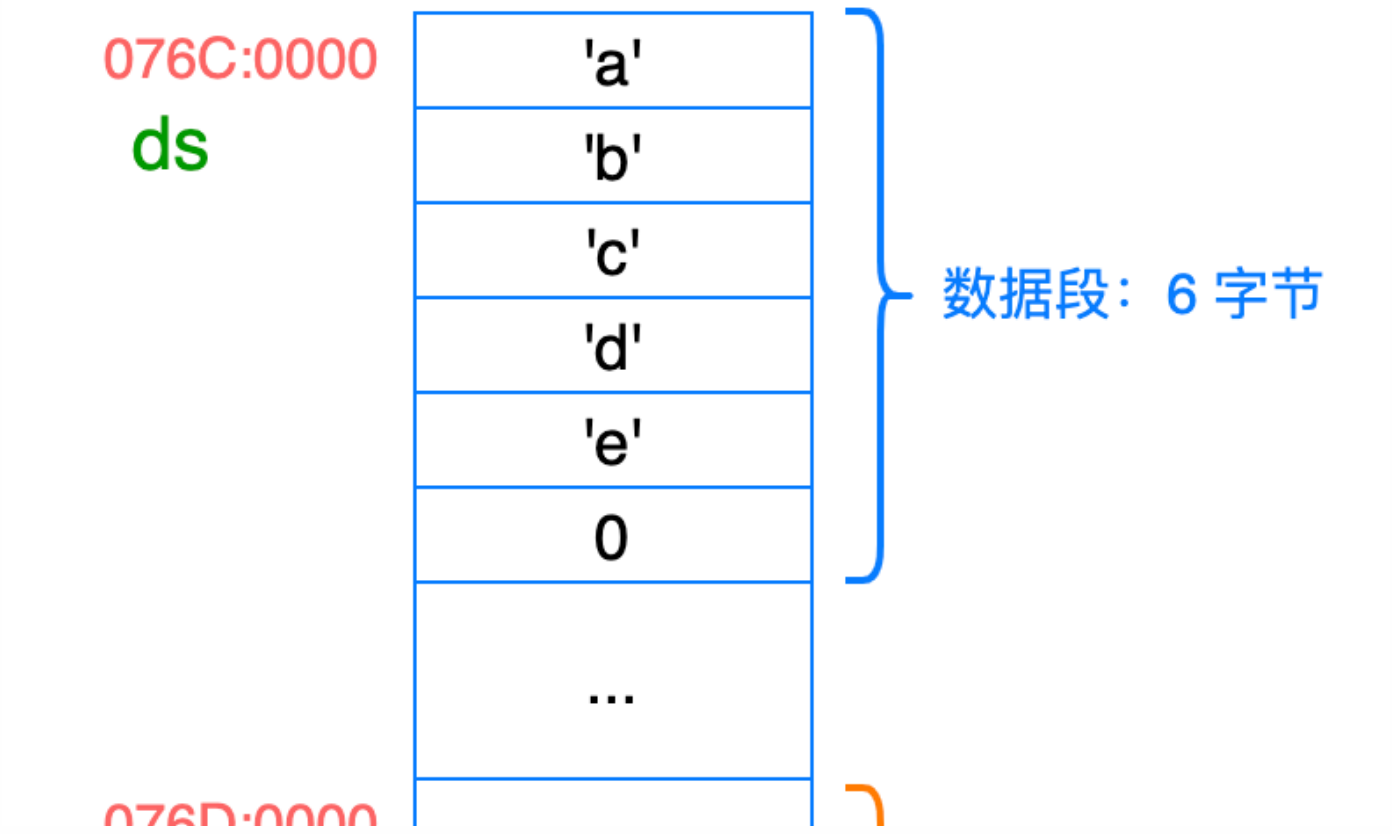
从以上这张图中可以看到编译器为程序安排了下面这几个地址：

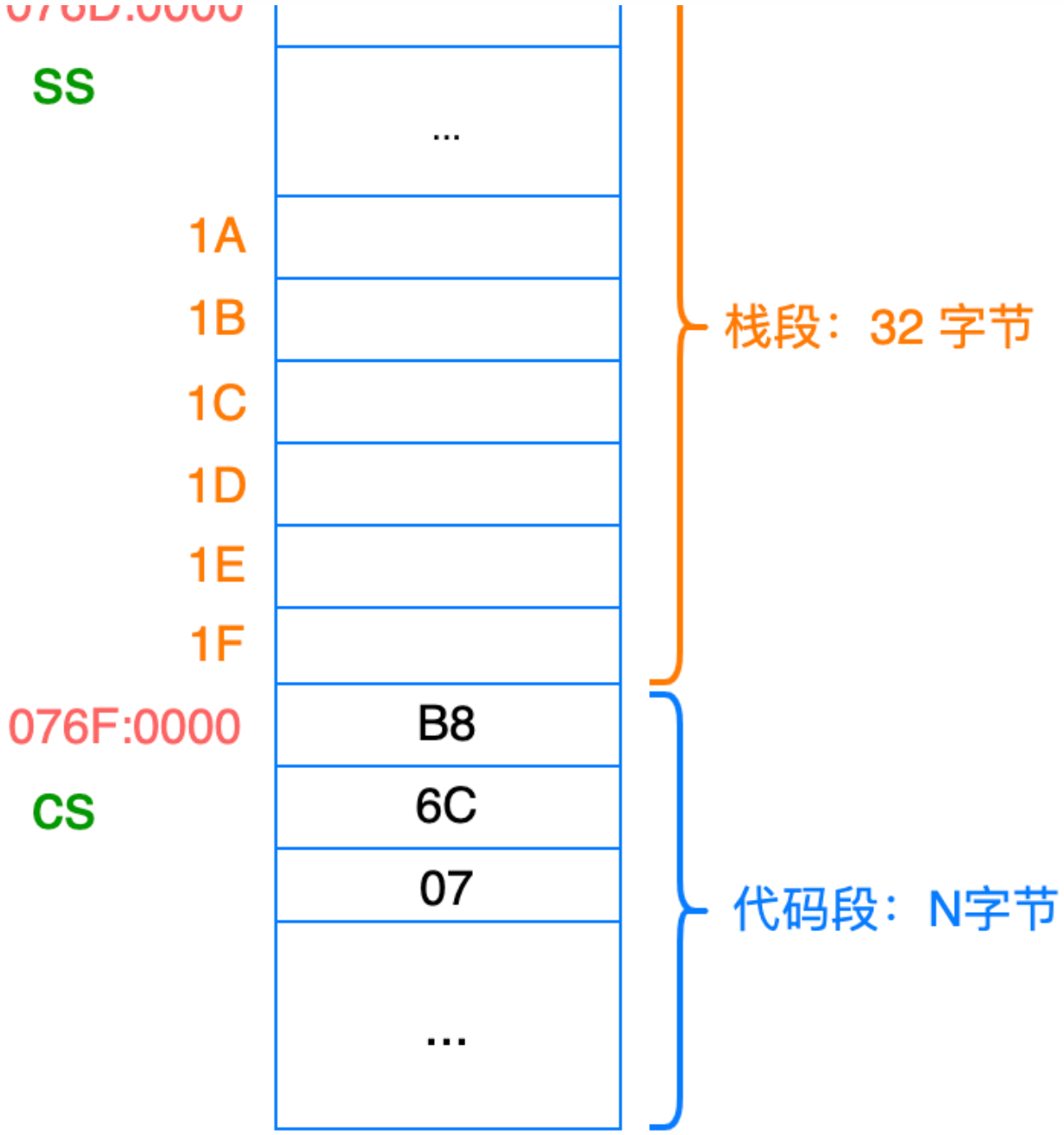
> 把【数据段】安排在 076C:0000 位置;
> 把【栈段】 安排在 076D:0000 位置;
> 把【代码段】安排在 076F:0000 的位置;



虽然数据段值定义了 6 个字节的数据(5 个字符 + 1 个结束符)，但是它与栈段的开始地址之间，还是预留了 16 个字节的空间。

我们把此时内存空间的整体布局画一下：





准备调用子程序

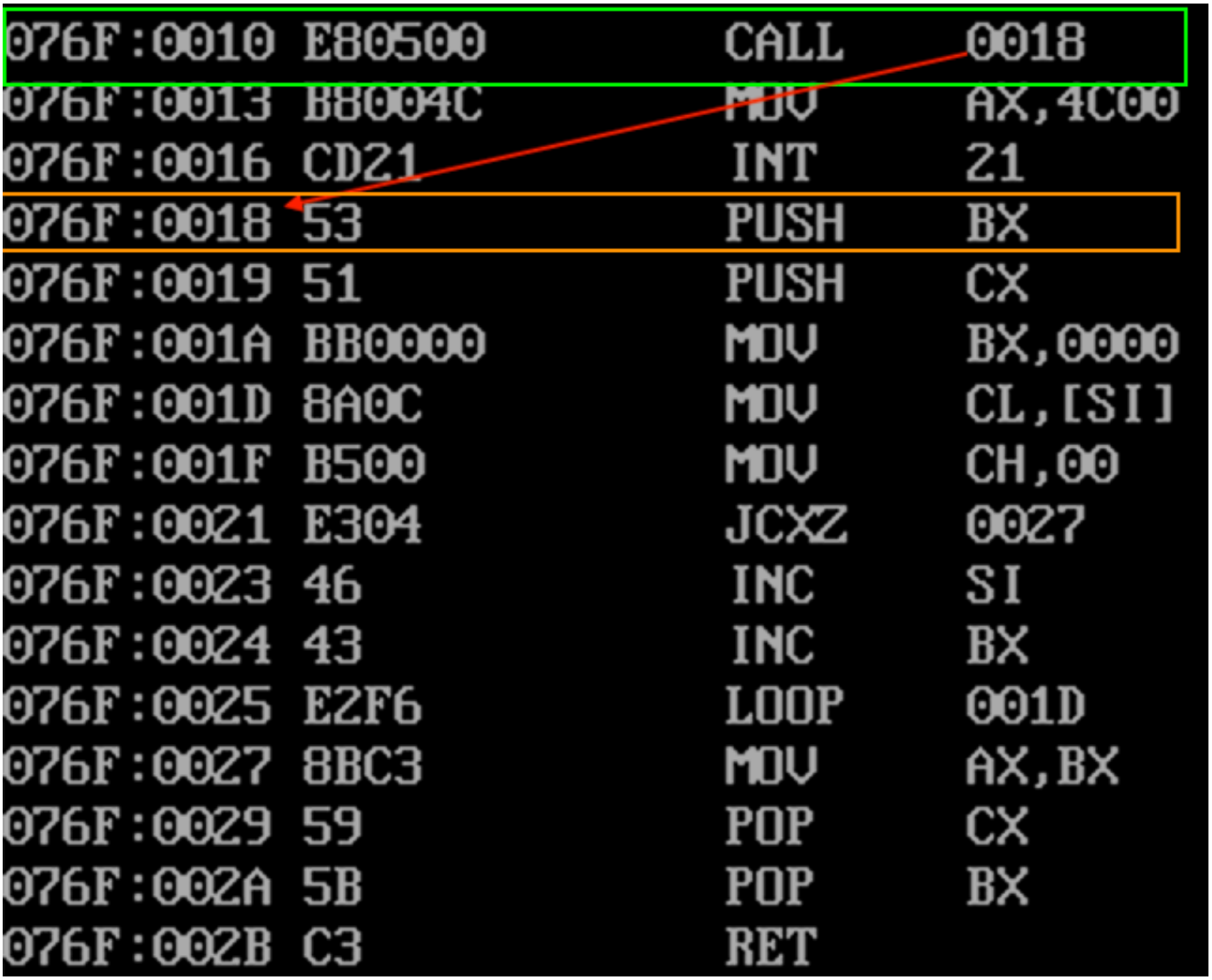
我们都知道，在调用函数的之后，需要把调用指令后面的那条指令的地址，压入到栈中。

只有这样，被调用函数在执行结束之后，才能继续返回到正确的指令处继续执行。

CPU 在执行 call 指令的时候，会自动把 call 指令的后面一条指令的地址，压入到栈中。

在执行 call 指令之前，我们先来看一下 2 张图片。

1. call 的指令码和汇编代码



`call` 的汇编代码是：`call 0018`。

`0018` 指的是指令寄存器 `ip` 的值，加上代码段寄存器 `cs`，就是：`076F:0018`，这个位置处存储的就是子程序的第一条指令：`push bx`。

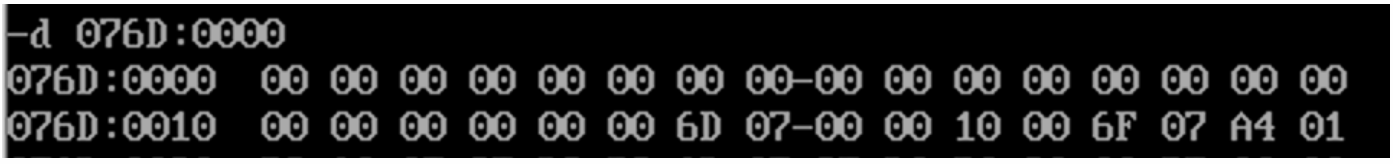
注意：`call` 的指令码是 `E80500`，`E8` 是 `call` 指令的操作码，`0005` 是指令参数(注意：低字节是放在低地址，即：小端模式)。

之前文章说过，CPU 在执行一条指令后，会自动把指令寄存器 `ip` 修改为下一条指令的地址。

当 `call` 这条指令执行时，`ip` 就自动变成下一条指令的地址，再加上 `call` 指令中的 `0005`，也就是说让 `ip` 再加上这个值，就是子程序的第一条指令的地址。

这也是相对地址的概念！在以后介绍到重定位的时候，再继续聊这个话题。

2. 栈空间的数据



此时，栈顶寄存器 sp 的值为 0020，即：栈的最高地址的下一个位置(为什么是这个位置？上一篇文章有说明)。

这 32 个字节的内容是没有任何意义的。

因为栈里数据是否有意义，是依赖于 sp 寄存器的，可以把它理解成一个指针，有些书籍中称呼它为：栈顶指针。

调用子程序

子程序的功能是计算字符串的长度，那么主程序一定要告诉子程序：字符串的开始地址在哪里。

在代码的开头，我们放置了 6 个字节的数据段空间，内容是 5 个字符，加上一个 0。

主程序把第一个字符的地址 0，通过寄存器 si 来告诉子程序：mov si, 0。

子程序在执行时，就从 si 的值所代表的地址处，依次取出每一个字符。

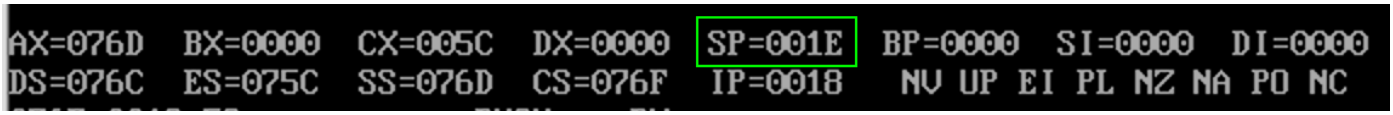
现在我们开始执行 call 指令。

从上面的描述中可以知道：call 的下一条指令的地址(076F:0013)，将会被压入到栈中。

由于这里 call 指令是段内跳转，不会把 cs 的值入栈，仅仅是把 ip 的值入栈。(如果是段间跳转的话，就会把 cs:ip 都压栈)

我们来看一下执行 call 指令之后的两张图：

1. 寄存器的值



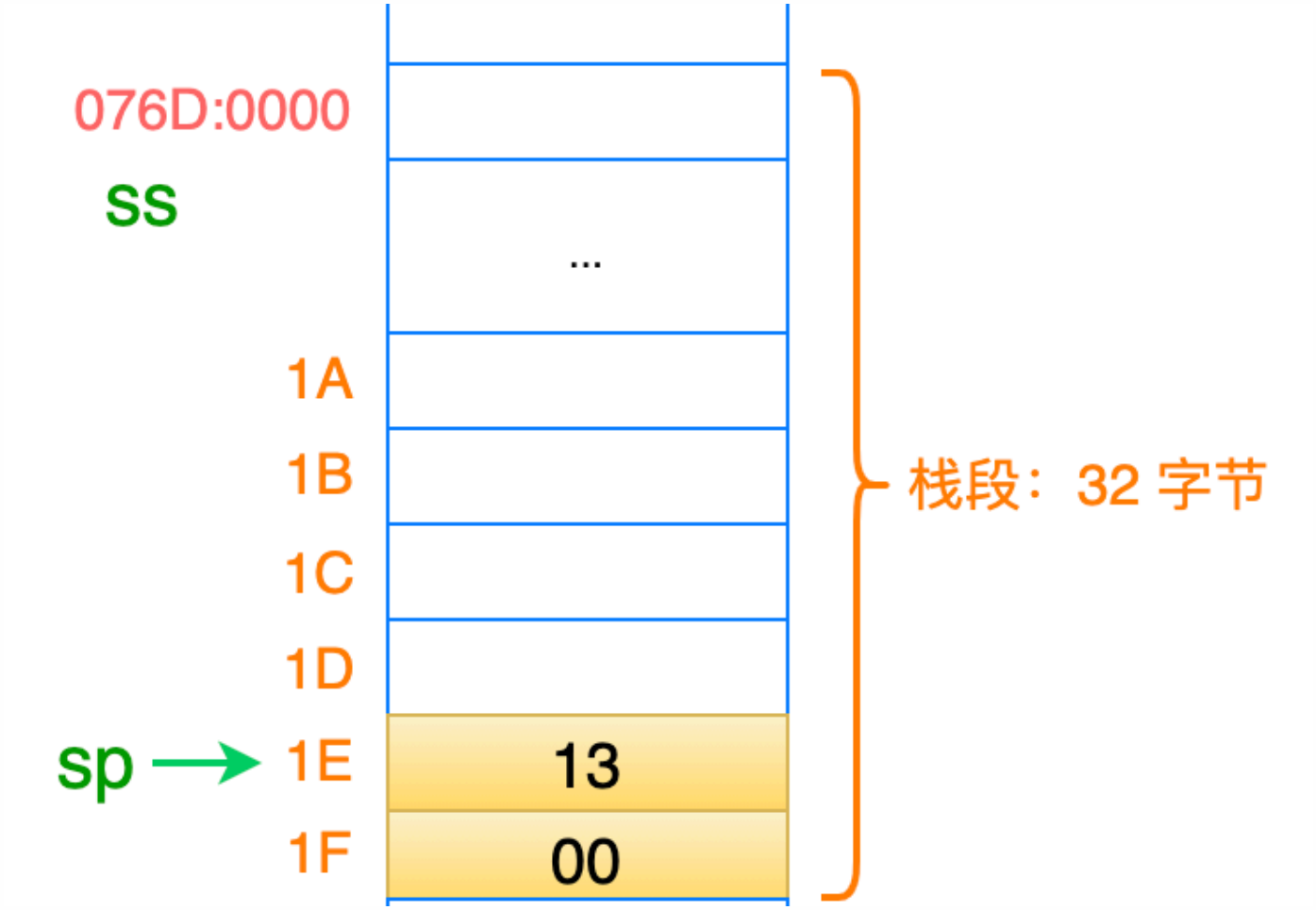
从图中看出 sp 的值变成了 001E。还记得之前文章说的入栈操作吗?

> Step1：sp = sp -2。由于 sp 的初值是 0020，减去 2 之后就是 001E（都是十六进制）；
>
> Step2：把要入栈的值(也就是下一条指令的地址 0013)放在 sp 指向的地址处。

从图中还可以看到，指令寄存器 ip 的值变成了 0018，也就是子程序的第 1 条指令(push bx)的地址。

2. 栈空间的数据

可以看到：最后 2 个字节是 0013，也即是下面的这样：



此时，指令寄存器 ip 指向了子程序的第一条指令 076F:0018 处，那就继续执行吧！

子程序

保护使用到的寄存器

我们知道：CPU 中寄存器都是公用的。

在子程序中，为了计算字符串的长度，代码中用到了bx, cx 这 2 个寄存器。

但是我们不知道这 2 个寄存器是否在主程序中也被使用了。

如果我们冒然直接使用它们，改变了它们的值，那么在子程序执行结束后，返回到主程序时，主程序如果也用了这 2 个寄存器，那就有麻烦了。

因此，在子程序的开始处，需要把 bx, cx 放在在栈中进行暂存保护。

当子程序返回的时候，再从栈中恢复它们的值，这样就不会对主程序构成潜在的威胁了。

1. push bx

在入栈之前，bx 的值是 0000，我们给他入栈。

还记得上篇文章中入栈的操作吗：

> Step1: 把 sp 的值减 2;
> Step2: 把要入栈的值放在 sp 地址处(2个字节);

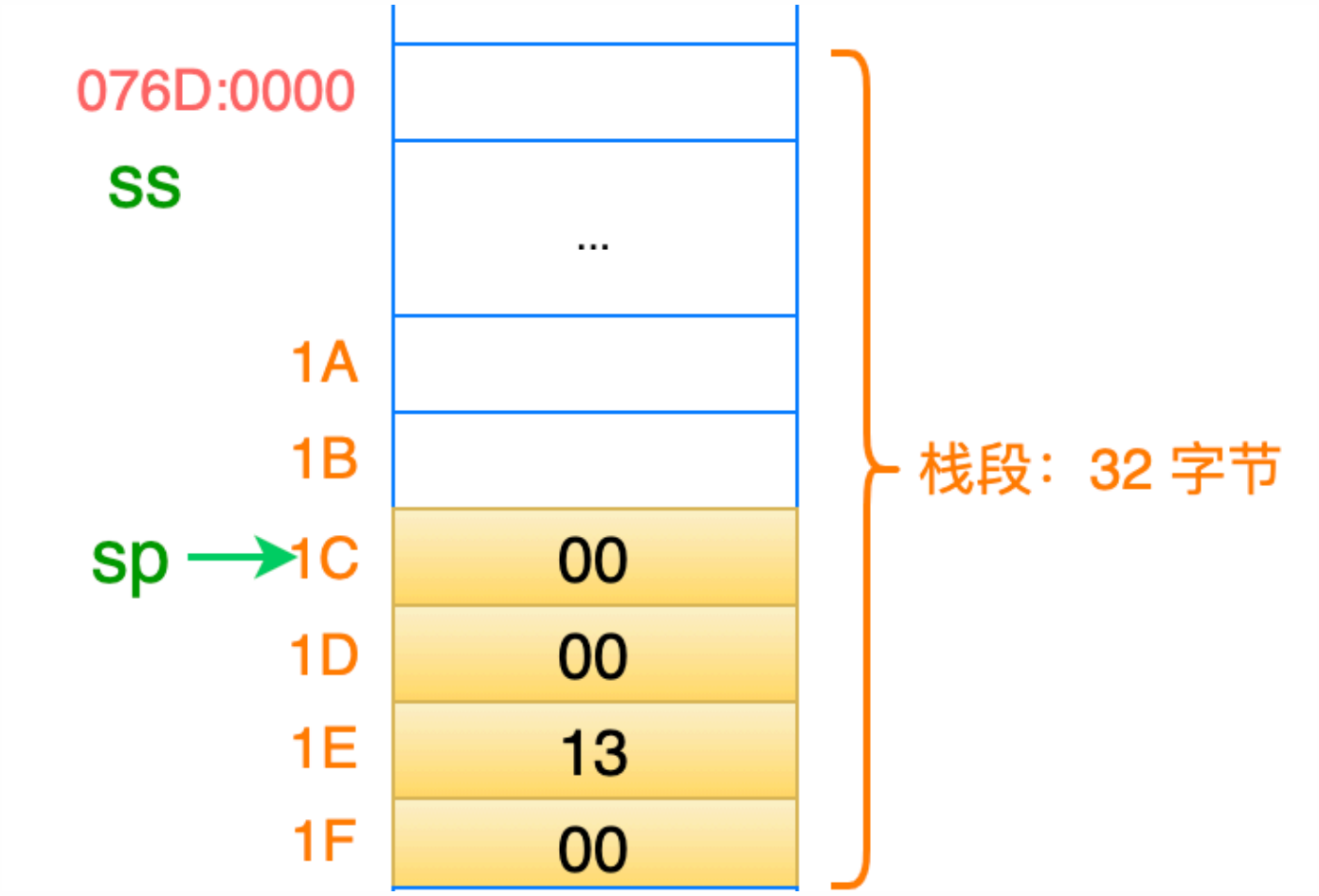
此时，栈顶寄存器 sp 变成 001C (001E - 2)。

```
AX=076D  BX=0000  CX=005C  DX=0000  SP=001C  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076D  CS=076F  IP=0019   NV UP EI PL NZ NA PO NC
```

ba

再来看一下栈空间的数据情况：

```
- d 076D:0000
076D:0000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
076D:0010  00 00 6D 07 00 00 19 00-6F 07 A4 01 00 00 13 00
```



此刻，栈中有意义的数据就有 2 个：返回地址，bx 的值。

2. push cx

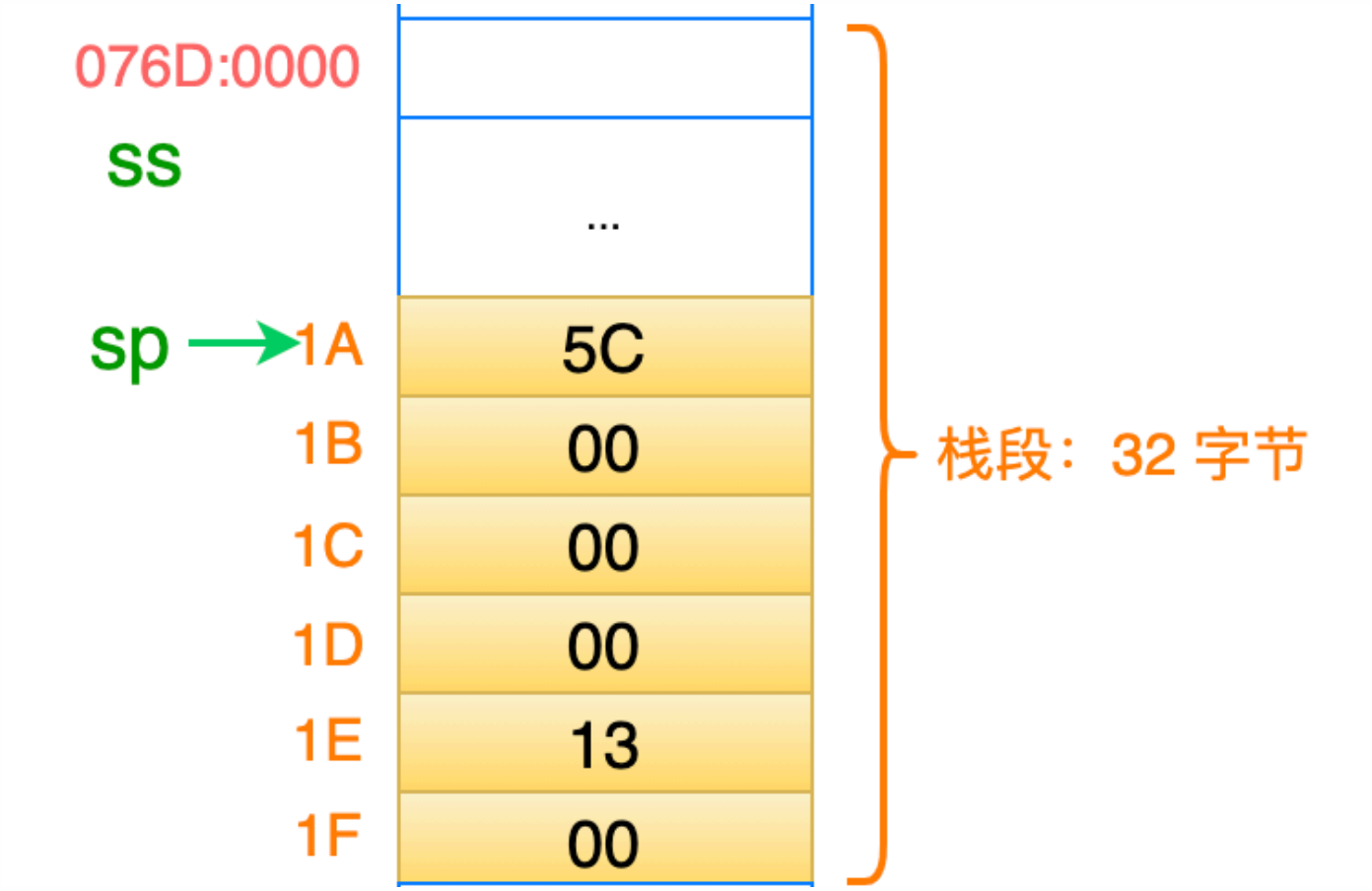
在入栈之前，cx 的值是 005C，我们给他入栈。

执行入栈的 2 步操作之后，栈顶寄存器 sp 变成 001A (001C - 2)。

```
AX=076D  BX=0000  CX=005C  DX=0000  SP=001A  BP=0000  SI=0000  DI=0000
DS=076C  ES=075C  SS=076D  CS=076F  IP=001A   NV UP EI PL NZ NA PO NC
076F:001A BB0000        MOV     BX,0000
```

栈空间的数据情况：

```
-d 076d:0000
076D:0000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
076D:0010  6D 07 00 00 1A 00 6F 07-A4 01 5C 00 00 00 13 00
```



3. 计算字符串的长度

字符串是放在数据段中的。数据段的段地址 ds，在主程序的开头已经设置好了。

字符串的首地址，主程序在执行 call 指令之前，已经放在寄存器 si 中了。

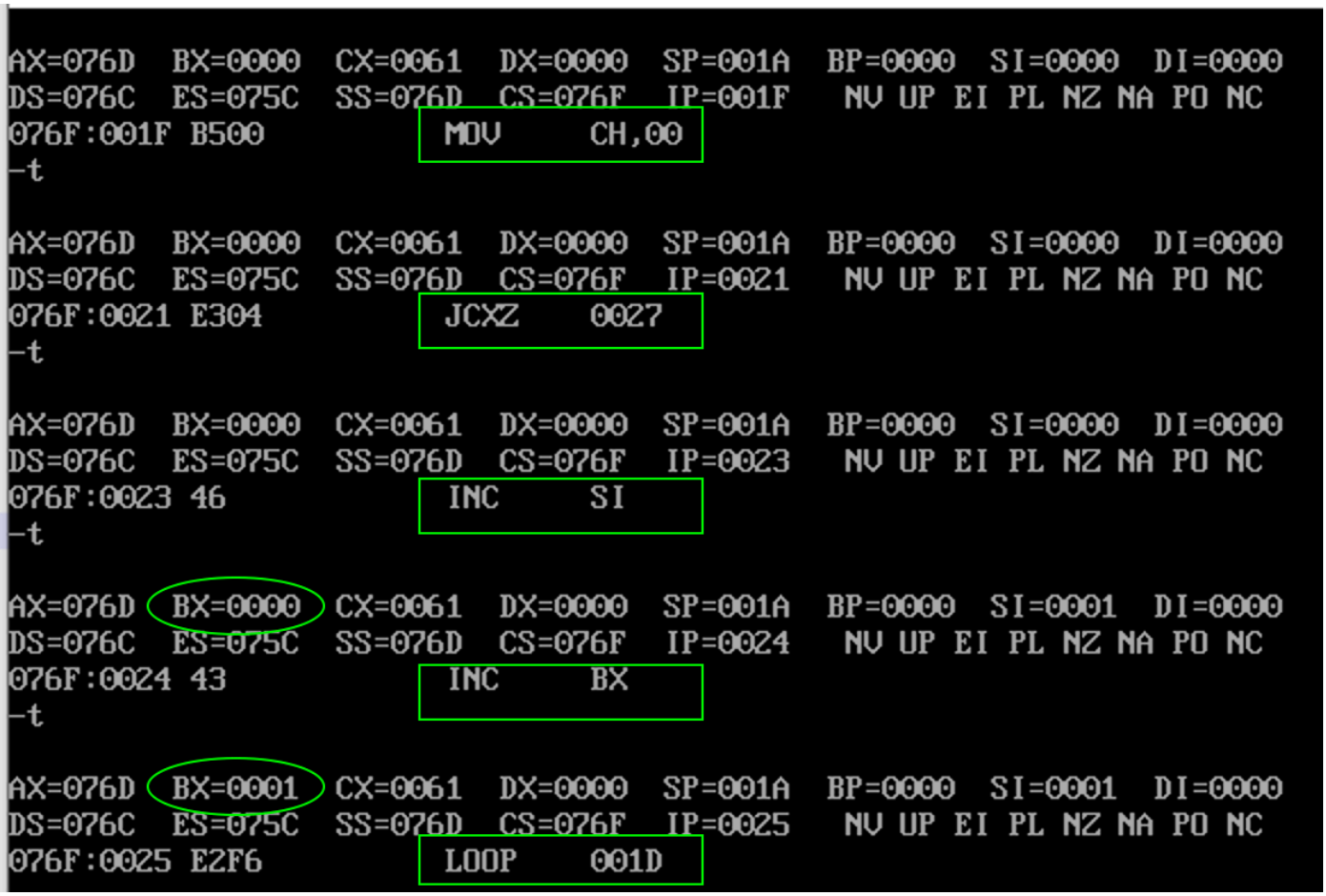
因此，子程序只要从 si 开始位置，依次取出每一个字符，然后检查它是否等于 0 (jcxz)。

> 如果不为0，就把长度值加 1 (inc bx)，然后继续取下一个字符(inc si)；
>
> 如果为0，就停止获取字符，因为已经遇到了字符串末尾的 0。

在循环获取每一个字符的时候，可以用 bx 寄存器来记录长度，所以在子程序的开头要让 bx 入栈。

读取的每个字符，放在 cx 寄存器中，所以在子程序的开头要让 cx 入栈。

我们来看一下检查第一个字符 'a' 的情况：



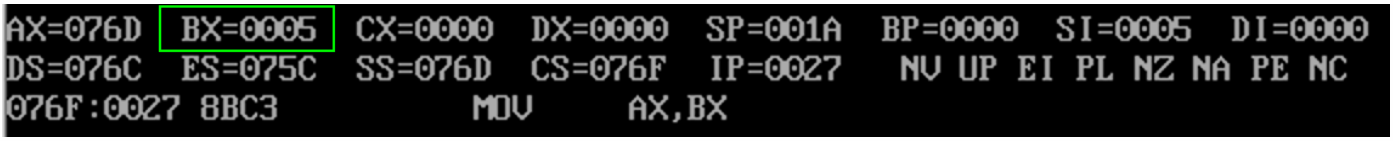
此时：

bx 的值为 0001，说明长度至少为 1。

si 的值为 0001，准备取下一个位置 ds:si（即：076C:0001）处的字符 'b'。

这个过程一直循环 6 次(loop s)，当 ds:si 指向 076C:0005，也就是取出的字符为 0 时，就直接跳转到标号为 over （即：076F:0027）的地址处。

此刻，寄存器 bx 中就存放着字符串的长度：0005：



4. 把字符串长度告诉主程序

字符串的长度计算出来了，我们要把这个值告诉主程序，一般都是通过通用寄存器 ax 来传递返回结果。

所以，执行指令 mov ax, bx 把 bx 的值赋值给 ax，主程序就可以从寄存器 ax 中得到字符串的长度了。

5. pop cx

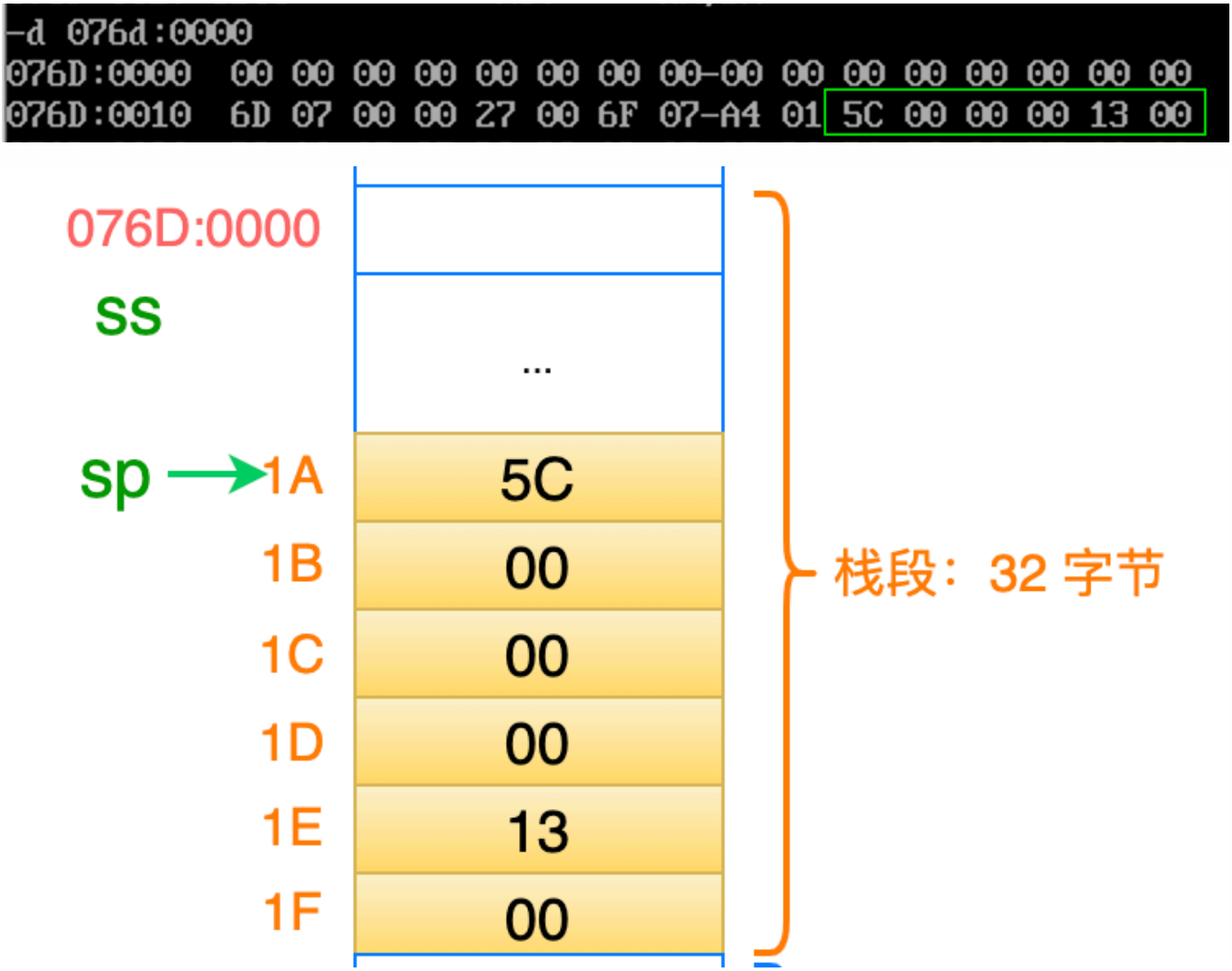
子程序在返回之前，需要把栈中保存的 bx、cx 值恢复到寄存器中。

另外，由于栈的后进先出特性，需要把栈顶数据先弹出到 cx 寄存器中。

在执行出栈之前：

> sp = 001A
> cx = 0000

栈中的数据情况如下：

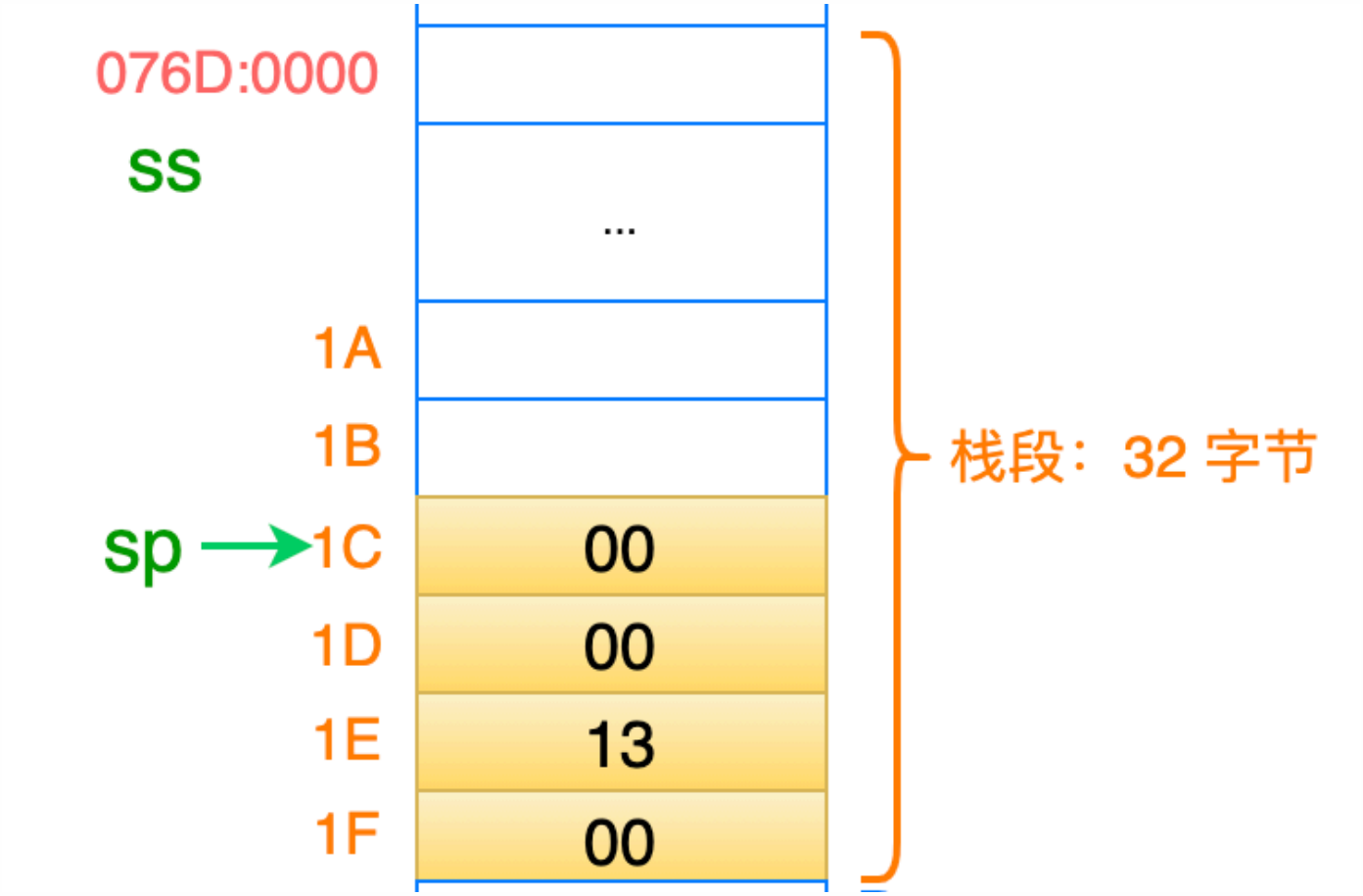


pop cx 指令分为 2 个动作：

> Step1：把 sp 指向的地址单元的中数据(2 个字节)，放入寄存器 cx 中，于是 cx 中的值变成了：005C；
>
> Step2：把 sp 的值自增 2，变成 001C (001A + 2)。

此时，栈中的数据情况：

```
-d 076d:0000
076D:0000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
076D:0010  05 00 05 00 00 00 2A 00-6F 07 A4 01 00 00 13 00
```

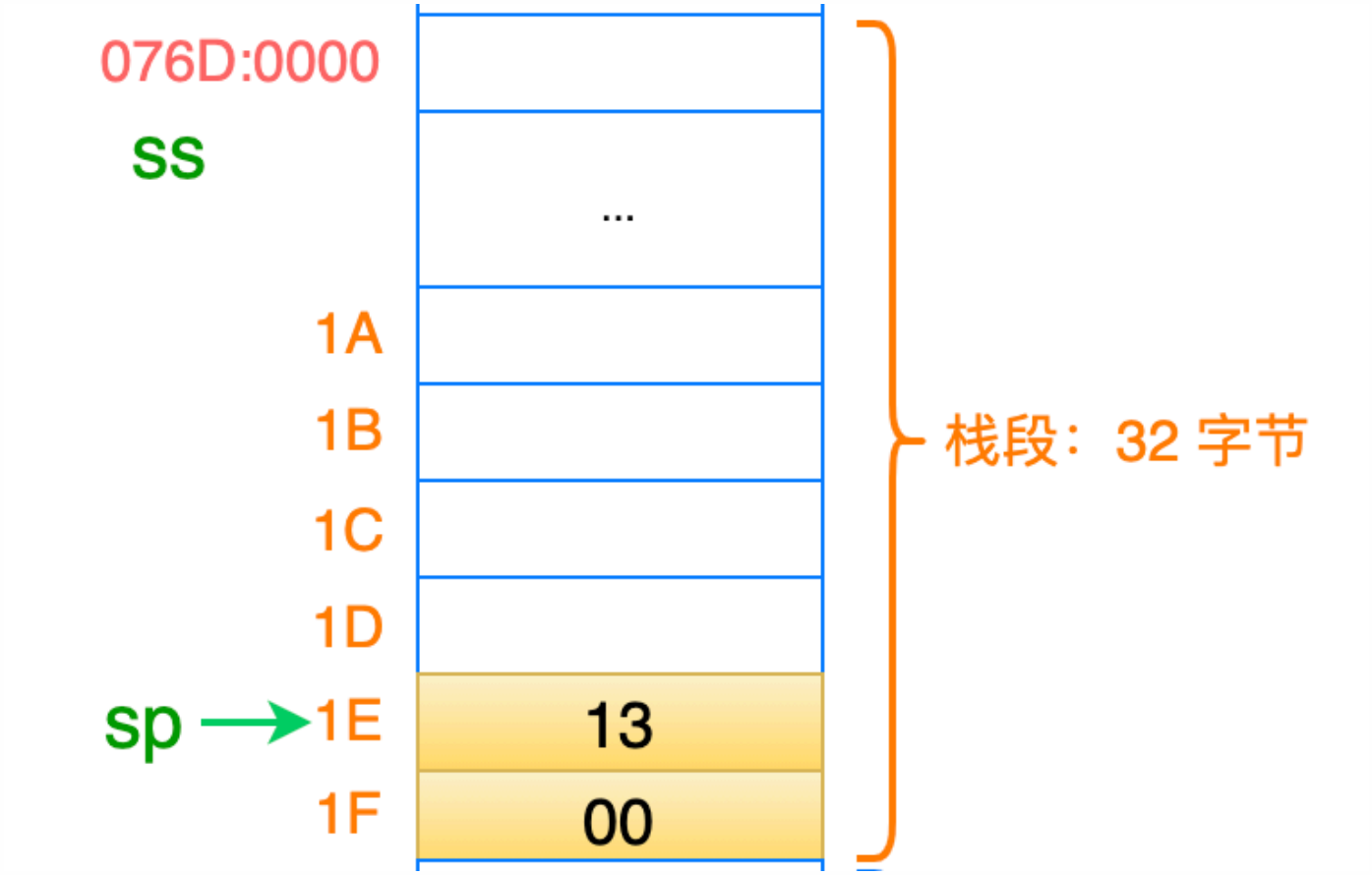


6. pop bx

执行过程是一样的：

> Step1：把 sp 指向的地址单元的中数据(2 个字节)，放入寄存器 bx 中，于是 bx 中的值变成了：0000；
>
> Step2：把 sp 的值自增 2，变成 001E (001C + 2)。

此时，栈中的数据情况：

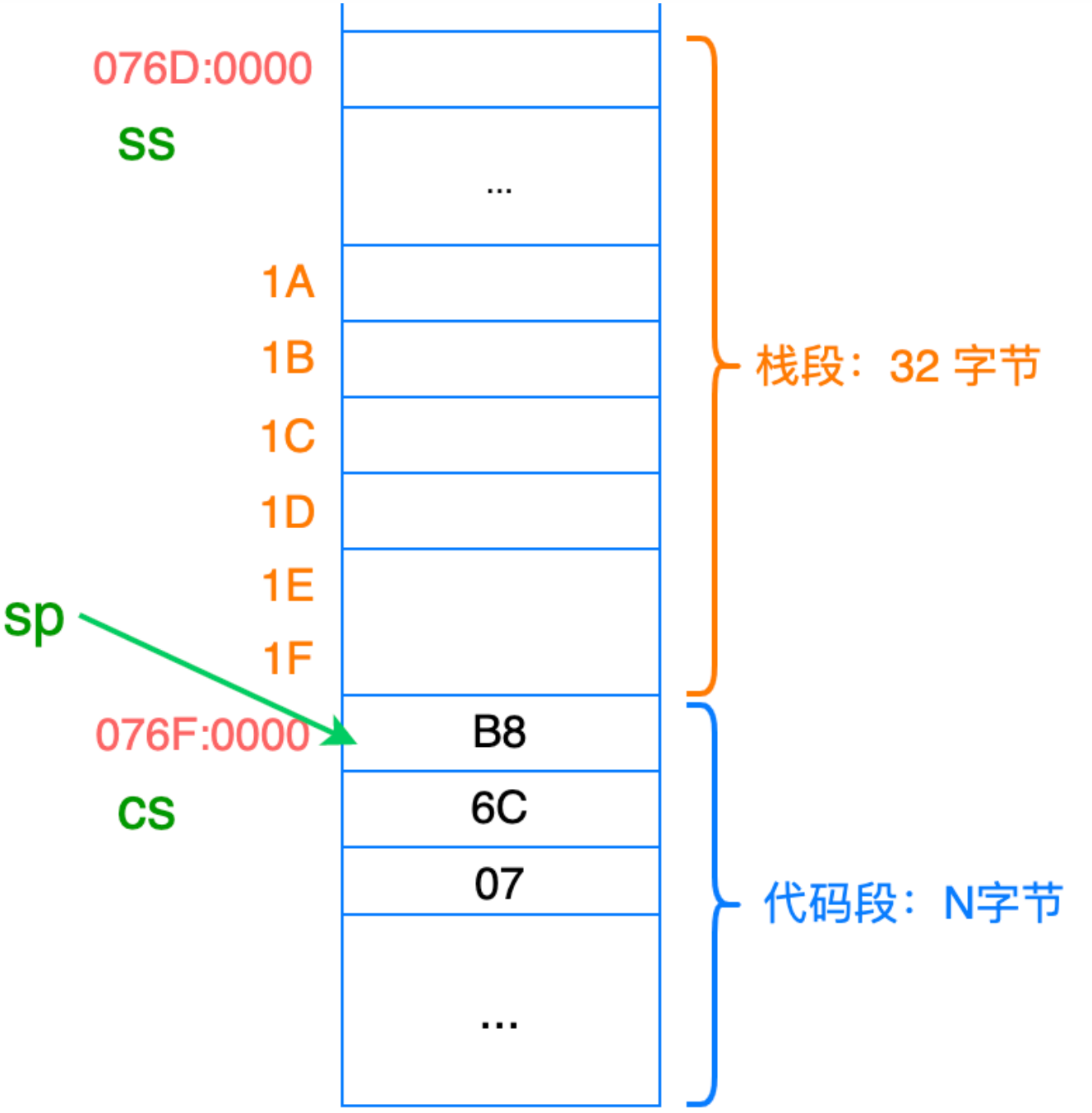


7. 返回指令 ret

CPU 在执行 ret 指令时，也有 2 个动作：

> Step1：把 sp 指向的地址单元的中数据(2 个字节)，放入指令寄存器 ip 中，于是 ip 中的值变成了：0013；
>
> Step2：把 sp 的值自增 2，变成 0020 (001E + 2)。

此时，栈中的数据情况是：



这时，栈顶寄存器 sp 已经指到了代码段的空间中。这是由于我们在刚开始安排的时候，没有在栈与代码之间，空出来一段缓冲空间。

不管怎样，此时:

> 栈空间中没有任何有意义的数据了;
> cs:ip 指向了主程序中 call 指令的下一条指令（mov ax,4c00h）;

所以，当 CPU 执行下一条指令的时候，又回到了主程序中继续执行。。。

------ End ------

推荐阅读

【1】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
【2】一步步分析-如何用C实现面向对象编程
【3】原来gdb的底层调试原理这么简单
【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、C语言、Linux操作系统、应用程序设计、物联网

星标公众号，能更快找到我！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。