目标操作数。

操作码	指　令	操作码	指　令
40	MOV　B, B	50	MOV　D, B
60	MOV　H, B	70	MOV　[HL], B
61	MOV　H, C	71	MOV　[HL], C
62	MOV　H, D	72	MOV　[HL], D
63	MOV　H, E	73	MOV　[HL], E
64	MOV　H, H	74	MOV　[HL], H
65	MOV　H, L	75	MOV　[HL], L
66	MOV　H, [HL]	76	HLT
67	MOV　H, A	77	MOV　[HL], A
68	MOV　L, B	78	MOV　A, B
69	MOV　L, C	79	MOV　A, C
6A	MOV　L, D	7A	MOV　A, D
6B	MOV　L, E	7B	MOV　A, E
6C	MOV　L, H	7C	MOV　A, H
6D	MOV　L, L	7D	MOV　A, L
6E	MOV　L, [HL]	7E	MOV　A, [HL]
6F	MOV　L, A	7F	MOV　A, A

其中的一些指令如：

```
MOV  A, A
```

并不会执行有意义的操作。而指令：

```
MOV  [HL], [HL]
```

是不存在的，事实上，与之对应的指令是 HLT（Halt）即停止指令，也就是说该指令的意义是停止。

研究 MOV 操作码的位模式能更好地了解它，MOV 操作码由 8 位组成：

```
01dddsss
```

其中 ddd 这 3 位是目标操作数的代码，sss 这 3 位是源操作数的代码。它们所表示的意义如下：

000 = 寄存器 B
001 = 寄存器 C
010 = 寄存器 D
011 = 寄存器 E
100 = 寄存器 H
101 = 寄存器 L
110 = 寄存器 HL 保存的存储器地址中的内容
111 = 累加器 A

例如，指令

```
MOV  L, E
```

对应的操作码为：

```
01101011
```

用十六进制数可表示为 6Bh。这与前面列出的表格是一致的。

可以设想一下，在 8080 的内部可能是这样的：标记为 sss 的 3 位用于 8-1 数据选择器，标记为 ddd 的 3 位用来控制 3-8 译码器以此确定哪一个寄存器锁存了值。

寄存器 B 和 C 也可以组合成 16 位的寄存器对 BC，同样我们还可以用D和 E 组成寄存器对 DE。如果这些寄存器对也包含要读取或保存的字节的存储器地址，则可以用下面的指令实现：

操作码	指　令	操作码	指　令
02	STAX　[BC], A	0A	LDAX　A, [BC]
12	STAX　[DE], A	1A	LDAX　A, [DE]

另一种类型的传送（Move）指令称做传送立即数（Move Immediate），它的助记符写做 MVI。传送立即数指令是一个双字节指令，第一个字节为操作码，第二个是数据。这个单字节数据从存储器转移到某个寄存器，或者转移到存储器中的某个存储单元，该存储单元由 HL 寄存器对寻址。

操作码	指　令
06	MVI　B, xx
0E	MVI　C, xx
16	MVI　D, xx
1E	MVI　E, xx
26	MVI　H, xx
2E	MVI　L, xx
36	MVI　[HL], xx
3E	MVI　A, xx

例如，当指令：

```
MVI     E, 37h
```

执行后，寄存器 E 存放的字节是 37h。这就是我们要介绍的第三种寻址方式——立即数寻址（immediate addressing）。

下面将列出一个操作码集，包括 32 个操作码，它们能完成 4 种基本的算术运算，这些运算在第 17 章设计处理器时我们已经熟悉了，它们是加法（ADD）、进位加法（ADC）、减法（SUB）和借位减法（SBB）。可以看到，在所有的例子中，累加器始终用于存放其中的一个操作数，同时用来保存计算结果。这些指令如下。

操作码	指　令	操作码	指　令
80	ADD　A, B	90	SUB　A, B
81	ADD　A, C	91	SUB　A, C
82	ADD　A, D	92	SUB　A, D
83	ADD　A, E	93	SUB　A, E
84	ADD　A, H	94	SUB　A, H
85	ADD　A, L	95	SUB　A, L
86	ADD　A, [HL]	96	SUB　A, [HL]
87	ADD　A, A	97	SUB　A, A
88	ADC　A, B	98	SBB　A, B
89	ADC　A, C	99	SBB　A, C
8A	ADC　A, D	9A	SBB　A, D
8B	ADC　A, E	9B	SBB　A, E
8C	ADC　A, H	9C	SBB　A, H
8D	ADC　A, L	9D	SBB　A, L

8E	ADC A, [HL]	9E	SBB A, [HL]
8F	ADC A, A	9F	SBB A, A

假如累加器 A 存放的字节是 35h，累加器 B 存放的字节是 22h，经过减法运算：

```
SUB      A, B
```

累加器中的值变为 22h，即两个字节的差。

如果累加器 A 中的值为 35h，寄存器 H 和 L 中的值分别是 10h 和 7Ch，而存储器地址 107Ch 处的字节为 4Ah，指令：

```
ADD  A, [HL]
```

把累加器中的值（35h）与寄存器对 HL 寻址（107Ch）存储器得到的数值（4Ah）相加，并把计算结果（7Fh）保存到累加器。

在 8080 中，使用 ADC 指令和 SBB 指令可以对 16 位数、24 位数、32 位数甚至更高位的数进行加法、减法运算。例如，假设现在寄存器对 BC 和 DE 各自保存了一个 16 位的数，我们要把这两个数相加，并且把结果保存在寄存器对 BC 中。具体做法如下：

```
MOV A, C     ; 低字节操作
ADD A, E
MOV C, A
MOV A, B     ; 高字节操作
ADC A, D
MOV B, A
```

在上面的计算中，用 ADD 指令对低字节相加，用 ADC 指令对高字节相加。低字节相加产生的进位会进入高字节的运算中。在这段简短的代码中，我们用到了 4 个 MOV 指令，这是因为在 8080 中只能利用累加器进行加法运算，操作数在累加器和寄存器之间来回地传送，因此在 8080 的代码中会大量使用 MOV 指令。

现在我们来讨论 8080 的标志位（flag）。第 17 章设计的处理器已经有了 CF（进位标志位）和 ZF（零标志位）两个标志位，在 8080 中又新增了 3 个标志位，包括符号标志位 SF，奇偶标志位 PF 和辅助进位标志位 AF。在 8080 中，用一个专门的 8 位寄存器来存放所有标志位，该寄存器称做程序状态字（Program Status Word，PSW）。不同的指令对标志位有不同的影响，LDA、STA 或 MOV 指令始终都不会影响标志位，而 ADD、SUB、ADC 以及 SBB 指令会影响标志位的状态，具体情况如下。

- 如果运算结果的最高位是 1，那么符号标志位 SF 标志位置 1，表示该计算结果是负数。
- 如果运算结果为 0，则零标志位 ZF 置 0。
- 如果运算结果中“1”的位数是偶数，即具有偶数性（even parity），则奇偶标志位 PF 置 1；反之，如果“1”的位数是奇数，即运算结果具有奇数性（odd parity），则 PF 置 0。由于 PF 的这个特点，有时会被用来进行简单的错误检查。PF 在 8080 程序中并不常用。
- 进位标志位 CF 的情况和第 17 章描述的稍有不同，当 ADD 和 ADC 运算产生进位或者 SUB 和 SBB 运算不发生借位时，CF 都置 1。
- 辅助进位标志位 AF 只有在运算结果的低 4 位向高 4 位有进位时才置 1。它只用于 DAA（Decimal Adjust Accumulator，十进制调整累加器）指令中。

下面的两条指令会直接影响进位标志位 CF。

操作码	指　令	含　义
37	STC	令 CF 置 1
3F	CMC	令 CF 取反

第 17 章设计的计算机可以执行 ADD、ADC、SUB 和 SBB 指令（虽然缺乏灵活性），而 8080 功能更为强大，它还可以执行 AND（与）、OR（或）、XOR（异或）等逻辑运算。不论是算术运算还是逻辑运算，都是由 8080 处理器的算术逻辑单元（ALU）来完成的。

以下是 8080 的算术运算和逻辑运算指令。

操作码	指　令	操作码	指　令
A0	AND　A, B	B0	OR　A, B
A1	AND　A, C	B1	OR　A, C
A2	AND　A, D	B2	OR　A, D
A3	AND　A, E	B3	OR　A, E
A4	AND　A, H	B4	OR　A, H
A5	AND　A, L	B5	OR　A, L
A6	AND　A, [HL]	B6	OR　A, [HL]
A7	AND　A, A	B7	OR　A, A
A8	XOR　A, B	B8	CMP　A, B
A9	XOR　A, C	B9	CMP　A, C

AA	XOR A, D	BA	CMP A, D
AB	XOR A, E	BB	CMP A, E
AC	XOR A, H	BC	CMP A, H
AD	XOR A, L	BD	CMP A, L
AE	XOR A, [HL]	BE	CMP A, [HL]
AF	XOR A, A	BF	CMP A, A

AND、XOR 和 OR 都是按位运算（bitwise operations）指令，也就是说对于这些逻辑运算指令，其操作数的每一个对应位都是独立运算的，例如：

```
MVI   A, 0Fh
MVI   B, 55h
AND   A, B
```

保存到累加器的结果将会是 05h。假如我们把 3 条指令换作 OR，则最终的结果将会是 5Fh；如果换作 XOR，则结果又变成了 5Ah。

CMP（Compare，比较）指令同 SUB 指令类似，也是把两个数相减，不同之处在于它并不在累加器中保存计算结果，计算的目的是为了设置标志位。这个标志位的值可以告诉我们两个操作数之间的大小关系。例如，我们考虑下面的指令：

```
MVI   B, 25h
CMP   A, B
```

指令执行后，累加器 A 中的值并没有变化。改变的是标志位的值，如果 A 中的值等于 25h，则零标志位 ZF 置 1；如果 A 中的值小于 25h，则进位标志位 CF 置 1。

同样的，也可以对立即数进行这 8 种算术逻辑操作。

操作码	指　令	操作码	指　令
C6	ADI A, xx	E6	ANI A, xx
CE	ACI A, xx	EE	XRI A, xx
D6	SUI A, xx	F6	ORI A, xx
DE	SBI A, xx	FE	CPI A, xx

例如，可以用下面的这条指令来替代上面列出的两条指令：

```
CPI     A, 25h
```

下面是两种特别的 8080 指令。

操作码	指　令
27	DAA
2F	CMA

CMA 是 Complement Accumulator 的简写。它对累加器中的数按位取反，即把 0 变为 1，1 变为 0。例如，累加器中的数如果是 01100101，使用 CMA 命令后，累加器中的数按位取反，得到 10011010。我们还可以使用如下指令对累加器中的数取反：

```
XRI  A, FFh
```

前面提到过，DAA 是 Decimal Adjust Accumulator 的缩写，即十进制调整累加器，它可能是 8080 中最复杂的一条指令。在 8080 微处理器中专门设计了一个完整的小部件用来执行该指令。

DAA 指令提供了一种用二进制码表示十进制数的方法，称为 BCD 码（binary-coded decimal），程序员可以在该指令的帮助下实现十进制数的算术运算。BCD 码采用的表示方式为，每 4 位为一段，每段所能表示数的范围是：0000～1001，对应十进制数的 0～9。因为 1 字节有 8 位故可分割为 2 个段，因此在 BCD 码格式下，一个字节可以表示两位十进制数。

假设累加器 A 存放的是 BCD 码表示的 27h，显然它就对应十进制数的 27（通常，十六进制的 27h 对应的十进制数是 39）。同时假设寄存器 B 中存放着 BCD 码表示的 94h。假如执行如下指令：

```
MVI     A, 27h
MVI     B, 94h
ADD     A, B
```

累加器中存放的最终结果是 BBh，当然，这肯定不是 BCD 码。因为 BCD 码中每 4 位组成的段所能表示的十进制数不会超过 9。然而，当我们执行指令：

```
DAA
```

那么累加器最后所保存的值是 21h，而且进位标志位 CF 置 1。因为十进制的 27 与 94 相加的结果为 121。由此可以看到，使用 BCD 码进行十进制的算术运算是很方便的。

在 8080 程序中，经常会对一个数进行加 1 或减 1 运算。在第 17 章的乘法程序中，为了实现对一个数减 1，我们把该数与 FFh 相加，它是-1 的补码。8080 提供了专门的指

令用来对寄存器或存储器中的数进行加 1（称作增量）或减 1（称作减量）操作。

操作码	指　令	操作码	指　令
04	INR　B	05	DCR　B
0C	INR　C	0D	DCR　C
14	INR　D	15	DCR　D
1C	INR　E	1D	DCR　E
24	INR　H	25	DCR　H
2C	INR　L	2D	DCR　L
34	INR　[HL]	35	DCR　[HL]
3C	INR　A	3D	DCR　A

INR 和 DCR 都是单字节指令，它们可以影响除 CF（Carry Flag）之外的所有标志位。

8080 还包括 4 个循环移位（Rotate）指令，这些指令可以把累加器中的内容向左或向右移动 1 位，它们的具体功能如下。

操作码	指　令	意　义
07	RLC	使累加器循环左移
0F	RRC	使累加器循环右移
17	RAL	带进位的累加器循环左移
1F	RAR	带进位的累加器循环右移

这些指令只对进位标志位 CF 有影响。

假设累加器中存放的数是 A7h，即二进制的 10100111。RLC 指令使其每一位都向左移一位。最终的结果是，最低位（左端为低位，右端为高位）移出顶端移至尾部成为最高位，这条指令也会影响 CF 的状态。在这个例子中 CF 置 1，最后的结果为 01001111。RRC 指令以同样的方式进行右移位操作。如果移位之前的数是 10100111，执行 RRC 之后将变为 11010011，同时 CF 置 1。

较之 RLC 和 RRC，RAL 和 RAR 指令的工作方式稍有不同。执行 RAL 指令时，累加器中的数仍然按位左移，把 CF 中原来的值移至累加器中数值的最后一位，同时把累加器中数据的原最高位移至 CF。例如，假设累加器中移位之前的数是 10100111 且 CF 为 0，执行 RAL 指令后，累加器中的数变为 01001110 而 CF 变为 1。类似的，如果执行的是 RAR 指令，累加器中的数变为 01010011 而 CF 变为 1。

当我们在程序中需要对某个数进行乘2（左移）或除2（右移）运算时，使用移位操作会使运算变得非常简单。

我们通常把微处理器可以寻址访问的存储器称为随机访问存储器（random access memory，RAM），主要的原因是：只要提供了存储器地址（有多种方式），微处理器可以用非常简便的方式访问存储器的任意存储单元。RAM就像一本书，我们可以翻到它的任意一页。这种方式很方便，它不像存储在一个微缩胶片上的整个星期的报纸，为了阅读星期六的内容，我们需要扫描几乎整个星期的内容。同样，它也比读取磁带快得多，因为当我们要听最后一首歌时，需要快进磁带的一整面。微缩胶片和磁带都不是随机访问的，它们是顺序访问（sequential access）的。

显然，随机访问存储器是非常好的一种寻址方式，对于经常访问存储器的微处理器来说更是如此。然而，在某些情况下使用不同的寻址方式访问存储器也是有好处的。例如，下面例子中的这种存储方式既不是随机的也不是顺序的：假设你在办公室工作，有人会到你办公桌前为你分配任务，每一项工作都用到某种文件夹。这些工作通常有这样的特点，在你完成某项工作之前首先要做另一项相关工作，并用到另一个文件夹。因此你只能放下第一个文件夹，并在它上面打开一个第二个文件夹继续工作。现在又有一个人给你分配了一个比前一项优先级更高的工作，于是你打开第三个文件夹放在前面两个上，继续工作。而这项工作也需要先做一项相关工作，于是你只好再打开第四个文件夹，现在你的办公桌上已经堆叠了四个文件夹了。

你可能已经注意到了，事实上，这些堆叠的文件夹很有序地保存了你干活的顺序轨迹。最上面的文件夹总是代表优先级最高的工作，完成该工作之后就可以做接下来的工作了，依此类推。最后当你处理完办公桌上最后一个文件夹（即接受的第一个任务）后，就可以回家了。

这种形式的存储器称作堆栈（stack）。使用堆栈时，我们以从底部到顶部的顺序把数据存入堆栈，并以相反的顺序把数据从堆栈中取出，因此该技术也称作后进先出存储器（last-in-first-out，LIFO）。堆栈的特点是，最先保存到堆栈中的数据最后被取出，而最后保存的数据则被最先取出。

同样，在计算机中也可以使用堆栈，当然计算机中的堆栈保存的是数据而不是工作。大量的实践证明，在计算机中使用堆栈技术是十分方便的。通常把将数据存入堆栈的过

程称作压入（push），把从堆栈取出数据的过程称作弹出（pop）。

假设你正在编写一个汇编语言程序，需要用到寄存器 A、B、C 来存储数据。在编写程序的过程中，你注意到程序需要做一个小的计算，并且该计算也需要用到寄存器 A、B、C。你希望在完成该计算之后仍然回到原来的地方，并且仍然使用寄存器 A、B、C 中原先存放的数据。

为了保存寄存器 A、B、C 原先存放的数据，可以简单地把这些数据保存到存储器中不同的地址中，需要进行的计算完成之后，再把这些数据从存储器转移到寄存器。但这种方式需要记录数据存放的地址。有了堆栈的概念之后，我们可以用一种更清晰的方式来处理这个问题，即把这些寄存器中的数据依次存放到堆栈中。

```
PUSH    A
PUSH    B
PUSH    C
```

稍后将具体解释这些指令实际的意义。现在需要知道的是，这些指令以某种方式把寄存器中的内容保存到先进后出存储器。这些指令执行之后，寄存器中原有的数据将妥善地保存下来，你就可以放心地使用这些寄存器进行别的工作了。为了取回原来的数据，可以使用 POP 指令把它们从堆栈中弹出，当然弹出的顺序和原来压入的顺序是相反的。相应的 POP 指令如下所示。

```
POP     C
POP     B
POP     A
```

再次谨记：后进先出。如果 POP 指令的顺序弄错了，将会引起严重的错误。

我们可以在程序中多次用到堆栈而不会引起混乱，这是堆栈机制的一个特殊优势。例如，在我们要编写的程序中已经把寄存器 A、B、C 中的数保存到了堆栈，在程序的另一段又需要把寄存器 C、D、E 中的数保存到堆栈，可以使用下面的指令：

```
PUSH    C
PUSH    D
PUSH    E
```

当然，在该段程序中还需要使用一些指令将保存到堆栈中的数据取回至寄存器，这些指令是：

```
POP     E
POP     D
POP     C
```

显然，由于先进后出的原则，这些数据在先前存放的 C、B、A 中的数据之前弹出堆栈。

堆栈的功能是怎样实现的呢？首先，堆栈其实就是一段普通的 RAM 存储空间，只是这段空间相对独立不另作他用。8080 微处理器设置了一个专门的 16 位寄存器对这段存储空间寻址，这个特殊的寄存器称为堆栈指针（SP，Stack Pointer）。

在我们所举的例子中，对于 8080 来说使用 PUSH 和 POP 对寄存器操作实际上是不准确的。在 8080 中，执行 PUSH 指令实际上是把 16 位的数据保存到堆栈，执行 POP 指令是把这些数据从堆栈中取回至寄存器。因此，对于上面的如 PUSH C，POP C 等指令，我们对其进行如下的修改。

操作码	指　令	操作码	指　令
C5	PUSH　BC	C1	POP　BC
D5	PUSH　DE	D1	POP　DE
E5	PUSH　HL	E1	POP　HL
F5	PUSH　PSW	F1	POP　PSW

PUSH　BC 指令将寄存器 B 和 C 中的数据保存到堆栈，而 POP　BC 则将这些数据从堆栈取回到寄存器 B 和 C 中，并且保持原来的顺序。最后一行指令中的 PSW 代表程序状态字，如前所述，这是一个 8 位的寄存器，用于保存标志位。最后一行的 PUSH 和 POP 指令的操作对象实际上是累加器和 PSW，即压入和弹出堆栈的数据由累加器和 PSW 中的内容组成。如果你想把所有寄存器中的数据及全部标志位都保存到堆栈，可以使用下面的指令：

```
PUSH    PSW
PUSH    BC
PUSH    DE
PUSH    HL
```

堆栈是怎样工作的呢？我们假设堆栈指针是 8000h，当执行 PUSH BC 指令时将会引发以下操作。

- 堆栈指针减 1，变为 7FFFh。

- 寄存器 B 中的内容被保存到堆栈指针指向的地址，即存储器地址 7FFFh 处。
- 堆栈指针减 1，变为 7FFEh。
- 寄存器 C 中的内容被保存到堆栈指针指向的地址，即存储器地址 7FFEh 处。

类似的，在堆栈指针仍为 7FFEh 的情况下，执行 POP BC 指令时会将上面的步骤反过来执行一遍：

- 堆栈指针指向的地址（7FFEh）的内容加载到累加器 C。
- 堆栈指针加 1，变为 7FFFh。
- 堆栈指针指向的地址（7FFFh）的内容加载到累加器 B。
- 堆栈指针加 1，变为 8000h。

每执行一条 PUSH 指令，堆栈都会增加两个字节，这可能会导致程序出现一些小错误——堆栈可能会不断增大，最终覆盖掉存储器中保存的程序所必需的代码或数据。这种错误被称作堆栈上溢（stack overflow）。类似的，如果在程序中过多地使用了 POP 指令，则会过早地取完堆栈中的数据从而导致类似的错误，这种情况称为堆栈下溢（stack underflow）。

如果 8080 连接的是一个 64 KB 的存储器，你可能会把堆栈指针初始化为 0000h。当执行第一条 PUSH 指令时，堆栈指针会减 1 变为 FFFFh，即存储器的最后一个的存储单元。这时，堆栈的初始位置将会是存储器的最高地址，因为程序的代码通常从 0000h 开始存放，因此两者将保持非常远的距离。

8080 使用 LXI 指令为堆栈寄存器赋值，LXI 是 Load Extended Immediate 的缩写，即加载扩展的立即数。下面的这些指令将把操作码后的两个字节保存到 16 位寄存器对中。

操作码	指　令
01	LXI　BC, xxxx
11	LXI　DE, xxxx
21	LXI　HL, xxxx
31	LXI　SP, xxxx

指令：

```
LXI  BC, 527Ah
```

和下面两条指令等价：

```
MVI  B, 52h
MVI  C, 7Ah
```

LXI 指令保存一个字节。而且上表中最后一条 LXI 指令为堆栈指针赋了一个特殊的值。通常下面的指令不作为微处理器复位之后首先执行的指令之一。

```
0000h:      LXI  SP, 0000h
```

类似的，还可以对寄存器对和堆栈指针进行加 1 或减 1 操作，即把它们看做 16 位寄存器。

操作码	指　令	操作码	指　令
03	INX　BC	0B	DCX　BC
13	INX　DE	1B	DCX　DE
23	INX　HL	2B	DCX　HL
33	INX　SP	3B	DCX　SP

对于要讨论的 16 位指令，我们可以再看一些例子。下面的指令把由任意 2 个寄存器组成的 16 位寄存器对的内容加到寄存器对 HL 中。

操作码	指　令
09	DAD　HL, BC
19	DAD　HL, DE
29	DAD　HL, HL
39	DAD　HL, SP

这些指令可以减少操作的字节数。例如，上面的第一条指令一般情况下需要 6 个字节。

```
MOV  A, L
ADD  A, C
MOV  L, A
MOV  A, H
ADC  A, B
MOV  H, A
```

DAD 指令一般用来计算存储器地址，只对进位标志位 CF 有影响。

接下来我们来认识一下各种各样的指令。下面两条指令的特点是操作码后面都跟着2字节的地址，第一条指令把HL寄存器对的内容保存到该地址，第二条指令把该地址的内容加载到HL寄存器对。

操作码	指　令	意　义
22h	SHLD　[aaaa], HL	直接保存HL中的数据
2Ah	LHLD　HL, [aaaa]	直接加载数据到HL

寄存器L的数据保存在地址aaaa，而寄存器H的数据保存在地址aaaa+1。

下面的两条指令把寄存器对HL保存的数据加载到程序计数器和堆栈指针。

操作码	指　令	意　义
E9h	PCHL　PC, HL	将HL保存的数据加载到程序计数器
F9h	SPHL　SP, HL	将HL保存的数据加载到堆栈指针

PCHL指令本质上是一种Jump指令，它把HL保存的存储器地址加载到程序计数器，而8080处理器要执行的下一条指令就是程序计数器所指明的存储器地址中存放的指令。SPHL指令可以作为另一种为堆栈指针赋值的指令。

下面的两条指令中，第一条将HL保存的数据与堆栈顶部的两个字节进行交换；第二条指令将HL保存的数据和寄存器对DE保存的数据进行交换。

操作码	指　令	意　义
E3h	XTHL　HL, [SP]	把HL中的内容和堆栈顶部2个字节进行交换
EBh	XCHG　HL, DE	把DE中的内容和HL中的内容进行交换

除了刚讲过的PCHL指令外，目前为止还没有介绍过8080的跳转指令。如第17章所述，在处理器中专门设置了一个称为程序计数器PC的寄存器，它用来保存处理器将要取出并执行的指令的存储地址。通常，处理器在PC的指引下顺序执行存储器中存放的指令，但有一些指令，如Jump（跳转）、Branch（分支）或Goto（无条件转移）——使处理器脱离原来的执行顺序。这些指令使PC重新加载另外的值，处理器要执行的下一条指令存放于存储器的其他位置，而不在按原来的顺序寻址。

当然，原始的普通跳转指令的确有一定的作用，但从第17章得来的经验可以知道，条件跳转（conditional jump）指令的作用更大。条件跳转指令使处理器根据某些标志位的

值转移到特定的地址，这些标志位可以是进位标志位 CF、零标志位 ZF 等。正是由于条件跳转指令的引入，第 17 章所设计的自动加法器才成为一般意义上的数字计算机。

8080 有 5 个标志位，其中有 4 个可用于条件跳转指令。8080 支持 9 种不同的跳转指令，包括了非条件跳转指令，还包含根据 ZF（Zero Flag）、CF（Carry Flag）、PF（Parity Flag）以及 SF（Sign Flag）是否为 1 而跳转的条件跳转指令。

在介绍这些指令之前，首先来介绍与 Jump 指令相关的另外两种指令。第一种是 Call（调用）指令，它和 Jump 指令类似，但是有所不同的是：执行 Call 指令后，程序计数器（Program Counter，在这部分讲解中简称 PC）加载一个新的地址，而处理器会把原来的地址保存起来，保存到何处呢？最好的选择自然是堆栈了。

这种策略使 Call 指令有效地记录了“从何处跳转”（where it jumped from），即保存了跳转之前的相关信息。堆栈中保存的地址可以使处理器最后返回到转移之前的位置。用于返回的指令称为 Return（返回）。Return 指令从堆栈中弹出两个字节，并把它们加载到 PC 中，这样就完成了返回到跳转点的工作。

对于任何处理器来说，Call 和 Return 指令都非常重要。在它们的帮助下，程序员可以在程序中使用子程序（subroutine），子程序是一段频繁使用的完成特定功能的代码（这里的“频繁”意味着“不止一次”）。对于汇编语言来说，子程序是其基本的组成部分。

让我们来看一个使用子程序的例子。假设你在编写一个汇编语言程序，在程序的某个位置你需要把两个字节相乘，因此你编写了一段用于两个数相乘的代码，然后继续向下写，在程序的另一个位置你发现需要再一次对两个字节相乘。因为你已经写过把两个字节相乘的代码，所以只需要重复使用这些代码就可以了。但是怎么做呢？只是简单地把这些代码重复输入到存储器吗？我们希望不是，因为这样做不仅耽误时间而且浪费存储空间，一个更好的方法是跳转到先前写的那段乘法代码所在的位置。但是普通的 Jump 指令不能完成这个操作，因为在执行乘法之后不能准确地返回程序的当前位置。因此你需要使用 Call 指令和 Return 指令来帮助你实现这个功能。

用来实现两个数相乘的一组指令可以作为一个子程序。下面我们将看到这个子程序。在第 17 章的乘法程序中，被乘数（还有乘积）被保存在存储器的特定位置；而在 8080 的子程序中，乘数和被乘数分别存放在寄存器 B 和寄存器 C 中，乘积保存到 16 位寄存器