第19章　两种典型的微处理器

微处理器 —— 集成计算机中央处理器（CPU）的所有组件在一个硅芯片上 —— 诞生于1971年。它的产生有一个很好的开端：第一个微处理器是 Intel 4004，其中有2300个晶体管。今天，差不多30年过去了，为家用计算机所制造的微处理器中将近有 10 000 000个晶体管。

微处理器实际的作用基本上保持不变。现在的芯片上附加的上百万个晶体管可以做许多有趣的事情，但在微处理器最初的探索过程中，这些事情更多的是分散我们的注意力而不是给我们以启迪。为了对微处理器的工作情况获得更清晰的认识，让我们先看一下最初的微处理器。

这些微处理器出现在1974年。在该年度，Intel公司在4月推出了8080，Motorola（摩托罗拉） —— 从20世纪50年代开始生产半导体和晶体管产品的公司 —— 在8月份推出了6800。它们并非是那年仅有的微处理器。同样是在1974年，德克萨斯仪器公司推出了4位的TMS 1000，用在许多计算器、玩具和设备上；National Semiconductor（国家半导体公司）推出了 PACE，它是第一个16位的微处理器。然而，回想起来，8080和6800是两个最具有重大历史意义的芯片。

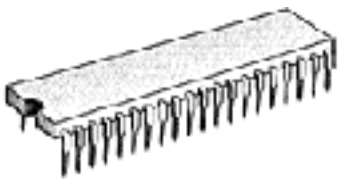
Intel设定8080最初价格为$ 360，这是对IBM System/360的一个讽刺。IBM System/360是一个大型机系统，由许多大公司使用，要花费几百万美元。（今天，你只花$ 1.95就可以买到一个8080芯片。）这并不是说8080可以与System/360相提并论，但不用几年，IBM公司也将会正视这些很小的计算机。

8080是一个8位的微处理器，有6000个晶体管，时钟频率为2MHz，可寻址64KB的存储空间。6800（今天也只卖$ 1.95）有大约4000个晶体管，也可寻址64KB的存储空间。第1代6800的时钟频率为1 MHz，但到1977年Motorola公司发布新款的6800时，其时钟频率已为1.5MHz和2 MHz。

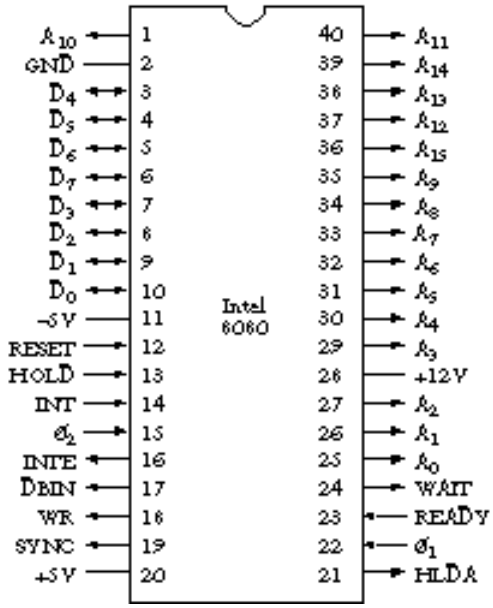
这些芯片称作“单芯片微处理器”，不太准确的名称为“一个芯片上的计算机”。处理器只是整个计算机的一部分。此外，计算机至少还需要一些随机访问存储器（ RAM）、供人们输入信息到计算机的方法（输入设备），供人们从计算机获取信息的方法（输出设备），以及其他可把所有这些东西连接在一起的芯片。这些组件将在第 21章详细介绍。

从现在起，我们只考察微处理器。描述微处理器时，通常是用框图来画微处理器的内部组件及它们是如何连接的。但在第 17章已有够多的图了，现在，我们将通过观察处理器与外界的相互作用来了解它的内部 。换句话说，为了弄清楚它的作用，可以把微处理器看成是一个黑盒子，它的内部操作不需要做详细研究。我们可以通过测试芯片的输入和输出信号，特别是芯片的指令集来掌握微处理器的功能。

8080和6800都是40管脚的集成电路。这些芯片最普通的IC封装大约是2英寸长，半英寸宽，1/8英寸厚：



当然，你看到的只是外包装。位于其内部的硅晶片非常小，就拿早期的8位微处理器来说，其硅晶片小于1/4平方英寸。外包装保护硅晶片并通过管脚提供对芯片的输入和输出点的访问。下图显示了8080的40个管脚的功能：



本书的所有电气或电子设备都需要某种电源供电。 8080的一个特别之处在于它需要三种电源电压：管脚20必须连到5伏电源上，管脚11连到-5伏电源上，管脚28连到12伏电源上；管脚2接地（1976年，Intel发布了8085芯片，简化了这些电源需求）。

其余管脚都画有箭头。从芯片中出来的箭头表示输出信号，这是由微处理器控制的信号，计算机中其余芯片对此作出响应。指向芯片的箭头表示输入信号，这是来自于其他芯片的信号，8080对它们做出响应。有些管脚既是输入又是输出。

第17章的处理器需要振荡器使它工作。 8080需要两个不同的2 MHz同步时钟输入，在22和15管脚上分别标记为 ϕ_1和 ϕ_2。这些信号可以很方便地由Intel公司生产的8224时钟信号发生器提供。给这个芯片连上一个18 MHz的石英晶体，剩下的工作它基本上可以完成。

一个微处理器通常有多个输出信号来寻址存储空间，这种信号的数目与微处理器可寻址的存储器空间的大小直接相关。 8080有16个地址信号，标为A0～A15，具有寻址2^{16}即65 536字节的存储空间的能力。

8080是一个8位微处理器，一次可从存储器中读出、写入 8位数据。它包括8个数据信号D_0~D_7，这些信号是在此芯片中仅有的几个既作为输入又作为输出的信号。当微处理器从存储器读数据时，这些管脚作为输入；当微处理器向存储器写数据时，这些管脚作为输出。

微处理器的另外10个管脚是控制信号。例如， RESET输入用来复位微处理器。输出信号$\overline{WR}$表示微处理器要向RAM中写数据。（$\overline{WR}$信号对应于RAM阵列的写入输入。）另外，当芯片读取指令时，其他控制信号会在某个时候出现在 D_0~D_7管脚。由8080构成的计算机系统通常使用8228系统控制芯片来锁存这些附加的控制信号。后面将会讲述一些控制信号。由于 8080的控制信号非常复杂，因此，除非你想基于 8080芯片来实际设计计算机，否则最好不要用这些控制信号来折磨自己。

假定8080微处理器连接了64KB的存储器，这样可以不通过微处理器来读写数据。

8080芯片复位后，它从存储器的地址0000h处读取该字节，送到微处理器中。这可以通过在地址信号端A_0～A_{15}输出16个0来实现。它读取的字节必须是8080指令，这种读取字节的过程叫作取指令。

在第17章构造的计算机里，所有指令（除了停止指令HLT）都是3个字节长，包括一个操作码和两个字节的地址。在8080中，指令长度可以是1个字节、2个字节或3个字节。有些指令可使8080从存储器的某一位置处读出一个字节送到微处理器中；有些指令可使8080从微处理器中把数据写入存储器的某一位置处；其他指令可使8080不使用RAM而在内部执行。第一条指令执行完后，8080访问存储器中的第二条指令，依此类推。这些指令组合在一起构成一个计算机程序，用来完成一些自己感兴趣的事情。

当8080运行在最高速度即2 MHz时，每个时钟周期为500纳秒（1除以2 000 000周等于0.000000500秒）。第17章中的每条指令都需要4个时钟周期，8080的每条指令则需要4～18个时钟周期，这意味着每条指令的执行时间为2～9微秒（即百万分之一秒）。

了解微处理器功能的最好方法可能是在系统方式下测试其完整的指令集。

第17章最后出现的计算机仅有12条指令。一个8位微处理器很容易就有256条指令，每个操作码对应于某个8位值。（如果一些指令有2个字节的操作码，则实际会有更多的指令）。8080虽没有那么多，但它也有244条操作码。这看起来似乎很多，但总的来说，却又不比第17章中的计算机功能多多少。例如，如果想用8080做乘法或除法，仍然需要写一段小程序来实现。

第17章中讲过，处理器指令集的每个操作码都和某个助记符相联系，有些助记符之后可能还有操作数。但这些助记符仅用来方便地表示操作码。处理器只读取字节，它并不懂组成这些助记符的字符的含义。

第17章中的计算机有两条很重要的指令，称作装载（Load）和保存（Store）指令。这些指令都占用三个字节的存储空间。装载指令的第一个字节是操作码，操作码后的两个字节表示16位地址。处理器把在此地址中的字节送到累加器。同样，保存指令把累加器中的内容存储到指令指定的地址处。

下面，我们用助记符来简写这两个操作：

```
LOD   A ，[aaaa]
STO   [aaaa]，A
```

在此，A表示累加器（装载指令的目的操作数，保存指令的源操作数），aaaa表示一个16位的存储器地址，通常用4位十六进制数来表示。

8080中的8位累加器称作A，就像第17章中的累加器。正如第17章中的计算机一样，8080也有两条与装载和保存指令功能一样的指令。8080中这两条指令的操作码为32h和3Ah，每个操作码后有一个16位地址。8080的助记符为STA（代表存储累加器的内容）和LDA（代表装载到累加器）：

操作码	指令
32	STA [aaaa],A
3A	LDA A,[aaaa]

除了累加器，8080微处理器内部还包括6个寄存器（register），每个寄存器可以保存8位的值。这些寄存器和累加器非常相似，事实上，累加器被看作是一种特殊的寄存器。和累加器

一样，这6个寄存器也是锁存器。处理器可以把数据从存储器传送到寄存器，也可以把数据从寄存器送回到存储器。然而，这些寄存器没有累加器的功能强大。例如，当两数相加时，其结果通常送往累加器而非其中一个寄存器。

在8080中，除累加器外的6个寄存器的名字分别为B，C，D，E，H和L。人们通常问的第一个问题是“用F和G干什么？”，第二个问题是“用I，J和K又要做什么？”，答案是使用寄存器H和L具有某种特殊的含义。H代表高（High），L代表低(Low)。通常把H和L的8位合起来记作HL来表示一个16位寄存器对，H作为高位字节，L作为低位字节。这个16位值通常用来寻址存储器。后面我们将看到它的简单用法。

所有这些寄存器都是必需的吗？为什么不在第17章中的计算机中用到它们呢？从理论上说，它们并非必需，但是使用它们会很方便。许多计算机程序在同一时刻要用到几个数据，如果所有这些数据都存储在微处理器的寄存器中而非存储器中，执行程序将会更快，因为程序访问存储器的次数越少，那么它的运行速度也就越快。

8088指令中，有一个至少63个指令码供一条8080指令使用的指令，它就是MOV指令，即Move的简写。该条指令只有一个字节，用于把一个寄存器中的内容传送到另一个寄存器中（或同一个寄存器中）。使用大量MOV指令是设计带有7个寄存器（包括累加器）的微处理器的正常结果。

下面是前32条MOV指令。记住目的操作数在左边，源操作数在右边：

操作码	指令	操作码	指令
40	MOV B，B	50	MOV D，B
41	MOV B，C	51	MOV D，C
42	MOV B，D	52	MOV D，D
43	MOV B，E	53	MOV D，E
44	MOV B，H	54	MOV D，H
45	MOV B，L	55	MOV D，L
46	MOV B，[HL]	56	MOV D，[HL]
47	MOV B，A	57	MOV D，A
48	MOV C，B	58	MOV E，B
49	MOV C，C	59	MOV E，C
4A	MOV C，D	5A	MOV E，D
4B	MOV C，E	5B	MOV E，E
4C	MOV C，H	5C	MOV E，H
4D	MOV C，L	5D	MOV E，L
4E	MOV C，[HL]	5E	MOV E，[HL]
4F	MOV C，A	5F	MOV E，A

这些都是很方便的指令。当一个寄存器中有值时，可以把它传送到其他寄存器中。注意，上述指令中有四条指令用到HL寄存器对，如：

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

前面列出的LDA指令把一个字节从存储器中传送到累加器中，这个字节的16位地址直接跟在LDA操作码的后面。这里的MOV指令把一个字节从存储器中传送到寄存器B中，但被装载到寄存器中的字节的地址是保存在寄存器对HL中。HL寄存器是怎样得到16位存储器地址的呢？它可以通过多种方法来实现，或许是通过某种方法计算出来的。

总之，这两条指令

```
LDA    A，[aaaa]
MOV    B，[HL]
```

都把一个字节从存储器中装载到微处理器中，但它们用两种不同的方法来寻址存储器地址。第一种方法叫作直接寻址方式，第二种方法叫作间接寻址方式。

第二批32条MOV指令表明用HL寻址的存储器地址也可以作为目的操作数：

操作码	指令	操作码	指令
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

其中一些指令如：

```
MOV  A,  A
```

做的是无用的事，而像：

```
MOV   [HL]，[HL]
```

这样的指令是不存在的。和这条指令相对应的操作码实际上是停止指令。

观察这些MOV操作码更明显的方法是考察它的位模式， MOV操作码由8位组成：

```
01dddsss
```

其中字母ddd代表指代目的操作数的3位代码，sss代表指代源操作数的3位代码。这3位代码是：

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= HL中保存的存储器地址中的内容
111= 累加器A

例如，指令：

```
MOV  L ,  E
```

相应的操作码表示为01101011，或6Bh。可以通过检查前面的表来验证。

因此可能在8080内部某个地方，标有sss的3位标识用在8-1数据选择器中，标有ddd的3位标识用于控制3-8译码器，此译码器用来决定哪一个寄存器锁存了一个值。

也可能使用寄存器B和C来构成一个16位寄存器对BC，用寄存器D和E来构成一个16位寄存器对DE。如果每一个寄存器对都包含用于装载或保存一个字节的存储器地址，则可以使用下述指令：

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]

另一种类型的传送指令叫做传送立即数，指定的助记符为 MVI。传送立即数指令占两个字节，第一个是操作码，第二个是1个字节的数据。此字节从存储器中传送到一个寄存器中或由HL寻址的存储单元中。

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中包含有字节37h。这就是第三种寻址方式，即立即数寻址方式。

32个操作码的集合完成四种基本算术运算，那是在第17章开发处理器时我们就已熟悉的运算，即加法（ADD）、进位加法（ADC）、减法（SUB）和借位减法（SBB）。所有情况中，累加器是两个操作数之一，也是结果的目的地址。

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

假设A中是35h,寄存器B中是22h,当指令：

```
SUB     A，B
```

执行后，累加器中的结果为13h。

若A中的值为35h,寄存器H中的值为10h，L中的值为7Ch，存储器地址107Ch中的值为4Ah，则指令：

```
ADD A，[HL]
```

把累加器中的内容（35h）和通过寄存器对HL寻址得到的值（4Ah）相加，并把结果（7Fh）保存到累加器中。

ADC和SBB指令允许8080加/减16位、24位、32位和更多位的数。例如，假设寄存器对BC和DE都包含16位数，你想把它们相加，并把结果存到BC中。下面是具体做法：

```
MOV   A，C     ；低位字节
ADD   A，E
MOV   C，A
MOV   A，B     ；高位字节
ADC   A，D
MOV   B，A
```

其中有两条加法指令，ADD指令用于低位字节相加，ADC指令用于高位字节相加。第一条加法指令的进位位包含在第二条加法指令中。因为只能利用累加器进行加法运算，所以在这么短的代码中也需要至少4条MOV指令。许多MOV指令常常出现在8080代码中。

该是谈论8080标志位的时候了。在第17章的处理器中，已有进位标志位CF和零标志位ZF。8080还有3个标志位，即符号标志位SF、奇偶标志位PF和辅助进位标志位AF。所有标志位都保存在另一个叫作程序状态字（PSW：program status word）的8位寄存器中。像LDA、STA和MOV这样的指令不影响标志位，而ADD、SUB、ADC和SBB指令却要影响标志位，影响的方式如下：

- 当运算结果最高位为1时，符号标志位SF为1，表示结果为负。
- 当结果为0时，零标志位ZF为1。
- 当运算结果中“1”的个数为偶数时，奇偶标志位PF=1；当运算结果中“1”的个数为奇数时，奇偶标志位PF=0。PF有时用来粗略地检测错误，此标志位在8080程序中不常用。
- 当ADD或ADC运算产生进位或SUB与SBB运算不发生借位时，进位标志位CF=1。（这点不同于第17章中的计算机进位标志的实现。）
- 当操作结果的低4位向高4位有进位时，辅助进位标志位AF=1。此标志位只用在DAA（十进制调整累加器）指令中。

有两条指令直接影响进位标志位CF：

操作码	指令	含义
37	STC	置CF为1
3F	CMC	CF取反

第17章中的计算机可执行ADD、ADC、SUB和SBB指令（尽管没什么灵活性），但8080还可以进行逻辑运算AND（与）、OR（或）和XOR（异或）。算术运算和逻辑运算都可通过处理器的算术逻辑单元（ALU）来执行：

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指令按位运算，即逻辑操作只是单独地在对应位之间进行。例如：

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

累加器中的结果将为05h。如果第三条指令为OR运算，则结果为5Fh；如果第三条指令为XOR运算，则结果为5Ah。

CMP（比较）指令与SUB指令基本上一样，除了结果不保存在累加器中。换句话说，CMP执行减法操作再把结果扔掉。这是为什么？是因为标志位。根据标志位的状态可知道所比较的两数之间的关系。例如，当如下指令：

```
MVI   B，25h
CMP   A，   B
```

执行完时，A中的内容没有改变。然而，若A中的值为25h，则ZF标志置位；若A中的值小于25h，则CF = 1。

这8个算术逻辑运算也可以对立即数进行操作：

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，上面列出的两条指令也可以用下面的指令来替换：

```
CPI     A，25h
```

下面是其他两条8080指令：

操作码	指令
27	DAA
2F	CMA

CMA即complement accumulator，它对累加器中的值进行取反操作。每个0变为1，每个1

变为0。如果累加器中的值为01100101，CMA指令使它变为10011010。也可以用下述指令来使累加器按位取反：

```
XRI    A，FFh，
```

DAA即Decimal Adjust Accumulator，如前所述，它可能是8080中最复杂的一条指令。微处理器中有一个完整的小部件专门用于执行这条指令。

DAA指令帮助程序员用BCD码表示的数来进行十进制算术运算。在BCD码中，每一小块数据的范围在0000～1001之间，对应于十进制的0～9。利用BCD码格式，每8位字节可存储两个十进制数字。

假设累加器中的值为BCD码的27h。由于是BCD码，则它实际上指的是十进制的27。（十六进制的27h等于十进制的39。）再假定寄存器B中的值为BCD码的94h。如果执行指令：

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

累加器中的值将为BB h，当然，它不是BCD码，因为BCD码中的每一块不能超过9。但是，现在执行指令：

```
DAA
```

则累加器中的值为21h，且CF = 1，这是因为27和94的十进制和为121。如果想进行BCD码的算术运算，这样做是相当方便的。

经常需要对一个数进行加1或减1操作。在第17章的乘法程序中，我们需要对一个数减1，使用的方法是加上FFh，它是-1的2的补码。8080中包含特殊的用于寄存器或存储单元的加1指令（称作增量）和减1指令（称作减量）：

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

单字节指令INR和DCR可影响除CF外的所有标志位。

8080也包含4个循环移位指令，这些指令可使累加器中的内容左移或右移1位：

操作码	指令	含义
07	RLC	累加器循环左移
0F	RRC	累加器循环右移
17	RAL	累加器带进位循环左移
1F	RAR	累加器带进位循环右移

这些指令只影响CF。

假定累加器中的值为A7h，即二进制的10100111。RLC指令使A中的内容向左移位，最高位（移出顶端）成为最低位（移进底端），同时决定进位标志位CF的状态。其结果为01001111且CF = 1。RRC指令用同样的方法向右移位。开始为10100111，执行RRC指令后，其结果为

11010011且CF = 1。

RAL和RAR指令有些不同。当向左移位时，RAL指令把CF移入累加器的最低位，而把最高位移入CF中。例如，如果累加器的内容为10100111，CF = 0，RAL指令执行的结果是累加器的内容变为01001110，且CF = 1。同样，在相同的初始条件下，RAR指令使累加器的内容变为01010011，CF = 1。

对于乘2（左移1位）和除2（右移一位）操作，移位指令非常方便。

把微处理器寻址的存储器叫作随机访问存储器（RAM）是有原因的：微处理器可以简单地根据提供的地址访问某一存储位置。RAM就像一本书一样，我们可以打开它的任何一页。它并不像做在微缩胶片上的一个星期的报纸，要找到周六版，需扫过大半周。同样，它也不同于磁带，要播放磁带上的最后一首歌需快进整个一面。微缩胶片和磁带的存储不是随机访问的，而是顺序访问的。

RAM确实效果不错，对于微处理器来说更是如此。但在使用存储器时有所差别是有好处的，下面就是一种既非随机又非顺序访问的存储方式：假定你在一个办公室里，人们到你桌前给你分配工作，每个工作都需要某种文件夹。通常你会发现你在继续某项工作之前，必须使用另外一个文件夹先做一些相关的工作。因此你把第一个文件夹放在桌子上，又拿出第二个文件夹放在它上面进行工作。现在又有一个人来让你做一个优先权高于前面工作的工作，你拿来一个新文件夹放在那两个上面继续工作。而此项工作又需要另外一个文件夹，这样在你的桌子上很快就摆了一堆文件夹了。

注意，这个堆非常明确地、有序地保存了你正在做的工作的轨迹。最上面的文件夹总是最高优先权的工作，去掉这个以后，下一个肯定是你就要做的，如此类推。当你最终去掉了桌子上的最后一个文件夹后（你开始的第1项工作），你就可以回家了。

以这种方式工作的存储器技术叫做作堆栈（stack）。从底向上压入堆栈，从顶向下弹出堆栈，因此这也叫后进先出存储器，或LIFO。最后放入堆栈中的数据最先被取出，最先放入堆栈中的数据最后被取出。

计算机中也可以使用堆栈，不是用来保存工作而是用来存储数据，且已被证明使用起来非常方便。向堆栈中放入数据叫作push（压入），从堆栈中取走数据叫作pop（弹出）。

假定你正在用汇编语言设计程序，程序中使用了寄存器A、B和C。但在编程过程中，你发现此程序需要去做另一件事——一个小的计算，其中也要使用寄存器A，、B、C。而你最终要回到先前的程序，并使用A、B、C中原有的值。

当然，你能做的工作只是简单地把寄存器A、B、C中的值保存到存储器中的不同位置，以后再把这些位置的值装载到寄存器中，但这样做需要保存值被保存的位置。一个显然的方法是把寄存器压入堆栈：

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

一会儿再解释这些指令的作用。现在，我们只需要知道它们以某种方式把寄存器的内容保存在一个后进先出的存储器中。一旦这些语句执行了，你的程序就可以毫无顾虑地利用这些寄存器来做其他工作。为了得到原来的值，只需简单地按与压入堆栈相反的顺序把它们从堆栈中弹出即可，如下所示：

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

记住是后进先出。如果用错了POP语句的顺序，就会引起错误。

堆栈机制的一个好处在于一个程序的不同部分都可以使用堆栈而不会出现问题。例如，在把A、B、C压入堆栈中后，程序的其他部分还可能需要把寄存器C、D、E的内容压入堆栈：

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

接着，这一部分程序所要做的就是在第一部分弹出C、B和A之前，用下述方法恢复寄存器的值：

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

堆栈是怎样实现的呢？首先，堆栈只是不被别的东西使用的正常的RAM的一部分。8080微处理器包含一个特殊的16位寄存器来对这一部分存储器进行寻址，这个16位寄存器叫作堆栈指针。

这里举的压入和弹出寄存器的例子对于8080来说不太准确。8080的PUSH指令实际上是存储16位的值到堆栈，POP指令用来恢复它们。因此8080不用像PUSH C和POP C这样的指令，它有下述8条指令：

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指令把寄存器B和C的内容保存到堆栈中，POP BC指令恢复它们。最后一行的缩写PSW指的是程序状态字，前面讲过，它是包含有标志位的8位寄存器。最后一行的两条指令实际上是把累加器和PSW都压入和弹出堆栈。如果你想保存所有寄存器和标志位的内容，可以使用：

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

当以后想恢复这些寄存器的内容时，按相反的顺序使用POP指令：

```
POP       HL
POP       DE
POP       BC
POP       PSW
```

堆栈是怎样工作的呢？假设堆栈指针为8000h，PUSH BC指令将引起下面这些情况发生：

- 堆栈指针减1至7FFFH
- 寄存器B的内容保存在堆栈指针所指的地址处，即7FFFh处

- 堆栈指针减1至7FFEH
- 寄存器C的内容保存在堆栈指针所指的地址处，即7FFEh处

当堆栈指针仍然为7FFEh时，POP BC指令执行，用来反向执行每一步：

- 从堆栈指针所指的地址（即7FFEh）处装载数据到寄存器C中
- 堆栈指针增1至7FFFh
- 从堆栈指针所指的地址（即7FFFh）处装载数据到寄存器B中
- 堆栈指针增1至8000h

对每个PUSH指令，堆栈都会增加2个字节，这可能导致程序出现小毛病——堆栈可能会变得很大以致会覆盖掉程序所需的一些代码和数据。这就是堆栈上溢问题。同样，过多的POP指令会过早用光堆栈内容，这就是堆栈下溢问题。

如果8080同一个64KB的存储器连接，你可能想把初始堆栈指针置为0000h。第一条PUSH指令使地址减1变为FFFFh，这时堆栈占用了存储器的最高地址。如果你的程序放在从0000h处开始的存储器区域，则它和堆栈离的就太远了。

对堆栈寄存器进行赋值的指令是LXI，即load extended immediate（装载扩展的立即数）。下面这些操作码后的指令也是把两个字节装载到16位寄存器：

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
```

等价于

```
MVI  B，52
MVI  C，7A h
```

LXI指令保存一个字节。另外，上表中最后一条LXI指令用来对堆栈指针赋值。微处理器复位后，这条指令并不常用来作为首先执行的指令之一：

```
0000 h：  LXI   SP，0000 h
```

也可以对寄存器对和堆栈指针执行加1和减1操作，就好像它们是16位寄存器一样：

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

即然是在讨论16位指令，可以看看更多一些这样的指令。下面的指令是把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

上面这些指令可节约几个字节。例如，第一条指令正常需要 6个字节：

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

DAD指令通常用于计算存储器地址，这条指令只影响 CF。

下一步让我们看以下各种指令。下面的两个操作码后都紧跟着一个 2字节地址，分别保存和装载寄存器对HL的内容：

操作码	指令	含义
2h	SHLD [aaaa]，HL	直接保存HL
2Ah	LHLD HL，[aaaa]	直接装载HL

寄存器L的内容保存在地址aaaa处，寄存器H的内容保存在地址aaaa+1处。

下面两条指令用来从寄存器对 HL中装载程序计数器PC或堆栈指针 SP：

操作码	指令	含义
E9h	PCHL PC，HL	把 HL中的内容装载到PC
F9h	SPHL SP，HL	把HL中的内容装载到 SP

PCHL指令实际上是一种转移指令， 8080执行的下一条指令保存在 HL寄存器对中的地址所对应的存储单元中。 SPHL是另外一个设置SP的方法。

下面两条指令中，第一条指令使 HL的内容与堆栈中最上面的两个字节进行交换，第二条指令使HL的内容与寄存器对DE的内容进行交换：

操作码	指令	含义
E3h	XTHL HL，[SP]	HL与堆栈顶端的内容交换
EBh	XCHG HL，DE	DE 和HL交换

除了PCHL外，还没有讲过8080的转移指令。前面第17章中讲过，处理器中有一个叫作程序计数器PC的寄存器，PC中包含处理器取回并执行的指令的存储器地址。通常 PC使处理器顺序执行存储器中的指令，但有些指令 —— 通常命名为 Jump（转移）、Branch（分支）或 goto（跳转） —— 能使处理器偏离这个固定的过程。这些指令使得 PC装载另外的值，处理器所取的下一条指令将在存储器的其他位置。

尽管简单、普通的转移指令很有用，但条件转移指令更有用。这些指令可使处理器根据某些标志，如CF或ZF，来转移到另外的地址处。条件转移指令的存在使得第 17章中的自动加法机成为一般意义上的数字计算机。

8080有5个标志位，其中4个对条件转移指令有用处。 8080支持9种不同的转移指令，包括无条件转移指令和基于 ZF、CF、PF、SF是1还是0的条件转移指令。

在介绍这些指令之前，先介绍一下与此相关的另外两种指令。第一个是 Call（调用）指令。Call指令与Jump指令的不同之处在于：前者把一个新值装入到程序计数器 PC中，处理器保存PC中原来的地址，保存在哪里？当然，在堆栈中。

这种策略意味着Call指令可有效地保存“程序从哪里跳转”的标记。处理器最终可利用此

地址返回到原来的位置。这个返回指令叫 Return。Return指令从堆栈中弹出两个字节，并把该值装载到PC中。

Call和Return指令是任何处理器中都很重要的功能。它们允许程序员编写子程序，子程序是程序中经常用到的代码段。（“经常”一般意味着“不止一次”。）子程序是汇编语言中的基本组成部分。

让我们看一个例子。假设你正在编写一个汇编语言程序，并且需要使两个数相乘，因此你可以写出一段代码来做这项工作，然后继续往下编写程序，现在又需要使两个数相乘。因为你已知道如何进行两数相乘，因此你只需简单地重复使用同样的指令来完成它。但只是简单地两次把这些指令输入到存储器吗？希望不是，这是对时间和存储空间的浪费，更好的方法是转送到原来的代码处。由于无法返回到程序的当前位置，所以一般的 Jump指令不能用。但使用Call和Return指令可以让你完成所需的功能。

进行两数相乘的一组指令可以作为一个子程序。下面就是这样的子程序。在第 17章中，被乘数（和结果）存放在存储器的某一地址中；而在 8080子程序中，寄存器B的值和寄存器C中的值相乘，然后把16位乘积装入寄存器HL中：

```
Multiply:       PUSH  PSW       ;保存要改变的寄存器
                PUSH  BC
                SUB   H,H       ;设置HL（结果）为0000h
                SUB   L,L
                MOV   A,B       ;乘数送到A
                CPI   A,00h     ;如果为0，结束
                JZ    AllDone

                MVI   B,00h     ;BC的高字节置0

Multloop:       DAD   HL,BC     ;BC 加到HL
                DEC   A         ;乘数减1
                JNZ   Multloop  ;不为0，转移

AllDone:        POP   BC        ;恢复保存的寄存器
                POP   PSW
                RET             ;返回
```

注意，上述子程序的第 1行开始有一个标号 Multiply。当然，这个标号对应于子程序所在的存储器地址。子程序开始用了两个 PUSH指令，通常在子程序开始处应先保存（以后恢复）它需要使用的寄存器。

然后该子程序把H和L寄存器置为0。虽然可以使用MVI指令而不用SUB指令，但那需要使用4个字节的指令而不是2个字节的指令。子程序执行完后，寄存器对HL中保存有相乘的结果。

下一步该子程序把寄存器 B的内容（乘数）移入 A中，并且检查它是否为 0。如果它为0，乘法子程序到此结束，因为结果为 0。由于寄存器H和L已经为0，因而子程序可以只使用 JZ指令跳转到末端的两个POP指令处。

否则，子程序把寄存器 B置为0。现在，寄存器对BC中包含一个16位的被乘数，A中为乘数。DAD指令把BC（被乘数）加到HL（结果）中。A中的乘数减1，且只要它不为0，JNZ指令就又使BC加到HL中。此小循环继续下去，直到 BC加到HL中的次数等于乘数。（可以用8080的移位指令编写一个更有效的乘法子程序。）

利用这个子程序完成25h与12h相乘的程序用下面的代码：

```
MOV  B,    25h
MOV  C,    12h
CALL  Multiply
```

Call指令把PC的值保存在堆栈中，该值是Call指令的下一条指令的地址。然后，Call指令使程序转移到标号Multiply所标识的指令，即子程序的开始。当子程序计算完结果后，执行RET（返回）指令，即从堆栈中弹出程序计数器的值，程序继续执行Call指令后面的语句。

8080指令集中包括条件CALL指令和条件Return指令，但它们远不如条件转移指令用得多。下表中完整地列出了这些指令：

条件	操作码	指令	操作码	指令	操作码	指令
None	C9	RET	C3	JMP aaaa	CD	CALL aaaa
Z not set	C0	RNZ	C2	JNZ aaaa	C4	CNZ aaaa
Z set	C8	RZ	CA	JZ aaaa	CC	CZ aaaa
C not set	D0	RNC	D2	JNC aaaa	D4	CNC aaaa
C set	D8	RC	DA	JC aaaa	DC	CC aaaa
Odd parity	E0	RPO	E2	JPO aaaa	E4	CPO aaaa
Even parity	E8	RPE	EA	JPE aaaa	EC	CPE aaaa
S not set	F0	RP	F2	JP aaaa	F4	CP aaaa
S set	F8	RM	FA	JM aaaa	FC	CM aaaa

你可能知道，存储器并不是唯一连接在微处理器上的设备。一个计算机系统通常需要输入输出设备以便于实现人机通信。输入输出设备通常包括键盘和显示器。

微处理器是怎样与外围设备(对于连接到微处理器而不是存储器的东西的称呼)进行通信的呢？外围设备具有与存储器相似的接口，微处理器可通过对应于外设的具体地址来对外设进行读写。在有些微处理器中，外围设备实际上占用了通常用来寻址存储器的地址，这种配置叫作内存映像I/O。然而在8080中，在65 536个正常地址外还有256个附加地址专门为输入输出设备预留，这些就是I/O端口（I/O Port)。I/O地址信号为A_0～A_7，但I/O访问与存储器访问不同，由8228系统控制芯片锁存的信号来区分。

OUT指令用于把累加器中的数据写到紧跟该指令的字节所寻址的I/O端口中。IN指令把端口的数据读入到累加器中。

操作码	指令
D3	OUT PP
DB	IN PP

外围设备有时需要引起微处理器的注意。例如，当你在键盘上按键时，如果微处理器能马上知道这件事通常是有帮助的。这由称作中断（interrupt）的机制来完成，这是连接至

8080INT输入端的，由外设产生的信号。

然而，当8080复位时，它不能对中断产生响应。程序必须通过执行 EI（Enable interrupts）指令来允许中断，通过执行DI（Disable Interrupts）指令来禁止中断。

操作码	指令
F3	DI
FB	EI

8080的INTE输出端信号表明允许中断。当外设需要中断微处理器当前工作时，它把 8080的INT输入端设置为1。8080通过从存储器中取出指令对它作出响应，但控制信号表明有中断发生。外设通常通过提供下述指令之一来响应 8080：

操作码	指令	操作码	指令
C7	RST 0	E7	RST 4
CF	RST 1	EF	RST 5
C7	RST 2	E7	RST 6
DF	RST 3	FF	RST 7

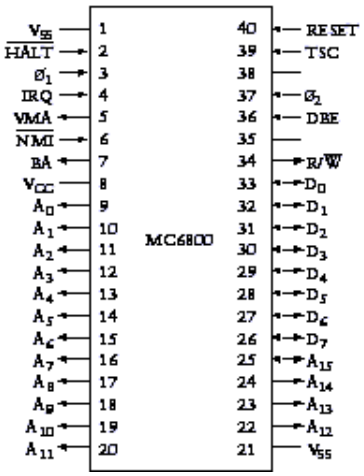
以上称作Restart指令，它们与CALL指令相似，也需要把当前程序计数器的值压入堆栈。但Restart指令随后转移到一个特定的位置： RST 0转移到地址0000h 处，RST 1转移到地址0008h处等等，直到RST 7转移到地址0038h处。位于这些地址中的代码段来处理中断。例如，来自键盘的中断引起RST 4指令执行，地址0020h处的一些代码从键盘读取数据（这将在第 21章做全面介绍）。

到此为止，已讲述了 243个操作码。下述12个字节与任何操作码无关： 08h、10h、18h、20h、28h、30h、38h、CBh、D9h、DDh、EDh和FDh。这样总共有255个操作码。下面还要提到一个操作码：

操作码	指令
00	NOP

NOP代表 no op，即no operation （无操作）。NOP指令使微处理器什么都不做。这有什么作用吗？用于填空。8080通常可以执行一批NOP指令而不会有任何坏情况发生。

以下不打算再详细讨论 Motorola 6800，因为它的设计与功能与 8080非常相似。下面是6800的40个管脚：



V_{SS}代表接地，V_{CC}是5V电源。与8080相似，6800有16个地址输出信号和既可作为输入又可作为输出的8个数据信号。它有RESET信号和R/$\overline{W}$信号。$\overline{IRQ}$信号代表中断请求。6800的时钟信号比8080的更加简单。6800没有I/O端口的概念，所有输入输出设备都必须是6800存储器地址空间的一部分。

6800有一个16位程序计数器PC、一个16位堆栈指针SP、一个8位状态寄存器（作为标志）以及两个8位累加器A和B。它们都被看成是累加器（B不是只作为一个寄存器）是因为没有能用A来做而不能用B来做的事。6800没有附加的8位寄存器。

6800中有一个16位索引寄存器（index register），可用来保存一个16位地址，很像8080中的寄存器对HL。对于许多指令来说，它们的地址都可以由索引寄存器和紧跟在操作码后的地址之和得到。

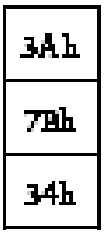
虽然6800和8080所实现的操作相同 —— 装载、保存、加法、减法、移位、转移、调用，但很明显的区别是：它们的操作码和助记符完全不同。例如，下面是6800的分支转移指令：

操作码	指令	含义
20h	BRA	转移
22h	BHI	大于则转移
23h	BLS	小于或相同则转移
24h	BCC	进位为0则转移
25h	BCS	进位置1则转移
26h	BNE	不等则转移
27h	BEQ	相等则转移
28h	BVC	溢出为0则转移
29h	BVS	溢出置1则转移
2Ah	BPL	为正则转移
2Bh	BMI	为负则转移
2Ch	BGE	大于或等于0则转移
2Dh	BLT	小于0则转移
2Eh	BGT	大于0则转移
2Fh	BLE	小于或等于0则转移

6800没有像8080中那样的奇偶标志位PF，但它有一个8080中没有的标志位 —— 溢出标志位（overflow flag）。上述转移指令中有些依赖于标志位的组合。

当然8080和6800指令集是不同的，这两个芯片是同一时间由不同的两个公司的两组不同的工程师设计的。这种不兼容性意味着每一种芯片不能执行对方的机器代码，为一种芯片开发的汇编语言程序也不能翻译成可在另一种芯片上执行的操作码。编写可在多于一种处理器上执行的计算机程序是第24章的主题。

8080和6800还有一个有趣的不同点：在两种微处理器中，LDA指令都是从一个特定的地址处装载到累加器。例如，在8080中，下列字节序列：



8080LDA指令

将把存储在地址347Bh处的字节装载到累加器。现在把上述指令与6800的LDA指令相比较，后者采用称作6800的扩展地址模式：

B6h	6800LDA指令
7Bh	
34h	

该字节序列把存储在地址7B34h处的字节装载到累加器A中。

这种不同点是很微妙的。当然，你也可能认为它们的操作码不同：对8080来说是3Ah，对6800来说是B6h。但主要是这两种微处理器处理紧随操作码后的地址是不同的，8080认为低位在前，高位在后；6800则认为高位在前，低位在后。

这种Intel和Motorola微处理器保存多字节数时的根本不同从没有得到解决。直到现在，Intel微处理器在保存多字节数时，仍是最低有效字节在前（即在最低存储地址处）；而Motorola微处理器在保存多字节数时，仍是最高有效字节在前。

这两种方法分别叫作little-endian(Intel方式)和big-endian (Motorola方式)。辩论这两种方式的优劣可能是很有趣的，不过在此之前，要知道术语big-endian来自于Jonathan Swift的《Gulliver's Travels》，指的是Lilliput和Blefuscu在每次吃鸡蛋前都要互相碰一下。这种辩论可能是无目的的。先不说哪种方法在本质上说是不是正确的，但这种差别的确当在基于little-endian和big-endian机器的系统之间共享信息时会带来附加的兼容性问题。

这两种微处理器后来怎样了呢？8080用在一些人所谓的第一台个人计算机上，不过可能更准确的说法是第一台家用计算机上。下图是Altair 8800，出现在1975年1月份的《Popular Electronics》杂志的封面上。

当你看到Altair 8800时，前面面板上的灯泡和开关看起来似乎很熟悉。这和第16章为64KB RAM阵列建议的初始“控制面板”的界面是同一类型的。

8080之后出现了Intel 8085，更具意义的是出现了Zilog制造的Z-80芯片。Zilog是Intel公司的竞争对手，是由Intel公司的前雇员，也曾在4004芯片上做出重要贡献的Federico Faggin

建立的。Z-80与8080完全兼容，且增加了许多很有用的指令。1977年，Z-80用于Radio Shack TRS-80 Model1上。

也是在1977年，由Steven Jobs和Stephen Wozniak建立的苹果计算机公司推出了APPLE II。APPLE II既不用8080也不用6800，而是使用了采用MOS技术的更便宜的6502芯片，它是对6800的增强。

1978年6月，Intel公司推出了8086，一个16位微处理器，它可访问的存储空间达到1MB。8086的操作码与8080不兼容，但它包含乘法和除法指令。一年后，Intel公司又推出了8088，其内部结构与8086相同，但其外部按字节访问存储器，因此该微处理器可使用较流行的为8080设计的8位支持芯片。IBM在其5150个人计算机——通常叫作IBM PC——上使用了8088芯片，这种个人计算机在1981年秋季推出。

IBM进军PC市场产生了巨大影响，许多公司都发布了与PC兼容的机器（兼容的含义在随后各章里将要详细讨论）。多年来，“IBM PC兼容机”也暗指“Intel inside”，特指所谓x86家族的Intel微处理器。Intel x86家族继续发展，1985年出现了32位的386芯片，1989年出现了486芯片。1993年初，出现了Intel Pentium微处理器，普遍地用在PC兼容机上。虽然这些Intel微处理器都不断增加了指令的指令集，但它们仍然支持从8086开始的所有以前处理器的操作码。

苹果公司的Macintosh首次发布于1984年，它使用了Motorola 68000——一个16位的微处理器，也即6800的下一代处理器。68000和它的后代（常称为68K系列）是制造出的最受欢迎的一类微处理器。

从1994年开始，Macintosh计算机开始使用Power PC，一种由Motorola、IBM和Apple公司联合开发的微处理器。PowerPC是由一种称作RISC（精简指令集计算机）的微处理器体系结构来设计的，它试图通过简化某些方面以提高处理器的速度。在RISC计算机中，每条指令通常长度相同，（在PowerPC中为32位），存储器访问只限于装载和保存指令，且指令做简单操作而不是复杂操作。RISC处理器通常有大量的寄存器以避免频繁访问存储器。

因为PowerPC具有完全不同的指令集，所以它不能执行68K的代码。但现在用于APPLE Macintosh的PowerPC微处理器可仿真68K。运行于PowerPC上的仿真程序逐个检验68K程序的每一个操作码，并执行适当的操作。仿真程序不如PowerPC自身代码那样快，但可以工作。

按照摩尔定律，微处理器中的晶体管数量应该每18个月翻一番。这些多增加的晶体管有什么用处呢？

有些晶体管用于增加处理器的数据宽度，从4位到8位到16位再到32位；另外一些增加的晶体管用于处理新的指令。现在大多数微处理器都有用于浮点算术运算的指令（这将在第23章解释）；还有一些新增加的指令用来进行一些重复计算，以便在计算机屏幕上显示图片和电影。

现代处理器使用了一些技术用来提高速度，其中之一是流水线技术，处理器在执行一条指令的同时读取下一条指令。由于转移指令会改变执行流程，实际上这样达不到预期效果。现在的处理器还包含一个Cache(高速缓冲存储器)，它是做在处理器内部的快速RAM阵列，用于保存最近执行的指令。因为计算机程序经常执行一些小的指令循环，因而Cache可以避免这些指令重复装载。所有这些速度提升措施都需要在处理器中有更多的逻辑器件和晶体管。

如前所述，微处理器只是完整的计算机系统的一部分（尽管是最重要的部分）。我们将在第21章构造这样一个系统，但首先必须学习怎样编码存在存储器中的除了操作码和数字外的其他东西。我们必须返回到小学一年级，再学习一下怎样读写文本。