操作数，那么它们之间就要用逗号分割。操作数的个数取决于指令的种类。也有不需要操作数的指令，比如用于停止 CPU 运转的 HALT 指令。

汇编语言的语法和英语祈使句的语法很像。若对比英语的祈使句 Give me money 和汇编语言的语句，就可以看出在英语的祈使句中，一开头放置了一个表示“做什么”的动词，这个动词就相当于汇编语言中的操作码。在动词后面放置了一个表示“动作作用到什么上”的宾语，这个宾语就相当于汇编语言中的操作数。因为程序的作用是向 CPU 发出指令，而且编程语言又是由说英语的人发明的，所以编程语言与英语祈使句类似也就不足为奇了。

构成机器语言的是二进制数，而在汇编语言中，则使用十进制数和十六进制数记录数据。若仅仅写出 123 这样的数字，表示的就是十进制数；而像 123H 这样在数字末尾加上了一个 H（H 表示 Hexadecimal，即十六进制数），表示的就是十六进制数。在代码清单 3.2 所示的程序中，使用的都是十进制数。

在表 3.1 中有这样几条指令希望诸位注意。在第 2 章中介绍过，Z80 CPU 的 $\overline{\text{MREQ}}$ 引脚和 $\overline{\text{IORQ}}$ 引脚实现了一种能区分输入输出对象的机制，可以区分出使用着相同内存地址的内存和 I/O。在汇编语言中，读写内存的指令不同于读写 I/O 的指令。一旦执行了读写内存的指令，比如 LD 指令，$\overline{\text{MREQ}}$ 引脚上的值就会变为 0，于是内存被选为输入输出的对象；而一旦执行了读写 I/O 的指令，比如 IN 或 OUT 指令，$\overline{\text{IORQ}}$ 引脚上的值就会变为 0，于是 I/O（这里用的是 Z80 PIO）被选为输入输出的对象。

表 3.1　Z80 CPU 中的主要指令

指令的种类	助记符	功能
运算指令	ADD A, num	把数值 num 加到寄存器 A 的值上
	ADD A, reg	把寄存器 reg 的值加到寄存器 A 的值上
	SUB num	从寄存器 A 的值中减去数值 num
	SUB reg	从寄存器 A 的值中减去寄存器 reg 的值
	INC reg	将寄存器 reg 的值加 1
	DEC reg	将寄存器 reg 的值减 1
	AND num	计算寄存器 A 的值和数值 num 的逻辑积
	AND reg	计算寄存器 A 的值和寄存器 reg 的值的逻辑积
	OR num	计算寄存器 A 的值和数值 num 的逻辑和
	OR reg	计算寄存器 A 的值和寄存器 reg 的值的的逻辑和
	XOR num	计算寄存器 A 的值和数值 num 的逻辑异或
	XOR reg	计算寄存器 A 的值和寄存器 reg 的值的逻辑异或
	SLA reg	对寄存器 reg 的值进行算数左移运算
	SRA reg	对寄存器 reg 的值进行算数右移运算
	SRL reg	对寄存器 reg 的值进行逻辑右移运算
	CP num	比较寄存器 A 的值和数值 num 的大小
	CP reg	比较寄存器 A 的值和寄存器 reg 的值的大小
内存与 CPU 之间的输入输出指令	LD reg, num	把数值 num 写入到寄存器 reg 中
	LD reg1, reg2	把寄存器 reg2 的值写入到寄存器 reg1 中
	LD (num), reg	把寄存器 reg 的值写入到地址 num 上
	LD (reg), reg	把寄存器 reg2 的值写入到存放在寄存器 reg1 中的地址上
	PUSH reg	把寄存器 reg 的值写入到栈中
	POP reg	把由栈顶读出的数据存放到寄存器 reg 中
I/O 与 CPU 之间的输入输出指令	IN A, (num)	从地址 num 中读出数据，存放到寄存器 A 中
	IN reg, (C)	从存储在寄存器 C 中的地址上读出数据，存放到寄存器 reg 中
	OUT (num), A	把寄存器 A 的值写入到地址 num 上
	OUT (C), reg	把寄存器 reg 的值写入到存储在寄存器 C 中的地址上
程序流程控制指令	JP num	使程序的流程跳转到地址 num 上，接下来从那个地址上的指令开始执行
	CALL num	调用存放在地址 num 上的子例程
	RET	从子例程中返回
	HALT	中止 CPU 的运行

num：表示 1 个数值，(num)：表示值为 num 的地址
reg、reg1、reg2：名为 reg、reg1、reg2 的寄存器，(reg)：存储在名为 reg 的寄存器中的地址

3.3 Z80 CPU 的寄存器结构

这里先稍微复习一下第 2 章的内容。计算机的硬件有三个基本要素，CPU、内存和 I/O。CPU 负责解释、执行程序，从内存或 I/O 输入数据，在内部进行运算，再把运算结果输出到内存或 I/O。内存中存放着程序，程序是指令和数据的集合。I/O 中临时存放着用于与周边设备进行输入输出的数据。

复习就到这里，下面就来扩充所学到的知识吧。既然数据的运算是在 CPU 中进行的，那么在 CPU 内部就应该有存储数据的地方。这种存储数据的地方叫作“寄存器”。虽然也叫寄存器，但是与 I/O 的寄存器不同，CPU 的寄存器不仅能存储数据，还具备对数据进行运算的能力。CPU 带有什么样的寄存器取决于 CPU 的种类。Z80 CPU 所带有的寄存器如图 3.2 所示[①]。A、B、C、D 等字母是寄存器的名字。在汇编语言当中，可以将寄存器的名字指定为操作数。

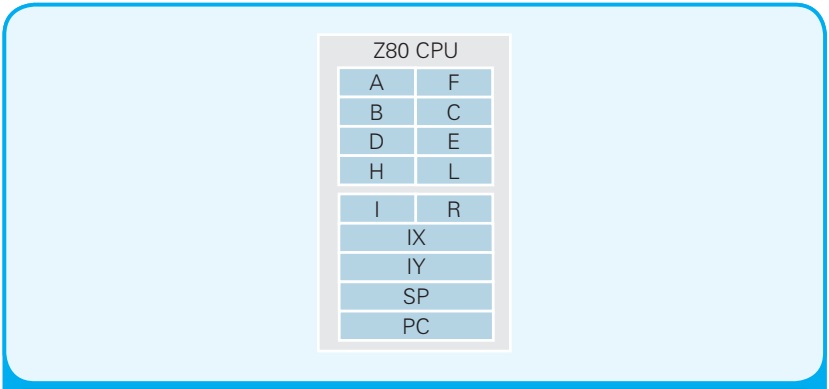

图 3.2 Z80 CPU 的寄存器

① A、B、C、D、E、F、H、L 每个寄存器都带有一个辅助寄存器，本节省略了对它们的介绍。

IX、IY、SP、PC 这 4 个寄存器的大小是 16 比特，其余寄存器的大小都是 8 比特。寄存器的用途取决于它的类型。有的指令只能将特定的寄存器指定为操作数。

举例来说，A 寄存器也叫作“累加器”，是运算的核心。所以连接到它上面的导线也一定会比其他寄存器的多。F 寄存器也叫作“标志寄存器”，用于存储运算结果的状态，比如是否发生了进位，数字大小的比较结果等。PC 寄存器也叫作“程序指针”，存储着指向 CPU 接下来要执行的指令的地址。PC 寄存器的值会随着滴答滴答的时钟信号自动更新，可以说程序就是依靠不断变化的 PC 寄存器的值运行起来的。SP 寄存器也叫作“栈顶指针”，用于在内存中创建出一块称为“栈”的临时数据存储区域。

既然诸位已经熟悉了寄存器的功能，下面笔者就开始介绍代码清单 3.2 的内容。这段程序大体上可以分为两部分——“设定 Z80 PIO”和“与 Z80 PIO 进行输入输出”。Z80 PIO 带有两个端口（端口 A 和端口 B），用于与周边设备输入输出数据。首先必须为每个端口设定输入输出模式。这里端口 A 用于接收由指拨开关输入的数据，为了实现这个功能，需要如下的代码。

```
LD   A, 207
OUT (2), A
LD   A, 255
OUT (2), A
```

这里的 207 和 255 是连续向 Z80 PIO 的端口 A 控制寄存器（对应该 I/O 的地址编号为 2）写入的两个数据。虽然使用 OUT 指令可以向 I/O 写入数据，但是不能直接把 207、255 这样的数字作为 OUT 指令的操作数。操作数必须是已存储在 CPU 寄存器中的数字，这是汇编语言

的规定。

于是，先通过指令“LD A, 207”把数字 207 读入到寄存器 A 中，再通过指令“OUT (2), A”把寄存器 A 中的数据写入到 I/O 地址所对应的寄存器中。像“(2)”这样用括号括起来的数字，表示的是地址编号。端口 A 控制寄存器的 I/O 地址是 2 号。

一旦把 207 写入到端口 A 控制寄存器，Z80 PIO 就明白了：“哦，想要设定端口 A 的输入输出模式啊。”而通过接下来写入的 255，Z80 PIO 就又知道：“哦，要把端口 A 设定为输入模式啊。”

同样地，通过下面的程序可以将端口 B 设定为输出模式。

```
LD  A, 207
OUT (3), A
LD  A, 0
OUT (3), A
```

先把 207 写入到端口 B 控制寄存器（对应的 I/O 地址为 3 号），然后写入 0。这个 0 表示要把端口 B 设定为输出模式。应该使用什么样的数字设定端口，在 Z80 PIO 的资料上都有说明。用 207、255、0 这样的数字来表示功能设定参数，这也是为了适应计算机的处理方式。

完成了 Z80 PIO 的设定后，就进入了一段死循环处理，用于把由指拨开关输入的数据输出到 LED。为了实现这个功能，需要如下的代码。

```
LOOP: IN  A, (0)
      OUT (1), A
      JP  LOOP
```

“IN A, (0)”的作用是把数据由端口 A 数据寄存器（连接在指拨开关上，对应的 I/O 地址为 0 号）输入到 CPU 的寄存器 A。“OUT (1), A”

的作用是把寄存器 A 的值输出到端口 B 数据寄存器上（连接在 LED 上，对应的 I/O 地址为 1 号）。

“JP LOOP”的作用是使程序的流程跳转到 LOOP（笔者随意起的一个标签名）标签所标识的指令上。JP 是 Jump 的缩写。“IN A, (0)”所在行的开头有一个标签“LOOP:”，代表着这一行的内存地址。正如刚才所讲的那样，在用汇编语言编程时，如果老想着“这一行对应的内存地址是什么来着?”就会很不方便，所以就要用“LOOP:”这样的标签代替内存地址。当把标签作为 JP 指令的操作数时，标签名的结尾不需要冒号“:”，但是在设定标签时，标签名的结尾则需要加上一个冒号，这一点请诸位注意。

3.4　追踪程序的运行过程

用汇编语言编写的程序是不能直接运行的，必须先转换成机器语言。机器语言是唯一一种 CPU 能直接理解的编程语言。从汇编语言到机器语言的转换方法将在稍后介绍，这里先来看一下代码清单 3.3，里面列出了事先转换出来的机器语言，以及对应的汇编语言。1 条汇编语言的指令所对应的机器语言由多个字节构成。而且，同样是汇编语言中的 1 条指令，有的指令对应着 1 个字节的机器语言，有的指令则对应着多个字节的机器语言。汇编语言中的 1 条指令能转换成多少条机器语言取决于指令的种类以及操作数的个数。代码清单 3.3 中第一个内存地址是 00000000（0 号地址），下一个地址是 00000010（2 号地址），中间隔了 2 个地址，这说明如果从 0 号地址开始存储一条 2 字节的机器语言，那么下一条机器语言就从 2 号地址开始存储。

下面就一边看着代码清单 3.3，一边跟随着 CPU 解释、执行机器

语言程序吧。在这里，我们假设机器语言的程序是像代码清单 3.3 那样被存储在内存中的。

一旦重置了 CPU，00000000 就会被自动存储到 PC 寄存器中，这意味着接下来 CPU 将要从 00000000 号地址读出程序。首先 CPU 会从 00000000 号地址读出指令 00111110，判断出这是一条由 2 个字节构成的指令，于是接下来会从下一个地址（即 00000001，1 号地址，代码清单 3.3 中并没有标记出该地址本身）读出数据 11001111，把这两个数据汇集到一起解释、执行。执行的指令是把数值 207 写入到寄存器 A，用汇编语言表示的话就是“LD A, 207”。这时，由于刚刚从内存读出了一条 2 字节的指令（占用 2 个内存地址），所以 PC 寄存器的值要增加 2，并接着从 00000010 号地址读出指令，解释并执行。

接下来的流程与此类似，通过反复进行“读取指令”“解释、执行指令”“更新 PC 寄存器的值”这 3 个操作，程序就能运行起来了。一旦执行完最后一行的 JP LOOP 所对应的机器语言，PC 寄存器的值就会被设为标签 LOOP 对应的地址 00010000，这样就可以循环执行同样的操作。请诸位重点观察 PC 寄存器是如何控制程序流程的。

代码清单 3.3　汇编语言与机器语言的对应关系

```
地址       机器语言                                  标签    操作码      操作数
00000000  00111110 11001111                          LD      A, 207
00000010  11010011 00000010                          OUT     (2), A
00000100  00111110 11111111                          LD      A, 255
00000110  11010011 00000010                          OUT     (2), A
00001000  00111110 11001111                          LD      A, 207
00001010  11010011 00000011                          OUT     (3), A
00001100  00111110 00000000                          LD      A, 0
00001110  11010011 00000011                          OUT     (3), A
00010000  11011011 00000000                  LOOP:   IN      A, (0)
00010010  11010011 00000001                          OUT     (1), A
00010100  11000011 00010000 00000000                 JP      LOOP
```

3.5　尝试手工汇编

在 CPU 的资料中，明确写有所有可以使用的助记符，以及助记符转换成机器语言后的数值。只要查看这些资料，就可以把用汇编语言编写的程序手工转换成机器语言的程序，这样的工作称为“手工汇编”。进行手工汇编时，要一行一行地把用汇编语言编写的程序转换成机器语言。下面就实际动手试一试吧。表 3.2 列出了汇编语言中必要指令的助记符、助记符所对应的机器语言，以及执行这些机器语言所需的时钟周期数。

表 3.2　从助记符到机器语言的转换方法

助记符	机器语言	时钟周期数
LD A, num	00111110 num	7
OUT (num), A	11010011 num	11
IN A, (num)	11011011 num	11
JP num	11000011 num	10

下面就从汇编语言的第 1 行开始转换。第一行的“LD A, 207”匹配“LD A, num”这个模式，所以可以先转换成“00111110 num”。然后将十进制数的 207 转换成 8 比特的二进制数，用这个二进制数替换 num。使用 Windows 自带的计算器程序就可以很方便地把十进制数转换成二进制数。从 Windows 的开始菜单中选择“运行”，输入 calc 后点击“确定”按钮，就可以启动计算器程序。

接下来，从计算器的“查看”菜单中选择“科学型”，这样就得到了一个可以用十进制数或二进制数表示数字的计算器了。首先选中“十进制”单选框，然后输入 207，接下来选中“二进制”单选框，这样 207 就变成了二进制数的 11001111（如图 3.3 所示）。至此，“LD A, 207”就转换成了机器语言 00111110 11001111。由于这条指令存储在内

存最开始的部分（00000000 号地址），所以要把这条指令和内存地址像下面这样并排写下来。

```
地址          汇编语言          机器语言
00000000    LD A, 207       00111110 11001111
```

图 3.3　用 Windows 的计算器程序把十进制数转换成二进制数

第 2 条指令“OUT (2), A”匹配“OUT (num), A”这个模式，所以可以先转换成“11010011 num”。然后把 num 的部分替换成 00000010，即用 8 比特的二进制数表示的十进制数 2，最终就得到了机器语言“11010011 00000010”。因为内存中已经存储了 2 字节的机器语言，所

以这条机器语言要从 00000010 号地址（用十进制表示的话就是 2 号地址）开始记录。

```
地址          汇编语言          机器语言
00000010    OUT (2), A      11010011 00000010
```

这之后由于 LD 指令和 OUT 指令又以相同的模式出现了 3 次，所以可以用相同的步骤转换成机器语言。请诸位注意，机器语言中每条语句的字节数是多少，内存地址就相应地增加多少。

```
地址          汇编语言          机器语言
00000100    LD A, 255       00111110 11111111
00000110    OUT (2), A      11010011 00000010
00001000    LD A, 207       00111110 11001111
00001010    OUT (3), A      11010011 00000011
00001100    LD A, 0         00111110 00000000
00001110    OUT (3), A      11010011 00000011
```

接下来是“IN A, (0)”匹配“IN A, (num)”这个模式，所以可以先转换成“11011011 num”。然后把 num 替换成 00000000，即用 8 比特的二进制数表示的十进制数 0，最终就得到了机器语言“11011011 00000000”。对于接下来的“OUT (1), A”，也可以按照同样的方法转换。

```
地址          汇编语言          机器语言
00010000    IN A, (0)       11011011 00000000
00010010    OUT (1), A      11010011 00000001
```

最后一句的 JP LOOP 匹配模式“JP num”，所以可以先转换成“11000011 num”。请注意这里要用 16 比特的二进制数替代作为内存地址的 num。在微型计算机中是以 8 比特为单位指定内存地址的，但在 Z80 CPU 中用于设定内存地址的引脚却有 16 个，所以在机器语言中也