第5章　控制器逻辑

中央处理器(CPU)是计算机的中枢,是计算机系统的运算核心和控制核心,主要由运算器(Arithmetic Logic Unit,ALU)、控制器(Control Unit,CU)、一组寄存器以及相关总线组成。了解计算机的工作原理的关键是搞清 CPU 的工作原理。关于 ALU 的简单原理已经在 1.3.1 节中介绍了,本章主要对 CU 进行逻辑分析。CU 的核心功能是分析指令,设计依据是指令系统。

5.1　处理器的外特性——指令系统

5.1.1　指令与指令系统

1. 指令及其基本格式

在计算机中,指令(instruction)是要求计算机完成某个基本操作的命令。计算机程序就是由一组指令组成的代码序列。这里所说的"基本"是针对具体的 CPU 而言的,不同的 CPU 所指的"基本"二字的意义不同,所能执行的基本操作的数量和种类也不相同。但是,任何 CPU 都必须满足最小完备性原则,即它所能执行的基本操作必须能组成该 CPU 所承担的全部功能。也就是说,有的 CPU 的基本操作多一些、复杂一些,有的 CPU 的基本操作少一些、简单一些,但它们的组合效果应当相同。例如,有的 CPU 将乘法作为基本操作之一;有的没有乘法操作,但可以用加法和移位操作组成乘法操作。

在 1.2.9 节中已经介绍过,一条指令由操作码和地址码两大部分组成,并且可以按照地址的数量把指令分为如下 4 种类型:

- 3 地址指令。
- 2 地址指令。
- 1 地址指令。
- 0 地址指令。

一条指令的长度由操作码和各地址码的长度决定。

指令地址码的长度由指令的寻址空间——存储器的容量决定。例如,一个 10 位的地址码的寻址空间为 2^{10}。一个 1GB 的内存需要的地址码为 30 位,对于 3 地址指令,指令的长度就需要 100 位左右。这样的指令就太长了。但是多地址指令比少地址指令编写出来的程序长度要小,并且执行速度要快。

为了解决用较短的地址码访问大容量存储器的问题,人们研究出了多种寻址方式。具体内容将在后面介绍。

操作码的长度由操作的种类决定,一个包含 n 位的操作码最多能表示 2^n 种操作。不同的计算机中,可以采用定长操作码——操作码占有固定长度的位数,也可以采用变长操作

码——操作码的长度不固定。变长操作码可以用扩展窗口将部分地址作为操作码使用，以增加指令条数。

例 5.1 某计算机字长为 16 位，操作码占 4 位，有 3 个 4 位的地址码，试说明如何扩展该机器的指令系统。

解：在定长操作码系统中，操作码的长度与地址码是冲突的，即地址码越长，操作码就要越短；而操作码短了，指令系统中的指令数就少了。例如在本例中，操作码只有 4 位，指令系统中最多只能有 16 条指令，这常常是不够的。为了解决这个矛盾，可以采用变长操作码。即减少地址数，扩展操作码。

在进行操作码扩展时，要特别注意一点：长码中不可出现短码。因为编译器会首先从短码开始区分不同的指令类型。因此，短码至少要留出一位用于连接扩展位，称为扩展窗口位。

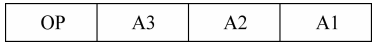

图 5.1 指令格式

本例的指令格式如图 5.1 所示。

则计算不同地址数指令的步骤如下。

(1) 原始 OP 占有 4 位，用 0000～1110 定义 15 条三地址指令，留下 1111 作为扩展窗口，与 A3 一起组成扩展字段。

(2) 由于长码中不可出现短码，所以只能用 1111 0000～1111 1110 定义 15 条二地址指令。留下 1111 1111 作为一地址指令的扩展窗口，与 A2 组成扩展操作码。

(3) 同理，用 1111 1111 0000～1111 1111 1110 定义 15 条一地址指令。

(4) 用 1111 1111 1111 0000～1111 1111 1111 1111 定义 16 条零地址指令。

2. 指令系统

一个 CPU 所能承担的全部基本操作由一组对应的指令描述。这组完整地描述该 CPU 的指令就称为该 CPU 的指令系统(instruction system)或指令集(instruction set)，也称该 CPU 的机器语言。

指令系统决定了 CPU 的外特性。一方面，指令系统表明了 CPU 能执行哪些基本操作，因此，指令系统是(系统)程序员在该 CPU 上进行程序设计的依据。另一方面，功能模拟和结构模拟是研究、制造任何一种机器的两个最重要的途径。20 世纪 60 年代人们从程序员的角度观察计算机的属性，开始用“计算机体系结构”来统一功能和结构这两个方面。但是由于功能和结构两者仍然存在差别，通常把计算机的功能方面叫做外(宏)体系结构，把计算机的实现方面叫做内(微)体系结构。由于 CPU 的功能是取指令—分析指令—执行指令，所以一个 CPU 设计所依据的功能也来自指令系统，即指令系统是 CPU 设计的基本依据。要设计一个 CPU，先要为它设计指令系统。

3. 指令系统的描述语言——机器语言与汇编语言

一个 CPU 的指令系统就是与该 CPU 进行交互的工具，用其可以让该 CPU 完成特定的操作。所以一个 CPU 的指令系统就可以看成该 CPU 的机器语言。显然，不同的 CPU 具有不同的机器语言。

在表现形式上，机器语言就是用 0、1 码描述的指令系统。用它编写的程序难读、难记、

难查错，给程序设计和计算机的推广、应用、发展造成了极大困难。面对这一不足，人们最先是采用一些符号来代替0、1码指令，如用ADD代替“加”操作码等。这种语言称为符号语言。下面是几条符号指令与其对应的机器指令代码的例子：

```
MOV    AH,   01H              ; 机器指令代码: B401H
XOR    AH,   AH               ; 机器指令代码: 34E2H
MOV    AL,   [SI+0078H]       ; 机器指令代码: 8A847800H
MOV    BP,   [0072H]          ; 机器指令代码: 8B2E7200H
DEC    DX                     ; 机器指令代码: 4AH
IN     AL,   DX               ; 机器指令代码: ECH
```

符号语言方便了编程，用它编写程序效率高，写出的程序易读性好，提高了程序的可靠性。但是，符号语言是不能直接执行的，必须将之转换为机器语言才能执行。

符号语言程序转换为机器语言程序的方法是查表。这是非常简单的工作。为了将这种查表工作自动化，除了正常的指令外，还需要添加一些对查表进行说明的指示性指令——伪指令。这种查表工作称为汇编。用符号语言描述并增加了指示性指令的指令系统称为汇编语言。汇编语言为程序员提供了极大的方便，也提高了程序的可靠性。通常在介绍指令系统时采用的都是汇编语言。

汇编语言指令的一般形式如下：

```
标号: 操作码   地址码  (操作数)              ; 注释
```

下面是一段用Intel 8086汇编语言描述的计算A＝2＋3的程序：

```
        ORG    C0H              ; C0H 为程序起始地址
START:  MOV    AX, 2            ; 2→AX,AX 为累加器,START 为标号
        ADD    AX,3             ; 3+ (AX)→AX
        HALT                    ; 停
        END    START            ; 结束汇编
```

由于汇编语言比机器语言的易读性好，又与机器语言一一对应，所以机器指令都可按汇编语言符号形式给出。

4. 汇编语言的基本语法

下面以Intel 8086汇编语言为例，介绍汇编语言中的几个基本语法。

1）数据类型

Intel 8086汇编语言中允许使用如下形式的数值数据。

- 二进制数据，后缀为B，如10101011B。
- 十进制数据，后缀为D，如235D。
- 八进制数据，后缀为Q(本应是O，为避免与数字0相混，用Q代替)，如235Q。
- 十六进制数据，后缀为H，如BAC3H。

加后缀的目的是便于区分。数值较多的是采用十六进制。有时也允许用名字来表示数据，如用PI代表3.141593等。

用引号作为起止界符的一串字符称为字符串常量，如'A'(等价于41H)，'B'(等价于

42H),'AB'(等价于 4142H)等。

2) 运算符

汇编语言中的运算符有以下 3 类:

- 算术运算符:+,-,*,/。
- 关系运算符:EQ(相等),NE(不相等),LT(小于),GT(大于),LE(小于等于),GE(大于等于)。
- 逻辑运算符:AND("与"),OR("或"),NOT("非")。

3) 操作码

可以用算术运算符,也可以用英文单词,如用 SUB 表示减去,用 ADD 表示相加等。

4) 地址码

指令中的地址码可以用十六进制或十进制表示,也可以用寄存器名或存储器地址表示。

5) 标号与注释

汇编语言还允许使用标号及注释,以增加可读性。这部分与机器语言没有对应关系,仅用于使人在阅读程序时容易理解。

5.1.2 寻址方式

在设计一个 CPU 时,往往要根据用户需求、技术条件和成本的权衡来,确定字长以及指令格式。指令格式一经确定了,指令中各个字段的布局就基本确定了,地址字段的长度也就基本确定了。一般说来,指令中地址字段的长度是非常有限的。指令设计的一个重要任务就是使用非常有限的地址字段在尽可能大的范围内寻找操作数,于是就变幻出许多寻址方式来。

下面结合 8086 介绍几种常用的寻址方式。

1. 立即寻址

采用立即寻址(immediate addressing)时,操作数不单独存放在存储器中,而是指令代码的一部分,只要将这一部分分离出来,便可以立即得到操作数。因此指令的执行速度很快,故将这种操作数称为立即操作数。图 5.2 为 8086 指令 ADD AX 3165H 的存储及执行示意图,这条指令的操作是:将 AX 的内容与立即数 3165H 相加,结果存入 AX 中。

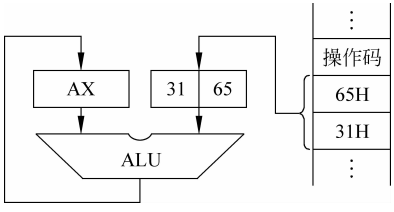

图 5.2 8086 指令 ADD AX 3165H 的存储及执行

2. 寄存器寻址

在程序执行的过程中,若把大量的中间结果和最终结果都送到存储单元中保存,则要付出时间和空间的代价,且往往是不必要的。因为中间数据仅仅是作为下一次运算的一个操作数,而还有一些数据只需立即输出而无须保存。为了提高 CPU 的工作效率,现代 CPU 中都开辟了寄存器组,用以保存运算过程中的中间结果和某些最终结果。在寄存器中的操作数称为寄存器操作数。相应的寻址方式称为寄存器寻址(register addressing)。

图 5.3　8086 的通用寄存器

寄存器设在 CPU 内部，存取的速度大大地高于存储器，所以使用寄存器存放中间结果可以提高程序的运行效率。而为了编出高效率的程序，必须先熟悉有哪些寄存器可供编程使用。如 8086 中有 8 个 8 位的通用寄存器：AL、BL、CL、DL、AH、BH、CH、DH。这 8 个 8 位的通用寄存器也可以并成 4 个 16 位的通用寄存器：AX(AL 与 AH)、BX(BL 与 BH)、CX(CL 与 CH)、DX(DL 与 DH)。它们的结构以及习惯用法如图 5.3 所示。

寄存器寻址就是按指定的寄存器代号对寄存器进行读写。如 8086 指令

```
MOV  AX,  BX
```

是将 BX 中的内容传送到 AX 中。BX 为源地址，AX 为目标地址。

寄存器寻址指令简单。巧妙地使用寄存器是提高汇编程序设计的一个关键。

3. 存储器直接寻址

指令的操作对象存放在内存的某单元中，称为存储器操作数。相应的寻址方式称为存储器寻址。存储器寻址有直接寻址、间接寻址和变址/基址寻址等。

存储器直接寻址就是把操作数的地址直接作为指令中的地址码。图 5.4 为直接寻址的示意图。其中 MOV 是操作码，△是寻址方式，AX 是寄存器代号，3056 是直接地址。这条指令的功能是把 3056H 单元的内容 A3CEH 取出来，送到累加器 AX 中。

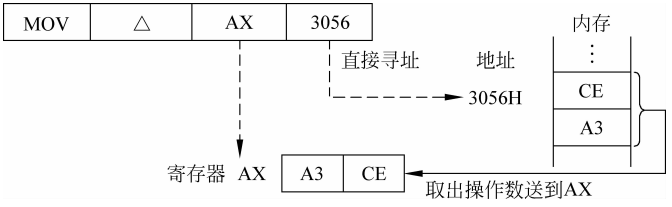

图 5.4　直接寻址

直接寻址灵活性较差，并且受指令长度的限制，寻址范围很有限。如一条 2 字节指令，操作码占 6 位后，地址码最多能占 10 位，寻址范围只有不足 1KB。

4. 存储器间接寻址

采用间接寻址方式，由地址码从存储器取出的数不是操作数本身，而是操作数的地址。还需要再以该地址从存储器取出一个数，这个数才是操作数。也就是说需要两次访问存储器，故称这种寻址方式为间接寻址方式。图 5.5 为间接寻址的示意图。这时 0B58 单元也被称为间址器或地址指针。这条指令的具体操作是从 0B58 单元中取出一个地址，再从该地址 (1A3C) 中取出操作数送到寄存器 AX 中。

间址器好像是一个“地址询问处”。这种寻址方式增加了指令的灵活性，只要改变间址器的内容，不改变指令，就可以访问到不同的操作数。同时，全部字长都可以用于存放操作

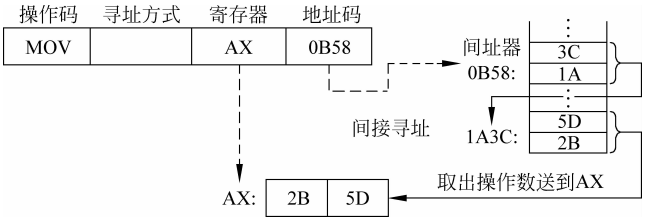

图 5.5　间接寻址示意图

数地址，能扩大寻址范围。但是，由于要多次访问内存，降低了指令的执行速度。提高速度的一个办法是用寄存器作为间址器。8086 中的间接寻址就是寄存器间接寻址方式。它规定可以用 BX、BP、SI 或 DI 作为(地址)指针。如指令

```
MOV  AX,  [SI]
```

其中 [SI]表明 SI 是(地址)指针，即 SI 中存的是操作数地址，而不是操作数。方括号是间址方式的一种表示。这条指令的功能是把 SI 所指向的数据传送到累加器 AX 中。

5. 变址/基址寻址

变址寻址和基址寻址是两种通过计算得到操作数有效地址的寻址方法，计算的方法为：基准地址＋偏移量。但是两种寻址方法的策略不同：变址寻址是用一个寄存器(称变址寄存器)存放偏移量(称为变址值)，在指令中给出基准地址，即形式地址，如图 5.6(a)所示。基址寻址相反，用寄存器(称基址寄存器)存放基准地址，在指令中给出偏移量，如图 5.6(b)所示。

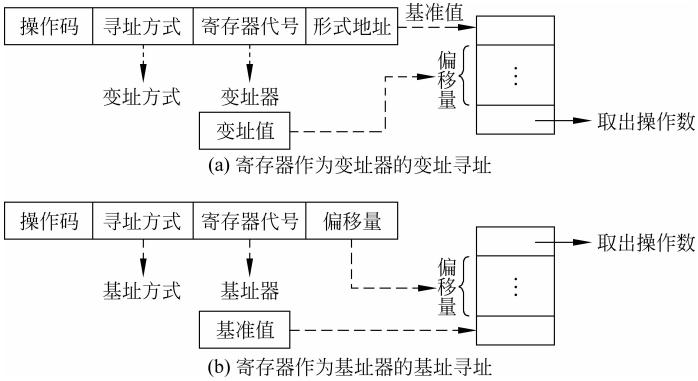

图 5.6　变址寻址与基址寻址

在 8086 中，通常用 SI 和 DI 作为变址寄存器。SI 常作为源变址寄存器，DI 常作为目标变址寄存器。采用变址寻址时，在指令中给出的是变址器的代号和偏移量，如在指令

```
MOV  CL,  [SI-100H]
```

中，将寄存器 SI 中的值减去 100H 作为操作数的地址。

在 8086 中，BX 与 BP 称为两个基址指针。采用基址寻址时，在指令中要给出基址寄存

器的代号和偏移量，如指令

```
MOV  AL,  [BX+10H]
```

其中，BX 为基址指针，10H 为逻辑地址。

基址寻址与变址寻址都能够有效地扩大寻址范围。一般变址寻址主要用于为数组等数据结构提供支持，这时可以把数组的起始位置存在变址器中。基址寻址主要用于为逻辑地址向物理地址的转换提供支持。有了这种支持，程序的装入就简单多了，只要把某道程序的装入地址放在基址寄存器中，程序员在编程时所使用的逻辑地址将通过基址寻址向物理地址变换。

基址寻址和变址寻址两种寻址方式的组合称为基址变址寻址，如指令

```
MOV  AX,  [BX+SI+3BH]
```

其中给出的是基址寄存器的代号 BX、变址寄存器的代号 SI 和偏移量 3BH。

6. 堆栈寻址

堆栈(stack)是在内存中开辟的一个存储数据的连续区域。其一端地址是固定的，称之为栈底；另一端的地址是活动的，称为栈顶。对堆栈数据的操作只能在浮动着的栈顶进行，为此设置了一个栈顶指针(SP)以指示当前栈顶的位置。一个新的数据存入堆栈，称之为进栈或压栈(PUSH)，存在原栈顶之上，成为新的栈顶，栈顶指针也随即上升；从堆栈取数据，就是将栈顶元素取出，称为退栈或弹出(POP)，栈顶指针也随即下降。堆栈就像一个弹夹，新压入的子弹总在最上边，最先打出去的子弹是最后压进去的，而最后打出去的子弹总是最先压进去的，即形成后进先出(Last In First Out，LIFO)的存储机制。

在程序设计中，堆栈主要用于子程序调用、递归调用等的断点保存和现场保护等场合。图 5.7 为使用堆栈保存子程序调用时的断点保护示意图。

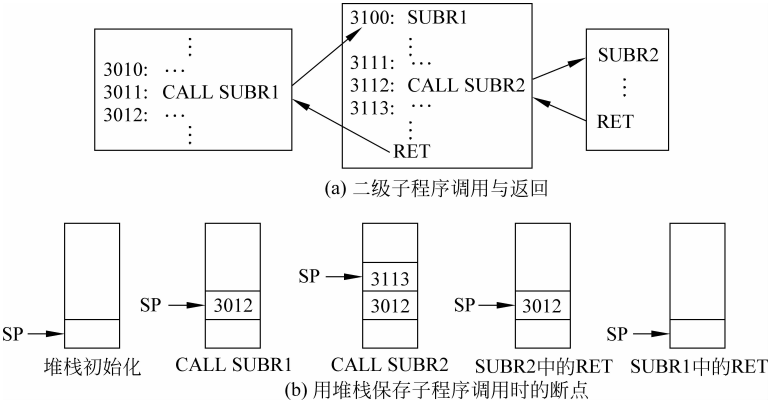

图 5.7　堆栈及其操作

8086 的当前程序的堆栈设置在堆栈段内，它的起始地址由段寄存器 SS 指示，栈顶指针由 16 位的寄存器 SP 担当，可以用下面的指令对 SP 进行初始化：

```
MOV  SP,  im
```

这时 SP 与 SS 之间的偏移量表示堆栈的大小，堆栈的最大空间是 64KB。

PUSH 和 POP 是只能对 16 位操作数执行进栈和出栈操作的两条堆栈操作指令，分别称为进栈指令和出栈指令。进栈操作时，PUSH 指令先执行 SP－2→SP，然后将一个字从源地址送到由现行 SP 寻址的堆栈两个单元；出栈操作时，POP 指令将一个字从现行 SP 寻址的堆栈两个单元送到目的地址，然后执行 SP＋2→SP 操作。例如

```
PUSH    DS                  ; 暂存 DS
PUSH    CS
POP     DS                  ; 传送 CS 到 DS
PUSH    [SI+06H]            ; 存储单元内容进栈
```

堆栈操作不仅可以暂存数据，还可以用来传送数据。但是要注意，一般情况下 PUSH 和 POP 指令应该配对使用，以保证堆栈中的数据不会紊乱。

7. 8086 的段寻址

8086 的地址总线为 20 条，所以内存地址位数是 20 位，具有 1MB 寻址能力，而地址寄存器的位数只有 16 位，可寻址的范围为 64KB。为此 8086 采用了分段技术，并且用 4 个 16 位的专门的寄存器 CS、DS、SS、ES 作为段寄存器，分别保存代码段、数据段、堆栈段和附加段的首地址的高 16 位，如图 5.8 所示。

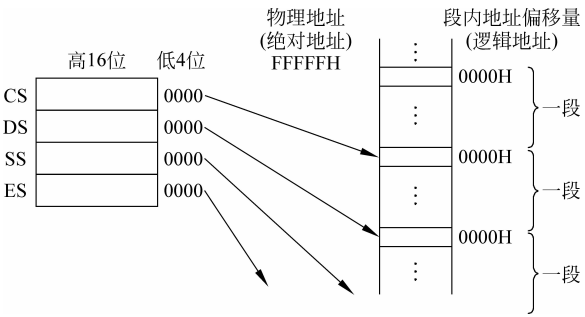

图 5.8　8086 的段地址、物理地址和逻辑地址

4 个当前段的最大长度是 64KB，相对于各段首地址，段内偏移地址都可以用 16 位的地址码来表示。这样，便可以使用 16 位的寄存器如 IP、SP、BP、SI、DI 等进行地址操作，只要确定了 CS、DS、SS、ES 的值，就可以只考虑 16 位的地址偏移量，即逻辑地址。当需要 20 位的地址时，CPU 会自动选择相应的段寄存器，并将其左移 4 位后与 16 位的地址偏移量相加，产生所需要的 20 位的物理地址。

段地址可以由段前缀指令指定，也可以由 8086 隐含地给出。

8086 的寻址都是基于段寻址的，如 8086 的直接寻址，实际上是一种段寻址。其他 8086 寻址方式也是以此为前提的。

5.1.3 Intel 8086 指令简介

本节介绍 Intel 8086 CPU 中的一些指令。

1. 数据传送类指令

数据传送指令用于寄存器、存储单元或输入输出端口之间的数据或地址传送。

1）通用数据传送指令

MOV 是最基本的通用数据传送指令。它可以在寄存器之间、寄存器和存储单元之间传送字节和字，也可以将一个立即数传送到寄存器或存储单元中。例如：

```
MOV  AL,  BL              ; 寄存器之间传送字节
MOV  SI,  [BP+8AH]        ; 存储单元和寄存器之间传送数据
                          ;(基址寻址)
MOV  AL,  06H             ; 立即数传送到寄存器
```

XCHG 是一条数据交换指令。操作数可以是寄存器或存储单元，但不能是段寄存器或立即数，也不能同时对两个存储器操作。例如：

```
XCHG  AL,  CL             ; 字节交换
XCHG  BX,  SI             ; 字交换
XCHG  AX,  [BX+SI]        ; 寄存器和存储单元之间交换数据
                          ;(基址变址寻址)
```

2）输入输出指令

这是专门用于累加器和输入输出端口之间进行数据传送的指令，而不是像通用数据传送指令那样只用于寄存器、存储单元或堆栈之间的数据传送。下面是输入输出指令的实例：

```
IN    AL,    05BH         ; 字节输入(端口地址 05BH)
OUT   0FAH,  AX           ; 字输出(端口地址 0FAH)
MOV   DX,    0358H        ; 设置端口地址(端口地址超出 8 位时
                          ; 必须先将端口地址送入 DX 寄存器
                          ; 再进行输入输出)
IN    AX,    DX           ; 字输入
```

3）地址传送指令

地址传送指令用于传送操作数的地址。LEA 用来将源操作数(必须是存储器操作数)的偏移地址传送到通用寄存器、指针或变址寄存器。例如：

```
LEA  SI,  [BX+36H]
```

LDS 是一条取(地址)指针到数据段寄存器和数据寄存器的指令。它要求源操作数是一个双字长存储器操作数，目的操作数是 16 位通用寄存器、指针或变址寄存器，但不能是段寄存器。指令执行时，双字长存储器操作数中的低地址字传送到指定的数据寄存器中，高地址字传送到数据段寄存器 DS 中。该指令用来设置字符串传送的源地址非常方便，例如指令

```
LDS  SI,  SRC
```

是把双字节(地址)指针变量 SRC 的低 16 位送 SI,高 16 位送 DS。

LES 是一条取(地址)指针到附加段寄存器的指令。该指令的操作与 LDS 指令基本类似,不同的只是把 32 位(地址)指针的高 16 位送到 ES 寄存器中,例如:

```
LES  DI,  DST
```

4) 标志传送指令

这种数据传送指令专门用于对标志寄存器进行操作。8086 指令系统提供了以下 4 条标志传送指令:

```
LAHF              ; 标志寄存器的低 8 位送 AH
SAHF              ; AH 中的内容送标志寄存器
PUSHF             ; 标志寄存器内容进栈
POPF              ; 栈顶内容送标志寄存器
```

注意,SAHF 和 POPF 指令直接影响标志位,其他传送指令均不会对标志位产生影响。

2. 算术运算指令

8086 的算术运算可以处理二进制数和无符号十进制数。其中,十进制数不带符号,用压缩码(紧凑格式)或非压缩码(非紧凑格式)保存。用压缩码表示的十进制数,每个字节可容纳两个 BCD 码,十进制数高有效位是 8 位二进制数中的高 4 位,低有效位是 8 位二进制数中的低 4 位,因此一个字节的数值范围是 00～99。而使用非压缩码时,每个字节只表示一位 BCD 码,用低 4 位二进制数表示,因此一个字节的数值范围是 0～9。做乘、除法时,高 4 位必须是 0,做加、减法时可以是任意值。

1) 加法指令

(1) 加法指令 ADD。

语法:ADD 目的,源

功能:对两个字节或字操作数相加,将其结果送到目的操作数地址。

受影响的状态标志:AF、CF、PF、OF、ZF、SF。

操作数可以是寄存器、存储器或立即数。例如:

```
ADD  AX,  BX
ADD  AL,  40H
ADD  [BX+SI+64H],  AX
```

注意:

- 参与运算的两操作数应该同时带符号或同时不带符号,并且长度必须一致。
- 两操作数不能同时为存储单元或段寄存器。
- 立即数不能作为目的操作数。

(2) 带进位加法指令 ADC。

语法:ADC 目的,源