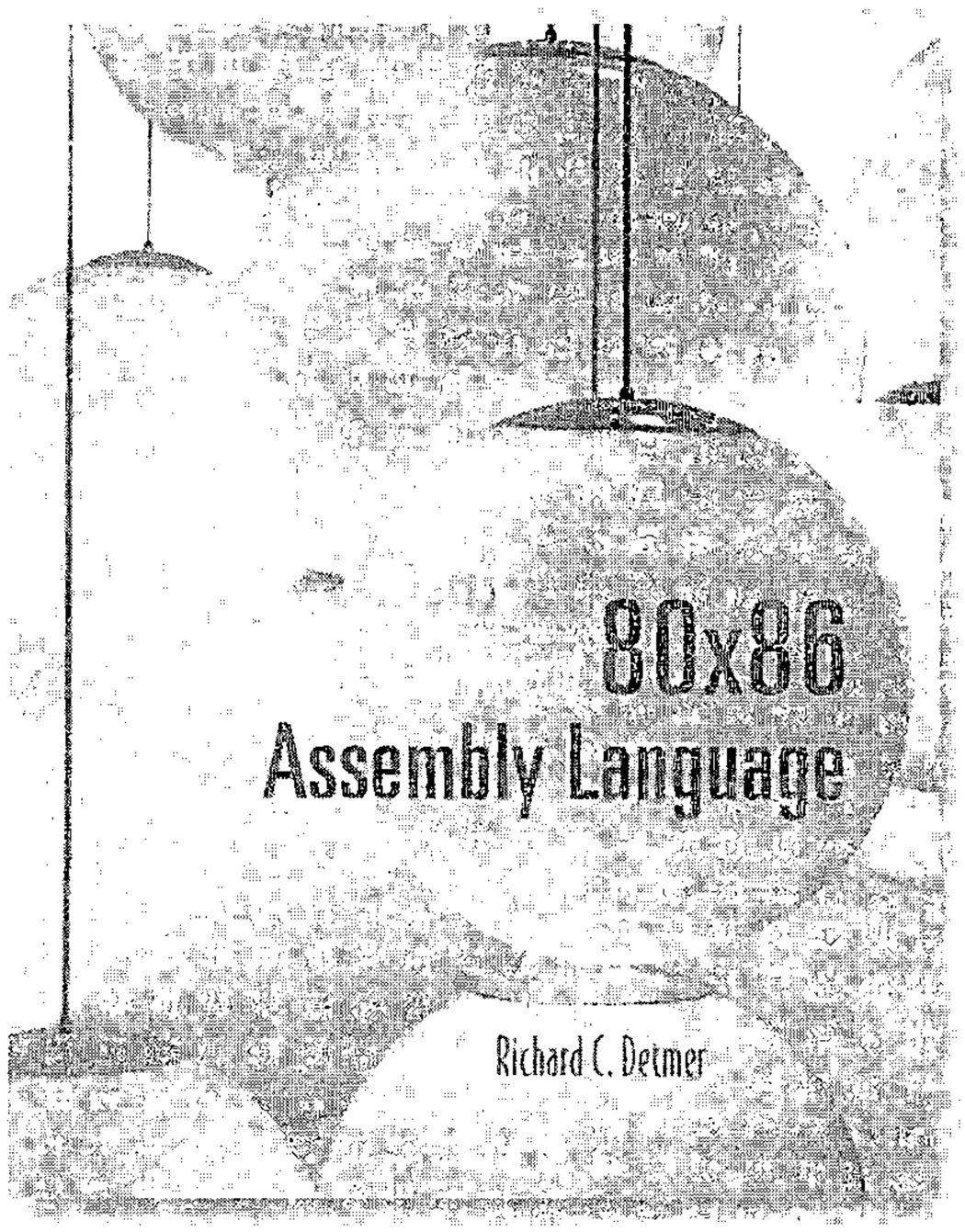
计　　算　　机　　书

80x86汇编语言基础教程

(美) Richard C. Detmer 著　郑红　陈丽琼　译



Essentials of 80x86 Assembly Language

机械工业出版社
China Machine Press

第 1 章　计算机中数的表示

用 Java 或 C++ 等高级语言编程时，要用到许多不同类型的变量（比如整型、浮点型或者字符型），变量一旦声明，就不需要考虑数据在计算机中是如何表示的。然而，用机器语言编程时，就必须考虑数据存储的位置和方式。本章将讨论 80x86 微处理器数据存储和处理的位置，以及常用的数据表示方式。

1.1　二进制数和十六进制数

计算机用**位**（bit）（二进制数制用 0 或 1 表示不同的电子状态）来表示数值。以 2 为基数，数字 0 和 1 表示**二进制**数。二进制数跟十进制数很相似，但二进制相应的权（从右到左）依次为 1、2、4、8、16 和 2 的更高次幂等，而十进制相应的权为 1、10、100、1000、10000 和 10 的更高次幂等。例如，二进制数 1101 可表示十进制数 13。

1		1		0		1		
one 8	+	one 4	+	no 2	+	one 1	=	13

二进制数很长，在读写时很不方便，比如，八位二进制数 11111010 表示十进制数 250，15 位二进制数 111010100110000 表示十进制数 30 000。而用**十六进制**（基数 16）表示时，大约只需要用到相应二进制表示的四分之一长度的位数。十六进制与二进制的转换很容易，因此，十六进制的表示可以简化二进制的表示。十六进制需要 16 个数字：其中 0、1、2、3、4、5、6、7、8 和 9 与十进制数相同；A、B、C、D、E 和 F 等同于十进制的 10、11、12、13、14 和 15。另外，这几个字母不论是小写还是大写都可用于表示数。

十六进制数中的权值对应 16 的幂，权值从右到左依次是 1、16、256 等等。十六进制数 9D7A 按如下方法计算：

```
9    ×    4096      36864    [ 4096 = 16^3 ]
+13   ×    256        3328    [ D is 13, 256 = 16^2 ]
+ 7   ×    16          112
+10   ×    1            10    [ A is 10 ]
=40314
```

得出表示的是十进制的 40314。

图 1-1 列举了一些二进制、十六进制和十进制表示的数。读者应该熟记这些数，或者能够很快地推导出来。

上面的例子给出了如何将二进制数或十六进制数转换为十进制数。那么如何将十进制数转换成二进制数或者十六进制数呢？以及如何将二进制数转换为十六进制数呢？下面将介绍如何手工实现转换。通常情况下，可用一个具有二进制、十进制和十六进制转换功能的计算器很容

十进制	十六进制	二进制
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

图 1-1　十进制、十六进制和二进制数

易实现转换，这样，数制转换只不过是按一两个键而已。这种计算器可以像十进制那样进行二进制和十六进制的数学运算，而且具有很多其他的用途。注意：这种计算器用七个显示段来显示一个数字。比如，显示小写字母 b 时，看起来像数字 6，其他有些字符也可能难以辨认。

计算器不需要把一个十六进制数转换为对应的二进制形式。事实上，许多二进制数太长，一般的计算器不易显示，因此，每四位二进制数用一个十六进制数表示。其对应的二进制位如图 1-1 的第 3 列所示。如果位数不够四位，必要的时候前面用 0 补充。例如：

$3E8E2_{16}$=11 1011 1000 1110 0010_2

转换数字下标处的 16 和 2 表示基数。如果不易混淆，则这些下标处的数字可以忽略。二进制数补齐位数是为了增强可读性，如十六进制的数字 2 转换为二进制时，最前面用 0 补齐得到 0010。但由于二进制数前面的零不会改变该二进制数的值，所以上例中十六进制数的最高位 3 不需要转换为 0011。

把二进制数转换为十六进制数格式，则与上面的步骤正好相反。把二进制数从右向左每四位进行分隔，每四位二进制数用对应的十六进制数表示，例如：

1011011101001101111 = 101 1011 1010 0110 1111 = 5BA6F

前面介绍了如何将二进制数转换为十进制数。通常，一个很长的二进制数直接转换为十进制数并不采用这种方法。更快的方法是先将二进制数转换为十六进制数，再将该十六进制数转换为十进制数。以上面的 19 位二进制数为例：

1011011101001101111_2
$=$ 101 1011 1010 0110 1111
$=$ $5BA6F_{16}$
$=$ $5\times65536+11\times4096+10\times256+6\times16+15\times1$
$=$ 375407_{10}

下面给出十进制数转换为十六进制数的算法，该算法从右到左依次生成十六进制数位。该

算法用伪代码来描述，本书中其他的所有算法和程序都将采用伪代码描述。

```
until DecimalNumber = 0 loop
        divide DecimalNumber by 16, getting Quotient and Remainder;
        Remainder (in hex) is the next digit (right to left);
        DecimalNumber := Quotient;
end until;
```

示例

以十进制数 5876 转换为十六进制数的过程为例：

- 因为这是一个 until 循环，当第一次执行程序体的时候就进行循环控制条件检查。（为什么不用 while 循环？）
- 16 整除 5876（十进制数）

```
   367      商   新的十进制数的值
16|5876
  5872
  ----
     4      余数   最右边的十六进制数位
```

当前结果：4

- 367 不等于 0，再用 16 整除。

```
    22      商   新的十进制数的值
16|367
  352
  ---
   15      余数   生成的第 2 个十六进制数位
```

当前结果：F4

- 22 不等于 0，用 16 整除。

```
     1      商   新的十进制数的值
16| 22
   16
   --
    6      余数   生成的下一个十六进制数位
```

当前结果 6F4

- 1 不等于 0，用 16 整除。

```
     0      商   新的十进制数的值
16| 1
    0
    -
    1      余数   生成的下一个十六进制数
```

当前结果 16F4

- 当前的十进制数为 0，循环终止。最后结果为 $16F4_{16}$

练习 1.1

请根据下面给出的数字将表中空白的另外两种进制形式补充完整。

	二进制	十六进制	十进制
1.	100	_____	_____
2.	10101101	_____	_____
3.	1101110101	_____	_____

4.	11111011110	_____	_____
5.	1000000001	_____	_____
6.	_____	8EF	_____
7.	_____	10	_____
8.	_____	A52E	_____
9.	_____	70C	_____
10.	_____	6BD3	_____
11.	_____	_____	100
12.	_____	_____	527
13.	_____	_____	4128
14.	_____	_____	11947
15.	_____	_____	59020

1.2 80x86 存储器

80x86 微机上的**随机存储器**（RAM）逻辑上可以看作是一个“条板单元”的集合，每个单元能存储一个**字节**（8 位）的指令或者数据。每个存储器字节都有一个 32 位的助记符，称为**物理地址**。一个物理地址通常用一个 8 位的十六进制数表示。第一个地址为 00000000_{16}，最后一个地址为无符号数的 $FFFFFFFF_{16}$。图 1-2 给出了一台 PC 中可能的存储器的逻辑图。由 $FFFFFFFF_{16} = 4\ 294\ 967\ 295$ 可知，PC 微机的存储器字节数可达到 4 294 967 296，即 4GB。

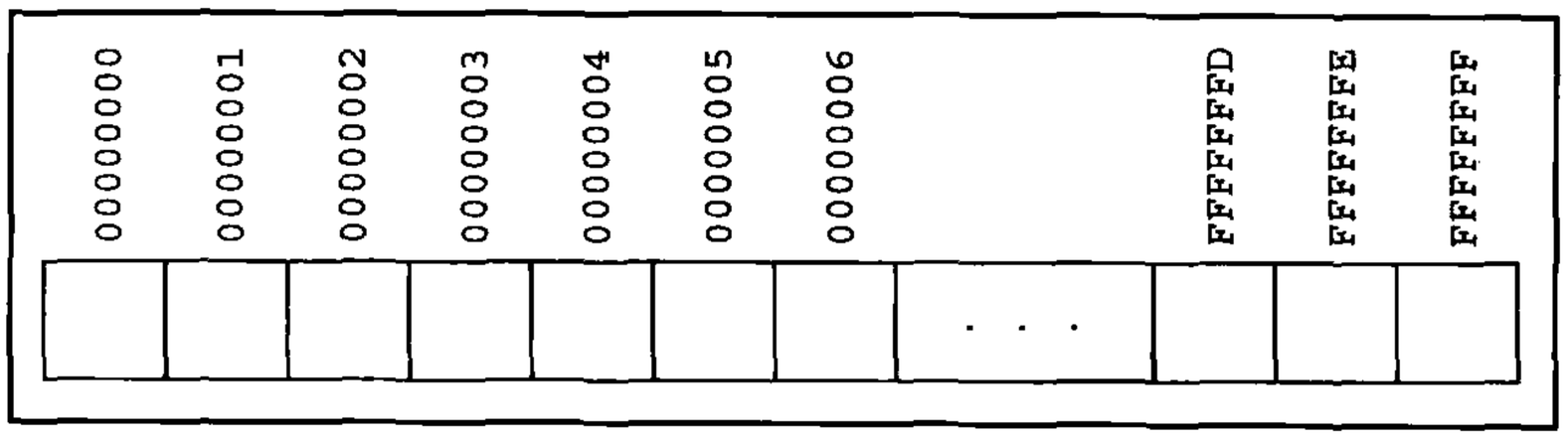


图 1-2 PC 存储器的逻辑图

在 80386 之前，Intel 80x86 处理器系列仅仅能够直接寻址的存储器为 2^{20} 字节，使用 20 位物理地址，通常用 5 个十六进制数表示，范围从 00000 ～ FFFFF。

本书中的汇编语言程序使用**平面存储**模式。这意味着，逻辑上指向存储数据和指令的存储单元的地址实际上是用 32 位的地址编码的。

Intel 80x86 体系结构还提供了**分段存储**模式，早期的 8086/8088 CPU 只用这种模式。在 8086 / 8088 中，PC 存储器可看作是段的集合，每个段是 64KB，以 16 的倍数作为一个段的开始地址。也就是说，如果一个段的起始地址是 00000，另一个段（重叠第一个段）的起始地址就是 16（00010_{16}），下一个段的起始地址是 32（00020_{16}），其他段的起始地址依此类推。一个段的**段号**由其物理地址的前 4 个十六进制数组成。8086/8088 微机中的程序并不能使用 5 位的十六进制数地址，事实上，每个存储单元的定位取决于段号和从该段开始处的一个 16 位的**偏移量**。通常，程序只写出偏移量，段号可通过上下文来推断。偏移量是指从段的第一个字节到要定位地址的距离。在十六进制中，偏移量大小从 0000 到 $FFFF_{16}$。

从 80386 开始，80x86 系列处理器既有 16 位也有 32 位的分段存储模式。段号仍是 16 位长，

但不直接指向存储器中的一个段。事实上，段号仅仅是包含真正 32 位段的起始地址的表中的索引。在 32 位分段模式中，32 位的偏移量加上该段的起始地址可得出内存操作数的实际地址。对编程人员而言，段逻辑上是很有用的：在 Intel 的分段模式下，编程人员通常为代码、数据和系统堆栈分配不同的内存段。80x86 平面存储模式是真正的 32 位分段模式，所有的段寄存器包含相同的值。

事实上，当程序执行时，由程序产生的 32 位地址不一定是某个操作数存储的物理地址，操作系统和 Intel 80x86 CPU 有另外的存储管理层。**分页**（paging）机制用于将程序的 32 位地址映射成物理地址，当程序所产生的逻辑地址超过计算机实际的物理内存空间时，分页机制就非常有用了。如果程序太大而不能全部装入物理内存时，分页机制也可用于将部分程序在需要的时候从磁盘交换到内存中。当用汇编语言编程时，分页机制对用户是透明的。

练习 1.2

1. 假定微机的 RAM 是 256MB，则最后一个字节的 8 位十六进制数的地址是多少？
2. 假定一个微机的视频适配器预定的 RAM 地址是从 000C0000 到 000C7FFF，则其存储空间的字节数是多少？

1.3 80x86 寄存器

早期的 8086/8088 CPU 能够执行 200 多种不同指令。随着 80x86 系列扩展到 80286、80386、80486 以及 Pentium 处理器，CPU 可执行更多的指令。本书探讨如何用这些指令执行程序，从而理解机器级的计算机的性能。其他生产商生产的 CPU 也执行基本相同的指令集，因此，在 80x86 上编写的程序在这些 CPU 上不用改变也可运行。

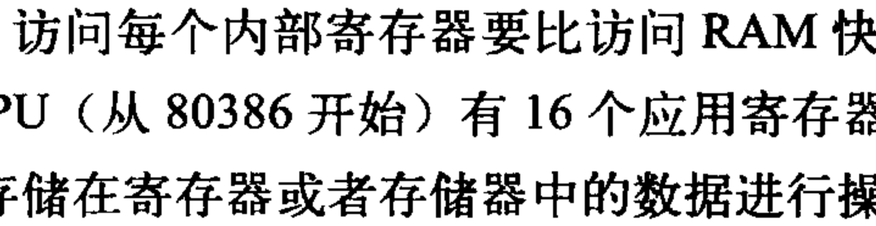
CPU 包含许多**寄存器**。访问每个内部寄存器要比访问 RAM 快得多。应用寄存器主要跟编程人员有关。一个 80x86 CPU（从 80386 开始）有 16 个应用寄存器。常用的指令可在寄存器与存储器间传输数据，也可对存储在寄存器或者存储器中的数据进行操作。所有的寄存器都有名字，一些寄存器有着特定的用途。下面将给出这些寄存器的名字，并详述它们的用途。

寄存器 EAX、EBX、ECX 和 EDX 称为**数据寄存器**或者**通用寄存器**。EAX 有时也是**累加器**，因为它用于存储许多计算的结果。下面是一条使用 EAX 寄存器的指令的例子：

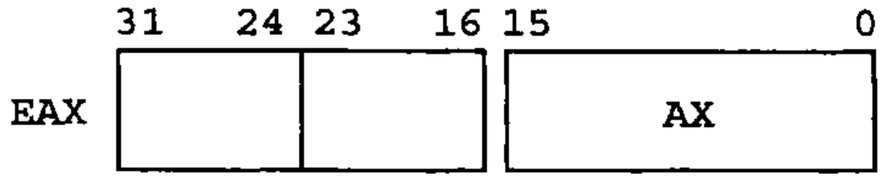
```
add eax , 158
```

将十进制数 158 与 EAX 中已有的数相加，用相加的和取代 EAX 中原来的数。（加法指令和下面提到的其他指令将在第 3 章详细讨论。）

寄存器 EAX、EBX、ECX 和 EDX 都是 32 位长，Intel 转换是以低位的 0 开始，从右向左，对数位转换。因此，如果把每个寄存器看成四个字节，则这些位用数表示为：



寄存器 EAX 整体上按照地址可分成若干部分。低位字位数从 0 ～ 15，就是常用的 AX 寄存器。

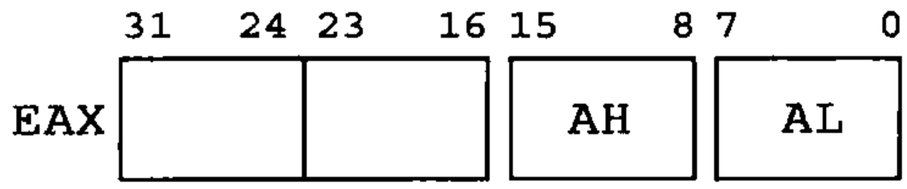


指令

```
sub   ax , 10
```

表示从存储在 AX 寄存器中的数中减去 10，而 EAX 寄存器的高位数（16 ～ 31）没有任何改变。

同样，AX 寄存器的低位字节（0 ～ 7 位）和高位字节（8 ～ 15 位）分别就是通常所说的 AL 和 AH。



指令

```
mov   ah , 02ah
```

复制 2A 到 8 ～ 15 位，不改变 EAX 的其他的任何位。这里的操作数是十六进制而不是十进制。

寄存器 EBX、ECX 和 EDX 也有低 16 位的 BX、CX 和 DX，它们又可按照高位和低位字节分别划分为 BH 和 BL、CH 和 CL、DH 和 DL。BH、BL、CH、CL、DH 和 DL 的每位的改变不会改变相应寄存器的其他位。要注意的是 EAX、EBX、ECX、EDX 的高位字并没有相应的名字——不能只使用名字引用 16 ～ 31 位。

8086 到 80286 处理器有 4 个 16 位的通用寄存器，称之为 AX、BX、CX 和 DX。增加“E”后表示“扩展”，将 16 位扩展成 32 位的 80386 寄存器。但是，80386 以及后来的体系结构都有效地包含了以前的 16 位的体系结构。

另外还有 4 个 32 位的通用寄存器：ESI、EDI、ESP 和 EBP。事实上，可以用这些寄存器做算术之类的操作，但通常必须保留它们，用于特定的用途。ESI 和 EDI 寄存器是**索引寄存器**，其中 SI 代表源索引，DI 代表目的索引。它们可用于实现数组索引，并且在某些字符串操作中必须使用，但本书不讨论这些内容。

ESP 寄存器是系统栈（内存中保留域）的**栈指针**，它很少直接通过程序来改变，但当数据入栈或者出栈时会改变。ESP 寄存器在堆栈的用途之一就是过程（子例程）调用。过程调用指令地址紧跟在存储于栈中的过程调用指令之后，当调用返回时，这条指令地址就可从堆栈中取出。第 5 章将会详细探讨堆栈以及栈指针寄存器。

EBP 寄存器是**基址寄存器**。通常，堆栈中被存取的数据项仅仅是存放在栈顶的数据项。然而，EBP 寄存器除了标识栈顶位置外，也经常用于标识栈中的某一个固定位置，因此，在这个固定位置附近的数据可被访问。EBP 也用于过程调用，尤其是带有参数的过程调用。

还有 6 个 16 位的**段寄存器**：CS、DS、ES、FS、GS 和 SS。在以前的 16 位分段存储器模式中，CS 寄存器包含有代码段的段号，即当前正在执行的指令所存储的存储器区域。由于一个段是 64K 长，一个程序的指令集通常在 64K 的范围内；如果一个程序长度超过 64K，则该程序在运行时，需要改变 CS 的值。同样，DS 包含数据段的段号，即数据存储在存储器中的区域。SS 寄存器包含有堆栈段的段号，即保留的栈。ES 寄存器包含有可用于乘法运算的附加数据段的段号。FS 和 GS 是 80386 增加的，它们便于访问两个附加数据段。

在平面 32 位存储器模式中，编程人员不太考虑段寄存器。操作系统给每个 CS、DS、ES 和 SS 相同的值。回想一下，这是一个表的入口指针，该表包含段的实际起始地址，也包含程序的大小。因此，当程序随意或者故意对另一个区域进行写操作时，操作系统可能提示错误。但是

这些对编程人员都没有什么关系，编程人员只根据 32 位地址来考虑。

32 位**指令指针**，或称为 EIP 寄存器，汇编语言的编程人员是不能直接对其进行访问的，CPU 必须从存储器中取出要执行的指令，并且，EIP 跟踪下一条待取指令的地址。如果是比较老式的、简单的计算机体系结构，下一条待取指令可能也是下一条将要执行的指令。事实上，80x86 CPU 在执行前一条指令时，它就取随后要执行的指令了。假设（通常是正确的）：下一次将执行的指令在存储器中是有序紧随（上一条指令）的。如果这种假设证明是错误的，例如，如果执行一个过程调用，则 CPU 取出存储的指令，设置 EIP 包含这个过程的偏移量，并且从新的地址取下一条指令。

另外，对于预取指令，80x86 CPU 在完成前一条指令的执行前，实际上它就开始执行这个预取指令了。**流水线**（pipelining）使用了这种方法，加快了处理器的有效速度。

最后的寄存器称之为**标志寄存器**（flag register），名为 EFLAGS 指的就是这种寄存器，但是指令中不使用这个助记符。32 位的一些位用于设置 80x86 处理器的某些特征，其他的位，称为**状态标志位**，表示指令执行的结果，常用的标志寄存器的 32 位中的有些位的名字在图 1-3 中给出。标志位的改变取决于所执行的指令。例如，第 6 位（零标志位 ZF）设定为 1，因为相加的和为 0。其他的标志将在下一节详细讨论。

位	助记符	作用
0	CF	进位标志位
2	PF	奇偶校验标志位
6	ZF	全零标志位
7	SF	符号标志位
11	OF	溢出标志位

图 1-3　部分 EFLAGS 位

总之，80x86 CPU 使用 16 个内部寄存器存储操作数和运算结果，并且跟踪段选择器和段地址，可执行很多指令，图 1-4 对这些寄存器的使用进行了总结。

名字	长度（位）	使用 / 说明
EAX	32	累加器，通用；低位字 16 位 AX，可分为 AH 和 AL
EBX	32	通用；低位字 16 位 BX，可分为 BH 和 BL
ECX	32	通用；低位字 16 位 CX，可分为 CH 和 CL
EDX	32	通用；低位字 16 位 DX，可分为 DH 和 DL
ESI	32	源索引；串复制源地址，数组索引
EDI	32	目的索引；目的地址，数组索引
ESP	32	栈指针；栈顶指针
EBP	32	基址指针；栈中引用位置的地址
CS	16	代码段选择器

图 1-4　80x86 寄存器

名字	长度（位）	使用 / 说明
DS	16	数据段选择器
ES	16	附加段选择器
SS	16	栈段选择器
FS	16	附加段选择器
GS	16	附加段选择器
EIP	32	指令指针；下一条待取指令的地址
EFLAGS	32	标志集；或者状态位

图 1-4 （续）

练习 1.3

1. 画图说明 ECX、CX、CH 和 CL 之间的关系。
2. 标志位 PF 是奇偶校验标志位，计算结果的低位字节的值如果为 1 表示计算结果是偶数，为 0 表示计算结果是奇数。如果计算结果的低字节是 $4E_{16}$，PF 是什么？

1.4 字符编码

字母、数字、标点符号等各种字符在计算机中都可用一个特定的数值来表示。字符编码方式有很多，微机中普遍采用的一种字符编码是美国信息交换标准代码（简称为 ASCII）。

ASCII 用 7 位表示字符，数值从 0000000 到 1111111，它有 128 种组合，可以表示 128 种字符。也可用十六进制数 00 ～ 7F 或者十进制数 0 ～ 127 表示[⊖]。附录 A 给出了 ASCII 的详细列表，在表中可以查到“Computers are fun .”用十六进制表示的 ASCII 码值：

43	6F	6D	70	75	74	65	72	73	20	61	72	65	20	66	75	6E	2E
C	o	m	p	u	t	e	r	s		a	r	e		f	u	n	.

注意：尽管空格字符不可见，但仍然有一个字符编码（十六进制数 20H）。

数字也可以用字符编码表示，例如用 ASCII 表示日期“October 23, 1970”：

4F	63	74	6F	62	65	72	20	32	33	2C	20	31	39	37	30
O	c	t	o	b	e	r		2	3	,		1	9	7	0

其中，日期中的数字字符 23 用 ASCII 码值 32 33 表示，1970 用 31 39 37 30 表示，这与 1.1 节所讲的二进制表示有所不同，上节中 $23_{10}=10111_2$，$1970_{10}=11110110010_2$。计算机使用这两种数字表示方法，其中 ASCII 表示法用于外设输入输出，二进制表示法用于计算机内部计算。

ASCII 的代码看起来似乎是任意指定的，但事实上是遵循某些规范的。大写字母的 ASCII

⊖ 许多计算机使用扩展字符集，它另外增加了十六进制字符 80 ～ FF（十进制 128 ～ 255）。本书不使用扩展字符集。

码值是相邻的，小写字母的 ASCII 码值也是如此。大写字母的编码与其相对应的小写字母的编码仅仅有一位不同，大写字母的第 5 位是 0，而小写字母的第 5 位是 1，其他各位都相同。（通常计算机用位来表示数时，从右到左，最右边的位以第 0 位开始。）例如，

大写字母 M 的编码为 $4D_{16}=1001101_2$

小写字母 m 的编码为 $6D_{16}=1101101_2$

打印输出字符从 20_{16} ～ $7E_{16}$。（空格字符也是打印输出字符。）数字 0，1，…，9 的 ASCII 值分别为 30_{16}，31_{16}，…，39_{16}。

ASCII 码值从 00_{16} 到 $1F_{16}$ 以及 $7F_{16}$ 都是**控制字符**，例如，ASCII 键盘上的 ESC 键的 ASCII 码值是 $1B_{16}$，简称 ESC，表示特殊服务控制，但经常被认为是“escape”的含义。ESC 字符经常与其他字符一起传给外部设备，比如，传给一台打印机，让它执行某种指定的操作。因为这样的字符序列没有标准化，所以本书将不作讨论。

CR 和 LF 是两个非常常见的 ASCII 控制字符，CR（$0D_{16}$）表示返回，LF（$0A_{16}$）表示换行。当按下 ASCII 键盘的 Return 或 Enter 键时，就会产生编码 $0D_{16}$，如果该编码送到 ASCII 显示器，则使光标移到当前行的开始处而不是到新的一行；如果该编码送到 ASCII 打印机，则会使打印头移到当前行的开始处。换行码 $0A_{16}$ 在 ASCII 显示器上会使光标垂直移到下一行或者使打印机将纸向上滚动一行。

使用较少的控制字符有“Form Feed”（$0C_{16}$），该字符使打印机换页；控制字符“Horizontal Tab”（09_{16}）在按下键盘的 Tab 键时生成；“Backspace”（08_{16}）在按下键盘的 Backspace 键时生成；“Delete”（$7F_{16}$）在按下键盘的 Delete 键时生成。注意：Delete 键和 Backspace 键生成的代码不同。响铃（Bell）字符（07_{16}）输出到显示器时会听见响铃声。

许多大型计算机用“扩展二进制编码一十进制信息编码”（简称 EBCDIC）。本书不讨论 EBCDIC 编码。

练习 1.4

1. 每个十六进制数可表示为一个十进制数或两个字符的 ASCII 值，请写出这两种表示。
   a. 2A45
   b. 7352
   c. 2036
   d. 106E
2. 写出下列字符串的 ASCII 值，不要忘记空格和标点符号。返回和换行字符用 CR 和 LF 表示，如果写在一起 CRLF（中间没有空格）表示回车换行功能。
   a. January1 is New Year’s Day. CRLF
   b. George said，“Ouch!”
   c. R2D2 was C3P0’s friend.CRLF[0 是数字 0]
   d. Your name? [问号后输入两个空格]
   e. Enter value: [冒号后输入两个空格]
3. 将下列 ASCII 序列输出到计算机显示器，会显示什么？
   a. 62 6C 6F 6F 64 2C 20 73 77 65 61 74 20 61 6E 64 20 74 65 61 72 73

b. 6E 61 6D 65 0D 0A 61 64 64 72 65 73 73 0D 0A 63 69 74 79 0D 0A

c. 4A 75 6E 65 20 31 31 2C 20 31 39 34 37 0D 0A

d. 24 33 38 39 2E 34 35

e. 49 44 23 3A 20 20 31 32 33 2D 34 35 2D 36 37 38 39

1.5 有符号整数的二进制补码表示

本节详细探讨计算机中数的表示。前面已经介绍了两种表示数的方法，一种是用二进制表示（通常用十六进制表示），另一种是用 ASCII 码表示，但是这两种表示法有两个问题：如何表示一个负数；表示数的有效位是有限的。

假定内存中的数用 ASCII 码存储，一个 ASCII 码通常用一个字节存储，而 ASCII 码长度是 7 位，附加位（左侧或最高位）置 0。为了解决上述表示法中的第一个问题，可在编码中包含一个负数符号。例如：四个字符的 −817 的 ASCII 编码表示就是 2D，38，31 和 37。为解决第二个问题，通常用固定长度的若干字节数，位数不足时，在 ASCII 码的左边用 0 或者空格补充；也可以使用一个长度可变的字节数，但必须规定要表示的数的最后一个数位以 ASCII 码结束，也就是说用一个非数字字符结束。通常，80x86 结构没有什么指令用来处理 ASCII 格式的数字，即使有这样的指令，也很少使用。

计算机内部使用二进制表示数时，80x86 和多数计算机选用固定长度的二进制数运算来解决长度问题，Intel 80x86 系列可用的长度有 8 位（1 字节）、16 位（1 **个字**）[⊖]、32 位（**双字**）和 64 位（**四字**）。

例如，697 的双字长二进制表示为：

$$697_{10} = 1010111001_2 = 00000000000000000000001010111001_2$$

二进制数表示的前面添加 0 是为了使长度是 32 位，该二进制数用十六进制数表示如下：

00	00	02	B9

前面介绍的表示法能够很好地表示非负数和**无符号**数，但不能表示负数。同时，对于任意给定长度都可表示一个最大无符号数，例如一个字节长度，表示的最大无符号数为 FF_{16} 或者 255_{10}。

二进制补码表示法与前面所讲的无符号数表示法很相似，但前者可以表示负数。二进制补码表示数时，应该指定其长度，所以可用“双字长的二进制补码表示法”表示一个数。二进制补码表示非负数与表示无符号数大致相同；也就是说，二进制补码表示数时，前面需要补充很多 0 来确保定长。只有一个限制：对于正数的表示有一个附加位，最左边的一位置 0。如：用字长二进制补码表示最大的正数就是 0111111111111111_2，即 $7FFFF_{16}$ 或者 32767_{10}。

如果根据正数的表示是最左边一位置 0，就推测负数的表示是将最左边一位置 1，其他各位跟对应正数的表示完全相同，那就大错特错了。因为负数的表示要比正数的表示复杂得多，所以不能简单地将最左边的位由 0 改为 1 来表示负数。

一个十六进制的计算器很容易将一个负的十进制数转换为二进制补码表示。例如，计算器

⊖ 其他计算机体系结构一个字的长度可能不是 16 位。

显示 −565，如果按“十六进制”转换键，计算器通常会显示 FFFFFFFDCB（有可能最前面 F 的个数不同）。如果用双字长表示，使用最后的 8 位十六进制数，则为：

FF	FF	FD	CB

或者二进制 1111 1111 1111 1111 1111 1101 1100 1011（注意：最高位用 1 表示负数）。只使用最后四个十六进制数，用字表示，则可表示为：

FD	CB

不用计算器也可实现负数的二进制补码表示。一种方法就是先用十六进制表示这个无符号数，然后用 100000000_{16} 减去这个十六进制数，得到该数的双字长二进制补码表示，十六进制的被减数由 1 后面紧跟表示长度所决定的若干个 0 组成。例如，10000_{16} 就是字长表示补码中的被减数。（字节长度的二进制补码表示的被减数是多少呢？四字长的二进制补码表示的被减数又是多少呢？）以二进制表示，被减数 0 的个数是二进制数位的长度。

示例

用字长的二进制补码表示法表示十进制数 − 76：首先转换无符号十进制数 76 为十六进制数 4C，然后用 10000 减去 4C。

```
  1 0 0 0 0
      - 4 C
```

因为 0 不够减 C，所以要从 1000 中借 1，剩下 FFF，即：

```
  F   F   F  10
      -   4   C
  ---------------
  F   F   B   4
```

借 1 之后做减法就容易多了，相应的数字变为：

$10_{16} - C_{16}=16_{10} - 12_{10}=4$（十进制或十六进制）

并且 $F_{16} - 4=15_{10} - 4_{10}=11_{10}=B_{16}$

如果已经了解十六进制数的加减法，就没有必要将十六进制数转换为十进制数后再做减法。

1 后面紧跟适当个数的 0 作为被减数，减去某个数的操作称为“**二进制取补**”。这个称谓既包含二进制表示的含义，又包含取补操作的含义，二进制取补操作相当于在十六进制计算器上按下符号转换键。

既然一个给定的二进制补码的表示是固定长度，那么，显然可以存储一个最大范围的数。给定的长度是一个字，则可存储的最大的正数是 7FFF，它是最大的 16 位长的正数，用二进制表示时最高位为 0。十六进制数 7FFF 也就是十进制数 32767。正数用十六进制表示的最高位是 0 到 7（用二进制表示最高位为 0），负数用十六进制表示的最高位为 8 到 F，（用二进制表示最高位是 1）。

怎样把一个二进制补码表示成相应的十进制数呢？首先，确定二进制补码的符号。将正的二进制补码数转换为十进制数，就像任何无符号二进制数转换成十进制数一样，可以手动完

成，也可用有十六进制功能的计算器转换。例如，一个字长的二进制补码数 0D43 表示十进制数 3395。

二进制数位的最高位是 1，即十六进制的 8 到 F 表示的二进制补码数，处理这类负数有些复杂。要注意的是，对一个数求二进制补码，得到结果，再对该结果求其二进制补码，就得到原数。对于长度为双字的数 *N*，通常的代数表达式为：

N=100000000 — (100000000 — *N*)

例如：双字长的二进制补码数 FFFFF39E

100000000 — (100000000 — FFFFF39E) =100000000 — C62=FFFFF39E

这再次说明二进制补码运算与取补运算一致。因此，对二进制位表示的负数求其二进制补码可得到对应的正数（无符号数）。

示例

字长的二进制补码数 E973 表示一个负数，因为符号位（最高位）是 1（E=1110）。取补码可得出相应的正数。

10000 — E973 = 168D =5773_{10}

这说明 E973 表示的十进制数为 −5773。

最高位为 1 的字长二进制补码的范围从 8000 ～ FFFF。转换为十进制数为：

10000 - 8000 = 8000 = 32768_{10}

因此，8000 表示的是 −32768。同样地，

10000 - FFFF = 1

因此，FFFF 表示的是 −1。由前面得知，字长的二进制补码表示的最大数为十进制正数 32767，因此，其表示的十进制数范围是 −32768 ～ 32767。

用计算器实现将二进制补码表示的负数转换为十进制数是件很棘手的事情。比如，以双字长表示 FFFFFF30，用计算器显示的是 10 位的十六进制数，因此，该数的十位十六进制数的序列为 FFFFFFFF30，前面附加了 6 个 F，然后按计算器的“十进制”转换按钮，计算器上显示的是 −208。

练习 1.5

1. 给出下列每个十进制数的双字长的二进制补码表示：
   a. 3874
   b. 1000000
   c. −100
   d. −55555
2. 给出下列每个十进制数的字长的二进制补码表示：
   a. 845
   b. 15000

c. 100

d. −10

e. −923

3. 给出下列每个十进制数的字节长的二进制补码表示：

a. 23

b. 111

c. −100

d. −55

4. 给出下列每个 32 位双字长的二进制补码数或无符号数所表示的十进制整数：

a. 00 00 F3 E1

b. FF FF FE 03

c. 98 C2 41 7D

d. FF FF FF 78

5. 给出下列每个 16 位字长的二进制补码数或无符号数所表示的十进制整数：

a. 00 A3

b. FF FE

c. 6F 20

d. B6 4A

6. 给出下列每个 8 位字节长的二进制补码数所表示的十进制整数：

a. E1

b. 7C

c. FF

d. 3E

7. a. 给出以字节长的二进制补码形式存储的有符号的十进制整数范围（最小到最大）。

b. 给出以字节形式存储的无符号数的十进制整数的范围。

8. a. 给出以字长的二进制补码形式存储的有符号的十进制整数范围（最小到最大）。

b. 给出以字长形式存储的无符号数的十进制整数的范围。

9. 本节介绍了如何用某个适当的 2 的幂的数作被减数来求一个数的二进制补码。还有一种方法就是以二进制表示该数（选用正确的数位作为表示长度），将每位的 0 变为 1，1 变为 0（也称对 1 取反），最后，对结果加 1（忽略进位）。这两种方法有异曲同工之妙。

1.6 整数的加减法

计算机普遍采用二进制补码表示法，其原因之一在于计算机常用二进制补码存储加减运算的有符号整数，无符号整数的加减法和有符号的加减法类似，这意味着 CPU 不需要增加额外的电路。本小节将讨论整数的加减运算，并介绍可用来确定运算结果是否正确的进位和溢出概念。

首先给出一些加法运算的例子。尽管在本书中 80x86 的指令常用到双字长操作数，为

了简单起见，例子中运用字长操作数。对于字节、字及双字长的操作数，其概念基本相同。80x86 结构系统对有符号数和无符号数使用相同的加法指令。给出的示例中的数都以字长度表示。

首先，0A07 和 01D3 相加。不管采用的是无符号数还是二进制补码表示，这两个数都是正数。右边表示十进制相加的算式。

```
  0A07          2567
+ 01D3        + 467
  0BDA          3034
```

由 $BAD_{16}=3034_{10}$ 可得，该计算结果是正确的。

然后，0206 与 FFB0 相加。显然，正数如同无符号数，但需用二进制补码的有符号数表示，0206 是正数，而 FFB0 为负数，这样就有两种十进制数相加的情况，第一种是有符号数的相加，第二种是无符号数的相加。

```
  0206          518           518
+ FFB0        + (-80)       + 65456
  101B6         438           65974
```

显然，运算结果不唯一。事实上，101B6 是十六进制数表示的 65974，但其不能用长度为 1 个字的无符号数表示（长度为字的无符号数能表示的最大正数为 65535）；如果用有符号数表示，并且忽略最左边的附加位 1，由 101B6 可得到 01B6，01B6 正是十进制数 438 的二进制补码。

现在 FFE7 与 FFF6 相加。如果用有符号数表示，则这两个数是负数。下面也给出有符号十进制数和无符号十进制数相加的表示。

```
  FFE7          (-25)         65511
+ FFF6        +(-10)        + 65526
  1FFDD         -35           131037
```

由于相加的结果值太大，不能用两个字节数表示，但是如果不考虑附加位 1，则 FFDD 就是 −35 的字长的二进制补码表示。

上面的例子中，最后两个加法运算都有一个向高位的**进位**，除去进位后，其他的数字位不是正确的无符号数的结果，但却是正确的二进制补码表示。即使对于有符号数，也不一定总能得到和的正确二进制补码表示。考虑下面两个正数的相加：

```
  483F          18495
+ 645A        + 25690
  AC99          44185
```

这两个加法运算中没有向高位的进位，但有符号数的表示却是错误的，因为 AC99 表示负数 −21351。直观上看，错误的原因在于，求得的和 44185 比用一个字即两个字节长度存储的最大有符号整数 32767 大（见 1.5 节练习 8）。但是，无符号数求和得到的结果却是正确的。

下面的两个有符号数表示的负数相加的例子，也会出现“错误”的结果。

```
  E9FF          (−5633)        59903
+ 8CF0        +(−29456)      +36080
 176EF          −35089         95983
```

两个数相加有一个进位，但除进位外的剩下四位数字 76EE 不是正确的有符号数的结果，因为 76EE 表示的是正数 30447，并且，−32768 是以字长存储的最小的负数，从直观上判断运算结果出错了。

上面例子出错的原因在于产生了**溢出**。计算机在做二进制加法时，硬件也可判断是否溢出，而且，如果没有溢出，计算机则认为得到的结果是正确的。事实上，计算机做二进制加法时，其运算过程逻辑上从右向左进行各位相加运算，与手动做十进制加法的过程很相似。计算机在逐位相加时，有时会向临近的左边一列产生进位“1”，这个进位会与左边的一列相加的结果一起求和，其他各列相加时依此类推。最特殊的一列是最左边的一列，即符号位，可能有进位到该位，也有可能该符号位进位到“附加位”。符号位进位输出（输出到“附加位”）即前面所谈到的“进位”，以附加的十六进制数 1 表示。图 1-5 给出了出现溢出和没有溢出的情况，从这可以总结出：向“附加位”的进位与向符号位的进位同时有或者同时没有时，不会发生溢出，否则发生溢出。

符号位是否有进位	符号位是否有进位输出	是否溢出
无	无	无
无	有	有
有	无	有
有	有	无

图 1-5　加法运算的溢出情况

下面以二进制形式再次给出上述加法的例子，进位写在两加数的上方。

```
                  111
  0000 1010 0000 0111          0A07
+ 0000 0001 1101 0011        + 01D3
  0000 1011 1101 1010          0BDA
```

该例没有向符号位（第 15 位）的进位，也没有符号位进位输出，所以没有溢出。对第 1、2 和 3 位的进位对溢出没有影响。

```
1 1111 11
  0000 0010 0000 0110          0206
+ 1111 1111 1011 0000        + FFB0
1 0000 0001 1011 0110          101B6
```

该例有向符号位的进位，并且有符号位进位输出，所以没有溢出。

```
1 1111 1111 11   11
  1111 1111 1110 0111          FFE7
+ 1111 1111 1111 0110        + FFF6
1 1111 1111 1101 1101          1FFDD
```

同样，该例既有向符号位的进位也有符号位的进位输出，所以也没有溢出。

```
  1          1111 11
  0100 1000 0011 1111          483F
+ 0110 0100 0101 1010        + 645A
  1010 1100 1001 1001          AC99
```

该例中，有向符号位的进位，但符号位没有进位输出，所以发生了溢出。

```
1    1    11 111
  1110 1001 1111 1111          E9FF
+ 1000 1100 1111 0000        + 8CF0
1 0111 0110 1110 1111        176EF
```

该例子因为没有向符号位的进位，但有符号位的进位输出，所以产生了溢出。

在计算机中，$a-b$ 这样的减法算式，通常是取 b 的二进制补码，然后将其与 a 相加，这就相当于 a 与 $-b$ 相加。例如：十进制减法 195−618 = −423，

```
  00C3
- 026A
```

可将 −026A 转换为加上 FD96，因为 FD96 是 026A 的二进制补码。

```
  00C3
+ FD96
  FE59
```

十六进制有符号数 FE59 表示十进制数 −432。观察上面的加法算式，有

```
       11      11
  0000 0000 1100 0011
+ 1111 1101 1001 0110
  1111 1110 0101 1001
```

注意，这个加法算式中没有进位，但是做减法时要借位。做减法 $a-b$ 时，如果无符号数 b 比 a 大则要**借位**。计算机硬件通过将减法运算转换为相应的加法运算，根据加法是否产生进位来判断减法是否要借位，如果加法运算没有进位，则对应的减法运算就需要借位，如果加法运算有进位，则对应的减法运算不需要借位（切记“进位”本身意味着“输出”）。

再举一个减法运算的例子：十进制数 985−411 = 574，用字长的二进制补码表示为：

```
  03D9
- 019B
```

可将减 019B 转换为加上 019B 的二进制补码 FE65。即：

```
                1 1111 1111 1       1
  03D9            0000 0011 1101 1001
+ FE65          +  1111 1110 0110 0101
  1023E         1 0000 0010 0011 1110
```

不考虑附加位 1，十六进制数 023E 表示十进制数 574。这个例子在做加法时有进位，所以相应的减法运算不需要借位。

减法运算也需要定义溢出。如果知道该运算结果已经超出了某种长度（如字长等）表示法所表示的范围，则可以人为断定这个运算结果出错了。而计算机通过对所做的加法运算进行分析来判断相应的减法运算是否产生了溢出。如果加法运算有溢出，则原始的减法运算也会有溢出；如果加法运算没有溢出，则原始的减法运算也不会有溢出。前面所列举的两个减法运算都没有溢出。如果用字长的二进制补码表示 −29123−15447，就会产生溢出。显然，正确的答案− 44570 已经超出了字长的二进制补码数所表示的范围 −32768 ～ 32767，而计算机硬件在做如下运算时

```
  8E3D
- 3C57
```

将减 3C57 计算转换为加上 3C57 的二进制补码 C3A9。

```
                1     1 11     111    1
  8E3D            1000 1110 0011 1101
+ C3A9          + 1100 0011 1010 1001
 151E6          1 0101 0001 1110 0110
```

由于符号位有进位输出，但没有向符号位的进位，所以产生了溢出。

本小节的例子是以字长的二进制补码来描述数的加减法运算，这种方法也适用于字节的二进制补码、双字的二进制补码或者其他长度的二进制补码的加减运算。

练习 1.6

完成下列字长的二进制补码数的运算。给出每个加减运算操作具体的和与差，并判断是否有溢出。对于加法运算，判断是否有进位，对于减法运算，判断是否有借位。将运算的结果转换成十进制数来核对所做的答案是否正确。

1. 003F + 02A4
2. 1B48 + 39E1
3. 6C34 + 5028
4. 7FFE + 0002
5. FF07 + 06BD
6. 2A44 + D9CC
7. FFE3 + FC70
8. FE00 + FD2D
9. FFF1 + 8005
10. 8AD0 + EC78
11. 9E58 − EBBC
12. EBBC − 9E58
13. EBBC − 791C
14. 791C − EBBC

1.7 本章小结

计算机用电子信号表示所有数据，这些数据可用二进制数（位）来表示，这种表示法可认为是数的二进制表示。数也可写成十进制、十六进制和二进制格式。

80x86 计算机在内存中存储数据和指令。逻辑上内存是一长串地址单元，每个地址单元可

以存储一个字节的信息。寄存器用来操作数据。

大多数计算机用 ASCII 码表示字符，每个字符包括不能打印的控制字符都有一个 ASCII 码值。

整数可用预定长度的二进制补码表示，如一个字节、一个字或双字，一个二进制表示可解释为无符号数或有符号数。

对于无符号数和二进制补码而言，加减运算是一样的，计算机硬件根据运算的情况判断是否有进位或溢出。对于无符号数的运算，通过进位可判断结果是否正确；对于有符号数的二进制补码的运算，通过溢出可判断结果是否正确。