# 计算机算数

10.1算术逻辑单元

10.2整数表示

符号-幅值表示法

补码表示法

数位扩展

定点表示法

10.3整数算数

取负

加减法

乘法

除法

10.4浮点表示法

原理

二进制浮点表示的IEEE标准

10.5浮点算数

浮点加法和减法

浮点乘法和除法

精度考虑

二进制浮点运算的IEEE标准

10.6关键术语、思考题和习题

**学习目标**

学习本章后，你应该能够：

* 理解数字不同表示方式（二进制格式）的差别，理解基本算术运算所使用的算法。
* 解释说明**二进制补码表示法**。
* 概述使用二进制补码进行基本算术运算的方法。
* 理解在**浮点数表示法**中有效数字、基数和指数的使用方法。
* 概述用于浮点数表示法的IEEE 754标准。
* 理解与浮点算法相关的一些主要概念，包括保护位、舍入、次正规数、下溢出和上溢出。

我们首先从算术逻辑单元（ALU）的概述开始讨论处理器。之后本章重点介绍了ALU最复杂的方面计算机算术。第11章将介绍简单逻辑和运算功能的数字逻辑实现，作为ALU一部分的逻辑函数将会在第12章介绍。

计算机算数通常对两种不同类型的数字类型，如整数和浮点，完成算术运算。在任何一种情况下，表示法的选择都是关键的设计问题。我们首先讨论数的表示法。然后再讨论算术运算。

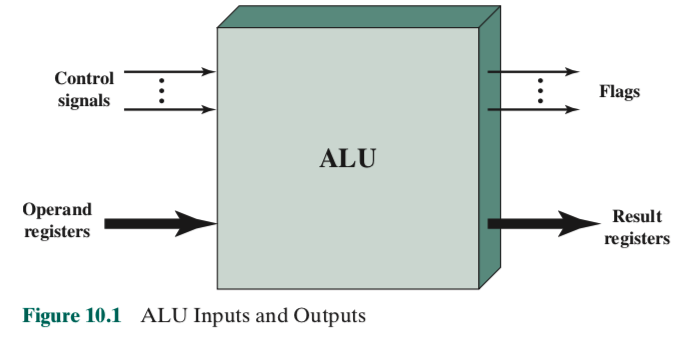
本章包括许多示例，每个示例在灰色方框中突出显示。

## 10.1算术逻辑单元

算数逻辑单元（ALU）是计算机中对数据实际执行算术和逻辑操作的部件。计算机系统的所有其他元件（控制单元、寄存器、存储器、I/O），主要是把数据传入ALU进行处理，然后把结果取出。在某种意义上，考察ALU涉及的是计算机的核心或本质。

算数逻辑单元，以及计算机中的所有电子部件都基于简单的数字逻辑设备的使用，这些设备可以存储二进制数字，并执行简单的布尔逻辑操作。

图10.1指出了ALU与处理器的其余部分互连的一般方式。算术和逻辑操作的运算符在寄存器中提交给ALU，运算结果存储在寄存器中。这些寄存器是处理器内的临时存储位置，通过信号路径连接到ALU（参见图2.3）。ALU也可以设置标记值作为操作的结果。例如，如果计算的结果超过要存储在其中的寄存器的长度，则溢出标志被设置为1。



标记值也存储在处理器内的寄存器中。处理器提供控制ALU的操作和数据进出ALU的信号。

## 10.2整数表示

在二进制数系统中1，可用数字0和1、负号（对于负数）、周期或**小数点**（对于具有小数分量的数字）来表示任意一个数。例如：

-1101.01012 = -13.312510

对于计算机存储和处理，负号和小数点的存在是不方便的。因为只有二进制数字（0和

1）可以用来表示数字。如果我们仅使用非负整数，那么表示起来就直截了当了。

一个8位的字可以表示从0到255的数字，例如：

 00000000 = 0

 00000001 = 1

 00101001 = 41

 10000000 = 128

 11111111 = 255

一般来说，如果一个n位二进制序列an-1an-2…a1a0表示一个无符号整数A，其值为：

A = Σ2iai

注释-------------------------------------------------------------------------------------------------

1有关数制系统（十进制、二进制、十六进制）的基本介绍，请参见第9章。

### 符号-幅值表示法

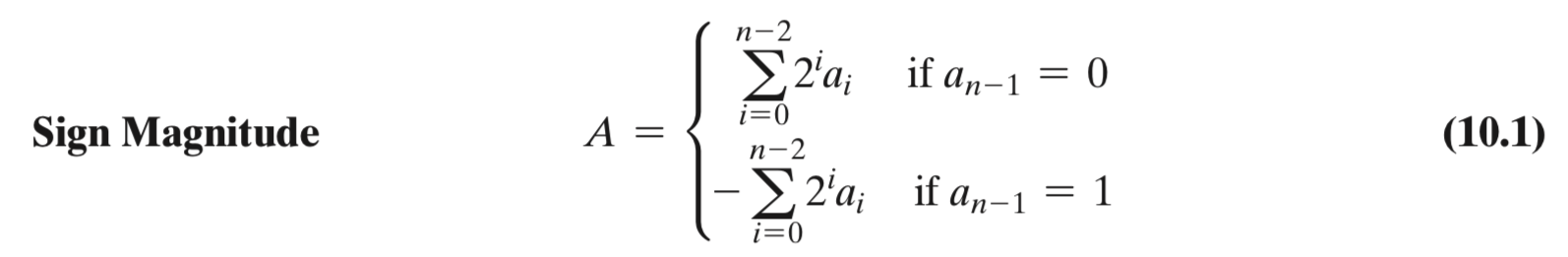
有几种可选的方式来表示负整数和正整数，这些表示方式都涉及将字的最高位（最左位）作为符号位处理。如果符号位为0，则数字为正；如果符号位为1，则数字为负。

采用符号位表示正负数的最简单表示法是符号-幅度表示法。以一个n位字为例，最右边的n-1位保存整数的幅值（绝对值，译者注）。

+18 = 00010010

-18 = 10010010（符号-幅值）

一般情况下，符号-幅值可以表示为：



符号-幅度表示法有几个缺点。一个缺点是加减运算需要同时考虑数字的符号和幅值大小，来能进行所要求的运算。在第10.3节的讨论中这点会变得更清楚。另一个缺点是，0有两种表示：

+ 010 = 00000000

- 010 = 10000000 （符号-幅值）

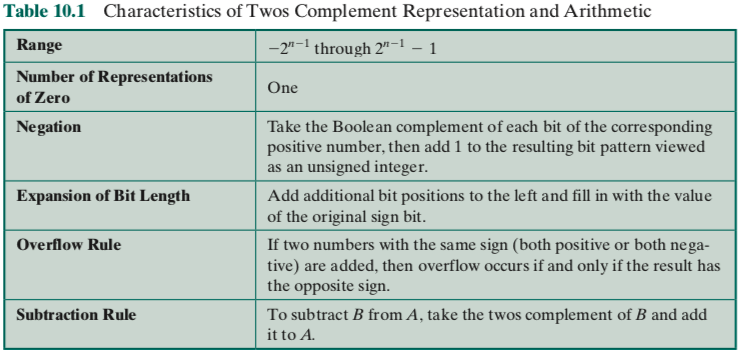
这样之所以不方便，是因为判断是否为0的操作（在计算机上频繁执行的一种操作）相对于只有一种0的表示法的情况下稍微困难一些。

由于这些缺点，符号-幅度表示法很少用于ALU的整数表示。最常用的方案是2的补码表示法2。

### 2的补码表示法

和符号-幅值表示法类似，2的补码表示法（twos complement representation）使用最高有效位作为符号位，使得很容易判断一个整数是正数还是负数。与符号-幅度表示法不同点在于其它位的解释方式。表10.1突出表示了2的补码表示法和算术的关键特征，这些将在本节和下一节中详细阐述。

大多数关于2的补码表示法的介绍都把重点放在了生成负数的规则上，但并没有正式给出这些规则可以成立的证明。

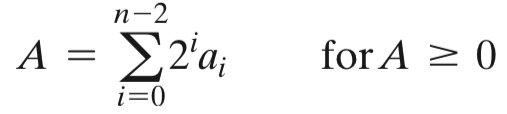


注释2-----------------------------------------------------------------------------------------------

在文献中，经常使用术语“2的补码表示”（two’s complement or 2’s complement）。这里，我们遵循标准文档中的使用时间，并省略撇号写作twos complement（例如，IEEE Std 100-1992，*The New IEEE Standard Dictionary of Electrical and Electronics Terms*）。

我们在本节和第10.3节中对2的补码表示的介绍是基于参考文献[DATT93]的，它建议，通过加权取和的方式来定义2的补码，正如我们之前对无符号和符号-幅度法所做的那样，这样有利于更好地理解2的补码表示法。这种解释的优点是，它可以消除2的补码表示算术规则可能不适用于某些特殊情况任何疑虑。

考虑使用2的补码表示法来表示一个n位整数A。如果A是正数，那么符号位an-1是零。其余的比特位表示此数的幅值，这和符号-幅值表示法一样，因此有：



数字0被认为是正数，因此具有符号位0和所位为0的幅值。我们可以看出，正整数的表示范围是从0（所有幅值位都是0）到2n-1-1（所有幅值位都是1）。表示再大的数字需要更多的位。

现在，对于一个负数A（A < 0），符号位an-1是1。剩余的n-1位可以取2n-1个值中的任意一个。因此，可以表示的负整数的范围是从-1到-2n-1。我们希望使用一种合适的方法指派负整数的位值，使得我们可以像无符号整数运算那样直截了当处理运算。在无符号整数表示中，为了从位表示中计算整数的值，最高有效位的权值是+2n-1。对于一个有符号位的表示法，我们将会在第10.3节中看到，如果最高有效位的权值是-2n-1，就会满足上文要求的算术运算性质。这是在2的补码表示法中使用的约定，这个约定产生的负数表达式如下所示：



等式（10.2）定义了正负数的2的补码表示法。如果an-1＝0，则-2n-1an-1 ＝0，这样的表达式定义了一个非负整数。

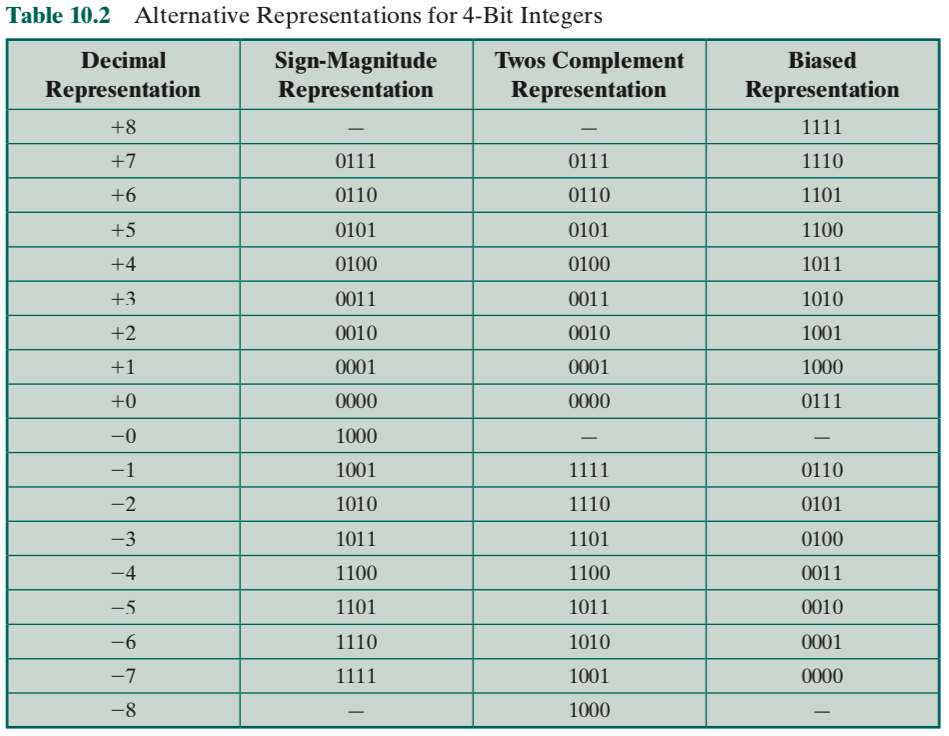
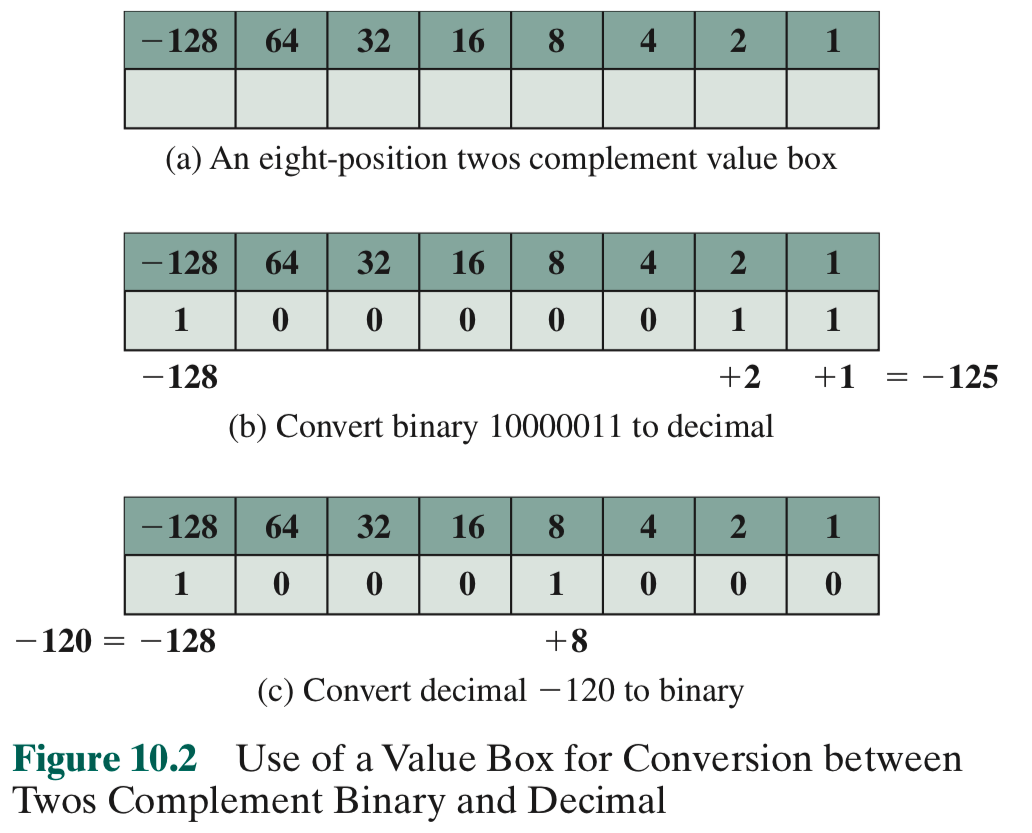


表10.2比较了4位整数的符号-幅度表示法和2的补码表示法。虽然2的补码表示法看似与人们的习惯有些别扭。但我们将看到，它有利于最重要的算术运算，加法和减法。正是由于这个原因，几乎所有的处理器都是采用此法来表示整数的。

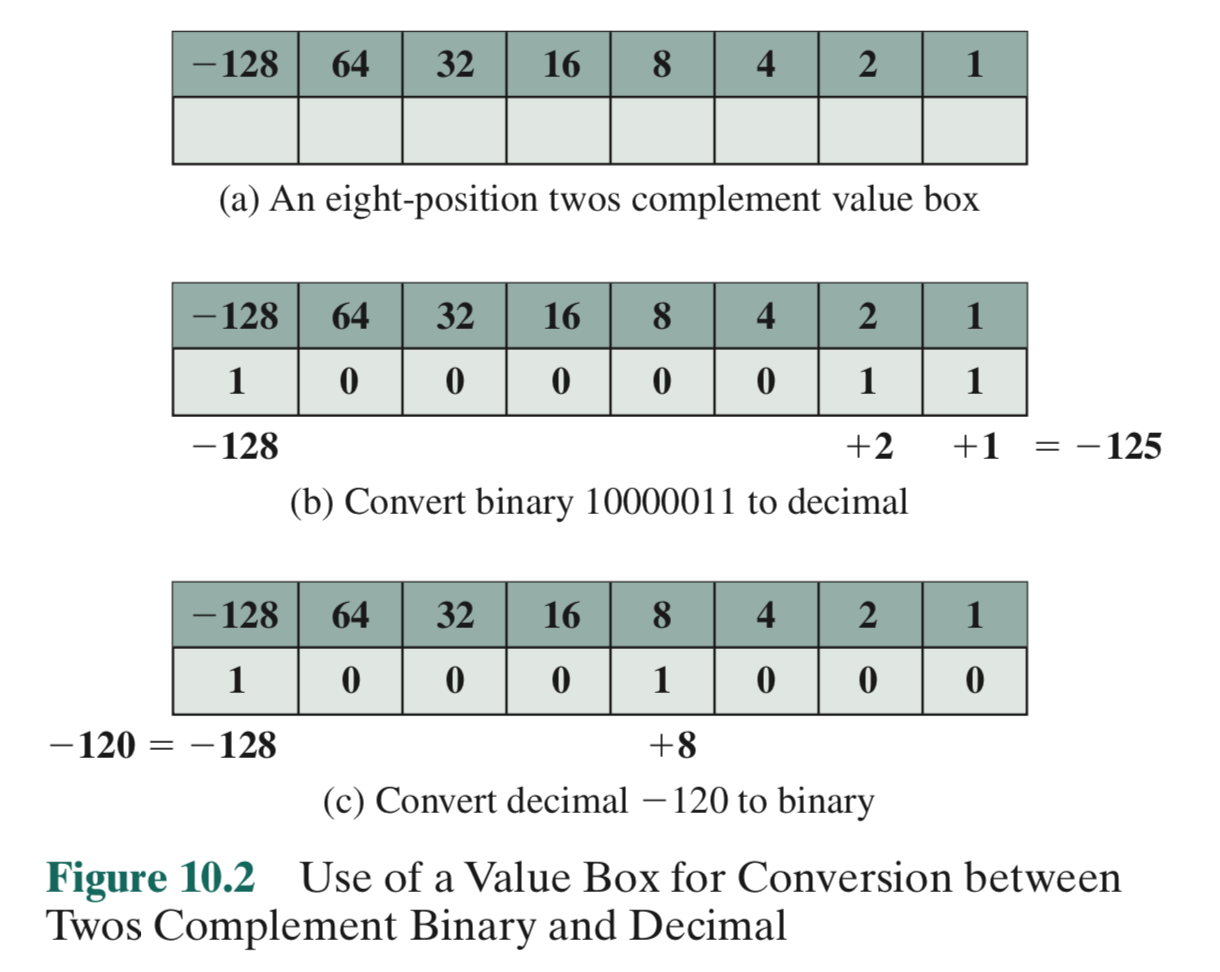
值盒子（value box）是说明2的补码表示法的一个很有用的方法，其中盒子最右端的值是1（20），往左每一个临接的位置其值加倍，直到最左边的位置，最左边的值是负的。如图10.2a所示，它能以2的补码形式表示的最小负数是-2n-1；如果除符号位以外其他位有1，则表示要把对应的正数加到最小负数-2n-1上。另外，很显然负数的最左位必定是1，正数的最左位必定是0。因此，最大的正数是0开头，后面跟着全部为1的数，其等于2n-1-1。

图10.2的其余部分说明了使用值盒子将2的补码转换为十进制，以及将十进制转换为2的补码的方法。



### 数位扩展

有时取一个n位整数需要将其存储在m位中，其中m > n。这种位长度的扩展称为**数位扩展**，因为可以通过增加位长度来扩展可表示的数字范围。



在符号-幅值表示法中，这很容易实现：只需将符号位移动到最左边的新位置上，并用零填充多余出的空位即可，例如：

+18 = 00010010 （符号大小，8位）

+18 = 0000000000010010 （符号大小，16位）

-18 = 10010010 （符号大小，8位）

-18 = 1000000000010010 （符号大小，16位）

这种做法对于2的补码的负数则不行。使用同一个例子，

+18 = 00010010 （2的补码，8位）

+18 = 0000000000010010 （2的补码，16位）

-18 = 11101110 （2的补码，8位）

- 32,658 = 1000000001101110 （2的补码，16位）

倒数第2行使用图10.2中的值盒子可以很容易地判断。最后一行可以使用公式（10.2）或16位值盒子来检验。

因此，2的补码整数的扩展规则是，将符号位移动到新的最左边位置，空出的位均以符号位的值填充。即对于正数，用零填充；对于负数，用1填充。这种方式叫做符号扩展（sign extension）。

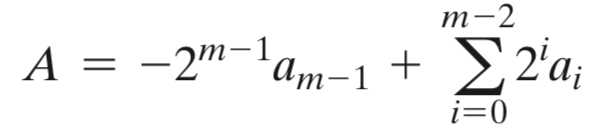
-18 = 11101110 （2的补码，8位）

-18 = 1111111111101110 （2的补码，16位）

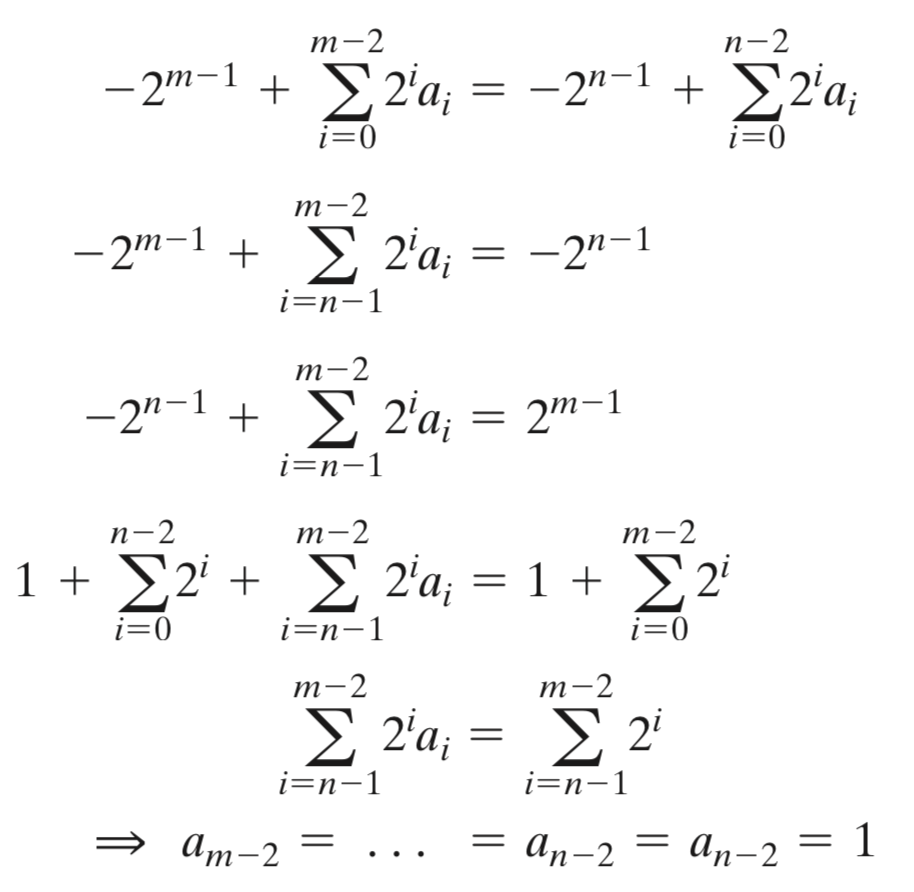
为了理解这个规则为什么可以工作，让我们再考虑一个n位的二进制数字序列an-1an-2…a1a0，当作一个2的补码表示的整数A，那么A的值是：



如果A是正数，则规则肯定是正确的。现在假设A是负数，我们想构成一个m位表示，其中m> n，则有：



这两个值必须相等：



从上面第一个等式走到第二个等式，我们要求最低的n-1位在两种表示中保持不变。对于倒数第二行，只有从第n-1位到第m-2位上的值都为1，等式才成立。因此，规则是正确的。研究下第10.3节开头的2的补码取负的讨论，你可能更容易理解这个符号扩展的规则。

### 定点表示法

最后需要指出，我们在本节中讨论的表示法有时被称为定点（fixed point）表示法。这是因为小数点（二进制小数点）是固定的，并且假定为在最低位数字的右边。程序员可以使用此方法来表示二进制小数，方法是适当降低数量级，以便使小数点隐含地设置在某个其他位置上。

## 10.3整数算数

本节考察2的补码表示的数的常用算术功能。

### 取负

在符号-幅值表示法中，求一个整数的负数的规则很简单：只需符号位取反。在2的补码表示法中，求一个整数的负数可用如下规则：

1. 将整数的每一位（包括符号位）取布尔反。也就是说，把每一个1变为0，每个0变为1。
2. 将结果作为一个无符号二进制整数对待，加1。

上述两步过程被称为求补运算(twos complement operation)，或者求整数的2的补。例如：

+18 = 00010010（2的补码）

按位取反＝11101101

+ 1

11101110 = - 18

对这个数字再取负，则正如所预料的，取负再取负就是原数本身：

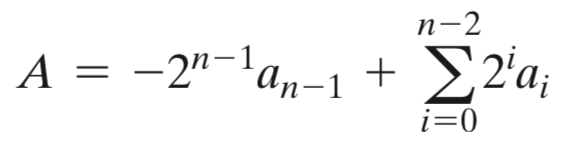
-18 = 11101110（2的补码）

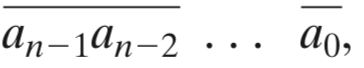
按位取反＝00010001

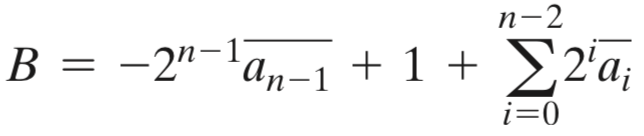
+ 1

00010010 = +18

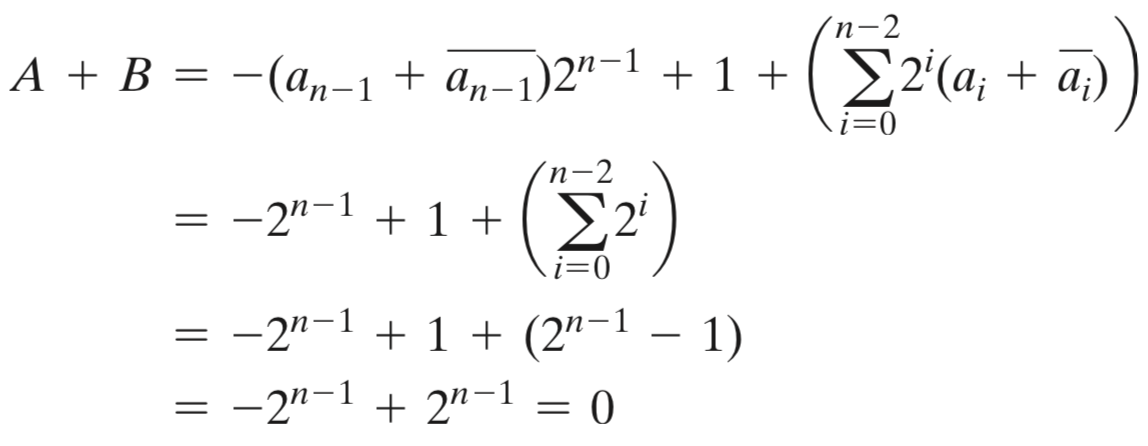
我们可以用方程（10.2）中2的补码表示式来证明刚才描述的操作的有效性。用n位二进制数字序列an-1an-2…a1a0来表示一个2的补码整数A，它的值是：



现在构造按位取反的位串，并将其视为无符号整数，加上1。最后，将得到的n位二进制数字当做一个2的补码表示的整数B。于是，B的值是：



现在，我们期望等式A = -B成立，即A + B = 0。这很容易验证是正确的。



前面的推导过程假定，我们可以将A取反后作为无符号整数处理，以便完成加1的操作，然后将结果作为一个 处理。有两个特殊的情况需要考虑。第一种情况，考虑A = 0。此时，对于一个8位的表示：

0 = 00000000（2的补码）

按位取反＝11111111

+ 1

100000000 = 0

上述算式加1时，有一个从最高位发出的进位(carry)，可以忽略该进位。结果是0求负还是0，正如期望的那样。

第二个特殊情况更是个问题。如果我们对位串1后面跟着的n-1个0取负，我们得到的是原来的数字。例如，对于一个8位的字(word)：

+128 = 10000000（两补）

按位取反＝01111111

+ 1

10000000 = 128

这样的一些意外是不可避免的。n位字可以表示的不同位串数目是2n，这是一个偶数。我们希望用这些位串来表示正负整数和0。如果能表示的正负整数数量一样（符号-幅值），那么0就会有两种表示。如果0只有一种表示（2的补码），那么能表示的正数和负数的数目必然不一样。在2的补码表示法中，一个n位长度的位串可以表示表示-2n-1，但不能表示+2n-1。

### 加法和减法

图10.3中表示了2的补码的加法，加法执行过程就和无符号整数加法一样。前四个例子显示的是成功的操作。如果运算的结果是正的，我们得到的2的补码表示会和无符号整数表示一样。如果运算结果为负，那么会得到负数的2的补码形式。注意，在某些情况下，最高位会产生一个进位，它超出了字的长度（图中以阴影表示），将被丢弃而忽略不计。

对于任意加法操作，如果出现结果的长度大于所使用的字的长度。这种情况称为**溢出**。当发生溢出时，ALU必须发出信号指出这个事实，以通知其他部件不要试图使用此结果。判断溢出的规则如下所示：

**溢出规则**：如果两个数字相加，并且它们同是正数或同是负数，则当且仅当结果的符号变为相反时才出现溢出。

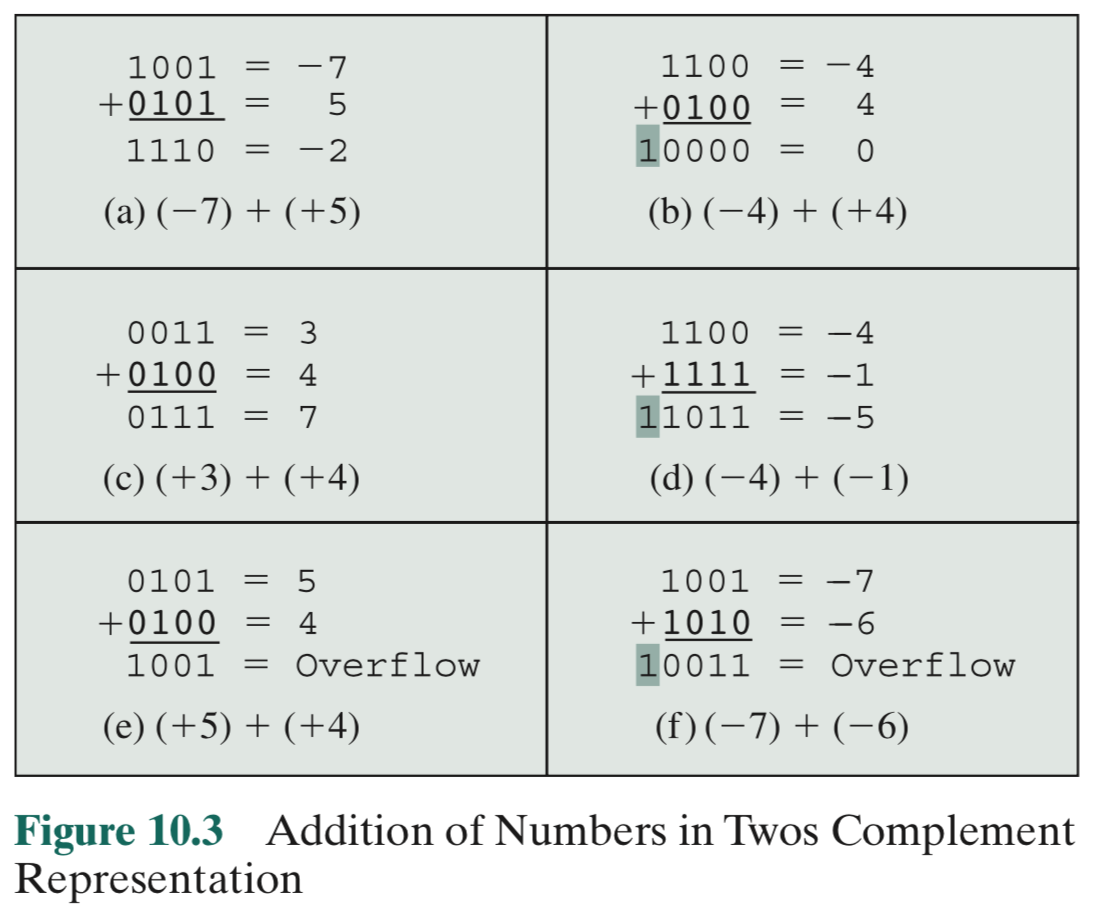
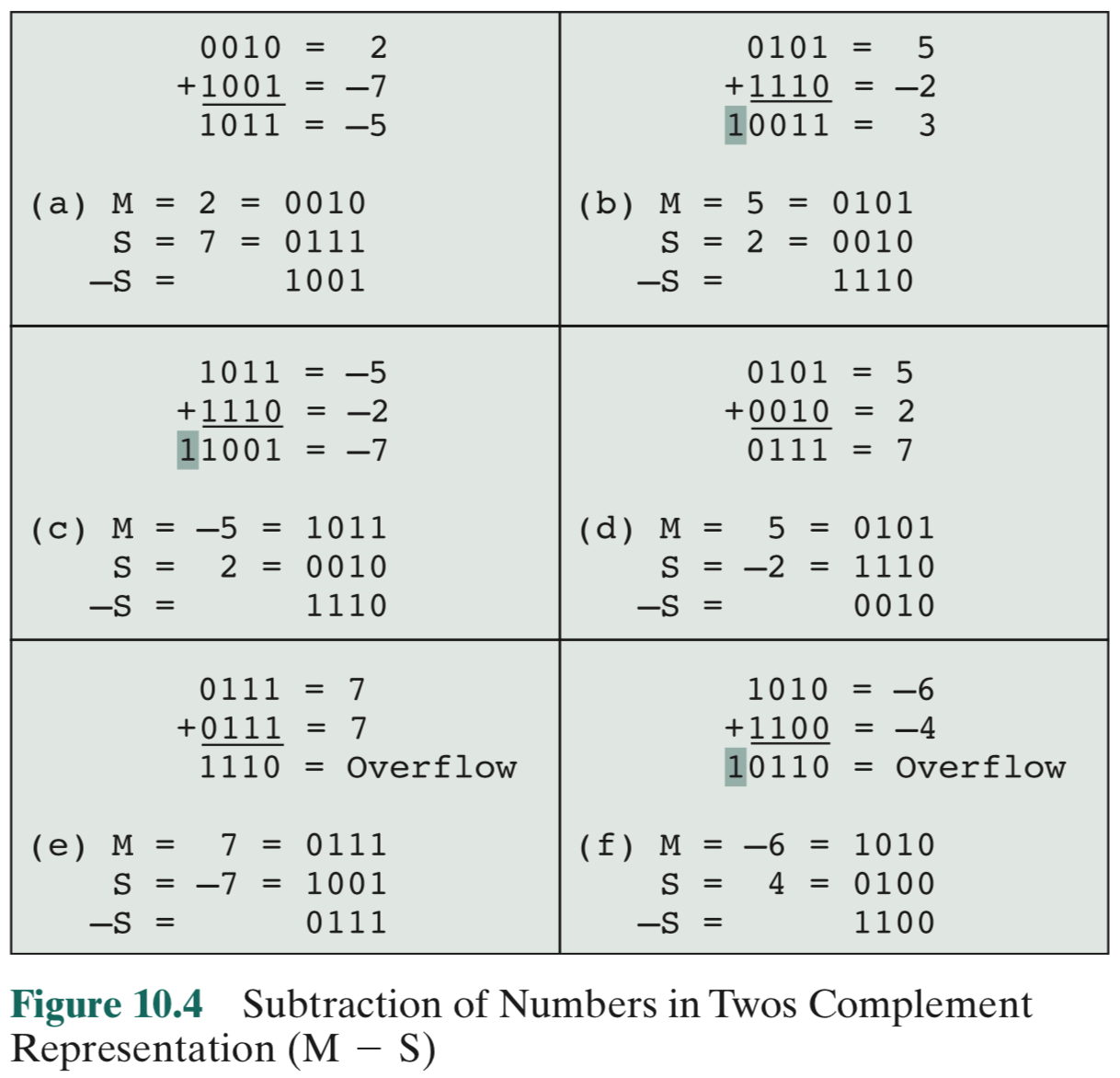


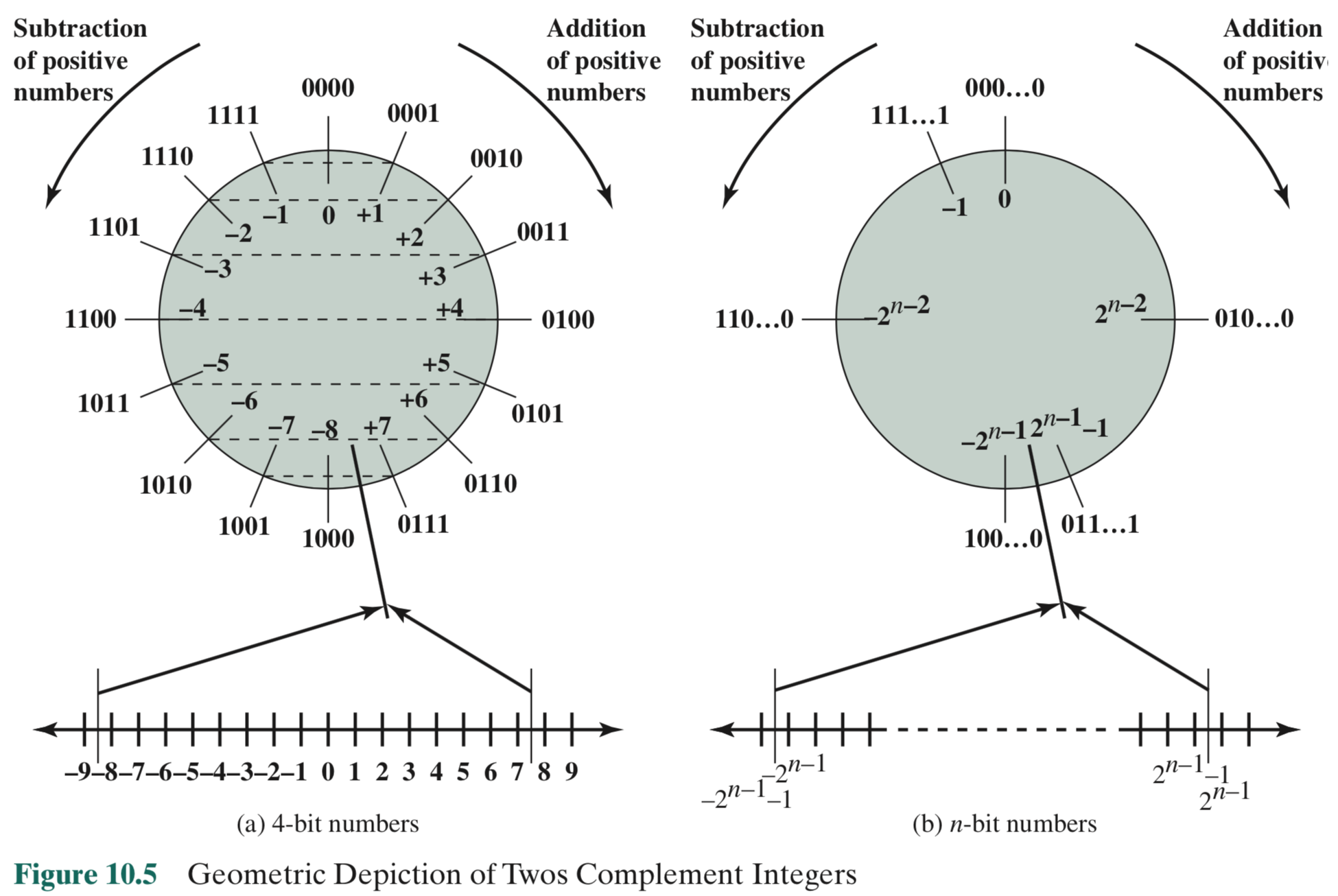
图10.3e和f显示的是溢出的例子。注意，不管是否存在进位，都可能发生溢出。

减法使用如下的规则可以很容易地处理溢出：

**减法规则**：若由一个数（被减数）减去另一个数（减数），则只需求出减数的2的补（取负），并把它加到被减数上。

因此，减法可以使用加法实现的，如图10.4所示。该图的最后两个例子说明了溢出规则仍然适用。

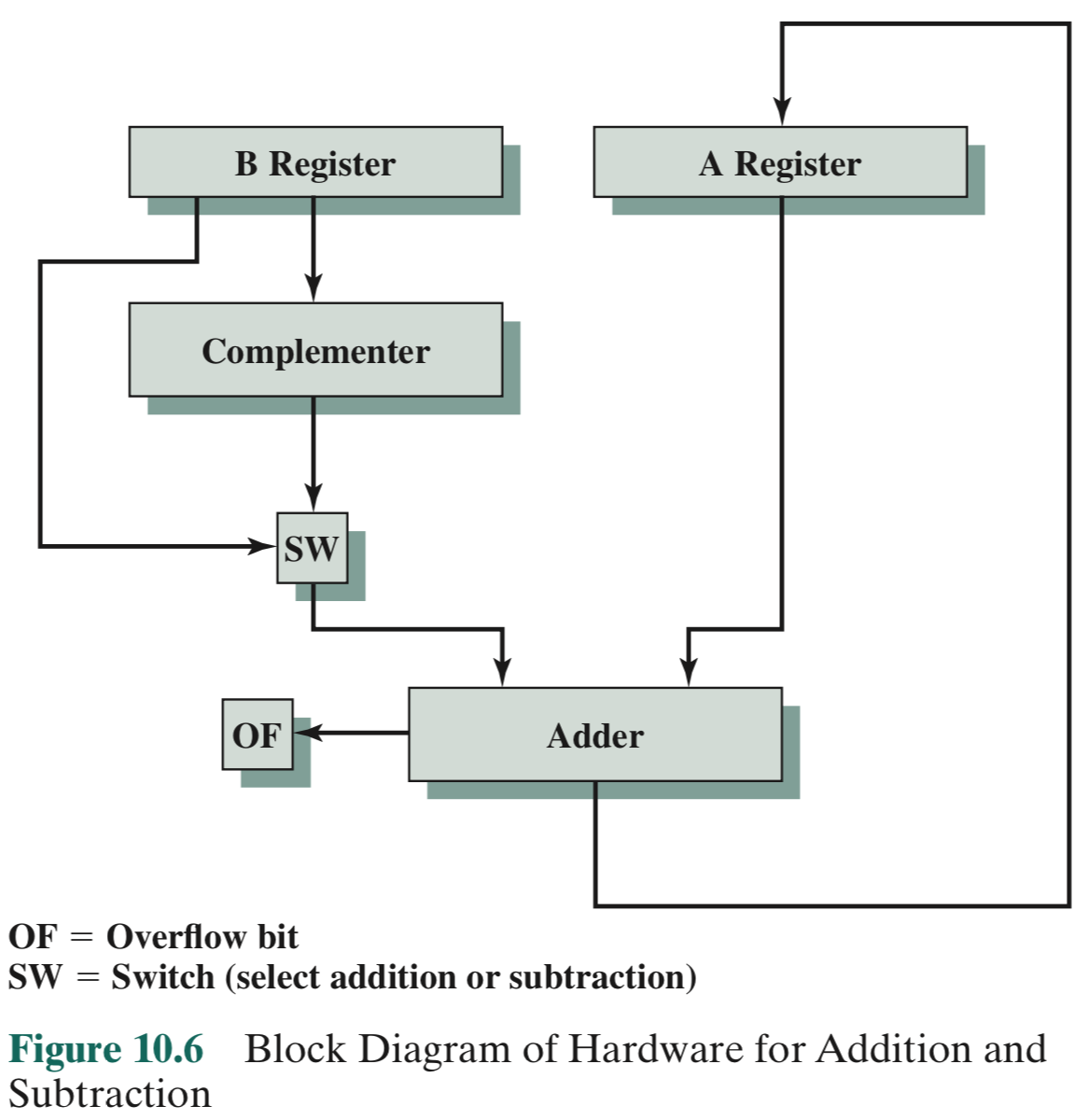




通过考察图10.5所示的几何描述法[BENH92]，可以加深对2的补码加减法的理解。 图中上部分的两个圆是通过将相应的数轴段以端到端的方式连接而构成的。注意，当数字被放置在一个圆上时，任一数求补数是其在水平方向上相对的那个数（由水平破折线指示）。从圆上的任意数字开始，我们通过顺时针移动k个位置来表示加正数k（或减负数k），逆时针方向移动k个位置来表示减正数k（或加负数k）。如果算术操作导致经过端到端的连接点，则给出一个不正确的答案（溢出）。

图10.3和10.4中的**所有**示例很容易使用图10.5的圆来描绘。

图10.6给出了实现加法和减法所需的数据通路和硬件元件。中心元件是一个二进制加法器，它送两个数字用于加法并产生一个求和结果和一个溢出指示。二进制加法器将两个数字视为无符号数（加法器的数字逻辑实现将在第11章给出）。对于加法，提交给加法器的两个数来自寄存器，在本例中指的是A和B寄存器。结果可以存储在这些寄存器中的一个或者另外的第三个寄存器中。溢出指示存储在一个1位的溢出标志中（0=无溢出；1=溢出）。对于减法运算，减数（B寄存器）要通过一个2的补码求补器（complementer），以便产生减数2的补提交给加法器。请注意，图10.6只显示了数据通路，



根据当前操作是加法还是减法，需要一些控制信号来控制是否使用求补器。

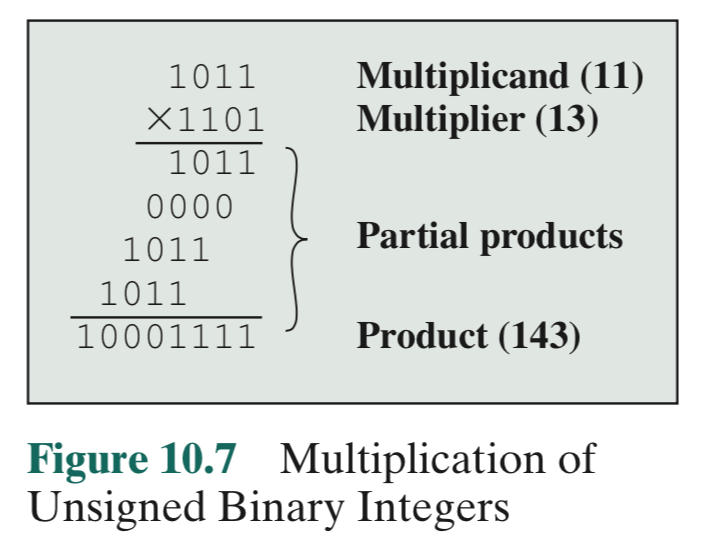
### 乘法

与加减法相比，无论是以硬件还是软件来完成，乘法都比较复杂。各种各样的算法已经用于各类计算机中。本小节的目的是给读者某些关于常用乘法算法的感性认识。我们先介绍如何实现两个无符号（非负）整数相乘的简单问题，然后再研究实现两个2的补码表示数乘法的最常见技术。

1. **无符号整数乘法**

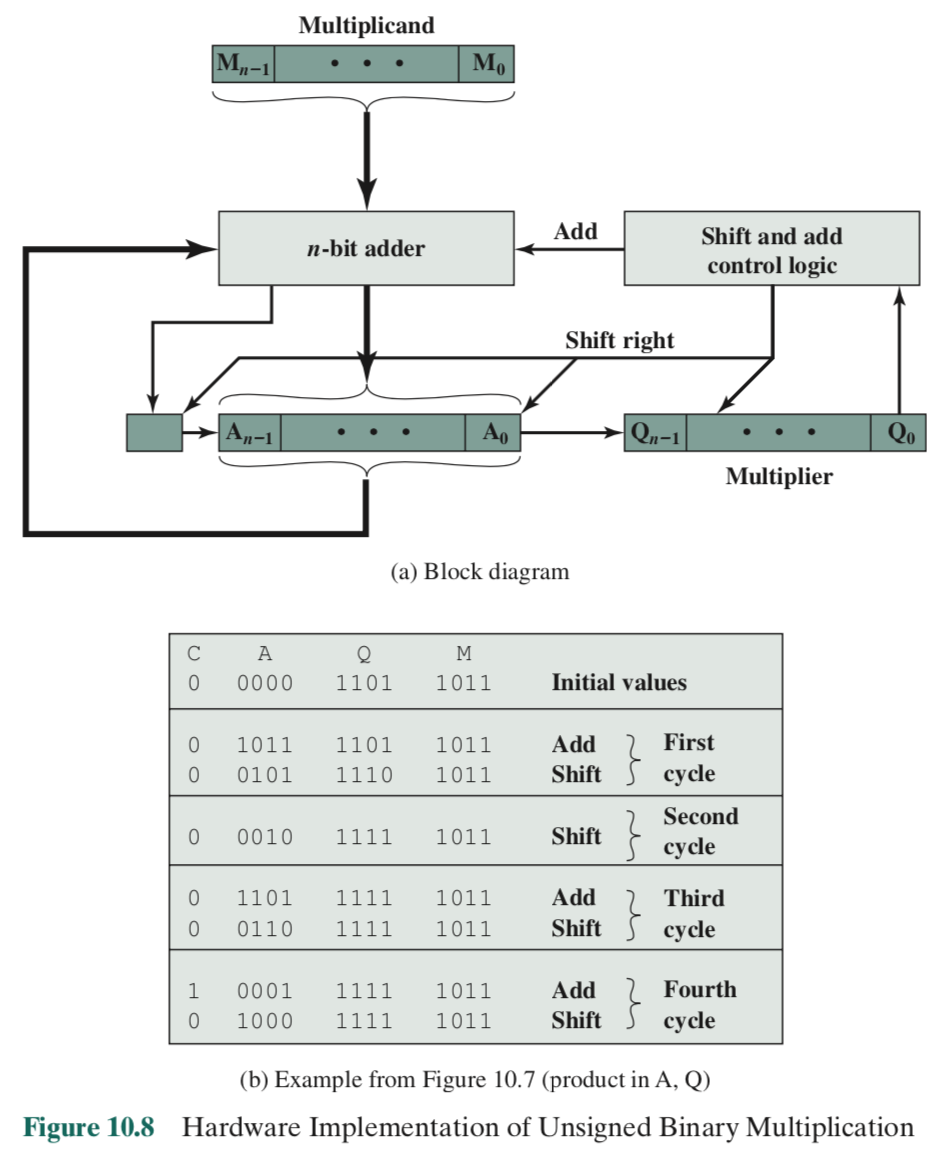
图10.7说明了无符号二进制整数的乘法，就像我们用纸和笔手工演算的那样。由此可以得出几点重要发现：

1. 乘法涉及部分积的生成，乘数的每一位对应一个部分积。然后将这些部分积相加得到最后的乘积。
2. 部分积是容易确定的。当乘数的位为0时，其部分积也为0。当乘数的位是1时，其部分积是被乘数。
3. 部分乘积通过求和而得到最后乘积。因此，后面的部分积总要比它前面的部分积左移一个位置。
4. 两个n位二进制整数的乘法可以产生最大长度为2n位的积（例如，11\*11=1001）。



与纸笔手工演算法相比，我们可以做几个改进来使计算机乘法操作更有效。首先，我们可以边产生部分积边做加法，而不是等到最后再相加。这样就消除了存储所有部分积的需求，减少了寄存器的使用数目。第二，我们可以节省某些部分积的生成时间，对于乘数的每个1，需要执行加法和移位两个操作；但是对于每个0，只需要执行移位操作即可。

图10.8a表示了一种使用上述改进的实现方案。乘数和被乘数被装入两个寄存器（Q和M）。还需



要第三个寄存器，即寄存器A，初始化设置为0。还需要一个1位寄存器C，初始化为0，用于保存加法可能产生的进位。

乘法器的操作如下。控制逻辑每次读取乘数的一位。如果Q0是1，则将被乘数与A寄存器相加，并将结果存储在A寄存器中，同时C位用于表示溢出。然后，C、A和Q寄存器的所有位都向右移一位，于是C位进入An-1，A0进入Qn-1而Q0丢失。如果Q0是0，则不执行加法，只需移位。对原始乘数的每一位重复上述过程。得到的2n位乘积存储在A和Q寄存器中。图10.9显示了该操作的流程图，图10.8b给出了一个示例。注意，在第二个周期中，当乘数位为0时，不执行加法运算。

1. **2的补码乘法**

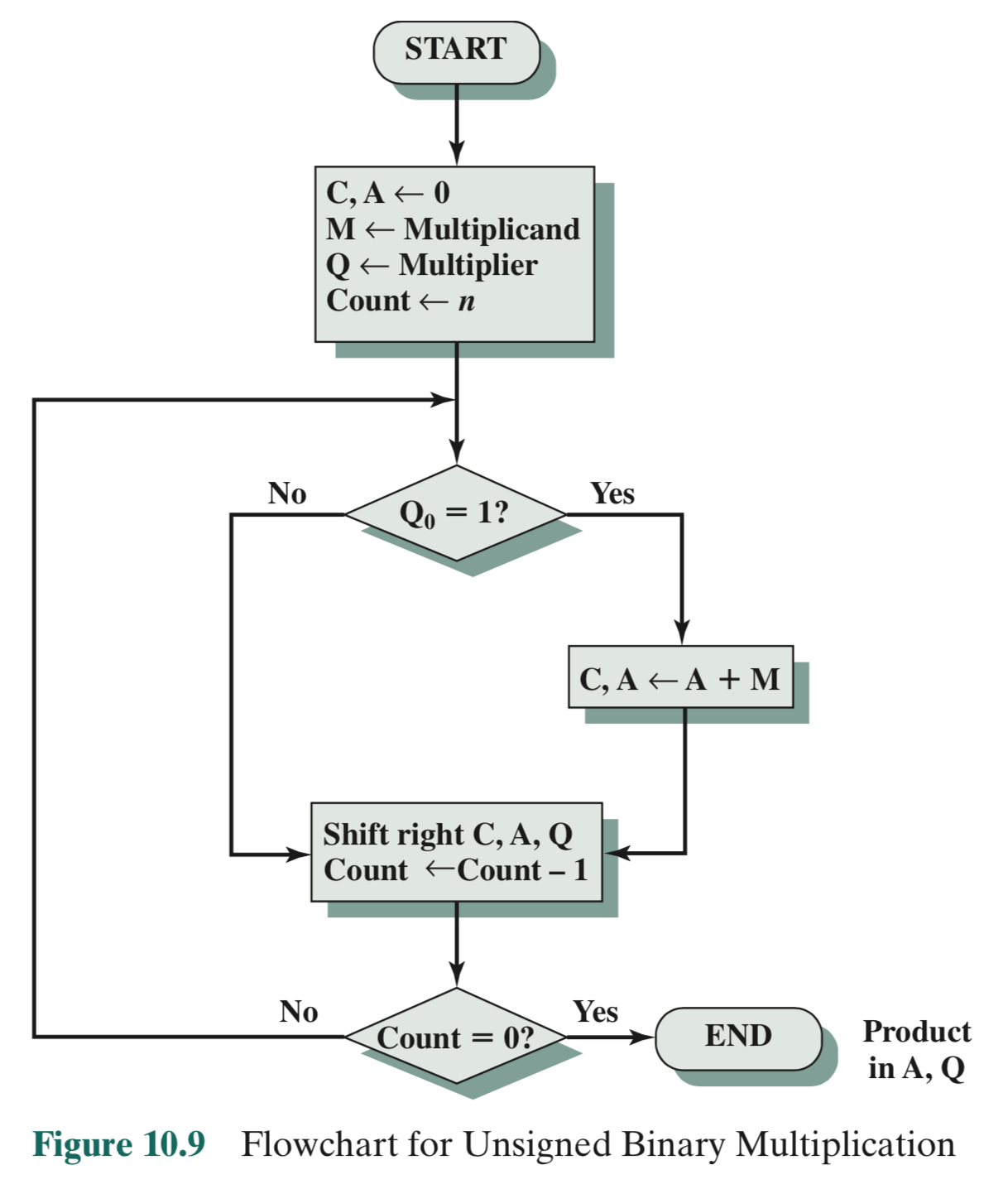
我们已经看到，通过把两个2的补码表示的数当作无符号整数看待，可以对它们进行加减运算。现在考虑：

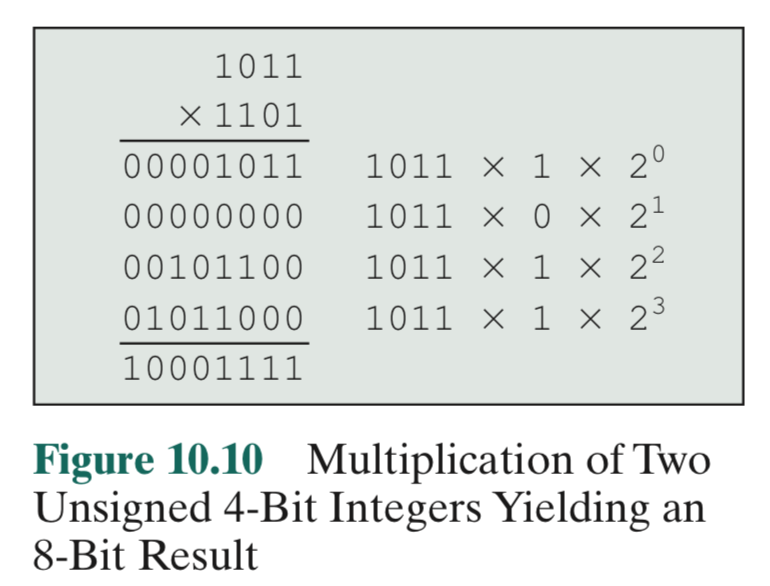
1001

+ 0011

1100

如果将这些数字看成是无符号整数，则是9（1001）加3（0011）得到12（1100）。若是看作是2的补码整数，则是-7（1001）加3（0011）得到-4（1100）。





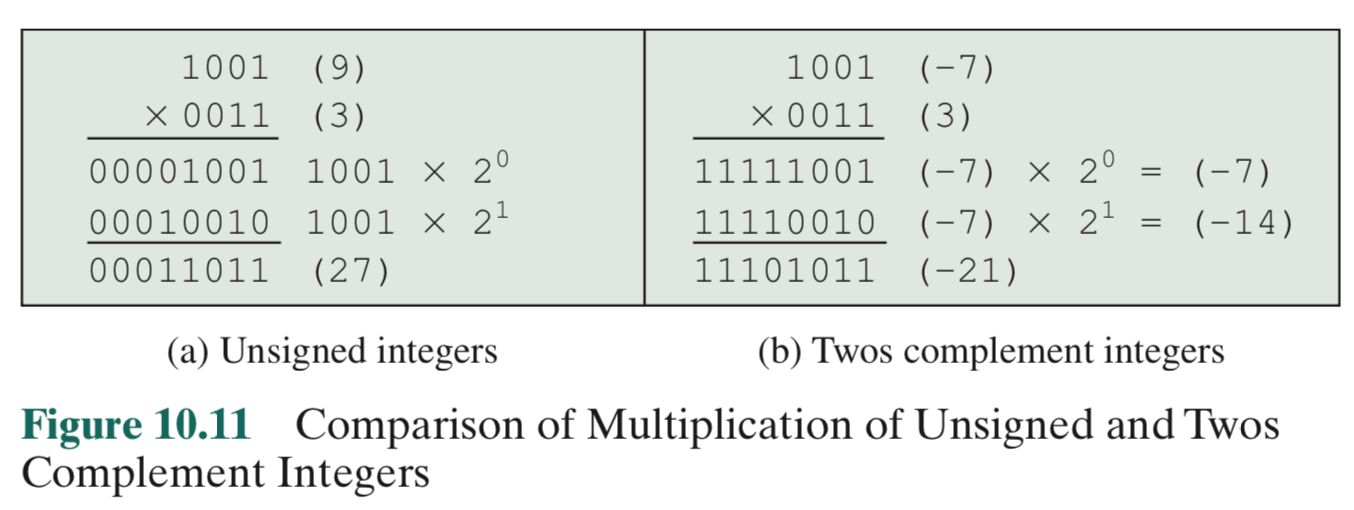
遗憾的是，这个简单的方案不能用于乘法了。为了说明这一点，再看图10.7。我们将11（1011）乘以13（1101），得到143（10001111）。如果我们将其解释为两个补码数，则是-5(1011)乘以-3(1101)得到的却是-113(10001111)。这个例子表明，如果被乘数和乘数都是负数，那么简单直接的乘法就行不通了。事实上，被乘数和乘数只要有一个是负数，那么它就不能工作。为了说明这种状况，需要返回到图10.7的例子，借助2次幂操作看看实际在做什么。回想一下，任何一个无符号二进制数都可以表示为2的幂之和。因此，

1101 ＝1×23＋1×22＋0×21＋1×20

＝23＋22＋20

此外，一个数乘以2n可以通过左移此数n位来实现。理解了这一点，可以看出图10.10改造了图10.7，以使得通过乘法生成的部分积更加明确。图10.10中唯一的不同之处在于，它把n位乘数产生的部分积当作是一个2n位的数。

因此，作为一个无符号整数，4位被乘数1011被存储在8位字00001011中。每个部分积（20对应的一位除外）都由这个数向左移动，并且空出位以0填充而形成（例如，此数左移两次产生00101100）。

现在我们可以说明，如果乘数是负的，那么直接的乘法不能工作的原因。问题在于，作为部分积的负的被乘数，其每次得出的部分积必须是2n字长的负数；部分积的符号位必须一同设置。这可由图10.11说明，它表示的是1001乘以0011。如果这些数被看作是无符号整数，则是9×3=27，处理很简

单。但是，如果把1001看作是补码数-7，则每个部分积必须是2n位(8位)的负的补码数，如图10.11b所示。注意，这要求用部分积左边填充1实现的。

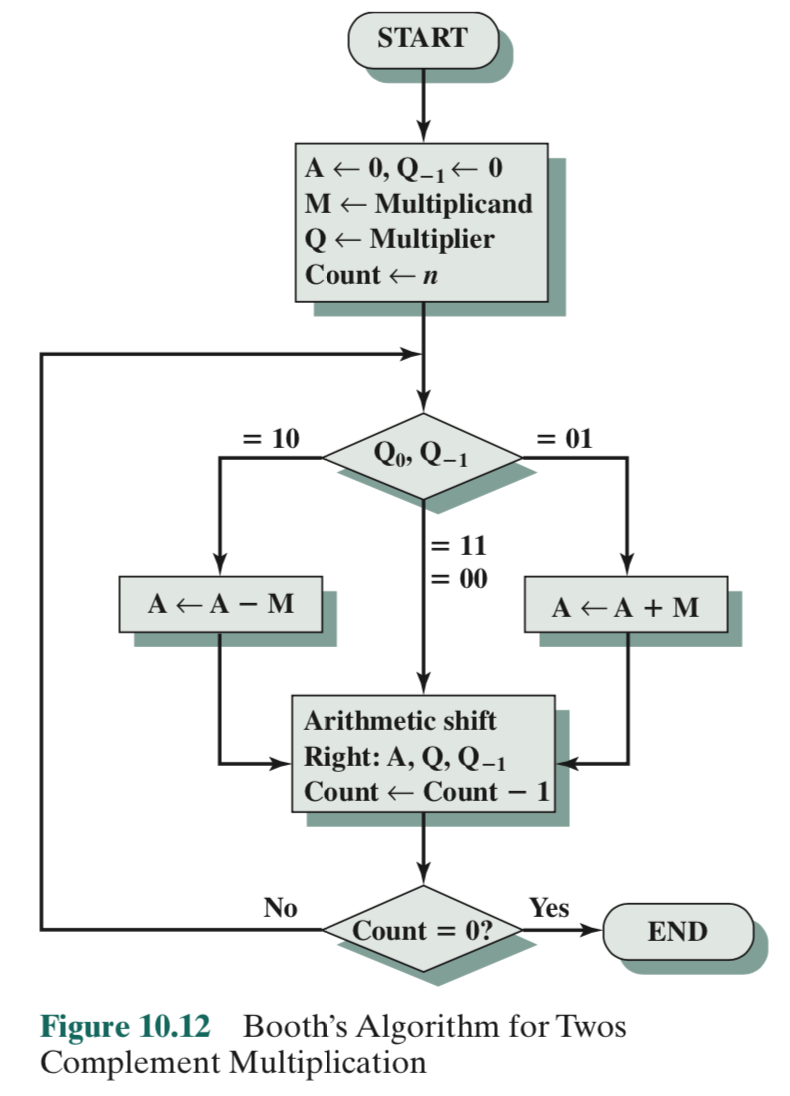
如果乘数是负的，那么简单直接的乘法也是不能工作的。原因是乘数的各位不再对应于必须发生的移位或乘法操作。例如，十进制数字-3的四位补码表示为1101。如果我们采用简单的按位操作来取部分积，则会有如下的对应结果：

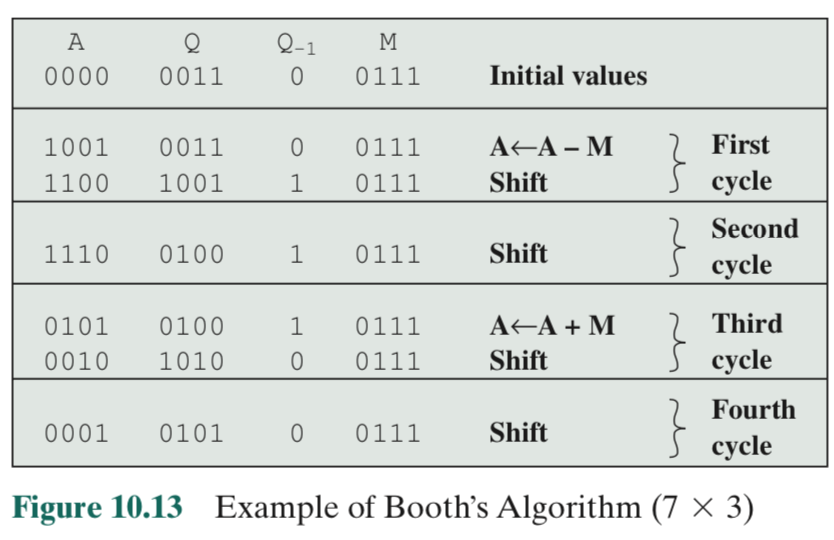


事实上，1101对应的是-（21 + 20）。因此负的乘数不能直接用于上面所描述的方式。

有摆脱这种困境的方法。一种是将被乘数和乘数都转换为正数再相乘，然后当且仅当两个原始数的符号不同时，其结果取2的补（即取负）。乘法电路实现者更倾向于使用不需要这种最终转换步骤的技术。其中最常见的是Booth算法[BOOT51]。相对于无符号数直接乘法，该算法也有利于加速乘法过程。

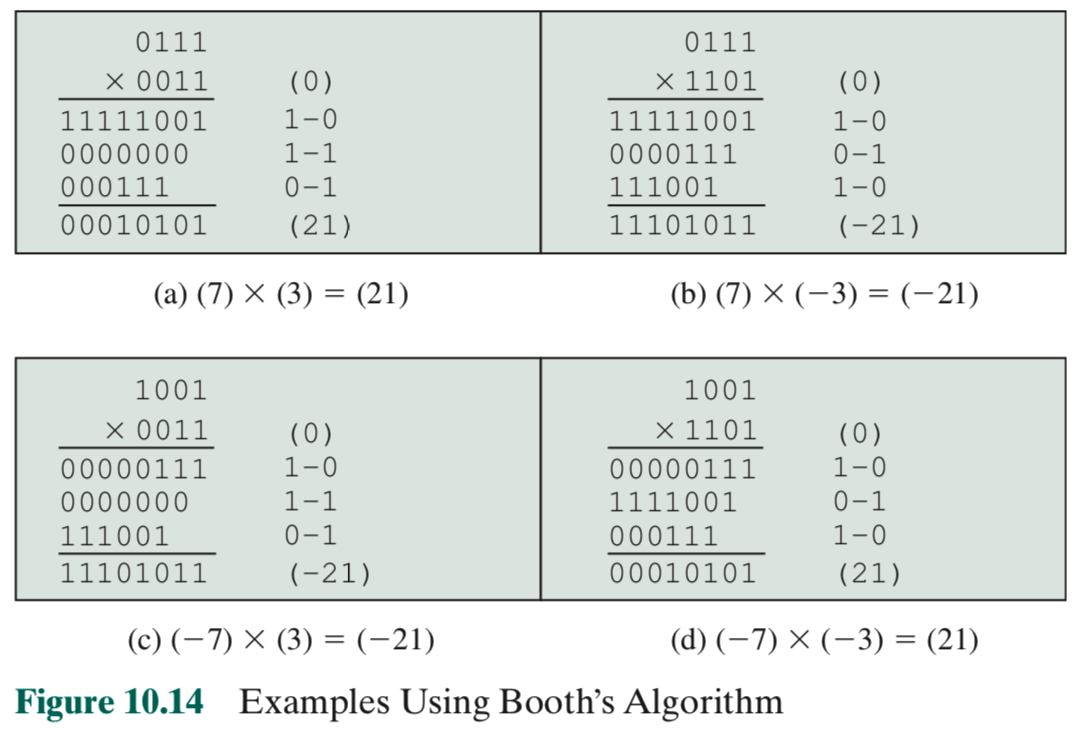
图10.12给出了Booth算法的框图，可以描述如下。与前面一样，乘数和被乘数分别被放置在Q和



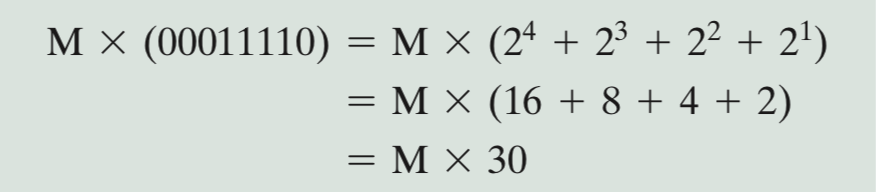


M寄存器中。这里也有一个1位寄存器，逻辑上位于Q寄存器的最低有效位（Q0）的右边，并命名为Q-1；它的用途下面即将说明。乘法的结果将保存在A和Q寄存器中。A和Q-1初始化为0。与前文一样，控制逻辑也是每次扫描乘数的一位。只是现在，当检查某一位时，同时也检查其右边的位。如果这两个比特是相同的（1 - 1或0 - 0），那么A、Q和Q-1寄存器的所有位都向右移动1位。如果这两个比特不同，则被乘数被加到A寄存器中或从A寄存器中减去，这取决于两位是0 – 1还是1 - 0。加减之后，再发生右移。无论哪种情况，右移是这样进行的：A的最左位，即An-1位，不仅被移入An-2,而且仍保留在An-1。这要求保留A和Q中数字的符号。这种移位被称为**算术移位**（arithmetic shift），因为它保留了符号位。

图10.13显示了7乘3时布斯算法的操作顺序。图10.14a给出了该操作更紧凑的表示，图10.14其余部分给出了算法的其他示例。可以看出，布斯算法适用于正数和负数的任意组合。同时还要注意该算法的效率。连续的1串或0串都可以跳过，每个串平均下来只有一次加法或减法运算。

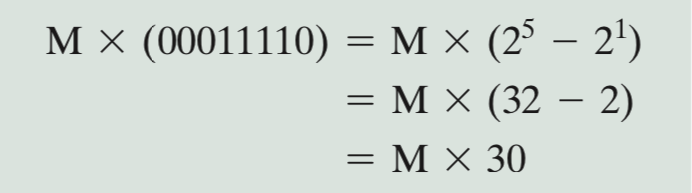


布斯的算法为什么能得到正确的结果呢？首先考虑一个正乘数的情况。具体来说，考虑一个正乘数，它由一个全为1的块和两边的0串组成（例如，00011110）。众所周知，乘法可以通过将被乘数适当左移的副本相加求和来实现：

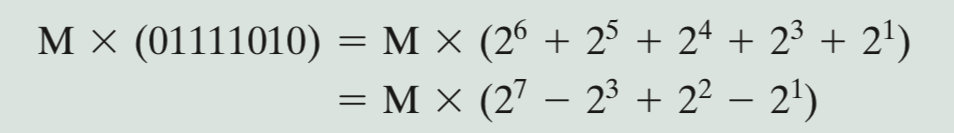


这种操作的数量可以减少到两个，如果我们认识到：

2n + 2n-1 + … + 2n-K  = 2n+1 - 2n-K (10.3)



因此，可以通过被乘数的一次加法和一次减法得到积。该策略客户以扩展到乘数中有任意数量（连续）的1块，包括把单个1当作一个块的情况。于是有：



布斯算法这样执行这种策略：当遇到一串1的第一个1时（1–0）时，执行一次减法，当遇到一串1的最后一个1（0–1）时，执行一次加法。

为说明对于一个负的乘数这样的策略依然可以工作，我们需要做如下推导。假设X是一个2的补

码表示的负数，其表示为：

X的表示 = {1xn-2xn-3…x1x0}

那么x的值可以表示为：

X= -2n-1 +（xn-2×2n-2）+（xn-3×2n-3）+（x1×21）+（x0×20） （10.4）

读者可以通过将此算法应用到表10.2中的数字来验证这一点。

X的最左位是1，因为X是负数。假设最左边的0在第k位上。于是，X的表示为：

X的表示＝{111…10xk-1xk-2…x1x0} （10.5）

其值为：

X= -2n-1 + 2n-2 +… + 2k+1（xk-1×2k-1）+ … +（x0×20） （10.6）

由式（10.3）可知：

2n-2 + 2n-3 + … + 2k+1  = 2n-1 - 2k+1

重新排列：

-2n-1 +2n-2 + 2n-3 + … + 2k+1  = - 2k+1 （10.7）

将式（10.7）代入式（10.6），可以得到：

X= -2k+1 +（xk+1×2k-1）+ … +（x0×20） （10.8）

最后，我们返回到布斯算法。记住X的表示(式(10.5))，很显然，从x0起到最左边的0的所有位都能被这样处理，因为它们产生式(10.8)中除去(-2k+1)项之外的所有项。当算法扫描经过最左的0而遇到下一个1（2k+1）时，会出现一个1-0转换，并且进行一个减法（-2k+1）。这是式（10.8）中的剩余项。

作为一个例子，让我们考虑某个乘数M乘以（-6）。在2 的补码表示中，使用8位字长，(-6)被表示为11111010。通过式（10.4），我们知道：

-6 ＝-27＋26＋25＋24＋23＋21

这个读者可以很容易验证。因此有：

M ×（11111010）＝ M ×（-27＋26＋25＋24＋23＋21）

使用式（10.7），

M ×（11111010）＝ M ×（- 23＋21）

读者可以验证这仍然是M ×（- 6）。最后，按照我们先前阐述的理由，可以得出：

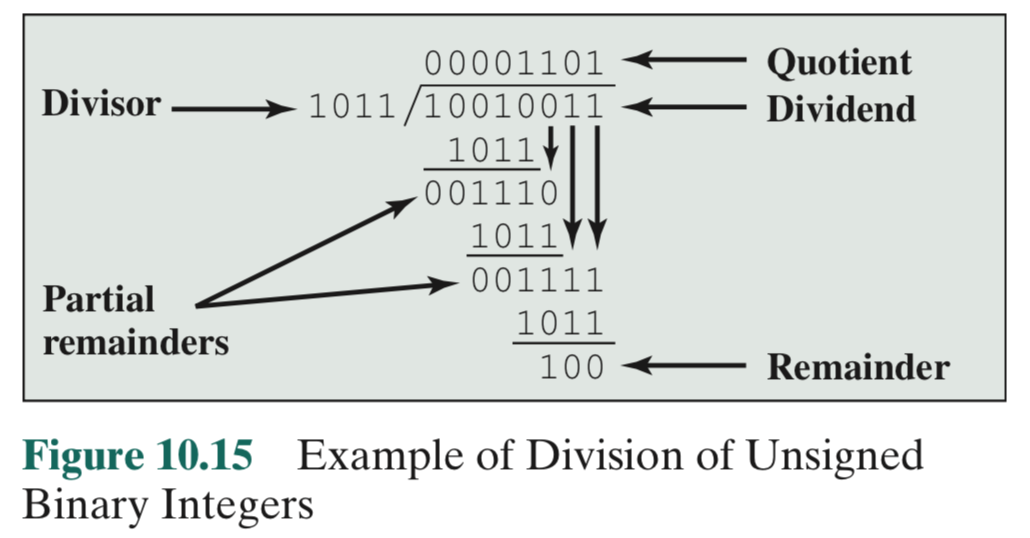
M ×（11111010）＝ M ×（- 23+22-21）

现在，我们清楚地看出布斯算法是遵守上述操作方式的。当遇到第一个1(1-0)时执行一次减法；遇到(0-1)时执行加法；最后遇到下一个全是1串的第一个1时又执行另一次减法。因此，布斯算法与直接移位相加的算法相比只需完成较少的加法和减法。

### 除法

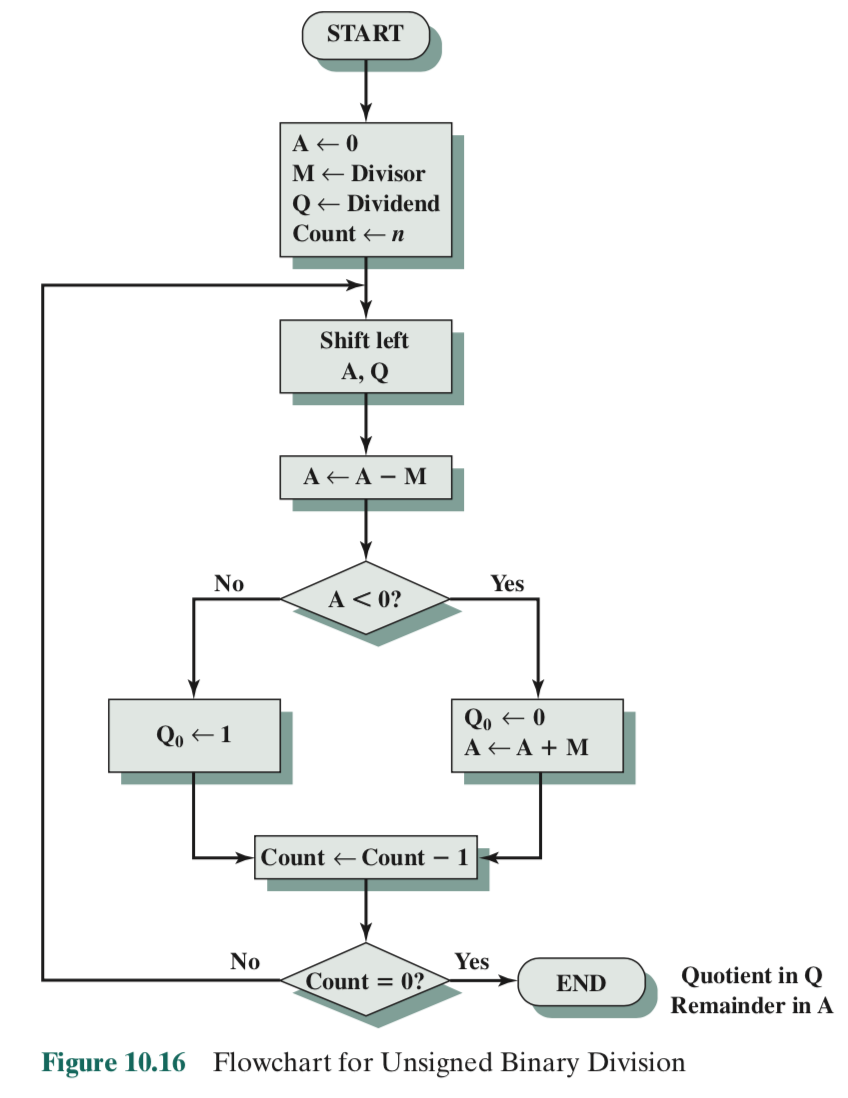
除法要比乘法稍微复杂一些，但是基于相同的通用原理。同前述一样，算法的基础是纸和笔的演算，并且操作包括重复的移位和加与减。

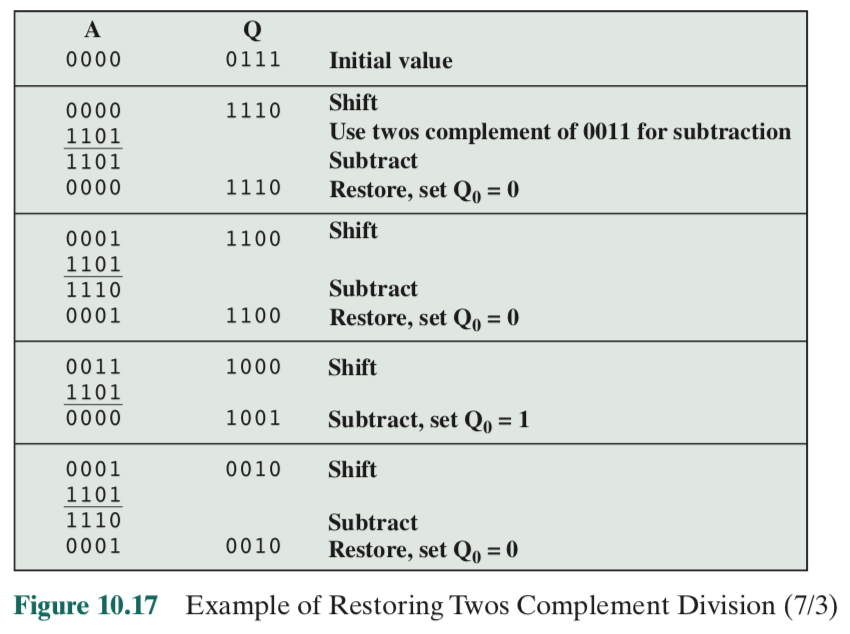
图10.15表示的是无符号二进制整数的长除法(long division)例子。详细描述该过程具有一定的指导意义。首先，从左到右检查被除数的位，直到所检查的位表示的数大于或等于除数；这被称为除数能够去“除”此数。直到这个事件发生之前，一串0从左到右被放入到商中。当上述事件发生时，将一个1放入到商中，并从这个部分被除数中减去除数。结果被称为部分余(partial remainder)。



由此开始除法呈现一种循环模式。在每个循环中，被除数的其他位续加到部分余上，直到所构成的数大于或等于除数。和以前一样，从该数中减去除数以产生新的部分余。这个过程一直持续下去，直到被除数的所有位被用完。

图10.16显示了与此长除过程相对应的机器算法。除数放在M寄存器中，被除数放在Q寄存器





中。每一步，A和Q寄存器一起向左移动1位。然后A减去M来确定A是否能分出部分余3来。如果够减，则Q0位变为1。否则，Q0位为0，并且M必须被返加到A中来恢复原先的值。计数值然后递减1，该过程持续进行n步。最后，商保存在Q寄存器中，余数保存在A寄存器中。

这个过程可以扩展到负数，但是有一些难度。我们给出一种用于2的补码数的方法。这种方法的一个例子显示在了图10.17中。

该算法假设除数*V*和被除数*D*都为正数。|*V*| < |*D*|。如果|*V*| = |*D*|，那么商*Q*=1且余数*R* = 0。如果|*V*| > |*D*|，则*Q*＝0，*R*＝*D*。算法可以概括如下：

1. 将除数的2的补加载到M寄存器，实际上是将除数的相反数装入到M寄存器中。被除数装入到A、Q寄存器。被除数必须表示为2n位正数。例如，4位0111变成00000111。
2. A，Q左移1位。
3. 执行A ← A - M。这个操作从寄存器A的值中减去除数。
4. a. 如果上一步的减法结果为非负（寄存器A的最高有效位为0），那么置Q0←1。

b. 如果上一步的减法结果为负（寄存器A的最高有效位为1），则置Q0←0，并恢复寄存器A的原值。

1. 重复步骤2到4，Q有多少位就重复多少次。
2. 余数在A中，商在Q中。

注释3-------

这是无符号整数的减法。如果结果产生了从最高位向上的借位，则是不够减。

为了处理负数，我们考虑余数定义为：

D = Q × V + R

（这里*D* = 被除数，*Q* = 商， *V* = 除数， *R* = 余数）

考虑以下整数除法例子，其中包含所有可能的D和V组合：

D = 7 V= 3 => Q = 2 R= 1

D = 7 V= -3 => Q = -2 R= 1

D = -7 V= 3 => Q = -2 R= -1

D = -7 V= -3 => Q = 2 R= -1

读者可能从图10.17中注意到（-7）/（3）和（7）/（-3）产生不同的余数。我们发现，Q和R的绝对值不受被除数和除数符号的影响。而Q和R的符号很容易从被除数D和除数V的符号导出。具体而言，sign(R) = sign(D)，而sign(Q) = sign(D) × sign(V)。因此，实现2的补码除法的一种方法是将操作数都转换为无符号绝对值，进行除法运算，并在最后根据需要通过被除数he除数的符号来计算商和余数的符号。这是恢复余数除法算法[PARH10]所选取的方式。

## 10.4浮点表示法

### 原理

使用定点表示法（例如，2的补码）可以表示以0为中心或接近0的一定范围内的正负整数。通过假定一个固定的小数点的位置，这种格式也可以用来表示带有小数部分的数。

这种方法有局限性，那就是它不能表示非常大的数字，也不能表示非常小的分数。此外，当两个大数相除时，商的小数部分可能会丢失。

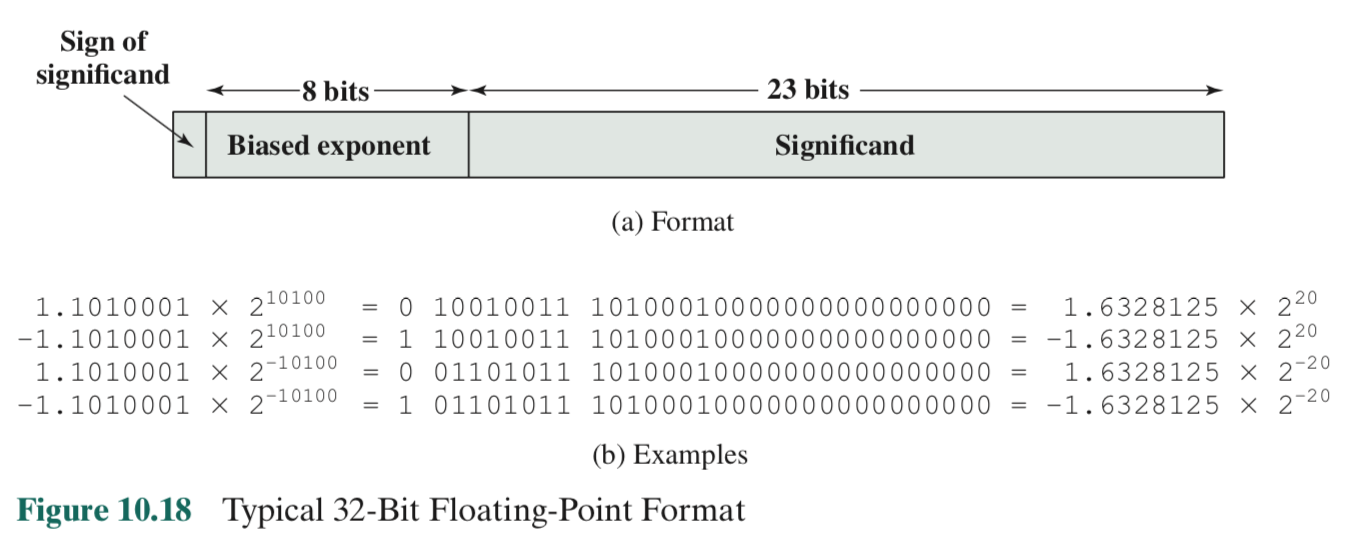
对于十进制数，我们可以使用科学计数法(scientific notation)来解除这个限制。于是，976 000 000 000 000可以表示为9.76×1014，而0.000 000 000 097 6可以表示为9.76×10-14，我们所做的实际上只是动态地将小数点移动到一个合适的位置，并使用10的指数来跟踪小数点。这就允许只用少数几个数字来表示很大范围的数和很小的数字。

同样的方法可以用于二进制数。我们可以用形式表示数字：

±S×B±E

这个数字可以存储在具有三个字段的二进制字中：

* 符号：正或负。
* 有效值S(significant)。
* 指数或称为阶E(exponent)



指数的**底**或者**基**B(base)是隐含的，不需要存储，因为它对所有数字都是一样的。通常，小数点的位置约定好在最左（最高）有效位的右边。也就是说，小数点左边有一位。

最好用一个例子来说明用于表示二进制浮点数的原理。图10.18a表示了一个典型的32位浮点格式。最左边的位存储数字的**符号**（0 = 正，1 = 负）。阶值存储在位1到位8中。所使用的表示称为**移码表示法(biased representation)**。从字段中减去一个称为偏移量(bias)的固定值，才可以获得真正的指数。通常，偏移量等于（2k-1-1），其中k是二进制指数的位数。在此例子中，一个8位字段能表示的数字是0到255。当取偏移量为127（即27-1）时，则真实的阶值范围为-127到+128。在这个例子中，阶值的底被认为是2。

表10.2给出了4位整数的移码表示。注意，当移码表示法的各位被当作一个无符号整数时，其数的相对大小关系不会改变。例如，在移码和无符号数两种表示法中，最大数字都是1111，最小数字都是0000。然而在符号-幅值或2的补码表示法来说不是这样的。使用移码表示法的一个优点是，为了便于比较，非负的浮点数可以被视为整数对待。

字（本例中为23位）的最后一部分是**有效值**4。

任何浮点数都可以用多种方式表示。

以下各式是等价的，其中有效数以二进制形式表示：

0.110 × 25

110 × 22

0.0110 × 26

为了简化对浮点数的操作，通常需要对它们进行规格化（normalize）。一个**规格化数**是

注释4-------

用于替代有效值的术语是**尾数**，但尾数有些过时了。尾数也被用于表示“对数的小数部分”，因此在此书中最好避免使用。

一个有效值的最高有效位非零的数。因此，对于二进制表示法，一个规格化数是指其有效值的最高有效位为1。如前所述，通常约定小数点左边有一位。于是，一个规格化的非零数具有如下的格式：

±1.bbb…b×2±E

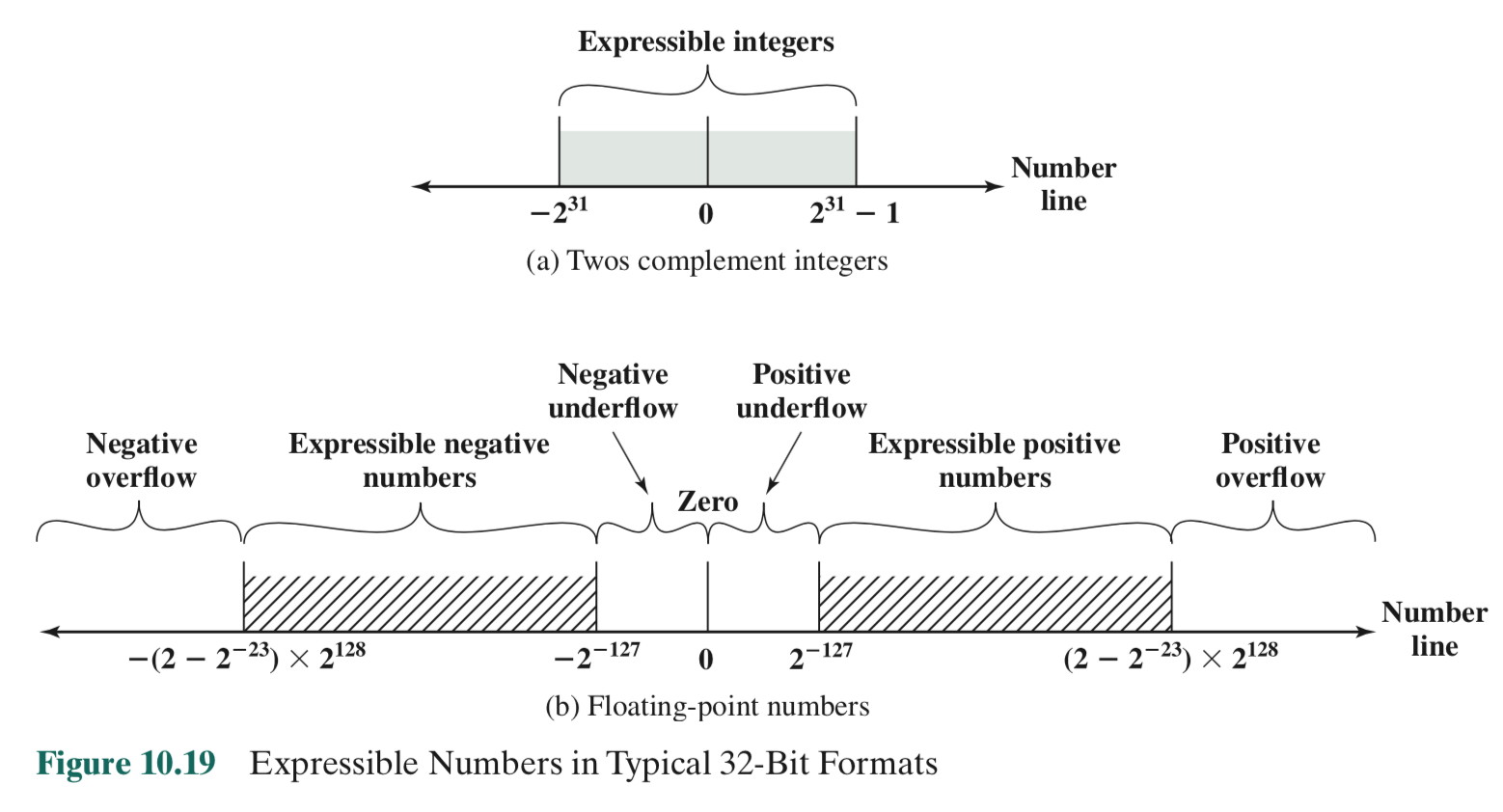
其中b是二进制数字（0或1）。这表示有效位的最左位必须总是1，所以没有必要总存储这个1，所以它是隐含的。因此，23位有效数字可以用于存储24位有效数字，其值范围在半开区间[1，2)中。对于一个非规格化数，可以通过将小数点移动到最左边的一个1的右边并相应地调整阶值，即可将此数规格化。

图10.18b给出了几个以这种规格化形式存储的数字的例子。其中每个示例的左边是二进制数，中间是对应的二进制位串表示，右边是十进制数值。注意如下特征：

* 符号总是存储在字的第一位。
* 真实有效值的第一位总是1，而且不需要存储在有效值字段中。
* 值127加到真实阶值后再存入阶值字段
* 阶的底是2。

为了进行比较，图10.19显示这种表示法的32位字能表示的数字范围。使用2的补码整数表示法，可以表示从-231到231-1的所有整数，总共232个不同的数字。以图10.18的浮点格式为例，可以表示如下范围的数：

* 介于-(2-2-23)×2128和-2-127之间的负数。
* 介于2-127和(2-2-23)×2128之间的正数



数轴上有五个区间不包括在这些范围内：

* 小于-(2-2-23) ×2128的负数，称为**负上溢**(negative overflow)。
* 大于-2-127的负数，称为**负下溢**(negative underflow)。
* 零。
* 小于2-127的正数，称为**正下溢**(positive underflow)。
* 大于(2-2-23) ×2128的正数，称为**正上溢**(positive overflow)。

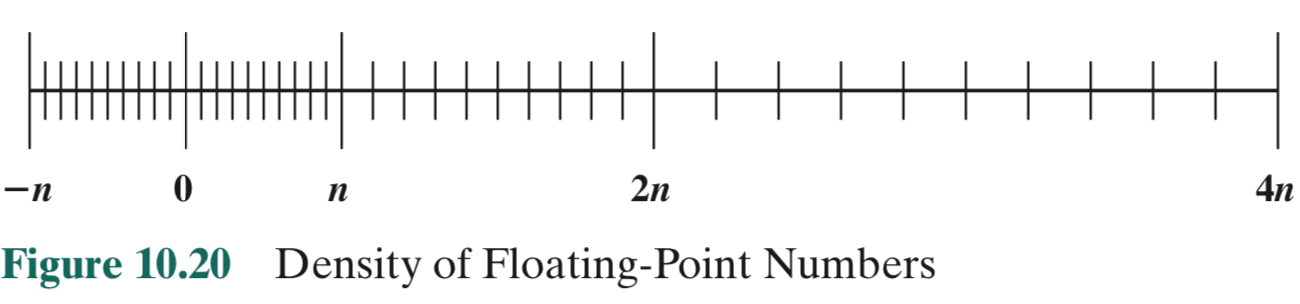
正如上文提到的，这种表示法不适合表示0。然而，下面将会看到，我们可以把一个专门的位串定义成零。当算术运算的结果幅值比指数128能表示的幅值还大时，会发生上溢（例如2120 × 2100 = 2220）。当小数的幅值太小（例如，2-120 × 2-100 = 2-220），会发生下溢。下溢是一个不太严重的问题，因为其结果通常足够小而可以近似为0。

注意，使用浮点表示法并不能表示更多的值。使用32二进制位串可以表示的不同值的最大数目仍然是232。我们所做的，实际上只是把这些数沿数轴正负两个方向在更大范围内分布。在实际应用中，人们希望表示的大多数浮点数仅仅是近似的表示。然而，对于不是很大的整数而言，浮点表示是精确的。

此外还应注意，用浮点符号表示的数字不再像定点数那样沿数轴等距分布，而是越靠近原点分布越密集，越远离原点，分布约稀疏，如图10.20所示。这是浮点数学的一个权衡：多数计算产生的结果并不是严格精确的，必须进行某种舍入，以使结果达到所能表示的最近似值。

在图10.18所示的格式类型中，范围(range)和精度(precision)之间有一个折衷。该示例中8位用于阶，23位用于有效数。如果我们增加阶的位数，就扩充了可表示数的范围。但是，因为可以表达的数总数是固定不变的，所以实际效果是减少了这些数字的密度，因此降低了精度。同时增加范围并提高精度的唯一方法是使用更多的位。于是，大多数计算机至少提供单精度数字和双精度数字两种浮点数。例如，一个处理器可以支持64位的单精度格式和128位的双精度格式。

因此，在阶中的位数和有效值位数之间存在一个折中。还有一个问题比这更复杂。隐含的阶值的底并不需要总是2。例如，IBM S/390使用的就是一个以16为底的体系结构[ANDE67b]。其浮点数格式由7位阶值和24位有效值组成。



在IBM BASE-16格式情况下：

0.11010001 × 210100＝0.11010001×16101

此时所存的阶值是5而不是20。

使用较大指数的优点在于，相同数量的阶值位可以表示的数的范围更大。但是请记住，我们并没有增加可以表示的数的总数。因此，对于一种固定格式而言，更大的底能给出更大的表示范围，但代价是精度会变低。

### 二进制浮点表示的IEEE标准

最重要的浮点表示是在1985年采用并于2008年修订的IEEE标准754中定义的浮点表示法。这个标准的指定是为了提高程序从一个处理器移植到另一个处理器的可移植性，并促进开发复杂、面向数值的程序。该标准已获得广泛的认可，并被应用于几乎所有现代处理器和算术协处理器中。IEEE 754-2008同时包含了二进制和十进制的浮点表示法。在本章中，我们只讨论二进制表示。

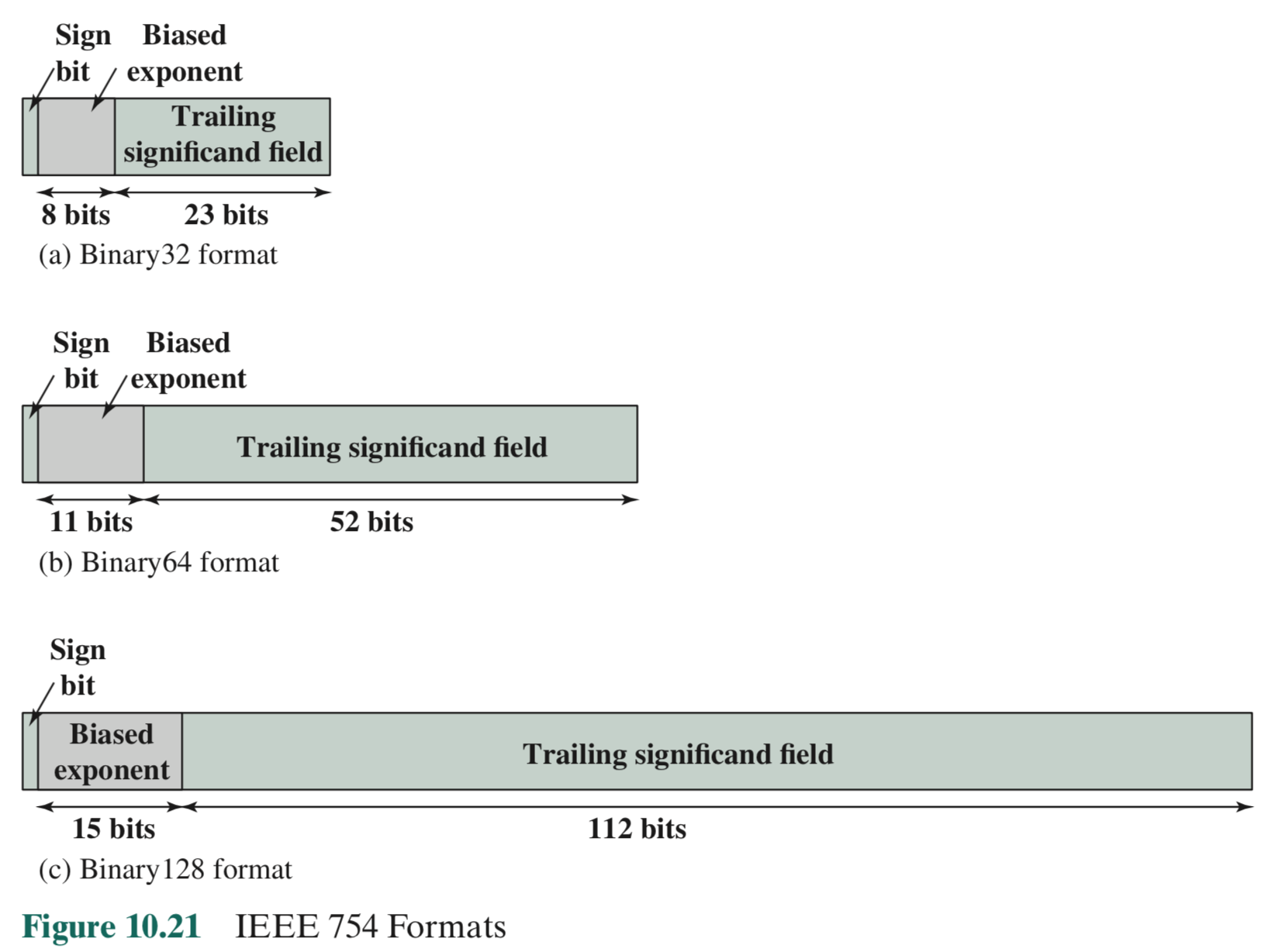
IEEE 754-2008定义了以下几个不同类型的浮点格式：

* + **算术格式(Arithmetic format)**：此格式支持标准定义的所有强制操作。该格式可用于表示标准中描述的操作的浮点操作数或结果。
  + **基本格式(Basic format)**：该格式包括五个浮点表示法，三个二进制和两个十进制， 其编码由标准指定，可用于算术。在任何一致实现中都实现了至少一种基本格式。
  + **交换格式(Interchange format)**：这是一种完全指定的固定长度的二进制编码，允许数据在不同平台之间进行交换，并且可以用于存储。

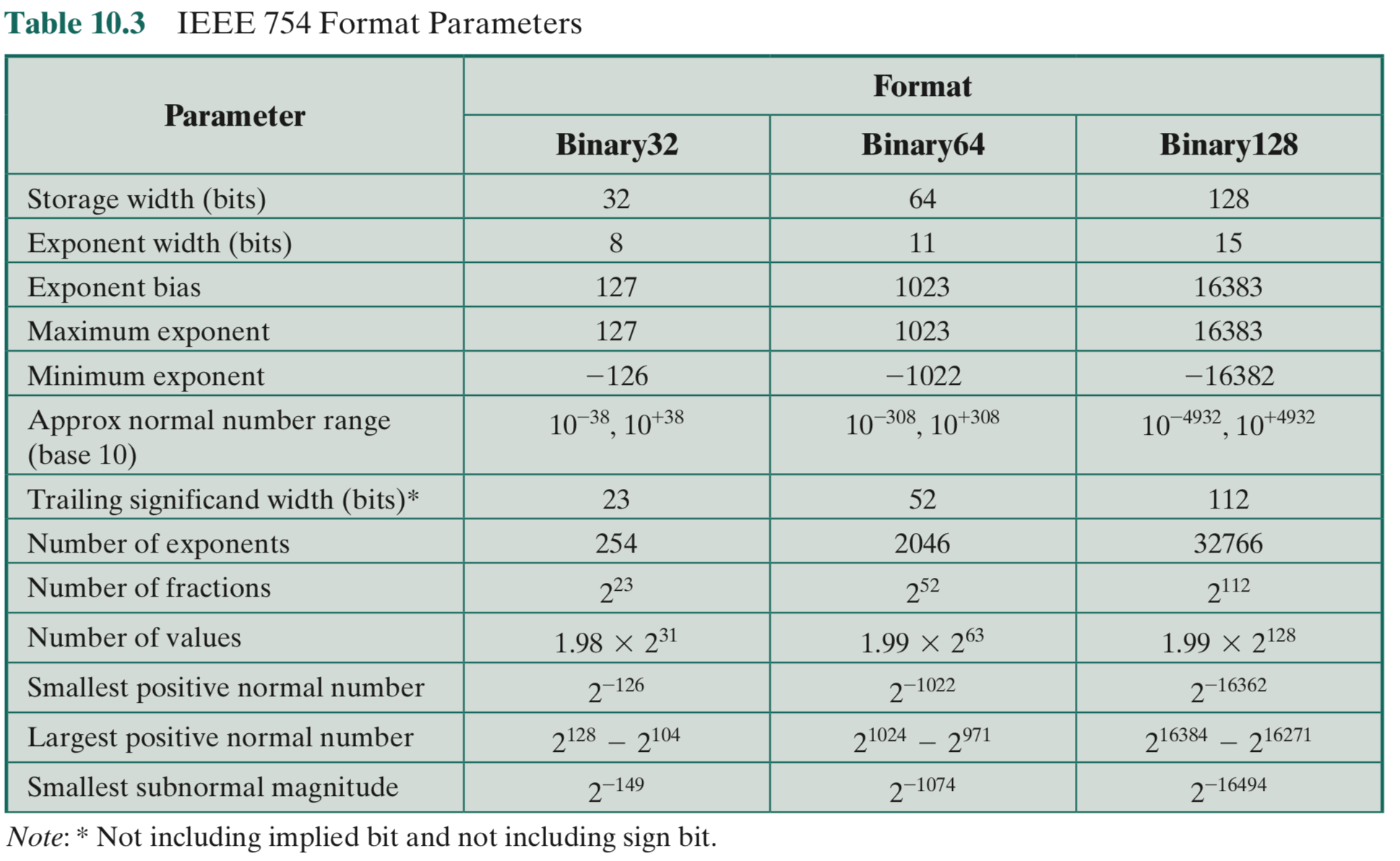
三种基本二进制格式的比特长度分别为32、64和128位，阶值分别为8、11和15位（如图10.21）。表10.3总结了三种格式的特点。这两种基本的十进制格式的位长度是64位和128位。所有的基本格式同时也是算术格式类型（可用于算术操作）和交换格式类型（独立平台）。

标准中指定了其他几种格式。二进制16(binary16)格式只是一种交换格式，并在不需要较高精确度的情况下存储数值。二进制{k}格式和小数{k}格式是具有总长度k位，并具有定义长度的有效数和指数的交换格式。格式必须是32位的倍数；因此格式定义为k ＝ 160, 192，等等。这两种格式的家族同时也属于算术格式。

此外，该标准定义了**扩展精度格式**，其中通过在指数（扩展范围）和有效值（扩展精度）中增加额外比特来扩展支持的基本格式。精确格式是依赖于实现的，但是标准对指数和有效值的长度有一定的限



制。这些格式是算术格式类型，但不是交换格式类型。扩展格式被用于中间计算。有了更高的精度，扩



展格式减少了最终结果被过度舍入错误污染的概率；由于它们的范围较大，它们也减少了中间上溢，从而中止计算的概率，该计算的最终结果可以以基本格式表示的。扩展格式的另一个有点是它在带来更大的基本格式的好处的同时，不会带来通常与较高精度相关联的时间损失。

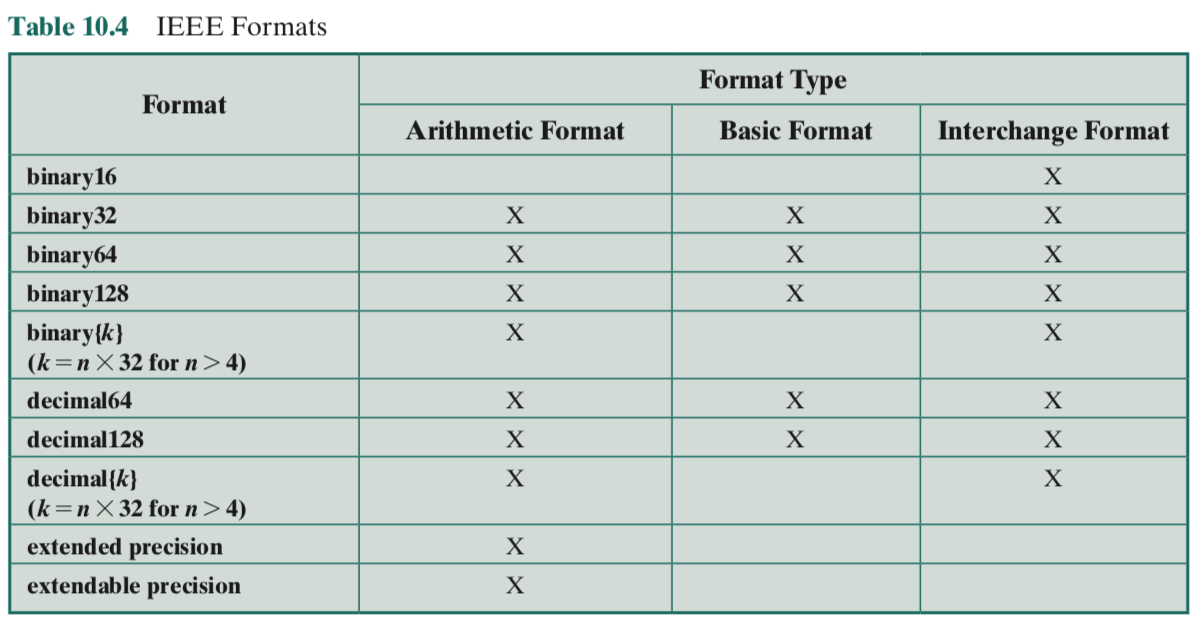
最后，IEEE 754-2008将**可扩展精度格式**定义为具有在用户控制下定义的精度和范围的格式。同样，这些格式可以用于中间计算，但是标准没有提供约束、格式以及长度。

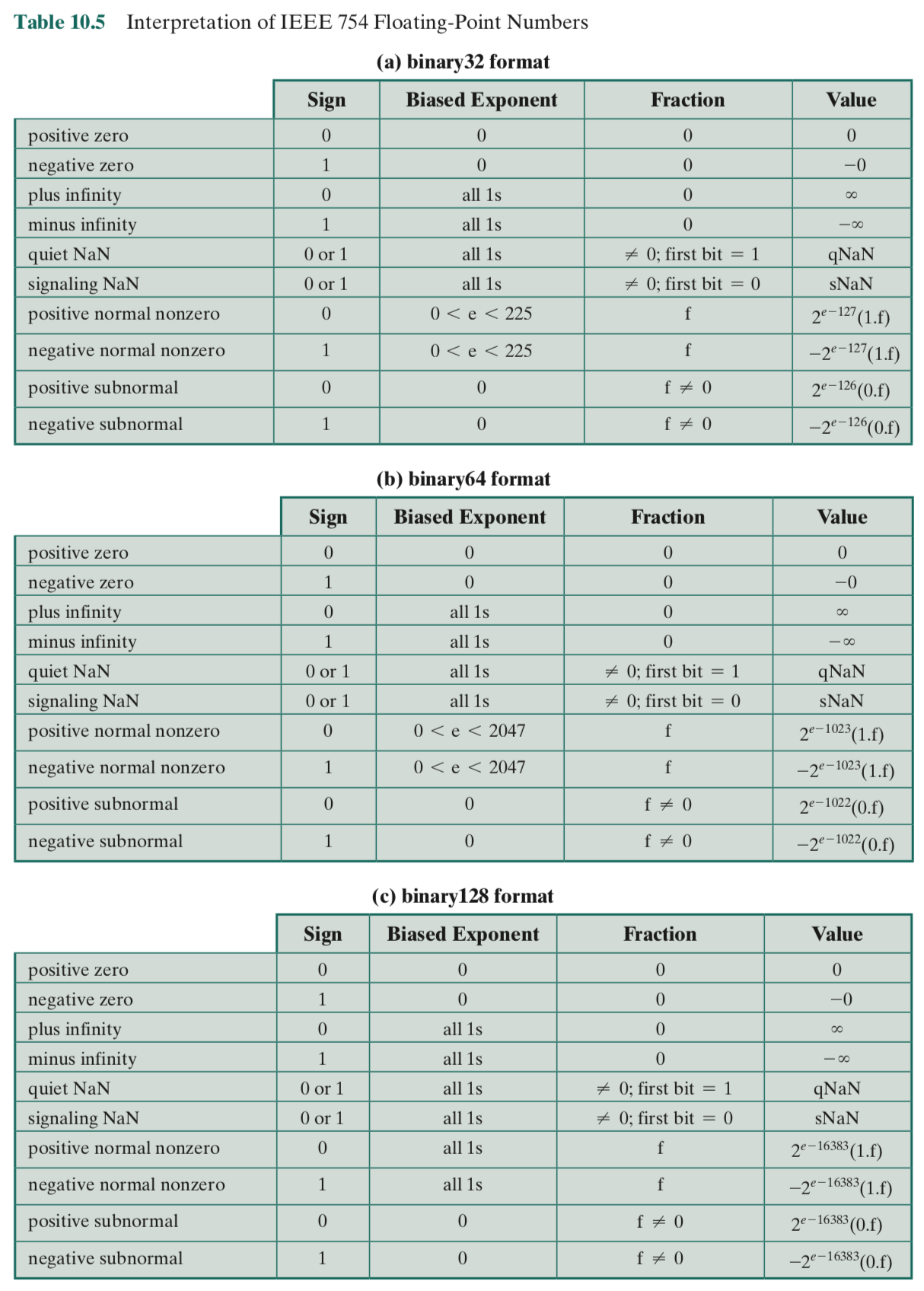
表10.4显示了定义的格式和格式类型之间的关系。

IEEE格式中的所有位模式(bit pattern)并不是都以通常方式解释；有些位模式用于表示特殊值。表10.5指出了各种位模式对应的值。全0（0串）和全1（1串）的指数值定义特殊值。下面的数字分类的

代表：

* + 对于在1-254之间的32位格式，1-2046之间的64位格式，以及1-16382之间的规格化非零浮点数都被表示。指数是有偏移量的，所以对于32位格式，指数范围是-126到+127，等等。规格化数需要小数点左侧一位为1；该位是隐含的，因此对应可以有24位、53位或113位有效数字。因为其中一位是隐含的，所以二进制格式的对应字段被称为**尾随有效数字域**(trailing significand field)。
  + 根据符号位不同，指数域为0同时小数域也为0可以表示正零或负零。如前所述，精确地表示0是有很意义的。





* + 根据符号位的不同，全1阶值与0有效值一起表示正或负无穷大。能表示无穷大也是有用的。将上溢作为错误条件而停止程序执行，还是把它看成是一个∞值带入程序并继续处理，这样的决定权留给用户。
  + 0阶值与非零有效值一起表示一个非规格化数(denormalized)。在这种情况下，二进制小数点左边的隐藏位是零，真正的阶值是-126或-1022。数字的正负取决于它的符号位。
  + 全1阶值与非0有效值一起给出非数NaN值，这表示不是一个数(Not a number)，非数(NaN)用以表示各种异常情况。

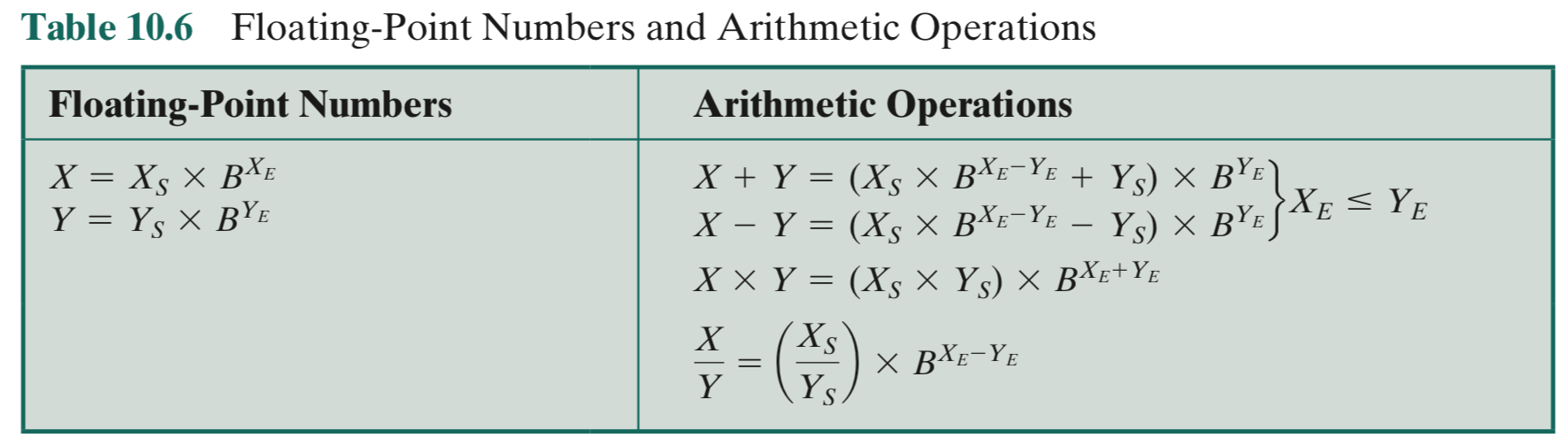
非规格化数和NaN的意义在第10.5节中讨论。

## 10.5浮点运算

表10.6总结了浮点运算的基本操作。对于加法和减法，必须确保两个操作数具有相同的阶。这可能需要移动其中一个操作数上的小数以实现对齐。乘法和除法更简单一些。

浮点操作可以产生如下几种特殊情况：

* + **阶值上溢**(exponent overflow): 正阶值超过最大允许的阶值。NANT值。在某些系统将其设计成+∞或-∞。
  + **阶值下溢**(exponent underflow)：负阶值小于最小允许阶值（例如，-200小于-127）。这意味着该数字太小而不能表示，一般可以报告为0。



举例：

X＝0.3×102＝30

Y＝0.2×103＝200

X + Y=（0.3×102 - 3 + 0.2）×103 = 0.23×103 = 230

X - Y=（0.3）×102 - 3 - 0.2）×103=（ - 0.17）×103 = 170

X×Y= （0.3）×0.2）×102 + 3 = 0.06 × 105 = 6000

X÷Y= （0.3÷0.2）× 102 - 3 = 1.5 ×10-1 = 0.15

* + **有效值下溢**(significand underflow)：处理有效值对齐时，数字可能被移出右端最低位而丢失。下面将要讨论，此时需要某种形式的舍入。
  + **有效值上溢**(significand overflow)：同符号的两个有效值相加可能导致最高有效位的进位。这可通过重新对齐来修补，后面将说明。

### 浮点加法和减法

在浮点算数中，加减运算比乘除运算更复杂，因为它需要对齐。加减法的四个基本阶段是：

1. 检查0。
2. 对齐有效值。
3. 加或减有效值。
4. 规格化结果。

一个典型的流程图如图10.22所示。下面逐步说明浮点加减法所需的主要操作。我们假设一个与图10.21类似的格式。为了加或减操作，两个操作数必须传送到可被算数逻辑单元(ALU)使用的寄存器。如果浮点格式包括一个隐藏有效位，则该位必须先变成显式再做操作。

* **步骤1：零校验**。因为加法和减法除了符号不同之外基本上都是相同的。所以如果是一个减法运算，则过程从一开始就改变减法的符号。接下来，如果任一操作数是0，则另一个操作数就是结果。
* **步骤2：有效值对齐：**下一个阶段是操作数字，使两个阶值相等。

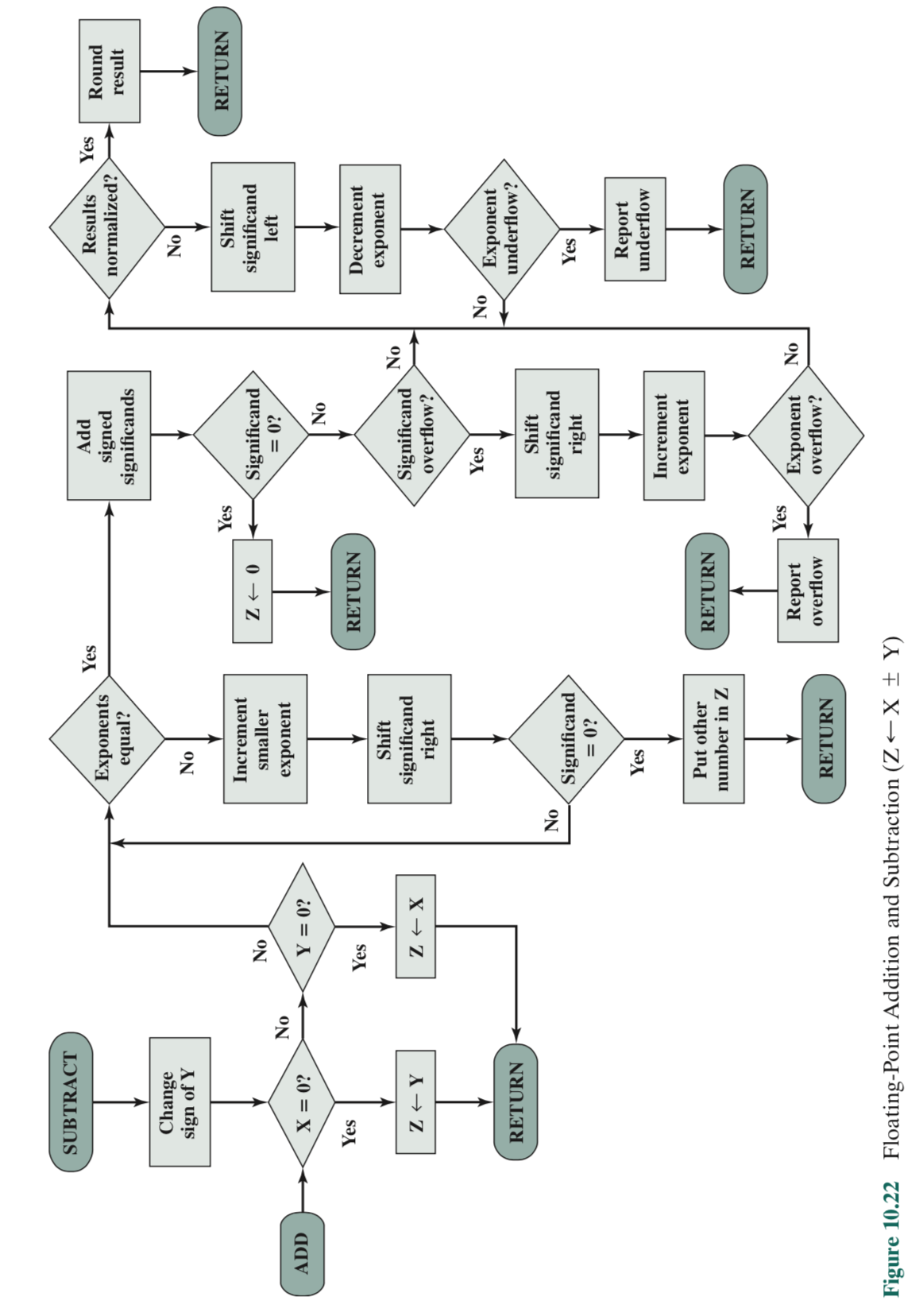
要说明为什么需要这样做，可考虑以下十进制加法：

（123×100）+（456×10-2）

显然，我们不能仅仅加有效值。数字必须首先设置为对等位置，也就是说，第二个数字的4必须与第一个数字的3对齐。在两个阶值相等的条件下两个数才能相加，这是数学的基本要求。因此有：

(123×100) + (456×10-2) = (123×100) + (4.56×100) = 127.56×100

可以通过向右移动较小的数（增加其阶值）或向左移动较大的数（减少它的阶值）来实现有效值对齐。因为任一操作都可能导致数字的丢失，一般来说，右移较小的数而丢失的数字，所造成的影响相对



较小。通过重复右移有效值幅值部分1位，并将其阶值加一，直到两个阶值相等可完成对齐。(注意，如果隐含的基为16，则上述1“位”数的移位相当于4位的移位。)如果该处理导致有效值变为0，则

另一个数字即为结果。因此，如果两个数字的阶值相差很大，则较小的数字丢失。

* **步骤3：加法**：将两个有效值相加，相加时考虑它们的符号。因为符号可能不同，结果可能是0。这里还有可能出现有效值上溢1个数字。如果是这样，则有效值要右移，阶值增加1。阶值增加1又可能发生阶值上溢；这是应当终止操作并报告。
* **步骤4：规格化：**最后一步将结果归一化。左移有效值直到最高有效数字（位，或4位----对底为16而言）为非零。每次左移都会引起阶值相应减1，从而可能导致阶值下溢。最后，结果必须进行舍入，然后再报告结果。我们将舍入的讨论推迟到乘法和除法的讨论之后进行。

### 乘法除法

浮点乘、除法要比加、减法简单得多，如下面的讨论可以看出。

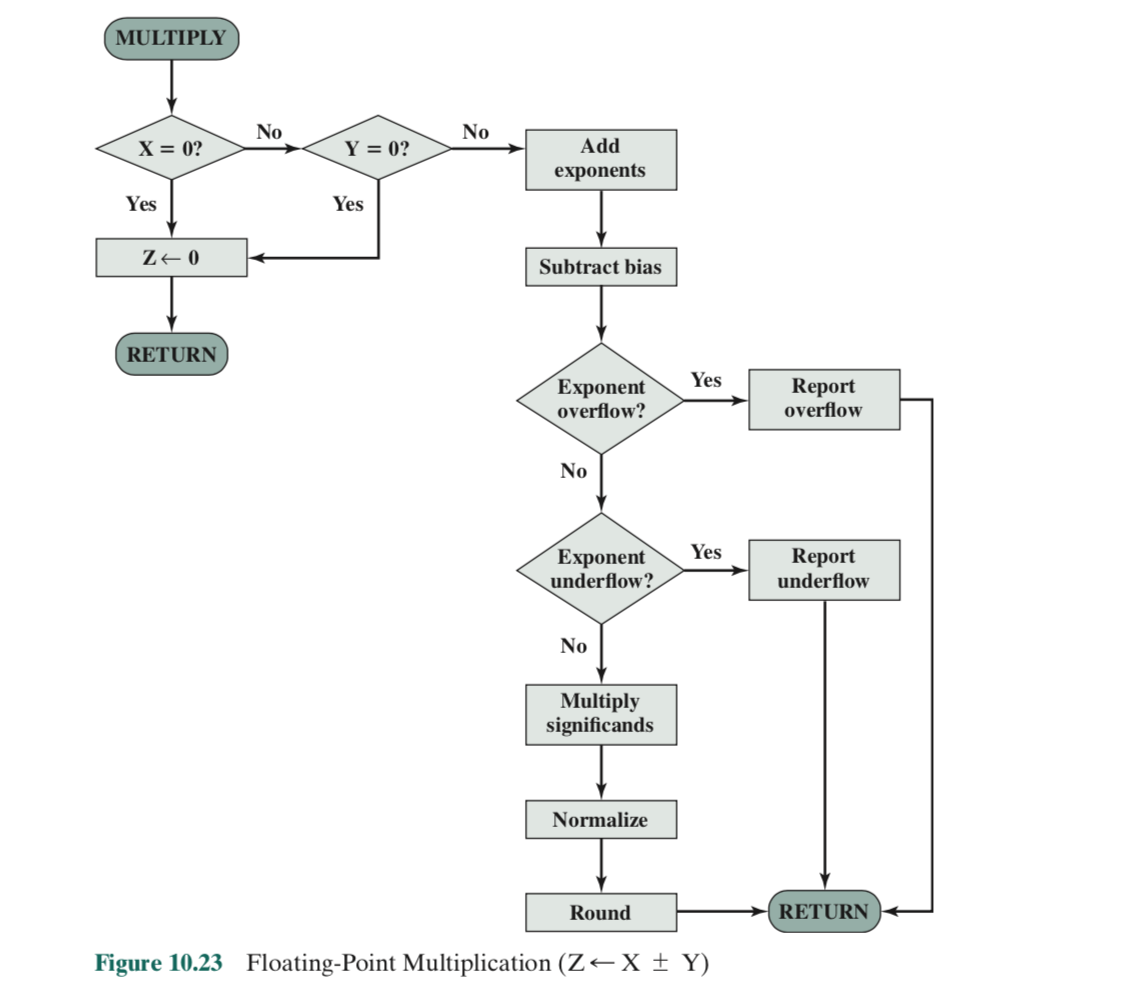
我们首先考虑乘法，过程如图10.23所示。首先，如果任一操作数为0, 则乘积结果为0。下一步是阶值相加。如果阶值是以移码形式存储，则两个阶值的和将是两倍的偏移量。因此，必须从总和中减去一个偏移量。阶值相加可能会出现阶值上溢或下溢，此时应当结束乘法并报告。

如果乘积的阶值在一个恰当的范围内，下一步是应当是有效值相乘，同时考虑它们的符号。有效值相乘与整数乘法的完成方式相同。在本例中，我们使用的是符号-幅值表示法，但是其实现细节与用于2的补码表示法的细节类似。这个乘积的长度将是被乘数和乘数长度的两倍。多余的位将在舍入期间丢失掉。

在计算出乘积之后，对结果进行规格化和舍入处理，就像加法和减法所做的那样。注意，规格化会导致阶值下溢。

最后，让我们考虑图10.24所示的除法流程图。同样，第一步是测试0。如果除数为0，则报告出错，或认为商是一个无穷大，这取决于具体的实现。若被除数是0，则结果为0。接下来，被除数的阶值减去除数的阶值。这个过程减掉了偏移量，故必须在阶值相减后再加上偏移量。然后检查阶值是否出现了上溢或下溢。

下一步是有效值相除。其次是规格化和舍入处理。

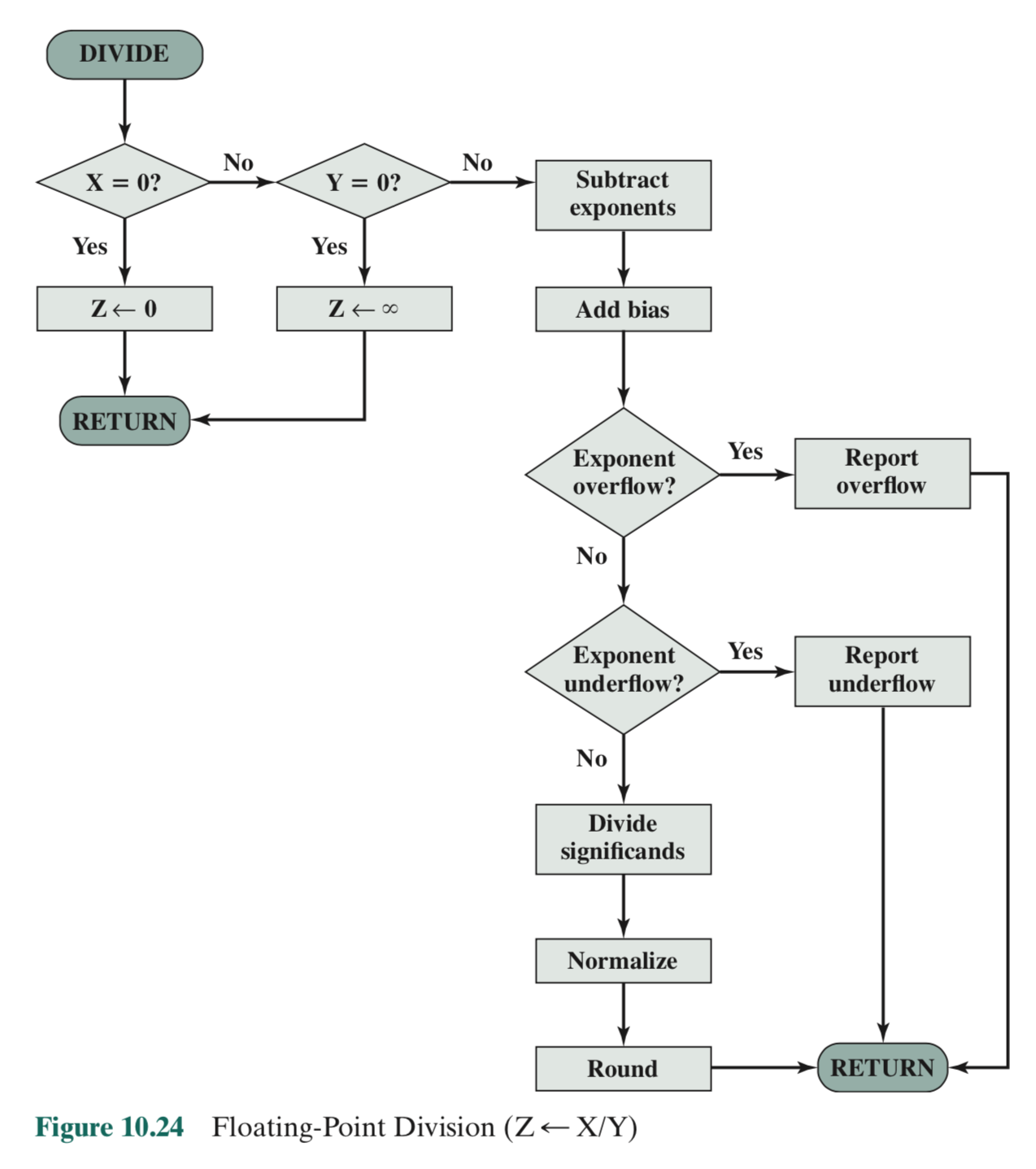


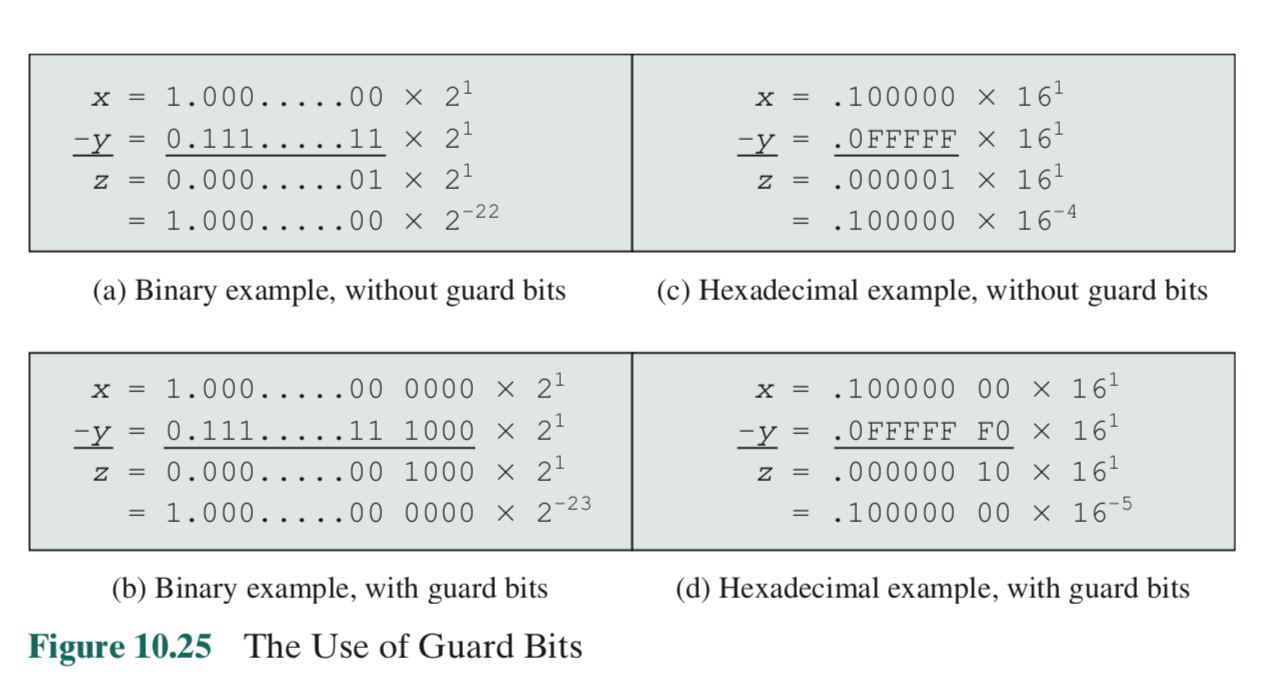
### 精度考虑

保护位

我们提到过，在浮点操作之前，每个操作数的阶值和有效值要被加载到算数逻辑单元的寄存器中。有效值的装入情况是，寄存器的长度几乎总是大于有效值位长与一个隐藏位之和。寄存器包含这些附加位，被称为保护位(guard bits)，有效值装入时，这些位以0填充，用于扩展有效值的右端。

使用保护位的原因如图10.25所示。考虑数字IEEE格式，它具有24位有效值，包括二进制小数点左侧的一个隐藏位。x = 1.00…00×21和y = x = 1.11…11×20是两个数值非常接近的数字。如果从较大的数中减去较小的数，它必须向右移动1位以对齐阶值。这如图10.25a所示。在这个过程中，y失去了





1位有效数字；结果是2-22。相同的操作在图10.24b中重复进行，但此时附加了保护位。现在最低有效位不会由于对齐而丢失，并且结果是2-23，与前一个答案相比差了一半。当基数为16时，精度的损失可能会更大。如图10.25c和10.25d所示，结果差了16倍。

舍入

影响结果精度的另一个细节是舍入策略(rounding policy)。对有效值进行任何操作的结果通常存储在较长的寄存器中。当结果转化到浮点格式时，必须消除多余的位，这样才能产生更精确的结果。这个过程叫作**舍入**。

已经开发了多种技术用于舍入处理。事实上，IEEE标准列出了四种可供选择的方法：

* + **就近舍入**(round to nearest)：将结果舍入到最近的可表示数。
  + **朝+∞舍入**(round toward +∞)：结果朝正无穷大方向向上舍入。
  + **朝-∞舍入**(round toward -∞)：结果朝负无穷大方向向下舍入。
  + **朝0舍入**(round toward 0)：结果朝0舍入。

让我们依次考虑这些策略。**就近舍入**是标准中列出的默认舍入方式，其定义如下：最接近无限精确结果的可表示值将被选择。

比方说，如果超出可以保存的23位的多余位是10010。那么多余位的值超过了最低可表示位值的一半。在这种情况下，正确的答案是最低可表示位加1，即四舍五入到下一个可表示位。现在，考虑多余的位是01111。在这种情况下，多余位的值小于最低可表示位值的一半。正确的答案是简单地删除多余的位（截断, truncate），这具有向下舍入到下一个可表示数的效果。

该标准还解决了多余位是10000……这种特殊情况的处理。这时的结果恰好介于两个可能的可表示值的正中点。一种可选的方法是截断，因为这种操作最简单。然而，这个简单方法的缺点是，它会给一个计算序列带来一个小的但可累积的偏差。我们需要的是一个没有偏差的舍入方法。一种可能的方法是根据随机数进行向上或向下舍入，这样平均来说，结果是没有偏差累积效应的。反对这种方法的理由是它不能产生可预测的、确定性的结果。IEEE标准采用的方法是强制结果为偶数：如果计算的结果恰好在两个可表示数字的正中间，则当结果的最低可表示位是1时，结果向上入；如果最低可表示位是0，则该值不向上舍入。

接下来的两个可选方法是**舍入到正或负无穷大**，它在实现一种称为区间算术的技术时很有用。区间算法通过为每个结果产生两个值，为监测和控制浮点计算中的误差提供了一种有效的方法。这两个值对应于包含真实结果的区间的上下两端。区间宽度，即上下两端之差，指示结果的精确度。如果一个区间的端点不可表示，那么该区间的端点分别执行舍入处理。虽然区间的宽度可以根据实现而变化，但是已有许多算法可以产生窄的区间。如果上界和下界之间的范围足够窄，则可获得了一个足够精确的结果。如果不是这样，至少我们知道这一事实，并且可以基于此进行进一步的分析。

标准中规定的最后一项舍入技术是**朝0舍入**。实际上，这是简单的截断：忽略多余的位。这当然是最简单的技术。然而，结果是，被截断值的幅值大小总是小于或等于更精确的原始值，从而在操作中向零引入一致的偏差。这是一个严重的偏差，因为它对任何产生非零多余位的运算都有影响。

### 二进制浮点运算的IEEE标准

IEEE 754超出了格式的简单定义，同时制定了特殊情况及其相应的处理方法，使得浮点算数可以产生独立于硬件平台的统一的、可预测的结果。这其中的一个方面已经讨论过了，即舍入处理。本小节将介绍其他三个论题：无穷大、非数(NaN)和非规格化数。

无穷大

无穷大在实数算术中被当作限界来对待，对无穷大值可以给出以下解释：

-∞ < （任何有限的数） < +∞

除了后面讨论的特殊情况外，任何涉及无穷大的算术运算都会得到明确的结果。

例如：

5 +（+∞）＝ +∞ 5 ÷（+∞） = +0

5 -（+∞）＝ -∞ （+∞）+（+∞） = +∞

5 +（-∞）＝ -∞ （-∞）+（-∞） = -∞

5 -（-∞）＝ +∞ （-∞） -（+∞） = -∞

5 ×（+∞）＝ +∞ （+∞） -（-∞） = +∞

静默式和通知式非数

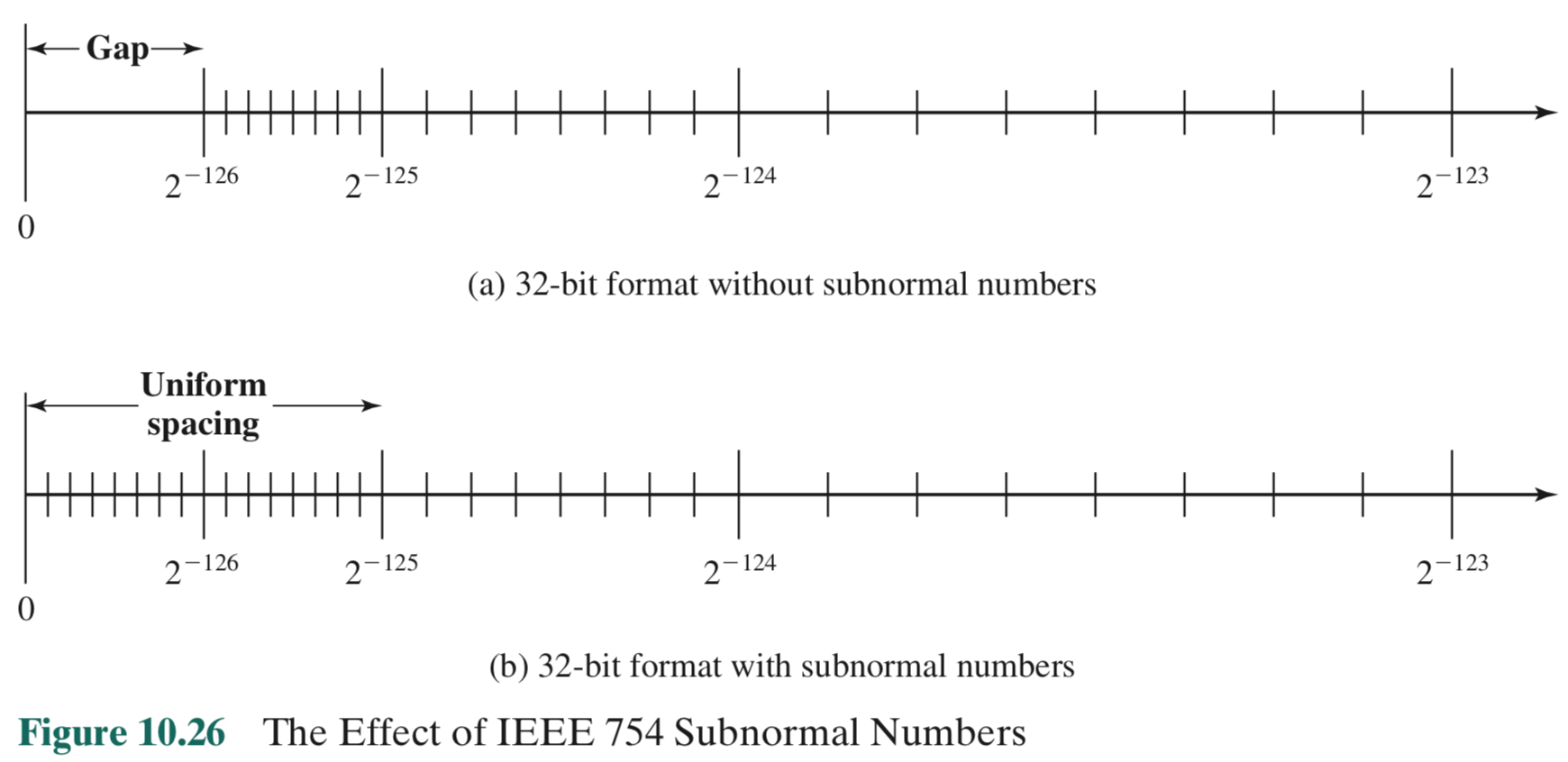
非数(NaN)是以浮点格式编码的符号实体，有两种类型：静默式和通知式。通知式(signaling)NaN在每次它作为一个操作数出现时，就产生一个无效操作异常的通知。通知式NaN可以为未初始化的变量，以及不属于标准的算数类增强，提供赋值。静默式(quiet)的NaN可以通过几乎每个算术操作，而不给出异常通知。表10.7指出了那些可以产生静默式NaN的操作。

注意，这两种类型的NaN具有相同的一般格式（如表10.4）：全1的阶值和非0的有效值。非0有效值的实际位模式取决于具体实现；有效值部分可用于区分静默式的NaN和通知式的NaN，并指定具体的异常条件。

非规格化数

在IEEE 754中包含了一种非规格化数(denormalized number)的情况，可用以处理阶值下溢的情况。当结果的阶值太小(大幅值的负阶值)时，通过右移进行非规格化，每次右移阶值增一，直到阶值落在可表示的范围之内。

图10.26说明了加入非规格化数后的效果。可表示数可以以[2n，2n+1]的区间分组。每个区间内，数



的阶值部分保持不变，而有效值变动，在区间内产生一致间隔的可表示数。当我们接近零时，每个后面的区间是前一区间宽度的一半，但是包含相同数量的可表示数。因此，可表示数的密度随着我们接近零而增加。然而，如果只使用规格化数，那么在最小规格化数和0之间存在一个间隙被浪费了。以32位IEEE 754格式而言，每个区间有232个可表示的数，最小可表示正数是2-126。使用非规格化数后，223-1个附加的数以一致的密度填加到0和2-126之间。

非规格化数的使用，被称为逐渐下溢(gradual underflow) [COON81]。在没有非规格化数的情况下，最小可表示的非0数与0之间的间隙，比最小可表示非零数和下一个更大非0数之间的间隙大得多。逐级下溢填补了这一间隙，并将阶值下溢的影响降低到与规格化数间的舍入相当的级别上。

## 关键术语、思考题和习题

|  |  |  |
| --- | --- | --- |
| Arithmetic and logic unit(AlU) ：算数逻辑单元 | minuend：被减数 | radix point：小数点 |
| arithmetic shift：算数移位 | multiplicand：被乘数 | range extension：数位扩展 |
| base：基值（或称为底） | multiplier：乘数 | remainder：余数 |
| biased representation：移码表示法 | negative overflow：负上溢 | rounding：舍入 |
| dividend：被除数 | negative underflow：负下溢 | sign bit：符号位 |
| divisor：除数 | normal number：规格化数 | sign-magnitude representation：符号-幅值表示法 |
| exponent：阶值（指数） | ones complement representation：1的补码（反码）表示法 | significand：有效值 |
| exponent overflow：阶值上溢 | overflow：上溢（或溢出） | significand overflow：有效值上溢 |
| exponent underflow：阶值下溢 | partial product：部分积 | significand underflow：有效值下溢 |
| fixed-point representation：定点表示法 | positive overflow：正上溢 | subnormal number：非规格化数 |
| floating-point representation：浮点表示法 | positive underflow：正下溢 | subtrahend：减数 |
| guard bits：保护位 | product：乘积 | twos complement representation：2的补码表示法 |
| mantissa：尾数（有效值） | quotient：商 |  |

**思考题**

10.1 简要解释以下表示法：符号-幅值、2的补码、移码。

10.2 下面三种表示法如何确定一个数字是否为负：

符号-幅值、2的补码、移码

10.3 简要说明符号-幅值表示法的缺点。

10.4 在2的补码表示法中如何求得一个整数的负数？

10.5 一般来说，什么情况下对一个n位整数的求2的补运算时会产生相同的整数？

10.6 一个数的2的补码表示和一个数的2的补码的区别是什么？

10.7 如果我们将一个2的补码表示的数当作无符号整数处理来做加法运算，则如果将结果解释为2的补码数，则结果是正确的。这对于乘法是不成立的。为什么？

10.8 浮点表示法中，一个数字的四个基本元素是什么？

10.9 浮点数的阶值部分使用移码表示法有什么好处？

10.10 正下溢、阶值下溢和有效值下溢有什么区别？

10.11 浮点加减法的基本要素是什么？

10.12 给出使用保护位的理由。

10.13 列出并简要讨论两种不同类型的NaN。

**习题**

10.1 在二进制符号-幅值表示法和2的补码表示法中分别表示以下十进制数，约定使用4位：

+4；- 6

10.2 用2的补码表示法表示以下十进制数：- 85；+ 52。

10.3 有时会遇到另一种二进制整数表示法，称为**1的补码表示法**。正整数以与符号-幅值表示法相同的方式表示。负整数是以对应正数的各位取反来表示的。

a. 以类似等式（10.1）和（10.2）的步骤，使用位的加权和给出1的补码表示法的定义。

b. 1的补码表示法表示的数字范围是什么？

c. 使用1的补码表示法，定义用于执行加法运算的算法。

注：虽然1的补码表示法在20世纪60年代从计算机硬件中消失了，但是仍然在因特网协议（IP）和传输控制协议（TCP）中用于校验和的计算。

10.4 将符号-幅值和1的补码表示法的特征添加到表10.1中。

10.5 考虑对两个二进制字的下列操作。对齐两个字的最低有效位并相加。如果有进位，则将进位加到到位左边一位。如果进位加到了字的末尾，则将该位添加到最右边的位。结果如何？

10.6 在第10.3节中，2的补码表示法定义如下。求取一个数X的2的补码，只需让X的每个位取反后加1。

1. 说明如下定义是等价的：对于n位二进制整数X，X的2的补码可通过将X看作是一个无符号整数然后计算（2n - X）获得。
2. 请通过顺时针移动来验证减法，展示图10.5也能对a.中所提出定义提供图形化的支持。

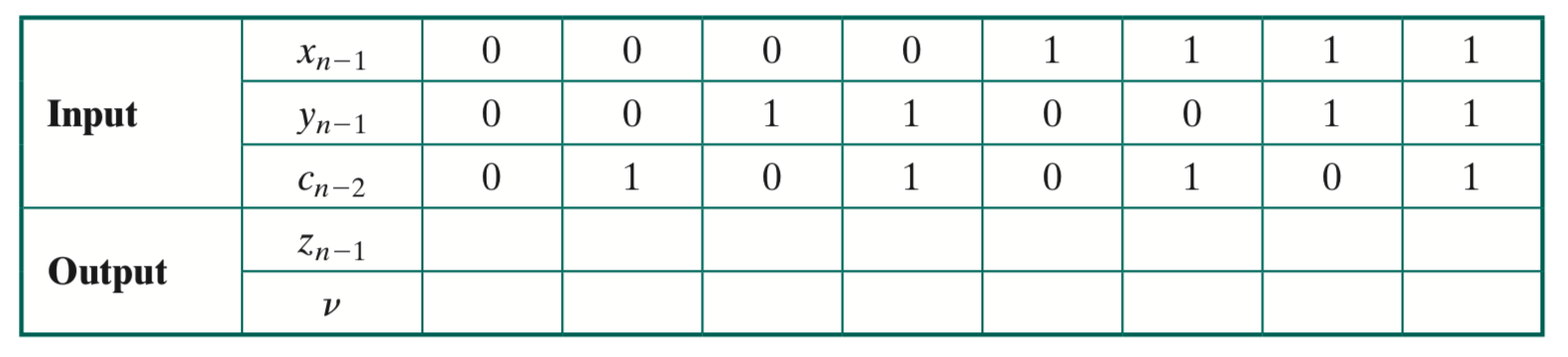
10.7 对于n位数字r进制数N，“r的补”定义为：当N≠0时，n的r的补是rn – N，当N = 0时，n的补为0。找出十进制数58230的十的补码表示。

10.8 使用10的补码算数来计算（67,350 - 23,560）。假设规则类似于2的补码表示算法。

10.9 考虑对两个n位数字的2的补码加法：



假设上式按位相加时使用了一个位间进位Ci，它是有xi 、yi和Ci-1相加产生的。令v表示溢出（v = 1）的二进制变量。填写表中的值。



* 1. 假设数字使用8位2的补码表示法表示。计算下列各式：

1. 5 + 10
2. 5 - 10
3. -5 + 10
4. - 5 - 10

10.11 使用2的补码算法，求如下的差：

a. 001000 b. 10100101 c. 100100001011 d. 10011011

- 110111 - 011110 - 101010110011 - 10100101

10.12 下面这个对上溢的定义对于2的补码表示算法是否正确？

如果向最左边的进位和最左位向上的进位的异或是1，则说明有上溢。否则，就没有。

10.13 图10.12中Q-1位的意义是什么？

10.14 给定x=0101和y=1010为两个用2的补码表示法表示的数（即，x=5，y=-6），使用布斯算法计算乘积p = x× y

10.15 使用布斯算法将23（乘数）乘以29（乘数），其中每个数字都用6位表示。

10.16 证明基数为B的两个n位数字相乘，其乘积不会超过2N个数字。

10.17 通过说明图10.15所示的除法的步骤，验证图10.16的无符号二进制除法算法的有效性。使用类似于图10.17的说明格式。

10.18 第10.3节中描述的2的补码表示整数除法算法，其被称为恢复余数算法，因为A寄存器中的值必须在减法不成功之后恢复。另一种稍微复杂一点的算法，称为不恢复余数除法，它避免了不必要的减法和加法。请为后一种方法设计算法。

10.19 在执行无符号整数减法时，如果需要从最高有效位借位，则结果是负数。这句话是真是假？

10.20 2的补码表示法中，计算-145除以13，使用12位字。请使用第10.3节中描述的算法。

10.21

* 1. 考虑一个十进制数字的定点表示法，其中隐含的小数点可以在任何位置（如在最低有效数字的右边、最高有效位数的右边，等等）。若近似表示普朗克常数（6.63×10-27）和阿伏加德罗数（6.02×1023）的近似值需要多少个十进制数字？对于两个数字，隐含的小数点必须位于相同的位置。
  2. 现在考虑十进制浮点格式，其中阶值以移码表示，偏移量为50。假设已经使用规格化数。用这种浮点格式表示以上两个常数需要多少个十进制数字？

* 1. 假设指数e被限制在0≤e≤x的范围内，偏移量为q，底为b，有效值长度为p。

1. 可以表示的最大和最小的正数是多少？
2. z作为规格化的浮点数，可以表示的最大和最小的正数是多少？

10.23 用IEEE 32位浮点格式表示以下数字：

a. -5 b.-6 c.-1.5 d.384 e.1/16 f.-1/32

10.24 以下数字使用IEEE 32位浮点格式。十进制的等值是多少？

a. 1 10000010 00100000000000000000000

b. 0 01111110 00001100110011001100110

c. 0 10000000 11001100110011001100110

10.25 考虑一种简化的7位IEEE浮点格式， 3位阶值，3位的有效值。列出所有127个值。

10.26 用IBM的32位浮点格式表示以下数字，该格式使用7位阶值，隐含基数为16，指数偏移量为64（十六进制的40）。规范化浮点数要求最左边的十六进制数字不为零；隐含的小数点在该数字的左边。

|  |  |  |  |
| --- | --- | --- | --- |
| 1. 1.0 2. 0.5 | 1. 1/64 2. 0.0 | 1. -15.0 2. 5.4 ×10-79 | 1. 7.2 × 1075 2. 65，535 |

10.27 设9ABD0000是用十六进制表示的IBM格式的浮点数。这个数字的十进制值是多少？

10.28 下述情况的偏移量分别是多少：

a. 9位字段中底为16的阶值。

b． 10位字段中底为16的阶值。

10.29 为图10.21b的浮点格式画出与图10.19b类似的数轴表示。

10.30 考虑一个浮点格式，它有8位移码阶值，23位有效值。以这种格式表示以下数字：

1. – 720
2. 0.645

10.31 正文提到，32位格式最多可以表示232个不同的数字。有多少不同的数字可以用IEEE 32位格式表示？

10.32 在计算机中使用的浮点表示法只能精确地表示某些实数，其他浮点数必定是近似的。如果A’是与实际值A近似的值，那么相对误差r表示为

r =

用下列浮点格式表示十进制数 +0.2，并计算相对误差

* 底为2
* 阶值：移码，4位；
* 有效值：9位。

10.33 如果 P = 9.938，如果P被截断到9.93，或被舍入到9.94，请计算相对误差。

10.34 当人们谈到浮点运算的不准确性时，他们常常把误差归因于相等数量的减法运算中出现的抵消。但是当X和Y近似相等时，可以精确地求出差值X - Y，没有误差。人们到底是指的是什么呢？

10.35 数值A和B作为近似A’和B’存储在计算机中，忽略任何可能的截断或舍入误差，表明乘积的相对误差近似为因子中相对误差的总和。

10.36 在计算机计算中最严重的错误之一发生在减去两个几乎相等的数字时。考虑A＝0.07199，B＝0.07119。计算机将所有值都调整为四位十进制数字。因此A’＝0.0719和B’＝0.0711。

a. A’和B’的相对误差是多少？

B.C’ ＝ A’ — B’的相对误差是多少？

10.37 为了了解非规格化和逐级下溢的影响，考虑一个十进制系统，它的有效数为6位十进制数，最小规格化数是10-99。一个规格化数要求在十进制小数点左边有一个非零的十进制数。执行以下计算并对结果进行非规格化。对结果进行解释。

a. （2.50000×10-60）×（3.50000×10-43）

b. （2.50000×10-60）×（3.50000×10-60）

c. （5.67834×10-97）-（5.67812×10-97）

10.38 显示如何执行以下浮点加法（其中将有效值截断到3位十进制数）。以规格化形式表示结果。

a.（9.123×10-1）+（8.432×10-3）

b.（9.123×10-4）+（0.438×104）

10.39 显示如何执行以下浮点减法（其中将有效值截断到3位十进制数）。以规格化形式表示结果。

a. (9.123×10-2) - (8.432×10-3)

b. (9.123×102) - (0.438×102)

10.40 显示如何执行以下浮点乘法（其中将有效值截断到3位十进制数）。以规格化形式表示结果。

1. （9.123×10-1）×（0.432×101）
2. （9.123×100）÷（0.438×101）