目　录

第一章　计算机算术运算和数的系统 ········· 1
1.1　计算机算术运算概论 ········· 1
1.2　机器算术运算的数的系统 ········· 3
1.3　算术运算操作的分类 ········· 4
1.4　普通基数的数的系统 ········· 6
1.5　带符号数字的数的系统 ········· 7
1.6　算术运算的算法及其实施 ········· 10
1.7　运算处理器的规格 ········· 12
1.8　定点数的表示方法 ········· 13
1.9　浮点数的表示方法 ········· 15
1.10　多倍精度的算术运算 ········· 18
1.11　参考文献注释 ········· 18
第二章　集成电路和数字器件 ········· 22
2.1　电子工艺和逻辑电路系列 ········· 22
2.2　多路转换器和分路转换器 ········· 23
2.3　基本的加法/减法逻辑 ········· 29
2.4　求补器和数值比较器 ········· 30
2.5　模块式阵列乘法器 ········· 34
2.6　多功能寄存器的特点 ········· 36
2.7　通用寄存器的设计 ········· 39
2.8　半导体随机存取存储器和只读存储器 ········· 40
2.9　可编程序逻辑阵列（PLA） ········· 42
2.10　磁泡存储器（MBM） ········· 45
2.11　参考文献注释 ········· 47
第三章　快速的双操作数加法器/减法器 ········· 51
3.1　引言 ········· 51
3.2　基本的带符号补数和带符号数值的加法器 ········· 51
3.3　非同步自定时的加法 ········· 55
3.4　进位完成检测加法器 ········· 57
3.5　条件和加法法则 ········· 57
3.6　条件和加法器的设计 ········· 59
3.7　进位选择加法器 ········· 60
3.8　进位产生传播及先行功能 ········· 63

3.9 先行进位加法器 …… 65
3.10 各种双操作数加法器的评价 …… 68
3.11 参考文献注释 …… 69
第四章 多操作数加法器,带符号数字算术运算以及运算逻辑部件 …… 72
4.1 引言 …… 72
4.2 多操作数进位存储加法器 …… 72
4.3 多级进位存储加法器 …… 74
4.4 按位划分的多操作数加法 …… 76
4.5 按位划分的多操作数加法器 …… 78
4.6 带符号数字加法/减法 …… 79
4.7 全并行 SD 加法器/减法器 …… 81
4.8 多功能算术运算逻辑部件 …… 83
4.9 ALU 的系统应用 …… 86
4.10 实例研究 I:SN74181 ALU 的设计和应用 …… 88
4.11 参考文献注释 …… 92
第五章 标准的和再编码的乘法器 …… 95
5.1 引言 …… 95
5.2 间接的乘法算法和硬件 …… 95
5.3 一个间接的通用乘法器的设计 …… 99
5.4 乘法中的多次移位 …… 103
5.5 重叠的多位扫描 …… 104
5.6 采用重叠扫描的乘法器设计 …… 106
5.7 典型的乘数再编码 …… 109
5.8 串再编码和 Booth 乘法器 …… 111
5.9 各种加法-移位乘法器的评价 …… 114
5.10 参考文献注释 …… 115
第六章 叠接单元的阵列乘法器 …… 118
6.1 阵列乘法的基础 …… 118
6.2 模块乘法与华莱士(Wallace)树 …… 121
6.3 直接的补码乘法 …… 125
6.4 Pezaris 阵列乘法器及其修改方案 …… 128
6.5 Baugh-Wooley 补码乘法器 …… 131
6.6 通用乘法阵列 …… 135
6.7 可编程序的相加乘法模块 …… 138
6.8 通用乘法网络 …… 142
6.9 再编码阵列乘法 …… 144
6.10 利用 ROM-加法器的乘法网络 …… 147
6.11 对数乘法/除法的原理 …… 151
6.12 参考文献注释 …… 153

第七章 标准的和高基数的除法器 ······ 156
7.1 引言 ······ 156
7.2 基本的“减法-移位”除法的特性 ······ 156
7.3 常规的带恢复的除法 ······ 158
7.4 二进制带恢复的除法器的设计 ······ 159
7.5 二进制不恢复除法 ······ 162
7.6 高基数不恢复除法 ······ 163
7.7 **SRT** 除法的原理 ······ 165
7.8 修改的二进制 **SRT** 除法 ······ 168
7.9 Robertson 的高基数除法 ······ 172
7.10 实例研究 II:ILLIAC-III 的算术运算处理器 ······ 175
7.11 参考文献注释 ······ 181
第八章 收敛除法与单元阵列除法器 ······ 184
8.1 引言 ······ 184
8.2 收敛除法的方法 ······ 184
8.3 采用乘法的除法器的设计 ······ 186
8.4 通过对除数求倒数实现除法 ······ 190
8.5 使用 CSA 树的二进制倒数器 ······ 191
8.6 带恢复的单元阵列除法器 ······ 193
8.7 不恢复的单元阵列除法器 ······ 196
8.8 进位-存储的单元阵列除法 ······ 198
8.9 先行进位的单元阵列除法器 ······ 200
8.10 各种单元阵列除法器的评价 ······ 202
8.11 参考文献注释 ······ 205
第九章 规格化的浮点运算处理器 ······ 207
9.1 浮点运算的基本理由 ······ 207
9.2 基数的选择和浮点的特殊性 ······ 208
9.3 规格化浮点运算操作 ······ 210
9.4 基本的浮点运算硬件 ······ 212
9.5 规格化浮点加法/减法 ······ 214
9.6 规格化的浮点乘法 ······ 218
9.7 规格化的浮点除法 ······ 220
9.8 实例研究 III:IBM **360** 系统 **91** 型的浮点运算处理器 ······ 223
9.9 参考文献注释 ······ 230
第十章 关于浮点运算的新课题 ······ 233
10.1 引言 ······ 233
10.2 非规格化浮点运算 ······ 233
10.3 截断和舍入操作 ······ 235
10.4 公理化舍入理论 ······ 237

10.5 浮点运算的误差分析 …… 238
10.6 关于浮点算术运算操作中的误差边界 …… 240
10.7 一个以 ROM 为基础的舍入方案 …… 243
10.8 浮点指令的比较 …… 246
10.9 多倍精度的浮点加法/减法 …… 250
10.10 多倍精度的浮点乘法和除法 …… 252
10.11 参考文献注释 …… 255
第十一章 基本函数,流水线算术运算和误差控制 …… 258
11.1 引言 …… 258
11.2 二进制数的平方根和平方 …… 258
11.3 基本函数的多项式计算 …… 263
11.4 Walther 统一 CORDIC 计算技术 …… 265
11.5 Chen 氏收敛计算方法 …… 268
11.6 流水线计算的概念 …… 272
11.7 实例研究 IV: 在 TI 公司先进科学计算机中的流水线运算 …… 274
11.8 通用的多功能运算流水线 …… 278
11.9 流水线快速傅里叶变换处理器 …… 281
11.10 运算处理器中的误差控制 …… 284
11.11 实例研究 V:CDC 公司 STAR-100 流水线运算处理器 …… 288
11.12 参考文献注释 …… 292

第一章　计算机算术运算和数的系统

1.1　计算机算术运算概论

算术运算在人类文明中，尤其在科学、工程和工艺技术领域中起着重要的作用．机器的算术运算可以追溯到公元前五百年在中国采用算盘的形式． 在整个计算机的历史中，运算处理器是计算机的主要工作力量． 运算处理器用来执行算术运算操作，并且为所要计算的问题产生数值解．由于中/大规模集成（MSI/LSI）电路的出现，愈来愈多的高级复杂的运算处理器已成为今日高性能数字计算机系统标准硬件的特点．一个现代化的数字计算机可以配备多个硬件运算处理器，这些处理器可以用来处理不同格式的数据，或者求解通用的和专用的特殊算术运算函数．这些硬件运算处理器提供比一般计算手段更快的计算速度．选择一个合适的算术运算系统，对计算机结构设计和程序设计两者都有重要影响．

令 $\mathbf{R}$ 是所有实数的集合．一个**实数算术运算**可以定义为一个代数映象

$$\mathbf{f}:\mathbf{R}\times\mathbf{R}\rightarrow\mathbf{R} \tag{1.1}$$

式中 $\mathbf{f}$ 的典型例子就是标准的两个操作数的算术运算操作 $+$，$-$，$\times$，$\div$．令 $\mathbf{M}$ 是机器表示数的集合．在集合 $\mathbf{M}$ 中每个数只能有有限数目的数位． 集合 $\mathbf{M}$ 必须是 $\mathbf{R}$ 的一个合适的有限子集，即 $\mathbf{M}\subset\mathbf{R}$． 一个**机器的算术运算**可以用以下映象建立模型：

$$\mathbf{g}:\mathbf{M}\times\mathbf{M}\rightarrow\mathbf{M} \tag{1.2}$$

机器算术运算与实数算术运算不同之处在于数的精度的基本结果．用运算处理器只能执行有限精度的计算，而实数运算可以产生字长无限制的任意精度的结果．

换句话说，我们可以把机器算术运算看成是一种具有恰当舍入控制的近似的实数算术运算． 用函数映象表示，以上机器算术运算函数 $\mathbf{g}$ 与实数算术运算函数 $\mathbf{f}$ 之间的关系，可用下列合成函数联系起来

$$\mathbf{g}=\mathbf{h}\circ\rho \tag{1.3}$$

映象 $\mathbf{h}$ 定义为

$$\mathbf{h}=\mathbf{f}|\mathbf{M}\times\mathbf{M}\rightarrow\mathbf{R} \tag{1.4}$$

实数 $\mathbf{f}$ 限制在机器域（Machine Domain）$\mathbf{M}\times\mathbf{M}$ 以内．定义为下式的映象

$$\rho:\mathbf{R}\rightarrow\mathbf{M} \tag{1.5}$$

是所选择的舍入方案，它将使任意长度的结果修整为一个机器可表示的数．注意，数字计算机与一般普通的算术运算设备对比，在给定足够的存储容量的情况下，可以执行任意精度的计算．有限精度的结果，只是指在受限制的字长（在寄存器或加法器）和局限的存储容量的机器中所得的结果．

数字计算机的算术运算是一个涉及数字计算机的系统结构和逻辑设计的领域．因此算术运算的设计要包括算术运算算法的发展及其逻辑上的实现．在计算机结构领域它与

实现算术运算操作的效率互相有关．在数值分析领域它又关系到近似实数算术运算的精度．图1.1示出影响计算机算术运算系统设计与其它有关计算机学科的相互作用关系．可以说，算术运算系统的设计涉及到计算机工程的许多领域，它是计算机科学和数字工程的混合学科．

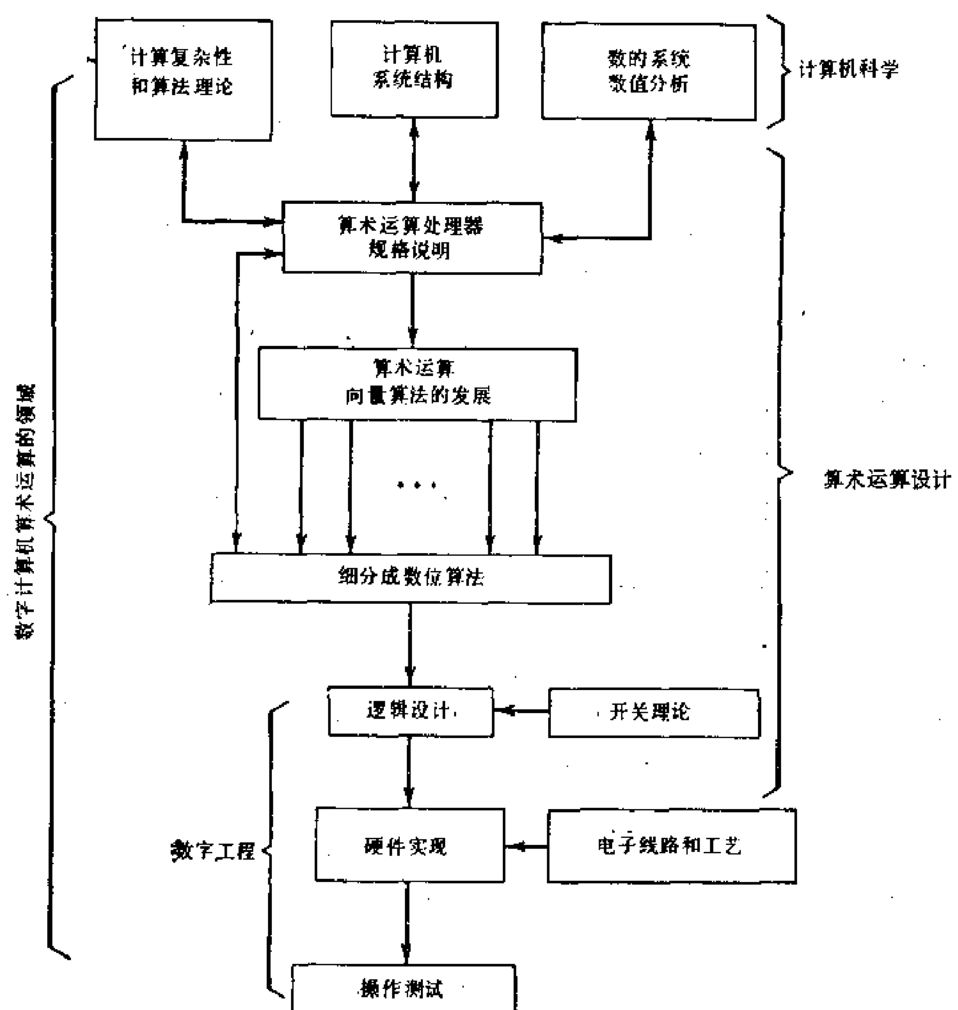


图1.1 算术运算设计与其它计算机科学和数字工程学科的相互关系

本书描述计算机运算系统的四个主要领域的原理和设计技术：基数算术运算，带符号数字的算术运算，初等函数计值以及流水线算术运算．第一和第二章叙述数的系统和数字电路工艺，并为以下章节提供基础．基数算术运算，正如大多数计算机中遇到的，可分成定点的和浮点的系统．在阐明标准的和非标准的加法、减法、乘法和除法的算法以及它们如何实现时，更强调用新的方法说明，诸如多操作数进位存储加法器，乘数再编码，收敛和SRT除法，单元阵列乘法器和除法器，带符号数字的算术运算以及其他．对浮点算术运算从初级的到高等的水平都将予以说明．最后一章讨论初等函数和流水线算术运算的设计．

笔者建议有兴趣的读者查阅每章最后的文献目录单和参考书单，可以得到进一步资料来源．这些材料的大多数是选自一般杂志或会议刊物．习题的安排将有助于学生消化

本书的要点，并希望能激励学生们发展自己新的和改进的技术.

1.2 机器算术运算的数的系统

在数字计算机中，算术运算算法的实现，在很大程度上取决于数值数据是如何存储在存储器和寄存器中的. 不同的内部数的表示方法会产生不同的运算硬件的设计. 选择一种合适的数的系统会影响从设计者角度看来的计算机结构，同时也影响用户所使用的数值分析方法.

因为在数字计算机中只实现有限精度的算术运算，所有允许的数值表示必须限制在有限的字长内. 用机器实现具有有限精度的实数算术运算操作是由于字长受到限制. 选择一个好的算术运算系统和内部数的表示方法会影响机器操作的有效实现以及近似的实数算术运算的精度这两个方面.

算术运算的设计者在解决范围很广的应用问题时，必须注意时间和空间效率结构方面的现实性以及提供充分精度的数值分析方面的现实性. 一般机器的算术运算中，数的系统可分为以下五种范畴.

普通基数的数的系统

普通计算机采用基数 $r \geqslant 2$ 的**固定基数**的算术运算系统，而且数字的集合为:

$$\{0, 1, \cdots, r-1\} \tag{1.6}$$

一个数的所有数字都用正的权，而每个数是单值地表示的. 有些特殊系统可能采用**混合基数**的数，其中一个数的不同数字位采取不同的基本数值. 我们将在第 1.4 节正式讨论固定基数的数的系统.

带符号数字的数的系统

这种系统可以认为是固定基数系统的一种扩展情况，其中正权和负权的数字都允许在一个数字集合中

$$\{-\alpha, \cdots, -1, 0, 1, \cdots, \alpha\} \tag{1.7}$$

其中 α 是一个有界的正整数. 对于一个给定的数值，带符号数字表示法可能不是单值的. 因为对于一个给定的数，可以有多于一个的带符号数字的表示方法，这种数的系统可以认为是冗余的. 冗余的带符号数字的数的系统将在第1.5节中详细讨论.

残数的数的系统

一个残数的每个数位并不赋予权的因数；因此，数字的次序对于决定这个数的值来说是不重要的. 而且混合基数也可赋给不同数位. 一个残数 $\mathbf{x}$ 是用一个 n 重元表示的整数

$$\mathbf{x} = \{r_1, r_2, \cdots, r_n\}_m \tag{1.8}$$

相对应于另一个 n 重元

$$\mathbf{m} = \{m_1, m_2, \cdots, m_n\} \tag{1.9}$$

每个 r_i 叫做模数为 m_i 的 $\mathbf{x}$ 的残数，其中所有 n 模值 $\{m_i | i = 1, 2, \cdots, n\}$ 双双相对地互为素数. 所有这 n 个残数数字 r_i，$i=1, 2, 3, \cdots, n$，可以各自独立地处理. 由于这一

点，在残数算术运算中，加法和乘法是固有地无进位的．残数算术运算最先是由 Svoboda[16]提出的，是用来设计可靠的计算系统时减少算术运算的错误．建议读者可阅读 Szabo 和 Tanaka[15]，以便了解残数算术运算及其应用．

有理数的系统

这种算术运算系统所表示的数值量，如同按照分子-分母整数对来表示分数．有理数的加法、减法、乘法和除法永远产生有理数，所以这些操作是可以有限制的而无需采用无限的精度，如数 $\frac{1}{3}=0.3333\cdots$．但实际的限制使得这种理想情况难以处理，即其中分子和分母就是在中等规模的计算中也会变得很大．

曾经建议对这种近似算术运算设计范围的研究只在于一个有限精度的有理系统中，其中精度的限制是通过约束分数的分子和分母大小来实现的．这种有限精度的约束可以分别或共同做，即限制分子和分母数位的数目，产生名称为固定分切或浮动分切的有理数系统．就考虑其实现来说，有理算术运算还只处在假设阶段．有兴趣读者如要进一步研究，可参阅 Matula[12] 以及 Hwang 和 Chang[7] 的著作．

对数的数的系统

这个系统采用一个实数 $\boldsymbol{\mu}>1$ 作为基数．实数的集合是用下列对数空间 $\mathbf{L}_\mu$ 来定义：

$$\mathbf{L}_\mu=\{\mathbf{x}\,|\,|\mathbf{x}|=\boldsymbol{\mu}^i,\ i\ \text{是整数}\}\cup\{0\} \tag{1.10}$$

应用指数表达式的表示方法就是建议用几何的舍入法，而不是用算术的舍入法，以便提高数的精度．通过整数算术运算和对数-反对数子程序以实现对数算术运算的详细论述，见 Marasa[10]．

本书中描述的算术运算设计主要以普通基数的数的系统或者以冗余带符号数字的数的系统为基础．残数，有理数和对数的数的系统不在本书范围内．

1.3 算术运算操作的分类

从用户和设计者的角度来看，在现代数字计算机中算术运算指令一般可以分成三种类型．

标准算术运算操作

这一类型主要包括四种简单的算术运算功能：定点或浮点形式的加法，减法，乘法和除法．其它所有的数学函数都可以用这四种标准操作来表示．下面我们主要区别两种操作形式．详细的含意将在以后章节中讨论．

(1) **定点** （以后简称 **FXP**）算术运算．通常用于基数点固定的数据中，如商业或统计计算中遇到的题目．**FXP** 操作还可根据其基数点出现的位置进一步分成两小类．在整数算术运算中所有结果都在寄存器的右端排列起来，如同在最右端有一个基数点．而在分数算术运算中，所有结果不管其长度如何，都是在寄存器的左端排列．

（2）**浮点**（以后简称 FLP）算术运算．主要在科学和工程计算中使用，其中需要经常改变数值大小的比例尺度． FLP 操作也可按照其数据格式分成两类． 当必须使用规格化的数据操作数时，我们称为**规格化 FLP 算术运算**操作． 当操作数在中间和最后过程中无需规格化，则称为**非规格化 FLP 算术运算**操作．目前使用的大部分计算机都采用规格化操作．

基本算术运算函数

基本函数指的是那些在数学计算中经常用到的特殊算术运算操作．这些函数包括指数、平方根、对数、三角、双曲函数及其他．但并非所有计算机都把这些函数作为标准硬件特征加以包括． 目前大部分计算机是用软件或固件例行子程序来实现这些基本函数的．随着硬件价格的下降，正在愈来愈普遍地采用专用的硬件函数求值器来产生这些基本函数．

伪算术运算操作

这种类型的操作需要一定程度的算术计算，但主要是在执行计算机程序时作为专门的应用．下面描述伪算术运算操作的两类．

（1）**地址运算**的执行基本上是为了计算有效的存储器地址，诸如变址，间址，相对地址或偏置编址方式．

（2）**数据编辑运算**包括逻辑上字母符号和数据传送操作，诸如比较，求补，输入，存储，组合，拆分，移位，规格化及其他． 这些操作用来把数据从一种格式变换到另一种，检查与源格式的一致性，或者为控制程序的顺序而作的测试．

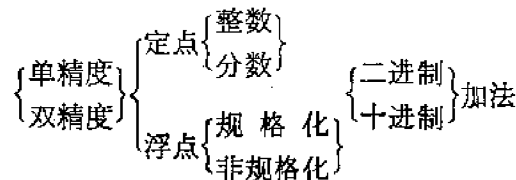
我们主要讨论标准的定点和浮点运算以及某些基本函数的计算．地址运算和数据编辑操作并不是我们的主要兴趣．本书中提出的理论和设计技术集中在前两种算术运算操作．在很多计算机中，算术运算指令是按照执行的数据精度来细分的．多倍精度的运算对定点和浮点操作都可应用．

（1）**单精度（SP）运算**指的是这样一些操作，其中所用的标准数据操作数的字长是和一个存储器字的字长相等的．

（2）**双精度（DP）运算**使用的每个操作数的字长加了一倍．三倍字长或更高精度的操作可以相似地定义．

某些计算机器提供分开的运算硬件来处理二进制和十进制数据，如 IBM 360 系统．在这些机器中，可以直接规定对十进制操作数进行操作．通常需要有码的转换，组合或拆分指令来直接处理十进制格式的数据．

表 1.1 对以上算术运算操作的分类进行总结． 举例说，一个加法（ADD）操作可以分成 16 种，如下所示：



在一个数字计算机中，加法操作的以上 16 种规定取决于数的精度，整数或分数的定点运

算,规格化或非规格化的浮点运算,以及对二进制还是对十进制数据进行操作.

假如对硬件的成本没有限制，而其应用确实需要大量的硬件运算功能以加快处理速度,那么我们还可以设计一个算术运算部件来处理实数或复数,或者设计某些专用的硬件机器来做一些专门的算术计算,诸如超高速傅里叶变换 (SFFT)，矩阵处理器及其他. 随着工艺的进步，这种高级专用的运算机器确实是可行的. 本书所谈到的设计技术为制造专用或通用计算机器提供了必要的知识.

表 1.1 算术运算操作的分类

标准算术运算操作	基本算术运算函数	伪算术运算操作
加法，减法，乘法和除法的操作方式：单精度、双精度、定点整数、定点分数、规格化浮点、非规格化浮点、二进制运算、十进制运算或高基数运算、负操作数	指数、对数、平方根、立方根、平方、立方、高次幂或高次根、三角和双曲线函数、特殊多项式、超越函数	**数据编辑运算**：比较、范围、求补、求反、输入、存储、组合、拆分、增量、减量、移位、旋转、规格化 **地址运算**：相对寻址、变址、基数寻址、偏置寻址

1.4 普通基数的数的系统

大多数现有的算术运算处理器的基础是具有普通基数的数的系统. 只有很少的机器用冗余带符号数字的算术运算[24]. 其余三种算术运算系统采用残数，有理数和对数的数的系统,目前尚未广泛应用.

一个以 r 为基数的数 $\mathbf{X}$ 在数字计算机中是用 $(n+k)$ 重数字向量来表示的

$$\mathbf{X}=(x_{n-1},\cdots,x_0,x_{-1},\cdots,x_{-k}),\tag{1.11}$$

其中每个分量 x_i (对于 $-k\leqslant i\leqslant n-1$) 称为向量 $\mathbf{X}$ 的第 i 个数字. 每位数字可以有 r 个不同的值

$$\{0,1,\cdots,r-1\}\tag{1.12}$$

其中 r 是数的系统的基数. 在一个固定基数的数的系统中,所有各位数字都具有相同的基数值. 常用的十进制数的表示法属于 $r=10$ 的范畴.

数 $\mathbf{X}$ 的整数部分为前 n 个数位 $(x_{n-1},\cdots,x_1,x_0)$，而其余 k 个具有负下标的数位 $(x_{-1},x_{-2},\cdots,x_{-k})$ 形成数 $\mathbf{X}$ 的分数部分. 基数点 (即通常所说的小数点) 是用来分隔这两部分的. 在实际的计算机线路中,这个基数点是不标明的; 就是说，它在存储设备中并不占有一个物理位置. 我们将讨论基数为正整数 $r\geqslant 2$ 的数系统. 以负数，分数和复数为基数的数系统不在本书的讨论范围内.

混合基数的数是指这样一种数，它在不同数位上具有不同的基数值. 我们用来计时的方法(小时,分,秒)就是采用了混合基数 (24, 60, 60). 在具有权的基数的数系统中,我们把每个数字向量 $\mathbf{X}$ 指定一个单值,表示为

$$\mathbf{X}_v=\sum_{i=-k}^{n-1}x_i\cdot\omega_i\tag{1.13}$$

其中每个 ω_i 称为第 i 位数字的加权因子. $n+k$ 个加权因子形成一个权向量,表示为

$$\mathbf{W}=(\omega_{n-1},\cdots,\omega_0,\omega_{-1},\cdots,\omega_{-k})\tag{1.14}$$

数 $\mathbf{X}$ 的值可以用 $\mathbf{X}\cdot\mathbf{W}$ 求出,即两个向量 $\mathbf{X}$ 和 $\mathbf{W}$ 的点积. 特别是权向量

$$\mathbf{W} = (r^{n-1}, \cdots, r^0, r^{-1}, \cdots, r^{-k}) \tag{1.15}$$

将引出一个基数为 r 的数 $\mathbf{X}$ 的常用记数表示法，其值

$$\mathbf{X}_v = \mathbf{X} \cdot \mathbf{W} = \sum_{i=-k}^{n-1} x_i \cdot \omega_i = \sum_{i=-k}^{n-1} x_i \cdot r^i \tag{1.16}$$

实际上，经常使用的有四种基数的数系统，即分别相应于基数值 $r = 2, 8, 10$ 和 16 的二进制，八进制，十进制和十六进制的数系统。

基数值 r 愈高，需要有更多的二进位数字（位数）来为每个基数 r 的数字进行编码。通常，至少要有 k 位数来为一个基数 r 的数字编码，其中

$$k = \lceil \log_2 r \rceil \tag{1.17}$$

符号 $\lceil x \rceil$ 表示不小于实数 x 的最小整数。举例说，我们想用 $k = \lceil \log_2 10 \rceil = 4$ 个二进位来对每个十进制数字（$r = 10$）编码。用于表示十进制数字的若干种 4 位二进制码见表 1.2，其中二-十进制和葛莱码（Gray code）是最常用的。

表 1.2　若干表示十进制数字的二进制码

十进制数字	二-十进制码 (8-4-2-1 码)	葛莱码 (Gray code)	5-2-1-1 码	余-3 码
0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 1 0	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 1 1	0 0 1 1	0 1 0 1
3	0 0 1 1	0 1 0 1	0 1 0 1	0 1 1 0
4	0 1 0 0	0 1 0 0	0 1 1 1	0 1 1 1
5	0 1 0 1	1 1 0 0	1 0 0 0	1 0 0 0
6	0 1 1 0	1 1 0 1	1 0 1 0	1 0 0 1
7	0 1 1 1	1 1 1 1	1 1 0 0	1 0 1 0
8	1 0 0 0	1 1 1 0	1 1 1 0	1 0 1 1
9	1 0 0 1	1 0 1 0	1 1 1 1	1 1 0 0

人类已经习惯于实行十进制系统，而在数字计算机中所有数据都是二进制编码的。内部二进制编码的恰当选择要立足于表示方法的效率，算术运算的设备以及操作的可靠性。

设计者必须注意内部基数转换对非二进制机器是需要的。然而，用户不必注意必要的码的转换。计算机对用户看来可以是八进制，十进制，十六进制或其它基数的机器。建议用更高基数的方法来设计高速硬件乘法和除法部件，如在 ILLIAC III 计算机中，用于乘法和除法的基数 $r = 256$[6]。

在数字计算机中，正和负的普通基数的数主要有两种不同的表示方法。方程式 1.6 所示的表示式相应于所谓定点表示法。另一种存储基数的数的方法是浮点表示法。我们将在第 1.8 和第 1.9 节中描述这两种基数的数的表示方法。

1.5　带符号数字的数的系统

冗余数的系统对于人工计算可能是不方便的，但它们在设计高速算术运算机器时是有用的。带符号数字（SD）的数的表示方法可以有冗余性。每个带符号的数字可以用几个二进位来表示它（包括每个 SD 的符号位），这样会增加数据存储空间并需要更宽的数

据总线．然而，提高速度这一点却是它的一大优点．我们将阐明使用 SD 表示法来设计第四章中的完全并行的加法器，第五章中的再编码乘法器，及第七章中的高基数除法器．

SD 数正式定义如下：给定一个基数 r，则一个 SD 数的每一个数字可具有以下 $2\alpha+1$ 个值

$$\Sigma_r = \{-\alpha, \cdots, -1, 0, 1, \cdots, \alpha\} \tag{1.18}$$

其中最大数字的数值 α 必须在以下范围内：

$$\left\lceil \frac{r-1}{2} \right\rceil \leqslant \alpha \leqslant r-1 \tag{1.19}$$

因为整数 $\alpha \geqslant 1$，必须认为 $r \geqslant 2$．为了在对称的数字集合 Σ_r 中得到最小的冗余，可选择以下值作为最大数值

$$\alpha = \left\lfloor \frac{r}{2} \right\rfloor \tag{1.20}$$

其中符号 $\lfloor x \rfloor$ 表示小于或等于实数 x 的最大整数．因此，其中如果 r_e 是偶数，$\alpha = \frac{r_e}{2}$．而如果 r_0 是奇数，则 $\alpha = \frac{(r_0-1)}{2}$．这意思是说相邻奇偶基数值可产生相同的数字集合．相应于这种 α 选择的数字集合可表示为以下两种不同方式，但实际上在 $r_0 = r_e + 1$ 时，两者是相同的集合．

$$\Sigma_{r_0} = \left\{-\frac{r_0-1}{2}, \cdots, -1, 0, 1, \cdots, \frac{r_0-1}{2}\right\}$$

$$\Sigma_{r_e} = \left\{-\frac{r_e}{2}, \cdots, -1, 0, 1, \cdots, \frac{r_e}{2}\right\} \tag{1.21}$$

举例说，基数为 2 的 SD 系统具有数字集合 $\Sigma_2 = \{-1, 0, 1\}$，而基数为 4 的 SD 系统具有数字集合 $\Sigma_4 = \{-2, -1, 0, 1, 2\}$，以此类推．一个 SD 数

$$\mathbf{Y} = (y_{n-1} \cdots y_0 y_{-1} \cdots y_k)_r \tag{1.22}$$

的代数值 $\mathbf{Y}_v$ 可以计算如下

$$\mathbf{Y}_v = \sum_{i=-k}^{n-1} y_i \cdot r^i \tag{1.23}$$

这和方程式 1.16 相似，只不过现在的 $\mathbf{Y}$ 本身可以是正的或负的，而无需一个显式符号．零只有一个唯一的表示式，即对方程式 1.22 中所有的 i，有 $y_i = 0$．一个 SD 数 $\mathbf{Y}$ 的负数 $-\mathbf{Y}$ 是通过改变 $\mathbf{Y}$ 中的所有非零数字的符号直接推导出来的．注意，数字"零"不考虑符号．实际上，只使用 2 的幂次作为基数值，即 $r = 2^k$．

使用 SD 数系统的最初目的是要去除在加法（或减法）中的进位链延迟．为了断开进位链，α 的下界应该缩紧一些

$$\left\lceil \frac{r+1}{2} \right\rceil \leqslant \alpha \leqslant r-1 \tag{1.24}$$

在第四章中我们将考虑使用的 α 在上述区间之内的 SD 加法．对于使用 SD 表示法的除法，方程式 1.19 中给出的界限是充分紧缩的．

让我们考虑若干数值例子．给定一个明确的数值 $\mathbf{Y}_v = -3$，其字长 $n = 4$，$k = 0$，基数为 2 的 SD 系统可以认为具有一个数字集合 $\{\bar{1}, 0, 1\}$．为了清楚起见，负的数字

-1 用上面带一道横线表示，即 $\bar{1}$. 有五种合法的 **SD** 表示法可产生数值 -3.

$$
\begin{aligned}
\mathbf{Y} &= (00\bar{1}\bar{1})_2 = -2 - 1 \\
&= (0\bar{1}01)_2 = -4 + 1 \\
&= (\bar{1}101)_2 = -8 + 4 + 1 \\
&= (0\bar{1}1\bar{1})_2 = -4 + 2 - 1 \\
&= (\bar{1}11\bar{1})_2 = -8 + 4 + 2 - 1
\end{aligned}
$$

这些表示方法具有不同的零和非零数字的分布. 在一个具有 n 个数字其值为 $\mathbf{Y}_v$ 的 **SD** 向量 **Y** 中，非零数字的数目叫做**权** $\omega(n, \mathbf{Y}_v)$. 上面的数所具有的权是从 2 到 4. 一般，一个二进制的 n 位数字的 **SD** 向量，它的权定义如下：

$$\omega(n, \mathbf{Y}_v) = \sum_{i=0}^{n-1} |y_i| \tag{1.25}$$

其中当 $y_i \neq 0$ 时，$|y_i| = 1$.

对应于给定的 n 值和 $\mathbf{Y}_v$ 值，具有最小权的 **SD** 向量叫做最小 **SD** 表示法. 在以上例子中，有两个 **SD** 向量最小，即为 $(00\bar{1}\bar{1})_2$ 和 $(0\bar{1}01)_2$. 最小 **SD** 表示法对于第五章中所讨论的乘数再编码是特别有意义的.

普通的基数为 r 的数可以按照下面的方法很容易转换成一个等价的 **SD** 形式：

令 $\mathbf{X} = (x_{n-1}, \cdots, x_1, x_0)_r$ 是一个普通的基数为 r 的数，而 $\mathbf{Y} = (y_{n-1}, \cdots, y_1, y_0)_r$ 是等价的 **SD** 数. 等价的意思是它们代表相同的代数值. 对于每一个普通数字 x_i，我们形成一个中间的差数字 d_i 为

$$d_i = x_i - r \cdot b_{i+1} \tag{1.26}$$

其中借位数字，

$$b_{i+1} = \begin{cases} 0, & \text{如 } x_i < \alpha \\ 1, & \text{如 } x_i \geq \alpha \end{cases} \tag{1.27}$$

于是第 i 位 **SD** 数字 y_i 可以用 d_i 和 b_i 相加来获得

$$y_i = d_i + b_i \tag{1.28}$$

举例说，给定 $\mathbf{X} = (0648)_{10}$，其 $r = 10$, $n = 4$，对给定的 **SD** 集合 $\{\bar{6}, \cdots, \bar{1}, 0, 1, \cdots, 6\}$，$\alpha = 6$. 我们按照表 1.3 所经过的转换顺序，可得到 $\mathbf{Y} = (1\bar{4}5\bar{2})_{10}$. 值得注意的是，还可以从另一端开始形成 **SD**；实际上这里没有借位的传播. 正如上例可看到的，所有带符号的数字是可以独立生成的.

表 1.3 一个普通十进制数转换成一个等价的 SD 形式

数字位置 i	4	3	2	1	0	注　释
x_i		0	6	4	8	普通十进制数
d_i		0	$\bar{4}$	4	$\bar{2}$	中间的差
b_i	0	1	0	1		借位数字
y_i		1	$\bar{4}$	5	$\bar{2}$	SD 形式

从一个 **SD** 数 **Y** 转换到普通基数形式 **X** 是靠两个数 $\mathbf{Y}^+$ 和 $\mathbf{Y}^-$ 的相加，这两个数分别由 **Y** 中正的数字和负的数字来形成. 上面的例子表示

$$\mathbf{Y} = \mathbf{Y}^+ + \mathbf{Y}^- = (1\bar{4}5\bar{2})_{10}$$

其中

$$
\begin{array}{rl}
Y^{+} = & +(1\ 0\ 5\ 0)_{10} \\
+)\ Y^{-} = & -(0\ 4\ 0\ 2)_{10} \\
\hline
X = & (0\ 6\ 4\ 8)_{10}
\end{array}
$$

注意,因为要执行普通的减法,所以在上述反转换的过程中存在着进位传送.

1.6 算术运算的算法及其实施

一个算术运算的算法是一组以一定形式顺序排列或者循环进行的过程，它去处理合法的输入数据并且按预先规定的数据格式产生有意义的结果. 这些过程在离散的时间域内必须是机器可以执行的.无限循环是不允许的,因为要求有计算结束时自己停止的算法.

任何算术运算设计任务可以分成两个阶段. 第一是研制有效的算法. 第二是研制其逻辑上的实施. 两个阶段都需要很大的设计上的努力. 算法的研制是更为复杂和有开创性的工作，而其实施则要求更多的工程上的背景. 然而，应该说这两个阶段是互相支持的. 没有一方,另一方也不会产生有效的设计. 按照其数据规模,算术运算的算法可以分成两大类:

(1) **数字算法**分别处理操作数的个别数字，并形成局部的结果. 决定一位全加器操作的运算法则可作为数字算法的一个典型例子. 一般，数字算法可用小型或中型规模的组合或时序逻辑电路来实现，这些电路具有有限的输入输出. 在非二进制系统的情况下，每个数字可以在内部用二进制编码表示. 数字算法的复杂性并不一定随着基数的数

表 1.4 典型的算术运算的数字算法 (Avizienis[13])

数的系统[1)	数字算法名称	算术运算或逻辑规定[2)
CR, SD	数字传送	$D_i \leftarrow S_i$
CR, SD	右移 k 个数字位置	$D_{i-k} \leftarrow S_i$
CR, SD	左移 k 个数字位置	$D_{i+k} \leftarrow S_i$
CR	$(r-1)$ 的补数	$D_i \leftarrow (r-1) - S_i$
SD	加法逆元素	$D_i \leftarrow -S_i$
CR	进位	$C_{i+1} = \lfloor (X_i + Y_i + C_i)/r \rfloor$ 或 $C_{i+1} = X_iY_i \vee Y_iC_i \vee X_iC_i$ (二进制)
CR	和	$S_i \leftarrow (X_i + Y_i + C_i)$ 模 r 或 $S_i = X_i \oplus Y_i \oplus C_i$ (二进制)
SD	进位(传送)	$t_{i+1} = \mathrm{Sign}((X_i + Y_i) \times \lfloor (X_i + Y_i)/\alpha \rfloor)$
SD	和	$S_i \leftarrow X_i + Y_i + t_i - t_{i+1} \times r$
CR	数字积残数	$P_{ij} = (X_i \times Y_j)$ 模 r
CR	数字积进位	$C_{ij} = \lfloor (X_i \times Y_j)/r \rfloor$
CR	积与和	$Z_{ij} = ((X_i \times Y_j) + U_{ij} + V_{ij})$ 模 r

1) CR 表示普通基数的数的系统, SD 表示带符号数字的数的系统.

2) $\lfloor x \rfloor$ = 不超过 x 的最大整数:

$$
\mathrm{Sign}(x) = \begin{cases} 1 & \text{如 } x > 0 \\ 0 & \text{如 } x = 0 \\ -1 & \text{如 } x < 0 \end{cases}
$$

α 是一个 SD 数的系统中允许的最大的数字数值.

值增加而增大. 然而,在较高基数值的系统中,线路复杂性却急剧增加. 表 1.4 列出某些普通的数字算法.

(2) **向量算法**把整个操作数作为向量来处理. 在某种意义上, 可以组织局部数字算法以形成向量算法. 可以把数字算法看成是在微观水平上的处理, 这有点象用布尔代数方程或者查表水平上的逻辑设计. 向量算法是用向量值运算方程, 在宏观水平上来处理数值数据. 在正常执行标准或基本的运算操作以后,其结果可以**数字向量**的形式出现;也可以**条件向量**的形式出现. 举例说,一位的条件向量可以是**奇异点**检测(如溢出和零的检测)的算法的结果,而两位的条件向量可以是数值大小比较和范围估计的算法的输出. 表 1.5 列出某些单操作数和双操作数的典型向量算法.

算术运算的算法在现代数字计算机中可以用三种方法实现: 软件、硬件和固件,或者是这些方法的组合. 早期计算机只有一些基本的硬件功能,如加法/减法,求补和移位. 另外一些标准操作,如乘法、除法,以及基本函数,如对数, 求平方根等. 它们都是用软件的例行子程序来实现的. 以后,固件的出现,用存储在只读存储器中的微程序来代替存储的

表 1.5 典型基数的运算向量算法 (Avizienis[1])

矢量算法的名称和规定[1]	所用的定点记号[2]	在特定位置上细分的数字算法	注 释
直接传送 $D \leftarrow S$	RC, DC	$D_i \leftarrow S_i$, 对 $n-1 \geqslant 1 \geqslant 0$	$\lvert D\rvert = \lvert S\rvert = n$
符号扩展 m 位 $D \leftarrow \mathrm{Ext}(S)$	RC, DC	$D_i \leftarrow S_{n-1} = \mathrm{Sign}(S)$, 对 $n \geqslant 1 \geqslant n+m-1$ $D_i \leftarrow S_i$, 对 $n-1 \geqslant 1 \geqslant 0$	$\lvert D\rvert = n+m$ $\lvert S\rvert = n$
运算左移 m 位 $D \leftarrow S \times r^m$	RC, DC	$D_{i+k} \leftarrow S_i$ 对 $n-1 \geqslant 1 \geqslant m$	当有效数字丢失时, 在左端会产生溢出
	RC	$D_i \leftarrow 0$, 对 $m-1 \geqslant i \geqslant 0$	
	DC	$D_i \leftarrow \begin{cases} 0, & 如\ S > 0 \\ r-1, & 如\ S < 0 \end{cases}$ 对 $m-1 \geqslant i \geqslant 0$	
和 $D \leftarrow S+T$ 或者 差 $D \leftarrow S-T$	RC, DC	$D_i \leftarrow (S_i \pm T_i \pm C_i)$ 模 r, 对 $n-1 \geqslant i \geqslant 0$	会产生相加上溢或相减下溢
	RC	$C_0 \leftarrow 0$ (初始进位或借位)	
	DC	$C_0 \leftarrow C_n$ (循环进位或借位)	
舍入 m 位 (1) 截断	RC, DC	$D_i \leftarrow S_i$, 对 $n-1 \geqslant i \geqslant m$ $D_i \leftarrow 0$, 对 $m-1 \geqslant i \geqslant 0$	m 位最低有效位被截尾
(2) 局部调整舍入	RC, DC	$D_i \leftarrow S_i$, 对 $n-1 \geqslant i \geqslant m+1$, $D_m \leftarrow S_m + \left\lfloor \frac{S_m}{r/2} \right\rfloor$, $D_i \leftarrow 0$, 对 $m-1 \geqslant i \geqslant 0$	
(3) 和调整舍入	RC, DC	$D_i \leftarrow (S_i + C_i)$ 模 r, 对 $n-1 \geqslant i \geqslant m+1$, 其中 C_i 是进位输入, 由于 $D_{m+1} \leftarrow S_m + 1$, 如 $S_{m-1} = 1$, $D_i \leftarrow 0$ 对 $m-1 \geqslant i \geqslant 0$	会发生上溢
零检测 $F_z \leftarrow S = 0$	RC	$F_z \leftarrow 1$, 如 $S_i = 0 \vee 0 \leqslant i \leqslant n-1$	清零
	DC	$F_z = 1$, 如或者 $S_i = 0$, 对所有 $n-1 \geqslant i \geqslant 0$ 或者 $S_i = r-1$, 对所有 i	使用 +0 和 −0

1) $D = D_{n-1} \cdots D_1 D_0$ 和 $S = S_{n-1} \cdots S_1 S_0$.

2) RC 表示定点基数补码的记号, DC 表示定点基数反码的记号.

软件子程序．现在，几乎所有计算机都有一些硬件实现的标准算术运算功能和一些基本函数．

软件的方法很费时间，而且为了存放例行子程序的语句要占用很大的存储器空间．然而，在早期的计算机工业中，这是比较便宜的．随着硬件价格的迅速下降，只有少量计算机还用软件来计算常用的函数．硬件方法提供较快的执行速度并占用极少的存储器空间，少到只有用于指令本身的一个或两个字．固件提供一个给算术运算算法的微操作顺序的方法．控制存储器提供远比软件例行子程序要快得多的执行速度．使用长控制字的横向微程序设计可以发挥最大的硬件并行性，它能使速度更快．在近年内，由于微电子学成功的发展，使得可以用大规模的，细胞单元式的，叠接的或流水线逻辑阵列的方法来实现算术运算操作．这种极先进的硬件方法会提供极快的计算速度．

我们将把算法及其实现方法两者和各种不同运算操作联在一起讨论．向量算法将用流程图来说明，而数字算法(如果需要)将用布尔代数方程．我们将提出利用中规模/大规模集成电路片的硬件方法来实现算术运算．计算机本身将用功能性的方框图表示，而不是用很多细微的逻辑设计图表示．

1.7 运算处理器的规格

运算处理器是数字计算机中实际执行算术运算任务的部分．下面给出数字运算处理器的一般规格．公式是按照设计者的观点来描述的．设计者必须考虑指导算术运算的算法，系统结构以及处理器逻辑上的实施．设计者必须把算术运算设计和逻辑设计分成两个相接连的阶段．这两阶段的办法使算法的描述可以分开，并与迅速变化着的器件工艺无关．

图 1.2 画出一个可以用 13 个分量描述的数字算术运算处理器

$$\mathbf{A} = \langle \mathbf{I}, \mathbf{R}, \mathbf{O}, \mathbf{F}, \mathbf{C}, \mathbf{S}, \mathbf{T}, \mathbf{\Delta}, \mathbf{f}, \mathbf{g}, \mathbf{h}, \mathbf{p}, \mathbf{q} \rangle \tag{1.29}$$

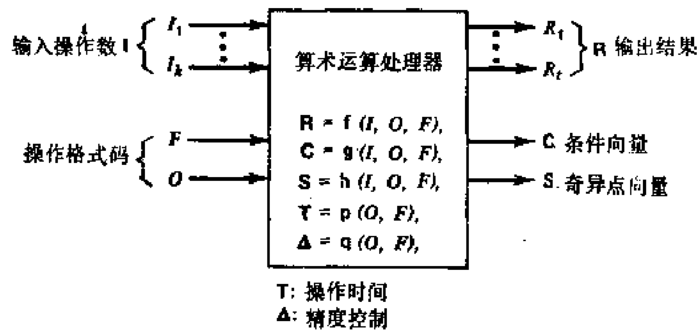


图 1.2 一个数字算术运算处理器的一般规格

其中

1. $\mathbf{I} = \{I_1, I_2, \cdots, I_k\}$ 是输入操作数的集合，若 $k = 1$，即为单操作数操作，而若 $k = 2$，则为双操作数操作，以此类推．每个操作数 I_i 是一个满足下述机器特性的，预先规定好的数表示法的数字向量：

1. a 每个 $I_i \in \mathbf{M}$，其中

$$\mathbf{M} = \{m \mid \mathbf{m}_{\min} \leqslant m \leqslant \mathbf{m}_{\max}\} \tag{1.30}$$

是一个有限精度机器可表示的数的有限集合。

1. b 每个 I_i 在下列已知精度之内

$$I_i - \Delta_L \leqslant I_i \leqslant I_i + \Delta_H \tag{1.31}$$

其中 $\Delta_L, \Delta_H \in \mathbf{\Delta}$ 将在下面第 7 点中说明。

2. $\mathbf{R} = \{R_1, R_2, \cdots, R_l\}$ 是一个满足 (1.a) 和 (1.b)的结果输出数字向量的集合，但允许有不同的预先规定的表示法。

3. 操作格式码 (OFC) 是一个包含有很多操作格式对 ($\mathbf{O}$, $\mathbf{F}$) 的向量

$$\mathbf{OFC} = (O_1, F_1; O_2, F_2; \cdots; O_r, F_r) \tag{1.32}$$

其中，$O_i \in \mathbf{O}$ 是可容许算子的集合，而 $F_i \in \mathbf{F}$，是可容许的数据格式的集合。

4. $\mathbf{C}$ 是条件向量和 $\mathbf{S}$ 是奇异点向量，用以确定与结果有关的特殊条件(如结果的符号及其它)或者指明在没有形成合法结果时的奇异点条件。奇异点条件包括上溢，下溢等等。

5. $\mathbf{f}$ 是与输入输出有关的向量值映象

$$\mathbf{f}: \mathbf{I} \times \mathbf{O} \times \mathbf{F} \rightarrow \mathbf{R}; \tag{1.33}$$

$\mathbf{g}$ 和 $\mathbf{h}$ 是与输入有关的向量值映象，而条件和奇异点向量分别为

$$\begin{aligned} \mathbf{g}: \mathbf{I} \times \mathbf{O} \times \mathbf{F} \rightarrow \mathbf{C}; \\ \mathbf{h}: \mathbf{I} \times \mathbf{O} \times \mathbf{F} \rightarrow \mathbf{S} \end{aligned} \tag{1.34}$$

6. $\mathbf{P}$ 是一个时间函数，它决定 OFC 所要求的执行时间 (T 是时间域)。

$$\mathbf{P}: \mathbf{O} \times \mathbf{F} \rightarrow \mathbf{T} \tag{1.35}$$

7. Δ 是精度控制参数的集合，而 $\mathbf{q}$ 规定精度控制机构

$$\mathbf{q}: \mathbf{O} \times \mathbf{F} \rightarrow \mathbf{\Delta} \tag{1.36}$$

上面列出的一个数字运算处理器的数学公式，用前面描述的两阶段算术运算设计过程是可以实现的。其细节应包括研制有效的向量算法，主机的系统结构，用以上映象表示的运算处理器的规格，以及数字电路构成算法的逻辑设计。在选定一个有效算法以后，一个算术运算设计者主要承担的工作就是去制定这些映象，使其形式能够很容易地翻译成逻辑设计任务。

1.8 定点数的表示方法

一个带符号的数必须是可正可负的，但不能又是正又是负。通常，具有 n 个数字位置的数，最左边的数字保留作为符号指示位。考虑下面基数为 r 的数

$$\mathscr{A} = (a_{n-1}a_{n-2}\cdots a_1a_0)_r \tag{1.37}$$

其中符号数字 a_{n-1} 采取的值为

$$a_{n-1} = \begin{cases} 0, & \text{如 } \mathscr{A} \geqslant 0 \\ r-1, & \text{如 } \mathscr{A} < 0 \end{cases} \tag{1.38}$$

$\mathscr{A}$ 中其余的数字表示 $\mathscr{A}$ 的真正数值或者是 $\mathscr{A}$ 的两种补值中的一种。由于 $\mathscr{A}$ 是一个位置表示的数，我们应该在其中置放一个基数点，把整数部分和分数部分分开。如果这个基数点是选择在固定位置上，则我们称它为定点 (FXP) 数。理论上说，在给定的表示方法中，这个基数点可以位于任何两个相邻的数字之间。

定点数表示法通常采用两种方式。这两种方式是可以互换的。每一个 n 数位的整数

可以看成是一个分数乘上一个常数因子 r^n，而任何一个 k 数位的分数可以看成是一个整数乘上一个常数因子 r^{-k}，其中 r 是固定的基数． 这样很容易在整数和分数之间转换．

因此，基数点位置的两种最常用的选择是：或者在这个数的数值部分的最左端，或者在最右端．前者的情况，基数点位于符号数位 a_{n-1} 和具有最高有效数值的数位 a_{n-2} 之间．这就使所有定点分数严格地小于 1． 后者的情况，基数点位于最低有效数位 a_0 右面，它使所有定点数都是整数．这两种方法本质上是等效的．我们可以很方便地乘以或除以一个常数因子 r^{n-1}，即能把一种方式转换成另一种． 在本书讨论中的大部分情况，我们都采取整数的形式，除非有另外的注明．固定基数点是不言而喻的，没有必要去明显标明．

对正的定点数，符号位 $a_{n-1}=0$，而其余的数字 $a_{n-2}, \cdots, a_1, a_0$ 永远表示其真正数值．我们把一个数 $\mathscr{A}$ 的真正或绝对数值表示为

$$|\mathscr{A}| = (m_{n-2}m_{n-3}\cdots m_1m_0)_r \tag{1.39}$$

其中当 $a_{n-1}=0$ 或者 $\mathscr{A} \geqslant 0$ 时；对 $i=n-2, \cdots, 1, 0$，有 $m_i = a_i$． 总之，我们把一个正的定点数表示为

$$\mathscr{A} = (0a_{n-2}\cdots a_1a_0)_r = (0m_{n-2}\cdots m_1m_0)_r \tag{1.40}$$

这种表示其数值等于

$$|\mathscr{A}| = \sum_{i=0}^{n-1} a_i \cdot r^i = \sum_{i=0}^{n-2} m_i \cdot r^i \tag{1.41}$$

为了表示一个负的定点数，有三种不同的表示方法． 如令 $\bar{\mathscr{A}}$ 是 (1.40) 式所定义的正数 $\mathscr{A}$ 的负数，则 $\bar{\mathscr{A}}$ 具有一个数值为 $r-1$ 的符号数位．下面给出负数的三种不同定点表示法：

带符号数值表示法

$$\bar{\mathscr{A}} = ((r-1)m_{n-2}\cdots m_1m_0)_r \tag{1.42}$$

其中对于 $(n-2) \geqslant i \geqslant 0$ 的 m_i 是真正的数值位．在这种表示方法中，正数形式 $\mathscr{A}$ 与负数形式 $\bar{\mathscr{A}}$ 不同之处只在于其符号位．

基数反码表示法

$$\bar{\mathscr{A}} = ((r-1)\bar{m}_{n-2}\cdots \bar{m}_1\bar{m}_0)_r \tag{1.43}$$

其中对于 $n-2 \geqslant i \geqslant 0$ 的 $\bar{m}_i = (r-1) - m_i$． 在这种表示方法中，$\bar{\mathscr{A}} = r^n - 1 - \mathscr{A}$． 它也叫做 $(r-1)$ 的补码表示法．

基数补码表示法

$$\bar{\mathscr{A}} = (((r-1)\bar{m}_{n-2}\cdots \bar{m}_1\bar{m}_0) + 1)_r \tag{1.44}$$

在这种表示方法中，$\bar{\mathscr{A}} = r^n - \mathscr{A}$． 它也叫做 r 的补码表示法．

对于两种特殊的基数系统，即二进制 $(r=2)$ 和十进制 $(r=10)$ 的数，以上的表示方法已归纳在表 1.6 中．在表 1.7 中给出一个用来分清不同的二进制表示法的具体例子．假设我们把数值为 $A=(547)_{10}=(1000100011)_2$ 的带符号定点数以正数和负数形式存入 16 位长的小型计算机中．可以注意到，为了填满整个 16 位的字，对于所有表示方法的正数和带符号数值表示方法的负数来说，要在前面填满"零"． 另一方面，对于"1"的补码或"2"的补码表示的负数*，要在前面填满"1"． 我们称这是符号补数系统的**符号延伸**．

---

* "1"的补码即通常所说的反码，"2"的补码简称补码．——译者注

表 1.6 负值的二进制或十进制数的定点表示法

表示方法	二进制($r=2$)	十进制($r=10$)
带符号数值	$(1m_{n-2}\cdots m_1m_0)_2$	$(9m_{n-2}\cdots m_1m_0)_{10}$
基数反码	1 的补码 $(1\bar{m}_{n-2}\cdots\bar{m}_1\bar{m}_0)_2$	9 的补码 $(9\bar{m}_{n-2}\cdots\bar{m}_1\bar{m}_0)_{10}$
基数补码	2 的补码 $(1\bar{m}_{n-2}\cdots\bar{m}_1\bar{m}_0+1)_2$	10的补码 $(9\bar{m}_{n-2}\cdots\bar{m}_1\bar{m}_0+1)_{10}$

表 1.7 数 ±547 的二进制定点表示法

定点表示法	二进制数	
	+547	−547
带符号数值	0000001000100011	1000001000100011
"1"的补码	0000001000100011	1111110111011100
"2"的补码	0000001000100011	1111110111011101

以上三种定点表示系统的任何一种所涉及的整数范围取决于字长 n. 在所有三种表示法中，范围的上界值决定于包括符号位在内的 n 位的字所能表示的最大正整数.这种上界的二进制形式为 $(011\cdots1)_2$,它等于十进制数 $2^{n-1}-1$. 对于带符号数值和反码的数来说，下界分别是 $(111\cdots1)_2=-(2^{n-1}-1)_{10}$ 和 $(100\cdots0)_2=-(2^{n-1}-1)_{10}$. 在"2"的补码运算中，最负的数是 $-2^{n-1}=(100\cdots0)_2$. 当一个正数超出上界时，叫做上溢. 同样，当一个负数超出下界时，叫做下溢. n 位计算机的二进制整数的范围归纳在表 1.8 中.

表 1.8 字长为 n 位的定点二进制算术运算系统中整数范围和零的表示方法

表示方法的系统	整数范围	零的表示方法	
		正	负
带符号数值	$-(2^{n-1}-1)\leqslant A\leqslant 2^{n-1}-1$ $111\cdots1\leqslant A\leqslant 011\cdots1$	000…0	100…0
反码	$-(2^{n-1}-1)\leqslant A\leqslant 2^{n-1}-1$ $100\cdots0\leqslant A\leqslant 011\cdots1$	000…0	111…1
补码	$-2^{n-1}\leqslant A\leqslant 2^{n-1}-1$ $100\cdots0\leqslant A\leqslant 011\cdots1$	000…0	000…0

所有的定点数都只有单一的表示法，但是，在带符号数值和反码表示系统中的"零"例外. 在这两种数的表示系统中，正零和负零的表示方法不同. 而对于补码表示系统，零的表示方式是唯一的，即为 000…0. 我们把这个唯一的零向量叫做"干净的零"，以区别于另外两种定点数表示方法中的"肮脏的零".

1.9 浮点数的表示方法

定点表示法对于表示有限幂值的小基数的数是很方便的.以字长为 32 位的二进制计算机而言，这个计算机能处理的定点整数的范围限制在 $\pm(2^{31}-1)$，它大约等于 $\pm10^{11}$. 这个范围对于工程和科学应用是不适应的. 为了表示范围更宽的数，我们采用分成两个部分的表示法，用

$$\mathrm{f}=(m,e) \tag{1.45}$$

来表示一个实数

$$\mathrm{f}=m\times r^{e} \tag{1.46}$$

其中 m 和 e 各是一个带符号的定点数，而 r 是给定的基数. 32 位操作数的一种可能表示

格式是为 m 域保留 22 个最左位，而其余的 10 位留给 e 域，所蕴含的基数 $r = 2$.

这两个分量 m 和 e 分别称为数 $\mathbf{f}$ 的尾数和阶. 一般尾数 m 可以采取上节中描述的三种定点表示法中的任一种. 举例说，可以考虑如下方式： m 是采用带符号表示法的一个分数，而阶 e 是具有偏置值或没有偏置值的以补码表示的整数（下面将简要说明）. 数 $\mathbf{f}$ 的基数 r 并不出现在表示式 (m, e) 中，因为它是不言自明的.

有意思的是可以观察到，尾数 m 中的基数点可以随着阶 e 数值的调整而浮动. 为此原因，我们称这种表示法 (m, e) 为数 $\mathbf{f}$ 的浮点表示法，而 $\mathbf{f}$ 这个数本身叫做浮点 (FLP) 数.

考虑十进制浮点系统中的一个数值例子

$$\mathbf{f} = -0.000005078125 \times 10^{+3} \tag{1.47}$$

它可以用以下两种浮点码表示

$$\begin{aligned} \mathbf{f}_1 &= (-0.000005078125, +3) \\ \mathbf{f}_2 &= (-0.507812500000, -2) \end{aligned} \tag{1.48}$$

我们可以把尾数向左(右)移动 k 个位置，而与此同时对阶的数值减去(增加) k，这样所表示的数的真正数值维持不变. 如在上例中，我们把尾数左移 5 个数位，同时相应地把阶减去 5. 在规格化浮点运算的操作中经常要执行尾数的移位和阶的增减.

所谓一个浮点数是规格化的，就是要求尾数的最高位具有一个非零的数字，如 (1.48) 式所示. 规格化一个浮点数的过程，需要把尾数向左移动，把前面多余的"0"推到更高的数位中去，同时相应地减小阶，一直到最高位出现一个非零的数字为止. 在规格化运算中，所有浮点数在它们被处理以前都必须预先规格化. 因此，在每一步中间计算以后，也必须进行算后规格化的过程，以确保规格化形式的完整性.

规格化后的尾数 m 所具有的绝对值在下列范围内

$$\frac{1}{r} \leqslant |m| < 1 \tag{1.49}$$

其中 r 是所蕴含的基数. 对于二进制的数 $(r = 2)$，其表示范围是 $0.5 \leqslant |m| < 1$. 换言之，所有规格化后的非零二进制数，其尾数不小于 1/2. 只有浮点零的表示是个例外.

阶 e 可以是一个正整数或负整数. 在对两个浮点数进行相加或相减以前，必须先对它们的阶进行比较并使之相等. 我们希望简化比较操作，使它不牵涉到阶的符号. 要绕过符号问题的一个办法是对每个阶都加上一个正的常数，使得所有的阶都转化为正整数，形成一个所谓偏置的阶. 这个偏置常数通常选择其值等于最负阶绝对值. 考虑一个具有 q 位阶域的浮点数表示法. 以补码表示的未偏置的阶所具有的值应在下列范围内

$$-2^{q-1} \leqslant e_{未偏置} \leqslant 2^{q-1} - 1 \tag{1.50}$$

我们可以选择偏置常数 $\mathbf{b} = 2^{q-1}$，使所有阶转化为正整数，其范围为

$$0 \leqslant e_{偏置} \leqslant 2^{q} - 1 \tag{1.51}$$

必须注意，真正的阶(未偏置的阶)可以用下式再从偏置的阶中恢复过来

$$e_{未偏置} = e_{偏置} - 2^{q-1} \tag{1.52}$$

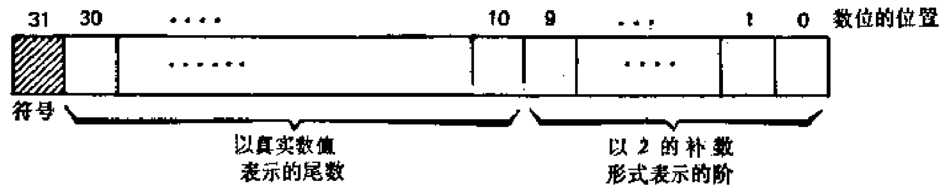
考虑把一个规格化的浮点数 $\mathbf{f} = -0.5078125 \times 2^{-2}$ 存储在一个字长为 32 位的数字计算机中，假定具有以下的数据格式：

第 31 位留作符号指示位，正数为"0"，负数为"1". 二进制的小数点位于第 31 位与第

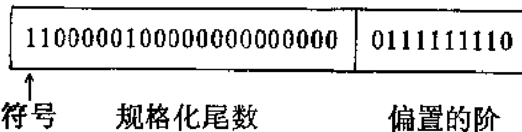
30位之间．因为，在带符号数值形式中的

$$m=(-0.5078125)_{10}=(110000001000000000000000)_2,$$

而 $e_{偏置}=(-2+512)_{10}=(510)_{10}=(0111111110)_2$，所以这个数的完整的浮点表示式



为



所用的偏置常数是 $2^{q-1}=2^9=(512)_{10}=(1000000000)_2$．为了比较起见，将三个特定的阶值列表如下：

$e_{未偏置}$	$(-512)_{10}=(1000000000)_2$	$(0)_{10}=(0000000000)_2$	$(+512)_{10}=(0111111111)_2$
$e_{偏置}$	$(0000000000)_2$	$(1000000000)_2$	$(1111111111)_2$

理论上，在浮点表示法中可以存在无穷多的零的表示式，即尾数等于零而不管阶的数值如何．在计算机设计中，我们希望对于定点和浮点数都只有单一的零的表示法．一个"干净"的定点零是由一串 0 组成的．我们也可以用带偏置的阶来为浮点运算定义这样一个"干净的零"．我们所必须做的是，指定用一个零尾数以及偏置形式的最负的阶来表示一个浮点零．"肮脏的零"包含有非零的阶，不作为是合法的表示法．采用了偏置的阶，我们就可以避免这种"肮脏零"所引起的混乱．

决定一个浮点数的精度是尾数长度．提高精度的方法在后面将要讨论．这个阶 e 显示出浮点数的数量级．在上例中，阶在 -512 和 $+511$ 之间．因此，我们可以表示正的规格化浮点数 F^+ 在以下范围内

$$0.5\times 2^{-512}\leqslant \mathbf{F}^+\leqslant(1-2^{-21})\times 2^{+511} \tag{1.53}$$

负的规格化浮点数 $\mathbf{F}^-$ 在以下范围内

$$-(1-2^{-21})\times 2^{+511}\leqslant \mathbf{F}^-\leqslant -0.5\times 2^{-512} \tag{1.54}$$

以上不等式中指出的浮点数的范围可用图 1.3 表示．由于尾数的规格化，在零（原

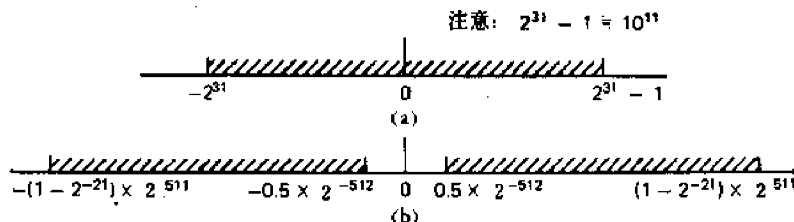


图 1.3 定点和浮点表示法中 32 位数的范围．
(a) 32 位定点数（整数）在补码表示法中的允许范围，
(b) 32 位规格化的浮点数（具有 21 位尾数和 10 位基数为 2 的阶）的允许范围

点）的周围有一个间隙. 对于普遍基数 $r \geq 2$ 的规格化的浮点运算以及有关浮点运算设计的高级讨论题目将分别在第九和第十章中研究.

1.10 多倍精度的算术运算

前面几节中提到的数字数据均具有固定长度和单字长精度的格式，其中每一个数只占据一个存储器字. 在某些应用中，为了得到较好的精度或较广的操作范围，需要采用可变字长或多倍精度的格式来增加数中的有效数位的数目. 为了增加包含在尾数中的有效数位的数目，可以用两个或多个字来表示一个浮点数. 作为一个例子，对每一个浮点数可以用两个 32 位的存储字. 假如我们仍把第一个字中的 10 位留作阶，那么，尾数的精度可以增加为 $22 + 32 = 54$ 位. 另一种办法，我们可以用 48 位表示尾数和用 16 位表示阶. 在这两个域的长度之间存在着折衷方案.

若干种需要多倍精度数据格式的标准算术运算操作描述如下：

1. 在两个 n 位单精度的数相加或相减以后，预期有一个单精度的和以及 1 位进位. 为了以后的计算，这个进位输出需要用一个专门的触发器来保持.

2. 在两个 n 位二进制数相乘以后，预期有一个 $2n$ 位的积（或者有时是 $(2n-1)$ 位的积）. 假如只允许用一个单精度的数，我们就必须把乘积的下半部舍入掉，这将引起舍入误差. 为了保持整个精度，需要产生一个双倍字长的乘积.

3. 一个双倍字长（$2n$ 位）整数被一个单字长（n 位）的整数除了以后，预期有一个单字长的商和一个单字长的余数. 如果预料到会有这一对结果（商；余数），则为了以后的使用，需要有一个双倍精度的字来保持它.

当计算中预期出现多倍精度的数时，必须采取措施来指出给定的数是单字长，双字长或三字长等. 几种常用的方法描述如下：

1. 使用两个特殊符号作为结束标志，用它来确定一个数字数的开始和结束.

2. 附加一个单独的字作为可变字长的数的长度指示符. 一般，这个长度指示符表示多倍精度的数所需存储器字的数目.

3. 使用一个指数，把定点数转换成带有一个有效标志向量或者一个保护数字的浮点数.

对于浮点运算几乎经常需要做双倍精度的计算，除非是在统计工作中，那里定点双倍精度运算对于计算平方和以及交叉乘积来说更为有利. 定点多倍精度计算比浮点多倍精度计算要简单得多. 还有其它一些控制数字数据的精度的方法，诸如使用舍入方法或者使用保护寄存器，这些将在以后几章中讨论.

1.11 参考文献注释

第一章概括地论述了计算机的算术运算，包括理论，设计和应用上的问题. 对于数的系统和算术运算的设计方法，Garner[5,6] 和 Tung[18] 曾做过很出色的综述. Matula[11,13] 曾经写过基数算术运算综合性的数论处理方法以及它与计算机工程其它学科的相互关系. Knuth[8] 提出了机器算术运算（尤其是对于浮点计算）完整的历史性的说明. Kulisch[9] 写

过计算机算术运算数学基础的公理化的论述，它总结了计算机中的数的空间，舍入方法以及算术运算结构. 冗余的带符号数字的数系统最早是在Robertson[14]和Avizienis[1,2]的著作中提出来的. 在1.5节中提供的材料是文献[2]中定义的，带符号数字的数系统的一种松弛形式. 它也包括了基数为2的冗余数，而SRT除法(见第七章)的理论正是在这基础上建立的.

残数算术运算系统在Szabo和Tanaka[15]的文章中有详细的讨论. Matula[12], Hwang和Chang[7]的文章研究了有理数系统. 对数的数系统在Marasa的论文中有深入的论述. 算术运算操作的分类可以与现有的算术运算机器来作比较，诸如Buckholz[4]中的IBM7030 (Stretch计划)，文献[6]中的IBM360系统，以及Thornton[17]中的CDC6600. 算术运算算法的分类和处理器的规格在Avizienis的著作[3]中得到扩充. 若干个定点，浮点和多精度数据表示方法的实例研究可查阅文献[4,6,8,17].

IEEE计算机协会曾举办过四次国际性的三年一度的计算机算术运算讨论会，即1969，1972，1975和1978. 电气电子工程师协会的计算机汇刊曾经出版过有关计算机算术运算的三期专刊，包括头三次讨论会上提出的主要文章. 这三期专刊是在1970年8月，1973年6月，以及1977年7月发行的. 这些专刊中报告的一些最新成果已选用在本书中. 然而，如果读者要更详细了解的话，最好去阅读这些原文.

参 考 文 献

[1] Avizienis, A., "A Study of Redundant Number Representations for Parallel Digital Computers," *Ph. D. dissertation*, University of Illinois, Digital Computer Laboratory, May 1960.

[2] Avizienis, A., "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IRE Trans. on Elec. Computers*, Vol. EC-10, September 1961, pp. 389—400.

[3] Avizienis, A., "Digital Computer Arithmetic :A Unified Algorithmic Specification," *Proc. Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, 1971, pp. 504—525.

[4] Buckholz, W. (ed.), *Planning A Computer System: Project Stretch*, McGraw-Hill, New York, 1961.

[5] Garner, H. L., "Number Systems and Arithmetic," in *Advances in Computers*, Vol. 6, Academic Press, New York, 1965, pp. 131—194.

[6] Garner, H. L., "A Survey of Recent Contributions to Computer Arithmetic," *IEEE Trans. Comp.*, Vol. C-25, No. 12, December 1976, pp. 1277—1282.

[7] Hwang, K. and Chang, T. P., "A New Interleaved Rational/Radix Number System for High-Precision Arithmetic Computations," *Proc. of the Fourth Symposium on Computer Arithmetic*, October 1978.

[8] Knuth, D. E., *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, Addison-Wesley, Reading, Mass, 1969, Chap. 4.

[9] Kulisch, U., "Mathematical Foundations of Computer Arithmetic," *IEE5 Trans. Computers*, Vol. C-26, No. 7, July 1977, pp. 610—620.

[10] Marasa, J. D., "Accumulated Arithmetic Error in Floating-Point and Alternative Logarithmic Number Systems," *M. S. Thesis*, Seven Institute Technology Washington University. St. Louis, Mo., June 1970.

[11] Matula, D. W., "Number Theoretic Foundations for Finite-Precision Arithmetic," in *Applications of Numbers Theory to Numerical Analysis*, W. Zaremba (ed.), Academic Press, New York, 1972, pp. 479—489.

[12] Matula, D. W., "Fixed-Slash and Floating-Slash Rational Arithmetic," *Proc. of the Third Symposium on Computer Arithmetic*, 1979 Catalog No. 75 CH1017-3C, November 1975, pp. 90—90—91.

[13] Matula, D. W., "Radix Arithmetic: Digital Algorithms for Computer Architecture," in *Applied Computation Theory: Analysis, Design, Modeling*, R. T. Yeh (ed.), Prentice-Hall, Englewood Cliffs, N. J., 1976, Chap. 9.

[14] Robertson, J. E., "Redundant Number Systems for Digital Computer Arithmetic," *Notes for the Univ. of Michigan Engineering Summer Conference*, in "Topics in the Design of Digital Computing Machines," Ann Arbor, Mich., July 6—10, 1959.
[15] Szabo, N. S. and Tanaka, R. I., *Residue Arithmetic and Its Applications to Computer Technology*, McGraw-Hill, New York, 1967.
[16] Svoboda, A., *Digitale Informationswandler*, Braunschweig, Germany, Vieweg and Sohn, 1960.
[17] Thorton, J. E., *Design of A Computer The Control Data 6600*, Scott, Foresman, Glenview, Illinois., 1970. Chap. V.
[18] Tung, C., "Arithmetic," in *Computer Science* (A. F. Cardenas et al., ed.) Wiley-Interscience, 1972, Chap. 3.

习　题

题 1.1　区别以下与数字计算机算术运算系统有关的各对术语：

(a) 实数算术运算与机器算术运算.

(b) 数字算法与向量算法.

(c) 固定基数与混合基数的数的系统.

(d) 冗余与非冗余数表示法.

(e) 定点与浮点数表示法.

(f) 整数与分数定点算术运算.

(g) 单精度与双精度运算操作.

(h) 规格化与非规格化浮点操作.

(i) 基数反码与基数补码的数的系统.

(j) 浮点表示法中偏置与未偏置的阶值.

(k) 内部数表示法中的"肮脏零"与"干净零".

题 1.2　给定两个代数值 $(39.25)_{10}$ 和 $(-39.25)_{10}$. 找出以下基数为 r 的数的系统中各自的定点表示法. 每个数由 n 个整数数字和 k 个分数数字组成.

(a) 基数补码系统,其 $(r, n, k) = (8, 4, 2)$.

(b) 基数反码系统,其 $(r, n, k) = (2, 8, 4)$.

(c) 对于数字集合 $\{\bar{2}, \bar{1}, 0, 1, 2\}$, 写出一种可行的带符号数字表示法,其 $(r, n, k) = (4, 8, 8)$. 在该表示法中,至少应有一个负的数字.

题 1.3　给定一个代数值 $(-14)_{10}$, 字长 $n = 6$, 数字集合为 $\{\bar{1}, 0, 1\}$. 找出所有可能的数值为 -14 的 6 位带符号数字表示法. 指出在所有可能的 SD 表示法中最小的带符号数字表示法. 这种最小表示法是唯一的吗?

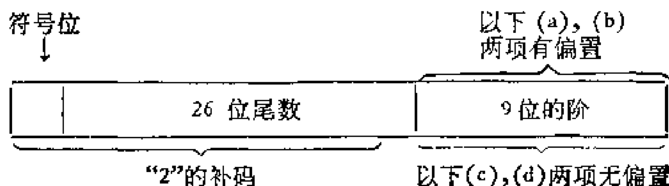
题 1.4　把普通的八进制数 $(376)_8$ 转变成一个 $r = 8$, $n = 6$, $\alpha = 5$ 的等价的带符号数字的数,数字集合为 $\{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$, 上面的 α 是最大的数字数值.

题 1.5　比较用硬件，固件和软件实现算术运算算法的优点和缺点. 评论其设计上需要花的功夫，计算的速度和对硬件的需求.

题 1.6　试找出以下浮点数的规格化的内部机器表示方法,这些浮点数是在一个 32 位的 FLP 的运算处理器中, FLP 数据的格式与 1.9 节中给定的相同.

(a) -1023.75×2^{-7}　(b) $+0.0025 \times 2^{+301}$　(c) $+932.875 \times 2^{+327}$　(d) $-0.007813 \times 2^{-490}$

题 1.7　某计算机具有下列36位规格化的浮点数据格式,其尾数和阶都是"2"的补码形式.



为以下极端的浮点数写出在36位计算机中的位组合格式．假定阶的基数($r=2$)

(a) 具有偏置的阶的最大正数．

(b) 具有偏置的阶的最负的数．

(c) 重复 (a)，不过阶域为未偏置的．

(d) 重复 (b)，不过阶域为未偏置的．

题 1.8 令 **A** 是一个基数为 r，n 位数字的定点整数．令 $\bar{\mathbf{A}}$ 是将 **A** 按位对值 $r-1$ 求补而获得的数．

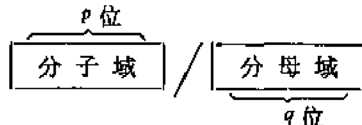
(a) 证明 $\mathbf{A}+\bar{\mathbf{A}}=r^n-1$，假定 **A** 是以基数反码的形式表示的．

(b) 证明 $\mathbf{A}=\bar{\mathbf{A}}+1=r^n$，假定 **A** 是以基数补码的形式表示的．

注意，在 (a) 中，如 1.43 式给定的那样 $\bar{\mathbf{A}}=\mathscr{A}$；在 (b) 中，如 1.44 式给定的那样 $\bar{\mathbf{A}}+1=\mathscr{A}$．

题 1.9 说明如何对一个浮点数进行规格化，假如它的尾数是以补码形式表示 (不象前面所述的以带符号数值形式表示)．又假如尾数是以反码形式表示，试重复回答这个问题．这两种规格化情况必须用尾数位和符号位推导出布尔方程式．

题 1.10 Matula 在文献[12]中建议用一种固定分切的有理数系统．这种系统的精度限制是由于分子和分母的大小分别都是有界限而引起的．分子和分母项可以按如下所示的标准基数格式存储在固定长度域中．为了简单起见，假定分子和分母两者都是不带符号的整数．



回答以下与固定分切有理数系统有关的问题：

(a) 这种有理数表示法系统所能提供的数的精度范围怎样?

(b) 假定 $p+q=n$ 是一个常数(固定大小)，而分切的位置可以改变．当这个分切号允许向右浮动 k 个位置，其中$1\leqslant k\leqslant q-1$，试评述精度范围的变化．

(c) 重复以上问题，但分切号向左浮动 k 个位置，其中 $1\leqslant k\leqslant p-1$．

在以上每个问题中，写出数的范围及在此范围内相邻有理数之间的最小间隙．

第二章　集成电路和数字器件

2.1　电子工艺和逻辑电路系列

数字计算机中使用的电子元件已经经历了四代的发展. 第一代数字计算机（在 1952 年以前）使用真空管作为基本电路元件.第二代的计算机电子学是以使用分立晶体管和二极管元件为标志（1958 年），这些元件比起以前的真空管来说,体积相对小 20 倍，速度快 100 倍，而且失效率低 100 倍. 第三代是以使用集成电路（IC）为特点的，其形式为小规模集成（SSI）和中规模集成（MSI）电路片. 每个 SSI/MSI 电路片能包含多至 100 个电路元件,其工作速度是以毫微秒来度量的. SSI/MSI 器件所需的功耗和封装价格大为减少，而可靠性大为提高. 自从 1969 年以来，大规模集成（LSI）电路器件变得愈来愈普遍，这表示电子计算机第四代的开始. 一个大规模集成电路的功能器件，可以在单片电路中包含 1000 个以上的电路元件,而且价格、功耗和失效率进一步降低,密度和速度进一步提高（比真空管快 2500 倍以上）.

在本书中，我们考虑中大规模集成（MSI/LSI）元件的数字设计. 本章中简要介绍常用的数字集成电路和存储器. 这些介绍不只限于标准的 MSI/LSI 电路，也介绍用户定做的标准电路功能的动向. 本章也描述主要类型的运算集成电路和积木式元件的线路功能方框图及其外特性. 由于制造工艺变化太快，这些描述不打算牵涉到制造工艺. 我们最感兴趣的是这些器件的功能和特性. 这些 MSI/LSI 器件的知识对于我们在以后章节中理解各种运算处理器硬件结构和设计时是很必要的. 很明显，一个简要的介绍不可能包括电子市场上供应的所有标准成品. 为了补充这一点,我们提供一些通用的文献,有助于有兴趣的读者去查找主要的集成电路手册、数据以及应用须知.

下面列出各种固体电子工艺及其相应的逻辑电路系列. 在现代计算机结构中常用的数字电子器件有两种类型: MOS（金属-氧化物-半导体）和双极型器件.

金属-氧化物-半导体（MOS）工艺

1. PMOS，它采用 P 沟道的 MOS 电路.

2. NMOS，它采用 n 沟道硅栅工艺.

3. CMOS，它把 P 沟和 n 沟晶体管结合在同一基片上,形成互补 MOS.

4. CCD，电荷耦合器件，它利用在硅表面上形成的势位井中存在或不存在少数载流子电荷的原理来表示“1”和“0”.

5. VMOS，V 形槽 MOS，它可以产生具有 V 形槽口沟道的 MOS 晶体管. 这种短沟道可以得到较高的击穿电压和较高的封装密度.

双极型工艺和逻辑电路系列

1. TTL 电路,它使用多发射极的晶体管-晶体管逻辑电路.

2. ECL 电路，它是由非饱和的发射极耦合逻辑电路组成的.

3. I^2L 电路，它使用双极型集成注入逻辑电路.

4. 肖特基 TTL 和肖特基 I^2L，它们是相应地用肖特基二极管附加在多发射极和多收集极上的 TTL 和 I^2L 电路.

逻辑门电路功能和存储器阵列都可用以上工艺制造出来. 未来工艺发展的动向是要达到更高的开关速度，更高的密度(存储容量)，较低的功耗，以及较低的价格. 为了比较各种运算部件设计的速度，我们定义Δ为相应于单级逻辑电路的单位门延迟. Δ值的一个很好的度量单位就是通过一个"与非"(NAND)门或者一个"或非"(NOR)门的时间延迟. 因为每一种开关功能都可以用 NAND 或者 NOR 门来实现，多级开关电路的时间延迟可以用 NAND 门的级数或者Δ的数目来度量. 用Δ表示的典型门功能的时间延迟列于表 2.1 中. 注意接线逻辑的 AOI 门具有延迟为 $\Delta + \Delta_{RC}$，其中 Δ_{RC} 取决于负载的 RC 时间常数. 对于很小的 Δ_{RC}，接线逻辑门功能的近似延迟只有 Δ.

表 2.1　典型门电路的逻辑符号和时间延迟

门的功能	逻辑符号（正逻辑）	以Δ的数目表示的时间延迟
NAND	A, B → $\overline{A \cdot B}$	Δ
NOR	A, B → $\overline{A+B}$	Δ
NOT	A → $\overline{A}$	Δ
AND	A, B → $A \cdot B$	2Δ
OR	A, B → $A+B$	2Δ
XOR	A, B → $A \oplus B$	3Δ
XNOR	A, B → $A \odot B$	3Δ
AOI	A, B, C, D → $\overline{AB+CD}$	$\Delta + \Delta_{RC}$

2.2　多路转换器和分路转换器

一个 2^n 输入端的多路转换器(MPX)具有 2^n 条输入线，$x_{2^n-1}, \cdots, x_1, x_0$，以及一条

输出线 f. 这种部件在 n 条选择线 S_{n-1}, $\cdots$, S_1, S_0 的控制下,把第 i 条输入线 x_i 的逻辑值传送到输出线. 即为

$$f = x_i \text{ 其中 } i = (S_{n-1}\cdots S_1S_0)_2 = \sum_{i=0}^{n-1} S_i \cdot 2^i \tag{2.1}$$

数字式多路转换器可以看成是一个与具有外部控制的与多路旋转开关相同的电子器件. 在某些应用中称它为**数据选择器**, 因为它从 2^n 个数据输入端中选出一个, 并把其信息传送到输出端. 具有两条控制线的 4 输入端多路转换器,从四个输入端中选出一个,并使其出现在输出端,这个线路见图 2.1. 逻辑线路图中指出了一个附加的输出端, 它具有以 $\bar{f}$ 表示的相补的输出. 此外,选通输入可以有一个或者多个,它使电路片具有"使能"的能力. 如用接线逻辑实现, 那么一个多路转换器的数据从输入到输出的总延迟时间只有 2Δ.

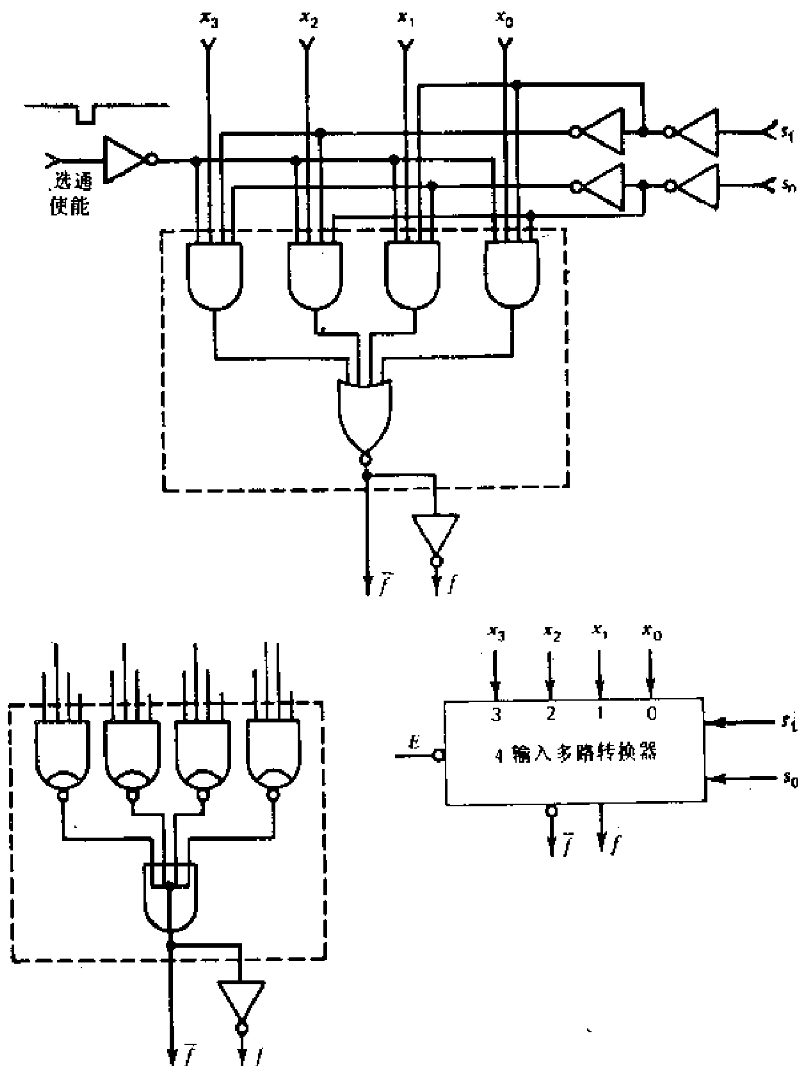


图 2.1 一个 4 输入端多路转换器的逻辑线路图, 以及它用与或非接线逻辑电路实现的方法 (在虚线以内)

由于受到目前封装工艺的限制, 数字式多路转换器以 14 到 24 条引线封装的单片集成电路有: 四个 2 输入端多路转换器,双 4 输入端多路转换器,双 8 输入端多路转换器以

及单个 16 输入端多路转换器．如要制造大型的多路转换器，可以把很多小型多路转换器按照多级树形结构连接起来．作为一个例子，可把四个双 4 输入端多路转换器的输出端接到一个 8 输入端多路转换器的输入端，构成一个 32 输入端的多路转换器，见图 2.2．在构造这种树形结构的多路转换器时，所必须遵循的一般规则是减少所用的集成电路封装片的总数，并使线路级数愈少愈好．一个 2^n 个输入端的多路转换器可以用来实现 $n+1$ 个变量的任何开关函数，其中有 n 个变量是在选择线上，而 2^n 条输入线中的每一条只是剩下的变量的一个函数．

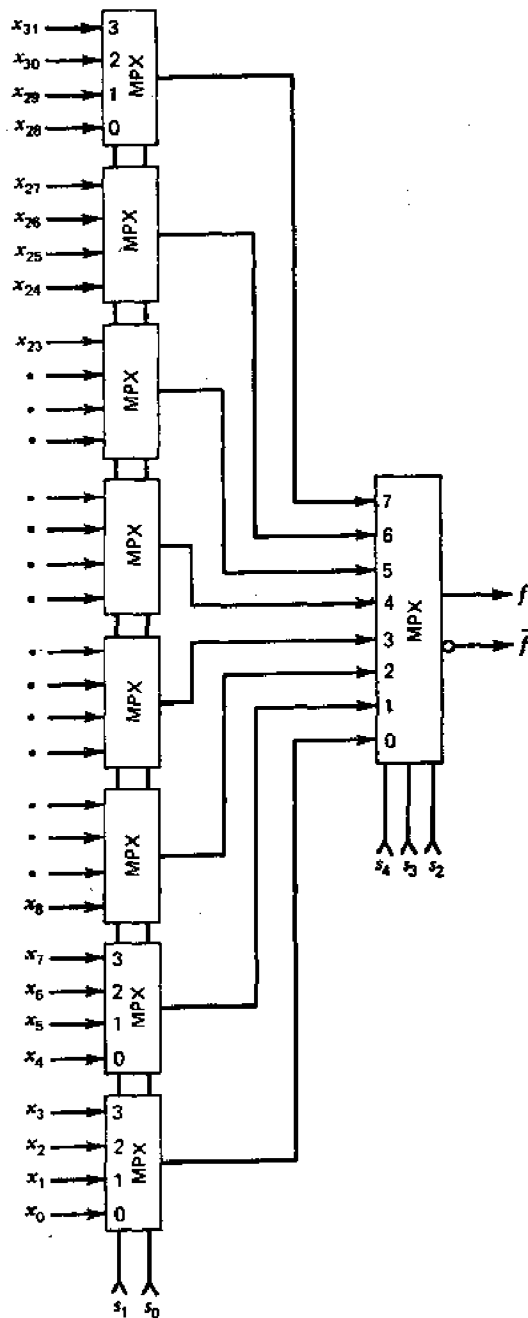


图 2.2　由 8 个 4 输入端多路转换器和一个 8 输入端多路转换器组成一个 32 输入端多路转换器

一个分路转换器和一个数字多路转换器刚好相反．这种电路在一条输入线上接受二进制数据，并把它传送到 2^n 条可能的输出线中的一条．所选择的输出线取决于 n 条选择线的二进位组合格式．一个 4 输出端的分路转换器的逻辑电路和逻辑符号见图 2.3．

行译码器是分路转换器的一种变化形式，其中伪数据输入线被去掉了．一个 3 到 8 的行译码器如图 2.4 所示，其中三条选择线 a_2，a_1，a_0 叫做地址线．这三条地址线各位数字的组合决定了八条输出线中哪一条应该变高电位．还有选通输入端可以用来扩展译码器．行译码器用于对存储器系统进行寻址，因而称为地址译码器，它也可以用于对向量输出模式进行译码．现在，单片封装的双 2 到 4 行译码器，3 到 8 行译码器以及 4 到 16 行译码器已可供使用，这些电路有的具有选通输入端，有的没有．同样，也可把很多具有选通输入端的小行译码器连接起来，组成一个大型行译码器．图 2.5 表示一个 6 到 24 行译码器，它是由四个各带两个选通输入端的 4 到 16 行译码器组成．地址寄存器中的高两位是用来区分四个行译码器的．

我们经常需要用组合逻辑模块来进行二进制数据到二-十进制（BCD）数据的转换，以及它的反转换．图 2.6 表示一个 6 位的二进制到 BCD 的转换器的功能框图和真值表．这个电路把 6 位二进制数转换成一个 7 位的包含两个十进制数字的 BCD 数．为了转换更长的二进制数，可以使用很多个这种模块．图 2.7 表示把三个 6 位的二进制到 BCD 的

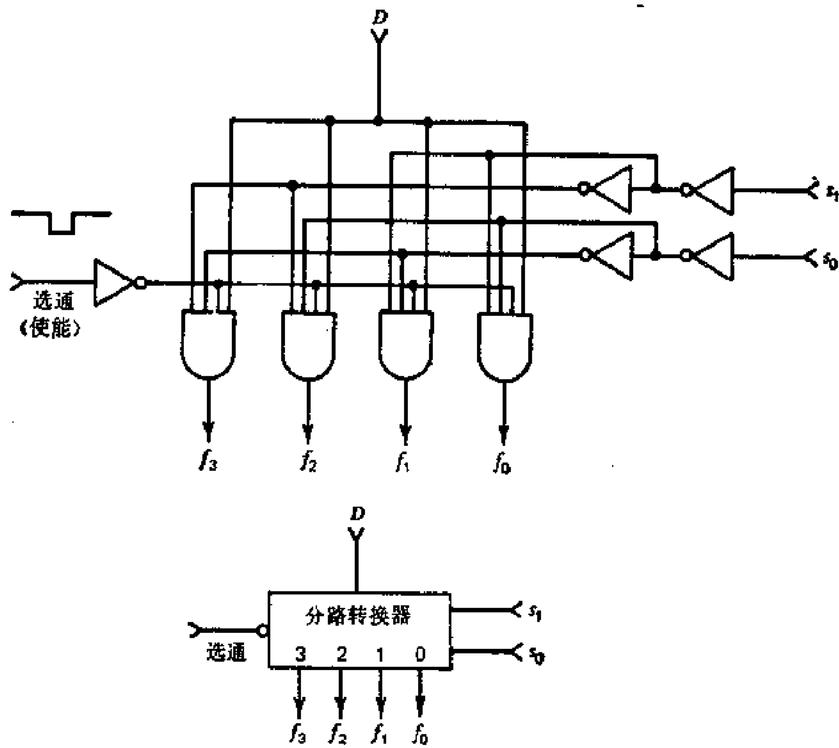


图 2.3 一个 4 输入端分路转换器的线路图和逻辑符号

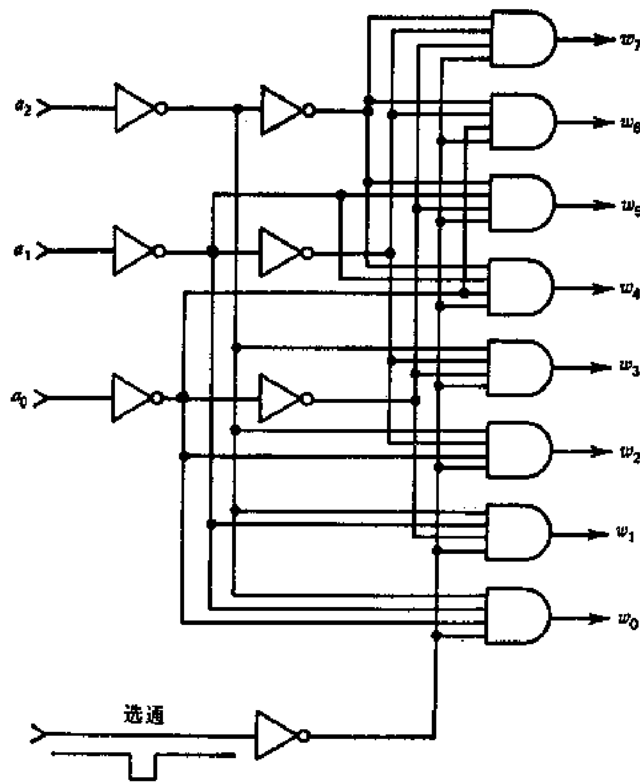


图 2.4 3 到 8 行译码器电路图

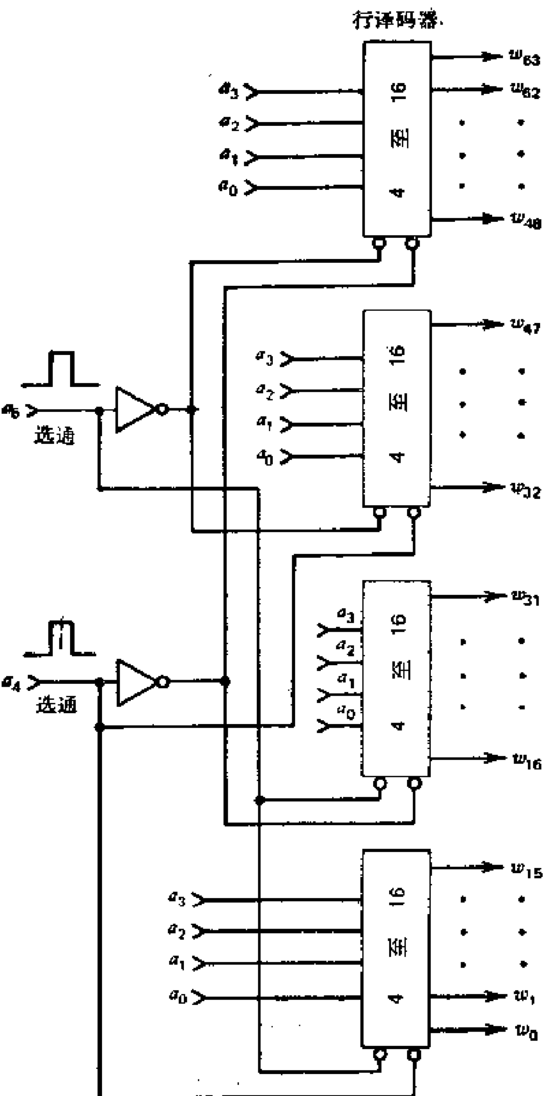
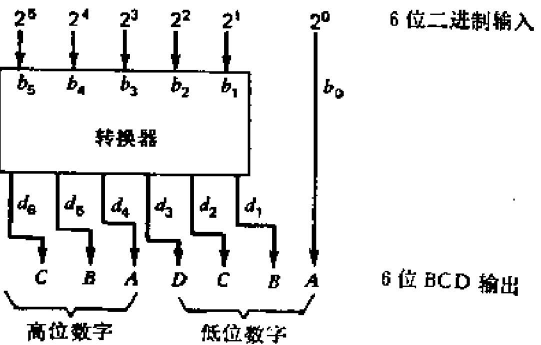


图 2.5　由四个各带两个选通输入端的 4 到 16 行译码器组成一个 6 到 64 行译码器



函数表

二进制字	输入 b_5 b_4 b_3 b_2 b_1	输出 d_6 d_5 d_4 d_3 d_2 d_1	二进制字	输入 b_5 b_4 b_3 b_2 b_1	输出 d_6 d_5 d_4 d_3 d_2 d_1
0,1	0 0 0 0 0	0 0 0 0 0 0	32,33	1 0 0 0 0	0 1 1 0 0 1
2,3	0 0 0 0 1	0 0 0 0 0 1	34,35	1 0 0 0 0	0 1 1 0 1 0
4,5	0 0 0 1 0	0 0 0 0 1 0	36,37	1 0 0 1 0	0 1 1 0 1 1
6,7	0 0 0 1 1	0 0 0 0 1 1	38,39	1 0 0 1 1	0 1 1 1 0 0
8,9	0 0 1 0 0	0 0 0 1 0 0	40,41	1 0 1 0 0	1 0 0 0 0 0
10,11	0 0 1 0 1	0 0 1 0 0 0	42,43	1 0 1 0 1	1 0 0 0 0 1
12,13	0 0 1 1 0	0 0 1 0 0 1	44,45	1 0 1 1 0	1 0 0 0 1 0
14,15	0 0 1 1 1	0 0 1 0 1 0	46,47	1 0 1 1 1	1 0 0 0 1 1
16,17	0 1 0 0 0	0 0 1 0 1 1	48,49	1 1 0 0 0	1 0 0 1 0 0
18,19	0 1 0 0 1	0 0 1 1 0 0	50,51	1 1 0 0 1	1 0 1 0 0 0
20,21	0 1 0 1 0	0 1 0 0 0 0	52,53	1 1 0 1 0	1 0 1 0 0 1
22,23	0 1 0 1 1	0 1 0 0 0 1	54,55	1 1 0 1 1	1 0 1 0 1 0
24,25	0 1 1 0 0	0 1 0 0 1 0	56,57	1 1 1 0 0	1 0 1 0 1 1
26,27	0 1 1 0 1	0 1 0 0 1 1	58,59	1 1 1 0 1	1 0 1 1 0 0
28,29	0 1 1 1 0	0 1 0 1 0 0	60,61	1 1 1 1 0	1 1 0 0 0 0
30,31	0 1 1 1 1	0 1 1 0 0 0	62,63	1 1 1 1 1	1 1 0 0 0 1

图 2.6　一个 6 位的二进制到 BCD 转换模块的方框图和真值函数表

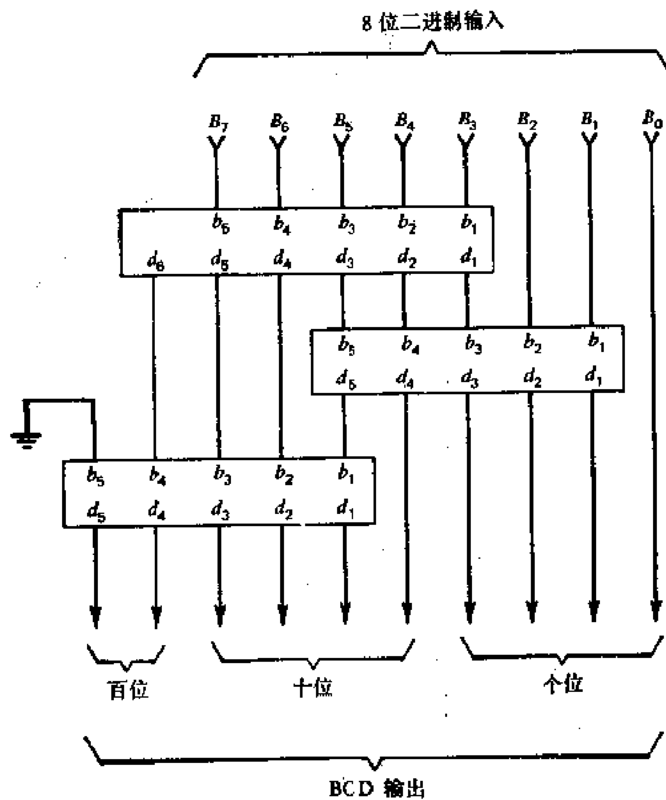
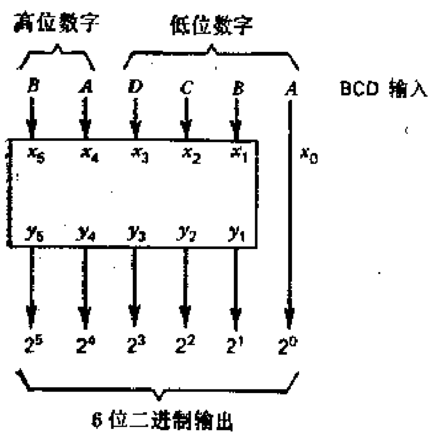


图 2.7 用三块二进制到 BCD 转换模块组成一个8位二进制到 BCD 的转换器



BCD字	输入					输出				
	x_5	x_4	x_3	x_2	x_1	y_5	y_4	y_3	y_2	y_1
0-1	0	0	0	0	0	0	0	0	0	0
2-3	0	0	0	0	1	0	0	0	0	1
4-5	0	0	0	1	0	0	0	0	1	0
6-7	0	0	0	1	1	0	0	0	1	1
8-9	0	0	1	0	0	0	0	1	0	0
10-11	0	1	0	0	0	0	0	1	0	1
12-13	0	1	0	0	1	0	0	1	1	0
14-15	0	1	0	1	0	0	0	1	1	1
16-17	0	1	0	1	1	0	1	0	0	0
18-19	0	1	1	0	0	0	1	0	0	1
20-21	1	0	0	0	0	0	1	0	1	0
22-23	1	0	0	0	1	0	1	0	1	1
24-25	1	0	0	1	0	0	1	1	0	0
26-27	1	0	0	1	1	0	1	1	0	1
28-29	1	0	1	0	0	0	1	1	1	0
30-31	1	1	0	0	0	0	1	1	1	1
32-33	1	1	0	0	1	1	0	0	0	0
34-35	1	1	0	1	0	1	0	0	0	1
36-37	1	1	0	1	1	1	0	0	1	0
38-39	1	1	1	0	0	1	0	0	1	1

图 2.8 标准的 BCD 到二进制转换模块的方框图和函数表

转换模块连接在一起，实现 8 位二进制数到 3 位 BCD 数的转换．相反，我们也可以组成一个标准的 6 位 BCD 到二进制转换器，如图 2.8 所示．为了把一个两位的 BCD 数转换成一个 7 位二进制数，需要有两块这种电路．位数更多的单片的 BCD 到二进制转换器或者二进制到 BCD 转换器可用大规模集成电路制造．这种大规模集成电路的内部结构应与文献[2]所论述的相似．

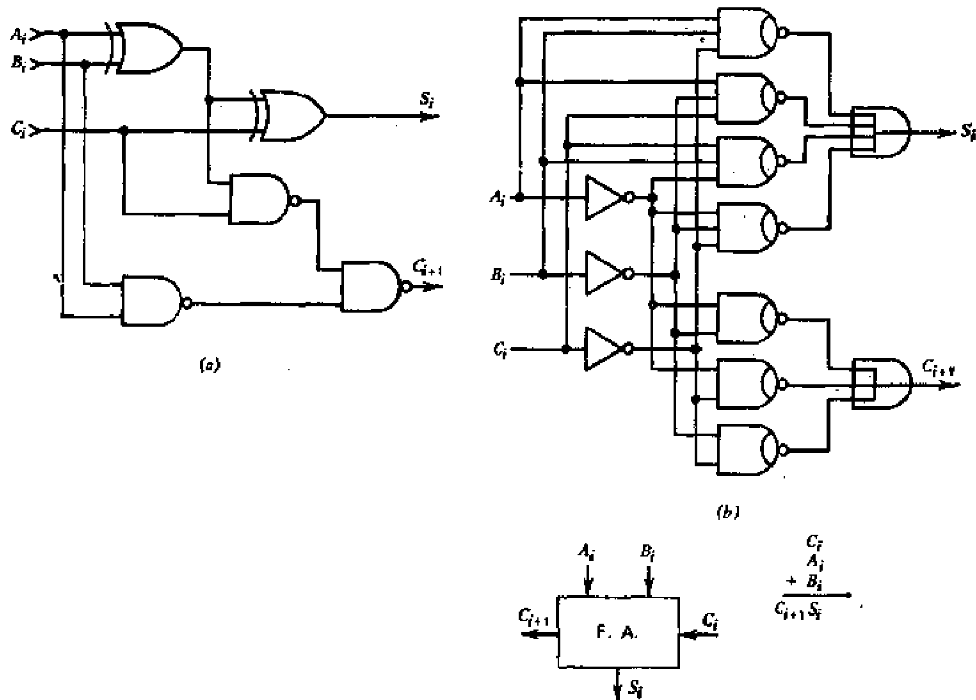
2.3 基本的加法/减法逻辑

下面几节介绍基本的 MSI/LSI 的算术运算功能，诸如加法器，求补器，比较器和乘法器．其他一些便于使用的算术运算功能，诸如先行进位部件（CLA），运算逻辑部件（ALU），以及其它部件，在以后几章要用到时再进行讨论．

一位全加器（FA）把两个二进制数字 A_i，B_i 和一个进位输入 C_i 相加起来，产生一个和输出 S_i，以及一个进位输出 C_{i+1}，见图 2.9．两个输出和三个输入可按以下布尔方程联系起来．实现全加器的不同方法可见文献[17]．

$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_iB_i + B_iC_i + C_iA_i \tag{2.2}$$

这种基本的加法单元可以修改成 4 个输入端和 4 个输出端的可控加/减单元（CAS），如图 2.10 所示．另加的输入 P 用来控制加法（$P = 0$）或减法（$P = 1$）操作．在减法操



$$C_i \times 2^{i+1} + S_i \times 2^i = A_i \times 2^i + B_i \times 2^i + C_i \times 2^i$$

图 2.9 一位全加器的两个可能实现方法的逻辑框图及其逻辑符号．(a) 按下列方程实现的全加器方案：

$$\begin{cases} S_i = A_i \oplus B_i \oplus C_i \\ C_{i+1} = A_iB_i + B_iC_i + C_iA_i \end{cases}$$

其中 S_i 的时间延迟为 6△，C_{i+1} 的时间延迟为 5△；(b) 全加器的接线逻辑实施方案，其中两个输出的时间延迟为最小，只有 2△

作情况下，输入 C_i 称为借位输入，而输出 C_{i+1} 称为借位输出. CAS 的输入与输出的关系，可以用以下一对布尔方程表示：

$$S_i = A_i \oplus (B_i \oplus P) \oplus C_i$$
$$C_{i+1} = (A_i + C_i) \cdot (B_i \oplus P) + A_i C_i \tag{2.3}$$

当 $P = 0$，方程式 (2.3) 就等于方程式 (2.2). 而当 $P = 1$，则有

$$S_i = A_i \oplus \bar{B}_i \oplus C_i$$
$$C_{i+1} = A_i \bar{B}_i + \bar{B}_i C_i + A_i C_i$$

这种 CAS 单元广泛用来组成第六，八和十一章中讨论的叠接运算逻辑阵列.

n 个 1 位的全加器可以级联成一个 n 位的行波进位加法器. 如采用接线逻辑实现内部求和的方法，则每个 1 位的全加器的时间延迟为 2Δ. 目前，以中规模集成封装形式的，产生两个 4 位或 8 位数的二进制和的 4 位或 8 位单片加法器已经可供使用. 其中进位输入与输出线可使它扩展成任何字长. 有些 n 位加法器还具有整个 n 位先行进位的特点. 关于高速加法器中采用的先行进位技术，将在第三章中详细讨论.

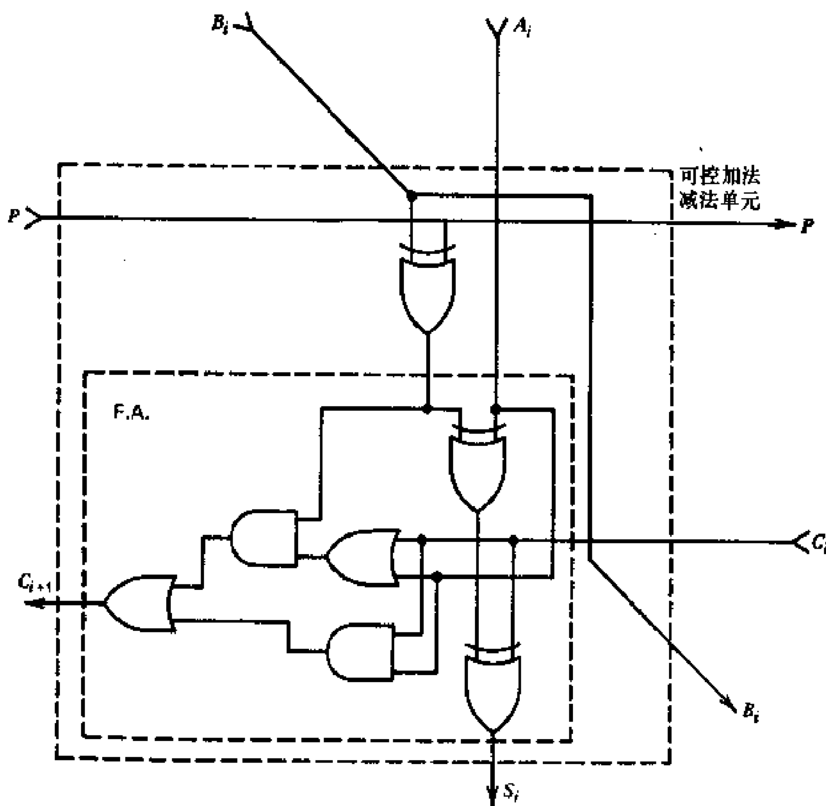


图 2.10 可控加法/减法 (CAS) 单元的逻辑图

2.4 求补器和数值比较器

在算术运算部件设计中，经常用到的二进位求补电路有两种. 对 1 求补 (求反) 器见图 2.11 (a)，当控制线信号为 1，它通过异或 (XOR) 门执行按位求反. 这些按位求反的操作是独立的，为了把任何带符号的二进位数变成反码形式，只需 3Δ 的时间延迟. 如果

要把一个 $(n+1)$ 位的带符号数值的数求补成 1 的补数，那么，前面的符号位可以用作控制信号.

在对 2 求补的情况下，要采用按位扫描技术来执行所需的求补操作. 令 $\mathbf{A}=a_n\cdots a_1a_0$ 是给定的 $(n+1)$ 位带符号的数，要求确定它的补码形式. 所谓按位扫描就是从数的最右端 a_0 开始，由右向左，直到找出第一个"1". 例如 $a_i=1$，其中 i 是最低的数值，而且 $0\leqslant i\leqslant n$. 于是，a_i 以右的每一个输入位，包括 a_i 自己，都保持不变，而 a_i 以左的每一个输入位都求反，即"1"变成"0"和"0"变成"1". 这种扫描方式的对 2 求补器可以用图 2.11 (b) 所示的电路来实现. 其中从右开始的输出位是和输入位相等的，直到遇到第一个"1"为止，而从这个特殊位的位置开始，执行按位求反的操作. 横向链式线路中的第 i 扫描级的输出 C_i 为 1 的条件是：第 i 级的输入位 $a_i=1$，或者第 i 级链式输入（来自右起前 $(i-1)$ 级的链式输出）$C_{i-1}=1$. 在最右端的起始链式输入 C_{-1} 必须永远置成"0". 控制线 E 启动对 2 求补的操作. 当控制线信号为"0"，输出将和输入相等. 同以前一样，可以利用符号位作为控制信号.

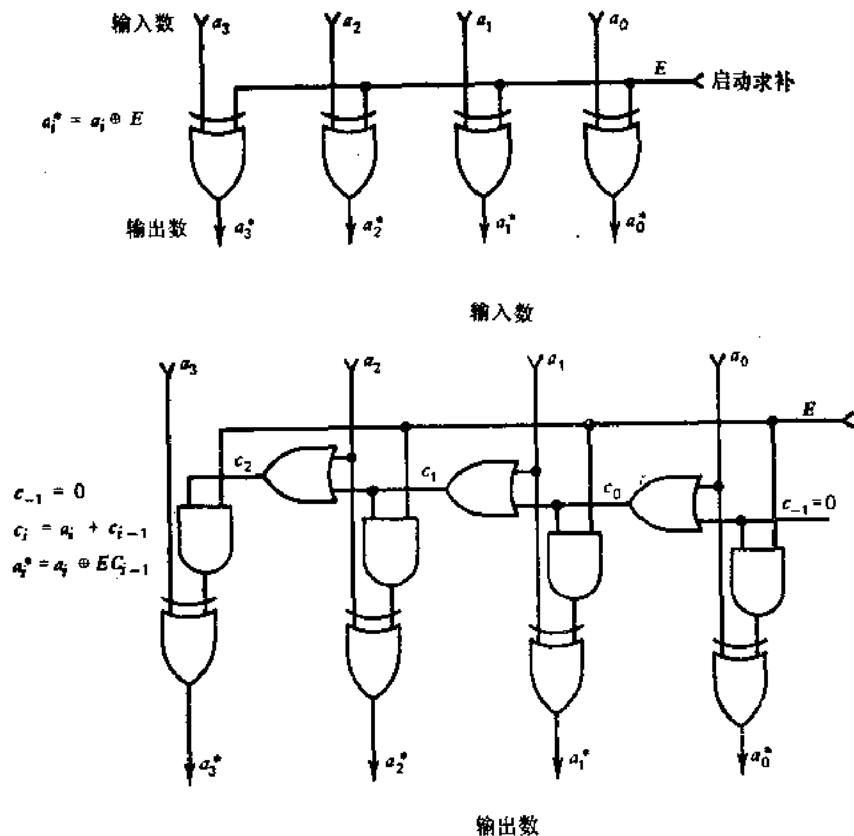


图 2.11 两个具有使能控制的二进位求补器的线路图.
(a) 4 位的对 1 求补器; (b) 4 位的对 2 求补器

举一个例子，在一个 4 位的对 2 求补器中，输入数如为 1010，则输出应是 0110，其中从右算起的第 2 位，就是所遇到的第一个"1"的位. 用这种组合逻辑对 2 求补器来转换一个 $(n+1)$ 位带符号的数，所需的总时间延迟为

$$\Delta_{TC}=n\cdot 2\Delta+5\Delta=(2n+5)\cdot\Delta \tag{2.4}$$

其中每个扫描级需 2Δ 延迟，而 5Δ 则是由于与门和异或门引起的。

4 位的数值比较器见图 2.12. 该部件比较两个 4 位的二进制数 $\mathbf{A}=a_3a_2a_1a_0$ 和 $\mathbf{B}=b_3b_2b_1b_0$，并产生三个输出：$\mathbf{A}=\mathbf{B}$，$\mathbf{A}>\mathbf{B}$，以及 $\mathbf{A}<\mathbf{B}$. 这种部件完全可以扩充到任何位数，而不需要外加门电路. 即可以用级联 4 位比较器的办法，来比较更长的字. 级联输入就是为此目的使用的. 图 2.13 阐明了用六个 4 位比较器组成 24 位比较器的连接方法. 目前，使用组合比较器，可以在大约 40ns 的时间内比较两个 24 位的数. 数值比较也可以用运算逻辑部件中的加法器来执行，这一点将在第 4 章中讨论.

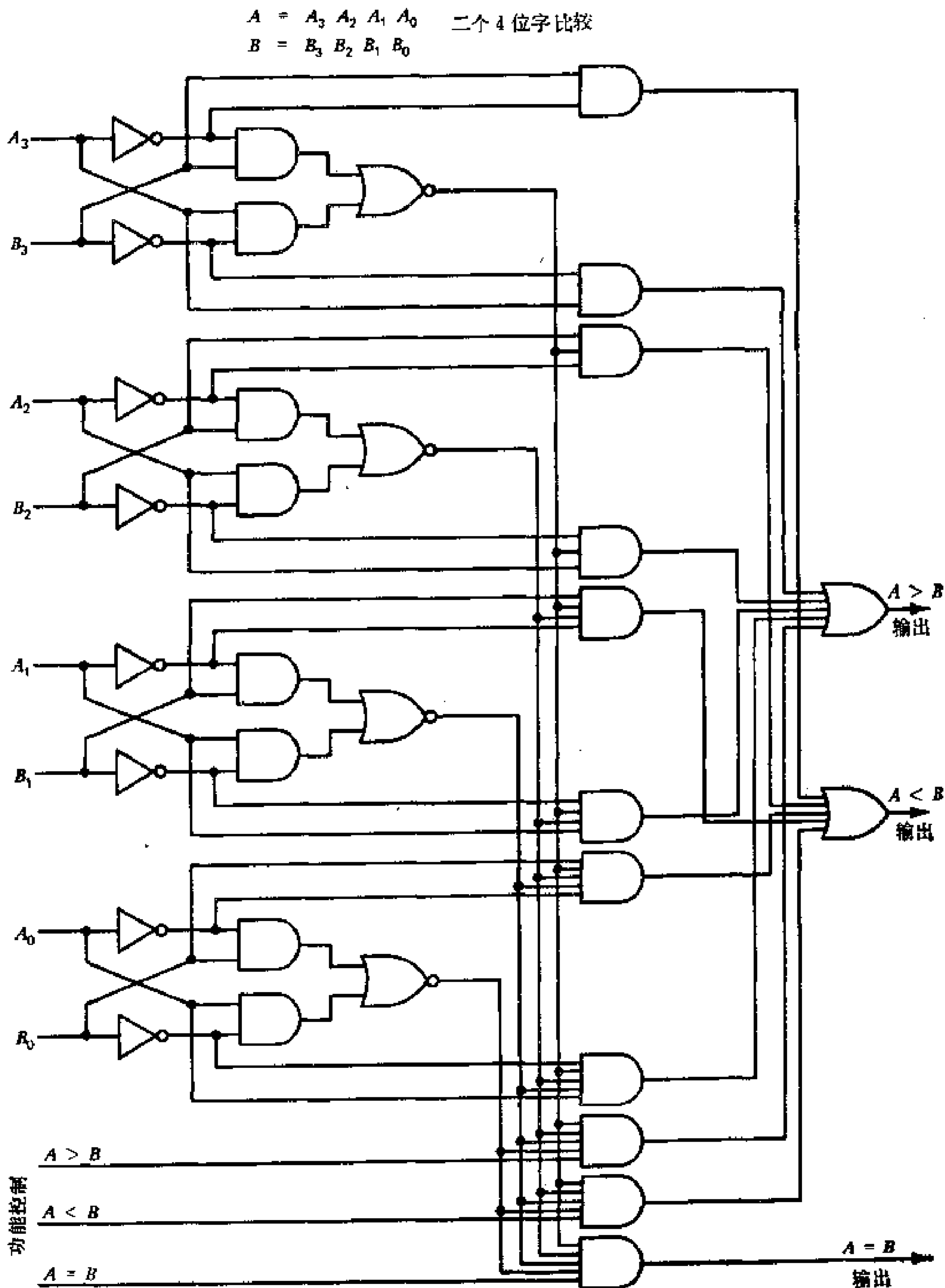
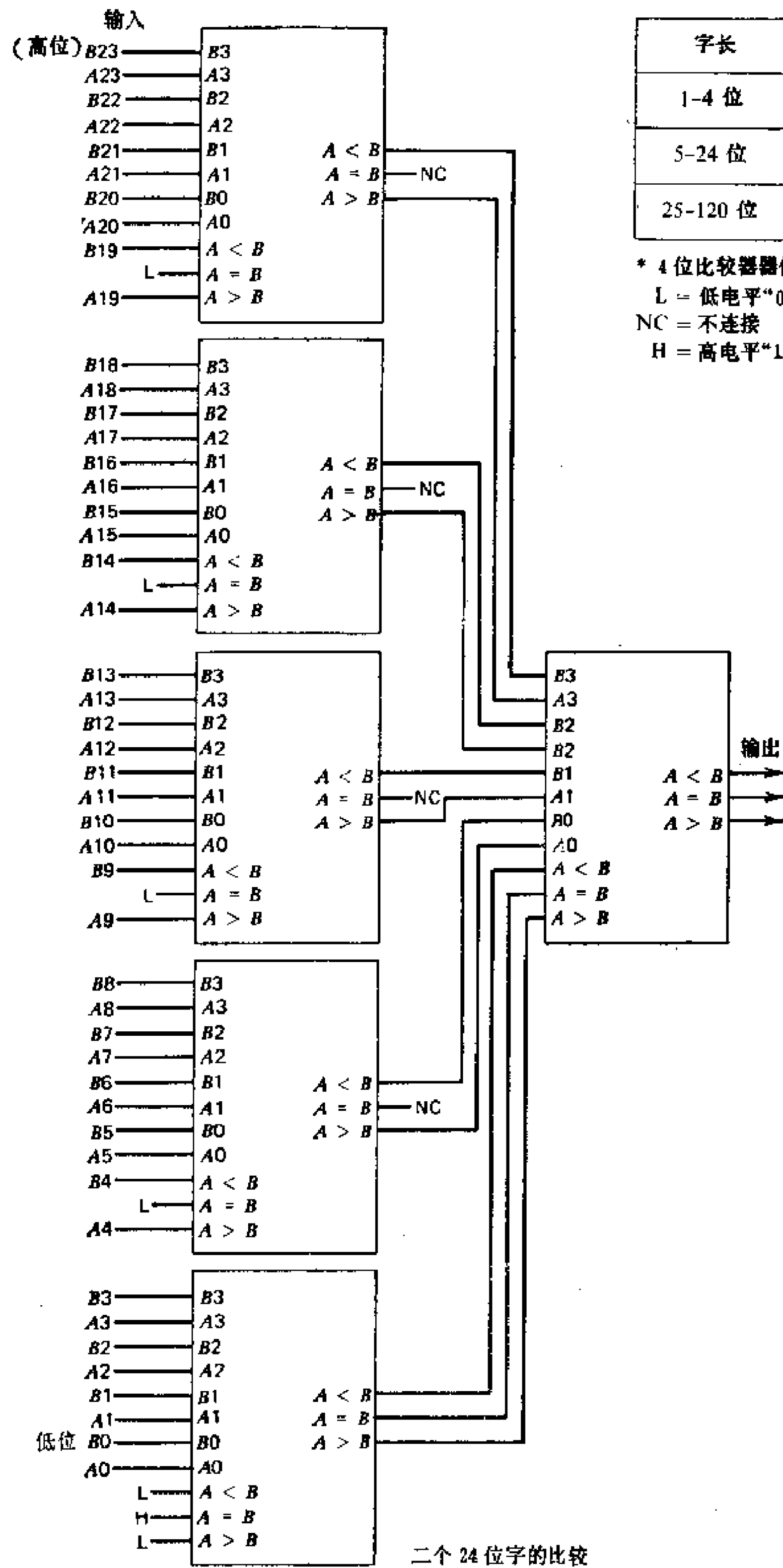


图 2.12 4 位数值比较器的逻辑图



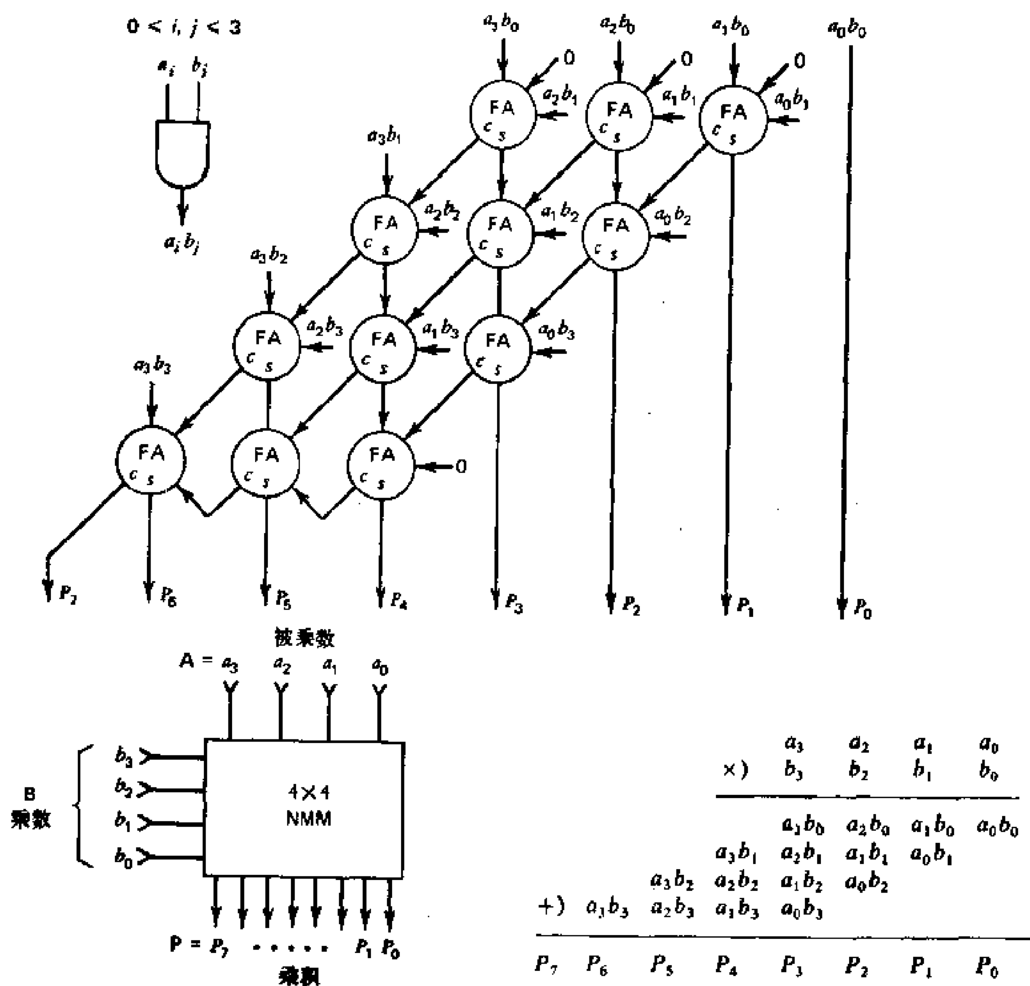
字长	器件数*
1-4 位	1
5-24 位	2-6
25-120 位	8-31

* 4位比较器器件
L = 低电平"0"
NC = 不连接
H = 高电平"1"

图 2.13 一个24位的数值比较器，它是由六个4位比较器级联组成的，表中列出不同字长的比较器所需的器件数

2.5 模块式阵列乘法器

以下介绍模块式组合逻辑阵列，它用于对短字长或中等字长的操作数执行快速乘法．阵列乘法模块可以分成两种类型．第一种叫做非相加乘法模块（NMM），用于执行局部乘法（经过移位的各被乘数的总和），它与其他乘法模块的结果无关．图 2.14 指出了用 12 个全加器和 16 个与门组成的、4 × 4 非相加乘法模块的设计．这个器件把两个 4 位数 $\mathbf{A}=A_3A_2A_1A_0$ 和 $\mathbf{B}=B_3B_2B_1B_0$ 相乘，产生一个 8 位的乘积 $\mathbf{P}=\mathbf{A}\times\mathbf{B}=P_7P_6\cdots P_1P_0$．对这种模块，18条引线的集成电路封装就已经足够了．



				a_3	a_2	a_1	a_0
			×)	b_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+)	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

图 2.14 4×4 非相加乘法模块（NMM）的线路图，逻辑符号及操作矩阵

这种基本的乘法模块可加以修改，接受某些附加的输入，并把它们与形成的积相加．这个相加的特点，使之有可能用很多较小的相加乘法单元连接起来，构成一个任意的 $n\times n$ 并行乘法阵列．图 2.15 表示 4 × 2 相加乘法模块（AMM）的功能结构，它计算算术函数 $\mathbf{P}=\mathbf{A}\times\mathbf{B}+\mathbf{C}+\mathbf{D}$．其中 $\mathbf{A}=a_3a_2a_1a_0$ 和 $\mathbf{B}=b_1b_0$ 是被乘数和乘数．$\mathbf{C}=c_3c_2c_1c_0$ 和 $\mathbf{D}=d_1d_0$ 是两个相加输入，而 $\mathbf{P}=P_5P_4P_3P_2P_1P_0$ 是所得到的积．为了产生乘积项 a_ib_j，其中 $0\leqslant i\leqslant 3$ 和 $0\leqslant j\leqslant 1$，共用了八个"与"门．而为了把乘积项和外部的附加输入相

表 2.2 各种实际规模的、基本阵列乘法模块（NMM 和 AMM）的应用参数

阵列大小 $m\times n$	非相加乘法模块（NMM）		相加乘法模块（AMM）	
	外部引线数[1)]	延迟时间[2)]	外部引线数[1)]	延迟时间[2)]
2×2	10	4△	14	6△
4×2 或 2×4	14	8△	20	10△
4×4	18	12△	26	14△
8×4 或 4×8	26	20△	38	22△
8×8	34	28△	50	30△

1）每个电路片包括电源和地两条附加引线.

2）用于同时产生所有被加数项的"与"门所引起的 2△ 延迟除外.

加，则必需有 8 个全加器的阵列．这些 4 × 2 乘法器已有 20 条外部引线的中规模集成电路可供使用．现在大小为 16 × 16 的单片阵列乘法器也已使用[36]．我们在第六章中将讨论，如何把这些基本乘法模块组成大型乘法网络． 表 2.2 总结了两种类型不同大小的乘法模块的应用参数．要使用非相加乘法模块，还必须有附加的求和器件，诸如多操作数华莱士（Wallace）树形加法器．这些设备的细节以及阵列乘法器结构的通用解决办法见第六章．

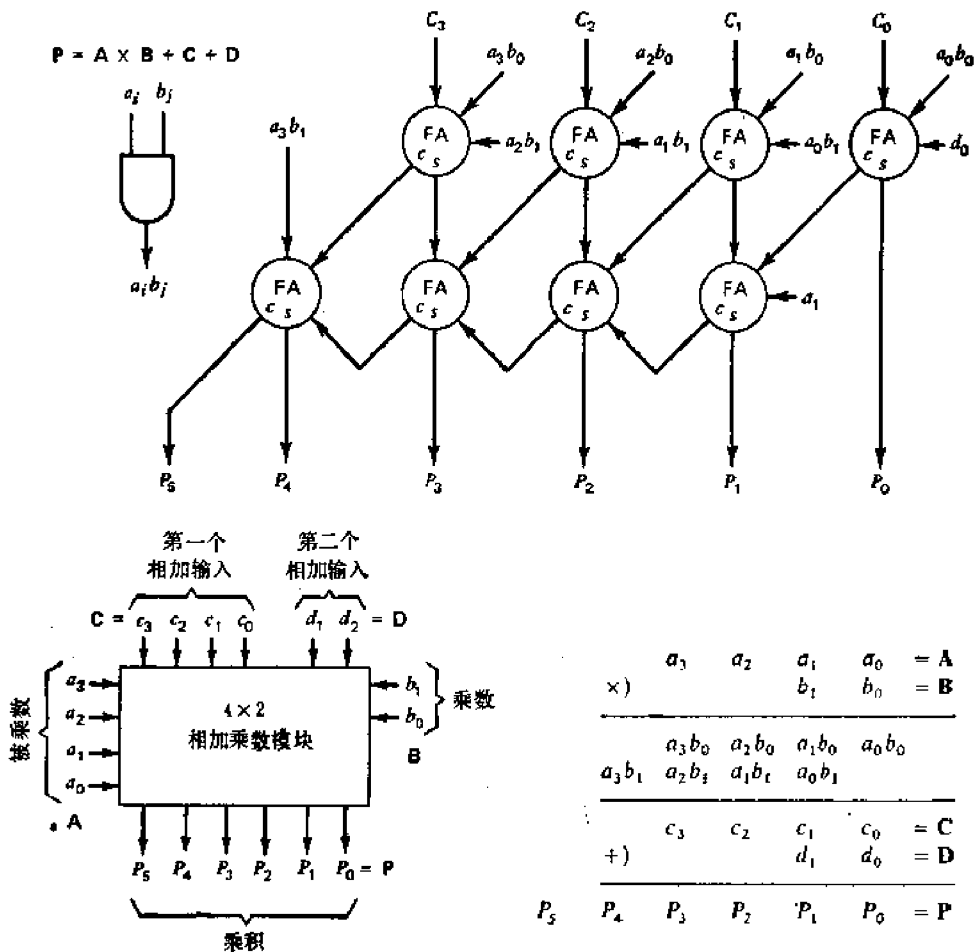
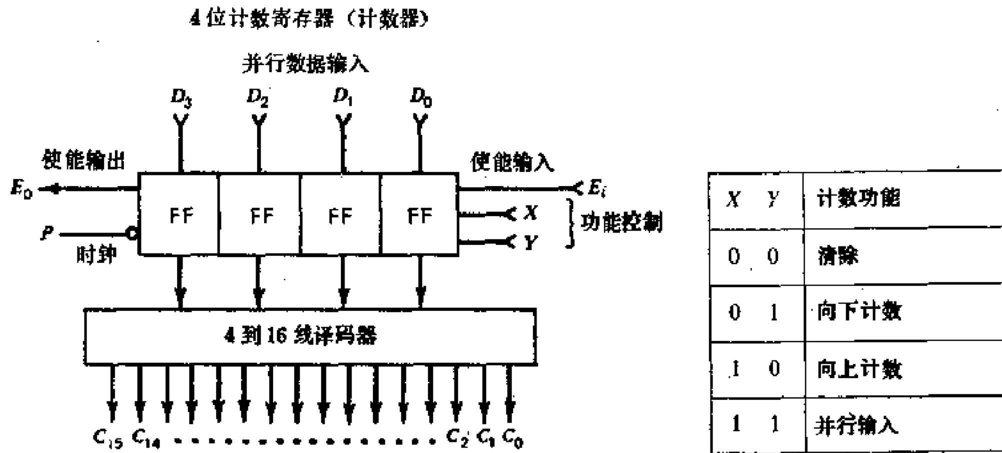


图 2.15 4×2 相加乘法模块（AMM）的线路图，逻辑符号及操作矩阵

2.6 多功能寄存器的特点

本节将介绍在数字算术运算处理器中工作寄存器的功能特点．这些硬件特点在算术运算处理器经常使用的控制状态计数器、移位器、求补器和通用寄存器的设计中，起着关键作用．

我们从计数寄存器的设计开始，当它和译码器一起工作时，可以控制数字系统中的状态转换．16 状态的控制计数器的方框图见图 2.16．在两条功能选择线 X 和 Y 的控制下，这个部件中的 4 位计数寄存器可以执行四种不同的功能——清除，向上计数，向下计数以及并行输入．4 到 16 的行译码器用来译码 16 个计数状态．



X	Y	计数功能
0	0	清除
0	1	向下计数
1	0	向上计数
1	1	并行输入

图 2.16　带有循环译码器的 16 状态控制计数器的方框图和功能表

X Y	0 0	0 1	1 0	1 1
功能	右移	左移	输入 A	输入 B

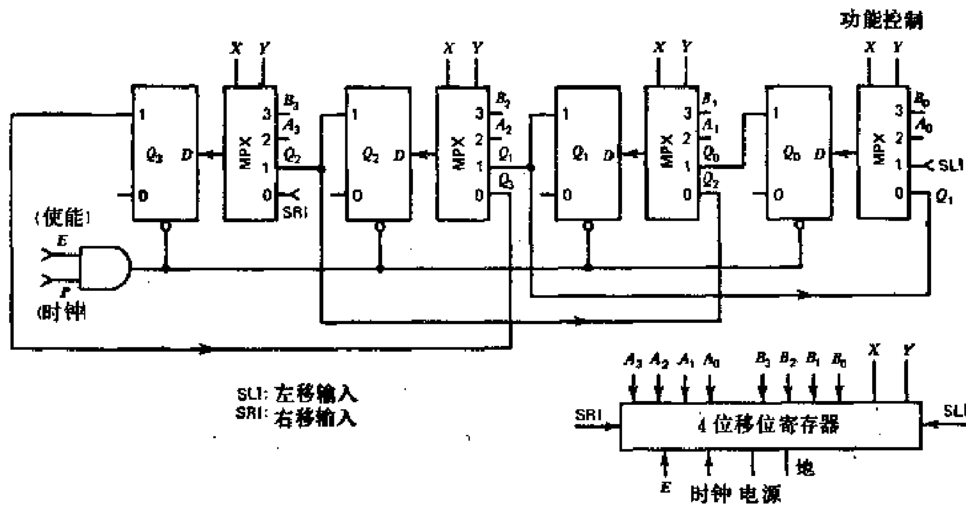


图 2.18　具有并行输入入口的 4 位双向移位寄存器的逻辑线路图

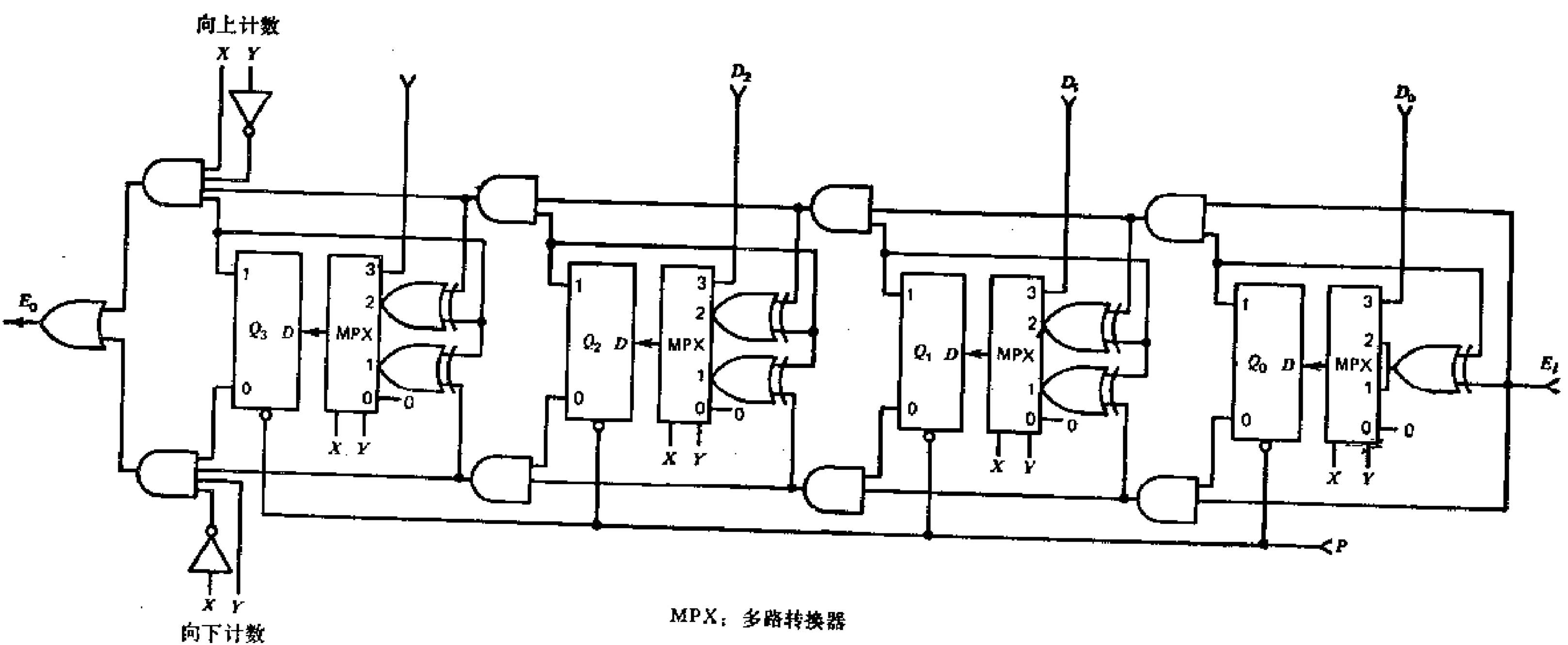


图 2.17　具有消除，向上计数，向下计数和并行输入功能的 4 位计数寄存器（图 2.16 所示）的逻辑线路图

D_3 D_2 D_1 D_0

PL OC CR

2 到 4 译码器 w_3 w_2 w_1 w_0

X Y

预先清除 PC 1 0 J K C Q_3 Q_2 Q_1 Q_0

时钟 P

T_0 T_i TC

PL：并行输入　　$D = D_3D_2D_1D_0$　　OC：1 的补码
TC：2 的补码　　　　　　　　　　　　CR：清除寄存器

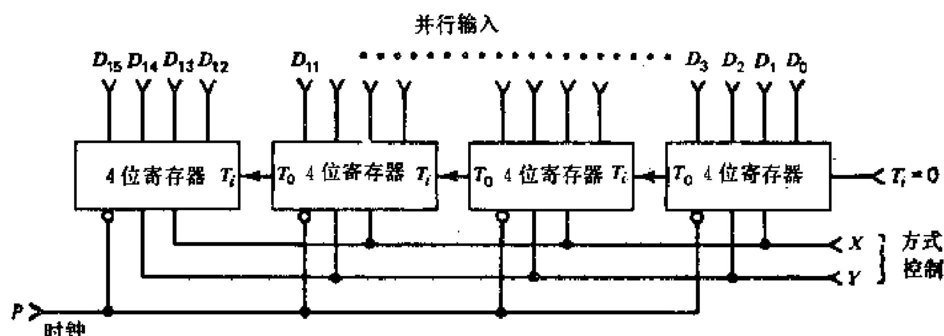
图 2.19　具有为扩展目的所需的进位输入 T_i 和进位输出 T_0 的 4 位和四种功能求补寄存器的逻辑图

4 位计数寄存器的内部线路如图 2.17 所示．在计数寄存器中每个触发器的触发逻辑是用一个 4 输入端的多路转换器来实现的．为了扩展这个 4 位寄存器，我们在每一个 4 位片上加了一条允许计数输入线 E_i 和一条允许计数输出线 E_0．例如，我们可以把两个 4 位计数寄存数连接起来，构成一个可以计数到 $2^8-1=255$ 状态的 8 位寄存器．较低一级的输出线 E_0 接到较高一级的输入线 E_i．最右边一级的输入置成 1．

在算术运算处理器中经常用到移位寄存器或简单的移位器．4 位双向的并行输入移位寄存器见图2.18．在左移时加到最右端触发器的外部输入用 SLI（左移输入）表示，而右移时加到最左端触发器的外部输入用 SRI（右移输入）表示．对于每个触发器的四种可能的输入选择为右移，左移，输入 A 和输入 B．允许操作端 E 用来控制时钟．当 E 为低，时钟脉冲不能通过．当时钟不允许进入时，寄存器将保持其原有数据．

可以设计出在外部控制下能对自己求补的寄存器．在图 2.19 中指出了一个 4 位的具有四种功能的求补寄存器．这四种功能是 CL(清除)，OC（对 1 求补)，TC（对 2 求补)，和 PL（并行输入)．这个线路并不用多路转换器来选择输入端，而是用 J-K 触发器来实现．输入触发逻辑则用随机逻辑门和一个功能译码器来实现．

在对 1 求补操作中执行按位求反．在对 2 求补操作中可能会产生各位之间的进位．在每个 4 位片中引入一条进位输入线 T_i 和一条进位输出线 T_0，这样它们可以级联起来形成任意字长．图 2.20 表示用四个 4 位求补寄存器构成一个 16 位多功能求补器的连接方法．和图 2.11 b 所示的一样，最右级的进位输入 T_i 永远是零．



X	Y	寄存器功能
0	0	清除 (CR)
0	1	1 的补码 (OC)
1	0	2 的补码 (TC)
1	1	并行输入 (PL)

图 2.20· 用四个 4 位求补寄存器(图 2.19) 构成的一个 16 位求补寄存器

2.7 通用寄存器的设计

除以上讨论的计数、移位和求补操作以外，还可以在一个通用寄存器中添设附加的逻

辑操作，诸如按位异或，与，或，置位以及其他．我们选择一个具有八种功能的寄存器的设计，来说明在构成通用寄存器时的两种不同方法． 表 2.3 规定用 D 触发器构成的寄存器所执行的八种功能．触发器下一状态的方程式表示在下面．

表 2.3 n 位通用寄存器功能说明和下一状态

FNC 译码	功能控制 X Y Z	寄存器功能	触发器的下一状态 $Q_{n-1}(t+1)$	$Q_i(t+1)$	$Q_0(t+1)$
CR	0 0 0	清除	0	0	0
SR	0 0 1	右移	SRI	$Q_{i+1}(t)$	$Q_1(t)$
SL	0 1 0	左移	$Q_{n-2}(t)$	$Q_{i-1}(t)$	SLI
PL	0 1 1	并行输入	D_{n-1}	D_i	D_0
XOR	1 0 0	异或	$Q_{n-1}(t)\oplus D_{n-1}$	$Q_i(t)\oplus D_i$	$Q_0(t)\oplus D_0$
AND	1 0 1	与	$Q_{n-1}(t)\cdot D_{n-1}$	$Q_i(t)\cdot D_i$	$Q_0(t)\cdot D_0$
OC	1 1 0	对 1 求补	$\bar{Q}_{n-1}(t)$	$\bar{Q}_i(t)$	$\bar{Q}_0(t)$
CU	1 1 1	向上计数	$Q_{n-1}(t)\oplus(Q_{n-2}(t)\cdots Q_0(t))$	$Q_i(t)\oplus(Q_{i-1}(t)\cdot Q_{i-2}(t)\cdots Q_0(t))$	$\bar{Q}_0(t)$

最高位触发器的下一状态的方程为

$$\begin{aligned}Q_{n-1}(t+1) = {} & \mathrm{SR}\cdot \mathrm{SRI} + \mathrm{SL}\cdot Q_{n-2}(t) + \mathrm{PL}\cdot D_{n-1} + \mathrm{XOR}\cdot(Q_{n-1}(t)\oplus D_{n-1}) \\ & + \mathrm{AND}\cdot(Q_{n-1}(t)\cdot D_{n-1}) + \mathrm{OC}\cdot \bar{Q}_{n-1}(t) \\ & + \mathrm{CU}\cdot(Q_{n-1}(t)\oplus(Q_{n-2}(t)\cdot Q_{n-3}(t)\cdots Q_1(t)\cdot Q_0(t)))\end{aligned} \tag{2.5}$$

对于第 i 位的触发器，$i=1, 2, \cdots, n-2$ 下一状态方程为

$$\begin{aligned}Q_i(t+1) = {} & \mathrm{SR}\cdot Q_{i+1}(t) + \mathrm{SL}\cdot Q_{i-1}(t) + \mathrm{PL}\cdot D_i + \mathrm{XOR}\cdot(Q_i(t)\oplus D_i) \\ & + \mathrm{AND}\cdot(Q_i(t)\cdot D_i) + \mathrm{OC}\cdot \bar{Q}_i(t) \\ & + \mathrm{CU}\cdot(Q_i(t)\oplus(Q_{i-1}(t)\cdots Q_0(t)))\end{aligned} \tag{2.6}$$

对于最低位的触发器的下一状态方程为

$$\begin{aligned}Q_0(t+1) = {} & \mathrm{SR}\cdot Q_1(t) + \mathrm{SL}\cdot \mathrm{SLI} + \mathrm{PL}\cdot D_0 + \mathrm{XOR}\cdot(Q_0(t)\oplus D_0) \\ & + \mathrm{AND}\cdot(Q_0(t)\cdot D_0) + (\mathrm{OC}+\mathrm{CU})\cdot \bar{Q}_0(t)\end{aligned} \tag{2.7}$$

这些方程式可以用 n 个 8 输入多路转换器来实现，类似于图 2.17 和图 2.18．随机逻辑也可以用来实现与图 2.19 类似的触发逻辑．后一种方法称为叠加，它需要更多种类的门功能和更多的集成电路片，而用多路转换器的方法所需的设计工作较少，并能减少电路片的数目．

2.8 半导体随机存取存储器和只读存储器

随机存取存储器（简称 RAM）指的是这样一种固态存储器，这里计算机可以用相等的存取时间，对任何存储单元进行信息的读出或写入，而不管这个存储单元在存储矩阵中处在什么物理位置．MOS 和双极型的 RAM 两者都有集成电路封装形式的片子．RAM 可以用来作为高速便笺存储器，缓冲存储器，或者甚至用作数字系统中的主存储器．目前，大规模集成 RAM 中大部分是用 MOS 工艺生产的．而较快的双极晶体管 RAM 的规模也在增加，但价格仍然较高．

图 2.21 表示一个用三发射极晶体管的标准 TTL 单元做成的，8 个字每字 n 位的双极

型 RAM 的基本结构．对存储器字的地址选择，是采用同时把 X 和 Y 选择线提到"高"电压的电压重合选择法．在读出时，读/写使能线处在高电位，位线"通导"一侧的信号电流将流入电流读出放大器．要把信息写入所选中的单元时，读/写使能线保持在低电位，允许外界数据"1"或"0"通过写入放大器传送，使该晶体管单元"通导"或"截止"．几种主要的集成半导体存储器工艺的取数时间，周期时间，工作功耗以及密度列于表 2.4 中．

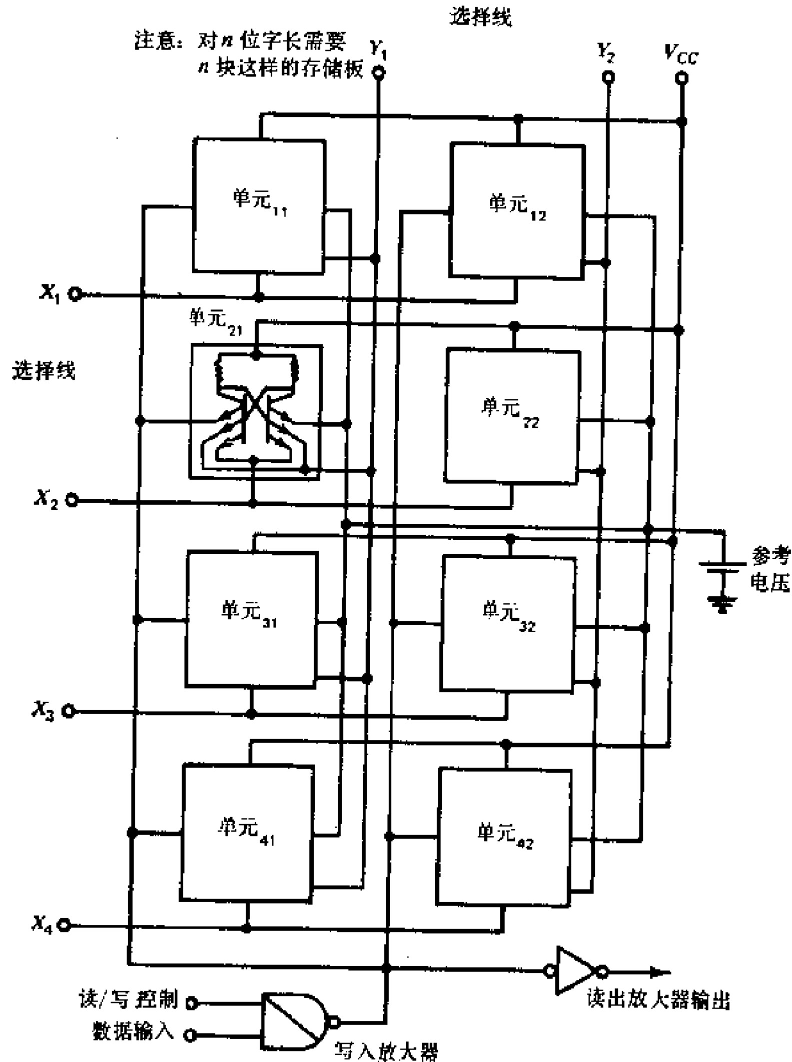


图 2.21　具有 8 个字（每字 n 位）的 RAM 系统中每位存储器阵列的略图，它采用电压重合技术，依靠四条 X 选择线和两条 Y 选择线来选择地址

用来存储固定程序或者永久性数据表的半导体存储器叫做只读存储器（简称 ROM）．ROM 的数据内容只能写一次．对 ROM 的编程通常是在制造时通过掩模做成的．某些类型的 ROM 可以由用户来编写程序，叫做可编程序只读存储器（PROM）．ROM 也可以随机存取．它与 RAM 不同之处仅仅在于当编好程序后，ROM 是不能重写的．ROM 没有重写能力，制造比较便宜．对于某些专门用途，有一种新的电子可变只读存储器（简称 EAROM）已可供使用．这种 EAROM 可以重新编写程序，其写入时间比 RAM 慢．因为 EAROM 主要仍用于读出，它们有时称作主读存储器（RMM）或者可写入的 ROM．ROM

的通常用途包括数码转换(编码或译码),字母数字符号的产生,微程序控制的时序逻辑的实现,控制状态的定时,以及用查表方法产生基本函数(三角、对数等等).

表 2.4 主要集成半导体存储器工艺的一般特性 (1977 年)

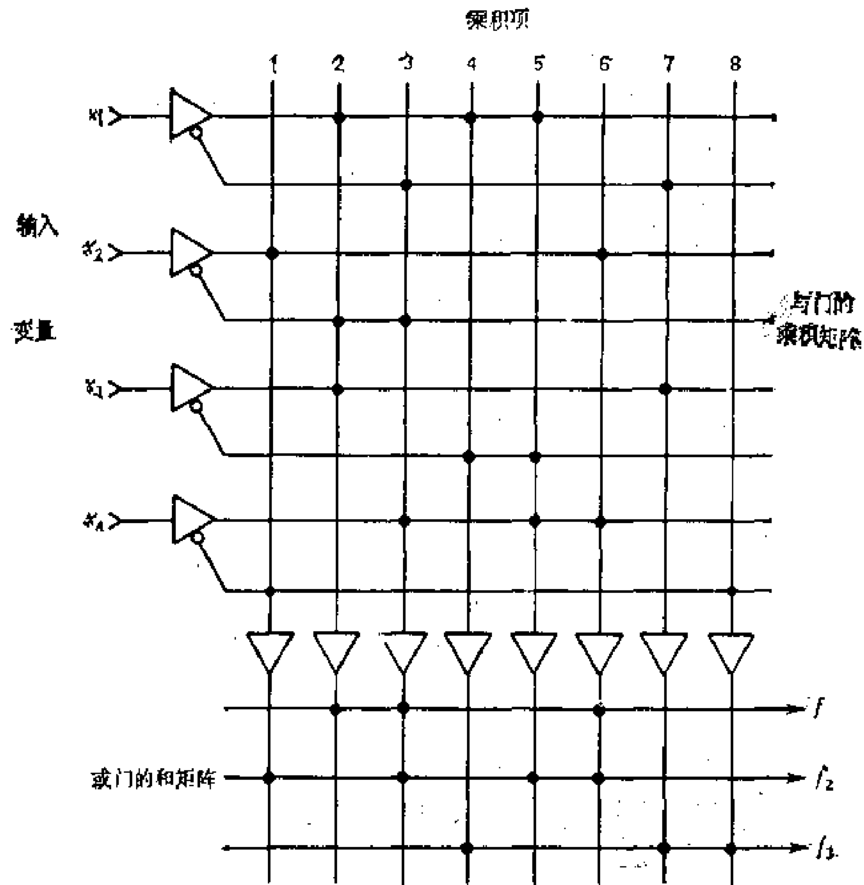
工 艺	特		性		
	取数时间 (ns)	典型密度 (位/片)	工作功耗(每位) (μW)	非易失性	用户价格 (美分/位)
PMOS	300—400	16K	100	用电池	0.25
NMOS	150—250	16K	100	用电池	0.20
CMOS	50—60	4K	20	用电池	1.20
CCD	80—250	64K	10	用电池	0.08
TTL	50—60	4K	500	用电池	0.80
ECL	45—60	1K	600	用电池	2.50
I^2L	50	1K	400	用电池	4.50
MBM	3000	128K	10	是	0.02

2.9 可编程序逻辑阵列 (PLA)

与存储有关的半导体器件系列中比较新的一种是字段可编程序逻辑阵列 (FPLA),它能满足微处理器和其它计算机系统中对非标准逻辑功能的需要.在实际的数字设计中,有时发现很难用有限个集成电路去组成一个最节省的电路,而用 FPLA 可以很容易解决这个问题. FPLA 在实现任意开关函数这一点上是和 PROM 类似的. 但这两种用户可编程序的器件之间区别在于: 在 FPLA 中应该只包括素项(必要的乘积项),而在 ROM 中则应对所有小项 (典型乘积项) 都编程序. 在 PROM 中,要用固定地址译码器来选择地址,这很费力,会使存储器编码效率变坏. 而 FPLA 废除了固定地址译码器,利用可编程序的地址矩阵来选择所需要的素项.

FPLA 在功能上相当于很多"与"门的集合,它们可以在任何一个输出端上"或"起来. 3个和项中每一个包含 8 个乘积项(对 4 个输入变量)的小型 PLA 的结构,见图 2.22. 每一个和项("或")控制一个输出函数,它可以用外界电脉冲来编排程序,直到包括 8 个乘积项. 交叉线上的点,在矩阵上部相当于"与"门,而在矩阵下部则相当于"或"门. 输入变量的每一行可以被地址矩阵的每一列识别为"真","假",或者"随意值". 其结果是,无用的小项所浪费的存储单元不再需要了. 对于不作用小项所必须的逻辑输出以"缺席"表示. 因此,为了实现相同的逻辑函数,用 FPLA 比用 PROM 所需要的位数要少,译码逻辑也少.

现在,具有 8 个输出函数、每一个包含 16 个输入和多到 48 个乘积项的 FPLA 已可供使用. 图 2.23 表示 Signetics 公司的 82S101 双极型 FPLA 的逻辑线路图,它具有供扩展乘积项用的三态输出. 片选端 ($\overline{CE}$) 控制输入变量的扩展,并禁止三态输出. 这种 Signetics 的 FPLA 的最大取数时间为 50 nS, 功耗为 600 mW. FPLA 的主要用途包括逻辑压缩,存储器重叠操作,组成故障检测网络,快速多位移位器以及优先中断系统.



乘积项	输入变量				输出功能		
	x_1	x_2	x_3	x_4	f_1	f_2	f_3
1	d	1	d	0	0	1	0
2	1	0	1	d	1	0	0
3	0	0	d	1	1	1	0
4	1	d	0	d	0	0	1
5	1	d	0	1	0	1	0
6	d	1	d	1	1	1	0
7	0	d	1	d	0	0	1
8	d	d	d	0	0	0	1

d = 任意

图 2.22 PLA 的逻辑矩阵和真值表的描述．这个 PLA 具有 3 个输出函数，每个输出可包含 8 个乘积项及 4 个输入变量

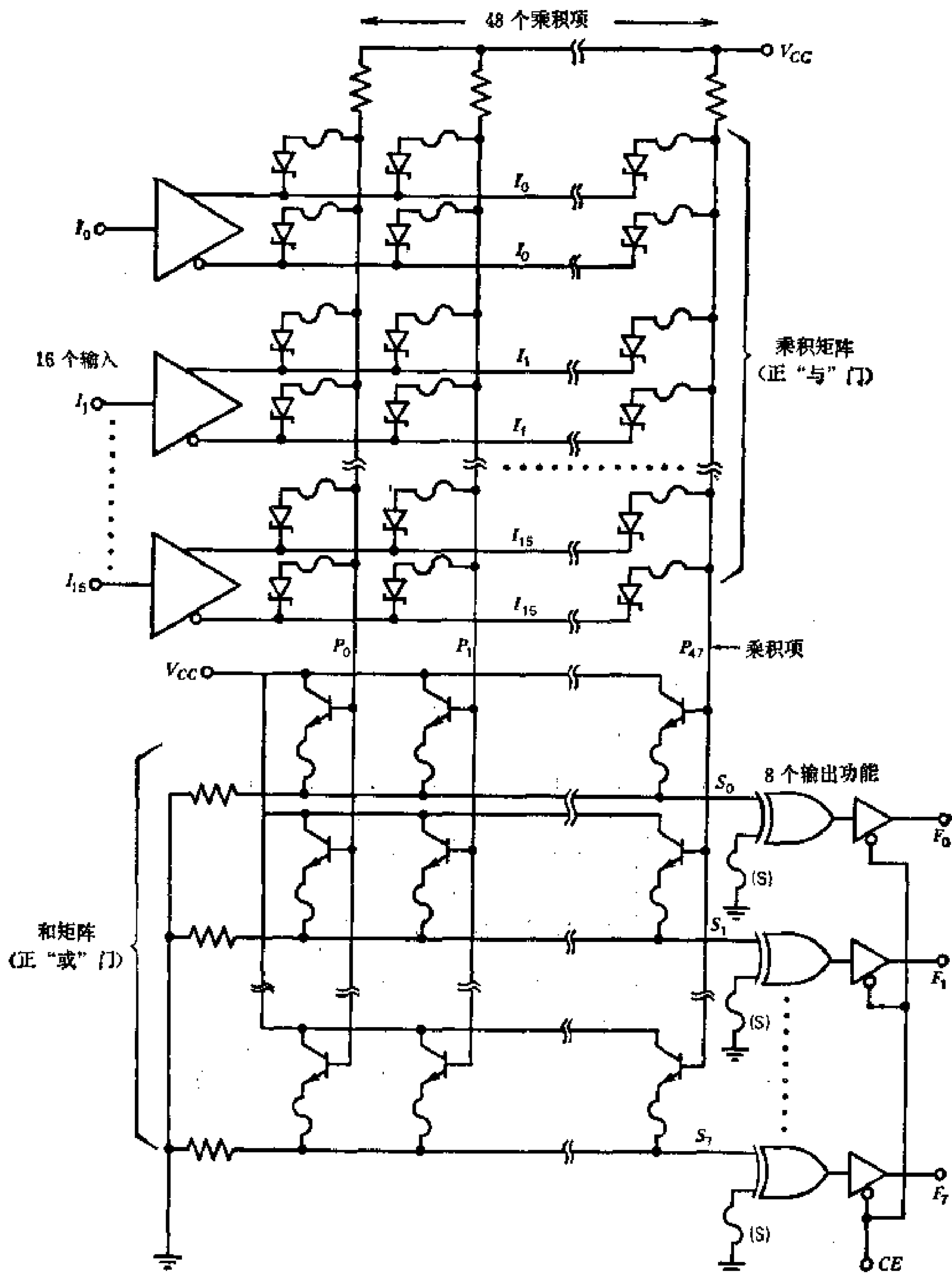


图 2.23 Signetics 82S 101 双极型字段可编程序逻辑阵列（FPLA）的线路图．
它有 16 个输入变量和每个可包含 48 个乘积项的 8 个输出函数

2.10 磁泡存储器(MBM)

磁泡存储器可以看成是与转动的机电存储器(如磁盘,磁鼓和磁带等)相似的固态集成部件. 两者都是把信息以磁化区域的形式存储. 集成磁泡存储器的磁化区域是在磁性物质的薄层中的圆柱形磁畴上,其磁化方向与周围区域相反. 在特定位置上存在或不存在磁泡,相应于在这些位置上存二进位"1"或"0". 这里通过把固态薄层中的磁畴移动到一个取数器件来实现对这些二进位的访问,与此相反,磁盘或磁带部件则是依靠在物理位置上移动存储介质.

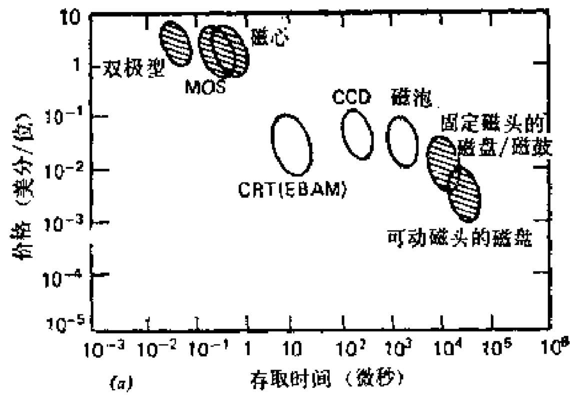
存储器材料可以是用磁性的石榴石外延隐埋在非磁性的石榴石基片内,也可以是在基片(如玻璃片)上溅射一层非晶形的金属磁性薄层. 由于在制造过程中所需的掩模次序较少,并且无需严格的对准,磁泡存储器的每位价格应比硅集成存储器要低. 用磁泡工艺制成的存储器系统有很多有意义的用途,值得注意的是,如用作微型外围设备,以代替一般旋转型磁性存储器.

磁泡存储器片具有下列基本特点: 非易失性、高密度、积木化和低功耗. 磁泡存储器的基本片子、系统特性以及某些定量的特点见表 2.5. 因为磁泡存储器具有移位寄存器的结构,而且在存储器存取时,只是按位地从每个局部磁滞迴线转移到主磁滞迴线,这就有可能采用存储块的结构. 然而,磁泡片是可以按位,按字节或按字访问的,因为坡莫合金的图案可以规定各个分离的存储单元.

把磁泡存储器与其他已有的存储器比较,诸如半导体 RAM,电荷耦合器件(CCD),固定头磁盘,以及软磁盘. Juliussen[15] 曾提出图 2.24 中的两张图. 第一张图表明,就访问时间和价格来说,在半导体 RAM 和旋转型磁性大容量存储器之间有一个空隙区,而电荷

表 2.5 磁泡存储器的一般特性 (Juliussen[15])

电路片特性		定性特点
存储密度	1M 位/in^2	串行存取存储器
片子容量	64K—100K 位	移位寄存器结构
存取时间	1—4 ms	按信息块存取
传送速率	100K 位/秒	按位可编地址
封装形式	10/16 引线双列直插	非易失性存储器
功耗	<1 W	非破坏读出
维持功率	无	读出-修改-写入周期
温度范围	(−25)°~(+75)℃	停止/启动操作
系统特性		降低有效存取时间
传送速率	(0.1M—1.5M)位/秒	降低工作功耗
组装	印刷电路板	允许可变传送速率
存储容量	(1M—3M) 位/板	减少缓冲器的需要
功耗	(10—20)W	积木化的存储器性能
平均失效间隔时间	10000 小时以上	较低的使用价格
控制器	用微处理器	存储器每位价格几乎与存储量无关
		封装引线数量与存储量无关
		生产工序较简便
		生产过程与集成电路相似



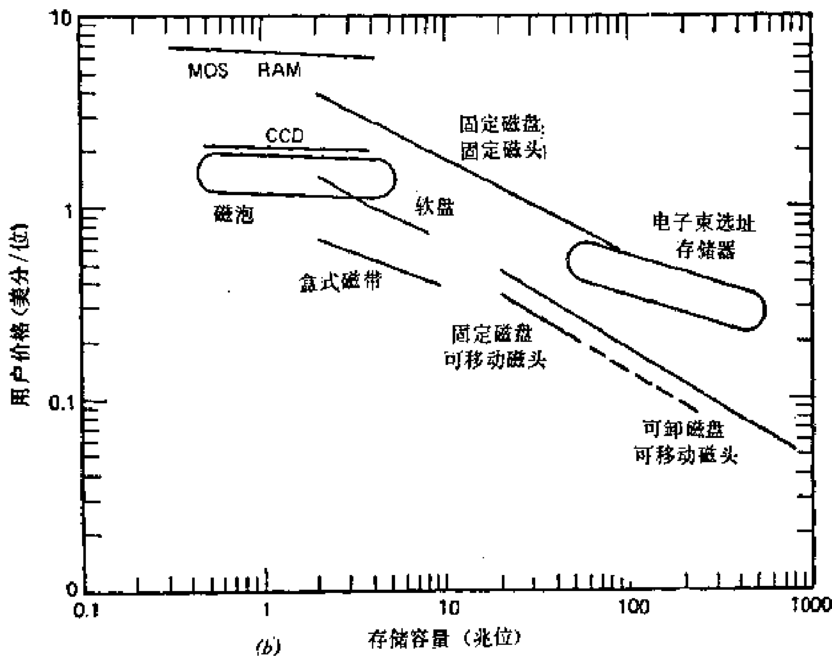


图 2.24 磁泡存储器（MBM）系统如何填补固态存储器与旋转型磁性大容量存储器之间，在存取时间和存储容量方面的间隙．(a) 每位价格与存取时间的关系，(b) 每位价格与存储容量的关系(取自 "Computer Design" 杂志1976[15]，获准重印)

耦合器件，磁泡存储器以及电子束选址存储器（EBAM）可以填补这个空隙．第二张图表示每位价格和存储容量之间的关系．这张图表明 MOS/RAM 和电荷耦合器件的每位价格几乎与存储容量无关，这种积木化可扩充的特点也出现在磁泡存储器中．另一方面，旋转型磁性存储器的价格却随存储容量迅速变化．磁泡存储器和电荷耦合器件在装配上也比旋转型大容量存储器优越，前两种在一块印制电路板上都可容纳几兆位，这完全可满足 CPU 的需要．

磁泡存储器比起电荷耦合器件，其主要优点在于它的非易失性和每片的位密度较高．磁泡存储器的主要缺点是它的传送速率相对地较低，但这可以用多磁泡片并行传送来解决．磁泡工艺是由 Bell 实验室在六十年代后期提出的．现在很多机构，如 Texas Instruments 公司，IBM 公司，Hewlett-Packard 公司，日立公司以及其他公司，都宣布提供 92K 位、256K 位甚至更大的磁泡存储器商品，磁泡存储器的应用包括主机外围设备，小型外围设

备,微型外围设备,小型非易失性存储器,以及作为磁盘和磁带记录器的代用品等等.

2.11 参考文献注释

读者如果愿意丰富自己在数字电子器件方面的知识,那就应该从 Kohonen[16]和 Blakeslee[3]的书开始学习. Kohonen 对于电子线路和数字逻辑原理作了组合的介绍. Blakeslee 的书对常用的 MSI/LSI 数字器件,按其实际应用的性能作了总结. Calhoun[5] 综述了包括四代计算机的计算机硬件技术. Wakerly[37] 最近写了关于用标准集成电路进行设计的新书. 目前已有很多关于大规模集成处理部件(PE)和微处理器及其应用的书籍[13,14,29]. 可能存在一种错误印象, 以为没有必要去设计算术运算部件, 只要简单使用这些大规模集成 PE 或者微处理器片子就可以了. 这种说法只正确一部分. 因为在我们涉及到设计字长较长的高速算术运算部件时, 对于计算功能有限的"慢"和"短"字长的 PE 或微处理器是没有兴趣的,至少在现阶段是如此. 当将来速度的缺陷被弥补,基本微处理器的字长进一步扩展时,这种情况可能改变.

实际的数字工程师可以从电子制造厂家提供的产品目录和数据手册中, 获得集成电路器件的详细特性和应用参数. 其中,有关现代数字集成电路的理论,设计和应用方面论述最全面的是 Texas Instruments 电子丛书[18,20], 以及由 TI 数据手册作为补充[30,31,32,33]. 如要用 TTL 和 ECL 逻辑系列进行设计, 读者可以看到这些数据手册非常有用[4,8,12,20,23,30]. 关于新型器件,例如 CCD 和磁泡存储器方面的资料, IEEE 出版社已经出版了一些有关理论和应用方面的书籍[6,19]. 最近, Juliussen[15] 和 Tung[35] 发表了两篇分别在 TI 和 IBM 研制磁泡存储器方面的有意义的报告. 对于想知道数字逻辑设计技术评述的读者, Mowle 的书[22]是很适宜的,它提供从布尔代数基础到基本算术运算线路设计的系统的方法.

电子制造厂家已经出版了许多目录,数据手册以及应用说明. 除了以上提到的 Texas Instruments 公司和 Intel 公司的刊物以外, 值得注意的还有 Motorola 公司的数据全书[21], Fairchild 公司的应用手册[7,8], RCA 公司的集成电路手册[26], National Semiconductor 公司的集成电路目录[23,24],以及 Signetics 的产品手册[27,28]. 可编程序逻辑阵列(PLA)和磁泡存储器(MBM)的材料分别选自[12,25,20]和[6,15,33]. 全加器电路的最优化见 Liu[17]等的文章. 阵列乘法器可见[1,8,11,36].

关于集成电路数字逻辑系列的评述, Garrett[10]曾对 1970 年以前的逻辑电路系列给予出色的特性分析. 有关数字电子器件和集成存储器的最近发展和将来的趋势, 读者可以阅读计算机硬件方面的最近文章[9, 34, 38]. 这个文献指南并非是完整的. 对于新型数字集成电路器件的详细特性和应用要求,读者有兴趣可以查阅很多集成电路数据手册.

参 考 文 献

[1] Advanced Micro Single-Spaced Devices, "TTL/MSI AM 2505 4-bit by 2-bit Two's Complement Multiplier." Sunnyvale, Calif.. 1972.

[2] Benedek, M., "Developing Large Binary to BCD Conversion Structures," *IEEE Trans. Comp.*, Vol. C-26., No. 7, July 1977, pp. 688—689.

[3] Blakeslee, T. R., *Digital Design with Standard MSI and LSI*, Wiley-Interscience, New York, 1975.

[4] Blood, W., Jr., *MECL System Design Handbook*, Motorola Semiconductor Products, Inc., Mesa,

Arizona, 1972.

[5] Calhown, D. F., "Hardware Technologies." in *Computer Science*, (Cardenas et al. ed.), Wiley-Interscience, New York, 1972, Chap. 1.

[6] Chang, H. (ed.), *Magnetic Bubble Technology: IC Magnetics for Digital Storage and Processing*, IEEE Press, New York, 1975.

[7] Fairchild, *Easy ECL: 9500 Series High-Speed Logic*, Fairchild Semiconductor, Mountain View, Calif., 1971.

[8] Fairchild, *The TTL Application Handbook*, Fairchild Semiconductor, Mountain View, Calif., 1973.

[9] Falk, H., "Computer Hardware Software," *IEEE Spectrum*, January 1974, pp. 39—43.

[10] Garrett, L. S., "Integrated-Circuit Digital Logic Families," Parts I, II, III, *IEEE Spectrum*, October, November, December 1970.

[11] Hughes Aircraft Co., *Bipolar LSI* 8-bit Multiplier H1002MC," Newport Beach, Calif. 1974.

[12] Hwang, K., "A TTL Programmable Logic Array and its Applications," *Proc. of the IEEE*, March 1976, pp. 368—369.

[13] Intel Corp. Staff, *Microcomputer Systems Data Book*, Intel Corp., Santa Clara, Calif., 1976.

[14] Intel Corp. Staff, *Intel Data Catalog*, Santa Clara, Calif., 1976.

[15] Juliussen, J. E., "Magnetic Bubble Systems Approach Practical Use," *Computer Design*, October 1976, pp. 81—91.

[16] Kohonen, T., *Digital Circuits and Devices*, Prentice-Hall, Englewood Cliffs, N. J. 1972.

[17] Liu, T. K. et al., "Optimal One-Bit Full Adders with Different Types of Gates," *IEEE Trans. Comp.*, Vol. C-23, No. 1, January 1974, pp. 63—70.

[18] Luecke, G., et al., *Semiconductor Memory Design of Applications*, McGraw-Hill, New York, 1973.

[19] Melen, R. and Buss, D., "Charge-Coupled Devices: Technology and Applications," *IEEE Press*, New York, 1977.

[20] Morris, R. L. et al. (eds.), *Designing with TTL Integrated Circuits*, McGraw-Hill, New York, 1971.

[21] Motorola Staff, *Semiconductor Data Library*. Motorola Semiconductor Products, Inc., Phoenix, Arizona 1976.

[22] Mowle, F. J., *A Systematic Approach to Digital Logic Design*, Addison-Wesley, Reading, Mass., 1976.

[23] National Semiconductor, *Digital Integrated Circuits*, National Semiconductor Corp., Santa Clara, Calif., 1974.

[24] National Semiconductor, *CMOS Integrated Circuits*, National Semiconductor Corp., Santa Clara, Calif., 1975.

[25] Priel, V. and Holland, P. "Application of a High-Speed Programmable Logic Array," *Computer Design*, December 1973, pp. 34—96.

[26] RCA Solid-State, *RCA Integrated Circuits*, RCA Corp., Somerville, N. J., 1976.

[27] Signetics Application Book: *Digital, Linear, MOS*, Signetics Corp., Sunnyvale, Calif., 1974.

[28] Signetics Application Notes, *Bipolar Field Programmable Logic Arrays*, Signetics Corp., Sunnyvale, Calif., 1976.

[29] Soucek, B., *Microprocessor and Microcomputers*, Wiley-Interscience, New York, 1976.

[30] Texas Instruments Staff, *TTL Data Book and Supplement*, Texas Instruments, Inc., Dallas, Texas, 1973.

[31] Texas Instruments Engineering Staff, *The Integrated Circuits Catalog for Design Engineers*, Components Group, Texas Instrument, Inc., CC-401, Dallas, Texas, 1973.

[32] Texas Instruments Engineering Staff, *The Semiconductor Memory Data* Book, Texas Instruments, Inc., CC-420, Dallas, Texas, 1975.

[33] Texas Instruments Semiconductor Staff, "Magnetic Bubble Memories and System Interface Circuits," T. I., Inc., Dallas, Texas, 1977.

[34] Torrero, E A., "Solid-State Devices," *IEEE Spectrum*, January 1977, pp. 48—54.

[35] Tung, C. et al., "Bubble Ladder for Information Processing," *IBM Research Report*, # RJ 1556, 1975.

[36] TRW Application Notes, "LSI Multipliers," Space Park, Radondo Beach, Calif., March 1977.

[37] Wakerly, J. F., *Logic Design Projects Using Standard Integrated Circuits*, Wiley, New York, 1976.

[38] Warner, R. W., Jr.. "I-Squared L: A Happy Merger," *IEEE Spectrum*, May 1976, pp. 42—47.

习　　题

题 2.1　试用一个异或门和两个 4 输入端的多路转换器实现一个 1 位的可控加/减单元（CAS），一端作为和的输出，一端作为进位输出．两个操作数位用作选择信号．这里假定真值的和求补的进位输入信号两者均可使用．

题 2.2　试确定如单独用四个 2 输入端，双 4 输入端，双 8 输入端和单 16 输入端多路转换器构成以下树形多路转换器所需的最少集成电路片数．因为每个双 8 输入端和每个单 16 输入端多路转换器要求较大的集成电路封装尺寸，在印刷电路板上占据两倍的面积．在计算它们时应把每一个当作两片来考虑．

(a) 一个64输入端多路转换器．

(b) 一个 128 输入端多路转换器．

(c) 一个 1024 输入端多路转换器．

题 2.3　构造一个 16 到 65536 的行译码器（2^{16} 个输出端中选出 1 个的地址译码器），试用最少数目的双 2 到 4，单 3 到 8 以及 4 到 16 行译码器来实现．画出你设计的连接线路图．这里假定电路片具有图 2.4 和图 2.5 所示的片选功能．

题 2.4　设计两个积木化的数码转换器：

(a) 用若干图 2.6 中的标准 6 位二进制到二-十进制转换模块，构成一个 16 位的二进制到二-十进制的转换器．

(b) 用若干图 2.8 中的标准二-十进制到二进制转换模块，构成一个 6 位二-十进制数位的二-十进制到二进制的转换器．

题 2.5　用若干图 2.12 中的 4 位比较器组成一个 48 位的数值比较器．所有不用的引线端应该适当规定．画出你设计的连接线路图．

题 2.6　设计一个 8×8 相加乘法模块，它能计算以下表达式

$$\mathbf{P} = \mathbf{A} \times \mathbf{B} + \mathbf{C} + \mathbf{D}$$

其中 **A**, **B**, **C** 和 **D** 都是 8 位数，而 **P** 是 16 位数．在你的设计中，只允许用全加器和 2 输入端的"与"门．

题 2.7　试用 D 型触发器和 16 输入端多路转换器设计一个 8 位 16 种功能的寄存器．功能表列在下面．画出你设计的线路图．

功能控制 WXYZ	寄存器功能	功能控制 WXYZ	寄存器功能
0 0 0 0	无操作	1 0 0 0	并行输入 **A**
0 0 0 1	清零	1 0 0 1	并行输入 **B**
0 0 1 0	置成 1	1 0 1 0	Q "或" **A** (按位)
0 0 1 1	求 1 的补数	1 0 1 1	求 2 的补数
0 1 0 0	向上计数	1 1 0 0	Q "异或" **A** (按位)
0 1 0 1	向下计数	1 1 0 1	Q "与" **A** (按位)
0 1 1 0	右移	1 1 1 0	Q "异或" **B** (按位)
0 1 1 1	左移	1 1 1 1	Q "与" **B** (按位)

注意：**A** 和 **B** 是 8 位外部输入信号，而 **Q** 是当前寄存器的内容．

题 2.8　扼要描述以下半导体存储器技术：

(a) 随机存取存储器(RAM)．

(b) 只读存储器 (ROM).

(c) MOS 与双极型工艺.

(d) 电荷耦合器件 (CCD).

(e) 集成注入逻辑(I^2L).

(f) 磁泡存储器 (MBM).

按存取时间,密度,功耗,非易失性和每位价格,比较 CCD, I^2L 和 MBM 存储器.

题 2.9 试对一个具有 3 个输出端，每个包含对 6 输入变量的、多到 24 个乘积项的字段可编程序逻辑阵列(与图 2.22 相似),来实现一个 3 位的数值比较器.

题 2.10 相对于以下技术考虑,比较只读存储器 (ROM) 和可编程序逻辑阵列 (PLA).

(a) 制造工艺.

(b) 位容量.

(c) 可编程序的能力.

(d) 算术运算设计应用.

题 2.11 设计以下带符号补码的数值比较器:

(a) 一个 4 位的对 1 求补比较器.

(b) 一个 4 位的对 2 求补比较器.

每个比较器应有三条输出线,指出结果“>”,“=”和“<”. 注意,每个输入操作数的最前面一位留作符号.

第三章　快速的双操作数加法器/减法器

3.1 引　　言

高性能加法器不仅对于加法，而且对于减法，乘法和除法都是很必要的．一个数字算术运算处理器的速度，在很大程度上取决于用在系统中的加法器的速度．本章将从各种定点二进制加法和减法的算术运算法则的规定开始．然后我们再研究在设计快速的双操作数加法器时各种加速进位/借位的技术．首先讨论非同步的进位完成加法器（carry-completion adders）．然后讨论三种同步加法器，即条件和（conditional-sum），进位选择（carry-select）以及先行进位（carry-lookahead）加法器．对于各种双操作数加法器的评价也作了叙述．

在五十年代和六十年代发表了大量有关快速加法器设计的文献．这里我们只介绍主要的双操作数加法器类型，但这些类型还有其它的变形．我们将着重说明与每一种加法器有关的设计方法．采用今天的大规模集成电路（LSI）工艺来构成一个加法器，对于设计者来说已不是一个很大的负担了．然而，深入了解加速进位的技术，一般会有助于算术运算的设计．例如，使用快速多操作数加法器，可以大大改进乘法速度．加法器应用的例子将在后几章中讨论．在大多数数字计算机中，要求的加法器不止一个．例如，一个浮点处理器至少需要两个加法器，一个处理尾数，一个处理阶．在存储器相对编址中，需要用一个专用的快速加法器来计算存储器有效地址．本章仅讨论常用的双操作数加法器．非常用的多操作数加法器，带符号数字加法器以及运算逻辑部件，将在下一章中讨论．

3.2 基本的带符号补数和带符号数值的加法器

以下给出在三种定点带符号数的系统中，加法或减法算术运算的算法．与这些加法算法相应的行波进位线路的具体实现是用二进制基数来阐明的．这里只考虑定点（FXP）整数的加法或减法就足够了．

令基数为 r 的正整数用正体字母 $\mathbf{A} = a_{n-1}\cdots a_1a_0$ 来表示，其中符号 $a_{n-1} = 0$．$\mathbf{A}$ 的基数反码用 $\bar{\mathbf{A}} = \bar{a}_{n-1}\cdots\bar{a}_1\bar{a}_0$ 来表示，其中对所有 i 的 $\bar{a}_i = r - 1 - a_i$ 以及 $\bar{\mathbf{A}} = (r^n - 1) - \mathbf{A}$．$\mathbf{A}$ 的基数补码用 $\bar{\mathbf{A}} + 1$ 表示，它等于数 $r^n - \mathbf{A}$．我们将用草体字母 $\mathscr{A} = a_{n-1}\cdots a_1a_0$ 表示一个选定求补形式的带符号整数，不论其是正或是负．

基数补码的算术运算

令 $\mathscr{A} = a_{n-1}\cdots a_1a_0$ 和 $\mathscr{B} = b_{n-1}\cdots b_1b_0$ 是两个基数补码整数．$\mathscr{A}$ 和 $\mathscr{B}$ 的加法按以下步骤进行：

第一步　一位一位直接相加．从最低的一对 (a_0, b_0) 开始到最左面的一对符号位为止，假定从右端进入的是一个零初始进位．

第二步　把进人符号位置的进位与从符号位置出去的进位相比较．假如以上两个进位是相同的（都是"1"或都是"0"），则结果是正确的基数补码形式，并把从符号位出去的进位忽略不计．当这两个进位不同时，表示有加法溢出，因而结果无效．

执行两个基数补码数的减法，首先是求出减数的基数补码，然后再按以上加法步骤，把它加到被减数上去．基数补码加法/减法的算法可以按以下三种情况来证明．

情况 1　$\mathscr{A}$ 和 $\mathscr{B}$ 两者都是正数，其中 $\mathscr{A} = \mathbf{A} = a_{n-1}\cdots a_1a_0$ 和 $\mathscr{B} = \mathbf{B} = b_{n-1}\cdots b_1b_0$，以及 $a_{n-1} = b_{n-1} = 0$．于是结果 $\mathscr{S} = \mathscr{A} + \mathscr{B} = \mathbf{A} + \mathbf{B} = s_{n-1}\cdots s_1s_0$ 在这种情况下，符号位置不能产生进位输出．假如 $\mathbf{A} + \mathbf{B} < r^{n-1}$，则所得到的和将是正的（$s_{n-1} = 0$）而且形式是正确的，这表示没有进位进入符号位置．否则（$\mathbf{A} + \mathbf{B} \geqslant r^{n-1}$），将发生加法溢出．

情况 2　$\mathscr{A}$ 和 $\mathscr{B}$ 两者都是负数．令 $\mathscr{A} = r^n - \mathbf{A}$ 和 $\mathscr{B} = r^n - \mathbf{B}$．则

$$\mathscr{S} = \mathscr{A} + \mathscr{B} = (r^n - \mathbf{A}) + (r^n - \mathbf{B}) = 2 \cdot r^n - (\mathbf{A} + \mathbf{B})$$

因为 $a_{n-1} = b_{n-1} = 1$，所以总有一个符号位置的进位输出．如把这个符号位置的进位输出忽略不计，则相当于把 $\mathscr{S}$ 被 r^n 除，并保留其余数，即为

$$\mathscr{S}/r^n = [2r^n - (\mathbf{A} + \mathbf{B})]/r^n = 1 + [r^n - (\mathbf{A} + \mathbf{B})]/r^n$$

于是，如 $\mathbf{A} + \mathbf{B} \leqslant r^{n-1}$，则结果具有正确的形式 $r^n - (\mathbf{A} + \mathbf{B})$．否则（$\mathbf{A} + \mathbf{B} > r^{n-1}$），产生加法溢出．

情况 3　$\mathscr{A}$ 是正的而 $\mathscr{B}$ 是负的（或者 $\mathscr{A}$ 负和 $\mathscr{B}$ 正，其情况是相似的）．在这种情况下，不产生溢出．令 $\mathscr{A} = \mathbf{A}$，$\mathscr{B} = r^n - \mathbf{B}$．于是

$$\mathscr{S} = \mathscr{A} + \mathscr{B} = \mathbf{A} + (r^n - \mathbf{B}) = r^n + (\mathbf{A} - \mathbf{B})$$

很明显，如 $\mathbf{B} > \mathbf{A}$，则得到的和将是负的，而且形式是正确的．如 $\mathbf{B} \leqslant \mathbf{A}$，则结果将为 $\mathscr{S} = r^n + (\mathbf{A} - \mathbf{B})$．我们让 $\mathscr{S}$ 被 r^n 除，$\mathscr{S}/r^n = 1 + (\mathbf{A} - \mathbf{B})/r^n$，忽略符号位置的进位输出，其余数 $\mathbf{A} - \mathbf{B} = |\mathbf{A} - \mathbf{B}|$ 即等于所得到的和．

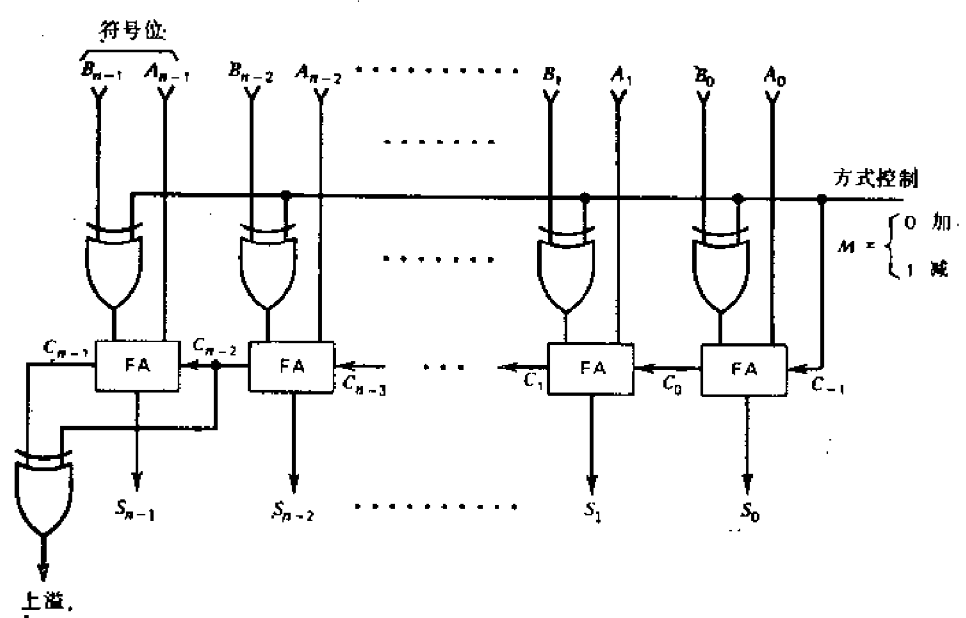
二进制基数补码（对 2 求补）的加法器/减法器的线路设计见图 3.1．图中还附有一些数值例子以验证其操作．最右面的全加器的起始进位输入端被连接到功能方式线 M 上，M 在做加法时等于"0"，而做减法时等于"1"．注意，减数的基数补码是用 $\bar{\mathbf{B}} + 1$ 的方法求得的．

基数减 1 补码的算术运算

令 $\mathscr{A}$ 和 $\mathscr{B}$ 是两个基数减 1 补码整数．其加法/减法的法则和基数补码运算法则是相似的，只有一点是例外，即当不发生溢出时，符号位置的进位输出反回去加到最低位上，以产生基数减 1 表示法的正确的"和"（或者"差"）．这个反馈回去的进位叫做循环进位．对于基数减 1 加法算法的证明留给读者作为一个练习．

图 3.2 给出了在二进制系统（$r = 2$）中基数减 1 加法和减法的数值例子，还给出一个具有外部加/减功能控制及溢出检测的 n 位对 1 求补的行波进位加法器/减法器线路图．注意，循环进位不会在闭合回路中引起任何振荡，因为受循环进位输入触发而产生的任何进位，只会停留在它原来产生的地方，而不会向前传播．无论是反码还是补码，一个 n 位的行波进位加法器均需要以下的时间延迟．

$$\Delta(n\text{位行波进位加法器}) = n \cdot 2\Delta + 6\Delta = (2n + 6)\Delta \qquad (3.1)$$



例如：

$$\begin{array}{rl} & \mathscr{A} = 0\,1\,0\,0\,1 = (+11)_{10} \\ +) & \mathscr{B} = 1\,1\,0\,0\,1 = (-7)_{10} \\ \hline & \mathscr{S} = 1\,\underbrace{0\,0\,1\,0\,0}_{结果} = (+4)_{10} \end{array}$$

忽略（最高位 1）　结果

$\Leftarrow$ 例如：

$$\begin{array}{rl} & \mathscr{A} = 0\,1\,0\,1\,1 = (+11)_{10} \\ -) & \mathscr{B} = 0\,0\,1\,1\,1 = (+7)_{10} \\ \hline \end{array}$$

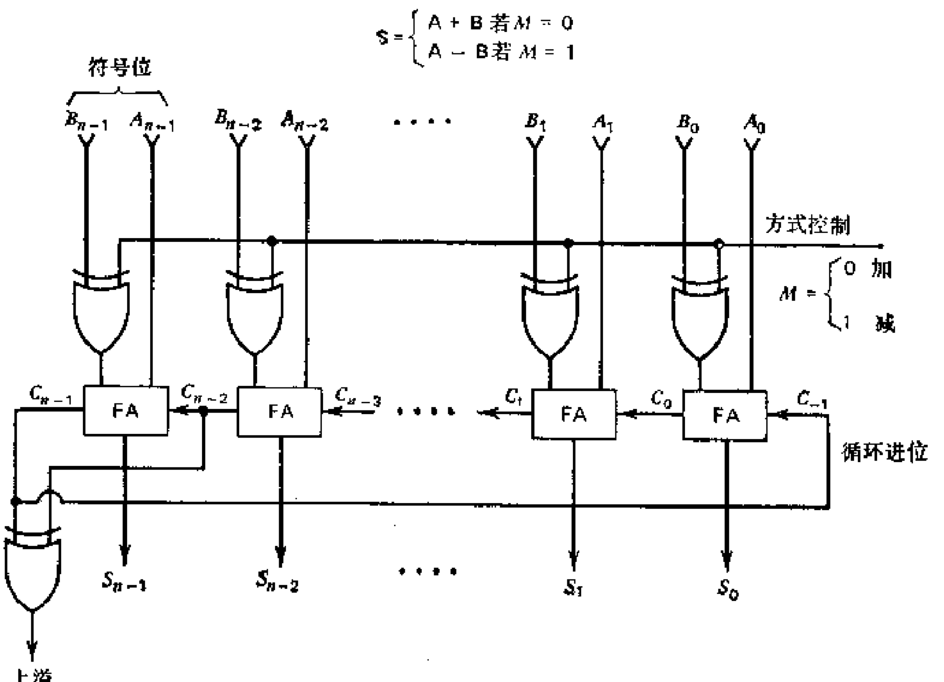
例如：

$$\begin{array}{rl} & \mathscr{A} = 0\,1\,0\,1\,1 = (+11)_{10} \\ +) & \mathscr{B} = 0\,0\,1\,1\,1 = (+7)_{10} \\ \hline & \underbrace{1\,0\,0\,1\,0}_{} \neq (+18)_{10} \end{array}$$

由于上溢使结果无效

$$\mathscr{S} = \begin{cases} \mathscr{A} + \mathscr{B} & 若\ M = 0 \\ \mathscr{A} - \mathscr{B} & 若\ M = 1 \end{cases}$$

图 3.1　补码运算的加法/减法和相应的行波进位加法器的设计



例如：

$$\begin{array}{rl} & \mathscr{A} = 1\,0\,0\,1\,1\,0\,1 = (-50)_{10} \\ +) & \mathscr{B} = 0\,1\,0\,1\,0\,1\,1 = (+43)_{10} \\ \hline & 0\,1\,1\,1\,1\,0\,0\,0 \\ & \quad \longrightarrow 0 \\ \hline & \mathscr{S} = 1\,1\,1\,1\,0\,0\,0 = (-7)_{10} \end{array}$$

例如：

$$\begin{array}{rl} & \mathscr{A} = 1\,1\,0\,0\,1 = (-6)_{10} \\ +) & \mathscr{B} = 1\,0\,1\,0\,1 = (-10)_{10} \\ \hline & 1\,0\,1\,1\,1\,0 \neq (-16)_{10} \end{array}$$

由于上溢使结果无效

$$\begin{array}{rl} & \mathscr{A} = 1\,1\,0\,0\,1 = (-6)_{10} \\ -) & \mathscr{B} = 1\,0\,1\,0\,1 = (-10)_{10} \\ \hline \end{array}$$

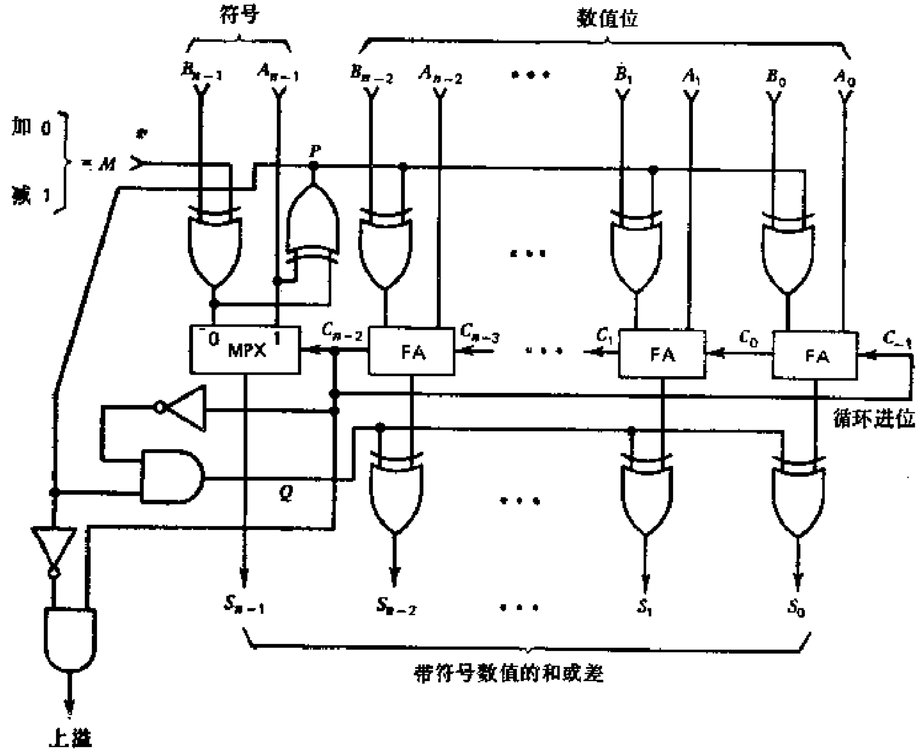
$$\Downarrow$$

$$\begin{array}{rl} & \mathscr{A} = 1\,1\,0\,0\,1 = (-6)_{10} \\ +) & \mathscr{B} = 0\,1\,0\,1\,0 = (+10)_{10} \\ \hline & 1\,0\,0\,0\,1\,1 \\ & \quad \longrightarrow 1 \\ \hline & \mathscr{S} = 0\,0\,1\,0\,0 = (+4)_{10} \end{array}$$

图 3.2　反码运算的加法/减法和相应的行波进位加法器/减法器

带符号数值的算术运算

当两个操作数是以带符号数值的形式给出时，一般希望设计一个线路能直接把这两个数相加或相减,并产生一个带符号数值的"和"或"差". 在这种情况下,算前求补和算后求补这两种线路都需要. 对这两种求补线路的控制逻辑以及溢出检测逻辑见图 3.3. 注



$$\mathscr{A} = 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0 = (-22)_{10}$$
$$-)\ \mathscr{B} = 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1 = (+77)_{10}$$

$$\Downarrow (P = 0)$$

$$0\ 0\ 1\ 0\ 1\ 1\ 0$$
$$+)\ 1\ 0\ 0\ 1\ 1\ 0\ 1$$
$$0\ 1\ 1\ 0\ 0\ 0\ 1\ 1$$
$$\longrightarrow 0 = C_{n-2}(\text{循环进位 } c_r)$$
$$1\ 1\ 0\ 0\ 0\ 1\ 1$$

$$M \oplus B_{n-1} = 1 \oplus 0 = S_n \qquad \Downarrow (Q = 0)$$
$$\mathscr{S} = 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1 = (-99)_{10}$$

图 3.3 带符号数值的行波进位加法器/减法器

意这里也需要循环进位,因为在这个系统中设置的也是基数减 1 补数操作.

表 3.1 对这三种基本的二进制行波进位加法器/减法器线路的主要硬部件和操作速度作了总结. 可以得出结论: 与带符号补数的设计相比较,带符号数值加法/减法需要的硬件最多,可是速度较慢. 无论采用反码或补码的算术运算，其加法/减法都需要相同数量的硬件和几乎相等的速度. 然而，补码加法器/减法器由于无需反馈循环进位，因而被认为比较简单. 下面我们将只讨论带符号补码的设计，在这个条件下减法可以通过补码相加的办法来实现.

表 3.1 三种定点行波进位加法器/减法器线路的硬件部件数和执行时间

n 位加法器/减法器的类型	硬件部件			执行时间（用△表示）
	全加器	异或门	其他部件	
反码	n	$n+1$	无	$(2n+6)$△
补码	n	$n+1$	无	$(2n+6)$△
带符号数值	$n-1$	$2n$	2 输入多路转换器 2 个"非"门 2 个"与"门	$(2n+13)$△

3.3 非同步自定时的加法

以上介绍的串行行波进位加法器是很慢的，因为每一位的位置必须等待一个行波进位传播通过其较低的位置。根据最坏情况的进位传播，行波进位设计所需的总时间延迟是正比于加法器的长度 n 的。以下描述的是一个较好的设计方法，它是通过检测加法器中实际最长的进位传播来大大节省时间的，就平均意义上说，可以比最坏情况传播的总延迟时间短得多。

当两个各为 n 位的二进制数相加时，有两种进位对最后的"和"是起作用的。如图 3.4 的数值例子所表明的，所有各列可以独立地产生进位，形成首次的进位向量 **C**。向量 **C** 中的各个"1"是这么决定的，即检查一下它前面一列的两个被加的数位是否都是 1。这样，向量 **C** 中所有各位可以并行地产生。而由前面进位来的传播进位所触发产生的进位，则形成图中所示的二次进位向量 **D**。依靠检查被向量 **C** 中首次进位所触发的进位，即可产生向量 **D** 中的非零数位。最后的"和" **S** 是用这四个向量 **A**，**B**，**C** 和 **D** 按位相加来求得的。

让我们把向量 **C** 中的四个非零进位表示为 a，b，c 和 d。在向量 **D** 下面指左的箭头指明了该进位传播的长度。而此例中最长的进位长度是 5，而不是 16 位加法器中的最坏情况 $n=15$。令 $E_n(p)$ 是任意选择的两个各为 n 位的二进制数相加时，所产生的进位传播平均最长的长度。在 1946 年，Burks, Goldstine 和 Von Neumann 通过对所有随机二

$$\mathbf{S} = \mathbf{A} \oplus \mathbf{B} \oplus \mathbf{C} \oplus \mathbf{D}\text{（按位模 2 加法）}$$
$$= \mathbf{A} + \mathbf{B}\text{（算术加法）}$$

位的位置	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
A =	0	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1	被加数
+) B =	0	1	0	0	1	1	1	0	0	0	0	1	0	1	0	1	加数
				ⓓ		ⓒ							ⓑ		ⓐ		
C =	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	第一次进位
D =	1	1	1	0	1	0	0	0	1	1	1	1	0	0	0	0	第二次进位
⊕) S =	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	和

进位传播长度：←ⓒ 2　←ⓑ 5（最长）　←ⓐ 1　←ⓓ 4

注：在这数值例子中没有出现最坏情况的进位传播长度 $n=16$

图 3.4 在做加法时首次与二次进位的产生及最长的进位传播长度

进制数的统计分析，证明这个平均最长进位长度有一个上限为

$$\mathbf{E}_n(p) \leqslant \log_2 n \tag{3.2}$$

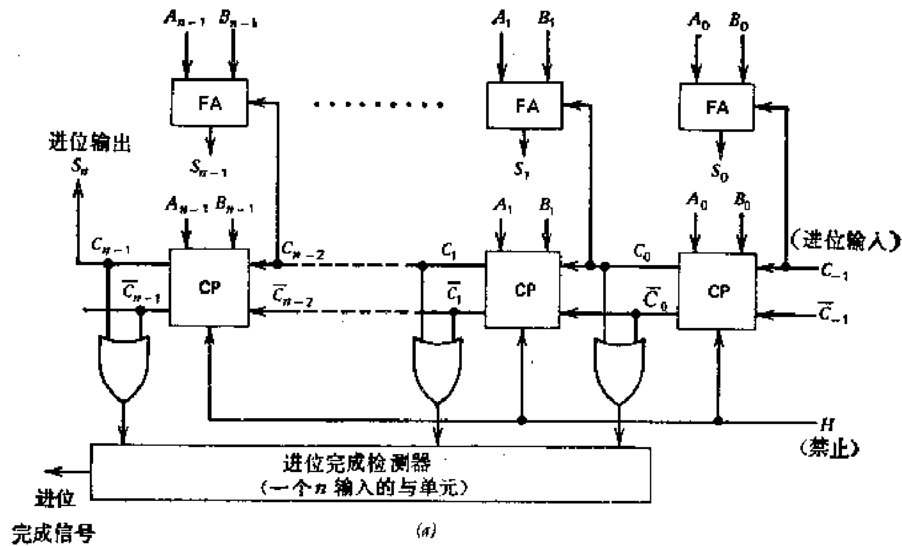
这样给出的平均最长进位长度，对于 32 位和的情况不大于 5，而对于 64 位和的情况则不大于 6。

最坏情况的最长进位对这个平均最长进位长度的比值为

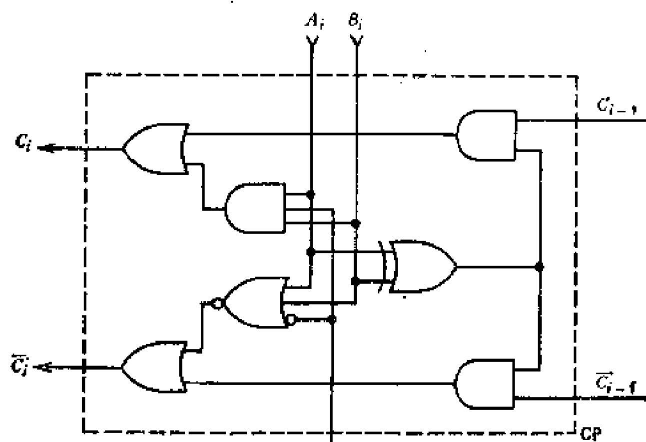
$$\theta = n/\log_2 n \tag{3.3}$$

这个比值在 n 很大时是很可观的。这意思是说，我们可以做一个加法器，它在平均意义上比行波进位加法器相加时要快 $n/\log_2 n$ 倍。一个加法器的工作速度取决于两个给定二进制数的最长的进位传播长度。举例说，为完成图 3.4 中所示的两个 16 位数的相加，只需要 4 级的进位传播延迟 ($4 \times 2\Delta = 8\Delta$)，这大约比一个行波进位加法器快 4 倍。不同的一对二进制数会形成不同的最坏情况时间延迟，但其平均时间趋向于 $\log_2 n$。

我们认为这种加法器是自己定时的 (Self-timing) 和非同步的。自己定时指的是，这



(a)



FA：全加器

CP：进位传播单元

(b)

图 3.5 一个非同步自定时的进位完成检测加法器.

(a) 进位完成检测加法器. (b) 进位传播 (CP) 单元的内部逻辑

个加法器与解的题目是有关的，而且完成操作需要不同的加法时间．非同步表示可以采用没有时钟节拍的控制逻辑，来缩短两个相邻算术运算操作之间的非必要的等待时间间隙．而通常的同步加法器则是由最坏情况时间延迟来决定工作速率的，这将在平均意义上降低整个操作速度．非同步加法器的主要缺点是控制逻辑大为复杂．

3.4 进位完成检测加法器

图 3.5 表示一个逻辑线路，它为以上定义的向量 **C** 中的所有首次进位提供并行的传播．当具有最长传播道路的首次进位到达向量 **D** 中它的指定目的位置时，产生一个进位完成信号，指出进位传播已经结束．然后这个加法在所有 n 个独立的全加器中进行模数为 2 的相加后，即可结束．因而只有 2Δ 的时间延迟．

图3.5中所示的进位传播(CP) 单元，可按以下布尔方程来描述，对于 $i=n-1,\cdots 2,1$

$$C_i = C_{i-1}\cdot(A_i\oplus B_i) + A_i\cdot B_i\cdot H \tag{3.4}$$

$$\bar{C}_i = \bar{C}_{i-1}\cdot(A_i\oplus B_i) + \bar{A}_i\cdot\bar{B}_i\cdot H \tag{3.5}$$

对于 $i=0$ 的情况

$$C_0 = (H\cdot C_{-1})\cdot(A_0\oplus B_0) + A_0\cdot B_0\cdot H \tag{3.6}$$

$$\bar{C}_0 = (H\cdot\bar{C}_{-1})\cdot(A_0\oplus B_0) + \bar{A}_0\cdot\bar{B}_0\cdot H \tag{3.7}$$

其中 C_{-1} 是可选择的进人最低位的初始进位．图中的禁止线H是用来控制加法开始的操作．如置 $H=0$，则两条进位链都被置成"低电位"．而进位序列(在 C_i 和 $\bar{C}_i$ 的两条链上)在解除所有各级的禁止 $(H=1)$ 以后才能开始，其中包括选择的初始进位．

假如第 i 级的 $A_iB_i=11$，则首次进位就从这第 i 级开始，它将经过那些满足 $A_iB_i=$ 01 或 10 的第 i 级向左传播．图中附加一个 n 输入端的"与"门，它在进位传播完毕时给出信号表明在所有被激励的各级上是否有进位存在(1 或 0)．然而，当加法还没有完时，进位完成信号不应该表示(即使是短时地)出进位完成．假如在加法已经结束后有一个输入信号发生变化，则进位完成信号应该立即去掉并保持为零，直到下一个新的"和"完成为止．这种进位完成加法器的样机是由 Gilchrist 等[8]做出的．实验结果表明，一个 40 位的进位完成加法器与一个 40 位行波进位加法器的速度相比，可以少化 8 倍的时间．

3.5 条件和加法法则

这里将描述两种相似的同步加法器．这些加法器使用产生间隔进位的方法，克服了进位传播的问题，并用这些进位在不同进位输入条件下从两个同时产生的暂定的"和"中选择真正的"和"输出．用低价格的，扇人和扇出系数较小的门电路实现经济的算术运算系统时，条件和加法器显得特别有吸引力．而进位选择加法器比起行波进位加法器，速度有很大提高，但硬件也要适当增加一些．

以下，加数、被加数和真正的"和"分别用 A，B 和 S 表示．进位用 C 表示．下标用来标明数位的位置，而上标"1"和"0"是指有进位，还是没有进位输入到某个区段的最低位的位置．没有上标时，表示是一个真正的"和"或一个真正的进位．表 3.2 描述了"条件和"加法的概念．$S^0(k)$ 和 $S^1(k)$ 表示两个暂定的"和"，每个包含多个区段，而每个区段有 k 列加数/被加数．对于一个 n 位数的加法，在 $S^0(k)$ 或 $S^1(k)$ 中有 $\lceil n/k\rceil$ 个区段．在所有

表 3.2 条件和加法表

	i	6	5	4	3	2	1	0	假设进入每一区段的进位	步
$A=(109)_{10}$	A_i	1	1	0	1	1	0	1		
$B=(54)_{10}$	B_i	0	1	1	0	1	1	0		
	$S_i^0(1)$	1	0	1	1	0	1	1	0	1
	$C_{i+1}^0(1)$	0	1	0	0	1	0	0		
	$S_i^1(1)$	0	1	0	0	1	0	//////	0	
	$C_{i+1}^1(1)$	1	1	1	1	1	1	//////		
	$S_i^0(2)$	1	0	1	0	0	1	1	0	2
	$C_{i+1}^0(2)$	0	1		1		0			
	$S_i^1(2)$	0	1	0	0	1	//////	//////	1	
	$C_{i+1}^1(2)$	1	1		1		//////	//////		
	$S_i^0(4)$	0	0	1	0	0	1	1	0	3
	$C_{i+1}^0(4)$	1			1					
	$S_i^1(4)$	0	1	0	//////	//////	//////	//////	1	
	$C_{i+1}^1(4)$	1			//////	//////	//////	//////		
$S=A+B=(163)_{10}$	S	0	1	0	0	0	1	1		
	C_{out}	1								

注：箭头表示区段之间的实际的进位
到最右边的区段的初始进位永远假设为零

区段上同时独立地执行加法．令 $C^0(k)$ 和 $C^1(k)$ 分别是从 $S^0(k)$ 和 $S^1(k)$ 的所有区段中输出的进位所形成的暂定进位序列．

加法过程在 t 步内完成，其中整数

$$t=\lceil \log_2 n \rceil \tag{3.8}$$

在第 i 步，$S^0(k)$ 和 $S^1(k)$ 被形成，其区段大小为

$$k=2^{i-1} \tag{3.9}$$

被加数各列组成分开的区段，从最低位的右端到左端．当 n 并不是 2 的整数幂时，最左的区段会小于 k 列．逐次的区段进位输出被用来选择真正的和输出．

在表 3.2 中我们有一例子 $n=7$．于是，共需要 ($t=\lceil \log_2 7 \rceil=3$) 步才能完成加法．第一步，每一区段只有简单一列，而在 $S^0(1)$ 和 $S^1(1)$ 中的七个区段，是用按位的模 2 加法与相应的进位序列 $C^0(1)$ 和 $C^1(1)$ 相加而形成的．在第二步，每个区段包含两列加数/

被加数位，在 $S^0(2)$ 和 $S^1(2)$ 中的四个区段每一个都是由 2 位加法形成的，而进位输出形成 4 位的暂定进位序列 $C^0(2)$ 和 $C^1(2)$。这个过程按与上相似的方式继续下去，不同的是每执行一步，区段的大小要加倍。最后一步得出真正的"和" S 以及真正的进位输出 C，这就是我们所希望的输出。

3.6 条件和加法器的设计

表 3.2 中所有暂定的"和"以及暂定的进位都能并行地产生。多级的两输入端多路转换器逻辑是用来选择真正"和"以及真正进位。图 3.6 表示一个用三级两输入端多路转换器做成的 7 位条件和加法器的线路图。

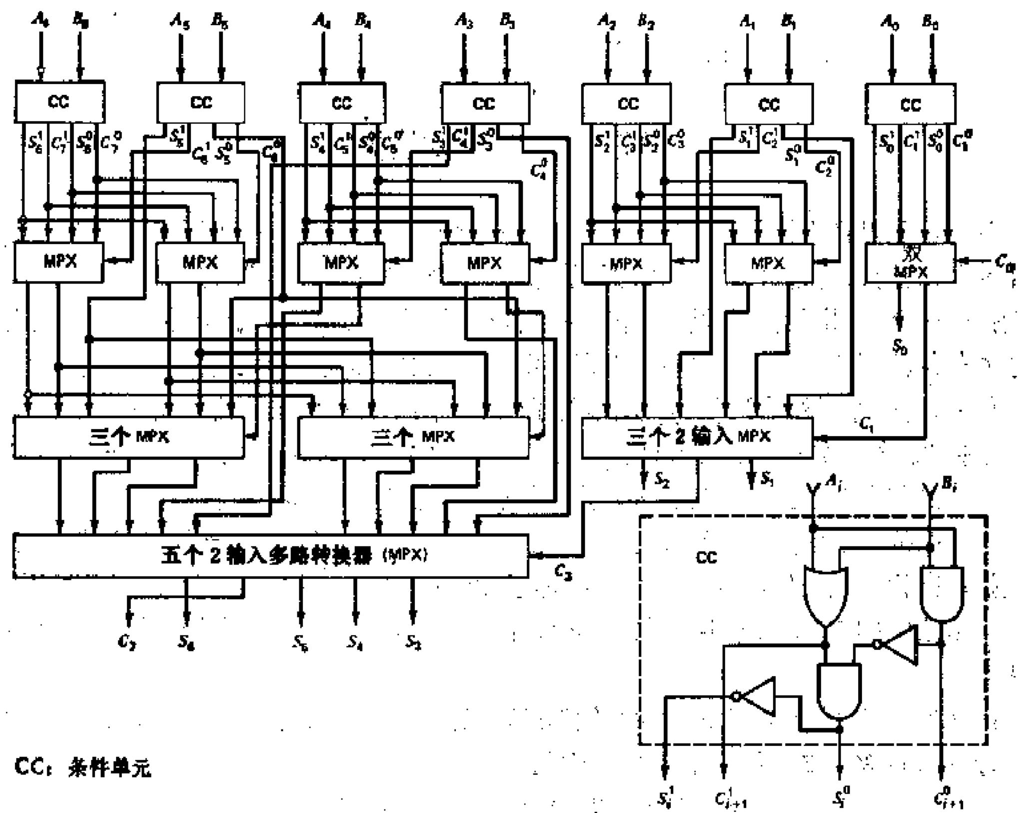


图 3.6　7 位条件和加法器的逻辑线路图。它是用 MSI 多路转换器实现的

每一个数字位置 $i=6,\cdots,1,0$ 中，需要有一个条件单元（CC），以产生如下的暂定"和"及进位

$$S_i^0 = A_i \oplus B_i = A_i\bar{B}_i + \bar{A}_iB_i \tag{3.10}$$

$$C_{i+1}^0 = A_iB_i \tag{3.11}$$

以及

$$S_i^1 = A_i \odot B_i = A_iB_i + \bar{A}_i\bar{B}_i \tag{3.12}$$

$$C_{i+1}^1 = A_i + B_i \tag{3.13}$$

在图3.6的顶部一行上有七个 CC。CC 的详细逻辑在图右下角表示出来。两个这样的 CC 可以做在 14 条引线的集成电路封装片内，如用线或逻辑，只有 2Δ 的时间延迟。在图 3.6 中的第 1，第 2 和第 3 级上分别使用双，三和五个 2 输入端多路转换器，它们相对应

于表 3.2 中的第 1、第 2 和第 3 步．这些多路转换器在第 1 级选择真正“和”位 S_0，在第 2 级选 S_2S_1，在第 3 级选 $S_6S_5S_4S_3$．相应地增加所用的 CC 数目以及多路转换器级数，即可使该线路扩展为字长更长的加法器．

一般，n 位条件和加法器需要

$$\Delta(\text{条件和加法时间}) = (\lceil \log_2 n \rceil + 1) \cdot 2\Delta \tag{3.14}$$

去完成两个 n 位数的加法．每一级的多路转换要构成 2Δ 的延迟．使用目前的 MSI 电路，为了实现一个 n 位的条件和加法器，需要 $\lceil n/2 \rceil$ 个条件单元的电路片和 $\lceil m/4 \rceil$ 个四 2 输入端多路转换器电路片，其中 2 输入端多路转换器的总数 m 可按下式计算

$$m = 2 \cdot n + 3 \cdot \left\lceil \frac{n-1}{2} \right\rceil + 5 \cdot \left\lceil \frac{\left\lceil \frac{n-1}{2} \right\rceil - 1}{2} \right\rceil + \cdots + (\lceil \log_2 n \rceil + 2) \cdot 1 \tag{3.15}$$

对于图 3.6 的设计，共需 11 块电路片．

3.7 进位选择加法器

如果不是从产生按位的暂定“和”与进位开始，我们可以把一个长加法器划分成固定大小的加法器分段，各分段同时执行加法，并用适当的进位输入去选择真正的“和”输出．我们挑选一个 16 位加法器的设计来阐明进位选择加法（见图 3.7）．16 位加法器划分为四个 4 位加法器分段．每个加法器分段有两个加法器，其中一个是有进位进入这个分段的低位的，另一个是没有进位的．图中为了区别这两个分段的加法器线路，一个输入端接 1，一个接 0．每个分段的局部加法器中假定都采用行波进位传播．

选择到每个分段去的进位输入，是由级联的进位选择（CS）部件顺序产生的．当添加更多的高位加法器分段以增长总加法器长度时，CS 线路的复杂性就很快增加．然而，可以扩展高位基数进位（在图 3.7 中基数等于 $2^4=16$）到更高位，以至可能用多级的进位选择．例如，图 3.8 表示使用十六个 4 位加法器，以构成一个具有多级进位分段的 64 位加法器．

图 3.8 中用实线和虚线表示的进位形成路径分别把一个进位“0”或“1”送到这个分段．小组进位 X 和 Y 是按以下方程定义的．

$$X = C_3^0 C_7^1 C_{11}^1 C_{15}^1 + C_7^0 C_{11}^1 C_{15}^1 + C_{11}^0 C_{15}^1 + C_{15}^0 \tag{3.16}$$

$$Y = C_3^1 C_7^1 C_{11}^1 C_{15}^1 \tag{3.17}$$

其中进位 C_i^0 和 C_i^1 是在每个 4 位加法器分段中产生的．标号 a, b, c 和 d 作为 16 位一组的标志．每 16 位一组的进位组合起来，按照下列方程产生进位 C_{15}, C_{31}, C_{41} 和 C_{63}．

$$\begin{aligned}
C_{15} &= C_{-1}Y_a + X_a \\
C_{31} &= C_{-1}Y_aY_b + X_aY_b + X_b \\
C_{47} &= C_{-1}Y_aY_bY_c + X_aY_bY_c + X_bY_c + X_c \\
C_{63} &= C_{-1}Y_aY_bY_cY_d + X_aY_bY_cY_d + X_bY_cY_d + X_cY_d + X_d
\end{aligned} \tag{3.18}$$

以上进位送回到适当的分段组去，在那里它们用来同时选择四个 16 位一组的真正“和”输出．把这个 64 位多级进位选择加法器与行波进位加法器比较，速度可快 15 倍左右，但硬件大约加了一倍．

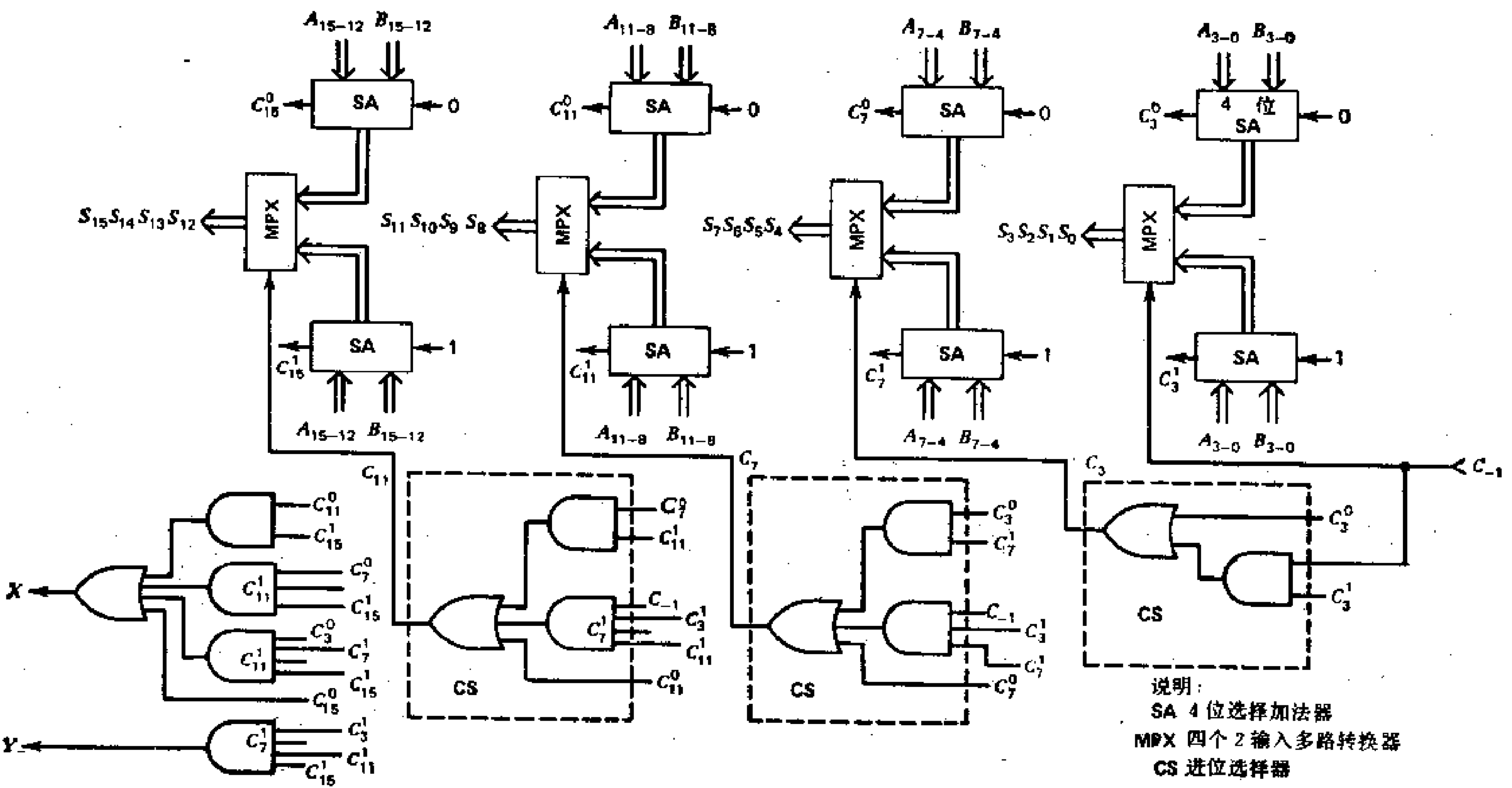


图 3.7 具有成组进位产生功能的 16 位进位选择加法器

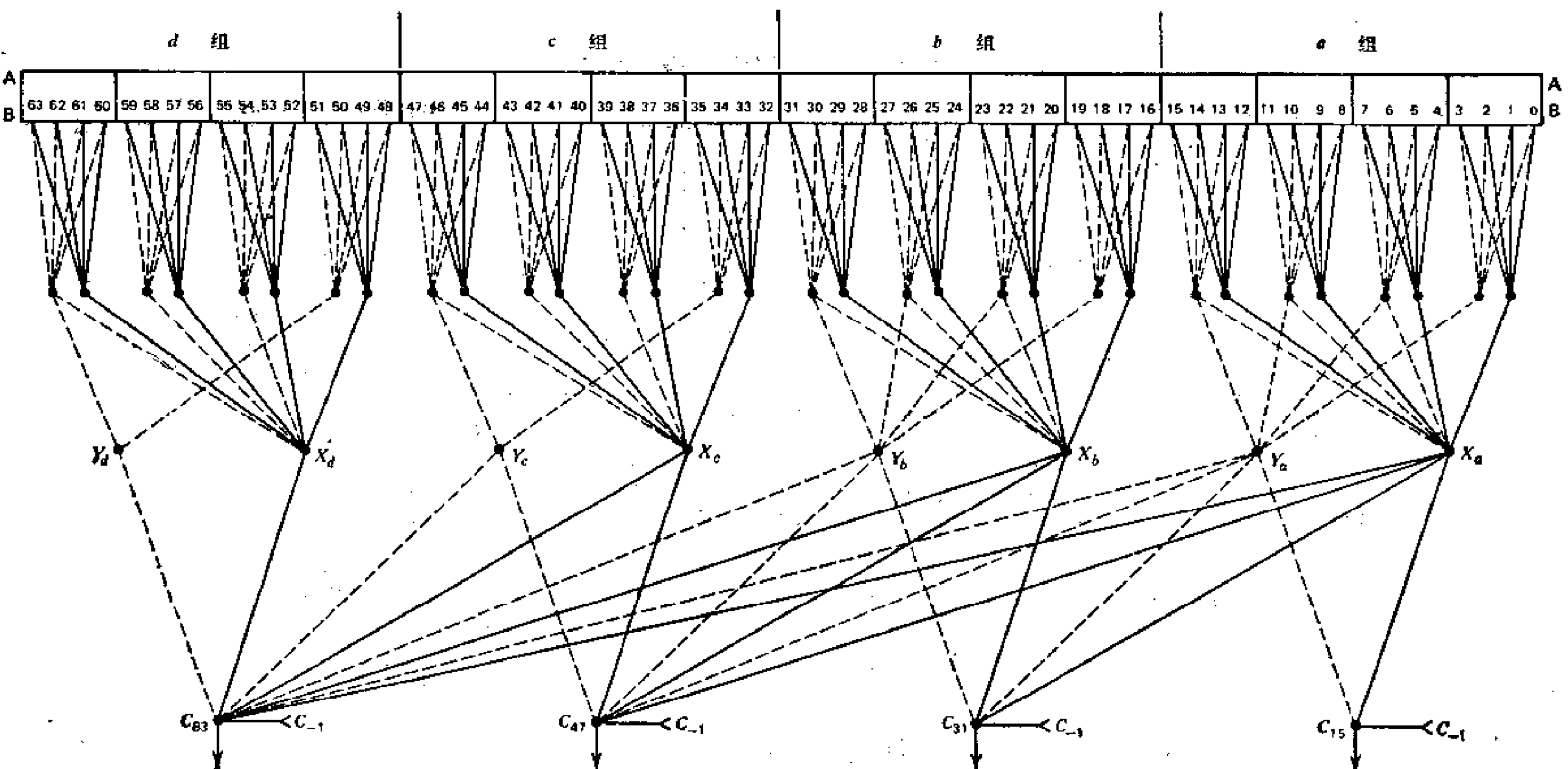


图 3.8 在一个 64 位进位选择加法器中 3 级进位的形成．该加法器具有十六个 4 位分段，分成四个 16 位小组

3.8 进位产生传播及先行功能

下面我们讨论一种先行进位（CLA）技术，它可用来加速在加法器/减法器组合体中进位（或借位）的传播. 进位进入一个"并行"加法器中所有位的位置，是由附加的逻辑电路同时产生的. 这样得到的加法时间是与加法器长度无关的常数.

令 $\mathbf{A}=A_{n-1}\cdots A_1A_0$ 和 $\mathbf{B}=B_{n-1}\cdots B_1B_0$ 是输入到 n 位加法器中的加数和被加数，其中 A_{n-1} 和 B_{n-1} 是符号位. 令 C_{i-1} 是加到第 i 位的进位输入. 加到最低位的进位输入用 C_{-1} 表示. 令 S_i 和 C_i 是第 i 级的"和"以及进位的输出. 我们定义两个辅助函数如下：

$$G_i=A_i\cdot B_i \tag{3.19}$$

$$P_i=A_i\oplus B_i \tag{3.20}$$

其中进位产生函数 G_i 表示这个 i 级发出一个进位的条件. 而 P_i 叫进位传播函数，它在第 i 级可以使进来的进位 C_{i-1} 通过，并进入下一级去的条件下，将为"真"（即逻辑"1"）. 把 P_i 和 G_i 代入方程式 2.2 中，我们得到（当 $i=n-1,\cdots,1,0$）

$$S_i=(A_i\oplus B_i)\oplus C_{i-1}=P_i\oplus C_{i-1} \tag{3.21}$$

$$C_i=A_i\cdot B_i+(A_i\oplus B_i)\cdot C_{i-1}=G_i+P_i\cdot C_{i-1} \tag{3.22}$$

这些方程表明一点，即对 $i=n-1,\cdots,1,0$ 的所有 P_i 和 G_i，都可以根据外部输入 **A** 和 **B** 同时产生，如图 3.9 所示. 方程式 3.21 指的是，只要所有进位输入 $C_{n-2},\cdots,C_1,C_0,C_{-1}$ 是同时提供的，则对 $i=n-1,\cdots,1,0$ 的所有的"和"数位 S_i 可以并行地产生，如图中所示. 我们下面将说明能预先决定这些进位的线路.

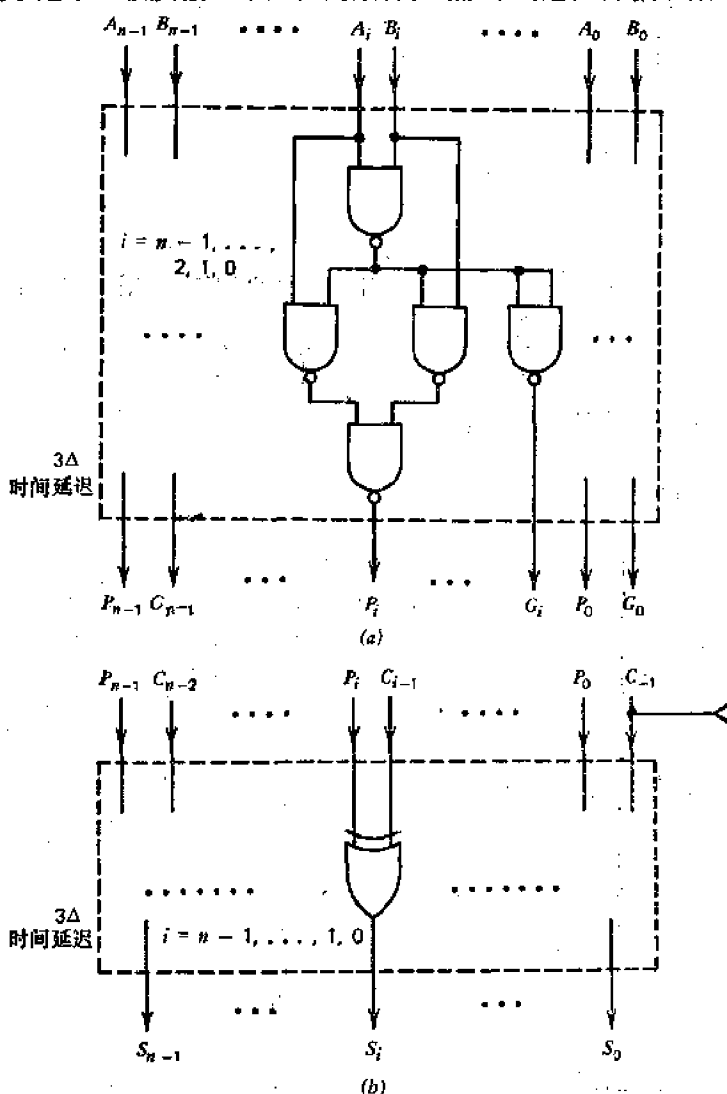


图 3.9 实现进位产生，进位传播和求和功能的逻辑电路.
(a) 进位产生/传播部件；(b) 求和部件

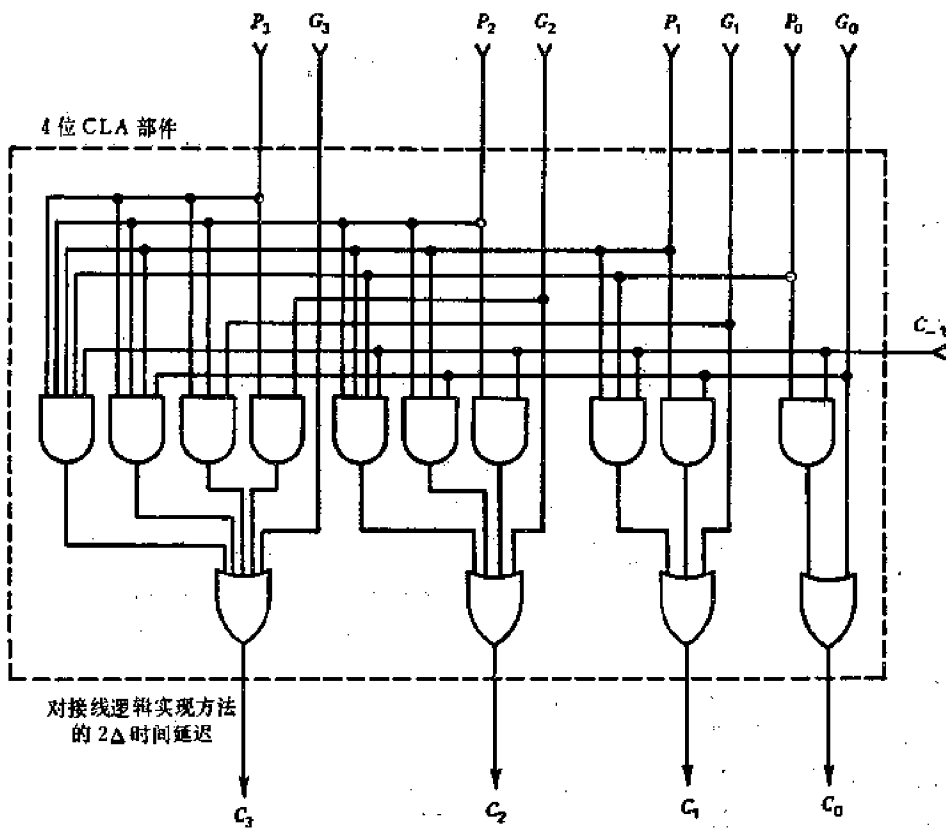


图 3.10　4位先行进位部件的逻辑电路

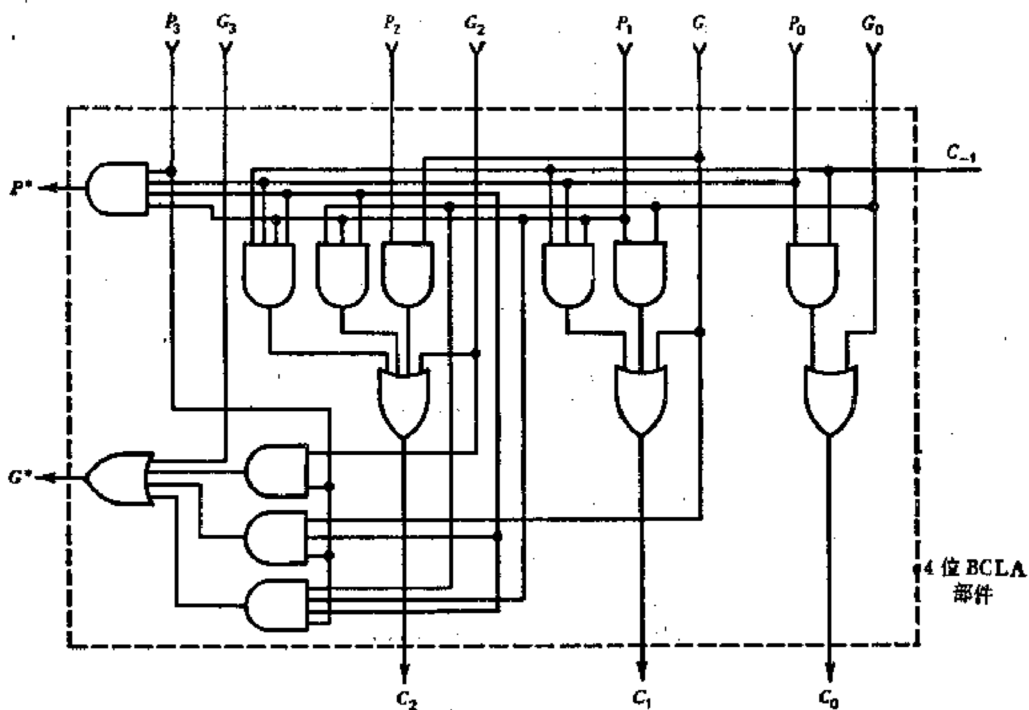


图 3.11　具有成组进位产生/传播功能的4位成组先行进位部件的逻辑电路图

重复使用方程式 3.22 中的递归公式，可得到用变量 P_i 和 G_i 以及初始进位 C_{-1} 表示的以下一组进位方程式。

$$
\begin{aligned}
C_0 &= G_0 + C_{-1}P_0 \\
C_1 &= G_1 + C_0P_0 = G_1 + G_0P_1 + C_{-1}P_0P_1 \\
C_2 &= G_2 + C_1P_2 = G_2 + G_1P_2 + G_0P_1P_2 + C_{-1}P_0P_1P_2 \\
&\vdots \\
C_k &= G_k + G_{k-1} \cdot P_k + G_{k-2}P_{k-1}P_k + \cdots \\
&\quad + G_0P_1P_2\cdots P_k + C_{-1}P_0P_1\cdots P_k \\
&\vdots \\
C_{n-1} &= G_{n-1} + G_{n-2}P_{n-1} + \cdots + C_{-1}P_0P_1\cdots P_{n-1}
\end{aligned} \tag{3.23}
$$

这组方程可以用组合逻辑电路实现，叫做 n 位的先行进位部件． $n=4$ 的电路见图 3.10．再增添两个附加端功能(称做成组进位产生 G^* 以及成组进位传播 P^*)就可形成另一方案，见图 3.11．其中只有三个进位输出端 C_0，C_1 和 C_2，以及另外两个输出端

$$P^* = P_0P_1P_2P_3 \tag{3.24}$$

$$G^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 \tag{3.25}$$

当一个进位进入这个小组（4 位片）后产生一个进位从这个小组输出时，变量 P^* 是“真”(逻辑“1”)．而变量 G^* 则相应于这样的条件，即从某组的最高位置产生出来的进位是来自该小组自身时，将为“真”(逻辑“1”)．于是，小组的进位输出可以写成

$$C_{out} = G^* + P^* \cdot C_{in} \tag{3.26}$$

其中 C_{in} 是进入小组的进位．我们把这种设计方案叫做成组先行进位（BCLA）部件．

目前，用双极型晶体管门做成的单片 4 位或 8 位 CLA 部件和 BCLA 部件已有供应．这种线路的位数 n 所以不能太大，如 $n>10$，主要原因是受到 TTL, ECL 和 I²L 的扇入与扇出的限制（如果坚持用两级（延迟 2Δ）CLA 部件来实现的话）．这种情况将来有可能改变．

3.9 先行进位加法器

把图 3.9 和 3.10 中的三个部件的电路结合起来，即可构成一个先行进位加法器，见图 3.12，其中 $n=8$．由于使用异或门，进位产生-传播部件及形成“和”的部件各需 3Δ 的时间延迟．单级的 CLA 加法器所需的时间延迟是常数．

$$\Delta(\text{单级 CLA 加法时间}) = 3\Delta + 2\Delta + 3\Delta = 8\Delta \tag{3.27}$$

理论上说，如果 CLA 部件可以自由扩充，则可以做成任何字长的完整 CLA 加法器．由于上节中所讨论的限制，目前，单级 CLA 只用来设计字长为 $5 \leqslant n \leqslant 16$ 的并行加法器．字长 $n \leqslant 4$ 的加法器中用 CLA 是不合算的．因为短字长行波进位加法器的速度也差不多快．对于特别长的加法器，如 $n>16$，则需用多级先行进位加法器．

图 3.13 表示一个 32 位的两级 CLA 加法器的设计，其中第一级用了八个 4 位 BCLA 部件，第二级用了一个 8 位 CLA 部件。加到 8 位先行部件的输入，是从八个 4 位小组的小组输出 P_j^* 和 G_j^* 得来的，其中 $j=0, 1, \cdots, 7$．第二级的进位输出连接到第一级中的小组进位输入端．在这样一个两级的设计中，产生所有所需的进位，要花 6Δ 门延迟．总共的时间延迟(它是常数)为

$$\Delta(\text{两级 CLA 加法时间}) = 3\Delta + 6\Delta + 3\Delta = 12\Delta \tag{3.28}$$

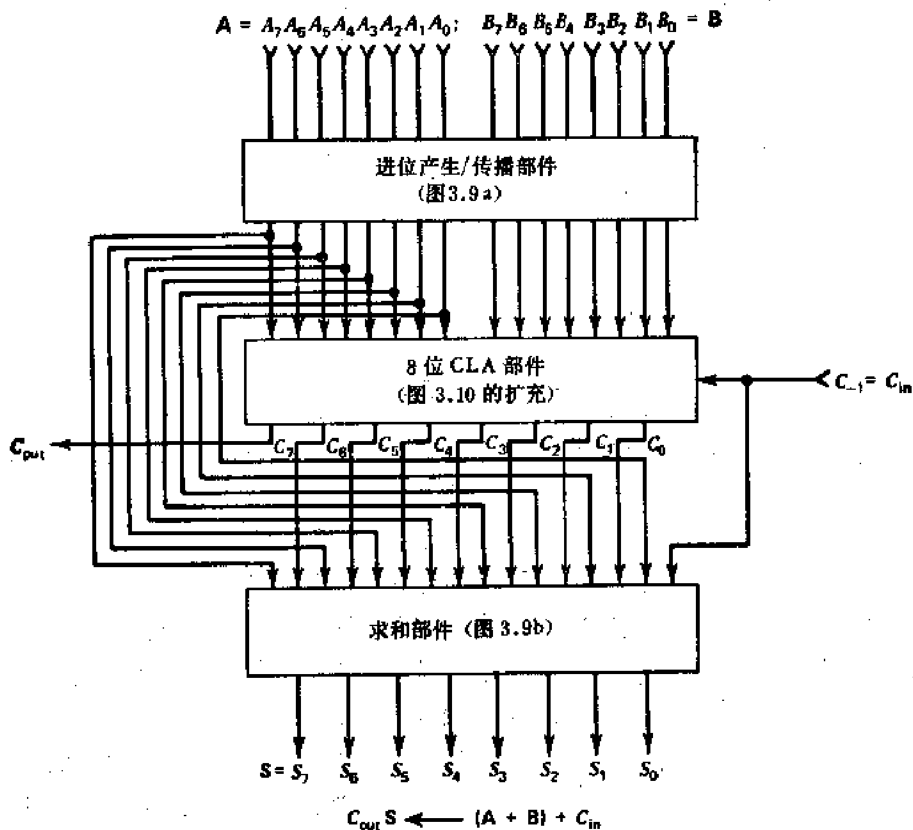


图3.12 全先行进位的加法器(图中是8位 CLA 加法器)的功能方框图

同样的思想可以扩展为设计多于两级的更长的加法器．每添加一级先行进位，则多加一个 4Δ 的时间延迟.在实际情况中,我们用的加法器，很少是多于 64 位的．因此考虑一级或两级的 CLA 加法器已经是足够的了．表 3.3 总结了与不同组合的 n 位先行进位加法器有关的设计和应用参数．符号 **CLA**($m \times k$) 用来表示两级的方式，其中 m 个 k 位 BCLA 部件用在第一级，而一个 m 位的 CLA 部件用在第二级，这样 $m \times k = n$．图 3.13 所示的设计相应于一个并行加法器方案 $32 = 8\times4$．如果考虑到为实行减法所需的

表 3.3 先行进位加法器构成方式、设计和应用参数

CLA 加法器方式	所需 CLA 部件	所需 BCLA部件	工作速度[1]
CLA(8)	CLA(8)	无[2]	8Δ
CLA(16)	CLA(16)	无[2]	8Δ
CLA(4×4)	CLA(4)	四个 BCLA(4)	12Δ
CLA(4×6)	CLA(4)	四个 BCLA(6)	12Δ
CLA(6×4)	CLA(6)	六个 BCLA(4)	12Δ
CLA(8×4)	CLA(8)	八个 BCLA(4)	12Δ
CLA(8×6)	CLA(8)	八个 BCLA(6)	12Δ
CLA(8×8)	CLA(8)	八个 BCLA(8)	12Δ
CLA(16×4)	CLA(16)	十六个 BCLA(4)	12Δ
CLA(16×8)	CLA(16)	十六个 BCLA(8)	12Δ

1) 如果包括求补减法和溢出检测所需的延迟，则还要添加 3Δ 的延迟．

2) 假设是全长的 CLA．

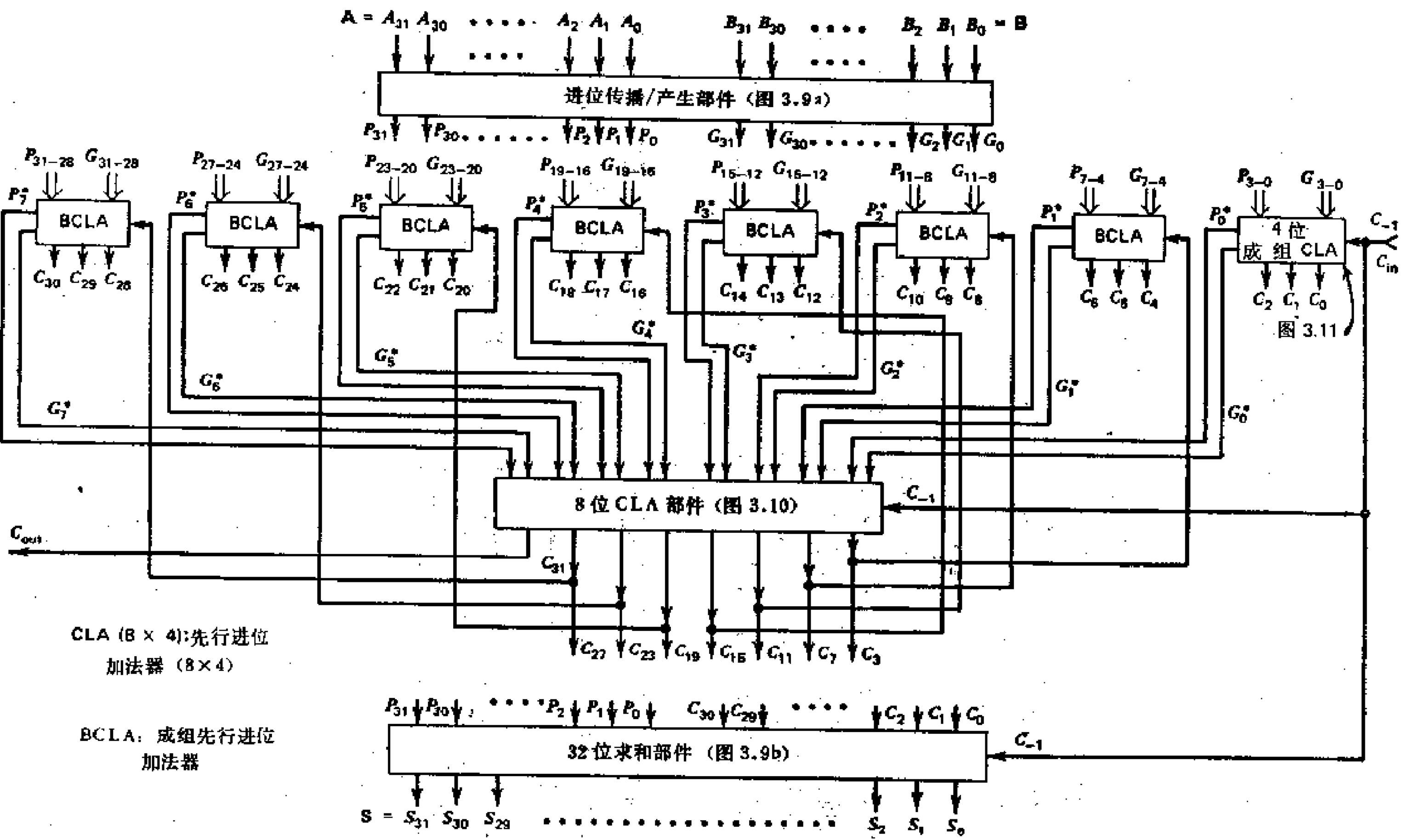


图 3.13　32 位字长的两级(安排成 8×4 的形式）先行进位加法器

算前求补和溢出检测电路，那么表 3.3 中每一项总加法时间还应加上三级附加门的延迟 3Δ.

3.10 各种双操作数加法器的评价

一个加法器的计算效率 η 和两个因素有关，即加法器的投资 **I** 和计算能力 **W**，按以下比值

$$\eta = \frac{\mathbf{W}}{\mathbf{I}} \tag{3.29}$$

投资 **I** 通常是由硬件复杂程度 ϕ 和总加法时间 ρ（它决定加法器工作速度）的乘积反映的. 计算能力 **W** 很容易用加法器的宽度 n 来衡量，它反映了加法器的工作范围. 对于不同工艺器件不同结构的加法器设计，没有衡量硬件复杂性 ϕ 的唯一的或通用的公式. Sklansky[7] 建议用一个加法器结构中的门数目 G 的对数，来定义 ϕ

$$\phi = \log_2 \mathbf{G} \tag{3.30}$$

对数中的基数 2，是因为在 Sklansky 的分析中只用两输入端的逻辑门 AND, OR 或反相器. 虽然以上方程式并不能直接用于现代的多输入端门功能的设计中，但其分析方法有助于研究评价计算机算术运算系统的更通用的方法. 表 3.4 提供我们在以前几节中研究的四种类型加法器的 **G** 和 ρ 的公式. **G** 和 ρ 两者都是在假设只有两输入门的条件下，用加数/被加数的长度 n 来表示的. 以上的讨论可得出下面衡量加法器效率的公式

$$\eta = \frac{n}{\rho \cdot \log_2 \mathbf{G}} \tag{3.31}$$

在图 3.14 中，把效率方程画成对所有四种加法器方案的加法器长度 n 的函数. 平均最长进位长度 ($\log_2 n$) 被用来估计一个进位完成加法器所预期的加法时间. 在先行进位加法器中的进位传播长度假设为 $p = 5$.

表 3.4 评价四种双操作数加法器的门数目和延迟时间的 Sklansky 公式 (Sklansky[17])

加法器类型	所需两输入端门的数目[1]	用 Δ[2] 表示的总延迟时间
(1) 行波进位加法器	$7n$	$2n\Delta$
(2) 进位完成加法器	$17n-1$	$(n+4)\Delta$（下限）
(3) 条件和加法器	$3n[2+\log_2(n+1)]$	$[2+2\log_2(n+1)]\Delta$
(4) 先行进位加法器 CLA $\left(\frac{n}{p}\times p\right)$	$6n+1+\frac{p+1}{p-1}\cdot q+\frac{p^2+2p-1}{p}\cdot\left[k\cdot\left(n+\frac{1}{p-1}\right)-\frac{p\cdot q}{(p-1)^2}\right]$，其中 $k \triangleq \log_p[1+n(p-1)]-1$，$q \triangleq 1+(n-1)p-n$	$[4+k(p+1)]\Delta$ 其中 $k=\log_p[1+n(p-1)]-1$

1) n 是加法器字长；p 是 CLA 加法器的进位跨度(即进位传播长度).
2) 不计求补器或溢出检测器产生的时间延迟. 其中认为只有两输入端的"与"门、"或"门和"非"门影响延迟时间，尤其是在 CLA 情况下.

一个有效的加法器设计应该考虑到器件工艺技术和许多其他系统因素，诸如器件扇

入与扇出能力，溢出的考虑，循环进位，各种求补线路以及与其他运算操作的联系．同步加法器需要更简单的时钟控制和具有固定的加法时间，而非同步加法器则可提供可变的加法时间，但其代价是要化费更复杂的局部定时控制逻辑．在最后决定加法器设计时，还应考虑到诸如设计和制造容易以及出错测检能力等其他因素．

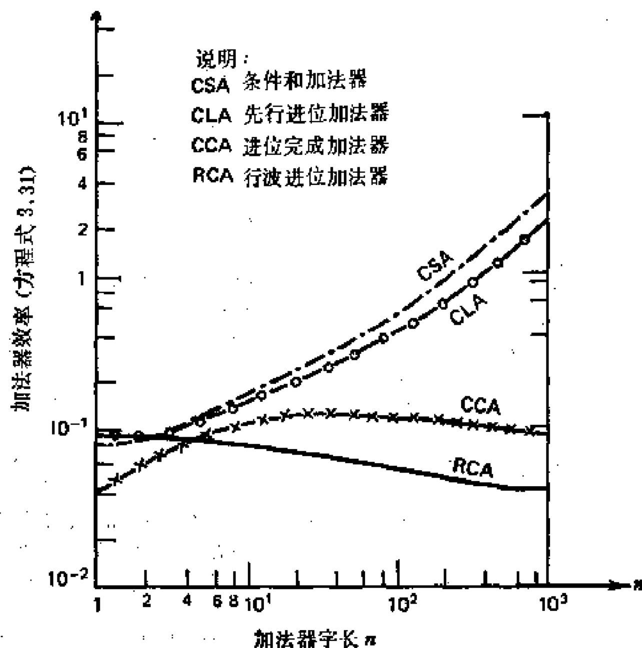


图 3.14 在方程式 3.31 基础上对四种双操作数加法器设计的效率进行比较（Slansky[17]）

3.11 参考文献注释

对机器算术运算的早期研究工作，大部分是为了使加法器设计的速度更快和更为经济．用现代器件具体实现加法器，可查找 Fairchild 公司的应用手册[6]．非同步加法器曾由 Gilchrist 等作了研究．对进位传播长度的研究在 Reitweisner[15] 和Briley[5]中有报道．条件和加法是由 Slansky[16] 提出的，Kruy[9] 做了进一步的研究．进位选择加法器是由 Bedrij[4]介绍的．

在许多其他研究者中间，Aleksander[1], Avizienis[2], Ferrari[7], Lehman[10,11], MacSorley[13], Sklansky 和 Lehman[19] 以及 Weller[21] 曾研究过加速进位传播的技术以及它们的可能实现方案．应该公正地说，先行进位加法器的一般类型，是这些并行进位产生的研究者们共同达到的成果．实际上，Lehman 和 Sklansky[19] 的工作是内容丰富的．对各种不同的双操作数加法器的评价是 Sklansky[17] 做的．

Bartee 和 Chapman 关于通用累加器的论文[3] 对现代数字加法器的设计很有影响．Majerski[14] 分析了加法器设计中的进位跳转分布．最近，Skedletsky[18] 评述了循环进位加法器的设计．在理论方面，Winograd[23] 和 Spira[21] 研究了机器加法所需时间的极限．

参 考 文 献

[1] Aleksander, I., "Array Networks for a Parallel Adder and Its Control," *IEEE Trans. on Electr. Computers*, Vol. EC-16, No. 2, April 1967.

[2] Avizienis, A., "Logic Nets for Carry and Borrow Propagation," *Class Notes*, Dept. of Engineering, University of California, Los Angeles, 1968.

[3] Bartee, T. C., Chapman, D. J., "Design of an Accumulator for a General Purpose Computer," *IEEE Trans.* EC-14, No. 4, August 1965, pp. 570—574.

[4] Bedrij, O. J., "Carry-Select Adders," *IRE Trans.*, EC-11, No. 3, June 1962. pp. 340—346.

[5] Briley, B. E., "Some New Results on Average Worst-Case Carry," *IEEE Trans. Compt.*, Vol. C-22, No. 5, May 1973, pp. 459—463.

[6] Fairchild Semiconductor Staff, *The TTL Applications Handbook*, Mountain View, Calif., August 1973.

[7] Feirari D., "Fast Carry-Propagation Iterative Networks," *IEEE Trans. Compt.*, Vol. C-17, No. 2, February 1968, pp. 132—145.

[8] Gilchrist, B. et al., "Fast Carry Logic for Digital Computers," *IRE Trans.* EC-4, December 1955, pp. 133—136.
[9] Kruy, J. F., "A Fast Conditional Sum Adder Using Carry Bypass Logic," *AFIPS Conf. Proceedings*, Vol. 27, FJCC 1965, pp. 695—703.
[10] Lehman, M. and Burla, N., "Skip Techniques for High-Speed Carry-Propagation in Binary Arithmetic Units," *IRE Trans.* EC-10, No. 4, December 1961, pp. 691—698.
[11] Lehman, M., "A Comparative Study of Propagation Speed-Up Circuits in Binary Arithmetic Units." *Inform. Processing*, 1962, Elsevier-North Holland, Amsterdam, 1963, pp. 671—677.
[12] Ling, H., "High-Speed Binary Parallel Adder," *IEEE Trans. Comput.*, EC-15, No. 5, October 1966, pp. 799—802.
[13] MacSorley, O. L., "High-Speed Arithmetic in Binary Computers," *Proc. IRE*, Vol. 49, No. 1, January 1961, pp. 67—91.
[14] Majerski, S., "On Determination of Optimal Distributions of Carry Skips in Adders," *IEEE Trans.* EC-16, No. 1, February 1967, pp. 45—58.
[15] Reitwiesner, G. W., "The Determination of Carry Propagation Length for Binary Addition," *IRE Trans.* EC-9, No. 1, March 1960, pp. 35—38.
[16] Sklansky, J., "Conditional-Sum Addition Logic," *IRE Trans.* EC-9, No. 2, June 1960, pp. 226—231.
[17] Sklansky, J., "An Evaluation of Several Two-Summand Binary Adders," *IRE Trans.* EC-9, No. 2, June 1960, pp. 213—226.
[18] Skedletsky, J. J., "Comment on the Sequential and Indeterminal Behavior of and End-Around-Carry Adder," *IEEE Tran. Comput.*, Vol. C-26, No. 3, March 1977, pp. 271—272.
[19] Sklansky, J. and Lehman, M., "Ultimate-Speed Adders," *IRE Trans.* EC-12, No. 2, April 1963, pp. 142—148.
[20] Spira, P. M., "Computation Times of Arithmetic and Boolean Functions in (*d*, *r*) Circuits." *IEEE Trans.* Comput. Vol. C-22, No. 6, June 1973, pp. 552—555.
[21] Weller, C. W., "A High-Speed Carry Circuit for Binary Adders." *IEEE Trans. Comput.* Vol. C-18, No. 8, August 1969, pp. 728—732.
[22] Winograd, S., "On the Time Required to Perform Addition," *Journal of ACM*, Vol. 12, No. 2, April 1965, pp. 277—285.

习　题

题 3.1 试证明,无论在反码或补码的加减法中，只要进入符号位置的进位是等于从符号位置出去的进位,则溢出不会产生. 这个证明应包括所有可能的输入组合情况.

题 3.2 试证明平均最长进位传播长度 $E_n(p)$ 上限受 $\log_2 n$ 限制,其中 n 是加法器字长(文献[15]).

题 3.3 画出一个完整的 16 位非同步自定时的、进位完成检测加法器的原理图. 这个检测加法器对普通 16 位行波进位加法器来说,设计速度提高的比值为多少?

题 3.4 用条件单元 (CC) 和 2 输入端多路转换器 (MPX) 设计一个 15 位条件和加法器. 画出逻辑线路图,并估计所用的集成电路片数,假设一个集成电路片中装有 3 个 CC 或两个 2 输入端 MPX.

题 3.5 试画出具有图 3.8 所示的，进位形成树形结构的 64 位进位选择加法器的完整逻辑线路图. 并在速度和所需硬件方面把这种进位选择加法器与一个 64 位,两级，8×8 先行进位加法器作一比较.

题 3.6 假设只有 4 位的成组先行进位部件 BCLA (见图 3.11) 以及 4 位先行进位部件 CLA (图 3.10). 试构成一个三级的，$4\times4\times4$ 的先行进位补码加法器/减法器. 画出与图 3.13 相似的线路连接图. 估计这个三级 CLA($4\times4\times4$) 加法器的总时间延迟，包括求补器和溢出检测逻辑的时间延迟.

题 3.7 一个 (d,r) 电路，是一个 d 值逻辑电路，它是由最大扇入为 r 输入端的逻辑模块组成的(注意,在二进制逻辑中 $d=2$). 每个模块在单位时间内计算一个 r 自变量的 d 值逻辑函数,

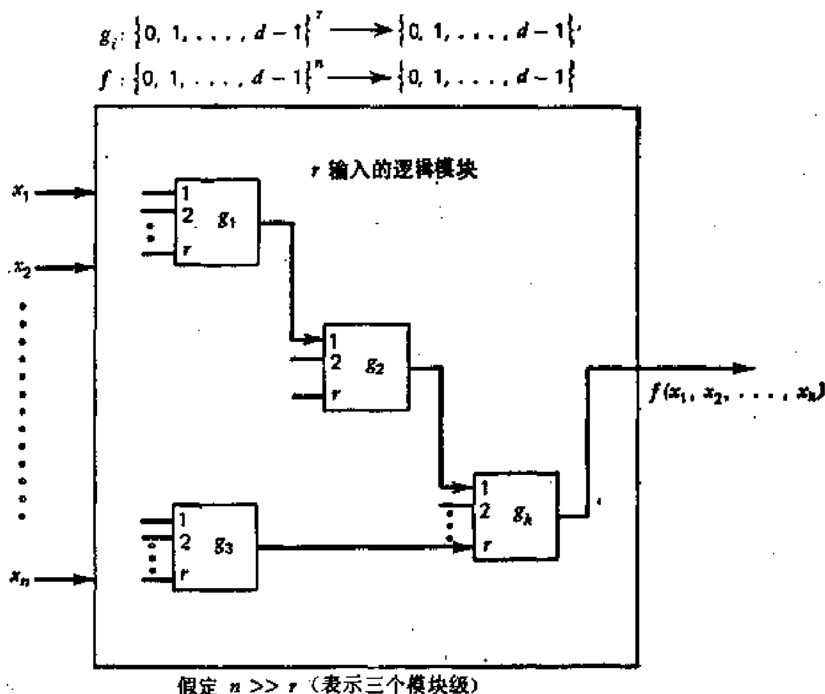
$$g:\{0,1,\cdots,d-1\}^r\to\{0,1,\cdots,d-1\}.$$

一个 (d,r) 电路的总共计算时间,取决于通过 (d,r) 电路 (图 3.15) 的最长通路中的模块级数.

(a) 试证明，任何 n 变量 d 值逻辑函数 $f:\{0, 1, \cdots, d-1\}^n \to \{0, 1, \cdots, d-1\}$ 可以在最少为 $\lceil \log_r n \rceil$ 计算时间内计算出来，你可假定 $n \gg r$(Spira[20])。

(b) 试证明 Ofman 界限：两个 n 位二进制数可以用(3,2)(即 $d=3$，$r=2$)电路在时间 $0(\log n)+1$ 内完成加法。

(c) 试证明 Winograd 界限[21]：两个 n 数位的 d 进制数，可以用 (d, r) 电路在时间 $c_1 + \lceil C_2 \log_r n \rceil$ 内完成加法，其中 $C_1 = 1 + \log_{\lfloor (r+1)/2 \rfloor} \lfloor r/2 \rfloor$ 及 $C_2 = \log_{\lfloor (r+1)/2 \rfloor} r (r \geqslant 3)$。在这个证明中可假设用模数 d^n 的基数补数算术运算。



假定 $n >> r$（表示三个模块级）

图 3.15 每个最多有 r 个输入端的逻辑模块所组成的典型 (d, r) 电路

题 3.8 试证明，在一个多级的长字长为 n 的先行进位加法器中，总加法时间数量级为 $0(2\log_r n)$，其中 r 是所用逻辑门的最大扇入的限制。

题 3.9 试证明直接反码加法/减法的算法适用于输入操作数的所有可能的各种情况。并证明必需使用循环进位。

题 3.10 重复题 3.9，证明图 3.3 中所示的带符号数值的加法/减法的算法。

第四章　多操作数加法器，带符号数字算术运算以及运算逻辑部件

4.1　引　　言

本章从三个方面对快速双操作数加法器的设计方法予以引伸讨论．在准备研究本章所提出的较深的方法以前，必须对前章所讨论的双操作数加法器设计有充分的了解．从普通的两个数的加法线路，可以引伸出以下三个方面：

多操作数加法

通过修改两个操作数的加法器，可以设计出功能强得多的加法器，它能够同时加很多个数，而不是一次只加两个数．这种多操作数的加法，对于以后几章中将要讨论的高速硬件乘法或除法部件的实现，起着重要的作用．我们将在以下两节中讨论进位存储加法器(carry save adders)，然后在第 4.4 和 4.5 节中讨论位片式划分加法器，这两种加法器建议用于处理多操作数加法．

带符号数字加法/减法

在 1.5 节中讨论的冗余带符号数字表示法（SD）被用来设计完全并行的加法器/减法器．在 SD 数表示法中的冗余码，可以实行快速加法/减法的方法，其中和/差的每个数字只取决于操作数的两个相邻数字位置中的数字．换言之，进位传播链被去除了．我们将在第 4.6 节中规定 SD 加法/减法的法则，在第 4.7 节中讨论其物理实现方法．

运算逻辑部件

我们感兴趣的是把一个并行加法器转变为一个功能更强的设备，叫做运算逻辑部件（ALU)．运算逻辑部件在构成实际的运算处理器中起着极其重要的作用．我们的讨论将集中于描述 ALU 的功能特点和系统应用．最后举例研究一个 32 个功能（SN 74181）的 ALU 的设计和应用．

4.2　多操作数进位存储加法器

传统加法器设计是一次加两个数．有些算术运算操作需要将很多的二进位数相加．下面我们提出一种多操作数加法器，它可用少量的硬件元件来执行快速的多个数的加法，乘法和特殊函数的产生．当执行多个数的加法时，有可能要存储进位的传播，直到所有加法被完成，然后拿最后一个工作周期（或很多个最后工作周期）去完成所有加法的进位传播．设计出具有如此进位存储能力的加法器，叫做进位存储加法器（CSA)．

图 4.1 表示一个 n 位的 **CSA**．加法器由 n 个全加器组成．与我们在行波进位加法器中把 n 级级联相接的方法不同，**CSA** 的所有全加器的进位输出都接到一个中间寄存器，叫做进位存储寄存器 (CSR)．n 个全加器的进位输入端形成一个到加法器去的第三个输入向量．这三组输入端，用 A，B 和 C 表示，它们可以互换，而不会影响输出．两个输出向量分别用 **S** 和 **R** 表示它们是“和”与“进位”输出．与前一样，下标 i $(0 \leqslant i \leqslant n-1)$ 指示所有数的数位的位置．上标用来识别输入数．

用 **CSA** 来执行 k 个 n 位数的加法，$\mathbf{N}^1 + \mathbf{N}^2 + \cdots + \mathbf{N}^k$ (模 2^n 的和)，可以用以下步骤描述：

第一步：把前面三个输入数 $\mathbf{N}^1$，$\mathbf{N}^2$ 和 $\mathbf{N}^3$ 输入到 **A**，**B** 和 **C** 输入线．

第二步：把第一步得到的输出 **S** 和 **R**，送回到 **A** 和 **B**，把第 4 个输入数 $\mathbf{N}^4$ 输入到 **C**．在这个操作中，对所有的数位线 $0 \leqslant i \leqslant n-1$，“和”位 S_i 均返回输入线 A_i，而进位位 R_i 则进入输入线 B_{i+1}，只有一个零进位是进入线 B_0，而最左边的进位 R_{n-1} 则忽略不计．

第三步：对余下的 $k-4$ 个输入数，重复第二步，每个加法周期进入一个数，直到所有输入数都进入加法器为止．完成以上几步共需 $k-2$ 个加法周期．

第四步：重复第三步，只不过现在是把零送入 **C** 线，一直到所有各级的 **R** 输出都变成零为止．于是在 **S** 输出端即可得到 k 个数的总和．这最后一步共需 $n-1$ 个加法周期，以完成最后的进位传播．

注意，当需要一个模为 (2^n-1) 的和时，应把最左进位 R_{n-1} 接到最右输入端 B_0，如

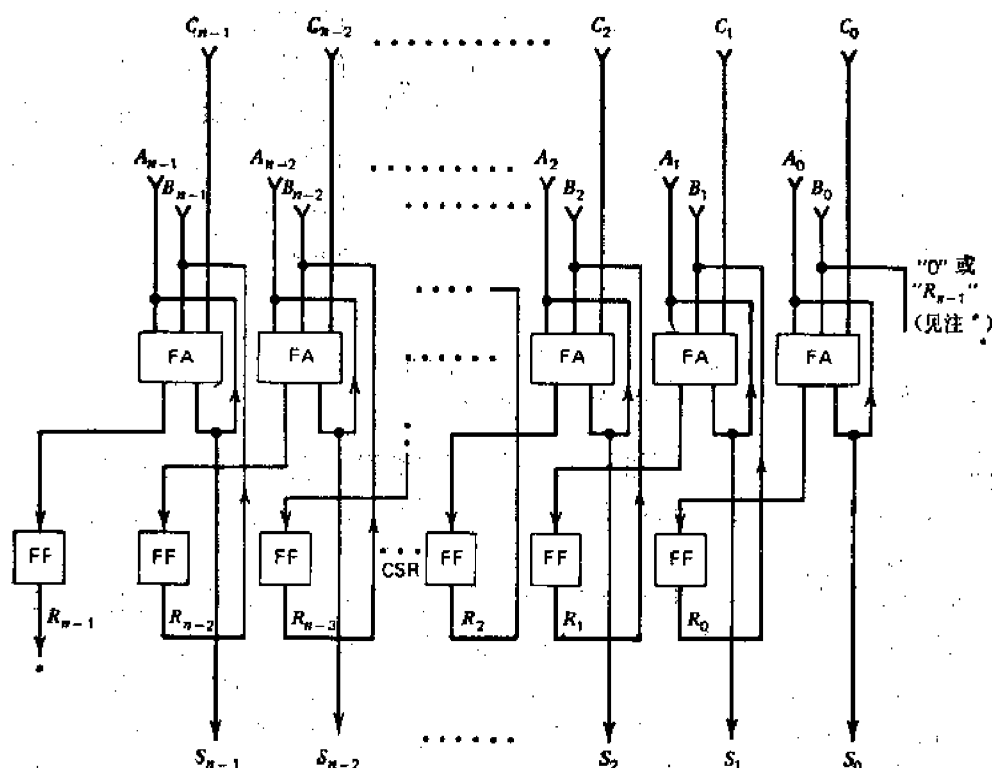


图 4.1 具有进位存储寄存器 (CSR) 的 n 位进位存储加法器 (CSA) 的原理逻辑图．
[*在补码(模数为 2^n 的和)加法时，进位端 R_{n-1} 忽略不计；而在反码(模数为 2^n-1 的和)加法时，R_{n-1} 接到 B_0 输入端．在补码操作中，B_0 接地(“0”)]

同在对 1 求补算术运算中的循环进位一样. 模为 2^n 的加法，就相当于对 2 求补的加法操作. 以上步骤的总的加法时间是 **p** 个工作周期,**p** 为

$$k-1 \leqslant p \leqslant n+k-3 \quad (4.1)$$

基本的 **CSA**，由于其只需较少的硬件，因而是很有吸引力的. 然而，以上步骤的第四步中，最后一级的进位传播导致行波进位的工作方式，它将使整个加法过程成为不可容忍地缓慢，当字长 n 很长时尤其如此. 要改进速度，可以在第四步中采用单独的 2 输入加法器，把 **CSA** 的"和"向量与"进位"向量合并(相加)成一个"最终"的"和". 这个普通的 2 输入加法器，叫做进位传播加法器 (**CPA**). **CSA** 和一个 **CPA** 的连接见图 4.2. **CPA** 在过程的结束时提供 k 个数的最终的"和". 使用这种组合的 **CSA/CPA** 加法部件，只需 $k-1$ 个加法周期，就可完成 k 个输入数的多次加法. **CPA** 可以采用第 3 章中讨论的 2 输入加法器的任何一种方案.

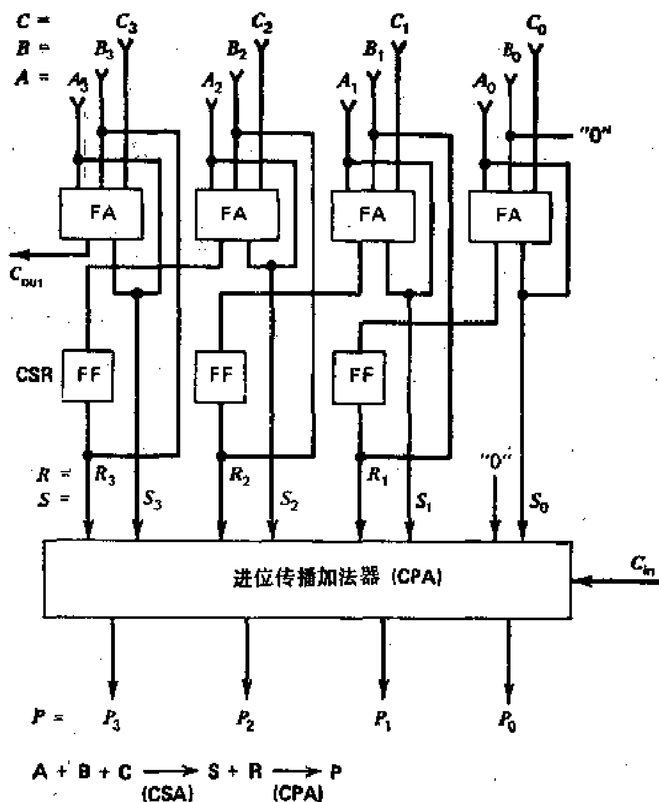


图 4.2 在补码算术运算 (模数为 2^n 的和) 中，多操作数相加所用的进位存储加法器和进位传播加法器 (CSA/CPA) 的组合

最后的进位传播是依靠在 **CPA** 中的先行进位逻辑来加速的. 在组合 **CSA/CPA** 设计中的基本周期，比起单独使用 **CSA** 要稍长一些，这里假定使用相同的时钟去控制 **CSA** 和 **CPA**. 然而，如果考虑到所有工作周期，则 **CSA/CPA** 线路仍然要快得多，因而更符合我们的要求.

4.3 多级进位存储加法器

在图 4.1 和图 4.2 的设计中，仍然是每一个周期输入一个相加的数. 因此，严格地说，这种操作仍是顺序串行的. 我们可以把很多个 **CSA** 连接在一起，组成一个树形多级加法器，这样在每个周期内就可以相加很多个数.

图 4.3 表示同时有 $k=4,5,6$ 和 7 个数相加的情况下的 **CSA** 树形结构. 在每一种情况下，需要有两个反馈输入端来把部分"和"以及左移一位的部分"进位"累加起来. **CSA** 树的级数决定了加法过程的基本周期时间. 每次添加一个输入操作数，就使 **CSA** 的数目增加一个. 为了把 k 个操作数减少到只有两个向量，需要 $k-2$ 个 **CSA**. 每一级 **CSA** 构成的时间延迟为 2Δ，这和一级全加器的延迟相等. 一个最佳设计应该使 **CSA** 树的级数最少. 令 v 是 **CSA** 树的级数. 一个 v 级 **CSA** 树所能处理的操作数的最大数目可以定义为

$$\theta(\nu) \tag{4.2}$$

其中 $\theta(\nu)$ 也包括两个反馈输入. Avizienis[2] 曾推导出一个计算这个数目的递归公式

$$\theta(\nu) = \left\lfloor \frac{\theta(\nu-1)}{2} \right\rfloor \times 3 + (\theta(\nu-1))\text{Mod}2 \tag{4.3}$$

其中 $\nu = 2, 3, \cdots$, 而初始时

$$\theta(1) = 3 \tag{4.4}$$

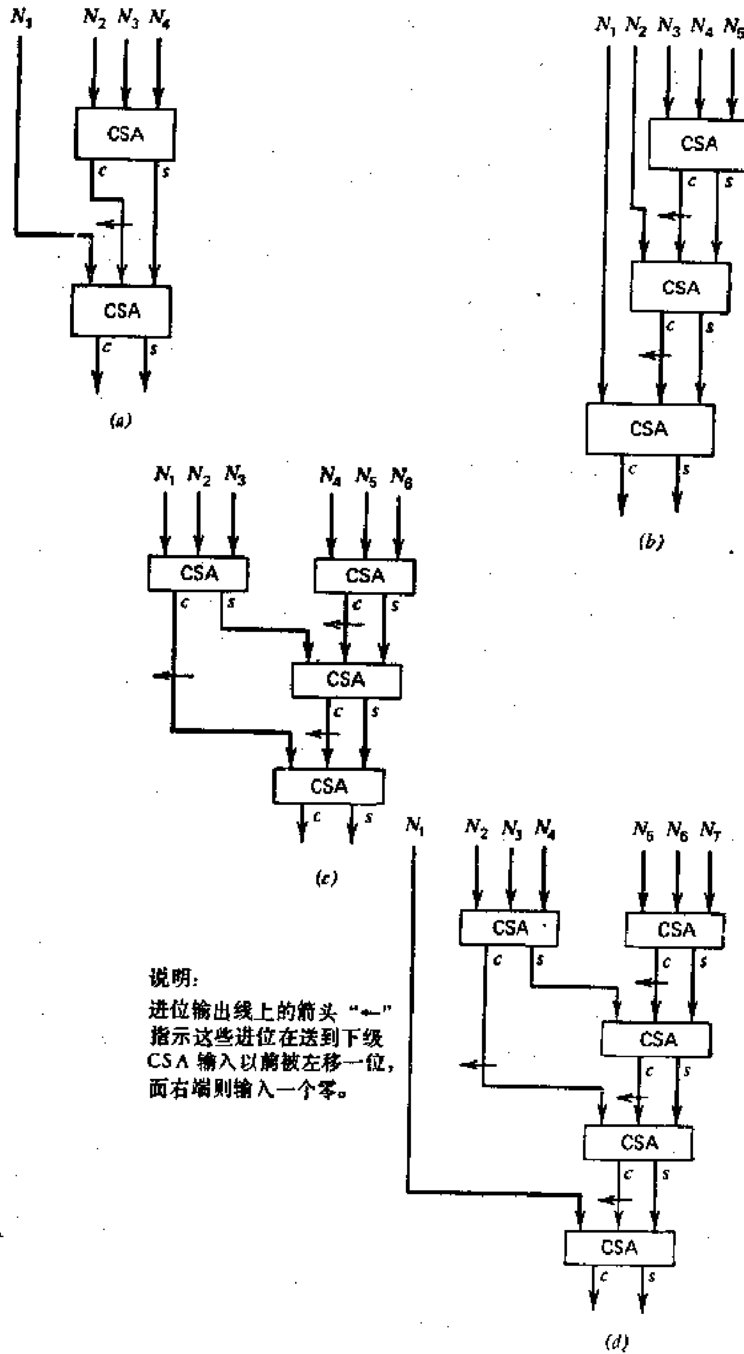


图 4.3 不同输入端数的进位存储加法器(CSA)树.

(a) 4 输入的 CSA 树, (b) 5 输入的 CSA 树, (c) 6 输入的 CSA 树, (d) 7 输入的 CSA 树

表 4.1 总结了在实际情况下不同 v 值时的 $\theta(v)$ 值. 例如,28个数相加,需要一个七级的 **CSA** 树,这表示要有 14Δ 的门延迟.

表 4.1 v 级的 CSA 树所能处理的操作数的最大数目 $\theta(v)$ (Avizienis[13])

v	$\theta(v)$	v	$\theta(v)$
CSA 树的级数	输入操作数的最大数目	CSA 树的级数	输入操作数的最大数目
1	3	6	19
2	4	7	28
3	6	8	42
4	9	9	63
5	13	10	94

注意: 其中包括了循环输入(反馈). v 级 CSA 树的加法时间等于 $2v\cdot\Delta$.

在以后几章中,这些 **CSA** 加法器树将用来设计高速乘法和除法处理器. 应该注意,要使 **CSA** 树工作正确,恰当地把输入和反馈输入按列对齐是很重要的. 为了确保操作正确进位存储寄存器的起始内容应该是零. 进位输出信号在送到下一级以前,必须左移一位.

4.4 按位划分的多操作数加法

与进位存储加法器中按行的并行加法相反,按位划分的加法器是以位片按列的方式执行 k 个输入数的并行加法. 所有的数按位列分成很多小组,每个小组包含相邻的若干列. 在每一个组内,多操作数是一列一列地相加的. 所有划分的子加法器是并行操作的. 所有划分的子加法器的各个"和"输出信号,组合起来形成 k 个输入数的最终的"和". 这种按位划分的线路,至少在三个方面是值得注意的. 首先,它适合于使用 ROM 或 PLA 的大规模集成电路. 其次,它比按行进位存储的方法要快得多. 第三,它很适用于第 11 章中要讨论的高速流水线算术运算.

按位划分的子加法器设计中所包含的基本思想可用图 4.4 中所示的例子来阐明. 这个设计考虑的是九个 3 位数 (划分规模 $m=3$) 的加法. 这九个数是从顺序访问的缓冲存储器(一组九个 3 位的移位寄存器)中取出的. 在每个周期中,存储在缓冲器矩阵中的列信息,向左移动一列的位置. 缓冲寄存器最左边一列的九个二进位 $a_0, a_1, \cdots, a_7, a_8$ 送人一个九输入端的列加法器 (CA) 中,这个 CA 同时产生三个进位 C_3, C_2, C_1 以及一个"和"位 S_0.

列加法器可以用随机逻辑实现,也可用单片 ROM 或 PLA 实现. 图 4.5 表示把部分列加法表编人一个 2048 位的 ROM (512×4) 中. 九个数位 $a_8, a_7, \cdots a_0$ 形成 ROM 的地址输入,它从 512 个字中选出一个字,作为 CA 的"和"输出 C_3, C_2, C_1, S_0. 图 4.4 表示一个用三个半加器,两个全加器以及一个 5 位进位寄存器组成的 5 位进位存储加法器.这个电路在每次列加法时用于把五个进位 $C_5, \cdots, C_1$ 传播到高位的位置. 下标相应于最后"和"输出各位的位置. 在开始按列相加以前,进位寄存器应该清零.

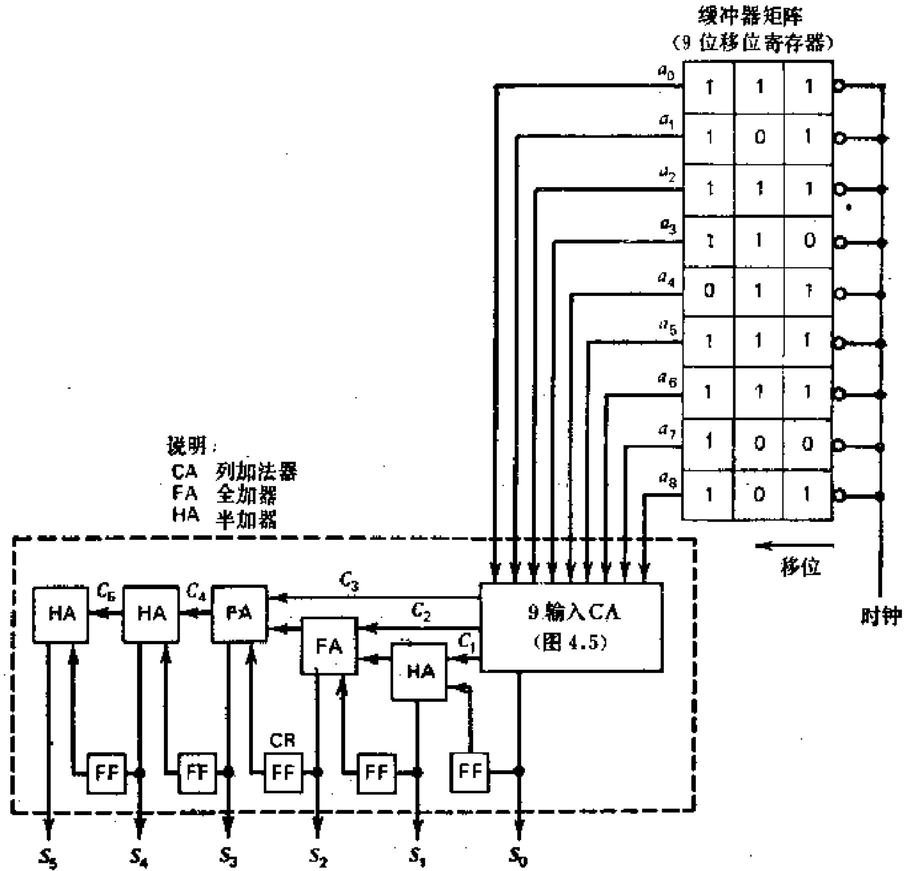


图 4.4 用九个移位寄存器，一个列加法器以及若干个全加器与半加器实现的9位片划分的加法器

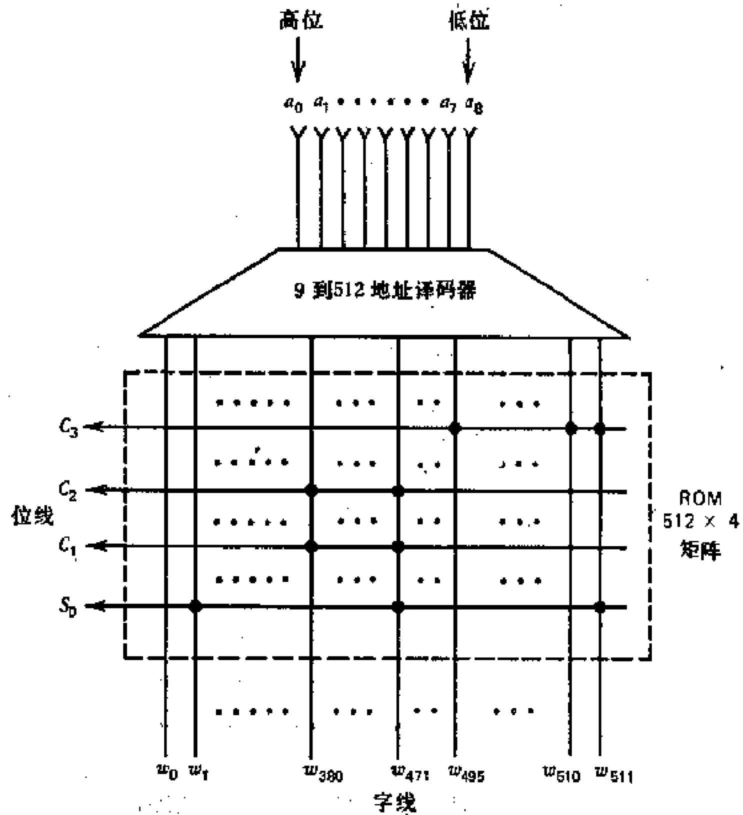


图 4.5 图 4.4 中所用的由 ROM 实现的9位片列加法器 (CA)

4.5 按位划分的多操作数加法器

按位多操作数加法详细的电路工作过程，可以用以下运算步骤以及在缓冲矩阵寄存器中的数值数据组来解释:

第一次加法周期: 把最左一列移位到列加法器，产生“进位”位与“和”位由于第一列的九位中有八个是“1”，所以 C_3, C_2, C_1 和 S_0 为 1000. 第一次的 6 位部分“和”出现在 S_5, S_4, S_3, S_2, S_1 和 S_0 各端，它们为 001000. 5 位进位寄存器现在输入的是 01000.

第二次加法周期: 把目前的最左一列（即在缓冲器矩阵中的第二列）移位到列加法器，以产生 C_3, C_2, C_1 和 S_0，为 0110. 三个进位 C_3, C_2, C_1 = 011 则与进位寄存器当前的内容相加，以产生第二次部分“和”为 01011(00011 + 01000). 这个完整的第二次部分“和”为 010110.

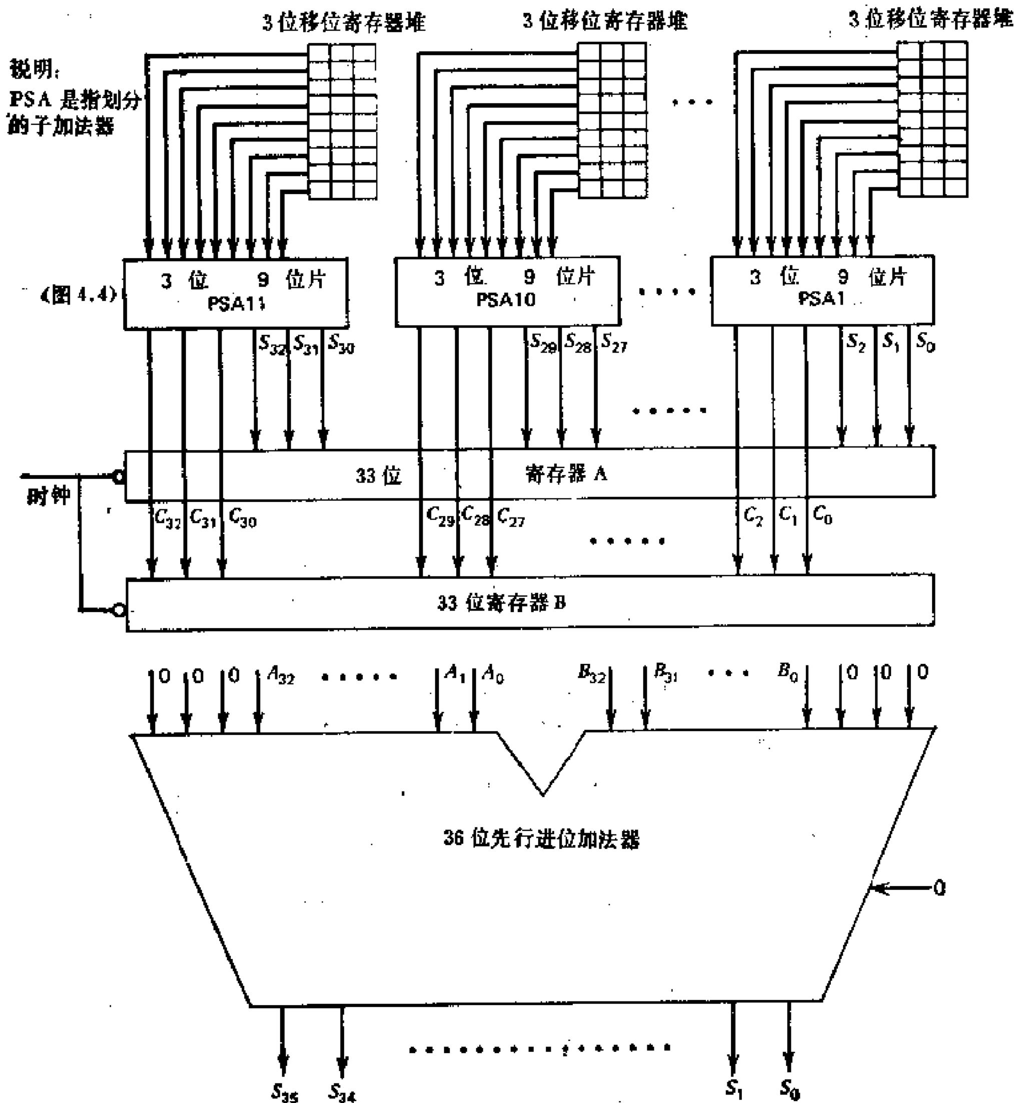


图 4.6 划分规模为 3 的 32 位 9 个数的按位划分加法器

第三次加法周期：把剩下的一列从缓冲器矩阵移位到列加法器．在下一个周期再重复这一过程，以产生这九个 3 位数的最终的"和" $S_5S_4S_3S_2S_1S_0 = 110011$．

以上过程使 k 个数的加法在 n 个周期内完成，其中 n 是操作数的字长．在图 4.4 中设计的按位划分的子加法器，其划分规模 $m = 3$．下面所选的设计例子是用十一个 3 位划分的子加法器，组成一个全字长 ($n = 32$ 位)的九个数的加法器．见图 4.6，所有十一个子加法器的较低 3 位的"和"输出存储在 33 位的寄存器 **A** 中，而所有较高 3 位的"和"是存储在 33 位的寄存器 **B** 中．为了从十一个划分的子加法器中获得在 **A** 和 **B** 中的寄存器"和"，如前所述需要三个加法周期．在第四个周期，寄存器 **A** 和 **B** 的内容，按以下列的排法，送到一个 36 位先行进位加法器中．

$$\begin{array}{r} 0\ \ 0\ \ 0\ A_{32}A_{31}\cdots A_4A_3A_2A_1A_0 \\ +)\ \ B_{32}B_{31}B_{30}B_{29}B_{28}\cdots B_1B_0\ 0\ \ 0\ \ 0 \\ \hline S_{35}\,S_{34}\,S_{33}\,S_{32}\,S_{31}\cdots S_4\,S_3\,S_2\,S_1\,S_0 \end{array},$$

总共需要四个周期以产生九个 32 位输入数的 36 位的"和"

Singh 和 Waxman[21] 曾经证明，在 $m+1$ 个周期内可以加 k 个数，其中 m 是划分宽度的下限

$$m \geqslant \lceil \log_2(k-1) \rceil \qquad (4.5)$$

图 4.6 中所示的多个数的加法器，执行 32 位字长的 $k = 9$ 个数的加法，只需 $m + 1 = 3 + 1 = 4$ 个周期．这比起用单个全字长 $m = n$ 的子加法器所需的 n 个周期，已有显著的改进．表 4.2 提供不同大小的多操作数加法器设计的最小划分宽度 m．举例说，5 位按列划分 ($m = 5$) 的 33 个全字长数(从 16 位到 64 位)的加法，只需要六个 ($m + 1 = 6$) 加法周期．注意总加法时间正比于 $m + 1$，但与加法器的字长 n 无关．

表 4.2 不同数目操作数的按位划分加法的最小划分宽度

相加操作数的数目	最小划分宽度
5	2
9	3
17	4
33	5
65	6
k	$m = \lceil \log_2(k-1) \rceil$

4.6 带符号数字加法/减法

下面我们讨论带符号数字 (**SD**) 算术运算的典型性质，这种 **SD** 算术运算是用有限类型的 **SD** 冗余码来执行数的"并行"加法或减法．在数字计算机中的加法和减法操作期间，这种方法把进位传播限制在向左一位的位置．因此，这种 **SD** 数无论字长有多长，它的加法时间与只有两个相邻数字的相加时间是相同的．为了在快速并行算术运算中做到相加时间与内部数字无关，对于 **SD** 数表示法必须有以下四个要求．

1. 基数 r 是一个大于 2 的正整数．

2. 零的代数值有唯一的 **SD** 表示法．

3. 在机器可表示的范围内，对于每一个代数值，普通的带符号数值 m 位数字表示法与 **SD** m 位数字表示法之间存在着换算关系．

4. 对于两个 **SD** 操作数的所有相应位置的数字，应能执行完全并行的加法/减法．

假如两个 SD 数 Z 与 Y 对所有数字位置 $n-1 \geqslant i \geqslant -k$（如方程式 1.22 所示）均能满足以下两个条件，那么这两个 SD 数的加法或减法可以认为是完全并行的。这里假设 SD 数有 $(n+k)$ 个数位。

$$\mathbf{Z}=(z_{n-1}\cdots z_1 z_0 . z_{-1}\cdots z_{-k})_r$$

及

$$\mathbf{Y}=(y_{n-1}\cdots y_1 y_0 . y_{-1}\cdots y_{-k})_r$$

1. 假设 S_i 是所得到的和 $S=(s_{n-1}\cdots s_1 s_0 . s_{-1}\cdots s_{-k})_r=\mathbf{Z}+\mathbf{Y}$ 的第 i 位"和"的数字，t_i 是从第 $(i-1)$ 个数字位置来的传送数字。于是 s_i 只是三个变量的函数

$$s_i=f(z_i,\ y_i,\ t_i) \tag{4.6}$$

其中 z_i，y_i 分别是被加数 $\mathbf{Z}$ 和加数 $\mathbf{Y}$ 的第 i 位数字。

2. 向左传送到第 $(i+1)$ 位数字位置的传送数字 t_{i+1} 只是被加数数字 z_i 和加数数字 y_i 的函数。

$$t_{i+1}=g(z_i,\ y_i) \tag{4.7}$$

从被减数数字 z_i 中减去减数数字 y_i 的全并行的减法，其执行方法与两个数 z_i 和 $\bar{y}^i$ 的全并行加法是一样的，其中 $\bar{y}_i$ 是 y_i 的加性逆元素

$$z_i-y_i=z_i+(\bar{y}_i) \tag{4.8}$$

注意，在 SD 加法或者 SD 减法中，传送数字 t_i 取正值和负值都可以，不象在普通加法或减法中的"进位"或"借位"只可取非负的数值。传送数字永远不可能传播到超过左面第一个加法器的位置。在给定数字 z_i，y_i 和 t_i 以后，以上定义包含以下描述的两步 SD 加法过程。

第一步，通过把 z_i 加到 y_i，产生一个逸出的传送数字 t_{i+1} 以及一个内部的"和"数字 ω_i

$$r\cdot t_{i+1}+\omega_i=z_i+y_i \tag{4.9}$$

第二步，"和"数字 s_i 是由 ω_i 加 t_i 得到的，t_i 是从数字位置 $i-1$ 来的传送数字

$$s_i=\omega_i+t_i \tag{4.10}$$

为了要满足方程式 4.6 和方程式 4.7 中给定的条件，方程式 4.10 中 s_i 可能取的数值范围，不应超过方程式 4.9 中数字 z_i 和 y_i 的数值范围。方程式 4.8 中规定的 SD 减法是可能的，原因是对于数字 y_i 的每个允许的非零数值，一定有一个加性逆元素 $\bar{y}_i$ 存在于 y_i 的所有允许数值的集合之中。即对每一个 $y_i=a$，必定存在 $\bar{y}_i=-a$，而有

$$y_i+\bar{y}_i=a+(-a)=0 \tag{4.11}$$

第二个要求是要有唯一的零的表示法，这一点是由下列条件来满足的，即允许的数字数值的大小不应超过 $r-1$，

$$|z_i|\leqslant r-1 \tag{4.12}$$

其中 z_i 是任何 SD 数 $\mathbf{Z}=(z_{n-1}\cdots z_0\cdot z_{-1}\cdots z_{-k})_r$ 的第 i 位数字。

第三个要求——可转换性是这样满足的，即完全并行的加法过程在应用于转换任何普通基数的数字 $x_i\in\{0,1,\cdots,r-1\}$ 时，将产生一个基数为 r 的 SD 数字 z_i 的允许数值

$$x_i=r\cdot t_{i+1}+\omega_i \tag{4.13}$$

$$z_i=\omega_i+t_i \tag{4.14}$$

以上的过程其实是对于方程式 1.26，1.27 和 1.28 所规定的过程的一种引伸，其中允

许传送数字 t_i 在必要时可以是负的．以上在方程式 4.9 到 4.12 中所规定的条件，为每个传送数字设立以下一组集合

$$t_i = -1, 0, 1 \tag{4.15}$$

下列条件

$$|\omega_i| \leqslant r - 2 \tag{4.16}$$

是在假设 t_i 限制为 $-1, 0, 1$ 的条件下，作为中间“和”ω_i 的数值的上限而求出的．方程式 4.16 的直接结果是以下的限制条件

$$r \geqslant 3 \tag{4.17}$$

因为，对 $r = 2$，唯一允许的数值 $\omega_i = 0$，这不能满足数值 $x_i = 1$ 的方程式 4.13．按方程式 4.13，4.15 和 4.16 中的要求，建立一组对每个中间数字 ω_i 都能允许的数值的集合

$$\omega_i \in \{-(r-2), \cdots, -1, 0, 1, \cdots, r-2\} \tag{4.18}$$

以上的推导，为全并行 SD 加法/减法系统中的操作数字得出两组扩展的数字集合．

对于一个 $r_0 \geqslant 3$ 的奇数基数，我们选择

$$\Sigma'_{r_0} = \left\{\frac{-(r_0+1)}{2}, \cdots, -1, 0, 1, \cdots, \frac{r_0+1}{2}\right\} \tag{4.19}$$

对于一个 $r_0 \geqslant 4$ 的偶数基数，我们有

$$\Sigma'_{r_e} = \left\{-\left(\frac{r_e}{2}+1\right), \cdots, -1, 0, 1, \cdots, \left(\frac{r_e}{2}+1\right)\right\} \tag{4.20}$$

这两组集合比方程式 1.18 和 1.21 给出的那些包含有更多的冗余性．否则，所需的完全并行性是不能达到的．举例说，给定 $r = 4$，使用数字集合 $\Sigma'_4 = \{-3, -2, -1, 0, 1, 2, 3\}$，取代方程式 1.21 所指的

$$\Sigma_4 = \{-2, -1, 0, 1, 2\}$$

也应当注意，对 $r > 4$，有可能存在多于一个允许数字的数值集合．当每个数字假定有以上规定的最大数值 $\alpha = (r_0+1)/2$ 或 $\alpha = r_e/2 + 1$ 时，一个 **SD** 数的冗余性可认为是最小的．如方程式 1.19 所示，当选择 $\alpha = r - 1$ 时，冗余性被认为是最大的．由于不同程度增加冗余性所引起的基数 $r > 4$ 的 **SD** 系统，其允许的数字集合的数目以 $r - 1$ 为上限，如方程式 4.12 所示．

4.7 全并行 SD 加法器/减法器

在前一节中描述的全并行 **SD** 加法和减法的实现将在本节中讨论．分两步的加法过程可用图 4.7 中的加法器线路来完成．罗马数字 I 和 II 分别用于识别实现方程式 4.9 和方程式 4.10 的两类加法单元．在方框图中只表示出全并行加法器中相应于三个相邻数字的一个部分．

SD 加法的详细执行步骤，可以用一个数值例子来说明．在一个通用的 **SD** 算术运算部件中，加法、减法、左移和右移是基本操作，而乘法和除法则是执行一系列的加法或减法以及移位．为执行 **SD** 减法，先对 **SD** 向量 **Y** 中的所有非零数字的符号求反，以改变减数的符号，然后执行相应数字的全并行 **SD** 加法．ω_i 的允许值按以下序列给定

$$\{\omega_{\min}, \cdots, -1, 0, 1, \cdots, \omega_{\max}\} \tag{4.21}$$

而形成 ω_i，t_{i+1} 和 s_i 的规则，可根据方程式 4.9 按递减的下标 $i = n-1, n-2, \cdots, -k$ 重写如下。

$$\omega_i = (z_i + y_i) - r \cdot t_{i+1} \tag{4.22}$$

其中

$$t_{i+1} = \begin{cases} 0, & \text{如 } \omega_{\min} \leqslant z_i + y_i \leqslant \omega_{\max} \\ 1, & \text{如 } z_i + y_i > \omega_{\max} \\ -1, & \text{如 } z_i + y_i < \omega_{\min} \end{cases} \tag{4.23}$$

“和”数字 s_i 仍旧从方程式 4.10 求得。值得注意的是，SD 数的加法或减法可以从数的任何一端开始。假定我们是从最高位(左端)开始到达右端，那么以下的例子可以用来说明两个基数为 10 的 SD 数的加法。

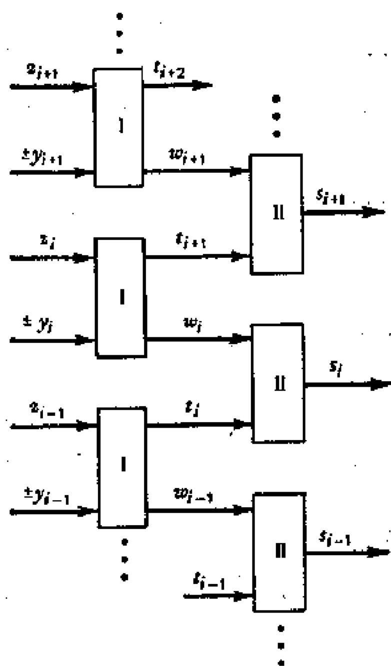


图 4.7 带符号数字 (SD) 运算中全并行加法器/减法器的框图的一部分（只表示 3 个相邻数字）

例如 给定 $r = 10$，则对基数为 10 的 SD 系统所允许的数字集合如下表示

$$\omega_i \in \{\bar{5}, \bar{4}, \bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4, 5\} \tag{4.24}$$

其中 $\omega_{\min} = \bar{5}$，$\omega_{\max} = 5$；

$$t_i \in \{\bar{1}, 0, 1\} \tag{4.25}$$

以及

$$y_i, z_i, s_i \in \{\bar{6}, \bar{5}, \bar{4}, \bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4, 5, 6\} \tag{4.26}$$

考虑以下一对 $n = 1$ 和 $-k = -5$ 的 SD 数的 SD 加法

$$\begin{aligned} \text{被加数 } \mathbf{Z} &= 1.\bar{3}65\bar{1}\bar{4} \\ \text{加　数 } \mathbf{Y} &= 0.\bar{4}053\bar{1} \end{aligned} \tag{4.27}$$

其代数值为

$$\begin{aligned} \tilde{\mathbf{Z}} &= (0.76486)_{10} \\ \tilde{\mathbf{Y}} &= (-0.39471)_{10} \end{aligned} \tag{4.28}$$

Z 和 **Y** 的 SD 加法的详细步骤在表 4.3 中描述。

表 4.3 两个基数为 10 的 SD 数的加法 S = X + Y

数字位置 i	1	0	−1	−2	−3	−4	−5
被加数 Z z_i		1	$\bar{3}$	6	5	$\bar{1}$	$\bar{4}$
加数 Y y_i		0	$\bar{4}$	0	5	3	$\bar{1}$
直接和 $z_i + y_i$		1	$\bar{7}$	6	10	2	$\bar{5}$
传递数字 t_i	0	$\bar{1}$	1	1	0	0	
中间和 ω_i		$(1-0)=1$	$(\bar{7}+10)=3$	$(6-10)=\bar{4}$	$(10-10)=0$	$(2-0)=2$	$(\bar{5}-0)=\bar{5}$
和 S s_i		$(\bar{1}+1)=0$	$(1+3)=4$	$(1-\bar{4})=\bar{3}$	$(0+0)=0$	$(0+2)=2$	$(0+\bar{5})=\bar{5}$

所得到的 **SD** 形式的"和"表示在表的最底下的一行中

$$\mathbf{S} = \mathbf{Z} + \mathbf{Y} = 0.4\bar{3}02\bar{5} \tag{4.29}$$

其代数值为

$$\tilde{\mathbf{S}} = \tilde{\mathbf{Z}} + \tilde{\mathbf{Y}} = (0.37015)_{10} \tag{4.30}$$

SD 数的移位（左移或右移）是这样执行的：假如一个数的数字 z_i 在移位操作时通过加法器逻辑，则当 $z_i > \omega_{max}$ 或 $z_i < \omega_{min}$ 时将产生一个传送数字 t_{i+1}，并将按照方程式 4.22 和 4.23 把这个数字加到中间"和" ω_{i+1} 的左边．当右移一位时，要引入 $z_0 = 0$，作为被移位的数的最高位．如果这个移位包含传送位的产生，则移位寄存器的最低位 z_{-k} 应保留下面的性质

$$|z_{-k}| \leqslant |z_i|_{max} - 1 \tag{4.31}$$

这在多精度操作中是有用的．在左移一位之前，要检查 z_1 和 z_2 数字，它将预告在移位之后是否会产生溢出．

对于字长各为 m 个数字（$m = n + k$，见方程式 1.18）的 **SD** 数的全平行加法器，应包含 m 个相同的数字加法器．每个数字加法器由图 4.7 所示的两个加法单元 I 和 II 组成．这 m 个数字加法器是用传送数字线把它们相邻的加法器连接起来，一个输出连接到左面，一个输入连接到它的右面．所有传送数字 t_i 以及中间"和"数字 ω_i 是同时产生的；因此，**SD** 加法时间与字长 m 无关．它只决定于一级数字加法器的加法时间．

为了节省存储容量，一般都愿意采用最小冗余度的 **SD** 表示法．因此，当使用尽可能最少的数字数值时，可以预期得到不太复杂的数字加法器逻辑．

基数 r 的选择取决于在存储容量的增加与相应的一位数字加法器的逻辑复杂性之间如何平衡．举例说，基数为 4 的 **SD** 算术运算，对于数字加法器输入端的减数的数字要求有七个数值（-3 到 3），这些数值中至少有六个（-3 或 3 可以不用）是为了"和"的数字 s_i，再加上商和乘数数字的不同要求．Avizienis[3] 得出结论，对于所有基数 $3 \leqslant r \leqslant 6$，要求每个数字有三个二进位存储元件，而对于基数 $7 \leqslant r \leqslant 14$，则要求有四个存储元件．举例说，基数 10（有 13 个数值）要求每个数字有四个存储元件，这和普通的 BCD 加法器是一样的．一个基数为 4 的 **SD** 数字加法器实际上设计有七个数值的数字集合（$-3, -2, -1, 0, 1, 2, 3$），它要求三个二进制数字的权，分别为 -4，2 和 1．一个基数为 4 的 **SD** 加法器所需要的逻辑线路，大致相当于普通的基数为 4 的加法器的 12 个半加器．这比具有行波进位传播的普通基数为 4 的加法器要多出三倍．

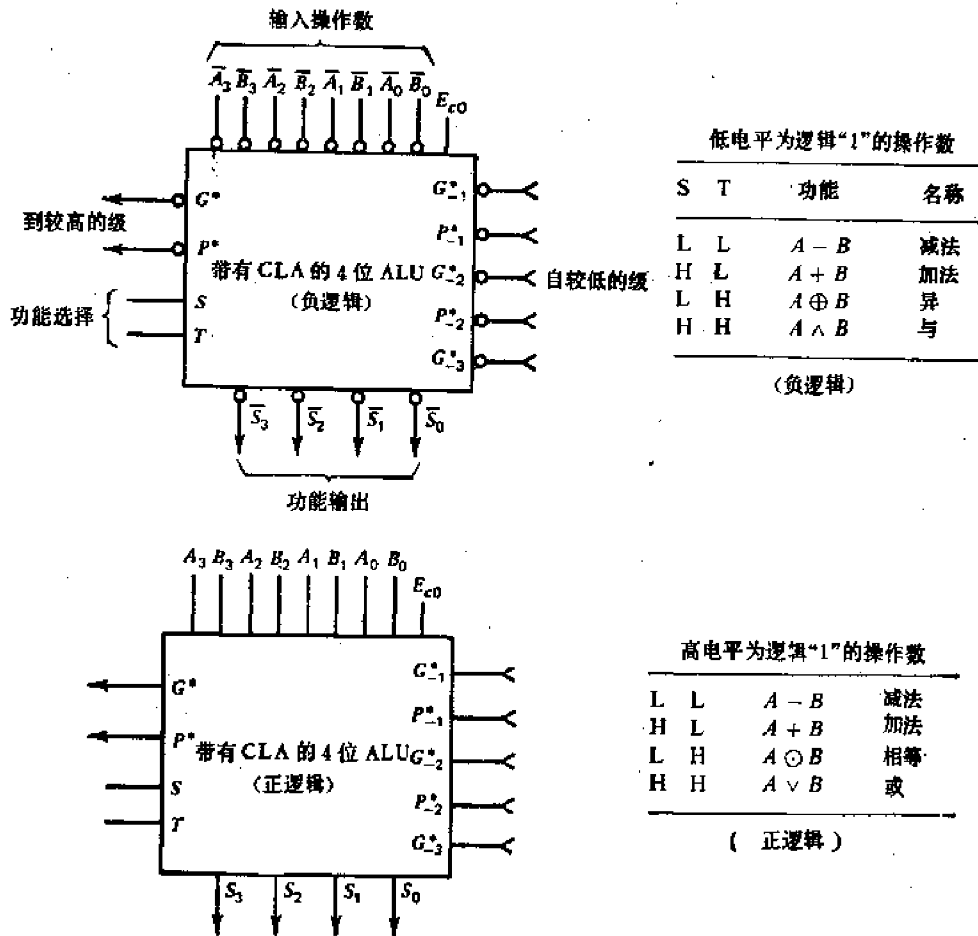
4.8 多功能算术运算逻辑部件

在前章讨论的并行双操作数加法器中，再加入一些附加的组合逻辑，即很容易得到一个算术运算逻辑部件（**ALU**），它不仅能执行两个输入数的很多种算术运算操作，也能执行各种逻辑操作．为了决定这些操作，需要有一些附加的外部功能的选择线．

目前，大部分 **ALU** 是以 MSI 的 4 位片形式提供的．用现有的 LSI 电路可以得到单片 8 位或者一直到 16 位的 **ALU**．因为全部先行进位通常是做在 4 位的位片中，并且为多级先行进位的互连作了准备，即通过将多个 4 位片连接在一起，可得到字长更长的 **ALU**（除非是受到特定的空间和功耗上的限制）．根据封装尺寸的不同，商用 **ALU** 所

能执行的功能数目从 4 到 32 不等.

现在以基本的 4 种功能，4 位 **ALU** 的设计作为例子. 这种部件能对两个 4 位的输入字 $\mathbf{A}=A_3A_2A_1A_0$ 和 $\mathbf{B}=B_3B_2B_1B_0$ 执行加法、减法以及若干逻辑功能，并产生一个 4 位的输出字 $\mathbf{S}=S_3S_2S_1S_0$. 有两条功能选择线 S 和 T 用来决定做什么操作. 如图 4.8 所示，该电路可以对高电平为逻辑"1"的数据总线工作，也可对低电平为逻辑"1"的数据总线工作. 反码与补码的算术运算操作都可用这个 **ALU** 实现.



低电平为逻辑"1"的操作数

S	T	功能	名称
L	L	$A-B$	减法
H	L	$A+B$	加法
L	H	$A\oplus B$	异
H	H	$A\wedge B$	与

（负逻辑）

高电平为逻辑"1"的操作数

S	T	功能	名称
L	L	$A-B$	减法
H	L	$A+B$	加法
L	H	$A\odot B$	相等
H	H	$A\vee B$	或

（正逻辑）

图 4.8 正逻辑和负逻辑操作方式的 4 位 ALU 的方框图和功能表

ALU 内部采用先行进位逻辑来提高其速度. 它具有从前级 **ALU** 位片来的为第二级先行进位作好准备的电路，而无需用到附加的外部逻辑. 为了说明这个 **ALU** 的内部电路操作，假定采用低电平为逻辑"1"的数据输入和输出信号. **ALU** 的内部逻辑可以分成几个功能部分，如图 4.9 所示. 其顶部包含一组门电路，它在 S，T 线的控制下，产生"与"的函数 $\wedge_i$ 以及"或"的函数 $\vee_i$，其中 $i=3,2,1$ 和 0.

$$\wedge_i=\begin{cases}A_i\bar{B}_i, & \text{假如 } S=0,\\ A_iB_i, & \text{假如 } S=1,\end{cases}$$

$$\vee_i=\begin{cases}A_i+\bar{B}_i, & \text{假如 } S=0,\\ A_i+B_i, & \text{假如 } ST=10,\\ 1, & \text{假如 } ST=11\end{cases} \tag{4.32}$$

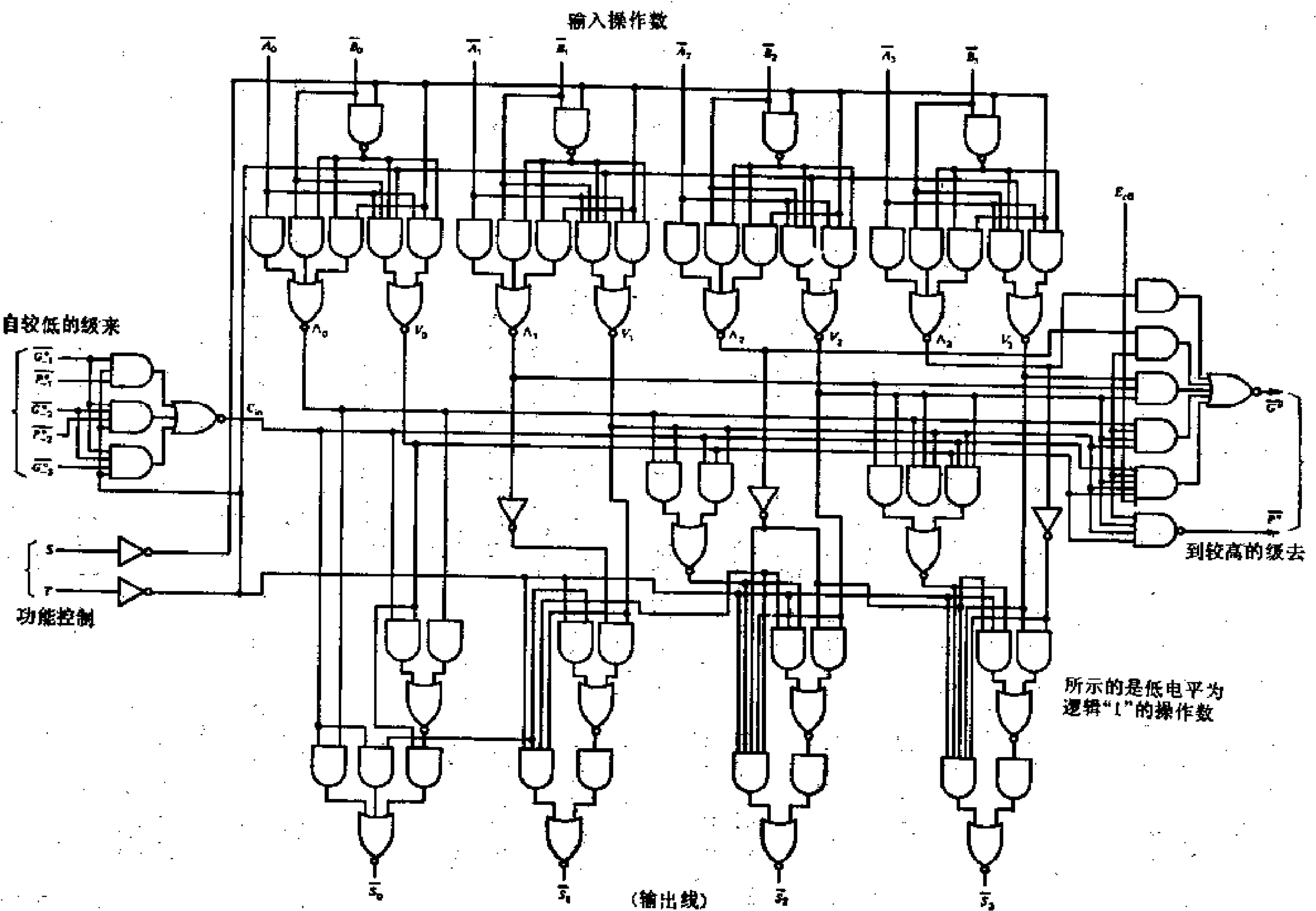


图 4.9　在图 4.8 中所表示的，具有先行进位的 4 位 ALU 原理逻辑图

门函数的不同形式 $\wedge_i$ 和 $\vee_i$ 是用来改变电路执行的功能。

电路的左面部分包含了一组用于产生先行进位输入的门电路。

$$C_{in} = G^*_{-1} + P^*_{-1}G^*_{-2} + P^*_{-1}P^*_{-2}G^*_{-3} \tag{4.33}$$

其中成组进位产生信号 G^*_i 以及成组进位传播信号 P^*_i 在方程式 3.20 中已有定义，只不过原来对于所有 i 的进位传播函数 $P_i = A_i \oplus B_i$，现在是用 $\vee_i = A_i + B_i$ 来代替了。其中负的下标－1，－2和－3，指的是分别从邻近前一级，前二级和前三级 ALU 位片产生的成组进位产生信号或成组进位传播信号。电路的底部包含加法逻辑，它把合适的 $\wedge$，$\vee$ 信号，以及进位信号在每个数位的位置上相加（异或）。进入四个数位位置上的内部进位可列在下面

$$\begin{aligned}
C_0 &= C_{in} + T \\
C_1 &= (\wedge_0 + C_{in}\vee_0) + T \\
C_2 &= (\wedge_1 + \wedge_0\vee_1 + C_{in}\vee_0\vee_1) + T \\
C_3 &= (\wedge_2 + \wedge_1\vee_2 + \wedge_0\vee_1\vee_2 + C_{in}\vee_0\vee_1\vee_2) + T
\end{aligned} \tag{4.34}$$

ALU 的第 i 位输出 S_i 可以写成

$$S_i = (\overline{\wedge}_i \cdot \vee_i) \oplus C_i \tag{4.35}$$

注意在 T 是高电位时的逻辑操作，有一个进位 $C_i = 1$ 强迫进入每个数位的位置，它使

$$S_i = (\overline{\wedge}_i \cdot \vee_i) \oplus 1 = \overline{\overline{\wedge}_i \cdot \vee_i} \tag{4.36}$$

每个输出数字现在正如我们所期望的那样，与进位传播无关。为了完整起见，电路的右边产生两个成组进位的辅助函数

$$\begin{aligned}
G^* &= \wedge_3 + \wedge_2\vee_3 + \wedge_1\vee_2\vee_3 + \wedge_0\vee_1\vee_2\vee_3 + C_{in}\vee_0\vee_1\vee_2\vee_3E_{co} \\
P^* &= \vee_0\vee_1\vee_2\vee_3
\end{aligned} \tag{4.37}$$

要提供一个允许进位输出信号 E_{co}，使得 ALU 位片的进位输出可以写成

$$C_{out} = G^* + P^* \cdot C_{in} \cdot E_{co} \tag{4.38}$$

相应于高电平为逻辑"1"的电路操作，将留下作为读者的练习。更为先进的 ALU 设计及其应用，将在 4.10 节中作为实例研究来介绍。

4.9 ALU 的系统应用

本节将描述使用很多个基本的 4 位 ALU 位片连接起来，构成一个全字长的 ALU 的连接方法。然后我们将用一个 ALU 去比较在不同定点表示法中的二进制数的数值。把 ALU 和其他运算操作联系在一起的系统匹配问题也将予以讨论。

在前一节中所讨论的 ALU 位片设计，可以作为一个积木块来构造一个完整的 ALU，其字长为 4 的倍数。在图 4.10 中构成的 16 位的 ALU 使用四个 4 位 ALU。在这个 16 位的 ALU 中，通过外部的小组进位的连接（见图所示）实现了全部先行进位。可以再连接上一些附加的 ALU 位片，以形成行波成组加法，从而在很长的字长时仍旧能够保持相当高速的性能。每增加 12 位，只添加两级的门延迟时间。举例说，可以在 16 位的 ALU 电路的左端，连接若干个如图中虚线所示的 12 位的积木块，以形成 28 位的 ALU 或者 40 位的 ALU，依此类推。用以上 ALU 位片级联构成的不同 ALU 的典型加法时间，见表 4.4。要做减法，只比加法时间多出一级门延迟时间。

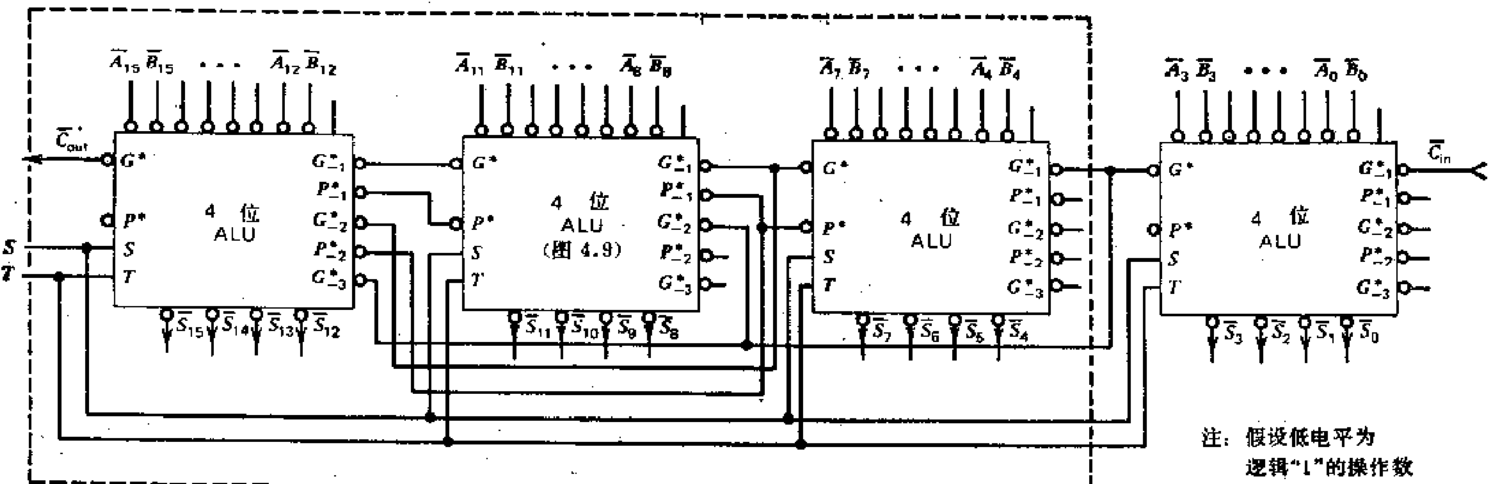


图 4.10　将内有先行进位电路的 4 位的 ALU 级联起来构成 16 位的 ALU

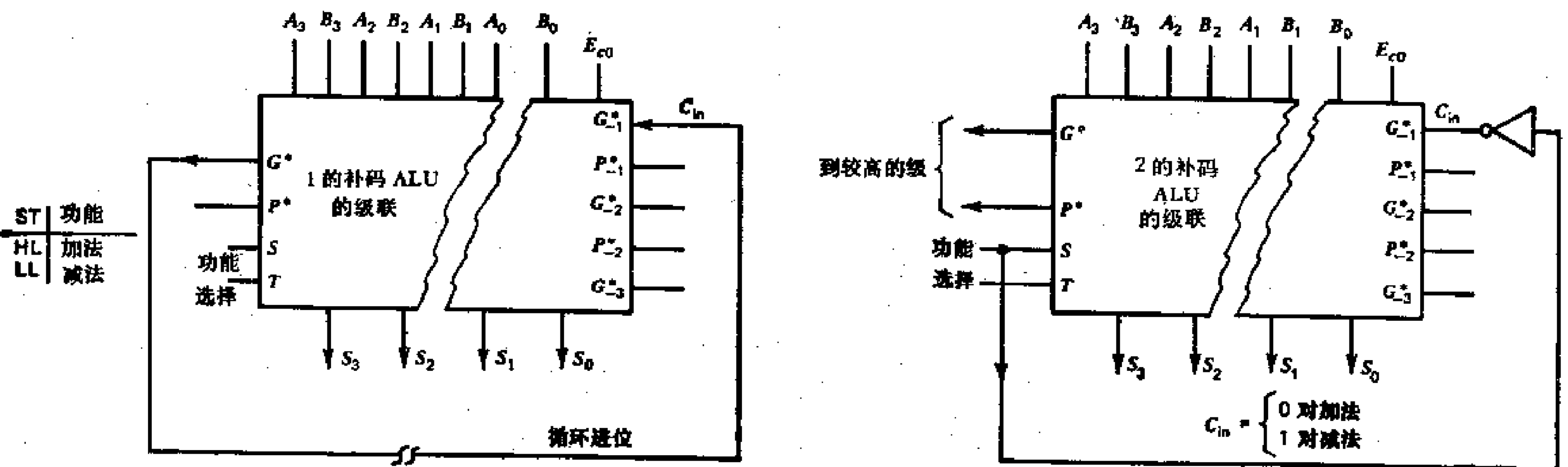


图 4.11　用 4 位 ALU（图 4.8）级联构成的，正逻辑工作的带符号求补算术运算

以上构成的 ALU，可以执行反码也可执行补码运算；可以用正逻辑，也可用负逻辑工作．用以上的 ALU，执行反码或者补码的算术运算操作时所需添加的外部线路的连接见图 4.11．反码运算时的循环进位会增加时间延迟．当需要减法时，采用补码的 **ALU** 的最低位，必须强迫加入一个进位．在 G^*_{-1} 端的辅助进位可以用来输入初始进位．

表 4.4 使用图 4.8 和 4.9 所示的 4 位位片所组成的 ALU 的典型加法时间与减法时间

字长(位)	加法(ns)[1]	减法(ns)[1]
1—4	24	27
5—16	38	41
17—28	53	56
29—40	68	71
41—52	83	86
53—54	98	101

1) 数据是从 Fairchild Semiconductor[1]得到的．

在算术运算过程中，诸如除法，开方等，经常需要用到两个二进制数的数值比较．这种比较操作的执行可用减法模式的电路并要观察其有无进位输出信号．在补码减法中(具有可检出的进位)，这个功能执行的是 **A** − **B**．假如来自符号位的进位输出信号是逻辑"1"，则 **A** ≥ **B**．假如进位输入取消，那么被执行的功能就是 **A** − **B** − 1．如果在这操作中进位输出是逻辑"1"，则 **A** > **B**．如果只有第一种情况是对的，则很明显 **A** = **B**．

用 **ALU** 的进位产生和进位传播输出可以直接检测 **A** 与 **B** 是否相等．如 **A** = **B**，则在减法工作方式或者异或工作方式中，所有内部 V_i 信号将是逻辑"1"，而所有内部 A_i 信号中没有一个是逻辑"1"的．这可以用以下条件在输出端检测到

$$\mathbf{A} = \mathbf{B}, \text{ 如 } P^* \cdot \bar{G}^* = 1 \tag{4.39}$$

要检测整个字是否全等，就必须为每一个部件形成上述条件，同时将所有信号"与"起来．

ALU 的选择或设计必须与计算系统的其余部分联系起来一起考虑．首先，要最大限度地利用 **ALU** 电路片中固有的硬件特点．它们必须与有关定点或浮点算术运算功能的各种微操作相适应．不仅是加法和减法，还要联系乘法，除法，开平方以及其他有用操作一起考虑．

ALU 还必须和逻辑方式或数据总线以及和控制电路相匹配．同步控制和非同步控制有很大差别．此外，**ALU** 应该与主存储器或者高速缓冲存储器相匹配．RAM，顺序存取存储器或者堆栈存储器所要求的中间数据格式也都不同．固定字长和可变字长存储字要求的数据访问线路也不同．如果考虑到所有这些因素，我们可以得到有意义的运算处理器的设计．

4.10 实例研究 I: SN 74181 ALU 的设计和应用

下面我们讨论一个实际的 4 位 **ALU**，它能执行 16 种算术运算操作，包括加法，减法，增量，加倍，求反，通过及其他，还可执行对两个 4 位并行字的所有 16 种可能的逻辑操作．这个器件商业标号为 74181 **ALU**．为了在 32 种操作中选择 1 种，至少需要五条控制线．相应于两种逻辑方式的 74181 **ALU** 的逻辑方框图见图 4.12，其中还列有功能表．工作方式选择线 M 把逻辑操作 ($M = 1$) 与算术运算操作 ($M = 0$) 区别开来．逻辑操作是在按位的基础上执行的．根据是否有进位输入这两种情况，在功能表中算术运算操作分成两列来叙述．

功 能 表							
工作方式选择输入				负逻辑输入与输出		正逻辑输入与输出	
S_3	S_2	S_1	S_0	逻辑 (M = H)	算术运算 (M = L)(C_n = L)	逻辑 (M = H)	算术运算[2] (M = L)(C_n = H)
L	L	L	L	$\bar{A}$	A 减 1	$\bar{A}$	A
L	L	L	H	$\overline{AB}$	AB 减 1	$\overline{\bar{A}+\bar{B}}$	$A+B$
L	L	H	L	$\overline{A+B}$	$A\bar{B}$ 减 1	$\bar{A}B$	$A+\bar{B}$
L	L	H	H	逻辑 1	减 1	逻辑 0	减 1
L	H	L	L	$\overline{A+B}$	A 加 $(A+\bar{B})$	$\overline{AB}$	A 加 $A\bar{B}$
L	H	L	H	$\bar{B}$	AB 加 $(A+\bar{B})$	$\bar{B}$	$(A+B)$ 加 $A\bar{B}$
L	H	H	L	$\overline{A \oplus B}$	A 减 B 减 1	$A \oplus B$	A 减 B 减 1
L	H	H	H	$A+\bar{B}$	$A+\bar{B}$	$A\bar{B}$	$A\bar{B}$ 减 1
H	L	L	L	$\bar{A}B$	A 加 $(A+B)$	$\bar{A}+B$	A 加 AB
H	L	L	H	$A \oplus B$	A 加 B	$\overline{A \oplus B}$	A 加 B
H	L	H	L	B	$A\bar{B}$ 加 $(A+B)$	B	$(A+\bar{B})$ 加 AB
H	L	H	H	$A+B$	$A+B$	AB	AB 减 1
H	H	L	L	逻辑 0	A 加 A[1]	逻辑 1	A 加 A[1]
H	H	L	H	$A\bar{B}$	AB 加 A	$A+\bar{B}$	$(A+B)$ 加 A
H	H	H	L	AB	$A\bar{B}$ 加 A	$A+B$	$(A+\bar{B})$ 加 A
H	H	H	H	A	A	A	A 减 1

H = 高电压电平，L = 低电压电平.

1) 每一位均移位到下一个更高位，即 $A^* = 2A$.

2) 算术运算操作用的补码表示法表示.

逻辑符号

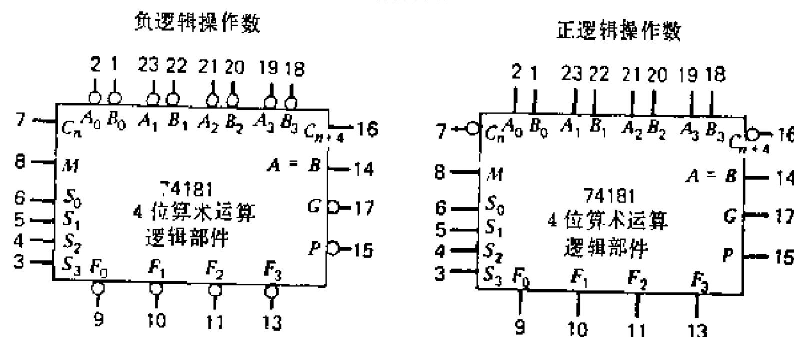


图 4.12 工作于正逻辑或负逻辑操作数方式的 74181 ALU 的方框图和功能表

图 4.13 表示 74181 ALU 的内部逻辑线路图. 在四条功能选择线 $S_3S_2S_1S_0$ 的控制下，辅助的与/或功能是在图的顶部产生的. 这些是用来在电路下部产生"和"以及"进位"项. 当 M 为高时，进位是禁止传播的.

这里采用了内部先行进位，但并没有措施可以预测从前面 **ALU** 位片来的进位. 有一个进位输入端和一个进位输出端，可用作相同部件行波进位的级联. 这种电路有两个辅助进位函数 $\bar{G}$ 和 $\bar{P}$，其定义与第三章中的 G^* 和 P^* 相似. 收集极开路 **A = B** 输出端可以与其他 **ALU** 的 **A = B** 输出端按"与"逻辑接起来，以检测若干部件的全"1"条件. 减法是用反码的加法来执行的，其中减数的反码是内部产生的. 其结果输出为 **A − B − 1**，它需要有一个循环进位或者一个强迫进位，以产生 **A − B**.

比较相对于正逻辑与负逻辑数据的 74181 的两种操作方式，我们可以观察到，这个器

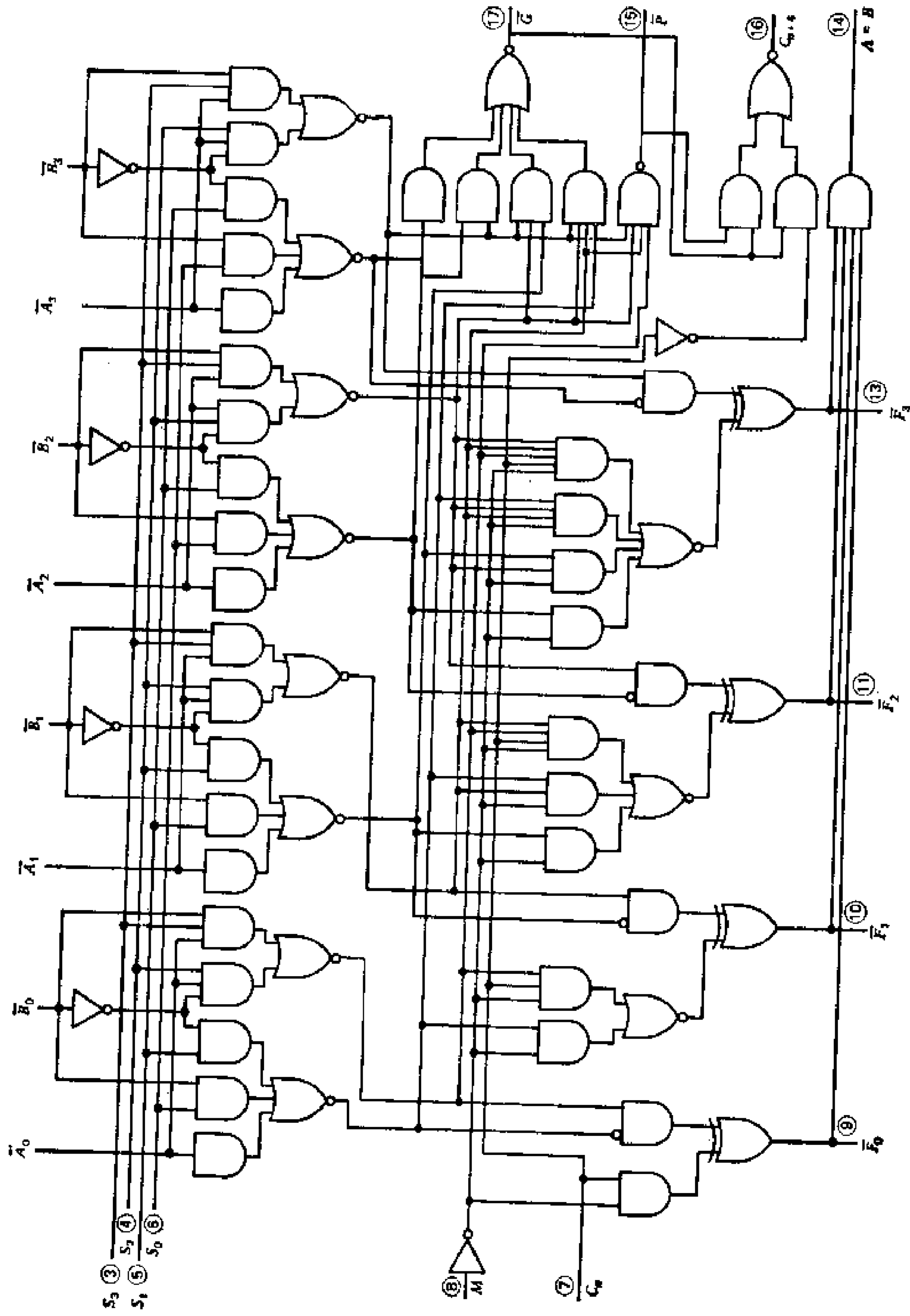


图 4.13 图 4.12（负逻辑操作数）中的 74181 ALU 的原理逻辑电路图

件提供的功能形成了一个封闭的集合，以至于把逻辑输入信号都反相所产生的功能，仍然是在这个集合之中. 因此，这个器件执行的正逻辑数据工作方式的一组运算和逻辑操作，与负逻辑数据工作方式的一组相同. 使用这种 **ALU** 也可执行正负逻辑的混合工作方式，这并不丧失大部分有用的功能. 两种混合工作方式，一种是 A 与 F 为负逻辑以及 B 为正逻辑，另一种是 A 与 F 为正逻辑以及 B 为负逻辑，将留作练习.

下面我们讨论如何用若干个 74181**ALU** 位片，与外部的 74182 先行进位部件(**CLA**)在一起构成一个全字长的 **ALU**. 具有行波先行进位的 32 位 **ALU** 见图 4.14. 在这个电路中用了八个 74181 **ALU** 和两个 4 位 74182 **CLA** 部件. 典型的加法时间以及用 74181 和 74182 设计的不同字长所需的电路片数目见表 4.5.

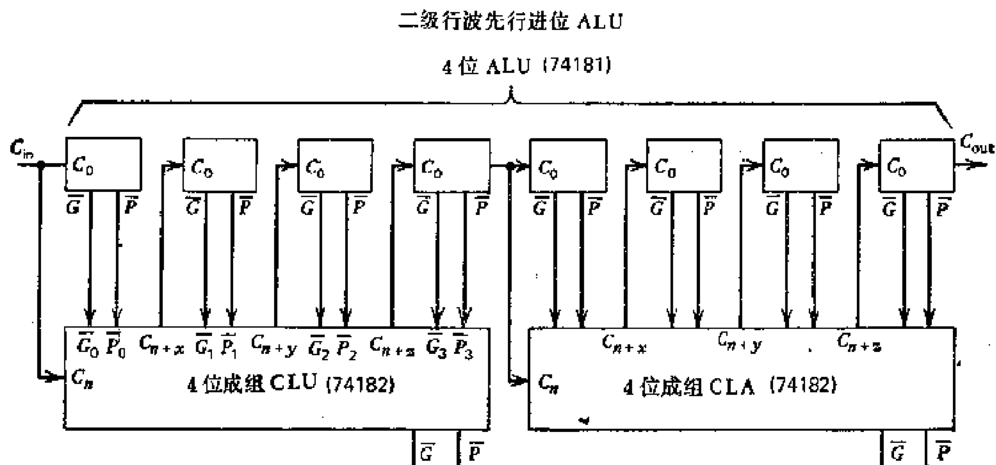


图 4.14 用两个 16 位全先行进位部分级联组成的 32 位 ALU（其中用了八个 74181 ALU 和两个 74182 成组 CLA 部件）

表 4.5 具有不同字长和结构的 ALU 设计的加法时间和电路片数

字长(位数)	ALU 的结构	总的加法时间[1](ns)	集成电路片数	
			74181	74182
4	一级 CLA	21	1	0
8	行波 CLA	36	2	0
12	行波 CLA	48	3	0
12	两级 CLA	36	3	1
16	行波 CLA	60	4	0
16	两级 CLA	36	4	1
32	行波 CLA	110	8	0
32	两级行波 CLA	62	8	2
48	行波 CLA	160	12	0
48	两级行波 CLA	136	12	3
48	三级 CLA	88	12	4
64	行波 CLA	210	16	0
64	两级行波 CLA	101	16	4
64	三级 CLA	64	16	5

1) 数据取自 Fairchild Semiconductor[5].

以下说明用 74181 **ALU** 执行的比较操作. **A = B** 输出是从函数输出 $F_3F_2F_1F_0$ 经

内部译码求得的，当两个数值相等的字加在 **A** 和 **B** 输入端时，**A = B** 输出端将为高电平，它指出两数相等. 74181 **ALU** 有若干种功能，它能用来表示两个输入数的以下关系

$$=,\ >;\ \geqslant;\ <;\ \leqslant,\ \neq \tag{4.40}$$

这些有用的功能见表 4.6. 检查输出端 **A = B** 和进位输出 C_{out} 的数值，可以按表 4.6 中右列所示的关系来决定 **A** 和 **B** 的相对大小. 例如，等值操作 $\overline{\mathbf{A}\oplus\mathbf{B}}$ 可以用来指示相补的关系式 $\mathbf{A}=\bar{\mathbf{B}}$. 由于减法是用反码相加来实现的，如果在输入进位（$C_{in}=0$ 或 1）有信号，将会使减法时差“1”，这就是为什么用减法来比较两个数时必须把输入进位确定好的原因. 对于最高位为正(一个“0”)的不带符号的数，**ALU** 的进位输出将指出它们的相对大小.

表 4.6 用 74181 ALU 执行比较操作

输　出	状　态	操　作	负逻辑	正逻辑
$A=B$	H	A 减 B	A = B	$A=(B$ 减 1)
	H	$\overline{A\oplus B}$	A ≠ B	A = B
	H	$A\oplus B$	A = B	A ≠ B
进位输出	H	A 减 B	A ≥ B	A < B
($\bar{C}_4$ 对正逻辑操作数	L	A 减 B	A < B	A ≥ B
C_4 对负逻辑操作数)	H	A 减 B 减 1	A > B	A ≤ B
	L	A 减 B 减 1	A ≤ B	$A>B$

4.11 参考文献注释

本章中所述的进位存储加法器，基本取材于 MacSorley[7]、Robertson[9]、Rohatsch[10] 和 Wallace[14] 的著作. 在 Wallace[14] 和 IBM 技术人员的报告[3,4]中，可以见到进位存储加法器树的典型应用：实现高速乘法/除法. 多操作数加法的按位划分线路是由 Singh 和 Waxman[11] 提出的. 他们也讨论了如何利用按位划分多操作数加法器以实现快速乘法. Kouvaras 和其他作者[6]，利用一组数字递归公式为主的算术运算过程，提出若干个二进位数同时相加的另一种系统[6].

带符号数字的运算法则最早是由 Aviziens[2,3] 形成的. 在本章中只提到 **SD** 双操作数的加法和减法. 其他的 **SD** 算术运算操作，诸如 **SD** 乘法，**SD** 除法，多操作数 **SD** 加法，溢出检测，多倍精度以及舍入操作，伊利诺埃大学的 Avizienis 和 Robertson[9] 曾予以研究. Metze 和 Robertson[8] 也提出了有关消除数字计算机中进位传播的有意义的报告.

读者如果想知道更多的 **ALU** 设计和应用的材料，可查看 Fairchild Semiconductor 公司的 TTL 应用手册[5]以及 TI 公司的 TTL 数据手册[12]. Texas Instruments 公司最近宣布了单片 I^2L 4 位并行二进制处理器单元[13]. 这个器件不仅包含有全部先行进位实现的 16 位操作的 **ALU**，而且在电路片中还有很多工作寄存器与控制逻辑. 在这个意义上说，单片微处理器明显的是基本 **ALU** 的引伸. 最近宣布的高速性能的双极型位片式微处理器，可以使数字算术运算处理器的设计与制造在性能/价格效率方面达到一个新的水平.

参考文献

[1] Anderson. S. F. et al., "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal of R & D*, Vol. 11, No. 1, January 1967, pp. 34—53.

[2] Avizienis, A., "A Study of Redundant Number Representations for Parallel Digital Computers," Ph. D. Thesis, University of Illinois, Urbana, Illinois, May 1960.

[3] Avizienis. A., "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IEEE Trans. Elec. Comp.*, Vol. EC-10, September 1961, pp. 389—400.

[4] Bratun, J. M. et al., "Multiply/Divide Unit for A High-Performance Digital Computer," *IBM Tech. Disc Bulletin*, Vol. 14, No. 6, November 1971, pp. 1813—1316.

[5] Fairchild Semiconductor Staff, *The TTL Application Handbook*, Fairchild Semiconductor, Mountain View, CA 1973.

[6] Kouvaras, N. D. et al., "Digital System of Simultaneous Addition of Several Binary Numbers," *IEEE Trans. Comput.*, Vol. C-17, No. 10, October 1968. pp. 992—997.

[7] MacSorley, O. L. "High-Speed Arithmetic in Binary Computers." *Proc. of IRE*, Vol. 49, No. 1, January 1961, pp. 67—91.

[8] Metze, G. and Robertson, J. E., "Elimination of Carry Propagation in Digital Computers," *Proc. International Conf. on Inf. Processing*, Paris, France, June 1959. pp. 389—396.

[9] Robertson. J. E., "A Deterministic Procedure for the Design of Carry-Save Adders and Borrow-Save Subtractors," University of Illinois, Dept. of Computer Science, *Report No.* 235, July 1967.

[10] Rohatsch, F. A., "A Study of Transformations Applicable to the Development of Limited Carry-Borrow Propagation Adders," Ph. D. Thesis. University of Illinois, Urbana. Illinois. June 1967.

[11] Singh, S. and Waxman, R., "Multiple Operand Addition and Multiplication," *IEEE Trans. Comput.*, Vol. C-22, No. 2, February 1973, pp. 113—119.

[12] Texas Instruments Staff, *The TTL Data Book and Supplement*, Dallas, Texas, 1973, pp. 38—395 and pp. S312—S313.

[13] Texas Instruments Application Note, *I²L SBP-0400 4-bit Parallel Binary Processing Element*, Texas Instrument Inc., Dallas, Texas 75222, 1976.

[14] Wallace. C. S., "A Suggestion for a Fast Multiplier," *IEEE Trans. Elec. Comp.*, Vol. EC-13, February 1964, pp. 14—17.

习　题

题 4.1 设计一个具有一级 **CSA** 和一个完全先行进位的 **CPA** 的 16 位，对 2 求补，多操作数加法器. 估计一下，用 **CSA/CPA** 加法部件产生 100 个 16 位补码数的"和"所需的总加法时间.

题 4.2 修改题 4.1 中的单级的 **CSA/CPA** 的设计使成为一个多级的设计，其中 **CSA** 循环在每次加法周期中可接受 10 个输入数. 画出逻辑线路图并估计一下，用多级 **CSA/CPA** 加法部件求 100 个以补码表示的 16 位数的加法总时间.

题 4.3 试证明：用于决定 v 级 **CSA** 树所能处理的操作数的最多个数 $\theta(v)$ 的 Avizienis 公式(方程式 4.3)，对所有可能的 v 级 **CSA** 树线路都是适用的.

题 4.4 试设计一个与图 4.4 相似的有移位寄存器阵列，ROM，全加器和半加器的 5 位，33 位片，按位划分的子加法器 (PSA). 然后再用 11 个 PSA，两个 55 位寄存器，以及一个 60 位 CLA 加法器组成一个 55 位，33 个数的加法器.试解释如何用这种多个数的相加部件在六个机器周期内完成 33 个数(55 位字长)的相加.

题 4.5 试证明：使用本文中描述的按位划分多操作数相加方法去相加 k 个数所需的工作周期数的下限，即 Singh 和 Waxman 的下限方程 (方程式 4.5). 把你的证明与文献[11]给出的原来的证明相比较.

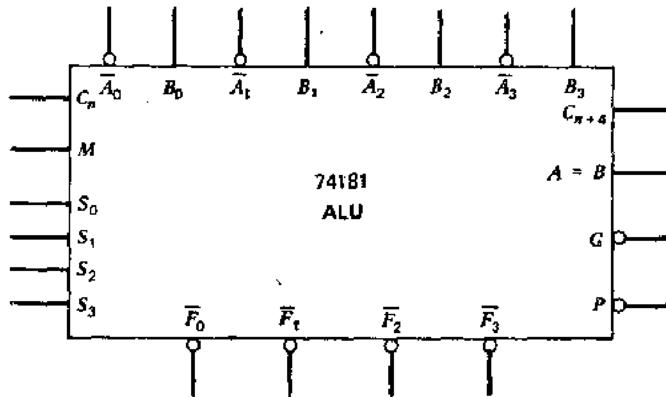
题 4.6 画出能对基数为 4 的 **SD** 数相加的全并行加法器的逻辑线路设计，并用类似表 4.3 的表格来验证你的设计. 注意，基数为 4 的 **SD** 的七个可能数值(−3, −2, −1, 0, 1, 2, 3)，是用三个权为

一4，2，1 的二进位数字来表示的。

题 4.7 画出使用七个如 4.8 节所表示的 4 位 **ALU** 位片组成的 28 位 **ALU** 的内部连接图。这里假设在你的设计中使用对 2 求补的算术运算。

题 4.8 证实 74181 **ALU** 的正逻辑与负逻辑操作方式的功能等价关系，列出对于所有这两种方式的 32 种功能的一一相对应的表。

题 4.9 按下图规定，画出 74181 **ALU** 在具有负逻辑 A 和 F 以及正逻辑 B 的混合操作方式时的功能表。



题 4.10 试说明所有三种 64 位 **ALU** 线路的原理方框图，这三种线路形式是行波 CLA，两级行波 CLA 以及三级 CLA，所用的集成电路片数 (74181 和 74182) 如表 4.5 中最后二行的规定。

题 4.11 给定两个 4 位不带符号的数 **A** = 1011 和 **B** = 1010，试验证表 4.6 中的各项，把这两个二进制数送人 74181 **ALU**，使用表中建议的函数，并计算在输出端 **A** = **B** 的输出或进位输出。

题 4.12 构成一个多级 CSA 树，它能同时接受 30 个外部输入数 (反馈循环输入除外)。试尽可能减少所用的进位存储加法器(**CSA**)的数目，并减少 **CSA** 树级数。

第五章　标准的和再编码的乘法器

5.1 引　言

采用集成电路工艺制成的近代数字计算机中，硬件乘法/除法部件已成为一个标准的特点．本章将讨论在各种数的表示法下，基于加法-移位方法的高速乘法运算处理器．给出通用乘法处理器的详细设计．阐述具有多位扫描和乘数再编码技术的快速乘法器．介绍用进位-存储加法器来实现的高速乘法器．Booth 乘法器[4] 则将按乘数再编码的观点给予讨论．

下面我们用"←"作为替换算子，用"·"作为连接或级联算子，用"(**R**)"表示寄存器或向量变量 **R** 的内容．逻辑"或"和"与"操作分别用"∨"和"∧"表示．另外，加、减、乘、除算术操作仍分别用＋、－、×、/表示．二个 n 位不带符号的整数 A 和 B 相乘，产生一个 $2n$ 位的乘积

$$P = A \times B \tag{5.1}$$

这里 A 是所谓被乘数，B 是乘数．当 A 和 B 均为带符号的整数(三种定点表示法中的任一种)时，乘积只有 $(2n-1)$ 位长．为了实现通用设计，以便能根据用户的选择来处理不带符号和带符号的两种数．我们假定：在处理带符号的数时，$2n$ 位乘积中符号位和最高位之间有一个空位．下面我们先讨论基本的硬件主机以及四种乘法算法，这四种算法分别相应于不带符号的数值，带符号的数值，反码和补码的运算系统．

5.2　间接的乘法算法和硬件

典型的间接乘法部件由图 5.1 所示的几个功能器件所组成．这里要求三个 n 位并行输入的寄存器，即累加器 (**AC**)、乘数寄存器 (**MR**) 和辅助寄存器 (**AX**)．被乘数 A 和乘数 B 一开始分别输入到寄存器 **AX** 和 **MR** 中．**AC** 的初始内容应该是零，即相当于初始部分乘积为零．标记为 A_s 和 B_s 的二个独立的触发器分别用于保留 A 和 B 的符号(对于不带符号的操作，则用零输入到 A_s 和 B_s 中)．另外还需要有一个 n 位的并行加法器，它可以从第三章给出的那些快速加法器方案中选择．

结果形成的 $2n$ 位乘积 P 出现在级联寄存器 **AC**·**MR** 中，这个级联寄存器是通过把 **MR** 连接到 **AC** 的右边构成的．当执行带符号的乘法时，**AC** 中最前一位 AC_{n-1} 载有计算结果的乘积的符号．控制计数器 (**CTR**) 用于记录加法-移位的次数，以及鉴别乘法是否完成．这些功能块的详细互连情况以及有关的控制逻辑将在后面几节中说明．

每一个计数周期都会产生下列操作．我们令

$$\mathbf{S} = S_{n-1} \cdots S_1 S_0$$

为加法器的和输出．C_{in} 和 C_{out} 代表加法器的初始进位-输入和进位-输出．我们定义一

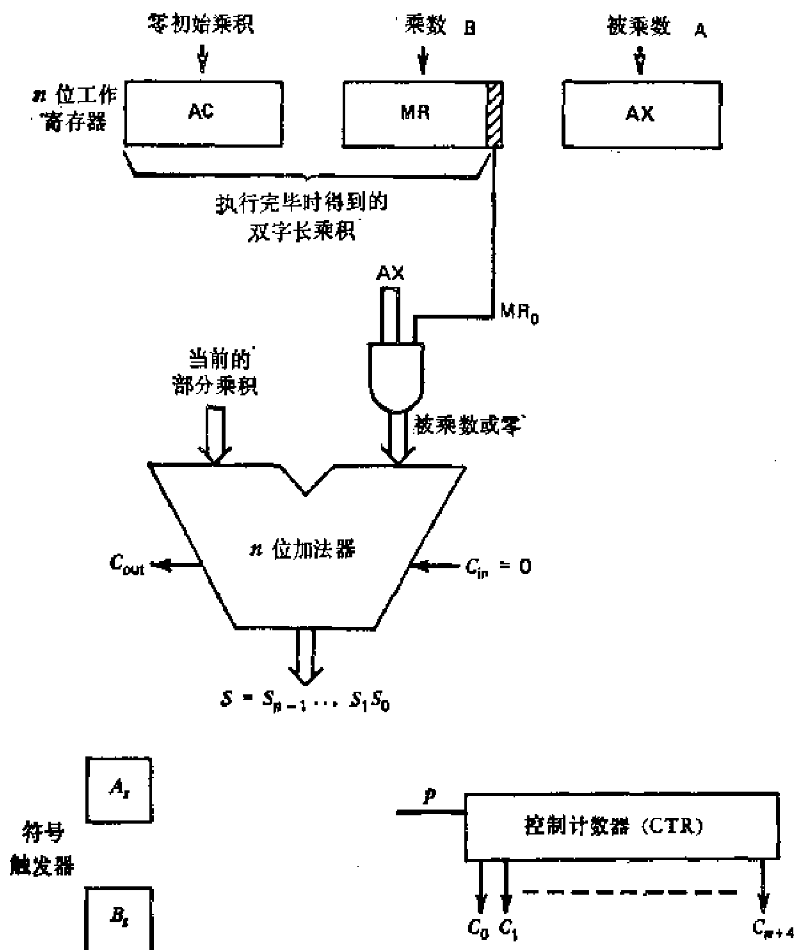


图 5.1　标准的加法-移位定点乘法的基本硬件，每个周期(**AC** · **MR**)均有一次右移

个辅助向量 $\mathbf{AX}\wedge \mathrm{MR}_0$，它是由 **AX** 寄存器的 n 个输出的每一个分别和 **MR** 当前的最低位 MR_0 相"与"得到的.

$$\mathbf{AX}\wedge \mathrm{MR}_0=\begin{cases}\mathbf{AX}, & \text{如果 } \mathrm{MR}_0=1\\ \mathbf{0}, & \text{如果 } \mathrm{MR}_0=0\end{cases} \tag{5.2}$$

黑体字 **0** 表示一个 n 位的零向量. 这个加法器执行下列加法

$$C_{\text{out}}\cdot \mathbf{S}\leftarrow(\mathbf{AC})+(\mathbf{AX}\wedge \mathrm{MR}_0) \tag{5.3}$$

其中每个计数周期都用一个零作为初始进位-输入 ($C_{\text{in}}=0$). 在这个加法中有可能产生进位 C_{out}. n 位加法器可以是一个具有固定时间延迟的全先行进位加法器. 当加法器输出端形成"和" S 后，可以立即执行下面的右移操作

$$\mathbf{AC}\cdot\mathbf{MR}\leftarrow C_{\text{out}}\cdot S_{n-1}\cdots S_1\cdot S_0\cdot \mathrm{MR}_{n-1}\cdots \mathrm{MR}_1 \tag{5.4}$$

这就是把"和"输出右移 (包括 C_{out} 位进入 AC_{n-1}) 以后输入 **AC**，S_0 输入到 MR_{n-1}，以及 MR_0 从 **MR** 的右端被推出. 这种带存储的移位操作可用多路转换器的逻辑来实现，后者能按所要求的移位方式把适当的输入选送到工作寄存器中.

上述步骤说明，在每个计数周期根据被检测的乘数位 MR_0 是"1"还是"0"，将把被乘数 A 或者把一个零向量加到部分乘积的前 n 位上. 部分乘积逐次右移等效于手算时所执行的左移，手算时被乘数的各位将按一定的列的位置写入. 上述过程可以重复 n 次，每次

检测乘数的一位，直到乘数的所有位都被检测到为止．计算结束时，n 位的乘数 B 将从 **MR** 的右端全部推出，代替它的是最终乘积的后半部．在 **AC** 中则保留着乘积的前半部．

不带符号和带符号数值的乘法

二个 n 位不带符号的数的乘法可用图 5.2 所示的流程图来详细说明．完成这个操作只需要 $n+1$ 个周期．把数据输入到适当的寄存器和触发器中，是在图中所指的周期 C_0 内实行的．乘法循环包括 n 个周期 $C_1, \cdots, C_n$．为了对二个 n 位带符号数值的数相乘，只需要 $n-1$ 个周期，这是因为只有 $n-1$ 个数值位．在起始周期 C_0，A 的符号位 a_{n-1} 和 B 的符号位 b_{n-1} 分别输入触发器 A_s 和 B_s．寄存器 AX 和 MR 的领前各位一开始用零输入．乘法过程结束时，预期的乘积符号 $A_s \oplus B_s$ 应输入到累加器中的符号位 AC_{n-1}．最

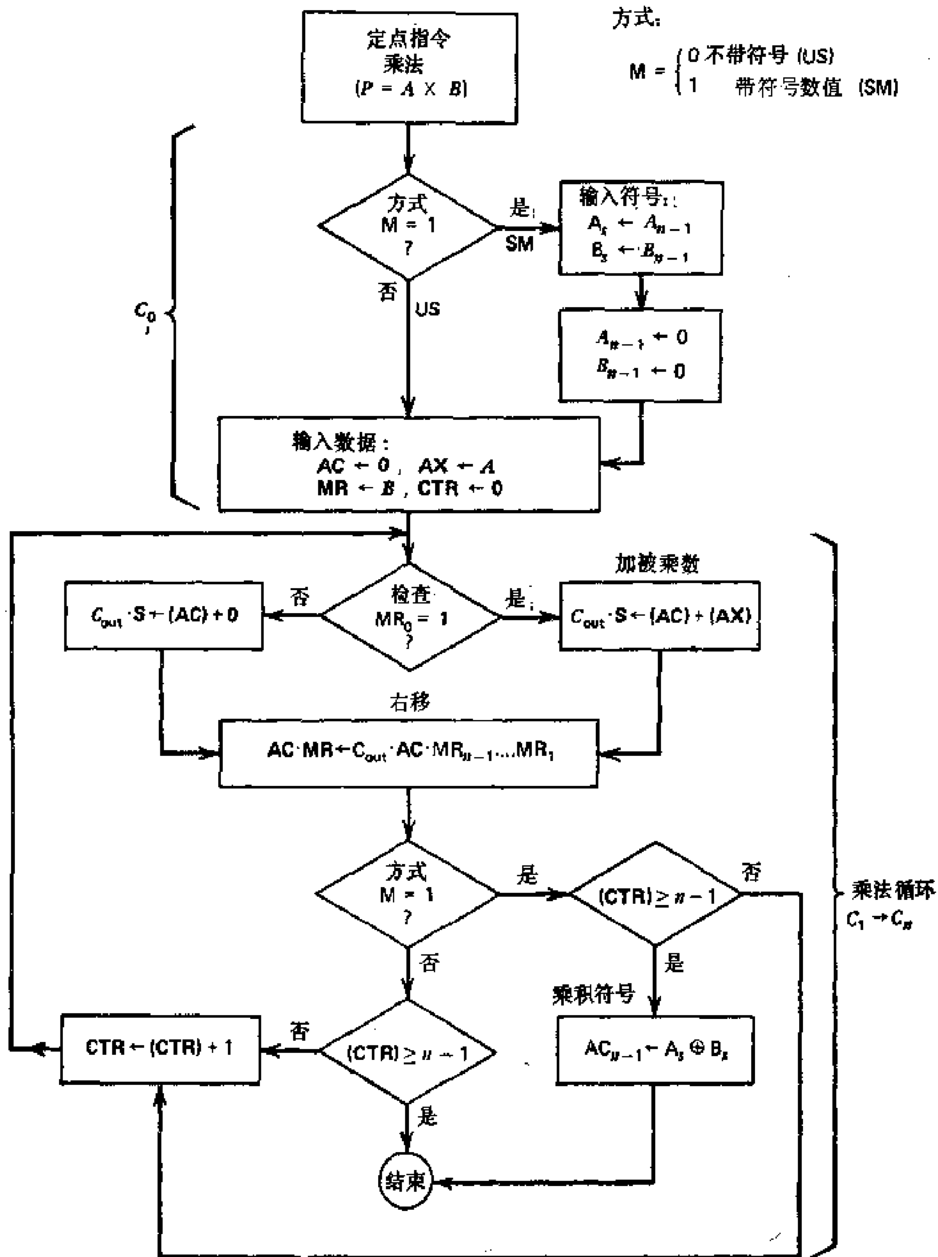
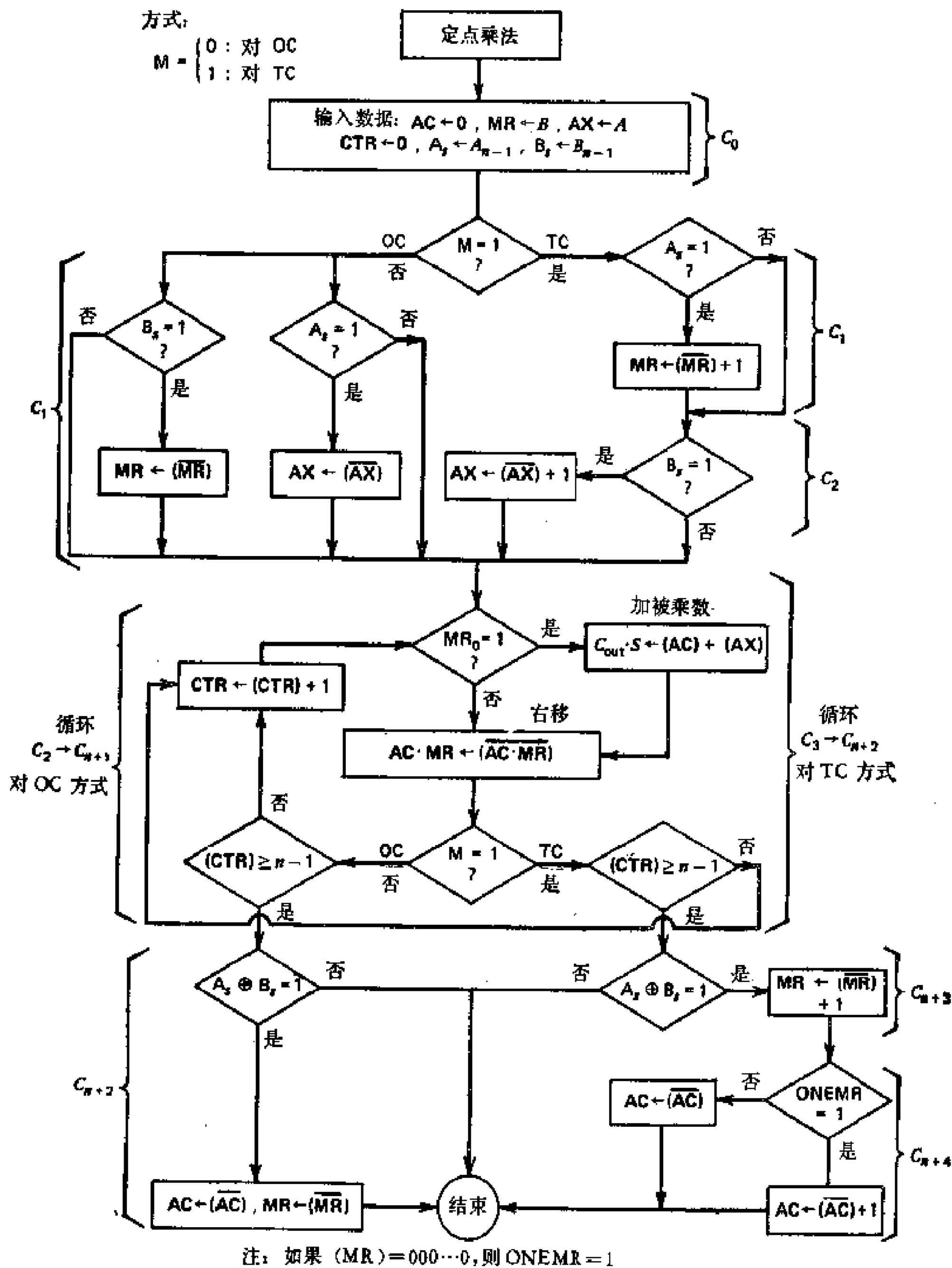


图 5.2　基本的不带符号与带符号数值“加法-移位”乘法的算法



注：如果（MR）＝000…0，则 ONEMR＝1

图 5.3 基本的、间接的反码和补码"加法-移位"乘法的算法

后得到的双倍字长的乘积将留在级联寄存器 **AC · MR** 中．一个寄存器的内容是指这个寄存器所存的数值，如图 5.2 所示，**AC** 的内容用 (**AC**) 来表示．

间接的带符号-补码乘法

带符号-补码数(1的补码或 2 的补码表示法)相乘的间接算法需要二次求补步骤：输入操作数如果是负的，则在进入乘法循环之前应转换成带符号数值的形式，计算结果的乘积如果是负的，也应该转换成所需要的形式．如图 5.3 所示，反码和补码的乘法两者均需

使用一个算前求补器和一个算后求补器来完成这些操作.

反码乘法需要用二个周期 (C_1 和 C_{n+2}) 来进行按位的算前求补和算后求补操作. 补码操作必须用四个周期 (C_1, C_2, C_{n+3} 和 C_{n+4}). 为了得到补码，可以先产生给定数的反码，然后在结果的最低位上加 1. 这个加法可以通过送一个 1 到加法器的 C_{in} 端来实现；这时将正在被增量的数输入到加法器的一个输入端，而将另一个零向量输入到加法器的另一个输入端，然后，经过加法器来执行这个加法. 也可以利用具有计数能力的寄存器来实现加 1.

当有 2 的求补器(图 2.11 b) 可用时，为了产生给定数的补码，只需要一个周期. 换句话说，算前求补周期 C_1 和 C_2 以及算后求补周期 C_{n+3} 和 C_{n+4} 两者都能合并成一个周期. 两个周期减为一个周期并不一定有益处. 这是因为要适应 2 的求补器电路中较长的进位延迟，周期时间可能被延长. 在表 5.1 中给出了以上四种间接乘法的概况. 不带符号的数值 (US)、带符号数值 (SM)、反码 (OC) 和补码 (TC) 的乘法分别需要 $n+1$, $n+1$, $n+3$ 和 $n+5$ 个机器周期.

5.3 一个间接的通用乘法器的设计

图 5.2 和 5.3 所描述的带符号数值和间接的带符号补码的乘法可以合并，得到一个通用设计. 如图 5.4 所示，为了确定工作方式，需要二条外部的方式控制线 X 和 Y. 缩写词 US, SM, OC 和 TC 分别用来表示被译码的方式控制信号，即不带符号的数值、带符号的数值、反码和补码乘法. 这个通用乘法的方法用图中所附的功能表来描述. 在 2 的补码相乘时，要求 $n+5$ 个计数周期. 对所有这四种工作方式来说，每个周期所进行的主要操作都已归纳在这个表中. 除了补码操作以外，这个处理器在某些周期可能是空闲的.

为了得到通用设计，必须把图 5.1 所描述的硬件主体加以扩充. 所有寄存器的功能以及控制线的分配的有关说明均已表示在图 5.5 中. 这里，功能方式控制用一个 2 至 4 行译码器来实现. 在三个工作寄存器 **AC**,**MR** 和 **AX** 中，每一个都能假定具有下列四种功能: 按位求补，增 1，并行输入，带有 SRI 串行输入的情况下右移一位，或者使寄存器清除为零. 例如，当 EAC 线①转向高电平，选择线②和 ③ 为低电平时，加法器输出将右移后送到 **AC**. 所有可能的寄存器功能都已表示在这个线路图中. 如果补码乘法的最终乘积是负的，则在最后的算后求补周期 (C_{n+4}) 应将 **AC** · **MR** 中的按位求补的乘积再加上 1. 这个加法可以分两步说明如下:

首先在周期 C_{n+3} 对二个寄存器求补，**AC** · **MR** ← $(\overline{\mathbf{AC}}) \cdot (\overline{\mathbf{MR}})$. 然后在周期 C_{n+4} 使寄存器 **MR** 增 1. **MR** 上的净操作为

$$\mathbf{MR} \leftarrow (\overline{\mathbf{MR}}) + 1 \tag{5.5}$$

如果 **MR** 的最后内容为零，则必有一个进位输出进入 **AC**. 这个条件用布尔变量 $ZEROMR$ 表示，后者是用 n 输入的或非逻辑来产生的. 当下列条件满足时

$$C_{n+4} \wedge \mathrm{TC} \wedge ZEROMR \wedge P_s = 1 \tag{5.6}$$

在周期 C_{n+4} 期间要求进行下面的操作

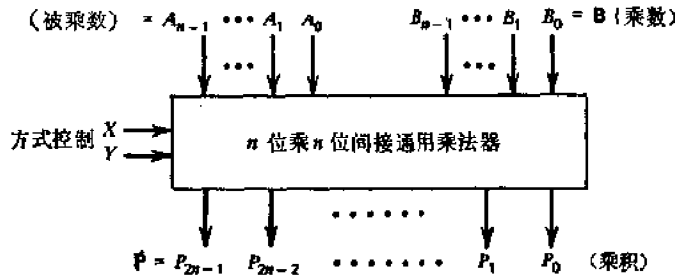
$$\mathbf{AC} \leftarrow (\mathbf{AC}) + 1 \tag{5.7}$$

这样，负的乘积便能以补码形式正确地出现在级联寄存器 **AC**·**MR** 中. 最终乘积的符号为

表 5.1　四种标准的加法-移位间接乘法线路的操作一览表

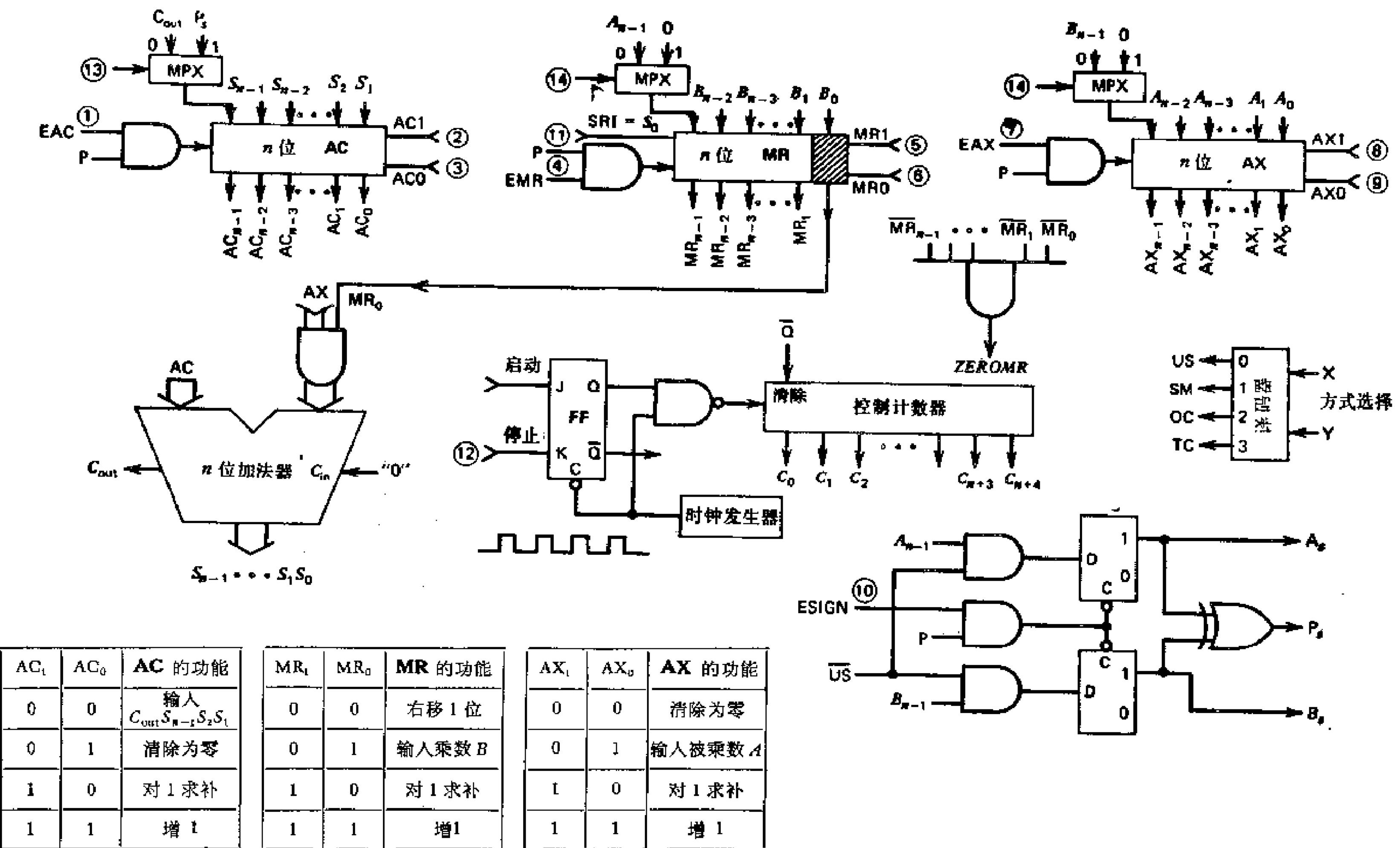
所执行的操作	乘法器周期			
	US	SM	OC	TC
将操作数从存储器送到寄存器，将符号送到标志触发器	C_0	C_0	C_0	C_0
对 OC 或 TC 方式，将负操作数按位求补。对 SM 方式将 0 输入到 MR_{n-1} 和 AX_{n-1}			C_1	C_1
对 TC 方式，将求补的负操作数增1				C_2
由每个周期一次的加法-移位操作所组成的乘法循环	$C_1 \to C_n$	$C_1 \to C_n$	$C_2 \to C_{n+1}$	$C_3 \to C_{n+2}$
对 OC 或 TC 方式，若乘积符号是负的，则将 AC · MR 中的乘积按位求补。对 SM 方式，将乘积的符号输入到 AC_{n-1} 中。			C_{n+2}	C_{n+3}
对 TC 方式，若乘积是负的，则将 AC · MR 中被求补的双倍字长的乘积增 1				C_{n+4}

注：n 为操作数的字长。



周期	方式 XY			
	0 0 US	0 1 SM	1 0 OC	1 1 TC
C_0	输入数据	输入数据和符号	输入数据和符号	输入数据和符号
C_1		对负的符号求补	对负的操作数按位求补	对负的操作数按位求补
C_2				求补的操作数增 1
C_3 ⋮ C_{n+2}	n 个周期每个周期均匀右移一位的乘法循环			
C_{n+3}		建立最后乘积的符号	对负的乘积按位求补	对 AC · MR 中负的乘积按位求补
C_{n+4}				MR 增 1，而且若方程式 5.6 正确，则 AC 也增 1

图 5.4　一个 n 位乘 n 位的间接通用乘法器的说明



AC_1	AC_0	**AC** 的功能
0	0	输入 $C_{out}S_{n-1}S_2S_1$
0	1	清除为零
1	0	对 1 求补
1	1	增 1

MR_1	MR_0	**MR** 的功能
0	0	右移 1 位
0	1	输入乘数 B
1	0	对 1 求补
1	1	增1

AX_1	AX_0	**AX** 的功能
0	0	清除为零
0	1	输入被乘数 A
1	0	对 1 求补
1	1	增 1

图 5.5　n 位乘 n 位间接通用乘法器的原理逻辑电路图

$$P_s=\begin{cases}A_s\oplus B_s, & \text{如果 SM}+\text{OC}+\text{TC}=1\\0, & \text{如果 UN}=1\end{cases} \tag{5.8}$$

控制计数器实质上是一个模-$(n+5)$的自激时钟($\lceil\log_2(n+5)\rceil$位),它具有译码输出 C_0, $C_1,\cdots,C_{n+4}$. 计数器的频率和处理器的主时钟一致. MULTIPLY(乘法)触发器用于控制乘法过程的开始和结束.

下面讨论的是确定每个处理器周期中给控制线(有圆圈标记)指定数值的过程. 控制线数值的各种组合形成许多本征操作,称为微操作. 用控制线作为列的标题,用计数周期作为行的标题,由此得到的矩阵如表 5.2 所示. 这样一个矩阵被称为微操作表或控制矩阵. 在这个表中,行的各项相应于每个操作周期所要求的控制线的数值. 这些数值能通过对照流程图检查来确定. 大多数项是一些 0 或 1. 有一些项带有逻辑条件. 例如,方程式 5.6 中的条件决定了 **AC** Enable 线 ① 在周期 C_{n+4} 时应该是"1". 于是,这个测试条件被列在表 5.2 中周期 C_{n+4} 和控制线①的交点上.

通过调查这个微操作表,可以立即为每一条控制线推导出逻辑方程式. 例如,AX Enable 线⑦应确定为:

$$⑦ = C_0 \vee C_1 \wedge A_s \vee C_2 \wedge A_s \wedge \text{TC} \tag{5.9}$$

AC Enable 线 EAC 为

$$① = \text{TC} \wedge ZEROMR \wedge P_s \wedge C_{n+4} \vee C_0 \vee C_3 \vee \cdots \vee C_{n+2} \vee C_{n+3} \wedge P_s \wedge X \tag{5.10}$$

送到 MULTIPLY 触发器 J 端的 START(启动)信号是由中央处理器内的指令译码器产生的. 每当执行 MULTIPLY 指令时,该触发器就应该置位,直到乘法操作完成为止. 另一方面,在最后一个计数周期应产生 STOP(停止)信号. 每当 MULTIPLY 触发器被清除时,就不会产生脉冲去启动乘法部件.

微操作表所描述的激励逻辑可以用二种不同的方法来实现. 第一种方法是使用硬接

表 5.2 图 5.5 中的通用乘法器的微操作表

机器周期	控制线													
	① EAC	② AC_1	③ AC_0	④ EMR	⑤ MR_1	⑥ MR_0	⑦ EAX	⑧ AX_1	⑨ AX_0	⑩ ESG	⑪ SR_1	⑫ STP	⑬ SPD	⑭ SMZ
C_0	1	0	1	1	0	1	1	0	1	1	d	0	0	0
C_1	0	d	d	B_s	1	0	A_s	1	0	0	d	0	0	SM
C_2	0	d	d	$B_s\wedge$TC	1	1	$A_s\wedge$TC	1	1	0	d	0	0	0
C_3	1	0	0	1	0	0	0	d	d	0	S_0	0	0	0
⋮	1	0	0	1	0	0	0	d	d	0	S_0	0	0	0
C_{n+2}	1	0	0	1	0	0	0	d	d	0	S_0	0	0	0
C_{n+3}	$P_s\wedge X$	1	0	$P_s\wedge X$	1	0	0	d	d	0	S_0	0	SM	0
C_{n+4}	$ZEROMR$ $\wedge P_s\wedge$TC	1	1	$P_s\wedge$TC	1	1	0	d	d	0	d	1	0	0

注: "d" 是指"随意"条件. $X=\text{OC}\vee\text{TC}$.

线的随机组合逻辑电路来实现控制线方程式，如象上面导出的方程式 5.9 或方程式 5.10. 用这种方法实现的运算处理器称为硬接线控制的运算处理器. 第二种方法用控制存储器(ROM) 来存储微操作表的编码方案. 微指令的格式是微程序的主要论题. 由固件控制来操纵的运算处理器被称为微程序的运算处理器. 时钟频率应决定于电路中沿着所有可能的信号通路的最长传播延迟. 这可能包括激励逻辑、多路转换器、寄存器、加法器等所引起的延迟. 采用目前常用的 7400 系列 TTL 门和触发器以及先行进位加法器时，在乘法处理器中最坏情况下的延迟可以小于 300 ns，这相当于时钟频率大于 3 兆赫.

5.4 乘法中的多次移位

数字计算机在执行乘法指令时，如果每个周期所检查的乘数位多于一位，乘法的速度便可以加快. 象这种多位扫描要求每次加法后作多次移位. 例如，每次检查二位，那么加法-移位周期的总数就可以减半. 这些逐次扫描的位组可以是分离的，也可以是重叠的. 在这一节中，我们先研究不重叠的多位扫描的乘法. 这二种方法均不难用 **CSA** 树来实现. 显然，在速度上的提高就要求消耗额外的硬件. 然而，在采用 **CSA** 回路时，硬件的增加是相当有限的. 为了便于说明，我们将介绍不带符号的整数乘法的方法. 只要作较小的修改就能适合其他的运算表示法.

初始的寄存器分配和前面相同，即被乘数 $A=(\mathbf{AX})$；乘数 $B=(\mathbf{MR})$；以及 $0=(\mathbf{AC})$. 现在让我们用每次检查二个乘数位的例子来说明它的工作原理. 我们假定字长为偶数 $n=2k$，这样假定并不失去一般性(如果不是偶数，我们可以在高位端人为地插入一个零，使字长成为偶数). 在乘数的低位端开始操作，一次右移二位.

在扫描了最低的一对乘数位 $(\mathrm{MR}_1, \mathrm{MR}_0)$ 后，有四种可能的动作，如表 5.3 所示. 对于 $m=(\mathrm{MR}_1,\mathrm{MR}_0)_2$ 来说，被乘数 A 的倍数 $m\times A$ 被加到当前的部分乘积上，用来生成下一个部分乘积. 必须指出，由于多次移位，所得到的和可能大于 n 位.

$$C_{\mathrm{out}}\cdot S\leftarrow(\mathbf{AC})+m\times A \tag{5.11}$$

每个被乘数的倍数是由若干个把 A 移位后的内容进行线性求和得到的. 二进制数左移 i 位，同时让零进入右端，即等效于将这个数乘以 2^i. 因此，如果 $m=(b_d,\cdots b_1b_0)_2$，我们有

$$m\times A=\sum_{i=0}^{d}b_i\times 2^i\times A \tag{5.12}$$

表 5.3 中每一行都可以用方程式 (5.11) 求出，这里 $m=(b_1b_0)_2=(\mathrm{MR}_1,\mathrm{MR}_0)_2$.

在加法后，级联寄存器 **AC · MR** 右移二位，即

$$\mathbf{AC}\cdot\mathbf{MR}\leftarrow\overbrace{C_{\mathrm{out}}S_nS_{n-1}\cdots S_2}^{n\text{位}}\cdot\overbrace{S_1S_0\mathrm{MR}_{n-1}\cdots\mathrm{MR}_3\mathrm{MR}_2}^{n\text{位}} \tag{5.13}$$

必须指出，下一对较高位的两个乘数位已经被移到最低的位置上，并且准备在下一个周期中接受检测. 这种 2 位扫描过程本身要重复 $k=\lceil n/2\rceil$ 个周期，一直到乘数的所有位都检测完为止. 至于最后乘积，可以从级联寄存器 **AC · MR** 中得到. 上述原理可以推广到任意

表 5.3 在扫描了一对乘数位之后，加到部分乘积上去的被乘数的倍数

MR_1	MR_0	要加的倍数
0	0	0
0	1	A
1	0	$2A$
1	1	A, $2A$

大小的扫描位组．一般地说，在一个 d 位扫描系统中，要加的被乘数的最大倍数可写成

$$(2^d - 1) \times A = 2^{d-1} \times A + \cdots + 4A + 2A + A \tag{5.14}$$

非重叠的多位扫描乘法可以用 4.2 节中的进位存储加法器（**CSA**）和进位传播加法器（**CPA**）来实现．我们用每个周期扫描二个乘数位（$d = 2$）来阐明这个设计．被乘数倍数的加法是用一个 3 输入、2 输出的 **CSA** 和一个 2 输入、1 输出的 **CPA** 按照图 5.6 互连构成的．**CSA** 的二个输入端是为倍数 $2A$ 和 A 所保留的．这些倍数分别依靠与 MR_1 和 MR_0 中的乘数位相"与"来启动操作的．**CSA** 的其余输入取自保存在 **AC** 中的当前的部分乘积．然后将 **CSA** 的二个输出用（$n + 2$）位的 **CPA** 求和，形成新的部分乘积，结束时得到最后乘积．

每次叠代后，按照方程式5.13所示的传送方式，新的部分乘积将右移 2 位．乘法循环包括了 $k = \lceil n/2 \rceil$ 次叠代．一个初始周期用于输入数据，总的乘法时间为 t 个周期，对于每个周期扫描 m 位的一般情况，这里

$$t = \left\lceil \frac{n}{m} \right\rceil + 1 \tag{5.15}$$

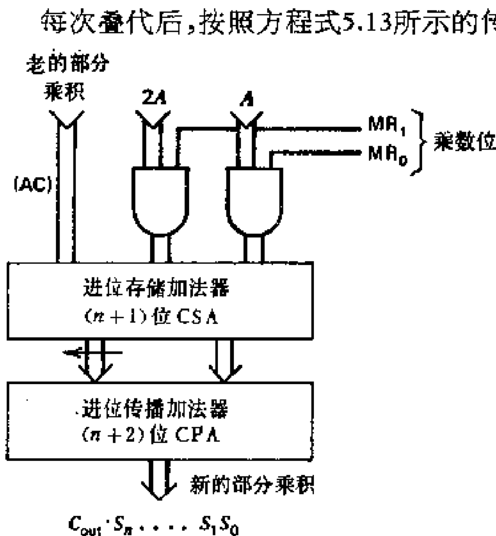


图5.6 在一个 2 位乘数扫描系统中，用于加被乘数倍数的进位-存储加法器/进位传播加法器的复合体

图 5.6 所示的硬件乘法部件要求用

$$t = \lceil n/2 \rceil + 1 = \lceil 2k/2 \rceil + 1 = k + 1$$

个周期来完成二个 $2k$ 位数的乘法．

为了增加扫描位组的位数，可能要用多级 **CSA** 树．每增加一级 **CSA** 就有 2Δ 的时间延迟．带有全 CLA 的 **CPA** 有固定的时间延迟，即对 2 级 CLA 来说为 12Δ．工作寄存器、多路转换器和倍数选择门的延迟归并在一起为 8Δ．因此，一个使用 v 级 **CSA** 树的 m 位扫描乘法器，它的基本乘法周期(一次叠代的时钟周期)可以估算如下：

$$p = 12\Delta + 8\Delta + 2v\Delta = (20 + 2v)\Delta \tag{5.16}$$

总的乘法时间由 t 乘以 p 求得

$$\text{乘法时间} = p \times t = (20 + 2v) \times \left(\left\lceil \frac{N}{m} \right\rceil + 1 \right) \Delta \tag{5.17}$$

对一个 m 位扫描的乘法器，要求 **CSA** 的个数等于

$$\#(\mathbf{CSA}) = m - 1 \tag{5.18}$$

在下一节中，我们将说明如何用重叠扫描来减少这个数字．应该指出，"m 位扫描"也可以想象成是高基数乘法的一种类型（$r = 2^m$），这将在后几节讨论．

5.5 重叠的多位扫描

在非重叠的位扫描方法中，乘数的每一位生成一个被乘数的倍数，加到部分乘积上．当扫描位组的位数 m 较大时，即意味着所要加的被乘数的倍数有许多个．MULTIPLY 指令的执行时间主要取决于执行加法的次数．因此，希望能减少上述倍数的个数．下面介绍一种重叠的多位扫描方法，它将使每个加法周期中被乘数倍数的数量减少一半．这就

是说，在一个重叠设计中，实际上只需要非重叠设计中所用 **CSA** 的一半.

这个想法是基于用移位越过乘数中的一串零，从而减小了执行时间. 乘数中零的个数愈多时，这个操作就愈快. 现考虑乘数中有一串 k 个连续的 1，如下：

$$\begin{array}{ll} \text{列的位置} & \cdots, i+k, i+k-1, i+k-2, \cdots, i, i-1, \cdots \\ \text{位的内容} & \cdots, 0, \underbrace{1, 1, \cdots, 1}_{k\text{个连续的}1}, 0, \cdots \end{array} \tag{5.19}$$

利用数字串的特性：

$$2^{i+k} - 2^i = 2^{i+k-1} + 2^{i+k-2} + \cdots + 2^{i+1} + 2^i \tag{5.20}$$

我们可以用下面的数字串来代替 k 个连续的 1

$$\begin{array}{ll} \text{列的位置} & \cdots, i+k+1, i+k, i+k-1, \cdots, i+1, i, i-1, \cdots \\ \text{位的内容} & \cdots, 0, \underset{\uparrow\,\text{一次加法}}{1}, \underbrace{0, \cdots, 0}_{k-1\text{个连续的}0}, \underset{\uparrow\,\text{一次减法}}{\bar{1}}, 0, \cdots \end{array} \tag{5.21}$$

上面划有一横的低次项 $\bar{1}$ 表示一个 -1，相应于执行一次减法. 利用这种乘数再编码的方法，我们只要在数字串开始时作一次加法，结束时作一次减法，使之能够代替原来的 k 次连续加法. 显然，当 k 很大时，能节省大量的加法时间. 我们用这个串特性来阐明为什么重叠的位扫描是有利的.

乘数位仍按二位一组分成许多组(对). 但一次扫描三位，二位来自现在的对，第三位来自下一个高次对的低位. 实际上每一对的低位被检测了二次. 这种扫描可从乘数的任何一端开始. 为了与前面的说明取得一致，我们假定扫描乘数的习惯是从右端(最低的一对)到左端. 重叠和非重叠两种扫描模式表示在图 5.7 中，这里乘数的字长是 14 位.

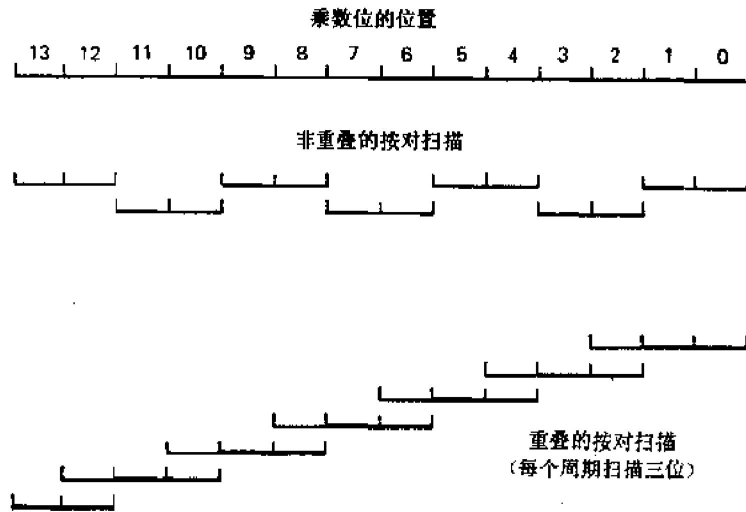


图 5.7　非重叠的与重叠的 2 位扫描模式的比较

每三位一组检测后的动作说明在表 5.4 中. 表中指出了每个机器周期或者执行一次单纯的移位，或者执行一次加法，或者执行一次减法. 这里只需要倍数 $2A$ 或 $4A$. 当下一对的低次位 x_{i+2} 为 0 时，三位中最左边的 1 经常指示一串 1 的左端 (结束). 依据方程式 5.20 中所描述的串特性，在具有非零的乘数位时应该执行加法. 另一方面，当 $x_{i+2}=1$ 时，即意味着是一串 1 的右端(开始)或中心，按照串特性需要作一次减法. 在每个加法周期中，部分乘积每次要右移二位. 这就使部分乘积比它应该具有的数值小了 4 倍被乘数

表 5.4 在重叠的按对扫描的系统中，扫描三个乘数位以后所要加的被乘数的倍数

乘数位			所要加的被乘数的倍数	用方程式 5.20 中的串特性来解释
下一个高次对的低位 X_{i+2}	现存的一对 X_{i+1}	现存的一对 X_i		
0	0	0	0	没有一串 1
0	0	1	$+2A$	串结束
0	1	0	$+2A$	孤立的 1
0	1	1	$+4A$	串结束
1	0	0	$-4A$	串开始
1	0	1	$-2A$	串开始和结束
1	1	0	$-2A$	串开始
1	1	1	0	串中心

$(-4A)$. 这可以用在下一步扫描中加上所需被乘数的倍数与 4 倍数的差值来校正. 倍数 $2A$ 或 $4A$ 进入加法器的地点是重要的. 如果一对的结尾是零，那么所得到的部分乘积是正确的，而且下一次的操作是一次加法. 如果一对的结尾是 1，则所得到的部分乘积太大，所以下一次操作将是一次减法.

应该指出，一个被求补的部分乘积会引起一些附加的周期. 这表示移位器必须设计成也能处理补码数的移位. 在最早的周期，有一个零初始部分乘积. 因此，部分乘积的输入线均为零. 如果乘数最低位是 1，那么 A 的补码通过部分乘积输入端进入加法器，否则就接收一些零. 同时，由第一对乘数位译码来选择的被乘数的倍数，将从另外的 **CSA** 输入端进入. 上述线路可以扩充到三位一步(如果包括来自下一个三位的重叠位，那么实际上是四位). 我们鼓励读者去探索与较高次的重叠位扫描有关的规则. 下面接着讨论基于上述重叠的 2 位扫描方法的硬件乘法部件的结构.

5.6 采用重叠扫描的乘法器设计

上述技术要求对一串 1 的开始和结束进行检测. 下面我们来描述一个叠代的硬件处理器，它将同时对六个重叠的三位一组的乘数对 (13 位)进行译码. 每次叠代只需要六个被乘数的倍数，其数值可取为 0，$\pm 2A$ 或 $\pm 4A$. 反之，利用等效的非重叠的设计，每次叠代需要 12 个倍数. 图 5.8 指出了重叠扫描的模式，这里每次叠代有六个生成的被乘数倍数，这个乘法器设计实际上已经用在 IBM 360/91 计算机中. 在每次 13 位的重叠方式中，五次叠代便足以解决所有 61 个乘数位. 注意在 IBM360/91 中用于乘法的是 56 位的分数或 14 位十六进制的尾数. 右端再补上四个零，使之成为 15 个十六进制数位，如图 5.8 所示. 包括符号位在内的操作数共有 61 位，虽然其中只有 56 位是分数的有效数值位.

图 5.9 指出了二种不同的 **CSA** (进位存储加法器) 树. 这二个 **CSA** 树把六个生成的倍数合并成二个向量——部分和与部分进位. 树上每一个 **CSA** 的长度等于被乘数的长度. 部分和与部分进位右移 12 个位置，循环地返回成为反馈输入. 在乘数被送入以后，这二个来自 **CSA** 的向量通过一个进位传播加法器相加以形成最后乘积. **CPA** 具有的长度应能容纳双倍字长的最后乘积，即对 56 位乘 56 位的乘法器应有 112 位.

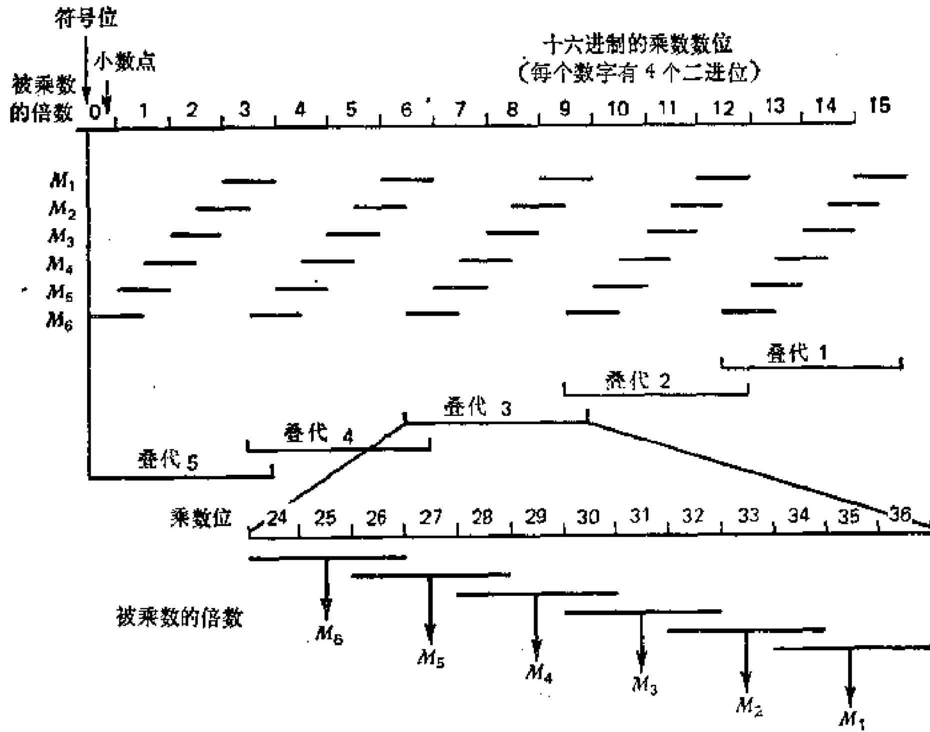


图 5.8　在 IBM 360/91 型系统的乘法部件中，叠代方式与倍数的生成

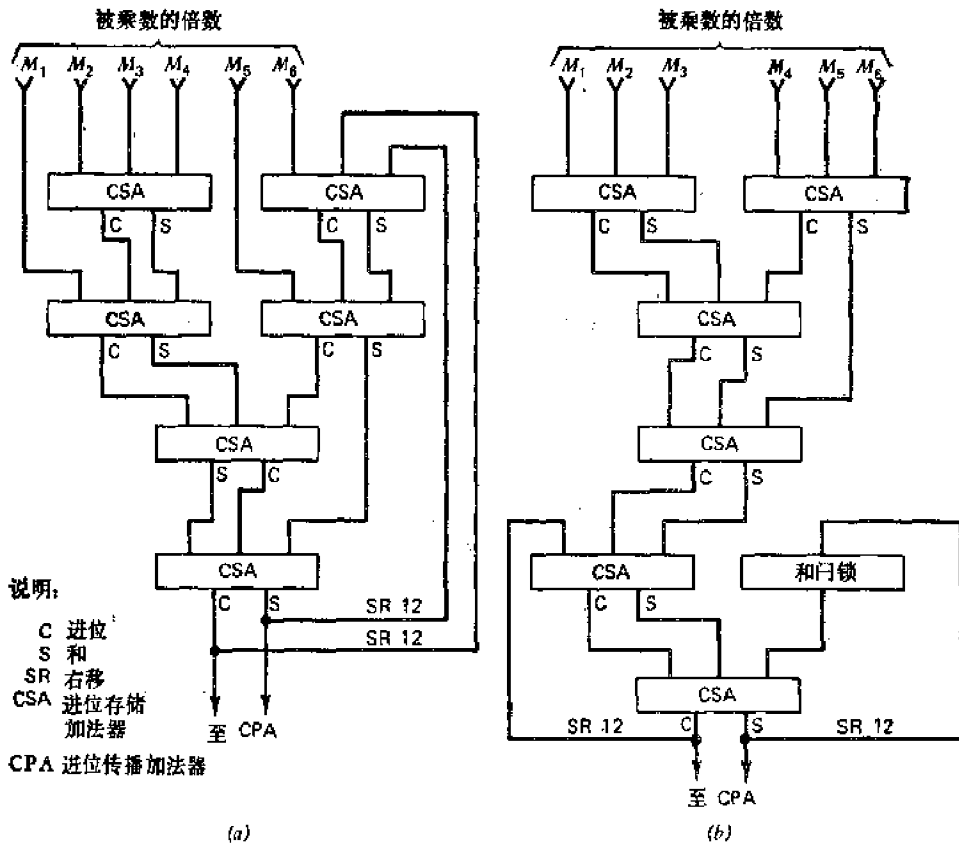


图 5.9　将六个被乘数倍数（图 5.8）加到累计的部分乘积上的二种可能的 CSA 树（Anderson等[1]）。
(a) 来自输出端的反馈回路；(b) 在输出端上的累计回路

这二个 **CSA** 树只是在形成反馈回路的方法上有所区别. 图5.9左边的 **CSA** 树有一个来自输出端的反馈回路,而右边的 **CSA** 树则在输出端上有一个累计回路. 主要区别在于每个树各有其不同的叠代周期. 左边的树,一个叠代周期等于通过整个四级的树所需要的时间. 右边的设计中的安排允许叠代周期接近于只通过最后二级 **CSA** 的延迟. 在IBM 360/91中,12位扫描的乘法部件的原理框图表示在图5.10中. 数据流在图中用箭头的指向来说明. 该硬件乘法部件更详细的情况将在第九章中作为一个实例研究.

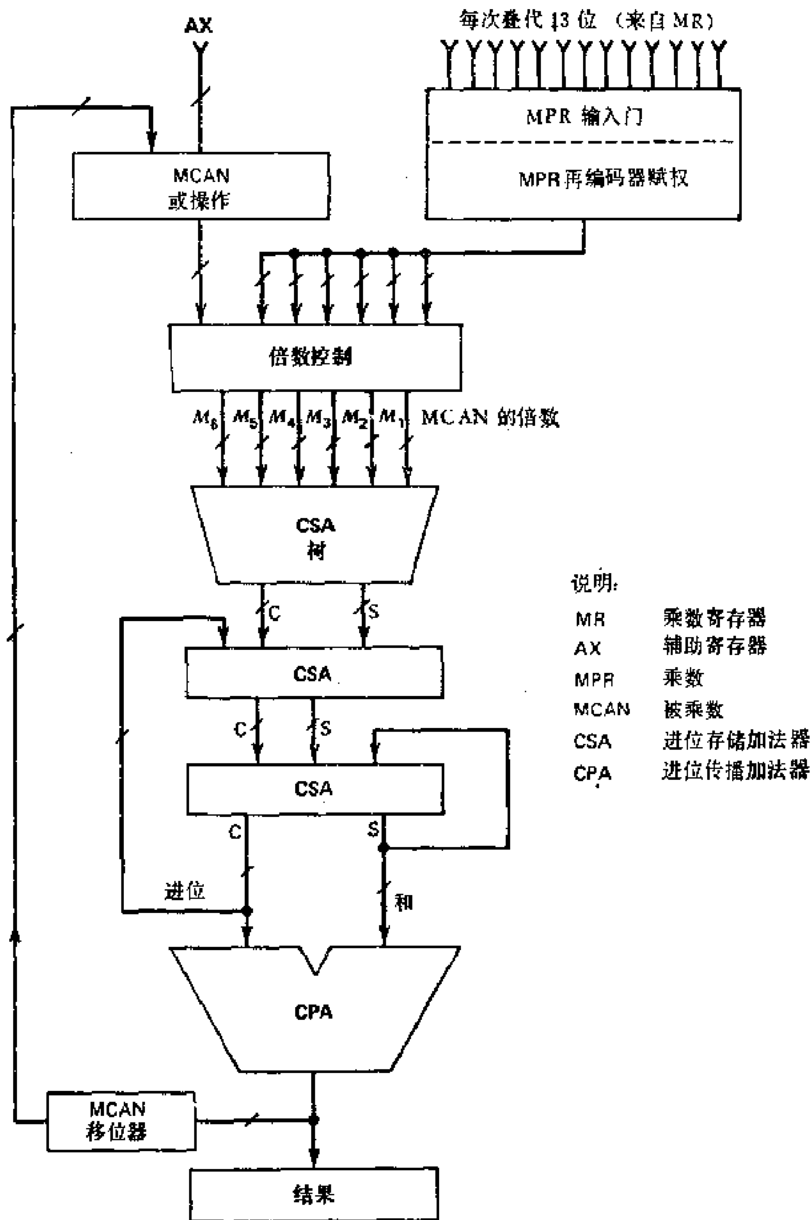


图5.10 在IBM 360 系统91型计算机中,具有重叠的12位扫描的56位乘56位再编码乘法器 (Anderson 等[1])

乘数的再编码是通过检查 **MR** 的后12位来实现的. 六种被乘数的倍数是由六组与门控制左移后的被乘数生成. 这里要求用按位对1求补的求补器来提供真值和补码的

乘法. CSA 树可以假定采用图 5.9 中任何一个设计,这依赖于所要求的速度和硬件的限制. 右移器应该能把带符号的补码数右移,并用扩充的符号去填满空出来的最左边的 12 个位置,因为部分乘积有时可以是补码形式表示的负数.

被乘数倍数的对准以及到相应的 CSA 树输入线去的反馈线是很重要的. 一开始,二个移位器的内容应该清除,因而初始部分乘积的反馈为零. 当乘数最低位是一个 1 时,补码表示的被乘数应该在最早的周期通过这些反馈线输入,因而可以执行相应的减法. 如果采用补码运算,那么在最后乘积求和期间,应该有一个初始进位进入 CPA 的最低位.

一个 m 位扫描的乘法器(这里 m 是偶数)可以完成二个 n 位数的乘法,用上述乘法器需要 k 次叠代,这里 $k=\lceil(n+4)m\rceil$,其中的 4 是考虑到用零增补了 4 个空出的低位. 用一个起始周期把数据输入到适当的寄存器中,用一个结束周期把部分和与部分进位向量相加,上述处理过程共需要 C 个周期来完成,这里

$$\mathbf{C}=\left\lceil\frac{n+4}{m}\right\rceil+2 \tag{5.22}$$

因为 CPA 要求一个固定的时间延迟,它通常小于 CSA 树的延迟,所以周期时间主要决定于通向CSA 树输出级的通路的延迟. 这可以包括乘数再编码器,倍数求补器和门的控制, CSA 的前三级,以及反馈回路的后二级加上移位器的延迟. 采用目前的集成电路器件,这些延迟加在一起可以小于 30 ns. 因此,用上述硬件乘法部件来完成二个 56 位的分数的乘法可以小于 $\{\lceil(56+4)/(12-1)\rceil+2\}\times 30\cong 225$ ns,这对于存储器周期在 1 μs 左右的大多数现代计算机来说是十分满意的.

5.7 典型的乘数再编码

在机器运算方面,最近的进展之一是采用冗余的 SD 编码来代替普通的乘数数位,这时将使越过乘数中的一串零的平均移位长度增加,从而减少乘法中的加法型操作.

典型的带符号-数位的编码

一个基数为 2 的 SD 数中,允许的数位集合为 $\{\bar{1}, 0, 1\}$. 在具有预定值 α 的 n 位数的所有可能的冗余表示法中,最使我们感兴趣的是最小的 SD 向量,如第 1.5 节所述,它具有最小的权. 对一个给定值 α,可以有不止一个 n 位的 SD 向量,它们都具有相等的最小的权. 例如,给定 $n=6$ 和 $\alpha=3$,就有八个不同的基数为 2 的 SD 向量,它们都具有数值 3. 这些向量的权列表如下. 在这八个向量中间,上面二个向量是具有权 2 的最小的向量.

一个最小的 SD 向量 $\mathbf{D}=D_{n-1}\cdots D_1D_0$ 如果不包含相邻的非零数位,则被称为典型的带符号-数位向量. 这意味着在典型的 SD 向量中,所有非零的数位都被一些零所隔开,这一点可以正式定义如下

$$D_i\times D_{i-1}=0 \quad 对\ 1\leqslant i\leqslant n-1 \tag{5.23}$$

Reitweisner[17] 已经证明: 对具有固定数值 α 和固定向量长度 n 的任意位数 D,假如在 D 中的两个最左边的数位的乘积不等于 1,即

$$D_{n-1}\times D_{n-2}\neq 1 \tag{5.24}$$

那么就存在一个“唯一的”标准 SD 形式，这里 $\mathbf{D}=D_{n-1}D_{n-2}\cdots D_0$. 只要在向量 $\mathbf{D}$ 的左端加一个附加数位 $D_n=0$，这个特性常常可以得到满足. 在下面的讨论中，我们将考虑领前的位为零的 $(n+1)$ 位 SD 向量，显然这并不丧失一般性.

带符号-数位向量	数　值	权
$(000011)_2$	$2+1=3$	2
$(00010\bar{1})_2$	$4-1=3$	2
$(001\bar{1}0\bar{1})_2$	$8-4-1=3$	3
$(01\bar{1}\bar{1}0\bar{1})_2$	$16-8-4-1=3$	4
$(1\bar{1}\bar{1}\bar{1}0\bar{1})_2$	$32-16-8-4-1=3$	5
$(001\bar{1}\bar{1}1)_2$	$8-4-2+1=3$	4
$(01\bar{1}\bar{1}\bar{1}1)_2$	$16-8-4-2+1=3$	5
$(1\bar{1}\bar{1}\bar{1}\bar{1}1)_2$	$32-16-8-4-2+1=3$	6

将一个普通的二进制向量转换成一个典型的 SD 向量的步骤说明如下.

典型的再编码算法

给定一个 $(n+1)$ 位二进制向量 $\mathbf{B}=B_nB_{n-1}\cdots B_1B_0$，这里 $B_n=0$，$B_i\in\{0,1\}$，$0\leqslant i\leqslant n-1$. 我们希望求出 $(n+1)$ 位典型的 SD 向量

$$\mathbf{D}=D_nD_{n-1}\cdots D_1D_0,$$

除了 $D_n=0$ 以外，具有 $D_i=\{\bar{1},0,1\}$，因而二个量 $\mathbf{D}$ 和 $\mathbf{B}$ 都代表同一个数值

$$\alpha=\sum_{i=0}^{n}B_i\times 2^i=\sum_{i=0}^{n}D_i\times 2^i \tag{5.25}$$

第一步：从 $\mathbf{B}$ 的低位端开始，令下标 $i=0$ 和初始进位 $C_0=0$.

第二步：以进位-输入 C_i 为条件来检查向量 $\mathbf{B}$ 的二个相邻位 B_{i+1} 和 B_i，并且按普通的二进制运算规则生成下一个进位 C_{i+1}，即只有当三个输入 B_{i+1}，B_i 和 C_i 中有二个或三个 1 时，才有 $C_{i+1}=1$.

第三步：按下列算术运算方程式来生成向量 $\mathbf{D}$ 的第 i 位 D_i

$$D_i=B_i+C_i-2C_{i+1} \tag{5.26}$$

第四步：下标 i 增 1，并检查是否 $i=n$. 如果不等，即转向第二步，否则就停止.

上述步骤归纳在表 5.5 中. 现在让我们用一个数字例子来解释一下，给定下列 9 位二进制向量

$$\mathbf{B}=(0\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 1)_2 \tag{5.27}$$

利用上述步骤，相应的典型 SD 向量可以求出如下：

$$\mathbf{D}=(0\ 1\ 0\ \bar{1}\ 0\ \bar{1}\ 0\ 0\ \bar{1})_{SD} \tag{5.28}$$

$\mathbf{D}$ 具有数值 $128-32-8-1=87$，等于 $\mathbf{B}$ 的数值. 注意 $\mathbf{D}$ 中所有的非零位（1 或 $\bar{1}$）都用零隔开了. 此外，向量 $\mathbf{D}$ 具有权 4，而向量 $\mathbf{B}$ 具有权 5. 实际上，数字数的普通二进制表示法可作为 SD 数的一种特例来考虑.

典型的再编码技术可以用在乘法器的设计中. 每个周期只要求执行对倍数 A 或 $-A$ 的加法. 在再编码乘数中，用来隔开非零位的所有的零将引起执行更多的“移位”. 这个方法可以推广到一次生成 2 个或更多个带符号的数位. 例如，根据对 $\mathbf{B}$ 的三个相邻位 B_{i+2}，B_{i+1}，B_i 和引入的进位 C_i 的检查，可以用一步来生成 $\mathbf{D}$ 的二个数位 D_i，D_{i+1} 以及下一个进位 C_{i+2}. 因为 $\mathbf{D}$ 必须是典型的，相邻数位 D_{i+1} 和 D_i 两者不能都是非零. 这就提出了下列五对 (D_{i+1}, D_i) 可能的选择

$$\{\bar{1}0,\ 0\bar{1},\ 00,\ 01,\ 10\} \tag{5.29}$$

这里排除了 11，$\bar{1}\bar{1}$，$\bar{1}1$ 和 $1\bar{1}$.

一次产生二个带符号数位的过程可以设想为一个基数 4 的 SD 向量，它具有下列数

位集合

$$\{\bar{2},\ \bar{1},\ 0,\ 1,\ 2\} \tag{5.30}$$

分别相应于方程式 5.29 中的五对.

基数为 4 的典型向量可以用于设计每个周期移二位的快速再编码乘法器，假设倍数 $0,\ \pm A,\ \pm 2A$ 均在运算处理器内生成. 采用冗余和高基数的方法已经用于 **ILLIAC** 计算机系列的运算部件设计.

表 5.5 基数 $r=2$ 的典型乘数再编码算法

普通的乘数位		假设的进位	再编码的位[1]	进位输出[1]
B_{i+1}	B_i	输入 C_i	D_i	C_{i+1}
0	0	0	0	0
0	1	0	1	0
1	0	0	0	0
1	1	0	$\bar{1}$	1
0	0	1	1	0
0	1	1	0	1
1	0	1	$\bar{1}$	1
1	1	1	0	1

1) 用方程式 5.26 求 D_i.

2) $C_{i+1}=B_{i+1}B_i+B_iC_i+B_{i+1}C_i$.

5.8 串再编码和 Booth 乘法器

另一种感兴趣的乘数再编码方案是基于第 5.5 节所描述的串特性. 我们把这种方法称为串再编码. 实际上、著名的 Booth 乘法器就是基于串再编码. 表 5.6 说明了与串再编码有关的一些规则. 输出数位 $D_i(0\leqslant i\leqslant n)$ 按下列鉴别规则求出

$$D_i=\begin{cases}0, & \text{如果 } B_i=B_{i-1}\\ 1, & \text{如果 } B_i<B_{i-1}\\ \bar{1}, & \text{如果 } B_i>B_{i-1}\end{cases} \tag{5.31}$$

表 5.6 中给出的注解相应于检测一串 1 的开始、中间和结束. 这个步骤需要在一个 n 位的二进制向量

$$\mathbf{B}=B_{n-1}\cdots B_1B_0$$

的二端附加二个空位 $B_{-1}=B_n=0$. 因为在这个步骤中没有进位产生，每个被变换的数位 D_i 只是输入串 $\mathbf{B}$ 的二个相邻位 B_i 和 B_{i-1} 的函数. 因此，通过同时检查向量 $\mathbf{B}$ 的所有相邻的对 $B_i,\ B_{i-1}$; $\mathbf{D}$ 中的所有数位可以并行地产生.

串再编码对于变换包含一长串 1 的二进制向量来说是很有用的. 然而，对于变换具有许多孤立的 1 的二进制向量则是无效的. 换句话说，非零数位在串再编码后可以出现在互相邻近的位置上，它会引起一个甚至比以前的权更高的 **SD** 向量. 这个现象可从下面的例子中看出

$$\text{二进制向量}\quad \mathbf{B}=(0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0)_2 \tag{5.32}$$

$$\text{变换后的串向量}\quad \mathbf{D}=(0\ 1\ \bar{1}\ 1\ \bar{1}\ 1\ \bar{1}\ 1\ \bar{1}\ 0)_{SD} \tag{5.33}$$

必须指出，在串再编码后，邻近的非零数位应具有相反的符号，即 $D_i \times D_{i+1} \neq 1$（对于所有的 i）。

为了将上述方法扩充到高基数，可以把再编码的串分成几段，每一段为二位(对)或三位．现在让我们来解释基数为 4 的串再编码，这里将再编码的 SD 向量分成许多间断的数位对．在 **D** 中的每一对 (D_{i+1}, D_i) 都可以看作一个基数为 4 的数位 F_i，于是便建立下列对应关系

$$\text{基数-2}(D_{i+1}, D_i):\ \bar{1}0,\ 0\bar{1},\ 00,\ 01,\ 10. \tag{5.34}$$

或 $\bar{1}1$　或 $1\bar{1}$

$$\text{基数-4}(F_i): \bar{2},\ \bar{1},\ 0,\ 1,\ 2 \tag{5.35}$$

表 5.6　基数为 2 的串乘数再编码算法

输入		输出	关于串特性的注解
B_i	B_{i-1}	D_i	
0	0	0	没有串
0	1	1	串的结束
1	0	$\bar{1}$	串的开始
1	1	0	串的中心

利用这个变换，我们可以设计一个基数为4、每步一律移 2 位的串再编码乘法器．必须指出，表 5.4 中所描述的乘法的重叠扫描法正好是一个基数为 4 的串再编码方案，这里每一步有三个输入数位 B_{i+1}，B_i，B_{i-1} 被扫描，有二个输出带符号数位 D_{i+1}，D_i 被生成．

基于串再编码理论的一个硬件乘法部件说明如下．该部件可以直接对二个补码数相乘，而不必关心这二个数的符号．换句话说，没有必要再进行算前求补和算后求补．这个方法就是熟知的 Booth 乘法．

在一对乘数位 B_i 和 B_{i-1} 被检测后，乘法部件将执行下列操作．

如果 $B_i = B_{i-1}$，则将部分乘积右移

如果 $B_i < B_{i-1}$，则将 A 加到部分乘积上，然后右移　　(5.36)

如果 $B_i > B_{i-1}$，则从部分乘积中减去 A，然后右移

移位可以设想成有一个零向量空加到部分乘积上．减法实际上是用补码加法来实现的．乘数各位的所有各对能同时比较．该设计表示在图 5.11 中．每叠代一次，要扫描乘数位的三个重叠的对（4 位）．相继的二次叠代也重叠一位．采用这种方法的 32 位 Booth 乘法器的扫描模式如图 5.12 所示．一次扫描三个乘数对意味着同时把三个被乘数的倍数加到部分乘积上．因此，一个五输入的 **CSA** 树被用于承担多操作数加法的任务．

最初，一个零部分乘积被送入 **AC**，被乘数 **A** 送入 **AX**．n 位的乘数 B 和一个附加的零送入 $(n+1)$ 位的 **MR**．每个加法周期要检测寄存器 **MR** 最右边的四位 MR_2，MR_1，MR_0 和 MR_{-1}，以便确定必要的操作．门逻辑用来选择适当的输入送往 **CSA** 树．该乘法部件能通过 $\lceil (n+1)/3 \rceil$ 次叠代完成二个 n 位数的乘法．假定每次叠代一律右移 3 位．以 $n = 32$ 为例，只需要 $\lceil (n+1)/3 \rceil = 11$ 个机器周期．在工作速度和硬件复杂性之间存在着一个折衷方案．最简单的设计是一次扫描一对乘数位，这时完成乘法需要 n 个周期．

上面的乘法部件如果备有左移的话，还能够从左端启动．事实上，所有乘数对能同时进行检测，以便产生出所需要的 SD 串代码．Booth 算法也能用快速叠接逻辑阵列来实现，这种逻辑阵列可以由“加法-减法-移位单元”组合而成，这留待第六章中加以研究．

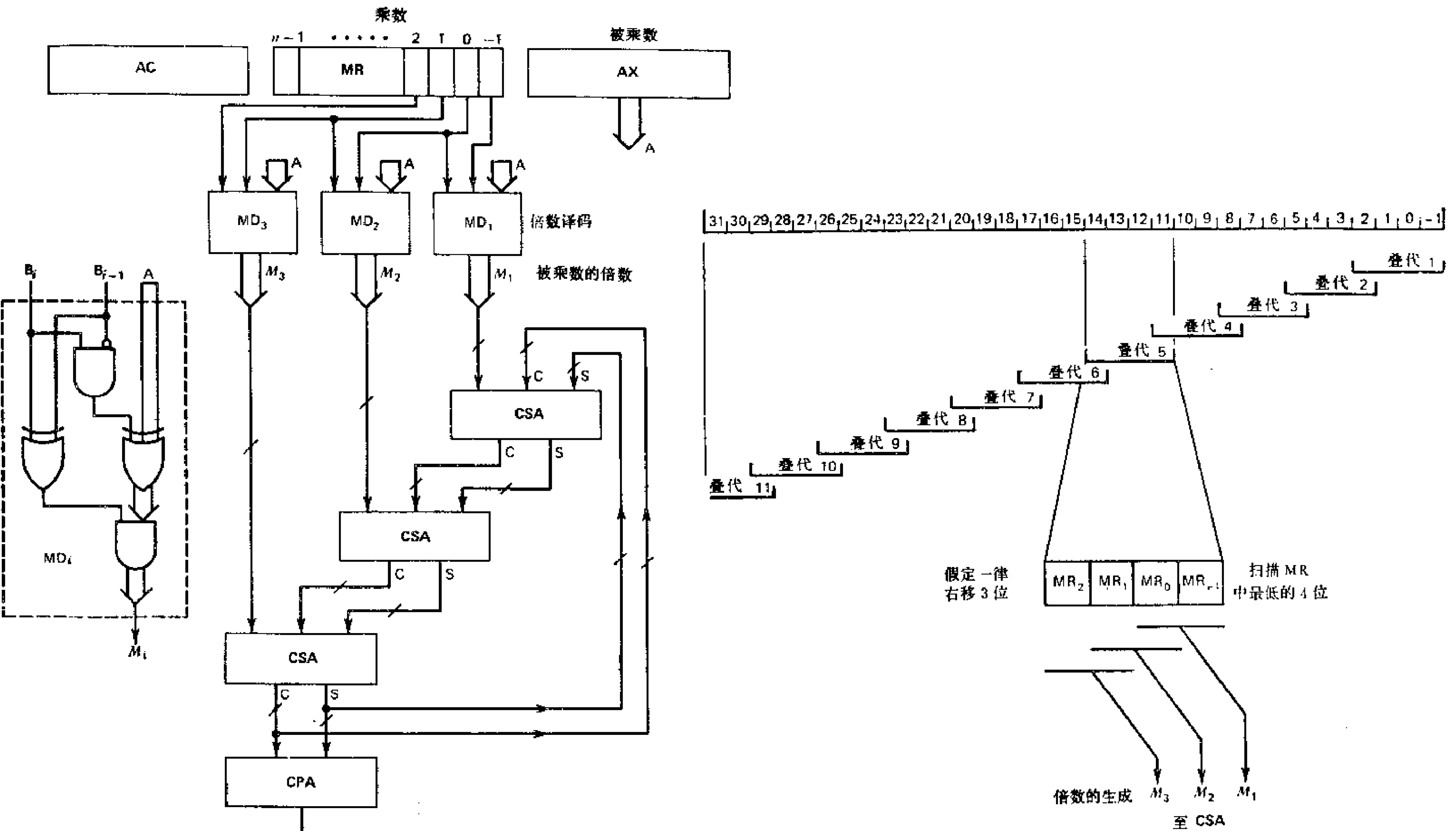


图 5.11　对一个 4 位扫描的一般化的 Booth 乘法器，为了说明乘数的译码、倍数的产生以及利用 **CSA/CPA** 作加法所用的原理逻辑图

图 5.12　用图 5.11 所示进位存储加法器来实现的 32 位一般化的 Booth 乘法器的 4 位扫描模式

5.9 各种加法-移位乘法器的评价

机器设计师常常要求较好地衡量运算处理器的效率．一种运算方法的有效性应该取决于对所有可能的输入操作数进行平均的衡量．乘法器效率的两种通常采用的度量标准是:

1. 在一次乘法中所需要的加法型操作的平均次数．

2. 在一次乘法中，相继二次加法之间的平均移位长度．

我们根据这二点来比较各种乘法器设计的性能以前，让我们把这些乘法器设计方案归纳成五类．

(1) 加法器旁路

这指的是在乘法器中越过一串零的移位．由于要求不均匀的移位，所以这种方法主要用于异步计算机中．缺点是增加了控制的复杂性．

(2) 减少加法时间

无论是用 **CLA** 技术来产生并行进位，还是用进位-存储的方法，都能减小基本的加法时间．采用 **SD** 运算可能实现全并行的加法器．

(3) 一次移几位

每次叠代必须准备移几个数位．多位译码会引起硬件的增加．最佳的选择则是围绕着这样的中心，即在速度和实际经济因素之间取折衷．

(4) 乘数再编码

通过再编码把冗余度引入乘数中，可以不增加线路的复杂性，这是因为产生被乘数的倍数 $-A$ 或 $+A$ 是比较简单的．再编码技术势必引起较长的平均移位长度．

表 5.7 标准的和再编码的乘法器方案的比较 (Garner[9])

类　型	乘 法 器 结 构	加法平均次数	平均移位长度
标准的乘法器	基本的加法-移位	n	1
	跳零加法	$n/2$	1
	具有移单/双位的跳零加法	$n/2$	$1\frac{1}{2}$
	具有移 1,2,3 位的跳零加法	$n/2$	$1\frac{3}{4}$
	具有所有各种可能的移位的跳零加法	$n/2$	2
再编码的乘法器	具有移 1 和 2 位的跳零加法	$n/3$	$1\frac{3}{4}$
	具有所有各种可能的移位的跳零加法	$n/3$	3
	只移双位(基数－4)	$n/3$	2
	只移三位(基数－8)	$<n/3$	3

(5) 高基数乘法

采用高基数乘法时，乘法中所需要的叠代次数可减小 k 倍，$k=\log_2 r$，这里 $r=2^k$ 是数的基数．在计算机商品中很少采用高基数，这主要是因为硬件复杂性增加不少．

与各种乘法方案有关的加法平均次数与平均移位长度已在表 5.7 中作了比较．最简单的设计是需要 n 次具有单位平均移位长度的加法．对各乘数中所有可能的移位进行平均，则移位长度的平均值将以 2 代替 1．采用乘数再编码时，平均移位长度有可能达到 3，

这是因为生成了更多个零，而所有非零的数位都是稀疏的．随着移位长度的增加，加法型操作的次数平均能减少到一半甚至 1/3，如表中所示．例如，采用基数 8 的具有平均移位长度 3 的再编码乘法器，为完成二个 n 位数的乘法所需要的加法次数可以小于 $n/3$．

在近代数字计算机中，乘法部件所需要的半导体元件数大致占总数的 10%，但价格却低于计算机总价格的 10%．因此，希望让一些其他的运算，如象平方、除法等来共享与乘法有关的硬件．一般地说，设备价格是随着字长的平方而上升的，而时间则是随着字长的对数而增长的．目前，定点机的乘法时间大约是机器的加法时间的四倍．但是对浮点运算来说，这个比值接近于 1．在科学用的计算机中，全部算术运算操作中大约有 1/3 是乘法．因此，高速乘法部件的采用将会使大多数计算机的计算速度大致加快一倍．

5.10 参考文献注释

普通的计算机都采用标准的加法-移位技术来实现乘法．Mowle[15]曾经对这些基本的乘法算法和它们的逻辑设计问题作了详细的描述．一个通用乘法器的设计阐明了典型运算设计的具体步骤，即从算法流程图开始到微操作表，然后再具体地加以实现．直接的补码乘法线路是由 Booth[4]、Robertson[18] 和 Kamal[10] 等提出的．1的补码运算由 Metze[14] 作了专门的研究．乘法中一次移多位的办法曾有许多作者讨论过，其中要算 MacSorley 的处理[13]最有启发性．他阐明了在实现带有 **CSA** 回路的乘法器时，采用均匀移位（每个周期移 2 或 3 位）或可变移位的设计特征．对于采用每次均匀移 2，3 或 4 位的计算机，Freeman[8] 还给出了平均移位长度的分析．

Robertson[20] 提议用冗余度来改进乘法器的效率，后来在 **ILLIAC III** 计算机中得到实现，这是由 Atkins[2] 报道的．Ling[12] 提出了一个用于快速乘法的多位译码线路．他的方法是利用一组简单的数学恒等式把乘法分解成少量的加法和减法．Chen[6] 提出的二进制乘法是基于平方原理．重叠的多位扫描的方法在参考文献[1]、[5]、[7]、[11]、[13]、[16] 和 [20] 中均有研究．Garner[9] 曾经提出一种环模型，用来研究带符号的补码乘法．Winograd[23] 研究了乘法所需时间在理论上的限制．

乘数再编码方法的起源尚不清楚．典型的再编码曾由 Reitwiesner[17] 作了研究，他证明了在平均的意义上没有别的方法能够比典型的 **SD** 再编码更有效．但这并不排除存在同样有效的办法．Pennhollow[16]发现了一类具有平均移位长度 3 的再编码，它的典型方案是很简单的．由 Wallace[22]提出的乘数再编码是利用 **CSA** 来实现乘法器的．Avizienis[3] 和 Robertson[20]已经对各种乘数再编码方案作了综合处理．串再编码曾被 Booth[4] 所应用．Robertson[19] 建立了数字除法和乘数再编码步骤之间对应关系．最近，Trivedi 和 Ercegovac[21] 发表过一些用于快速乘法和除法的联机算法．

参　考　文　献

[1] Anderson, S. F. et al. "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal*, Jan. 1967, pp. 34—53.

[2] Atkins, D. E., "Design of the Arithmetic Units of Illiac III: Use of Redundancy and Higher Radix Methods," *IEEE Trans. Comput.*, Vol. C-19, No. 8, Aug. 1970, pp. 720—733.

[3] Avizienis, A., "Recoding of the Multiplier," *Class Notes*, Engr. 225A, UCLA, Los Angeles, CA.,

1971.
[4] Booth, A. D., "A Signed Binary Multiplication Technique." *Quart. Journ. Mech. and Appl. Math.*, Vol. 4, Part 2, 1951, pp. 236—240.
[5] Bursten, V. S., "Accelerating Multiplication and Division Operations in High-Speed Digital Computers," *Tech. Report*, Institute of Exact Mechanics and Computing Technique, The Academy of the USSR, Moscow, 1958.
[6] Chen, T. C., "A Binary Multiplication Scheme Based on Squaring," *IEEE Trans. Comput.*, Vol. C-20, No. 6, June 1971, pp. 678—680.
[7] Fenwick, P. M., "Binary Multiplication with Overlapped Addition Cycles," IEEE *Trans. Comput.*, Vol. C-18, No. 1, Jan. 1969, pp. 71—74.
[8] Freeman, H., "Calculation of Mean Shift for a Binary Multiplier Using 2, 3, or 4-bit at a Time," *IEEE Trans.*, Vol. EC-16, No. 6, Dec. 1967, pp. 864—866.
[9] Garner, H. L., "A Ring Model for the Study of Multiplication for Complement Codes," *IRE Trans.*, Vol. EC-8, No. 1, March 1959, pp. 25—30.
[10] Kamal, A. A., and Ghanam, M., "High-Speed Multiplication Systems," *IEEE Trans., Comput.*, Vol. C-21, No. 9, Sept. 1972, pp. 1017—1021.
[11] Lehman, M., "Short-Cut Multiplication and Division in Automatic Binary Digital Computers," *Proc. IEEE*, Vol. 10, Sept. 1958, pp. 496—504.
[12] Ling, H., "High-Speed Computer Multiplication Using a Multiple-Bit Decoding Algorithm," *IEEE Trans. Comput.*, Vol. C-19, No. 8, Aug. 1970, pp. 706—709.
[13] MacSorley, O. L., "High-Speed Arithmetic in Binary Computers," *Proc. of IRE*, Vol. 49, Jan. 1961, pp. 91—103.
[14] Metze, G., "A Study of Parallel One's Complement Arithmetic Units with Separate Carry or Borrow Storage," *Ph. D. Thesis*, Univ. of Illinois, Urbana, Ill., 1958.
[15] Mowle, F. J., "Simplified Logic Deisgn Using Digital Circuit Elements," Vol. 3, *Class Notes*, Dept. of Elec. Eng. Purdue University, Sept. 1974, Chap. 14.
[16] Pennhollow, J. O., "Study of Arithmetic Recoding with Applications in Multiplication and Division," *Ph. D. Thesis*, Univ. of Illinois, Urbana, Ill., Sept. 1962.
[17] Reitwiener, G. W., "Binary Arithmetic," in *Advances in Computers*, Vol. 1, Academic Press, N. Y. 1960, pp. 261—265.
[18] Robertson, J. E., "Two's Complement Multiplication in Binary Parallel Digital Computers," *IRE Trans.*, Vol. EC-4, No. 3, Sept. 1955, pp. 118—119.
[19] Robertson, J. E., "The Correspondence Between Methods of Digital Division and Multiplier Recoding Procedures," *IEEE Trans. Comput*, Vol. C-19, No. 8, Aug. 1970, pp. 692—701.
[20] Robertson, J. E., "Increasing the Efficiency of Digital Computer Arithmetic Through Use of Redundance," *Class Notes*, Univ. of Illinois, Urbana, Illinois.
[21] Trivedi, K. S. and Ercegovac, M. D., "On-Line Algorithms for Division and Multiplication," *IEEE Trans. on Computers*, Vol. C-26, No. 7, July 1977, pp. 681—687.
[22] Wallace, C. S., "A Suggestion for a Fast Multiplier," *IEEE Trans. on Electronic Computers*, Vol. EC-14, No. 1, Feb. 1964, pp. 14—17.
[23] Winograd, S., "On the Time Required to Perform Multiplication," *Journal of ACM*, Vol. 14, No. 4, Oct. 1967, pp. 793—802.

习　　题

题5.1 考虑图5.5中的通用n位乘n位乘法器，它能以四种不同的方式操作，字长$n=16$. 试分别给四种操作方式 US, OC, TC 和 SM 的每一种，列出所有机器周期中相继出现的寄存器内容 (**AC**)、(**MR**)、(**AX**)，以及相继出现的加法器输出 $S_{n-1}\cdots S_1S_0$. 从主存储器取出的二个操作数具有下列二进制形式

$$被乘数=(0101101101111010)_2\quad 乘数=(1010111100000101)_2$$

注意：这些二进制模式在不同的定点表示法中可以代表不同的操作数数值.

题5.2 根据表5.2所给定的微操作，列出通用乘法器所有14条控制线的方程式，并用最小的3级与非电路加以实现. 估算出图5.5的电路中最长延迟通路的门的级数，这里假定寄存器具有8△延迟，加法器具有12△延迟，门的延迟参照表2.1. 若令单位门延迟△的平均值为4ns. 试确定可以用来驱

动通用乘法器的最大时钟频率（机器周期时间）。

题5.3 试说明每个机器周期均匀移5位的40位乘40位快速乘法器的原理线路图，这里假定采用非重叠的5位扫描系统。允许使用多级进位-存储加法器。假设每级进位-存储加法器带来2△延迟，其中△ = 4ns，试估算这个多位扫描乘法器的机器周期时间。

题5.4 对一个重叠的4位扫描乘法系统，其中每个周期要扫描当前的乘数位 $X_{i+2}X_{i+1}X_i$ 的位组（三位一组）再加上下一个高次位组的低位 X_{i+3}，试列出类似于表5.4的被乘数倍数表。根据串特性来解释对16种可能的4位乘数模式的扫描。

题5.5 证明：对具有给定值 α 和给定字长 n 的数，存在着唯一的典型带符号数位的形式，这里假定满足方程式5.24（Reitweisner[137]）。

题5.6 给定一个16位的二进制数 **B** =（0 0 1 1 0 1 0 1 1 1 1 1 0 1 0 1），试求出相应的典型带符号数位向量 **D**，它代表着和 **B** 相同的数值。并说明 **D** 中每个再编码带符号数位是怎样一步一步生成的。

题5.7 将表5.5扩充到能一次产生二个典型的带符号数位（基数4）。在这个扩充的典型再编码表中，列的标题应该是普通的二进制数位 B_{i+2}，B_{i+1}，B_i；假定 C_i 为进位输入；D_{i+1}，D_i 为再编码带符号数位；以及 C_{i+2} 为进位输出。注意：这个扩充的基数-4的典型再编码表具有四个输入位，因而有16行组合。

题5.8 试用5.8节中叙述的串再编码技术，列出基数-8的串再编码表（类似于表5.6）。将所得到的表与题5.4中得到的表相比较。证实这二个表应该是等效的。

题5.9 图5.11所示的Booth串再编码乘法器的设计是基于4位扫描系统。试对这个设计加以修改，改成每次叠代扫描七对重叠的乘数位（8位）。试解释这个一般化的串再编码乘法器的8位扫描模式（类似于图5.12）。用多级进位-存储加法器给这个修改的设计提供一个原理逻辑图。

第六章　叠接单元的阵列乘法器

6.1　阵列乘法的基础

如前章所述，硬件乘法器的常规设计是采用串行移位和并行加法相结合的方法，这种方法并不需要很多器件，在用微程序控制时尤其是这样．然而“加法-移位”的方法毕竟太慢，它不能满足目前的科学和工程对高速乘法所提出的要求．自从大规模集成电路出现以来，高速单元阵列乘法器已经成为对串-并行设计的一个可行的合乎逻辑的改进．在过去十年左右的时间内，已经提出了各种各样的叠接阵列乘法线路．正如所预期的那样，它们在速度上的提高是通过增加硬件的投资而得到的．

本章将引出带符号的和不带符号的阵列乘法器，以及它们用模块网络实现的方法．介绍几种 2 的补码数直接相乘的方法．重点阐明一种通用的乘法阵列和网络．其他一些阵列乘法线路，如像采用可控加法/减法单元、ROM-加法器网络和对数变换等也在这一章中给予讨论．

考虑两个不带符号的二进制整数 $\mathbf{A} = a_{m-1}\cdots a_1a_0$ 和 $\mathbf{B} = b_{n-1}\cdots b_1b_0$，它们的数值分别为 A_v 和 B_v，则

$$A_v = \sum_{i=0}^{m-1} a_i 2^i \qquad B_v = \sum_{i=0}^{n-1} b_i 2^i \tag{6.1}$$

在二进制乘法中，被乘数 $\mathbf{A}$ 与乘数 $\mathbf{B}$ 相乘产生 $(m+n)$ 位乘积 $\mathbf{P} = P_{m+n-1}\cdots P_1P_0$．乘积 $\mathbf{P}$ 的数值为

$$P_v = A_vB_v = \left(\sum_{i=0}^{m-1} a_i 2^i\right)\left(\sum_{j=0}^{n-1} b_j 2^j\right) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(a_ib_j)2^{i+j} = \sum_{k=0}^{m+n-1} P_k 2^k \tag{6.2}$$

图 6.1 中指出了实现这个乘法过程所需要的操作．每一个部分乘积项 a_ib_j 叫做一个被加数．这 $m \times n$ 个被加数 $\{a_ib_j \mid 0 \leqslant i \leqslant m-1$ 和 $0 \leqslant j \leqslant n-1\}$ 可以用 $m \times n$ 个与门并行地产生（图 6.2）．所以设计高速并行乘法器的基本问题在于缩短被加数矩阵中每一列所包含的 1 的加法时间．下面给出三种不同的线路来实现被加数求和的过程．

$$\begin{array}{rcccccccccccl}
 & & & & a_{m-1} & a_{m-2} & & \cdots & & a_1 & a_0 & = \mathbf{A} \\
 & & & \times) & & & & b_{n-1} & \cdots & b_1 & b_0 & = \mathbf{B} \\
\hline
 & & & & a_{m-1}b_0 & a_{m-2}b_0 & & \cdots & & a_1b_0 & a_0b_0 & \\
 & & & a_{m-1}b_1 & a_{m-2}b_1 & & \cdots & & a_1b_1 & a_0b_1 & & \\
 & & \cdot^{\cdot^{\cdot}} & & & & & \cdot^{\cdot^{\cdot}} & \cdot^{\cdot^{\cdot}} & & & \\
+) & a_{m-1}b_{n-1} & a_{m-2}b_{n-1} & & \cdots & & a_1b_{n-1} & a_0b_{n-1} & & & & \\
\hline
P_{m+n-1} & P_{m+n-2} & P_{m+n-3} & & \cdots & & & P_{n-1} & \cdots & P_1 & P_0 & = \mathbf{P} \\
 & & & & & & & & & & & \mathbf{P} = \mathbf{A} \times \mathbf{B}
\end{array}$$

图 6.1　在 m 位乘 n 位不带符号整数的阵列乘法中，用来说明加法-移位操作的被加数矩阵

我们选择 5 位乘 5 位不带符号的阵列乘法器 $(m = n = 5)$ 的设计来阐明并行阵列

乘法的原理．所有被加数项的排列如同图 6.1 中的矩阵． 5 位乘 5 位的 Braun 阵列乘法器的电路图如图 6.3 所示． 这种乘法器要实现 n 位乘 n 位时，就需要 $n(n-1)$ 个全加器和 n^2 个与门．该乘法部件总的乘法时间可以估算如下：

令 Δ_a，Δ_x 和 Δ_f 分别为与门（AND）、异或门（XOR）和全加器（FA）的传输延迟时间．从表 2.1 可见，若假定用 2 级与非逻辑或者与或非接线逻辑来实现这些基本功能，那么我们就有 $\Delta_x = 3\Delta$ 和 $\Delta_a = \Delta_f = 2\Delta$．找出最坏情况下的延迟途径，即沿着矩阵最右边的对角线和最下面的一行，便得到 n 位乘 n 位不带符号的（US）阵列乘法器的总的乘法时间．

$$
\begin{aligned}
\mathbf{\Delta}_{US} &= \Delta_a + [(n-1) + (n-1)] \times \Delta_f \\
&= 2\Delta + (2n-2) \times 2\Delta \\
&= (4n-2)\Delta \qquad (6.3)
\end{aligned}
$$

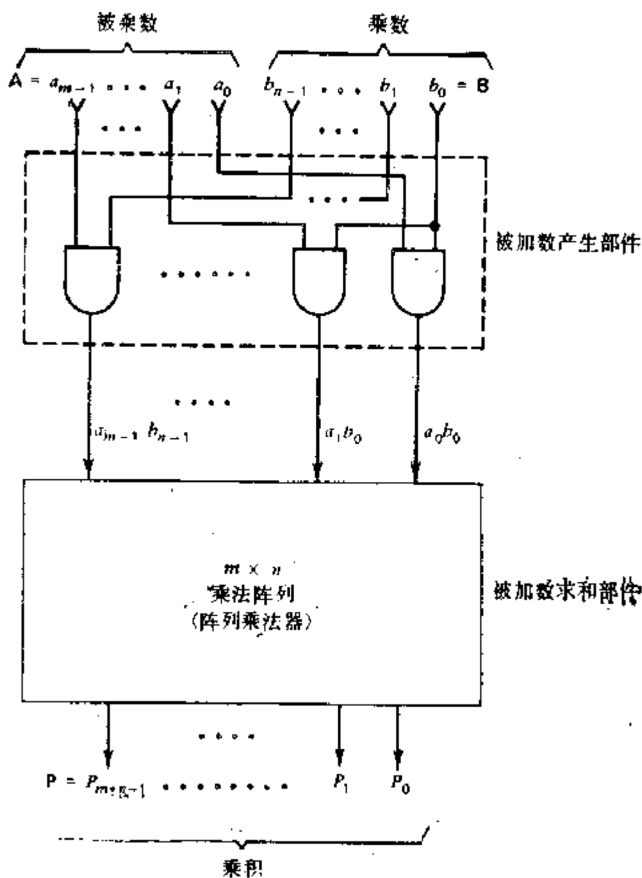


图 6.2　m 位乘 n 位不带符号的阵列乘法器

用虚线围住的阵列中最后一行构成了一个时间延迟为 $(n-1)2\Delta$ 的行波-进位加法器．这一行可用具有固定时间延迟的先行进位加法器（CLA）来代替，如假定全部进位均为先行的时间延迟为 8Δ．利用这种改进措施，总的乘法时间可减为

$$\mathbf{\Delta}_{US}(CLA) = \Delta_a + (n-1)\Delta_f + 8\Delta = 2\Delta + (n-1)2\Delta + 8\Delta = (2n+8)\Delta \quad (6.4)$$

带符号阵列乘法器的结构随所用的数的表示法而不同． 我们将说明如何把第 2.4 节所述的求补器插入到不带符号的阵列乘法器中，用来实现带符号数值的乘法、反码乘法、以及补码乘法．图 6.4 指出：算前求补器用于将两个操作数在被不带符号的乘法阵列（核心部件）相乘以前先变成正整数；算后求补器则在二个输入操作数的符号不一致时，把结果变换成带符号的数．

包含这些求补级的乘法器被称为符号求补的阵列乘法器．令 $\mathbf{A} = a_n a_{n-1} \cdots a_1 a_0$ 和 $\mathbf{B} = b_n b_{n-1} \cdots b_1 b_0$ 均为用定点表示的 $(n+1)$ 位带符号整数．在必要的求补操作以后，$\mathbf{A}$ 和 $\mathbf{B}$ 的真值输送给 n 位乘 n 位不带符号的阵列乘法器，并由此产生 $2n$ 位真值乘积 $\mathbf{A} \times \mathbf{B} = \mathbf{P} = p_{2n-1} \cdots p_1 p_0$，其符号位为 $p_{2n} = a_n \oplus b_n$． 在带符号数值的乘法中，算前求补和算后求补都不需要，因为这些真值都是立即可用的．在 1 的补码乘法中，求补器只用异或门来组成（图 2.11 a），每一个的时间延迟为 3Δ．在 2 的补码乘法中，假定三个求补器中每一个都是图 2.11 b 中的电路，则时间延迟为

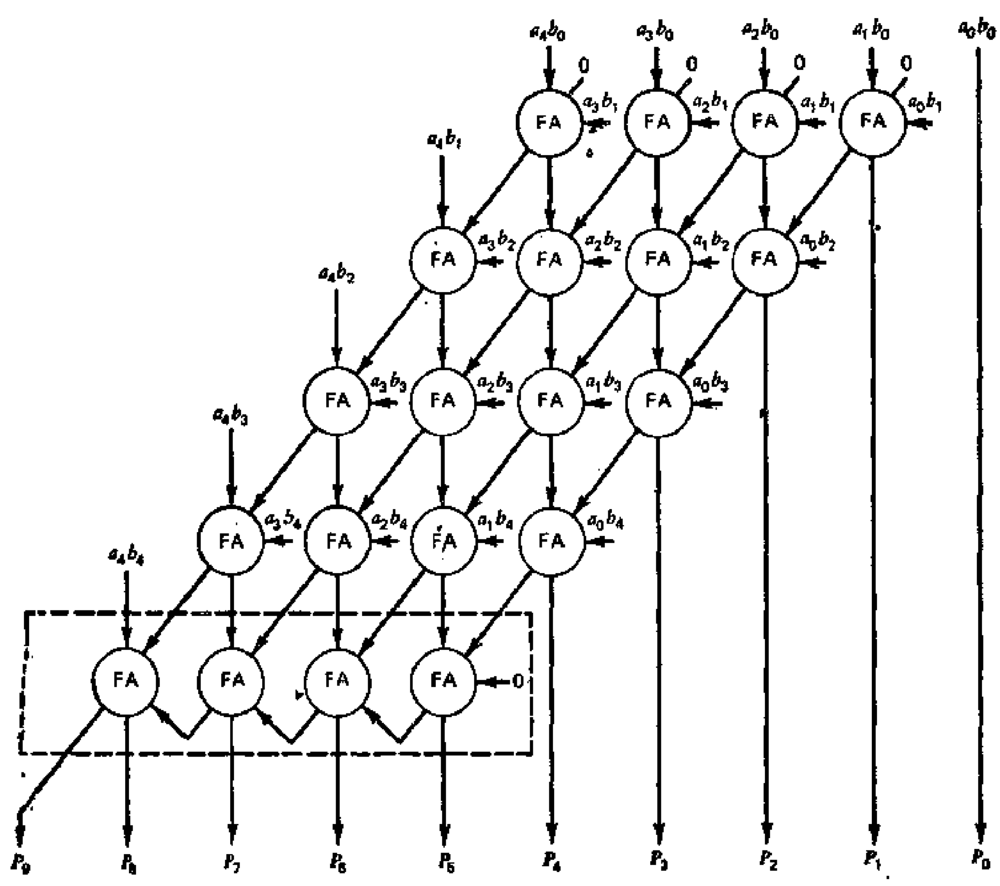
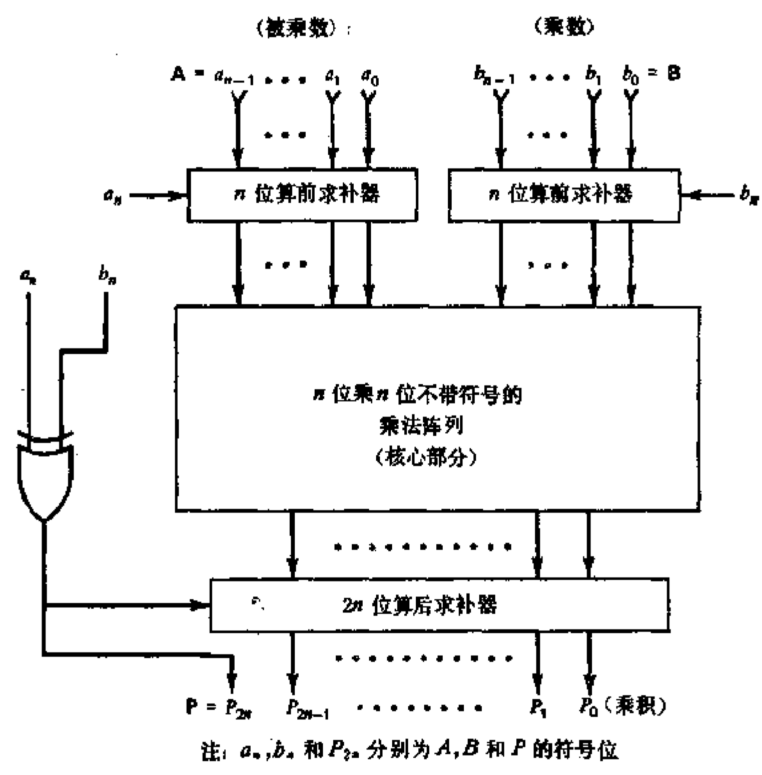


图 6.3　5 位乘 5 位不带符号的阵列乘法器的原理电路图(Braun[57])



注：a_n,b_n 和 P_{2n} 分别为 A,B 和 P 的符号位

图 6.4　$(n+1)$ 位乘 $(n+1)$ 位带求补级的阵列乘法器方框图

$$\Delta(n\text{位求补器}) = (n-1)\times 2\Delta + 5\Delta = (2n+3)\Delta \tag{6.5}$$

令 Δ_{SM}, Δ_{OC} 和 Δ_{TC} 分别为阵列乘法器对带符号数值，1 的补码和 2 的补码的总时间延迟（按图 6.4 中的规定）．我们有

$$\Delta_{SM} = (4n-2)\Delta \tag{6.6}$$

$$\Delta_{OC} = 3\Delta + (4n-2)\Delta + 3\Delta = (4n+4)\Delta \tag{6.7}$$

$$\Delta_{TC} = 2(2n+3)\Delta + (4n-2)\Delta = (8n+4)\Delta \tag{6.8}$$

$(n+1)$ 位乘 $(n+1)$ 位不带符号和带符号数值、1 的补码和 2 的补码等阵列乘法器所需要的硬件数量和时间延迟均已归纳在表 6.1 中．从这个表中我们可以得出一个结论：带符号的数值以及 1 的补码表示法两者均适合于高速的带求补级的阵列乘法，而间接的 2 的补码阵列乘法所需要增加的硬件太多，并且为了完成所必需的求补和乘法操作，时间大约长了一倍．克服这些困难的补救办法将在后面几节中给出．

表 6.1　基本乘法阵列所需硬件的元件数和时间延迟

硬件与速度	表示法			
	带符号的数值	1 的补码	2 的补码	不带符号的数值
异或门	1	$4n+1$	$4n+1$	0
与门	n^2	n^2	n^2+4n	$(n+1)^2$
或门	0	0	$4n-3$	0
全加器	$n(n-1)$	$n(n-1)$	$n(n-1)$	$n(n+1)$
总时间延迟	$(4n-2)\Delta$	$(4n+4)\Delta$	$(8n+4)\Delta$	$(4n+2)\Delta$

注：假定每个操作数都有 $(n+1)$ 位．

6.2　模块乘法与华莱士（Wallace）树

这一节阐明用模块网络实现大规模乘法阵列的方法，这里用到了第 2.5 节中引出的基本乘法模块．首先提出一种进位存储加法器的位片方案，它类似于第 4.4 节中用过的列加法器．这些位片加法器即所谓华莱士树，它们将用于对非相加乘法模块（NMM）所

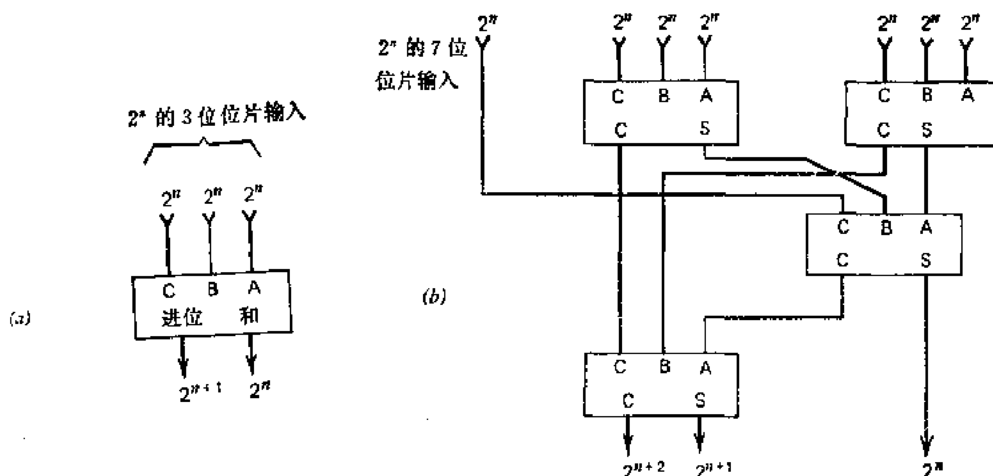


图 6.5　用进位存储全加器组成的位片式华莱士树．
(a) 3 位位片华莱士树 W_3，(b) 7 位位片华莱士树 W_7

独立产生的子乘积进行求和.

一个 k 输入的华莱士树是一个位片求和电路,它能产生 k 位位片输入的和. 图 6.5 给出了二个华莱士树,其中一个具有 3 个输入端,另一个有 7 个输入端. 3 输入的华莱士树只不过是一个 3 至 2 的进位-存储全加器, 它有二个输出,用来代表 3 个输入二进位的二进制和. 7 输入的华莱士树可以对 7 位的位片输入相加,并得出一个 3 位的和. Texas 仪器公司已经用 TTL 大规模集成电路生产了 7 位的华莱士树位片, 典型时间延迟为 45ns. 图中已表示出五个 3 至 2 的进位-存储全加器可以用来构成一个 7 输入的华莱士树. 对于较大的华莱士树则可用更多级进位-存储全加器位片 (CSA) 来实现.

4 位乘 4 位的 NMM 被用于产生局部的部分乘积,即所谓每 8 位一组的子乘积. 为

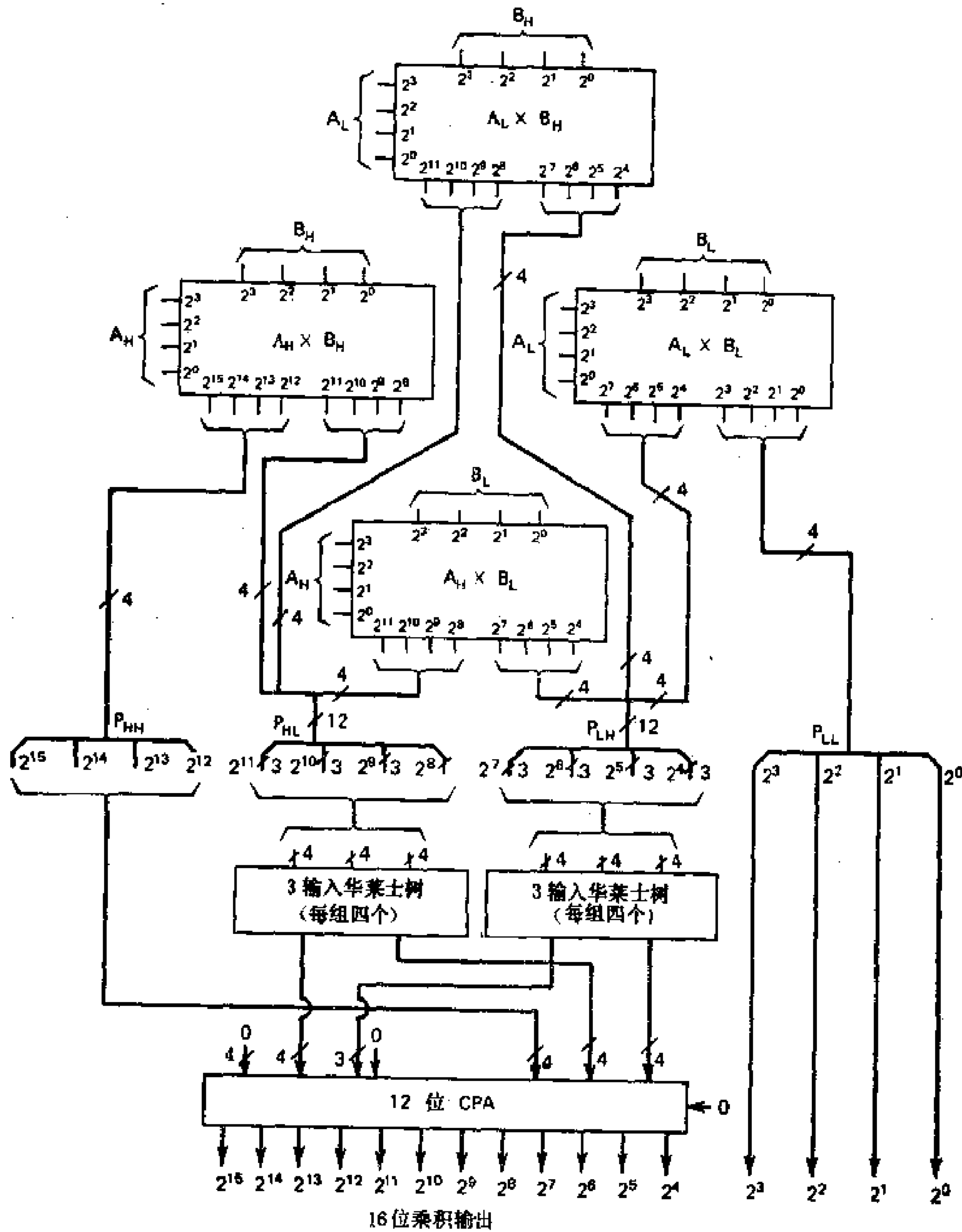


图 6.6 利用 4 位乘 4 位的非相加乘法模块 (NMM)、华莱士树与进位传播加法器 (CPA) 组成一个 8 位乘 8 位的阵列乘法器

了同时产生所有的子乘积，输入操作数（被乘数和乘数）必须加以分解使能进入 4 输入的位片．让我们考虑用四个 4 位乘 4 位的 NMM 来构成一个 8 位乘 8 位的阵列乘法器．16 位乘积可以写成

$$\begin{aligned}\mathbf{P}=\mathbf{A}\times\mathbf{B}&=(A_H\cdot A_L)\times(B_H\cdot B_L)\\&=A_H\times B_H+A_H\times B_L+A_L\times B_H+A_L\times B_L\\&=P_{HH}+P_{HL}+P_{LH}+P_{LL}\end{aligned}\tag{6.9}$$

这里

$$\mathbf{A}=A_H\cdot A_L\quad \mathbf{B}=B_H\cdot B_L\tag{6.10}$$

下标H和L用来标明每个 8 位数中的"高" 4 位和"低" 4 位．点"·"指的是高低位之间的连接．最后乘积是四个 8 位子乘积的和，这四个子乘积各自分别移了 4 位．

这四个子乘积的产生示于图 6.6 的上部．这些子乘积的求和则是由网络中部所用的 3 输入华莱士树来实现的．列的对准对于保证两个向量形式的结果的正确性是很重要的，这两个向量是指和向量与进位向量．网络的下部是一个 12 位的进位传播加法器（CPA），它把两个向量合并成最后的乘积．注意：相应于子乘积 $P_{LL}=A_L\times B_L$ 的几个最低有效

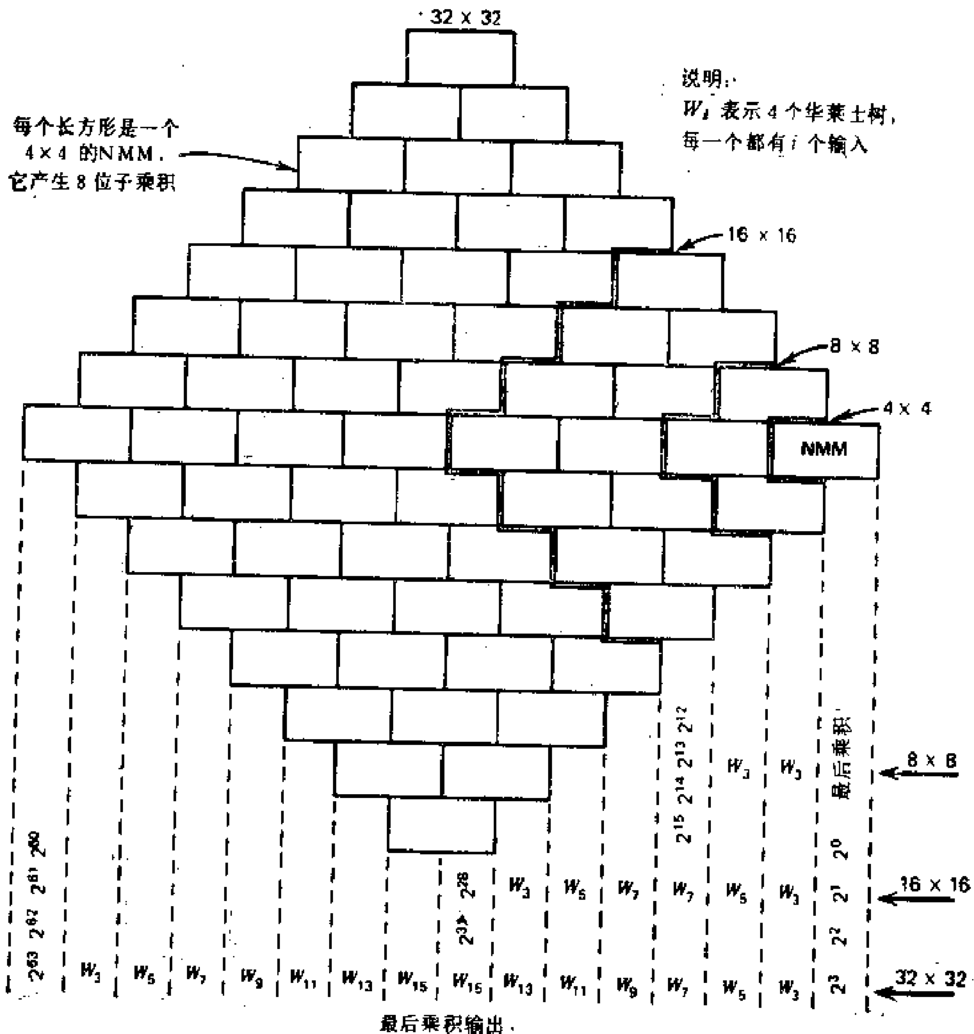


图 6.7 从 4 位乘 4 位到 32 位乘 32 位的各种乘法器的阵列布置

位不必通过 CPA.

在图 6.7 中，我们阐明了规模从 4 位乘 4 位到 32 位乘 32 位的各种阵列乘法网络的模块排列. 每个长方形代表一个 8 位的子乘积，它被分成“高”、“低”两个 4 位的位片. 所有这些位片在列的方向由奇数输入的华莱士树来叠加. 对一个 8 位乘 8 位的乘法网络，只需要用一些 3 输入的华莱士树. 对 16 位乘 16 位的网络，则必须有规模为 3,5,7,7,5,3 的华莱士树；而对于 32 位乘 32 位的乘法，所需的华莱士树的规模将大到有 15 个输入. 必须指出，位于二端的二个 4 位的位片不需要华莱士树. 15 个输入的华莱士树可用三个 7 输入的华莱士树来构成. 华莱士树中所有没有用到的输入端均须接地，以保证它为“0”输入. Texas 仪器公司的 NMM (SN74S274) 和华莱士树 (SN74S275) 采用上述网络实现方法可以在 75 ns 内获得 16 位乘积，以及在 116 ns 内获得 32 位乘积.

下面将描述第二章中导出的相加乘法模块 (AMM) 的互连方法，它不需要位片求和

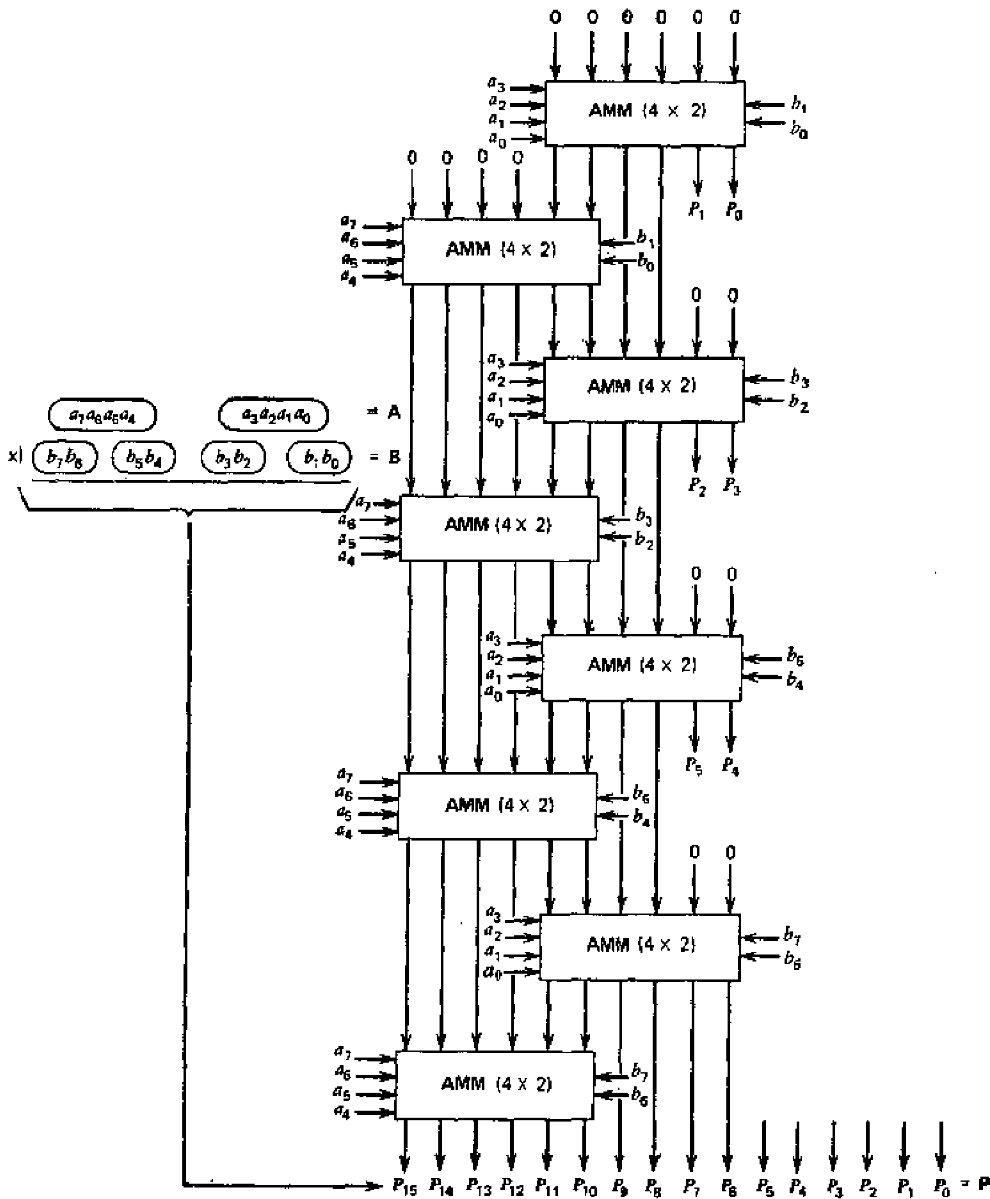


图 6.8 由 8 个 4 位乘 2 位的 AMM 组成一个 8 位乘 8 位的乘法网络的互连线路图 (AMM 的细节见图 2.15)

树. 图 6.8 指出了用八个 4 位乘 2 位的 AMM 所构成的一个 8 位乘 8 位的阵列乘法器. 一般地说，$4m$ 位乘 $4m$ 位的乘法网络可以用 $2m^2$ 个 4 位乘 2 位的 AMM 来构成. 这 $2m^2$ 个模块被排列成斜矩阵的形式，与图 6.8 相仿，它有 $2m$ 条 45° 的对角线和 m 个相互交织的列. 同一列的 AMM 都具有同样的 4 位被乘数，同一对角线上的 AMM 都具有同样的 2 位乘数. 根据 4 位乘 2 位的 AMM，这种 $4m$ 位乘 $4m$ 位的乘法网络的总时间延迟可以估算如下：

$$\Delta_N(4m \times 4m) = (3m - 1) \times 10\Delta + 2\Delta = (30m - 8)\Delta \tag{6.11}$$

这里的 10Δ 是指每个 4 位乘 2 位的模块的时间延迟，而 2Δ 是由于产生被加项的与门的延迟. 一个高速 TTL 门的延迟 $\Delta = 3$ ns，因此用这种 AMM 网络乘法来产生 32 位乘积所需要的时间不到 400 ns. 这种方法可以推广到使用较大的 AMM，如象 4 位乘 4 位或 8 位乘 8 位的模块. 表 6.2 归纳了各种规模的阵列乘法器所需要的 NMM. AMM 以及华莱士树等模块的数量和时间延迟.

表 6.2 三种实际规模的模块乘法网络的模块数量、华莱士树类型以及速度

网络规模	非相加的网络			相加的网络	
	4×4 的 NMM 的数量	华莱士树类型	乘法时间	4×2的 AMM 的数量	乘法时间
16 位乘 16 位	16	W_3, W_5, W_7	65Δ	32	112Δ
32 位乘 32 位	64	$W_3, W_5, W_7, W_9, W_{11}, W_{13}, W_{15}$	90Δ	128	232Δ
64 位乘 64 位	256	$W_3, W_5, W_7, \cdots W_{25}, W_{27}, W_{29}, W_{31}$	250Δ	512	472Δ

6.3 直接的补码乘法

本节讨论的阵列乘法器可以完成补码数的“直接”乘法，而不需要求补级. 这种直接方法排除了较慢的对 2 求补操作，大大加速了乘法过程. 我们先说明与这种直接的 2 的补码乘法相联系的数学特征. 在描述了几个一般化的全加器形式之后，我们将讨论几种直接的 2 的补码阵列乘法的算法，并且在下二节中指出它们可能实现的方案.

到此为止，我们一直把 2 的补码数作为“位置数”来处理，这个位置数有一个非权的符号和一个带正权的系数. 对于计算补码数的数值来说，一种较好的方法是使它的位置数有一个带负权的符号和带正权的系数. 让我们先简单地评论一下补码整数的常规的定点表示法. 今考虑一个补码数 $\mathbf{N} = (a_{n-1}a_{n-2}\cdots a_1a_0)_2$，这里 a_{n-1} 指定为符号. 根据 $\mathbf{N}$ 的符号，数 $\mathbf{N}$ 的值 N_v 可以表示为

$$N_v = \begin{cases} +\sum_{i=0}^{n-2} a_i 2^i & \text{如果 } a_{n-1} = 0 \ (\mathbf{N}\text{为正}) \\ -\left[1 + \sum_{i=0}^{n-2} (1 - a_i) 2^i\right] & \text{如果 } a_{n-1} = 1 (\mathbf{N}\text{ 为负}) \end{cases} \tag{6.12}$$

习惯上把补码数 $-\mathbf{N}$ 表示为

$$-N = \bar{a}_{n-1}\bar{a}_{n-2}\cdots\bar{a}_1\bar{a}_0 + 1 \tag{6.13}$$

式中 $\bar{a}_i = 1 - a_i$（对 $0 \leqslant i \leqslant n-1$）. 如果我们把负权因数 -2^{n-1} 强加到符号位 a_{n-1} 上，那么可以把方程式 6.12 中的二个位置表示式合并成下面的形式

$$N_v = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \tag{6.14}$$

它足以证明方程式 6.13 中的数 $-\mathbf{N}$ 能够用下式来计算.

$$-N_v = -(1-a_{n-1})2^{n-1} + \left[\sum_{i=0}^{n-2}(1-a_i)2^i\right] + 1 \tag{6.15}$$

在方程式 6.14 两边同乘以 -1，我们得到

$$\begin{aligned}
-N_v &= a_{n-1}2^{n-1} - \sum_{i=0}^{n-2} a_i 2^i = a_{n-1}2^{n-1} + (2^{n-1} - 2^{n-1}) - \sum_{i=0}^{n-2} a_i 2^i \\
&= (a_{n-1} - 1)2^{n-1} + 2^{n-1} - \sum_{i=0}^{n-2} a_i 2^i \\
&= -(1-a_{n-1})2^{n-1} + \left(1 + \sum_{i=0}^{n-2} 2^i\right) - \sum_{i=0}^{n-2} a_i 2^i \\
&= -(1-a_{n-1})2^{n-1} + \left[\sum_{i=0}^{n-2}(1-a_i)2^i\right] + 1
\end{aligned} \tag{6.16}$$

因此，我们从 6.16 式可以作出结论，即方程式 6.14 和 6.15 中给出的表达式是等效的，两者都是补码数的合理表达式．如果用一个数值例子，就能清楚地说明上面的推导结果．我们考虑 $n=5$ 的情况，$\mathbf{N}=(+13)_{10}=(01101)_2$，而 $-\mathbf{N}=(-13)_{10}=(10011)_2$．不难证明，$\mathbf{N}=(01101)_2$ 具有数值

$$N_v = -0\times 2^4 + 1\times 2^3 + 1\times 2^2 + 1\times 2^0 = (+13)_{10},$$

而 $-\mathbf{N}=(10011)_2$ 具有数值

$$-N_v = -1\times 2^4 + 0\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0 = -32 + 19 = (-13)_{10}.$$

常规的一位全加器假定它的 3 个输入和 2 个输出都是正权．这种加法器通过把正权或负权加到输入/输出端，可以归纳出四类加法单元．表 6.3 列出了这四类一般化的全加器的名称和逻辑符号．每一类全加器都是用它所包含的负权输入的个数来命名的．因此，0 类全加器没有负权输入，2 类全加器则有 2 个负权输入和一个正权输入，等等．

下面列出这四种运算的方程式，它们将说明这四类一般化的全加器的输入/输出关系．

$$0\text{类}\quad C2^1 + S2^0 = X2^0 + Y2^0 + Z2^0 \tag{6.17a}$$

$$1\text{类}\quad C2^1 + (-S)2^0 = X2^0 + Y2^0 + (-Z)2^0 \tag{6.17b}$$

$$2\text{类}\quad (-C)2^1 + S2^0 = (-X)2^0 + (-Y)2^0 + Z2^0 \tag{6.17c}$$

$$3\text{类}\quad (-C)2^1 + (-S)2^0 = (-X)2^0 + (-Y)2^0 + (-Z)2^0 \tag{6.17d}$$

从这四个运算方程式中导出四个一般化的加法器的真值表，如表 6.4 所示．注意 0 类和 3 类加法器可用同样的真值表来描述，但是带权的列的标题不相同．1 类全加器和 2 类全加器也使用同一个真值表，但其列的标题不相同．根据真值表中各项很容易导出支配这四类全加器的布尔方程式．

$$\begin{matrix}0\text{类}\\3\text{类}\end{matrix}\begin{cases}S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ\\ C = XY + YZ + ZX\end{cases} \tag{6.18a}$$

$$\begin{matrix}1\text{类}\\2\text{类}\end{matrix}\begin{cases}S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ\\ C = XY + X\bar{Z} + Y\bar{Z}\end{cases} \tag{6.18b}$$

表 6.3 四类一般化的全加器的名称和逻辑符号

类型	逻辑符号	操作
0 类 全加器	X Y Z 0 C S	X Y +) Z CS
1 类 全加器	X Y Z 1 C S	X Y +) −Z C(−S)
2 类 全加器	X Y Z 2 C S	−X −Y +) Z (−C) S
3 类 全加器	X Y Z 3 C S	−X −Y +) −Z (−C) (−S)

表 6.4 描述四类一般化全加器的真值表

全 加 器	带 权 输 入			带 权 输 出	
0 类	$X\cdot 2^0$	$Y\cdot 2^0$	$Z\cdot 2^0$	$C\cdot 2^1$	$S\cdot 2^0$
3 类	$-X\cdot 2^0$	$-Y\cdot 2^0$	$-Z\cdot 2^0$	$-C\cdot 2^1$	$-S\cdot 2^0$
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	0	1
	0	1	1	1	0
真值表	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
	1	1	1	1	1
1 类	$X\cdot 2^0$	$Y\cdot 2^0$	$-Z\cdot 2^0$	$C\cdot 2^1$	$-S\cdot 2^0$
2 类	$-X\cdot 2^0$	$-Y\cdot 2^0$	$Z\cdot 2^0$	$-C\cdot 2^1$	$S\cdot 2^0$
	0	0	0	0	0
	0	0	1	0	1
	0	1	0	1	1
	0	1	1	0	0
	1	0	0	1	1
真值表	1	0	1	0	0
	1	1	0	1	0
	1	1	1	1	1

必须指出，0 类和 3 类全加器是用同一对逻辑方程式来表征的，它和普通的一位全加器（0 类）是一致的．这是因为 3 类全加器可以简单地把 0 类全加器的所有输入和输出值全部反相来得到，反之也一样．在 1 类和 2 类全加器之间也能建立类似的关系．方程式 6.18 中的逻辑具有 2 级乘积-和的形式，它可以用接线逻辑（与或非 TTL 门）来实现，延迟为 2Δ．在下几节中，我们将用不同的逻辑符号来代表 0 类和 3 类全加器，尽管它们具有同样的内部结构；对 1 类和 2 类全加器也是如此．这样区分会使乘法阵列的线路图更便于理解．

6.4 Pezaris 阵列乘法器及其修改方案

使用混合型全加器的直接的 2 的补码阵列乘法器介绍如下．为清楚起见，我们解释 5 位乘 5 位的 2 的补码阵列乘法器的结构，通过它来阐明 Pezaris 的原始设计．然后我们再介绍他设计的二个可行的修改方案．考虑两个 5 位的 2 的补码数，被乘数

$$\mathbf{A} = (a_4)a_3a_2a_1a_0$$

和乘数 $\mathbf{B} = (b_4)b_3b_2b_1b_0$，它们具有带负权的符号 a_4 和 b_4，并用括弧标注．下面我们将用括弧来标注负的被加项，如 (a_ib_j)．**A** 和 **B** 相乘的过程中所包括的操作可以用图 6.9 中

						(a_4)	a_3	a_2	a_1	a_0	$= \mathbf{A}$
					×)	(b_4)	b_3	b_2	b_1	b_0	$= \mathbf{B}$
						(a_4b_0)	a_3b_0	a_2b_0	a_1b_0	a_0b_0	
					(a_4b_1)	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
				(a_4b_2)	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
			(a_4b_3)	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
+)	a_4b_4	(a_3b_4)	(a_2b_4)	(a_1b_4)	(a_0b_4)						
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0		$= \mathbf{P}$

					(0)	1	1	0	1	= +13
				×)	(1)	1	0	1	1	= − 5
					(0)	1	1	0	1	
				(0)	1	1	0	1		
			(0)	0	0	0	0			
		(0)	1	1	0	1				
+)	0	(1)	(1)	(0)	(1)					
	0	(1)	0	1	1	1	1	1	1	
	(1)	1	0	1	1	1	1	1	1	= −65

扩充的符号位 ↑

符号位

图 6.9 两个异号的补码数的乘法

的被加数矩阵和数值例子来说明．Pezaris 乘法器的逻辑电路原理图示于图 6.10．在 n 位乘 n 位的一般情况下，Pezaris 乘法器需要 $(n-2)^2$ 个 **0** 类全加器，$n-2$ 个 **1** 类全加器，$2n-3$ 个 **2** 类全加器和一个 **3** 类全加器，总共是 $n(n-1)$ 个全加器．Pezaris 乘法器所需要的总的乘法时间为

$$\mathbf{\Delta}_P = \Delta_a + 2(n-1)\Delta_i = 2\Delta + (2n-2)2\Delta = (4n-2)\Delta \qquad (6.19a)$$

如果在最后一行中全部采用先行进位，那么总的延迟可以减为

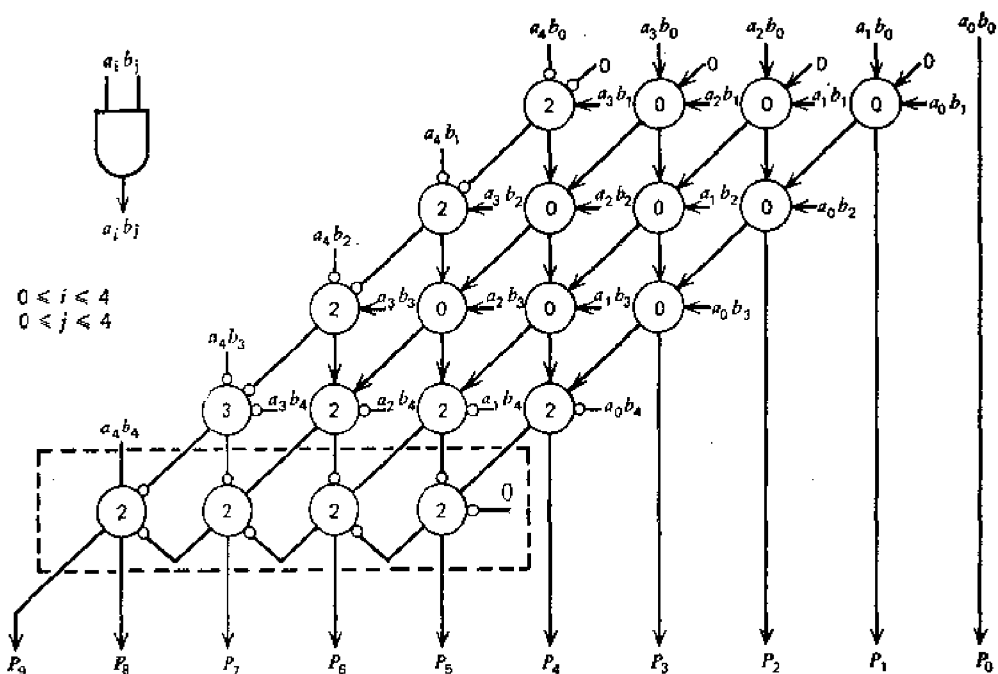


图 6.10 5 位乘 5 位的 Pezaris 阵列乘法器的原理电路图

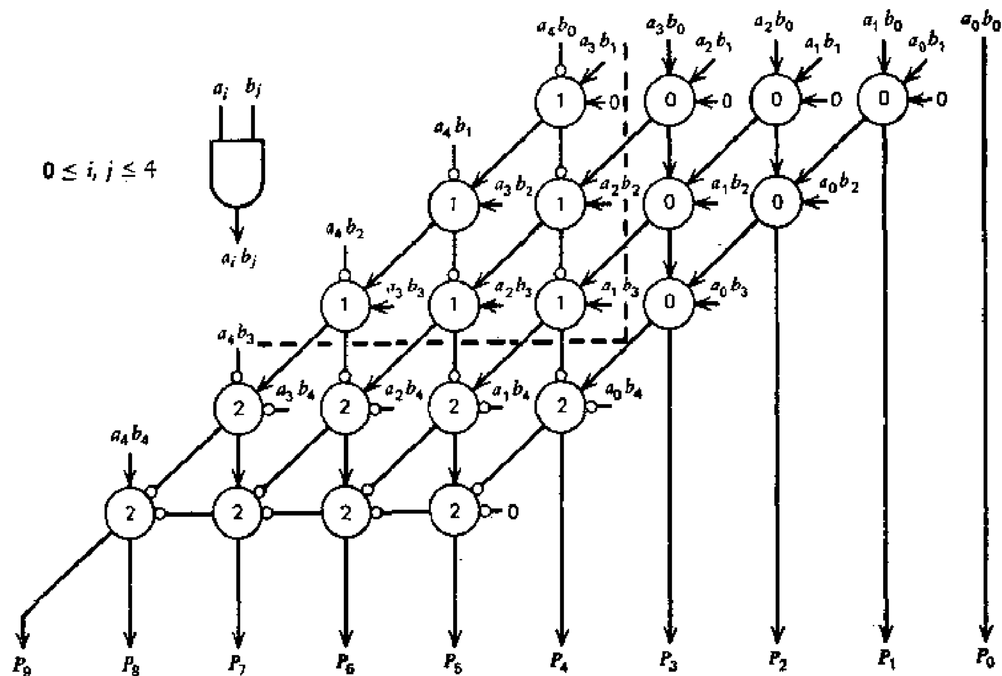


图 6.11 通过修改 Pezaris 乘法器得到的 5 位乘 5 位三段阵列乘法器的逻辑电路原理图

$$\Delta_P(\mathrm{CLA}) = \Delta_a + (n-1)\Delta_f + 8\Delta = 2\Delta + (n-1)2\Delta + 8\Delta = (2n+8)\Delta \quad (6.19b)$$

将图 6.10 左上角的那些 **2** 类全加器用 **1** 类全加器来代替，我们就能把 Pezaris 设计修改成图 6.11 所示的阵列．这个修改方案就是所谓三段阵列乘法器．阵列中各段之间用虚线隔开，每一段中只用一种全加器．右上角的三角形中只用 **0** 类全加器，左上角的三角形中只用 **1** 类全加器，阵列的最后二行只用 **2** 类全加器．图 6.9 中的被加数矩阵可以重新

排列成一个结构更均匀的阵列，而不必改变列方向的求和关系．该方法可用图 6.12 所示的重新排列的矩阵来阐明．利用这个矩阵，我们可以把正的被加数与负的被加数分开．这种分离使得有可能只用二种类型的全加器，即如图 6.13 中的二段阵列乘法器．上半部由 $(n-1)\times(n-2)$ 个 **0** 类全加器组成，其功能如同一个 $(n-1)$ 位乘 $(n-1)$ 位的整数乘法器，这里没有任何负的被加数．下半部由二行共 $2(n-1)$ 个 **2** 类全加器组成，把所有负的被加数加到从上半部来的所有正的被加数的中间和上．

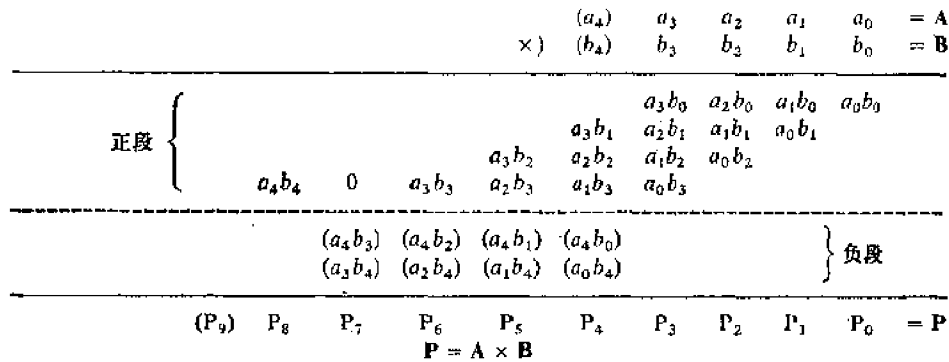


图 6.12　在 5 位乘 5 位补码数的二段阵列乘法器中，把正的和负的被加数分开

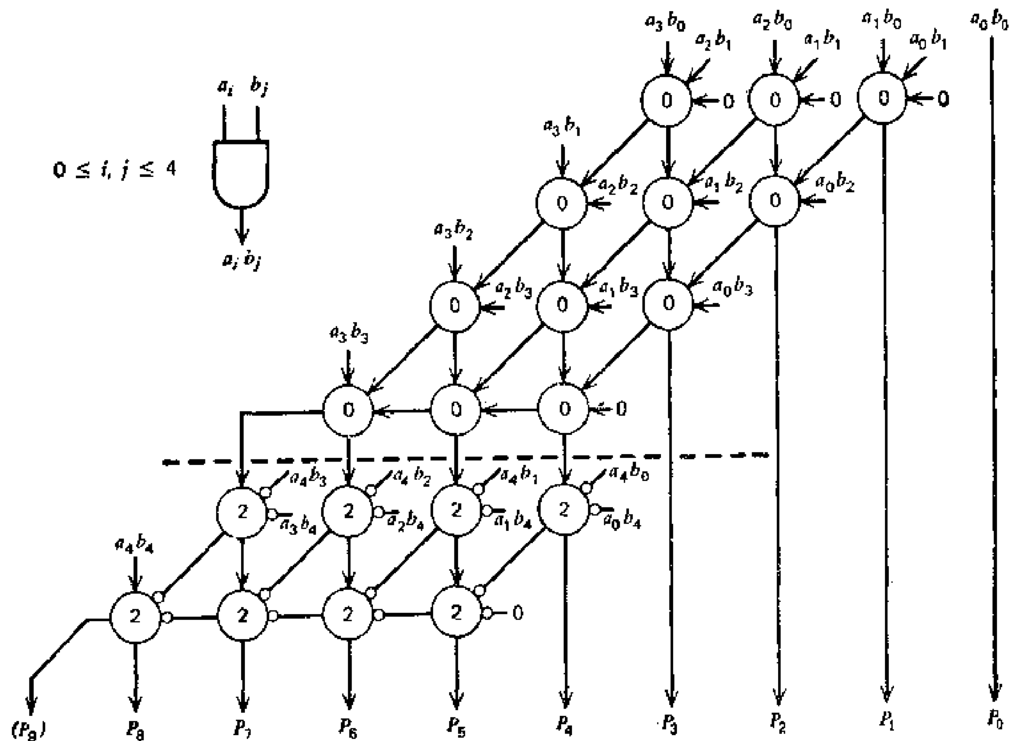


图 6.13　根据图 6.14 中的矩阵所设计的 5 位乘 5 位 2 的补码二段阵列乘法器

二段阵列乘法器的总的乘法时间比前两种阵列稍慢一些．最长的进位传播途径是沿着最右边的对角线以及最后一行，总的时间延迟为

$$\Delta_{BS} = \Delta_a + (n-1)\Delta_f + (n-1+1)\Delta_f = 2\Delta + (2n-1)2\Delta = 4n\Delta \quad (6.20)$$

在最后一行中全部采用先行进位时，我们就能达到下列速度

$$\Delta_{BS}(CLA) = \Delta_a + n\Delta_f + 8\Delta = (2n+10)\Delta$$

6.5 Baugh Wooley 补码乘法器

Baugh 和 Wooley[3] 曾经提出过一种直接的 2 的补码阵列乘法的算法．他们的算法主要优点是所有被加数的符号都是正的，因此该阵列可以全部采用普通的 0 类全加器来构成．这种均匀的结构对于大规模集成电路是很有吸引力的．

今考虑两个补码数，一个是 m 位的被乘数 $\mathbf{A} = (a_{m-1}a_{m-2}\cdots a_1a_0)_2$，另一个是 n 位的乘数 $\mathbf{B} = (b_{n-1}b_{n-2}\cdots b_1b_0)_2$．$\mathbf{A}$ 和 $\mathbf{B}$ 的数值用 A_v 和 B_v 表示，它们可以用方程式 6.14 写出

$$A_v = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \quad B_v = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \tag{6.21}$$

用补码表示的乘积 $\mathbf{P} = A \cdot B = (p_{m+n-1}p_{m+n-2}\cdots p_1p_0)_2$ 的数值设为 P_v，它可以用 a_i 和 b_j 的系数的乘积再配上相应的权因数写出．

$$\begin{aligned} P_v &= -p_{m+n-1}2^{m+n-1} + \sum_{i=0}^{m+n-2} p_i 2^i = A_v B_v \\ &= \left(-a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i\right)\left(-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i\right) \\ &= a_{m-1}b_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\ &\quad - \sum_{i=0}^{m-2} a_i b_{n-1} 2^{n-1+i} - \sum_{i=0}^{n-2} a_{m-1} b_i 2^{m-1+i} \end{aligned} \tag{6.22}$$

在方程式 6.22 中，被加数 a_ib_{n-1} 的符号对于 $i = 0, \cdots, m-2$ 各项来说都是负的，$a_{m-1}b_j$ 的符号对于 $j = 0, \cdots, n-2$ 各项也都是负的．如图 6.14 所示，将所有负的被加数都排在最后二行，那么通过前 $n-2$ 行被加数相加，再减去最后二行来形成乘积，就如同我们过去在二段阵列乘法器所做的那样．不过并不是减去负的被加数，而是用加上被加数的反码来代替．利用方程式 6.15，我们可以把方程式 6.22 中的第 3 项减法

$$\sum_{i=0}^{m-2} a_i b_{n-1} 2^{n-1+i} = 2^{n-1}\left(-0\cdot 2^m + 0\cdot 2^{m-1} + \sum_{i=0}^{m-2} a_i b_{n-1} 2^i\right) \tag{6.23a}$$

改成对下式的加法

$$2^{n-1}\left(-1\cdot 2^m + 1\cdot 2^{m-1} + 1 + \sum_{i=0}^{m-2} \bar{a}_i \bar{b}_{n-1} 2^i\right) \tag{6.23b}$$

注意方程式 6.23b 具有下列数值

$$\begin{cases} 0 \quad 对\ b_{n-1} = 0 \\ 2^{n-1}\left(-2^m + 2^{m-1} + 1 + \sum_{i=0}^{m-2} \bar{a}_i 2^i\right) \quad 对\ b_{n-1} = 1 \end{cases} \tag{6.23c}$$

根据方程式 6.23c，方程式 6.23b 可以写成

$$2^{n-1}\left(-2^m + 2^{m-1} + \bar{b}_{n-1}2^{m-1} + b_{n-1} + \sum_{i=0}^{m-2} \bar{a}_i b_{n-1} 2^i\right) \tag{6.23d}$$

$P = A \times B$

a_{m-1} a_{m-2} $\cdots$ a_3 a_2 a_1 a_0 $= \mathbf{A}$

×) b_{n-1} $\cdots$ b_2 b_1 b_0 $= \mathbf{B}$

正的被加数:

$a_{m-2}b_0$ $\cdots$ a_3b_0 a_2b_0 a_1b_0 a_0b_0

$a_{m-2}b_1$ $\cdots$ a_3b_1 a_2b_1 a_1b_1 a_0b_1

$a_{m-2}b_2$ $\cdots$ a_3b_2 a_2b_2 a_1b_2 a_0b_2

$\cdot^{\cdot^{\cdot}}$

$a_{m-1}b_{n-1}$ 0 $a_{m-2}b_{n-2}$ $\cdots$ a_2b_{n-2} a_1b_{n-2} a_0b_{n-2}

负的被加数:

0 0 $a_{m-2}b_{n-1}$ $a_{m-3}b_{n-1}$ $\cdots$ a_1b_{n-1} a_0b_{n-1}

0 0 $a_{m-1}b_{n-2}$ $a_{m-1}b_{n-3}$ $\cdots$ $a_{m-1}b_0$

P_{m+n-1} P_{m+n-2} P_{m+n-3} P_{m+n-4} $\cdots$ P_{m-1} $\cdots$ P_{n-1} $\cdots$ P_2 P_1 P_0 $= \mathbf{P}$

注：上面的矩阵表示 $m>n$ 的情况

图 6.14 在 m 位乘 n 位的 2 的补码乘法中，将正的和负的被加数分开

$P = A \times B$

a_{m-1} a_{m-2} $\cdots$ a_3 a_2 a_1 a_0 $= \mathbf{A}$

b_{n-1} $\cdots$ b_2 b_1 b_0 $= \mathbf{B}$

$a_{m-2}b_0$ $\cdots$ a_3b_0 a_2b_0 a_1b_0 a_0b_0

$a_{m-2}b_1$ $\cdots$ a_2b_1 a_1b_1 a_0b_1

$a_{m-2}b_2$ $\cdots$ a_1b_2 a_0b_2

$\cdot^{\cdot^{\cdot}}$

$a_{m-1}b_{n-1}$ 0 $a_{m-2}b_{n-2}$ $\cdots$ a_2b_{n-2} a_1b_{n-2} a_0b_{n-2}

$\bar{a}_{n-1}$ $\bar{a}_{m-2}b_{n-1}$ $\bar{a}_{m-3}b_{n-1}$ $\cdots$ $\bar{a}_2b_{n-1}$ $\bar{a}_1b_{n-1}$ $\bar{a}_0b_{n-1}$

$\bar{b}_{n-1}$ $a_{m-1}\bar{b}_{n-2}$ $a_{m-1}\bar{b}_{n-3}$ $\cdots$ $a_{m-1}\bar{b}_1$ $a_{m-1}\bar{b}_0$

1 a_{m-1} b_{n-1}

P_{m+n-1} P_{m+n-2} P_{m+n-3} P_{m+n-4} $\cdots$ P_{m-1} $\cdots$ P_{n-1} P_3 P_2 P_1 P_0 $= \mathbf{P}$

图 6.15 利用图6.14中负被加数的补码，得出全部为正被加数的 Baugh-Wooley 补码乘法的算法（这里再次假定 $m>n$）

这意味着图 6.14 中倒数第二个行向量

$$0\ 0\ a_{m-2}b_{n-1}a_{m-3}b_{n-1}\cdots a_0b_{n-1} \tag{6.24a}$$

可以用下面二个行向量的和来代替

$$\begin{matrix} 0 & \bar{b}_{n-1} & \bar{a}_{m-2}b_{n-1} & \bar{a}_{m-3}b_{n-1} & \cdots \bar{a}_0 b_{n-1} \\ 1 & 1 & 0 & 0 & \cdots b_{n-1} \end{matrix} \tag{6.24b}$$

同样地，我们也可以把方程式 6.22 中第四项 $\sum_{i=0}^{m-2} a_{m-1}b_i \cdot 2^{m-1+i}$ 的减法替换为对下式的加法

$$2^{m-1}\left(-2^n + 2^{n-1} + \bar{a}_{m-1}2^{n-1} + a_{m-1} + \sum_{i=0}^{n-2} a_{m-1}\bar{b}_i 2^i\right) \tag{6.25}$$

这意味着图 6.14 中最后一个行向量

$$0\ \ 0\ \ a_{m-1}b_{n-2}\ \ a_{m-1}b_{n-3}\cdots a_{m-1}b_0 \tag{6.26a}$$

可以用下列二个行向量的和来代替

$$\begin{matrix} 0 & \bar{a}_{m-1} & a_{m-1}\bar{b}_{n-2} & a_{m-1}\bar{b}_{n-3} & \cdots a_{m-1}\bar{b}_0 \\ 1 & 1 & 0 & 0 & \cdots a_{m-1} \end{matrix} \tag{6.26b}$$

因此，方程式 6.24b 中第二个行向量和方程式 6.26 b 中第二个行向量可以按下面的方法组合成单一的行向量，由于 2 的补码加法的性质，符号列 P_{m+n-1} 的进位输出可忽略。

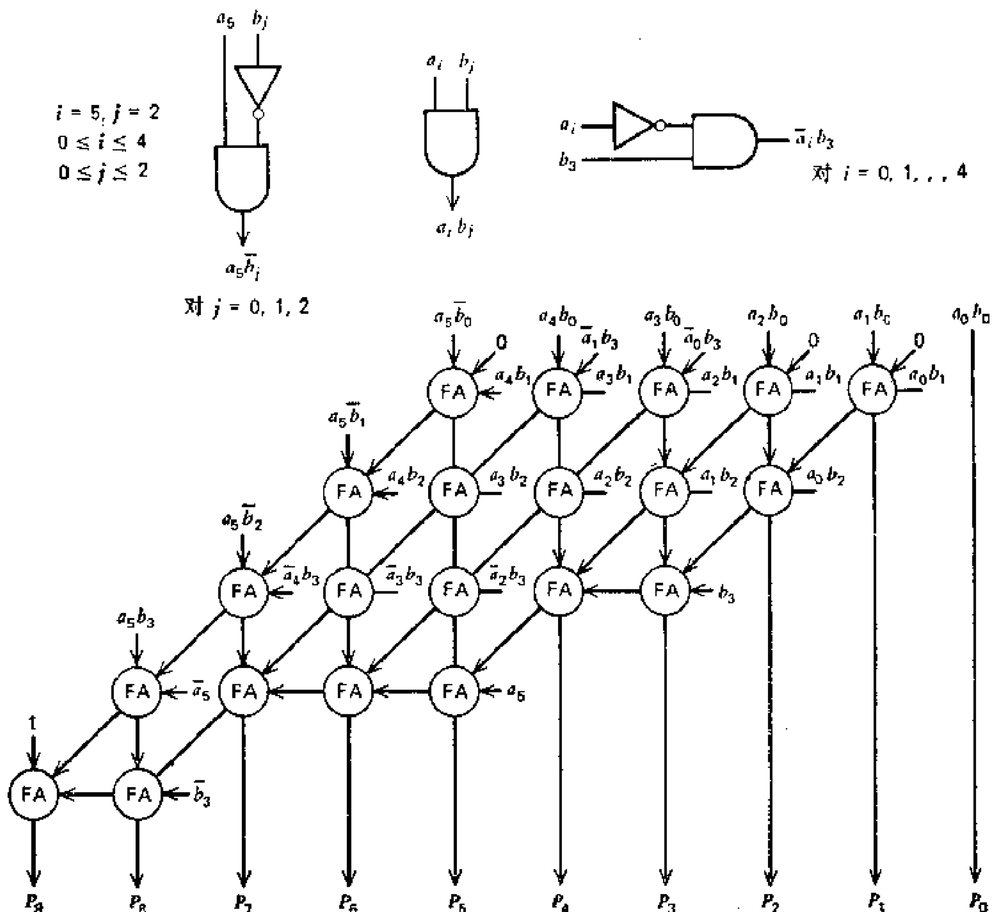


图 6.16 6 位乘 4 位的 Baugh-Wooley 补码阵列乘法器的逻辑电路原理图

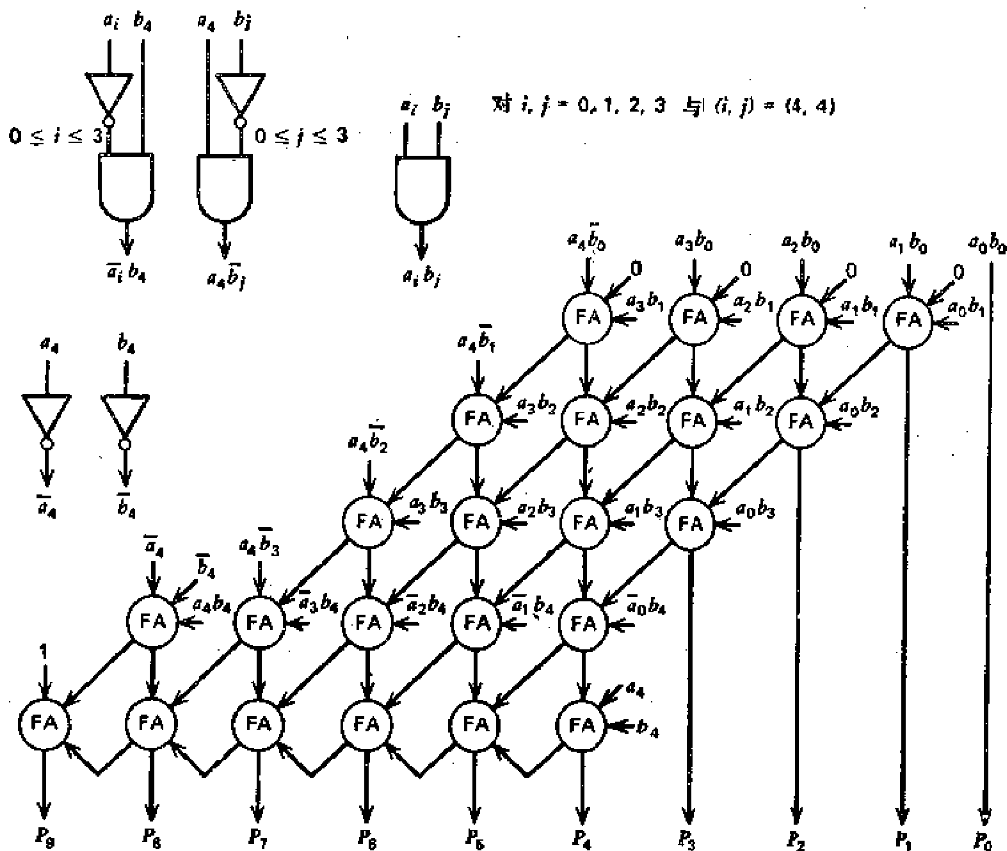


图 6.17 5 位乘 5 位的 Baugh-Wooley 补码阵列乘法器的逻辑电路原理图 (Hwang[18])

$$\begin{array}{ccccccc} 1 & 0\cdots0 & a_{m-1} & 0\cdots & b_{n-1} & 0\cdots & 0 \\ \uparrow & & \uparrow & & \uparrow & & \uparrow \\ p_{m+n-1} & & p_{m-1} & & p_{n-1} & & p_0 \end{array} \quad \text{列} \tag{6.27}$$

图 6.14 中最后二行用公式 (6.24a)、(6.26b) 和 (6.27) 代替以后，我们便得到新的被加数矩阵，如图 6.15 所示．这个新矩阵的主要特征在于它的均匀性，它只包含正的被加数．因此，只要用 **0** 类全加器来执行加法就能获得乘积．根据 Baugh-Wooley 的算法，6×4 的 2 的补码乘法器的逻辑电路原理图如图 6.16 所示．必须指出，如果反相输入不能直接从数据总线上取用，那么就要用一些反相器．一般地说，m 位乘 n 位的 Baugh-Wooley 乘法器需要 $m(n-1)+3$ 个全加器来实现．当 $m=n$ 时，电路设计稍有区别，在图 6.17 中指出了 5 位乘 5 位的 Baugh-Wooley 乘法器的原理电路．在 n 位乘 n 位的乘法器中，需要

表 6.5 补码数的 Pezaris 乘法器、三段乘法器、二段乘法器以及 Baugh-Wooley 乘法器的比较

		$m\times n$ 补码阵列乘法器			
		Pezaris (图 6.10)	三段 (图 6.11)	二段 (图 6.13)	Baugh-Wooley (图 6.16)
全加器使用数量	**0** 类	$(m-2)(n-2)$	$(m-1)(n-2)/2$	$(m-2)(n-1)$	$m(n-1)+3$
	1 类	$m-2$	$(m-1)(n-2)/2$	0	0
	2 类	$m+n-3$	$2(m-1)$	$2(m-1)$	0
	3 类	1	0	0	0
总时间延迟(乘法时间)		$2(m+n)\Delta-2\Delta$	$2(m+n)\Delta-2\Delta$	$2(m+n)\Delta$	$2(m+n)\Delta$

n^2-n+3 个全加器. 表 6.5 中，我们对直接的补码乘法的四种不同设计方案作了比较. 这些补码乘法器的速度几乎相同.

6.6 通用乘法阵列

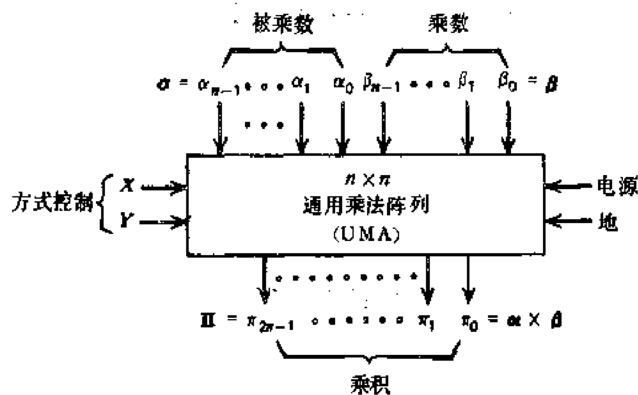
到现在为止，我们已经讨论了多种并行的阵列乘法方案，并且把不带符号和带符号的数分别加以处理. 这一节我们将研究用一般化的通用方法来设计高速阵列乘法器. 所谓乘法器的通用，是指它能对以任何预先规定的数的表示法表示的两个二进制数相乘，并求出乘积. 这意味着它可能是下列四种表示法中的任何一种：不带符号的数值，带符号的数值，反码和补码. 一个 n 位乘 n 位的通用乘法阵列 (UMA)，它的输入输出关系将如图 6.18 所示. 两个输入是被乘数

$$\boldsymbol{\alpha}=\alpha_{n-1}\cdots\alpha_1\alpha_0$$

和乘数 $\boldsymbol{\beta}=\beta_{n-1}\cdots\beta_1\beta_0$，输出为 $\boldsymbol{\alpha}$ 和 $\boldsymbol{\beta}$ 的乘积，用符号表示为

$$\boldsymbol{\Pi}=\pi_{2n-1}\cdots\pi_1\pi_0.$$

$\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ 和 $\boldsymbol{\Pi}$ 假定采用同一种数的表示法.



方式控制 XY	所执行的乘法操作
00	不带符号的乘法
01	带符号数值的乘法
10	1 的补码乘法
11	2 的补码乘法

图 6.18 具有操作方式控制的 $n\times n$ 通用乘法阵列的框图

这种 UMA 所涉及的求和操作由图 6.19 中的流程图来说明. 算前求补和算后求补级只是在变换反码数时才有需要. 这两级在处理不带符号的数、带符号的数以及补码数时均被旁路.

相应于四种性质不同的数的表示法，分别给出并行乘法的算法. 令 $\mathbf{A}=a_{n-1}\cdots a_1a_0$ 和 $\mathbf{B}=b_{n-1}\cdots b_1b_0$，它们可以根据 $\boldsymbol{\alpha}$ 和 $\boldsymbol{\beta}$ 的各位 α_i 和 β_i 与符号位 α_{n-1} 和 β_{n-1} 进行异或得到，对 1 的补码方式 $(XY=10)$ 来说，在 $0\leqslant i\leqslant n-1$ 时

$$\begin{aligned}a_i&=\alpha_i\oplus(X\overline{Y}\alpha_{n-1})\\ b_i&=\beta_i\oplus(X\overline{Y}\beta_{n-1})\end{aligned}\tag{6.28}$$

令 $\mathbf{P}=P_{2n-1}P_{2n-2}\cdots P_1P_0$ 为 $\mathbf{A}$ 与 $\mathbf{B}$ 相乘后得到的乘积，对于 2 的补码乘法只需要加上适当的校正项. 今定义一个布尔变量 $\gamma=\alpha_{n-1}\oplus\beta_{n-1}$. $\mathbf{P}$ 与最终输出 $\boldsymbol{\Pi}$ 之间有如下关系.

$$\begin{aligned}\Pi_{2n-1}&=P_{2n-1}(\overline{X\oplus Y})+\gamma(X\oplus Y)\\ \Pi_{2n-2}&=P_{2n-2}(\overline{X\oplus Y})+X\overline{Y}\gamma\end{aligned}\tag{6.29}$$

注意：只有 1 的补码乘法才需要上面的求反操作. 对不带符号、带符号数值以及补码方式的乘法，我们有简单的关系式 $\mathbf{A}=\boldsymbol{\alpha}$, $\mathbf{B}=\boldsymbol{\beta}$ 和 $\Pi_k=P_k$，这里 $0\leqslant k\leqslant 2n-2$；但在带符号数值的乘法中，我们置符号 $a_{n-1}=b_{n-1}=0$.

在不带符号的乘法中，$(X, Y)=(0, 0)$，如图 6.1 所示，乘积 $\mathbf{P}=\mathbf{A}\times\mathbf{B}$ 可用方程

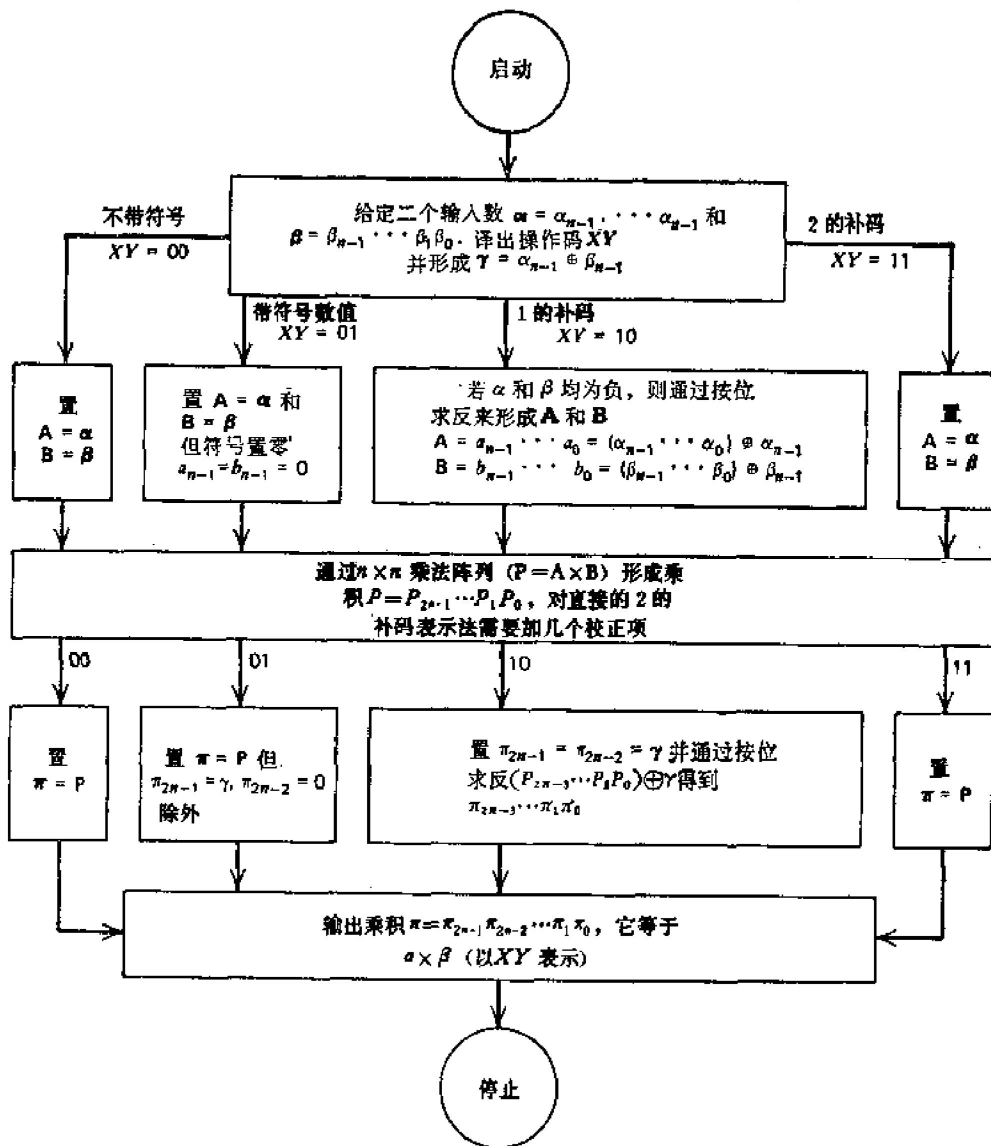


图 6.19 通用乘法阵列（UMA）所涉及的操作流程图说明

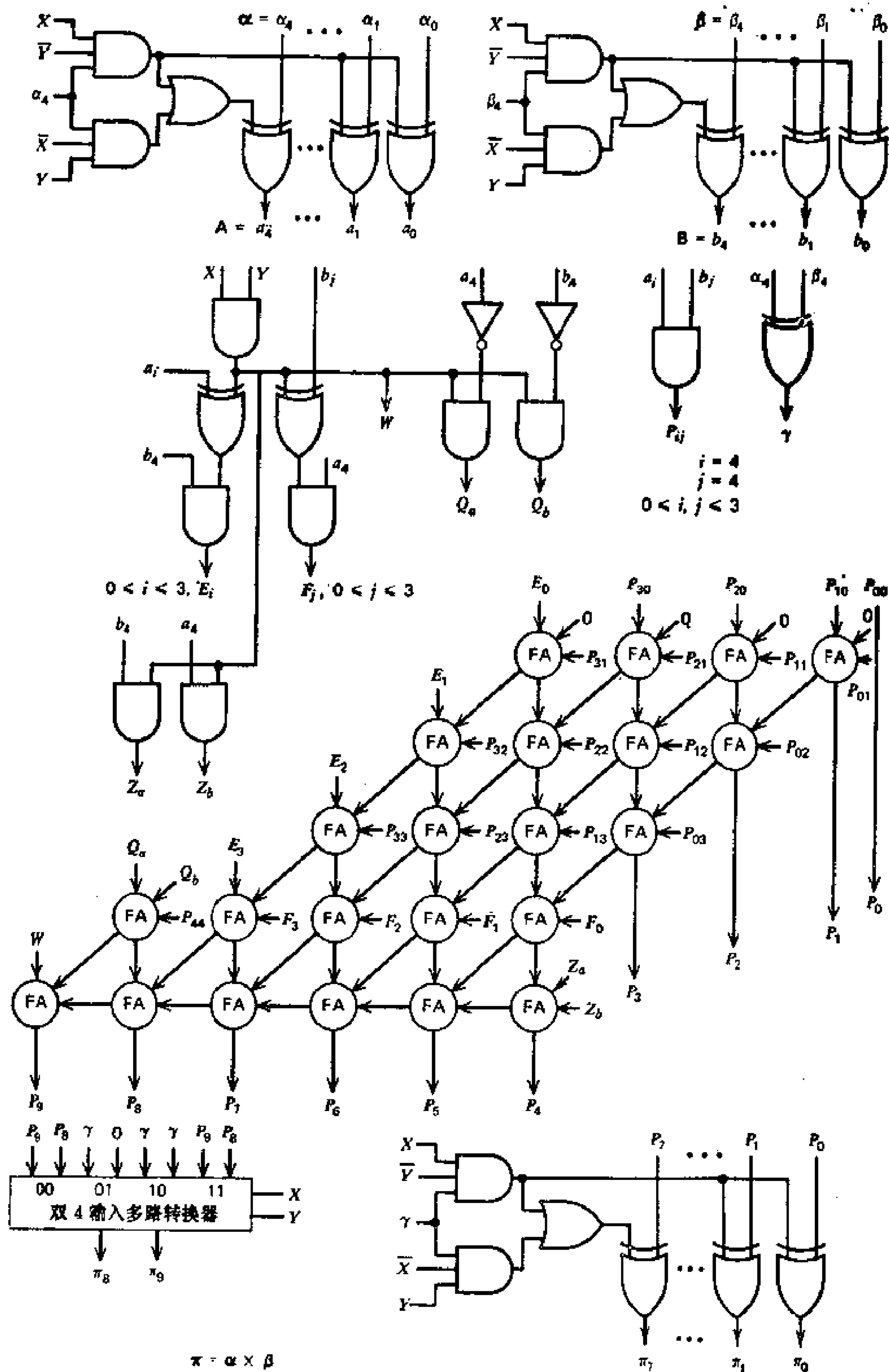
式 6.2（对 $m = n$）直接相乘得到。

对于带符号数值的乘法，其 $(X, Y) = (0, 1)$，乘积的符号位 $P_{2n-1} = \gamma$，而下一位 $P_{2n-2} = 0$。在应用方程式 6.2 来求其余的乘积位 $P_{2n-3} \cdots P_1 P_0$ 之前，应使符号位为零 $a_{n-1} = b_{n-1} = 0$。

对 1 的补码乘法，有 $(X, Y) = (1, 0)$，前面二位（包括扩充的符号位）可求之如下：$P_{2n-1} = P_{2n-2} = \gamma$。在算前求补操作之后，**A** 和 **B** 两者都成为 $a_{n-1} = b_{n-1} = 0$ 的正的带符号数值的数。这就意味着其余位 $P_{2n-3} \cdots P_1 P_0$ 也能用方程式 6.2 求出。

补码乘法［相应于 $(X, Y) = (1, 1)$］推荐用第 6.5 节中描述的 Baugh-Wooley 算法。对于一般的 $n \times n$ Baugh-Wooley 乘法器，在图 6.15 的被加数矩阵中加进若干项以后，便得能够产生下列补码数

$$\mathbf{P} = 1 \times 2^{2n-1} + (\bar{a}_{n-1} + \bar{b}_{n-1} + a_{n-1} b_{n-1}) \times 2^{2n-2}$$



$\pi = \alpha \times \beta$

图 6.20　5 位乘 5 位的通用乘法阵列（UMA）的原理图

$$+\left(\sum_{i=0}^{n-2} a_i \times 2^i\right) \times \left(\sum_{j=0}^{n-2} b_j \times 2^j\right) + (a_{n-1} \times 2^{n-1}) \times \left(\sum_{j=0}^{n-2} \bar{b}_j \times 2^j\right)$$

$$+\left(\sum_{i=0}^{n-2} \bar{a}_i \times 2^i\right) \times (b_{n-1} \times 2^{n-1}) + (a_{n-1} + b_{n-1}) \times 2^{n-1} \qquad (6.30)$$

在这四种操作方式中间，2 的补码乘法涉及的被加数的项数最多.带符号数值和 1 的补码乘法涉及被加数的项数最少. 这也反映了增加这些项使硬件需要量也增多了，正如前节所指出的那样.

现在介绍实现通用并行乘法的逻辑电路. 一个 5 位乘 5 位的 UMA 的原理图表示在图 6.20 上. 算前求补和算后求补级在符号位和方式线的控制下，由二级异或门来实现. 前面二位 π_{2n-1} 和 π_{2n-2} 通过双 4 至 1 的多路转换器来检索所有这四种情况. UMA 的核心部分是叠接的门控全加器阵列，表示在图的中部. 核心部分实现对移位后的被加数进行求和.

大多数全加器的输入端被连接到一些激励逻辑电路. 这些逻辑电路的逻辑方程式列在下面.

$$P_{ij} = a_i b_j \ \text{对} \ \begin{matrix} 0 \leqslant i \leqslant n-2 \\ 0 \leqslant j \leqslant n-2 \end{matrix}$$

$$\begin{aligned}
P_{n-1,\ n-1} &= a_{n-1} b_{n-1} \\
E_i &= (XY \oplus a_i) b_{n-1} \ \text{对} \ 0 \leqslant i \leqslant n-2 \\
F_j &= a_{n-1} (XY \oplus b_j) \ \text{对} \ 0 \leqslant j \leqslant n-2 \\
Q_a &= XY \bar{a}_{n-1}; \quad Q_b = XY \bar{b}_{n-1} \\
Z_a &= XY a_{n-1}; \quad Z_b = XY b_{n-1} \\
W &= XY
\end{aligned} \qquad (6.31)$$

UMA 的设计参数可以概括如下：一个一般化的 UMA 表示为 UMA ($n \times n$)，它需要用 $n^2 - n + 3$ 个全加器，$6n - 3$ 个异或门，$n^2 + 11$ 个与门，3 个或门和一个双 4 至 1 的多路转换器来组成. 在 UMA($n \times n$) 中总共需要 $4n + 4$ 个外部连接端点. UMA($n \times n$) 的时间延迟将在下一节中分析.

6.7 可编程序的相加乘法模块

UMA 的硬件元件数和连接端数随着它的规模的增大而迅速增加. 这说明了对于很大的 n，要想把整个 $n \times n$ 的 UMA 做在一个集成电路片上是有困难的. 因此，必须用小的乘法模块来构成大的通用乘法网络. 这些模块不仅应该具有相加的能力，而且具有外部可编程序的能力.

可编程序的相加乘法 (PAM) 模块的框图如图 6.21 所示. 我们用符号 $M(k \times k)$ 来表示一个 PAM 模块. 作为一个例子，我们选择 $M(4 \times 4)$ 模块的设计来加以解释. 一个 $M(k \times k)$ PAM 模块可以在可编程序的控制下表现为四种不同的工作方式

$$M_{00}(k \times k); M_{01}(k \times k); M_{10}(k \times k); M_{11}(k \times k) \qquad (6.32)$$

式中的下标相应于出现在 U, V 控制端上的数值. 根据这四种操作方式，一个 PAM 模块 $M(k \times k)$ 能够用来计算下列四种算术表示式.

$M_{00}(k \times k)$:

$$\begin{aligned}\mathbf{P} &= \mathbf{A} \times \mathbf{B} + \mathbf{C} + \mathbf{D} \\ &= \left(\sum_{i=0}^{k-1} a_i \times 2^i\right) \times \left(\sum_{j=0}^{k-1} b_j \times 2^j\right) + \sum_{r=0}^{k-1} c_r \times 2^r + \sum_{r=0}^{k-1} d_r \times 2^r \\ &= \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} a_i b_j \times 2^{i+j} + \sum_{r=0}^{k-1}(c_r + d_r) \times 2^r \end{aligned} \tag{6.33a}$$

$M_{01}(k \times k)$:

$$\begin{aligned}\mathbf{P} &= \left(\sum_{i=0}^{k-1} a_i \times 2^i\right) \times \left(\sum_{j=0}^{k-2} b_j \times 2^j\right) + \left(\sum_{i=0}^{k-1} \bar{a}_i \times 2^i\right) \times b_{k-1} \times 2^{k-1} \\ &\quad + \sum_{r=0}^{k-1} c_r \times 2^r + \sum_{r=0}^{k-1} d_r \times 2^r \\ &= \sum_{i=0}^{k-1}\sum_{j=0}^{k-2} a_i b_j \times 2^{i+j} + \sum_{i=0}^{k-1} \bar{a}_i b_{k-1} \times 2^{k+i-1} + \sum_{r=0}^{k-1}(c_r + d_r) \times 2^r \end{aligned} \tag{6.33b}$$

$M_{10}(k \times k)$:

$$\begin{aligned}\mathbf{P} &= \left(\sum_{i=0}^{k-2} a_i \times 2^i\right) \times \left(\sum_{j=1}^{k-1} b_j \times 2^j\right) + a_{k-1} \times 2^{k-1} \times \left(\sum_{j=0}^{k-1} \bar{b}_j \times 2^j\right) \\ &\quad + \sum_{r=0}^{k-1} c_r \times 2^r + \sum_{r=0}^{k-1} d_r \times 2^r \\ &= \sum_{i=0}^{k-2}\sum_{j=0}^{k-1} a_i b_j \times 2^{i+j} + \sum_{j=0}^{k-1} a_{k-1} \bar{b}_j \times 2^{k+j-1} + \sum_{r=0}^{k-1}(c_r + d_r) \times 2^r \end{aligned} \tag{6.33c}$$

$M_{11}(k \times k)$:

$$\begin{aligned}\mathbf{P} &= \sum_{i=0}^{k-2}\sum_{j=0}^{k-2} a_i b_j \times 2^{i+j} + a_{k-1} b_{k-1} \times 2^{2k-2} \\ &\quad + \sum_{j=0}^{k-1} a_{k-1} \bar{b}_j \times 2^{k+j-1} + \sum_{i=0}^{k-1} a_i b_{k-1} \times 2^{k+i-1} + 1 \times 2^{2k-1} \\ &\quad + (\bar{a}_{k-1} + \bar{b}_{k-1}) \times 2^{k-1} + \sum_{r=0}^{k-1}(c_r + d_r) \times 2^r \end{aligned} \tag{6.33d}$$

这里 $\mathbf{A} = a_{k-1}\cdots a_1 a_0$ 为被乘数输入，$\mathbf{B} = b_{k-1}\cdots b_1 b_0$ 为乘数输入，而 $\mathbf{C} = c_{k-1}\cdots c_1 c_0$ 和 $\mathbf{D} = d_{k-1}\cdots d_1 d_0$ 则为两个相加的输入．结果输出取为下面的形式

$$\mathbf{P} = \begin{cases} P_{2k-1}P_{2k-2}\cdots P_1 P_0, & \text{对 } M_{00}(k \times k); M_{01}(k \times k) \text{ 和 } M_{10}(k \times k) \\ P^*_{2k-1}P_{2k-2}\cdots P_1 P_0, & \text{对 } M_{11}(k \times k) \end{cases} \tag{6.34}$$

PAM 模块 $M(k \times k)$ 的四种操作方式的矩阵描述在图 6.22 中给出，那里取 $k = 4$．这个 PAM 模块 $M(4 \times 4)$ 用门控全加器实现的方法表示在图 6.23 中．内部激励逻辑可以用下列方程组来说明

$$\begin{aligned} &T_{i3} = (v \oplus a_i) b_3 \quad \text{对 } i = 0, 1, 2 \\ &T_{3j} = a_3 (b_j \oplus u) \quad \text{对 } j = 0, 1, 2 \\ &T_{33} = (\overline{u \oplus v}) a_3 b_3 + u\bar{v}\bar{a}_3 b_3 + \bar{u} v a_3 \bar{b}_3 \\ &R_a = uv\bar{a}_3; \quad R_b = uv\bar{b}_3 \quad S = uv \end{aligned} \tag{6.35}$$

乘数　　第一项加法输入

$B_{3-0}=b_3\ b_2\ b_1\ b_0$　　$c_3\ c_2\ c_1\ c_0=C_{3-0}$

被乘数 A_{3-0}：a_3，a_2，a_1，a_0

方式控制：U，V

4 位乘 4 位可编程序加法的乘法模块

第二项加法输入 D_{3-0}：d_3，d_2，d_1，d_0

电源

地

P_7^*　P_7　P_6　P_5　P_4　P_3　P_2　P_1　P_0

输出 $\begin{cases} P_{7-0}, & 当\ UV=00,01,10 \\ P_7^* P_{6-0}, & 当\ UV=11. \end{cases}$

方式控制 U　V	操作方式
0　0	M_{00}
0　1	M_{01}
1　0	M_{10}
1　1	M_{11}

B_{3-0}　C_{3-0}

A_{3-0}　$M_{UV}(4\times4)$　D_{3-0}

P_{7-4}　P_{3-0}

工作在 M_{UV} 方式的 PAM 模块 $M(4\times4)$ 的简化图

图 6.21　4 位乘 4 位的可编程序相加乘法模块的输入-输出图及其简化图 $M_{UV}(4\times4)$

$$
\begin{array}{rcccccccc}
 & & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & \times) & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
 & & 0 & & & c_3 & c_2 & c_1 & c_0 \\
+) & 0 & 0 & & & d_3 & d_2 & d_1 & d_0 \\
\hline
 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

(a) 方式 $M_{00}(4\times4)$

$$
\begin{array}{rcccccccc}
 & & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & \bar{a}_3b_3 & \bar{a}_2b_3 & \bar{a}_1b_3 & \bar{a}_0b_3 & & & \\
\hline
 & & 0 & & & c_3 & c_2 & c_1 & c_0 \\
+) & 0 & 0 & & & d_3 & d_2 & d_1 & d_0 \\
\hline
 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

(b) 方式 $M_{01}(4\times4)$

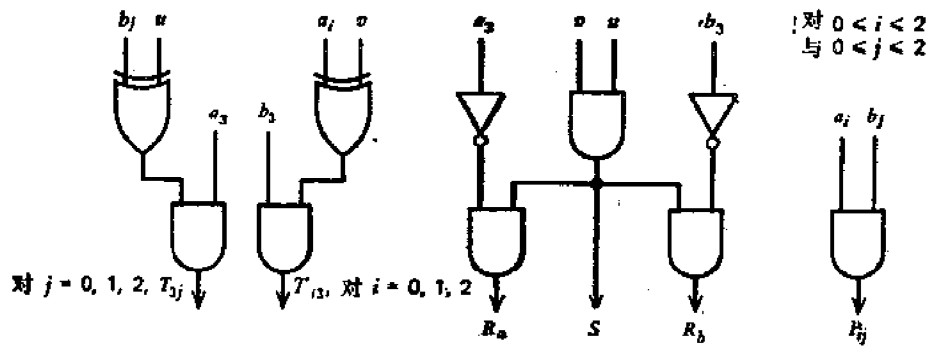
$$
\begin{array}{rcccccccc}
 & & & & & a_3\bar{b}_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & a_3\bar{b}_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & a_3\bar{b}_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & a_3\bar{b}_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
 & & 0 & & & c_3 & c_2 & c_1 & c_0 \\
+) & 0 & 0 & & & d_3 & d_2 & d_1 & d_0 \\
\hline
 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

(c) 方式 $M_{10}(4\times4)$

$$
\begin{array}{rcccccccc}
 & & & & & a_3\bar{b}_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & a_3\bar{b}_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & a_3\bar{b}_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & a_3b_3 & \bar{a}_2b_3 & \bar{a}_1b_3 & \bar{a}_0b_3 & & & \\
\hline
 & & \bar{a}_3 & & & c_3 & c_2 & c_1 & c_0 \\
+) & 1 & \bar{b}_3 & & & d_3 & d_2 & d_1 & d_0 \\
\hline
 & P_7^* & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

(d) 方式 $M_{11}(4\times4)$

图 6.22　可编程序相加乘法模块 $M(4\times4)$ 的四种工作方式用被加数矩阵的说明



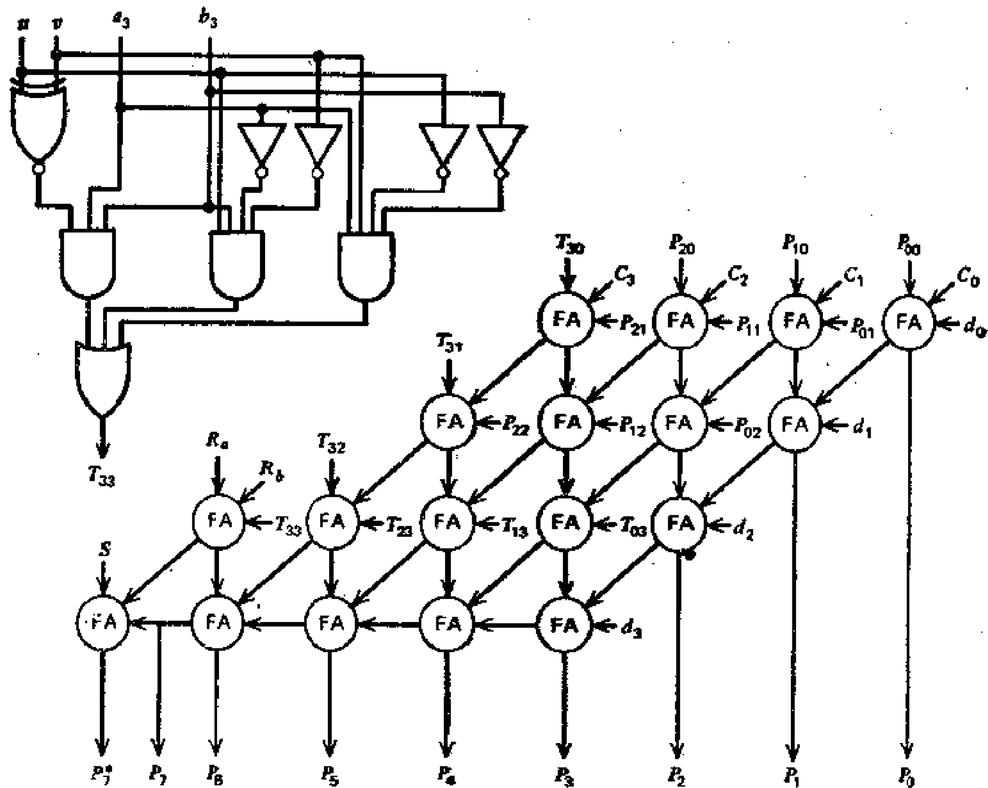


图 6.23 具有方式控制与激励逻辑的 $M(4\times4)$ 可编程序相加乘法模块的原理图 (Hwang[181])

表 6.6 各种规模的可编程序相加乘法 (PAM) 模块的硬件复杂性与应用参数

PAM 模块 $M(k\times k)$	主要硬件元件数		每个封装的外引线数	进位传播延迟[1)]	
	全加器数	激励逻辑门数		PAM 模块 M_{00}, M_{01}, M_{10}	PAM 模块 M_{11}
$M(2\times2)$	6	23	14+1	6△	8△
$M(4\times2)$	10	17	20+1	10△	12△
$M(4\times4)$	18	39	28+1	14△	16△
$M(8\times8)$	26	95	52+1	30△	32△
$M(k\times k)$	k^2+2	$k^2+2k+15$	$6k+5$	$(4k-2)$△	$4k$△

1) 在计算该延迟时,已经排除了激励逻辑,这是因为激励逻辑同时加到所有的 PAM 模块上. 由于激励逻辑所产生的延迟,对整个 PAM 模块的网络来说只能计算一次.

该 PAM 模块 $M(4\times4)$ 需要一个具有 29 条外引线的集成电路封装. 同样的研制技术

可以用于生产标准的不同规模的 PAM 模块，如象 $M(2\times 2)$，$M(4\times 2)$，$M(8\times 8)$ 等. PAM 模块的应用参数归纳在表 6.6 中.

6.8 通用乘法网络

现在讨论用上一节提出的小型 PAM 模块来构成大型通用乘法网络 (UMN) 的叠接方法. 我们用符号 UMN$(n\times n)$ 来表示一个 n 位乘 n 位的 UMN. 必须指出，当 k 较小时(如 $k=4$)，UMA$(k\times k)$ = UMN$(k\times k)$. $M(4\times 4)$ 的 PAM 模块可以编程为四种不同的方式 $M_{00}(4\times 4)$，$M_{01}(4\times 4)$，$M_{10}(4\times 4)$ 或 $M_{11}(4\times 4)$，只用这些模块就能建立起 UMN$(4k\times 4k)$. 在具有其他规模的标准 PAM 模块的设计中也能应用类似的方法.

图 6.24 和图 6.25 绘制了两个 UMN(8×8). 每一种场合都使用了四个 PAM 模块 $M(4\times 4)$. 第一个网络已经被编程为一个不带符号的乘法网络，它用了四个 $M_{00}(4\times 4)$. 我们将简单地把这个不带符号的 UMN 表示为 $N_{US}(8\times 8)$. 第二个网络包含 $M_{00}(4\times 4)$，$M_{01}(4\times 4)$，$M_{10}(4\times 4)$ 和 $M_{11}(4\times 4)$ PAM 模块各一个，它被编程为一个 2 的补码乘法网络，并用符号 $N_{TC}(8\times 8)$ 表示.

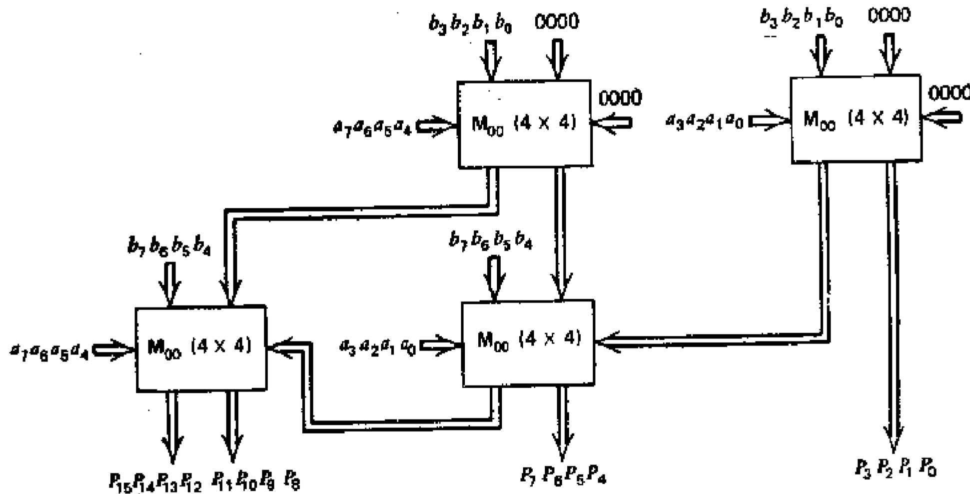


图 6.24 包含四个 $M_{00}(4\times 4)$ PAM 模块的 $N(8\times 8)$ 通用阵列乘法网络，它被编程为一个不带符号的阵列乘法器 $N_{US}(8\times 8)$

这种叠接的线路图可扩充到设计任何 UMN$(4k\times 4k)$ 的网络. 一般地说，一个 UMN$(4k\times 4k)$ 需要 k^2 个 PAM 模块 $M(4\times 4)$. 例如，16 个 PAM 模块 $M(4\times 4)$ 互连可得一个 $N_{TC}(16\times 16)$ 的网络，如图 6.26 所示，它工作在 2 的补码方式. 注意在这两个网络图中，规定用 A_{3-0} 代表二进制数 $a_3a_2a_1a_0$. 在这 16 个模块中间，我们已经编程了九个 $M_{00}(4\times 4)$、三个 $M_{01}(4\times 4)$、三个 $M_{10}(4\times 4)$ 和一个 $M_{11}(4\times 4)$. 当然，用 16 个 $M_{00}(4\times 4)$ 的 PAM 模块也能构成一个不带符号的网络 $N_{US}(16\times 16)$.

在 2 的补码 UMN $N_{TC}(4k\times 4k)$ 中所需要的每一种方式的 PAM 模块 $M(4\times 4)$ 的数量为：$(k-1)^2$ 个 $M_{00}(4\times 4)$，$k-1$ 个 $M_{01}(4\times 4)$，$k-1$ 个 $M_{10}(4\times 4)$ 和一个 $M_{11}(4\times 4)$. 同类模块叠加的结构、通用的可编程序的能力以及在中小规模的 PAM 模块中只要求有限数目的输入/输出引线等，使得这些可编程序的乘法模块具有很大的吸引力. 规模为 16×16 (或更小)的典型 UMA$(n\times n)$ 能做成一个引线数小于 64 的单片集

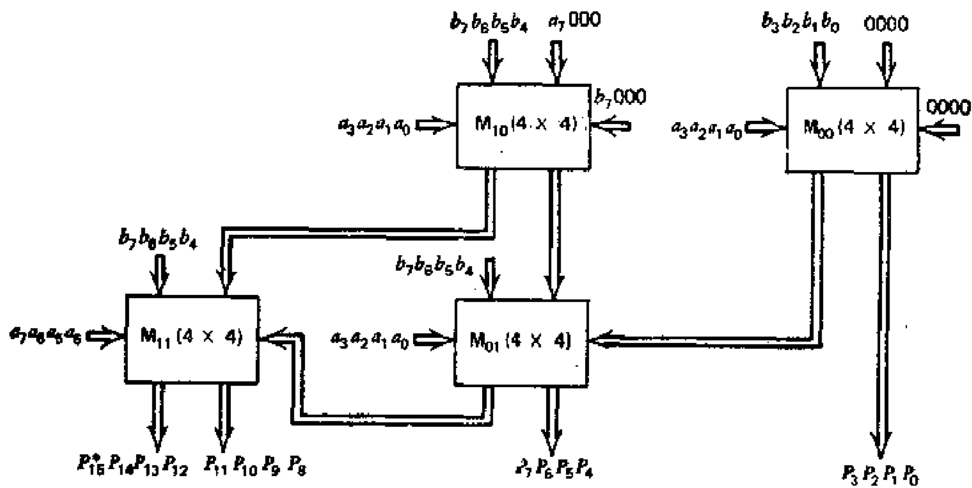


图 6.25 与图 6.10 相同的网络 N(8×8)，重新编程为一个 2 的补码阵列乘法器 $N_{TC}(8\times8)$

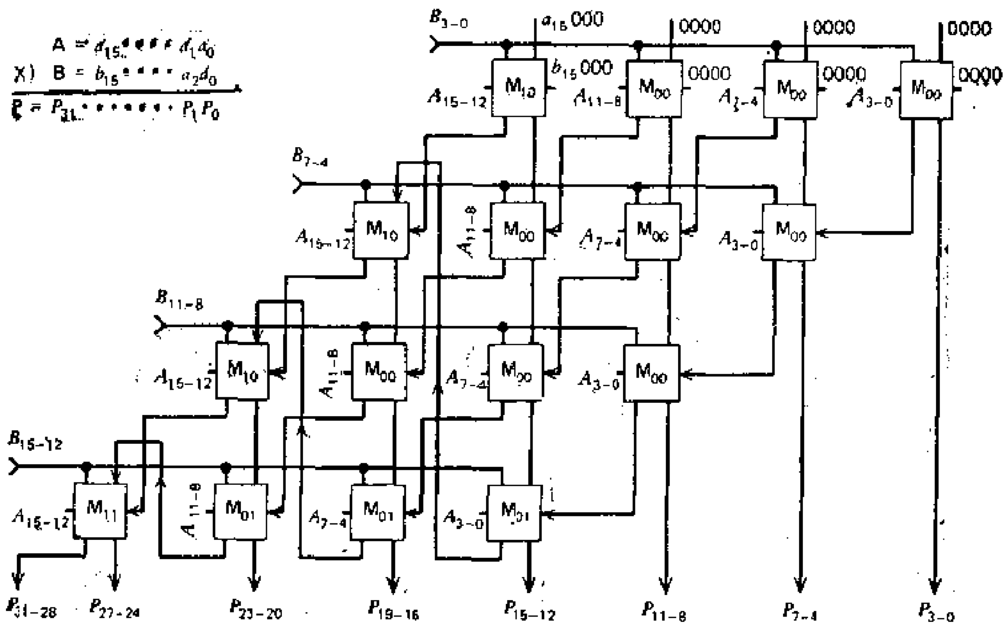


图 6.26 包含 16 个 PAM 模块的 16 位乘 16 位的乘法网络，它被编程为补码方式 (Hwang[18])

成电路. 4 位乘 4 位的标准 PAM 模块则能装在小于 30 条引线的集成电路单片上.

下面我们分析该乘法阵列和网络的速度特性. 在一个双极型的 UMA($n\times n$) 中，所需要的总的乘法时间可按图 6.20 所示的电路来求出，两个求补级每一个为 5Δ 时间延迟，输入激励逻辑为 7Δ 附加延迟. 一旦进入全加器阵列，最长的进位传送途径（沿着最右边的对角线和最底下的一行全加器）给出的延迟为 $[(n-2)+(n+1)]2\Delta=(4n-2)\Delta$. 而且，如果我们在最后一行中使用 CLA，对二级 CLA 加法器来说设有 12Δ 的固定延迟，那么可以求出减小后的阵列延迟为 $(n-2)2\Delta+12\Delta=(2n+8)\Delta$. 总之，UMA($n\times n$) 所需的总乘法时间为

$$\Delta_{UMA}(n\times n)=\begin{cases}(4n+15)\Delta & \text{最后一行没有 CLA}\\(2n+25)\Delta & \text{最后一行有 CLA}\end{cases}\tag{6.36}$$

在 UMA 的四种操作方式中间，不带符号的和补码的操作用得最多. 如果不出现反

码运算，求补逻辑就可以完全取消，从而获得既快而又便宜的电路，它只要用一条方式控制线来识别补码操作和不带符号的操作．总的乘法时间比方程式 6.36 减小了 12Δ，这是由于取消了算前和算后求补电路引起的．

令 $\Delta_{rs}(4\times4)$ 为 PAM 模块 $M(4\times4)$ 工作在 $M_{rs}(4\times4)$ 方式时的进位传送延迟．分析图 6.23 给出的电路，找出 PAM 模块 $M(4\times4)$ 中的延迟，

$$\Delta_{00}(4\times4)=\Delta_{01}(4\times4)=\Delta_{10}(4\times4)=14\Delta,\quad \Delta_{11}(4\times4)=16\Delta.$$

利用这些数值，下面来估算任何一个 $\mathrm{UMN}(4k\times4k)$ 的总乘法时间，假定 $\mathrm{UMN}(4k\times4k)$ 由若干 PAM 模块 $M(4\times4)$ 组成．令 $\boldsymbol{\Delta}_{TC}(4k\times4k)$ 和 $\boldsymbol{\Delta}_{US}(4k\times4k)$ 分别为网络 $N_{TC}(4k\times4k)$ 和 $N_{US}(4k\times4k)$ 的延迟．我们有

$$\boldsymbol{\Delta}_{US}(4k\times4k)=(2k-1)\times14\Delta+7\Delta=(28k-7)\Delta \tag{6.37}$$

$$\boldsymbol{\Delta}_{TC}(4k\times4k)=\Delta_{US}+2\Delta=(28k-5)\Delta \tag{6.38}$$

式中 7Δ 考虑了全加器激励逻辑中的延迟，2Δ 则是由于 $M_{11}(4\times4)$ 模块中附加的延迟．现在已可用高速 TTL 或电流型 ECL 来构成上述阵列乘法器、PAM 模块或乘法网络了．用这两种逻辑系列，每一级逻辑的单位延迟 Δ 可以不到 1 ns、例如，采用 ECL 电路的 Pezaris 乘法器，两级延迟为 $2\Delta=1.6\text{ns}$，因此单位延迟为 $\Delta=1.6\text{ns}/2=0.8\text{ns}$．按照这个单位延迟的数值，最后一行带有 CLA 的 16 位乘 16 位通用阵列乘法器完成乘法的时间为

$$\boldsymbol{\Delta}_{UMA}(16\times16)=(2\times16+25)\Delta=45.6\text{ns}$$

在速度上稍作牺牲，我们还能仿照图 6.26 用一个通用乘法网络 $N(16\times16)$ 来获得 32 位乘积，这时

$$\boldsymbol{\Delta}_{TC}(16\times16)=(28\times4-5)\times0.8=85.6\text{ns}$$

或者 $\boldsymbol{\Delta}_{US}(16\times16)=(28\times4-7)\times0.8=84\text{ns}$．从表 6.7 归纳的数据来看，UMA 的方法要比模块网络提供更快的乘法时间．然而，模块方法在当前的电子学与组装工艺中更具有实际意义．通常可以在操作速度以及积木式模块的规模之间取折衷方案．模块规模的最佳选择要受逻辑系列、制造方法以及系统所要求的通用性和模块化程度的影响．

表 6.7　对于典型规模的阵列和网络，$\mathrm{UMA}(n\times n)$ 和 $\mathrm{UMN}(n\times n)$ 所需要的主要元件数和操作速度

	$\mathrm{UMA}(n\times n)$		$\mathrm{UMN}(n\times n)$[2]	
规模 $n\times n$	全加器需要量	乘法时间[1] $\Delta_{UMA}(n\times n)$	PAM $M(4\times4)$ 的需要量	乘法时间 $\Delta_{TC}(n\times n)$
8×8	53	47Δ	4	51Δ
16×16	237	79Δ	16	107Δ
32×32	989	143Δ	64	219Δ
$n\times n$	n^2-n-3	$(4n+15)\Delta$	$(n/4)^2$	$(7n-5)\Delta$

1) 最后一行没有 CLA．

2) 对 $\mathrm{UMN}(n\times n)$ 来说，其中 $n=4k$．

6.9　再编码阵列乘法

再编码的 2 的补码阵列乘法器是基于本节所要介绍的 Booth 算法．利用 ROM、加法

器和对数表的大型阵列乘法的实际线路则将在下二节中讨论。 Majithia 和 Kita[19] 提出了一种能直接乘补码数的叠接逻辑阵列. 原文是基于分数的补码表示法. 引入一个比例因数后，该阵列可推广到去处理任何补码数. 我们仍用分数表示法来阐明这些阵列的工作原理.

令乘数 $\mathbf{Y} = Y_0.Y_{-1}Y_{-2}\cdots Y_{-(n-1)}$ 是以补码表示的分数. 按照方程式 6.14，$\mathbf{Y}$ 应具有数值 Y_v

$$Y_v = -Y_0 + \sum_{k=0}^{n-1} Y_{-k} \times 2^{-k} \tag{6.39}$$

最后乘积 $\mathbf{Z} = Z_0.Z_{-1}\cdots Z_{-2(n-1)}$ 可以类似地计算. 注意这里假定两个操作数均为 n 位(每一个都包括符号)，因此乘积 Z 有 $2n-1$ 位.

分数乘法的算法可用图 6.27 的流程图来解释. 逐位扫描操作是从最高位 Y_0 开始的，一直要进行到最低位. 在下一次加法或减法的操作之前，将当前的部分乘积的和 $S^{(k)}$ 右

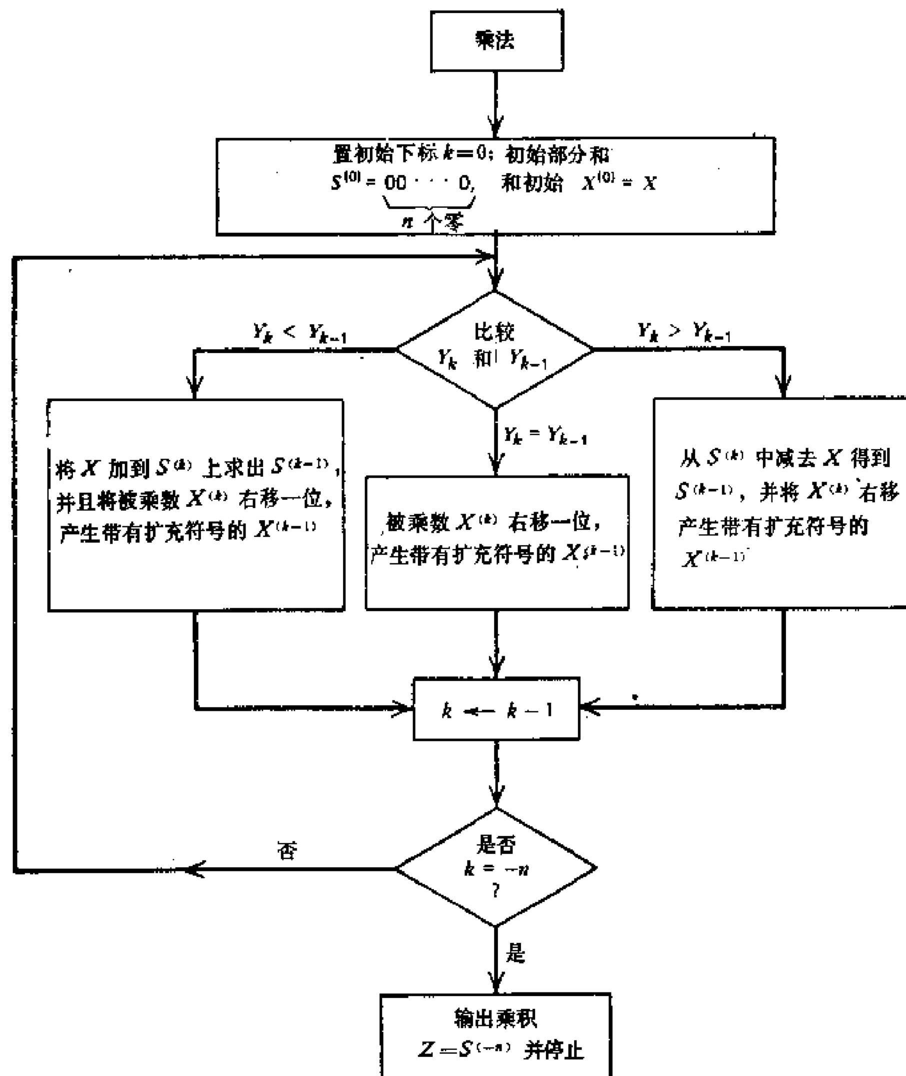


图 6.27 2 的补码阵列乘法的 Booth 再编码乘法算法

移一位就等效于把被乘数 X 右移了一位. 利用被乘数移位可获得同样的最后结果，而其优点是：如果采用存储移位的技术，那么速度便能加快. 应该注意，当补码数右移时，扩充的符号位将被移入已经腾空的最高位的位置上；即如果最高位为“1”，则移入“1”，否则就移入“0”.

可控加法-减法-移位（CASS）单元的设计如图 6.28 所示，它有五个输入和五个输出. 表征该 CASS 单元的逻辑方程式为

$$S = A\oplus BP\oplus CP;\ T = (A\oplus D)(B + C) + BC;\ U = B;\ Q = P;\ R = D \quad (6.40)$$

输入线 A 和 B 传送两个操作数的数位. P 和 D 是两条方式控制线，它控制同一列的所有单元进行同一个操作. 当 $P = 0$ 时，除了把扩充的被乘数通过 B 到 D 右移一位以外，该单元不执行算术运算操作. 当 $P = 1$ 和 $D = 0$ 时，该单元作为一个加法器工作. 当 $P = 1$ 和 $D = 1$ 时，该单元作为一个减法器工作.

两个 3 位补码分数的直接乘法的叠接阵列如图 6.29 所示. 一般地说，如果 **X** 和 **Y** 两者都是包括符号位在内的 n 位数，那么该阵列就需要 $(3n^2 - 2)$ 个 CASS 单元和 n 个 1 位数值比较器. 该设计验证如下：设两个相乘的数为 $\mathbf{X} = 0.01$ 和 $\mathbf{Y} = -0.11$，用补码表示时，$\mathbf{X} = X_0X_{-1}X_{-2} = 001$ 和 $\mathbf{Y} = Y_0Y_{-1}Y_{-2} = 101$，这种情况下 $n = 3$. 详细步骤为

第一步、预置 $k = 0$，$S^{(0)} = 000$ 和 $X^{(0)} = 001$.

第二步、比较 $Y_0 = 1, Y_{-1} = 0$，从 $S^{(0)} = 000$ 中减去 $X^{(0)} = 001$，得到 $S^{(-1)} = 1110$，将 $X^{(0)}$ 移一位，得到 $X^{(-1)} = 0001$.

第三步、比较 $Y_{-1} = 0$ 和 $Y_1 = 1$，将 $X^{(-1)}$ 加到 $S^{(-1)}$ 上得出 $S^{(-2)} = 1111$，使 $X^{(-1}$ 移位得到 $X^{(-2)} = 00001$.

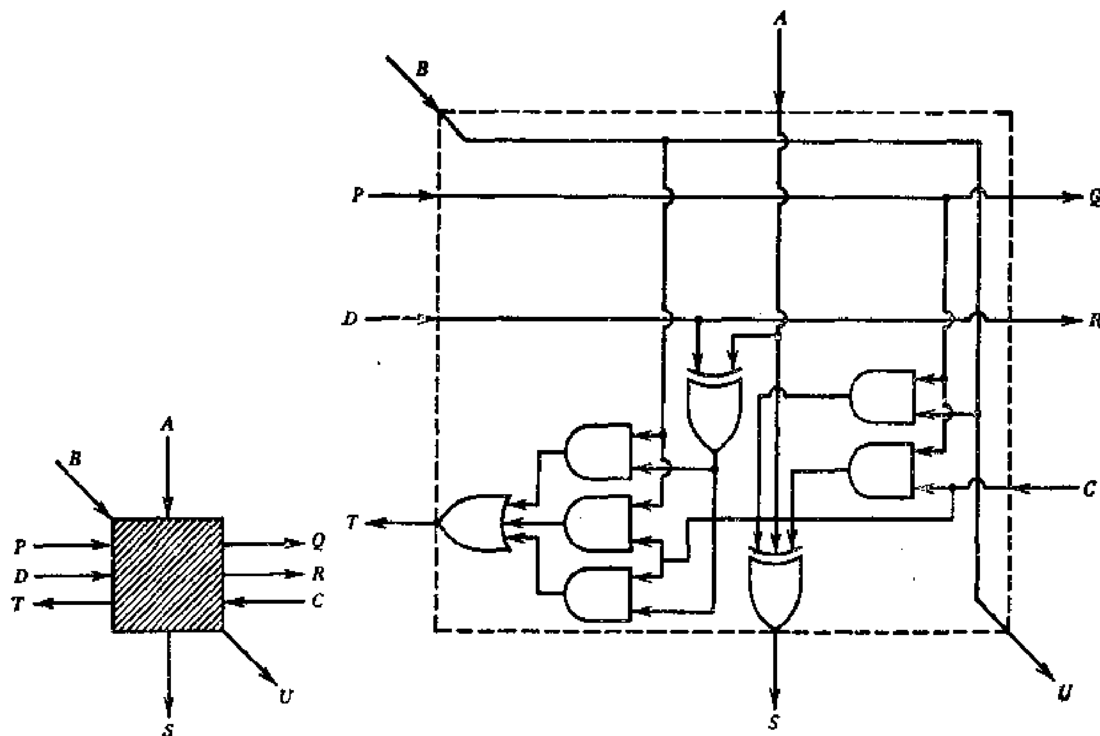


图 6.28 可控加法-减法-移位（CASS）单元的原理逻辑电路图

第四步、比较 $Y_{-2}=1$ 和 $Y_{-3}=0$，从 $S^{(-2)}$ 中减去 $X^{(-2)}$ 得出 $S^{(-3)}=11101$，这就是所要的乘积 **P**，于是到此停止. 答案为 $\mathbf{P}=1.1101$，正如预期的那样，它等于 -0.0011.

该比较器每一个具有延迟 $\Delta_c=3\Delta$. 这意味着每一行的操作方式在 Δ_c 之后被调整. CASS 单元最坏情况下产生输出 S 的延迟为 $\Delta_t=8\Delta$. 乘积的第一位在时间 $\Delta_c+\Delta_t$ 以后出现，第二位在延迟 $\Delta_c+2\Delta_t$ 后出现，第三位在延迟 $\Delta_c+3\Delta_t$ 后出现，等等. 因为有 $2n-1$ 组对角线单元，所以获得整个乘积的总延迟为

$$\Delta_T=\Delta_c+(2n-1)\Delta_t=3\Delta+(2n-1)8\Delta=(16n-5)\Delta \tag{6.41}$$

它的速度比前面的阵列乘法器的速度大约慢四倍，但比起普通的串-并行乘法器仍旧快得多.

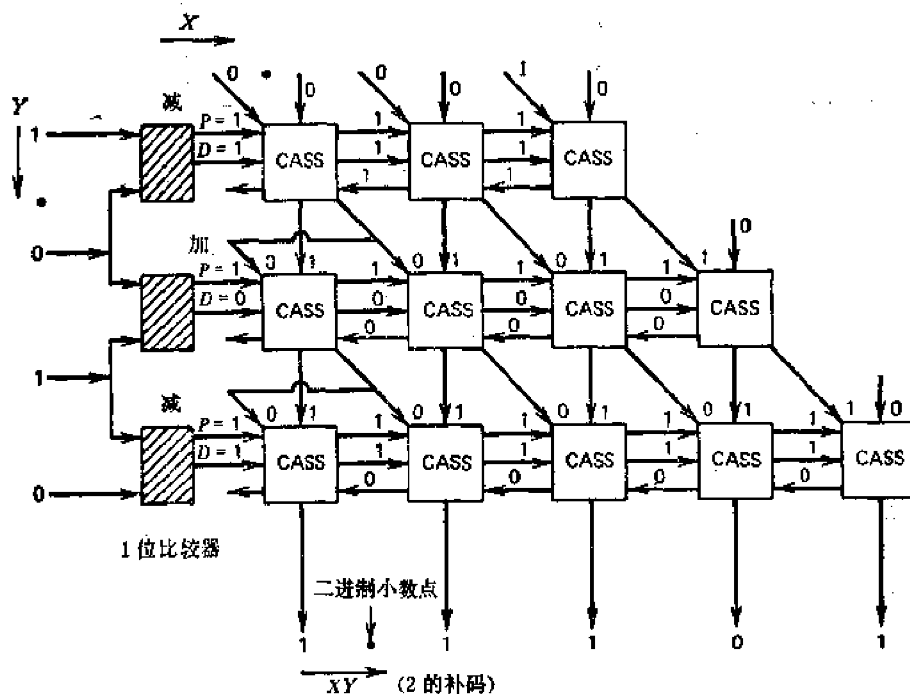


图 6.29 两个 3 位补码分数的再编码乘法所采用的叠接逻辑阵列（Majithia 和 Kita[19]）

6.10 利用 ROM-加法器的乘法网络

由于 ROM 的价格便宜、使用方便，它已广泛地用来作为查表的工具，对于产生任何两个二进制数的乘积来说，这也是一种可行的方法. 乘法表存储在 ROM 中，它包含输入操作数的所有可能组合的乘法结果. 这里主要的困难是 ROM 所需要的位数.

对于直接乘法来说，m 位被乘数和 n 位乘数的每一个组合都确定了存储器中一个唯一对应的地址. 从 2^{m+n} 个可能的字中取出该地址的内容，形成一个 $(m+n)$ 位的乘积，这里假定没有舍入. 如图 6.30 所示，要求 ROM 的总存储容量为

$$N=2^{m+n}\times(m+n) \tag{6.42}$$

即使对中等的 m 和 n 值，例如 $m=n=8$，也要求 ROM 的容量高达 $2^{8+8}\times 16=2^{20}=1,048,576$ 位，为了存储整个乘法表，容量已超过一百万位. 因此，在许多实际应用中，乘积往往被舍入成 p 位，$p<m+n$，比例因子 2^{m+n+p} 已被施加到舍入后的乘积的最

低有效位上．于是所需的存储容量减小到

$$N_t = 2^{m+n} \times p \tag{6.43}$$

这种舍入可能产生的舍入误差 ε 为

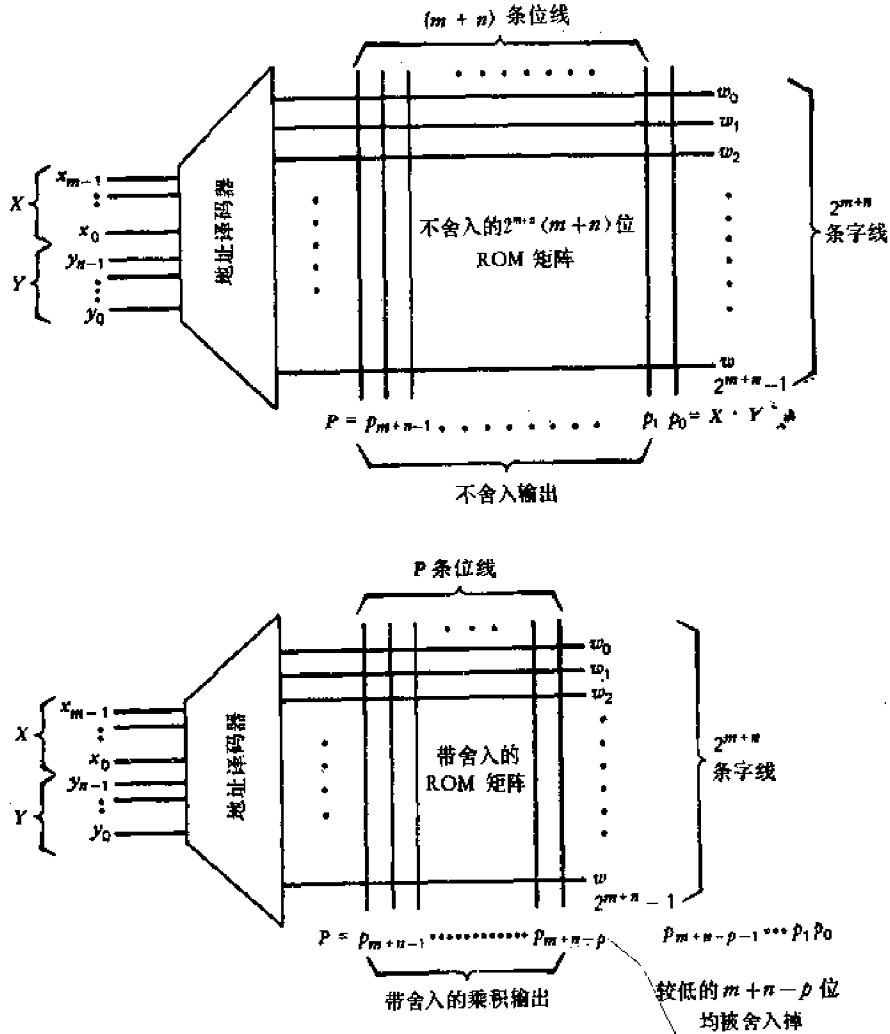


图 6.30　用 ROM 做成带舍入和不带舍入的 m 位乘 n 位的乘法器

$$\frac{-q}{2} < \varepsilon < \frac{q}{2} \tag{6.44}$$

这里 q 表示舍入后乘积的最低有效位的数值．对于不同的 q，p 值和 $m = n = 4, 8, 10$ 等各种情况，ROM 所需要的存储容量的计算结果列于表 6.8．在 $m = n = p$ 的场合，最低的 n 位均被舍入掉．最大误差用输入量的数值范围的百分数来表示时，即为

$$\varepsilon_{\max} = \frac{0.5}{2^n - 1} \times 100 \tag{6.45}$$

$\varepsilon_{\max}$ 的曲线在图 6.31 中给出，它是输入范围 n 的函数．

目前电子存储器工业所生产的标准流行产品中，可供使用的单片 ROM 模块的容量大到 16K 位．例如，容量为 $256 \times 8 = 2048$ 位的标准 ROM 已有几家公司生产，象美国

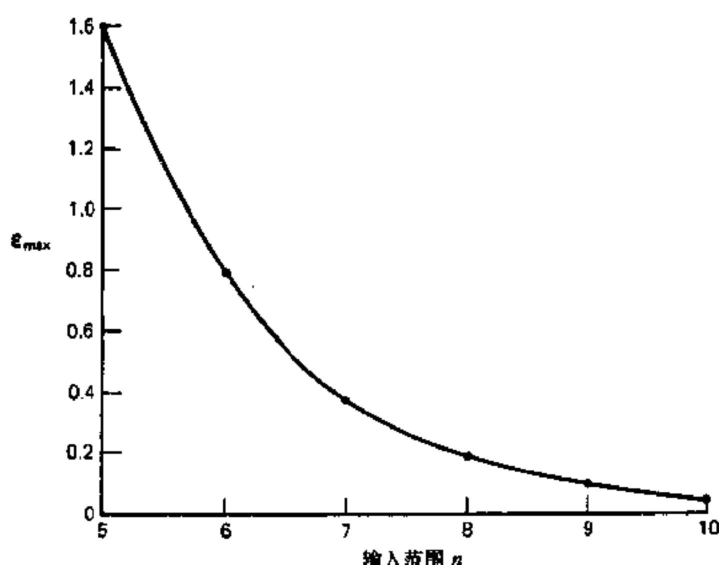


图 6.31 带舍入乘积的最大误差百分数随输入范围 n 的变化曲线

表 6.8 在采用 ROM 的直接乘法中，字长与存储容量之间的关系

输入范围 $m=n$	字长 p	带舍入乘积 q 中最低有效位的数值	存储容量 $N_r = 2^{2m} \times p$
4	8	0	2,048
4	4	2^4	1,024
8	16	0	1,048,576
8	14	2^2	917,504
8	12	2^4	786,432
8	8	2^8	524,288
10	20	0	20,971,520
10	16	2^4	16,777,216
10	10	2^{10}	10,485,760

微系统公司、电子阵列公司和全国半导体公司．这种 2048 位的 ROM 能用于存储整个 4 位乘 4 位的不带舍入的乘法表．大型表格则可用若干较小的存储器模块来构成．为了能使用较小的存储器，就要求把给定的表格分成许多相互独立的段．我们将查表、ROM 以及算术运算功能的划分这三种技术结合起来，便产生一种比较实用的组成大型乘法网络的方法，这种方法只需要用到规模较小的 ROM 和较便宜的加法器．正如前一节所指出的那样，一个不带舍入的 8×8 的乘法表会要求 ROM 的容量超过一百万位．因此，从实际制造的观点来看，很希望把整个表格分成几个较小的子表格．一种办法是把给定的数 $A_8 = a_7a_6a_5a_4a_3a_2a_1a_0$ 分成二个数的和：其中的一个数只取数 A_8 的前半部，其后四位为零，另一个数则取数 A_8 的后半部，前四位为零，即

$$A_8 = A_4^U + A_4^L \tag{6.46}$$

这里 $A_4^U = a_7a_6a_5a_4 0000$ 和 $A_4^L = 0000a_3a_2a_1a_0$．同样可以定义 $B_8 = B_4^U + B_4^L$．利用这些划分好的数，原来的乘法便能重新组成为四个 4×4 的乘法的和．

$$A_8 \times B_8 = [A_4^U + A_4^L] \times [B_4^U + B_4^L] = A_4^U B_4^U + A_4^U B_4^L + A_4^L B_4^U + A_4^L B_4^L \tag{6.47}$$

这四个乘积以及它们的和能用四个 2048 位的 ROM 和五个 4 位的行波进位加法器来实现，它的框图如图 6.32 所示．图中的 ROM 和加法器均采用标准的流行产品，如象全国半导体公司生产的 MM523 型 256 × 8 位的 ROM 和 Texas Instruments (TI) 公司的 SN7483 型 4 位全加器．前四个零和后四个零在电路中相当于不连接．这些零决定着在所有各段的乘积项中列方向的关系．外接的提升电阻在电路中没有标出，这些电阻是使 MOS 存储器的输出信号能与 TTL 加法器的输入要求相兼容所必需的．此外，±12V 电源是供 MOS ROM 用的，而 5V 电源只供 TTL 器件用．尽管增加了这些兼容性和电源种类方面的要求，ROM-加法器方案仍被认为是便宜的，也是容易实现的．

表 6.9 提供了 $4k$ 位乘 $4k$ 位乘法网络所需要的 ROM 和加法器数量的计算值．一般地说，一个 $4k$ 位乘 $4k$ 位的 ROM-加法器乘法网络可以仿照图 6.32 来构成，它需要 k^2 个 256 字节的 ROM 模块和 $k(3k-1)/2$ 个 4 位加法器．这种 ROM-加法器乘法网络的总延迟将为

$$\Delta_{RA} = \Delta_r + (2k-1)\Delta_a \tag{6.48}$$

这里 Δ_r 为 ROM 的存储器取数时间，Δ_a 为 4 位加法器中总的进位延迟．按照现有的存

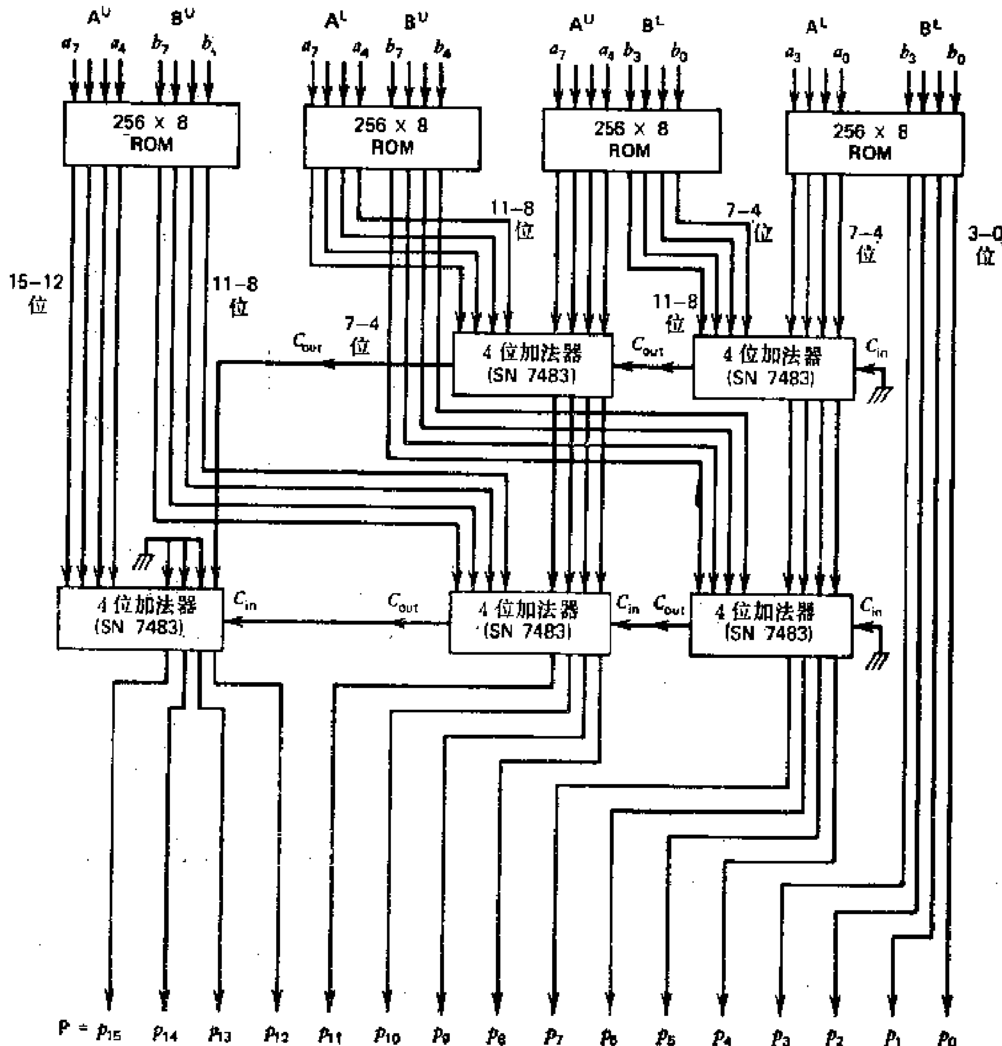


图 6.32 用 256 字节的 ROM 和 4 位加法器来实现 8 位乘 8 位的阵列乘法器 (Hemel[16])

储器和加法器的工艺，8 × 8 的 ROM-加法器乘法网络的速度可小于 1 μs，因此这个图对许多小型计算机和微型计算机都是适合的.

表 6.9′ 用 4 位加法器和 256 字节的 ROM 组成的乘法网络对硬件的需要量

参 数 k	网络规模 $4k \times 4k$	256 字节 ROM 的需要量	4 位加法器的需要量
1	4×4	1	0
2	8×8	4	5
3	12×12	9	12
4	16×16	16	22
6	24×24	36	51
8	32×32	64	100

6.11 对数乘法/除法的原理

减小 ROM 乘法表容量的另一个方法是：采用对数及反对数变换，把乘/除转换成加/减. 原先所要存储的整个表格现在换成只需要存储对数、反对数和加法表，这些表格通常只要求一些较小的 ROM. 用对数来实现乘/除是基于下列关系式

$$A \times B = \mathbf{antilog}(\mathbf{log}A + \mathbf{log}B) \tag{6.49}$$

$$A/B = \mathbf{antilog}(\mathbf{log}A - \mathbf{log}B) \tag{6.50}$$

式中 A 和 B 为二进制数，对数采用以 2 为底. 注意 A 和 B 只取绝对值，最后乘积的正确符号假定另外产生.

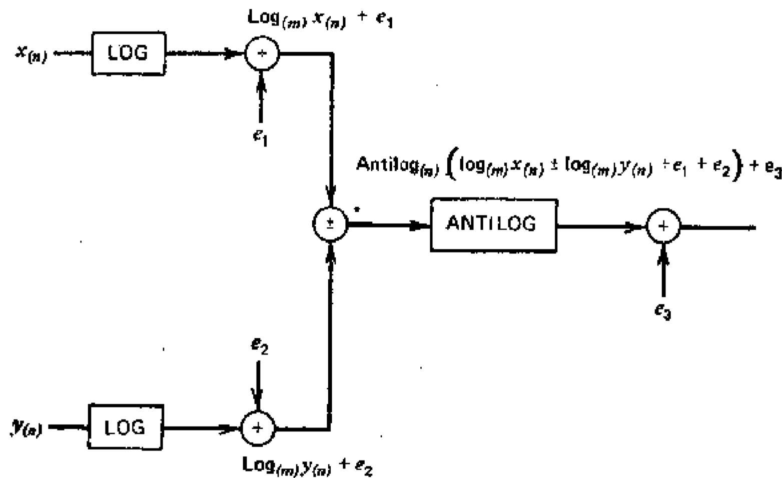


图 6.33 说明所有可能的误差来源的对数乘法模型（*对乘法用加，对除法用减）

采用查找对数表的乘法器/除法器的完整模型可用图 6.33 来描述，它指明了所有的误差来源. 在 $X_{(n)}$，$Y_{(n)}$，$\mathbf{log}_{(m)}$ 和 $\mathbf{antilog}_{(n)}$ 的括弧中，下标用来表示两个输入操作数以及相应的带舍入的对数和带舍入的反对数的字长. 误差 e_1，e_2 和 e_3 是用 ROM 实现带舍入的对数和带舍入的反对数时产生的. 加法/减法可以用任一种高速硬件并行加法器或通过查表的方法来实现. 图 6.34 中绘出了最大误差作为半范围的百分数随着带舍入对数的字长 m 而变化的函数关系，图中以乘积的位数 n 作为参变量.

表 6.10 中的各项指出了乘积和对数字长的各种不同组合下，ROM 所需要的位数，

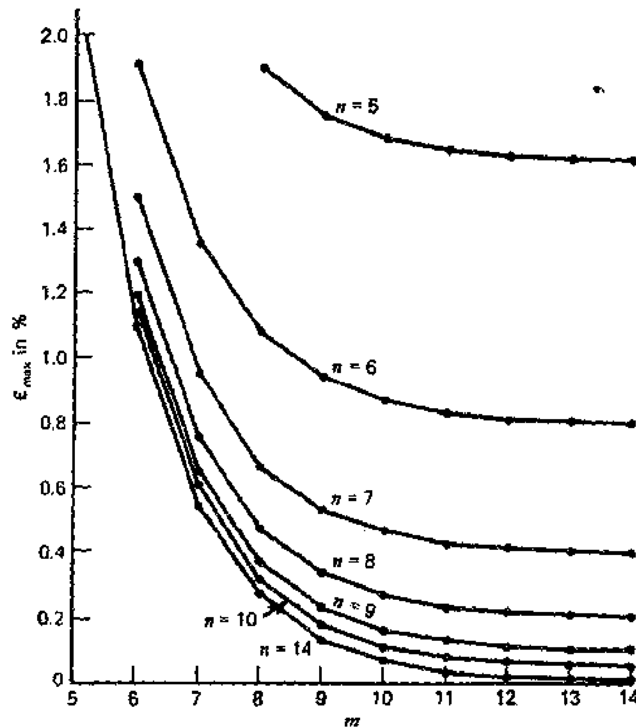


图 6.34 作为半范围百分数的最大误差 ε_{max} 随参数 m 的变化曲线，这里以 n 为参变量（Brubaken 等[6]）

ROM 的最佳容量可用图 6.34 和表6.10来决定．例如，假定误差为 0.5%，那么就有几种不同的 m 和 n 的组合可以工作．一种组合是 $m = n = 8$ 位，它给出位的需要量只有 6144 位，而用一个 8×8 的直接乘法表在采用半范围舍入时，则如表 6.8 所示需要 524288位，更不用说不带舍入的直接乘法，后者将超过一百万位．存储容量减少的倍数达 $524288 \div 6144 = 85.3$，显然是很可观的，这对于那些允许有小误差的应用来说显得尤其重要．

使用 ROM 的直接乘法/除法需要访问一次存储器，而对数乘法则需访问二次，还要外加一次加法时间．因此，对数乘法的速度大致是查表直接乘法的速度的三分之一．此外，该速度也受存储器工艺的支配．现有的对数乘法/除法的速度已到达几百毫微秒．

表 6.10 对数乘法所需要的 ROM 的总位数，它是带舍入对数和乘积字长的函数

log 的位数 (m)	乘积的位数 (n) 6	7	8	9	10	11	12	13	14
5	832	1,504							
6	1,152	1,984	3,584						
7	1,664	2,688	4.608	8.320					
8	2,560	3,840	6.144	10,496	18,944				
9	4,224	5,888	8,704	13,824	23,552	42,496			
10		9,728	13.312	19,456	30,720	52,224	94,208		
11			22,016	29,696	43.008	67,584	114,688	206,848	
12				49,152	65,536	94,208	147,456	249,856	450,560
13					108,544	143,360	204,800	319,488	540,672
14						237,568	211,296	442,368	688,128

6.12 参考文献注释

本章题材的范围很宽，涉及了用于高速并行乘法的各种叠接阵列和网络线路．早期关于整数阵列乘法的处理可以在 Braun[5] 以及在 Chu[7] 的文献中找到．较晚一些的则有 Wallace[28] 和 Dadda[8]，以及其他研究者[9,14,15,22]，他们推广了 Braun 的原始设计．现有基本的乘法积木式模块及其网络应用的介绍可在许多电子厂商的手册中找到，如象 4 × 2 的乘法单元可参阅先进微器件公司[1]、特克萨斯仪器公司[26]和仙童半导体公司的手册[11]；4 × 4 乘法模块可参阅特克萨斯仪器公司的手册[26]；8 × 8 单片乘法器可参阅休斯飞机公司的手册[17]．大规模集成电路的单片 16 位乘 16 位乘法器也已经由 TRW[27] 和 McIver[20] 宣布．

间接的和直接的带符号补码阵列乘法器曾经由 Deegan[10]，Gibson 和 Gibbard[13]，Mowle[23] 以及 Toma[27] 等人作过讨论．麻省理工学院林肯实验室用混合型全加器做成的 17 位乘 17 位的阵列乘法器是由 Pezaris[25] 报道的，它采用 2 的补码数直接乘法．Baugh-Wooley[3] 提出了一个均匀的补码直接乘法的算法．对 Pezaris 原设计的几种可行的修改方案有：采用 Baugh-Wooley 算法的门控全加器方案以及作者提出的可编程序通用并行乘法网络[18]．Majithia 和 Kita[19] 的阵列乘法器的结构则是基于 Booth算法[4]．采用 ROM-加法器经济地实现并行乘法的方法可从 Hemel 的著作[16]中读到．最后，通过查表的方法来实现对数乘法以及用 ROM 来组成的线路图则是根据 Michell[21] 以及 Brubaker 和 Becker[6]的论文．最近关于阵列乘法器的论文可参阅 Agrawal[2]．

参 考 文 献

[1] Advanced Micro Dvices, "TTL/MSI AM2505 4-bit by 2-bit 2's Complement Multiplier," 901 Tompson Pace, Sunnyvale, CA.

[2] Agrawal, D. P., "Optimum Array-Like Structures for High-Speed Arithmetic," *Proc. of 3rd Symposium on Computer Arithmetic*, IEEE Computer Society, # 75C1017-3C, Nov. 1975, pp. 208—219.

[3] Baugh, C. R. and Wooley, B. A., "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Trans. Computers*, Vol. C-22, No. 1—2, December 1973, pp. 1045—1047.

[4] Booth, A. D., "A Signed Binary Multiplication Algorithm," *Quart. J. Mech. Appl. Math.*, Vol. 4, Pt. 2, 1951 pp. 236—240.

[5] Braun, E. L., *Digital Computer Design*. Academic Press, New York, 1963.

[6] Brubaker, T. A. and Becker, J. C., "Multiplication Using Logarithms Implemented with Read-Only Memory," *IEEE Trans. Computers.*, Vol. C-24, 1975.

[7] Chu, Y., *Digital Computer Design Fundamentals*, McGraw-Hill, New York, 1962.

[8] Dadda, L., "Some Schemes for Parallel Multipliers," *Alta Frequenza*, Vol. 34, March 1965, pp. 349—356

[9] Dean, K. J., "Design of a Full Multiplier," *Proc. IEEE*, Vol. 115, Nov. 1968, pp. 1592—1594.

[10] Deegan I. D., "Cellular Multiplier for Signed Binary Numbers," *Electronic Letters*, Vol. 7, 1971, pp. 436—437.

[11] Fairchild Semiconductors, "TTL/MSI 9344 Binary (4-bit by 2-bit) Full Multiplier," 313 Fairchild Dr.; Mountain View, CA., 1971.

[12] Flores, I., *The Logic of Computer Arithmetic*, Prentice Hall, Englewood Cliffs, N. J., 1963.

[13] Gibson, J. A. and Gibbard, R. W., "Synthesis and Comparison of Two's Complement Parallel Multipliers," *IEEE Trans. Computers*, Vol. C-24, Oct. 1975, pp. 1020—1027.

[14] Guild, H. H., "Fully Iterative Fast Array for Binary Multiplication and Fast Addition," *Electronic Letters*, Vol. 5, May 1969, p. 263.

[15] Habibi, A. and Wintz, P. A., "Fast Multipliers," *IEEE Trans. Computers*, Vol. C-19, Feb. 1970, pp. 153—157.
[16] Hemel, A., "Making Small ROMs Do Math Quickly, Cheaply and Easily," *Electronic Computer Memory Technology*, W. B. Riley, (ed.), McGraw-Hill, New York, 1971, pp. 133—140.
[17] Hughes Aircraft Co., "Bipolar LSI 8-bit Multiplier H1002MC." 500 Superior Avenue, Newport Beach, CA., 1972.
[18] Hwang, K., "Global Versus Modular Two's Complement Array Multipliers," *IEEE Trans. Computers*, to appear.
[19] Majithia, J. C. and Kita, R., "An Iterative Array for Multiplication of Signed Binary Number," *IEEE Trans. Computers*., Vol. C-20, Feb. 1971, pp. 214—216.
[20] McIver, G. W. et al., "A Monolithic 16×16 Digital Multiplier," *Dig Tech. Paper Int. Solid State Circuits Conf.*, Feb. 1974, pp. 54—55.
[21] Michell, J. N., "Computer Multiplication and Division Using Binary Logarithms," *IRE Trans Elec. Computers*, Vol. EC-11, Aug. 1962, pp. 512—517.
[22] Mori, R. D., "Suggestion for an IC Fast Parallel Multiplier," *Electronic Letters*, Vol. 5, Feb 1969, pp. 50—51.
[23] Mowle, F. J., *A Systematic Approach to Digital Logic Design*, Addison Wesley, Reading, Mass., 1976.
[24] National Semiconductor Corp. *Data Sheets* MM 521, MM 522, MM 523, and DM 8200.
[25] Pezaris, S. D., "A 40ns 17-bit-by-bit Array Multiplier," *IEEE Trans. Computers*, Vol. C-20, No. 4, April 1971, pp. 442—447.
[26] Texas Instruments, Inc., *TTL Databook and Supplement to TTL Databook*, T. I., Dallas, Texas 1974, pp. 496—498, S262—S270.
[27] TRW, "MPY-LSI Multipliers: AJ 8×8, 12×12 and 16×16," LSI Products, TRW, Redondo Beach, Calif., March 1977.
[28] Wallace, C. S., "A Suggestion for Parallel Multipliers," *IEEE Trans. Electronic Computers*, Vol. EC-13, Feb. 1964, pp. 14—17.

习　　题

题6.1　试画出不带符号的16位乘16位 Braun 阵列乘法器的原理框图，要求阵列的最后一行用二级先行进位。当假设 $\Delta = 5ns$ 时，求出该乘法器的速度。

题6.2　试用进位存储全加器来建立下列规模的华莱士树：

(a) 9位位片式华莱士树；

(b) 15位位片式华莱士树。

题6.3　试用下列元件来构成一个12位乘12位的阵列乘法器，这些元件是：九个4位乘4位非相加乘法模块（NMM），多至5个输入的华莱士树以及两个进位传播加法器。证明你所设计的原理框图在形式上类似于图6.6。这里规定所有数据线的宽度按照它们的数位所在位置的权与输入、输出一致。

题6.4　试画出用十六个8位乘8位的相加乘法模块（AMM）组成的32位乘32位的乘法网络的原理设计图。求出你的设计的总乘法时间，这里假定每个 AMM 的延迟为 40 ns。

题6.5　证明按方程式6.14定义的带负号的补码表示法确实有效，即 $N_e + (-N_e) = 0$，这里 $-N_e$ 为 $-N$ 的数值。

题6.6　试用0类、1类、2类和3类全加器构成一个8位乘8位的 Pezaris 阵列乘法器，并用下面的一对2的补码数

$$A = 10110111 \quad B = 01110110$$

来验证你的设计。说明阵列中所有单元的输入/输出值与最后乘积一样，都是以补码表示的形式。

题6.7　重复题6.6，要求只用0类、1类和2类全加器单元来构成一个8位乘8位的三段阵列乘法器。

题6.8　重复题6.6，要求只用0类、1类和2类全加器来构成一个8位乘8位的二段阵列乘法

器.

题 6.9 重复题 6.6，要求只用 0 类全加器来构成一个 8 位乘 8 位的 Baugh-Wooley 阵列乘法器.试画出包括所有激励逻辑在内的原理设计图，并且对阵列中所有中间级用两个给定的操作数 A 和 B 来验证该设计.

题 6.10 给定 16 位乘 16 位可编程序的相加乘法模块（PAM）——$M(16 \times 16)$．试用 16 个这样的 PAM 来构成一个 64 位乘 64 位的通用乘法网络．分别求出所得到的网络在两种工作方式下的乘法时间 $\Delta_{US}(64 \times 64)$ 和 $\Delta_{TC}(64 \times 64)$.

题 6.11 试用 256 字节的 ROM 和 4 位全加器（SN7483）来构成一个 12 位乘 8 位的阵列乘法器．在 $\Delta_{ROM} = 300ns$ 和 $\Delta_{adder} = 20ns$ 的假定之下，估算出你的设计方案的速度，并对你的 ROM 加法器网络的原理图给予说明.

第七章　标准的和高基数的除法器

7.1　引　　言

虽然除法是乘法的逆运算，但是它在许多方面与乘法不同．首先，除法是移位与减除数的操作，与乘法对比，后者是移位与加被乘数的操作．每次减法（比较）的结果决定着除法程序中的下一次操作．因此，除法在各个操作周期之间有着内在的串行依存性．这个问题在乘法中并不存在，因为乘法中所有被加数是同时产生的．第二，除法不是一个很确定的过程，而是一个反复试验的过程．从一个数字集合，经过对每位数字的鉴别过程，选出商的逐位数字．

机器中的除法过程一般包括三个部分：操作数的预置，商的产生，以及余数的确定．开始一步要求被除数和除数都标准化，并检查是否可能出现商的溢出．商的各位是从最高位到最低位逐位选出的．余数则通常在商的产生过程结束时自动得出．对于改善除法效率的种种努力集中在寻找一些既快而又方便的办法来产生商的各位数字．

根据所产生商的每一位数字的允许值，可以把除法线路分成四类．在具有基数 r 的**带恢复的除法**中，商的每一位数字是从普通的数字集合

$$\{0, 1, 2, \cdots, r-1\} \tag{7.1}$$

中选择的．**不恢复的除法**，商的各位数字是从带符号的数字的集合

$$\{-(r-1), -(r-2), \cdots, -1, +1, \cdots, r-2, r-1\} \tag{7.2}$$

中选出的，这里零除外．在基数为 2 的 **SRT 除法**中，带符号数字的集合为

$$\{-1, 0, +1\} \tag{7.3}$$

包括零在内．基数为 r 的**一般化的 SRT 除法**则用了商的数字的集合

$$\{-m, \cdots, -1, 0, 1, \cdots, m\} \tag{7.4}$$

这里

$$\frac{r-1}{2} \leqslant m \leqslant r-1$$

本章将研究这四种"比较-移位"除法的原理以及实现的方法．在下一章中，将研究一些特殊的方法，如利用乘法通过收敛或求倒数来实现除法，和利用叠接单元阵列的高速除法等．

7.2　基本的"减法-移位"除法的特性

目前大多数数字计算机中，除法指令的执行是借助于一种所谓递归步骤．数字除法所需的时间主要耗费在递归步骤的重复执行上．各种除法都可以用下列递归公式来描述

$$R^{(j+1)} = r \times R^{(j)} - q_{j+1} \times D \tag{7.5}$$

这里 $j = 0, 1, \cdots, n-1$ 为**递归下标**

D 为除数

q_{j+1} 为小数点右边第 $j+1$ 位商

n 为商的字长，q_0 为符号

r 为基数

$r \cdot R^{(j)}$ 为确定第 $(j+1)$ 位商之前的**部分被除数**

$R^{(j+1)}$ 为确定了第 $(j+1)$ 位商之后的**部分余数**

$R^{(0)}$ 为被除数（初始部分余数）

$R^{(n)}$ 为最后余数

我们假定被除数 $R^{(0)}$ 和除数 D 都是分数（这样假定并不丧失一般性），于是所产生的商 Q

$$Q = q_0 \cdot q_1 q_2 \cdots q_{n-1} q_n \tag{7.6}$$

式中 q_0 为商的符号，它取决于下列操作

$$q_0 = r_0^{(0)} \oplus d_0 \tag{7.7}$$

这里 $r_0^{(0)}$ 和 d_0 分别为被除数和除数的符号. 小数点位于符号 q_0 和最高位 q_1 之间. 最后余数可以是正的，也可以是负的，这决定于所用的方法. 对于常规的带恢复的除法，余数的符号与被除数是一致的.

为了使讨论简化，被除数和除数两者均假定为正的分数，即 $r_0^{(0)} = d_0 = 0$，根据方程式 7.7，商 $q_0 = 0$.

当较大的被除数 $R^{(0)}$ 被较小的除数 D 除时，所得到的商会超过一个 n 位字所能表示的最大分数值. 这样一种商的溢出是由于下列条件引起的.

$$R^{(0)} \geqslant D \tag{7.8}$$

溢出意味着需要有比 n 位更多的数位才能表示这个商，它的数值已不小于 1. 当出现溢出时，机器的除法器将会发出一个误差信号. 我们在整个这一章中都蕴含地假定了 $R^{(0)} < D$ 和 $D \neq 0$. 对于浮点运算（在第九章中介绍），我们将进一步假定规格化除数的最高位为非零值，即 $d_1 \neq 0$. 通过把被除数或除数适当移位，上述假定常常可以得到满足.

除法过程可通过重复应用递归方程式 7.5 来得到证明.

对 $j = 0$

$$R^{(1)} = r \times R^{(0)} - q_1 \times D \tag{7.9}$$

对 $j = 1$

$$R^{(2)} = r \times R^{(1)} - q_2 \times D = r^2 \times R^{(0)} - (r \times q_1 + q_2) \times D \tag{7.10}$$

对 $j = n-1$

$$R^{(n)} = r^n \times R^{(0)} - (r^{n-1} \times q_1 + r^{n-2} \times q_2 + \cdots + q_n) \times D \tag{7.11}$$

上述叠代推导过程说明了除法过程是由一系列加法、减法或移位所组成，它们相应于逐位产生的商 q_{j+1} 是负、正或零值，这里 $j = 0, 1, \cdots n-1$. 方程式 7.11 可改写如下：

$$\frac{R^{(0)}}{D} = \sum_{i=1}^{n} r^{-i} \times q_i + \frac{r^{-n} \times R^{(n)}}{D} \tag{7.12}$$

这里，商为

$$Q = \sum_{i=1}^{n} r^{-i} \times q_i \tag{7.13}$$

最后余数为

$$R = r^{-n} \times R^{(n)} \tag{7.14}$$

它由余数 $R^{(n)}$ 右移 n 位求出. 商的选择过程是基于下列运算条件之一. 对于带恢复的除法,我们用

$$0 \leqslant R^{(j+1)} < D \tag{7.15}$$

对于不恢复的除法,我们用

$$|R^{(j+1)}| \leqslant |D| \tag{7.16}$$

对于一般化的 **SRT** 除法,我们用

$$|R^{(j+1)}| \leqslant k \times |D| \tag{7.17}$$

这里 $\frac{1}{2} \leqslant k \leqslant 1$. 商的各种选择判据细节将在以后各节中讨论.

7.3 常规的带恢复的除法

常规的基数为 r 的除法利用了数字集合 $\{0, 1, 2, \cdots, r-1\}$. 商的每一位

$$q_{j+1}(j = 0, 1, \cdots, n-1)$$

的数值的选择应满足方程式 7.15, 从而保证不出现溢出条件(方程式 7.8). 商的选择判据可以这样实现, 即从当前的部分被除数 $r \times R^{(j)}$ 中重复地减去除数 D,直到差成为负数为止. $r \times R^{(j)}$可以简单地通过把当前的部分余数 $R^{(j)}$ 左移一位数字来得出.在余数转为负值之前,执行减法的次数决定了商的被选位的数值. 附加的一步是要求恢复新的部分余数 $R^{(j+1)}$,后者必须满足方程式7.15.恢复是用一次加法来完成的, 即把除数加到减法过程结束时所得到的负的差值上.

需要做 $k+1$ 次减法和一次加法,才能决定商的一个数位, 这里 $k \in \{0, 1, \cdots, r-1\}$. 因此,在最坏情况下,产生一位商可能需要 r 次减法和一次加法. 这个过程与习惯上用纸和笔进行的十进制除法是一样的,如图 7.1 中的说明.

对二进制运算来说,基数 $r = 2$, 这时上述恢复过程可以大大简化. 二进制带恢复的除法每产生一位商至多只要一次减法和一次加法.在二进制的情况下,方程式 7.5 可以写成

给定: 被除数 $R^{(0)} = 0.1257$ 与
除数 $D = 0.39$ 对 $r = 10$
求出: 商 $Q = 0.q_1q_2$ 对 $n = 2$,
余数 $R^{(2)} = 0.r_1r_2 \times 10^{-2} = 0.00r_1r_2$

$R^{(0)} =$	0.1257		
$r \cdot R^{(0)} =$	1.257		
$-D =$	-0.39	第一次减法	$q_1 = 3$
	0.867	>0	
$-D =$	-0.39	第二次减法	
	0.477	>0	
$-D =$	-0.39	第三次减法	
	0.087	>0	
$-D =$	-0.39	第四次减法	
	-0.69	<0	
$+D =$	$+0.39$	恢复用加法	
$R^{(1)} =$	0.087		
$r \cdot R^{(1)} =$	0.87		
$-D =$	-0.39	第一次减法	$q_2 = 2$
	0.48	>0	
$-D =$	-0.39	第二次减法	
	0.09	>0	
$-D =$	-0.39	第三次减法	
	-0.70	<0	
$+D =$	$+0.39$	恢复用加法	

$R^{(2)} = 0.09 \times 10^{-2} = 0.0009$ 余数

商 $Q = 0.q_1q_2 = 0.32$

图 7.1 常规的十进制(基数=10)带恢复的除法的一个数值例子

$$R^{(j+1)} = 2R^{(j)} - q_{j+1} \times D \tag{7.18}$$

这里每一位商 $q_{j+1} \in \{0, 1\}$. 方程式 7.15 给出的选择条件即成为

$$q_{j+1} = \begin{cases} 0 & \text{如果 } 2R^{(j)} < D \\ 1 & \text{如果 } 2R^{(j)} \geqslant D \end{cases} \tag{7.19}$$

这个鉴别过程不难用一次减法求出暂定的部分余数来实现

$$\underline{R}^{(j+1)} = 2R^{(j)} - D \tag{7.20}$$

如果 $R^{(j+1)}$ 的符号为正，那么

$$q_{j+1} = 1;\ R^{(j+1)} = \underline{R}^{(j+1)} \tag{7.21}$$

否则，我们得到 $q_{j+1} = 0$，并且必须用一次加法来恢复正确的部分余数

$$R^{(j+1)} = \underline{R}^{(j+1)} + D = 2R^{(j)} \tag{7.22}$$

带恢复的二进制除法可以用 Robertson 图来说明，如图 7.2 所示. 在这个图中，水平轴指示当前的部分被除数 $2R^{(j)}$ 的数值. 垂直轴则相应于下一个部分余数 $R^{(j+1)}$. 所有的值都用除数 D 的倍数来度量.

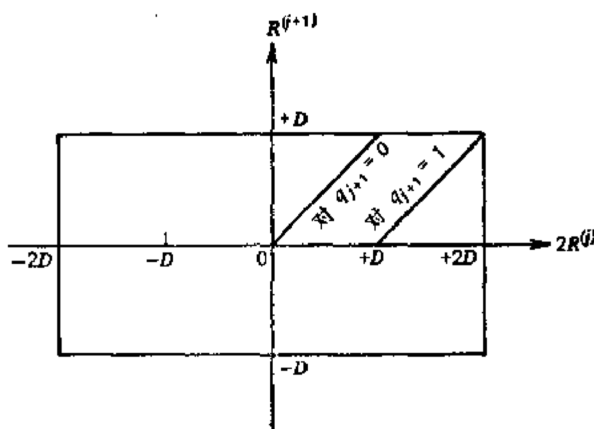


图 7.2 具有数位集合 {0, 1} 的带恢复二进制除法的 Robertson 图

右边的直线 $2R^{(j)} - D = R^{(j+1)}$ 指示 $q_{j+1} = 1$，因为 $D \leqslant 2R^{(j)} < 2D$，下一个部分余数为 $2R^{(j)} - D$. 左边的直线 $2R^{(j)} = R^{(j+1)}$ 指示 $q_{j+1} = 0$，因为 $0 \leqslant 2R^{(j)} < D$，且余数为 $2R^{(j)}$. 上述带恢复的二进制除法在每次叠代开始时强制选择 $q_{j+1} = 1$. 当初始猜测不正确时，在恢复阶段就要付出代价.

总之，如果假定产生商的数位为"0"或为"1"的概率相等，那么加法次数平均为 $n/2$ 次，而减法必定为 n 次. 为了逐个地产生部分被除数 $2R^{(j)}$，这里 $j = 0, 1, \cdots, n-1$，总共要求 n 次左移一位. 我们将指出，用具体的电路实现时，依靠移位器来进行存储移位，旧的部分余数就可以得到保留，而用于恢复的加法则可以绕过.

7.4 二进制带恢复的除法器的设计

这一节要讨论执行二进制除法指令的运算处理器的设计，这里采用了刚才描述过的带恢复的方法. 该部件包括三个 n 位的工作寄存器，即**累加器**（AC）、**辅助寄存器**（AX）和**商-乘数寄存器**（QM），如图 7.3 所示. 四个触发器用于存储链（L）、溢出（F）和符号（S_a 和 S_x）. 一个 $2n$ 位的被除数开始时存储在级联的寄存器 AC · QM 中. n 位除数在整个执行期间始终存储在 AX 寄存器中. 被除数和除数的符号分别存储在 S_a 和 S_x 中. L 和 F 的初始内容为零.

对第 j 个周期的部分被除数 $2R^{(j)}$ 可以通过级联寄存器 $L \cdot$ AC · QM 左移一位求出，新产生的商位进入 QM 寄存器的右端. 从 AC 的左端推出的位存储在 L 触发器中. L 触发器用作缓冲寄存器，它能实现下列循环旋转操作

$$L \leftarrow AC_1;$$

W	X	AC 的功能
0	0	并行输入
0	1	具有 SRI=L 的右移
1	0	具有 SLI=QM_1 的左移
1	1	清除

Y	QM 的功能
0	并行输入
1	具有 SLI=q_{j+1} 的左移

Z	AX 的功能
0	并行输入
1	求反

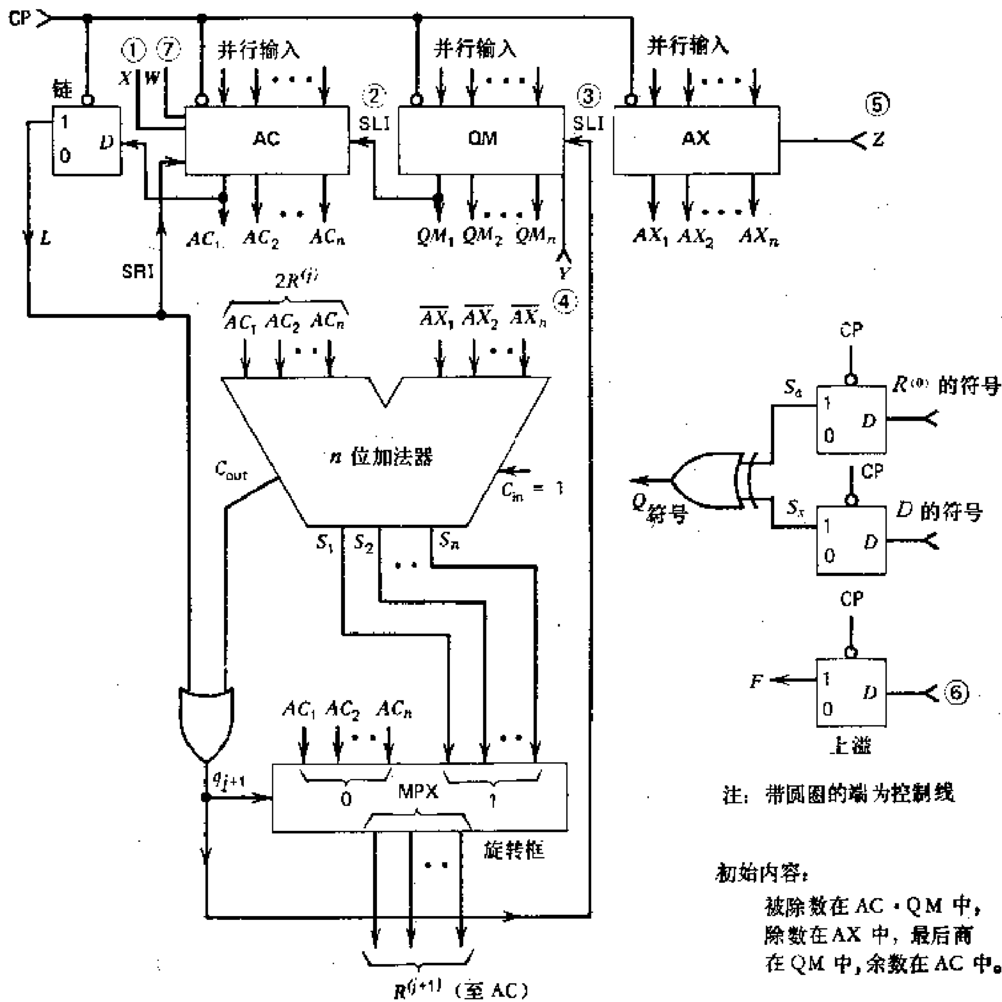


图 7.3 基于带恢复除法的二进制除法器的原理图

$$AC_1AC_2\cdots AC_n \leftarrow AC_2AC_3\cdots AC_nQM_1;$$

$$QM_1QM_2\cdots QM_n \leftarrow QM_2QM_3\cdots QM_nq_{i+1} \tag{7.23}$$

这里 q_{i+1} 就是所得到的新的商.

方程式(7.19)中规定的比较操作是用 2 的补码减法来实现的，如图 7.3 所示. **AX** 中的除数从移位后的部分余数中减去，后者存放在扩充的 $(n+1)$ 位寄存器 $L\cdot$**AC** 中. n 位加法器的输入-输出关系可用下列运算方程式来描述

$$C_{out}\cdot S_{1-n}=AC_{1-n}-AX_{1-n}=AC_{1-n}+\overline{AX}_{1-n}+1 \tag{7.24}$$

这里 $S_{1-n}=S_1S_2\cdots S_n$ 为和输出，C_{out} 为 n 位加法器的进位输出. 该加法器和它所延伸的链接位基本上实现了方程式 7.20. 第 $j+1$ 位商决定于下面的逻辑方程式

$$q_{j+1}=L\vee C_{out} \tag{7.25}$$

该硬件电路的结构验证如下. 当 $R^{(j)}$ 的最高位为"1"时，则它在移位成 $2R^{(j)}$ 后将留在链接位 L 中，于是，不管减的结果如何，有 $(L\cdot \mathbf{AC})>(\mathbf{AX})$. 另一方面，如果 $L=0$，那么 $C_{out}=1$，即意味着 $(\mathbf{AC})\geqslant(\mathbf{AX})$. 这两种情况均导致选择 $q_{j+1}=1$. 因而方程式(7.25)得到验证.

当 $q_{j+1}=0$ 时，利用多路转换器逻辑(旋转框)来保留旧的部分余数，则可省去恢复余数用的加法. 硬件的除法部件把初始部分余数逐步从 $2n$ 位缩减到 n 位，与此同时，商将增加到 n 位. 在中间阶段，有效部分余数出现在级联寄存器 **AC · QM** 最左边的 $2n-j$ 位，而商出现在右边的 j 位. 双字的长度和通常是 $2n$ 位. n 位商从右端逐位进入级联寄存器，而把前半部初始余数(被除数)从左端逐位推出. 由于初始周期用作溢出检测，最后余数 $R^{(n)}=R\times 2^n$ 将停留在 $L\cdot AC_1\cdot AC_2\cdots AC_{n-1}$ 中，其中 $R=R^{(n)}\times 2^{-n}$ 是所要求的 $2n$ 位余数. 计算结束时，用一次右移，使余数的后半部调整在累加器中. 以上硬件操作可用表 7.1 给出的数值例子来验证. 为了完成这个设计，必须决定所有控制端的方程式. 这将留给读者作为一个练习. 带恢复的二进制除法要求用 $n+1$ 个周期来实现，一个周期用于输入数据，n 个周期用于真正的比较-移位操作.

表 7.1　二进制带恢复除法的一个数值例子(这里采用图 7.3 所示的硬件部件)

周期 i	部分余数		除数 D	链接位 L	加法器输入		加法器输出		商	备　注 $(r=2,\ n=4)$
	$R^{(i)}$				$r\times R^{(i-1)}$	$\bar{D}$	进位	和		
	AC	**QM**	**AX**		**AC***	$\overline{\mathbf{AX}}$	C_{out}	$S_1S_2S_3S_4$	q_i	
0	0110	1101	1011	0	xxxx	xxxx	x	x x x x	符号 q_0	输入数据，检测溢出. 按方程式 7.7 来确定符号
1	0010	1011	1011	0	1101	0100	1	0 0 1 0	1	$R^{(1)}=r\times R^{(0)}-D$ (减法-移位)
2	0101	0110	1011	0	0101	0100	0	1 0 1 0	0	$R^{(2)}=r\times R^{(1)}$ (左移)
3	1010	1100	1011	0	1010	0100	0	1 1 1 1	0	$R^{(3)}=r\times R^{(2)}$ (左移)
4	1010	1001	1011	1	0101	0100	0	1 0 1 0	1	$R^{(4)}=r\times R^{(3)}-D$ (减法-移位)
5	1010	.1001	1011	1	0101	xxxx	x	x x x x	x	$L\cdot$ **AC** 右移，使余数调整在 **AC** 中
结果	余数	商	$0.q_1q_2q_3q_4=QM$ 中的 Q							

注：$\frac{A}{B}=Q+\frac{R}{B}$；$\frac{0.01101101}{0.1011}=0.1001+\frac{0.00001010}{0.1011}$，这里 $n=4$. AC^* 指左移后的 **AC**."x"指该条件为任意.

7.5 二进制不恢复除法

带恢复的除法与用“纸-笔”的除法是一致的．它的实现方法很简单，也很容易理解．但是，这些供恢复用的加法步骤（只是在采用带存储的多路转换时才能旁路）会使处理速度变慢，当商中出现许多零时尤其是这样．这一节我们要研究一种改进的方法，它将在不用存储多路转换逻辑的情况下完全取消恢复步骤．

以前，在每次叠代开始时均强制选商位为 1，不恢复除法与此不同，它是按状态指示来选择 +1 或是 −1 作为商位的．注意“0”不能作为合法的选择．这种商的选择是使每次选择所带来的误差在以后的步骤中能得到抵销，而且这种修正并不花费额外的加法、减法或移位延迟时间．它的思想是基于把方程式 7.15 中商的选择判据放宽到方程式 7.16 所规定的条件．在假定除数 $D > 0$ 时，我们可以把方程式 7.16 改写为

$$|R^{(j+1)}| < D \tag{7.26}$$

绝对值意味着相继的各个部分余数 $R^{(j+1)}$（其中 $j = 0, 1, \cdots, n-1$）都可正可负．只要余数的绝对值小于除数，就没有必要把负的余数恢复成正的余数．每次叠代时，除数或者加到部分被除数上，或者从部分被除数中减去．每一步所实现的指定操作取决于

$$R^{(j+1)} = \begin{cases} 2R^{(j)} - D, & \text{如果 } 2R^{(j)} > 0 \\ 2R^{(j)} + D, & \text{如果 } 2R^{(j)} < 0 \end{cases} \tag{7.27}$$

相应的商位按下列条件产生

$$q_{j+1} = \begin{cases} 1, & \text{如果 } 0 < 2R^{(j)} < 2D \\ -1, & \text{如果 } -2D < 2R^{(j)} < 0 \end{cases} \tag{7.28}$$

当 $2R^{(j)} = 0$ 时，处理过程即可结束．最后的商用不包含零的带符号数位编码来表示．因此，这个商既不是典型的，也不是最小的．反之，非零数位的数目却最大．为了与算术运算处理器中的其他操作相适应，必需把带符号数位的商转换成普通的二进制编码．

二进制不恢复除法的 Robertson 图在图 7.4 中给出．注意商的数位选 +1 或 −1 值，这对原点来说是对称的．只要 $D > R^{(0)}$（不溢出），那么就有 $|R^{(j+1)}| < D$，$|2R^{(j)}| < D$ 以及 $q_{j+1} \in \{-1, 1\}$．二条 45° 斜率的直线符合于方程式 7.27 中的二个式子．它们分别与按方程式 7.28 选择 $q_{j+1} = 1$ 和 $q_{j+1} = -1$ 相适应．不恢复方法需要有一个精确的符

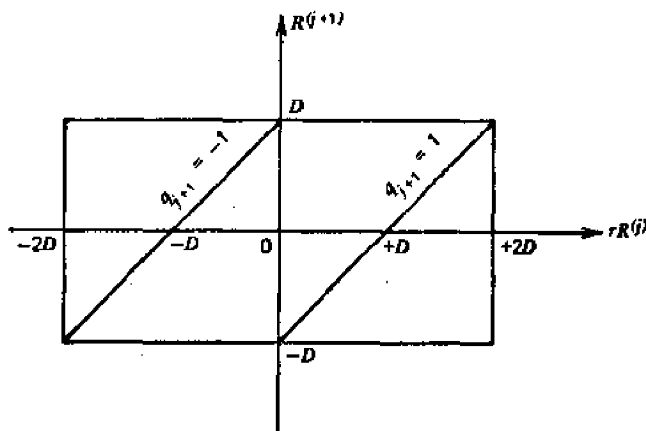


图 7.4 具有数位集合 {−1, 1} 的不恢复二进制除法的 Robertson 图

号确定机构，后者能检测相继的部分余数的符号，一直到整个除法过程完成为止.

为了比较二个二进制数所需要的时间是与字长成正比的. 在带恢复的除法中不存在冗余，这就要求除数中的所有数位对部分余数进行比较，以便确定商位的值. 在不恢复除法中产生的商当然是冗余的. 施加了冗余就允许通过估计除数和部分余数的数值来选择商位. 在引入较多的冗余时，这些估计可以比较粗略（长度上较短）. 为了选择商的数位所需要的比较时间在估计的字长较短时将减小.

一个负的商位 $q_i = -k$，（对一些正整数 k）意味着已经出现了 k 次加法. 对一个正的商位 $q_i = k$，则表示执行过同样次数的减法. 因为加法器在每一步中都要用到（加法或减法，但没有单纯的移位），所以在不恢复除法中平均移位长度是 1，而且在不恢复除法中多次移位的措施是无效的. 这就是与带恢复除法相区别的地方，因为在带恢复除法中，当商中产生一串零时，允许移多位来旁路加法器.

7.6 高基数不恢复除法

计算机的运算常常认为采用二进位的"组"要比用互相独立的每个二进位更方便一些. 这种成组的方法可以用高基数（大于 2）的数位来解释. 例如，一对二进位可以当作一个基数为 4 的数位，三个二进位可以当作一个基数为 8 的数位，等等. 一般地说，一串 l 个二进位等效于 m 个基数为 r 的数位，这里

$$m = \frac{l}{\lceil \log_2 r \rceil} \tag{7.29}$$

对实际情况 $r = 2^k$（其中整数 $k > 0$），因而 $l = m \times k$.

二进制的带恢复和不恢复的方法二者均能加以一般化，来设计出具有高基数的系统. 执行除法（DIVIDE）指令所需要的叠代次数是随着基数的增加而迅速减小的. 这意味着当采用高基数除法方案时，可以预期获得较快的执行时间. 这里需要从以下二个方面折衷考虑，即：与高基数除数的倍数的生成有关的硬件复杂性的增加，以及在选择高基数商数位时的步骤的复杂化.

具有任意基数 r 的带恢复除法已经在第 7.3 节中作了讨论. 下面我们将说明基数 r 的不恢复除法的原理. 递归过程仍用方程式 7.5 来描述. 每个商位均从方程式 7.2 给出的数字集合中选出. 例如，当基数 $r = 4$ 时，商的数字集合 $= \{-3, -2, -1, +1, +2, +3\}$. 当 $R^{(0)} < D$ 时，不存在溢出.

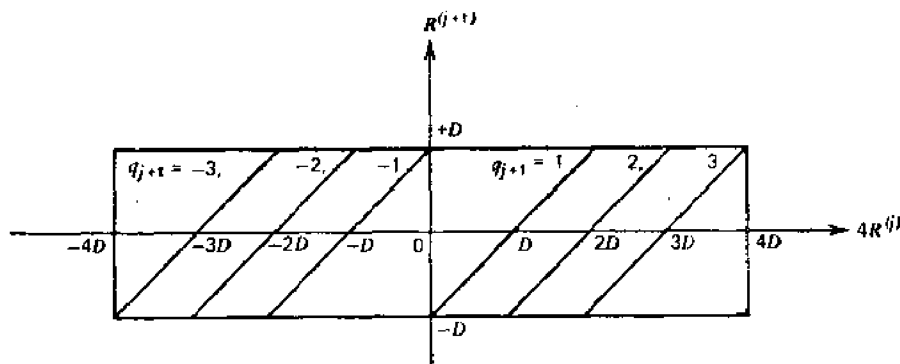


图 7.5 具有数字集合 $\{-3, -2, -1, +1, +2, +3\}$、基数为 4 的不恢复除法的 Robertson 图

第 $(j+1)$ 位商 q_{j+1} 是这样选择的，即让部分余数的绝对值处在下列区间内

$$-D < R^{(j+1)} < D \tag{7.30}$$

图 7.5 中的 Robertson 图描绘了一个具有基数 $r=4$ 的不恢复 的除法方案. 六条直线相应于每个商位的六种可能的数值. 应该指出，在某些区域，商 q_{j+1} 值会有多种选择. 这可以通过这些线在 $r \times R^{(j)}$ 轴上重叠的投影来看出. 这些倍数的选择列表如下

$$q_{j+1} = \begin{cases} -3, & \text{如果 } -4D < 4R^{(j)} < -3D \\ -3 \text{ 或 } -2, & \text{如果 } -3D < 4R^{(j)} < -2D \\ -2 \text{ 或 } -1, & \text{如果 } -2D < 4R^{(j)} < -D \\ -1, & \text{如果 } -D < 4R^{(j)} < 0 \\ +1, & \text{如果 } 0 < 4R^{(j)} < D \\ +1 \text{ 或 } +2, & \text{如果 } D < 4R^{(j)} < 2D \\ +2 \text{ 或 } +3, & \text{如果 } 2D < 4R^{(j)} < 3D \\ +3, & \text{如果 } 3D < 4R^{(j)} < 4D \end{cases} \tag{7.31}$$

图 7.6 的流程图所描述的是变换的一般规则，即把一个基数为 r 的冗余的 **SD** 商

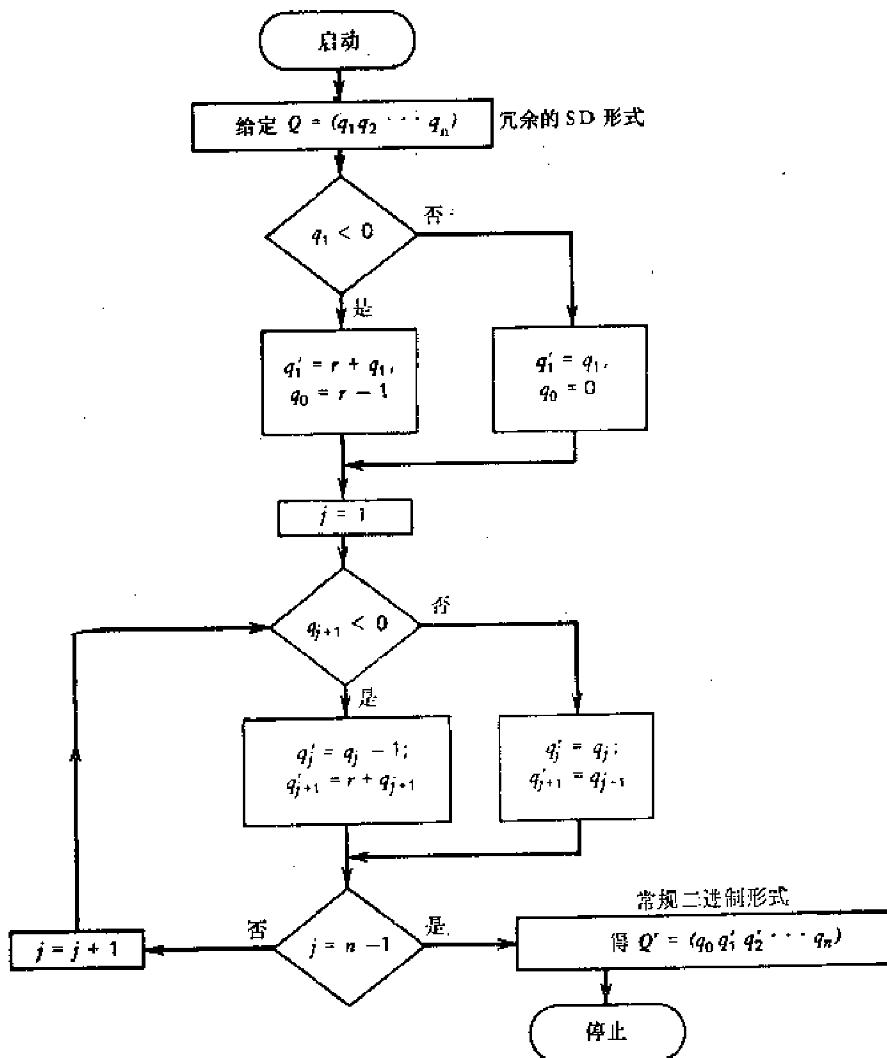


图 7.6 基数为 r 的不恢复除法中商的换算

$(q_1q_2\cdots q_n)$，其中每一个数位

$$q_i \in \{-(r-1), \cdots, -1, +1, \cdots, +(r-1)\}$$

（对所有的 i），变换成普通的以基数-补码（对 r 的补码）表示的 r 进制数 $q_0.q_1'q_2'\cdots q_n'$，其中每个 $q_i' \in \{0, 1, \cdots, r-1\}$（对所有的 i）。此外对正数来说，符号位 $q_0 = 0$，对负数，

$$q_0 = r - 1$$

换算的过程要求对商的各位，首先是最高有效位进行一系列检测。数位 q_{j+1} 在它为正数时是不变的。当遇到一个负的数位时，就要加上基数，同时向相邻的高位借位。注意，由于零不是一个允许的数位，所以不需要借位传播。下面给出一个 $r = 4$ 和 $n = 3$ 的数值例子。

例 给定 $Q = (q_1q_2q_3) = (\bar{2}\,1\,\bar{3})$。$Q$ 的固有值为

$$Q = \left(-2 \times \frac{1}{4}\right) + \left(1 \times \frac{1}{4^2}\right) + \left(-3 \times \frac{1}{4^3}\right) = \frac{-2}{4} + \frac{1}{16} + \frac{-3}{64}$$

$$= \frac{-32 + 4 - 3}{64} = \frac{-31}{64}$$

这个冗余的商 Q 可以变换成 4 的补码数 $Q' = (q_0.q_1'q_2'q_3')$。详细步骤说明如下：

第一步。这里 $q_1 = \bar{2} < 0$，导至负的符号，所以

$$q_0 = r - 1 = 4 - 1 = 3$$

$$q_1' = r + q_1 = 4 + \bar{2} = 2 \tag{7.32}$$

对 $j = 1$。因 $q_1' = 2$ 和 $q_2 = 1$，即有

$$q_2' = q_2 = 1 \tag{7.33a}$$

对 $j = 2$。因 $q_2' = 1$ 和 $q_3 = \bar{3}$，即意味着 q_2' 应修改为

$$q_2' = q_2 - 1 = 1 - 1 = 0 \tag{7.33b}$$

和

$$q_3' = r + q_3 = 4 + \bar{3} = 1 \tag{7.34}$$

方程式 7.32 至 7.34 给出了最后得到的用基数-补码形式表示的四进制编码，

$$Q' = (q_0.q_1'q_2'q_3') = (3.210)_4$$

Q' 有一个固有值 $-31/64$，这可以通过把 Q' 重新求补，返回到它的正数表示式来得到验证，即

$$\bar{Q} = (0.133)_4 = 1 \times \frac{1}{4} + 3 \times \frac{1}{16} + 3 \times \frac{1}{64} = \frac{31}{64}$$

7.7 SRT 除法的原理

二进制除法的 **SRT** 方法是 Sweeney, Robertson 和 Tocher 三人几乎在同时发现的[11,14,15]。它的提出是为了改进二进制浮点运算。这个方法涉及一个规格化的除数，以及若干相继的部分被除数，后者在下列区间内也被规格化：

$$\frac{1}{2} < |D| < 1 \tag{7.35}$$

$$\frac{1}{2} < |2R^{(j)}| < 1 \tag{7.36}$$

在不恢复除法中，"0"属于被禁止使用的商位. 这就意味着不用单纯移位的操作. 但这并不是所希望的省略，因为移位所需要的时间通常比加法/减法时间少得多. **SRT** 除法则允许商的数字集合为 $\{-1, 0, 1\}$，这里把"0"也作为一个合法的选择，从而使不恢复除法得到了改进.

根据部分被除数对除数进行比较的结果，除数或者加到部分被除数(指移位后的部分余数)上，或者被移位，或者从部分被除数中减去. 每个周期要执行的指定操作可以用下式来描述

$$R^{(j+1)}=\begin{cases}2R^{(j)}+D, & \text{如果 } 2R^{(j)}<-D\\ 2R^{(j)}, & \text{如果 } -D\leqslant 2R^{(j)}\leqslant D\\ 2R^{(j)}-D, & \text{如果 } D<2R^{(j)}\end{cases}\tag{7.37}$$

商位 q_{j+1} 的选择规则是

$$q_{j+1}=\begin{cases}-1, & \text{如果 } R^{(j)}<-D\\ 0, & \text{如果 } -D\leqslant 2R^{(j)}\leqslant D\\ 1, & \text{如果 } 2D<2R^{(j)}\end{cases}\tag{7.38}$$

上述规则以图解的形式说明在图 7.7 中. 由于除数和部分被除数二者都是按方程式 7.35 和 7.36 规格化的分数，所以我们可以排除按方程式 7.37 和 7.38 对 $2R^{(j)}$ 与 D 和 $-D$ 之间所作的全部比较. 这里只要用除数的最小值作为阈值便能实现. 比较操作可以缩减为下面的简化形式

$$R^{(j+1)}=\begin{cases}2R^{(j)}+\frac{1}{2}, & \text{如果 } 2R^{(j)}\leqslant-\frac{1}{2}\\ 2R^{(j)}, & \text{如果 } -\frac{1}{2}<2R^{(j)}<\frac{1}{2}\\ 2R^{(j)}-\frac{1}{2}, & \text{如果 } \frac{1}{2}\leqslant 2R^{(j)}\end{cases}\tag{7.39}$$

商的选择规则也能相应地简化为

$$q_{j+1}=\begin{cases}-1, & \text{如果 } 2R^{(j)}\leqslant-\frac{1}{2}\\ 0, & \text{如果 } -\frac{1}{2}<2R^{(j)}<\frac{1}{2}\\ 1, & \text{如果 } \frac{1}{2}\leqslant 2R^{(j)}\end{cases}\tag{7.40}$$

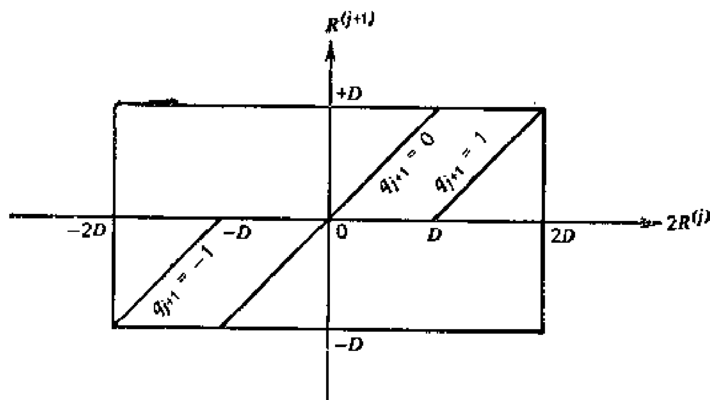


图 7.7 允许数字集合为 $\{-1, 0, 1\}$ 的二进制不恢复除法的 Robertson 图

采用方程式 7.40 中的一组规则的优点在于只需要将 $2R^{(j)}$ 对常数 $\frac{1}{2}$ 或 $-\frac{1}{2}$ 进行比较. 这在二进制系统中是很容易实现的. 刚才描述的过程具有使部分余数规格化的效果：若部分余数是正的，前置位为零就移位，若部分余数是负的，前置位为 1 就移位.

这样生成的商位具有冗余的带符号数字的形式. 如果除数是已知的，或者开始标准化操作时已经被变换成正的，那么这个过程能进一步得到简化. 在这种情况下，应该采用下面一组规则

$$q_{j+1}=\begin{cases}-1, \text{ 如果 } |2R^{(j)}| \geqslant \frac{1}{2} \text{ 和 } (2R^{(j)}) \text{ 的符号为负} \\ 0, \text{ 如果 } |2R^{(j)}| < \frac{1}{2} \\ +1, \text{ 如果 } |2R^{(j)}| \geqslant \frac{1}{2} \text{ 和 } (2R^{(j)}) \text{ 的符号为正}\end{cases} \tag{7.41}$$

上述选择被归纳在图 7.8 所示的 Robertson 图中. $2R^{(j)}$ 的范围限制在 $-1<2R^{(j)}<1$，$R^{(j+1)}$ 的范围限制在 $-\frac{1}{2}< R^{(j+1)} <\frac{1}{2}$. 这些 q 线表示着三种情况，分别相应于数值 $D=1/2, 3/4$ 和 1，即二个极值加上一个中间值.

举例来说，如果被除数 $R^{(0)}>0$ 与除数 $D=1/2$，那么所有部分余数都是正的，商的每一位可以是 0，也可以是 +1，如第一象限中的线 $R^{(j+1)}=2R^{(j)}-\frac{1}{2}$ 所指定的那样. 在这种特殊情况下，商具有普通的二进制形式，根据表 5.7，它的平均移位长度为 2. 另一方面，对 $D=3/4$ 来说，商将具有典型的最小带符号数字的形式，平均移位长度为 3. 在 $D=1$ 的情况下，商的表示式的特点是：连续的非零数位具有相反的符号. 从图 7.8 的第二象限和第四象限中的二条 q-线可以看出这一点. 对 $D=1$，将会产生具有平均移位长度 2 的冗余码.

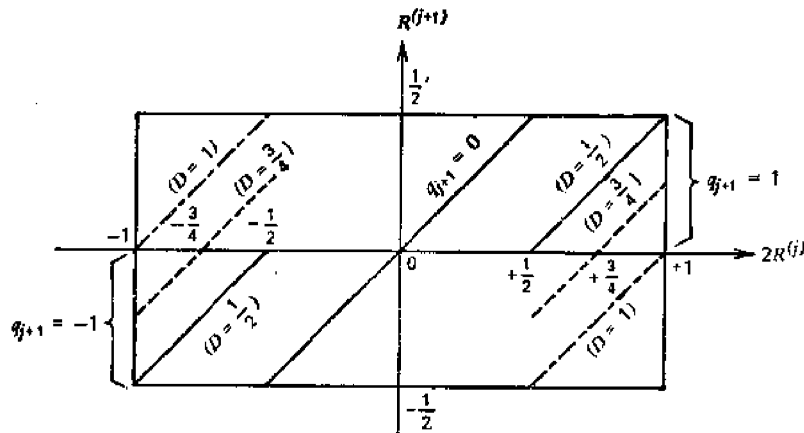


图 7.8 描述二进制 SRT 除法的 Robertson 图，图中的直线分别对应于除数的三个可能值 $\frac{1}{2}$，$\frac{3}{4}$ 和 1

上述分析可以导出以下结论：一个 SRT 除法系统的效率同时依赖于规格化除数的数值. Freiman[5] 求出了规格化除数所有可能值的平均移位长度. 就是说，SRT 除法的性能可以通过控制它的除数的范围而获得进一步提高. SRT 除法的具体实现将在第 7.10 节

中说明.

7.8 修改的二进制 SRT 除法

自从 1950 年末发表 **SRT** 除法之后，为了使上一节讨论的原始的 **SRT** 除法最佳化，已经有许多研究者探索出一些改进方法.**SRT** 除法的效率的分析是相当复杂的. Freiman[5] 曾经指出：**SRT** 除法所求出的商是最小的形式，但不一定是典型的形式，对下列范围内的除数来说，具有平均移位长度 3

$$\frac{3}{5} \leqslant |D| \leqslant \frac{3}{4} \tag{7.42}$$

实际上，Robertson[12] 已经指出过对具有下列精确值的除数

$$|D| = \frac{3}{4} \tag{7.43}$$

商将为典型的最小形式. Shively[13] 则给出这个范围的一个严格而完整的证明.

Wilson 和 Ledley[18] 指出了 **SRT** 除法的第一个变体，它对 $\frac{1}{2} < |D| < 1$ 的范围具有均匀的平均为 3 的移位长度. 他们的方法提供了一个除正的规格化分数的简化途径. 这个途径是基于快速乘法的逆过程，在快速乘法中位的模式被分解成一些用"0"和"1"隔离的"1"和"0"的串，就象图 7.9 中的例子所指出的那样.

在假定除数 D 是正的规格化分数，被除数 N 也是规格化的或者是在二进制小数点后面只有(至多)一个零时，Wilson-Ledley 除法的方法可以用图 7.10 中的流程图来描述. 这个过程是以形成第一个部分余数开始的

$$N^{(i)} = N^{(i-1)} - D \tag{7.44}$$

因为在上述假定下 $N^{(i)}$ 是负的，所以进入图 7.10 中负的循环. 如果 $N^{(i)}$ 在二进制小数点右边有 α 个零，那么商 Q 在二进制小数点右边至少有 $\alpha - 1$ 个 1 (在开始的一步中，$q_i = 0$，其中 $i = 0$，这只是说 $Q < 1$). 现在 $N^{(i)}$ 被规格化，第二个部分余数按下式形成

$$N^{(i+1)} = N^{(i)} + D \tag{7.45}$$

图 7.9 将二进制数分解成四类子串的说明

如果 $N^{(i+1)}$ 是负的，那么 $q_{0+\alpha} = 0$，后继的位都是 1. 如果 $N^{(i+1)}$ 是正的，那么 $q_{0+\alpha} = 1$，后继的位都是 0. 该过程按照这种方式继续进行下去，每次求差和规格化，每一步通过 $q_{i+\alpha-1}$ 来决定 q_i 和 q_{i+1}. 注意每一步可以决定几个商的数位. 当递归的下标到达所要求的商的位数时，这个过程便告结束.

图 7.11 中给出了这种除法的一个数值例子，那里被除数是

$$N = 0.1001111100001100,$$

除数 $D = 0.1101$. 这个例子很好地证实了上面的过程. 这个方法使普通二进制除法中的

加法或减法次数平均节省了 2/3. 它还能得出一个对所有规格化的除数来说是最小的,但不一定是典型的形式的商. 这个方法还表明了一种设计，其中可以不需要每一步都进行全字长的比较.

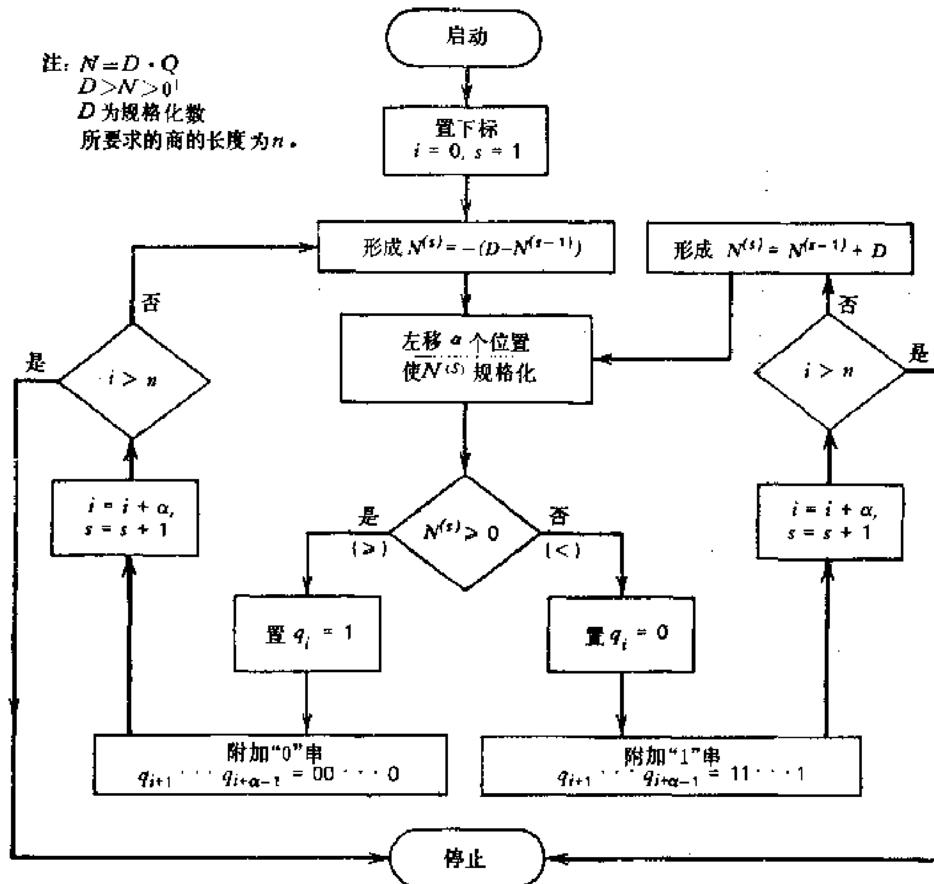


图 7.10 用于快速二进制除法的 Wilson和 Ledley 算法

另一类修改的 **SRT** 除法是 Metze[10] 提出的，它是基于 Freiman 对 **SRT** 算法的统计分析. Metze 方法所依据的基本思想在于每次叠代所生成的商位个数将由于部分余数的规格化而增加. 部分余数不必作高精度检查,可是商要表示成最小的形式,但不一定是典型形式.

方程式 7.39 中规定的 **SRT** 除法涉及移位后的部分余数 $2R^{(j)}$ 与常数 $K=1/2$ 的比较. 显然,2^{-1} 的精度对这个比较已经足够了. 只要除数落在方程式 7.42 的范围内，根据比较常数 $K=1/2$,有可能通过求 K/D 的极值来确定局部的商的范围,即

$$\frac{2}{3}\leqslant|Q|\leqslant\frac{5}{6} \tag{7.46}$$

这里

$$\left(\frac{K}{D}\right)_{\min}=\frac{1/2}{D_{\max}}=\frac{1/2}{3/4}=\frac{2}{3} \tag{7.47}$$

和

$$\left(\frac{K}{D}\right)_{\max}=\frac{1/2}{D_{\min}}=\frac{1/2}{3/5}=\frac{5}{6} \tag{7.48}$$

给定 $N = 0.1001111100001100$ 和 $D = 0.1101$

求 $Q = N/D = q_0 q_1 \cdots q_n$ 对 $n = 10$.

			所产生的商的位串
$+D$		0.1101	
$N^{(0)} = N$:	−)	0.1001 1111 0000 1100	$q_0 . q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10}$
$N^{(1)} = -(D - N^{(0)})$:		−0.0011 0000 1111 0100	$(\alpha = 2)$
规格化:		−0.11 0000 1111 0100	0.1?
$+D$:	+)	0.11 01	
$N^{(2)} = N^{(1)} + D$:		+0.00 0011 0000 1100	$(\alpha = 4)$
规格化:		+0.11 0000 1100	0.11000?
$-D$:	−)	0.11 01	
$N^{(3)} = -(D - N^{(2)})$:		−0.00 0011 0100	$(\alpha = 4)$
规格化:		−0.11 0100	0.110000111?
$+D$:	+)	0.11 01	
$N^{(4)} = N^{(3)} + D$:		0.00 0000	$(\alpha > 6)$
规格化:		$(\alpha = 6)$	0.1100001111
			$q_0 . q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10}$

图 7.11　阐明图 7.10 中所描述的 Wilson-Ledley 快速除法的一个数值例子

这就是说，当除数落在方程式 7.42 的范围内时，Metze 方法与 **SRT** 除法是相同的. 对于这个范围之外的除数，通过选择一个不同的比较常数来改进商的选择过程，Metze 方法可以取代 **SRT** 除法. 对于不同的除数区域，比较常数应该这样来选择，使得局部的商的范围符合方程式 7.46 中的规定.

为了实现比较常数的动态选择，对于第 i 个除数区域来说，可以简单地选择常数 K_i 使它满足下列边界

$$\frac{6}{5} K_i \leqslant |D_i| \leqslant \frac{3}{2} K_i \tag{7.49}$$

这里 D_i 为第 i 个区域中的除数，同时利用了下列极限

$$D_{\min} = \frac{K_i}{Q_{\max}} = \frac{K_i}{5/6} = \frac{6}{5} K_i \tag{7.50}$$

$$D_{\max} = \frac{K_i}{Q_{\min}} = \frac{K_i}{2/3} = \frac{3}{2} K_i \tag{7.51}$$

一般地说，大量的区域都允许进行低精度的递归比较，反之亦然. 因此，这种修改的 **SRT** 除法包括二个阶段. 一开始，应用方程式 7.49 至方程式 7.51 来确定适当的比较常数 K_i. 每一个递归步骤，除了使 $K = K_i$ 以外，都遵守 **SRT** 除法的规则.

表 7.2 列出了五个除数区域以及相应的比较常数，它们的选择应同时满足方程式 7.46 和方程式 7.49. 与常数 K_i 的递归比较要求的精度最高可到 2^{-4}. 表 7.2 中的最后一列指出了没有重叠的实用的除数区域. 这些区域的选择保证了鉴别精度可达 2^{-4}. 图 7.12 和图 7.13 指出了三个数值例子，它们对原始的 **SRT** 除法和 Metze 的 **SRT** 除法用同一组数据作了比较. 例 A 涉及的一个除数在 $\frac{3}{5} \leqslant |D| \leqslant \frac{3}{4}$ 的范围内，对它来说 **SRT** 除法可以获得用最小形式表示的商. 例 B 则指出了除数在上述范围之外的一个 **SRT** 除法. 这样产生的商得不到最小的表示式. 在图 7.13 中的例 C 指出了修改的 **SRT** 除法，它用了例 B 中同样的数据. 一个最小形式表示的商是可以用 Metze 方法来求出的.

表 7.2　关于 Metze 修改的 SRT 除法的一组比较常数和相应的除数区域

区　域	比较常数	理论的除数区域	没有重叠的实用除数区域
1	$K_1=\frac{3}{8}$	$\frac{9}{20}\leqslant\|D_1\|\leqslant\frac{9}{16}$	$\|D_1\|<\frac{9}{16}$
2	$K_2=\frac{7}{16}$	$\frac{21}{40}\leqslant\|D_2\|\leqslant\frac{21}{32}$	$\frac{9}{16}\leqslant\|D_2\|<\frac{5}{8}$
3	$K_3=\frac{1}{2}$	$\frac{3}{5}\leqslant\|D_3\|\leqslant\frac{3}{4}$	$\frac{5}{8}\leqslant\|D_3\|<\frac{3}{4}$
4	$K_4=\frac{5}{8}$	$\frac{3}{4}\leqslant\|D_4\|\leqslant\frac{15}{16}$	$\frac{3}{4}\leqslant\|D_4\|<\frac{15}{16}$
5	$K_5=\frac{3}{4}$	$\frac{9}{10}\leqslant\|D_5\|\leqslant\frac{8}{9}$	$\frac{15}{16}\leqslant\|D_5\|$

注：初步确定的除数区域和递归比较的最大精度等于 2^{-4}.

例 A．除数在 $\frac{3}{5}<|D|<\frac{3}{4}$ 的区域内

给定 $N=\frac{77}{256}$，$D=\frac{11}{16}$ 以及 $K=\frac{1}{2}$

$R^{(0)}=N=$　0.01001101　　商

$2R^{(0)}=$　0.1001101　　$2R^{(0)}>\frac{1}{2}\rightarrow q_1=1$

$-D=$　−) 0.1011　　$Q=0.q_1q_2q_3q_4$

$R^{(1)}=2R^{(0)}-D=$　−0.0001001　　$=0.100\bar{1}$

$R^{(2)}=2R^{(1)}=$　−0.001011　　$|2R^{(1)}|<\frac{1}{2}\rightarrow q_2=0$　　$=\frac{7}{16}$

$R^{(3)}=2R^{(2)}=$　−0.01011　　$|2R^{(2)}|<\frac{1}{2}\rightarrow q_3=0$　　（最小形式）

$R^{(4)}=2R^{(3)}=$　−0.1011　　$|2R^{(3)}|<\frac{-1}{2}\rightarrow q_4=\bar{1}$

$+D=$　+0.1011

　0.0000

例 B．除数在 $\frac{3}{5}<|D|<\frac{3}{4}$ 的区域之外

给定 $N=\frac{63}{256}$ 和 $D=\frac{9}{16}$

$R^{(0)}=N=$　0.00111111　　商

$R^{(1)}=2R^{(0)}=$　0.0111111　　$|2R^{(0)}|\leq\frac{1}{2}\rightarrow q_1=0$

$2R^{(1)}=$　0.111111　　$2R^{(1)}>\frac{1}{2}\rightarrow q_2=1$

$-D=$　−) 0.1001　　$Q=0.q_1q_2q_3q_4$

$R^{(2)}=2R^{(1)}-D=$　0.011011　　（非最小形式）

$2R^{(2)}=$　0.11011　　$2R^{(2)}\frac{1}{2}q_3=1=0.0111=\frac{7}{16}$

$-D=$　−) 0.1001

$R^{(3)}=2R^{(2)}-D=$　0.01001

$2R^{(3)}=$　0.1001　　$2R^{(3)}>\frac{1}{2}\rightarrow q_4=1$

$-D=$　−) 0.1001

　0.0000

图 7.12　描述 SRT 除法的数值例子，应用于二个不同的除数区域

例C. 修改的(Metze) SRT除法用于和例 B 相同的操作数。

给定 $N=\frac{63}{256}$ 和$D=\frac{9}{16}$，比较常数 $K=\frac{7}{16}$ (根据表 7.2)

$$
\begin{array}{lcrl}
R^{(0)}=N & = & 0.00111111 & \quad\text{商} \\
2R^{(0)} & = & 0.0111111 & \quad 2R^{(0)} < \frac{7}{16} \rightarrow q_1 = 1 \\
-D & = & -)\ \ 0.1001 & \\
\hline
R^{(1)}=2R^{(0)}-D & = & -0.0001001 & \\
R^{(2)}=2R^{(1)} & = & -0.001001 & \quad |2R^{(1)}| < \frac{7}{16} \rightarrow q_2 = 0 \\
R^{(3)}=2R^{(2)} & = & -0.01001 & \quad |2R^{(2)}| < \frac{7}{16} \rightarrow q_3 = 0 \\
R^{(4)}=2R & = & -0.1001 & \quad 2R^{(3)} < \frac{-7}{16} \rightarrow q_4 = \bar{1} \\
+D & = & +)\ \ 0.1001 & \\
\hline
 & & 0.0000 &
\end{array}
$$

答：$Q=0.q_1q_2q_3q_4=0.100\bar{1}$ 得到最小形式。

图 7.13 用 Metze 修改的 SRT 除法来求得最小形式的 SD 商

7.9 Robertson 的高基数除法

Robertson 最早以一般化的形式提出了 SRT 除法[11]，那里基数 r 是任意的，商的数字集合如方程式 7.4 所示。对不恢复除法来说，可用同样的递归公式（方程式 7.5）来描述 Robertson 的一般化方法。然而，逐次的商位的选择应满足方程式 7.17。对已知的除法来说，用在 $|R^{(j+1)}| \leqslant k \times |D|$ 中的常数 k 的几个离散值处在下列范围内

$$\frac{1}{2} \leqslant k \leqslant 1 \tag{7.52}$$

特别是，$k=1$ 相应于不恢复除法(方程式 7.16)。

Robertson 除法的机械实现，要求三个明确的步骤：

第一步，将部分余数 $R^{(j)}$ 左移一个数位的位置（在基数 r 中，即指移 $\lceil \log_2 r \rceil$ 个二进位），以便得到 $r \times R^{(j)}$，这和第 7.6 节中所述的相同。

第二步，选择几个许可的运算步骤之一，使得 $r|R^{(j)}|$ 的最大绝对值(即如方程式 7.17 中所示的 $rk|D|$) 减小到总数的 $1/r$，这样下一个部分余数 $R^{(j+1)}$ 即满足方程式 7.17。

第三步，相应于所选的运算步骤，产生商的一个数位。

这种高基数除法成功的关键在于对运算步骤的分析，它使上述第二步成为可能。我们把注意力集中在根据下列方程组把 $r \cdot R^{(j)}$ 变换成 $R^{(j+1)}$ 的运算步骤上：

$$R^{(j+1)} = r \times R^{(j)} - i \times |D| \tag{7.53}$$

这里 $i=-m, \cdots, -2, -1, 0, 1, 2, \cdots, m$ 和$(r-1)/2 < m < r-1$。

我们曾把方程式 7.53 中的每一个式子都称做一条 **q-线**。这组 **q-线**可以解释成一个

$R^{(j+1)}$随 $r \times R^{(j)}$ 变化的曲线图，其中以 q_{j+1} 作为参变量. 我们曾经把这个曲线图称为 Robertson 图. 这 $2m+1$ 条 **q-线**被限制在以原点为中心的一个矩形内，这个矩形的顶点的坐标为

$$(\pm r k \times |D|, \pm k \times |D|) \tag{7.54}$$

如图 7.14 所示. 在矩形内部，**q-线**在 $r \times R^{(j)}$ 轴上的投影覆盖了矩形中 $r \times R^{(j)}$ 轴的一部分. **q-线**具有斜率r，而四个顶点（方程式 7.54）落在通过原点的斜率为 $\pm(1/r)$ 的线上. 这个矩形必须足够大，以保证 $k \geq 1/2$

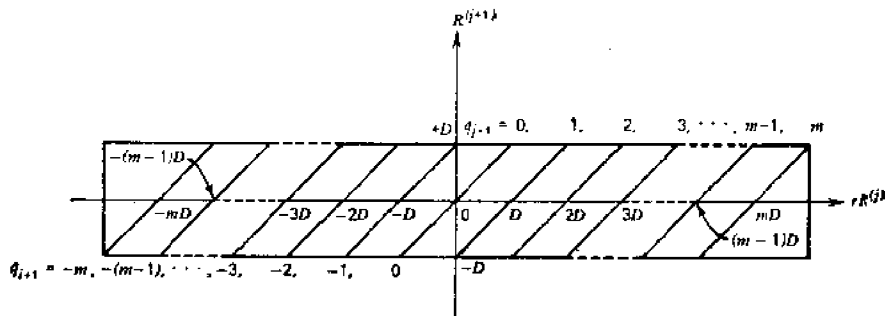


图 7.14 具有数位集合 $\{-m, \cdots, -2, -1, 0, 1, 2, \cdots, m\}$ 的 Robertson 高基数 **SRT** 除法的 Robertson 图

以下二条理由支配着对 k 和 m 值的选择：

1. **q-线**的数目应该最小化. 这是因为在除法期间必须形成的除数倍数的个数正比于 **q-线**的数目.

2. **q-线**在 $r \times R^{(j)}$ 轴上的投影的重叠应求最大. 在商的选择过程中，所需精度是随重叠的增加而降低的. 应该记住，降低了商的精度要求，选择也会较快.

这二个条件实际上是互相矛盾的. 因此常数 k 限制在方程式 7.52 的范围内，它的选择应该取自一组数值的离散的集合，即落在最右边的 **q-线**上的矩形右上角的顶点

$$(r k \times |D|, k \times |D|) \tag{7.55}$$

最右边的 **q-线**即为

$$R^{(j+1)} = r \times R^{(j)} - m \times |D| \tag{7.56}$$

k 的数值作为 r 和 m 的函数，它可以通过求解线 $R^{(j+1)} = r \times R^{(j)}$ 和方程式 7.56 所确定的 **q-线**的交点来求出. 交点上的 $R^{(j+1)}$ 值为 k，它等于 $m/(r-1)$. 因此我们得到

$$k = \frac{m}{r-1} \tag{7.57}$$

利用条件 $k \geq 1/2$，我们得到

$$m \geq \frac{r-1}{2} \tag{7.58}$$

m 具有明显的上界 $r-1$，我们有

$$\frac{r-1}{2} \leqslant m \leqslant r-1 \tag{7.59}$$

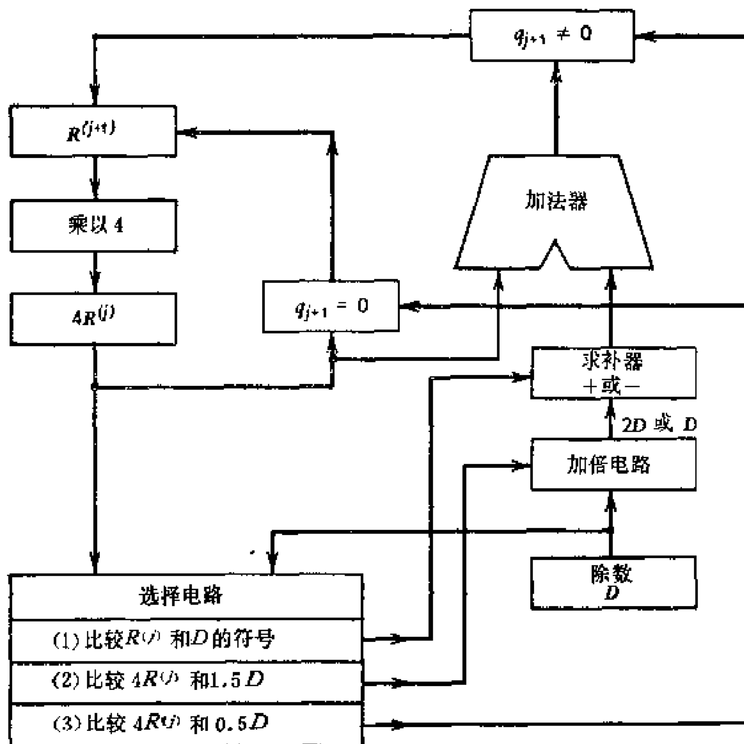
给定一个基数 r 后，m 和 k 的最佳选择严重地依赖于具体设计上的折衷，如象时间和设备的成本.

在二进制 **SRT** 除法中，$r=2$，我们用 $m=r-1=2-1=1$，因而上限 $k=m/(r-1)=1/(2-1)=1$. 在下面的例子中，基数 $r=4$，并具有数字集合 $\{-2,-1,0,1,2\}$，我们选 $m=2$. 所有必需的除数倍数 $\pm 2D$ 和 $\pm D$ 都可以通过移位和求补来形成，这比普通的具有数字集合 $\{0,1,2,3\}$ 的基数-4 的带恢复除法中倍数 D, $2D$ 和 $3D$ 的生成要简单得多.

基数-4 的 Robertson 除法方案表示在图 7.15 中. 这个部件包括一个简单的二进制加法器，一个条件求双倍与求补电路，以及一个选择逻辑. 选择电路将 $r \times R^{(j)} = 4R^{(j)}$ 与 $0.5D$ 和 $1.5D$ 相比较，精度到 7 位，这里假定

$$\frac{1}{4} \leqslant |D| < 1 \tag{7.60}$$

$$R^{(j+1)} = \begin{cases} 4R^{(j)}, \text{若 } q_{j+1} = 0 \\ \text{来自全加器输出，若 } q_{j+1} \neq 0 \end{cases}$$



注：1) 当 $R^{(j)}$ 和 D 的符号一致时，求补器被置成减法，否则被置成加法

2) 如果 $4|R^{(j)}| \geqslant \frac{5}{3}|D|$，则加倍电路输出 $2D$；如果 $4|R^{(j)}| \leqslant \frac{4}{3}|D|$，则输出 D；

如果 $\frac{4}{3}|D| < 4|R^{(j)}| < \frac{5}{3}|D|$，则二者均可

图 7.15 Robertson 提出的基数-4的 **SRT** 除法方案[11]

表 7.3 提供了相应的被选商位的数值以及在图 7.15 的三个选择电路中所执行的操作。

表 7.3　图 7.15 中商的选择和相应的选择逻辑

选择电路 (1)	选择电路(2)	选择电路(3)	q_{i+1} 的数值
减法	$2D$	$q_{i+1}\neq 0$	+2
减法	$1D$	$q_{i+1}\neq 0$	0
减法	$1D$	$q_{i+1}\neq 0$	+1
加法	$2D$	$q_{i+1}=0$	−2
加法	$1D$	$q_{i+1}=0$	0
加法	$1D$	$q_{i+1}\neq 0$	−1

7.10　实例研究 II：ILLIAC-III 的算术运算处理器

本节介绍采用带符号数位 (**SD**) 冗余码和高基数方法的算术运算处理器，它包含乘数再编码和 **SRT** 除法．这个基数为 256 的算术运算处理器的设计已经在 Illinois 大学的 ILLIAC-III 计算机中研制成功．这个实例研究中仔细推敲了所需功能块的硬件要求．这里提出了一个具体的实现办法，其中要用到前几章和本章中为高速乘法和除法所描述过的那些理论模型。

应该指出，这个设计与具体工艺无关．这里主要的努力集中在用功能子块组成一个有效的算术运算处理器．在一个高速算术运算部件中，为了加快乘法的执行，通常会涉及硬件中很大一笔投资；我们希望这笔投资也能有助于加快除法指令的执行。

这个运算器中所采用的策略是：设计一个高速乘法方案，同时让一般化的高基数 **SRT** 除法也能包含在里面．为了突出大量的递归步骤的设计，一些预备的和结束的步骤如象溢出检测和浮点数的准备等在解释时没有包括进去．今假定采用七字节 (56 位) 的分数数据．如果用一个附加的字节作为阶，那么这个运算器能扩充到处理 60 位 (8 字节) 的浮点操作。

图 7.16 指出了 ILLIAC-III 算术运算处理器的功能块图．图中所用的习惯说明如下：

1. 工作寄存器用 7 字节的长方形来表示，这些长方形再经过细分来指示字节．换句话说，每个寄存器有 56 位，每 56 个数位的 **SD** 数存储在二个 56 位的寄存器中 (每个带符号数字用二位)。

2. 组合逻辑模块用圆圈或带圆角的长方形来表示，逻辑门用符号 AND(∧)，OR(∨) 以及 XOR(⊕) 来表示。

3. 粗的总线表示数据是 **SD** 形式的．细的总线则相应于每个数位用 1 个二进位的普通数据．保存冗余 **SD** 数的"寄存器对"用这一对寄存器的缩写符号来表示，如象用 **USM** 来代表 **US** 和 **UM**。

4. 寄存器对寄存器的传送的控制信号表示成如下形式

$$\mathbf{S}D\mathbf{T}\quad \mathbf{S}R_n\mathbf{T}\quad \mathbf{S}L_n\mathbf{T} \tag{7.61}$$

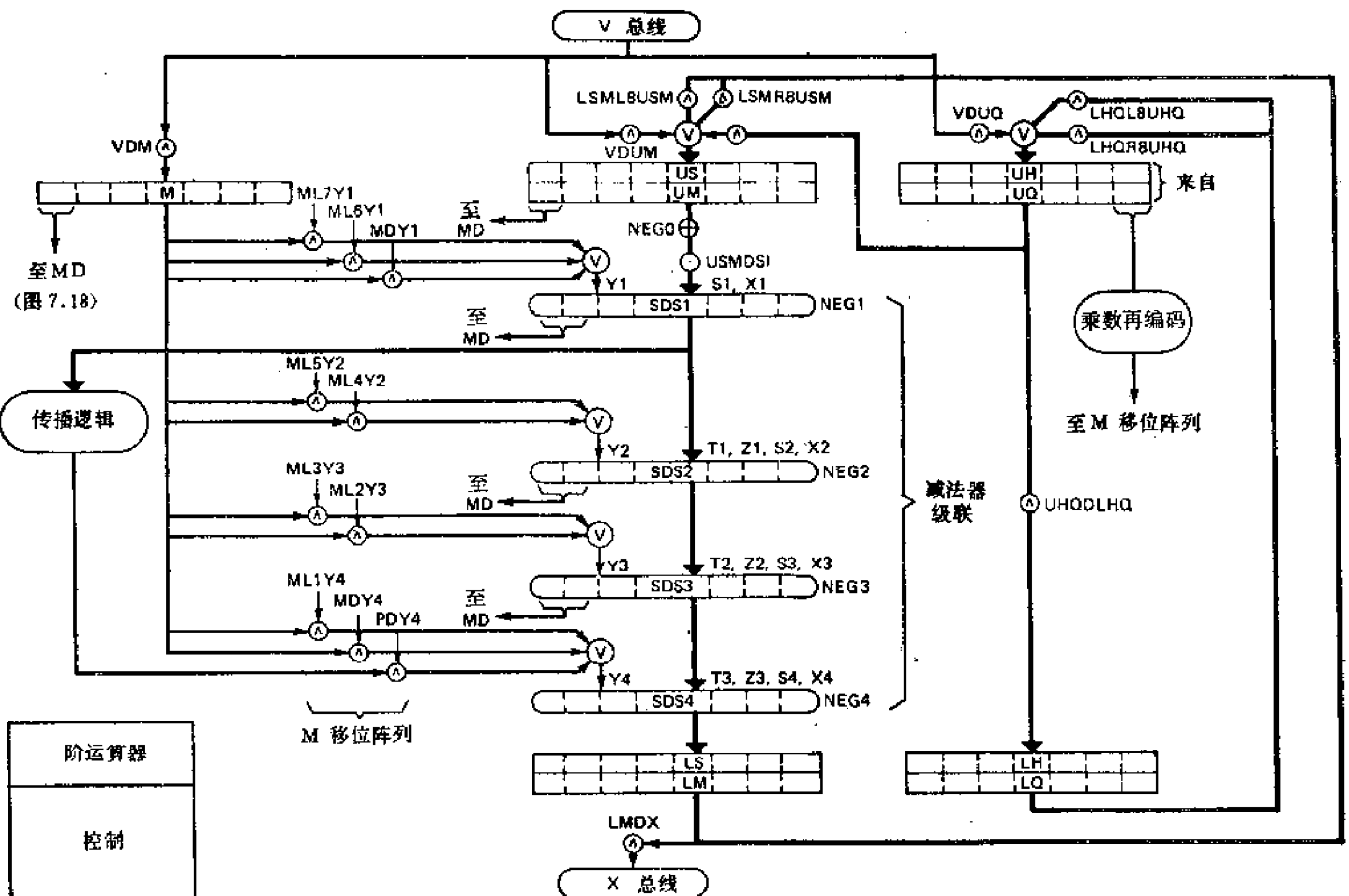


图 7.16 ILLIAC-III 算术运算处理器的功能块图（Atkins[13,4]）

这里 **S** 和 **T** 分别为源和目的寄存器. 中间的 "D" 是指把 **S** 的内容"直接传送"到 **T**, "R_n" 和 "L_n" 分别表示在向目的寄存器传送的过程中，源寄存器的内容右移或左移了几个位置.

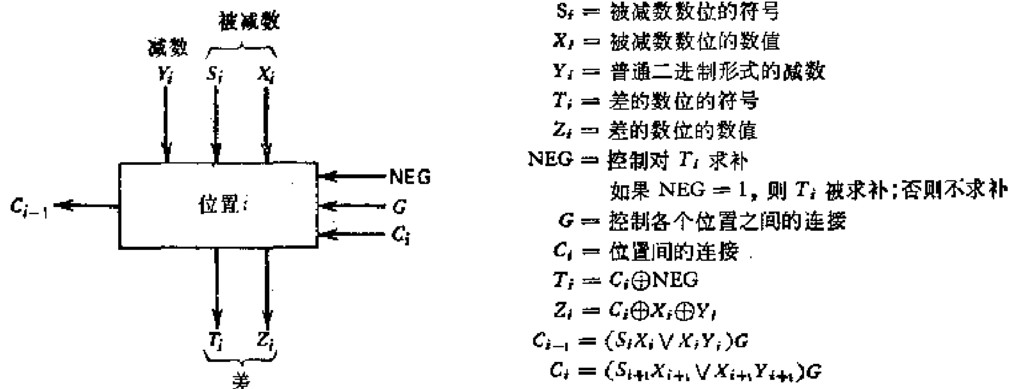


图 7.17　带符号数字的减法器 (SDS) 单元的一个典型级 (Atkins[4])

加法/减法逻辑用四个带符号数字的减法器 (**SDS**) 级联组成，在方块图中用 SDSi (其中 $i = 1, 2, 3, 4$) 来表示. 基本的 **SDS** 单元在图 7.17 中作了描述. **SDS** 的级联将对以普通形式存放在寄存器 **M** 中的数据和以 **SD** 形式存放在 **USM** 寄存器中的数据执行加法或减法. 这个级联引入了推迟进位/借位传播的装置，一直推迟到最后一步为止. 控制信号 NEG_i (对 $i = 1, 2, 3, 4$) 支配着所需要的"加"或"减"操作.

每条 Y_i 线(对 $i = 1, 2, 3, 4$) 是被乘数或除数的一位,它来自 **M** 寄存器. 注意: 下列操作是由 **SDS** 级联来执行的.

$$(\text{旧的部分乘积}) + (\text{被乘数}) = (\text{新的部分乘积}); \tag{7.62}$$

$$(\text{旧的部分余数}) - (\text{除数}) = (\text{新的部分余数}) \tag{7.63}$$

旧的部分乘积或余数中的每一个数位是由二个二进位 S_iM_i 组成的，它们存储在寄存器对 **USM** 中,这里 M_i 是数值位, S_i 是相联系的符号位. SDSi 的输出被用作 SDS$(i+1)$ 的输入,这里 $i = 1, 2, 3$. SDS4 的最后输出送到寄存器对 **LSM**.

这个 **SDS** 级联提供了基数为 256 的乘法;即同时扫描乘数的 8 位. 被乘数从 **V**-总线送入 **M** 寄存器，乘数进入 **UQ**. **UQ** 的低位字节驱动再编码逻辑,后者耦合到移位阵列的控制线上. 再编码要求对被乘数的 1, 2, 4, 8, 16, 32, 64 和 128 倍等倍数进行加和减. 这些倍数由"与"逻辑控制的移位阵列来形成. 在乘法的执行过程中, **LSM** 和 **LHQ** 的内容 (部分乘积和乘数) 右移 8 位返回到 **USM** 和 **UQ** 寄存器. 这样连续进行 8 个周期,在第 9 个周期通过转换逻辑把乘积转换成 **SD** 形式. 转换逻辑的输出送到 SDS4, 并且被转换成普通的表示式.

对于除法的执行, 除数放在寄存器 **M** 中,被除数以及下一个部分余数均以 **SD** 形式放在 **USM** 寄存器中. 商的数位以冗余的 **SD** 形式存放在 **UHQ** 寄存器中，符号位在 **UH** 中，数值位在 **UQ** 中. 除法中所用的转换逻辑与执行乘法时所用的相同. **UHQ** 的内容用控制信号 **UHQ**D**USM** 送到 **USM**，然后在最后一个周期转换成普通的二进制形式. 该硬件将要求在控制信号 **LSM**$L8$**USM** 的作用下使 **LSM** 左移 8 位进入 **USM**，以

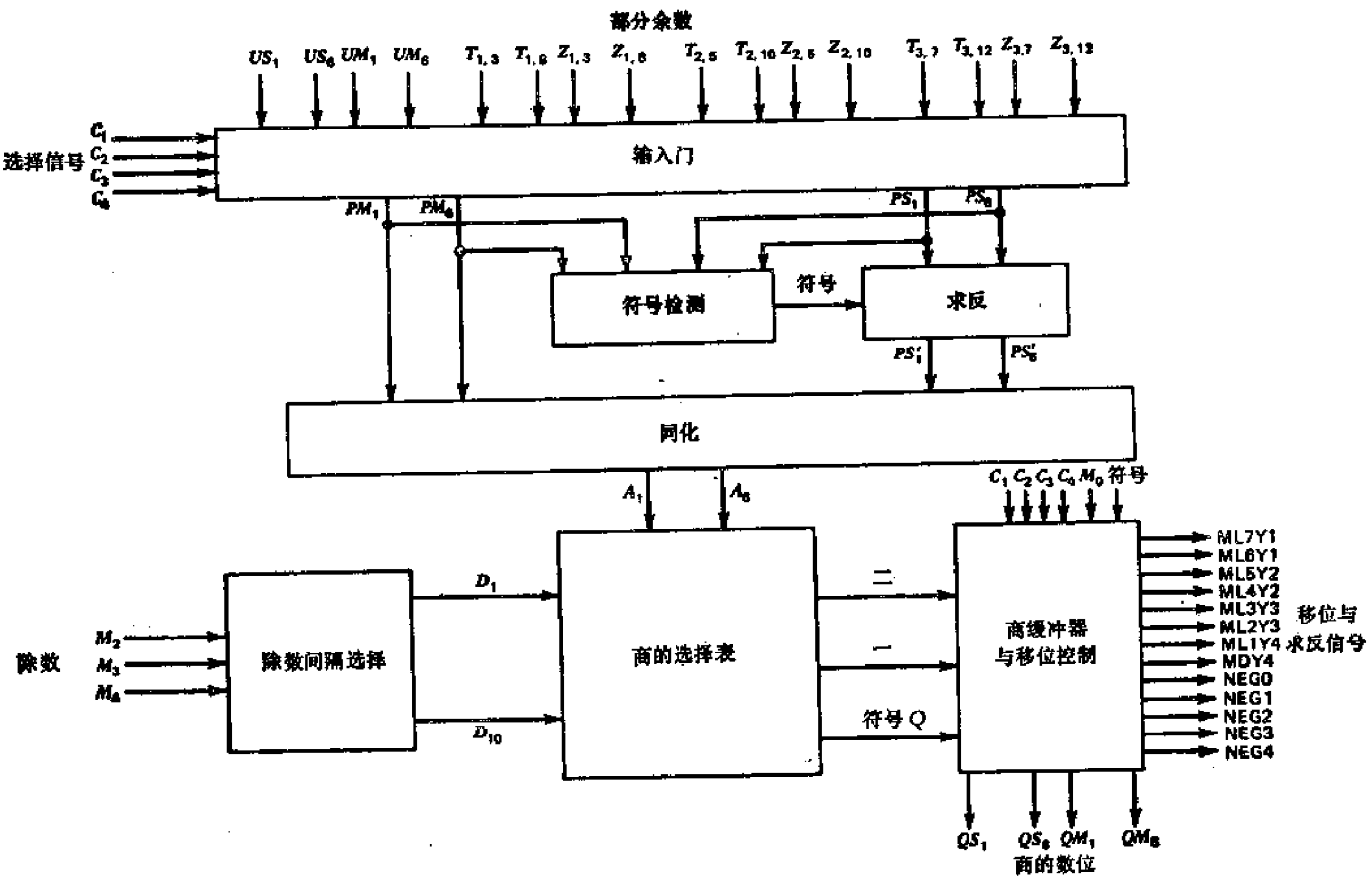


图 7.18 模型除法 (MD) 的功能块图 (Atkins[3])

及在控制信号 **LHQ**$R8$**UHQ** 的作用下从 **LHQ** 到 **UHQ** 等等.

在图 7.18 中粗略地画出了一个基数-4 的模型除法(**MD**),这里商的数位仅仅根据对当前的除数和部分余数中 6 个高位的检测来选择. 这个模型除法确定了通过 **SDS** 级联将除数的倍数从部分余数中减去. 下一个部分余数被存储在 **LSM** 中，然后左移 8 位进入 **USM**. 这些周期一直继续到产生满精度的商为止. 于是，这个商从 **UHQ** 直接送到 **USM**，再经过转换并送到 **LM**，在那里供控制处理器使用. 基数为 256 的除法可以通过连续应用四次基数为 4 的模型除法来达到.

表 7.4 提供了图 7.16 所示运算处理器和图 7.18 所示的 **MD** 部件的所有子块的功能一览. 必须指出,基数-4的乘法是在每个 SDS 级上进行的. 每个 SDS 级所选择的倍数的数值如下:(见图 7.18)

$$\begin{array}{llll} \text{SDS1} & 0,\ \pm 128A,\ \pm 64A & \text{SDS2} & 0,\ \pm 32A,\ \pm 16A \\ \text{SDS3} & 0,\ \pm 8A,\ \pm 4A & \text{SDS4} & 0,\ \pm 2A,\ \pm A \end{array} \tag{7.64}$$

其中 A 是被乘数. 按照表 7.5 中的规定，这里用了一些 3 位重叠的组来执行再编码. 模型除法可看作商的再编码器,它决定着要减去除数的哪些倍数. 与乘数再编码相似,这里已经把冗余引进商的表示式. 除数间隔选择逻辑 (**DISL**) 说明在表 7.6 中. 对二个非零分数 f_1 和 f_2 的除法,商的范围给定为 $\frac{1}{4}<f_1/f_2<16$. 带有零的除法在操作的第一步即被检测出来.

表 7.4 图 7.16 和图 7.17 中所有子块的功能一览表

缩写名称	名　称	说明[1]
M	被乘数寄存器	保存一个以普通二进制形式表示的数,以便从 **US-UM** 内以冗余形式表示的数中加上或减去
US	上符号寄存器	双列寄存器的第一列与 **SDS** 阵列相联系. 保存一些以 **SD** 格式表示的二进制数的符号位，以便用作 SDS1 的输入
UM	上数值寄存器	双列寄存器的第一列与 **SDS** 阵列相联系. 包含以 **SD** 格式表示的二进制数的数值,以便用作 SDS1 的输入
LS	下符号寄存器	作为 **US** 寄存器的第二列
LM	下数值寄存器	作为 **UM** 寄存器的第二列
UH	上 **H** 寄存器	双列寄存器的第一列，用于保存商的一些符号位以及使操作数对准和规格化
LH	下 **H** 寄存器	作为 **UH** 寄存器的第二列
UQ	上 **Q** 寄存器	双列寄存器的第一列,用于保存商的一些数值位,以及使操作数对准和规格化. 在乘法期间用于保存乘数
LQ	下 **Q** 寄存器	作为 **UQ** 的第二列

（续　表）

缩写名称	名　称	说明[1]
V-BUS	输入总线（"V"表示进入系统的箭头）	提供 AU 和交换网络之间的输入（至 AU）接口，包括电缆端接器和奇偶检测
X-BUS	输出总线	提供 AU 和交换网络之间的输出（自 AU）接口，包括电缆端接器和奇偶位的产生
MR	乘数再编码	将 UQ 寄存器的低 9 位再编码成一些信号，在乘法期间，利用这些信号来操纵 M 移位阵列的门
SDS1-SDS4	带符号-数字的减法器 1 至 4	带符号数字的减法器
PL	传播逻辑	由 S1 的 T_i 和 Z_i 输出产生 P_i 位．然后这些 P 位与 $S4$ 中的 Z_i 位相结合，产生同化结果．这个部件类似于先行借位逻辑
MSA	M 移位阵列	选择 M 寄存器的内容的倍数，以便加到减法器级联的 SDS1，SDS2，SDS3 和 SDS4 中
MD	模型除法	一种基数-4 的查表除法，它产生的商的数位被存储在 UH-UQ 中，并且在形成满精度的部分余数时用于控制 M 移位阵列
EAU	阶运算器	对浮点操作数的 7 位阶码执行算术运算
IG	输入控制	与-或门线路，用于使控制信号 C_i 选出的部分余数送到模型的下一级
SDT	符号检测	确定被选输入的符号，即领前的非零数位的符号．用来控制求反和形成商的数位的符号
NEG	求　反	通过对所有的 PS 位求补来使被选输入求反．利用这个特性，商的选择表只需要实现 Robertson 图的第一象限
ASM	同　化	将部分余数从 SD 格式转换成普通的二进制数．利用先行借位技术来加速这个步骤
DISL	除数间隔选择逻辑	对除数（即 M_1 至 M_4）译码．因为 $M_1=1$，所以 M_1 可以不要
QST	商选择表	从逻辑上实现 Robertson 图．它可以用二级管矩阵逻辑来构成
QBSC	商的缓冲器和移位控制	构成存储商的所有 8 个数位的单元，并且控制着到 UH-UQ 去的低位字节．产生 M-移位阵列的控制信号和 NEG1 信号，并用来控制对被选除数倍数进行 SDS 加法还是减法

1）AU＝运算器

表 7.5 在 ILLIAC-III 中采用的乘数再编码方案 (Wallace)

三位一组的乘数位			再编码数位/倍数选择¹⁾	三位一组的乘数位			再编码数位/倍数选择¹⁾
X_{i-1}	X_i	X_{i+1} ²⁾		X_{i-1}	X_i	X_{i+1} ²⁾	
0	1	1	$+2A$	1	1	1	·
0	1	0	$+A$	1	1	0	$-A$
0	0	1	—	1	0	1	·
0	0	0	0	1	0	0	$-2A$

1) A 为被乘数. 2) 开始时 $X_{i+1}=0$.

表 7.6 模型除法的除数间隔选择逻辑 (DISL) 的说明

间隔名称	逻辑方程式	除数 d 的表示范围	间隔名称	逻辑方程式	除数 d 的表示范围
D_1	$\bar{M}_{10}\bar{M}_{11}\bar{M}_{12}$	$\frac{1}{2}\leqslant d<\frac{9}{16}$	D_7	$D_1\vee D_2\vee D_3$	$\frac{1}{2}\leqslant d<\frac{11}{16}$
D_2	$\bar{M}_{10}\bar{M}_{11}M_{12}$	$\frac{9}{16}\leqslant d<\frac{5}{8}$	D_8	$D_4\vee D_5\vee D_6$	$\frac{11}{16}\leqslant d<1$
D_3	$\bar{M}_{10}M_{11}\bar{M}_{12}$	$\frac{5}{8}\leqslant d<\frac{11}{16}$	D_9	$D_4\vee D_5$	$\frac{11}{16}\leqslant d<\frac{7}{8}$
D_4	$\bar{M}_{10}M_{11}M_{12}$	$\frac{11}{16}\leqslant d<\frac{3}{4}$	D_{10}	$(D_5\vee D_6)=M_{11}$	$\frac{3}{4}\leqslant d<1$
D_5	$M_{10}\bar{M}_{11}$	$\frac{3}{4}\leqslant d<\frac{7}{8}$	D_{11}	$(D_1\vee D_2\vee D_3\vee D_4)=\bar{M}_{12}$	$\frac{1}{2}\leqslant d<\frac{3}{4}$
D_6	$M_{10}M_{11}$	$\frac{7}{8}\leqslant d<1$			

7.11 参考文献注释

本章涉及的材料参照了 Robertson[11] 及其后继者的思想. Mac Sorley[14], Garner[6] 和 Tung[17] 等曾经对这个课题作了杰出的综述. **SRT** 除法的发明应归功于 Sweeney[14], Robertson[11] 和 Tocher[15]. Metze[10], Wilson 和 Ledley[18] 则对 **SRT** 除法作了改进. 高基数除法的方法最早是由 Robertson[11] 提出的，后来又由 Atkins[2,3,4] 和 Kalayciogln[8] 作了论述. Freiman[5] 和 Shively[13]通过统计分析对 **SRT** 除法的特性作了评价. ILLIAC-III 算术运算设计的实例研究选自 Atkins 的著作,特别是选自他的理科硕士 (M.S.) 论文[3],同时也参考了[4]. Robertson 图是在 Robertson 的开创性工作之后命名的. 我们鼓励有兴趣的读者去研究一下 Robertson 的最近的著作[12],这篇著作中建立了数字除法的方法和乘数再编码参数之间的对应关系. 正如第 7.10 节所述，这种对应关系已被用在 ILLIAC-III 的算术运算结构中.

参 考 文 献

[1] Avizienis. A., "The Recursive Division Algorithm," *Class Notes*. Engr. 225A. UCLA. Los Angeles, California. 1971.

[2] Atkins. D. E., "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders." *IEEE Trans. Comput.*, Vol. C-17, No. 10. October 1968, pp. 925—934.

[3] Atkins. D. E., "The Theory and Implementation of SRT Division," *Tech. Report No. 230*. Dept of Computer Science, University of Illinois. Urbana, Illinois, 1967.

[4] Atkins, D. E., "Design of Arithmetic Units of ILLIACIII: Use of Redundancy and Higher Radix Methods," *IEEE Trans. Computer*, August 1970, pp. 720—733.
[5] Freiman, C. V., "Statistical Analysis of Certain Binary Division Algorithms." *Proc. of IRE*, Vol. 49, No. 1, January 1961, pp. 91—103.
[6] Garner, H. L., "Number Systems and Arithmetic," in *Advances in Computers*, Vol. 6, 1965, pp. 168—177
[7] Gilman, R. E., "A Mathematical Procedure for Machine Division," *Comm. of Ass. for Comp Mach.*, Vol. C, No. 4, April 1959, pp. 10—12.
[8] Kalaycioglu, V. U., "Analysis and Synthesis of Beneralized-Radix Additive Normalization Division Techniques," *Tech. Report*, Dept. of Elec. & Comp. Engr., University of Michigan, Ann Arbor, Michigan, May 1975.
[9] Krishnamurthy, E. V., "On Range-Transformation Techniques for Division," *IEEE Trans. Comput.*, Vol. C-19, No. 3, March 1970, pp. 227—231.
[10] Metze, G., "A Class of Binary Divisions Yielding Minimally Represented Quotients', *IRE Trans.*, Vol. EC-11, No. 6, December 1962, pp. 761—764.
[11] Robertson, J. E., "A New Class of Digital Division Methods," *IEEE Trans. Comput.*, Vol. C-7 September 1958, pp. 218—222.
[12] Robertson, J. E., "The Correspondence Between Methods of Digital Division and Multiplier Recording Procedures," *IEEE Trans. Comput.*, Vol. C-19, No. 8, August 1970, pp. 692—701.
[13] Shively, R. R., "Stationary Distribution of Partial Remainders in SRT Digital Division," *PH. D. Thesis*, University of Illinois, Urbana, Illinois, 1963.
[14] MacSorley, O. L., "High-Speed Arithmetic in Binary Computers," *Proc. of IRE*, Vol. 49 January 1961, pp. 67—91.
[15] Tocher, K. D., "Techniques of Multiplication and Division for Automatic Binary Computers," *Quart. J. Mech. Appl. Math.*, Vol. XI, Pt. 3, 1958, pp. 364—384.
[16] Tung, C., "A Division Algorithm for Signed-Digit Arithmetic," *IEEE Trans. Comput.*, Vol. 17 1968, pp. 887—889.
[17] Tung, C., "Arithmetic," Chap. 3 in *Computer Science*, A. F. Cardenas, et al. (eds.), Wiley-Interscience, N. Y., 1972.
[18] Wilson, J. B. et al., "An Algorithm for Rapid Binary Division," *IRE Trans.*, Vol. EC-10, No. 4, December 1961, pp. 662—670.

习　　题

题 7.1　将下列数据应用到 16 位的带恢复的除法器(图 7.3)，列出相继的部分余数、除数、链接位、加法器输入和输出以及生成的商位等内容．排列成类似于图 7.1 的表格形式．

被除数 $A = 0.0110110111100111$

除　数 $B = 0.10100111$

题 7.2　利用图 7.6 中给出的算法，将下列基数为 r 的冗余带符号数字的数变换成普通的二进制形式

(a) $Q = (7\bar{6}9\bar{2})_{10}$ 对 $r = 10$;

(b) $Q = (\bar{7}65\bar{4})_8$ 对 $r = 8$;

(c) $Q = (1\bar{1}\bar{1}1\bar{1}1)_2$ 对 $r = 2$.

通过检查所表示的数值来验证你的答案．

题 7.3　说明为什么一个二进制 SRT 除法器的效率对所用除数的范围是很敏感的．试证明当 $D = 3/4$ 时，商将成为典型的最小带符号数字的形式．

题 7.4　给定 $N = 0.0101100010110111$ 和 $D = 0.1001$，应用 Wilson-Ledley 算法，在 $n = 10$ 的情况下求出商 $Q = N/D$.

题 7.5　试证明当除数处在 $\frac{3}{5} \leqslant |D| \leqslant 3/4$ 的范围内时，Metze 修改的 SRT 方法与原始的 SRT 除法是一致的．

题 7.6 试证明 Metze 修改的 SRT 除法常常得出最小带符号数字的商，甚至在除数超出题 7.5 指定的范围时也是如此。

题 7.7 试证明：为满足基数-r 的 SRT 除法的要求，方程式 7.55 中所用的参数 m 必须在

$$(r-1)/2 \leqslant m \leqslant r-1$$

的范围内选择。

题 7.8 给图 7.17 中定义的带符号数字的减法器（SDS）单元提出一个可行的实现方案，指明用 56 个这样的 SDS 单元来实现 7 字节 SDS 减法器设计的完整线路图。然后，再以数值例子来阐明如何用这个部件从一个二进制编码的带符号数字的数中减去一个普通的二进制数。

题 7.9 试比较二进制带恢复除法和二进制不恢复除法的处理速度和硬件要求。在不恢复除法的情况下，将带符号数字的商转换成普通二进制形式所需要的硬件在比较时也应该包括进去。

题 7.10 从设计者和用户二个方面的观点来说明二进制 SRT 除法与其他快速标准除法相比较时的特点。确定高基数 SRT 除法的硬件复杂性比二进制 SRT 除法相对地增加了多少。

第八章　收敛除法与单元阵列除法器

8.1 引　　言

大多数普通的计算机都采用上一章所描述的试凑减法的除法方案．本章要介绍几种非普通的除法方法，它们不同于试凑减法的除法．

我们首先说明用重复的乘法来实现除法．这种乘法性的除法有二种类型．第一种方法把被除数和除数当作一个分数的分子和分母来处理．通过对分子和分母用同一收敛因子的序列相乘，直到分母趋向于单位值为止，由此来完成除法．最后得到的分子就成为所需要的商．第二种实现除法的方法是求除数的倒数，然后将被除数与这个倒数相乘来求出商．这二种方法都是按二次幂收敛于商．收敛除法已被用在某些大型计算机中，如象 CDC**6600** 和 **7600**，IBM **360** 系统 **91** 型．这类除法方案的主要优点在于共享了乘法所需要的硬件．具备一个快速的硬件乘法器，对于通过二次收敛来实现乘法性的除法是很关键的．

本章后半部用来阐明并行和半并行二进制除法的各种高速叠接单元阵列的设计和组成．大型组合逻辑阵列被用于实现除法中所需要的比较操作．在阵列中，移位是通过实际接线来实现的．单元阵列的方法特别适合于大规模集成电路的制造．单元阵列除法器所需要的控制线路较少，与标准除法器相比还能提供较满意的速度．这里要介绍的三类除法阵列是：带恢复的、不恢复的以及增量的除法阵列．我们将详细介绍 Cappa-Hamacher[3] 阵列除法器如何利用进位存储和先行技术来加速阵列中进位的传播．提出了对所有这些单元阵列除法器的价格-效率的分析．为了与前面的除法器说明取得一致，所有被处理的数都假定是正的分数．

8.2 收敛除法的方法

下面提出的一类除法的方法与上一章介绍的那些方法完全不同．这类方法使用了乘法硬件，通过收敛逼近来完成除法．为了产生所需要的商，要求执行叠代的乘法．

让我们来考虑除法操作

$$\frac{N}{D} = Q \tag{8.1}$$

这里商可以当作一个分数来处理，这个分数的分子是 N，分母是 D．每次叠代，选择一个常数因子 R_k 去乘分子和分母，这并不改变比值的实际数值．乘法因子的序列 $R_k(k=0, 1, \cdots, n)$ 是这样选择的，即让所得到的分母按二次幂向单位值收敛，从而使分子也按二次幂向所要求的比值或商 Q 收敛．

这个概念可以从数学上说明如下

$$\frac{N}{D}=\frac{N}{D}\times\frac{R_0}{R_0}\times\frac{R_1}{R_1}\times\cdots\times\frac{R_n}{R_n} \tag{8.2}$$

对足够大的 n，使

$$D\times R_0\times R_1\times\cdots\times R_n\rightarrow 1$$

于是我们得到

$$N\times R_0\times R_1\times\cdots\times R_n\rightarrow Q \tag{8.3}$$

为了适应浮点运算的设计，我们假定N和D二者均为正的规格化分数，它们满足下列范围

$$\frac{1}{r}\leqslant N,\quad D<1 \tag{8.4}$$

一般的规则是：选择相继的乘数 R_i（对 $i=0,1,\cdots,n$），使得

$$D_{i-1}<D_i \tag{8.5}$$

（对所有的 $i=1,2,\cdots,n$），这里

$$D_0=D\times R_0 \tag{8.6}$$

和

$$D_i=D\times R_0\times R_1\times\cdots\times R_i=D_{i-1}\times R_i \tag{8.7}$$

再用新的因子 R_{i+1} 与 D_i 相乘，这个过程将一直继续到某个整数 $i=n$，使分母在最大的机器精度内

$$D_n\rightarrow 1.0 \tag{8.8}$$

方程式 8.4 中规格化的范围意味着除数能表示为

$$D=1-\delta \tag{8.9}$$

对于这个范围内的某个 δ，有

$$0<\delta\leqslant\frac{r-1}{r} \tag{8.10}$$

我们可以选择第一个乘法因子

$$R_0=1+\delta, \tag{8.11}$$

于是

$$D_0=D\times R_0=(1-\delta)\times(1+\delta)=1-\delta^2 \tag{8.12}$$

显然，$1-\delta^2$ 大于 $1-\delta$，而且更接近于单位值。对第二次叠代，选择乘法因子

$$R_1=1+\delta^2 \tag{8.13}$$

因此

$$D_1=D_0\times R_1=(1-\delta^2)\times(1+\delta^2)=1-\delta^4 \tag{8.14}$$

一般地说，在第 i 次叠代时，有

$$R_i=1+\delta^{2^i} \tag{8.15}$$

和

$$D_i=D_{i-1}\times R_i=(1-\delta^{2^i})\times(1+\delta^{2^i})=1-\delta^{2^{i+1}} \tag{8.16}$$

方程式 8.5 所以能得到满足，是因为 δ 是一个适当的分数，

$$\delta^{2^i}>\delta^{2^{i+1}} \tag{8.17}$$

所以

$$D_{i-1}=1-\delta^{2^i}<1-\delta^{2^{i+1}}=D_i \tag{8.18}$$

在二进制的情况下，$\delta < \frac{1}{2}$，因此

$$\delta^{2^i} < 2^{-2^i} \tag{8.19}$$

于是

$$1 - D_i = \delta^{2^{i+1}} < 2^{-2^{i+1}} \tag{8.20}$$

对足够大的 i，即 $i = n$，方程式 8.20 中所指出的误差可以做到相当小，以至可以忽略.

值得指出的是：方程式 8.15 中规定的因子 R_i 可以直接依靠求 D_{i-1} 对基数的补码(在二进制的情况下即为 2 的补码)得出如下

$$R_i = 2 - D_{i-1} = 2 - (1 - \delta^{2^i}) = 1 + \delta^{2^i} \tag{8.21}$$

从 D_{i-1} 求 R_i 的简单性是这种方法的主要特征. 在 n 次叠代以后，分母

$$D_n = 1 - \delta^{2^n} \tag{8.22}$$

成为给定机器精度范围内所表示的一个 1，即全 1 向量. 因为在方程式 8.22 中的误差 $\delta^{2^n} < 2^{-2^{n+1}}$，即小于全字长的最低有效位，于是分子就成为所要求的商，它具有下列确切数值

$$Q = \frac{N}{D} = N \times R_0 \times R_1 \times \cdots \times R_n$$

$$= N \times (1 + \delta) \times (1 + \delta^2) \times \cdots \times (1 + \delta^{2^n}) \tag{8.23}$$

在输入除数 D 的 δ 值较小时，收敛的速度也较快. 这种二次收敛除法的方法对二进制数的系统特别适合，这是因为求二进制数的补码是很容易的. 如果除数允许表示为按位规格化的形式，那么它也能应用到高基数的系统. 关于三次或高次收敛除法的方法将作为练习留给读者.

8.3 采用乘法的除法器的设计

前节描述的收敛除法方案实际上已经在几种现有的计算机中实现了. IBM **360** 系统 **91** 型的算术运算执行部件就是一个很好的例子. 我们将说明在 **91** 型计算机中怎样把乘法收敛方法应用到设计有效的除法器.

IBM 的机器除法处理 56 位长的分数. 除数 D 是一个具有下列形式的按位规格化的浮点表示的分数

$$D = 0.1\underbrace{\text{xxx}\cdots\text{x}}_{56\text{位}} \tag{8.24}$$

因此，收敛常数 δ 以 0.5 为上界

$$\delta = 1 - D \leqslant 0.5 \tag{8.25}$$

具有 56 位的精度时，单位(1)的内部表示式通常是

$$1 = 0.\underbrace{111\cdots1}_{56\text{位}} \tag{8.26}$$

每当收敛因子 δ^{2^k} 小到足以使

$$\delta^{2^k} < 2^{-56} \tag{8.27}$$

(这里 2^{-56} 相应于除数的最低位)时，所得到的分母将趋近于方程式 8.26 所指示的 1.

第 i 次叠代的乘法因子 R_i 可以通过对分母 D_{i-1} 取 2 的补码来得出（D_{i-1} 是在第 $(i-1)$次叠代时求出的），正如方程式 8.21 所指定的那样．对一个 56 位的分数，用五个乘法因子 R_0, R_1, R_2, R_3, R_4 与插在中间的求 2 的补码的操作就已经足够了，这时商

$$Q = N \times R_0 \times R_1 \times R_2 \times R_3 \times R_4 \tag{8.28}$$

式中 $R_4 = 2 - D_3$ 和 $D_3 = D \times R_0 \times R_1 \times R_2 \times R_3$.

如果乘数中的位数减少，那么每次乘法的时间可随之减小．为了求出收敛的 V 位，初始乘数 R_0 被截断到 t 位，作为 $1+\delta_t$，这里 $\delta_t - \delta < 2^{-t}$．可以指出，最后得到的分母等效于

$$D_1 = (1 - \delta) \times (1 + \delta_t) = 1 - \delta^2 + \Delta_t \tag{8.29}$$

这里因数 Δ_t（对 $0 < \Delta_t < 2^{-t}$）是由于截断引起的．因为附加的 Δ_t 常常是正的，所以分母可以从上或从下二种形式向单位值收敛

$$D_i = \begin{cases} 0.11111\cdots xxxx\cdots \\ 1.0000\cdots xxxx\cdots \end{cases} \tag{8.30}$$

根据所需要的收敛速度，乘数可以用 0 的子串或 1 的子串来改编．正因为串的各位是全零或全 1，所以它们在乘法过程中可以跳过．于是，在 IBM**360/91** 中乘法时间显著地得到改进，除法时间也是如此．

为了改进初始的最小串长，并减少叠代次数，第一个乘数 $R_0 = 1 + \delta$ 由查表产生（用除数的前七位寻址）．第一次相乘保证了结果的小数点右边具有七个相同的位，即 $1 \pm \delta$ 具有形式 $\bar{a}.aaaaaaa\cdots$．以上与除法（DIVIDE）的执行相联系的操作均归纳在图 8.1 的流程图中．

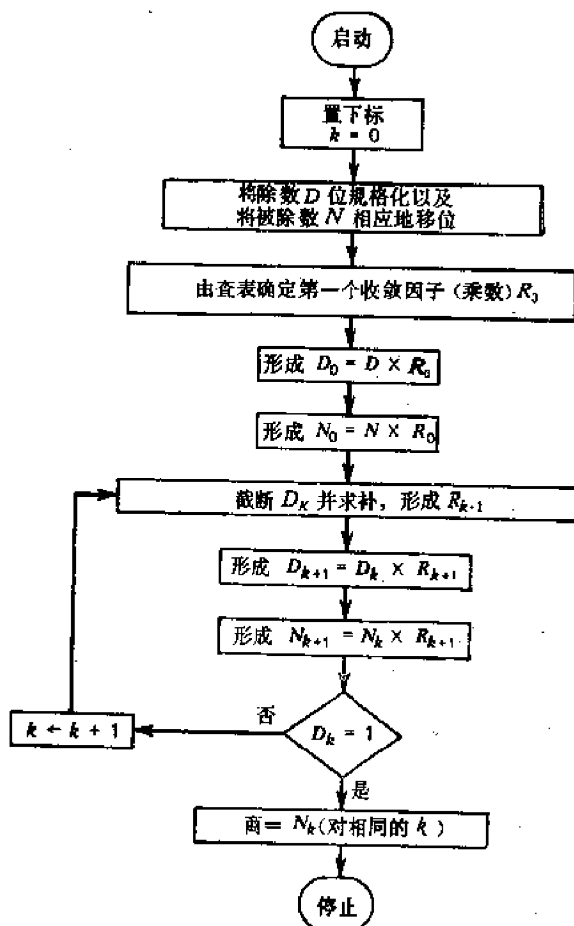


图 8.1 利用乘法收敛的方法在执行除法时的操作步骤

这个设计企图利用图 8.2 所画的乘法（MULTIPLY）硬件来实现流程图中规定的操作．进位-存储加法器迴路被用于完成每步乘法所需的六个操作数的加法．在设计可以认为完成之前，应该考虑三个主要问题．

1. 乘数 R_i 在每次叠代时具有可变的长度．由查表确定的第一个乘数长为 10 位，并且得出最小的 0 或 1 的串的长度为 7 位；第二个乘数为 14 位；第三个乘数为 28 位，等等．最小的 0 或 1 的串长在每次叠代时加倍．

2. 一次叠代所得到的结果就是下一次叠代的被乘数．进位传播加法器必须在除法循环中把进位-存储加法器的"和"与"进位"输出合并在一起．

3. 每次叠代需要二个乘数——一个

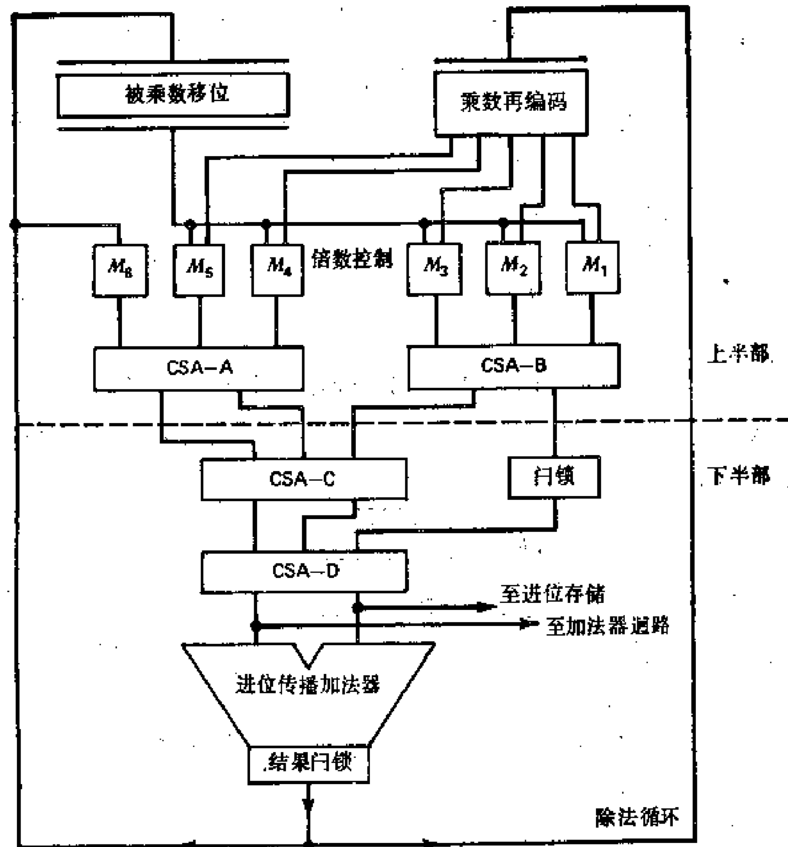


图 8.2 乘法性除法方案用 **CSA** 实现的方法（Anderson 等[1]）

用于乘分子，一个用于乘分母．但我们力求使二个乘法交叠，并用一个乘法线路来执行它们．这样会使硬件和时间两者都得到节省．

表 8.1 中的项目指出了五次叠代中相继的分母和它们的乘数的格式． 表 5.4 中的项目则指出了在一个乘数中领前的"1"或"0"是可以跳过的，这是因为它们引起乘数再编器有一个零倍输出．如果乘数再编码器的输入被求补，那么输出符号的变化不会影响到它的数值．这个特性将被用在再编码器的输出端产生 $\pm\delta^2$．

为了开始执行除法，除数 D 与第一个乘数 R_0 相乘，在 CSA 树的输出端产生第一个分母 D_0． 这二个输出由进位传播加法器组合在一起，并且返回到输入端作为新的被乘数．截断并求补后的输出形成了新的乘数．在除法循环上、下二半部的输出端用了两个暂存器闩锁．因此，一旦 $D \times R_0$ 被控制进入 CSA-C，下一个乘法 $N \times R_0$ 就可以开始．现在，$D_0 = D \times R_0$ 推入结果闩锁，循环返回到启动下一个乘法 $D_0 \times R_1$． 这时，被闩锁在 CSA-C 中的 $N \times R_0$ 通过加法器树推人结果闩锁．于是，这二个乘法互相跟随环绕着除法循环进行．第一个决定着第二个应该怎样乘才能最终收敛于商．

这一连串操作将一直继续到最后一个乘数 R_4 被使用为止． 由于分母 D_4 等效于 1，所以乘法已经完成． 如表 8.1 中所示，乘数一次控制 12 位． 最后乘的结果（$N \times R_0 \times R_1 \times R_2 \times R_3$）$\times R_4$ 就是最后的商 Q． 图 8.3 指出了时间图，图中显示的除法循环是重叠并行的．每完成一次分母的乘法，乘数再编码器的闩锁就会有变化． 在该部件中常常有二个乘法在执行，一个在除法器的上半部，一个在除法器的下半部，这二部分在图 8.2 中

表 8.1 在用乘法收敛的除法过程中，分母以及它们的乘数的格式（Anderson 等[1]）

数位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
D	0	1xxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0000	0000	
R_0	[01	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xx}$0]	由分母查表决定													
$D \times R_0 = D_0$	{0	1111	111x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	
	1	0000	000x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxx	xxxxx	xxxx	xxxx	xxxx	
R_1	{1	0000	[000$\overline{x}$	$\overline{xxxx}$	xx11	1]	由分母求补决定											
	0	1111	[111$\overline{x}$	$\overline{xxxx}$	$\overline{xx}$11	1]												
$D_0 \times R_1 = D_1$	{0	1111	1111	1111	11xx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	
	1	0000	0000	0000	00xx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	
R_2	{1	0000	0000	000[0	00$\overline{xx}$	$\overline{xxxx}$	$\overline{xxx}$1]											
	0	1111	1111	111[1	11$\overline{xx}$	$\overline{xxxx}$	$\overline{xxx}$1]											
$D_1 \times R_2 = D_3$	{0	1111	1111	1111	1111	1111	111x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	
	1	0000	0000	0000	0000	0000	000x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	
R_3	{1	0000	0000	0000	0000	0000	[000$\overline{x}$	$\overline{xxxx}$	$\overline{xxxx}$	1]								
	0	1111	1111	1111	1111	1111	[111$\overline{x}$	$\overline{xxxx}$	$\overline{xxxx}$	1]								
$D_2 \times R_2 = D_4$	{0	1111	1111	1111	1111	1111	1111	1111	1111	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	
	1	0000	0000	0000	0000	0000	0000	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	
R_4	{1	0000	0000	0000	0000	0000	0000	0000	0[000	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{x[x]xx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{x[x]xx}$	$\overline{xxxx}$	$\overline{xxxx}$	11]
	0	1111	1111	1111	1111	1111	1111	1111	1[111	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{x[x]xx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{x[x]xx}$	$\overline{xxxx}$	$\overline{xxxx}$	11]
D_4	{0	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111x	
（不形成）	1	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	

注：短精度的除法结果为 $N_4 = N \times R_0 \times R_1 \times R_2 \times R_3$。
长精度的除法结果为 $N_5 = N_4 \times R_4$。

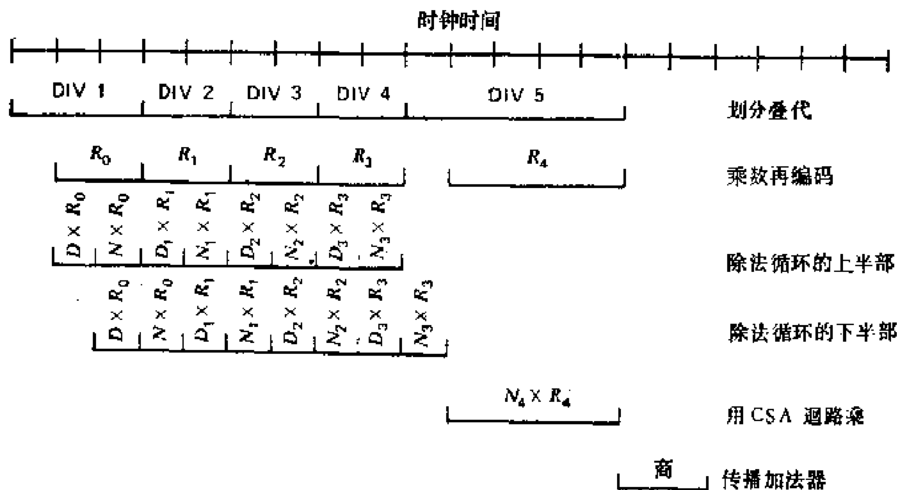


图 8.3 说明图 8.2 的乘法性除法迴路中执行情况的时间图

用虚线分隔开.

8.4 通过对除数求倒数实现除法

这一节中给出用于二进制除法的另一种乘法收敛的方法. 这个方法利用叠代过程来产生除数的倒数. 然后用被除数与除数的倒数相乘来求出商. 为了求出所需要的倒数,下面介绍一种收敛算法. 这种方法对于求二进制倒数来说证明特别有效,这是因为它便于用组合逻辑电路来实现. 在这个除法方案中也需要一个有效的硬件乘法器.

令方程式 8.1 中定义的 Q 就是所要求的商. 除数 D 假定是一个正的、规格化的分数,它在方程式 8.4 所规定的范围之内. D 的倒数用 $1/D$ 来表示,因此它的范围是

$$1 < \frac{1}{D} \leqslant 2 \tag{8.31}$$

对 $1/D$ 的初始逼近值 p_0 可以用组合逻辑部件或通过一个 ROM 查表求出. 这样一个初始逼近可以用图 8.4 中的方块图来描述. 为了减少硬件的复杂性,只是除数 D 的数位 $d_1 = 1$ 右边前 k 个数位 $d_2d_3\cdots d_{k+1}$ 需要作为逼近器方框的输入,而这个方框的输出则形成具有下列形式的二进制逼近值 p_0

$$p_0 = 1.s_1s_2\cdots s_t \tag{8.32}$$

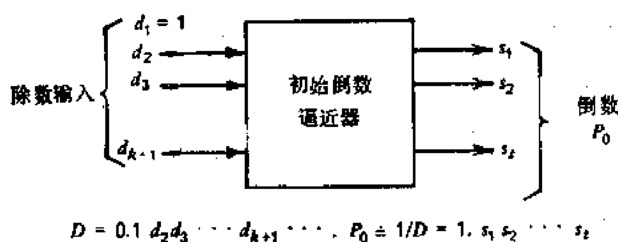
这里 k 和 t 值取决于硬件和精度要求. 在图 8.4 中还给出了一个局部表格,它说明了在 $k = 2$ 和 $t = 4$ 的具体情况下逼近器的转换逻辑.

显然,p_0 是对 $1/D$ 的一个分段逼近值,在 1/2 与 1 之间分成 2^i 个相等的间隔,每个间隔内 p_0 保持不变. 不难证明,对第 h 个间隔 ($h = 1, 2, \cdots, 2^i$),p_0 的最佳值是相应于它的中点的数的倒数,即

$$p_0(h) = \frac{2^{i+1}}{2^i + h - \frac{1}{2}} \tag{8.33}$$

对于 $i = 2$ 的情况,初始逼近值 p_0 随除数 D 的数值的变化曲线在图 8.5 中给出.

利用初始条件 p_0 和 $a_0 = p_0 \times D$,最后的倒数可以通过下列递归过程求出



输入		输出			
d_2	d_3	S_1	S_2	S_3	S_4
0	0	1	1	1	1
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	0

图 8.4 说明初始倒数逼近器的方块图和部分表格

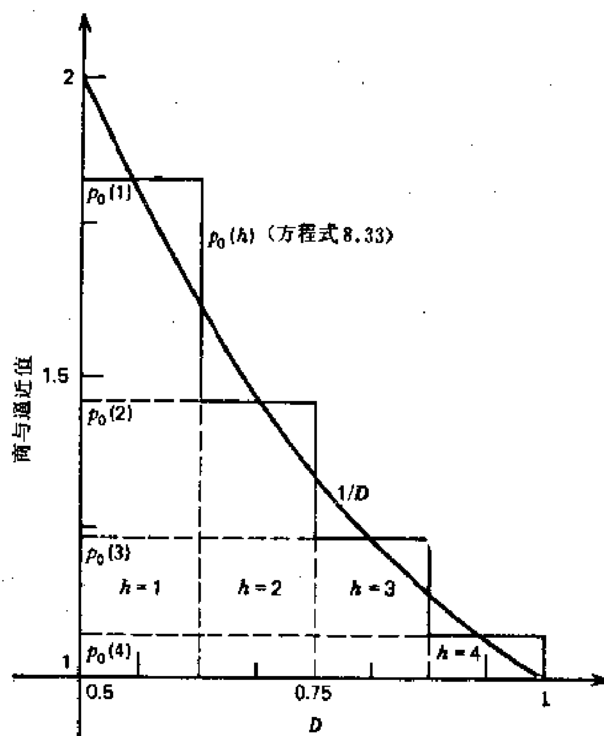


图 8.5 倒数 $1/D$ 及其分段逼近 $p_0(h)$ 随规格化除数 D 的变化 (Ferrari[4])

$$p_i = p_{i-1} \times (2 - a_{i-1}) \tag{8.34}$$

这里

$$a_i = a_{i-1} \times (2 - a_{i-1}) \tag{8.35}$$

和 $i = 1, 2, \cdots$ 为递归下标．理论上，对于所有的 i，随着叠代次数的增加，逐次逼近值 p_i 将按二次幂趋向于最后的倒数，即

$$\lim_{i \to \infty} p_i = \frac{1}{D} \tag{8.36}$$

和

$$\lim_{i \to \infty} a_i = 1 \tag{8.37}$$

实际叠代次数决定于机器的精度．令 ε 是一个极小的数，它反映机器字的最低有效位的数量级．要求 n 次叠代满足

$$|1 - p_n \times D| < \varepsilon \tag{8.38}$$

这里 p_n 就是所要计算的倒数的最后逼近值． 这个倒数算法的电路实现方法将在下一节中给出．

8.5 使用 CSA 树的二进制倒数器

目前已经有几种运算电路来实现上述倒数算法． Wallace[18] 提出用多级进位-存储加法器(**CSA**)树来执行倒数收敛过程中所需要的一对重叠的乘法．Stefanelli[16] 提出用一个不同的叠接阵列来产生以冗余 **SD** 编码表示的除数的倒数，那里需要一个代码转换器，

以便将冗余 **SD** 码变回到普通的二进制码．这一节只研究 Wallace 树的方法．

方程式 8.34 和 8.35 中规定的二个乘法可以用图 8.6 所示的 **CSA** 树来执行．在收敛过程的每一次叠代中，该加法器树允许二个乘法重叠执行．值得指出的是：第 i 次叠代时，二个操作共享同一个乘数 $(2-a_{i-1})$．二个乘法操作所用到的被乘数 p_{i-1} 和 a_{i-1} 在不同级上输入到 **CSA** 树；这就使重叠成为可能．

使用冗余 **SD** 码的乘数再编码是由 Wallace 提出的，目的是为了减少被加数（被乘数的倍数）的总个数，这些被加数都是加法器树作加法时所需要的．完成这些被加数加法的时间正比于被加数个数的对数．再编码方案只需要那些通过移位和求补便能求出的被乘数倍数．这也是一种局部的再编码，每个再编码数位仅仅依赖于一小组原始的乘数位．例如，基数 4 的再编码乘数数字属于集合 $\{-2,-1,0,+1,+2\}$，每一个都能用三个相邻的原始位来确定．这将使第五章中描述的被加数的个数减半．用 2 的补码校正电路可以获得负的倍数．

在图 8.4 所示的逼近器方框中，根据所选择的 $k=6$ 输入和 $t=4$ 输出，初始逼近值 p_0 可以经过再编码只给出三个被加数．在 Wallace 的原始设计中

$$|1-a_0| \leqslant 2^{-5} \tag{8.39}$$

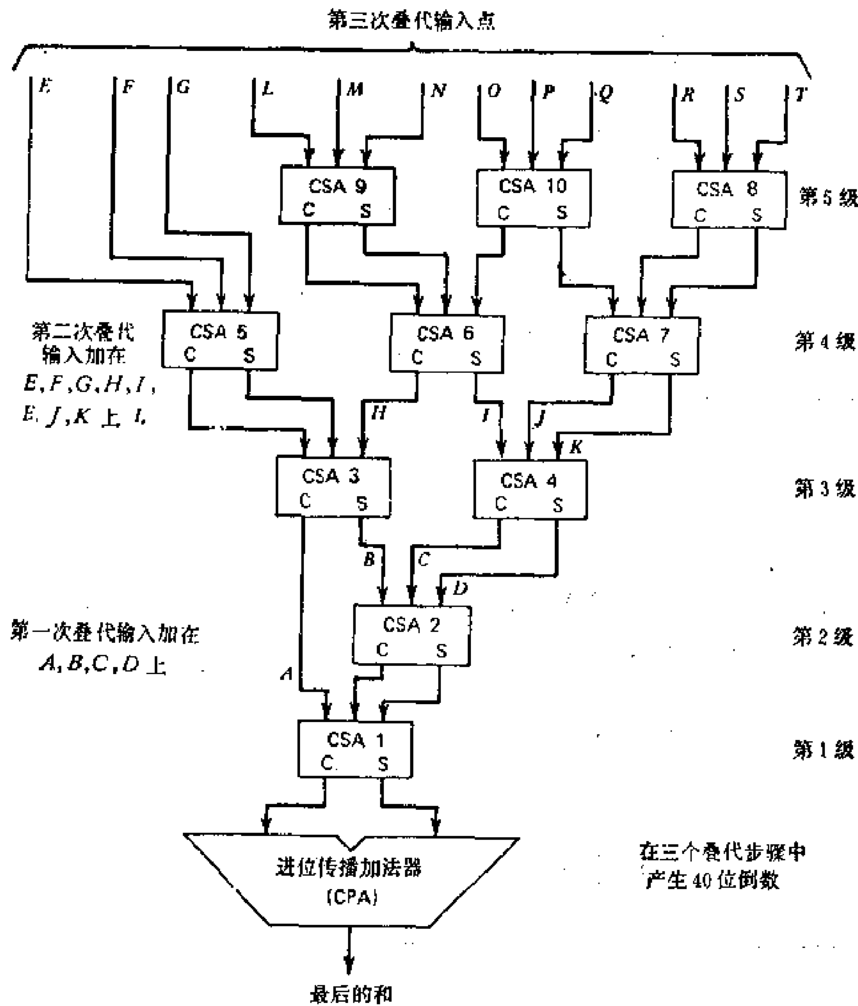


图 8.6 Wallace[17] 给除法提出的进位-存储加法器 (**CSA**) 树，这里用了叠代的二进制倒数

而 a_0 具有形式

$$a_0 = p_0 \times D = \bar{x}.xxxxxefghjk \tag{8.40}$$

第一个叠代步骤应该在二进制小数点后面立即产生 10 个相同的数位. 可以证明,具有方程式 8.40 中的形式的下一个 $a_1 = a_0 \times (2 - a_0)$ 能用近似的乘数来求出，这个近似的乘数未必就是 $2-a_0$,但是

$$2 - a_0 \simeq 1 - 2^{-5} \times (\bar{x}.efghj) \tag{8.41}$$

这里,括弧中的数可看作一个带符号的 2 的补码分数. 这个近似的乘数可以直接从 a_0 求出，如方程式 8.40 所示. 在这个乘数再编码之后,只需要四个被加数. 同样地,一个近似的乘数也能用在下一个要求 7 个被加数的叠代步骤中,以及用在要求 12 个被加数的第三个叠代步骤中. 产生一个 40 位的倒数,总共只需要三个叠代步骤.

使用近似乘数的二个明显优点说明如下:

1. 再编码乘数数位的个数少，使得只要用 **CSA** 树的一部分来做乘法，因而乘法时间缩短了(特别是最初的几步). 例如,对 $p_0 \cdot D$ 的四个被加数可以在第一个叠代步骤中从点 A, B, C 和 D 上引出. 对第二步的那七个被加数则在点 E, F, G, H, I, J, K 上.

2. 最初几步,p_i 中的数位个数是较少的. 再加上 a 的某些领前的数位不必形成,这就意味着: 对于一个 40 位的字长来说，可以通过把 **CSA** 树分割成字长较短的二部分,在第一个和第二个叠代步骤中均同时执行二个乘法,这类似于在 IBM**360/91** 的执行部件中的除法循环.

CSA 树的级数决定于字长;字长较长，所需要的级数也较多. 最后一步"a"的乘法(如果 n 是最后一步)即 $a_n = a_{n-1} \times (2 - a_{n-1})$ 可以不做,这是因为 $Q = p_n$ 与它无关. 利用 Wallace 树(图 8.6), 只要四次通过便能获得一个 40 位的倒数. 由于再编码近似乘数的重叠使用,至少前三次通过比一个完全的乘法要快.

8.6 带恢复的单元阵列除法器

带恢复的除法每个周期需要二次操作. 第一次操作从商的最高有效位开始，它常常是在被除数中对除数进行一次试凑性的减法. 如果这次减法不成功(即不够减), 便产生负的差值,于是得到一个商位"0",第二次操作要求再把除数加回去,以便恢复到旧的部分余数,这是因为已经减过头了. 试凑减法和恢复加法的效果相互抵销,从而引出一个正确的部分余数. 当前的周期通过将余数简单地左移一位来完成，并为下一个试凑周期作好准备. 另一方面,如果当前的试凑减法是成功的,那么选择商位"1",而差也就等于新的部分余数. 这种情况下,不需要恢复加法.

这些带条件的恢复操作可以用单元叠接阵列来实现. 阵列结构中所用的基本单元是一个具有四个输入和四个输出的可控减法器 (**CS**), 如图 8.7 所示. **CS** 单元的内部逻辑由下列方程式来说明

$$P = \bar{A}B + \bar{A}C + BC \tag{8.42}$$

$$S = AD + A\bar{B}\bar{C} + ABC + \bar{A}B\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} \tag{8.43}$$

这里 A, B 和 C 分别为余数、除数和借位-输入等输入二进位; P 为借位-输出信号, D 为同一行所有单元的操作控制信号. 下一个部分余数的数位 S 在逻辑上可以用下式来说明

$$S = \begin{cases} A \oplus B \oplus C, & \text{如果 } D = 0 \\ A, & \text{如果 } D = 1 \end{cases} \tag{8.44}$$

换句话说，当 $D = 0$ 时，CS 单元的性能象一个减法器，而若 $D = 1$，那么旧的余数就可以保留作为下一个余数．这里通过从向量 A（旧的部分余数）中减去向量 B（除数）来执行减法．图 8.8 指出了整个带恢复除法阵列的轮廓，这里被除数是一个 6 位的分数

$$N = .n_1n_2n_3n_4n_5n_6,$$

它由顶部一行和最右边的对角线上的垂直输入线来提供，除数是一个 3 位的分数

$$D = .d_1d_2d_3,$$

它沿对角线方向进入这个阵列．商是一个 3 位的分数 $Q = .q_1q_2q_3$，如果 $D > N$，它就在阵列左边产生．每一行决定一个商位，它或者是等于最后的部分余数的领前位，或者是等于这一行最左边的 CS 单元借位-输出的补码．对于真正的数值减法，到所有行去的初始借位-输入均被置为"0"．在除法中所需要的部分余数的左移可以用下列等效的操作来代替，即让余数保持固定，而将除数沿对角线右移．

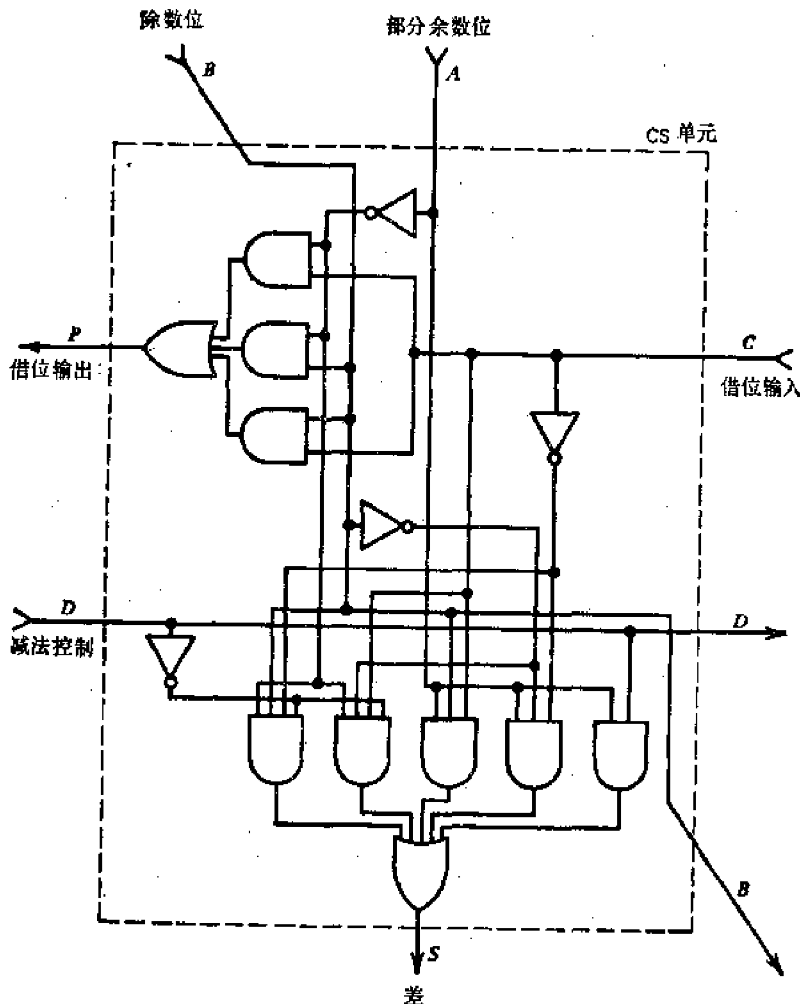


图 8.7 可控减法器（CS）单元的内部逻辑原理图

假如最后余数的领前一位数字是一个"1"，或者第 i 行的进位输出线上是一个"0"，即

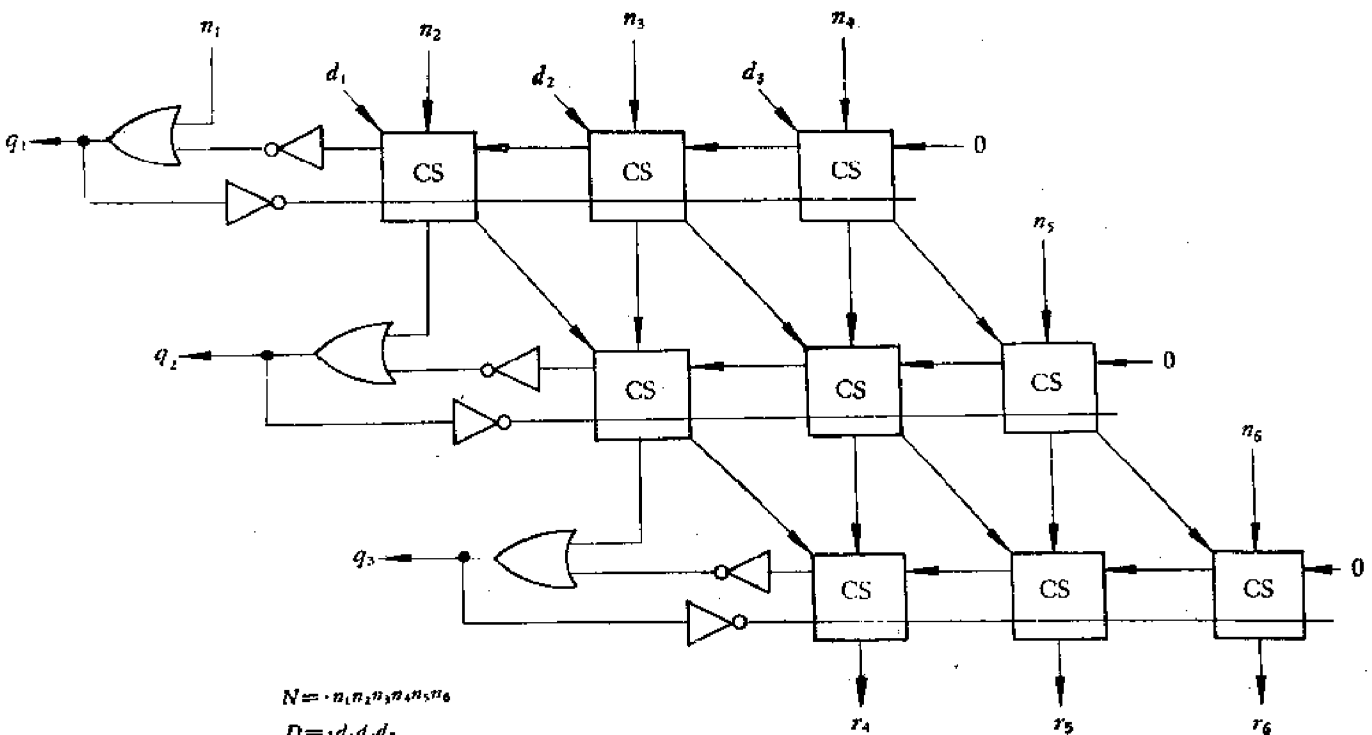


图 8.8　3 位除 3 位的带恢复阵列除法器的原理逻辑图

表示试凑减法的结果是正的，那么第 i 位商的数字产生一个数值“1”. 在这种情况下（$q_i=1$），试凑减法的差作为新的部分余数通过进入下一行. 另一方面，当 $q_i=0$（以上两种情况都不为“真”时），试凑减法可“旁路”，即让旧的部分余数通过进入下一行. 最后的余数，$R=0.000r_4r_5r_6$，将出现在底部的一行的输出，其中前面三个零相应于商长度（在图 8.8 中 $n=3$）.

方程式 8.42 和 8.43 指出了每个 **CS** 单元可用 3 级逻辑来实现，它具有 3Δ 的门延迟. 一般地说，一个二进制的单元阵列除法器接收一个 $2n$ 位的被除数和一个 n 位的除数，它产生一个 $2n$ 位的余数，其中前面 n 位是零. 我们把这样一个阵列称为 n 位除 n 位的除法器. n 位除 n 位的带恢复阵列除法器（在图 8.8 中，$n=3$）要求 n^2 个 **CS** 单元、$2n$ 个反相器以及三个或门. 由于逐行操作的时序特性，n 位除 n 位带恢复除法器应具有的除法执行时间在 $O(n^2)$ 的数量级. 这种阵列除法器与其它类型的除法阵列比较时的速度特性将在后面详细研究.

这里要提出一个重要的意见. 在带恢复的除法中，当这一行生成的商位具有数值“0”时，旧的部分余数即被保留到下一行. 这就使得有可能要在每个 **CS** 单元中安插多路转换器逻辑. 因此，在带恢复阵列的任一行上，并不执行实际的恢复加法. 反之，只有当旧的部分余数在一次试凑减法中被破坏，恢复步骤才成为必要. 这种使用多路转换器的存储移位技术已经用在标准的带恢复二进制除法器的设计中，如第 7.4 节所述. 这个特性使得带恢复除法能对它的“对手”不恢复除法相竞争. 如果没有存储多路转换逻辑，那么带恢复除法通常要比不恢复除法慢 33%.

8.7 不恢复的单元阵列除法器

在不恢复除法中并不需要供恢复用的加法. 不恢复阵列每一行所执行的操作究竟是加法还是减法，取决于前一行输出的符号与被除数的符号是否一致. 当出现不够减时，部分余数相对于被除数来说要改变符号. 这时应该产生一个商位“0”，除数首先沿对角线右移，然后加到下一行的部分余数上. 当部分余数不改变它的符号时，即产生商位“1”，下一行的操作应该是减法.

在说明完整的阵列结构之前，让我们先检验一下图 8.9 中给出的数值例子. 这将有助于指明在试凑法不成功的情况下，右移（除以 2）和加法操作所引起的“净”操作就是加除数的一半. 在带恢复的二进制除法中，通过加上整个除数，再将除数右移，然后减去除数的一半，这就得到正确的结果. 在表 8.2 中说明了这种对应关系.

不恢复的二进制除法器可以用一个由可控加法/减法（**CAS**）单元所组成的叠接阵列来实现，关于 **CSA** 单元已经在第 2.3 节作过描述. 一个 $(n+1)$ 位除 $(n+1)$ 位的不恢复除法阵列由图 8.10 所示的 $(n+1)^2$ 个 **CSA** 单元组成，其中两个操作数（除数与被除数）都是正的. 单元之间的互连是用 $n=3$ 的阵列来说明的. 用于输入操作数的端头分配和前面的相同. 最上面一行所执行的初始操作经常是减法. 因此最上面一行的控制线 P 固定置成“1”. 减法是用 2 的补码运算来实现的，这时右端各单元上的反馈线用作初始的进位输入. 必须指出，当这一行执行加法时（$P=0$），初始进位输入为零. 每一行最左边的单元的进位输出决定着商的数值. 把当前的商反馈到下一行，我们就能确定下一

表 8.2 带恢复的和不恢复的阵列除法的对应关系

	带恢复阵列除法	不恢复阵列除法
第一个周期	$-D$ $+D$(恢复)	$-D$
第二个周期	$-D/2$	$+D/2$
净操作	$-D/2$	$-D/2$

被除数 $N=(0.101001)_2=(41/64)_{10}$

除　数 $D=(0.111)_2=(7/8)_{10}$

TC: 2的补码. PR: 部分余数

	符号位 ↓	商的数位
被除数N	0.101001	
减 D(TC 形式)	1.001	
PR 为负	$1.110001<0$	$\Rightarrow q_0=0$
移位 PR	1.10001	
加D	0.111	
PR 为正	$0.01101>0$	$\Rightarrow q_1=1$
移位 PR	0.1101	
减D	1.001	
PR 为负	$1.1111<0$	$\Rightarrow q_2=0$
移位 PR	1.111	
加D	0.111	
	$0.110>0$	$\Rightarrow q_3=1$

商 $Q=q_0.q_1q_2q_3=(0.101)_2=(5/8)_{10}$

余数 $R=(0.00r_3r_4r_5r_6)_2=(0.000110)_2=(6/64)_{10}$

图 8.9 使用图 8.10 中给出的单元阵列除法器，执行普通不恢复的阵列除法的一个数值例子

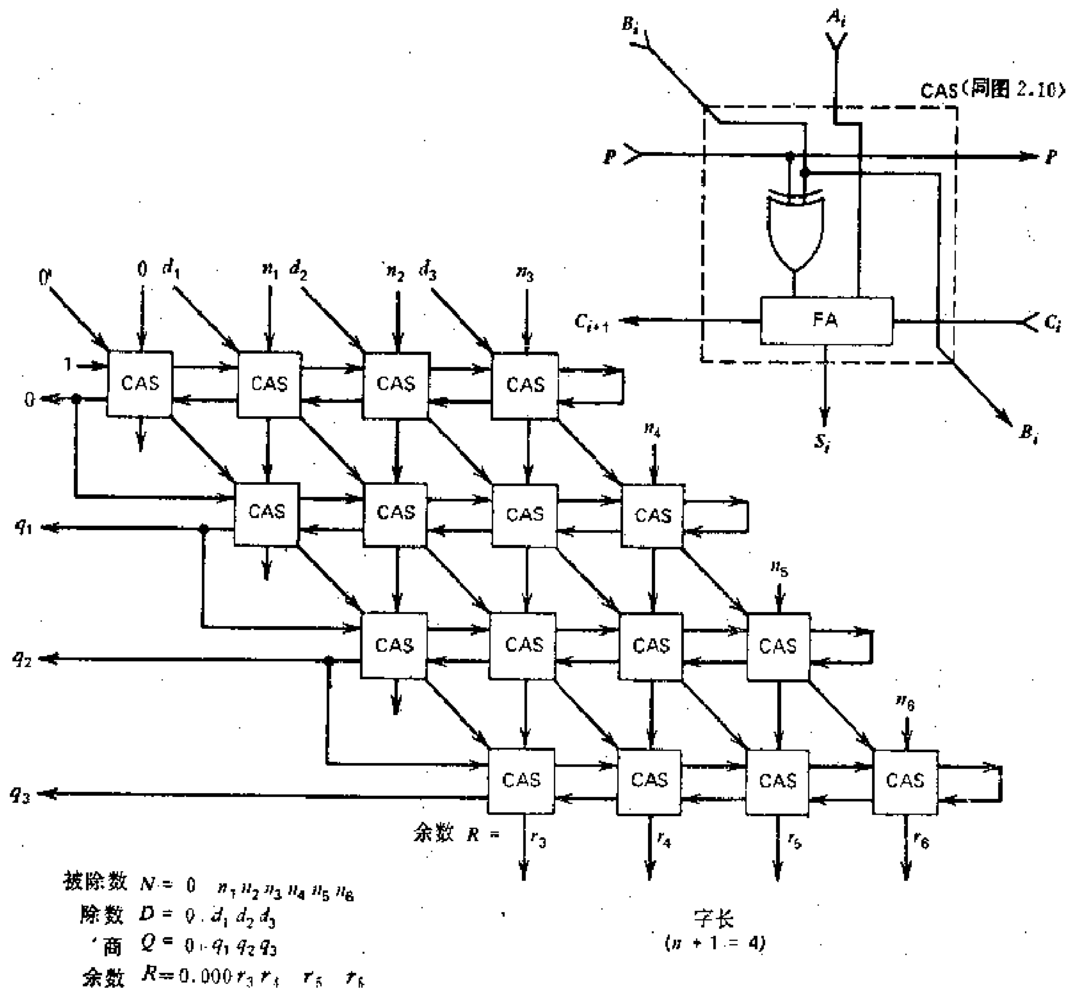


图 8.10 4 位除 4 位的不恢复阵列除法器的原理逻辑图 (Guild[11])

行的操作．进位输出信号指示出当前的部分余数的符号，因此，如前所述，它将决定下一行的操作．

使用上述不恢复除法阵列也需要 $O(n^2)$ 数量级的执行时间，这里由于沿着每一行有进位（或借位）传播，以及由于所有行在它们的进位线上都是串联连接的．为了用这个不恢复阵列估算除法执行时间，我们说明了 **CAS** 单元的实际内部电路的实现．**CAS** 单元的"和"以及"进位"的方程式可从方程式 2.3 修改成为下列形式

$$\begin{aligned} S_i = A_i \oplus (B_i \oplus P) \oplus C_{i-1} &= A_i B_i \bar{C}_{i-1} P + A_i \bar{B}_i \bar{C}_{i-1} \bar{P} \\ &+ \bar{A}_i B_i C_{i-1} P + A_i B_i C_{i-1} \bar{P} + A_i \bar{B}_i C_{i-1} P + \bar{A}_i \bar{B}_i \bar{C}_{i-1} P \end{aligned} \tag{8.45}$$

$$\begin{aligned} C_{i+1} = (A_i + C_{i-1})(B_i \oplus P) + A_i C_{i-1} &= A_i B_i \bar{P} + A_i \bar{B}_i P + B_i C_{i-1} \bar{P} \\ &+ \bar{B}_i C_{i-1} P + A_i C_{i-1} \end{aligned} \tag{8.46}$$

这二个方程式中，每一个都能用一个三级组合逻辑电路（包括反相器）来实现．因此，每一个基本的 **CAS** 单元的延迟为 3Δ 单位．在第 8.10 节中，我们将利用这个单元延迟来确定精确的除法时间．

8.8 进位-存储的单元阵列除法

到此为止我们已经学过的所有的除法阵列都需要在阵列的每一行中传播进位（借位）．这就使得总的除法时间成为与商的长度的平方成正比．这一节和下一节要描述利用进位存储和先行技术的方法，这是一种消除行波进位传播的可行的方法．这种改进方案可以用一个增量单元阵列来实现，它允许采用 MSI/LSI，对外部时序和控制逻辑的要求也最少．

在不恢复除法中，每一个相继的部分余数都能分解成二个向量的组合："和"向量与"进位"向量．这种分解使得进位的预测有可能采用普通的先行电路．因此，每一行左端的每个部分余数的符号位可以在最小的进位延迟时间内被确定．显然，进位的产生和传播的功能应该插入阵列的每个加法单元中去，从而使进位先行成为可能．这个方法所引起的总的除法时间几乎随商的长度线性地增加．在速度上的改进只要求适当增加硬件便能获得．

有三类逻辑单元可以用在这种阵列结构中．这些单元由图 8.11 所示的原理逻辑电路来描述．这些单元的具体实现可采用级数最少的电路．这些逻辑图并不意味着电路的复杂性最小．它们仅仅说明了那些单元的逻辑．在阵列中的 **A** 单元基本上是一些可控的 3 至 2 进位-存储全加器/全减器．上标 $i = 0, 1, \cdots, n$ 指示从上到下行的名称，下标 $j = 0, 1, 2, \cdots, n$ 指示每一行的数位的位置．最左边的位常常是符号位．

第 i 行的和的数位输出 S_j^i 与左移的进位输出 C_{j-1}^i 在逻辑上表示为

$$S_j^i = S_j^{i-1} \oplus C_j^{i-1} \oplus (D_j \oplus K^i) \tag{8.47}$$

$$C_{j-1}^i = (D_j \oplus K^i)(S_j^{i-1} + C_j^{i-1}) + S_j^{i-1} C_j^{i-1} \tag{8.48}$$

这里 S_j^{i-1} 和 C_j^{i-1} 为第 $(i-1)$ 行第 j 个"和"与"进位"输出的数位，D_j 为除数的第 j 位，K^i 为第 i 行的操作控制信号．在第 i 行上执行的操作决定于

$$K^i = \begin{cases} 0 & \text{对加法} \\ 1 & \text{对减法} \end{cases} \tag{8.49}$$

每个 **A** 单元包括一个与门和一个或门，它提供了进位生成和进位传播的功能，这些功能都是行的进位先行所必需的。

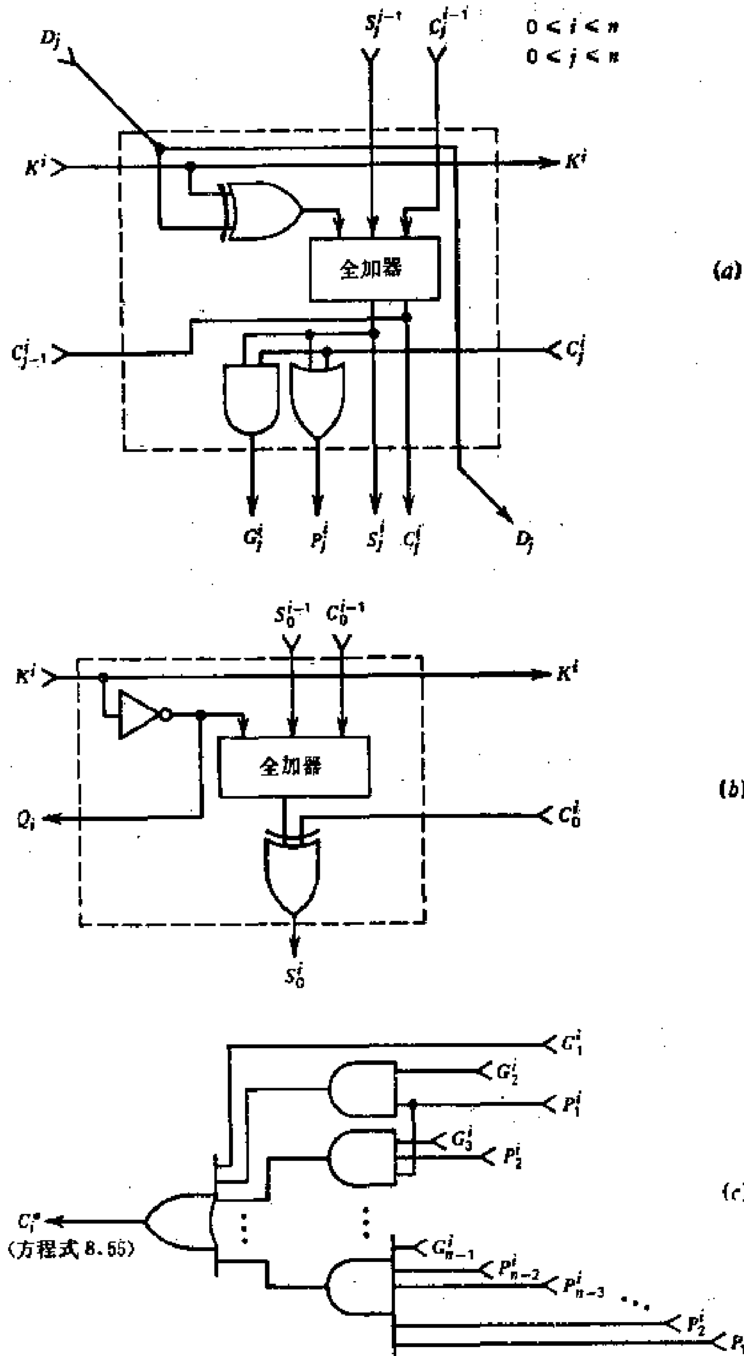


图 8.11 用来构成图 8.12 的除法阵列的三类基本逻辑单元.

(a) **A** 单元位于第 i 行第 j 个数位的位置上，(b) **S** 单元位于第 i 行的符号位的位置上，(c) **CLA** 单元在第 i 行上

$$G_j^i = C_j^i S_j^i \tag{8.50}$$

$$P_j^i = C_j^i + S_j^i \tag{8.51}$$

这里 C_j^i 来自同一行上右边最靠近的 **A** 单元的进位输出。

S 单元位于每一行的符号位的位置上．它产生一个伪符号 S_0^i，表示为

$$S_0^i = (S_0^{i-1} \oplus C_0^{i-1} \oplus \overline{K}^i) \oplus C_0^i \tag{8.52}$$

在第 i 行上，被求补的商的数位 $\overline{Q}_i$ 是由异门来产生的

$$\overline{Q}_i = S_0^i \oplus C_i^* \tag{8.53}$$

这里 C_i^* 为第 i 行产生的先行进位．信号 $\overline{Q}_i$ 用于控制下一行的加法-减法操作．对于 $i = 0, 1, \cdots, n-1$，我们有

$$K_{i+1} = Q_i \tag{8.54}$$

最上面的一行，$K_0 = 0$．

先行进位单元（**CLA**）的输出 C_i^* 可写成

$$C_i^* = G_1^i + P_1^i G_2^i + P_1^i P_2^i G_3^i + \cdots + P_1^i P_2^i \cdots P_{n-2}^i G_{n-1}^i \tag{8.55}$$

利用现有的技术，在 $n < 10$ 时，允许用一个具有 2Δ 延迟的电路来产生这个先行进位．

8.9 先行进位的单元阵列除法器

现在，我们准备利用所提到的各种积木式元件来描述执行高速除法的单元阵列结构．这个电路画在图 8.12 中．它用了一些 **A** 单元，**S** 单元和 **CLA** 单元，再加上若干异门和反相器．第一行经常依靠在控制线 K^0 上施加一个"1"来执行一次减法．每一行右端的初始进位输入被连接到上一行所产生的商的数位 Q_{i-1}．这就是说，减法是以 2 的补码加法的形式来完成的．

一个 $2n$ 位的被除数 $N = N_0.N_1N_2\cdots N_n$ 从顶部的输入线送人，这些输入线是指顶部一行以及最右边的对角线上那些单元的 S_j^{-1}．初始进位向量 C_j^{-1}（对 $j = 0, 1, \cdots, n$）应为零．被求补的除数通过对角线方向的输入线送人．阵列除法的操作用图 8.13 中的流程图来描述．每一步中的不恢复操作可用下列调整位置的公式来说明：

$$\begin{aligned} S^{i-1} &= S_0.S_1S_2\cdots S_{n-1}N_{n+i} = \text{旧的和向量} \\ C^{i-1} &= C_0.C_1C_2\cdots 0Q_{i-1} = \text{旧的进位向量} \\ \underline{\pm D} &\underline{= D_0.D_1D_2\cdots D_{n-1}D_n = \text{加/减除数}} \\ S^i &= S'_0.S'_1S'_2\cdots S'_{n-1}S'_n = \text{新的和向量} \\ C^i &= C'_0.\underbrace{C'_1C'_2\cdots C'_{n-1}}_{\text{去决定 CLA 的位 } C_i^*}0 = \text{新的进位向量} \end{aligned}$$

图 8.14 中给出一个用来说明这些操作步骤的数值例子．我们鼓励读者针对着图 8.12 中的阵列电路，去证实这个数值例子．图 8.12 所示的加速进位的逻辑具有一个单级先行的结构．目前只对字长 $n < 10$ 是可行的．对较长的字，每一行可能要求二级或多级先行进位．

总之，上述除法阵列可以用进位存储单元再加上每一行的先行进位逻辑来实现，它可能需要 $n(n+1)$ 个 **A** 单元，$n+1$ 个 **S** 单元，$n+1$ 个 **CLA** 电路，以及 $n+1$ 个异门，再加上一个反相器，这里 n 是指商的长度（包括符号位）．总的除法时间取决于所用先行进位的级数．然而，十分明显，每一行的时间延迟是一个与字长 n 无关的常数．利用这个阵列，总的除法时间应该是每一行延迟的 $(n+1)$ 倍．

$$\Delta_{CLA}(\text{阵列除法器}) = (n+1) \times \Delta_{row} + \Delta \tag{8.56}$$

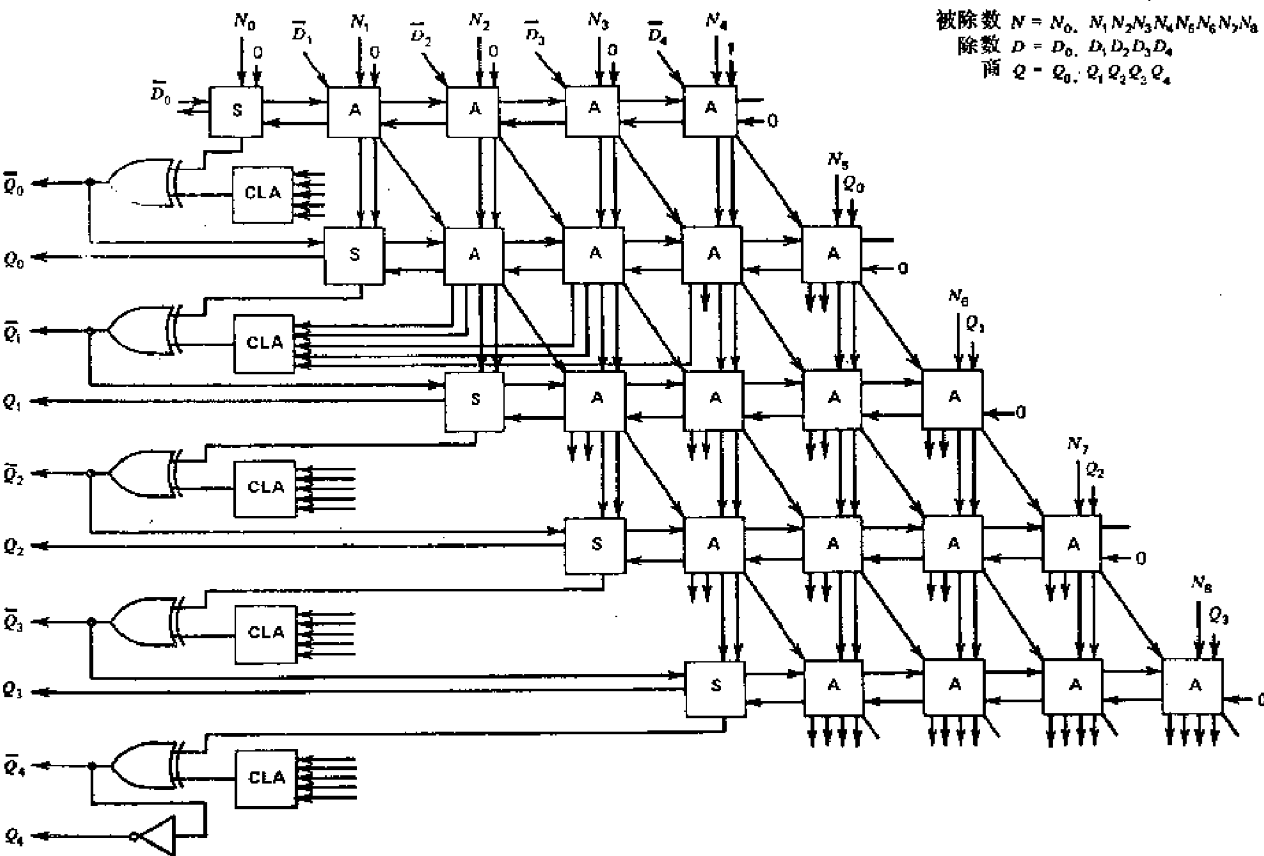


图 8.12　具有 8 位被除数与 4 位除数(以 2 的补码表示)的先行进位阵列除法器 (Cappa 与 Hamacher[3])

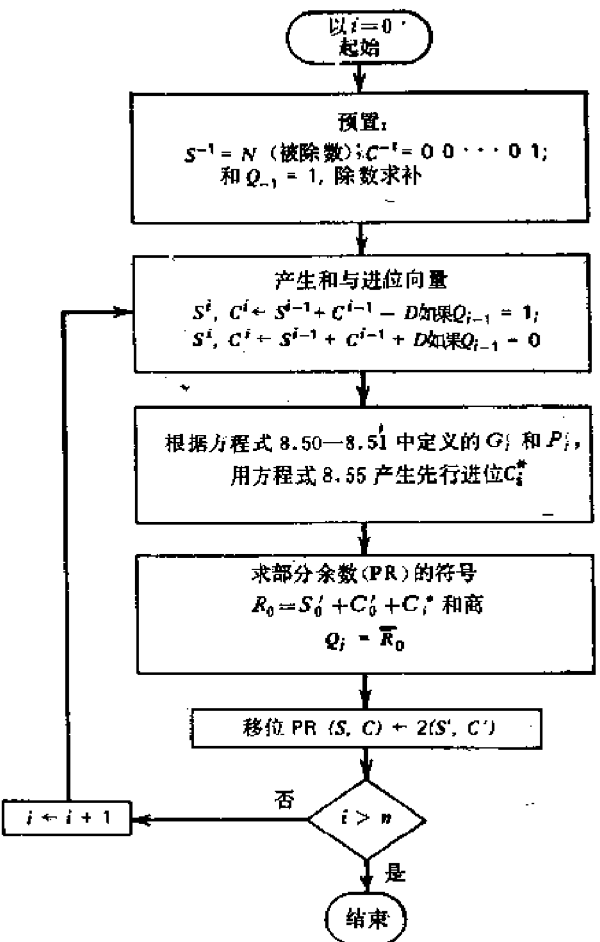


图 8.13　采用先行与进位-存储技术的不恢复除法的算法，用以描述图 8.12 所示的 Cappa-Hamacher 阵列除法器

式中 Δ_{row} 是下一节要分析的行延迟．Δ 是底部一行的反相器延迟，用于产生商的最后一个数位．

方程式 8.56 意味着总的除法时间是字长的线性函数 $O(n)$，它对上一节提出的非先行进位除法阵列的 $O(n^2)$ 延迟来说，有了很大的改进．严格地说，行延迟 Δ_{row} 应该按照 $O(\log_m n)$ 以对数的规律上升，这里 n 为字长，m 为所用门的最大扇人．目前的 TTL 逻辑一般具有 $m = 10$．$\log_m n$ 这个数字反映着先行的级数．因此，先行进位阵列除法器总的除法时间应具有 $O(n\log_m n)$ 的数量级．

对于实际应用来说，字长 n 通常较小，例如 $n \leqslant 100$．因此，对 $m = 10$ 来说，第二个因数 $\log_m n$ 就是 1 或 2，它可以作为常数来处理．方程式 8.56 指出的延迟对于实际设计是相当精确的．至少，这个延迟可看作分段线性增长，它在 $n = 10$ 处有一个间断点，超出这个间断点，则以更高的斜率直线上升．有关阵列除法方案的硬件和速度的详细分析将在下一节中讨论．

被除数 $= (0.10100100)_2 = \left(\frac{164}{256}\right)_{10}$　除数 $= (0.1110)_2 = \left(\frac{14}{16}\right)_{10}$

被除数	0.10100100			0.001	
强制进位	0.0001		先行	0←	
除数(1 的补码)	1.0001		真符号	1	$Q_2 = 0$
和向量	1.10100100	商位 ↓	移位和	1.10100	
进位向量移位	0.001		移位进位	0.0010	
先行进位位	0←		真除数	0.1110	
真符号	1	$Q_0 = 0$		1.01100	
移位和向量	1.0100100			1.010	
移位进位向量	0.0100		先行	0←	
真除数	0.1110		真符号	0	$Q_3 = 1$
和向量	1.1110100		移位和	1.0100	
进位向量	0.100		移位进位加 1	0.0101	
先行	1←		除数反码	1.0001	
真符号	0	$Q_1 = 1$		0.0000	
移位和	1.110100			0.101	
移位进位加 1	1.001		先行	0←	
除数反码	1.0001		真符号	0	$Q_4 = 1$
	1.110100				

商 $= Q_0 . Q_1 Q_2 Q_3 Q_4 = (0.1011)_2 = \left(\frac{11}{16}\right)_{10}$

图 8.14　具有进位先行和进位-存储的 Cappa-Hamacher 不恢复阵列除法的例子

8.10　各种单元阵列除法器的评价

前几节已经描述了三类除法阵列，这一节将对它们的操作速度和硬件要求加以比较．速度用基本门的延迟级数来度量，硬件要求用阵列结构中所需要的逻辑门的数量来评价．

用来构成带恢复除法阵列的 **CS** 单元要求用 10 个门来实现，每个单元具有 3Δ 的延迟．在计算门的个数时只涉及必需的几类基本门(与，或，非)．同理，用来构成不恢复阵列的 **CAS** 单元，如方程式 8.45 和 8.46 所指定的那样，每个单元需要 12 个门，并且每个单元也具有 3Δ 延迟．

在一个 n 位除 n 位的带恢复阵列中，最坏情况下信号的延迟就是阵列底部一行生成商的最低有效位 q_n 所需要的时间．在带恢复的阵列中，直到最左边的借位输出线上出现一个稳定的信号之前，每一行的和输出（新的部分余数）不能认为是稳定的．这就是说，每一行本身必须取 $n \times 3\Delta$ 时间延迟来产生一个正确的、下一行要用到的部分余数．因为相继的部分余数应以串行的方式一行一行地生成，所以在图 8.8 底部一行上生成最后一位商所需要的总时间是

$$\Delta_{restoring} = (3n^2 + 1)\Delta \tag{8.57}$$

这里 1Δ 是由最后一行反相器引起的．

对一个 $2n$ 位除以 n 位的不恢复阵列除法器，如果顶部一行需要用来决定商的符号，并以此作为一个附加的要求，那么我们可以获得类似的结果

$$\Delta_{nonrestoring} = 3(n + 1)^2\Delta \tag{8.58}$$

上面的分析指明了对于较大的 n，带恢复的和不恢复的除法阵列提供大致相同的速度．对短字长来说，带恢复的阵列比较快．

为了评价一个 $2n$ 位除以 n 位的先行进位阵列除法器的硬件要求，我们必须确定在基本单元中门的个数．代替图 8.11 中给出的逻辑图，我们用具有最小延迟的 3 级电路来估算增量阵列除法器所用到的 **A** 单元和 **S** 单元中门的个数和延迟．通过将 **A** 单元和 **S** 单元的方程式加以展开，便能得到这些电路．为清楚起见，所有下标和上标均取消．代替的是，单元的输出都加上一撇，而输入不加．展开以后，方程式 8.47 成为

$$\begin{aligned} S' = \bar{S}\bar{C}\bar{D}\bar{K} + \bar{S}\bar{C}D\bar{K} + \bar{S}C\bar{D}\bar{K} + \bar{S}CDK + S\bar{C}\bar{D}\bar{K} \\ + S\bar{C}DK + SCD\bar{K} + SC\bar{D}K \end{aligned} \tag{8.59}$$

方程式 8.48 成为

$$C' = \bar{D}KS + \bar{D}KC + D\bar{K}S + D\bar{K}C + SC \tag{8.60}$$

方程式 8.50 和 8.51 成为

$$G = C \wedge \text{（方程式 8.59 中的} S'\text{）} \tag{8.61}$$

$$P = C \vee \text{（方程式 8.59 中的} S'\text{）} \tag{8.62}$$

注意方程式 8.61 和 8.62 中的 C 来自同一行右边的 **A** 单元的进位输出，而方程式 8.59 和 8.60 中的 C 则来自最接近的上面的一行．方程式 8.52 成为

$$\begin{aligned} S_0^i = \bar{S}C\bar{K}\bar{C}^i + \bar{S}CKC^i + S\bar{C}\bar{K}\bar{C}^i + S\bar{C}KC^i + \bar{S}\bar{C}\bar{K}C^i + \bar{S}\bar{C}K\bar{C}^i \\ + SC\bar{K}C^i + SCK\bar{C}^i \end{aligned} \tag{8.63}$$

由这些最小级数的电路，可知每个 **A** 单元将包含 17 个门．对于 S'（方程式 8.47 中的 S_j^i）和 C'（方程式 8.48 中的 C_{j-1}^i）的延迟，每一个为 3Δ．对于 G 和 P 功能的延迟，每一个为 $3\Delta + \Delta = 4\Delta$．**S** 单元需要 9 个门，它的时间延迟为 3Δ．由于它用 3Δ 延迟来产生 C^i，进位输入信号从同一行最左边的 **A** 单元的进位输出进入到 **S** 单元．因此，当输入信号从最靠近的上一行的输出到来之后，总共用了 $3\Delta + 3\Delta = 6\Delta$ 来产生每一行的伪符号 S_0^i．

CLA 单元的门的成本是随字长而改变的．对 $n < 10$，一级 **CLA** 至多需要 $n - 1$ 个门．对 $10 \leqslant n \leqslant 64$，二级 **CLA** 大致需要 $n + \sqrt{n}$ 个门．把这些门的个数加在一起，我们便得到 $2n$ 位除以 n 位的具有二级 **CLA** 的除法阵列所需要的门的数量．

$$\text{门的数量} = 17n(n + 1) + (n + 1) \times (n + \sqrt{n} + 9 + 1) + 1$$

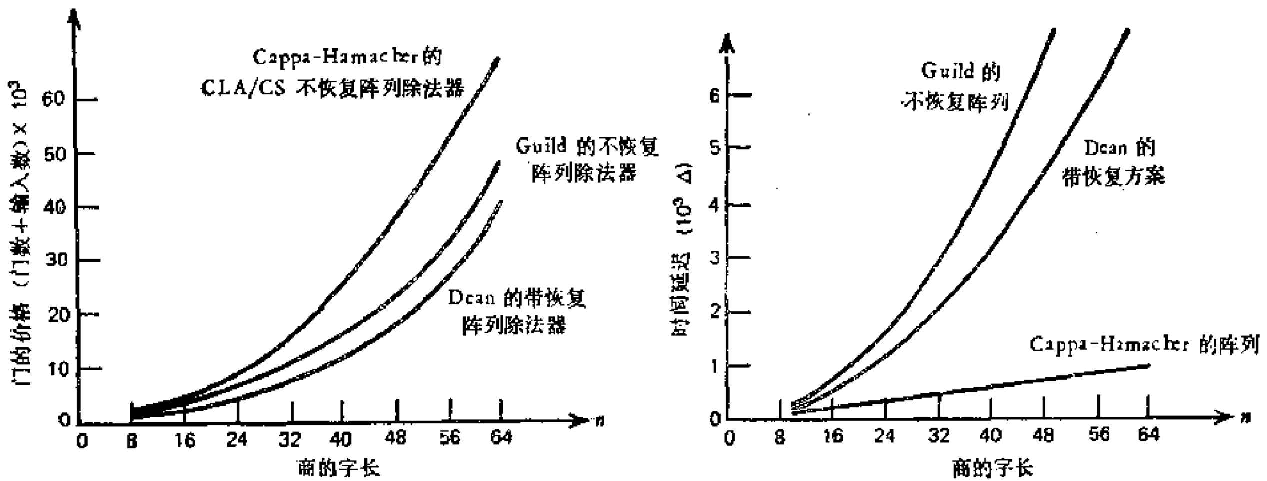


图 8.15　三类叠接阵列除法器的成本与延迟随商的字长而变化（Cappa[33]）

$$= (n+1)(18n+\sqrt{n}+10)+1 \tag{8.64}$$

方程式 8.56 中提到的每一行时间延迟 Δ_{row} 现在可以决定如下：Δ_{row} 决定于生成每个商位 $\overline{Q}_i$ 的时间，它等于几个电路延迟的总和，3Δ 来自异门的输出，加上 6Δ 来自产生伪符号的逻辑(一个 **A** 单元加一个 **S** 单元)，或者先行进位逻辑，对单级先行来说，后者等于 4Δ + 2Δ = 6Δ. 因此，我们有

$$\Delta_{row}(\text{单级 CLA}) = 3\Delta + 6\Delta = 9\Delta \tag{8.65}$$

对二级先行进位，第二项延迟将是 6Δ + 2Δ = 8Δ. 因此

$$\Delta_{row}(\text{二级 CLA}) = 3\Delta + 8\Delta = 11\Delta \tag{8.66}$$

将上面得到的行延迟代入方程式 8.56，我们得到增量 **CLA** 阵列除法器的总的除法时间

$$\Delta_{CLA}(\text{阵列除法器}) = \begin{cases}(9n+10)\Delta & \text{对单级 CLA;}\\ (11n+12)\Delta & \text{对二级CLA}\end{cases} \tag{8.67}$$

方程式 8.67 对于用在实际设计问题中是完全足够的．严格地说，对一个具有 $k=\lceil \log_m n \rceil$ 级先行逻辑的除法阵列，我们可以得出下列估算除法时间的公式

$$\Delta_{k级CLA}(\text{阵列除法器}) = (n+1)\times[9\Delta+(\log_m n-1)\times 2\Delta]+\Delta$$
$$= [2(n+1)\times \log_m n + 7n + 8]\Delta \tag{8.68}$$

式中 m 是该电路中许可的最大门扇入数．

表 8.3 归纳了与三类叠接阵列除法器中的每一类相联系的总门数和除法时间．在图

表 8.3　三类阵列除法器的门的数量和时间延迟．商的字长为 n，单位门的延迟为 Δ.

阵列除法器类型	门的数量	除法时间[1]
带恢复阵列 (Dean)	$14n^2$	$(3n^2-1)\Delta$
不恢复阵列 (Guild)	$17(n+1)^2$	$3(n+1)^2\Delta$
具有二级 CLA 和进位存储的不恢复阵列 (Cappa-Hamacher)	$(n+1)\times(18n+\sqrt{n}+10)+1$	$(11n+12)\Delta$
具有一级 CLA 和进位存储的不恢复阵列 (Cappa-Hamacher)	$18n^2+28n+11$	$(9n+10)\Delta$

1) 字长 n 不包括涉及符号位的附加要求，即对不恢复阵列的实际字长为 $n+1$.

8.15 中给出了速度和门的成本的曲线图. 从这些曲线中我们可以作出下列结论: 先行进位阵列除法器的成本平均比不恢复除法阵列高 50%,比带恢复除法阵列高 70%. 平均地说,**CLA** 阵列除法器要比另外二个不用先行进位逻辑的阵列除法快五倍. 相应于带恢复和不恢复阵列的速度曲线几乎互相重合. 在这三类中,带恢复阵列除法器所用的硬设备最少.

8.11 参考文献注释

用乘法收敛的除法最初是由 Goldschmidt[10] 提出的. 该方案曾经用在 **IBM360/91** 的浮点执行部件的设计中[1]. 求倒数的除法是 Wheeler 在[19]中最早介绍的. 倒数收敛的理论是在 Wallace[18] 和 Ferrari[6] 的一系列论文中相继提出的. Stefanelli[16]曾经提出一种倒数除法方案,它采用了具有带符号数字代码的叠接阵列. Socemeantu[15]也研究过这种新方法. 许多作者提出了叠接单元阵列除法. Dean[4] 提出了带恢复的阵列除法器. Majithia[13] 和 Guild[11] 提出了不恢复的除法阵列. Gardiner 等[8]对带恢复的和不恢复的单元阵列除法器作了比较.进位存储的除法和先行进位阵列除法器是 Cappa 和 Hamacher[3] 提出的. Cappa 的论文[2]提出了高速阵列运算的一种综合处理,其中使用了先行进位和进位存储的方法. 在 Deverell[5], Gex[9] 和 Hamacher 等[12] 的最近著作中也提出了一些用于快速乘法和除法的叠接阵列. 一种可能的发展趋向是用大规模的阵列逻辑来实现带符号数字的运算.

参 考 文 献

[1] Anderson, S. F. et al., "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal*, January 1967, pp. 34—53.

[2] Cappa, M., "Cellular Iterative Arrays for Multiplication and Division," *M. S. Thesis*, Dept. of Elec. Eng., Univ. of Toronto, Canada, October 1971.

[3] Cappa, M.. and Hamacher, V. C., "An Augmented Iterative Array for High-Speed Binary Division," *IEEE Trans. on Computers*, Vol. C-22, February 1973, pp. 172—175.

[4] Dean, K. J., "Binary Division Using a Data Dependent Iterative Arrays," *Electronics Letters*, Vol. 4. July 1968, pp. 283—284.

[5] Deverell, J., "The Design of Cellular Arrays for Arithmetic," *The Radio and Electronic Engineer*, Vol. 44. No. 1. January 1974. pp. 21—26.

[6] Ferrari, D., "A Division Method Using a Parallel Multiplier," *IEEE Trans. Comput.*, Vol. EC-16, April 1967, pp. 224—226.

[7] Flynn, M. J, "On Division by Functional Iteration," *IEEE Trans. Comput.*, Vol. C-19, August 1970 pp. 702—706.

[8] Gardiner, A. B. and Hont, J., "Comparison of Restoring and Nonrestoring Cellular Array Dividers," *Electronics Letters*, Vol. 7, April 1971, pp. 172—173.

[9] Gex, A., "Multiplier-Divider Cellular Array." *Elec. Letters*, Vol. 7, July 1971, pp. 442—444.

[10] Goldschmidt, R. Z., "Applications of Division by Convergence," *M. S. Thesis*, M. I. T. Cambridge, Mass., June 1964.

[11] Guild, H. H., "Some Cellular Logic Arrays for Nonrestoring Binary Division," *The Radio and Elec. Engr.*, Vol. 39, June 1970, pp. 345—348.

[12] Hamacher, V. C. and Gavilan, J., "High-Speed Multiplier/Divider Iterative Arrays," *Proc. of 1973 Sagamore Computer Conf. on Parallel Processing*. 1973, pp. 91—100.

[13] Majithia, J. C., "Nonrestoring Binary Division Using a Cellular Array," *Electronics Letters*, Vol 6, May, 1970, pp. 303—304.

[14] Robertson, J. E., "Theory of Computer Arithmetic Employed in the Design of New Computer at the University of Illinois," *Tech. Rept.*, No. 319, Dept. of Computer Science, University of Ellinois, Urbana, 1960.

[15] Socemeantu, A., "Cellular Logic Array for Redundant Binary Divisions," *Proc. IEE* (London), Vol. 119, No. 10, October 1972, pp. 1452—1456.

[16] Stefanelli, R., "A Suggestion for High-Speed Parallel Binary Divider," *IEEE Trans. Comp.*, Vol. C-21, January 1972. pp. 42—55.

[17] Svobada, A., "An Algorithm for Division," *Inf. Proc. Machines*, No. 9, Prague, Czechoslovakia, 1963, pp. 25—34.

[18] Wallace, C. S., "A Suggestion for a Fast Multiplier," *IEEE Trans. Comp.*, Vol. EC-13, February 1964, pp. 14—17.

[19] Wilkes, M. V., Wheeler, D. J., and Gills, S., *Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, Reading, Mass., 1951.

习　　题

题 8.1　在第 8.2 节中描述的采用乘法的除法算法按二次幂收敛于所要求的商，那里用了收敛因数的集合

$$\{\delta^{2^i} | i = 0, 1, 2, \cdots, n\}$$

其中 $\delta = 1 - D$，D 为给定的除数.

现在将这个二次收敛除法推广到三次收敛的除法方案，这里利用下列相继的收敛因数

$$\{\delta^{3^i} | i = 0, 1, 2, \cdots, k\}$$

其中 k 为所需的叠代次数. 试证明：$k<n$，并且解释收敛速率的改进是 p 的函数，p 是操作数的分数的长度.

题 8.2　对问题 8.1 所拟定的三次收敛除法的方法，试阐明：为了产生各个相继的乘数所必需的运算操作. 说明有三个运算操作与每个乘数的形成相联系，即每次叠代可能需要较长的周期时间. 指明在二次和三次收敛的除法系统中，收敛速度(叠代次数的倒数)和叠代周期之间的折衷方法.

题 8.3　试证明：利用 Ferrari 的分段逼近方法（图 8.5）对第 h 个间隔的 $p_0(h)$ 的初始近似的最佳选择就是相应于它中点的数的倒数，即方程式 8.33.

题 8.4　试对图 8.6 中所示的 Wallace 树的级数加以扩充，使它只要几次叠代就能产生一个 72 位的倒数. 此外，要求确定产生 72 位倒数所需要的最小叠代次数.

题 8.5　试用可控减法器(CS)单元设计一个 16 位除 16 位的带恢复阵列除法器，用它来实现 31 位不带符号的分数除以 16 位不带符号的分数的除法. 然后再用被除数

$$N = 0.0000111001000110111111110101101010$$

和除数

$$D = 0.1101010110101111$$

来检查你的设计是否正确. 估算出使用该阵列的总的除法时间，并以单位门延迟的倍数来表示.

题 8.6　试用可控加法/减法单元（CAS）设计一个 16 位除 16 位的不恢复阵列除法器，用它来实现 32 位 2 的补码数除以 16 位 2 的补码数的除法. 并用 2 的补码形式表示的被除数

$N = 0.0110111101111010100110110011110$ 和除数 $D = 0.101100100111011$

来检查你的设计是否正确. 估算出总的除法时间，以单位门延迟来表示.

题 8.7　设计一个 16 位除 16 位的先行进位阵列除法器，用它来实现 32 位被除数和 16 位除数的 2 的补码运算. 并用问题 8.6 中给出的同一对 2 的补码数 N 和 D 来检查你的设计. 估算出你的设计的总除法时间，这里假定阵列结构的每一行用二级先行进位.

题 8.8　从以下二个方面来对三类 32 位除 32 位的阵列除法器进行比较，这三类除法器类似于上述问题中提到的那些设计.

(a) 总的除法时间，这里假定 $\Delta = 4$ ns.

(b) 总的门数.

用除法时间和门的数量的乘积的倒数来评价这些阵列除法器，以便确定它们的"成本/效率"。在你的评价中，这三类 32 位除 32 位的阵列设计中，哪一类的成本/效率最高?

第九章 规格化的浮点运算处理器

9.1 浮点运算的基本理由

大多数早期的计算机都采用定点运算，它们基本上能处理商业应用方面以精确形式表示的“小”整数．对于科学计算来说，为了减少数字的位数使其数量便于处理，我们经常需要对这些数进行舍入．定点运算在处理科学和工程计算时则存在着某些困难．这些问题主要是由于机器中所表示的数的范围、精度以及有效位等引起的．

定点数最常用的范围是从－1到＋1的单位区间．当数的范围在计算过程中变成很大或很小时，程序员要关心所有的中间数据的小数点，这显然是很不方便的．超出范围的数通常利用软件、固件或硬件的方法通过选取适当的比例来加以处理．显而易见，科学计算中用到的数当然并不一定都落在单位区间内．在大多数场合，给定的数必须按比例增大或缩小，以便正确地符合于单位区间，在计算结束时，再把结果变回到用户的数域．如果没有这些变换，那么定点硬件就会产生毫无意义的结果．

比例问题包括选择适当的比例因子，以及对某些复杂情况建立适当的比例环，在这个环中的比例因子可以随情况的变更而加以修改．任何一个基数为 r 的 n 位定点数，如果绝对值小于 r^k，那么最大误差就是 r^{k-n}．常数 r^k 是由整个数的集合所共享的公共比例因子．$k=0$ 和 $k=n$ 这二个特例分别相应于定点分数和定点整数．

一个 n 位定点分数的固有精度要受到最大误差 r^{-n} 的限制．为了提高精度，提出了多精度的定点运算．每个定点数使用几个字，这意味着程序编制会更复杂，而且数据和指令所消耗的存储空间也会更多．尽管如此，多精度的定点运算不需要很多误差分析就能提供精确的结果．

对于复杂的计算问题，这些比例和精度的扩展步骤会涉及广泛的数学分析以及附带的计算，以便跟踪比例因子或者控制可变的字长．通常，整个数的集合的公共比例因子就是所用的最大比例因子．比例的引入会引起有效位丢失的问题．例如，在一个公共比例因子 r^p 和一个数的数量级 r^t 之间有差别时，如果 $t<p$，在定点数中就会产生 $p-t$ 个领前的零，因而最多只剩下 $n-p+t$ 个有效位，而不是实际提供的 n 个有效位．在一连串计算中，有效精度的不断丢失就会引起定点硬件不能处理的特殊情况，这种情况只有通过程序员干预才能解决．

为了克服上述与定点运算相联系的一些困难，早在 1940 年便提出了浮点运算．尽管浮点运算具有较复杂的舍入线路，并且硬件的需要量也可能加倍或加二倍，但它仍然为高速科学计算所普遍接受．浮点运算有二类：非规格化的和规格化的．规格化的浮点处理器只能对规格化的浮点数进行操作，并且要对所有的中间和最后结果必须施加算后规格化步骤．非规格化的浮点运算涉及的浮点操作不必要求规格化的操作数．规格化操作所具有的优点是：程序上的方便、唯一的浮点数表示式以及在每一级计算的尾数中能获得最大的有效数字．浮点运算是致力于研究自动定比例（用来解决与定点运算相联系的有

限的范围和严格的精度问题）过程的结果．大多数近代的通用数字计算机经常同时备有定点和浮点二种运算处理器．

9.2 基数的选择和浮点的特殊性

大多数计算机设计成带有二进制浮点运算，如象从 Digital Equipment 公司的 PDP-11 系列，一直到 IBM 360 系统的出现（后者选择基数 16 来做浮点运算）．其他的机器如象 ILLIAC II 采用基数 2，Burroughs 的机器采用基数 8，ILLIAC III 则用基数 256 来做乘法和除法，但存储数时用基数 16．Sweeney[14] 曾经指出，随着基数的增加，对调准和规格化移位的需要就会减少．以二进制和八进制为基础的浮点机用于科学计算时，在有关它们的数值特性方面很少有弊病．然而十六进制的机器，当用户把它们用到科学问题时，就会带来许多不方便的注释．这就促使将浮点数系统的数值特性当作基数值的函数来加以研究．这些研究将涉及表示方法的误差分析，舍入的模拟，在数学函数计算中确定相对误差的严格边界，以及相应的机器设计考虑等．

对二进制、八进制、十六进制以及其它浮点系统的优缺点的论证和讨论，主要集中在一些数值特性上，如机器表示的阶的范围，数的密度以及最大相对误差．我们用三个主要参数 (r, p, q) 来描述浮点表示法的特征，这里 r 是基数，p 是尾数的长度，$q+1$ 是阶的长度．注意 $p+q+2=n$ 即等于包括符号位在内的字长．我们将留到第十章再去分析浮点运算的误差与基数选择的关系．

我们在对合法的浮点数的正常浮点运算操作下定义之前，先考虑一下异常的情况．用浮点硬件可以产生单一的结果；这就要对合法的浮点数执行合法的操作．我们用符号 $+\infty$ 和 $-\infty$ 表示正的和负的准无穷大，相当于浮点数的阶超过最大允许的正值．二个无穷小量 $+\varepsilon$ 和 $-\varepsilon$ 类似地定义为浮点数的阶超出最大的负值．下标“u”和“n”用来区别非规格化的无穷小 $\pm\varepsilon_u$ 和规格化的无穷小 $\pm\varepsilon_n$．

令 p 为浮点数尾数字段中的位数，$q+1$ 为阶的位数．对一台基数为 $r=2^k$ 的浮点机（这里 $k=\log_2 r$ 是指要表示每一个基数为 r 的数字所需的二进位位数），则上述无穷大和无穷小应该满足以下特性．注意：二进制中最小的非零分数 2^{-p} 等于最小的基数 r 的非零分数 $r^{-p/k}$．

$$+\infty > (1-2^{-p}) \times r^{2^q-1};\ -\infty < -(1-2^{-p}) \times r^{2^q-1} \tag{9.1}$$

$$0 < +\varepsilon_u < 2^{-p} \times r^{-(2^q-1)};\ -2^{-p} \times r^{-(2^q-1)} < -\varepsilon_u < 0 \tag{9.2}$$

$$0 < +\varepsilon_n < r^{-1} \times r^{-(2^q-1)};\ -r^{-1} \times r^{-(2^q-1)} < -\varepsilon_n < 0 \tag{9.3}$$

对于规格化的和非规格化的二种浮点运算系统，上述范围均已表示在图 9.1 中．

现在，我们准备来定义数字计算机执行浮点指令期间可能出现的一些特有的结果．

阶的上溢 指的是所得到的数的阶超出了方程式 9.1 所指出的上限或下限．换句话说，$+\infty$ 和 $-\infty$ 二者均被认为是上溢出的条件．

阶的下溢 指的是所得到的数的阶超出了最小允许值，进入到下列区间 $(-\varepsilon, 0_-)$ 和 $(0_+, +\varepsilon)$，ε 值如方程式 9.2 和 9.3 所示．应该注意，在实轴原点上的真正的零值是一个例外．也就是说，零并不进入到无穷小的区间 $(-\varepsilon, 0_-)$ 和 $(0_+, +\varepsilon)$．每当阶发生上溢或下溢时，将由浮点硬件发出相应的指示信号．有时候，上溢/下溢信号可以旁路而不致

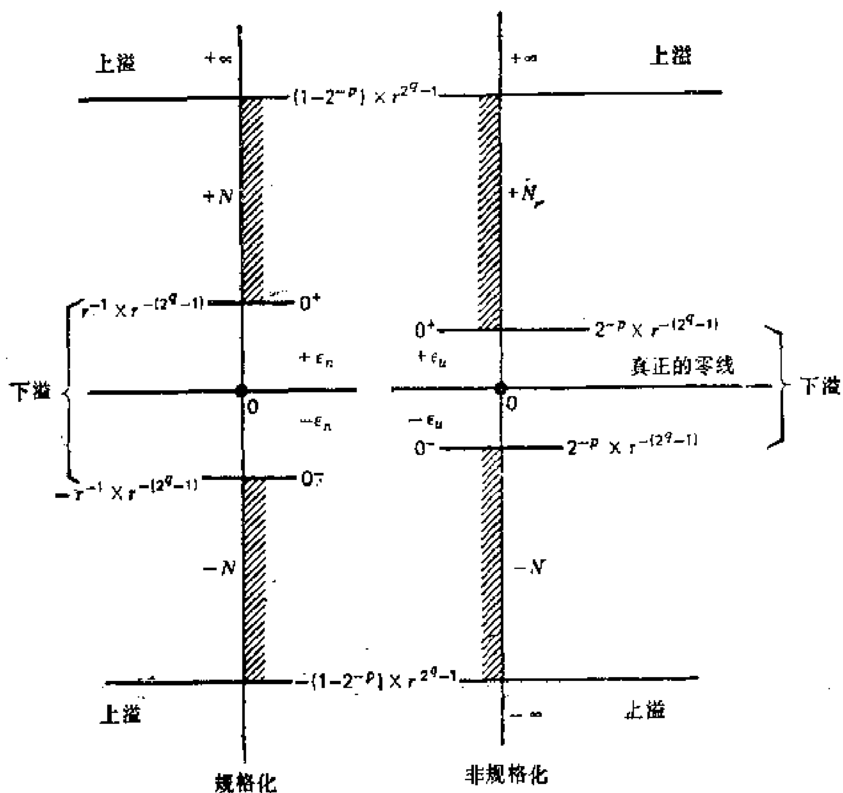


图 9.1 机器可表示的浮点数的范围,浮点奇异数,以及上溢/下溢条件

中断计算;也有时候,程序员可能需要加以干预,以便保存有意义的计算.

与定点运算不同,在浮点表示法中的零可以具有多个意义.零的更为广泛的含义将在下面给出.

数量级零 任何一个尾数为零的浮点数,即 $(m, e)=(0, e)$,均被称作数量级零(OMZ),这里阶 e 假定是阶的字段中所能表示的任意合法值. 这个 OMZ 可以是一个数减去它本身所得到的结果.

$$(m, e)-(m, e)=(0, e) \tag{9.4}$$

由 $(0, e)$ 来表示的数值意味着这个不确定的数有很宽的范围,即满足下列不等式

$$-r^{e-p}<(0, e)<r^{e-p} \tag{9.5}$$

这里 p 为尾数的数位个数,r 为蕴含的基数.

在特殊情况下,一个具有 $e=0$ 的 OMZ$(0, e)$ 被表示为 $(0, 0)$,我们称之为**真零**.真零并不是一个规格化数,也不是无穷小集合中的一员.

合法的浮点操作数(范围以内的数)在图 9.1 中表示为 $+N$ 或 $-N$. 可以看出

$$+\varepsilon<+N<+\infty \qquad -\infty<-N<-\varepsilon \tag{9.6}$$

真零相当于正数和负数之间的分界线. OMZ 则不能表示在实轴上. 表 9.1 列出了对一个合法操作数 $\pm N$ 和一个无穷大 $\pm\infty$ 或一个无穷小 $\pm\varepsilon$ 的所有可能的操作.这些操作所涉及的二个奇异数或不定数 OMZ 也在这个表格中说明了. 用 OMZ 除是被禁止的,这

表 9.1 涉及无穷大、无穷小、OMZ 和正规浮点数的浮点运算操作

加法+和减法−	备　注
$\infty\pm N=\infty$; $N\pm\varepsilon=N$; $N-\infty=-\infty$ $\varepsilon-N=-N$; $\infty+\infty=\infty$; $\infty\pm\varepsilon=\infty$ $\varepsilon\pm\varepsilon=\varepsilon$; $\varepsilon-\infty=-\infty$; $\infty-\infty=\infty$ $\varepsilon-\varepsilon=\varepsilon$; $N\pm Z=N$; $Z-N=-N$ $z_1\pm z_2=z_3$	N:正规浮点数　∞: 无穷大　ε: 无穷小 $z_i=(0, e_i)$: 一个 OMZ
乘法*	
$\infty*N=\infty$; $N*\varepsilon=\varepsilon$, $\infty*\infty=\infty$ $\varepsilon*\varepsilon=\varepsilon$; $\infty*\varepsilon=\infty$(或$\infty*\varepsilon=\varepsilon$) $z_1*z_2=z_3$; $N*z_1=z_2$	大多数机器选用 $\infty*\varepsilon=\infty$ 或 $\varepsilon/\varepsilon=\infty$ 来代替 $\infty*\varepsilon=\varepsilon$ 或 $\varepsilon/\varepsilon=\varepsilon$，以便强调要报警的溢出情况．遇到 N/z 或 $N/0$ 将使零除数标志置位
除法	
$\infty/N=\infty$; $\varepsilon/N=\varepsilon$; $N/\infty=\varepsilon$; $N/\varepsilon=\infty$ $\infty/\varepsilon=\infty$; $\varepsilon/\infty=\varepsilon$; $\infty/\infty=\infty$; $\varepsilon/\varepsilon=\infty$ (或 $\varepsilon/\varepsilon=\varepsilon$); $z_1/N=z_2$; $N/z=N/0$ （N/z 与 $N/0$）两者均被抑制	

将接通一个零除数指示器.

表 9.1 中定义的那些操作只是作为一种参考性的选择提出来的．实际上，系统的设计者可以定义他自己的一组不确定操作，以适应特殊的应用要求．一般的规则是：用这样的方法来处理不确定的奇异数时，浮点计算可以在没有外部中断的情况下有效地继续进行下去，同时希望最后结果不会受到严重影响，并且能作出可寻找的误差分析.

9.3 规格化浮点运算操作

四种标准的算术运算操作——加法、减法、乘法和除法．可以用具有较宽工作范围和较佳精度控制的浮点硬件来执行．我们将要描述对规格化浮点数 $x_1=(m_1, e_1)$ 和 $x_2=(m_2, e_2)$ 进行的这些操作，这里 $x=m\times r^e$，r 是蕴含的基数．尾数 m 是一个具有 p 个有效数位(符号除外)的带符号分数，它处在下列规格化的范围之内

$$\frac{1}{r}\leqslant |m|\leqslant 1-r^{-p}<1 \tag{9.7}$$

阶 e 是一个带符号的整数，它有 q 个有效数位(符号除外)，即

$$0\leqslant |e|\leqslant r^q-1 \tag{9.8}$$

阶是一个变量，它决定小数点的实际位置．浮点加法/减法 $x_1\pm x_2$ 正式定义如下：

$$(m_1, e_1)\pm(m_2, e_2)=\begin{cases}((m_1\pm m_2\times r^{-(e_1-e_2)}), e_1), & \text{如果 } e_1>e_2\\((m_1\times r^{-(e_2-e_1)}\pm m_2), e_2), & \text{如果 } e_1\leqslant e_2\end{cases} \tag{9.9}$$

上面的方程式指出：在执行有意义的加法/减法之前，二个数 x_1 和 x_2 的小数点必须对准．这一点是通过对二个阶的相对数值进行比较，并且将阶较小的那个尾数右移 $|e_1-e_2|$ 个位置来实现的．这个操作也反映在方程式 9.9 中，即用一个移位因子 $r^{-|e_1-e_2|}$ 来乘尾数．然后进行尾数的加法/减法，将较大的阶用作结果的阶．注意：结果的尾数绝对值（即 $|m|$)一定限制在下列范围内

$$0 \leqslant |m| < 2 \tag{9.10}$$

这个比较-移位-加法的过程基本上是串行的，因此浮点加法/减法与对应的定点相比需要较长的执行时间．即使在对准以后，也还会存在二个可能的复杂因素．第一个复杂因素相应于 $1 \leqslant |m| < 2$ 的情况，当同号的二个数相加或异号的二个数相减时，就可能出现这种情况．在这些场合，和或差的绝对值 $|m|$ 超过了单位值．有时候把这叫做尾数溢出问题．我们不打算把这个问题作为一种实际的上溢来加以讨论．不过这个问题是不难解决的，即把上溢的尾数右移一位，同时使阶增 1，

$$(m_1, e_1) \pm (m_2, e_2) = \begin{cases} (r^{-1} \times (m_1 \pm m_2 \times r^{-(e_1-e_2)}), e_1 + 1) & \text{如果 } e_1 > e_2 \\ (r^{-1} \times (m_1 \times r^{-(e_2-e_1)} \pm m_2), e_2 + 1) & \text{如果 } e_1 \leqslant e_2 \end{cases} \tag{9.11}$$

第二个复杂因数来自结果的尾数等于零，如方程式 9.4 所示．在这种情况下，有一个 OMZ 被形成．浮点加法/减法的最后一步是使结果的尾数规格化（如果它带有领前的零位）．但是，当 OMZ 出现时，算后规格化就不可能进行．这时必须产生一个专门的信号去向用户指示 OMZ 的存在．

浮点乘法 $x_1 \times x_2$ 和浮点除法 x_1/x_2 由下列二个运算方程式来定义

$$(m_1, e_1) \times (m_2, e_2) = (m_1 \times m_2, e_1 + e_2) \tag{9.12}$$

$$(m_1, e_1)/(m_2, e_2) = (m_1/m_2, e_1 - e_2) \tag{9.13}$$

尾数的乘法/除法和相应的阶的加法/减法是可以同时执行的．因此，对同样的分数长度，这些浮点乘法/除法的执行时间基本上与相应的定点操作相同．由于不执行定点运算中求比例因子的指令而节省下来的时间，被预置操作数和使最后乘积或商算后规格化所需要的时间抵消了．

这里有几点意见应该提一下．一般地说，如果使用方程式 9.12 和 9.13，那么结果的尾数值落在下列间隔内

$$\frac{1}{r^2} \leqslant |m_1 \times m_2| < 1; \tag{9.14}$$

$$\frac{1}{r} < |m_1/m_2| < r \tag{9.15}$$

这里假定了 $m_1 \neq 0 \neq m_2$．

当 $1/r \leqslant |m_1 \times m_2| < 1$ 时，不需要校正，最后乘积已经具有规格化的形式．然而当

$$\frac{1}{r^2} \leqslant |m_1 \times m_2| < \frac{1}{r} \tag{9.16}$$

时，最后乘积没有规格化．但左移一个数位就能使它规格化．在这种情况下，方程式 9.12 应该用下列方程式来代替：

$$(m_1, e_1) \times (m_2, e_2) = (r \times m_1 \times m_2, e_1 + e_2 - 1) \tag{9.17}$$

浮点除法不需要算后规格化，这是因为对 $m_1 < m_2$ 来说（方程式 (9.15)），最后的商 m_1/m_2 经常是规格化的．但是，当 $m_1 \geqslant m_2 \neq 0$ 时，将由于

$$1 \leqslant |m_1/m_2| < r \tag{9.18}$$

而出现除法上溢（或商上溢）．

右移一个数位就能消除商的上溢．如果商上溢的话，方程式 9.13 应该用方程式 9.19 来代替．

$$(m_1, e_1)/(m_2, e_2) = (m_1 \times r^{-1}/m_2, e_1 - e_2 + 1) \tag{9.19}$$

非规格化数的浮点运算规则将在下一章中讨论. 在以后各节中，我们将用有规则的流程图来详细介绍上述步骤，并且提出有关的浮点运算硬件处理器. 在第 9.8 节中将给出一个现有的浮点机的实例研究.

9.4 基本的浮点运算硬件

在这一节将描述基本浮点运算部件的硬件构造. 这个部件包括许多寄存器，二个加法器，以及控制和时序逻辑；我们用这个部件作为硬件的主体，以便于后面的几节中用来阐明标准的浮点操作加、减、乘、除的执行情况. 我们假定：浮点数据的格式具有 32 位字长，如图 9.2 所示. 假定基数 $r=2$，阶的偏置常数 $b=128$. 23 位尾数加上符号位就是一个具有带符号数值形式的规格化分数，并且意味着有一个二进位小数点紧跟在符号位 M_s 的右边. 按照通常规定符号的习惯

$$M_s=\begin{cases}0, & \text{对正数}\\1, & \text{对负数}\end{cases} \tag{9.20}$$

尾数的 23 位数值(分数) $\mathbf{M}=M_{1-23}$ 处在

$$0.5\leqslant \mathbf{M}<1 \tag{9.21}$$

的范围内. 这表示最高有效位 $M_1=1$. 被偏置的阶用小写字母 $\mathbf{m}=\mathbf{e}+b=m_{1-8}$ 来表示. 换句话说，$\mathbf{m}$ 是一个 8 位的不带符号的整数，它处在下列十进制数的范围内

$$0\leqslant \mathbf{m}\leqslant 255_{10} \tag{9.22}$$

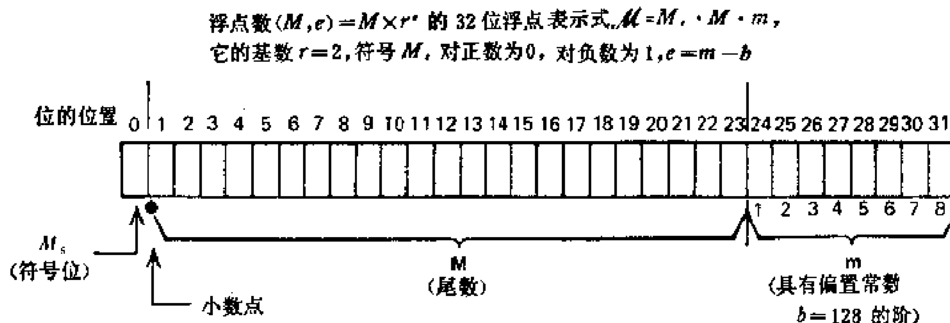


图 9.2 用来说明规格化浮点运算的 32 位浮点数据的格式

由下式

$$\mathbf{e}=\mathbf{m}-b=\mathbf{m}-128 \tag{9.23}$$

真正的阶 $\mathbf{e}$ 可以根据偏置值得到恢复.

必须指出，$\mathbf{e}$ 是以 2 的补码形式表示的带符号的整数，它在 $-128\leqslant \mathbf{e}\leqslant 127$ 的范围内. 真正的 $\mathbf{e}$ 值由程序员使用，而偏置值 $\mathbf{m}$ 由设计者使用. 阶加法器只处理偏置的阶.

从现在起，我们将用草体字母 $\mathcal{M}=M_s\cdot\mathbf{M}\cdot\mathbf{m}$ 表示从主存储器中取出的 32 位浮点数据字，这里 M_s 为符号，大写字母 $\mathbf{M}=M_{1-23}$ 为规格化尾数，小写字母 $\mathbf{m}=m_{1-8}$ 为偏置的阶.

图 9.3 指出了 32 位浮点运算处理器的硬件组成. 在这个处理器中有六个寄存器，分别用 $\mathbf{A}$, $\mathbf{a}$, $\mathbf{B}$, $\mathbf{b}$, $\mathbf{Q}$, $\mathbf{q}$ 表示. 由大写字母 $\mathbf{A}$, $\mathbf{B}$, $\mathbf{Q}$ 标记的寄存器每一个都有 23 位长，

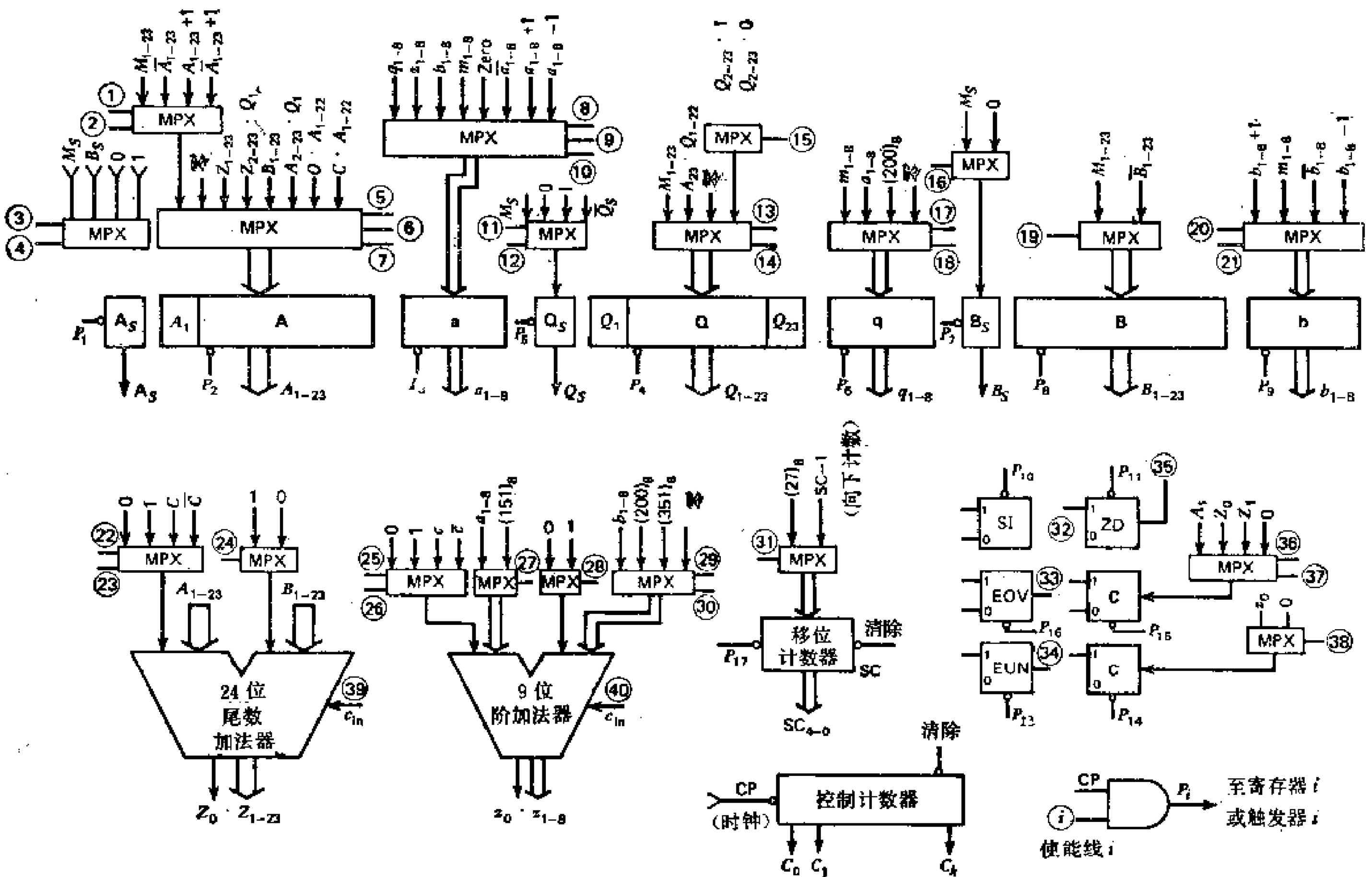


图 9.3　32 位浮点运算处理器的功能方框图

用来存储尾数. 小写字母 **a**, **b**, **q** 标记的寄存器每一个有 8 位长,用来存储阶. 下标用于指示这些寄存器中位的位置. 例如, $\mathbf{A}=A_{1-23}$, A_1 即指寄存器 **A** 的最高有效位, $\mathbf{a}=a_{1-8}$ 等等. 三个触发器 A_s, B_s, Q_s 被用于存储保留在相应的寄存器中的数的符号. 长度为 32 位的三个级联寄存器可以形成如下:

$$\mathscr{A}=A_s\cdot\mathbf{A}\cdot\mathbf{a};\quad \mathscr{B}=B_s\cdot\mathbf{B}\cdot\mathbf{b};\quad \mathscr{Q}=Q_s\cdot\mathbf{Q}\cdot\mathbf{q} \tag{9.24}$$

级联寄存器 $\mathscr{A}$, $\mathscr{B}$, $\mathscr{Q}$ 分别称为累加器,辅助寄存器和商-乘数寄存器. 当一个浮点数(数据)从存储单元 $\mathscr{M}$ 传送到寄存器 $\mathscr{B}$ 时,我们简单地写成 $\mathscr{B}\leftarrow\mathscr{M}$, 实际上这意味着有如下三项独立的寄存器数据传送

$$B_s\leftarrow M_s;\quad \mathbf{B}\leftarrow\mathbf{M};\quad \mathbf{b}\leftarrow\mathbf{m} \tag{9.25}$$

此外,还有二个用 C 和 c 表示的专用触发器,它们分别用来存储尾数加法器最高有效位的进位输出以及阶加法器的输出. 进位位是在二个尾数或二个阶的加法/减法执行过程中产生的,它能用来鉴别二个数的相对大小,或者用来决定上溢或下溢的条件. 如何使用这些进位条件的细节将在后几节中提到.

很明显,这里需要二个并行二进制加法器,一个用于处理尾数,另一个用于处理阶.尾数加法器 (**MA**) 长 24 位, 阶加法器 (**EA**) 长 9 位. 令 $Z=Z_0\cdot Z_{1-23}$ 和 $z=z_0\cdot z_{1-8}$ 分别为 **MA** 和 **EA** 的输出标记. 这二个加法器的输入和输出之间的关系用下面二个方程式来描述,这里运算符"+"和"·"分别表示算术加法和串连接.

$$Z_0\cdot Z_{1-23}=A_0\cdot A_{1-23}+0\cdot B_{1-23}+C_{in} \tag{9.26}$$

$$z_0\cdot z_{1-8}=a_0\cdot a_{1-8}+0\cdot b_{1-8}+c_{in} \tag{9.27}$$

式中 A_{1-23}, a_{1-8}, B_{1-23} 和 b_{1-8} 分别代表寄存器 **A**, **a**, **B** 和 **b** 的内容. 通过一个 4 输入的多路转换器, 领前的位 A_0 (或 a_0) 可以取 0, 1, C 和 $\bar{C}$ (或 0, 1, c 和 $\bar{c}$) 四个数值中的一个. 最后一项 C_{in} 和 c_{in} 分别代表二个加法器 **MA** 和 **EA** 的进位输入.

在每个处理器周期,多路转换器用于选择相应的数据输入到每个寄存器和加法器.移位则是利用接入多路转换器逻辑中的实际接线来实现的. 几个触发器用于指示特定的状况,如象阶上溢 (EOV), 阶下溢 (EUN),零除数 (ZD), 数量级零 (OMZ); 以及结果的符号指示器 (SI)等.

方式控制译码器没有画在图 9.3 中, 它用于从四种标准的浮点运算操作中选出一种. 带译码的计数器状态的控制计数器 (CC) 用于控制与主时钟耦合的处理器的操作. 启动线用于控制所有寄存器和触发器中状态的改变. 此外,移位计数器 (**SC**) 用于在乘法、除法和规格化的过程中, 记录移位的次数. 所有的控制线用带圆圈的数字来标记. 与每个运算功能相联系的详细的硬件操作将在下面几节中分别介绍.

9.5 规格化浮点加法/减法

在执行浮点加法/减法之前, 具有规格化形式的二个操作数——被加数/被减数和加数/减数分别输入到累加器 $\mathscr{A}$ 和辅助寄存器 $\mathscr{B}$. 结果的和/差也将被规格化,并且回送到累加器 $\mathscr{A}$. 在概念上,这些操作可以写成

$$\mathscr{A}\leftarrow\mathscr{A}\pm\mathscr{B} \tag{9.28}$$

为了完成二个浮点数的加法/减法,有四个主要步骤必须执行.

1. 检测零操作数；

2. 通过使它们的阶相等来调整尾数；

3. 尾数的加法/减法；

4. 使所得到的和/差规格化。

这四个步骤画在图 9.4 中．每一步中详细的硬件操作都用独立的流程图来说明．头二步零检测和使阶相等说明在图 9.5 中．当第二个操作数(寄存器 $\mathscr{B}$ 中的加数/减数)为零时，即不需要执行．当第一个操作数(在 $\mathscr{A}$ 中的被加数/被减数)为零时，对浮点加法来说结果就等于加数，对浮点减法来说，则等于减数的负数．只有当二个操作数均不为零时才进入第二步．

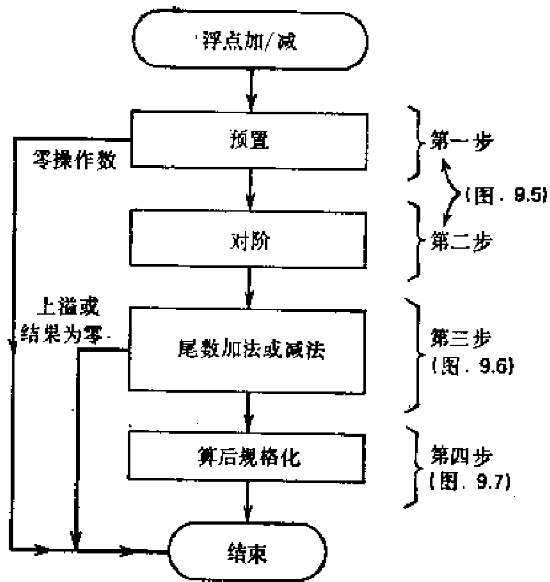


图 9.4　规格化浮点加法或浮点减法指令的四个执行步骤

方程式 9.9 和 9.10 中规定的阶的比较是用 2 的补码减法来实现的．寄存器级联(连接)通过介于二者之间的"点"算符来表示．箭头"←"用作置换算符．流程图中的算符"+"和"−"分别指示算术加法和减法(另有说明者除外)．上面一横表示按位求补．移位计数器从数值 27_8 (即 23_{10}) 开始计数．当二个阶 **a** 和 **b** 不等时，较小的那一个的尾数将右移足够次数，使得较小的阶可以增加到与较大的阶的数值相符合．

然后，我们进行第三步：尾数加法/减法．某些情况下，较小的阶与较大的阶相比时显得太小，这时经过最多 23 次移位之后尾数的数列必然排出在外．因此，具有较小的阶的操作数现在成了一个带尾数零的 OMZ．将这个 OMZ 加到一个合法的操作数上是在第三步中实现的，这里假定二个操作数现在有了相等的阶，但是 OMZ 具有零尾数，这种情况仍旧产生一个合法的结果，它等于把所涉及的较大的数取成正数和负数．

图 9.6 指明了尾数的加法或减法操作．另外，在二个加法器中假定都采用补码运算．图 9.6 中的流程图右半部描述了一个真值加法，它相应于下列四种情况之一：

$$(+A)+(+B);\ (+A)-(-B);\qquad (9.29)$$
$$(-A)+(-B);\ (-A)-(+B)$$

当真值加法的"和"超过单位值时，结果的尾数右移一位，并用"1"(进位)补入最高有效位，同时按方程式 9.11 将阶增 1．必须指出，寄存器 **A** 的最低有效位 A_{23} 已经被移到寄存器 **Q** 的最高位的位置 Q_1 上．有可能这样一个阶的增量(尽管只是 1)会引起阶的上溢，即 $(\mathbf{a}) > 377_8$．当出现上溢时，触发器 EOV 应被置位，在阶溢出以后，没有必要再进入规格化的程序．

图 9.6 中的流程图左半部相应于真值减法，它将处理下列四种情况之一：

$$(+A)+(-B);\ \ (+A)-(+B);\qquad (9.30)$$
$$(-A)+(+B);\ \ (-A)-(-B)$$

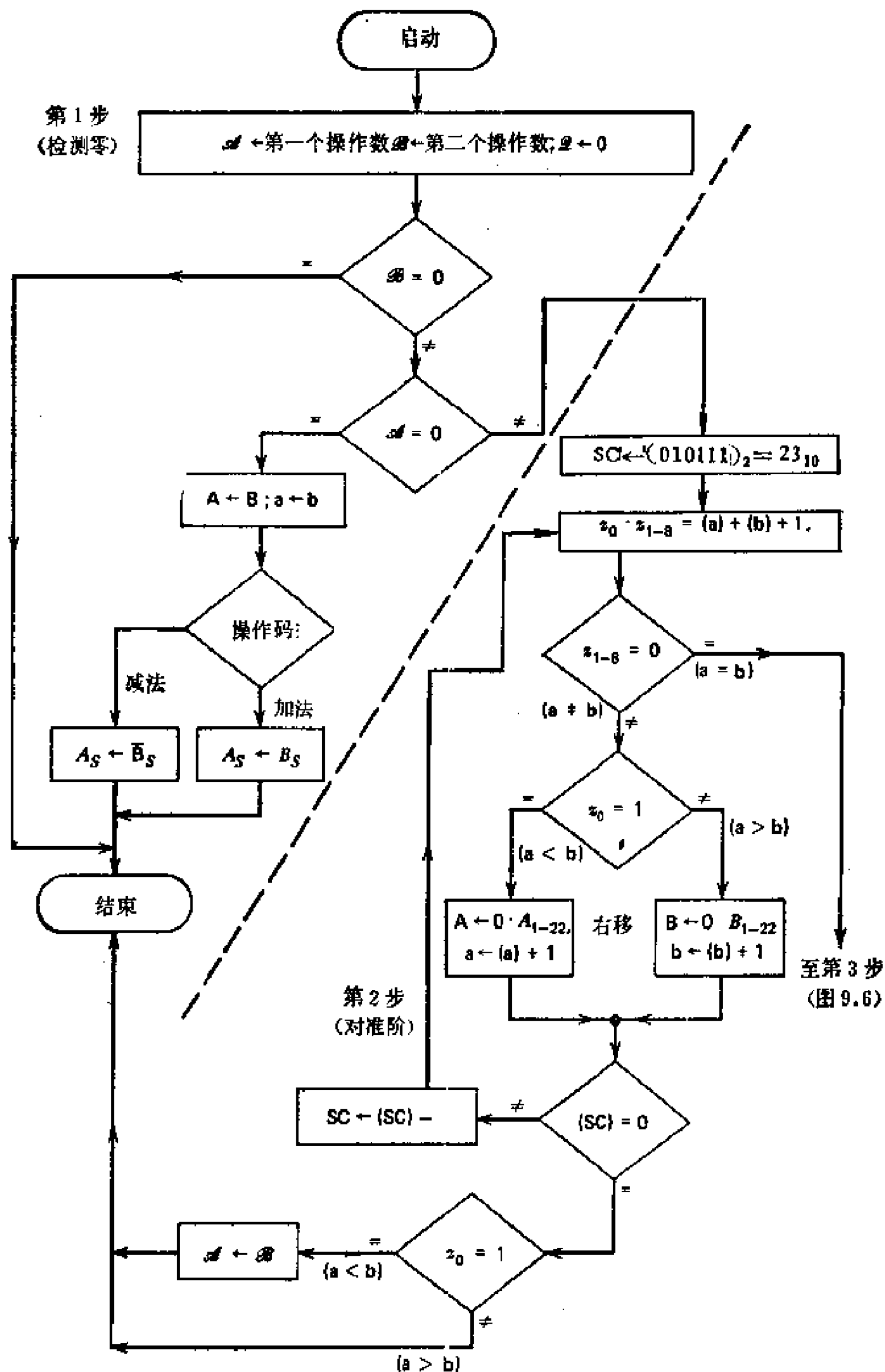


图 9.5 说明浮点加法/减法指令头二步的详细的微操作流程图

真值减法是通过补码加法来完成的．得到的差可以是正数，也可以是负数或零．当它是负数时（$A < B$），以补码表示的结果应该重新求补，回到它的带符号数值的形式．另外，当差为零时，操作便结束．规格化步骤只是当差为非零时才进入．

上述关于真值加法和真值减法的规定，指出了最高有效位有一个进位时，并不一定象定点操作中所做的那样指示一次溢出．代替它的是将阶增1，也就是二进位小数点向左移动一位，从而结果的尾数能正确保留．实际上，这是通过将尾数右移一位来实现的，如方

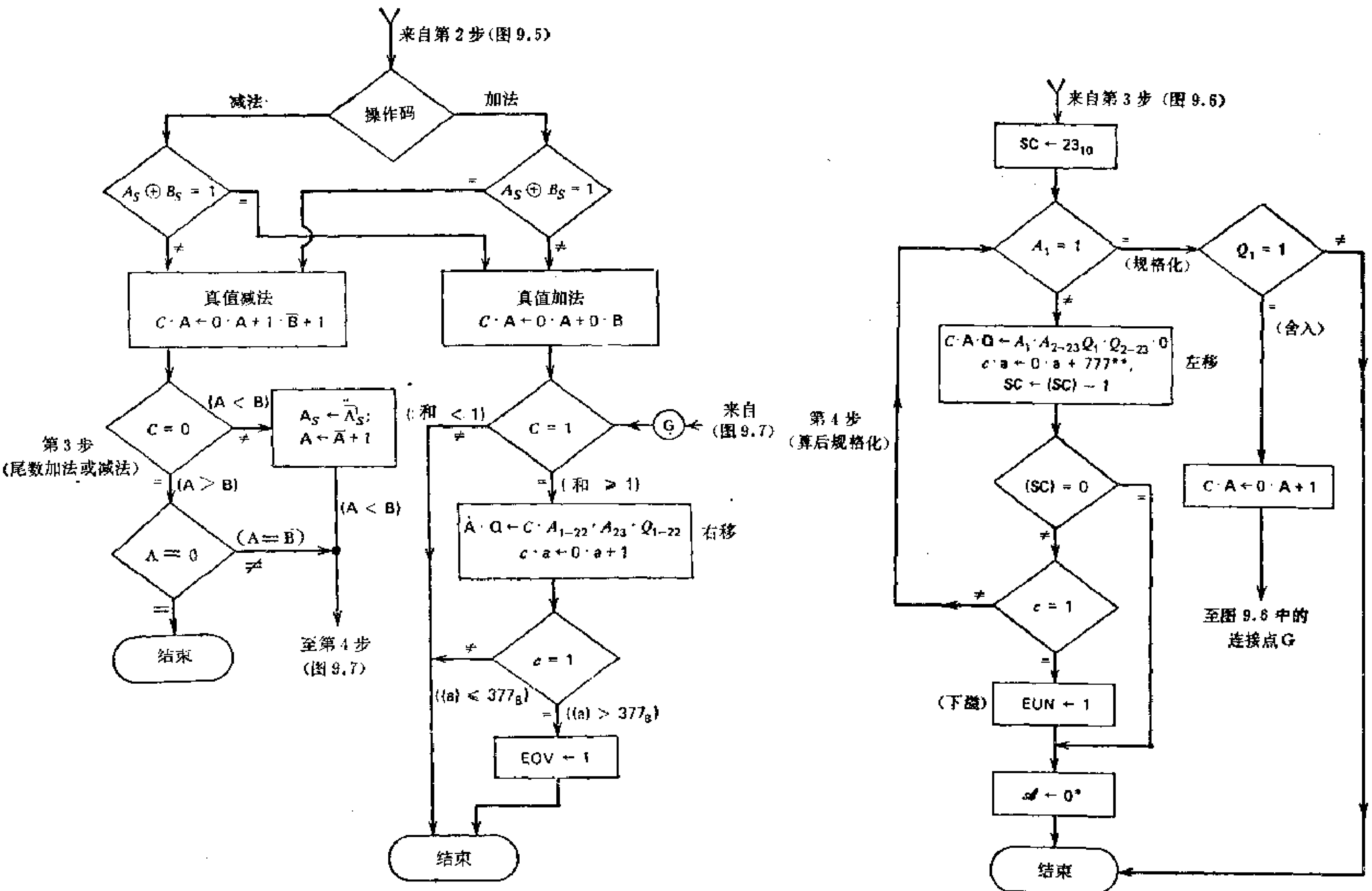


图 9.6　在执行浮点加法或浮点减法指令的第 3 步时，用来说明尾数加法/减法程序的流程图

图 9.7　规格化浮点加法或减法指令的规格化步骤(*产生一个 **OMZ**，** 利用补码运算，通过 **EA** 将 a 减 1)

程式 9.11 所示．最低有效位可能在舍入时丢失．在真值加法过程中，只有当阶超过极限时才出现上溢．

跟在尾数加法/减法后面的规格化操作在图 9.7 中作了描述．级联寄存器 $C \cdot \mathbf{A} \cdot \mathbf{Q}$ 一次左移一位，直到寄存器 **A** 中领前的位 A_1 成为"1"为止．于是，若 $Q_1 = 1$，就把舍入的 1 加到尾数上．最多经过 23 次左移后，如果结果仍未规格化，那么最后结果应该当作一个 OMZ 或零．在左移操作的过程中，也可能造成阶的下溢．当出现下溢时，最后结果可以指定为零，或者作为 OMZ 来处理．在流程图的说明中对这种情况仅仅指定零值．

9.6 规格化的浮点乘法

按照方程式 9.12 中的规定，浮点乘法是依靠二个操作数的尾数相乘，以及它们相应的阶相加来实现的．在对同号的二个阶执行真值加法时可能出现阶的上溢或下溢．与浮点乘法相联系的主要操作有四个．这些操作分三个阶段执行，如图 9.8 所示．预置阶段检测是否为零操作数，并且使乘积的符号置位．尾数的乘法和阶的加法这二步可同时执行．然而这二个并行的步骤必须在规格化步骤开始以前适当同步．最后一步规格化只有在方程式 9.16 中的条件出现时才有需要．

一开始，被乘数送入 $\mathscr{B}$ 寄存器，乘数送入 $\mathscr{Q}$ 寄存器，零送入累加器 $\mathscr{A}$．执行结束时，规格化的乘积留在累加器中．预置阶段中详细的微操作表示在图 9.8 中．当二个操作数具有同样的符号时，乘积为正，反之则为负．在对相应的尾数子寄存器 **B** 和 **Q** 进行检测以后，如果操作数中有一个为零，则所得乘积应置为零．接着，我们进行清除累加器 $\mathscr{A}$ 和使执行终止．如果二个操作数均不为零，我们便继续进行中间的步骤．图 9.9 中的流程图指出了与尾数相乘和阶相加有关的微操作．这二个操作可以并行执行．在某些廉价的机器中，只有一个加法器可以供算术运算操作使用，这时就可能要一个接一个地串行地执行，尽管这些操作本身是互相独立的．如流程图的左边所示，阶的加法下面跟着一个没有循环的程序．这个程序一开始是将乘数的阶从 **q** 寄存器送到子寄存器 **a**．阶加法器产生二个被偏置的阶的和；因此，所得的和的偏置加了一倍．下一步操作要求从加倍偏置的阶的和中减去偏置常数 $b = 128_{10} = 200_8$．对 200_8 的减法实际上是利用对 200_8 的 2 的补码(即 600_8)执行加法来实现的．当 **EA** 的进位输出是一个"1"时，对于 $(\mathbf{a}) > 377_8$ 或 $(\mathbf{a}) < 0$ 这二种情况，分别出现阶的上溢或下溢．否则，这个程序便进入规格化的步骤．

尾数乘法可以用第五章和第六章中介绍过的任一种定点乘法方案来实现．有了第 9.4 节中介绍的基本浮点硬件主机，我们便能用简单的加法-移位方案来完成浮点乘法，如流程图的右边所示．乘法程序一开始，就把 $27_8 = 23_{10}$ 送入移位计数器，以及把一个零作为初始部分乘积置入尾数寄存器 $C \cdot \mathbf{A}$．然后，我们检测乘数尾数的最低有效位 Q_{23} 的数值．如果它是 1，则通过 **MA** 将乘数尾数 **B** 加到部分乘积上．反之，就不需要作加法．以后，级联寄存器 $C \cdot \mathbf{A} \cdot \mathbf{Q}$ 的内容右移一位，移位计数器减 1．当 SC 成为零时，这个过程便告完成．否则，就进行下一次叠代，对乘数位下一个高位加以检测．这个过程一直继续到所有乘数位均被用到为止．如方程式 9.14 所示，在最后乘积中至多有一个领前的零，在这种情况下，只要左移一次便能使它规格化．实际上，上述操作会在级联寄存器 $\mathbf{A} \cdot \mathbf{Q}$ 中产生双倍字长的乘积．这个程序指出：所得乘积的下半部留在 **Q** 中，当 $Q_1 = 1$

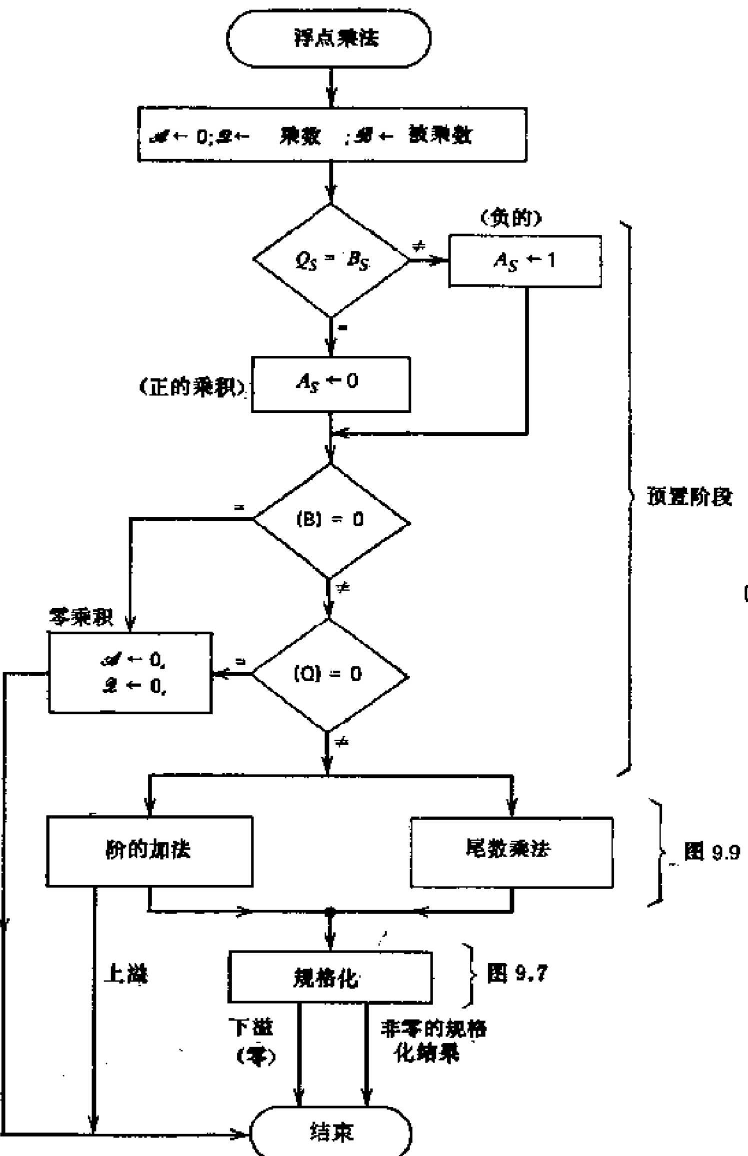


图 9.8 用来描述规格化浮点乘法指令的执行程序的流程图

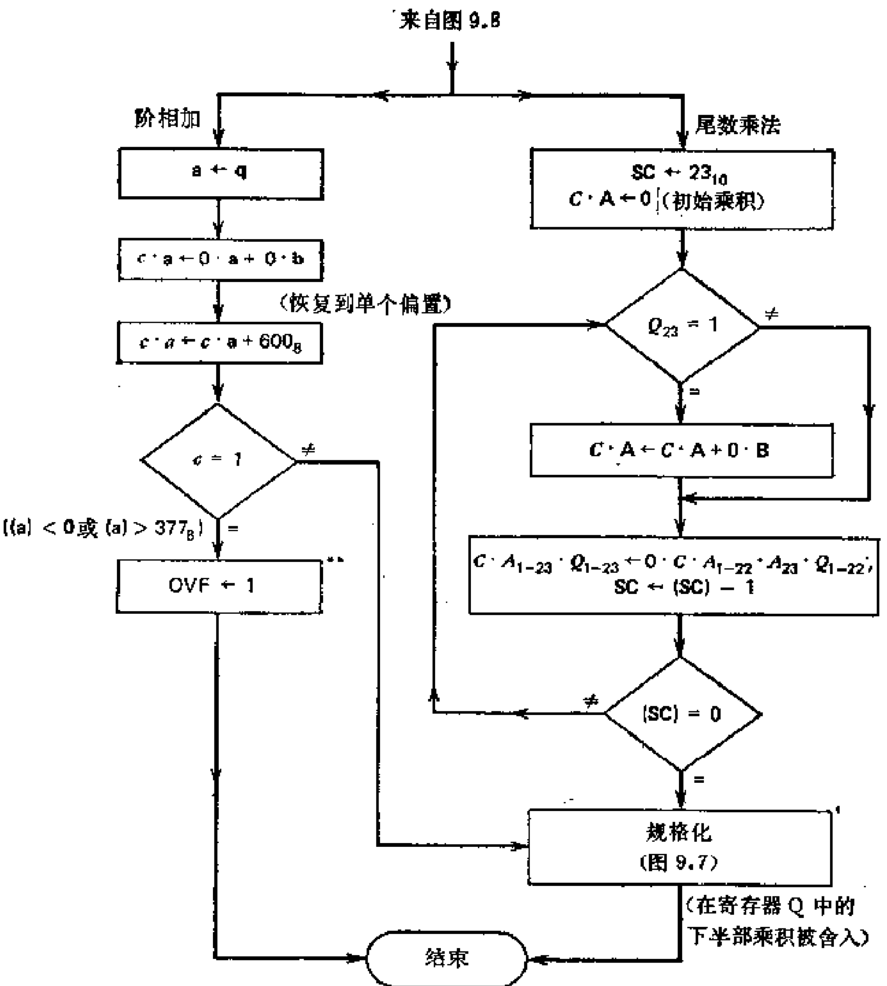


图 9.9 用于执行浮点乘法指令的阶的加法和尾数乘法的程序.
(* 为了使乘积规格化最多只需要左移一位.
** OVF 是一个标志信号,它或者是 EOV,或者是 EUN.)

时，能通过舍入将一个“1”进到 **A** 寄存器的最低有效位 A_{23} 上。在规格化步骤中的操作要求按方程式 9.17 的说明完成尾数的移位和阶的减量，如图 9.7 所示.

9.7 规格化的浮点除法

执行浮点除法指令的方法与浮点乘法相类似，所不同的只是用尾数的除法代替尾数乘法，用阶的减法代替阶的加法．当符号相反的二个阶执行真值加法时，有可能引起阶的上溢或下溢．除法方案还必须避免除数尾数小于被除数尾数的情况，包括除数为零的特殊情况．有了这个约束，在浮点除法中就不需要算后规格化，而只要求算前规格化来导致避免商的上溢.

与浮点除法有关的四种主要操作是：预置，尾数调整，阶的减法和尾数的除法．阶和尾数的操作是可以并行执行的，因此如图 9.10 所示，整个过程只包括三个相继的阶段．这二个并行的处理过程之间必须建立适当的同步．在浮点除法执行前和执行后，初始的和最后的寄存器分配画在下面：

初始内容	寄存器分配			最后内容
被除数	A_s	A	a	余　数
除　数	B_s	B	b	除　数
零	Q_s	Q	q	商

图 9.11

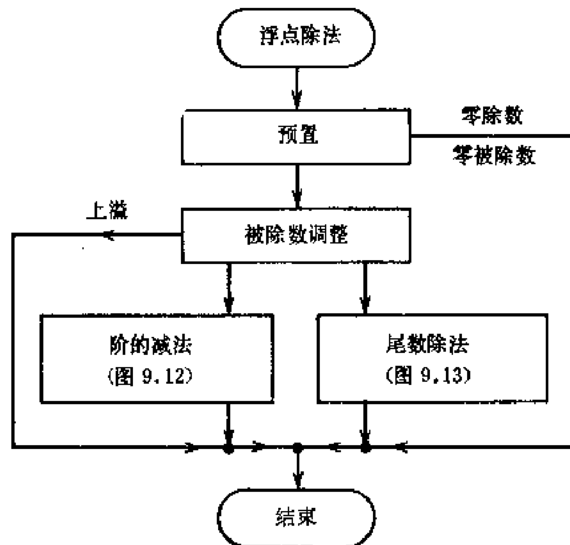


图 9.10　描述规格化浮点除法指令执行程序的流程图

预置和尾数调整的程序阐明在图 9.11 中．假定被除数和除数二者均以规格化形式给出．当被除数和除数具有相同的符号时，商的符号 Q_s 是正的，否则就是负的．当除数为零时，即引起商上溢，并使 ZD 标志置位．当除数不为零而被除数为零时，商被置成零，余数也置成零．当二个操作数均不为零时，我们才能进行尾数调整.

尾数调整基本上是一种防止上溢的操作，它保证了被除数的尾数比除数的尾数小．否则就不能进行有意义的尾数除法．这种调整经常产生一个规格化的商，因而不需要算后规格化．二个尾数的相对大小由比较操作来揭示．跟在比较操作后面的相应的动作取决于比较的结果．借助于第二章描述过的组合数值比较器，可以迅速地执行数值比较．利用图 9.3 中提供的硬件，我们能实现寄存器 **B** 中除数的尾数对寄存器 **A** 中被除数的尾数进行补码加法．必须指出，2 的补码加法是这样构成的，即对 B 按位求补，并且将一个“1”送到 **MA** 的进位输入端，然后再把它们加到 A 上.

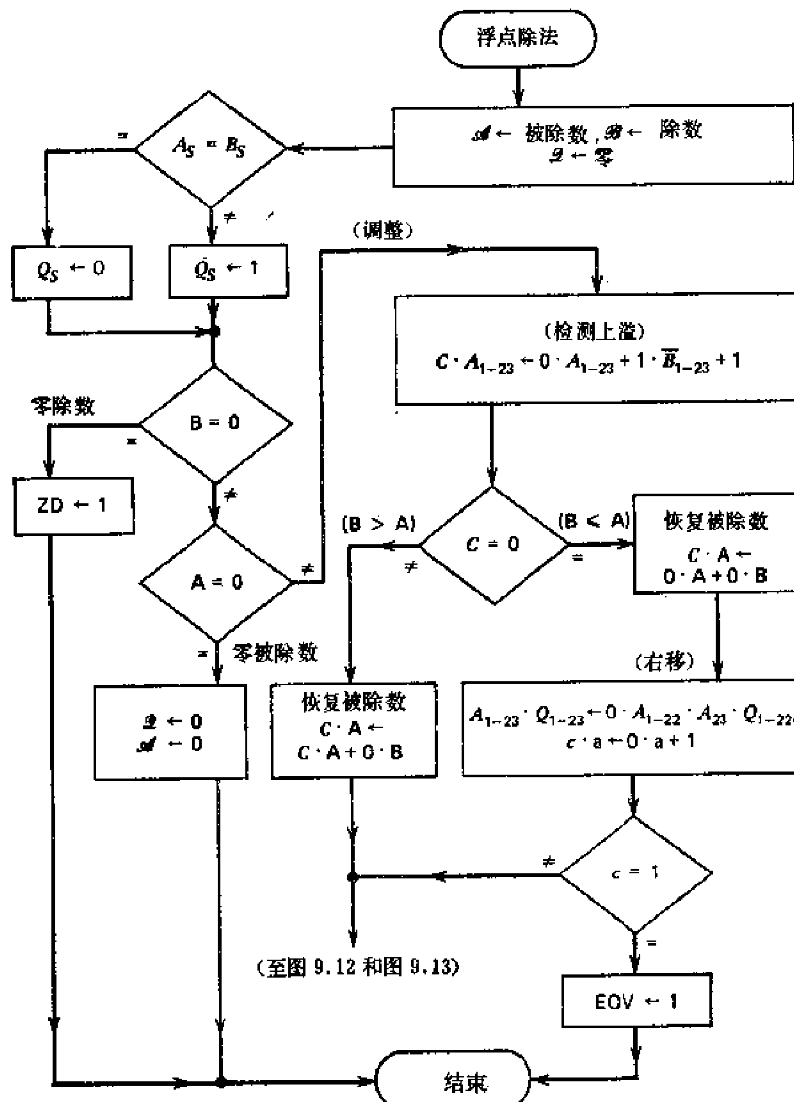


图 9.11 在浮点除法中，预置和被除数调整程序

当来自 **MA** 中的上述比较减法的进位 C 是一个零时，被除数（在 **A** 寄存器中）即大于或等于除数（在 **B** 寄存器中），简单地表示成 $(\mathbf{A} \geqslant \mathbf{B})$. 因此我们必须把除数加回去，以便恢复原来的被除数，即 $\mathbf{A} \leftarrow \mathbf{A} - \mathbf{B} + \mathbf{B}$, 然后将级联寄存器 $\mathbf{A} \cdot \mathbf{Q}$ 右移一个位置，并且从左端补入一个零. 这就使得被除数比除数小. 同时，被除数的阶按方程式 9.17 的要求增"1". 与这个增量操作相联系，有可能出现阶的溢出. 领前于阶的进位 C 能用来检测这个状态. 另一方面，在数值比较之后，C 为 1 时便揭示出相反的情况 $(\mathbf{A} < \mathbf{B})$, 即不产生上溢. 这时仍然需要用一次加法来恢复原来的被除数，但不需要移位. 这就完成了调整的程序.

阶的减法和尾数的除法二者都能在被除数尾数经过适当调整以后开始. 阶的减法操作说明在图 9.12 中. 对于廉价的设计，这二个操作可以共用同一硬件设备顺序执行. 阶的减法是通过 **EA** 用补码加法来实现的.

阶的进位 c 可以用来区分情况 $(\mathbf{a} \geqslant \mathbf{b})$（对 $c = 0$）和情况 $(\mathbf{a} < \mathbf{b})$（对 $c = 1$）. 在这二种情况中，我们都要恢复在阶的减法期间丢失的偏置常数 200_8. 偏置常数恢复到阶

寄存器 **a** 以后，我们可以简单地检测寄存器 **a** 中领前的位 a_1，以便揭示出任一奇异条件：对阶的上溢为 EOV = 1，对阶的下溢为 EUN = 1．如果没有奇异条件，那么我们就把阶的差从寄存器 **a** 传送到寄存器 **q**，后者将存储最后得到的商的阶．

余数的阶应该置 $23_{10} = 27_8$，它小于被除数的阶．这是通过 **EA** 从寄存器 $c\cdot\mathbf{a}$ 的内容中减去 27_8 来实现的．这个操作等效于把 751_8 加到寄存器 $c\cdot\mathbf{a}$ 的内容上去．必须注意，当 $\mathbf{a} < \mathbf{b}$ 时，阶的差 $\mathbf{a} - \mathbf{b}$ 是负的．从负的差中减去 23_{10} 有可能引起余数的阶下溢．这种情况下，我们简单地假定余数的阶是一个零值（以偏置形式表示的最负的阶），它在计算结束时保留在寄存器 **a** 中．

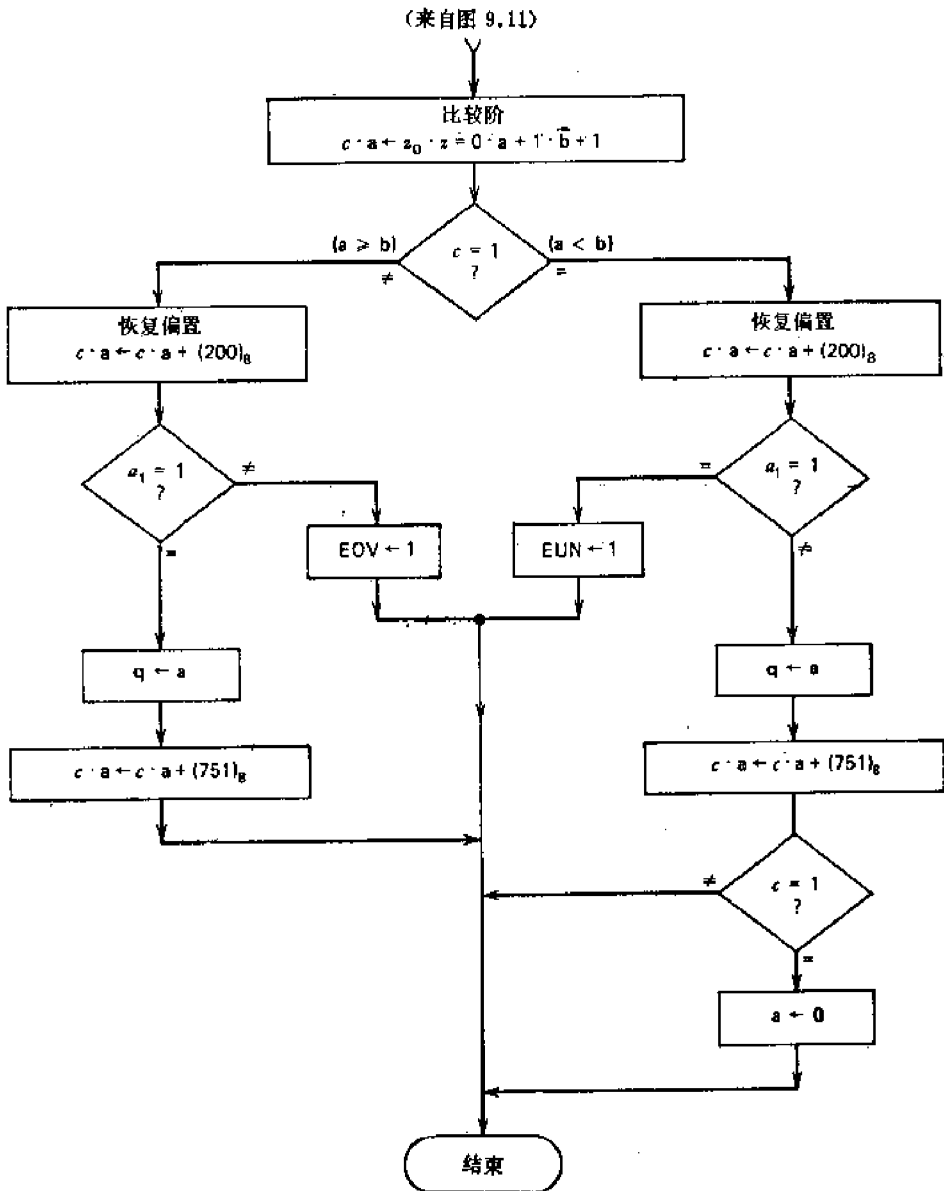


图 9.12　用于执行浮点除法指令的阶-减法程序

图 9.13 描述了尾数除法的程序． **SC** 应预置一个值 $23_{10} = 27_8$，它相当于尾数的长度．存储-移位带恢复除法的实现类似于第 7.4 节中的说明．级联寄存器 $C\cdot\mathbf{A}\cdot\mathbf{Q}$ 具有硬件移位的能力．换句话说，由于在这个寄存器中已设置了存储-移位特性，就不再需要

图 7.3 中用到的专设的移位器(旋转方框). $C\cdot\mathbf{A}\cdot\mathbf{Q}$ 寄存器与第 7.4 节中用过的 $L\cdot\mathbf{AC}\cdot\mathbf{QM}$ 寄存器相似. 另外，比较操作是用 2 的补码运算来完成的. 24 位 **MA** 控制着所需要的减法. 方程式 7.24 中规定的操作取代为

$$Z_0 \cdot Z_{1-23} = C \cdot A_{1-23} + 1 \cdot \bar{B}_{1-23} + 1 \tag{9.32}$$

生成的商从级联寄存器 $C\cdot\mathbf{A}\cdot\mathbf{Q}$ 右端移入.每叠代一次后，**SC** 减 1,直到到达数值"0"为止. 最后,还应该用一次右移将余数的尾数重新安置在寄存器 **A** 中. 这就完成了整个浮点除法的程序.

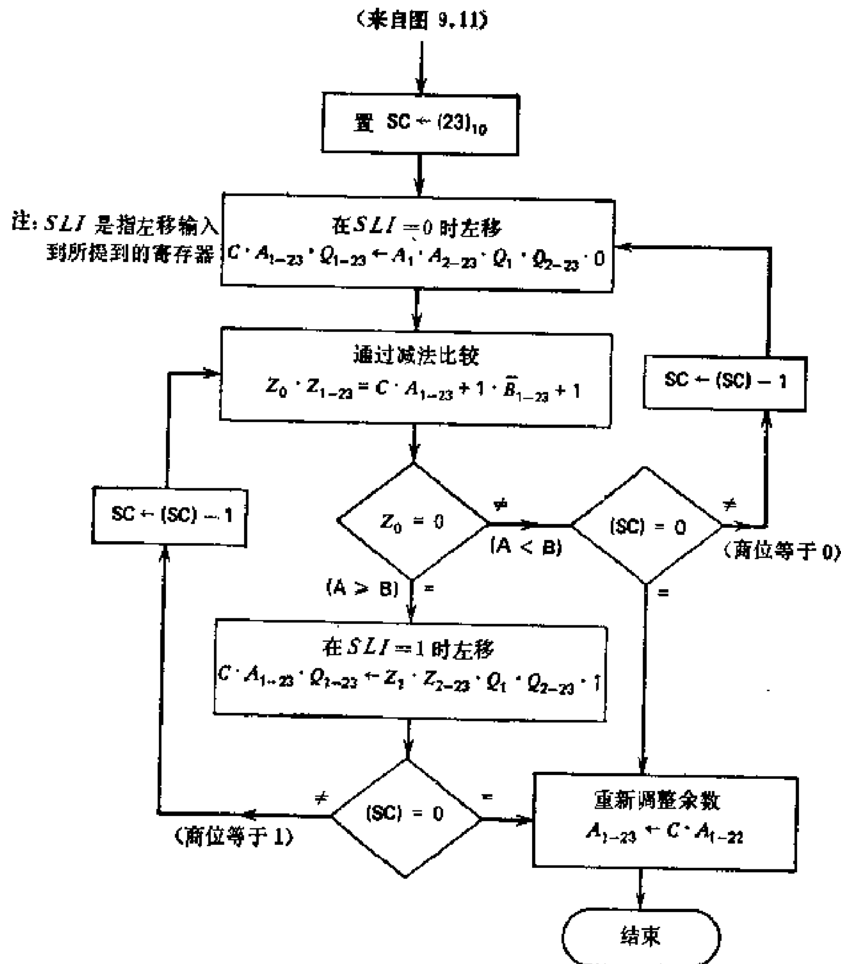


图 9.13 用于执行规格化浮点除法指令的尾数除法程序

9.8 实例研究 III:IBM360 系统 91 型的浮点运算处理器

这一节阐明一个实际的浮点处理器. IBM360 系统 91 型计算机有一个如图 9.14 所示的浮点运算处理器，这个处理器对加法功能和乘法/除法功能采用了面向指令的算法. 通过与浮点指令部件连接在一起，乘法浮点执行部件能以每个周期一条指令的快速速率执行指令. 下面我们介绍浮点处理器的结构，浮点数据的格式，浮点指令的定义,以及在 91 型浮点处理器中所包含的浮点加法部件和浮点乘法/除法部件. 在系统中配备了附加暂存器,使得流水线的运算操作有了可能,这些将在第 11 章中讨论.

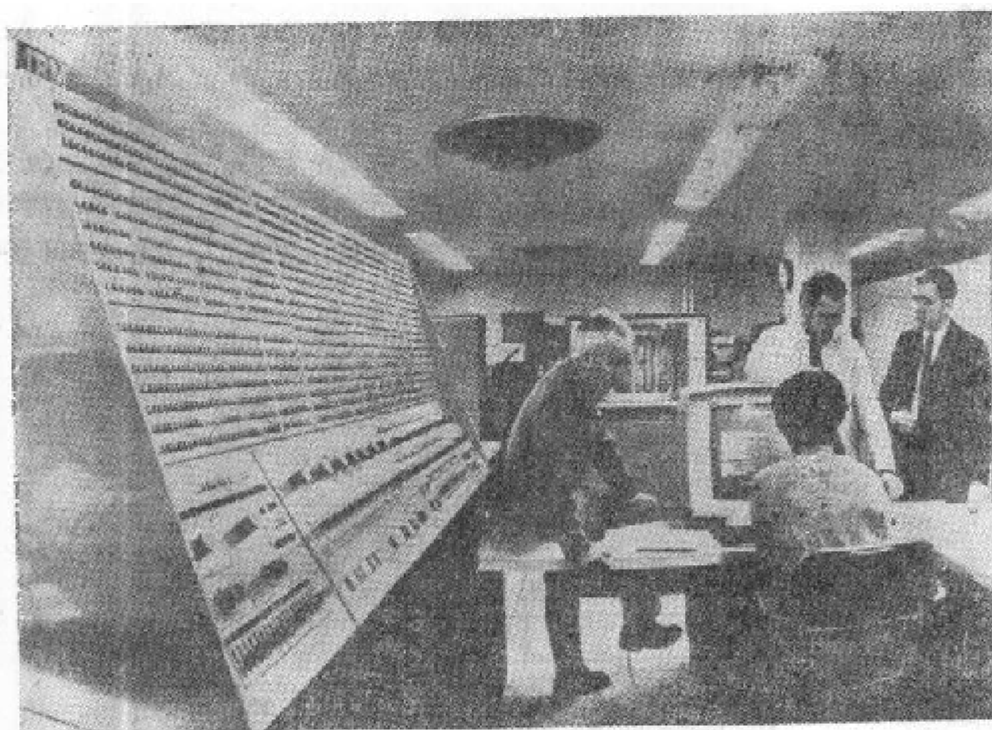


图 9.14 IBM360 系统 91 型计算机的主机架

在设计 **91** 型中的浮点执行部件（**FLEU**）时的第一件事就是研制一个完整的机构，这种机构应能适应 **91** 型中的浮点指令部件（**FLIU**）的特性. 浮点指令的执行时间比起由 **FLIU**发出这些指令的速率来说通常是比较长的. 最明显的方法是应用具有专门设计技术的快速工艺来减小每一条浮点指令的浮点执行时间. 但是，尽管采用现代化的算法，也不存在能与 **FLIU** 执行一个或二个周期相匹配的 **FLEU**. 另一种加快速度的方法是提供指令之间执行的并行性；这显然会要求二个或更多个 **FLEU**. 一个具有并行执行能力的浮点运算处理器的一般结构表示在图 9.15 中. 在这个图中指出了一个 **FLIU** 和二个 **FLEU**. **FLIU** 的主要功能是使操作数不断地从存储器送到适当的 **FLEU**. 这就必须缓冲这些指令，并且将每一条指令分配给不忙的 **FLEU**. 因为对所有的浮点指令，执行时间并不都相同，所以，有脱离程序执行的可能性. 指令之间的依赖性也应该加以控制. 如果第（N+1）条指令要依赖于第N条指令的结果，那么在第N条指令完成之前不允许开始第 N+1 条指令. **FLEU** 的职能就是缓冲和顺序控制所有这些指令、存储器操作数以及浮点累加器. 每个 **FLEU** 均能执行所有的浮点指令.

91 型计算机将它的浮点指令系统分成二个子系统：加法型指令系统和乘法/除法指令系统. 表 9.2 列出了这些指令，并且标出了执行每一指令的部件. 按照这个划分，在 **91** 型计算机中，有一个 **FLEU** 称为浮点加法部件，它能在二个周期内执行所有的加法型指令(加法或减法)，另一个 **FLEU** 称为浮点乘法/除法部件（**M/D** 部件)，它执行浮点乘法要用六个周期，执行浮点除法用 18 个周期. **IBM360/91** 浮点运算处理器的系统组成如图 9.16 所示. 浮点加法部件有三个保留站（reservation station)，它们作为三个独立的加法部件 $\mathbf{A}_1$, $\mathbf{A}_2$ 和 $\mathbf{A}_3$ 来处理. 浮点乘法/除法部件有二个保留站——**M/D1** 和 **M/D2**. 继 **360** 系统的结构和浮点指令的简述之后，下面将给出这些功能部件详细的设计.

在 **91** 型中浮点数据占用固定长度的格式，它可以具有一个全字的短格式，或是一个

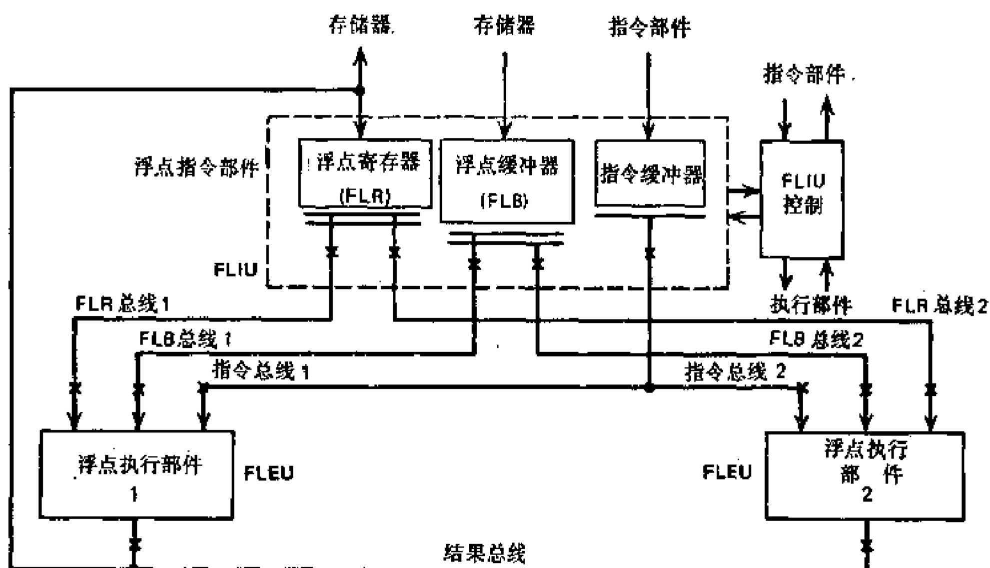


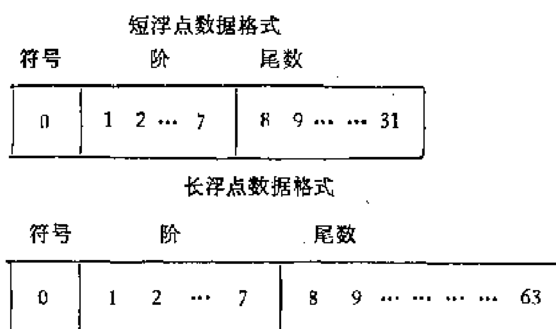
图 9.15 能并行执行浮点指令的浮点运算处理器的组成(Anderson 等[2])

表 9.2 IBM360 系统 91 型的浮点指令．这些指令由系统的各个不同的运算部件来执行

浮点指令	运算部件	机器周期	浮点指令	运算部件	机器周期
取数 (*S*/*L*)	FLIU	1	非规格化加法 (*S*/*L*)	加法部件	2
取数与测试(*S*/*L*)	FLIU	1	规格化减法 (*S*/*L*)	加法部件	2
存数 (*S*/*L*)	FLIU	1	非规格化减法(*S*/*L*)	加法部件	2
取补数 (*S*/*L*)	加法部件	2	比较 (*S*/*L*)	加法部件	2
取正数 (*S*/*L*)	加法部件	2	取半 (*S*/*L*)	加法部件	2
取负数 (*S*/*L*)	加法部件	2	乘法	乘法/除法部件	6
规格化加法 (*S*/*L*)	加法部件	2	除法	乘法/除法部件	18

缩写词：*S* = 短数据格式； *L* = 长数据格式； FLIU = 浮点指令部件．

双字的长格式：



最左边的位置是符号位：正的为“0”，负的为“1”．紧接着的七个位置被特征（偏置的阶）所占据．分数(尾数)包含十六进制的真正的数值位，对短格式来说是 6 位，对长格式来说是 14 位．小数点假定在位置 8 的二进位的前面．设具有偏置常数 64 的阶的基数为 16．因此，阶的范围从 −64 到 +63，相应于被偏置的二进制数值从 0 到 127．规格化减法、乘法和除法均保持以最高精度来执行．在这个系统中，也执行非规格化的加法或减法，如表 9.2 所示．无论是规格化的还是非规格化的和或差，都能依靠发出适当的指令来求出．

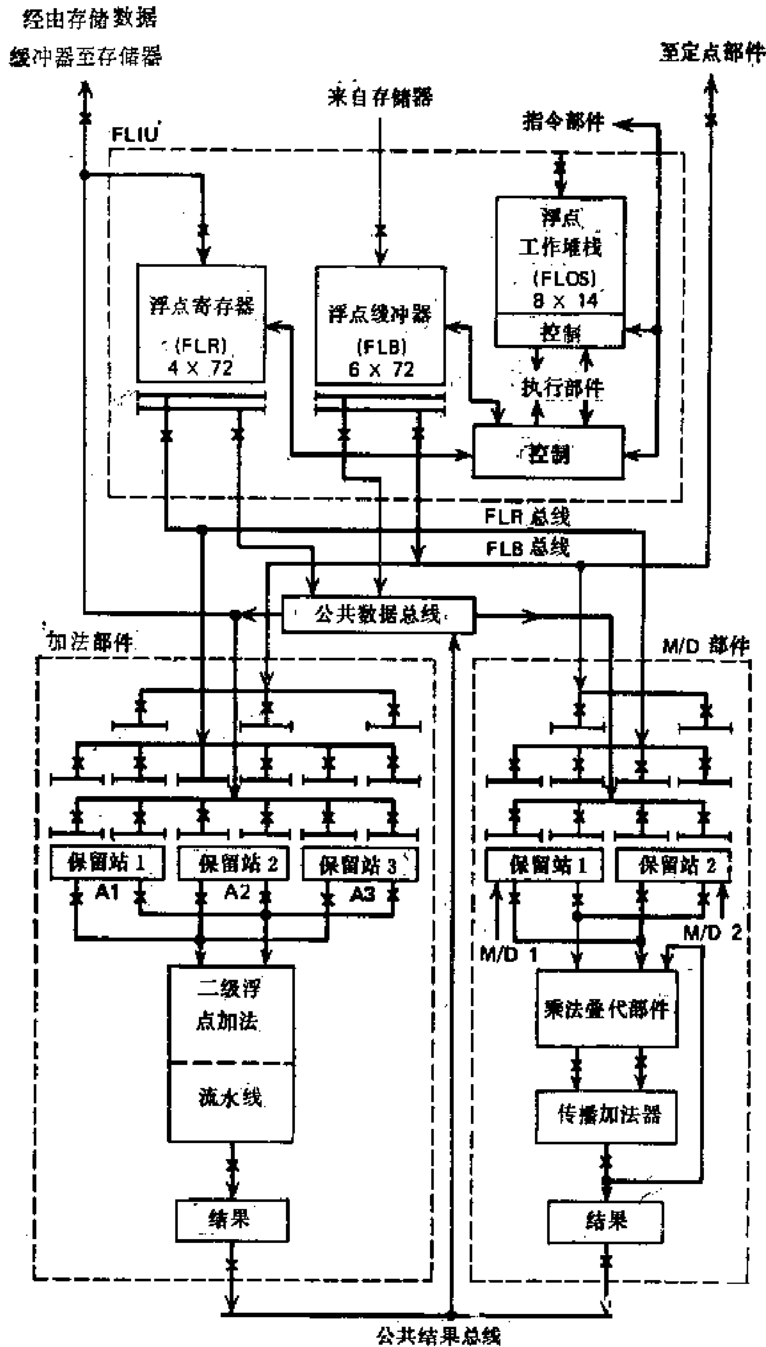


图 9.16 具有三个功能部件的 IBM 360 系统 91 型浮点运算处理器的原理方块图 (Anderson 等[2])

IBM360/91中的浮点加法部件

91 型中的浮点加法部件在设计上的困难在于使最长延迟通路中的逻辑级数最小化. **91** 型中的浮点加法/减法操作涉及较多的比较，如第 9.5 节所述. 加法部件的算法分成三部分：(1)特征比较与算前移位 (**CCP**)，(2)分数加法，(3)算后规格化. 浮点加法部件中的数据流说明在图 9.17 中. 该部件执行下列操作程序：

(a) 比较二个特征，并建立它们的差值.

(b) 将差值译码成移位次数，然后将具有较小特征的分数右移，使之与较大的特征相等.

(c) 使第二个分数通过真值-补码 (T/C) 逻辑，从而减法能用 2 的补码加法来完成.

(d) 二个分数在一个并行加法器中相加.

(e) 提供真值的和还是求补的和，取决于高位的进位.

(f) 如果需要的话，用规格化指令使所得结果的分数规格化(左移).

(g) 为了使所得结果的分数规格化，有必要在特征中减去左移的总次数.

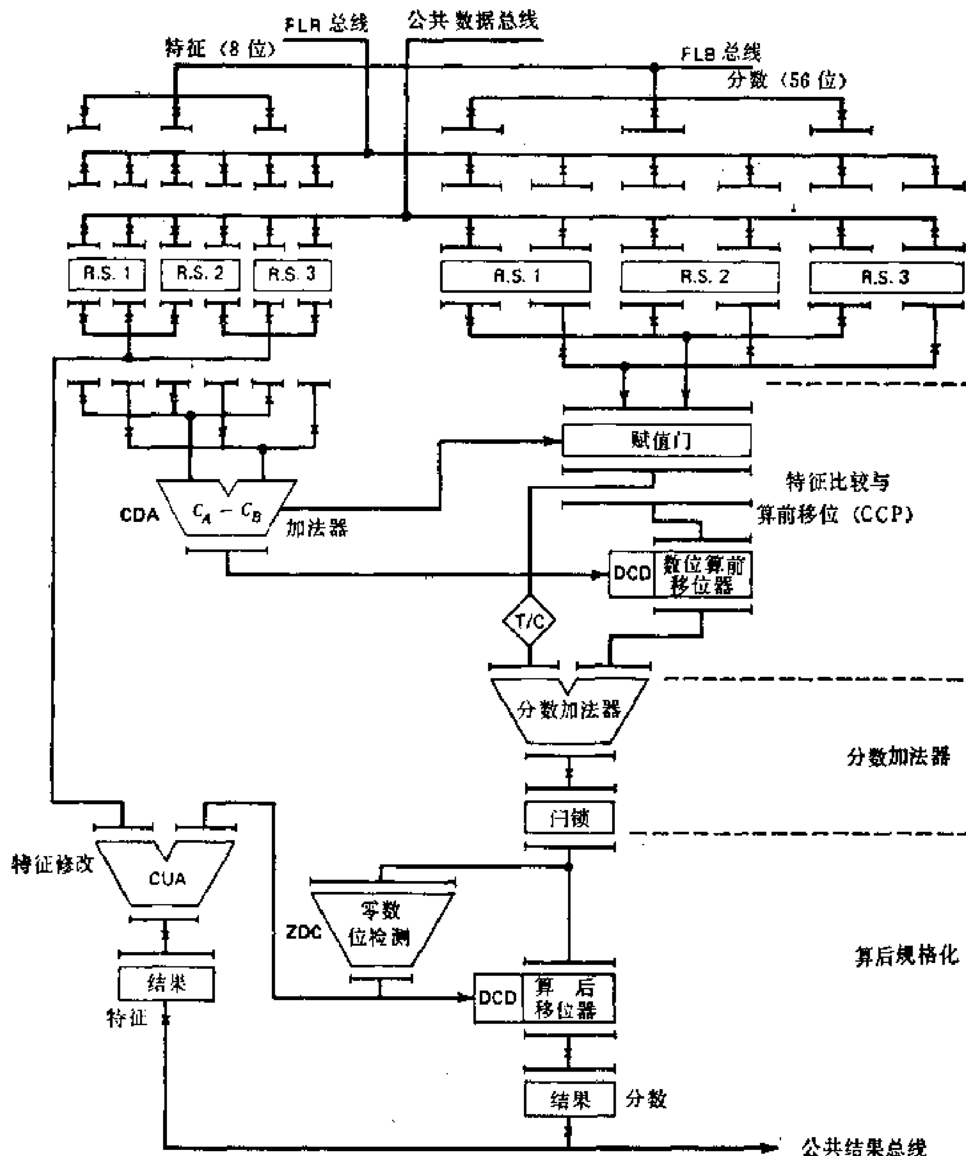


图 9.17 在 IBM360 系统 91 型计算机中，浮点加法器的原理方块图(Anderson 等[17])

(h) 将最后的操作数存入适当的累加器.

操作 (a)、(b) 和 (c) 由图 9.17 中的第一部分 CCP 来执行. 操作 (d) 和 (e) 归并在第二部分分数加法器中. 第三部分算后规格化进行最后三个操作 (f)、(g) 和 (h). 在 CCP 部分用了特征差的加法器 (CDA). 如果 $C_A \geq C_B$, 那么在 CDA 高位的位置

上就有一个进位输出，这个进位用来控制到算前移位器去的分数 B. 如果 $C_B > C_A$，CDA 即没有进位输出，控制分数 A 到算前移位器. 算前移位器是一个并行的数位-移位器，它使 14 个十六进制数位每次右移零到 15 位中的任意值.

分数加法器是一个二级 CLA 加法器，长 56 位，分成 14 组.第三部分零-数字检验器（ZDC）就是一个大型译码器，它检测出领前的零数字的个数，并且向算后移位器提供移位次数. 数位算后移位器的执行过程与数位算前移位器基本相同，所不同的只是算后移位为左移. 特征修改加法器（CUA）与分数移位并行地执行. ZDC 将最后的特征应减少的数值以 2 的补码形式提供给 CUA. 在这个浮点加法部件中总共用了三个高速并行加法器和二个移位器. 在第 9.4 节的基本设计上大量增加硬件，便能在二个处理器周期内高速度地执行双字长的浮点加法，几乎与中央处理器的指令发出的速率相适应.

IBM360/91 中的浮点乘法/除法部件

在 91 型计算机中，为浮点乘法和浮点除法所拟定的算法基本上共享同一硬件. 乘法算法基于重叠的 3 位扫描，它已经在第 5.5 节作了描述，并在第 5.6 节中加以实现. 56 位乘数（只指分数部分）的扫描模式说明在图 5.7 中. 五次重叠的叠代（每次叠代用 12 位）以及在每次叠代中如何产生被乘数的六个倍数均说明在图 5.8 中. 乘数再编码的规则列于表 5.4，在每三位一组的"窗"被扫描之后即产生一个倍数. 在图 5.9 中曾提出的二个进位存储加法器树，它们被用于实现被乘数倍数的加法. 基于这种叠代算法所必需的乘法硬件如图 5.10 所示.

另一方面，基于乘法的二次收敛的除法算法已在第 8.2 节中作了描述，并在第 8.3 节中加以实现. 图 8.1 阐明了 91 型中浮点操作的准确程序. 使用 CSA 和CPA 的硬件除法循环在图 8.2 中作了描述. 在收敛过程中，分母以及它们的乘数的格式列于表 8.1. 用来指明除法循环中的并行性的时间图如图 8.3 所示. 现在，我们将阐明上述部件设计怎样组成一个完整的系统，用以处理 91 型计算机的浮点乘法和浮点除法指令.

浮点乘法/除法部件完整的原理图如图 9.18 所示. 图中指出了执行规格化乘法或规格化除法的数据流. 数据流可分成叠代硬件和外围硬件（对叠代硬件来说是外围的硬件）两部分. 叠代硬件已经在第 5.6 和 8.3 节中作了描述. 外围硬件包括输入保留站，算前规格化，算后规格化，进位-传播加法器，结果寄存器以及特征运算等. 出现在 FLIU 中的二个保留站指定用作二个不同的乘法部件. 如果输入操作数为非规格化数，那么它们必须送往算前规格化器进行规格化，然后再回到原来的保留站. 为使一个操作数规格化所必需的左移次数被送到特征运算逻辑，那里用移位次数对特征加以修改.

所要求的特征加法与分数乘法的执行是重叠的. 由于输入被规格化，因此算后移位决不超过一个数位. 乘积是二个操作数累加得到的，叠代硬件的输出送往 CPA，去形成最后乘积. 算后规格化所需要的检测是与 CPA 并行进行的，结果送到公共的数据总线，或者左移一个数位或者不移位. 二次收敛除法的叠代操作在第 8.3 节中已经作了详细的描述. 浮点除法中所需要的特征减法与分数除法是并行执行的. 如第 9.7 节所述，除法不需要算后规格化.

并行性是 91 型浮点运算的高性能的关键所在. 我们已经研究的处理器组成中呈现出几级并行，现归纳如下:

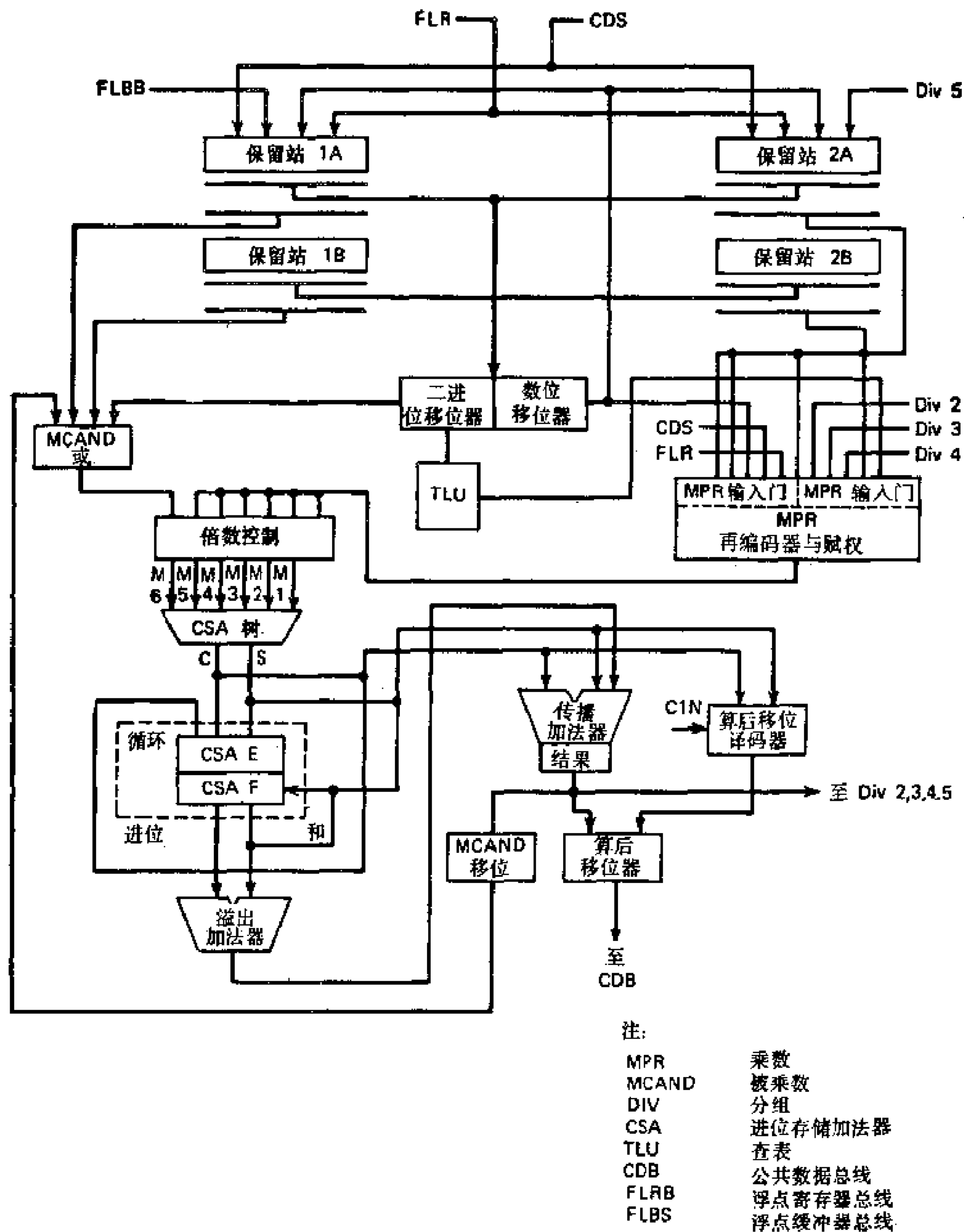


图 9.18 在 IBM 360 系统 91 型计算机中，浮点乘法/除法部件的原理方块图 (Anderson 等[2])

(a) 各类指令之间的并行执行.

(b) 同类指令之间的并行执行(加法部件).

(c) 在一条指令内部的并行执行(多次叠代的硬件和除法循环).

在 **91** 型中，公共的数据总线与寄存器特征线路允许同时执行一些独立的指令，而保持指令流中固有的基本步骤. 硬件先行约八条指令，并在局部基础上使程序执行自动最佳化. 这些技术的应用并不限于浮点运算或 **360** 系统的结构. 它几乎可以用于任何具有多个执行部件和多个累加器的计算机.

9.9 参考文献注释

本章介绍了有关浮点运算设计的一些基本论点. 除了第 9.1 节中给出的基本理由外，读者还能从 Kunth[10] 和 Sterbenz[15] 找到另外一些理由. 浮点运算严格的数学处理可在 Kunth 的著作中找到. Sterbenz 的著作详细描述了 IBM **360** 系统，包括对用户广泛应用的要求. 尤其是，在 STRETCH (IBM 7030 系统)计划中关于浮点奇异性的处理，已在 Campbell 的著作[3]中作了报道. 他绘制了 7030 的浮点特性的一份杰出的草图. 读者如希望研究更多的浮点处理器实例,应该读一读 Buckholz[3] 编写的作品[3]. IBM **360** 系统中的浮点操作在 [1, 8] 中作了描述. 第 9.3 节中列举的规格化浮点指令是在不涉及舍人或截断误差的情况下提出的. Chu[4] 和 Mano[12] 曾对规格化运算处理器作出扼要的解释. Chu 利用计算机设计语言 (CDL) 来说明浮点硬件，后者不难用一个通用计算机来模拟. 这种 CDL 模拟器现在可供各种类型的机器使用. 浮点运算的基数的选择曾由 Kahan[9], Matula[11] 和 Richman[13] 等研究过. 浮点运算的数值特征则在 Cody[5]; Gray 和 Harrison[7], 以及 Sweency[14] 中有过研究. 浮点操作较先进的课题将在下一章介绍.

IBM **360** 系统 **91** 型浮点指令部件、浮点加法部件和浮点乘法/除法部件的实例研究是依据 Anderson 等人的报告[2]. Flynn 和 Low[6] 提出同一系统的一篇重要报告. 对于那些想研究具有多个运算部件的系统中最大并行度的人来说，Tomasulo[16] 的论文是值得一读的.

参　考　文　献

[1] Amdahl, G. M., "The Structure of System/360, Part III, Processing Unit Considerations," *IBM System Journal*, Vol. 3, No. 1964, pp. 144—164.

[2] Anderson, S. F. et al., "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal*, January 1967, pp. 34—53.

[3] Campbell, S. G., "Floating-Point Operation," *Planning a Computer System*, W. Buckholz (ed.), McGraw-Hill, New York, 1962, Chapter 8.

[4] Chu, Y., *Computer Organization and Microprogramming*, Prentice-Hall, Englewood Cliffs, N. J., 1972, Chapter 5.

[5] Cody, W. J., Jr., "Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. C-28, June 1973, pp. 596—601.

[6] Flynn, M. J. and Low, P. R., "The IBM System/360 Model 91: Some Remarks on System Development," *IBM Journal*, January 1967, pp. 2—7.

[7] Gary, H. L. and Harrison, C. Jr., "Normalized Floating-Point Arithmetic with an Index of Significance," *Proc. of the Eastern Joint Computer Conference*, 1959, pp. 244—248.

[8] IBM Staff, "Floating-Point Arithmetic," in *IBM System/360 Principles of Operation*, IBM System Ref. Lib. From A22-6821-7, September 1968, pp. 41—50.3.

[9] Kahan, W., "What Is the Best Base for Floating-Point Arithmetic, Is Binary Best?," *Lecture Notes*, Dept. of Computer Sci., University of California, Berkeley, California, 1970.

[10] Knuth, D. E., *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, Addison-Wesley, Reading, Mass., 1969, Chapter 4.

[11] Matula, D. W., "A Formalization of Floating-Point Numeric Base Conversion," *IEEE Trans. Comput.*, Vol. C-19, August 1970, pp. 681—692.

[12] Mano, M. M., *Computer System Architecture*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976, Chapter 10.

[13] Richman, P. L., "Floating-Point Number Representations: Base Choice Versus Exponent Range," *Tech. Rep. No. CS-64*, Dept. of Computer Sci., Stanford University, Stanford, Califo-

rnia, 1967.
[14] Sweeney, D. W., "An Analysis of Floating-Point Addition," *IBM System Journal*, Vol. 4, No. 1. 1965, pp. 31—42.
[15] Sterbenz, P. H., *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.
[16] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, January 1967, pp. 25—33.

习　　题

题9.1　试就数的范围、精度、程序设计以及设计考虑等几方面对定点运算与浮点运算进行比较，说明它们的优缺点。

题9.2　设有一个48位的浮点处理器，它有37位用带符号数值表示的尾数，11位用2的补码形式表示的阶，基数 $r=16$. 试分别说明该机器对规格化和非规格化操作可表示的合法浮点数的范围．相邻浮点数之间的数值间隔有多大，并阐明该机器的规格化无穷小 $\pm\varepsilon_n$ 和非规格化无穷小 $\pm\varepsilon_u$ 之间的区别．最后证明在表9.1中所列举的奇异数的浮点操作。

题9.3　设有一个32位浮点运算处理器如图9.3所示，它具有40个控制端和16条用于寄存器或触发器的启动线．试列出逻辑和运算条件，这里控制线5,15和39被置成逻辑"1"，并用图9.4至图9.13给出的各流程图来给予解释．

题9.4　什么是"真值加法"和"真值减法"？证明阶上溢条件与图9.6中真值加法有关，阶下溢条件与图9.7中的算后规格化有关．解释为什么第9.7节描述的规格化除法不需要算后规格化．

题9.5　在规格化浮点加法/减法期间，图9.5中阶的调整程序指出：当二个阶的差超过最大的尾数长度 p 时，具有较小的阶的尾数全部从寄存器移出（剩下一个零尾数或一个OMZ）．在上述情况下，如果只用较大的数来决定结果，那么可能出现的最大误差是多大？你能提出一种减小这个误差的方法吗？

题9.6　试分别阐述2的补码减法和1的补码减法怎样用来比较二个带符号数值的数的相对大小．并说明阶的上溢（EOV）或阶的下溢（EUN）能用这二个带符号补码运算系统来检测．从设计观点来看，这二个系统中哪一个硬件费用少，为什么？

题9.7　试利用本章描述的基本的32位浮点运算硬件，说明下列浮点数的数值计算的具体步骤．并且逐步地列出工作寄存器，二个加法器的输入和输出以及某些标志值的内容．

(a) $(+0.75\times2^{-5})+(-0.625\times2^{+15})$.

(b) $(+0.5625\times2^{+251})-(-0.75\times2^{247})$.

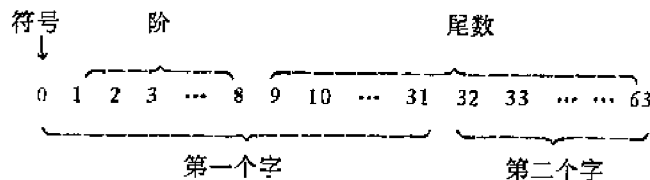
(c) $(+0.75\times2^{-5})\times(-0.625\times2^{+15})$.

(d) $(+0.5625\times2^{+251})\times(+0.75\times2^{+18})$.

(e) $(+0.75\times2^{-5})\div(-0.625\times2^{+15})$.

(f) $(+0.5625\times2^{-152})\div(+0.75\times2^{+18})$.

题9.8　假定用下列数据格式将该32位机器扩充成双精度浮点计算．试确定：为了执行双精度浮点运算，需要哪些必要的硬件元件（类似于图9.3的方块图）．注意每个操作数现在都是64位长，并占据二个32位的存储器字．



题9.9　考虑本章正文中描述的具有带偏置的阶和规格化的尾数的32位浮点运算处理器．令 $\mathscr{M}=(M,m)$ 和 $\mathscr{N}=(N,n)$ 为二个非零的规格化浮点数，它的格式如图9.2．试说明引起下列每一

种奇异结果的条件:

(a) 在 $\mathscr{M}+\mathscr{N}$ 后上溢;

(b) 在 $\mathscr{M}\times\mathscr{N}$ 后上溢;

(c) 在 $\mathscr{M}\div\mathscr{N}$ 后上溢;

(d) 在 $\mathscr{M}+\mathscr{N}$ 后下溢;

(e) 在 $\mathscr{M}\times\mathscr{N}$ 后下溢;

(f) 在 $\mathscr{M}\div\mathscr{N}$ 后下溢。

第十章　关于浮点运算的新课题

10.1　引　　言

在第九章中仅仅研究了规格化的浮点运算操作以及有关的处理器的设计．在这一章中，我们要研究几个与有效的浮点运算设计有关的新课题．我们首先从非规格化浮点运算操作的说明出发．在非规格化运算中并不需要算后规格化，这有优点也有缺点．我们将对规格化的和非规格化的二种浮点运算系统进行比较，指出它们设计上的差别．

浮点计算结果中引起的误差主要来自规格化、截断或舍入操作．首先阐明如象截断与最接近的邻域的舍入那样一些基本的舍入方案．在第10.4节中，根据从实数的集合到机器可表示的数的集合之间的代数映象，提出了公认的舍入理论的一种较严格的处理方法．利用这些映象，仿造了五种不同的舍入方案，并且从设计者的观点作了比较．第10.7节中利用ROM或PLA提出了一种新的舍入装置的方案．然后将这种ROM舍入方案与几个以加法器为基础的舍入方案进行比较．本章还提出了浮点机精度的统计分析．对于有效的浮点系统的设计与应用，根据最佳的基本选择，导出了运算误差的界限．为了寻求较好的舍入方案和较好的误差分析方法，这些结果对设计者和用户均能加以利用．对各类浮点指令算法的理论研究在10.8节中给出．根据数学映象进行比较研究，以及为每一类浮点指令的定义作出预测．这些映象和预测用来把操作数对的数据空间划分成不连贯的几类，它为浮点操作中预测规格化、截断和舍入的影响提供了有价值的信息．

然后，我们描述多倍精度浮点运算的特殊要求，即在消耗较多存储器空间和较复杂的程序控制的情况下提供较好的精度．特别是对一个具有48位单字长和96位双字长的计算机系统，阐明了双倍精度加法、减法、乘法和除法的算法．读者在研究本章提到的材料之前，应该对规格化浮点操作有全面的了解，本章的这些材料包括形式上的证明，数学描述以及浮点操作的新算法．这些研究为读者提供了发展有效的浮点运算系统所必需的工具．本章列举的参考文献提出了应该做的进一步探索工作的方向．

10.2　非规格化浮点运算

在第九章中描述的浮点运算操作是处理规格化的操作数．这一节我们将提出采用非规格化浮点运算操作的优点和缺点．非规格化浮点运算的一个很好的例子就是控制数据公司的CDC **6600**计算机，它的所有指令均执行非规格化运算．下面我们来说明非规格化数据的四种基本的运算操作，并指明它们与规格化设计的差别．

一个基数r的非规格化浮点数(f, e)有一个单位区间内的尾数f

$$|f| < 1 \tag{10.1}$$

这个非规格化尾数f可能具有小于r^{-1}的数值．这意味着一个非规格化浮点数可以有许多内部表示式．每一个合法的浮点表示式(f, e)相当于尾数移了不同位数，同时阶相应

地上升或下降．让我们假定尾数 f 是一个带符号的分数，在小数点右边有 p 个真值数位，阶 e 是一个带符号数值的整数，它有 $q+1$ 个数位，包括符号位在内，并满足方程式 9.8．虽然对二个字段均假定为带符号数值的形式，但是这种推导也能很好地用补码表示式来实现．当然，在它用规格化浮点硬件进行处理之前，常常要对一个非规格化的操作数进行规格化．这种规格化操作会使处理速度降低，而且要求较多的硬件．计算机系统 CDC**6600** 允许直接对非规格化数执行算术运算操作，只是在需要时才用一条单独的 NORMALIZE（规格化）指令使它们变换成规格化的形式．IBM**360** 系统具有规格化和非规格化两种运算硬件，因而能执行任何一种浮点指令．

与处理二个非规格化数 (f_1, e_1) 和 (f_2, e_2) 有关的运算规则定义如下，这里假定基数 $r=2$．浮点加法和浮点减法可以理解成同一类操作，因为尾数是可正可负的．我们假定阶 $e_1 \geqslant e_2$，这并不丧失一般性（加法是可交换的）．对非规格化运算，曾经提出过下列浮点加法规则

$$(f_1, e_1)+(f_2, e_2)=\begin{cases}(f_1+f_2\times 2^{e_2-e_1}, e_1), & \text{如果 } S<1\\((f_1+f_2\times 2^{e_2-e_1})\times 2^{-1}, e_1+1), & \text{如果 } S\geqslant 1\end{cases} \tag{10.2}$$

这里 $S=|f_1+f_2\times 2^{e_2-e_1}|$．$S\geqslant 1$ 是指所得到的和有一个超出分数极限值的尾数．因此，必须右移一位．除了最后的规格化步骤被省略以外，这个定义类似于规格化浮点加法．这些操作数不需要规格化，可以预计，在非规格化运算中上溢是很少出现的．

令 z 为非规格化浮点数 (f, e) 的尾数字段中领前的零的个数．注意：当 $z=p$ 时，$f=0$，这里 p 为尾数的长度．令 (m, e') 按照下式由 (f, e) 求得

$$\begin{cases}m=f\times 2^z\\e'=e-z\end{cases} \tag{10.3}$$

换句话说，(m, e') 为等效的规格化浮点数．我们可以将上式简写成

$$(m, e')=(f\times 2^z, e-z) \tag{10.4}$$

具有操作数 (f_1, e_1) 和 (f_2, e_2) 的浮点乘法和浮点除法可以用变量 λ 来定义，后者是 m_1 和 m_2 的函数．假设 $|f_1|\leqslant|f_2|$，这就保证了 $z_1\geqslant z_2$；非规格化乘法定义为

$$(f_1, e_1)\times(f_2, e_2)=(m_1\times m_2\times 2^{-z_1+\lambda}; e_1+e_2-z_2-\lambda) \tag{10.5}$$

若取 $\lambda=0$，那么这个操作可以这样实现：先让尾数较大的（绝对值）操作数规格化，然后再让两个尾数相乘，并把两个阶相加．在 $\lambda=0$ 的假定之下，我们可以把方程式 10.5 改写如下：

$$(f_1, e_1)\times(f_2, e_2)=(f_1\times m_2, e_1+e_2-z_2) \tag{10.6}$$

对于 $\frac{1}{2}\leqslant|m_2|<1$，乘积的尾数 $f_1\times m_2$ 大致具有 f_1 相同的数值．而且，对非零操作数，$\frac{1}{4}\leqslant|m_1\times m_2|<1$，显然，在所有场合对 $\lambda=0$ 有 $z=z_1$ 或 z_1+1，这里 z 为最终乘积的尾数中领前的零的个数．如果选 $\lambda=1$，乘积的尾数将是 $2f_1\times m_2$．并且在所有场合 $z=z_1-1$ 或 z_1．一般地说，在具有 $\lambda=0$ 或 1 的浮点乘法之后，我们为参数 z 指定下列数值

$$z=z_1 \text{ 或 } z_1\pm 1 \tag{10.7}$$

同理，我们定义非规格化浮点除法为

$$
(f_1, e_1)/(f_2, e_2) = \begin{cases} \left(\frac{m_1}{m_2} \times 2^{-z_1-\lambda}, e_1 - e_2 + z_2 + \lambda\right), & \text{如果 } |f_1| \leqslant |f_2|; \\ \left(\frac{m_1}{m_2} \times 2^{-z_2-\lambda}, e_1 - e_2 - z_1 + 2z_2 + \lambda\right), & \text{如果 } |f_1| > |f_2|. \end{cases} \tag{10.8}
$$

特别是，当遇到一个具有 $f_2 = 0$ 和 $m_2 = 0$ 的零除数时，商便成为

$$
(0, e_1 - e_2 - z_1 + 2p + \lambda) \tag{10.9}
$$

方程式 10.8 中求出的商，它的尾数具有 z 个领前的零，这里的 z 由下式给出

$$
z = \max(z_1, z_2) \text{ 或 } \max(z_1, z_2) \pm 1 \tag{10.10}
$$

二者分别对应于选择 $\lambda = 0$ 或 1.

上述操作指出：在一个 p 位的乘积或 p 位的商中，有效数位的个数与有效数位较少的操作数大致相同．由于这个原因，非规格化浮点运算有时被称为有效浮点运算．数 $p - z$ 反映了所得出的浮点数中有效数位的个数．

将非规格化浮点运算与规格化浮点运算进行比较，我们看到后者因为没有领前的非零位，因此在全部 p 个数位上，由于经常出现舍入而带来的舍入误差比较小．芝加哥大学研制的 MANIAC III 计算机有一个能够执行所有以下三类运算的运算器，这三类运算是指：定点运算，规格化的和非规格化的浮点运算．

非规格化运算的主要优点在于通过重新定比例而逐步下溢，以及使用了有效运算．逐步下溢允许下溢的结果用一个非规格化的数来代替．有效运算虽然会产生非规格化的结果，但它取消了算前和算后的规格化步骤．在这二类浮点运算系统之间进行折衷的分析涉及许多因素，如象：精度的等级，误差分析，运算硬件的设计和机械化，使用要求，程序设计是否容易等等．下面将对这些问题提供一些答案．

10.3 截断和舍入操作

在许多场合，浮点操作的结果可以超过 p 个数位，这里 p 为最大的尾数长度．例如，二个 p 位的规格化尾数相加，当结果超过数值 1 时，会产生 $p+1$ 个数位．另一个例子是，二个 p 位的尾数相乘预期可得 $2p$ 个数位．

在第一个例子中，我们处理"尾数和"上溢的办法是使所得到的"和"规格化，即通过把它右移一个位置以及使原来的最低有效位从右端推出，而不管它的数值如何．这样一个操作被称为**截断**．在截断运算中，结果的尾数首先被规格化，然后它的最低位被丢掉，它的前 p 个数位仍保留不变．在上述截断的情况下，我们能同样地处理正数和负数．

如果 x 是一个用基数 r 的真值表示的实分数（正的），我们将用 $(x)_p$ 来表示对 x 截取 p 个基数为 r 的数位时的大小．

$$
x = 0.130581 \quad (x)_4 = 0.1305 \quad (x)_2 = 0.13 \tag{10.11}
$$

我们将规定机器的算术运算操作⊕、⊖、⊗、⊘如下：首先执行相应的真实的算术运算操作 +、−、×、/，然后再把结果截取 p 个数位

$$
a \oplus b = (a+b)_p, \quad a \ominus b = (a-b)_p,
$$
$$
a \otimes b = (a \times b)_p, \quad a \oslash b = (a/b)_p \tag{10.12}
$$

例如，如果应用方程式 10.12 中的乘法规则，我们只能从 $2p$ 位实际乘积中截得一个

p 位的乘积，这里假定所得到的乘积已经是规格化的形式．如果所得到的乘积包括一个领前的零，如方程式 9.16 所保证的那样，那么必须左移一位才能使它规格化；于是只丢掉了 $p-1$ 个低位．

让我们回顾一下第 9.6 节中实现尾数相乘的方法，在 **Q** 寄存器中乘积的后 p 位被丢掉了．但是，每当寄存器 **Q** 中领前的位 Q_1 为"1"时，就有一个"1"进到 **A** 寄存器中的乘积的前 p 位上去．（否则就进"0"）这说明了将非有效的数位进行舍入的一种方法．在这种方法中，舍入意味着结果被取成最接近的 p 位数．这种舍入方法可以从上方或从下方趋向结果的真正数值．当被舍入的低位位置上有一个数值，它大于或等于其最大值的一半

$$\varepsilon = r^{-p}/2 \tag{10.13}$$

时，进一个"1"就会使机器表示的数从上方趋向于真正的数值，否则就从下方趋近．

对于舍入的阈值，不难依靠检查被舍入部位上的最高有效位来实现．如果该数位具有下列集合中的数值

$$\left\{\frac{r}{2},\ \frac{r}{2}+1,\ \cdots,\ r-1\right\}$$

这里假定 $r=2k$，那么通过进一个"1"到高位的位置上，便能完成舍入．如果这个数位的值在 $\{0,1,\cdots,(r/2)-1\}$ 中，那么舍入等效于截断．因此，截断可以当作是舍入的一种特殊情况，那里经常是从下方趋近于实际值的．其它舍入方案，如象不管舍入部分的数值如何，始终把一个"1"强加到高位部分，这就是经常从上方趋向实际值．

我们用 $(x)_p^*$ 代表一个被舍入成 p 个基数为 r 的数位的实数 x．用方程式 10.11 中同样的数值例子，我们有

$$x = 0.130581 \quad (x)_4^* = 0.1306 \quad (x)_2^* = 0.13 \tag{10.14}$$

注意：在这个例子中 $(x)_4^* \neq (x)_4$，但是 $(x)_2^* = (x)_2$．这意味着用不同的舍入方案预期可以获得不同的结果．用在 $(x)_4^*$ 中的阈值等于 $\varepsilon = 10^{-4}/2 = 0.000050$．因为

$$0.000081 > 0.000050,$$

所以有一个"1"被进到前 4 个十进制数位上去．

同样地，我们可以将带舍入的机器运算定义为在原有的操作之后，把结果舍入成 p 个有效数位．

$$a \oplus b = (a+b)_p^*; \quad a \ominus b = (a-b)_p^*$$
$$a \otimes b = (a \times b)_p^*; \quad a \oslash b = (a/b)_p^* \tag{10.15}$$

直观地可以看到，在设计与程序编制工作这二个方面，截断运算要比舍入运算更简单．有些计算机系统所用的浮点系统接近于上述舍入运算．这些计算机用了一个保护-数位寄存器来完成舍入运算，它的长度为 w 个数位．保护-数位寄存器保存着前 p 个有效数位右边的低位数位．例如，IBM **360** 系统用了一个保护数位 $(w=1)$，IBM **7090** 则有 27 个保护数位．这些保护寄存器的功能类似于第 9.6 节中执行 MULTIPLY（乘法）指令时所用到的 **Q** 寄存器．这些保护寄存器能用来实现误差控制线路．某些别的机器则更加灵活，如象 CDC **6600**，它利用操作码的一个子集合来执行截断运算，而另一个的子集合执行舍入运算．这就使程序员能够更灵活地去控制运算误差，但这是以消耗更多的硬设备为代价的．

10.4 公理化舍入理论

上一节描述的舍人技术只考虑尾数的绝对值。广义地说，一个好的舍入方案应该把正数和负数区别开。由于这个原因，需要用更严格的代数描述来模拟机器可能实现的各种舍入方案。我们希望这些方法能把现有的舍入方法加以归纳，并且为进一步的机器运算设计引出更有价值的方案。

令 $\mathbf{R}$ 为实数系统，$\mathbf{M}$ 为机器可表示的数的集合。显然，我们有 $\mathbf{M} \subseteq \mathbf{R}$。舍入 ρ 是对所有的 $a, b, \in \mathbf{R}$ 定义的一个映象

$$\rho: \mathbf{R} \rightarrow \mathbf{M} \tag{10.16}$$

因而

$$\rho(a) \leqslant \rho(b),\ \text{当}\ a \leqslant b\ \text{时} \tag{10.17}$$

若一个舍入对所有的 $a \in \mathbf{M}$，有

$$\rho(a) = a \tag{10.18}$$

则被称为最佳。实际上，这必须对任何合理的机器表示的数都正确。最佳舍入意味着，如果 $a \in \mathbf{R}$，以及 m_1，m_2 为 $\mathbf{M}$ 的二个相继的数，并且 $m_1 < a < m_2$，于是 $\rho(a) = m_1$ 或 $\rho(a) = m_2$。这就要求经常有硬件实现最佳舍入。

如果对所有的 $a \in \mathbf{R}$，舍入是向下或向上的，那么我们分别有

$$\rho(a) \leqslant a \tag{10.19}$$

或

$$\rho(a) \geqslant a \tag{10.20}$$

如果对所有的 $a \in \mathbf{R}$，舍入是对称的，那么

$$-\rho(-a) = \rho(a) \tag{10.21}$$

最佳的向下舍入 $\nabla: \mathbf{R} \rightarrow \mathbf{M}$ 定义为

$$\nabla(a) = \mathrm{Max}\{m \in \mathbf{M} \mid m \leqslant a\} \tag{10.22}$$

最佳的向上舍入 $\Delta: \mathbf{R} \rightarrow \mathbf{M}$ 定义为

$$\Delta(a) = \mathrm{Min}\{m \in \mathbf{M} \mid m \geqslant a\} \tag{10.23}$$

上述定义意味着 Δ 是这样一种舍入方法，当数为正时远离零，当数为负时趋向零。反之 ∇ 是一种正好相反的方法。这二种舍入 Δ 和 ∇ 已经被用在区间算术运算装置中。

对我们感兴趣的三种对称舍入是：截断 $\mathbf{T}$，增量 $\mathbf{A}$ 和近似 $\mathbf{P}$。这些舍入正式定义如下，对所有的 $a \in \mathbf{R}$，依据上述映象 Δ 和 ∇ 有

$$\mathrm{T}(a) = \begin{cases} \nabla(a), & \text{如果}\ a \geqslant 0 \\ \Delta(a), & \text{如果}\ a < 0 \end{cases} \tag{10.24}$$

$$\mathrm{A}(a) = \begin{cases} \Delta(a), & \text{如果}\ a \geqslant 0 \\ \nabla(a), & \text{如果}\ a < 0 \end{cases} \tag{10.25}$$

$$\mathbf{P}(a) = \begin{cases} \nabla(a), & \text{如果}\ \nabla(a) \leqslant a < \dfrac{\nabla(a) + \Delta(a)}{2} \\ \Delta(a), & \text{如果}\ \dfrac{\nabla(a) + \Delta(a)}{2} \leqslant a < \Delta(a) \end{cases} \tag{10.26}$$

截断舍入 **T** 始终舍向零，增量舍入 **A** 始终舍离零. 近似舍入 **P** 始终选最接近的机器数，在这种情况下需要把向上和向下舍入结合起来选择数值较接近的一个. 近似舍入 **P** 是最常用的，因为它提供了最好的精度. 现有的机器常常采用 **T** 或某些近似的 **P** 方案. 对 **P** 的一个普通的近似是在算后规格化移位之前进行舍入.

表 10.1 中给出了上述五种舍入方式 **Δ**，**∇**，**T**，**A** 和 **P** 的一览表. 显然，**P** 舍入在增加硬件费用的情况下提供了最有效的选择. 第 10.3 节中描述的截断和舍入方案分别类似于 **T** 舍入和 **P** 舍入. 一般地说，**P** 方式优于 **T** 方式. 但是，这两种情况在经过一系列计算后，误差生长的速率基本相同. 因此，可以预期舍入趋向于延缓误差的生长.

Yohe[28] 建议每一台计算机应能在程序控制下执行表 10.1 中列出的五种舍入方式. 一个浮点运算系统的设计者必须确定舍入对计算速度和精度的影响. 最近 Garner[9] 断言，用小基数值和伪舍入算法能获得较好的精度，而用大基数值和粗略的舍入步骤（如象截断）则可提高计算速度. 一个好的舍入方案能大大提高浮点处理器的精度性能，但它也会使整个系统的速度降低.

表 10.1　五种舍入方案以及它们对具有 p 个分数数位的浮点运算的含义

舍入方式	操作符号	数值含义	
		正数	负数
最佳向上	**Δ**	舍离零	舍向零
最佳向下	**∇**	舍向零	舍离零
截　断	**T**	舍向零	舍向零
增　量	**A**	舍离零	舍离零
近　似	**P**	π = 结果的绝对值，θ = 第 $(p+1)$ 位分数数位	
$\pi \geq \min$ 和 $\theta < r/2$		舍向零	
$\pi \geq \min$ 和 $\theta \geq r/2$		舍离零	
$\pi < \frac{1}{2}\min$		舍向零	
$\frac{1}{2}\min \leq \pi < \min$		舍离零	
阶上溢		舍离零	

10.5　浮点运算的误差分析

浮点运算的误差分析已经证明是比较复杂的. 对浮点误差的特征没有很好的了解，就不可能实现在精度和速度上具有最佳性能的设计. 这一节我们将描述近几年提出的一些误差分析模型. 特别是，我们选择三个特定的浮点表示法 $(r, q, p) = (2, 9, 22)$；$(4, 8, 23)$ 和 $(16, 7, 24)$ 来介绍表示法的误差特性. 这些参数的选择对一个 32 位的字来说，所给出的数的表示范围基本上与流行的机器中大多数单精度浮点运算所遇到的相同. 对这三个浮点系统所作的比较研究，不难加以修改后再去分析其他系统.

表示法的误差是由于机器可表示的数的集合 **M** 只是实数集合 **R** 的一个子集. 一个数的每一种机器表示法 $m \in \mathbf{M}$ 可以看作实数的一个等价类

$$m = \{x \mid x \in R \text{ 和 } x \equiv m\} \tag{10.27}$$

这里 $x \equiv m$，表示在机器精度以内的等价关系. 这种**表示法误差**是用 m 表示这个等价类

的一个元素 x 时造成的. 对一个 (r, q, p) 浮点系统，邻近的规格化浮点数之间的间距等于

$$2^{-p} \times r^{e} \tag{10.28}$$

式中 e 为未偏置的阶的数值. 尤其是,当 $r = 2^k$ 时,我们有

$$2^{-p} \times 2^{ke} = 2^{ke-p} \tag{10.29}$$

表示法误差的大小等于上述间距值的一部分,因而是这个数的数值的函数.

$m \in \mathbf{M}$ 与它的等效 $x \in \mathbf{R}$ 的相对误差 δ 定义为表示法误差 $x - m$ 被 x 除的数值

$$\boldsymbol{\delta} = \frac{x - m}{x} \tag{10.30}$$

如果我们假定规格化数在 $(1/r) \leqslant x < 1$ 的范围内是按对数概率分布的

$$P(x) = \frac{1}{x \times \ln r} \tag{10.31}$$

以及通过适当舍入得到均匀分布的最小误差为

$$Q(x) = \frac{2^{-p}}{4x} \tag{10.32}$$

那么,**平均相对表示法误差** (**ARRE**) 定义为

$$\mathbf{ARRE}(p, r) = \int_{1/r}^{1} P(x) \times Q(x) dx = \int_{1/r}^{1} \frac{2^{-p} dx}{4x^2 \times \ln r} = \frac{(r-1)}{4 \ln r} \times 2^{p} \tag{10.33}$$

对于所有规格化尾数的**最大相对表示法误差** (**MRRE**) 定义为

$$\mathbf{MRRE}(p, r) = 2^{-p-1} \times r \tag{10.34}$$

表 10.2 中给出了三个 32 位浮点系统的 **ARRE** 和 **MRRE** 的数值. 二进制表示法的 **MRRE** 是相应的十六进制表示法的一半,这一点已被用于论证二进制优于十六进制. 然而这种优越性并不是很显著的,如果再考虑 **ARRE** 的数值,那就更是这样. 这个表格

表 10.2　三个 32 位系统的浮点数的统计特性，这三个系统具有基数 r，以及 p 个分数数位和 q 个阶数位[1](Cody[6])

r	q	p	MRRE	ARRE	阶的范围	数的范围
2	9	22	0.5×2^{-21}	0.18×2^{-21}	$2^{2^9} - 1$	$2^{2^9} \times (1 - 2^{-22})$
4	8	23	0.5×2^{-21}	0.14×2^{-21}	$2^{2^9} - 2$	$2^{2^9} \times (1 - 2^{-23})$
16	7	24	1×2^{-21}	0.17×2^{-21}	$2^{2^9} - 4$	$2^{2^9} \times (1 - 2^{-24})$

1) $p + q + 1 = 32$，其中符号占一位.

指出了在这三个浮点系统中间，基数 4 的系统产生的误差最小. 因此，一个数的四进制表示法并不比相应的二进制表示法不精确，它的 **ARRE** 的数值要比相应的二进制方案小 20%.

Brent[3] 使用**均方根** (rms)来对范围几乎相同、但具有不同基数 $r = 2^k$ 的浮点系统的相对误差进行比较. 对于对数系统，Brent 曾导出下述 rms 相对误差 ($\mathbf{RE}_{rms}$) 函数,后者定义为标准偏差 $\boldsymbol{\delta}_{rms}$ 与相对误差 $\boldsymbol{\delta}$ 的平均值 $\boldsymbol{\delta}_0$ 的比值. $\mathbf{RE}_{rms}$ 仅仅表示为基数 $r = 2^k$ 的幂 $k = \log_2 r$ 的函数,只考虑具有一个显式领前尾数位“1”的显式规格化浮点数时

$$\mathbf{RE}_{rms} = \frac{\boldsymbol{\delta}_{rms}}{\boldsymbol{\delta}_0} = \sqrt{\frac{4^k - 1}{2(k \log 2)^3}} \tag{10.35}$$

表 10.3 具有不同基数 $r=2^k$ 的浮点系统的 RMS 相对误差（Brent[13]）

基数 r	第一个分数位	$RE_{rms}(k)$[1]	基数 r	第一个分数位	$RE_{rms}(k)$[1]
2	隐	1.06	32	显	3.51
2	显	2.12	64	显	5.34
4	显	1.68	128	显	8.47
8	显	1.87	256	显	13.90
16	显	2.45			

1) $RE_{rms}(k)$ 由方程式 10.35 定义，对于一个显式规格化系统只等于具有隐式最高有效位的规格化浮点系统的一半.

表 10.3 列出了 $k=1,2,\cdots,8$（相对于基数 $r=2,4,\cdots,256$）时，RE_{rms} 的数值. 这个表格指出：基数 4（$r=4$ 和 $k=2$）是最好的. 在最坏情况的研究中，基数 2 和基数 4 的系统是同样好的. 然而，在均方根的研究中，基数 4 肯定是比较好的. 由于基数 4 和基数 8 可能有不同的范围，有一些最小可接受范围的选择，对于这些范围基数 8 略优于基数 4，但是大于 8 的基数在 RE_{rms} 判据上经常劣于基数 4. 某些计算机系统，如象 **PDP-11**，它采用了隐式规格化二进制浮点系统，其中尾数长度增加一位，领前的分数"1"即被丢掉. Brent 已经证明，对这样的系统，方程式 10.34 中求出的结果应该减半. 这说明隐式二进制系统超过上述所有显式系统.

10.6 关于浮点算术运算操作中的误差边界

有效数位的个数可以粗略度量一个数所包含的相对误差. 领前的零和结尾的零并不传递有效信息. 自从数字计算机中普遍接受浮点算术运算以来，计算过程中的误差传播问题便愈来愈显得重要. 不同的舍入方案可能引起不同的精度等级. 前几节中说明并实现的算术运算操作都应该补上适当的误差控制机构. 对误差来源和误差边界有较好的了解，会有助于设计有效的误差控制方案.

在浮点计算中产生的误差会通过一系列操作不断传播下去. 如果不加适当的控制，误差将会累积到使最后结果成为毫无意义的程度（象这样的偏离往往与所要求的精度相差太远）. 误差控制机构严重地依赖于所用算术运算系统的类型. 与规格化浮点系统有关的设计重点不同于有效浮点设计中的那些重点. 此外，舍入也造成差异. 下面我们引出与浮点算术运算有关的误差分析方法. 我们用误差边界来估算与所加的舍入有关的最大误差. 为了降低复杂性，这里只给出规格化算术运算的分析. 但这种方法有可能推广到去分析其他类型的浮点算术运算系统.

第 10.5 节中用到的表示法将继续使用. 令 $\mathscr{E}_a$@为机器操作@在情况 a 遇到的绝对误差，机器结果与固有操作 α 的结果相比较时，后者由于没有施加截断或舍入而没有误差. 在我们介绍带舍入的机器操作之前，有几个参数必须定义一下.

根据方程式 10.13 中所规定的 ε，一个小分数 η 可定义如下：

$$-\varepsilon \leqslant \eta < \varepsilon \tag{10.36}$$

这里 ε 反映了第 $(p+1)$ 位中最大的舍入误差. 我们将用不同的下标来区别不同的分数 η_i，这些分数均满足方程式 10.36.

令 σ 为 p 个数位的基数为 r 的分数，$0 \leqslant |\sigma| < 1$，而且 $\lfloor \log_r|\sigma| \rfloor$ 是小于或等于实数

$\log_r|\sigma|$ 的最大整数，可以看出

$$-p \leqslant \lfloor \log_r|\sigma| \rfloor < 0$$

或

$$r^{-p} \leqslant r^{\lfloor \log_r|\sigma| \rfloor} < 1 \tag{10.37}$$

加法与减法

我们从规格化浮点加法 $N_1 \oplus N_2$ 开始分析．机器减法 $N_1 \ominus N_2$ 可当作加法的特殊情况来处理．假定 $N_1 = (m_1, e_1)$，$N_2 = (m_2, e_2)$，以及 $N_3 = N_1 \oplus N_2 = (m_3, e_3)$，这里 $e_2 > e_1$.

情况 ***A*** **在尾数加法以后没有溢出**．带舍入的机器操作⊕产生以下结果

$$m_3 = \sigma \times r^{-\lfloor \log_r|\sigma| \rfloor - 1} \tag{10.38}$$

$$e_3 = e_2 + \lfloor \log_r|\sigma| \rfloor + 1 \tag{10.39}$$

这里

$$\sigma = (m_1 \times r^{e_1 - e_2})_p^* + m_2 = m_1 \times r^{e_1 - e_2} + \eta + m_2 \tag{10.40}$$

从上面的机器结果中减去方程式 9.9 所确定的真正的加法 + 的结果，我们得到下列误差函数

$$\begin{aligned}
\mathscr{E}_a \oplus &= |(N_1 \oplus N_2) - |(N_1 + N_2)| = |(m_1 r^{e_1 - e_2} + m_2 + \eta) \\
&\quad \times r^{-\lfloor \log_r|\sigma| \rfloor - 1} \times r^{e_2 + \lfloor \log_r|\sigma| \rfloor + 1} - (m_1 r^{e_1 - e_2} + m_2) \times r^{e_2}| \\
&= |\eta \times r^{e_2}| \leqslant \varepsilon \times r^{e_2} = \varepsilon \times r^{e_3 - \lfloor \log_r|\sigma| \rfloor - 1} \\
&\leqslant \varepsilon \times r^{e_3 + p - k}
\end{aligned} \tag{10.41}$$

当 σ 的数值为零时，p 的数值即用来代替 e_3 中的 $\lfloor \log_r|\sigma| \rfloor$，这意味着没有舍入误差．

情况 ***B*** **在尾数加法以后有上溢**．机器结果 (m_3, e_3) 可以求之如下

$$\begin{aligned}
m_3 &= (m_1 \times r^{e_1 - e_2 - 1})_p^* + (m_2 \times r^{-1})_p^* \\
&= (m_1 + \eta_1) \times r^{e_1 - e_2 - 1} + (m_2 + \eta_2) \times r^{-1}
\end{aligned} \tag{10.42}$$

以及

$$d_3 = e_2 + 1 \tag{10.43}$$

从上式减去方程式 9.11 中固有的结果，我们得到

$$\begin{aligned}
\mathscr{E}_b \oplus &= |(N_1 \oplus N_2) - (N_1 + N_2)| \\
&= |((m_1 + \eta_1) \times r^{e_1 - e_2 - 1} + (m_2 + \eta_2) \times r^{-1}) \times r^{e_2 + 1} \\
&\quad - (m_1 \times r^{e_1 - e_2 - 1} + m_2 \times r^{-1}) \times r^{e_2 + 1}| \\
&= |(\eta_1 \times r^{e_1 - e_2 - 1} + \eta_2 \times r^{-1}) \times r^{e_2 + 1}| \\
&\leqslant \varepsilon r^{-1} \times (r^{e_1 - e_2} + 1) \times r^{e_2 + 1} \leqslant \varepsilon r^{e_1} + \varepsilon r^{e_2} \\
&\leqslant 2\varepsilon r^{e} = 2\varepsilon r^{e_3 - 1}
\end{aligned} \tag{10.44}$$

上述误差边界是用已知值 e_3 来表示的，它可以在机器操作⊕或⊖之后使用．以上二个误差边界均满足下列关于加法误差的一般边界

$$\mathscr{E} \oplus = |(N_1 \oplus N_2) - (N_1 + N_2)| < \frac{r^{e_3 - 1}}{2} \tag{10.45}$$

乘法

假设由 ***A*** 和 ***Q*** 寄存器级联形成的双倍长度的累加器可供尾数乘法使用．今考虑

$N_3 = N_1 \otimes N_2 = (m_3, e_3)$. 对 $\sigma = m_1 \times m_2$，我们有

$$m_3 = (\sigma \times r^{-\lfloor\log_r|\sigma|\rfloor-1})^*_p = (m_1 \times m_2 \times r^{-\lfloor\log_r|\sigma|\rfloor-1} + \eta) \tag{10.46}$$

$$e_3 = e_1 + e_2 + \lfloor\log_r|\sigma|\rfloor + 1 \tag{10.47}$$

从方程式 9.12 和上列方程式可得乘法的误差函数如下

$$\begin{aligned}\mathscr{E}\otimes &= |N_1\otimes N_2 - N_1 \times N_2| = |m_1 - m_2 \times r^{-\lfloor\log_r|\sigma|\rfloor-1} + \eta) \\ &\times r^{e_1+e_2+\lfloor\log_r|\sigma|\rfloor+1} - (m_1 \times m_2) \times r^{e_1+e_2}| \\ &\leqslant \varepsilon \times r^{e_1+e_2+\lfloor\log_r|\sigma|\rfloor+1} = \varepsilon \times r^{e_3}\end{aligned} \tag{10.48}$$

我们已经用了特性

$$-2 \leqslant \lfloor\log_r|\sigma|\rfloor \leqslant -1 \tag{10.49}$$

这是由于方程式 9.14（如果 m_1 和 m_2 均不为零）。在 m_1 或 m_2 或二者均为零的情况下，就没有舍入误差。因此，乘法误差可以根据已知结果来确定边界。如果 $\sigma = m_1 \times m_2$ 的数值为零，$\lfloor\log|\sigma|\rfloor$ 的数值再次用 e_3 中的 $-p$ 代替。

除法

为了避免上溢，假定规格化被除数 N_1 和除数 N_2 已适当调整，因此在产生规格化的商时，没有算后移位操作。令机器除法为 $N_3 = N_1 \oslash N_2$，并带有舍入。我们有

$$\begin{aligned}m_3 &= \left(\frac{m_1}{m_2}\right)^*_p \times r^{-\lfloor\log_r|\sigma|\rfloor-1} = \left(\frac{m_1}{m_2} + \eta\right) \times r^{-\lfloor\log_r|\sigma|\rfloor-1} \\ &= \left(\frac{m_1}{m_2} + \eta\right) \times r^{+1-1} = \frac{m_1}{m_2} + \eta\end{aligned} \tag{10.50}$$

$$e_3 = e_1 - e_2 + \lfloor\log_r|\sigma|\rfloor + 1 = e_1 - e_2 \tag{10.51}$$

表 10.4　规格化浮点操作误差边界一览（Carr[21]）

浮点操作	算术运算误差函数	误差函数上界[1)]
加法 或 减法	$\lvert(N_1\oplus N_2) - (N_1 + N_2)\rvert$ $\lvert(N_1\ominus N_2) - (N_1 - N_2)\rvert$	情况 A：无尾数上溢 $\varepsilon \times r^{e_3+p-1}$ 情况 B：尾数上溢 $\varepsilon \times r^{e_3-1}$
乘法	$\lvert(N_1\otimes N_2) - (N_1 \times N_2)\rvert$	$\varepsilon \times r^{e_3}$
除法	$\lvert(N_1\oslash N_2) - (N_1/N_2)\rvert$	$2\varepsilon \times r^{e_3}$

1) r = 阶的基数．　p = 允许尾数长度．　e = 和、差、积、商的结果中阶的数值．　ε = 超出允许尾数长度的第 $(p+1)$ 个数位中的最大舍入误差．

这里 $\frac{1}{2} \leqslant |\sigma| = |m_1/m_2| < 1$，因此 $\lfloor\log_r|\sigma|\rfloor = -1$ 和 $-\varepsilon \leqslant \eta < \varepsilon$．我们有误差函数

$$\begin{aligned}\mathscr{E}\oslash &= |(N_1\oslash N_2) - (N_1/N_2)| \\ &= \left|\left(\frac{m_1}{m_2} + \eta\right) \times r^{e_1-e_2} - \left(\frac{m_1}{m_2}\right) \times r^{e_1-e_2}\right| \\ &\leqslant |\eta| \times r^{e_1-e_2} \leqslant \varepsilon \times r^{e_1-e_2} = \varepsilon \times r^{e_3}\end{aligned} \tag{10.52}$$

在表 10.4 中给出了上述误差边界的一览表．这些边界均以基数 r、尾数长度 $p = \log_r 2\varepsilon$ 以及结果的阶 e_3 的函数来表示．

10.7 一个以 ROM 为基础的舍入方案

这一节我们研究一种最近提出的舍入方案，这种方案不难用 ROM 或 PLA 来实现．普通的以加法为基础的舍入方案涉及到用加法器硬件来处理进位传播，后者是由于"1"进到被舍入浮点数的低位部分引起的．这就造成实现舍入操作时附加的时间延迟．现在要描述的 ROM 舍入并不需要增加与加法器级有关的时间和硬件．这个方法从价格-效率和误差的抑制二个观点来看都是引人注目的．

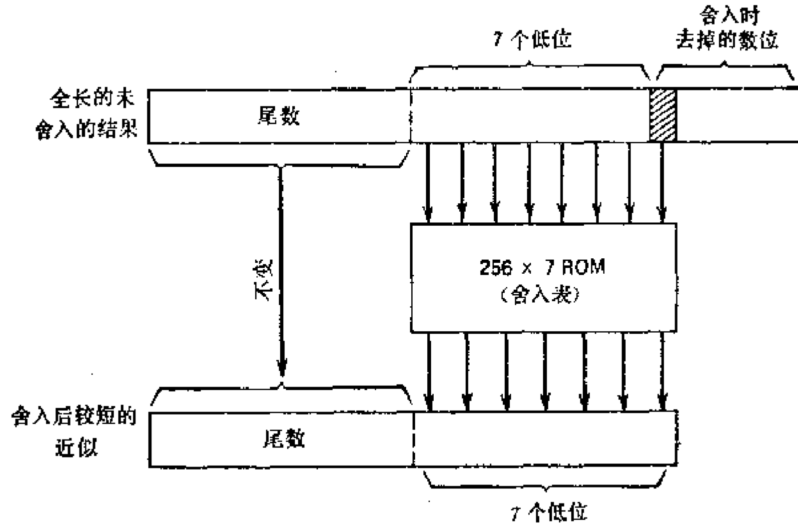


图 10.1 以 ROM 为基础的舍入方案的概念 (Kuck 等[16])

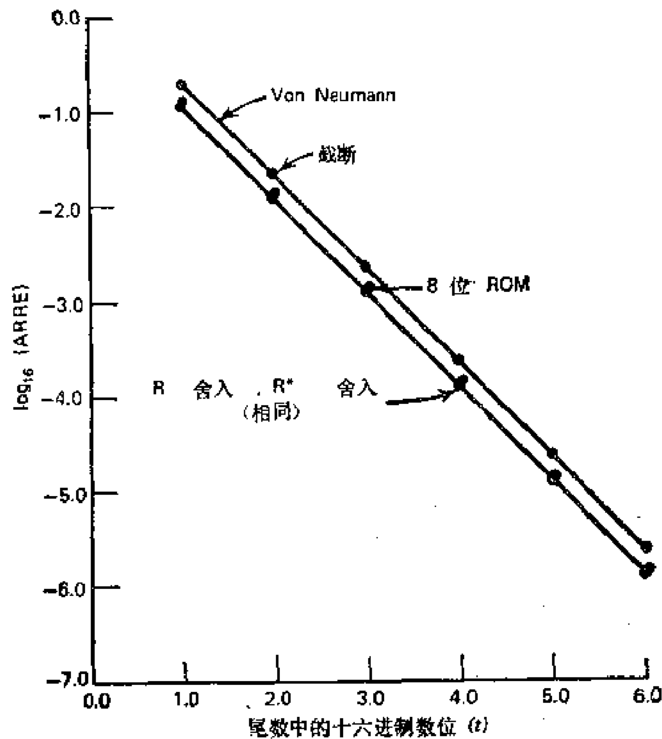


图 10.2 各种舍入方案的平均相对表示法误差 (ARRE)

ROM 表的低位部分的舍入在 6.10 节中已作了讨论，它与廉价的以 ROM 为基础的乘法网络有关．下面将说明为了把较长的浮点数简化为较短的精确近似值，这里用 ROM 或 PLA 来实现舍入的机理．这个方案说明在图 10.1 中．长的表示式相当于不用任何舍入的浮点计算中全长的中间或最后结果．较短的近似则是作了适当舍入步骤后，机器所接受的表示式．假定所用的 ROM 有 8 条地址线和 7 条输出线，存储容量为 $2^8 \times 7$ 位．尾数有效部分的 7 位低位与被舍入的最高位一起，形成 8 位地址输入到 ROM 或 PLA 去．ROM 的输出是这些有效位的 7 位舍入值．

舍入规则用微程序编制在 ROM 中，它包含必要的加法操作．在舍入上溢的情况下，即所有 7 个较低的有效位均为1，这时不执行舍入，而是简单地截断．其余 255 种情况（对 8 位的线路有 256 个输出），则通过 ROM 和 PLA 来实现适当的加法操作，从而真正执行舍入．用近代的存储器工艺来实现这种方案比起用硬接线的舍入逻辑电路更便宜．因此，从设计者的观点来看，ROM 舍入对于在浮点处理器中引进舍入运算是比较有效的．这种新方案的误差抑制能力也得到了证实．

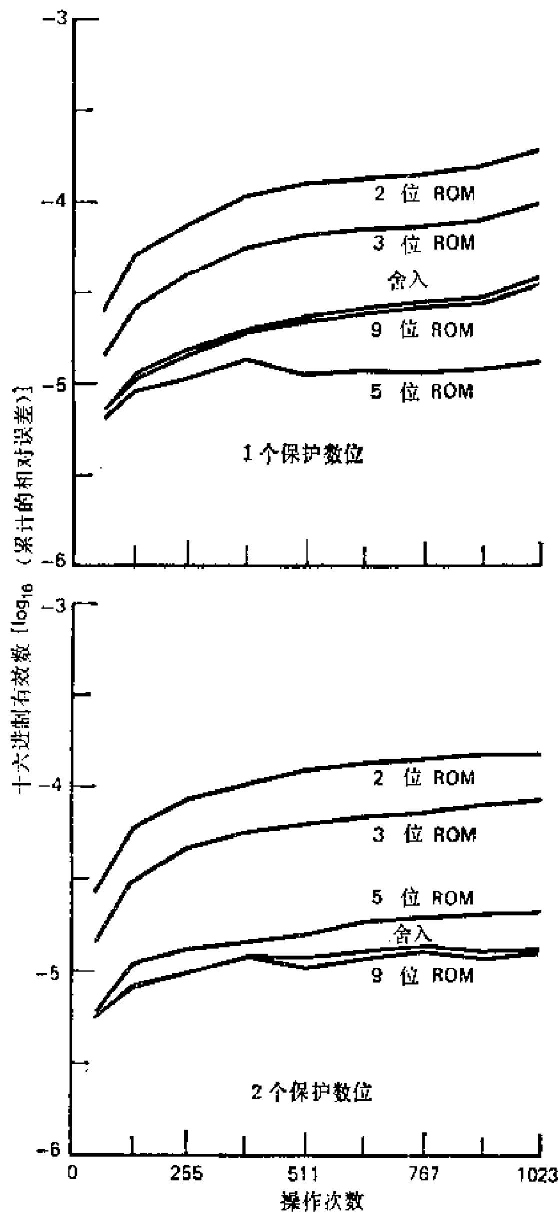


图 10.3　ROM 长度对误差生成的影响 (Kuck 等[16])

舍入方案有效性的二个指标被用来阐明 ROM 舍入应用到现有几种舍入方法时的误差抑制能力，它类似于前几节说明的截断 **T**（强制截断），近似 **P**（最近的邻域）以及增量 **A**（远离零）．这二个有效性指标是指：

1. **ARRE**（平均相对表示法误差）用方程式 10.33 来定义．

2. 具有特定问题类型的统计测试，如象一串加法、减法、乘法等．

使用保护数位来保留部分被调整的操作数（它在阶的比较期间右移）的效果在于给出精度较高的最后结果．保护数位的方法取决于伴随的调整方法．所谓调整，我们指的是舍入方法的使用简化了被调整的操作数，使之在执行加法或减法之前能适应机器字和保护数位．

五种现有的舍入方案的 **ARRE** 已说明在图 10.2 中，其中对不同尾数长度均假定基数为 16 ($r = 16$)．Von Neumann 舍入除了结果尾数的最低位强置为“1”以外，等效于截断．**R** 舍入广义地说是近似舍入，如第 10.3 节所述．**R***

舍入等效于 **R** 舍入，但当被舍入的数位具有二进位数值 1000，···时除外，这时要用 Von Neumann 舍入．曲线指出：就**ARRE**来说，8 位 ROM 舍入只是比正规的舍入 **R**（或 **R***）稍差一些（这是合理的，它不仅因为二者几乎在 256 个中有 255 个是相同的，而且也因为它们相区别的一个情况具有最低的相对误差）．图 10.3 指出，在具有一个或二个十六进制保护数位的混合加法型操作中，对于不同长度的 ROM 的累计相对误差．

具有各种基数和舍入方案的浮点系统，常常依靠把不同的数据集合应用到典型问题，以及统计地测量总的舍入误差来加以鉴定．图 10.4 中仅给出与重复加法型操作有关的结果．图中的曲线指出了对一个或二个保护数位的系统，上述五种舍入方案的误差的生长情况．初始拐点是由于十六进制算后规格化移位时，有效位丢失而引起的．对大多数方法，由于和的尾数的数值比累计的舍入误差增加得更快，因而相对误差实际上是减小

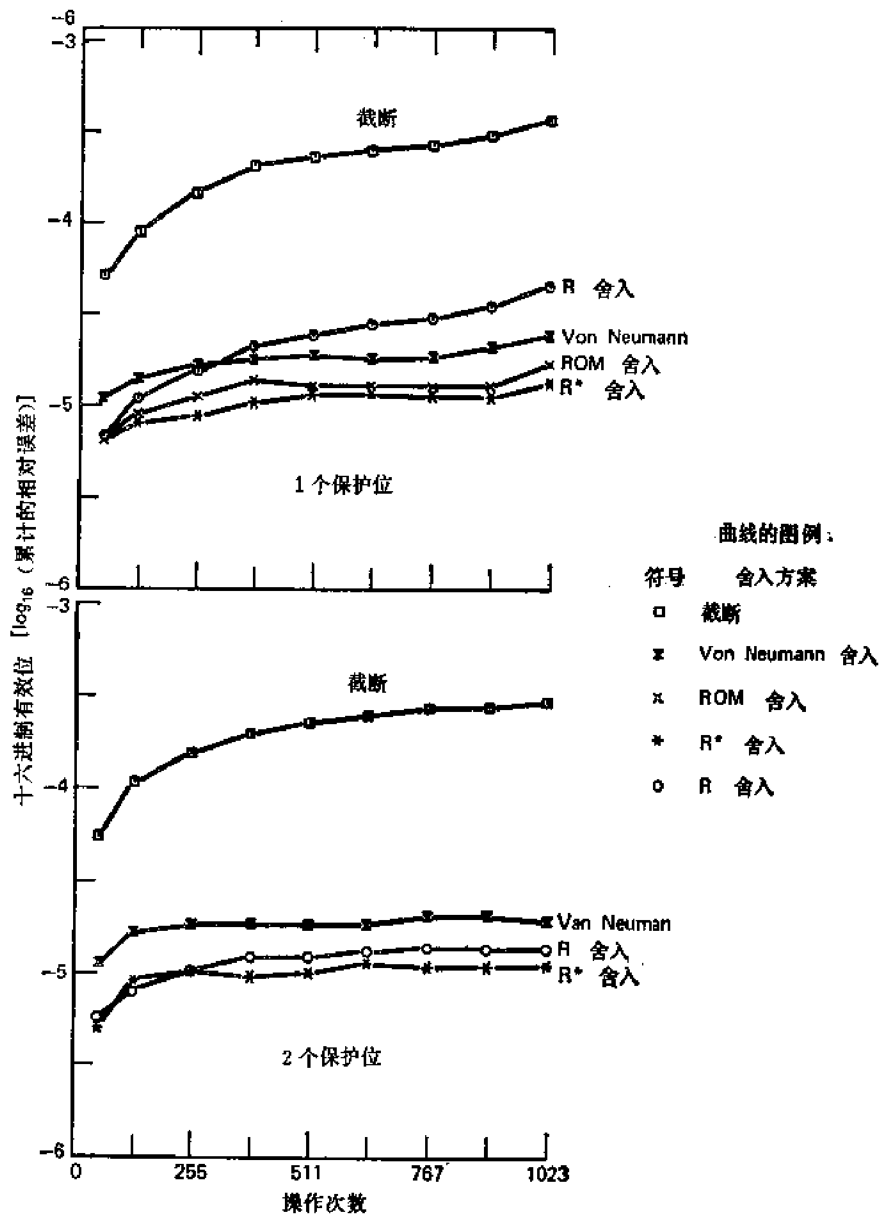


图 10.4 一连串加法型操作的误差生长（混合的加法和求补减法）（Kuck 等[143]）

的，但是围绕着 256 个操作，最后要出现进位输出；尾数的规格化引起一个领前的数位"1"加到丢失的低位上，同时使相对误差的结果中带来一个突变. 保护数位用得较少，舍入的调整将更加有效.

10.8 浮点指令的比较

这一节介绍对浮点指令的集合论研究. 这种研究为学术研究者和先进的设计师提供了有用的数学工具. 他们可以利用这种工具来对浮点操作加以形式定义、分析、比较和评价. 对于三类浮点算法（加法/减法，乘法和除法）中的每一类，我们都将使用比较集合的新概念来划分浮点操作数对的空间. 这个理论对于预示大多数浮点操作中规格化、截断和舍入的效果是特别有效的.

在这个研究中，浮点数 F 定义为三元函数

$$F(\sigma F, \varepsilon F, \phi F) \tag{10.53}$$

式中 $\sigma F \in \{+, -\}$ 为符号，εF 为整数的阶，ϕF 为 F 的分数（尾数）. 分数 ϕF 用一个有限的数字基数（LDR）多项式来模拟，它具有基数 $r \geqslant 2$ 以及正的模 r 的系数. 分数长度 p 假定为 $p \geqslant 3$；$\tilde{\phi}F$ 代表分数 ϕF 的数值. 零分数表示为 0，实际算术运算操作 +、-、*、/以及机器算术运算操作⊕、⊖、⊛、⊘的表示法和前面相同.

基数 r 的多项式通常定义为

$$P = \sum_{i=-\infty}^{\infty} p_i \times r^{-i} \tag{10.54}$$

式中系数 p_i 对所有的 i 均为整数（无论正或负）. 当 P 具有有限个非零系数时，我们称之为有限多项式. 当有限多项式有一个非负的数值 $\tilde{P} \geqslant 0$ 时，我们称它为有限正多项式. 此外，当一个有限正多项式它的所有系数均为正并在 $\{0, 1, \cdots, r-1\}$ 的范围内时，我们称它为 LDR 多项式. 在形式上，一个 LDR 多项式 Q 被写成

$$Q = \sum_{i=-s}^{t} q_i \times r^{-i} \tag{10.55}$$

这里 $q_i \in \{0, 1, \cdots, r-1\}$，$s+t+1$ 为多项式长度. 实际上，每一个有限正多项式 P 可以经过下列映象转换成一个 LDR 多项式 Q

$$\lambda: P \longmapsto \lambda P = Q \tag{10.56}$$

该映象定义如下：定义进位

$$C_i = 0,\ \text{对所有的}\ i \geqslant t+1. \tag{10.57}$$

$$C_i = \begin{cases} 0, & \text{如果}\ p_{i+1} + c_{i+1} < r; \\ 1, & \text{如果}\ p_{i+1} + c_{i+1} \geqslant r, \quad \text{对所有的}\ -s \leqslant i \leqslant t, \end{cases} \tag{10.58}$$

于是，对所有的 $-s \leqslant i \leqslant t$，求得 LDR 系数为

$$q_i = (p_i + c_i)\ \text{模}\ r \tag{10.59}$$

在具有正的分数值、长度为 p 的 LDR 多项式中，我们有下标 $s = -1$ 和 $t = p$，这样

$$Q = \sum_{i=1}^{p} q_i \times r^{-i} \tag{10.60}$$

的确小于 1.

令 S_r^p 为基数 r 的所有浮点数的集合，$\hat{S}_r^p \subset S_r^p$ 为所有规格化浮点数的集合，二者的分数长度均为 p.符号 $\zeta_k F$ 表示通过把阶增大 k 同时把分数乘以 r^{-k}（右移）来对 F 定比例.考虑任意二个浮点数 $G, H \in S_r^p$. 用 $e = \varepsilon G - \varepsilon H$ 表示它们的阶的差. 在我们按照映象

$$G \circ H: S_r^p \times S_r^p \to S_r^p \tag{10.61}$$

定义浮点操作之前需要有二个规定. 我们定义

$$\begin{aligned} G \geqslant H &= \varepsilon G > \varepsilon H \text{ 或 } (\varepsilon G = \varepsilon H \text{ 和 } \tilde{\phi} G \geqslant \tilde{\phi} H) \\ G < H &= \varepsilon G < \varepsilon H \text{ 或 } (\varepsilon G = \varepsilon H \text{ 和 } \tilde{\phi} G < \tilde{\phi} H) \end{aligned} \tag{10.62}$$

浮点加法

$$G \oplus H = \begin{cases} (\sigma G, \varepsilon G, \lambda(\phi G + \phi H^* r^{-e})); & G \geqslant H, \ \sigma G = \sigma H \\ (\sigma G, \varepsilon G, \lambda(\phi G - \phi H^* r^{-e})); & G \geqslant H, \ \sigma G \neq \sigma H \\ (\sigma H, \varepsilon H, \lambda(\phi H + \phi G^* r^{e})); & G < H, \ \sigma G = \sigma H \\ (\sigma H, \varepsilon H, \lambda(\phi H - \phi G^* r^{e})); & G < H, \ \sigma G \neq \sigma H \end{cases} \tag{10.63}$$

浮点减法

$$G \ominus H = G \oplus (-1 \cdot \sigma H, \varepsilon H, \phi H) \tag{10.64}$$

浮点乘法

$$G \circledast H = (\sigma G \cdot \sigma H, \ \varepsilon G + \varepsilon H, \ \lambda(\phi G * \phi H)) \tag{10.65}$$

浮点除法

$$G \oslash H = (\sigma G \cdot \sigma H, \varepsilon G - \varepsilon H, \lambda(\phi G / \phi H)) \tag{10.66}$$

这里假定了 $H \neq 0$.

上述操作是对非规格化数 $S_r^p \times S_r^p$ 定义的. 下面,我们将定义在操作数对的规格化空间上的操作为

$$S = \hat{S}_r^p \times \hat{S}_r^p \tag{10.67}$$

浮点算术运算中的规格化、截断和舍人均依赖于规格化以前的中间结果的分数的大小分数的值可以大于或等于 1. 于是，可以认为，当它大于 1 时，中间结果有进位，当分数的数值小于 $1/r$ 时，中间结果有删除. 为了描述上述进位和删除条件，下面定义二个谓语.

进位谓语

$$F \in S_r^p \text{ 和 } F \neq 0; \ \tilde{\phi} F \geqslant 1 \tag{10.68}$$

删除谓语

$$F \in S_r^p \text{ 和 } F \neq 0; \ \tilde{\phi} F < \frac{1}{r} \tag{10.69}$$

“操作数对” S 的空间可分成若干子集合，它们分别相应于不同的浮点操作. 这些子集合被称为比较集合. 每一操作种类能用这些比较集合的一个集合体来描述. 在每个集合体中，这些子集合是互异的. 这些比较是基于在转换映象 λ 中引起的进位-借位项.

对于浮点加法算法的比较集合

将进位/删除谓语应用到理想浮点加法算法的中间结果 $G \oplus \zeta_e H$ 上，可以用来划分空

间S．这些谓语缩写如下：分数值 $(G\oplus\zeta_c H)$ 写成 ω^+．于是所用到的谓语为：

$$
\begin{aligned}
&\omega^+ \geqslant 1 \text{ 指 } G\oplus\zeta_c H \text{ 有进位} \\
&\omega^+ < 1 \text{ 指 } G\oplus\zeta_c H \text{ 没有进位} \\
&\omega^+ \geqslant \frac{1}{r} \text{ 指 } G\oplus\zeta_c H \text{ 没有删除} \\
&\omega^+ < \frac{1}{r} \text{ 指 } G\oplus\zeta_c H \text{ 有删除}
\end{aligned}
\tag{10.70}
$$

这些比较集合的并集定义如下：它一般只包括那些 $G \geqslant H$ 的操作数对，这是因为$\oplus$操作在G和H中是对称的．定义 $\hat{S} \subset S$ 为

$$\hat{S} = \{(G, H) | (G, H) \in S \text{ 和 } G \geqslant H\} \tag{10.71}$$

在下列定义中，我们考虑操作数对$(G, H) \in \hat{S}$．浮点加法的比较集合为

$$
\begin{aligned}
S_0 &= \{(G, H);\ G = 0\} \\
S_1 &= \{(G, H);\ G \neq 0 \text{ 和 } \sigma G = \sigma H \text{ 和 } \varepsilon G = \varepsilon H\} \\
S_{2a} &= \{(G, H);\ G \neq 0 \text{ 和 } \sigma G = \sigma H \text{ 和 } 1 \leqslant \varepsilon G - \varepsilon H \leqslant p \text{ 和 } \omega^+ < 1\} \\
S_{2b} &= \{(G, H);\ G \neq 0 \text{ 和 } \sigma G = \sigma H \text{ 和 } \varepsilon G - \varepsilon H \geqslant p + 1\} \\
S_3 &= \{(G, H);\ G \neq 0 \text{ 和 } \sigma G = \sigma H \text{ 和 } 1 \leqslant \varepsilon G - \varepsilon H \text{ 和 } \omega^+ \geqslant 1\} \\
S_4 &= \{(G, H);\ G \neq 0 \text{ 和 } \sigma G \neq \sigma H \text{ 和 } \varepsilon G = \varepsilon H\} \\
S_{5a} &= \left\{(G, H);\ G \neq 0 \text{ 和 } \sigma G \neq \sigma H \text{ 和 } 1 \leqslant \varepsilon G - \varepsilon H \leqslant p \text{ 和 } \omega^+ \geqslant \frac{1}{r}\right\} \\
S_{5b} &= \left\{(G, H);\ G \neq 0 \text{ 和 } \sigma G \neq \sigma H \text{ 和 } \varepsilon G - \varepsilon H \geqslant p + 1 \text{ 和 } \omega^+ \geqslant \frac{1}{r}\right\} \\
S_6 &= \left\{(G, H);\ G \neq 0 \text{ 和 } \sigma G \neq \sigma H \text{ 和 } \varepsilon G - \varepsilon H = 1 \text{ 和 } \omega^+ < \frac{1}{r}\right\} \\
S_7 &= \left\{(G, H);\ G \neq 0 \text{ 和 } \sigma G \neq \sigma H \text{ 和 } 2 \leqslant \varepsilon G - \varepsilon H \text{ 和 } \omega^+ < \frac{1}{r}\right\}
\end{aligned}
\tag{10.72}
$$

这十个集合的并集覆盖了子空间 $\hat{S}$．当G和H有 $\varepsilon G - \varepsilon H \geqslant p$ 时，它们不能成为 S_3 的一项．因为在 S_7 中 $\varepsilon G - \varepsilon H \geqslant 2$，所以 $\omega^+ > 1/r^2$ 指示出只有一个位置删除．

对于浮点乘法的比较集合

分数值 $\tilde{\varphi}(G\oplus H)$ 被写成 ω^*．于是谓语为：

$$
\begin{aligned}
&\omega^* \geqslant \frac{1}{r}, \text{ 指 } G\circledast H \text{ 没有删除} \\
&\omega^* < \frac{1}{r}, \text{指 } G\circledast H \text{ 有删除}
\end{aligned}
\tag{10.73}
$$

对浮点乘法的比较集合为

$$
\begin{aligned}
S_8 &= \left\{(G, H);\ (G, H) \in S \text{ 和 } \omega^* \geqslant \frac{1}{r}\right\} \\
S_9 &= \left\{(G, H);\ (G, H) \in S \text{ 和 } \omega^* < \frac{1}{r}\right\}
\end{aligned}
\tag{10.74}
$$

乘积 $G\circledast H$ 永远不会有进位．并集 $S_8 \cup S_9$ 包括了空间 S．前面指出，当 $(G, H) \in S_9$，

$$(1/r^2) \leqslant \omega^* \leqslant \frac{1}{4}$$

时，将只出现单个删除.

对于浮点除法的比较集合

该比较集合为

$$\begin{aligned} S_{10} &= \{(G,H);(G,H)\in S \text{ 和 } \tilde{\phi}G \geqslant \tilde{\phi}H\} \\ S_{11} &= \{(G,H);(G,H)\in S \text{ 和 } \tilde{\phi}G < \tilde{\phi}H\} \end{aligned} \tag{10.75}$$

表 10.5　与每一类操作的算法有关的浮点操作数对在比较集合中的百分数（Kent[14]）

操作类型	问题	A	B	C	D	E	总计
加法或减法	浮点加法/减法指令的百分比	10.6	5.2	15.1	5.6	5.3	6.1
	S_0	0.4	1.0	0	1.6	2.2	1.4
	S_{0a}	23.8	27.7	1.7	12.1	0	10.5
	S_1	4.6	5.2	50.9	12.2	2.3	14.4
	S_{2a}	22.6	47.9	38.9	26.7	14.7	27.6
	S_{2b}	0.5	0.9	0	0.6	0	0.5
	S_3	6.8	5.5	3.3	9.2	3.7	7.2
	S_4	10.1	3.0	0.9	8.6	57.6	15.2
	S_{5a}	15.6	4.7	2.0	15.4	6.0	11.5
	S_{5b}	0.1	0	0	0.3	0	0.2
	S_6	10.8	2.7	0.6	9.3	12.0	8.3
	S_7	4.7	1.4	1.7	4.0	1.5	3.2
	Σ	100	100	100	100	100	100
乘法	问题	A	B	C	D	E	总计
	浮点乘法指令的百分比	10.4	2.2	3.2	6.9	3.0	5.7
	S_8	38.6	49.3	38.9	51.1	39.7	49.1
	S_9	61.4	50.7	61.1	48.9	60.3	50.9
	Σ	100	100	100	100	100	100
除法	问题	A	B	C	D	E	总计
	浮点除法指令的百分比	2.0	4.1	5.9	1.4	1.2	1.5
	S_{10}	55.2	69.5	68.0	54.4	58.2	57.7
	S_{11}	44.8	30.5	32.0	45.6	41.8	42.2
	Σ	100	100	100	100	100	100

显然，$S_{10}\cup S_{11}=S$，而且在 $G\oslash H$ 中能出现唯一的进位和没有进位.

Kent[14,15] 曾经用上述比较集合来研究下列计算机系列中加法/减法，乘法和除法等浮点指令的算法：CDC **6000**–Cyber **70**；CDC **3000**；Univac **1100**，SM3，SM4；以及 IBM **360–370**. 操作数对的子集合的划分，可以用来论述大多数现代计算机的浮点指令中规格化和截断的影响. 关于浮点操作数对在一些比较集合上的统计分布数据与具体的问题有关. 表 10.5 给出了对五个程序问题，浮点操作数对在上面定义的比较集合中所占的百分比. 通过这五个程序的调查，得出在 750000 条指令中有 46000 条浮点加法/减法，43000 条浮点乘法，11000 条浮点除法.

10.9 多倍精度的浮点加法/减法

在这一节中，我们介绍一种加/减**双倍精度**（DP）浮点数的算术运算的算法．这里所介绍的算法能用硬件或软件来实现．为了加深读者对实际情况的印象，我们假定机器字长为 48 位．因此，双倍精度浮点操作数占据二个相邻的存储器字，共 96 位．假定 DP 浮点数具有图 10.5 所示的数据格式

$$F = f \times 2^{\beta} \tag{10.76}$$

这里 $|f|$ 处在下列范围内

$$\frac{1}{2} \leqslant |f| \leqslant 1 - 2^{-84} \tag{10.77}$$

第一个字最左边的二位分别代表 F 的符号和阶 β 的符号．第一个字接下去的十位代表 β 的数值．用来表示 f 的其余 84 位，前 36 位在第一个字中，剩下的后 48 位在第二个字中，蕴含的小数点如图所示．我们可以写出

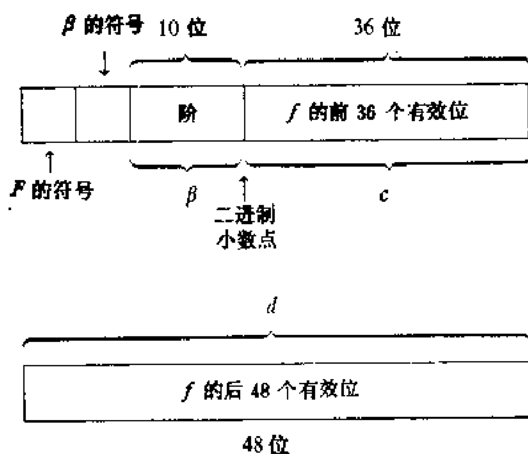


图 10.5 基本字长 48 位的二进制机器中双倍精度的数据格式

$$|f| = c + d \times 2^{-36} \tag{10.78}$$

这里 c 和 d 处在下列范围内

$$\frac{1}{2} \leqslant c \leqslant 1 - 2^{-36}$$

$$0 \leqslant d \leqslant 1 - 2^{-48} \tag{10.79}$$

分数 c 表示在第一个字中 $|f|$ 的前 36 位，d 表示第二个字中的 $|f|$ 的后 48 位．第二个 DP 浮点数 $G = g \times 2^{\delta}$ 可以类似地定义，这里

$$\frac{1}{2} < |g| < 1 - 2^{-84};$$

而

$$|g| = a + b \times 2^{-36} \tag{10.80}$$

其中 $\frac{1}{2} \leqslant a \leqslant 1 - 2^{-36}$ 和 $0 \leqslant b \leqslant 1 - 2^{-48}$．因此，$a$ 和 b 分别代表 $|g|$ 的高位和低位部分．负数用 1 的补码系统表示．改变 DP 浮点数符号的步骤是对整个 85 位分数进行按位求补（包括符号，但阶除外）．我们规定，如果

$$\beta \geqslant \delta \tag{10.81}$$

则 F 大于 G，如果相反的话，则 F 小于 G．符号 L 代表一对辅助的相邻存储器字，它包含二个数 F 和 G 中的较大者．另一方面，符号 S 也表示一对相邻字，它包含二个数中的较小者．现在，我们来说明与 DP 浮点加法/减法有关的标准 DP 算术运算算法．DP 浮点乘法和除法将在下一节叙述．

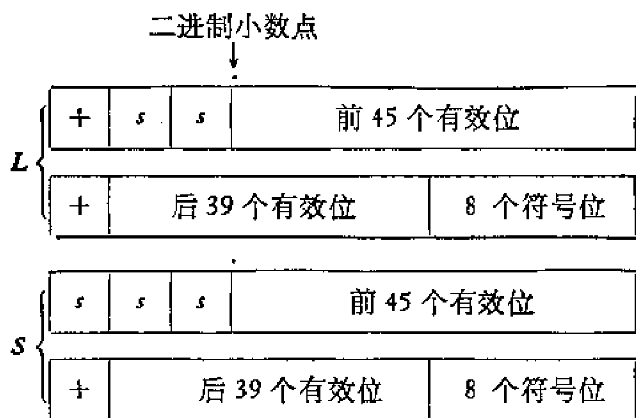
双倍精度的浮点加法

$$F + G = f \times 2^{\beta} + g \times 2^{\delta} \tag{10.82}$$

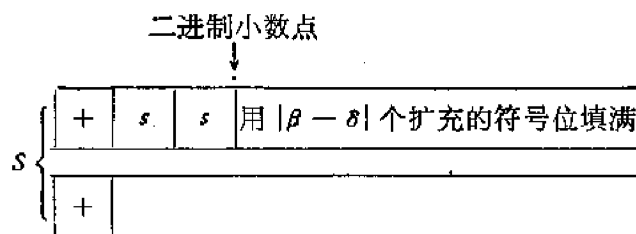
我们假定 F 大于 G，这并不丧失一般性. **DP** 加法是按下列算术运算操作顺序来执行的.

1. 记录 δ 和 β，并确定 $\beta-\delta$ 的符号. 然后，按方程式 10.81 来决定 F 是否大于或小于 G.

2. 因为 F 大于 G，所以将 f 和 g 装入 L 和 S. 求出下列二进位模式，这里"s"代表"符号位".



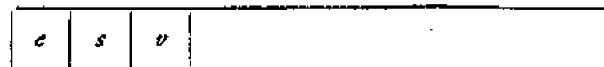
3. S 右移 $|\beta-\delta|$ 个位置，并且在这二个字的每一个字开始处放一个"+"位. 如果 $|\beta-\delta|\geqslant 84$，那就没有必要继续下去，因为 S 中所有的有效位将被丢失.



4. 将 L 和 S 的后半部相加得到

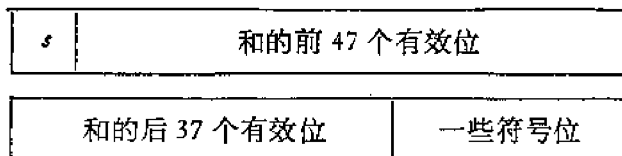
如果这个和的第一位 c 是一个"1"，那么实际上就有一个进位；否则就没有进位.

5. 将 L 和 S 的前半部相加，再加上第 4 步中得到的进位位.



如果 $c=1$，那么循环进位必须加到第 4 步中产生的字的最低有效位上. 第 4 步中的 c 在循环进位以前必须清除为零. 如果 $v=s$，那么 v 就是一个符号位. 然而，如果 $v\neq s$，那么在加法期间便会有上溢. 为了得到正确的结果，必须进行阶的调整.

6. 和的后半部左移一个位置，清除进位位 c. 于是，如果 $v\neq s$，就把双倍长度的和左移一个位置，如果 $v=s$，则移二个位置. 和保留为下面的形式:



7. 通过左移以及相应地调整阶来使和规格化. 如果 84 次左移仍不足以规格化,那么和应该取为零. 这样, 和的 84 个最高有效位与代表符号和阶的 12 位一起,装入二个 48 位的字中,如图 10.5 所示. 如果符号是负的,那么前 12 位在装入前必须求补.

双倍精度的浮点减法

$$F - G = F + (-G) \tag{10.83}$$

如上所述,在进入加法程序之前只对 G 求补. 作一些少量修改,上述算法便能推广到三倍精度的浮点系统.

10.10 多倍精度的浮点乘法和除法

下面采用上一节所述 48 位机器同样的 **DP** 数据格式. 我们将介绍 **DP** 浮点乘法和 **DP** 浮点除法的算法. 这二个算法也能用硬件或软件的方法来实现. 它们也能推广到处理三倍精度的浮点数. 缩写词"sgn$F$"用来表示浮点数 F 的符号.

双倍精度的浮点乘法

$$\begin{aligned} F \times G &= (f \times 2^{\beta}) \times (g \times 2^{\delta}) \\ &= (\text{sgn}F \cdot G) \times |f| \times |g| \times 2^{\beta+\delta} \end{aligned} \tag{10.84}$$

因为 $(\text{sgn } F \cdot G)$ 是外部记录的, 所以这个计算步骤主要涉及 $|f| \times |g| \times 2^{\beta+\delta}$ 的结构. 而且,我们将记录阶 β 和 δ, 一直到分数乘积 $|f| \times |g|$ 形成为止. 对 **DP** 浮点乘法, 可采用下列算法.

1. 从外部记录 $(\text{sgn } F \cdot G)$, 形成 $|F|$ 和 $|G|$,然后记录 $|F|$ 和 $|G|$ 最左边的 12 位, 分别作为 β 和 δ.

2. 将 $|f|$ 和 $|g|$ 的 84 位移到左边,直到每一个都具备下列二进制模式为止:

+	前 47 个有效位	
0	后 37 个有效位	10 个零

这意味着方程式 10.78 表示的 $|f|$ 应该调整为下面的形式.

$$|f| = C + D \times 2^{-47} \tag{10.85}$$

这里 C 和 D 在下列范围内

$$\begin{aligned} \frac{1}{2} \leqslant C \leqslant 1 - 2^{-47} \\ 0 \leqslant D \leqslant 1 - 2^{-37} \end{aligned} \tag{10.86}$$

同样, $|g|$ 具有形式

$$|g| = A + B \times 2^{-47} \tag{10.87}$$

而 A, B 在下列范围内

$$\frac{1}{2} \leqslant A \leqslant 1 - 2^{-47}; \ 0 \leqslant B \leqslant 1 - 2^{-37} \tag{10.88}$$

3. 用浮点操作来形成分数乘积

$$|f| \times |g| = (C + D \times 2^{-47}) \times (A + B \times 2^{-47})$$
$$= CA + (CB + DA) \times 2^{-47} + DB \times 2^{-94}$$
$$\cong CA + (CB + DA) \times 2^{-47} \quad (10.89)$$

注意：$DB \times 2^{-94}$ 这一项由于最多只保留 84 个有效位而被删掉．方程式 10.89 的计算用图 10.6 中的流程图来说明．

4．将方程式 10.89 中得出的乘积进行舍入和规格化．由于必须执行规格化，所以阶 $\beta + \delta$ 应作适当调整．然后，把 84 位规格化乘积以及 12 位符号和调整好的阶装人二个48位的字中，与前面所述相同．如果 $(\mathrm{sgn}\ FG)$ 是负的，则应再做最后求补的步骤．

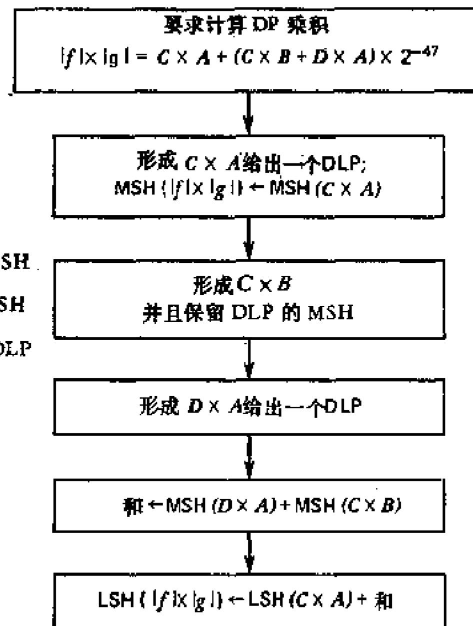


图 10.6 方程式 10.89 中规定的双倍长度乘积的计算步骤

双倍精度的浮点除法

$$\frac{G}{F} = \frac{g \times 2^{\delta}}{f \times 2^{\beta}}$$
$$= (\mathrm{sgn}\ G \cdot F)\left|\frac{g}{f}\right| \times 2^{\delta-\beta} \quad (10.90)$$

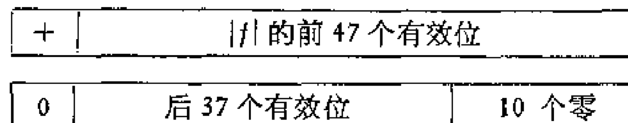
因为我们希望 $|g/f| < 1$，所以我们必须把分子缩小一倍，得到

$$\frac{G}{F} = (\mathrm{sgn}\ G \cdot F)\left|\frac{g/2}{f}\right| \times 2^{\delta-\beta+1} \quad (10.91)$$

下面的算法就是为 **DP** 浮点除法设计的

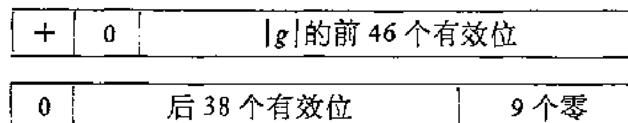
1．重复上述 **DP** 乘法算法中的第一步．

2．安排 $|f|$ 的 84 位，得出二进制模式



+	$|f|$ 的前 47 个有效位	
0	后 37 个有效位	10 个零

安排 $|g|$ 的 84 位，得到



+	0	$|g|$ 的前 46 个有效位
0	后 38 个有效位	9 个零

因此，和乘法的情况一样，$|f|$ 用方程式 10.85 和 10.86 来表示．然而，我们有

$$\left|\frac{g}{2}\right| = A + B \times 2^{-47} \quad (10.92)$$

这里 A 和 B 处在下列范围内

$$\frac{1}{4} \leqslant A \leqslant \frac{1}{2} - 2^{-47}; \quad 0 \leqslant B \leqslant 1 - 2^{-56} \tag{10.93}$$

3. 用浮点操作来形成商

$$\begin{aligned}
\frac{|g/2|}{f} &= \frac{A + B \times 2^{-47}}{C + D \times 2^{-47}} \\
&= \left[\frac{A + B \times 2^{-47}}{C}\right]\left[\frac{1}{1 + (D/C) \times 2^{-47}}\right] \\
&= \left[\frac{A}{C} + \frac{B}{C} \times 2^{-47}\right]\left[1 - \frac{D}{C} \times 2^{-47} + \frac{D^2}{C^2} \times 2^{-94} - \cdots\right] \\
&= \frac{A}{C} + \frac{B}{C} \times 2^{-47} - \frac{AD}{C^2} \times 2^{-47} - \frac{BD}{C^2} \times 2^{-94} + \cdots \\
&\cong \frac{A}{C} + \left[\frac{B}{C} - \frac{AD}{C^2}\right] \times 2^{-47} = \frac{A + [B - (AD/C)] \times 2^{-47}}{C}
\end{aligned} \tag{10.94}$$

方程式 10.94 的计算由图 10.7 中的流程图来说明.

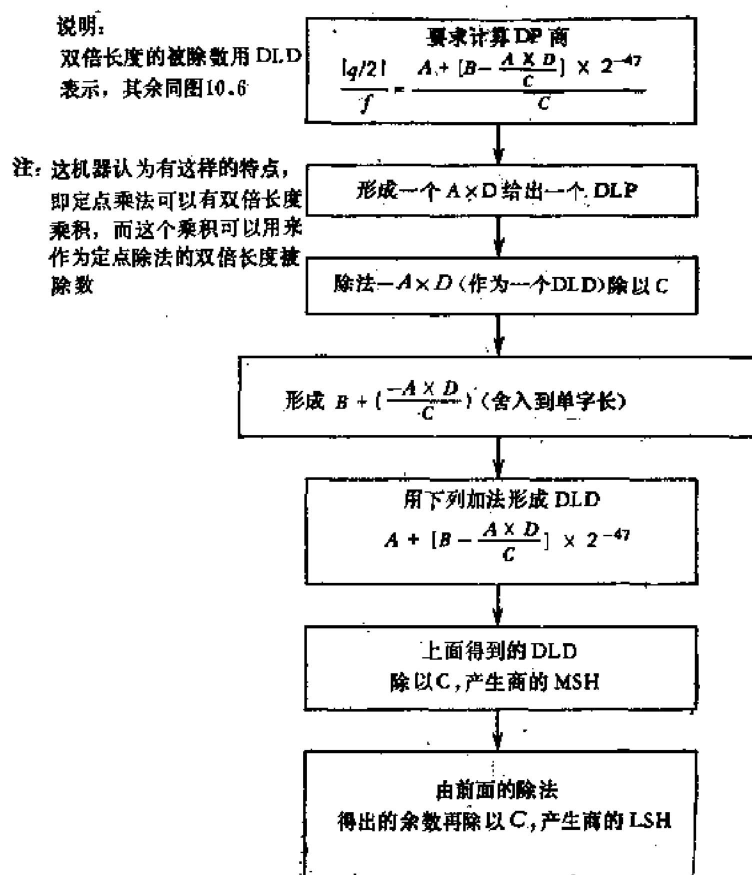


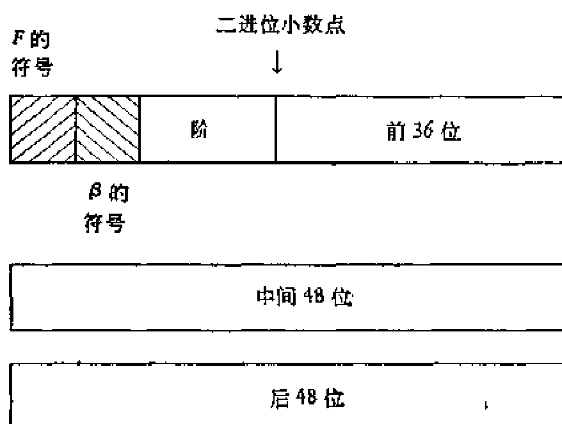
图 10.7　方程式 10.94 中所规定的双倍字长的商的计算步骤

4. 将流程图中用方程式 10.94 求得的商加以舍入和规格化. 因为 $|g/2|/|f|$ 的规格化，在阶 $\delta - \beta$ 中必须作一些调整. 然后重复 DP 乘法算法的第 4 步所描述的装入步骤.

以上关于 84 位数的算术运算算法是由 Gregory 和 Raney[10] 发表的. Ikebe[11] 曾修改了这些浮点 DP 乘法和 DP 除法的算法，使之适合三倍精度的数，即具有

$$132 = 36 + 48 + 48$$

位分数. Ikebe 提出的数据格式可作为 3 字浮点数的一个例子表示如下:



10.11 参考文献注释

本章介绍了有关浮点算术运算系统的一些新课题以及最近的发展. 非规格化浮点算术运算操作由 Ashenhurst 和 Metropolis[1,22] 在研制芝加哥大学的 MANIAC III 计算机时作过研究. 在 Yohe 的著作[27,28]中可以找到对各种舍入方案的杰出的处理. Wilkinson[26] 曾经对代数处理过程中的舍入误差进行了综合研究. 浮点计算中的舍入误差则有 Caprani[4], Cody[6], Dorr 和 Moler[8], Keneko 和 Liu[13], 以及 Tsao[25] 等人作过探讨. 公理化的舍入理论是基于 Kulisch[19] 和 Yohe[28] 的著作.

在以上关于舍入误差的论述中, Bohlender[2], Brent[3], Carr[5], Kuki 和 Cody[18] 等人曾对浮点算术运算的误差分析作了推演. 对于浮点算术运算设计的最佳基数的选择在 Cody[6] 和 Garner[9] 中有过描述. Matula[21]使浮点基数变换技术得到定形. Kuck, Parker 和 Sameh[16,17], Kent[14,15] 提出了新的 ROM 舍入方案, 介绍了浮点指令的集合论研究, 他们用比较集合来划分规格化操作数对的空间. Lunde[20] 用程序跟踪的办法来评价浮点指令. 有关双倍精度浮点算术运算的算法是 Gregory 和 Raney[10] 发明的. 感兴趣的读者应该检验一下 Ikebe[11] 中三倍精度乘法/除法的细节. Sterbenz[23]对双倍精度计算作了很好的处理. 现有计算机系统有关典型浮点算术运算指令的目录资料已发表在 CDC **6000** 系列参考手册[7], Thornton[24] 以及 IBM **370** 系统的操作原理[12]中, 也包括在一些先进的计算机工厂新近发表的许多系统手册中.

参　考　文　献

[1] Ashenhurst, R. L. and Metropolis, N., "Unnormalized Floating-Point Arithmetic," *Journal of ACM*, Vol. 6, March 1959, pp. 415—428.

[2] Bohlender, G., "Floating-Point Computation of Functions with Maximum Accuracy," *IEEE Trans. Computers*, Vol. C-26, No. 7, July 1977, pp. 621—632.

[3] Brent, R. P., "On the Precision Attainable with Various Floating-Point Number Systems," *IEEE Trans. Computers*, Vol. C-22, June 1973, pp. 601—607.

[4] Caprani, O., "Roundoff Errors in Floating-Point Summation," *BIT Vol. 15*, No. 1, January 1975, pp. 5—9.

[5] Carr, J. W. III, "Error Analysis in Floating-Point Arithmetic," *Comm. of ACM*, Vol. 2, No. 5, May 1959. pp. 10—15.
[6] Cody, W. J., Jr., "Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. C-22, June 1973, pp. 596—601.
[7] Control Data Corp., *CDC 6000 Series Computer Systems*, Reference Manual, Pub. No. 60100000-M, 1972.
[8] Dorr F. W. and Moler, C. B., "Roundoff Errors on the CDC 6600/7600 Computers," *SIGNUM Newsletter, Ass. for Comp. Mach.*, Vol. 8, 1973, pp. 24—26.
[9] Garner, H. L., "A Survey of Some Recent Contributions to Computer Arithmetic," *IEEE Trans. Computers*, Vol. C-25, No. 12, pp. 1277—1282.
[10] Gregory, R. T. and Raney, J. L., "Floating-Point Arithmetic with 84-bit Numbers," *Comm. of ACM*, Vol. 7, No. 1, January 1964, pp. 10—13.
[11] Ikebe, Y., "Note on Triple-Precision Floating-Point Arithmetic with 132-bit Numbers," *Comm. of ACM*, Vol. 8, No. 3, March 1965.
[12] IBM Staff, *IBM System/370. Principles of Operation*, Pub. No. GA 22-7000-2.
[13] Kaneko, T. and Liu, B., "On Local Roundoff Errors in Floating-Point Arithmetic," *Journal of ACM*, Vol. 20, No. 3, July 1973, pp. 391—398.
[14] Kent, J. G., "Highlights of A Study of Floating-Point Instructions," *IEEE Trans. Computers*, Vol. C-26, No. 7, July 1977, pp. 660—666.
[15] Kent, J. G., "Comparison Sets: A Useful Partitioning of the Space of Floating-Point Operand Pairs," *3rd Symposium on Computer Arithmetic*, IEEE Computer Society, Catalog No. 75CH1017-3C, November 1975, pp. 37—39.
[16] Kuck, D. J. et al., "ROM-Rounding: A New Rounding Scheme," *3rd Symposium on Computer Arithmetic*, IEEE Computer Society Catalog No. 75CH1017-3C, 1975, pp. 67—72.
[17] Kuck, D. J. et al., "Analysis of Rounding Methods in Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. C-26, No. 7, July 1977, pp. 643—650.
[18] Kuki, H. and Cody, W. J., "A Statistical Study of the Accuracy of Floating-Point Number System," *Comm. of ACM*, Vol. 16, No. 4, April 1973, pp. 223—230.
[19] Kulisch, U., "Mathematical Foundation of Computer Arithmetic," *IEEE Trans. on Computers*, Vol. C-26, No. 7, July 1977, pp. 610—620.
[20] Lunde, A., "Evaluation of Instruction Set Processor Architecture by Program Tracing," *Ph. D. Thesis*, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., July 1974.
[21] Matula, D. W., "A Formalization of Flaoting-Point Numeric Base Conversion," *IEEE Trans. Computers*, Vol. C-19, August 1970, pp. 581—691.
[22] Metropolis, N. and Ashenhurst, R. L., "Basic Operations in an Unnormalized Arithmetic System," *IEEE Trans.*, Vol. EC-12, 1963, pp. 896—904.
[23] Sterbenz, P. H., *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, N. J., 1974.
[24] Thornton, J. E., *Design of A Computer: The CDC 6600*, Scott, Foresman and Company, Glenview, Illinois, 1970, Chapter 5.
[25] Tsao, N. K., "On the Distribution of Significant Digits and Roundoff Errors," *Comm. of ACM*, Vol. 17, No. 5, May 1974, pp. 260—271.
[26] Wilkinson, J. H., *Rounding Errors in Algebraic Processes* Prentice-Hall, Englewood Cliffs, N. J., 1963.
[27] Yohe, J. M., "Foundations of Floating-Point Computer Arithmetic," *Tech. Rep. 1302*, Math. Res. Center, University of Wisconsin, Madison, January 1973.
[28] Yohe, J. M., "Rounding in Floating-Point Arithmotic," *IEEE Trans. Computers*, Vol. C-22, No. 6, June 1973, pp. 577—586.

习　　题

题 10.1 考虑具有 n 位字长的浮点数中，尾数字段为 p 位，阶字段为 q 位，则 $p+q+1=n$，这里浮点数的符号占据一位。试在规格化与非规格化浮点加法型操作进行比较时，通过统计分析来确定加法上溢或下溢的相对频率。

题 10.2 阐明下列与浮点算术运算的计算有关的几个术语：

(a) 算前规格化和算后规格化；

(b) 有效数位的算术运算；

(c) 带截断的机器算术运算;

(d) 带舍入的机器算术运算;

(e) 平均相对表示法误差 (ARRE);

(f) 显式与隐式规格化数的比较;

(g) 保护数位;

(h) 浮点指令的比较集合.

题 10.3 考虑一个具有真值形式的八进制分数 $x=(0.3725765436)_8$ 试说明下列表示式,并决定每一种情况下所引起的相对误差.

(a) $(x)_8$ } 在截断以后.
(b) $(x)_6$

(c) $(x)_8^*$ } 在舍入以后.
(d) $(x)_6^*$

题 10.4 令 $a=(0.10101010111000101110101011010110)$, 是一个 32 位正的实际分数,$\bar{a}$ 是 a 的负数. 现在考虑尾数字段有 $p=24$ 位的机器. 试分别回答下列问题

(a) 确定二个连续的机器可表示的分数 m_1 和 m_2, 这里 $m_1<a<m_2$.

(b) 定义一种舍入 ρ, 使它属于向下舍入的方案,同时对着上述机器来说明 $\rho(a)$.

(c) 对向上舍入的方案,重复 (b).

(d) 你所定义的上述舍入方案是对称的吗? 如果不是,你能画一个对称的吗?

题 10.5 考虑题 10.4 中同样的机器, 试对第 10.4 节中定义的五种舍入方案决定下列机器表示式

(i) $\Delta(a)=$ ______________

$\nabla(a)=$ ______________

$\mathrm{T}(a)=$ ______________

$\mathrm{A}(a)=$ ______________

$\mathrm{P}(a)=$ ______________

(ii) 对负的分数 (a), 重复 (i), 即决定

$\Delta(\bar{a})=$ ______________

$\nabla(\bar{a})=$ ______________

$\mathrm{T}(\bar{a})=$ ______________

$\mathrm{A}(\bar{a})=$ ______________

$\mathrm{P}(\bar{a})=$ ______________

题 10.6 对题 10.4 中提到的同一机器, 试说明一个以 4 位的 ROM 为基础的舍入方案的查表内容. ROM 的 4 位地址由 3 个低位加一个被舍入的最高位所组成,如图 10.1 所示. 该 ROM 应具有 3 位输出. 试决定 $\rho_{ROM}(a)$. 其中 ρ_{ROM} 是 ROM 舍入, 而 "a" 已在题 10.4 中说明. 又问 $\rho_{ROM}(a)$ 等于题 10.5中第 (i) 部分的哪一个舍入表示式?

题 10.7 试将第 10.9 节说明的双倍精度浮点加法的算法扩展到三倍精度浮点加法的算法,这里采用 Ikebe[11] 所用的同一数据格式.

题 10.8 试用计算机模拟的方法,通过一对给定数值的操作数 $|f|$ 和 $|g|$ 来验证图 10.6 和 10.7 中所指定的步骤. 并且分别用方程式 10.89 和方程式 10.94 中的近似式求出最大相对误差.

第十一章　基本函数,流水线算术运算和误差控制

11.1 引　　言

在本章内,我们学习数字计算机中算术运算系统的三个重要方面. 首先,我们将讨论计算基本函数的数字方法,如三角函数,反三角函数,指数函数,对数函数,双曲和反双曲函数,求平方根,求平方,以及其他超越函数. 我们先从设计一个求平方根的单元阵列部件和一个求平方的阵列部件开始,然后说明如何用多项式近似的方法来计算有界的基本函数. 在11.4一节中,将提出以 Walther 统一方法[35] 为基础的求基本函数的 CORDIC 计算技术. Chen[5] 提出的计算指数、对数、比例和平方根的收敛方法,将和其起始、变换与结束的法则一起讨论.

其次,我们研究流水线计算的原理及其相应的算术运算处理器设计. Texas Instruments 公司的先进科学计算机 (**ASC**) 中采用的流水线算术运算设计将在 11.7 节中论述. 我们将研究用单元阵列逻辑构成的,执行乘法、除法、平方根,以及平方操作的通用算术运算流水线. 在 11.9 节中,我们描述流水线如何用来设计一个执行快速傅里叶变换 (**FFT**) 对的专用运算处理器. 在 11.11 节中将提供控制数据公司 **STAR-100** 计算机中流水线运算处理器的详细研究实例.

第三,我们简要介绍在运算处理器中可能遇到的各种逻辑故障,并提出若干种在容错算术运算设计中对付这些故障的方法. 对以上论题进行进一步研究的文献介绍将列在本章的末尾.

11.2 二进制数的平方根和平方

LSI 工艺的出现,不仅使逻辑电路价格降低,而且使速度提高,尺寸缩小. 由于现在强调功能上的划分,使得专用算术运算功能发生器显得更有吸引力. 近年来,对于设计和实现叠接算术运算逻辑阵列产生相当的兴趣. 高速乘法和除法的叠接单元阵列在第六和第八章中已广泛论及. 在这一节里,我们介绍求二进制数的平方根和平方的算术运算算法,以及相应的单元阵列. 这两种操作可以成对地放在一起,在这种情况下,用一个单一的阵列,再加上一些控制逻辑,就可以选择两种操作中的一种.

考虑有两个正的,二进制分数 A 和 Q,其中 $Q=\sqrt{A}$ 或 $A=Q^2$. 我们称 Q 是 A 的平方根以及 A 是 Q 的平方. 表示为

$$\begin{aligned} Q &= \sqrt{A} = 0.q_1q_2\cdots q_n \\ A &= Q^2 = 0.a_1a_2\cdots a_{2n-1}a_{2n} \end{aligned} \tag{11.1}$$

在图 11.1 中指出了一种常用的不恢复求平方根的算法. 二进制数 A 从二进位小数点开始,成对地分成 a_1a_2, a_3a_4, $\cdots$, $a_{2n-1}a_{2n}$. 从 a_1a_2 中减去第一次减数 $D_0=0.01$. 假如余数 R_1 是正, $q_1=1$, 下一对 a_3a_4 连接到 R_1, 从这个形成的数中再减去 $D_1=0.q_101$. 而假

如余数 R_1 是负，则 $q_1=0$，而 a_3a_4 连接到 R_1，并把 $D_1=0.q_1 11$ 加到这个数中去．一般来说，在决定了平方根的第 k 位 q_k 以后，第 k 次的中间步骤应该做以下操作：

$$R_{k+1} \leftarrow R_k \cdot a_{2k+1}a_{2k+2} - q_1q_2\cdots q_k01, \text{ 如 } q_k = 1, \tag{11.2}$$

$$R_{k+1} \leftarrow R_k \cdot a_{2k+1}a_{2k+2} + q_1q_2\cdots q_k11, \text{ 如 } q_k = 0, \tag{11.3}$$

其中“·”是连接操作，它的实现办法是把旧的余数左移两位，而新的一对数字从右端进入．

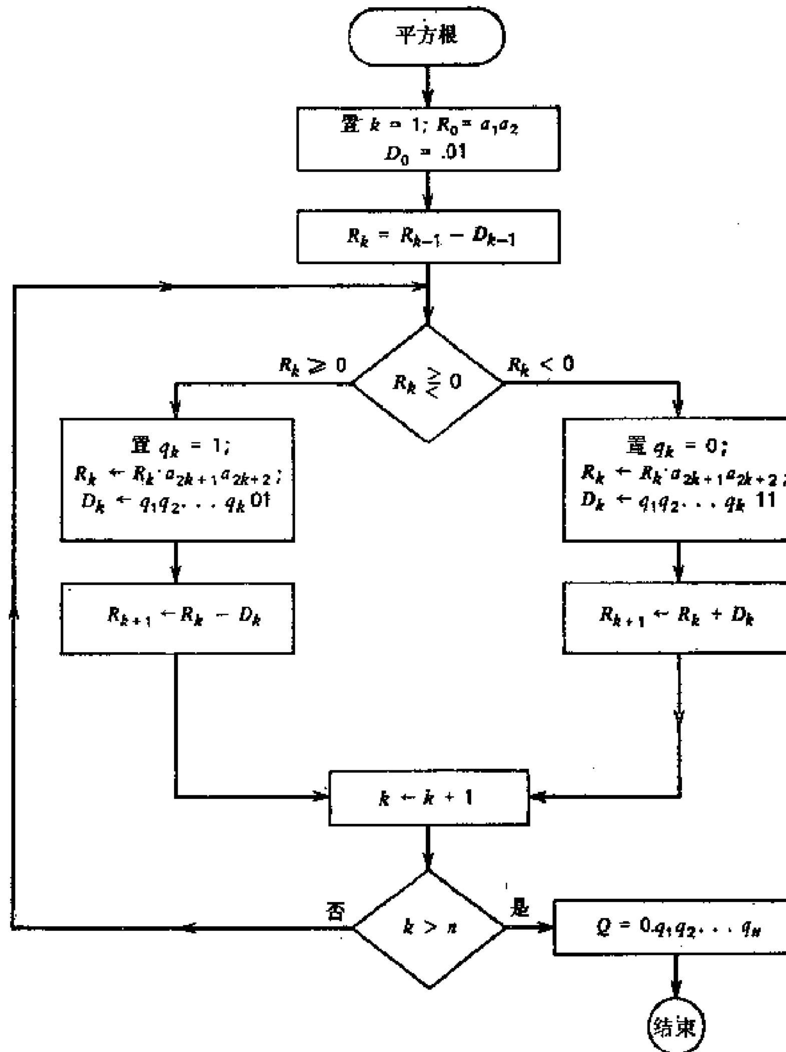


图 11.1　常用的不恢复求平方根的算法

在图 11.2 中表示在不恢复求平方根过程中可能用到的逐次减数的树，图中还有一个 8 位 ($2n=8$) 的二进制求平方根的部件．这个部件是用图 2.10（也在图 8.10）所描述的可控加法-减法 (CAS) 单元所组成的单元阵列构成的．阵列的每行接受一个新的输入数位对．执行减法是用了补码的算术运算，即用补码相加的方法．当一行的左端的进位输出是“1”，则余数是正，$q_k=1$，而下一行应执行减法；否则，$q_k=0$，而下一行的操作执行加法．要实现这种功能控制，只需把本行的左端进位输出信号，连接到功能控制线和下一行的起始进位输入线就可以了．逐次的平方根数位在每行的左端产生，如同图 8.10 中商

的产生一样．图中也列出验证这个阵列所用的数值例子，其中用 $A=(0.10101001)_2=(169/256)_{10}$ 以及 $Q=\sqrt{A}=\sqrt{169/256}=(13/16)_{10}=(0.1101)_2$.

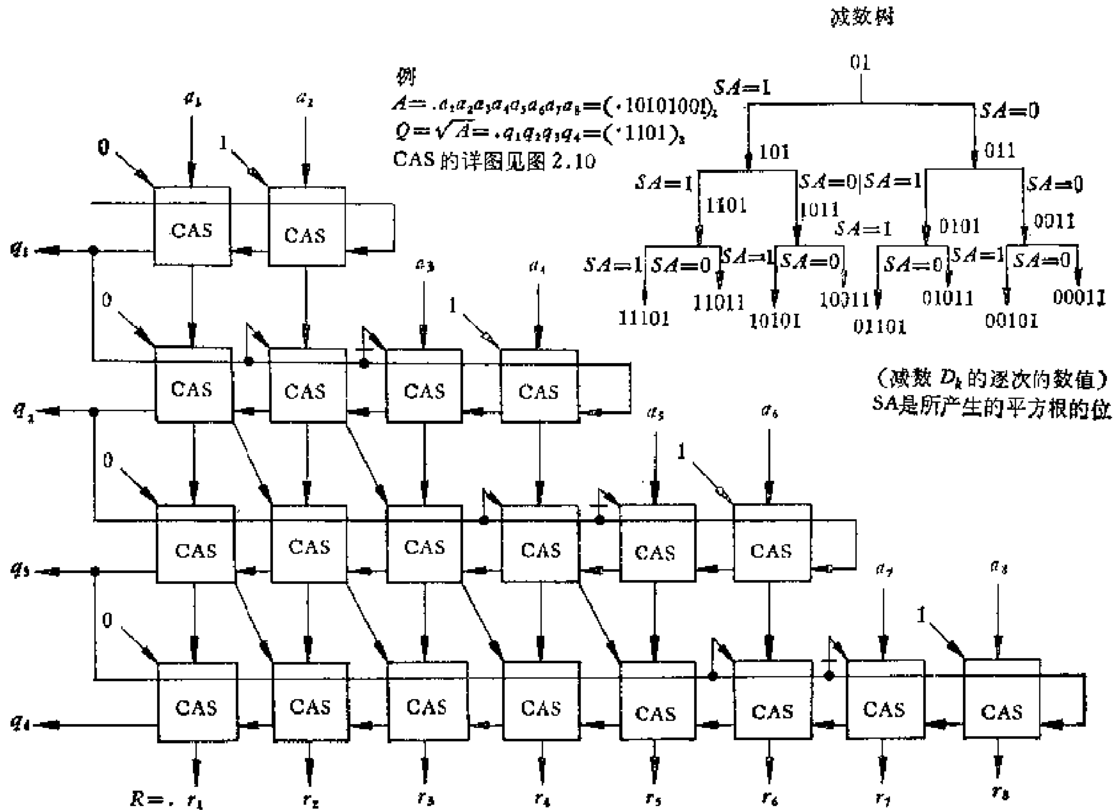


图 11.2 减数的树以及用可控加法减法（CAS）单元构成的 8 位不恢复求平方根部件的线路

下面讨论的是求平方根的另一种方法，它还可同时实现求平方的操作．考虑在方程式 11.1 中所表示的关系式，如 $q_1=1$，则 $A\geqslant(0.1)^2=0.01$．这样，在 A 和 0.01 之间要实行比较，如 $A\geqslant0.01$，则 $q_1=1$，否则 $q_1=0$．同样，如 $A\geqslant(0.q_11)^2$，即

$$A\geqslant(0.q_1)^2+0.0q_101,$$

则 $q_2=1$，如此类推．这个过程一般用以下公式表示：

$$(0.1)^2=0.01$$

$$(0.q_11)^2=(0.q_1)^2+0.0q_101,\ \text{或}\ F_2=F_1+0.0q_101 \tag{11.4}$$

其中如 $q_1=1, F_1=0.01$，否则 $F_1=0.00$．同样，我们有

$$(0.q_1q_21)^2=(0.q_1q_2)^2+0.00q_1q_201,$$

或

$$F_3=F_2+0.00q_1q_201$$

一般情况下，如 $q_{r+1}=1$，则

$$F_{r+1}=F_r+D_r \tag{11.5}$$

其中 $F_r=(0.q_1q_2\cdots q_r)^2$ 是第 r 次**平方**，而 $D_r=0.\overbrace{00\cdots0}^{r\text{次}}q_1q_2\cdots q_r01$ 叫做第 r 次**被开方数**．很明显，如 $q_{r+1}=0$，则 $F_{r+1}=F_r$．以上叠代公式对于所有 $r=1,\cdots,n$ 都适用．

用以上关系式来求平方根的过程可以描述如下：平方根的第一位 q_1 取决于从 A 减

去第一个被开方数 $D_1 = 0.01$. 假如余数是正,则 $q_1 = 1$; 否则 $q_1 = 0$. 在后一情况下,新的余数必须被 A 所取代. 为了获得 q_2, 要从新的余数(或者是从 A, 如 $q_1 = 0$)减去第二个被开方数 $D_2^* = 0.0q_101$. 在经过同样的 n 次叠代以后,即可取得整个平方根.

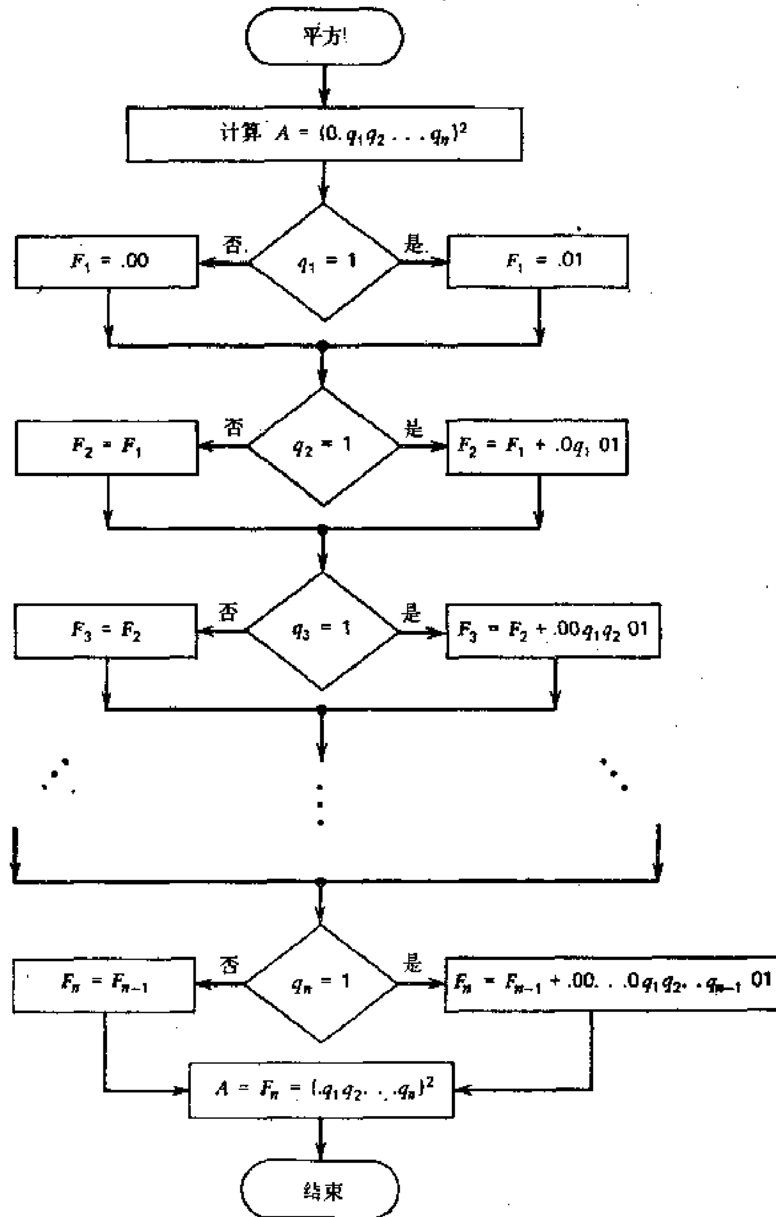


图 11.3 二进制求平方的算法

注意,在以上算法的每一个步骤中,一个数的平方也已在内部计算了. 因此,一个计算平方根的阵列也能够计算平方. 二进制的求平方的过程在图 11.3 中予以描述,它实际上是对逐次的 q_i 数字重复地应用方程式 11.5 所规定的递归方法,直到输入数的整个 $2n$ 位的平方值产生为止.

* 原文为 D_7, 系 D_2 之误. ——译者注

例如:

$Q = .q_1q_2q_3q_4 = (0.1101)_2$

$A = Q^2 = (0.a_1a_2 \cdots a_7)_2 = (0.10101001)_2$

$E = \begin{cases} 0 \text{ 移位} \\ 1 \text{ 加法} \end{cases}$

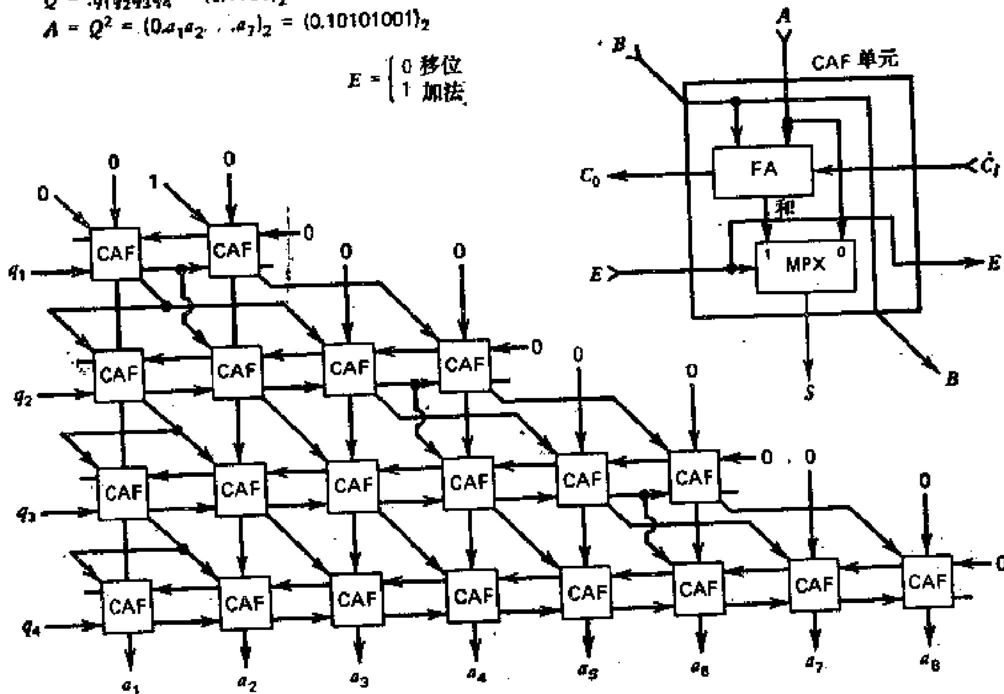


图 11.4　应用可控加法-移位（CAF）单元的 4 位求平方部件的阵列线路

$Q = \sqrt{A}$ 具有 $M = 1$ 和 $B = 0$.

$B^2 = S$ 具有 $M = 0$ 和 $A = 0$.

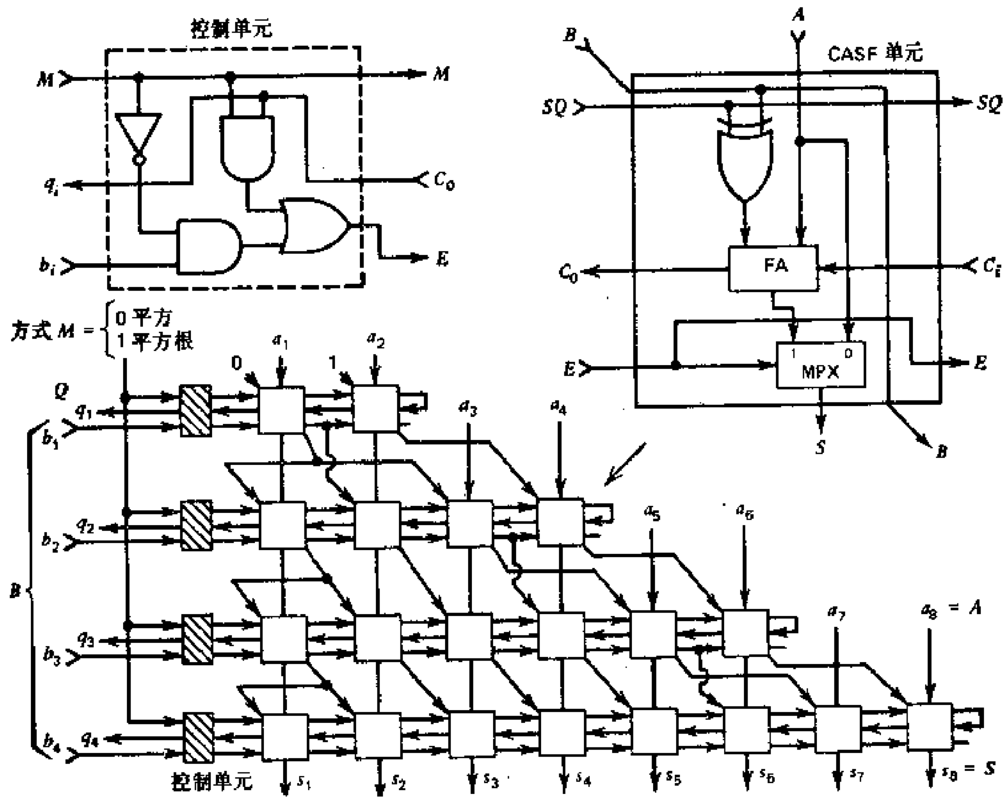


图 11.5　应用可控加法-减法-移位（CASF）单元的求二进制平方根和平方的组合部件阵列线路图

图 11.4 说明上图给定的求平方的算法是如何用阵列实现的. 在这个阵列结构中应用了一种新型的**可控加法移位** (CAF) 单元. 这个 CAF 单元是由一个全加器 (FA) 和一个两输入端的多路转换器 (MPX) 组成, 这个 MPX 用一条外部的允许 (Enable) 线控制,以便从旧的余数输入或者 FA 的和-输出中选择一个作为该单元的输出. 一个二进制数 $Q = 0.q_1q_2q_3q_4$ 从左边的允许线输入,每一行输入一位. 从阵列顶部输入的初始被开方数是零. 最后的平方值 $Q^2 = A = 0.a_1a_2\cdots a_8$ 出现在底部的输出线上. 在图 11.2 中所举的数值例子在这里也可用来验证此设计.

如果引入一个如图 11.5 所示的修改过的算术运算单元, 我们可以设计一种混合的阵列部件, 它既可用来求二进制的平方根, 也可求二进制的平方. **可控加法-减法-移位** (CASF) 只是一个 CAS 单元和一个 CAF 单元的组合. 每一行左端的控制单元是把 q_i 输入或者左边单元的进位输出 C_0 连接到该行的允许输入端. 这样, 在计算平方值时 (方式线 $M = 0$), 输入是加在 $b_1b_2b_3\cdots b_n$ 上. 输出可以在最后一行的"和"输出端 $s_1s_2\cdots s_8$ 处得到. 每一行是在进入该行 q_i 位的控制下进行相加或者移位的.

在求平方根的过程中,方式 $M = 1$. 把所有 SQ 控制线置为 1, 每一行进行相减或者移位. 这里再次假定是用对 2 求补的算术运算. 求平方根的数是加在顶端 $a_1, a_2\cdots a_8$, 而 4 位平方根的解则从左端 $q_1q_2q_3q_4$ 得到. 一般情况下, 这阵列需要有 $n(n+1)$ 个 CASF 单元和 n 个控制单元,才可计算一个 n 位数的平方,或者一个 $2n$ 位数的平方根.

11.3 基本函数的多项式计算

某些有界的基本函数,诸如三角,指数,对数,以及其他有界超越函数, 可以用多项式近似法做给定精度范围内的计算. 数学上,这些函数可以写成如下的无穷项的幂级数:

$$f(x) = \sum_{i=0}^{\infty} a_i \times x^i \tag{11.6}$$

以下例举几个常用的有界函数的幂级数展开式

$$e^{-x^2} = \sum_{j=0}^{\infty} \frac{(-1)^j}{j!} \times x^{2j} \tag{11.7}$$

$$\log_e(1 + x) = \sum_{j=1}^{\infty} \frac{(-1)^{j+1}}{j} \times x^j, \quad 对 -1 < x \leqslant 1; \tag{11.8}$$

$$\sin x = \sum_{j=0}^{\infty} \frac{(-1)^j}{(2j+1)!} \times x^{2j+1}; \tag{11.9}$$

$$\sin^{-1}x = \sum_{j=0}^{\infty} \frac{1 \cdot 3 \cdots (2j-1)}{2^j \cdot (j)! \cdot (2j+1)} \times x^{2j+1}, \ 对\ x^2 < 1; \tag{11.10}$$

$$\cosh x = \sum_{j=0}^{\infty} \frac{1}{(2j)!} \times x^{2j} \tag{11.11}$$

本章讨论一种快速硬件方法, 它使用截断幂级数以趋近基本函数的数值. 截断级数展开式应该如此选择,使它迅速地收敛为所需要的函数值. 在截断的级数中,非零系数的个数愈少, 则计算愈快. 具有零系数的幂次项在计算中并不出现. $k + 1$ 项的截断级数

具有以下形式:

$$f_k(x) = \sum_{i=0}^{k} a_i \times x^{m+i\cdot n} \tag{11.12}$$

其中 $m \geqslant 0$ 是变量 x 的初始幂，而 $n \geqslant 1$ 则是相邻两个非零系数的幂次项的幂增量. 上限 k 通常是由最小整数 $i = k$ 决定,对于一个给定容许误差 ε，应满足下式:

$$|a_k| \geqslant \varepsilon,\ 但\ |a_{k+1}| < \varepsilon \tag{11.13}$$

注意,这种方法只应用于具有单调递减系数的级数,即对于所有 i, $|a_i| > |a_{i+1}|$.作为一个例子,对于给定精度 $\varepsilon = 0.018$,在方程式(11.10)中我们有 $m = 1$,$n = 2$,以及 $k=5$. 这意味着,以下截断级数

$$\sin^{-1}x = x + \frac{x^3}{6} + \frac{3}{40} \times x^5 + \frac{5}{112} \times x^7 + \frac{35}{1152} \times x^9 + \frac{63}{2816} \times x^{11} \tag{11.14}$$

对于计算 $\sin^{-1}x$ 的数值到精度 $\varepsilon = 0.018$ 来说是足够的.

对于 $k + 1$ 项的普遍情况,我们用方程式 11.12 来计算截断级数的方法，应重写成如下所示的嵌套乘积的方式

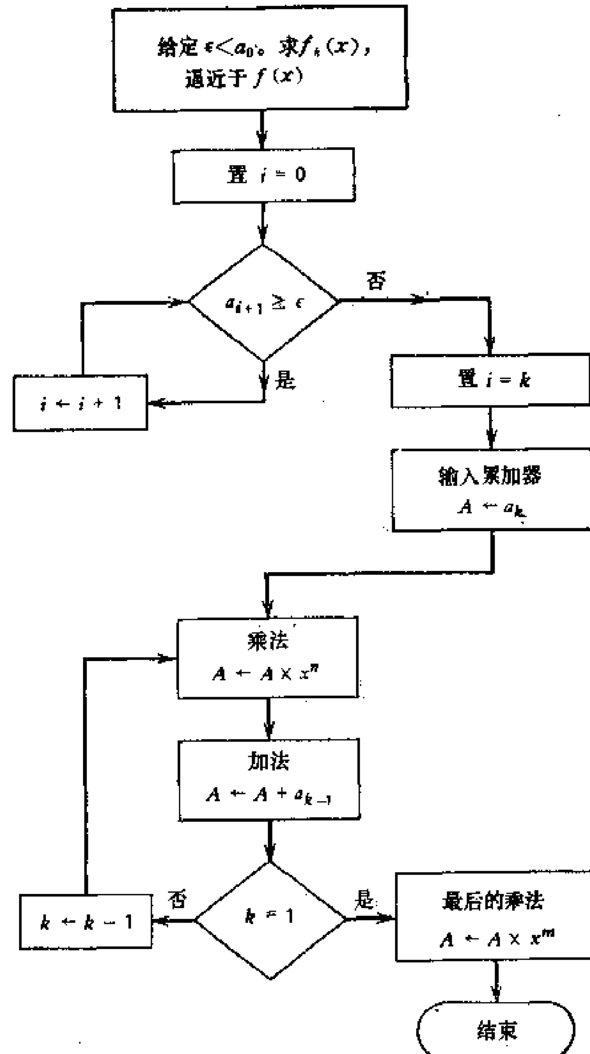


图 11.6　一个有界的基本函数的多项式计算

$$f_k(x) = x^m \times \left[\sum_{j=0}^{k} a_j \times x^{jn}\right] = x^m \times [((\cdots(((a_k \times x^n) + a_{k-1}) \times x^n + a_{k-2}) \times x^n + \cdots + a_1) \times x^n + a_0)] \tag{11.15}$$

因此,计算 $f_k(x)$ 最多只需要变量 x 的两个幂项,即 x^m 和 x^n. (实际上,如果这是可行的话,只需 x 的一个幂项就够了,即 x^{m+n}.)注意,如果不用这个嵌套式子,则需要计算 x 的 $k+1$ 个幂项,这将化费较长的计算时间. 简单的叠代乘法和加法可用来计算11.15式. 正如图 11.6 的流程图所阐明的那样,一个 $(k+1)$ 项幂级数,总共需要 $k+1$ 次乘法,还带有 k 次加法.

在很多情况下,逐项的系数也可以用叠代计算来求得. 举例说,方程式 11.14 中逐项系数可用下式计算

$$a_j = a_{j-1} \times \frac{(2j-1)^2}{(2j) \times (2j+1)} \tag{11.16}$$

其中 $j = 1, 2, 3, 4, 5$,而起始值 $a_0 = 1$. 通常,在按方程式 11.15 开始递归计算以前,所有非零系数和所用的整数幂(x^m 和 x^n)必须预先确定,并且存储在一个高速缓冲存储器或者某些工作寄存器中. 对于 $\sin^{-1}x$ 的例子,具有 $(m, n, k) = (1, 2, 5)$,方程式 11.15 应有以下形式:

$$\begin{aligned}
\sin^{-1}x &= x \times [(((((a_5 \times x^2) + a_4) \times x^2 + a_3) \times x^2 + a_2) \times x^2 + a_1) \times x^2 + a_0] \\
&= x \times \left[\left(\left(\left(\left(\left(\frac{63}{2816} \times x^2\right) + \frac{35}{1152}\right) \times x^2 + \frac{5}{112}\right) \times x^2 + \frac{3}{40}\right) \times x^2 + \frac{1}{6}\right) \times x^2 + 1\right]
\end{aligned} \tag{11.17}$$

如果所有系数 a_i 以及幂项 x 和 x^2 可以很快地从某些预处理级中得到,那末完成 11.17 式计算共需六次乘法和五次加法.

11.4 Walther 统一 CORDIC 计算技术

使用座标旋转向量最初是由 Valder[34] 提出的,这个方法用于研制座标旋转数字计算机(CORDIC),它能计算三角函数,乘法,除法以及在二进制的与混合基数的数系统之间进行转换. 以下讨论的是 Walther 的统一算法[35],它用来计算基本函数,包括乘法,除法,正弦,余弦,正切,反正切,双曲正弦,双曲余弦,双曲正切,反双曲正切,对数,指数以及平方根. 所需的基本操作为移位,加法,减法以及重新调用预先存储的常数. Hewlett-Packard 实验室曾研制应用此算法的硬件浮点处理器. 在 11.12 节中将讨论在设计计算器时,对这种算法如何进一步简化.

这种算法的基础是在直线座标,圆座标和双曲线座标系统中作座标旋转,用什么座标,取决于要计算的是什么函数. 这些座标系统采用 m 参量为表征,其中半径 R 和向量 $P = (x, y)$ 的夹角 A 如图 11.7 所示,定义为

$$R = (x^2 + m \times y^2)^{1/2} \quad A = \frac{1}{\sqrt{m}} \times \tan^{-1}(\sqrt{m} \times y/x) \tag{11.18}$$

参量 m 的三种不同的值,相应于图中所示的圆 $(m = 1)$,直线 $(m = 0)$,以及双曲线

$(m=-1)$ 座标系统。从 $P_i=(x_i, y_i)$，按照下式可求出新向量 $P_{i+1}=(x_{i+1}, y_{i+1})$

$$x_{i+1}=x_i+my_i\delta_i$$
$$y_{i+1}=y_i-x_i\delta_i \tag{11.19}$$

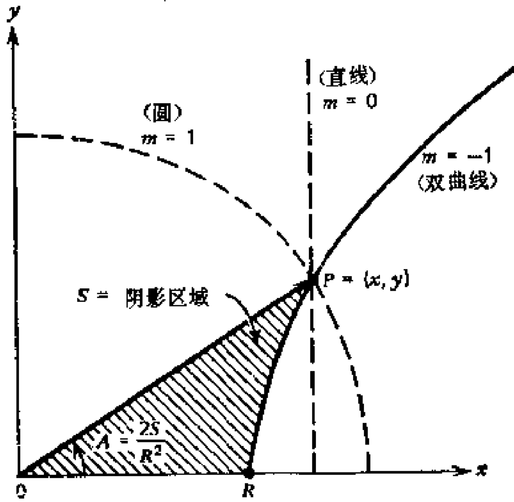


图 11.7 在三种座标系统中的向量 $P=(x, y)$ 的夹角 A 以及半径 R

其中 δ_i 是一个任意的增量或减量值。新向量的夹角和半径按下式求出

$$A_{i+1}=A_i-\alpha_i,$$
$$R_{i+1}=R_i\times K_i \tag{11.20}$$

其中

$$\alpha_i=\frac{1}{\sqrt{m}}\times\tan^{-1}(\sqrt{m}\times\delta_i),$$
$$K_i=(1+m\times\delta_i^2)^{1/2} \tag{11.21}$$

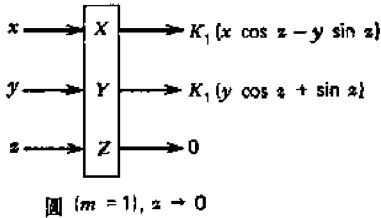
表 11.1 给出这三种座标系统的 α_i 和 K_i 的方程式。

在 n 次叠代以后，我们得到

$$A_n=A_0-\alpha$$
$$R_n=R_0\times K \tag{11.22}$$

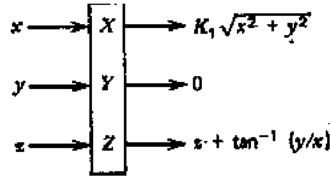
其中

$$\alpha=\sum_{i=0}^{n-1}\alpha_i \quad K=\prod_{i=0}^{n-1}K_i \tag{11.23}$$

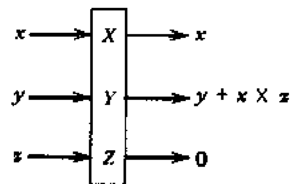


圆 $(m=1)$, $z\to 0$

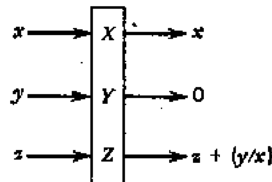
$K_1=\prod_{j=0}^{n-1}(1+\delta_j^2)^{1/2}$ 对 n 次叠代



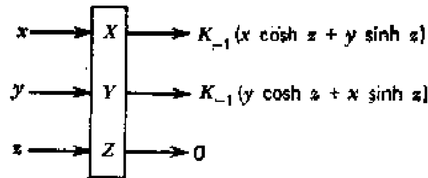
圆 $(m=1)$, $A\to 0$



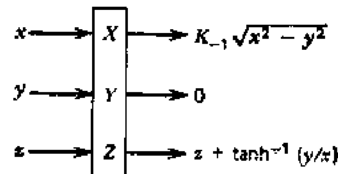
直线 $(m=0)$, $z\to 0$



直线 $(m=0)$, $A\to 0$



双曲线 $(m=-1)$, $z\to 0$



双曲线 $(m=-1)$, $A\to 0$

$K_{-1}=\prod_{j=0}^{n-1}(1-\delta_j^2)^{1/2}$，对 n 次叠代

图 11.8 使用 Walther 算法[35]来产生不同 CORDIC 函数的输入-输出方框图的描述

表 11.1　在三种座标系统中夹角 α_i 和半径因子 K_i 的方程式

座标系统 m	夹角 α_i	半径因子 K_i
1 （圆座标）	$\tan^{-1}\delta_i$	$(1+\delta_i^2)^{1/2}$
0 （直线座标）	δ_i	1
−1 （双曲线座标）	$\tanh^{-1}\delta_i$	$(1-\delta_i^2)^{1/2}$

注：δ_i 是增量变化

夹角的总的变化正好是增量变化的和，而半径的总的变化是增量变化的乘积。

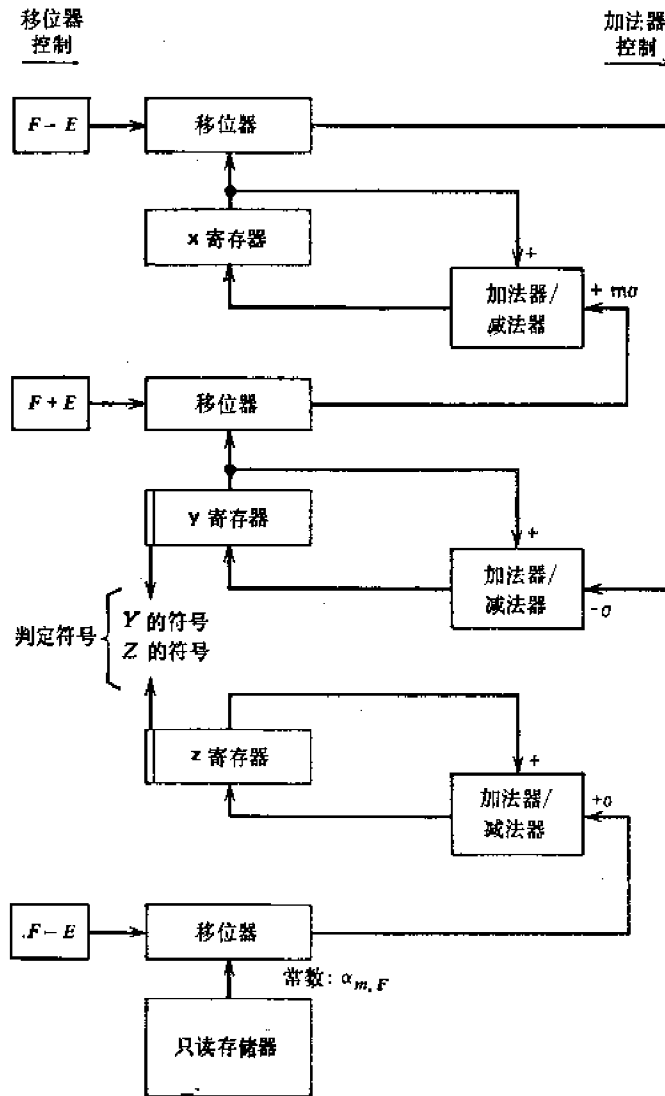


图 11.9　产生基本函数的 Hewlett-Packard 运算处理器的功能方框图（Walther[34]）

为了累计夹角的变化，要引入第三个变量 z。

$$z_{i+1} = z_i + \alpha_i \qquad (11.24)$$

解 11.19 式和 11.24 式中的差分方程，作 n 次叠代后可得

$$x_n = K \times [x_0\cos(\sqrt{m}) + y_0\sqrt{m}\sin(\sqrt{m})] \qquad (11.25)$$

$$y_n = K \times [y_0 \cos(\sqrt{m}) - x_0\sqrt{m}\sin(\sqrt{m})] \quad (11.26)$$

$$z_n = z_0 + \alpha \quad (11.27)$$

其中 α, K 在 11.23 式中已经定义.

图 11.8 归纳了不同座标系统的上述关系. 初始值 x_0, y_0, z_0 表示在每个方框的左边, 而最终值 x_n, y_n, z_n 表示在右边. 适当选择初始输入值,可以在输出端直接得到以下函数

$$\{x \times z,\ y/x,\ \sin z,\ \cos z, \tan^{-1} y, \sinh z, \cosh z, \text{和} \tanh^{-1} y\} \quad (11.28)$$

此外,以下函数可以通过恒等式间接地产生

$$\tan z = \sin z / \cos z \quad (11.29)$$

$$\tanh z = \sinh z / \cosh z \quad (11.30)$$

$$\exp z = \sinh z + \cosh z \quad (11.31)$$

$$\ln \omega = 2\tanh^{-1}[y/x],\ \text{其中}\ x = \omega + 1\ \text{和}\ y = \omega - 1 \quad (11.32)$$

$$\sqrt{\omega} = (x^2 - y^2)^{1/2},\ \text{其中}\ x = \omega + \frac{1}{4}\ \text{和}\ y = \omega - \frac{1}{4} \quad (11.33)$$

图 11.9 是 Hewlett-Packard 实验室中研制的使用上述统一 CORDIC 算法的硬件浮点处理器. 该运算处理器由三个相同的运算部件组成,它们并行操作,分别处理 x, y 和 z 的叠代. 每个运算部件包括一个 64 位寄存器,一个 8 位并行加法器/减法器,以及一个 48 到 8 的多路转换移位器.运算部件的组合体是用存储在一个 ROM 中的微程序控制的,ROM 中也存有夹角和半径的常数. ROM 的容量为 512 字, 每字 48 位,取数时间为 200ns. 处理器接收三种数据类型, 48 位浮点数, 32 位定点数以及 32 位整数. 所有函数计算到 40 位的精度(大约 12 位十进位数字). 表 11.2 列出大部分重要函数在 HP 处理器中的最长执行时间. 对通用计算机来说,这个速度是慢的, 但对于大多数计算器的应用来说, 这是颇有吸引力的.

表 11.2 在 HP 运算处理器中(图 11.9)不同基本函数的最长执行时间

函数	执行时间(μs)	函数	执行时间(μs)
取数	30	反正切	90
存数	15	双曲正弦	130
相加	40	双曲余弦	130
相减	50	双曲正切	190
相乘	100	反双曲正切	120
相除	100	指数	130
正弦	160	对数	120
余弦	160	平方根	100
正切	220		

11.5 Chen 氏收敛计算方法

在第八章中讨论的收敛除法,由 Chen[5] 加以普遍化, 以计算分数变量的指数, 对数, 比例,以及平方根. 这种方案的基本点是: 对一个数对 (x, y) 求共变换,使 $F(x, y)$ 不变; 当 x 趋向于已知值 x_ω, y 即趋向于所要求的结果. 以下讨论的是使用收敛方法的 Chen 氏硬件算法,用于计算 ωe^x, $\omega + \ln x$, ω/x, 以及 $\omega/\sqrt{x}$,这里 ω 和 x 是给定的数.

首先说明一些简要的函数理论基础，以后讨论特殊函数的算法，然后是实现这些任务的硬件设备。

考虑计算一个函数 $z_0 = f(x)|_{x=x_0}$，可以引入一个参数 y，形成一个具有两个变量的收敛函数 $F(x, y)$，使得

(a) 有一个已知初始值 $y = y_0$，而 $F(x_0, y_0) = z_0$。

(b) 存在一种方便的手段，把数对 (x_k, y_k)（对 $k \geq 0$）变换成 (x_{k+1}, y_{k+1})，以使 $F(x_{k+1}, y_{k+1})$ 不变。

(c) x 变换的程序收敛到一个已知目的值 $x = x_\omega$；相应的 y-变换收敛到 $y = y_\omega$，使 $y_\omega = F(x_\omega, y_\omega) = z_0$。

以上函数都可以在一个三维立方体中，用一条曲线表示在 $z = z_0$ 的平面内，并且通过图 11.10 中所示的点 $P_0(x_0, y_0, z_0)$。我们有

$$z_0 = f(x_0) = F(x, y) \tag{11.34}$$

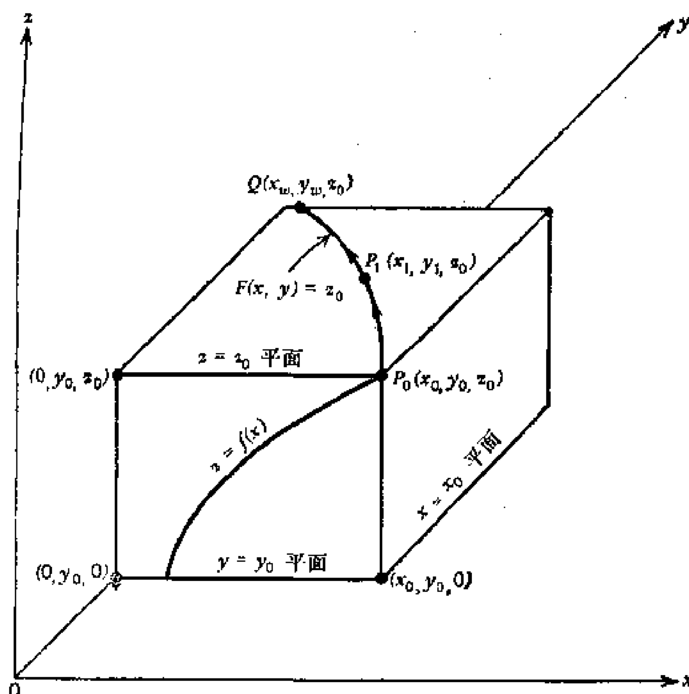


图 11.10　为计算基本函数的收敛函数的几何解释

在 (b) 中叙述的变换的不变性只是在下列情况下才有需要：即假如点 $P_k(x_k, y_k, z_k)$ 落在曲线 F 上，同样 $P_{k+1}(x_{k+1}, y_{k+1}, z_{k+1})$ 也是如此。或者我们可把它写成如下序列

$$\begin{aligned} z_0 &= f(x_0) = F(x_0, y_0) = F(x_1, y_1) = \cdots \\ &= F(x_k, y_k) = \cdots\cdots = F(x_\omega, y_\omega) = y_\omega \end{aligned} \tag{11.35}$$

条件 (c) 表示曲线 F 通过目的点

$$Q(x_\omega, y_\omega, z_0)$$

在 (b) 中，数对 (x, y) 的叠代变换表示这样一种法则，它把一个点从 P_0 开始，沿着曲线 F 通过 P_1，P_2 等，一直移动到目的点 Q。

逐次变换的数对，可以写成两个递归函数

$$x_{k+1} = \phi(x_k, y_k) \quad y_{k+1} = \psi(x_k, y_k) \tag{11.36}$$

变换法则ϕ或ψ的选择取决于价格-有效性的考虑．把ϕ规定为只是x_k的函数较为方便，对x_k的运算只是在某个闭合区间$[a, b]$内进行．x变换应该产生一个$x_{k+1} \in [a, b]$，使得

$$|x_{k+1} - x_\omega| < |x_k - x_\omega| \tag{11.37}$$

换句话说，点P_{k+1}应该比P_k更进一步地靠近目的点$Q(x_\omega, y_\omega, z_0)$．$\psi$的选择也可同样得到．这些选择将在下面用四种特殊的基本函数来说明．

计算基本函数的收敛算法

指数算法　$f(x) = \omega e^x$ 对于 $0 \leqslant x < \ln 2 = 0.693\cdots$

收敛函数　$$F(x, y) = y \times e^x \tag{11.38}$$

其中初始值 $y_0 = \omega$，以及目的值 $x_\omega = 0$

变换法则

$$\begin{cases} x_{k+1} = \phi(x_k) = x_k - \ln a_k \\ y_{k+1} = \psi(y_k) = y_k \times a_k \end{cases} \tag{11.39}$$

a_k 的选择将在以后讨论

对数算法　$f(x) = \omega + \ln x$ 对于 $\frac{1}{2} \leqslant x < 1$

收敛函数　$$F(x, y) = y + \ln x \tag{11.40}$$

其中初始值 $y_0 = \omega$，以及目的值 $x_\omega = 1$

变换法则

$$\begin{cases} x_{k+1} = x_k \times a_k \\ y_{k+1} = y_k - \ln a_k \end{cases} \tag{11.41}$$

比例算法　$f(x) = \omega / x$ 对于 $1/2 \leqslant x < 1$

收敛函数　$$F(x, y) = y / x \tag{11.42}$$

其中初始值 $y_0 = \omega$，以及目的值 $x_\omega = 1$．

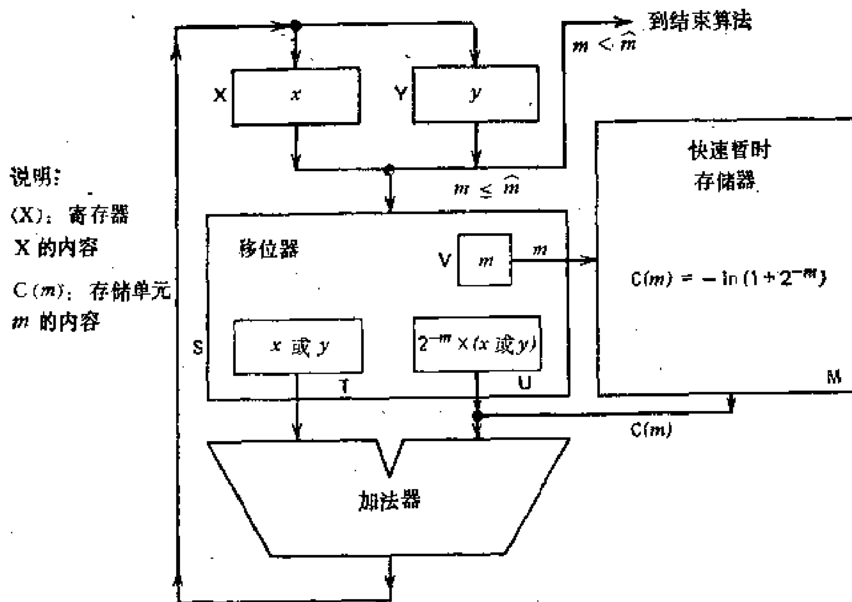


图 11.11　Chen 提出的计算基本函数的硬件运算处理器（Chen[13]）

变换法则

$$\begin{cases}x_{k+1} = x_k \times a_k \\ y_{k+1} = y_k \times a_k\end{cases} \tag{11.43}$$

平方根倒数算法 $f(x) = \omega/\sqrt{x}$ 对于 $1/4 \leqslant x < 1$

收敛函数 $$F(x, y) = y/\sqrt{x} \tag{11.44}$$

其中初始值 $y_0 = \omega$，以及目的值 $x_\infty = 1$。

变换法则

$$\begin{cases}x_{k+1} = x_k \times a_k^2 \\ y_{k+1} = y_k \times a_k\end{cases} \tag{11.45}$$

注意，$\sqrt{x}$ 可以在方程式(11.44)中用置 $\omega = y = x$ 来求得。

令 $k = n$ 是叠代的最后一步，使 $x \to x_\infty$ 以及 $y_n \to y_\infty$。以下所列的是**结束法则**，以及与以上各运算算法有关的**计算误差**。

结束法则

指数算法

$$z_0 = \omega \times e^{x_0} = y_0 \times e^{x_0} = (y_0 a_0) \times e^{x_0 - \ln a_0} = y_1 \times e^{x_1} = \cdots = y_n + y_n\mu \tag{11.46}$$

结束法则 $$x_n = \mu \to 0, \; z_0 \sim y_n + y_n \times x_n \tag{11.47}$$

相对误差 $$0 \leqslant \varepsilon \leqslant \mu^2(1 + 2\mu/3) \tag{11.48}$$

对数算法

$$\begin{aligned} z_0 &= \omega + \ln x_0 = y_0 + \ln x_0 = (y_0 - \ln a_0) + \ln x_0 a_0 \\ &= y_1 + \ln x_1 = \cdots = y_n + \ln(1 - \mu) \sim y_n - \mu \end{aligned} \tag{11.49}$$

结束法则 $$1 - x_n = \mu \to 0; \; z_0 \sim y_n - (1 - x_n) \tag{11.50}$$

绝对误差 $$0 \geqslant \delta \geqslant -\mu^2/2(1 - 2\mu/3) \tag{11.51}$$

比例算法

$$z_0 = \omega/x_0 = y_0/x_0 = (y_0 a_0)/(x_0 a_0) = y_1/x_1 = \cdots = y_n/(1 - \mu) \sim y_n + y_n\mu \tag{11.52}$$

结束法则 $$1 - x_n = \mu \to 0; \; z_0 \sim y_n + y_n \times (1 - x_n) \tag{11.53}$$

相对误差 $$0 \leqslant \varepsilon = \mu^2/(1 - \mu^2) \tag{11.54}$$

平方根倒数算法

$$\begin{aligned} z_0 &= \omega/\sqrt{x_0} = y_0/\sqrt{x_0} = (y_0 a_0)/(x_0 a_0^2)^{1/2} = y_1/\sqrt{x_1} \\ &= \cdots\cdots = y_n/(1 - \mu)^{1/2} \sim y_n + y_n\mu/2 \end{aligned} \tag{11.55}$$

结束法则 $$1 - x_n = \mu \to 0; \quad z_0 = y_n + y_n(1 - x_n)/2 \tag{11.56}$$

相对误差 $$0 \leqslant \varepsilon = -1 + 1/(1 - \mu)^{1/2}(1 + \mu/2) \leqslant 3\mu^2/8 \tag{11.57}$$

我们把 a_k 选择成如下形式

$$a_k = (1 + 2^{-m}) \tag{11.58}$$

使得与 a_k 相乘可以用一个移位和一个加法所代替。m 值通常选为在 $|x_k - x_\infty|$ 中领前的 1 位在二进位小数点右面的位数，但对于平方根倒数来说，需要增 1。表 11.3 归纳了以上算法，并且用 11.58 式代替 a_k，其中 p 代表第 m 位以后数位的模式，即，

$$0 \leqslant p \leqslant 2^{-m} \tag{11.59}$$

Chen 曾经设想过用一个统一的硬件处理器实现以上四种叠代算法。处理器表示在图

表 11.3　计算基本函数的 Chen 氏收敛算法小结

x_0的范围	函数 $f(x_0)$	$F(x_k, y_k)$	x_k	x_{k+1}	y_{k+1}	x_n	结束法则
[0, ln 2)	ωe^{x_0}	$y_k \times e^{x_k}$	$2^{-m} + p$	$x_k - \ln \cdot (1 + 2^{-m}) \sim p$	$y_k + 2^{-m} y_k$	$0 \leqslant \mu$ $\mu < 2^{-N/2}$	$y_n \times e^{\mu} \sim y_n + y_n \times \mu$
[1/2, 1)	$\omega + \ln x_0$	$y_k + \ln x_k$	$1-(2^{-m}+p)$	$x_k + 2^{-m} \times x_k \sim 1 - p$	$y_k - \ln \cdot (1 + 2^{-m})$	$1 - \mu$	$y_n + \ln(1-\mu) \sim y_n - \mu$
[1/2, 1)	ω / x	y_k / x_k	$1-(2^{-m}+p)$	$x_k + 2^{-m} \times x_k \sim 1 - p$	$y_k + 2^{-m} \times y_k$	$1 - \mu$	$y_n/(1-\mu) \sim y_n + y_n \times \mu$
[1/4, 1)	$\omega/\sqrt{x}$	$y_k/\sqrt{x_k}$	$1-2(2^{-m}+p)$	$x_k(1+2^{-m})^2 \sim 1 - 2p$	$y_k + 2^{-m} \times y_k$	$1 - \mu$	$y_n/(1-\mu)^{1/2} \sim y_n + y_n \times \mu/2$

注：N 是字长，p 应满足方程式 11.59

11.11 中，其中列有微操作．从 x_0 输入到寄存器 X，$\omega(=y_0)$ 输入到寄存器 Y 开始，计算四种函数的程序为：

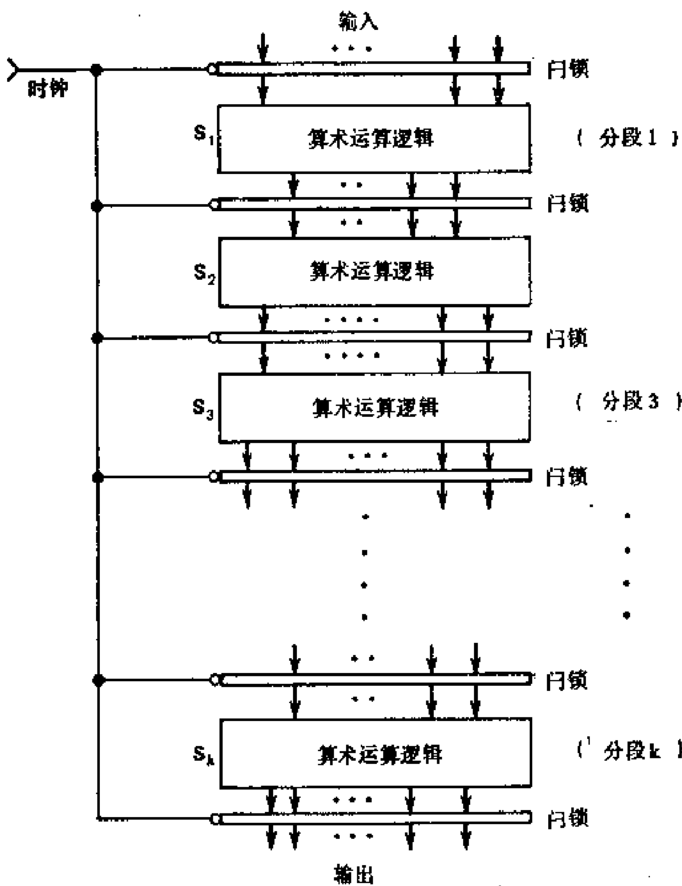


图 11.12　具有 k 个分段的运算流水线的功能结构，各个分段与快速闩锁相连接

$\omega \times e^x$：*abcdefgh*；然后，

$$(Y) + (Y) \times (X) \sim \omega e^{x_0} \tag{11.60}$$

$\omega + \ln x_0$：*abfdecgh*；然后，

$$(Y) + 1-(X) \sim \omega + \ln x_0 \tag{11.61}$$

ω / x_0：*abfdefgh*；然后，

$$(Y) + (Y) \times [1-(X)] \sim \omega / x_0 \tag{11.62}$$

$\omega/\sqrt{x_0}$：*abfdafdefgh*；然后，

$$(Y) + (Y) \times [1-(X)]/2 \sim \omega/\sqrt{x_0} \tag{11.63}$$

令 $T = N$，T 是一个普通的 N 位乘 N 位的乘法器的加法-移位次数．所要求的 $N/4$ 次叠代中，对于求平方根倒数，要做 $3N/4$ 次加法-移位，对其他三种函数来说，要做 $N/2$ 次加法-移位．于是计算时间可以估算出来，对平方根为 $3T/4$，对指数，对数以及 N 位分数的比例来说，则为 $T/2$．

11.6　流水线计算的概念

有两个术语经常被用来判定一台计算机的计算能力，一个是频宽，另一个是执行时间．**频宽**是在单位时间间隔内能够执行的任务的数目．**执行时间**是执行一个单任务所需

的时间长度．对于一台在同一时刻只执行一个任务的机器，执行时间就是频宽的倒数．在常规设计中，增加运算处理器频宽，大部分是由于使用快速逻辑电路以减少执行时间而达到的．举例说，快速加法器设计使进位传播延迟最小化．乘法器设计使很多数的加法可以同时进行．然而，电路工艺几乎已经到达其光速的极限．单是快速电路不能使计算能力有根本性的增长．

频宽问题的一个明显的解决办法是采用多运算部件，使得很多任务能同时执行．然而，直接复制很多硬件来做并行处理可能是不经济的，价格上也是不合算的．而流水线的方法是这样一种结构设计，它可以使频宽显著增加，而在硬件费用上只不过适当地增加一些．

流水线最初是从工业上装配线处理中发展出来的一种运筹学技术．流水线计算是把总的计算工作负荷细分成个别的任务，使它们按一定优先次序的规定，通过高速运算流水线以重叠的方式来执行．这种重叠的执行已经用在中央处理器的设计中，其中取指，译码，有效地址的计算，以及取下一指令的操作数，可以与当前指令的执行重叠地进行．如果准备一条指令化费的时间和执行时间几乎相等，那末重叠的处理器可比常规设计快一倍．这里，我们主要关心的是在复杂的运算功能中有顺序的任务的重叠执行．

流水线的运算部件可以概括地定义为很多称为“分段”的硬件资源的集合，这些分段组成一条线性装配线或者具有同步定时控制的流水线，这样细分的任务的流程可被流水线的逐个分段同时地执行．如图 11.12 所示，流水线是由连续的分段所表征．每个分段 S_i 本质上是一个专用的组合运算逻辑电路，其延迟为 τ_i，诸如求补器，加法器，乘法器以及其他．逐个分段都与数据闩锁（同步的寄存器）相连接，这些闩锁保存着逐个分段的输入和输出的各位数字．分段 S_i 的输出位 b_i 作为分段 S_{i+1} 的输入位 a_{i+1}，这样数据宽度应该

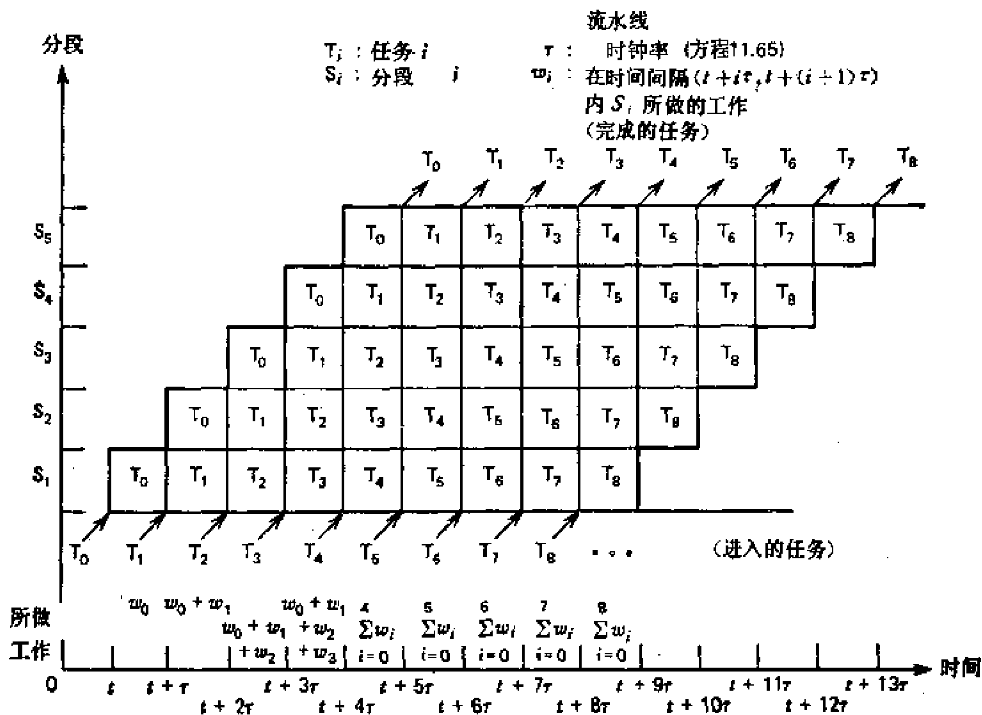


图 11.13　5 个分段的线性流水线中相连续的任务流（Chen[45]）

能够跨过边界，互相匹配

$$b_i = a_{i+1} \tag{11.64}$$

令 w_i 是分段 S_i 在时间间隔 τ_i 内所做的局部工作．不同的分段可以在不同时间间隔内完成它们的任务．为了调节流水线操作，同步时钟脉冲应该具有下列时钟周期

$$\tau = \text{Max}\{\tau_i\}_0^{k-1} + \tau_l \tag{11.65}$$

其中 τ_l 是单个闩锁的传播时间延迟．因此，流水线系统的速率取决于一个分段的最大延迟．每个闩锁，当它被外部时钟信号触发时，即把它的信息释放给在它右端的分段．上下两边的分段具有附加的闩锁，它处理全部 k 个分段的流水线的输入与输出．

为了把流水线充满或者漏空，需要 k 个时钟周期．每当流水线充满到一个稳定状态，每个分段忙于执行第 i 个任务 T_i（它需要 w_i 工作量）．于是流水线的稳态工作速率应为每个周期 $\sum_{i=0}^{k-1} w_i$．流水线的频宽是每个分段的最大执行时间 τ 的倒数，而不是整个流水线的执行时间或时间间隔 $k \times \tau$ 的倒数．因此，增加流水线的长度（分段的数目）并不影响频宽．

至于考虑任务的处理，在稳定状态流水线每个周期完成一个任务；在流水线上同时执行的 k 个任务，可以在 k 个相继的周期内做好，就如同一条汽车装配线一样．图 11.15 表示相连续的任务如何通过五个分段的运算流水线．流水线会使频宽大大增加，若干个线性流水线并行操作还可以处理多数据流．这种并行流水线系统可以应用在**单指令流和多数据流**（SIMD），或者**多指令流和多数据流**（MIMD）的计算机系统中．

11.7　实例研究 IV：在 TI 公司先进科学计算机中的流水线运算

在本节中，我们将讨论运算功能是如何被流水线处理器执行的．前节中描述了抽象的流水线模型，现在用 TI 公司先进科学计算机（ASC）中设计的实际流水线处理器来阐明．ASC 系统的圆视图见图 11.14．所有部件在照片中均已标明．

我们先从一个简单的流水线浮点加法部件开始．为执行这条指令，必须完成以下操作：

1. 减阶以求出差值．
2. 把较小的操作数右移，以调整其尾数．
3. 加尾数．
4. 把得出的和规格化．

按照不同设计目的，上述这些步骤的执行可以用一至四个分段的流水线部件来完成．一个流水线运算设计的目的，主要取决于以下因素：

1. 指令的系统（诸如定点还是浮点）以及字的长短（诸如半字长，单字长或双倍字长的操作数）．
2. 速度要求以及标量指令或向量指令．
3. 控制机构和指令顺序的安排．
4. 工艺限制和硬件价格．

这些因素的每一个都必须检验，因为每个因素对总的决定都有影响．大多数指令都

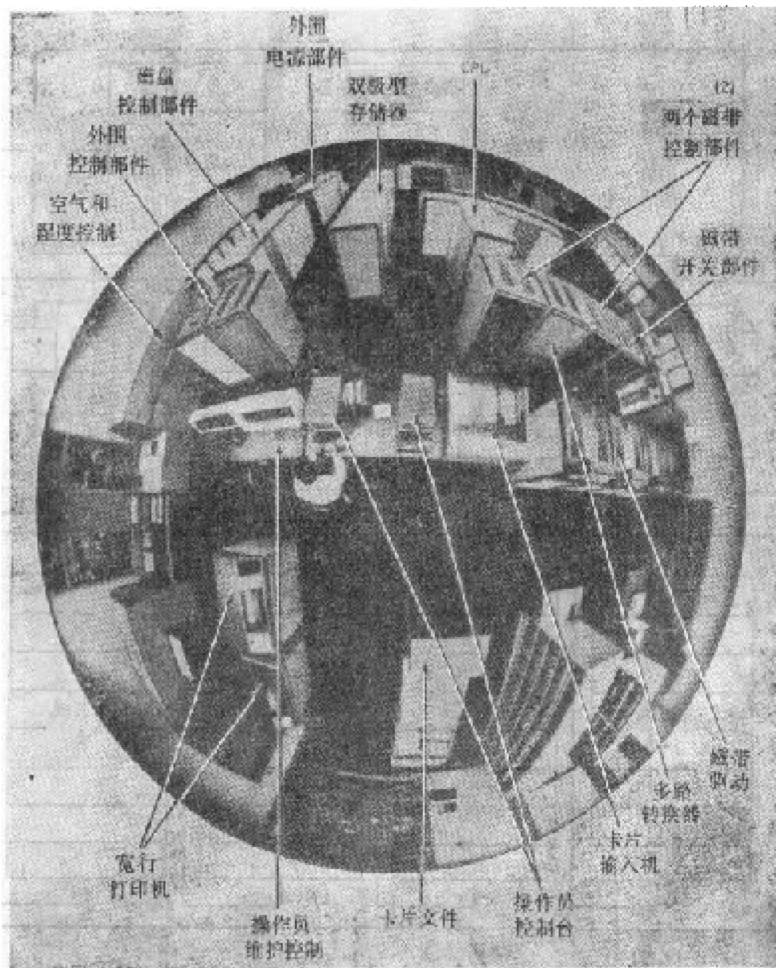


图 11.14 TI 公司的先进科学计算机 (ASC) 的圆视图 (1978 年 TI 公司提供)

认为是标量指令，它只对一个操作数或者对一对操作数执行运算，并产生一个数值的结果. 向量指令是指向量值的函数,它对一系列操作数进行运算，并产生一系列结果 (譬如两个向量相加);或者产生一个简单结果(譬如两个向量的点积). 向量运算指令的处理速率的基本计量,是每个操作数所需的机器周期的数目. 理想的运算流水线在每个周期内,可以对一个新的操作数集合进行运算. 对于产生具有 n 个分量的向量结果的向量，每个分量操作数一个周期的有效向量率，将需要 n 个周期并加上充满流水线所需的时间. 假如一条指令在两个周期内使用同一个流水线分段,因而不允许下一个操作数进入,则向量率将提高到每个分量操作数为两个周期. 在 **ASC** 中几乎所有的标量指令都有与向量相似的内容,即对大量数据执行操作. 如果主要关心的是高速,那末并行流水线部件应该用来处理向量指令.

图 11.15 表示 TI 公司 **ASC** 的中央处理器的功能方框图. 处理器由三种类型的功能部件组成: 指令处理部件 (IPU) 用于取指令,对操作码译码,并且产生操作数的地址. 它还包含 64 个可编址的寄存器. IPU 有流水线的功能. 存储器缓冲部件 (MBU) 用于从存储器取操作数,并把结果存入存储器. MBU 在执行向量指令时具有完善的控制，并且为向量指令计算所有所需的存储器地址. 处理器可以有多到四个流水线运算部件 (PAU),如图所示. 每个 PAU 具有以下要求的特点:

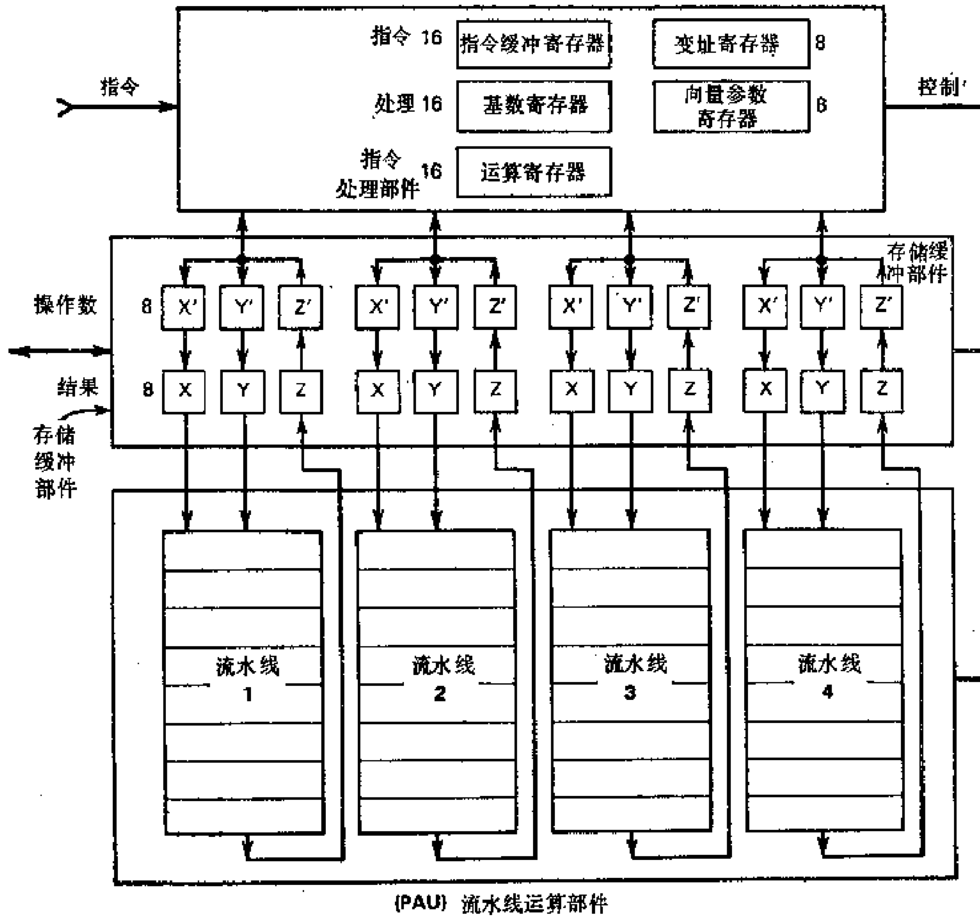


图 11.15 TI 公司先进科学计算机(ASC)的中央处理器的方框图 (Watson[56,371])

1. 每个 PAU 有八个带分路的不同的分段,执行很多定点或浮点格式的运算功能. 每个流水线有一个 MBU, 前次执行的结果可以发送回去供以后使用. 基本机器周期时间为 60 ns, 这表示每个分段的执行时间最多为 60 ns.

2. 该部件可以有效地执行标量指令.

3. 该部件也提供很快的向量点积. 每个向量指令对每个分量操作数的操作可在一个时钟周期内执行完毕. 可以有三维的下标. 每个 PAU 是一个 64 位的并行部件, 它能处理 64 位, 32 位和 16 位数据字.

4. 每个 PAU 的八个分段表示在图 11.16 中,其中使用的内部连接是为了执行浮点加法和定点乘法. 大部分标量和向量指令是可以由流水线部件来执行的. 只有双倍字长乘法和各种类型除法是例外,它们的执行需要多于一个时钟周期才能完成.

不同的运算指令允许取不同的途径通过 PAU, 并不限制其通路必须经过可操作的每个分段. 图 11.17 表示了在 TI 公司的 ASC 中, 为执行运算功能的各种可能的流水线内部级联. 乘法分段执行完整的 32 位乘 32 位的乘法. 它产生两个 64 位的输出数,作为伪"和"以及伪"进位",这两个数被累加器分段所接收并且相加,产生所要求的乘积. 累加器也采取措施,能通过它本身把其结果输送回去,通过重复相乘来执行双倍字长的乘法或除法. 回送的通路也用来完成定点向量点积的指令.

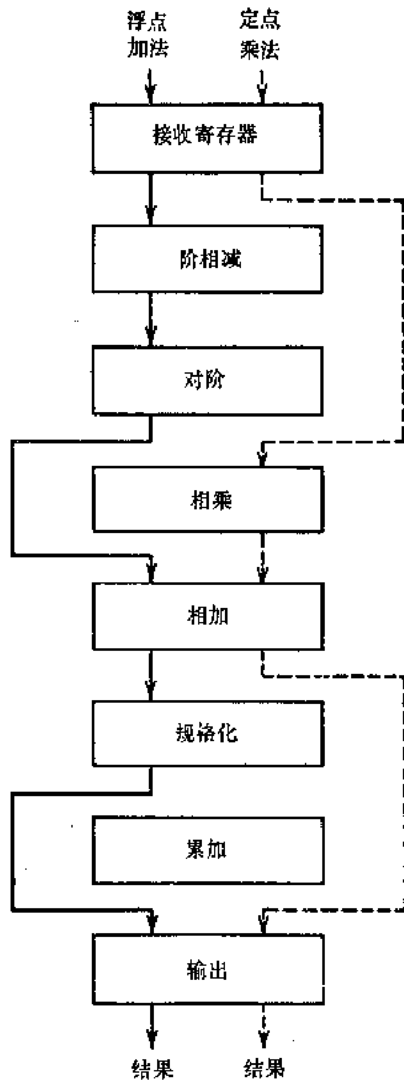


图 11.16 TI 公司 ASC 的运算流水线中的八个分段

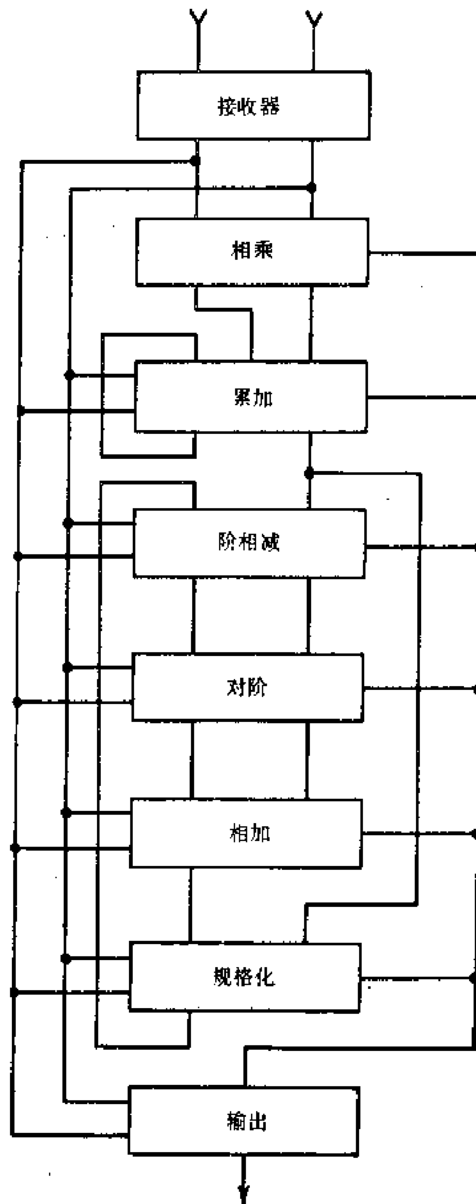


图 11.17 ASC 运算流水线执行不同指令时的可能有的分段内部联接。(Stephenson[33])

阶相减分段决定阶的差值，并且为**调整分段**提供移位次数的计数值，以调整浮点运算的分数部分。这种浮点运算要用到加法器。执行所有的右移时共享同一逻辑，这里包括逻辑移位，算术移位以及循环移位。**加法分段**是一个 64 位的两级 CLA 加法器。**规格化分段**在把所有浮点结果送到输出分段以前对它们进行规格化。所有左移指令共享这个硬件。此外，从定点转换成浮点以及它们的反转换也要用到规格化分段。

输出分段为所有指令的结果提供一个公共点。简单指令，诸如取数或存数，逻辑"与"，"或"等只使用这个分段。注意，阶相减分段的硬件也为下列操作所共享，如定点或浮点的比较操作，以及特殊向量指令（诸如检索最大值或检索最小值）。另一个流水线运算设计的实例研究是 CDC STAR-100 计算机，这将在 11.11 节中讨论。

11.8 通用的多功能运算流水线

本节讨论一个多功能单元阵列运算流水线．这个部件能执行所有基本运算操作，如乘法，除法，求平方根以及求平方．这个阵列使用一种把过去用过的 CAS 单元加以修改的运算单元，每个单元有六个输入和六个输出，如图 11.18 所示．

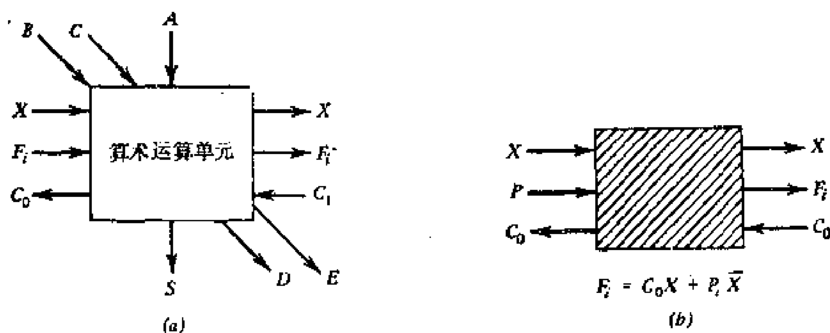


图 11.18　图 11.19 所示的通用运算流水线中所用的运算单元和控制单元．
(a) 修改过的可控加-减单元(式 11.66)；(b) 控制单元

运算单元的布尔方程式如下：

$$\begin{aligned} S &= [A\oplus(B\oplus X)\oplus C_1]F_i + A\bar{F}_i \\ C_0 &= (B\oplus X)(A + C_1) + AC_1 \\ D &= BC + CF_i = C(B + F_i) \\ E &= B + CF_i = (B + C)(B + F_i) \end{aligned} \tag{11.66}$$

在图 11.19 中，每一行分段的左端需要有一个控制单元．控制单元的布尔方程式为

$$F_i = C_0X + P_i\bar{X} \tag{11.67}$$

对于乘法和平方操作，控制线 X 置成逻辑“0”；对于除法和求平方根，则置成逻辑“1”．在介绍完整的单元阵列流水线以前，让我们先扼要地说明使用这些单元的四种操作．

乘法和除法

在阵列流水线中，右移和加/减方法是用来做乘法或除法的．被乘数或除数在通过流水线阵列中所有分段进行传送时保持不变．这一点是这样做到的：即把每个单独的运算单元的 C 输入位保持与式 11.66 中的 B 输入相同．

平方根

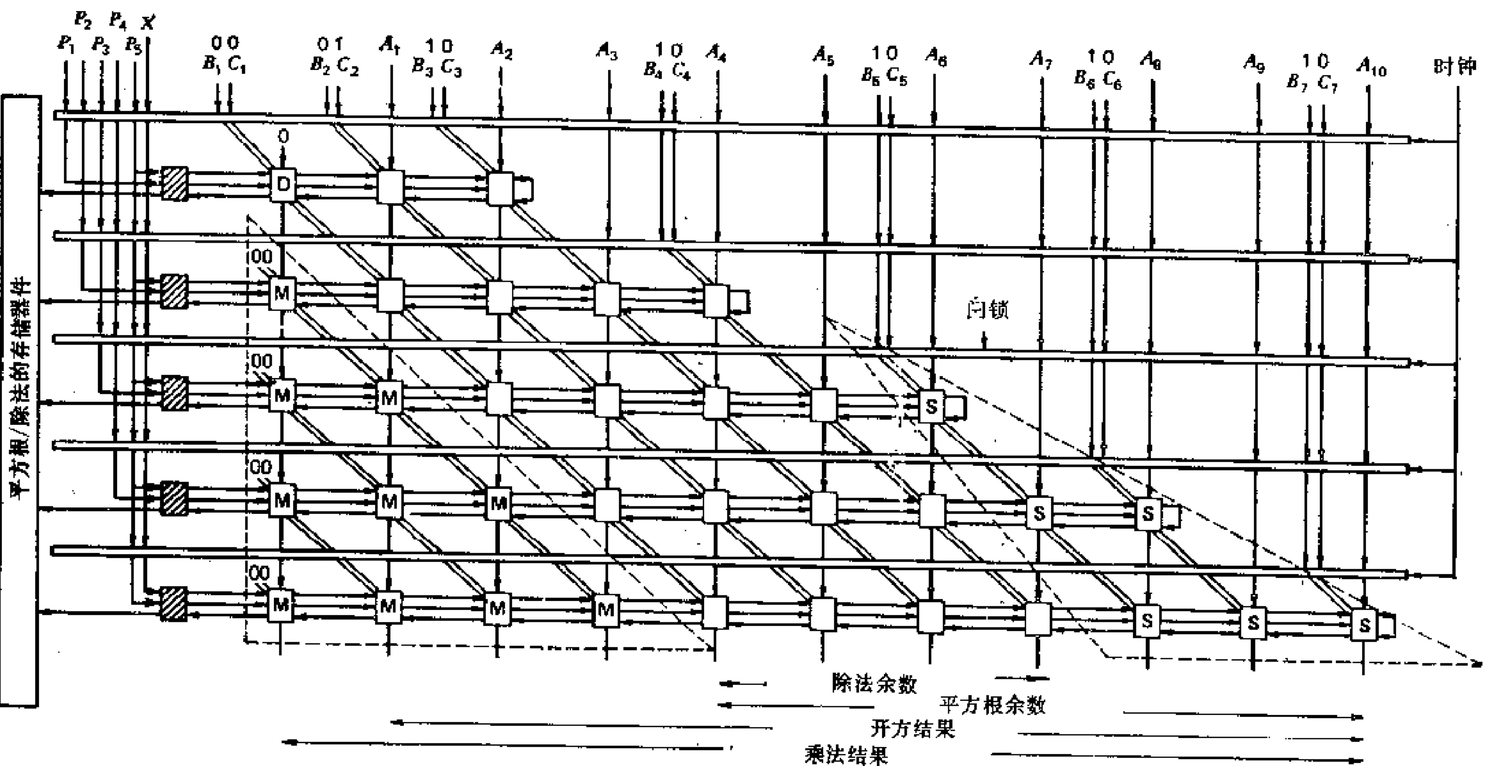
求平方根的实现方法与除法相似，只不过减法改变为表 11.4 中所示的特殊方式．令一个 $2n$ 位数 A 的平方根表示为

$$\sqrt{A} = F_1F_2\cdots F_i\cdots F_n \tag{11.68}$$

第一个分段永远从 A 的最高数位中减去 01．假如余数是正，F_1 将是“1”，否则是“0”．当 $F_i = 0$ 时，旧的余数将保留起来，如式 11.5 所述．当 $F_i = 1$ 时，逐个减数按表 11.4 所示的方法推导出来．

阵列流水线需要用三种不同方法改变每个分段的减数位：

1．维持 C 输入位为“0”，把任何分段中的“1”改变成下一分段的“0”．

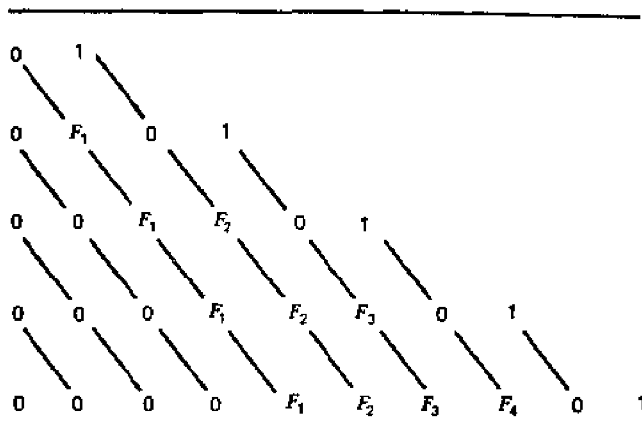


11.19 一个多功能流水线运算阵列的线路图(Kamal 等[20])

2. 维持 C 为1，把“0”改变为 F_i.

3. 维持 C 输入位与 B 相同，使减数位保持不变.

表 11.4 图 11.19 的流水线运算阵列中求平方根或求平方的不同分段上的减数



0	1								
0	F_1	0	1						
0	0	F_1	F_2	0	1				
0	0	0	F_1	F_2	F_3	0	1		
0	0	0	0	F_1	F_2	F_3	F_4	0	1

平方

执行平方操作也要用到表 11.4. 假定一个 n 位的数 $F_1F_2\cdots F_n$ 的平方为 S_n. 如果 $F_1=1$，S_1 取为 01，否则为 00. 如果 $F_2=1$，表 11.4 中第二行 $0F_101$ 加到 S_1 的各相应位上，给出 S_2. 相加或者跳越的过程取决于图 11.3 中 F_i 的数值. 因此，n 个这样的操作即完成一个给定 n 位数的求平方的过程.

图 11.19 中所示的流水线阵列是由 Kamal 等[20]提出的. 标有M的单元只是用来做乘法操作和溢出检测. 标有 S 的单元只用作求平方和平方根. 图示的阵列给出一个 10 位二进制数 $A=(A_1A_2\cdots A_{10})_2$ 的平方根. P 输入置成 0，x 置成 1，以求得 B 输入的补数，并使进位进入最低位（每行分段的最右单元）. 为了产生表 11.4 中的各项，B 和 C 输入到第一分段的闩锁的值为 00，01，10，10，10，10. 为了求 $2n$ 位数的平方根，该阵列需要 $n\times(n+2)$ 个运算单元和 n 个控制单元. 要计算其平方的数在 $P_1, P_2, \cdots, P_n$ 端送到部件中去. 当 $X=0$ 时，运算单元作为加法器工作，而控制单元把 P_i 转换成 F_i，于是完成了求一个二进制数 $P=(P_1P_2\cdots P_n)_2$ 的平方.

阵列把被乘数 B 和乘数 P 相乘. X 和 A 输入置成零，并使所有 C 位与相应的 B 输入相等. 考虑这种情况，当 B 和 P 各为四位，P_1 保持为“0”，则从第二分段开始做四次连续的加法，而 P 的其他四位给出乘数的值. 最后乘积是作为最后分段的输出而获得的. 该阵列也能执行 $A+B\times P$ 的运算，其中 $A=(A_1A_2\cdots A_7)_2$. 一般说，为了求 $2n$ 位数的平方根而设计的阵列流水线，可以把两个数 B 和 P 相乘，如果 n 是奇数，则 B 和 P 各为 $(n+3)/2$ 位；如果 n 是偶数，则 B 为 $(n+2)/2$ 位，P 为 $(n+4)/2$ 位.

阵列流水线可使一个 7 位数 $A=(A_1A_2\cdots A_7)_2$ 被一个 4 位数 $B=(B_1B_2B_3B_4)_2$ 相除，给出一个 4 位的商和一个 4 位的余数. 在这种情况，控制线 X 置成 1，P 输入置为零. C 输入保持与 B 输入相同. 标有 S 的运算单元在除法过程中并不使用. 把 00 输入送给 B_5C_5，B_6C_6 和 B_7C_7，才有可能执行补码除法. 和乘法一样，除法过程也从第二分段开始. 这里要求加到第一分段的控制单元的进位 C_0 为零. 通常这种阵列流水线可使一个（$n+$

2) 位的数 A 被 B 除. B 的最大位数为 $(n+3)/2$ 或 $(n+2)/2$，分别取决于 n 是奇数或偶数.

流水线部件的时钟频率由最后分段的处理延迟时间来决定，最后一个分段用了 $2n+1$ 个运算单元

$$\tau = (2n+1)\tau_a + \tau_c + \tau_l \tag{11.69}$$

其中 τ_a, τ_c 和 τ_l 分别是每个运算单元，控制单元和闩锁寄存器的延迟.

11.9 流水线快速傅里叶变换处理器

本节中，我们应用前面和本章学过的计算和设计的技术，讨论一种大家很熟悉的快速傅里叶变换 (**FFT**) 算法的硬件可能实现的方法. **FFT** 在现代数字信号处理中起着重要作用. 离散 **FFT** 对可以写成如下形式:

$$X(n) = \frac{1}{N}\sum_{m=0}^{N-1} X(m) \times W^{mn}$$

$$x(m) = \sum_{n=0}^{N-1} X(n) \times W^{-mn} \tag{11.70}$$

其中对 $n=0,1,\cdots,N-1$, $m=0,1,\cdots,N-1$, W 代表单位值的 n 次复数根,

$$j = \sqrt{-1}$$

$$W = e^{-j2\pi/N} \tag{11.71}$$

FFT 算法是计算以上复数级数的 **FFT** 对的一系列计算步骤. 为叙说简便，我们只讨论具有 $N=2^\lambda$ (λ 为某整数) 的基数为 2 的 **FFT** 算法.

对于特殊情况 $N=2^3=8$，用信号流程图表示的 Cooley-Tukey **FFT** 算法如图 11.20 所示. 图中的节点代表进入节点的复数输入的加法运算符. 联接各节点的流动通路中所附的常数指示出旋转向量 W^ϕ (假如单位值除外). 这种方法也称做按时间均分算法. 与每个节点有关的必要的计算 A_{ij} 具有以下形式

$$A_{ij} = a_i \times W^{\phi_i} + a_j \times W^{\phi_j} \tag{11.72}$$

其中 a_i 与 a_j 是复数输入变量，而 ϕ_i 与 ϕ_j 分别是相关的旋转向量. 这种 **FFT** 算法中的初始数据是以按位倒置的方式排序的，其中每个复数数据点的倒置二进位序数是用来作为输入的次序，而变换器的结果则是为正常次序.

按时间均分算法列出的方程，在 $\log_2 N$ 级中的每一级，存在 $N/2$ 个计算对，其中 W 的二个幂次的差正好为 $N/2$. 因为 $W^{N/2}=-1$，在每个节点对上的计算可以写成为

$$A' = A + B \times W^\phi$$

$$B' = A - B \times W^\phi \tag{11.73}$$

其中 A' 和 B' 是用前级变换结果表示的当前变换级的结果. A, B 和 ϕ 作为在计算序列中的标记. 图 11.21 表示计算方程式 11.73 的运算线路. 下标 R 和 I 分别相应于实数和虚数部分. 注意这两部分可以并行计算.

方程式 11.73 的实数和虚数部分可以分别写成

$$A'_R = A_R + B_R\cos\phi - B_I\sin\phi$$

$$B'_R = A_R - B_R\cos\phi + B_I\sin\phi \tag{11.74}$$

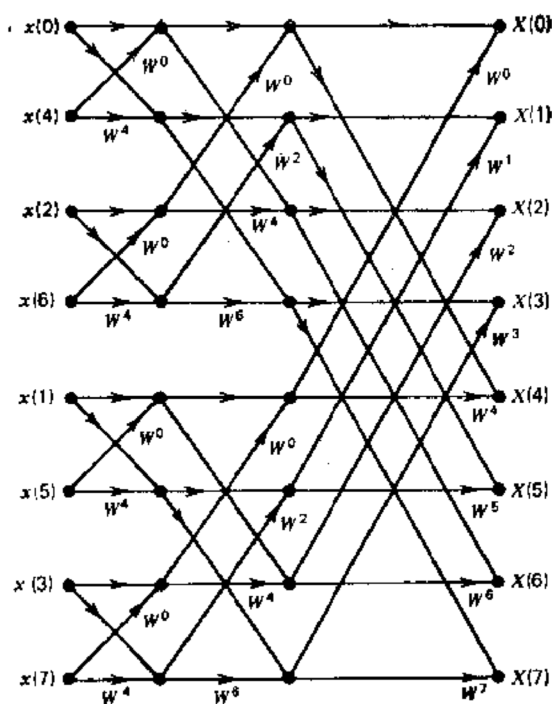


图 11.20 Cooley-Tukey 的具有 $N=8$ 个取样点的基数为 2 的 FFT 算法

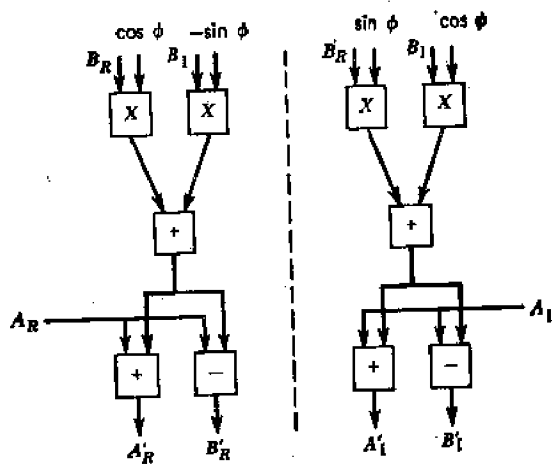


图 11.21 为实现算法的全蝴蝶式单元（BE）的线路框图

以及

$$A'_I = A_I + B_R \sin\phi + B_I \cos\phi$$

$$B'_I = A_I - B_R \sin\phi + B_I \cos\phi \tag{11.75}$$

其中 $A = A_R + jA_I$ 以及 $B = B_R + jB_I$ 是输入变量．为计算方程式 11.74 或 11.75 的运算线路的详细方框图见图 11.22．因为右半和左半是相同的，两个这样的线路就可以实现图 11.21 中的设计．注意实际的乘法由 8×8 乘法器执行，产生每个操作数所需的 16 位精度．

图 11.21 所示的运算部件称做蝴蝶式单元（BE）．蝴蝶的一翼在图 11.22 中实现．除了输入不同以外，两个翼是相同的．一个完整的 **FFT** 处理器共需要 k 个 BE，其中

$$k = \frac{N}{2}\log_2 N \tag{11.76}$$

可以把这 k 个 BE 安排成如图 11.23 所示的流水线 **FFT** 处理器．这个流水线由 $\log_2 N$ 个分段组成，每个分段有 $N/2$ 个 BE．分段到分段之间的内部联接已由图 11.20 中的数据流图说明．所有这些分段的旋转向量 $\sin\phi$ 和 $\cos\phi$ 可以用查表方法检索出来，表格放在 ROM 中． 图 11.20 表示 $N/2$ 个不同旋转向量如何被读出来，去处理 N 个抽样的一组信息．实际上，如果它们是按最后分段所要求的次序产生的，那么前面分段所要求的旋转向量是在需要它以前的适当时刻，靠选通这个表格来获得的．这样，每个分段所要求的正弦和余弦可以这样取得：即为每个运算 BE 提供一个寄存器，而所有寄存器由公共的总线来驱动．第一次变换的输出，在接收 N 组最后的数据抽样以后立即可以得到．

另一种实现蝴蝶式单元的方法是使用 CORDIC 算法，用它来产生每个分段中所用的权因子 $\cos\phi$ 和 $\sin\phi$．图 11.24 表示 CORDIC 全蝴蝶设计的原理框图．把复数量 B 旋转

16 × 16 相乘

实数或虚数部分

图 11.22　在图 11.21 中的半个蝴蝶式单元的运算线路框图（Larson[20]）

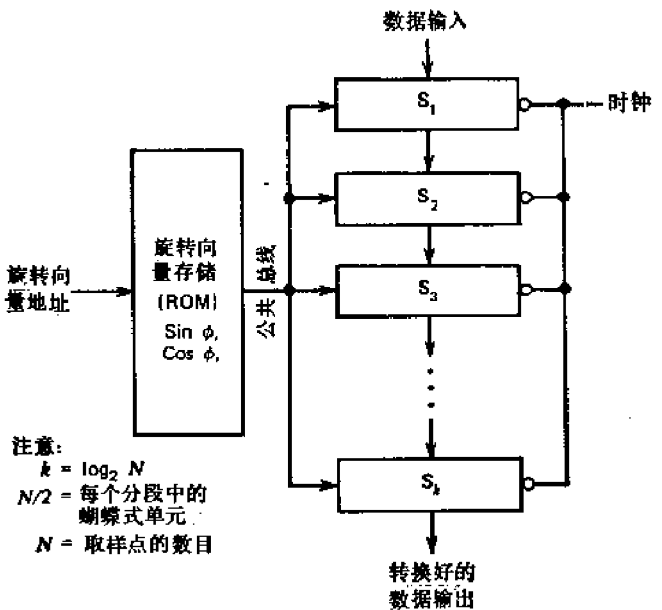


图 11.23　流水线 **FFT** 处理器的结构

一个角度 $\Phi=(2\pi/N)\phi$（或者 $BW^{\Phi}=(x+jy)W^{\Phi}=(x+jy)e^{-j\Phi}=Be^{-j\Phi}$）的基本 CORDIC 算法，由下面的叠代公式给出：

$$x_{i+1}=x_i+a_iy_i\times 2^{-i};\quad y_{i+1}=y_i-a_i\times a_ix_i\times 2^{-i};\quad z_{i+1}=z_i-a_i\times\theta_i \tag{11.77}$$

其初始值 $x_0=x$，$y_0=y$，$z_0=\Phi$，而如果 $z_i\geqslant 0$，则 $a_i=1$；否则 $a_i=-1$. 这 16 个符号系数 $a_i(i=0,1,\cdots,15)$ 是由 Φ 级联产生的，而其中 15 个输入常数 θ_i（对于 $i=0,1,2,\cdots,14$）有以下数值

$$\theta_{i+1}=\tan^{-1}(2^i) \tag{11.78}$$

对应于 $i=0,1,\cdots,\mu-1$，这个叠代要运行 μ 次循环，其中 μ 是每个操作数 x，y 或 Φ 的位数. 此图表示出一个字长 16 位的 16 个加法器级. 在级联的各级之间，用硬接线方式通过内部联接可实现 2^i 乘法器，它只做简单移位操作. 对于各种蝴蝶式设计的细节，读者可以参看 Larson 的博士论文[21].

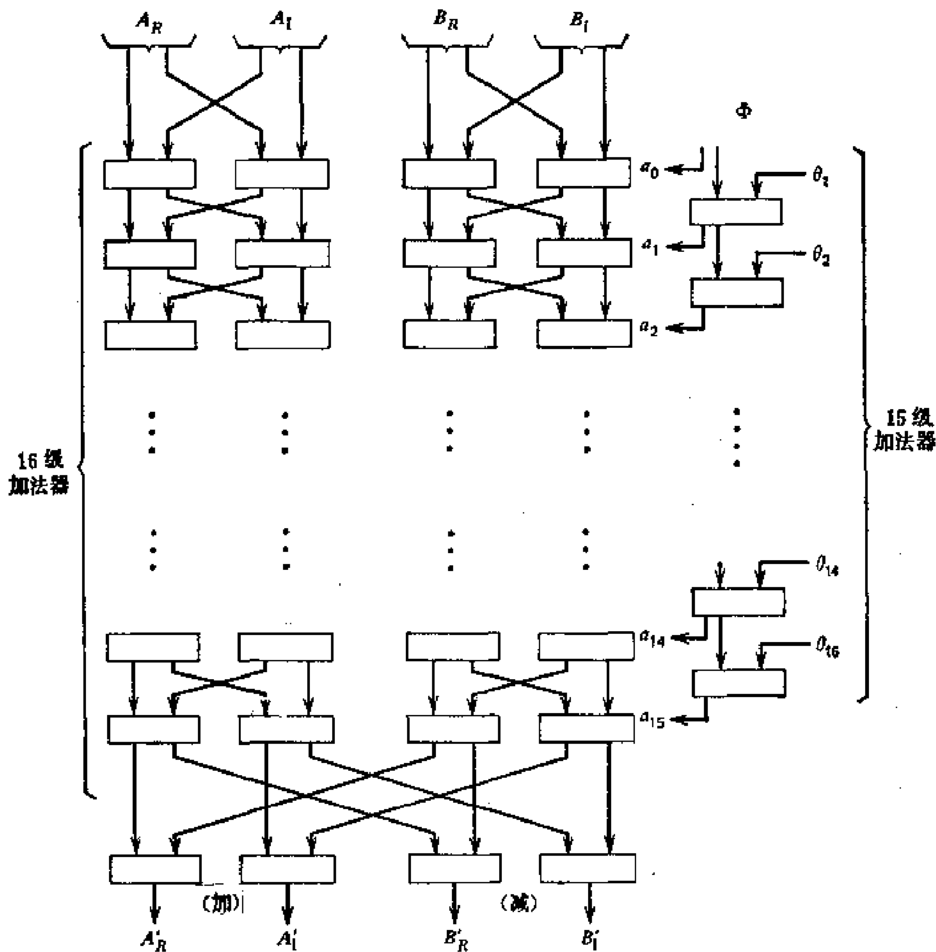


图 11.24 CORDIC 全蝴蝶式设计的运算线路框图（Larson[21]）

11.10 运算处理器中的误差控制

本节中，我们主要介绍逻辑故障种类以及对付在运算电路中这些逻辑故障的可能方

法．有各种原因会引起运算处理器出错．诸如由于过热而使电路元件误动作，接触不良，固定类型故障，电磁辐射干扰，机械震动以及其他．数字电路中的逻辑故障可以有很多不同方法来分类．图 11.25 表示逻辑故障的树形分类．

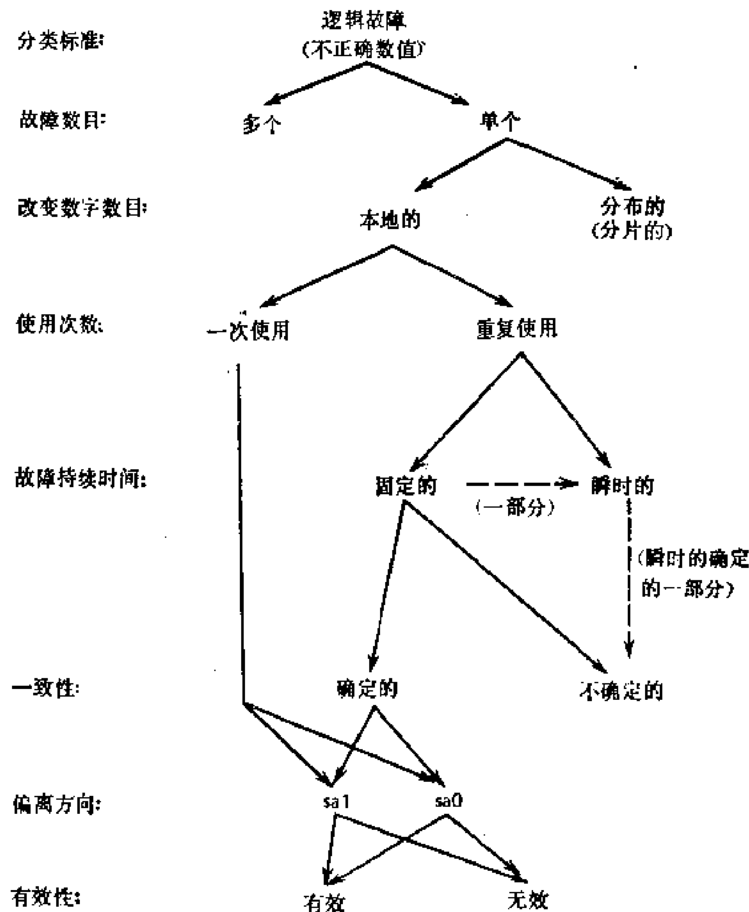


图 11.25　运算电路中逻辑故障的分类

单个故障指的是在电路中发生一个不正确的变量(终值)．当有多于一个变量终值与其所要求的值不符，我们认为电路中已有多个故障．所谓不符，我们是指当希望为零时出现逻辑“1”，以及相反情况．当其直接效果改变一个数位的值时，这个单一的故障就是本地的故障．否则，就会产生分布的或分片的故障．分布故障的效果有时可以是若干单个故障累积的效果所引起的．

在实际情况中，本地单个故障发生的概率最高．**一次使用故障**指的是这样一种故障条件，它在执行一个运算算法时，只是在电路的一次非重复使用中存在．**重复使用故障**是当有故障的电路在计算过程的一个叠代循环中重复使用时发生的．按照其逻辑值的偏离方向，一次使用类型的故障可以进一步细分成两种类型．固定在 0(s_a0) 以及固定在 1(s_a1) 的故障分别指的是永久性地产生故障值“0”和“1”．固定类型的误差可以是有效的或是无效的，取决于这些误差在某个特定时刻内是否与所要求的逻辑值相一致．

重复发生的故障可以是固定的，也可以是瞬时的．如果它在有故障的部件中并不是在所有叠代使用中都会发生，那末它就是瞬时的；否则，就称作固定的．我们可以把瞬时

的故障看作是在某些操作循环中无效的固定故障. 因此某些研究者把瞬时的故障当作固定故障的一个子集来对待. 如果固定故障 s_a0 或者 s_a1 只有一种形式(不是两种都有)出现,那末叫做确定的故障. 不确定故障可以是 s_a0, 或是 s_a1. 某些工程人员简单地称不确定故障为固定在 x 的故障 (s_ax),其中 x 可以是 0, 或是 1. 一个 s_ax 故障与发生在相同终值变量的两个瞬时故障 (s_a0 和 s_a1) 具有同样的效果. 因此, 不确定故障可以认为是瞬时的确定故障的一个子集. 分类树中的虚线表示了这个子集的关系.

与以上在运算电路中的故障条件作斗争的方法可以简要介绍如下. 在有故障的电路的误动作被检测以后以及在它被修复以前, 可以用备用件替换有故障电路. 替换通常是在逐步降格基础上进行的,其中余下的功能元件用开关技术重新组合,使有故障元件不能操作,它们的工作任务暂时指派给其他工作部件,直到恢复过程完成为止. 逐步降格方法的优点是在计算没有完成时不会停止. 但由于重新组合后的部件往往是负荷过重, 而且执行作业的排队时间将会较长,所以虽然计算仍将继续,但在恢复时期内是在降低的水平上进行的.

另一种常用的方法是选择有效的运算内部编码方式来检出运算故障, 然后通过复杂的误差纠正过程来自动纠正它们. 这将要求附加的硬件,如编码器和译码器. 通常,采用冗余技术来屏蔽输出误差. 为与运算误差作斗争的误差码和编码技术在图 11.26 中说明.

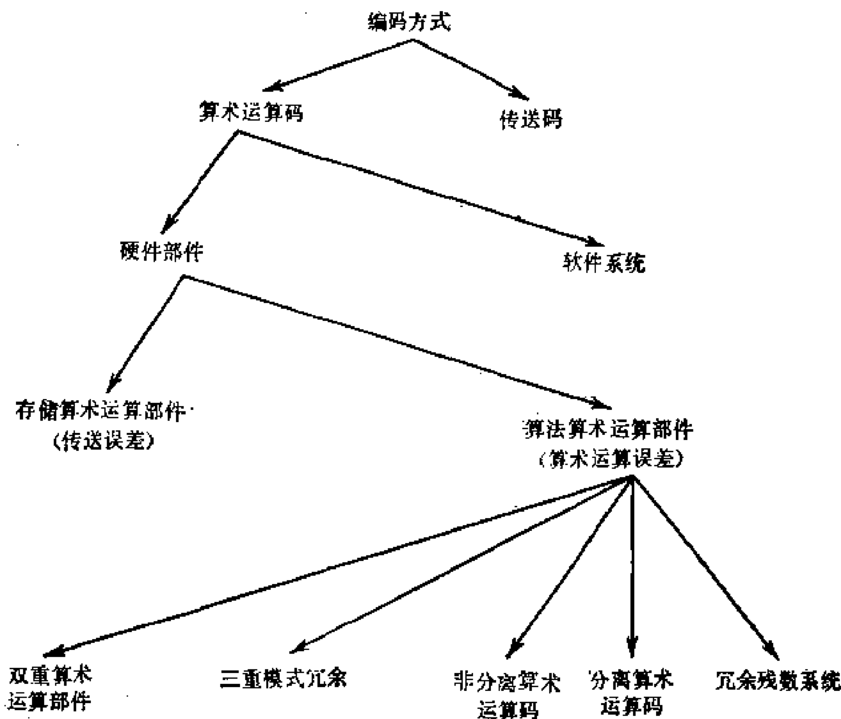


图 11.26 对付运算误差的编码方式

用在计算机系统中的内部码有两种, 算术运算码和传送码. 我们主要关心的是算术运算码的使用. 误差编码方案可以用硬件,也可以用软件来实现. 和以前一样,我们更关心硬件方法. 硬件编码方式对算法和存储的运算部件都可以应用. 前者产生算术运算误差,后者产生传送误差. 奇偶位校验码是典型的存储码.

为克服算法部件中的误差,通常采用以下方法中的一种或多种:

1. 直接复制一套硬件部件可提供检测单个误差的能力，但不能纠正。

2. 三重模块冗余可提供三中取二的多数表决，以克服单个误差，见图 11.27.

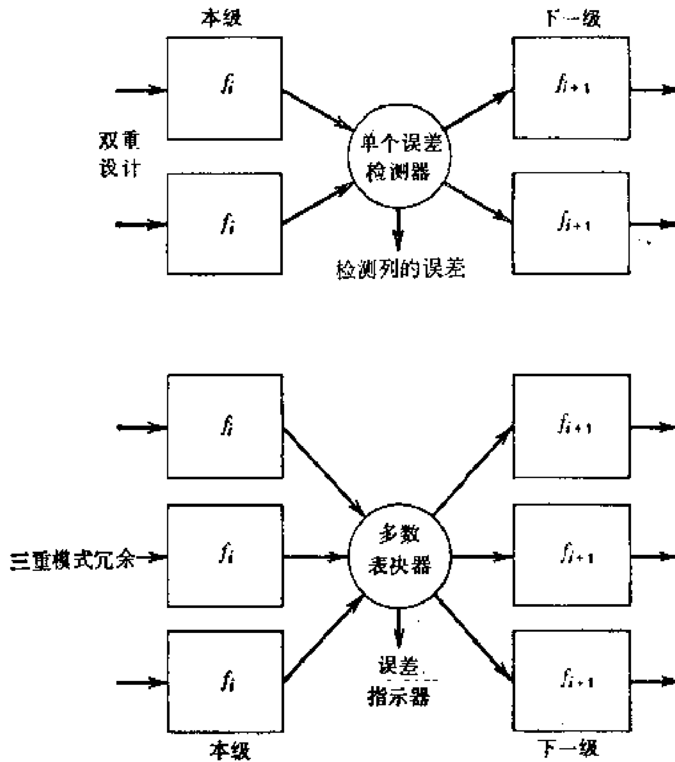


图 11.27 在运算处理器中用于误差控制的两重和三重模块冗余方案

3. 非独立的算术运算码，把检验位设置在被编码的字中. 表 11.5 所示的 $5N$ 码，表示非独立码的一个例子. 其中每个码字，$(5N)_2$，是把给定的三位信息字 $(N)_2$ 乘 5 所形成的，并把所得到的积表示为 6 位的二进位码. 注意，原来的信息位和检验位在 $5N$ 码字中并不是独立的.

4. 独立的算术运算码在提供检验能力时，不会弄乱给出的信息字. 在表 11.5 中所示的 $(N, |N|_5)$ 码提供独立码的一个例子. 其中检验码是靠把二进位编码 N（模 5）附加在原来信息码中产生的.

5. 使用冗余的残余数系统提供一种内在的纠错能力. 这种算术运算码扩展了残余编

表 11.5 误差控制用的独立的与非独立的运算码

十进位 N	二进位 N	表示 N 的二进位运算码	
		独立的 $[N, \|N\|_5]$ 码	非独立的 $5N$ 码
0	0 0 0	(000,000)	0 0 0 0 0 0
1	0 0 1	(001,001)	0 0 0 1 0 1
2	0 1 0	(010,010)	0 0 1 0 1 0
3	0 1 1	(011,011)	0 0 1 1 1 1
4	1 0 0	(100,100)	0 1 0 1 0 0
5	1 0 1	(101,000)	0 1 1 0 0 1
6	1 1 0	(110,001)	0 1 1 1 1 0
7	1 1 1	(111,010)	1 0 0 0 1 1

码的概念.

有兴趣的读者如欲深入了解能用来控制运算处理器中误差的各种误差编码方案，可以研究 Rao[30] 的书.

11.11 实例研究 V: CDC公司 STAR 100 流水线运算处理器

控制数据公司 STAR-100（串-阵列）计算机是一个流水线处理器，它的结构是围绕着一个 4M 字节（可选 8M 字节）高频宽的存储器设计的. STAR-100 计算机包括很多先进的设计特点，诸如流处理，虚拟编址，硬件宏指令，以及一个半导体存储器寄存器堆. 流水线运算部件是专门为可变字长的数据流能进行顺序的或并行的操作而设计的，如单个二进位，8 位字节以及 32 位或 64 位浮点操作数和向量. 不同的数据长度可以充分利用存储器的频宽以及运算流水线. 虚拟编址使用高速映象技术，把一个逻辑地址转换成绝对的存储器地址. 在数据流工作方式，系统具有每秒产生 1 亿个 32 位浮点结果的能力. 流水线处理器以及相联存储器总线的设计方式将在下面讨论.

STAR-100 处理器、存储器结构的方块图见图 11.28. 磁心存储器的访问周期为 1.28μs. 它有 32 个交叉存储体，每个存储体包含 2048 个 512 位的字. 存储周期分成 32 个小周期，每个小周期为 40ns，它也等于运算流水线的周期时间. 因此，32 个体的存储系统提供的总频宽为每个小周期有 512 位数据. 512 位可以细分成四个数据流，每个为 128 位；两个为操作数，一个为结果，最后一个为输入输出请求和控制向量. 这四个数据总线结构允许流水线可工作到每秒 1 亿个结果的最高速率.

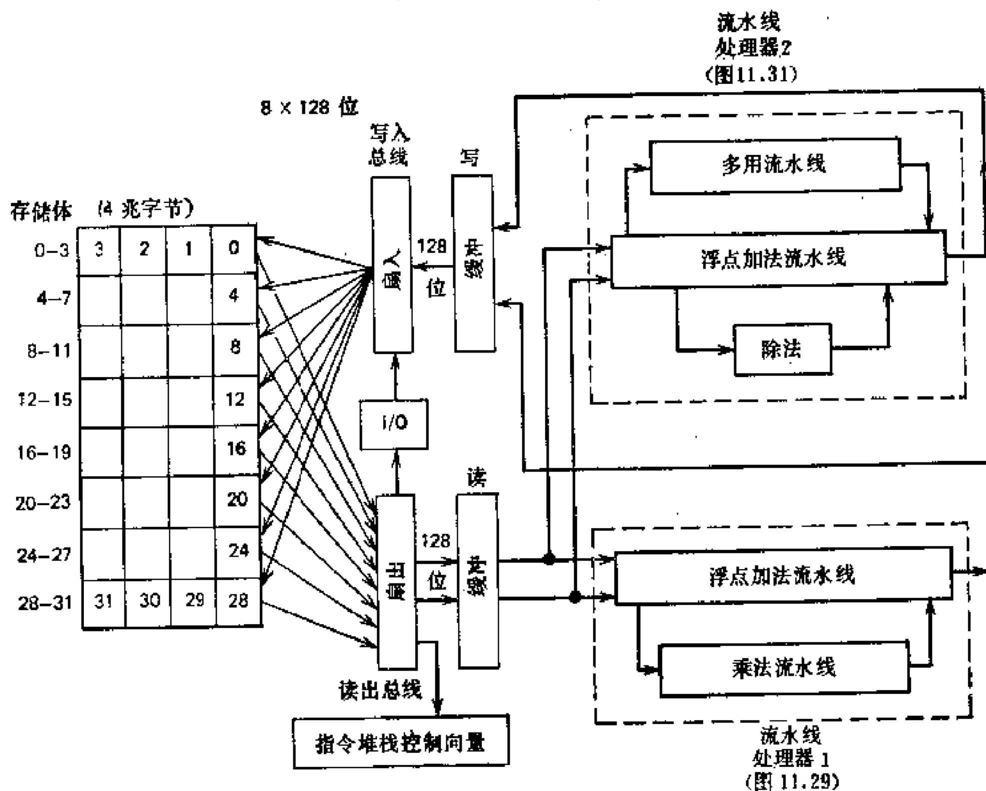


图 11.28 CDC STAR-100 的存储体，流水线处理器，以及数据通路的框图（Hintz 和 Tate[14]）

读出和写入缓冲器被用于同步四个工作总线.存储器请求时用缓冲器分隔成八个体，这样访问存储器的冲突可以显著减少. 因此不管在四条工作总线上地址如何分布，最高的流水速率可以保持.

在 STAR-**100** 中浮点运算是用两个独立的流水处理器执行的. 流水处理器 1 包括一个 64 位流水线浮点加法部件以及一个 32 位流水线浮点乘法部件，见图 11.29. 64 位浮点加法流水线主要包括四个级联的分段. 第一分段比较阶，并保存其较大的一个.阶差用来为第二分段中的移位的计数，其中具有较小的阶的分数，按照移位计数进行右移. 在第三分段，将移位的和不移位的分数相加，其"和"以及较大的阶被控制去第四分段，后者选择所需的"和"的上半部和下半部，并把结果传送到结果数据总线. 这样有可能把 64 位浮点加法流水线分解成两个独立的 32 位流水线，而只需很少的附加硬件. 流水线分解的概念广泛地用在 STAR-**100** 中，使机器除了能进行全长 64 位的操作以外，还能进行半宽

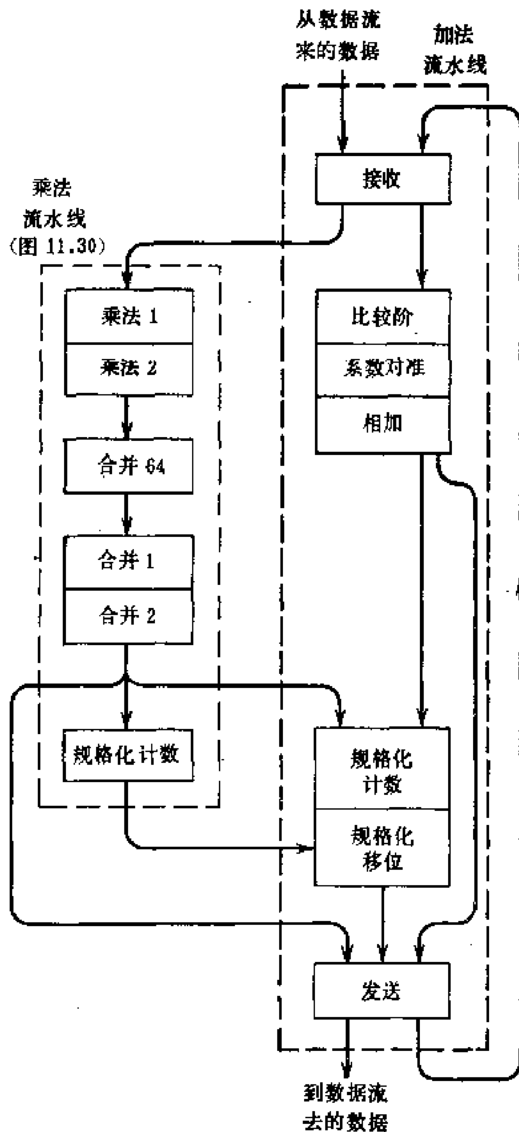


图 11.29 在 STAR-100 计算机中浮点运算流水线处理器 1 的原理图[7]

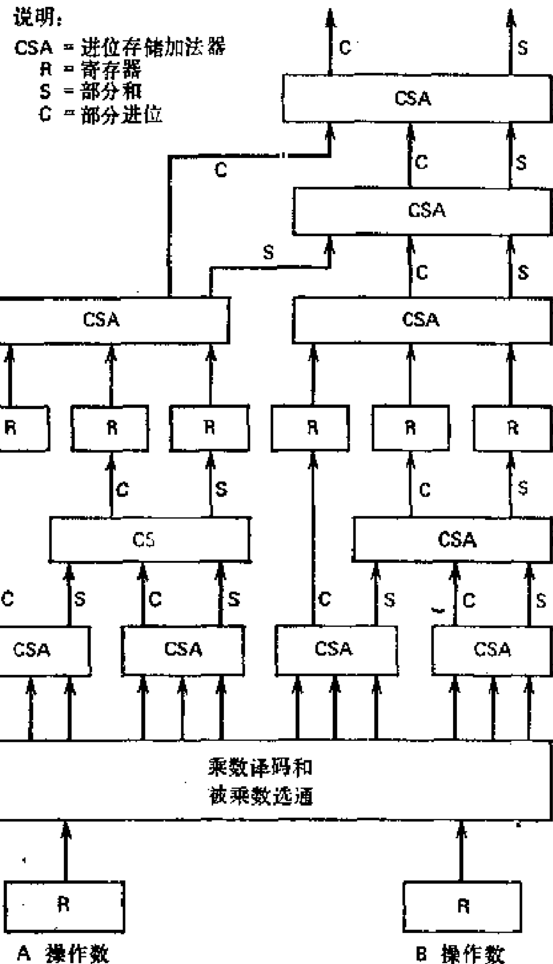


图 11.30 用在 STAR-100 中的基本的 32 位浮点乘法流水线

(32 位)的运算操作.

图 11.30 表示在 STAR-100 计算机中所用的 32 位乘法流水线的详细结构. 操作数 A 和 B 被控制送入乘数译码器以及被乘数控制网络. 乘数被译码成 12 个两位的小组.被乘数的倍数被控制送人逐个分段中的进位存储加法器 (CSA),最后合并成为两个输出数,部分被加数以及部分进位. 当这两个数相加在一起时,即形成两个输入数 A 和 B 的乘积. 在浮点乘法以后所需要的规格化动作是由加法流水线的第四分段执行的,见图 11.29.

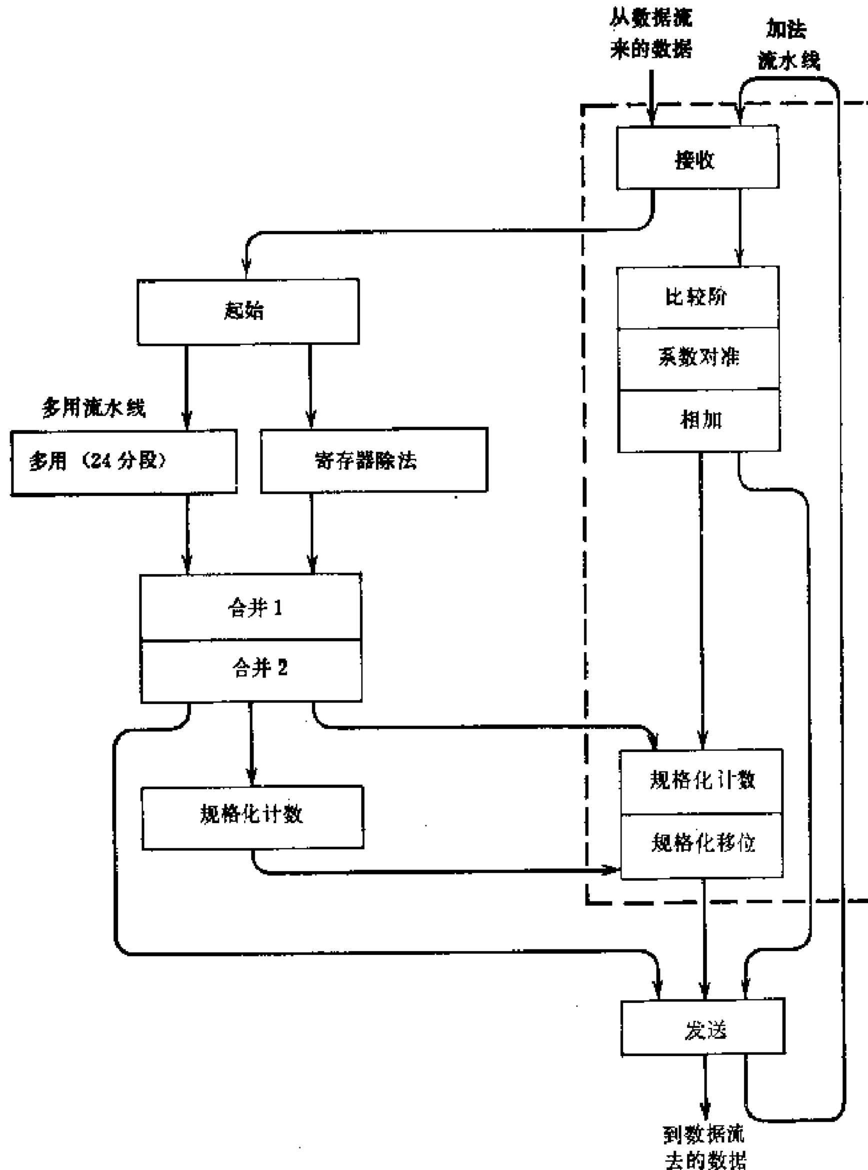


图 11.31 在 STAR-100 计算机中浮点运算流水线处理器 2 的原理图

流水线运算处理器 2 见图 11.31. 处理器包括一个流水线浮点加法部件,这与处理器 1 相似. 还有一个非流水线的浮点寄存器除法部件,一个 24 分段流水线多用途部件,以及若干流水线合并部件. 处理器可以用于寄存器除法,寄存器开平方,以及执行所有向量指令. 寄存器除法部分是一个单一分段的除法器,它也能执行二进制到 BCD 以及 BCD 到二进制的转换. 多用途流水线执行开平方向量除法,以及向量乘法指令的一半. 在多用

途流水线中有 24 个分段.

图 11.32 表示如何用两个基本的乘法流水线（图 11.30）形成一个功能更强的乘法流水线处理器. 处理器可以同时执行两个 32 位乘法或者一个 64 位乘法. 当处理器执行 32 位操作时，左边的乘法流水线把输入 A 和 B 相乘. 从流水线束的"部分和"以及部分进位在加法器 E 中合并. 右边流水线形成的两个部分数也同样地被加法器 F 合并.

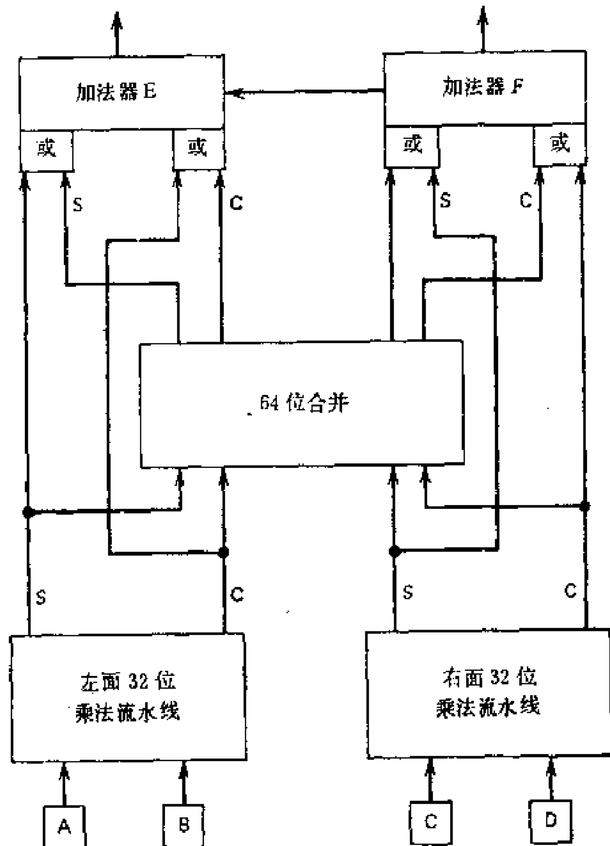


图 11.32 两个 32 位的乘法流水线并行形成一个 64 位的乘法流水线

为了执行 64 位乘法，我们把每个乘数和被乘数都分解成两部分，乘数为

$$A = A_0 + A_1 \times 2^W \quad (11.79)$$

被乘数为

$$B = B_0 + B_1 \times 2^W \quad (11.80)$$

其中 W 是基本流水线乘法器的宽度（32 位）. 于是，我们得到

$$A \times B = A_0 \times B_0 + (A_0 \times B_1 + A_1 \times B_0) \times 2^W + A_1 \times B_1 \times 2^{2W} \quad (11.81)$$

于是 64 位的乘法通过执行两对 32 位乘法来实现：即在第一个周期执行 $A_0 \times B_0$ 和 $A_0 \times B_1$，在第二个周期执行 $A_1 \times B_0$ 和 $A_1 \times B_1$. 于是，这些分解的乘法的四个"部分和"以及部分进位被 64 位合并流水线所合并. 然后，合并流水线的 64 位输出在最后的加法器中相加在一起，产生 64 位的乘积.

以上描述的各种运算流水线的最高运算速度归纳在表 11.6 中. 各项是以 40ns 的流水线周期时间为基础的. 总之，STAR-100 是具有多方面用途的设计，它充分利用了流水线运算的执行以及很高的存储器频宽这两个优点. 这些特性使得向量能以 40ns 的速率通过流水线.

表 11.6 在不同字长的情况下，STAR-100 每秒所能执行的运算操作的最高次数

浮点操作	32 位结果	64 位结果
加法/减法	100×10^6/秒	50×10^6/秒
乘法	100×10^6/秒	25×10^6/秒
除法/平方根	50×10^6/秒	12.5×10^6/秒

11.12 参考文献注释

关于用单元阵列求平方根和平方的材料是基于 Majithia 的著作[22]. 将带符号数字的技术应用到求平方根最佳化上已经在 Metze[24]中作了报道. Ramamoorthy 等[29] 研究了若干用于求平方根的叠代乘法,其中不需要除法. Chen 曾提出计算二进位数的平方根倒数和平方根的收敛方法[5]. 在 11.3 节中提出的有界基本函数的多项式近似方法,对于 Flores[14] 提出的多项式计算器设计有了改进, Flores 的方法要求产生变量 x 的每个幂项.

CORDIC 三角计算技术最初是由 Valder[34] 提出的. 以后,很多作者修改了原来的算法. 统一 CORDIC 方法是基于 Walther[35] 的工作. DeLugish 描述了乘法,除法, $\ln x$, e^x, 平方根,以及三角函数的一种算法[12].他的方法是基于在快速除法算法中采用冗余再编码技术. Chen 氏计算方法[5] 相似于 IBM360 系统 91 型中除法的收敛变换技术. Chen 提出的计算指数、对数、比例和平方根的叠代方法,共享着标准加法,乘法和除法操作所需硬件的大部分. Senzig[32] 曾经简化了 CORDIC 算法,使之适应计算器的结构,其字长在 10 个十进位数字的数量级. 他的计算器算法在计算三角,双曲线,指数和对数等函数时大约需要100到150次加法-移位操作,这种速度对于大多数计算器应用是可以接受的. Ercegovac[13] 和 Sarkar 等[31]曾经研究了获得基本函数的高基数和经济的计算过程.

Chen[6] 在 Stone 主编的计算机结构的新书中写了一章关于重叠和流水线处理技术. 原始的成果也曾发表在他的早期的著作中[4,5]. 在 TI 公司的 **ASC** 系统中实现的流水线运算,曾经被 Watson[36], Watson 和 Carr[37], Ramamoorthy 和 Li[27], 以及 Stephenson[33] 所报和评价. 其中,Stephenson 的处理方法体现了大多数的逻辑设计考虑. 通用的流水线阵列则是由于 Kamal 等人[20]的工作. 流水线 FFT 处理器设计是基于 Larson 的论文[21]. 具有不同分段设计的流水线 FFT 处理器的结构最初是由 Groginsky[16] 提出的. CDC STAR-**100** 流水线运算处理器的实例研究是依据[3,7,17,18]中的论述.图 11.28、11.30 和 11.32 中所示的原理图选自 Hintz 和 Tate [18]. 最近宣布了两个高吞吐率的流水线计算机. 一个是 **CRAY-1**,在[19]中由 Johnson 描述,以及在[3]中由 Baskette 和 Keller 作了评价;另外一个是[1]中描述的 AMDAHL **470 V/6**. 最近, Ramamoorthy 和 Li 提供了一篇关于流水线计算机结构的极好的综述. 其他有关流水线运算设计的重要文章可参考 [9, 10, 11, 15, 17, 25, 26, 27].

运算误差控制技术曾为很多作者所研究. Avizienis[2] 描述 AN 误差-检测码如何可以用在容错的运算设计中,如在喷气推进实验室的 **STAR**(自检测和自维修)系统中就有这些特点.运算处理器误差码的广泛论述可查找 Rao[30] 的文章. 对于那些愿意进一步研究运算误差纠正码的人 Chien 和 Hong[8] 以及 Massey[23] 的著作是值得一读的.

参 考 文 献

[1] Amdahl Corp., *Amdahl 470 V/6 Machine Reference*, Sunnyvale, California, 1975.

[2] Avizienis, A., "Fault Detection in Digital Arithmetic Processors," in *Theory of Digital Computer Arithmetic*, Class Notes (Engineering 225A), University College of Los Angeles, Los Angeles, California, 1971.

[3] Baskett, F. and Keller, T. W., "An Evaluation of the CRAY-1 Computer," in *High-Speed Computer and Algorithm Organization* (Kuck, et al., eds.), Academic Press, New York, 1977.

pp. 71—84.
[4] Chen, T. C., "Parallelism, Pipelining, and Computer Efficiency," *Computer Design*, January 1971. pp. 69—74.
[5] Chen, T. C. "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots," *IBM Journal Res. and Dev.*, July 1972. pp. 380—388.
[6] Chen, T. C., "Overlap and Pipeline Processing," Chapter 9 in *Introduction to Computer Architecture*, SRA, Inc., Palo Alto, California, 1975, pp. 375—429.
[7] Control Data Corp., "Control Data STAR-100 General System Descriptions," St. Paul, Minnesota, Pub. No. 60256000-03, October 1973.
[8] Chien, R. T. and Hong, S. J., "Error Correction in High-Speed Arithmetic," *IEEE Trans. Comp.*, Vol. C-21, No. 5, May 1972.
[9] Cotton, L. W., "Circuit Implementation of High-Speed Pipeline Systems," *IEEE Trans. Comp.*, Vol. C-20, January 1971, pp. 33—38.
[10] Cotton, L. W., "Maximum-Rate Pipeline Systems," *Proc. SICC*, 1969, pp. 581—586.
[11] Davidson, E. A. et al., "Effective Control for Pipeline Computers," *IEEE 1975 Compcon Reader Digest*, Spring 1975.
[12] De Lugish, B. G., "A Class of Algorithms for Automatic Evaluation of Certain Elementary Functions in a Binary Computer," *Technical Report* No. 399, Dept. of Computer Science, University of Illinois, Urbana, Illinois, 1970.
[13] Ercegovac, M. D., "Radix-16 Evaluation of Some Elementary Functions," *IEEE Trans. Comp.*, Vol. C-20, No. 12, December 1971, pp. 1617—1618.
[14] Flores, I., *The Logic of Computer Arithmetic*, Prentice-Hall, Englewood Cliffs, New Jersey, 1963, Chapter 17.
[15] Graham, W. R., "The Parallel and the Pipeline Computers," *Datamation*, April 1970, pp. 68—71.
[16] Groginsky, H. L., "A Pipeline Fast Fourier Transform," *IEEE Trans. Comp.*, Vol. C-19, November 1970, pp. 1015—1019.
[17] Hallin, T. G. and Flynn, M. J., "Pipelining of Arithmetic Functions," *IEEE Trans. Comp.*, Vol. C-21, August 1972, pp. 880—886.
[18] Hintz, R. G. and Tate, D. P., "CDCSTAR-100 Processor Design," *Compcon Proc.*, September 1972, pp. 1—4.
[19] Johnson, P. M., "CRAY-1 Computer System," Pub. No. 2240002A Cray Research, Inc., Minnesota, 1977.
[20] Kamal, A. K. et al., "A Generalized Pipeline Array," *IEEE Trans. Comp.*, Vol. C-23, May 1974, pp 533—536.
[21] Larson, A. G., "Cost-Effective Processor Design with an Application to Fast Fourier Transform Computers," *Ph. D. Thesis*, Stanford University, 1973.
[22] Majithia, J. C., "Cellular Array for Extraction of Squares and Square Roots of Binary Numbers," *IEEE Trans. Comp.*, Vol. C-20, No. 12, December 1971, pp. 1617—1618.
[23] Massey, J. L. et al., "Error Correcting Codes in Computer Arithmetic," in *Advances in Information System Sciences*, Chap. 5, Plenum Press, New York, 1972.
[24] Metze, G., "Minimal Square Rooting," *IEEE Trans. Elec. Comp.*, Vol. EC-14, No. 2, 1965, pp. 181—185.
[25] Patel, J., "Improving the Throughput of Pipelines with Delays and Buffers," *Ph. D. Thesis*, Stanford University, September 1976.
[26] Peatman, J. B., *The Design of Digital Systems*, McGraw-Hill, New York, 1972.
[27] Ramamoorthy, C. V. and Kim, K. H., "Pipelining—The Generalized Concept and Sequency Strategies," *National Computer Conf.*, 1974. pp. 289—297.
[28] Ramamoorthy, C. V. and Li, H. F., "Pipeline Architecture," *ACM Computer Surveys*, Vol. 9, No. 1, March 1977, pp. 61—102.
[29] Ramamoorthy, C. V. et al., "Some Properties of Iterative Square-Rooting Methods Using High-Speed Multiplication," *IEEE Trans. Comp.*, Vol. C-21, No. 8, August 1972, pp. 837—847.
[30] Rao, T. R. N., *Error Codes for Arithmetic Processors*, Academic Press, New York, 1974.
[31] Sarakar, B. P. et al., "Economic Pseudodivision Processes for Obtaining Square-Root, Logarithm and Arc Tan," *IEEE Trans. Comp.*, Vol. C-20, No. 12, December 1971, pp. 1589—1593.
[32] Senzig, D., "Calculator Algorithms," *IEEE Compcon Reader Digest*, Spring 1975, IEEE Catalog No. 75 CH 0920-9C, pp. 139—141.