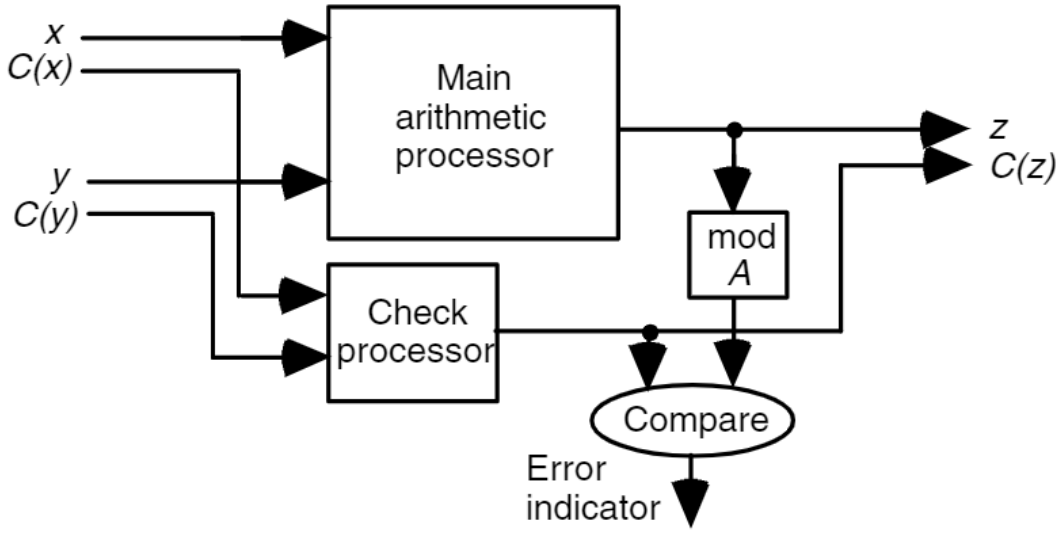


Figure 27.4 Arithmetic processor with residue checking.

【硬件算法笔记27】容错计算技术

Vortex 关注他

基于这本书：

resens：计算机硬件算法目录汉化
COMPUTER ARITHMETIC : Algorithms...
220 赞同 · 26 评论 文章

现代数字电路组件（components）非常稳健（robust），但超大规模的电路会将任何细小的错误与故障放大。例如数据通讯中，诺每 bit 发生错误的概率仅为 10^{-10} ，则其会被认为是相当好的。但在每秒数百万次运算操作的速度下，这个错误概率意味着每秒都会出现数个 bit 发生错误。

在计算机通信中，可以利用编码技术防止数据传输或存储中的错误。但用于通信领域的方法并不一定适用于算术容错，因此本章将讨论一些可用于提高算术系统（arithmetic system）稳健性（robustness）与可靠性（reliability）的办法。

故障、错差与错差编码（FAULTS,ERRORS,AND ERROR CODES）

在此之前，我们在设计中都一直采用着这个假定：

- 算术或逻辑元素（arithmetic and logic elements）总是按照预期的方式行为运行。

例如，与门的输出总是其所有输入的逻辑与，LUT表项总是能保存正确的初始值，以及导线一直都是保持连接的。

即便是非常可靠的现代集成电路，在漫长的计算过程、极端恶劣的运行环境、极端/不可预测的工会发生。

比如，与门的输出可能会一直“卡在1上”，于是只要此时有一个输入为 0，这个与门的输出就不符合预期了。此外，串扰（cross talk）或外部干扰（external interference）也可能与门的“瞬时故障（transient fault）”，在出现这种情况的几个时钟周期内，与门可能偏离预期行为。

内存单元的制造缺陷或读写电路的逻辑故障，可以导致可读条目（entry）的损坏。过热、制造缺陷等因素也可能造成导线的断裂或与另一根导线短路。

确保数字系统在发生故障（永久（permanent）或瞬时（transient）性的）的情况下正确运行，是容错计算（fault-tolerant computing）领域的主题，也叫作可靠计算（reliable(dependable) computing）[Parh94]。

在本章，我们将了解一些容错计算中的思想，这些思想与算术函数的计算关系特别大。

检测或纠正数据错差

检测或纠正数据错误的技术起源于通信领域，早期的通信信道非常不可靠，也不稳定，信号在传输过程中，很容易发生失真或改变。通信工程师想出的对策就是，以冗余格式编码(encode)数据，即“编码(code)”或“错差码(error codes)”。具体的编码方法例子包括：增加 1 个校验位（SED，single-error-detecting）、校验和（checksum）、汉明码（Hamming code）、SEC(或DED)码（double-error-detecting）等等。

如今，错差（error）检测与纠错码依旧被广泛应用于通信领域，尽管这些系统的可靠性与降噪/屏蔽技术有了很大进步，但在另一方面，数据速率和数据传输量的快步提升，也使得错误概率仍然不可忽视。

曾为通信而开发的编码也可用于防止存储错差。自早期集成电路存储器被证明比当时场景的磁芯技术更不可靠时，集成电路设计者就将SEC/DED编码加入它们的设计中了。

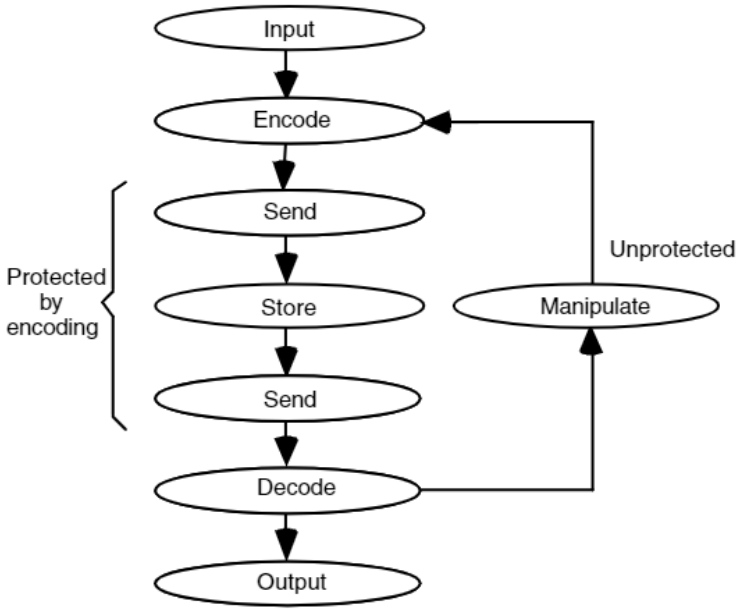


Figure 27.1 A common way of applying information coding techniques.

上图Fig 27.1展示了一个通用的信息编码技术流程：

1. 对输入数据进行编码（错差、错差码等）
2. 在收发、存储数据时，使用编码后的数据，以获得容错、纠错能力
3. 在输出处，对编码后的数据进行解码，以还原信息，然后对其进行操作等

这个方案常被应用于现代数字系统中，注意其中数据操作（data manipulate）部分是不受保护的。就是说，只在数据收发（向远处的）、存储的部分提供了保护。

编解码是必要的，因为许多常见编码在算术运算下都是不封闭的，例如两个偶数的和可能不一定偶须计算一个新的校验

和以替代旧元素。

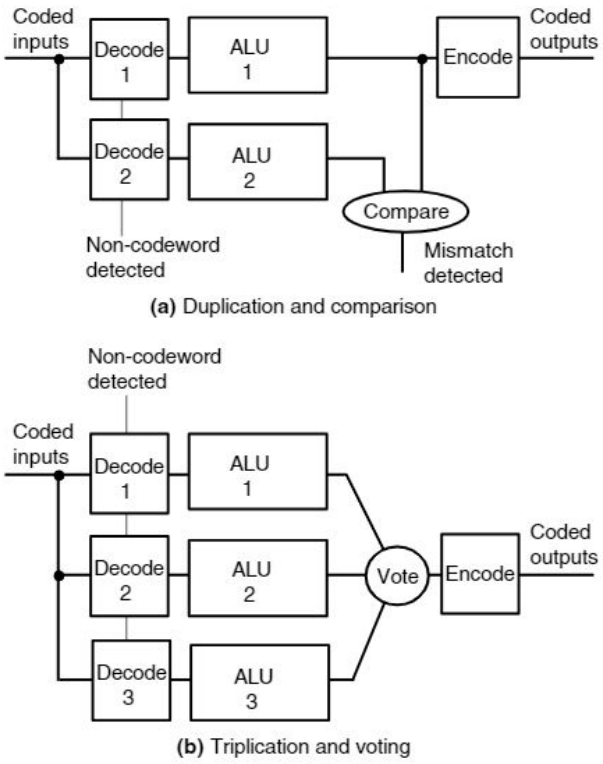


Figure 27.2 Arithmetic fault detection or fault tolerance (masking) with replicated units.

保护算术计算不受故障引起影响的一种办法是，重复进行多次相同操作流程，并对这些重复过程的结果进行比较。

Fig 27.2就展示了这种思路的实现，其中(a)比较两个重复ALU的结果（可用于单一故障（single fault）/错差检测）；(b)比较了3个，并通过多数表决的方式确定正确结果（可用于单一故障的掩盖或纠正）。

具体地，(a)中有两个重复的ALU，因此能保证单一故障被检测。但另一方面，其中的编码器（encode）依旧是一个关键元件，它的故障将无法被检测。

编码器（encoder）输出是冗余的（或coded），因此我们完全可以设计出一种编码电路，任何错误发生时，在其输出端产生一个非编码字（non-code word）。这种设计被称作自效验（self-checking），这使得我们可以传递“数据错误/失真”的信息，只要在后面的电路前加入（如存储器或ALU输入）一个校验器（checker），检测编码是否为非编码字，就能判断这个数据的有效性。

假定我们使用了自校验编码器（self-checking encoder），则Fig 27.2a中的双重复制结构可以检测任何由图内任何一个块的故障引起的错误。这也包括“比较块（图中compare）”，其故障也可能产生错误警告。只有两个或以上的故障（且在不同块）发生时，才可能检测不到错误。

Fig 27.2b中的三重复制ALU设计与此类似。注意其中的投票器（vote）是一个关键元素，必须谨慎设计。自校验设计不能用于投票器，因为它的输出是非冗余的。不过，通过结合投票与编码功能，我们也可能设计出一个高效的自校验投票编码器。

图中的三通道计算策略（三个重复复制的ALU）也可以推广为 N 通道，以提供容错性，但同时也将伴随成本开销，并且通常是无法接受的。

算术故障与通信故障的区别

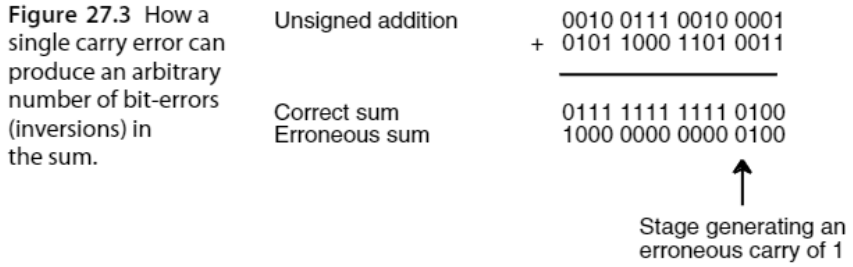


Figure 27.3 How a single carry error can produce an arbitrary number of bit-errors (inversions) in the sum.

前面的多重复制方案将涉及大量硬件开销，因此我们应该尝试在ALU内应用编码技术以进行故障检测或容错。

但第一个问题是，通信中的许多故障（如单次(single)、多次(double)、簇发(burst)有关的故障）的处理方案并不能照搬至算术设计中。

在通信中，噪声引起的尖峰（spike）可能使得单个 bit（随机错差random error），或少量连续 bit （簇发错差burst error）的错差。

但在加法器中，加法器一个错差的进位信号（例如由进位链某个逻辑门的故障导致）可能导致加法器输出中出现任意数量的bit被翻转。图Fig 27.3就提供了一个例子，其中加法器的一个单点故障就造成了结果中的12bit被翻转，或者用编码的术语说：正确结果与错误结果间的汉明距离（hamming distance）为12，或错差的汉明权（hamming weight）为12。

编码的检错或纠错能力与码字（codeword）间的最小汉明距离有关，这里提供3个例子：

Single-error-detecting (SED)	Min. Hamming distance = 2
Single-error-correcting (SEC)	Min. Hamming distance = 3
SEC/DED	Min. Hamming distance = 4

例如在SED编码中，仅翻转其码字中的任意 1 位，并不会导致其变成另一个码字（而是变成无效字），从而提供了检错。而对于SEC，仅翻转了其中 1 位的无效码字，相较于其它有效码字更接近原始的码字（从汉明距离上看），从而提供了纠错。

在Fig 27.3中的加法例子中可以看到，仅靠SED是无法检测所有情况中的故障的。

以及，应当注意的一点是，在计算机算数中，我们处理的是数字，而不是任意的位串。因此我们感兴趣的是错误值与正确值间的数字大小差别，而不是其中位的差异（汉明距离）。例如在Fig 27.3中，正确值与错误值相差 24。

据此，我们可以将算术权（arithmetic weight）定义为：加到正确值上后，能产生错差结果的（反之，加到错差结果能产生正确值亦然）的，最小的 有符号 2 的次幂数。（原文说的这么绕，可能是想避免减法）

这里提供两个例子：

Correct result	0111 1111 1111 0100	1101 1111 1111 0100
Erroneous result	1000 0000 0000 0100	0110 0000 0000 0100
Difference (error)	$16 = 2^4$	$-32\,752 = -2^{15} + 2^4$
Error, in minimum-weight BSD form	0000 0000 0001 0000	$^-$1000 0000 0001 0000
Arithmetic weight of the error	1	2
Type of error	Single, positive	Double, negative

于是，上述例子中的错误可以被看做是算术意义上的“单（single）”、“双（double）”错差。一些已被提出的算术检错编码能够检测或纠正那些以算术（而非汉明权）为特征的错差，我们

注意，最小权重（minimum-weight）的二进制有符号数（BSD）表示的错差幅度（error magnitude）将至多有 $\lceil (k+1)/2 \rceil$ 个非 0 位（digit），且总是可写成无任何连续非 0 位的规范二进制数（canonic BSD）形式。BSD数的规范形式（canonic form）是唯一的，且与算术错差权重（arithmetic error weight）概念关系密切。

算术检错编码（ARITHMETIC ERROR-DETECTING CODES）

算术检错编码的两个特点包括：

1. 以可检测错差（detectable error）的算术权（arithmetic weight）为特征
2. 允许我们直接对编码后的操作数（coded operands）进行算术操作

上节末讨论了其中第一点的重要性。第二点非常重要，因为它允许我们以比多重复制低得多的硬件冗余（开销）来保护算术计算免受电路故障的影响。

在本章，我们将讨论两类算术检错编码：乘积码（product code）与剩余码（residue code）。

在这两者的讨论种，我们都将假设操作数是无符号整数。因为浮点数的有关编码往往比较复杂，且不太受算术和容错研究员的关注。

乘积码（product codes）

所谓乘积码（也被称作 AN码 或 AN编码），就是将一个数 N 表示为乘积 AN，其中校验模数 A（check modulus）是一个常量。

要确定一个 AN 编码的操作数的有效性（validity），我们需要检查它关于 A 的整除性（divisibility by A），即是否能被 A 整除。

奇数 A 可以检测出所有权重为 1 的算术错差（包括所有单bit错差），但权重为 2 或更高的算术错差可能无法检测。例如错差 $32\,736 = 2^{15} - 2^5$ 在 A=3, 11, 或31 的情况下就无法被检测，因为它可以被这些校验模数整除。

乘积码的编/解码过程将涉及乘除法（乘/除以 A）。且很快也将看到，乘积码上的算数操作也需要涉及乘或除以 A。因此，为使这些编码实用，我们应确保校验模数 A 的乘除法很简单。即，我们可以考虑一类低成本乘积码（low cost product codes），其校验模数形式为 $a = 2^a - 1$ 。

乘以 $A = 2^a - 1$ 特别简单，只需要移位和减法（而且“-1”在减法中也是特别的）。除以 $A = 2^a - 1$ 也同样简单。

于是，给定 $y = (2^a - 1)x$ ，我们可以通过计算 $2^a x - y$ 以获得 x 。

$A = 2^a - 1$ 是奇数，因此低成本乘积码可以检测任何权重为 1 的算术错差。一些权重更高的错差可能无法发现，但这些错误的比例将随 A 的增加而减小。

单向错差（unidirectional errors），即所有错差位要么都是 0到1 的翻转，要么都是 1到0 的翻转，构成了数字电路实现中的一类重要错误。对于单向错误，其错差幅度（error magnitude）为为诺干同号 2 的次幂之和。

THEOREM 27.1

任何算术权不超过 $a - 1$ 的单项错误，都可以通过校验模数为 $A = 2^a - 1$ 的低成本乘积码检测到。

例如，A=15 除可以检测所有权重为 1 的错差外，还可以检测权重为 2 或 3 的错差。

可以被检测：

$$8+4=12$$
$$128+4=132$$
$$16+4+2=22$$
$$256+16+2=274$$

乘积码是不可分（nonseparate/nonseparable）编码的例子，在这些编码中，原始数据与用于检查的冗余信息是被混杂在一起的。换句话说，我们不能从 AN码 上直接获得原始数值 N，而是需要通过一个解码过程（即通过除以校验模数 A）。

乘积码的算术操作

乘积码的算术操作比较简单，可以直接执行加减法，因为：

$$Ax \pm Ay = A(x \pm y)$$

而乘积码直接相乘的结果则为：

$$Aa \times Ax = A^2ax$$

因此必须将乘积码的乘积除以 A，以将其修正为正确值。

对于除法，诺 $z = qd + s$ ，其中 q 是商， s 是余数，则有：

$$Az = q(Ad) + As$$

因此，直接进行除法算法所得到的将是商 q 和余数 As 。即，余数是以编码形式得到的，但商却必须将其乘以 A 才能得到编码后的商。

由于 q 是以非冗余方式获得的，其计算过程中发生的错误将无法被发现。为防止数据在除法过程中出错，我们可以先给被除数 Az 预乘一个 A ，然后再将其照常除以 Ad 。这种方法的问题是，所导出的商 q^* 和余数 s^* 将满足下式：

$$A^2z = q^*(Ad) + s^*$$

而这可能与预期的结果 Aq, A^2s 不同（后者 As 需除以 A 以修正）。 q^* 可能比 Aq 大至多 $A-1$ 个单位，因此可能需要进行修正，才能得到正确的商和余数。但在处理未受保护的 q^* 时（不一定是A的倍数），也可能出现无法发现的错误。

在 $A = 2^a - 1$ 的情况下，应对这个问题的一种可能解决方案是：在处理（除数为） a bits 的除法时，调整余数 q^* 的最后一个基数为 2^a 的位，而得到关于 A 成倍数关系的新余数 q^{**} 。这可通过保持前面的商位具有模 A 的校验和（modulo-A checksum of previous quotient digits）来实现。可以证明的是，只要在 $[-2^a+2, 1]$ 上选择 q^{**} 的最后一个基数为 2^a 的位，就足以解决这个问题。

然后要做的就是通过减法将 q^{**} 转为标准二进制表示的结果，详细过程可以参考[Aviz73]。

平方根导致的有关问题与除法中的问题类似。假设我们将被开方数 Az 乘以 A ，然后执行标准平方根算法计算过程，将计算：

$$\lfloor \sqrt{A^2x} \rfloor = \lfloor A\sqrt{x} \rfloor$$

这个计算结果与一般的正确结果 $A\lfloor \sqrt{x} \rfloor$ 不同，因此需要修正。同样， $\lfloor A\sqrt{x} \rfloor$ 也可以比 $A\lfloor \sqrt{x} \rfloor$ 多至多 A-1 个单位，上面针对除法的修正过程也同样适用于平方根。

在剩余码下，操作数 N 将被表示为一个数对 $(N, C(N))$ ，其中 $C(N) = N \bmod A$ 是校验部分，常数 A 为校验模数。

剩余码是可分（separate/separable）编码的例子，因为其中数据和校验部分并没有混杂在一起，也因此使得解码变得很简单。

将 N 编码为剩余码，需要计算 $C(N) = N \bmod A$ ，然后在编码格式中将其与 N 拼起来，以形成编码表示 $(N, C(N))$ 。

与乘积码中的情况一样，我们也可以定义低成本剩余码（low cost residue codes）为具有校验模数 $A = 2^a - 1$ 的剩余码，此时编码计算 $N \bmod A$ 会很简单。

剩余码的算术操作

剩余码的算术操作部非常简单，尤其是在使用低成本校验模数 $A = 2^a - 1$ 时。

如下，剩余码的加减法是通过对数据、校验部分分别操作来完成的：

$$(x, C(x)) \pm (y, C(y)) = (x \pm y, (C(x) \pm C(y)) \bmod A)$$

因此，如图Fig 27.4所示，将剩余码作为操作数的将有一个主要的加法器用于数据的加减法，此外还有一个很小的模A加法器（图中check processor），用于处理校验部分的模加法。

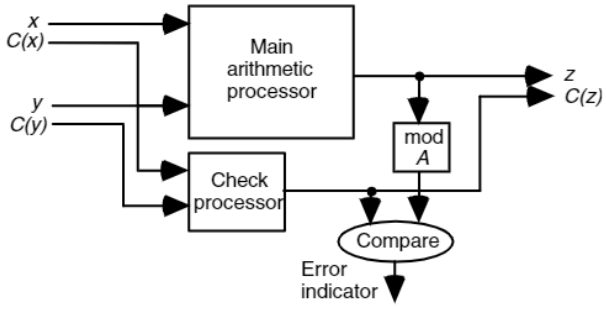


Figure 27.4 Arithmetic processor with residue checking.

为检测算术单元内的故障，我们需要将这个小型模加法器（check processor）输出于主加法器的模A余数进行比较。

剩余码的乘法也很简单：

$$(a, C(a)) \times (x, C(x)) = (a \times x, (C(a) \times C(x)) \bmod A)$$

于是，Fig 27.4的结构也适用于剩余码乘法。这种检验乘法运算的方式，本质上就是我从来都没听说过的去九法（一种演算加减乘运算的的方法）。

就像剩余数表示系统（RNS）算术一样，剩余码的除法和平方根很复杂。对于这些运算，其小型的余数校验器（residue check processor）并不能独立于主算术处理器（Main arithmetic processor），就是说必须与主算术处理器交互计算结果的校验，模数。

也如同乘积码，为奇数的校验模数 A 可以确保剩余码检测到所有权重为 1 的算术错差。但剩余码在检测多个单向错差方面的能力不如乘积码。例如我们先前看到的， $15N$ 码（ $A = 15$ 的乘积码）可以检测所有权重为 2、3 的单向算术错差，但 $A = 15$ 的剩余码并不能以类似的方式检测权重为 2 的错差，因为它在数据和余数上加了 1 ，而使得结果成为了一个“有效”的码字。

为应对上述问题，有人就提出了反剩余码（inverse residue code），其中校验部分 $C(N)$ 被表示为 $A - (N \bmod A)$ 而不再是 $N \bmod A$ 。对于 $A = 2^a - 1$ ，它就是 $N \bmod A$

单向错差（unidirectional error）现在以相反的方向影响数据与校验部分，因此更容易被发现。通过注意到，对于 a bits 的反剩余码，将 $C'(N) = A - (N \bmod A)$ 附着在 k bits 的 N 的最小有效端上，就能得到一个关于 $A = 2^a - 1$ 成倍数关系的 $(k + a)$ bits 的结果，容易证明以下事实：

THEOREM 27.2

任何算术权重不超过 $a - 1$ 的单向错差，都可通过低成本反剩余码检测，该反剩余码将使用校验模数 $A = 2^a - 1$ 。

一个检测码（error-detecting code）的额外成本和开销主要来自 2 个部分：

1. 编码后操作数的字宽（word width）的增加，这将导致寄存器、存储器、数据链路成本的增加
2. 校验算术或更宽的操作数使得 ALU 更加复杂

有关其中的第 1 部分，乘积、剩余、反剩余码都是类似的。例如它们的低成本版本的校验模数 $A = 2^a - 1$ 都需要给操作数外加一个附加位。

在算术方面，剩余码和反剩余码比乘积码简单，不过在除法反而比乘积码更复杂。

有趣的一点是，剩余类编码（residue-class codes）是用于校验加法器的唯一可能的可分编码（separable code）[Pete58]。

此外，已经证明的是，与、或、异或（AND、OR、XOR）等位级逻辑运算并不存在低于 100% 冗余程度的检错编码方案；就是说，对于逻辑运算检错，最好的办法就是重复复制并比较[Pete59]，如图Fig 27.2a。

算术纠错码（ARITHMETIC ERROR-CORRECTING CODES）

在这里，我们通过一类双剩余码（class of biresidue code）来阐述关于算术纠错码的主要思想。（“双剩余（biresidue）”的意思是用到了两个余数）

双剩余码（biresidue code）将数 N 编码为一个三元组 $(N, C(N), D(N))$ ，其中 $C(N) = N \bmod A$ 是校验部分， $D(N) = N \bmod B$ 是校验模数 A 除以校验模数 B 的余数。

诺原始数据需要 k $bits$ 以表示，则其对应的二进制剩余码表示将需要 $k + \lceil \log_2 A \rceil + \lceil \log_2 B \rceil$ $bits$ 以表示。

双剩余码类（class of biresidue code）的编码过程（encoding）与单剩余码（single-residue codes）类似，只是现在需要计算两个余数了。

双剩余码操作数的加法和乘法可以通过类似Fig 27.4所示的算术单元来处理，但将具有两个校验处理器（check processor）。由于这两个余数可以并行地进行计算与校验，因此不会损失速度。

诺错差仅影响数值 N 或其中一个余数（即 $C(N)$ ），则其可按照下方式修正：

1. 错差发生在 $C(N)$ 时， $C(N)$ 将无法通过余数校验（residue check），但 $D(N)$ 通过了它的校验，因此可通过重新计算 $N \bmod A$ 以纠正。

1. 错差发生在 N 时，除非错差幅度恰好为 A 或 B 的倍数（此时无法完成检错，或与余数错差无法区分），否则两个余数的校验都会失败，从而说明错差发生在 N。

$N)(D(N))$ 间的

3. 这两个差，$[(N_{wrong} \bmod A) - C(N)] \bmod A$ 和 $[(N_{wrong} \bmod B) - D(N)] \bmod B$ 构成了一个错差伴随式（error syndrome），诺不同错差的伴随式（syndrome）是不同的，则我们可以将其纠正。

作为例子，现考虑带有低成本校验模数 $A = 7, B = 15$ 的双剩余码。表Table 27.1显示，任何权重为 1 的算术错差 E （满足 $|E| \leq 2048$ ）都将导致一个独特的错差伴随式（error syndrome），从而可以通过给 N_{wrong} 减去相关错差值以修正它。

> syndrome，在本领域内被翻译为“伴随式”或“校验因子”，“伴随式”这个翻译很烂，所以这里做出一定注解：“syndrome”在医学上的意思是“综合征”，可以理解为各种“error”综合起来，所表现出的“综合征”，这是“error syndrome”所想传达的意思。但总的来说这几个翻译都做不到信达雅。

对于 $|E| \geq 4096$ ，其值将与 $E/4096$ 的相同，因此我们只能保证 12 bits 部分的数据的权重为 1 的错差能被纠正。由于这两个余数的表示总共需要 7 bits，因此这个双剩余码的冗余度为 $7/12 \approx 58\%$ 。

Table 27.1 Error syndromes for weight-1 arithmetic errors in the (7, 15) biresidue code.

Positive error	Error syndrome		Negative error	Error syndrome	
	Mod 7	Mod 15		Mod 7	Mod 15
1	1	1	−1	6	14
2	2	2	−2	5	13
4	4	4	−4	3	11
8	1	8	−8	6	7
16	2	1	−16	5	14
32	4	2	−32	3	13
64	1	4	−64	6	11
128	2	8	−128	5	7
256	4	1	−256	3	14
512	1	2	−512	6	13
1024	2	4	−1024	5	11
2048	4	8	−2048	3	7
4096	1	1	−4096	6	14
8192	2	2	−8192	5	13
16384	4	4	−16384	3	11
32768	1	8	−32768	6	7

校验模数为 $A \times B = 7 \times 15 = 105$ 的乘积码，也同样可以纠正权重为 1 的错差，通过码字关于 7 和 15 的整除性，并注意余数。然而，这么做的效率会低得多，因为必须将总字宽限定在 12bits，才能完全覆盖全部错差。因此，最大的可表示数为 $4095/105 = 39$ ，大约相当于 5.3 bits的数据，导致冗余度高达 127%。

一般来说，具有互素低成本模（relatively prime low-cost check moduli）$A = 2^a - 1, B = 2^b - 1$ 的双剩余码，可以为 $ab\ bits$ 的数据部分提供 权重为 1 的错误的纠正，其表示冗余度为 $(a + b)/(ab) = 1/a + 1/b$ 。因此，我们只需要选择较大的 a, b ，就可以获得较低的冗余度。

基于上述有关算术检错、纠错码的讨论，我们可得出结论：这种编码不仅对保护算术计算过程的故障导致的错差有效，也对存储和传输中的错差有用。在整个系统中使用单一的编码，可以避免频繁的编、解码，并最大限度地减少处理未编码数据时发生错差的可能性。

自校验功能单元（SELF-CHECKING FUNCTION UNITS）

自校验功能单元（self-checking function）可被设计为输入、输出不带编码的形式。例如Fig 27.4中，$x \bmod A$ 和 $y \bmod A$ 都是在功能单元内部计算的，而不是输入本身提供的。总之

有关自校验逻辑设计（self-checking logic design）的理论已经相当成熟，而可用于实现高可靠，或至少是安全失败（fail-safe）的算术单元。其想法就是，设计所需逻辑电路，并确保任何故障都来自一个我们的规定故障集中，其中包含了我们希望保护的故障，对于这些故障，要么不影响输出正确性（被屏蔽，masked），要么导致一个非字码输出（可观察，observable）。

在后一种情况下，无效结果要么被附加在该单元输出处的校验器立即检查到，要么在下一个下游的校验器上检查到，或者一直保持“非代码”编码并传播下去，类似于浮点运算中的 NaN 及其在运算操作中的传递。

在这种自校验单元的设计中，一个重要的问题是能否建立一个这样的自校验代码检查器，保证其不会验证一个非代码，尽管内部出现故障。例如，一个反剩余码 $(N, C'(N))$ 的自校验器可设计如下：

首先计算 $N \bmod A$，诺输入是有效的码字，则该计算的结果应当是 $C'(N)$ 的按位补码（bitwise complement）。我们可以把验证 $x_{b-1} \ldots x_1 x_0$ 是 $y_{b-1} \ldots y_1 y_0$ 的按位补码的过程看作是确保信号对 (x_i, y_i) 全为 $(1,0)$ 或 $(0,1)$ 的过程。即相当于计算一组逻辑与的布尔值，这些布尔值的编码规则如下：

$$\begin{array}{ll} 1 & \text{encoded as} \quad (1,0) \text{ or } (0,1) \\ 0 & \text{encoded as} \quad (0,0) \text{ or } (1,1) \end{array}$$

注意，诺输入正确，则编码校验器产生的两个输出将携有 $(1,0)$ 或 $(0,1)$，否则不正确的话则将携有 $(0,0)$ 或 $(1,1)$。

对应所需的，能保证不会因任何门或线路故障，使得非字码输入可以导致 $(1,0)$ 或 $(0,1)$，的所需与逻辑电路很简单。例如我们可以基于Fig 27.5中的二输入与门构建一个与逻辑树。要注意的是任何只有一根输出线的编码校验器都不可能是自校验的，因为它的输出线上的一个信号卡住故障（signal stuck-at fault）就能导致误导性的结果。

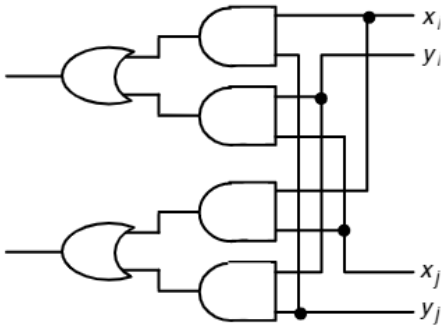


Figure 27.5 Two-input AND circuit, with 2-bit inputs (x_i, y_i) and (x_j, y_j), for use in a self-checking code checker.

基于结果检查的故障检测（验收测试）

故障检测也可通过结果检查（result checking）来实现。这类似于软件容错领域（software fault tolerance）里的验收测试（acceptance testing）。验收测试是一个（希望是简单的）验证过程（verification process）。例如可以计算平方根的平方，并将其与原被开方数比较，以验证平方根计算过程是否正确，诺我们假设平方过程中的任何错差都是独立的，则其不太可能补偿开方过程中的错误，则验收测试结果是可信的概率会非常高。

验收测试不必是完美的。覆盖率不完美（imperfect coverage）的测试（如比较余数）可能不会在每个故障发生后立即检测到它，但随时间推移，验收检验到故障的概率会非常高。另一方面，诺假设故障时永久性的，且很少发生，则定期验证（而不是同时或及时验证）也可能足以完成故障检测的任务要求。这种周期性的检查可能包括用几个随机的操作数进行计算，并验算结果的正确性，从而减少补偿性错误（compensating error）使故障无法被检测的可能性[Blum96]。

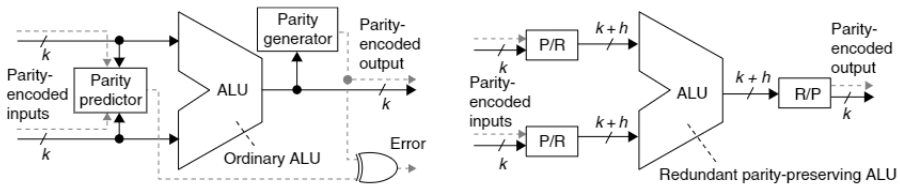


Figure 27.6 Self-checking adders with parity-encoded inputs and output.

鉴于奇偶校验码非常简单，且具有低冗冗余度，我们可能基于奇偶校验编码的输入和输出得到一种特别具有吸引力的自校验设计策略。由此产的方案的故障检测覆盖率通常是相当低的，但如刚才说的，我们可以在一个相对长期的时间窗口内一直进行校验，而只要其中一次检测出了故障，任务就完成了。因此这种低冗余度方案也可以被认为是具有成本效益（cost-effective）的。

基于奇偶校验的方案的主要挑战是，奇偶校验码在算术运算下不是封闭的。因此我们必须设计出用于预测输出奇偶校验结果（Parity predicting）的策略，以确保预测电路中的故障不会导致无法检测的错差。Fig 27.6a将该方法应用到了一个双操作数加法器上[Nico93]，要使该方案有效，必须使奇偶校验预测器（Parity predictor）完全独立于 ALU。

另一种设计策略如Fig 27.6b所示。其思想是在计算的每个阶段都保持数据的偶校验（even parity），并最终得到偶校验的输出。如图Fig 27.6b所示，奇偶编码的输入首先通过“P/R”被转为偶校验的的BSD数（二进制有符号数）。然后通过一个保持奇偶校验的BSD加法器（parity-preserving BSD adder）来计算该两数的偶校验冗余和（even-parity redundant sum）。最后通过“R/P”将结果转为标准输出。上述方案的关键在于，我们有能力将两个相邻的BSD位（digit）编码为一个 4bit 的偶校验码，并将BSD加法器设计成保持奇偶校验的[Thor97]。所采用的的编码可以基于BSD的有符号数编码（signed-magnitude encoding of BSD），于是 -0 和 +0 都将表示 0。正是这种关于 0 的偶、奇校验（even、odd parity）编码，使得该方案成为了可能。实现细节，包括在处理 2的补码 操作数时的小调整，可以参考[Parh02]。

基于算法的容错（ALGORITHM-BASED FAULT TOLERANCE）

到目前为止，我们讨论的用于检错、纠错的方案都是针对单个基本算术操作（如加法和乘法）的。除此之外的另一种策略是，接受“运算可能产生不正确结果”这一事实，并在数据结构或应用层上建立检错、纠错的机制。

作为这种方法的一个例子，现考虑矩阵乘法 $\boldsymbol{XY = P}$ 。一个数列（一个向量）的校验和（checksum）是所有数关于某个校验模数 A 的模，的代数和。

对任意 $\boldsymbol{m \times n}$ 矩阵 $\boldsymbol{M}$ ，我们定义一个 $\boldsymbol{m \times (n+1)}$ 的行校验矩阵 $\boldsymbol{M_r}$ （row-checksum matrix），其第 0～n-1 列都和矩阵 $\boldsymbol{M}$ 一样，多出来的第 n 列是各行的校验和。

类似地， $\boldsymbol{(m+1) \times n}$ 的列校验矩阵（column checksum matrix） $\boldsymbol{M_c}$ 的第 0～m-1 行与 $\boldsymbol{M}$ 一样，而第 m 行则是各列的校验和。

然后， $\boldsymbol{(m+1) \times (n+1)}$ 的全校验矩阵（full-checksum matrix） $\boldsymbol{M_f}$ 的定义则是： $\boldsymbol{M}$ 的行校验矩阵 $\boldsymbol{M_r}$ 的列校验矩阵 $\boldsymbol{(M_r)_c}$ 。下图Fig 27.7就提供了一个例子，其中校验模数为 $\boldsymbol{A = 8}$ ：

Figure 27.7 A 3×3 matrix M with its modulo-8 row, column, and full checksum matrices M_r, M_c, and M_f.

$$M = \begin{bmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{bmatrix} \quad M_r = \begin{bmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{bmatrix}$$

$$M_c = \begin{bmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{bmatrix} \quad M_f = \begin{bmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \\ 2 & 6 & 1 & 1 \end{bmatrix}$$

THEOREM 27.3

给定矩阵 X, Y ，以及 P 满足 $P = XY$ ，则有 $P_f = X_c Y_r$ 。

基于THEOREM 27.3，我们可以对编码矩阵 X_c, Y_r 进行标准矩阵乘法，然后将乘积 P_f 最后一行、列中的值与基于其它元素计算的校验和进行比较，以检测任何可能发生的错差。

诺矩阵元素是浮点数，则等价性（equalities）将近似地保持，意味着我们需要选择一个合适的阈值来定义数值相等，这是困难的。[Dutt96]关于此提供了一些解决方法。

全校验矩阵 M_f 是一个很好的稳健数据结构（robust data structure）的例子，它的错差检验、修正的特性如下：

THEOREM 27.4

在全校验矩阵中，任何单个元素的错差都能被纠正，任何三个元素的错差都可以被检出。

因此，全校验矩阵可用于具有高度局域性的故障的检测与纠正，如硬件乘法中影响不超过三个元素的的瞬时故障。虽然不能保证能检测更广泛的错误，但这也是很有可能的，因为很难出现几个具有补偿性的错误，以至于可以逃避任何校验与检测（但要注意很难出现并不意味着不会发生）。

有关这种具备检错、纠错能力的数据结构的设计，以及提供关于这些数据结构可直接操作的算术算法，仍是门艺术。不过这个领域也正在稳步进展，有关基于算法的容错基数的综述，可以参考[Vija97]。

容错RNS算术（FAULT-TOLERANT RNS ARITHMETIC）

可以将冗余编码与任何数表示系统结合，以提供检错或纠错能力。尤其是剩余数表示系统（RNS，residue number system），利用与额外模数向对应的冗余剩余数，可以实现非常优雅且有效的错差检测、纠正方案。

假定我们在RNS中选定的模数集（set of moduli）冗余地多出了一个余数（即从冗余模数集中去掉任意一个模数，剩下的模数构之集仍满足所需动态范围（dynamic range））。则，任何单一余数错差都可以被检测，因为这种错差会使得受影响的余数与其它余数不一致。显然要使这个方案可行，冗余模数必须是最大的那个（即 m ）。因此，检错方案如下：

用所有其他的余数来计算这个数模 m 的余数。这是通过由一个被称为基拓展（base extension）的过程完成的，有许多算法。然后计算出模 m 余数，并与数表示中的模 m 余数相比较，以检测可能的错差。

这种方法的美妙之处在于，算术算法完全不受影响；只需要简单的扩大RNS的动态范围，就能实现错差检测。且在 RNS 处理器中，由于其它原因，经常会提供基拓展操作，而为错差检测服务 —— 例如，需要用它作为构建，以综合出其它RNS操作。这种情况下，不需要额外的硬件来完成错差检测，唯一的开销就是额外余数所需的硬件。而且事实上，我们也可以在进行不那么关键的计算时，临时禁用检错功能，从而进一步降低开销。

也可以通过提供多个冗余余数（redundant residues）来获得对更多错差的检测、校验的能力[Etze80]，其方法类似于前文Section 27.3节中的双剩余码和多剩余码的纠正。同样，唯一需要增加的只是模数和动态范围，检查算法和相应的硬件结构，算术算法不会发生改变。

作为例子，现考虑在模集为 $\{8,7,5,3\}$ （动态范围为 840）的RNS上加入两个冗余模数：13和11。在由此产生的 6模数冗余RNS 表示下，数字 25 将被表示为 $(12,3,1,4,0,1)$ ，其中余数从大模数到小模数排列。现假定模7的余数故障了，该数变为 $(12,3,1,6,0,1)$ 。

利用基拓展，我们可以从其它四个余数中计算出两个冗余的余数；即，我们将 $(-,-,1,6,0,1)$ 转为 $(5,1,1,6,0,1)$ 。而原被破坏的数和重建的数间的这前两个成分的差为 $(+7,+2)$ ，这就是错差伴随式（error syndrome），其指向需要纠正的特定余数。我们看到，这里的纠错方案与Table 27.1中所示的双剩余码非常类似。

参考文献

[Aviz72] Avizienis, A., “Arithmetic Error Codes: Cost and Effectiveness Studies forApplication in Digital System Design,” IEEE Trans. Computers, Vol. 20, No. 11,pp. 1322–1331, 1971.

[Aviz73] Avizienis, A., “Algorithms for Error-Coded Operands,” IEEE Trans. Computers,Vol. 22, No. 6, pp. 567–572, 1973.

[Bars73] Barsi, F., and P. Maestrini, “Error Correcting Properties of Redundant ResidueNumber Systems,” IEEE Trans. Computers, Vol. 22, pp. 307–315, 1973.

[Blum96] Blum, M., and H. Wasserman, “Reflections on the Pentium Division Bug,” IEEETrans. Computers, Vol. 45, No. 4, pp. 385–393, 1996.

[DiCl93] Di Claudio, E. D., G. Orlandi, and F. Piazza, “A Systolic Redundant ResidueArithmetic Error Correction Circuit,” IEEE Trans. Computers, Vol. 42, No. 4,pp. 427–432, 1993.

[Dutt96] Dutt, S., and F. T. Assaad, “Mantissa-Preserving Operations and RobustAlgorithm-Based Fault Tolerance for Matrix Computations,” IEEE Trans. Computers,Vol. 45, No. 4, pp. 408–424, 1996.

[Etze80] Etzel, M. H., and W. K. Jenkins, “Redundant Residue Number Systems for ErrorDetection and Correction in Digital Filters,” IEEE Trans. Acoustics, Speech, andSignal Processing, Vol. 28, No. 5, pp. 538–545, 1980.

[Huan84] Huang, K. H., and J. A. Abraham, “Algorithm-Based Fault Tolerance for MatrixOperations,” IEEE Trans. Computers, Vol. 33, No. 6, pp. 518–528, 1984.

[Mand67] Mandelbaum, D., “Arithmetic Codes with Large Distance,” IEEE Trans. InformationTheory, Vol. 13, pp. 237–242, 1967.

[Nico93] Nicolaidis, M., “Efficient Implementations of Self-Checking Adders and ALUs,” Proc. 23rd Int’ l Symp. Fault-Tolerant Computing, pp. 586–595, 1993.

[Parh78] Parhami, B., and A. Avizienis, “Detection of Storage Errors in Mass Memories UsingArithmetic Error Codes,” IEEE Trans. Computers, Vol. 27, No. 4, pp. 302–308, 1978.

[Parh94] Parhami, B., “A Multi-Level View of Dependable Computing,” Computers andElectrical Engineering, Vol. 20, No. 4, pp. 347–368, 1994.

[Parh02] Parhami, B., “An Approach to the Design of Parity-Checked Arithmetic Circuits,” Proc. 36th Asilomar Conf. Signals, Systems, and Computers, pp. 1084–1088, 2002.

[Parh06] Parhami, B., “Fault-Tolerant Reversible Circuits,” Proc. 40th Asilomar Conf. Signals,Systems, and Computers, pp. 1726–1729, 2006.

[Pete58] Peterson, W. W., “On Checking an Adder,” IBM J. Research and Development,

[Pete59] Peterson, W. W., and M. O. Rabin, "On Codes for Checking Logical Operations," IBM J. Research and Development, Vol. 3, No. 2, pp. 163–168, 1959.

[Rao74]Rao, T. R. N.,Error Codes for Arithmetic Processors, Academic Press, 1974.

[Thor97] Thornton, M. A., "Signed Binary Addition Circuitry with Inherent Even ParityOutput," IEEE Trans. Computers, Vol. 46, No. 7, pp. 811–816, 1997.

[Vija97] Vijay, M., and R. Mittal, "Algorithm-Based Fault Tolerance: A Review," Microprocessors and Microsystems, Vol. 21, pp. 151–161, 1997.

编辑于 04-15

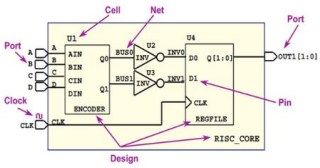
数字电路 数字集成电路 数字IC设计

推荐阅读

firrtl在基于chisel的项目开发中的意义

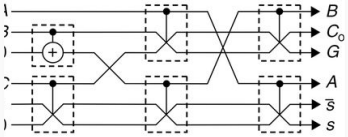
很长时间以来，我认为firrtl这个中间态完全没有意义。chisel足够简洁足够漂亮，verilog足够实用足够完备，从chisel直接走到verilog看起来是自然而然的事情。至于firrtl这个中间产物，我一直...

子非鱼花花 发表于深度学习与...



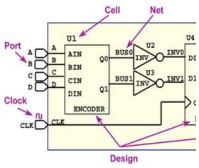
《数字集成电路静态时序分析基础》笔记③

空白的贝塔... 发表于从零开始的...



【硬件算法笔记26】低功耗计算技术

Vortex



《数字集成电路静态时序分析基础》笔记⑥

空白的贝塔... 发表于

2 条评论 切换为时间排序

写下你的评论...

周琦 04-15

你的硬件笔记算法对我很有用。我很多要做的东西都非常需要这方面知识

赞

Vortex (作者) 回复 周琦 04-15

谢谢鼓励！
之前有点赶所以前面大部分都翻的很随意，可能之后有空才会整理一下内容。当然原书就写得特别棒，建议看原书！

赞