[19]中华人民共和国国家知识产权局

[51]Int.Cl[6]
G06F 9/302
G06F 9/38

[12]发明专利申请公开说明书

[21]申请号 97196725.3

[43]公开日 1999年8月18日

[11]公开号 CN 1226325A

[22]申请日 97.8.22 [21]申请号 97196725.3
[30]优先权
[32]96.9.23 [33]GB[31]9619826.2
[86]国际申请 PCT/GB97/02260 97.8.22
[87]国际公布 WO98/12627 英 98.3.26
[85]进入国家阶段日期 99.1.25
[71]申请人 ARM有限公司
地址 英国剑桥郡
[72]发明人 D·V·贾加尔
S·J·格拉斯

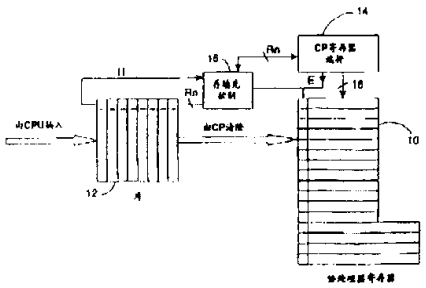
[74]专利代理机构 中国专利代理(香港)有限公司
代理人 王 勇 李亚非

权利要求书2页 说明书73页 附图页数7页

[54]发明名称 数据处理系统中的输入操作数控制

[57]摘要

一个数据处理系统包括多个寄存器(10)和一个算术逻辑单元(20,22,24),该数据处理系统中的程序指令字具有一个源寄存器位字段Sn,它指定存储一个输入操作数数据字的一个寄存器;一个输入操作数长度标志,用于指定所述输入操作数数据字是否具有N位的长度或者N/2位的长度;以及一个高/低位置标志,如果一个输入操作数的长度较小时,该高/低位置标志指示该输入操作数存储在位于所述源寄存器的高位位置和低位位置中的哪一个。该算术逻辑单元最好也能够执行并行操作的程序指令字,以独立操作存储在一个寄存器的相应的各半部分中的(N/2)位输入操作数数据字。



ISSN1008-4274

权　利　要　求　书

1. 一个用于数据处理的装置，所述装置包括：

用于存储要处理的数据字的多个寄存器，每一个所述寄存器具有至少N位的容量；以及

一个算术逻辑单元，用于响应程序指令字执行由所述程序指令字指定的算术逻辑操作；其中

所述算术逻辑单元响应至少一个程序指令字，该程序指令字包括：

(i)　一个源寄存器位字段，它指定存储所述程序指令字的一个输入操作数数据字的所述多个寄存器中的一个源寄存器；

(ii)　一个输入操作数长度标志，用于指定所述输入操作数数据字是否具有N位的长度或者N/2位的长度；以及

(iii)一个高/低位置标志，在所述输入操作数长度标志指示为一个(N/2)位的长度时，该高/低位置标志指示该输入操作数数据字位于所述源寄存器的高位位置和低位位置中的哪一个。

2.　根据权利要求1的装置，还包括一个N位数据总线，用于在一个数据存储设备和所述多个寄存器之间传送数据字。

3.　根据权利要求2的装置，还包括一个输入缓存器，用于从所述N位数据总线接收数据字以及将所述N位数据字提供给所述多个寄存器。

4. 根据前述任一权利要求的装置，其中所述算术逻辑单元响应至少一个并行操作的程序指令字，在第一个(N/2)位输入操作数数据字上和第二个(N/2)位输入操作数数据字上执行单独的算术逻辑操作，这些输入操作数数据字分别存储在一个源寄存器中的高位位置和低位位置。

5. 根据权利要求4的装置，其中所述算术逻辑单元具有一个信号通路，在算术逻辑操作中它起到位位置之间的进位链的作用，并且在执行一个并行操作的程序指令字时，所述信号通路在所述第一个(N/2)位输入操作数数据字和所述第二个(N/2)位输入操作数数据字之间断开。

6. 根据权利要求4或5的装置，其中所述并行操作的程序指令字执行下列算术逻辑操作之一：

(i)　并行加，其中执行两个并行的(N/2)位加；

(ii)　并行减，其中执行两个并行的(N/2)位减；

(iii) 并行移位，其中执行两个并行的(N/2)位移位操作；

(iv)　并行加/减，其中并行执行一个(N/2)位加和一个(N/2)位减。

7. 根据前述任一权利要求的装置，其中在所述输入长度标志指示一个N位长度时，所述高/低位置标志指示在作为一个N位输入操作数数据字使用之前是否将存储在所述高位位置的那些位移动到所述低位位置，以及将存储在所述低位位置的那些位移动到所述高位位置。

8. 根据前述任一权利要求的装置，其中所述算术逻辑单元具有一个N位的数据通路。

9. 根据权利要求8的装置，还包括至少一个多路转换器，该多路转换器响应所述高/低位置标志，选择向所述数据通路的低(N/2)位提供存储在所述源寄存器的高位位置和所述源寄存器的低位位置之一的一个(N/2)位输入操作数数据字。

10. 根据权利要求8或9的装置，还包括一个电路，以在将一个(N/2)位输入操作数数据字输入到所述N位数据通路之前对其进行符号扩展。

11. 一种处理数据的方法，所述方法包括以下步骤:

将要进行处理的数据字存储在多个寄存器中，每一个所述寄存器具有至少N位的容量；以及

响应程序指令字执行由所述程序指令字指定的算术逻辑操作；其中至少一个程序指令字包括:

(i) 一个源寄存器位字段，它指定存储所述程序指令字的一个输入操作数数据字的所述多个寄存器中的一个源寄存器;

(ii) 一个输入操作数长度标志，用于指定所述输入操作数数据字是否具有N位的长度或者N/2位的长度；以及

(iii)一个高/低位置标志，在所述输入操作数长度标志指示为一个(N/2)位的长度时，该高/低位置标志指示该输入操作数数据字位于所述源寄存器的高位位置和低位位置中的哪一个。

说　明　书

数据处理系统中的输入操作数控制

本发明涉及数据处理系统。更具体地，本发明涉及具有多个寄存器的数据处理系统，所述多个寄存器在程序指令字的控制下用于存储由算术逻辑单元操作的数据字。

提供程序指令以指定是否对存储在一个寄存器中的输入操作数进行操作，或者对存储在两个寄存器中的但作为一个输入操作数对待的一个输入操作数进行操作是众所周知的。

从本发明的一方面来看，本发明提供了用于数据处理的装置，所述装置包括:

用于存储要处理的数据字的多个寄存器，每一个所述寄存器具有至少N位的容量；以及

一个算术逻辑单元，用于响应程序指令字执行由所述程序指令字指定的算术逻辑操作；其中

所述算术逻辑单元响应至少一个程序指令字，该程序指令字包括:

(i)　一个源寄存器位字段，它指定存储所述程序指令字的一个输入操作数数据字的所述多个寄存器中的一个源寄存器;

(ii)　一个输入操作数长度标志，用于指定所述输入操作数数据字是否具有N位的长度或者N/2位的长度；以及

(iii) 一个高/低位置标志，在所述输入操作数长度标志指示为一个(N/2)位的长度时，该高/低位置标志指示该输入操作数数据字位于所述源寄存器的高位位置和低位位置中的哪一个。

在数据处理中已经出现提高系统的数据通路宽度的趋势。早期的系统具有8位的数据通路。然后发展成为16位的数据通路，现在常见的是具有32位和64位数据通路。随着数据通路宽度的增加，数据处理系统中的寄存器的宽度也随之增加以与其匹配。本发明认识到，在要处理的数据字的宽度小于数据通路的宽度时，则使用一个全寄存器来存储这些字是对设备的寄存器资源的浪费。这种情况在加载/存储体系结构的机器中尤其如此，其中，要处理的所有数据必须在寄存器中，而且希望减少需要从高速缓存或主存储器中取出数据的次数。本发明认识到上述考虑，并且提供了使用一个输入操作数长度标志和一个高/低位置标志的方案以指示该输入操作数的长度和其存储于寄存器的哪一部分。依次方式，

一个寄存器可以保存一个以上的输入操作数，从而更有效地利用了设备的寄存器资源，并且这些输入操作数仍可以单独地处理。

在利用一个N位数据总线将数据存储设备连接到寄存器时，本发明的这一优点得以增强。在这种情况下，可以利用数据总线一次传送两个操作数，从而更有效地利用了总线带宽，减少了出现性能瓶颈的可能性。

在本发明的较佳实施例中，所述算术逻辑单元响应至少一个并行操作的程序指令字，在第一个(N/2)位输入操作数数据字上和第二个(N/2)位输入操作数数据字上执行单独的算术逻辑操作，这些输入操作数数据字分别存储在一个源寄存器中的高位位置和低位位置。

所提供的并行操作的程序指令字允许算术逻辑单元充分利用它的N位数据通路能力执行两个独立的计算，即使这些输入操作数在长度上小于最大的数据通路宽度。这极大地提高了该系统的数据处理能力，同时没有引起显著的其他开销。

可以进行的一个改进是所述算术逻辑单元具有一个信号通路，在算术逻辑操作中它起到位位置之间的进位链的作用，并且在执行一个并行操作的程序指令字时，所述信号通路在所述第一个(N/2)位输入操作数数据字和所述第二个(N/2)位输入操作数数据字之间断开。

虽然并行操作的程序指令字可以采取许多形式，但最好所述并行操作的程序指令字执行下列算术逻辑操作之一：

(i) 并行加，其中执行两个并行的(N/2)位加；

(ii) 并行减，其中执行两个并行的(N/2)位减；

(iii) 并行移位，其中执行两个并行的(N/2)位移位操作；

(iv) 并行加/减，其中并行执行一个(N/2)位加和一个(N/2)位减。

本发明的更进一步的改进是，在所述输入长度标志指示一个N位长度时，所述高/低位置标志指示在作为一个N位输入操作数数据字使用之前是否将存储在所述高位位置的那些位移动到所述低位位置，以及将存储在所述低位位置的那些位移动到所述高位位置。

在变换操作期间这一特征尤其有用。这一功能的极其有效的硬件实现包括至少一个多路转换器，该多路转换器响应所述高/低位置标志，选择向所述数据通路的低(N/2)位提供存储在所述源寄存器的高位位置和所述源寄存器的低位位置之一的一个(N/2)位输入操作数数据字。

为了处理带符号的运算而又没有不应有的复杂性，最好提供一个电路以在将一个(N/2)位输入操作数数据字输入到所述N位数据通路之前对其进行符号扩展。

从本发明的另一方面来看，本发明提供了用于数据处理的方法，所述方法包括以下步骤:

将要进行处理的数据字存储在多个寄存器中，每一个所述寄存器具有至少 N 位的容量；以及

响应程序指令字执行由所述程序指令字指定的算术逻辑操作；其中至少一个程序指令字包括:

(i) 一个源寄存器位字段，它指定存储所述程序指令字的一个输入操作数数据字的所述多个寄存器中的一个源寄存器;

(ii) 一个输入操作数长度标志，用于指定所述输入操作数数据字是否具有 N 位的长度或者 N/2 位的长度；以及

(iii)一个高/低位置标志，在所述输入操作数长度标志指示为一个(N/2)位的长度时，该高/低位置标志指示该输入操作数数据字位于所述源寄存器的高位位置和低位位置中的哪一个。

下面参照附图以示例方式描述本发明的实施例，附图中:

图 1 示出数字信号处理装置的高层配置;

图 2 示出协处理器的寄存器配置的输入缓冲器;

图 3 示出通过协处理器的数据路径;

图 4 示出从寄存器中读取高或低位位的多路复用电路;

图 5 为示出较佳实施例中的协处理器所使用的寄存器重新映象逻辑的框图;

图 6 更详细地示出图 5 中所示的寄存器重新映象逻辑；以及

图 7 为示出块过滤算法的表。

下面描述的系统是关于数字信号处理（DSP）的。DSP 可采取许多形式，但一般可以认为是需要高速（实时）处理大量数据的处理。这一数据通常表示某种模拟物理信号。DSP 的好的实例便是用在数字移动电话中的，其中所接收与发送的无线电信号需要解码成模拟声音信号及将模拟声音信号编码（通常采用卷积、变换及相关运算）。另一实例是盘驱动器控制器，其中处理从盘头恢复的信号以产生头跟踪控制。

在上面的上下文中，下面是对基于与协处理器合作的微处理器核（在本例中为英国剑桥先进 RISC 机器有限公司设计的微处理器范围内的 ARM 核）的数字信号处理系统的描述。微处理器与协处理器的接口及协处理器体系结构本身是专门为提供 DSP 功能配置的。微处理器

核将被称作 ARM 而协处理器称作 Piccolo。ARM 与 Piccolo 通常制造成包含作为 ASIC 的一部分的其它元件（如片上 DRAM、ROM、D/A 与 A/D 转换器等）的单一集成电路。

Piccolo 为 ARM 协处理器，因此它执行一部分 ARM 指令集。ARM 协处理器指令允许 ARM 在 Piccolo 与存储器之间传送数据（利用加载协处理器 LDC 及存储协处理器 STC 指令），以及向与从 Piccolo 传送 ARM 寄存器（利用传送到协处理器 MCR 及从协处理器传送的 MRC 指令）。观察 ARM 与 Piccolo 的协作交互作用的一种方式是 ARM 作为 Piccolo 数据的强有力的地址发生器工作，而使 Piccolo 有时间执行需要实时处理大量数据来产生对应的实时结果的 DSP 运算。

图 1 示出 ARM 2 与 Piccolo 4，ARM 2 发布控制信号到 Piccolo 4 来控制向 Piccolo 4 传送数据以及从 Piccolo 4 传送数据字。指令高速缓冲存储器 6 存储 Piccolo 4 所需要的 Piccolo 程序指令字。单个 DRAM 存储器 8 存储 ARM 2 与 Piccolo 4 两者所需要的所有数据与指令字。ARM 2 负责寻址存储器 8 及控制所有数据传送。只带单个存储器 8 及一组数据与地址总线的布置比需要多个存储器及高总线带宽的总线的典型 DSP 方法简单与低廉。

Piccolo 执行来自控制 Piccolo 数据路径的指令高速缓冲存储器 6 的第二指令流（数字信号处理程序指令字）。这些指令中包含诸如乘－累加等数字信号处理型操作及诸如零开销循环指令等控制流指令。这些指令在保持在 Piccolo 寄存器 10（见图 2）中的数据上操作。这一数据是早先 ARM 2 从存储器 8 传送来的。指令流自指令高速缓冲存储器 6；指令高速缓冲存储器 6 作为完全的总线主驱动数据总线。小的 Piccolo 指令高速缓冲存储器 6 为 4 线、每线 16 个字的直接映象高速缓冲存储器（64 条指令）。在一些实现中，令指令高速缓冲存储器更大是值得的。

从而两个任务是独立运行的，ARM 加载数据而 Piccolo 处理它。这允许在 16 位数据上持续的单周期数据处理。Piccolo 具有使 ARM 预取顺序数据，在 Piccolo 需要它之前加载数据的数据输入机制（示出在图 2 中）。Piccolo 能以任何次序存取加载的数据，随着老数据的最后一次使用自动地重新填充其寄存器（所有指令的每一源操作数都有一位来指示应重新填充源寄存器）。这一输入机制称作再定序缓

冲器并包括输入缓冲器 12。加载进 Piccolo 的每一个值（见下面通过 LDC 或 MCR）携带有指定该值的目的地寄存器的标记 Rn。标记 Rn 与数据字一起存储在输入缓冲器中。当通过寄存器选择电路 14 存取寄存器而指令指定要重新填充该数据寄存器时，便通过确立信号 E 来标记该寄存器。然后重新填充电路 16 用输入缓冲器 12 中以该寄存器为目的地的最老的加载值自动重新填充该寄存器。重定序缓冲器保持 8 个带标记的值。输入缓冲器 12 具有类似于 FIFO 的形式，但除外可从队列中央抽取数据字，而此后较晚存储的字向前传递来填充空位。距离输入最远的数据字便相应地是最老的，并在输入缓冲器 12 保持带有正确的标记 Rn 的两个数据字时便用它来确定应当用哪一个数据字来重新填充输入缓冲器 12。

如图 3 中所示 Piccolo 通过将数据存储在输出缓冲器 18（FIFO）中输出它。数据是顺序地写入 FIFO 中的，并由 ARM 以相同的次序读出到存储器 8。输出缓冲器 18 保持 8 个 32 位值。

Piccolo 通过协处理器接口（图 1 的 CP 控制信号）连接在 ARM 上。在执行 ARM 协处理器指令时，Piccolo 能执行该指令；在执行该指令之前令 ARM 等待直到 Piccolo 就绪；或拒绝执行该指令。在最后一种情况中，ARM 将引起未定义的指令异常。

Piccolo 执行的最普通的协处理器指令为 LDC 与 STC，它们分别通过数据总线向与从存储器 8 加载与存储数据字，而 ARM 生成所有地址。便是这些指令将数据加载到重定序缓冲器中并存储来自输出缓冲器 18 的数据。如果在 LDC 上输入重定序缓冲器中没有足够的空间来加载数据时，及如果在 STC 上输出缓冲器中没有足够的数据供存储，即 ARM 正在期待的数据不在输出缓冲器 18 中时，Piccolo 将阻止 ARM。Piccolo 还执行 ARM/协处理器寄存器传送使 ARM 能存取 Piccolo 的特定寄存器。

Piccolo 从存储器取出其本身的指令来控制图 3 中所示的数据路径及从重定序缓冲器到寄存器及从寄存器到输出缓冲器 18 传送数据。Piccolo 的执行这些指令的算术逻辑单元具有执行乘法、加法、减法、乘－累加、逻辑运算、移位与循环的乘法器/加法器电路 20。在数据路径中还设置有累加/累减（decumulate）电路 22 及定标/饱和电路 24。

Piccolo 指令是初始时从存储器加载进指令高速缓冲存储器 6 中的，其中 Piccolo 能存取它们而不需要返回去存取主存储器。

Piccolo 不能从存储器失败中恢复。因此，如果在虚拟存储器系统中使用 Piccolo，在整个 Piccolo 任务中所有 Piccolo 数据都必须在物理存储器中。对于诸如实时 DSP 等 Piccolo 任务的实时性质，这不是重大的限制。如果出现存储器失败，Piccolo 将停止并在状态寄存器 S2 中设置标志。

图 3 示出 Piccolo 的整体数据路径功能。寄存器组 10 使用 3 个读端口与 2 个写端口。利用一个写端口（L 端口）从重定序缓冲器重新填充寄存器。输出缓冲器 18 是直接从 ALU 结果总线 26 更新的，从输出缓冲器 18 的输出是在 ARM 程序控制下的。ARM 协处理器接口执行到重定序缓冲器中的 LDC（加载协处理器）指令及从输出缓冲器 18 的 STC（存储协处理器）指令，以及在寄存器组 10 上的 MCR 与 MRC（传送 ARM 寄存器至/自 CP 寄存器）。

其余寄存器端口用于 ALU。两个读端口（A 与 B）驱动输入到乘法器/加法器电路 20，C 读端口用于驱动累加器/累减器电路 22 输入。其余写端口 W 用于将结果返回给寄存器组 10。

乘法器 20 执行 16×16 带符号或不带符号乘法，带有可选用的 48 位累加。定标器单元 24 能提供 0 至 31 位立即算术或逻辑右移，后面跟随可选用的饱和。移位器与逻辑单元 20 每一周期能执行一个移位或逻辑运算。

Piccolo 具有称作 D0－D15 或 A0－A3、X0－X3、Y0－Y3、Z0－Z3 的 16 个通用寄存器。第一组四个寄存器（A0－A3）预定作为累加器并且是 48 位宽，额外的 16 位提供在许多连续的计算中对溢出的保护。其余寄存器为 32 位宽。

可将各 Piccolo 寄存器作为包含两个独立的 16 位值对待。位 0 至 15 包含低的一半，位 16 至 31 包含高的一半。指令能指定各寄存器特定的 16 位的一半作为源操作数，或可指定整个 32 位寄存器。

Piccolo 还提供饱和的运算。如果结果大于目的地寄存器的大小，乘法、加法与减法指令的变型提供饱和的结果。当目的地寄存器为 48 位累加器时，将值饱和到 32 位（即无法饱和 48 位值）。在 48 位寄存器上没有溢出检测。由于会占用至少 65536 条乘法累加指令才

能导致溢出所以这是合理的限制。

各 Piccolo 寄存器是标记为“空”（E 标志，见图 2）或包含值（不可能有半个寄存器是空的）之一的。初始时，将所有寄存器标记为空。在各周期上 Piccolo 试图用重新填充控制电路 16 将来自输入重定序缓冲器的值填充空的寄存器之一。此外如果将来自 ALU 的值写入寄存器便不再将它标记为“空”的。如果从 ALU 写入寄存器，同时有值等待从重定序缓冲器放置到该寄存器中，则结果是不确定的。如果对空寄存器进行读取，Piccolo 的执行单元将停止。

输入重定序缓冲器（ROB）位于协处理器接口与 Piccolo 的寄存器组之间。用 ARM 协处理器传送将数据加载进 ROB 中。ROB 包含若干 32 位值，各带有指示作为该值的目的地的 Piccolo 寄存器的标记。该标记还指示该数据应传送给整个 32 位寄存器还是只给 32 位寄存器的底部 16 位。如果数据的目的地为整个寄存器，则将该项的底部 16 位传送给目标寄存器的底部一半并将顶部 16 位传送给寄存器的顶部一半（如果目标寄存器为 48 位累加器则扩展符号）。如果该数据的目的地只是寄存器的底部一半（所谓“半寄存器”），首先传送底部 16 位。

寄存器标记总是参照物理目的地寄存器，不执行寄存器重新映射（见下面关于寄存器重新映射。）

在每一个周期上 Piccolo 试图如下地将数据项从 ROB 传送到寄存器组：

- 检验 ROB 中各项并将标记与空寄存器比较，确定是否能从一部分或全部项对寄存器进行传送。

- 从能进行传送的项组中，选择最老的项并将其数据传送给寄存器组。

- 将该项的标记更新为标记该项是空的。如果只传送了该项的一部分，只将传送的部分标记为空的。

例如，如果目标寄存器完全是空的且选择的 ROB 项包含以整个寄存器为目的地的数据，便传送全部 32 位并标记该项为空的。如果目标寄存器的底部一半是空的而 ROB 项包含目的地为寄存器的底部一半的数据，则将该 ROB 项的底部 16 位传送给目标寄存器的底部一半并将 ROB 的底部一半标记为空的。

可以独立地传送任何项中的数据的高与低 16 位。如果没有项包含能传送给寄存器组的数据，该周期中不进行传送。下面的表描述目标 ROB 项与目标寄存器状态的所有可能组合。

	目标，Rn，状态		
目标 ROB 项状态	空	低一半空	高一半空
全寄存器，两半都有效	Rn.h<-entry.h Rn.l<-entry.l 项标记为空	Rn.l<-entry.l entry.l 标记为空	Rn.l<-entry.h entry.h 标记为空
全寄存器，高一半有效	Rn.h<-entry.h 项标记为空		Rn.h<-entry.h 项标记为空
全寄存器，低一半有效	Rn.l<-entry.l 项标记为空	Rn.l<-entry.l 项标记为空	
半寄存器，两半都有效	Rn.l<-entry.l entry.l 标记为空	Rn.l<-entry.l etntry.l 标记为空	
半寄存器，高一半有效	Rn.l<-entry.h 项标记为空	Rn.l<-entry.h 项标记为空	

总结一下，可以独立地从 ROB 重新填充寄存器的两半，ROB 中的数据标记为以整个寄存器为目的地或以寄存器的底部一半为目的地的两个 16 位值。

用 ARM 协处理器指令将数据加载进 ROB 中。如何在 ROB 中标记数据取决于用哪一条协处理器指令来执行传送。下述 ARM 指令可用于以数据填充 ROB:

```
LDP{<cond>}<16/32>    <dest>,[Rn]{!},#<size>
LDP{<cond>}<16/32>W   <dest>,<wrap>,[Rn]{!},#<size>
LDP{<cond>}16U        <bank>,[Rn]{!}
MPR{<cond>}           <dest>,Rn
MRP{<cond>}           <dest>,Rn
```

提供了下列 ARM 指令用于配置 ROB:

```
LDPA<bank list>
```

前三条被汇编为 LDC，MPR 与 MRP 被汇编为 MCR，LDPA 被汇编为 CDP 指令。

上面<dest>代表 Piccolo 寄存器（A0－Z3），Rn 代表一个 ARM 寄存器，<size>代表必须是 4 的非零倍数的固定字节数，而<wrap>代表常量（1、2、4、8）。用{}括起的字段为选用的。为了使传送能符合重定序缓冲器，<size>至多为 32。在许多场合中，为了避免死锁，<size>将小于这一限制。<16/32>字段指示是否应将加载的数据作为 16 位数据对待并指示要采取的结尾（endian）特定的动作（见下面），或者是 32 位数据。

注 1：在下面的正文中，当引用 LDP 或 LDPW 时它指指令的 16 位与 32 位变型两者。

注 2：‘字’为来自存储器的 32 位块，它可包含两个 16 位数据项或一个 32 位数据项。

LDP 指令传送若干数据项，将它们指派到一个全寄存器。这一指令将从存储器中地址 Rn 加载<size>/4 个字，将它们插入 ROB 中。能传送的字数受下面的限制：

－量<size>必须是 4 的非零倍数；

－<size>必须小于或等于特定实现的 ROB 的大小（在第一版本中为 8 个字，未来版本中保证不少于此）。

将传送的第一数据项标记为指派到<dest>的，第二数据项指派到<dest>＋1 等等（从 Z3 绕回到 A0）。如果指定了！，则此后将寄存器 Rn 增量<size>。

如果采用 LDP16 变型，随着它们从存储器系统返回，在构成 32 位数据项的 2 个 16 位半字上执行对结尾（endian）特定的操作。详情见下面大结尾（Big Endian）与小结尾（Little Endian）支持。

LDPW 指令传送若干数据项到一组寄存器。将传送的第一数据项标记为指派到<dest>，第二到<dest>＋1，等等。当出现<wrap>传送时，将下一个传送的项标记为指派到<dest>，等等。<wrap>量是在半字的量指定的。

对于 LDPW，适用下述限制：

－量<size>必须是 4 的非零倍数；

－<size>必须小于或等于特定实现的 ROB 的大小（在第一版中为 8 个字，未来版本中保证不小于此）；

－<dest>可以是{A0、X0、Y0、Z0}之一；

- 对于 LDP32W，<wrap>可以是{2、4、8}个半字之一，对于 LDP16W 可以是{1、2、4、8}个半字之一；

- 量<size>必须大于 2*<wrap>，否则不出现回绕而应用 LDP 指令来代替.

例如，指令

LDP32W　　X0，2，[R0]!,#8

将两个字加载进 ROB 中，将它们指派给整个寄存器 X0. R0 将被增量 8. 指令

LDP32W　　X0，4，[R0]，#16

将四个字加载进 ROB 中，将它们标记为指派给 X0,X1，X0，X1（按此次序）。R0 不受影响.

对于 LDP16W，可将<wrap>指定为 1、2、4 或 8. 1 的回绕将导致所有数据标记为指派给目的地寄存器<dest>.1 的底部一半. 这是‘半寄存器’情况.

例如，指令

LDP16W　　X0，1，[R0]!,#8

将两个字加载进 ROB 中,将它们标记为指派给 X0.1 的 16 位数据. R0 将被增量 8. 指令

LDP16W　　X0，4，[R0],#16

的表现类似于 LDP32W 实例，但是在它从存储器返回时在数据上执行对于结尾特定的操作除外.

LDP 指令所有未使用的编码可为将来扩展保留.

LDP16U 指令是为支持 16 位不对齐的数据的高效传送而提供的. LDP16U 支持是为寄存器 D4 至 D15（X、Y 与 Z 组）提供的. LDP16U 指令将一个 32 位数据字（包含两个 16 位数据项）从存储器传送到 Piccolo 中. Piccolo 将丢弃这一数据的底部 16 位而将顶部 16 位存储在保持寄存器中. X、Y 与 Z 组有一保持寄存器. 一旦装填了组中的保持寄存器，如果将数据指派给该组中的寄存器，便改变了 LDP{W}指令的表现. 加载进 ROB 中的数据由保持寄存器与正在用 LDP 指令传送的数据的底部 16 位的连接构成. 将正在传送的数据的高 16 位放入保持寄存器中:

entry<-data.l¦holding_register

holding_register<-data.h

这一操作模式一直持续到用 LDPA 指令关闭为止。保持寄存器并不记录目的地寄存器标记或大小。这一特征是从提供 data.l 的下一个值的指令获得的。

结尾的特定行为可永远出现在存储器系统返回的数据上。由于假定所有 32 位数据项在存储器中都是字对齐的，不存在等效于 LDP16U 的非 16 位指令。

LDPA 指令用于关闭 LDP16U 指令起动的不对齐操作模式。可以在组 X、Y、Z 上独立关闭不对齐模式。例如指令，

LDPA {X, Y}

将关闭组 X 与 Y 上的不对齐模式。这些组的保持寄存器中的数据将被丢弃。

允许在不处于非对齐模式的组上执行 LDPA，这将使该组在对齐模式中。

MPR 指令将 ARM 寄存器 Rn 的内容放入 ROB 中，指派给 Piccolo 寄存器<dest>。目的地寄存器<dest>可以是范围 A0－Z3 中的任何全寄存器。例如指令，

MPR X0, R3

将 R3 的内容传送到 ROB 中，将数据标记为指派给全寄存器 X0。

由于ARM是内部小结尾(endian)的，将数据从ARM传送到Piccolo时不出现对结尾特定的表现。

MPRW 指令将 ARM 寄存器 Rn 的内容放置在 ROB 中，将其标记为指派给 16 位 Piccolo 寄存器<dest>.l 的两个 16 位数据项。对<dest>的限制与对 LDPW 指令的相同（即 A0、X0、Y0、Z0）。例如指令，

MPRW X0, R3

将 R3 的内容传送到 ROB 中，将数据标记为指派给 X0.l 的两个 16 位量。应指出对于带有 1 回绕的 LDP16W，只能针对 32 位寄存器的底部一半。

至于 MPR，在数据上不作用对于结尾特定的操作。

将 LDP 编码为：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COND				110			P	U	N	W	1	Rn				DEST				PICCOLO1				SIZE/4								

其中 PICCOLO1 为 Piccolo 的第一协处理器号（当前为 8）。N 位在 LDP32（1）与 LDP16（0）之间选择。

LDPW 编码为:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COND				110			P	U	N	W	1	Rn				DES		WRA		PICCOLO2				SIZE/4								

其中 DEST 对于目的地寄存器 A0、X0、Y0、Z0 为 0－3 而 WRAP 对于回绕值 1、2、4、8 为 0－3。PICCOLO2 为 Piccolo 的第二协处理器号（当前为 9）。N 位在 LDP32（1）与 LDP16（0）之间选择。

将 LDP16U 编码为:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COND				110			P	U	0	W	1	Rn				DES		01		PICCOLO2				00000001								

其中 DEST 对于目的地组 X、Y、Z 为 1－3。

将 LDPA 编码为:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COND				1110				0000				0000				0000				PICCOLO1				000			0	BANK			

其中 BANK[3: 0]用于在每组的基础上关闭不对齐模式。如果设置了 BANK[1]，则关闭组 X 上的不对齐模式。BANK[2]与 BANK[3]分别关闭组 Y 与 Z 上的不对齐模式，如果设置的话。注意，这是 CDP 操作。

将 MPR 编码为:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COND				1110				0	1	0	0	DEST				Rn				PICCOLO1				000			1	0000			

将 MPRW 编码为:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COND				1110				0	1	0	0	DEST		00		Rn				PICCOLO2				000			1	0000			

其中 DEST 对于目的地寄存器 X0、Y0、Z0 为 1－3。

输出 FIFO 能保持多达 8 个 32 位值。它们是用下述（ARM）操作码之一从 Piccolo 传送的:

STP{<cond>}<16/32>　　[Rn]{!}, #<size>

MRP　　Rn

第一个将来自输出 FIFO 的<size>/4 个字保存在 ARM 寄存器 Rn 给定的地址上，如果！存在，变址 Rn。为防止死锁，<size>不得大于输出 FIFO 的大小（本实现中为 8 项）。如果采用 STP16 变型，在存储器系统返回的数据上可出现对于结尾特定的表现。

MRP 指令从输出 FIFO 中消除一个字并将其放置在 ARM 寄存器 Rn 中。对于 MPR 在数据上不作用对于结尾特定的操作。

STP 的 ARM 编码为:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COND				110			P	U	N	W	0	Rn				0000				PICCOLO1				SIZE/4							

其中 N 在 STP32（1）与 STP16（0）之间选择。对于 P、U 与 W 位的定义，参见 ARM 资料手册。

MRP 的 ARM 编码为:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COND	1110	0	1	0	1	0000	Rn	PICCOLO1	000	1	0000	

Piccolo 指令集内部假定小结尾(little endian)操作。例如，在存取作为两个 16 位的 32 位寄存器时，假定低一半占用位 15 至 0。Piccolo 可在带有大结尾（big endian）存储器或外围设备的系统中操作，因此必须注意以正确方式加载 16 位分组数据。

诸如 ARM（如英国剑桥的先进 RISC 机器有限公司生产的 ARM7 微处理器）等 Piccolo 具有程序员能控制的 'BIGEND' 配置管脚，控制可以是用可编程外围设备进行的。Piccolo 利用该管脚来配置输入重定序缓冲器及输出 FIFO。

当 ARM 将分组的 16 位数据加载到重定序缓冲器中时，它必须用 LDP 指令的 16 位格式指示这一点。这一信息与 'BIGEND' 配置输入的状态组合以适当的次序将数据放置在保持锁存器与重定序缓冲器中。尤其是在大结尾模式中，保持寄存器存储加载的字的底部 16 位，并与下一次加载的顶部 16 位配对。保持寄存器内容永远结束在传送到重定序缓冲器中的字的底部 16 位中。

输出 FIFO 可包含分组 16 位或 32 位数据。程序员必须使用 STP 指令的正确格式以便 Piccolo 能保证将 16 位数据提供在数据总线的正确一半上。当配置成大结尾时，在使用 16 位格式的 STP 时，上与下 16 位两半互换。

Piccolo 具有只能从 ARM 存取的 4 个专用寄存器。它们称作 S0-S2。它们只能用 MRC 与 MCR 指令存取。操作码为:

MPSR Sn, Rm

MRPS Rm, Sn

这些操作码在 ARM 寄存器 Rm 与专用寄存器 Sn 之间传送 32 位值。它们是作为协处理器寄存器传送在 ARM 中编码的:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
COND	1110	001	L	Sn	Rm	PICCOLO	000	1	0000

其中对于 MPSR，L 为 0 而对 MRPS，L 则为 1。

寄存器 S0 包含 Piccolo 唯一的 ID 及修订版本代码。

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
实现者		体系结构		部件号			修订版本

位[3: 0]包含处理器的修订版本号。

位[15:4]包含以二进制编码的十进制格式的 3 位部件号:piccolo 为 0x500

位[23:16]包含体系结构版本: 0×00 = 版本 1

位[31: 24]包含实现者商标的 ASCII 码: 0×41 = A = ARM 有限公司

寄存器 S1 为 Piccolo 状态寄存器。

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
N	Z	C	V	SN	SZ	SC	SV	保留			D	A	H	B	U	E

一级状态码标志（N、Z、C、V）

二级状态码标志（SN、SZ、SC、SV）

E 位: Piccolo 已被 ARM 禁止并已停止。

U 位: Piccolo 遇到未定义的指令并已停止。

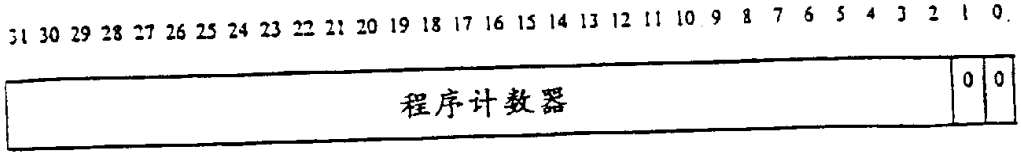
B 位: Piccolo 遇到断点并已停止。

H 位: Piccolo 遇到停止指令并已停止。

A 位: Piccolo 遇到存储器失败（加载、存储或 Piccolo 指令）并已停止。

D 位: Piccolo 检测到死锁条件并已停止（见下）.

寄存器 S2 为 Piccolo 程序计数器:



写入程序计数器起动 Piccolo 在该地址上执行程序（如果停止则离开停止状态）. 复位时程序计数器是无定义的, 因为 Piccolo 总是通过写入程序计数器起动的.

执行期间, Piccolo 监视指令的执行及协处理器接口的状态. 如果它检测到:

- Piccolo 已停止运行等待重新装填寄存器或等待输出 FIFO 具有可利用的项.

- 协处理器接口为忙等待, 由于 ROB 中空间不够或输出 FIFO 中项不够.

如果检测到这两种状态, Piccolo 置位其状态寄存器中的 D 位、停止并拒绝 ARM 处理器指令, 导致 ARM 进入未定义指令陷阱.

死锁状态的检测允许将系统构成为通过读取 ARM 与 Piccolo 程序计数器及寄存器至少能警告程序员已出现该状态及报告精确的故障点. 应强调死锁只能由于不正确的程序或系统的另一部分破坏 Piccolo 的状态引发. 死锁不能由数据不足或‘过载’引发.

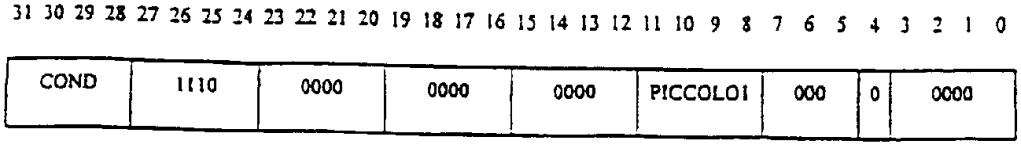
可采用若干种操作从 ARM 控制 Piccolo, 它们是由 CDP 指令提供的. 这些 CDP 指令只在 ARM 在特权状态中才接受. 如果不在该状态中 Piccolo 将拒绝 CDP 指令而导致 ARM 处于未定义的指令陷阱. 下面为可利用的操作:

- 复位
- 进入状态访问模式
- 启动
- 禁止

Piccolo 可用 PRESET 指令在软件中复位.

```
PRESET          ; 清除 piccolo 的状态
```

将这一指令编码为



执行这一指令时出现以下情况:

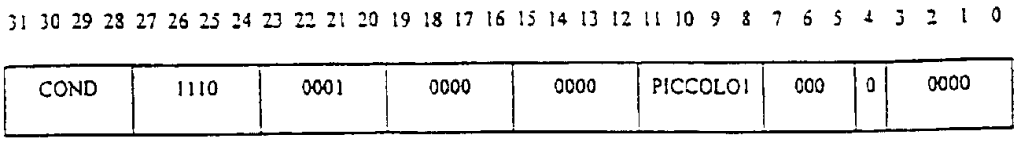
- 将所有寄存器标记为空（准备好重新填充）。
- 清除输入 ROB。
- 清除输出 FIFO。
- 复位循环计数器。
- 将 Pioccolo 置于停止状态（将置位 S2 的 H 位）。

执行 PRESET 指令可占用若干周期来完成（对于本实施例 2－3）。在正在执行它时，后面要在 Piccolo 上执行的 ARM 协处理器指令将处于忙等待。

在状态访问模式中，可使用 STC 及 LDC 指令保存与恢复 Piccolo 的状态（见下面关于从 ARM 访问 Piccolo 状态）。为了进入状态访问模式，必须首先执行 PSTATE 指令:

PSTATE　　进入状态访问模式

将这一指令编码为:



在执行时，PSTATE 指令将:

- 停止 Piccolo（如果它尚未停止），置位 Piccolo 的状态寄存器中的 E 位。
- 配置 Piccolo 进入其状态访问模式中。

执行 PSTATE 指令可占用若干周期来完成，由于在停止以前 Piccolo 的指令流水线必须用完。当正在执行时，后面要在 Piccolo 上执行的 ARM 协处理器指令将是忙等待。

将 PENABLE 与 PDISABLE 指令用于快速上下文切换. 当 Piccolo 被禁止时, 只能访问专用寄存器 0 与 1 (ID 与状态寄存器), 并且只是从特权模式时. 访问任何其它状态或从用户模式的任何访问将导致 ARM 未定义指令异常. 禁止 Piccolo 导致它停止执行. 当 Piccolo 已停止执行时, 它通过置位状态寄存器中的 E 位来确认这一事实.

Piccolo 是通过执行 PENABLE 指令启动的:

PENABLE; 启动 Piccolo

将这一指令编码为:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
COND	1110	0010	0000	0000	PICCOLO1	000	0	0000

Picclol 是通过执行 PDISABLE 指令禁止的:

PDISABLE ; 禁止 Piccolo

将这一指令编码为:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
COND	1110	0011	0000	0000	PICCOLO1	000	0	0000

在执行这一指令时, 出现以下情况:

- Piccolo 的指令流水线将流完.

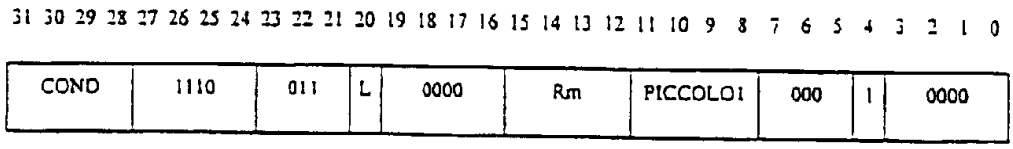
-Piccolo 将停机及置位状态寄存器中的 H 位.

Piccolo 指令高速缓冲存储器保持控制 Piccolo 数据路径的 Piccolo 指令. 如果存在, 它保证保持至少 64 条指令, 起始在 16 个字边界上. 下面的 ARM 操作码汇编进 MCR 中. 其操作为强制高速缓冲存储器取出起始在指定地址上 (必须是 16 字边界) 的一行 (16 条) 指令. 即使高速缓冲存储器已保持有关于这一地址的数据也发生这一取出.

PMIR Rm

在能执行 PMIR 之前 Piccolo 必须停止.

这一操作码的 MCR 编码为：



本节讨论控制 Piccolo 数据路径的 Piccolo 指令集.各指令为 32 位长。指令是从 Piccolo 指令高速缓冲存储器中读取的。

解码指令集是相当直观的。高 6 位（26 至 31）给出主操作码，位 22 至 25 为少数特定指令提供次要操作码。带灰色阴影的位当前不使用而为扩展保留（当前它们必须包含指定值）。

有 11 个主要指令类。这并不完全对应于提出在指令中的主操作码，这是为了便于解码某些子类。

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0																														
0	OPC					F	SD	DEST			S1	R1	SRC1						SRC2											
1	000			OPC		F	SD	DEST			S1	R1	SRC1						SRC2											
1	001			0	OP	F	SD	DEST			S1	R1	SRC1						SRC2											
1	0011																													
1	010			OPC		F	SD	DEST			S1	R1	SRC1						SRC2_SHIFT											
1	011			00		F	SD	DEST			S1	R1	SRC1						SRC2_SEL									COND		
1	011			01																										
1	011			1	OP	F	SD	DEST			S1	R1	SRC1						SRC2_SEL									COND		
1	10		0	OP	Sa	F	SD	DEST			A1	R1	SRC1						SRC2_MULA											
1	10	1			0																									
1	10	1	OP		1	F	SD	DEST			A1	R1	SRC1			0	A0	R2	SRC2_REG									SCALE		
1	110																													
1	11100					F	SD	DEST			IMMEDIATE_15																	- /-		
1	11101																													
1	11110					0	RFIELD_4				0	R1	SRC1						#INSTRUCTIONS_8											
1	11110					1	RFIELD_4				#LOOPS_13								#INSTRUCTION_8											
1	11111					0	OPC				REGISTER_LIST_16																	SCALE		
1	11111					100					IMMEDIATE_16																	COND		
1	11111					101					PARAMETERS_21																			
1	11111					11	OP																							

上表中的指令具有以下名称:

标准数据运算

逻辑运算

条件加/减

未定义

移位

选择

未定义

并行选择

乘累加

未定义

双倍乘

未定义

移动带符号立即数

未定义

重复

重复

寄存器列表操作

转移

重新命名参数传送

停止/中断

在下面的节中详细描述各类指令的格式。对于大多数指令，源与目的地操作数字段是通用的并在单独的节中详细描述，寄存器重新映射也一样。

大多数指令需要两个源操作数；源 1 与源 2。某些例外是饱和绝对值。

源 1（SRC1）操作数具有以下 7 位格式:

18	17	16 15 14 13	12
大小	再填充	寄存器号	高/低

字段元素具有下述含义:

- 大小 - 指示要读的操作数大小（1 = 32 位，0 = 16 位）。

- 再填充 - 规定读后应将寄存器标记为空的并能从 ROB 再填充。

- 寄存器号 - 编码要读取的 16 个 32 位寄存器中哪一个.

- 高/低 - 对于 16 位读指示读取 32 位寄存器的哪一半. 对于 32 位操作数, 置位时指示应互换两个 16 位一半.

大小	高/低	存取的寄存器部分
0	0	低 16 位
0	1	高 16 位
1	0	全 32 位
1	1	全 32 位, 两半交换

在汇编程序中通过在寄存器号上加上后缀来指定寄存器大小: l 为低 16 位, h 为高 16 位或.x 为具有高与低 16 位交换的 32 位.

通用的源 2 (SRC2) 具有以下三种 12 位格式之一:

11	10	9	8	7	6	5	4	3	2	1	0
0	S2	R2	寄存器号				高/低	标度			
1	0	ROT	IMMED_8								
1	1	IMMED_6						标度			

图 4 示出根据高/低位与大小位将所选择的寄存器的适当的一半切换到 Piccolo 数据路径上的多路复用器装置. 如果大小位指示 16 位, 则符号扩展电路根据需要用 0 或 1 填充数据路径的高位.

第一种编码指定源为寄存器, 这些字段具有与 SRC1 说明符相同的编码. 标度 (SCALE) 字段指定要作用在 ALU 的结果上的标度.

标度				操作
3	2	1	0	
0	0	0	0	ASR #0
0	0	0	1	ASR #1
0	0	1	0	ASR #2
0	0	1	1	ASR #3
0	1	0	0	ASR #4
0	1	0	1	保留
0	1	1	0	ASR #6
0	1	1	1	ASL #1
1	0	0	0	ASR #8
1	0	0	1	ASR #16
1	0	1	0	ASR #10
1	0	1	1	保留
1	1	0	0	ASR #12
1	1	0	1	ASR #13
1	1	1	0	ASR #14
1	1	1	1	ASR #15

带有循环编码的 8 位立即数允许生成可用 8 位值及 2 位循环表示的 32 位立即数。下表示出能从 8 位值 XY 生成的立即数值:

循环	立即数
00	0x000000XY
01	0x0000XY00
10	0x00XY0000
11	0xXY000000

6 位立即数编码允许使用 6 位不带符号的立即数（从 0 到 63），以及作用在 ALU 的输出上的标度。

通用源 2 编码对于大多数指令变型是通用的。对这一规则存在一些例外，它们支持源 2 编码的有限子集或将其稍加修改:

- 选择指令。
- 移位指令。
- 并行操作。
- 乘累加指令。
- 乘双倍指令。

选择指令只支持寄存器或 6 位不带符号立即数的一个操作数。由于这些位由指令的状态字段使用而得使该标度不可用。

SRC2_SEL

11	10	9	8	7	6	5	4	3	2	1	0
0	S2	R2	寄存器号				高/低	状态			
1	1	IMMED_6						状态			

移位指令只支持 16 位寄存器或 1 与 31 之间的 5 位无符号立即数的一个操作数。不能得到结果的标度。

SRC2_SHIFT

11	10	9	8	7	6	5	4	3	2	1	0
0	0	R2	寄存器号				高/低	0	0	0	0
1	0	0	0	0	0	0	IMMED_5				

在并行操作情况中，如果指定寄存器作为操作数的源，则必须执行 32 位读。并行操作的立即数编码略为不同。它允许将一个立即数复制到 32 位操作数的两个 16 位一半中。并行操作可利用稍加限制范围的标度。

SRC2_PARALLEL

11	10	9	8	7	6	5	4	3	2	1	0
0	1	R2	寄存器号				高/低	SCALE_PAR			
1	0	ROT	IMMED_8								
1	1	IMMED_6						SCALE_PAR			

如果使用 6 位立即数，则总是将它复制到 32 位量的两个一半上。如果使用 8 位立即数，只有当循环指示应将 8 位立即数循环到 32 位量的顶部一半上时才复制。

循环	立即数
00	0x000000XY
01	0x0000XY00
10	0x00XY00XY
11	0xXY00XY00

并行选择操作不使用标度；必须将这些指令的标度字段设置为0。

乘累加指令不允许指定 8 位循环立即数。该字段的位 10 用来部分地指定使用哪一累加器。源 2 蕴含 16 位操作数。

SRC2_MULA

11	10	9	8	7	6	5	4	3	2	1	0
0	A0	R2	寄存器号				高/低	标度			
1	A0	IMMED_6						标度			

乘双倍指令不允许使用常量。只能指定一个 16 位寄存器。该字段的位 10 用来部分地指定使用哪一个累加器。

SRC2_MULD

11	10	9	8	7	6	5	4	3	2	1	0
0	A0	R2	寄存器号				高/低	标度			

某些指令总是蕴含 32 位操作（如 ADDADD），并在这些情况中应将大小位设置为 1，高/低位用来有选择地互换 32 位操作数的两个 16 位一半。某些指令总是蕴含 16 位操作（如 MUL）并应将大小位设置为 0。而高/低位选择所使用的寄存器的哪一半（假定清除了失去的大小位）。乘累加指令允许独立说明源累加器与目的地寄存器。对于这些指令，大小位用来指示源累加器，而大小位则由指令类型为 0 来蕴含。

当读取 16 位值时（通过 A 或 B 总线）自动进行符号扩展将其扩展成 32 位量。如果读取 48 位寄存器（通过 A 或 B 总线），在总线上只出现底部 32 位.从而在所有情况中都将源 1 与源 2 转换成 32 位值.只有使用总线 C 的累加指令能存取累加器寄存器的整个 48 位.

如果置位再填充位，使用后便将该寄存器标记为空的并将由通常的再填充机制从 ROB 再填充（见关于 ROB 的节）。除非在进行再填充以前该寄存器再一次用作源操作数，Piccolo 不会停止运行。在再填充的数据有效以前的最小周期数（最佳情况－数据等待在 ROB 头部）

为 1 或 2. 因此建议在再填充请求后面的指令上不要使用再填充的数据。如果能避免在后面两条指令上使用操作数，应当这样做，由于这可防止深层流水线实现上的性能损失。

在汇编程序中通过在寄存器号上加上后缀“^”来指定再填充位。标记为空的寄存器段取决于寄存器操作数。可将各寄存器的两半标记为独立地再填充的（例如 X0.l^只标记再填充 X0 的底部一半，X0^则标记再填充整个 X0）。当再填充 48 位寄存器的顶部“一半”（位 47:16）时，将 16 位数据写入位 31: 16 并符号扩展到位 47。

如果试图再填充同一寄存器两次（如 ADD X0, X0^, X0^），只进行一次填充。汇编程序只允许语法 ADD X1, X0, X0^。

如果在再填充一寄存器之前试图读该寄存器，Piccolo 停止运行等待再填充该寄存器。如果标记寄存器为再填充，而在读取再填充的值之前更新了该寄存器，结果是不可预测的（例如 ADD X0, X0^, X1 是不可预测的，由于它标记 X0 再填充，然后通过将 X0 与 X1 之和放置在其中来再填充）。

4 位标度字段编码 14 种标度类型:

- ASR #0, 1, 2, 3, 4, 6, 8, 10
- ASR #12 至 16
- LSL #1

并行最大/最小指令不提供标度，因此不使用源 2 的 6 位常量变型（汇编程序设置为 0）。

在重复指令内支持寄存器重新映射，允许重复指令访问寄存器的移动‘窗口’而不回绕循环。下面更详细描述这一点。

目的地操作数具有以下 7 位格式:

25	24	23	22	21	20	19
F	SD	HL	DEST			

这一基本编码有 10 种变型：

汇编程序助记符		25	24	23	22	21	20	19
Dx	1	0	1	0	Dx			
Dx^	2	1	1	0	Dx			
Dx.l	3	0	0	0	Dx			
Dx.l^	4	1	0	0	Dx			
Dx.h	5	0	0	1	Dx _			
Dx.h^	6	1	0	1	Dx			
未定义		0	1	1	0000			
.l（无寄存器写回 16 位）	7	1	1	1	0	0	00	
""（无寄存器写回 32 位）	8	1	1	1	0	1	00	
.l^ (16-位)输出	9	1	1	1	1	0	00	
^(32-位)输出	10	1	1	1	1	1	00	

寄存器号（DX）指示正在寻址的是16个寄存器中哪一个。高/低位与大小位一起工作来寻址作为一对16位寄存器的各32位寄存器。大小位定义如何设置指令类型中所定义的适当标志，不论将结果是否写入寄存器组与/或输出FIFO，这允许构成比较及类似指令。带累加的加法类指令必须将结果写回寄存器。

下表示出各编码的表现：

编码	寄存器写	FIFO写	V标志
1	写整个寄存器	不写	32位溢出
2	写整个寄存器	写32位	32位溢出
3	写低16位到Dx.l	不写	16位溢出
4	写低16位到Dx.l	写低16位	16位溢出
5	写低16位到Dx.h	不写	16位溢出
6	写低16位到Dx.h	写低16位	16位溢出
7	不写	不写	16位溢出
8	不写	不写	32位溢出
9	不写	写低16位	16位溢出
10	不写	写32位	32位溢出

在所有情况中，任何操作写回寄存器或插入输出FIFO之前的结果为48位量。存在着两种情况：

如果写是16位的，通过选择底部16位[15: 0]将48位量减少到16位量。如果指令饱和，则值将饱和在范围－2^15至2^15－1中。然后将16位值写回到指定的寄存器，如果设置了写FIFO位，则写到输出FIFO。如果将其写到输出FIFO，则将其保持到直到写入下一个16位值将这两个值配对并作为单一的32位值放入输出FIFO中时。

对于32位写，通过选择底部32位[31: 0]将48位量减少到32位量。

对于32位与48位写两者，如果指令饱和，便将48位值转换成范围－2^31－1至2^31中的32位值。接着该饱和：

－如果执行写回到累加器，则写入整个48位。

－如果执行写回到32位寄存器，则写位[31: 0]。

－如果指示写回到FIFO，又一次写位[31:0].

目的地大小是由汇编程序在寄存器号后面用.l或.h指定的。如

果不执行寄存器写回，则寄存器是不重要的，因此省略目的地寄存器来指示不写到寄存器或使用^来指示只写入输出 FIFO。例如，SUB，X0，Y0 等效于 CMP X0，Y0 而 ADD^，X0，Y0 将 X0 + Y0 之值放入输出 FIFO 中。

如果输出 FIFO 没有值的空间，Piccolo 停止运行等待空间成为可利用的。

如果写出 16 位值，例如 ADD X0.h^，X1，X2，则锁存该值直到写第二个 16 位值。然后将两个值组合并作为一个 32 位数放入输出 FIFO 中。写入的第一个 16 位值总是出现在 32 位字的低位一半。将进入输出 FIFO 的数据标记为 16 或 32 位数据，以允许在大结尾系统上校正结尾。

如果在两次 16 位写之间写入 32 位值，则操作是未定义的。

重复指令内支持寄存器重新映射，允许重复指令访问寄存器的移动‘窗口’而不回绕循环。下面更详细地描述这一点。

在本发明的较佳实施例中，重复指令提供修改在循环中指定寄存器操作数的方式的机制。在这一机制下，要访问的寄存器是用指令中的寄存器操作数及在寄存器组中的编移量的一个函数来确定的。该偏移量是以可编程方式改变的，最好在各指令循环的末尾。该机制可独立地在位于 X、Y 与 Z 组中的寄存器上操作。在较佳实施例中，这一设施对于 A 组中的寄存器不能利用。

可使用逻辑与物理寄存器的概念。指令操作数为逻辑寄存器引用，然后将其映射到标识特定 Piccolo 寄存器 10 的物理寄存器引用。包含再填充在内的所有操作都在物理寄存器上操作。只在 Piccolo 指令流一侧出现寄存器重新映射 - 加载进 Piccolo 的数据总是指派给物理寄存器而不执行重新映射。

进一步参照图 5 讨论重新映射机制，图 5 为示出 Piccolo 协处理器 4 的若干内部部件的方框图。将 ARM 核 2 从存储器中检索到的数据项放在重定序缓冲器 12 中，而 Piccolo 寄存器 10 则以较早参照图 2 描述的方式从重定序缓冲器 12 再填充。将存储在高速缓冲存储器 6 中的 Piccolo 指令传递给在 Piccolo 4 内的指令解码器 50，在那里在将它们传递给 Piccolo 处理器核 54 之前进行解码。Piccolo 处理器核 54 包含较早参照图 3 讨论的乘法器/加法器电路 20、累加/累减

电路 22 及定标/饱和电路 24。

如果指令解码器 50 正在处理构成用重复指令标识的指令循环的一部分的指令，而该重复指令指示了应进行若干寄存器的重新映射，便利用寄存器重新映射逻辑 52 来执行必要的重新映射。可将寄存器重新映射逻辑 52 认为是指令解码器 50 的一部分，虽然熟悉本技术的人员清楚可将寄存器重新映射逻辑设置成完全与指令解码器 50 分开的实体。

指令中通常包含标识包含指令所需的数据项的寄存器的一个或多个操作数。例如，典型的指令可包含两个源操作数及一个目的地操作数，标识包含该指令所需的数据项的两个寄存器及应将指令的结果放入其中的寄存器。寄存器重新映射逻辑 52 从指令解码器 50 接收指令的操作数，这些操作数标识逻辑寄存器引用。根据逻辑寄存器引用，寄存器重新映射逻辑确定应不应施加重新映射，然后根据需要将重新映射作用在物理寄存器引用上。如果确定不应施加重新映射，便提供逻辑寄存器引用作为物理寄存器引用。稍后将详细讨论执行重新映射的较佳方式。

将来自寄存器重新映射逻辑的各输出物理寄存器引用传递给 Piccolo 处理器核 54，使得随后处理器核能将指令作用在由物理寄存器引用标识的特定寄存器 10 中的数据项上。

较佳实施例的重新映射机制允许将各寄存器组分成两部分，即可以重新映射的寄存器部分及保持它们原来的寄存器引用不重新映射的寄存器部分。较佳实施例中，重新映射部分起始于重新映射的寄存器组的底部。

重新映射机制采用若干参数，这些参数将参照图 6 详细讨论，图 6 为示出寄存器重新映射逻辑 22 如何使用各种参数的方框图。应指出这些参数是相对于正在重新映射的组内的一点的给定值，这一点为例如该组的底部。

可认为寄存器重新映射逻辑 52 包括两个主要逻辑块，即重新映射块 56 及基更新块 58。寄存器重新映射逻辑 52 采用提供加在逻辑寄存器引用上的偏移值的基指针，由基更新块 58 将这一基指针值提供给重新映射块 56。

可用基起始（BASESTART）信号来定义基指针的初始值，例如这

通常是零，虽然一些其它值也可指定。将这一基起始信号传递给基更新块 58 内的多路复用器 60。在指令循环的第一次重复中，多路复用器 60 将基起始信号传递给存储单元 66，而对于循环的后面的重复，由多路复用器 60 将下一基指针值提供给存储单元 66。

将存储单元 66 的输出作为当前基指针值传递给重新映射逻辑 56，并且还传递给基更新逻辑 58 内的加法器 62 的输入之一。加法器 62 还接收提供基增量值的基增量（BASEINC）信号。加法器 62 配置成将存储单元 66 所提供的当前基指针值增加该基增量值，并将结果传递给模电路 64。

这一模电路还接收基环绕（BASEWRAP）值并将这一值与来自加法器 62 的输出基指针信号比较。如果增量后的基指针值等于或大于基环绕值，便将新基指针绕回到新的偏移值。这时模电路 64 的输出便是要存储在存储单元 66 中的下一基指针值。将这一输出提供给多路复用器 60，并从那里到存储单元 66。

然而，在存储单元 66 从管理重复指令的循环硬件接收到基更新（BASEUPDATE）信号之前不能将这一下一个基指针值存储在存储单元 66 中。循环硬件周期性地生成基更新信号，例如每当要重复指令循环时。当存储单元 66 接收到基更新信号时，存储单元便用多路复用器 60 提供的下一基指针值改写前一基指针值。以这一方式，提供给重新映射逻辑 58 的基指针值将改变成新基指针值。

要在寄存器组的重新映射的部分内存取的物理寄存器由包含在指令的操作数内的逻辑寄存器引用与基更新逻辑 58 提供的基指针值之和确定。这一加法是由加法器 68 执行的并将输出传递给模电路 70。在较佳实施例中，模电路 70 还接收寄存器环绕值，如果来自加法器 68 的输出信号（逻辑寄存器引用与基指针值之和）超过寄存器环绕值，结果将环绕回到重新映射区的底部。然后将模电路 70 的输出提供给多路复用器 72。

将寄存器计数（REGCOUNT）值提供给重新映射块 56 内的逻辑 74，标识组中要重新映射的寄存器的数目。逻辑 74 将这一寄存器计数值与逻辑寄存器引用比较，并根据比较结果将控制信号传递给多路复用器 72。多路复用器 72 作为其两个输入接收逻辑寄存器引用及模电路 70 的输出（重新映射的寄存器引用）。本发明的较佳实施例中，如果

逻辑寄存器引用小于寄存器计数值，逻辑 74 便指令多路复用器 72 输出重新映射的寄存器引用作为物理寄存器引用．然而，如果逻辑寄存器引用大于或等于寄存器计数值，逻辑 74 便指令多路复用器直接输出逻辑寄存器引用作为物理寄存器引用。

如上所述，在较佳实施例中，重复指令调用重新映射机制．如稍后要详细讨论的，重复指令在硬件中提供四个零周期循环。这些硬件循环作为指令解码器 50 的一部分示出在图 5 中．每一次指令解码器 50 请求来自高速缓冲存储器 6 的指令时，高速缓冲存储器便将该指令返回给指令解码器，此时指令解码器判定返回的指令是否是重复指令。如果是，便配置硬件循环之一来处理该重复指令。

各重复指令指定循环中的指令数及环绕循环的次数（它是常量或读自 Piccolo 寄存器）。提供了两个操作码‘重复’（REPEAT）及下一个（NEXT）来定义硬件循环，‘下一个’操作码只用作分界符并不汇编成指令。重复从循环的起点开始，而‘下一个’界定循环的结束，允许汇编程序计算循环体中的指令数。在较佳实施例中，重复指令可包含要由寄存器重新映射逻辑 52 使用的诸如寄存器计数（REGCOUNT）、基增量（BASEINC）、基环绕（BASEWRAP）及寄存器环绕（REGWRAP）参数等重新映射参数。

可设置若干寄存器来存储寄存器重新映射逻辑所使用的重新映射参数。在这些寄存器内，可提供若干组预定义的重新映射参数，同时保留一些寄存器供存储用户定义的重新映射参数。如果用重复指令指定的重新映射参数等于预定义的重新映射参数组之一，则采用适当的重复编码，这一编码导致多路复用器之类将适当的重新映射参数直接从寄存器提供给寄存器重新映射逻辑。反之，如果重新映射参数与任何预定义的重新映射参数组都不同，则汇编程序生成重新映射参数传送指令（RMOV），它允许配置用户定义的寄存器重新映射参数，RMOV 指令后面是重复指令．最好 RMOV 指令将用户定义的重新映射指令放置在为存储这种用户定义的重新映射参数留出的寄存器中，然后将多路复用器编程为将这些寄存器的内容传递给寄存器重新映射逻辑。

在较佳实施例中，寄存器计数、基增量、基环绕及寄存器环绕参数取下表中确定的值之一：

参数	描述
REGCOUNT （寄存器计数）	它确定在上面执行重新映射的16位寄存器数并可取值0，2，4，8．REGCOUNT以下的寄存器是重新映射的，以上或等于REGCOUNT的是直接存取的。
BASEINC （基增量）	这定义在各循环重复结束时将基指针增量多少个16位寄存器。在较佳实施例中它可取值1，2或4，虽然如果需要事实上它可取其它值，适当时可包含负值。
BASEWRAP （基环绕）	它确定基计算的上限．基环绕模可取值2，4，8．
REGWRAP （寄存器环绕）	它确定重新映射计算的上限．寄存器环绕模可取值2，4，8．REGWRAP可选择为等于REGCOUNT

参见图6，重新映射块56如何使用各种参数的示例如下（在本例中，逻辑与物理寄存器值是相对于特定组的）：

```
if(逻辑寄存器<REGCOUNT)
物理寄存器 = （逻辑寄存器 + 基）MOD REGCOUNT
else
物理寄存器 = 逻辑寄存器
end if
```

在循环结束处，在循环的下一次重复开始前，基更新逻辑58执行对基指针的下述更新:

```
基=(基+BASEINC)MOD BASEWRAP
```

在重新映射循环结束处，关闭寄存器重新映射，然后作为物理寄存器存取所有寄存器。较佳实施例中，任何一个时间上只有一个重新映射REPEAT（重复）是活跃的。循环还可嵌套，但在任何特定时刻只有一个循环能更新重新映射变量 。然而如果需要，可以嵌套重新映射重复。

为了展示作为采用按照本发明的较佳实施例的重新映射机制的结果所达到的关于代码密度的好处，下面讨论典型的块过滤算法．首

先参照图 7 讨论阻塞过滤器算法的原理. 如图 7 中所示，将累加器寄存器 A0 配置成累加若干次乘法运算的结果，乘法运算为系数 C0 乘以数据项 d0 的乘法，系数 c1 乘以数据项 d1 的乘法，系数 c2 乘以数据项 d2 的乘法等. 寄存器 A1 累加类似的乘法运算组的结果，但这时系数集合已移位使得 c0 现在乘以 d1，c1 乘以 d2，c2 乘以 d3 等. 类似地，寄存器 A2 累加数据值乘以又向右移位一步的系数值的结果，使得 c0 乘以 d2，c1 乘以 d3，c2 乘以 d4 等. 然后重复这一移位、乘及累加进程，将结果放在寄存器 A3 中.

如果不采用按照本发明的较佳实施例的寄存器重新映射，则需要下面的指令循环来执行块过滤指令:

```
;  以 4 个新数据值开始

ZERO {A0-A3}                          ;  清零累加器



REPEAT  Z1                            ;  Z1=(系数个数/4)

;在第一轮进行下面四个系数

; a0 += d0*c0+d1*c1+d2*c2+d3*c3

; a1 += d1*c0+d2*c1+d3*c2+d4*c3

; a2 += d2*c0+d3*c1+d4*c2+d5*c3

; a3 += d3*c0+d4*c1+d5*c2+d6*c3

MULA    A0, X0.l^, Y0.l , A0     ; a0 += d0*c0, 及加载  d4

MULA    A1, X0.h , Y0.l , A1     ; a1 += d1*c0
```

```
MULA    A2, X1.l , Y0.l , A2     ; a2 += d2*c0

MULA    A3, X1.h , Y0.l^, A3     ; a3 += d3*c0, 及加载 c4

MULA    A0, X0.h^, Y0.h , A0     ; a0 += d1*c1, 及加载 d5

MULA    A1, X1.l , Y0.h , A1     ; a1 += d2*c1

MULA    A2, X1.h , Y0.h , A2     ; a2 += d3*c1

MULA    A3, X0.l , Y0.h^, A3     ; a3 += d4*c1, 及加载 c5

MULA    A0, X1.l^, Y1.l , A0     ; a0 += d2*c2, 及加载 d6

MULA    A1, X1.h , Y1.l , A1     ; a1 += d3*c2

MULA    A2, X0.l , Y1.l , A2     ; a2 += d4*c2

MULA    A3, X0.h , Y1.l^, A3     ; a3 += d5*c2, 及加载 c6

MULA    A0, X1.h^, Y1.h , A0     ; a0 += d3*c3, 及加载 d7

MULA    A1, X0.l , Y1.h , A1     ; a1 += d4*c3

MULA    A2, X0.h , Y1.h , A2     ; a2 += d5*c3

MULA    A3, X1.l , Y1.h^, A3     ; a3 += d6*c3, 及加载 c7

NEXT
```

在本例中，将数据值放在 X 寄存器组中而将系数值放在 Y 寄存器组中。作为第一步，将四个累加器寄存器 A0、A1、A2 与 A3 设置为零。一旦复位了累加器寄存器，便进入指令循环，该循环是用‘重复’（REPEAT）及‘下一个’（NEXT）指令定界的。值 Z1 确定指令循环应重复的次数，为了下面将要讨论的原因，它实际上等于系数（c0，c1，c2 等）的个数除以 4。

指令循环包括 16 条乘累加指令（MULA），在第一次通过循环之后这些提令将导致在寄存器 A0，A1，A2，A3 中包含上述重复与第一条 MULA 指令之间的代码中所示的计算结果。为了说明乘累加指令如何操作，我们将考虑前四条 MULA 指令。第一条指令将 X 组寄存器 0 的第一或低 16 位的数据值乘以 Y 组寄存器 0 中的低 16 位，并将结果加到累加器寄存器 A0 中。同时用再填充位标记 X 组寄存器 0 的低 16 位，这指示该寄存器的该部分现在可用新数据值再填充。以这一方式标记是因为从图 7 中可看出，一旦将数据项 d0 乘以系数 c0（由第一条 MULA 指令表示），对于其余块过滤指令 d0 便不再需要，因此能用新数据值取代。

然后第二条 MULA 指令将 X 组寄存器 0 的第二或高 16 位乘以 Y 组寄存器 0 的低 16 位（这表示图 7 中所示的乘法 d1×c0）。类似地，第三与第四条 MULA 指令分别表示乘法 d2×c0 及 d3×c0。从图 7 中可见，一旦执行过这四个计算，系数 c0 便不再需要，因此用再填充位标记寄存器 Y0.l 使它能用另一系数（c4）改写。

下面四条 MULA 指令分别表示计算 d1×c1、d2×c1、d3×c1 与 d4×c1。一旦执行过 d1×c1，便用再填充位标记寄存器 x0.h，因为不再需要 d1。类似地，一旦执行过全部四条指令，便将寄存器 Y0.h 标记为供再填充，因为不再需要系数 c1。类似地，下面四条 MULA 指令对应于计算 d2×c2、d3×c2、d4×c2 及 d5×c2，而最后四条指令则对应于计算 d3×c3、d4×c3、d5×c3 及 d6×c3。

在上述实施例中，由于寄存器是不能重新映射的，各乘法运算必须用操作数中指定的所需特定寄存器明显地再生。一旦执行过 16 条 MULA 指令，便能为系数 c4 至 c7 及数据项 d4 至 d10 重复这一指令循环。并且由于每一次重复该循环在四个系数值上操作。所以系数值的数目必须是 4 的倍数并必须计算 Z1 = 系数数/4。

通过采用按照本发明的较佳实施例的重新映射机制，可以极大地缩小指令循环，使得它只包含 4 条乘累加指令而不是否则所需要的 16 条乘累加指令。采用重新映射机制，将代码编写成如下所列：

```
; 以 4 个新数据值开始

ZERO {A0-A3}                          ; 清零累加器


REPEAT  Z1, X++ n4 w4 r4, Y++ n4 w4 r4; Z1= (系数的个数)

; 对 X 与 Y 组进行重新映射

; 重新映射这些组中四个 16 位寄存器

; 在循环的每一次重复上将两组的基指针递增。


; 当基指针到达该组中第四个寄存器时便绕回。


MULA    A0, X0.l^, Y0.l , A0    ; a0 += d0*c0, 及加载 d4

MULA    A1, X0.h,  Y0.l , A1    ; a1 += d1*c0

MULA    A2, X1.l,  Y0.l , A2    ; a2 += d2*c0

MULA    A3, X1.h,  Y0.l^, A3    ; a3 += d3*c0, 及加载 c4

NEXT                            ; 绕回到循环并进行重新映射
```

如上所述，第一步将四个累加器寄存器 A0－A3 设置成 0．然后进入用‘重复’与‘下一个’操作码定界的指令循环．重复指令拥有与之关联的若干参数，它们是:

X++: 指示对于 X 寄存器组基增量为“1”．

n4:指示寄存器计数为“4”，因此要重新映射前四个 X 组寄存器 X0.l 至 X1.h

w4: 指示对于 X 寄存器组基环绕为“4”

r4: 指示对于 X 寄存器组寄存器环绕为“4”

Y++: 指示对于 Y 寄存器组基增量为“1”

n4: 指示寄存器计数为“4”因此要重新映射前 4 个 Y 组寄存器 Y0.l 至 Y1.h。

w4: 指示对于 Y 寄存器组基环绕为“4”

r4:指示对于 Y 寄存器组寄存器环绕为“4”

还应指出，现在值 Z1 等于系数数目而不是先有技术示例中等于系数数目/4。

对于指令循环的第一次循环，基指针值为 0，因此无重新映射。然而下一次执行循环时，对于 X 与 Y 组基指针值都将是“1”，因此将操作数重新映射如下:

X0.l 成为 X0.h

X0.h 成为 X1.l

X1.l 成为 X1.h

X1.h 成为 X0.l（由于基环绕为“4”）

Y0.l 成为 Y0.h

Y0.h 成为 Y1.l

Y1.l 成为 Y1.h

Y1.h 成为 Y0.l（由于基环绕为“4”）

因此，在第二次重复时可看出，四条 MULA 指令实际上执行较早讨论的不包含本发明的重新映射的示例中用第五至第八条 MULA 指令所指示的计算。类似地，第三与第四次重复通过循环执行前面用先有技术代码的第九至第 12 及第 13 至第 16 条 MULA 指令执行的计算。

因此可以看出上述代码执行与先有技术代码完全相同的块过滤算法，但将循环体内的代码密度改进了一个因子 4，由于只需要提供 4

条指令而不是先有技术所需的 16 条。

通过采用按照本发明的较佳实施例的寄存器重新映射技术，能实现下述优点：

1. 改进代码密度；
2. 在一定场合中隐藏从标记寄存器为空到 Piccolo 的重定序缓冲器再填充该寄存器的等待时间。这可以以增加代码大小的代价通过解开循环来达到。
3. 能存取可变数目的寄存器－通过改变执行的循环重复次数，可改变存取的寄存器数目；以及
4. 便于算法展开。对于适当的算法，程序员可为算法的第 n 阶段生成一段代码，然后利用寄存器重新映射将公式应用在一个滑动数据组上。

很明显可以不脱离本发明的范围对上述寄存器重新映射机制作出某些改变。例如，有可能为寄存器组 10 提供比程序员在指令操作数中所能指定的更多的物理寄存器。这些额外的寄存器不能直接存取，而寄存器重新映射机制能利用这些寄存器。例如，考虑早先讨论的 X 寄存器组具有程序员可利用的 4 个 32 位寄存器并因而可用逻辑寄存器引用指定 8 个 16 位寄存器的示例。有可能使 X 寄存器组实际上包含例如 6 个 32 位寄存器，在这一情况中将有 4 个附加的 16 位寄存器不能由程序员直接存取。然而，这四个额外的寄存器能被重新映射机制利用，借此为存储数据项提供附加的寄存器。

可使用以下的汇编程序语法：

>>表示逻辑右移，或者在移位操作数为负时左移（见下面<lscale>）。

－>>表示算术右移，或者在移位操作数为负时左移（见下面<scale>）。

ROR 表示循环右移

SAT（a）表示 a 的饱和值（取决于目的地寄存器的大小饱和到 16 或 32 位）。具体地，为了饱和到 16 位，任何大于 + 0x7fff 的值用 +0x7fff 代替，而任何小于 - 0x8000 的值则用 - 0x8000 代替。类似地饱和到 32 位用极限 + 0x7fffffff 与 - 0x80000000. 如果目的地寄存器为 48 位，饱和仍然在 32 位上。

源操作数 1 可用下述格式之一:

<Src1>将用作[Rn|Rn.1|Rn.h|Rn.x][^]的简写。换言之，源说明符的所有 7 位都有效，并作为（可选择地互换的）32 位值或 16 位符号扩展的值读取寄存器。对于累加器只读取底部 32 位。^指示寄存器再填充。

<src1_16>是[Rn.1|Rn.h][^]的简写。只能读取 16 位值。

<src1_32>是[Rn|Rn.X][^]的简写。只能读取 32 位值，高与低一半有选择地互换。

<src_2>(源操作数 2)可以是下述格式之一:

<src2>是三种选项的简写

- 形式[Rn|Rn.1|Rn.h|Rn.x][^]的源寄存器，加上最终结果的标度（<scale>）。

- 可选择的移位的 8 位常量（<immed_8>），但无最终结果的标度。

- 6 位常量（<immed_6>）加上最终结果的标度（<scale>）。

<src2_maxmin> 与<src2>相同但不允许定标。

<src2_shift> 提供<src2>的有限子集的移位指令。见上述详细情况。

<src2_par> 在<src2_shift>方面

对于指定第三操作数的指令:

<acc> 四个累加器寄存器[A0|A1|A2|A3]中任何一个的简写。读取全部 48 位。不能指定再填充。

目的地寄存器具有格式:

<dest> 它是[Rn|Rn.1|Rn.h|.1|][^]的简写。不带“.”扩展写入整个寄存器（在累加器情况中为 48 位）。在不需要写回到寄存器的情况中，所使用的寄存器是不重要的。汇编程序支持省略目的地寄存器来指示不需要写回，或用“.1”来指示不需要写回，但应设置标志，犹如结果为 16 位量。^表示将值写到输出 FIFO 中。

<scale> 表示若干算术标度。可以利用的有 14 种标度:

ASR #0, 1, 2, 3, 4, 6, 8, 10

ASR #12 至 16

LSL #1

<immed-8> 代表不带符号的 8 位立即值. 这包含循环左移 0, 8, 16 或 24 的一个字节. 因此能为任何 YZ 编码值 0xYZ000000、0x00YZ0000、0x0000YZ00、及 0x000000YZ. 循环是作为 2 位的量编码的.

<imm_6> 代表不带符号的 6 位立即数.

<PARAMS> 用来指定寄存器重新映射并具有下述格式:

<BANK><BASEINC>n<RENUMBER>w<BASEWRAP>

<BANK>	可以是	[X|Y|Z]
<BASEINC>	可以是	[++|+1|+2|+4]
<RENUMBER>	可以是	[0|2|4|8]
<BASEWRAP>	可以是	[2|4|8]

表达式<cond>为下述状态码中任何一种. 注意编码与 ARM 稍有不同, 因为不带符号的 LS 与 HI 码已被更有用的带符号的上溢/下溢测试所替代. Piccolo 上的 V 与 N 标志的设置与 ARM 的不同, 因此从状态测试到标志检验的翻译与 ARM 也不同.

0000	EQ	Z=0	上一次结果为 0.
0001	NE	Z=1	上一次结果非 0.
0010	CS	C=1	在移位/最大操作后使用.
0011	CC	C=0	
0100	MI/LT	N=1	上一次结果为负
0101	PL/GE	N=0	上一次结果为正
0110	VS	V=1	上一次结果带符号溢出/饱和
0111	VC	V=0	上一次结果无溢出/饱和
1000	VP	V=1 & N=0	上一次结果正溢出
1001	VN	V=1 & N=1	上一次结果负溢出
1010	保留		
1011	保留		
1100	GT	N=0 & Z=0	
1101	LE	N=1 | Z=1	
1110	AL		
1111	保留		

由于Piccolo处理带符号的量，弃掉不带符号的LS与HI状态而用描述任何溢出的方向的VP与VN来代替。由于ALU的结果为48位宽，MI与LT现在执行相同功能，类似地PL与GE。这留下3个空槽供未来扩展。

除非另有说明，所有运算都是带符号的。

一级与二级状态码各包含:

N－负。

Z－零。

C－进位/不带符号溢出。

V－带符号溢出。

算术指令可分成两类：并行与“全宽度”。“全宽度”指令只设置一级标志，而并行运算符根据结果的高与低16位一半设置一级与二级标志。

在已施加定标但写到目的地之前，N、Z与V标志是根据整个ALU结果计算的。ASR将总是减少存储结果所需位数，而ASL则增加位数。为了防止在施加ASL定标时Piccolo截尾48位结果，将位数限制在必须进行零检测与溢出上。

N标志是假设正在进行带符号算术运算时计算的。这是因为在发生溢出时，结果的最高位是C标志或N标志之一，这取决于输入操作数是带符号还是不带符号的。

V标志指示作为将结果写到选择的目的地的结果是否出现任何精度损失。如果选择了不写回，仍然蕴含‘大小’，并正确地设置溢出标志。在下述情况中出现溢出:

－当结果不在范围－2^15至2^15－1中时写入16位寄存器。

－当结果不在范围－2^31至2^31－1中时写入32位寄存器。

并行加/减指令在结果的高与低一半上独立地设置N、Z与V标志。

当写入累加器时和写入32位寄存器一样设置V标志。这是允许饱和指令使用累加器作为32位寄存器。

饱和绝对值指令（SABS）在输入操作数的绝对值不符合指定的目的地时也设置溢出标志。

进位标志由加与减指令设置并由MAX/MIN、SABS及CLB指令用作

'二进制'标志。包含乘法运算在内的所有其它指令保留进位标志。

对于加与减运算，根据目的地是32还是16位宽，进位便是由位31或位15或结果生成的。

根据如何设置标志，可将标准算术指令分成若干类型:

在加与减指令的情况中，如果N位是置位的则保持所有标志。如果N位不置位则将标志更新如下:

如果全48位结果为0便置位Z。

如果全48位结果中位47置位（是负的）则置位N。

如果下述条件之一成立则置位V:

目的地寄存器为16位而带符号的结果放不进16位寄存器中（不在范围 - 2^15<= x<2^15 内）。

目的地寄存器为 32/48 位寄存器而带符号的结果放不进 32 位中。

如果在求和<src1>与<src2>时从位31有进位或者从<src1>减去<src2>时位31不出现借位，则如果<dest>为32或48位寄存器时便置位C标志（与ARM上的所期望的相同进位值）。如果<dest>为16位寄存器，则如果和的位31进位便置位C标志。

保留二级标志（SZ、SN、SV、SC）。

在从48位寄存器执行乘法或累加指令的情况中。

如果全48位结果为0便置位Z。

如果全48位结果中位47置位（是负的），则置位N。

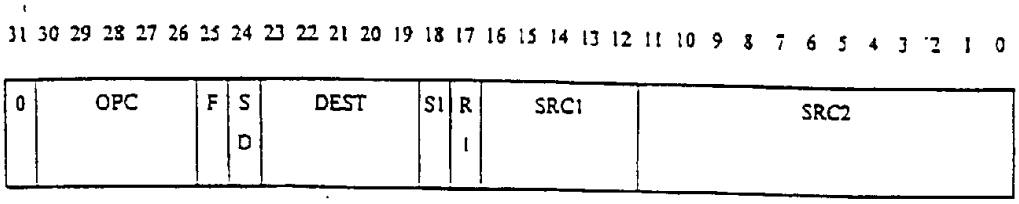
如果（1）目的地寄存器为16位而带符号的结果放不进16位寄存器（不在范围 - 2^15<= x<2^15 内）或（2）目的地寄存器为32/48位寄存器而带符号的结果放不进32位中，便置位V。

保留C。

保留二级标志（SZ、SN、SV、SC）。

下面讨论包含逻辑运算、并行加与减、最大与最小、移位等在内的其它指令。

加与减指令将两个寄存器相加或相减，定标该结果，然后存储回到一个寄存器。将操作数作为带符号的值对待。对于非饱和变型，标志更新是供选用的，并可通过在指令尾部附加一个 N 来抑制标志更新。



OPC 指定指令的类型

操作（OPC）：

100N0	dest = (src1 + src2) (->> scale) (, N)
110N0	dest = (src1 - src2) (->> scale) (, N)
10001	dest = SAT((src1 + src2) (->> scale))
11001	dest = SAT((src1 - src2) (->> scale))
01110	dest = (src2 - src1) (->> scale)
01111	dest = SAT((src2 - src1) (->> scale))
101N0	dest = (src1 + src2 + Carry) (->> scale) (, N)
111N0	dest = (src1 - src2 + Carry - 1) (->> scale) (, N)

助记符:

100N0	ADD{N}	<dest>, <src1>, <src2> {,<scale>}
110N0	SUB{N}	<dest>, <src1>, <src2> {,<scale>}
10001	SADD	<dest>, <src1>, <src2> {,<scale>}
11001	SSUB	<dest>, <src1>, <src2> {,<scale>}
01110	RSB	<dest>, <src1>, <src2> {,<scale>}
01111	SRSB	<dest>, <src1>, <src2> {,<scale>}
101N0	ADC{N}	<dest>, <src1>, <src2> {,<scale>}
111N0	SBC{N}	<dest>, <src1>, <src2> {,<scale>}

汇编程序支持下述操作码

CMP <src1>,<src2>,

CMN <src1>,<src2>,

CMP 为减法，它设置标志并禁止寄存器写。CMN 为加法，它设置标志并禁止寄存器写。

标志：上面已讨论过。

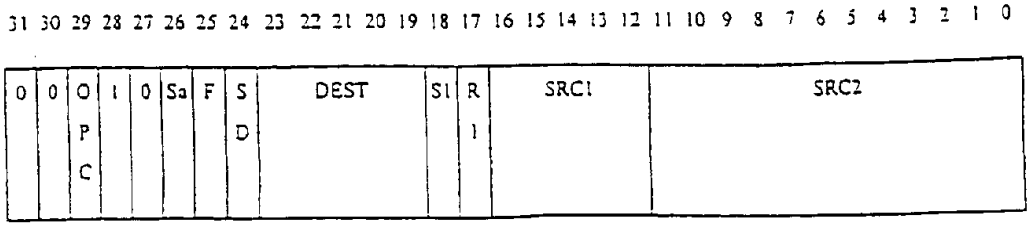
包含的理由：

在移位/最大/最小操作之后将进位插入寄存器底部 ADC 是有用的。它也用来进行 32/32 位除法。它也提供扩充精度加法。N 位加法给出更精密的标志控制，特别是进位。这使得 32/32 位除法能在每位 2 个周期上进行。

G.729 等需要饱和的加与减。

增量/减量计数器。RSB 对于计算移位是有用的（x = 32 - x 是常用运算）。对于饱和的求反（用在 G.729 中）需要饱和的 RSB.

加/减累计指令执行带累计与定标/饱和的加法与减法。与乘累加指令不同，不能独立于目的地寄存器指定累加器号。目的地寄存器的底部两位给出要累计到其中的 48 位累加器号 acc。因此 ADDA X0, X1, X2, A0 与 ADDA A3, X1, X2, A3 是有效的，而 ADDA X1, X1, X2, A0 则无效。对于这类指令，必须将结果写回寄存器-不允许目的地字段的不写回编码。



OPC 指定指令的类型。下面的 acc 为（DEST[1:0]）。Sa 位指示饱和。

操作（OPC）：

0 dest = {SAT}(acc + (src1 + src2)) {->> scale}

1 dest = {SAT}(acc + (src1 - src2)) {->> scale}

助记符

0 {S}ADDA <dest>, <src1>, <src2>, <acc> {,<scale>}

1 {S}SUBA <dest>, <src1>, <src2>, <acc> {,<scale>}

命令前面的 S 表示饱和。

标志：见上面。

包含的理由：

ADDA（加累计）指令对于用累加器每一周期求和整数数组的两个字是有用的（例如找出它们的平均值）。SUBA（减累计）指令在计算

差之和（用于相关）中是有用的；它将两个独立的值相减并将差加到第三寄存器中。

带四舍五入的加法可用与<acc>不同的<dest>进行。例如，X0=（X1+X2+16384）>>15 可通过将 16384 保持在 A0 中而在一个周期中完成。带四舍五入的常量的加法可用 ADDA X0, X1, #16384, A0 来完成。

对于（(a_i*b_j)>>k）之和（在 TrueSpeech 中相当常用）的位精确实现：

标准 Piccolo 代码为：

```
MUL   t1, a_0, b_0, ASR#K
ADD   ans, ans, t1
MUL   t2, a_1, b_1, ASR#k
ADD   ans, ans, t2
```

这一代码有两个问题：它太长以及不是加到 48 位精度，因此不能用保护位。较好的解决方法为使用 ADDA：

```
MUL    t1, a_0, b_0, ASR#k
MUL    t2, a_1, b_1, ASR#k
ADDA   ans, t1, t2, ans
```

这提高 25% 速度并保持 48 位精度。

并行加/减指令在成对保持在 32 位寄存器中的两个带符号的 16 位量上执行加法与减法。一级状态码标志从高 16 位的结果设置，而二级标志则从低位一半更新。只能指定 32 位寄存器作为这些指令的源，虽然这些值是可以半字互换的。将各寄存器的各个一半作为带符号的值对待。计算与定标是不损失精度完成的。因此 ADD ADD X0, X1, X2, ASR #1 将在 X0 的高位与低位一半中产生正确的平均值。为必须置位 Sa 位的各指令提供了选用的饱和。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	0	OPC	Sa	F	SD	DEST	S1	R1	SRC1	SRC2

OPC 定义操作

操作（OPC）：

000 dest.h = (src1.h + src2.h) ->> {scale},
dest.l = (src1.l + src2.l) ->> {scale}
001 dest.h = (src1.h + src2.h) ->> {scale},
dest.l = (src1.l - src2.l) ->> {scale}
100 dest.h = (src1.h - src2.h) ->> {scale},
dest.l = (src1.l + src2.l) ->> {scale}
101 dest.h = (src1.h - src2.h) ->> {scale},
dest.l = (src1.l - src2.l) ->> {scale}

如果置位了Sa位，各和/差是独立饱和的。

助记符:

000	{S}ADDADD	<dest>, <src1_32>, <src2_32> {,<scale>}
001	{S}ADDSUB	<dest>, <src1_32>, <src2_32> {,<scale>}
100	{S}SUBADD	<dest>, <src1_32>, <src2_32> {,<scale>}
101	{S}SUBSUB	<dest>, <src1_32>, <src2_32> {,<scale>}

命令前的 S 表示饱和。

汇编程序还支持

CMNCMN	<dest>, <src1_32>, <src2_32> {,<scale>}
CMNCMP	<dest>, <src1_32>, <src2_32> {,<scale>}
CMPCMN	<dest>, <src1_32>, <src2_32> {,<scale>}
CMPCMP	<dest>, <src1_32>, <src2_32> {,<scale>}

它们是不带写回的标准指令生成的。

标志:

C 如果在相加两个高 16 位一半时从位 15 进位，便置位。

Z 如果高 16 位一半之和为 0，便置位。

N 如果高 16 位一半之和为负，便置位。

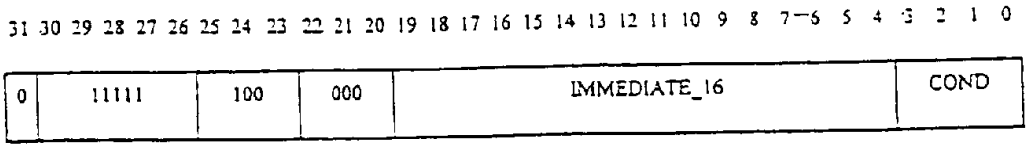
V 如果高 16 位一半的带符号的 17 位和不能装入 16 位中（定标后），便置位。

类似地为低 16 位一半置位 SZ、SN、SV 与 SC。

包含的理由:

并行加与减指令对于在保持在单个 32 位寄存器中的复数上执行运算是有用的。它们用在 FFT（快速傅里叶变换）核心中。它对于 16 位数据的简单矢量加法/减法也是有用的，允许在一个周期中处理两个元素。

转移（条件）指令允许控制流中的条件改变。Piccolo 占用三个周期来执行所取的转移。



操作:

如果根据一级标志<cond>成立，用偏移量转移。

偏移量为带符号的 16 位字数。当前偏移的范围限制在 - 32768 至+32767 个字。

执行的地址计算是

目标地址 = 转移指令地址 + 4 + 偏移量

助记符:

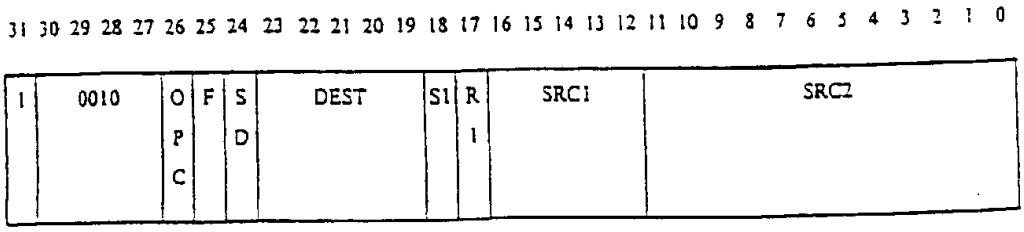
B<cond> <destination_label>

标志: 不受影响。

包含的理由:

在大多数例程中高度有用。

条件加或减指令有条件地将 src1 加在 src2 上或从 src1 中减去 src2。



OPC 指定指令的类型。

操作（OPC）：

如果（进位置位）temp=src1-src2 否则 temp=src1+src2

dest=temp{->>scale}

如果（进位置位）temp=src1-src2 否则 temp=src1+src2

dest=temp{->>scale}但是如果定标是左移位

则将（来自 src1-src2 或 src1+src2 的）进位的新值移位进底部中。

助记符:

0 CAS <dest>,<src1>,<src2>,{,<scale>}

1 CASC <dest>,<src1>,<src2>,{,<scale>}

标志: 见上面:

包含的理由:

条件加或减指令使高效除法代码能构成。

例 1: 将 X0 中的 32 位不带符号值除以 X1 中的 16 位不带符号值（假设 X0<（X1<<16）及 X1.h=0）。

```
LSL   X1, X1, #15          ;  上移除数
SUB   X1, X1, #0           ;  置位进位标志
REPEAT #16
CASC X0, X0, X1, LSL#1
NEXT
```

在循环末尾，X0.1 保持除法的商。取决于进位的值可从 X0.h 恢复余数。

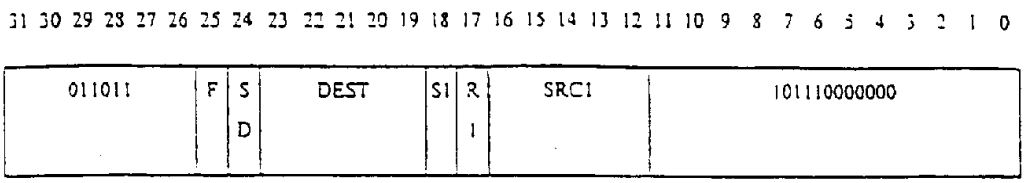
例 2: 将 X0 中的 32 位正值除以 X1 中的 32 位正值，

带早结束。

```
        MOV     X2, #0          ;清除商
        LOG     Z0, X0          ;  X0 可移位的位数
        LOG     Z1, X1          ;  X1 可移位的位数
        SUBS    Z0, Z1, Z0      ;X1 向上移位因此 1 匹配
        BLT     div_end         ; X1>X0 因此答数为 0
        LSL     X1, X1, Z0      ;   匹配前面的 1
        ADD     Z0, Z0, #1      ;进行的测试数
        SUBS    Z0, Z0, #0      ;  置位进位
        REPEAT  Z0
        CAS     X0, X0, X1, LSL#1
        ADCN    X2, X2, X2
        NEXT
div_end
```

在结束处，X2 保持商而余数可从 X0 恢复。

计数前导位指令使数据能正规化。



操作:

将 dest 设定为为了使位 31 与位 30 不同而 src1 中的值必须左移的位数。这是范围 0－30 中的一个值，但除外 src1 为－1 或 0 的特殊情况，这时返回 31。

助记符:

CLB <dest>,<src1>

标志:

Z 如果结果为 0，便置位。

N 是消除的。

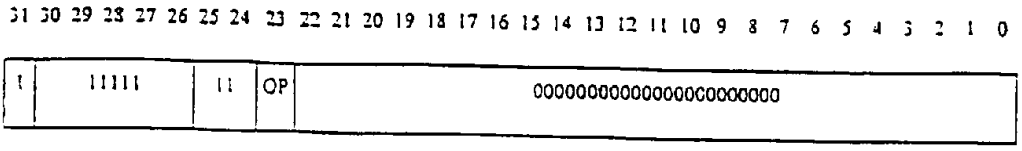
C 如果 src1 为－1 或 0 之一，便置位。

V 保持。

包含的理由:

正规化需要的步骤.

设置了停止与断点指令用于停止 Piccolo 的执行



OPC 指定指令的类型.

操作 (OPC) :

0 Piccolo 执行被停止并在 Piccolo 状态寄存器中置位停止位.

1 Piccolo 执行停止, 并在 Piccolo 状态寄存器中置位中断位, 并中断 ARM 报告已到达断点.

助记符:

0 HALT

1 BREAK

标志: 不受影响.

逻辑运算指令在 32 或 16 位寄存器上执行逻辑运算。将操作数作为不带符号值对待。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	000	OPC	F	SD	DEST	S1	RI	SRC1	SRC2

OPC 编码要执行的逻辑运算

操作（OPC）：

00 dest = (src1 & src2) {->> scale}

01 dest = (src1 ¦ src2) {->> scale}

10 dest = (src1 & –src2) {->> scale}

11 dest = (src1 ^ src2) {->> scale}

助记符:

00 AND <dest>, <src1>, <src2> {,<scale>}

01 ORR <dest>, <src1>, <src2> {,<scale>}

10 BIC <dest>, <src1>, <src2> {,<scale>}

11 EOR <dest>, <src1>, <src2> {,<scale>}

汇编程序支持下述操作码:

TST <src1>, <src2>

TEQ <src1>, <src2>

TST 为禁止寄存器写的“与”. TEQ 为禁止寄存器写的“EOR”.

标志:

Z　　如果结果为全 0, 便置位

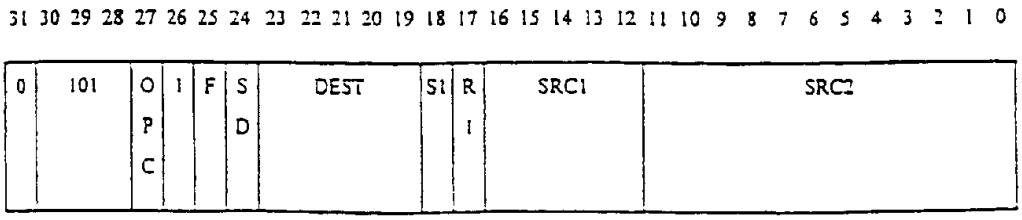
N、C、V　保持

SZ、SN、SC、SV　保持

包含的理由:

话音压缩算法采用组合位字段来编码信息。位屏蔽指令协助抽取/组合这些字段。

Max 与 Min 操作指令执行最大与最小值运算。



OPC 指定指令的类型。

操作（OPC）:

0　　dest = (src1 <= src2) ? src1 : src2

1　　dest = (src1 > src2) ? src1 : src2

助记符:

0　　MIN　<dest>, <src1>, <src2>

1　　MAX　<dest>, <src1>, <src2>

标志:

Z 如果结果为 0, 便置位。

N 如果结果为负, 便置位。

C 对于 Max: 如果 src2>=src1（dest=src1 情况）, 置位 C

　对于 Min: 如果 src2>=src1（dest=src2 情况）, 置位 C

V 保持

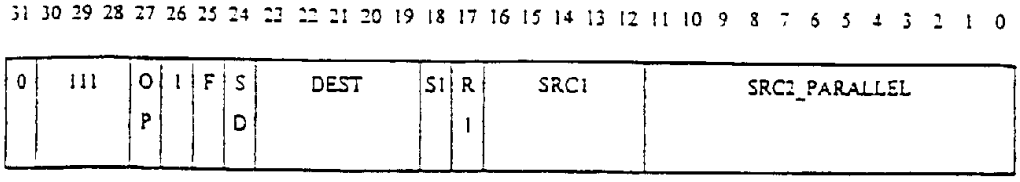
包含的理由:

为了找出信号强度，许多算法扫描样本来找出样本的绝对值的最大/最小值. 对此，MAX 与 MIN 是无价之宝. 取决于要找出信号中第一还是最后的最大值，操作数 src1 与 src2 可以互换.

MAX XO,XO,#0 将 XO 转换成从下面修剪掉的正数.

MIN XO, XO, #255 从上面修剪掉. 这对于图形处理有用.

并行指令中的 Max 与 Min 运算在并行的 16 位数据上执行最大值与最小值运算.



OPC 指定指令的类型.

操作（OPC）：

0 dest.l = (src1.l <= src2.l) ? src1.l : src2.l

dest.h = (src1.h <= src2.h) ? src1.h : src2.h

1 dest.l = (src1.l > src2.l) ? src1.l : src2.l

dest.h = (src1.h > src2.h) ? src1.h : src2.h

助记符:

0 MINMIN <dest>, <src1>, <src2>

1 MAXMAX <dest>, <src1>, <src2>

标志:

Z 如果结果的高 16 位为 0，便置位。

N 如果结果的高 16 位为负，便置位。

C 对于 Max: 如果 src2.h>=src1.h

（dest=src1 情况），置位 C

对于 Min:如果 src2.h=src1.h

（dest=src2 情况），置位 C。

V 保持。

SZ、SN、SC、SV 类似地为低 16 位一半置位。

包含的理由:

关于 32 位 Max 及 Min。

传送长立即数操作指令允许将寄存器设置成任何带符号的 16 位、符号延伸的值。两条这种指令能将 32 位寄存器设置成任何值（通过顺序存取高位与低位一半）。对于寄存器之间的传送见选择操作。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	11100	F	S D	DEST	IMMEDIATE_15	+/-	000

助记符

MOV <dest>,#<imm_16>

汇编程序利用 MOV 指令提供非互锁的 NOP（空操作）操作，即，NOP 等效于 MOV, #0。

标志: 标志不受影响。

包含的理由:

初始化寄存器/计数器。

乘累加运算指令执行带符号乘法与累加或累减（de-accumulation），定标与饱和。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	10	OPC	Sa	F	S D	DEST	A 1	R 1	SRC1	SRC2_MULA

字段 OPC 指定指令的类型。

操作（OPC）：

00 dest = (acc + (src1 * src2)) {->> scale}

01 dest = (acc - (src1 * src2)) {->> scale}

在各情况中，如果置位了 Sa 位，在写到目的地之前将结果饱和。

助记符:

00 {S}MULA <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}

01 {S}MULS <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}

命令前的 S 指示饱和。

标志: 见上节。

包含的理由:

对于 FIR 代码需要单周期持续的 MULA。MULS 用在 FFT 蝶形电路中。对于带四舍五入的乘法 MULA 也是有用的。例如通过将 16384 保持在另一累加器（例如 A1）中可在一个周期中完成 A0 = （X0*X1+16384）>>15。对于 FFT 核心还需要不同的<dest>与<acc>。

乘双倍运算（Multiply Double Operation）指令执行单符号乘法，在累加或累减、定标与饱和之前将结果加倍。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	10	1	OPC	1	F	SD	DEST	A1	R1	SRC1	0	A0	R2	SRC2	SCALE

OPC 指定指令的类型。

操作（OPC）：

0 dest = SAT((acc + SAT(2 * src1 * src2)) {->> scale})

1 dest = SAT((acc - SAT(2 * src1 * src2)) {->> scale})

助记符:

0 SMLDA <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}

1 SMLDS <dest>, <src1_16>, <src2_16>, <acc> {,<scale>}

标志: 见上节。

包含的理由:

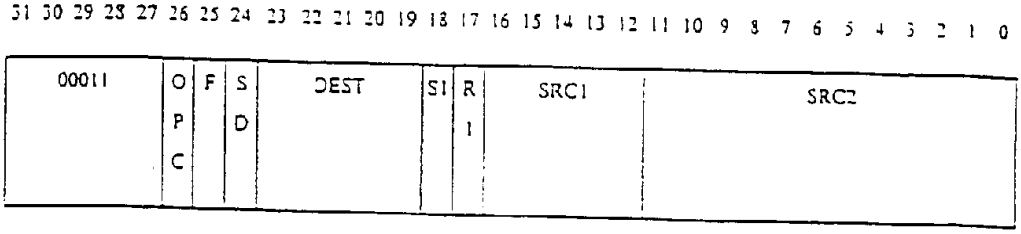
G.729及使用小数算术运算的其它算法需要MLD指令。大多数DSP提供能在累加或写回之前在乘法器的输出上左移一位的小数模式。作为特定的指令支持它提供更大的编程灵活性。等价于某些G系列基本运算的名称为:

L_msu=>SMLDS

L_mac=>SMLDA

在左移一位时它们利用乘法器的饱和。如果需要一序列的小数乘累加而不损失精度,可采用MULA,其和保持在33.14格式中。必要时,可在结束时利用左移及饱和转换到1.15格式。

乘法运算指令执行带符号乘法,及选用的定标/饱和。将源寄存器(只是16位)作为带符号数对待。



OPC指定指令的类型。

操作(OPC):

0 dest = (src1 * src2) {->> scale}

1 dest = SAT((src1 * src2) {->> scale})

助记符:

0 MUL <dest>, <src1_16>, <src2> {,<scale>}

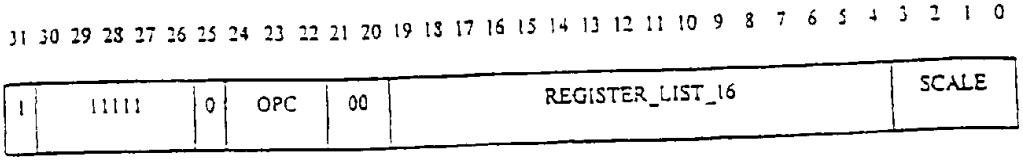
1 SMUL <dest>, <src1_16>, <src2> {,<scale>}

标志：见上节。

包含的理由：

许多处理需要带符号与饱和的乘法。

寄存器列表操作用来在一组寄存器上执行操作。提供了空与零指令用于在例程之前或之间复位选择的寄存器。提供了输出指令将列出的寄存器的内容存储到输出 FIFO 中。



OPC 指定指令的类型。

操作（OPC）：

000 对于（k = 0; k<16; k++）如果置位了寄存器列表的位 k，则将寄存器 k 标记为空。

001 对于（k = 0; k<16; k++）如果置位了寄存器列表的位 k，则将寄存器 k 设置成包含 0。

010 未定义

011 未定义

100 对于（k = 0; k<16; k++）如果置位了寄存器列表的位 k，则将（寄存器 k->>scale）写到输出 FIFO 中。

101 对于（k = 0; k<16; k++）如果置位了寄存器列表的位 k，则将（寄存器 k->>scale）写入到输出 FIFO 中并将寄存器 k 标记为空。

110 对于（k = 0; k<16; k++）如果置位了寄存器列表的位 k，则将 SAT（寄存器 k->>scale）写到输出 FIFO 中。

111 对于（k = 0; k<16; k++）如果置位了寄存器列表的位 k，则将 SAT（寄存器 k->>scale）写到输出 FIFO 中并将寄存器 k 标记为空。

助记符:

000	EMPTY	<register_list>
001	ZERO	<register_list>
010	Unused	
011	Unused	
100	OUTPUT	<register_list> {,<scale>}
101	OUTPUT	<register_list>^ {,<scale>}
110	SOUTPUT	<register_list> {,<scale>}
111	SOUTPUT	<register_list>^ {,<scale>}

标志:

不受影响

示例:

EMPTY	{A0, A1, X0-X3}
ZERO	{Y0-Y3}
OUTPUT	{X0-Y1}^

汇编程序还支持语法

OUTPUT	Rn

在这一情况中，利用 **MOV^, Rn** 指令输出一个寄存器。**EMPTY** 指令将停止直到所有要清空的寄存器包含有效数据

（即不空）。

寄存器列表操作不得在重新映射 REPEAT（重复）循环内使用。

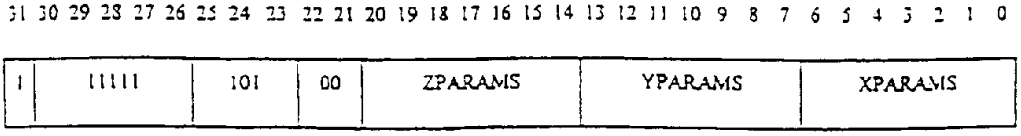
输出（OUTPUT）指令最多只能指定输出 8 个寄存器。

包含的理由：

例程结束后，下一个例程期望所有寄存器是空的以便它能从 ARM 接收数据。需要 EMPTY 指令来做到这一点。在执行 FIR 或过滤器之前，需要将所有累加器及部分结果清零。ZERO（零）指令协助做到这一点。通过取代一系列单个寄存器传送，两者都设计成改善代码密度。包含 OUTPUT（输出）指令通过取代一系列 MOV^, Rn 指令来改善代码密度。

提供了重新映射参数传送指令 RMOV 来允许配置用户定义的寄存器重新映射参数。

该指令编码如下：



各 PARAMS 字段包含以下的项：

6	5	4	3	2	1	0
BASEWRAP		BASEINC		0	RENUMBER	

这些项的含义如下:

参数	说明
RENUMBER	要在其上执行重新映射的16位寄存器数，可取值0、2、4、8。RENUMBER以下的寄存器重新映射，以上的直接存取。
BASEINC	各循环结束时基指针增加的量。可取值1、2或4。
BASEWRAP	基环绕模可取值2、4、8。

助记符:

RMOV <PARAMS>, [<PARAMS>]

<PARAMS>字段具有以下格式；

<PARAMS> ::= <BANK><BASEINC> n<RENUMBER> w<BASEWRAP>

<BANK> ::= [X¦Y¦Z]

<BASEINC> ::= [++¦+1¦+2¦+4]

<RENUMBER> ::= [0¦2¦4¦8]

<BASEWRAP> ::= [2¦4¦8]

如果使用 RMOV 指令同时重新映射是活动的，其行为是 UNPREDICTABLE（不可预测）。

标志：不受影响

重复指令提供硬件中的 4 个零周期循环。重复指令定义新的硬件循环。Piccolo 为第一条重复指令利用硬件循环 0，为嵌套在第一重复指令内的重复指令利用硬件循环 1 等等。重复指令不需要指定正在使用哪一个循环。重复循环必须严格嵌套。如果试图嵌套循环到大于 4 的深度，则行为是不可预测的。

各重复指令指定循环中的指令数（紧接在重复指令后面的）及通过循环的次数（它是常量或读自 Piccolo 寄存器）。

如果循环中的指令数较少（1 或 2）则 Piccolo 可用额外周期来建立循环。

如果循环计数是寄存器指定的，则蕴含 32 位存取（S1 = 1），但只认为底部 16 位是有效的并且数字是不带符号的。如果循环计数为 0，则循环的操作是未定义的。采取循环计数的复制，因此立即可以重用该寄存器（甚至再填充）而不影响循环。

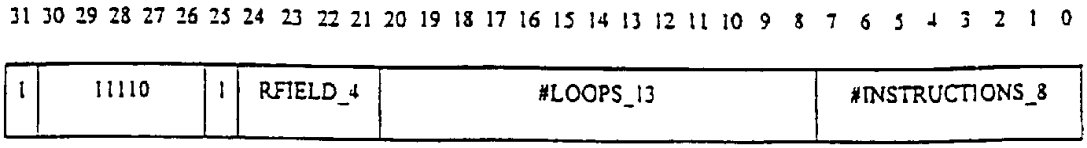
重复指令提供修改指定循环内的寄存器操作数的方式的机制。细节上面已描述过。

带有寄存器指定的循环数的重复的编码：

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	11110	0	RFIELD_4	00	0	R1	SRC1	0000	#INSTRUCTIONS_8

带固定的循环数的重复的编码:



RFIELD 操作数指定在循环内使用 16 种重新映射参数配置的哪一种。

RFIELD	重新映射操作
0	不执行重新映射
1	用户定义的重新映射
2..15	预置的重新映射配置 TBD

汇编程序提供两个操作码 REPEAT 与 NEXT 来定义硬件循环，REPEAT 在循环开始时而 NEXT 界定循环的结束，允许汇编程序计算循环体内的指令数。至于 REPEAT，它只须作为常量或寄存器指定循环次数。例如:

```
REPEAT    X0
MULA      A0, Y0.l , Z0.l , A0
MULA      A0, Y0.h^, Z0.h^, A0
NEXT
```

这将执行两条 MULA 指令 X0 次。同时，

```
REPEAT    #10
MULA      A0, X0^, Y0^, A0
NEXT
```

将执行 10 次乘累加。

汇编程序支持语法:

REPEAT # iterations[,<PARAMS>]以指定重复所用的重新映射参数。如果所需的重新映射参数等于预定义的参数组之一，则使用适当的 REPEAT 编码。如果不是，则汇编程序将生成 RMOV 来加载用户定义的参数，后面跟随 REPEAT 指令。见上面的节中的 RMOV 指令及重新映射参数格式的细节。

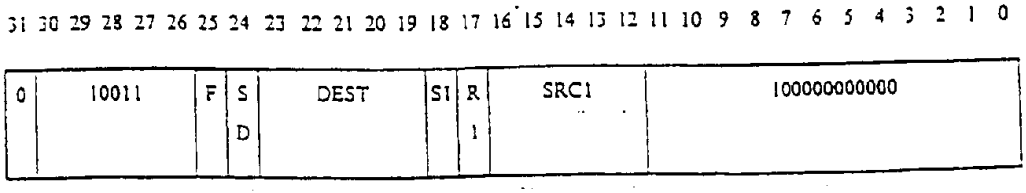
如果循环的重复次数为 0 则 REPEAT 的操作是不可预测的。

如果将指令字段的数字设置为 0 则 REPEAT 的操作是不可预测的。

循环只包含一条指令而该指令为转移时，则具有不可预测的表现。

从 REPEAT 循环界内转移到该循环的界外是不可预测的。

饱和绝对值指令计算源 1 的饱和的绝对值。



操作:

dest=SAT((src1>=0)?src1:-src1)。该值总是饱和的。具体地，0x80000000 的绝对值为 0x7fffffff 而不是 0x80000000！

助记符:

SABS <dest>, <src1>

标志:

Z 如果结果为 0，便置位。

N 保持。

C 如果 src1<0(dest=-src1 情况)，便置位。

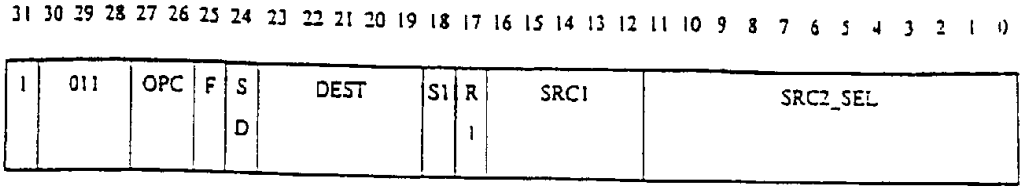
V 如果出现饱和，便置位。

包含的理由:

在许多 DSP 应用中有用。

选择操作（条件传送）用来有条件地将源 1 或源 2 传送到目的地寄存器中。选择总是等效于传送。还有在并行加/减以后使用的并行操作。

注意为了实现的原因，可读取两个源操作数，如果其中之一为空，指令将停止，不管该操作数是否是严格需要的。



OPC 指定指令的类型.

操作（OPC）：

00 如果<cond>对一级标志成立 则 dest = src1 否则 dest = src2

01 如果<cond>对一级标志成立 则 dest.h = src1.h 否则 dest.h=src2.h

如果<cond>对二极标志成立 则 dest.l = src1.l 否则 dest.l=src2.l

10 如果<cond>对一级标志成立 则 dest.h = src1.h 否则 dest.h=src2.h

如果<cond>对二级标志成立 则 dest.l = src1.l 否则 dest.l=src2.l

11 保留

助记符

00 SEL<cond> <dest>, <src1>, <src2>

01 SELTT<cond> <dest>, <src1>, <src2>

10 SELTF<cond> <dest>, <src1>, <src2>

11 不用

如果将寄存器标记为再填充，无条件将其再填充. 汇编程序还提供下列助记符:

MOV<cond> <dest>, <src1>

SELFT<cond> <dest>, <src1>, <src2>

SELFF <cond> <dest>, <src1>, <src2>

MOV<cond>A, B 等效于 SEL<cond>A, B, A. 通过互换 src1 与 src2 及使用 SELTF、SELTT 得到 SELFT 及 SELFF.

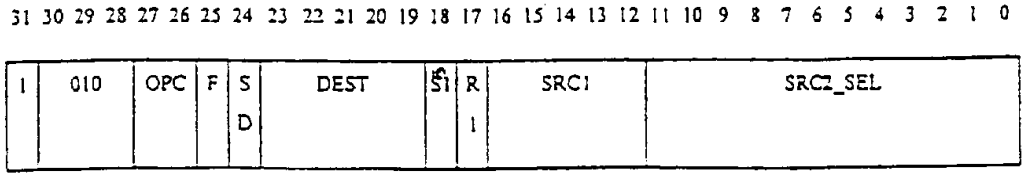
标志: 保持所有标志以便可以执行一序列选择。

包含的理由:

用于在线作出简单决定而无须依靠转移. 用于 Viterbi 算法及在样本或矢量中扫描最大元素时。

移位操作指令提供逻辑左与右移, 算术右移及循环指定的量. 认为移位量是取自寄存器内容的低 8 位的 - 128 与 + 127 之间的带符号整数或者在范围 + 1 至 + 31 中的立即数. 负数量的移位导致反方向上移位 ABS (移位量) .

将输入操作数符号扩展到 32 位; 在写回前将得出的 32 位输出符号扩展到 48 位从而写到 48 位寄存器表现合理。



OPC 指定指令的类型。

操作 (OPC) :

00 dest = (src2>=0) ? src1 << src2 : src1 >> -src2

01 dest = (src2>=0) ? src1 >> src2 : src1 << -src2

10 dest = (src2>=0) ? src1 ->> src2 : src1 << -src2

11 dest = (src2>=0) ? src1 ROR src2 : src1 ROL -src2

助记符:

```
00    ASL   <dest>, <src1>, <src2_16>
01    LSR   <dest>, <src1>, <src2_16>
10    ASR   <dest>, <src1>, <src2_16>
11    ROR   <dest>, <src1>, <src2_16>
```

标志:

Z 如果结果为 0，便置位。

N 如果结果为负，便置位。

V 保持

C 设置成移位出来的最后一位的值（和在 ARM 上一样）

寄存器指定的移位的行为为:

- LSL 移位 32 得出结果 0，C 设置为 src1 的位 0.

- LSL 移位 32 以上得出结果 0，C 设置为 0.

- LSR 移位 32 得出结果 0，C 设置为 src1 的位 31.

- LSR 移位 32 以上得出结果 0，C 设置为 0.

- ASR 移位 32 或以上得出用 src1 的位 31 填充及 C 设置为 src1 的位 31.

- ROR 移位 32 具有结果等于 src1 并将 C 设置成 src1 的位 31.

- ROR 移位 n 位，其中 n 大于 32，给出执行 ROR 移位 n-32 位相同的结果；因此从 n 中重复减去 32 直到该量在 1 至 32 范围中为止，见上。

包含的理由:

用 2 的幂乘/除。位与字段抽取。串行寄存器。

将未定义的指令在指令集清单中陈述如上。它们的执行将导致 Piccolo 停止执行，并置位状态寄存器中的 U 位，及禁止它本身（似乎清除了控制寄存器中的 E 位）。这允许截获指令集的任何未来扩充并在现有实现上有选择地仿真。

从 ARM 访问 Piccolo 状态如下。状态访问模式用来观察/修改 Piccolo 的状态。为两种目的设置这一机制:

- 上下文切换。

- 调试。

通过执行 PSTATE 指令将 Piccolo 置于状态访问模式中．这一模式允许用一序列 STC 与 LDC 指令保存及恢复所有 Piccolo 状态．当进入状态访问模式时，将 Piccolo 协处理器 ID PICCOLO1 的使用修改成允许访问 Piccolo 的状态．有 7 组 Piccolo 状态．可以用单一的 LDC 或 STC 加载与存储特定组中的所有数据．

组 0: 专用寄存器．

- 一个 32 位字，包含 Piccolo ID 寄存器的值（只读）．

- 一个 32 位字，包含控制寄存器的状态．

- 一个 32 位字，包含状态寄存器的状态．

- 一个 32 位字，包含程序计数器的状态．

组 1: 通用寄存器（GPR）

- 16 个 32 位字，包含通用寄存器状态．

组 2: 累加器

- 4 个 32 位字，包含累加器寄存器的高 32 位（注意，为了恢复的目的，以 GPR 状态进行复制是必要的 - 否则会蕴含该寄存器组上的另一次写使能）．

组 3: 寄存器/Piccolo ROB/输出 FIFO 状态．

- 一个 32 位字，指示哪些寄存器标记为再填充（每一个 32 位寄存器 2 位）．

- 8 个 32 位字，包含 ROB 标签的状态（存储在位 7 至 0 中的 8 个 7 位项）．

- 3 个 32 位字，包含不对齐的 ROB 锁存器的状态（位 17 至 0）．

- 一个 32 位字，指示输出移位寄存器中哪些槽包含有效数据（位 4 表示空，位 3 至 0 编码所用项的号码）．

- 一个 32 位字，包含输出 FIFO 保持锁存器的状态（位 17 至 0）．

组 4: ROB 输入数据．

- 8 个 32 位数据值．

组 5: 输出 FIFO 数据．

- 8 个 32 位数据值．

组 6: 循环硬件．

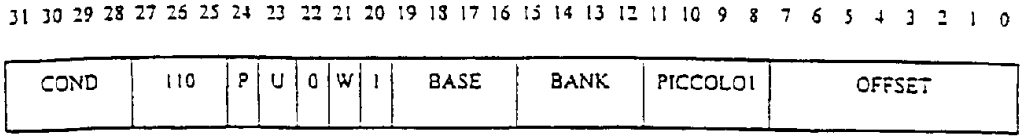
- 4 个 32 位字，包含循环起始地址．

- 4 个 32 位字，包含循环结束地址．

- 4 个 32 位字，包含循环计数（位 15 至 0）.

- 一个 32 位字，包含用户定义的重新映射参数及其它重新映射状态.

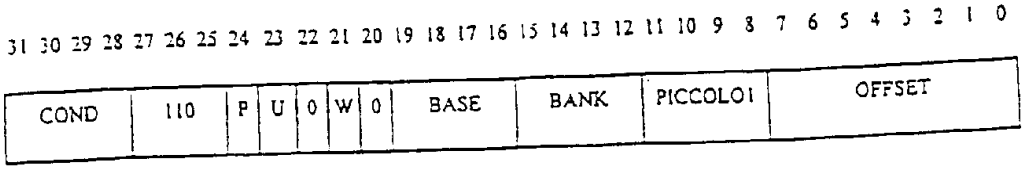
LDC 指令用于在 Piccolo 在状态访问模式中时加载 Piccolo 状态. BANK 字段指示正在加载哪一个组.



以下序列加载来自寄存器 R0 中的地址的所有 Piccolo 状态.

```
LDP B0,[R0],#16!; 专用寄存器
LDP B1,[R0],#64!; 加载通用寄存器
LDP B2,[R0],#16!; 加载累加器
LDP B3,[R0],#56!; 加载寄存器/ROB/FIFO 状态
LDP B4,[R0],#32!; 加载 ROB 数据
LDP B5,[R0],#32!; 加载输出 FIFO 数据
LDP B6,[R0],#52!; 加载循环硬件
```

STC 指令用于在 Piccolo 在状态访问模式中时存储 Piccolo 状态. BANK 字段指定正在存储哪一个组.



以下序列将所有 Piccolo 状态存储到寄存器 R0 中的地址.

```
STP B0,[R0],#16!; 保存专用寄存器
STP B1,[R0],#64!; 保存通用寄存器
STP B2,[R0],#16!; 保存累加器
STP B3,[R0],#56!; 保存寄存器/ROB/FIFO 状态
STP B4,[R0],#32!; 保存 ROB 数据
STP B5,[R0],#32!; 保存输出 FIFO 数据
```

STP B6,[R0],#52!; 保存循环硬件

调试模式－Piccolo需要响应与ARM所支持的相同的调试机制，即软件通过Demon与Angel，以及带有嵌入的ICE的硬件，下面是调试Piccolo系统的若干机制：

－ARM指令断点。

－数据断点（观察点）。

－Piccolo指令断点。

－Piccolo软件断点。

ARM指令与数据断点是由ARM嵌入的ICE模块处理的；Piccolo指令断点是由Piccolo嵌入的ICE模块处理的；Piccolo软件断点是由Piccolo核处理的。

硬件断点系统可配置成使ARM与Piccolo两者都有断点。

软件断点由Piccolo指令（停机或中断）处理，导致Piccolo停止执行，进入调试模式（置位状态寄存器中的B位）及禁止本身（似乎已用PDISABLE指令禁止Piccolo）。程序计数器保持有效，允许恢复断点地址。Piccolo不再执行指令。

单步进Piccolo可通过在Piccolo指令流上设定一个断点接一个断点来完成。

软件调试－Piccolo提供的基本功能便是在状态访问模式中通过协处理器指令加载与保存所有状态到存储器中的能力。这允许调试程序将所有状态保存在存储器中，读取与/或更新它及恢复到Piccolo中。Piccolo存储状态机制是非破坏性的，即Piccolo的存储状态操作不会破坏任何Piccolo内部状态。这意味着Piccolo在转储其状态之后不首先再一次恢复它便能重新起动。

要确定找出Piccolo高速缓冲存储器的状态的机制。

硬件调试－硬件调试由Piccolo的协处理器接口上的扫描链提供。然后可将Piccolo置于状态访问模式中并通过该扫描链检验/修改其状态。

Piccolo状态寄存器包含单一的位来指示它已执行了断点指令。在执行断点指令时，Piccolo置位状态寄存器中的B位，并停止执行。为了能查询Piccolo，调试程序必须启动Piccolo并通过在能出现随后的存取之前写入其控制寄存器而将其置于状态访问模式中。

图 4 示出响应高/低位及大小位将选择的寄存器的适当的一半切换到 Piccolo 数据路径上的多路复用器配置. 如果大小位指示 16 位, 则符号扩展电路用适当的 0 或 1 填充数据路径的高位.

说 明 书 附 图

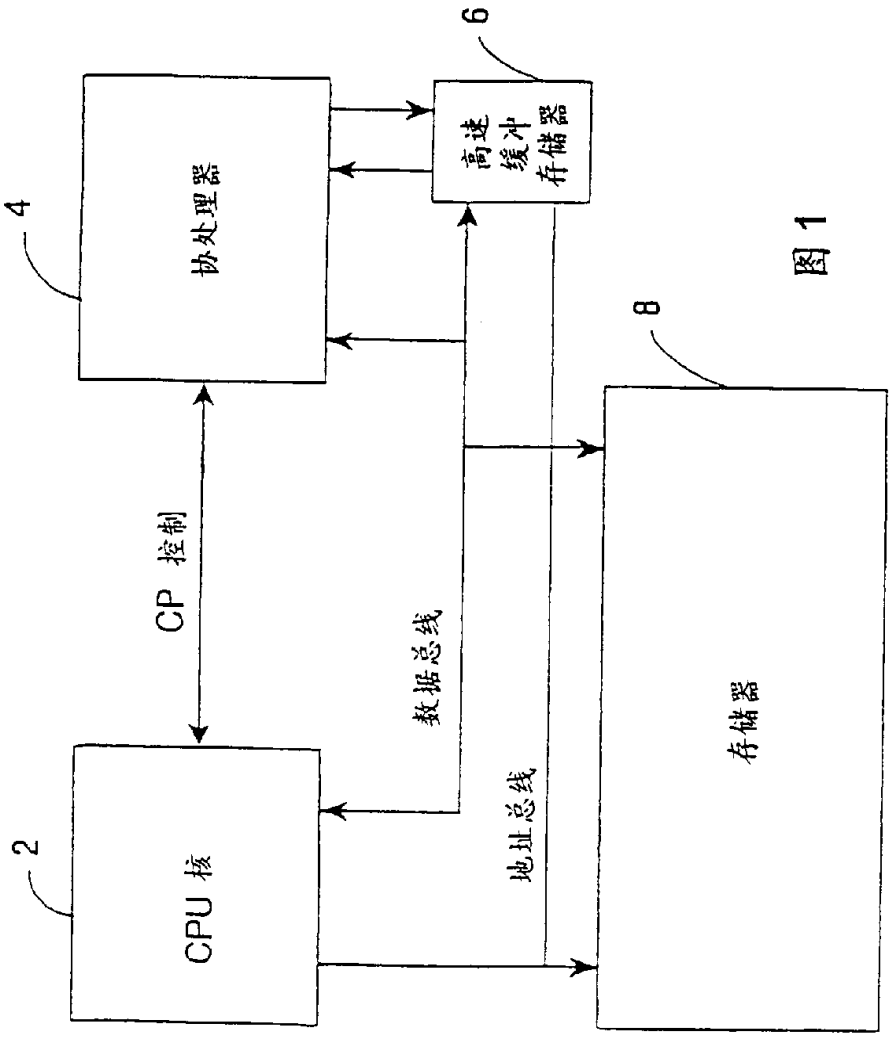


图 1

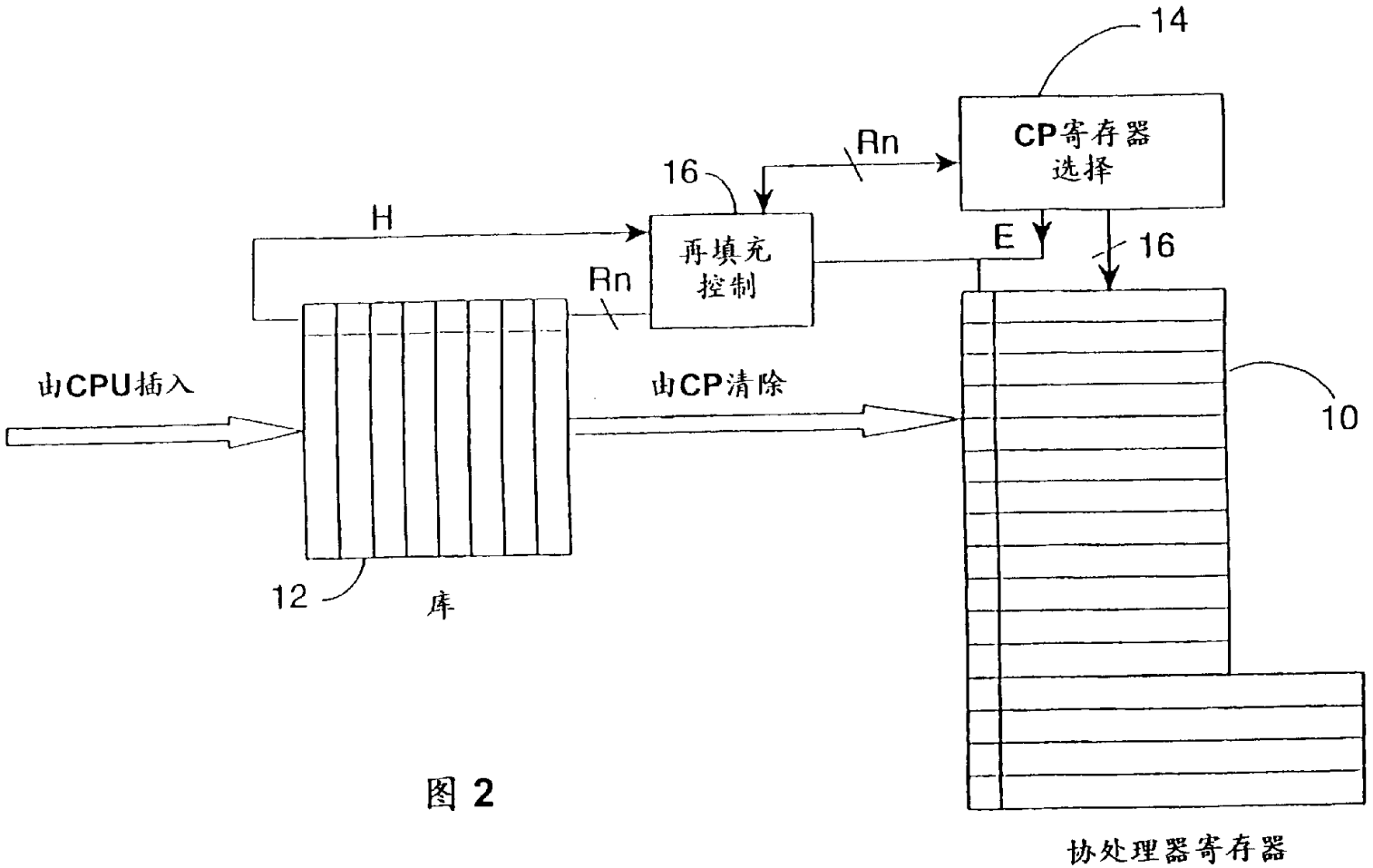


图 2

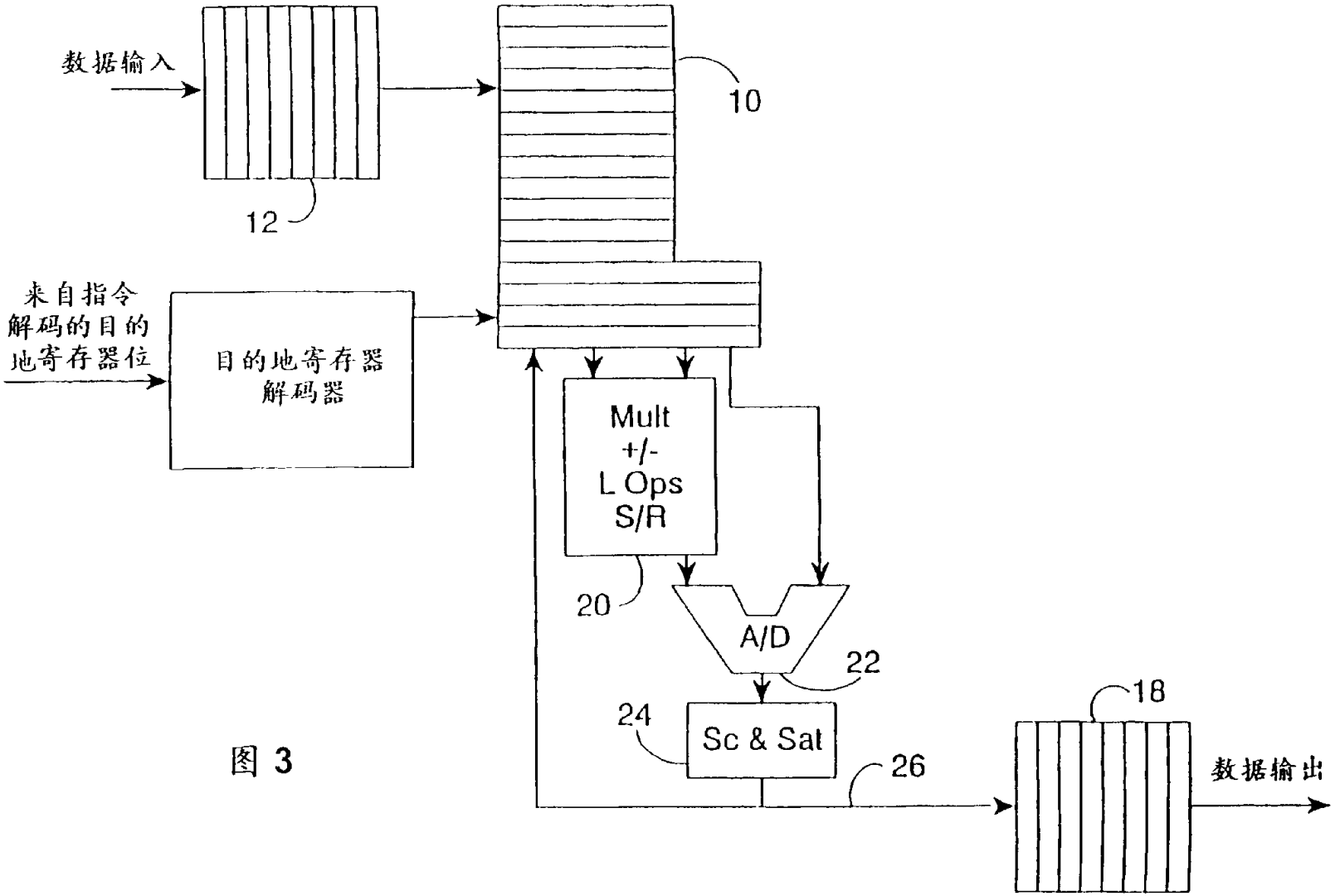

数据输入
12
10
来自指令
解码的目的
地寄存器位
目的地寄存器
解码器
Mult
+/-
L Ops
S/R
20
A/D
22
24
Sc & Sat
26
18
数据输出

图 3

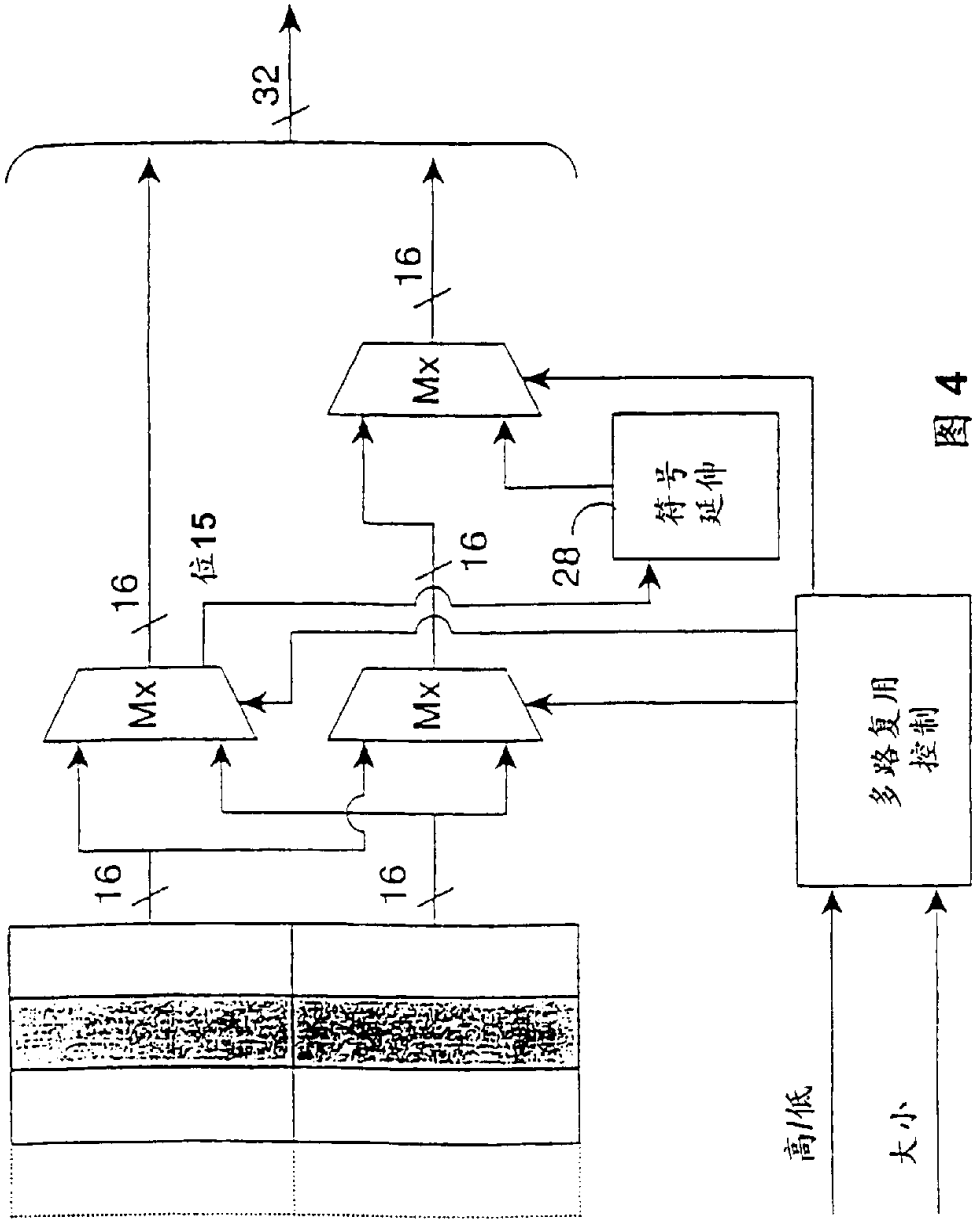


图 4

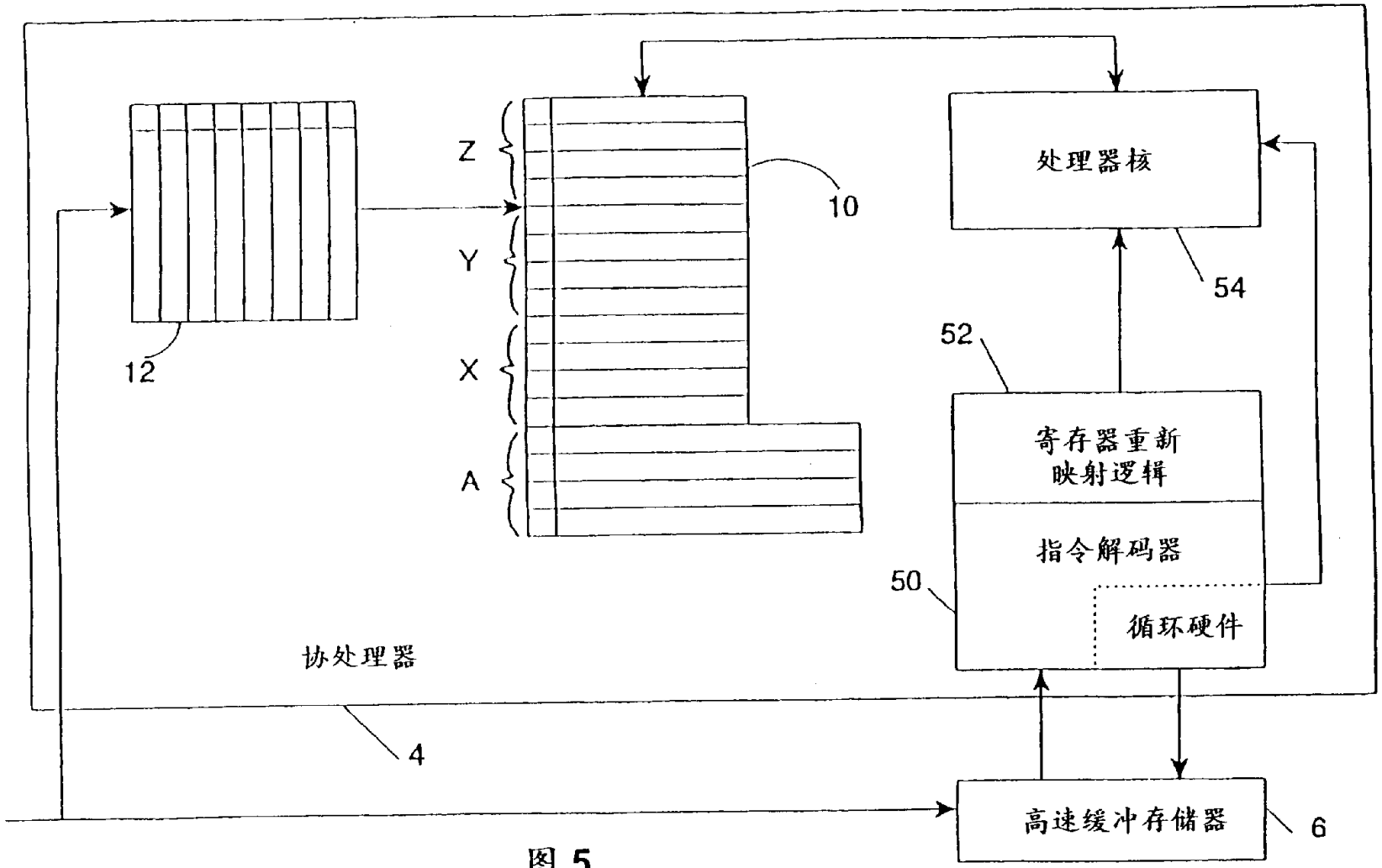


图 5

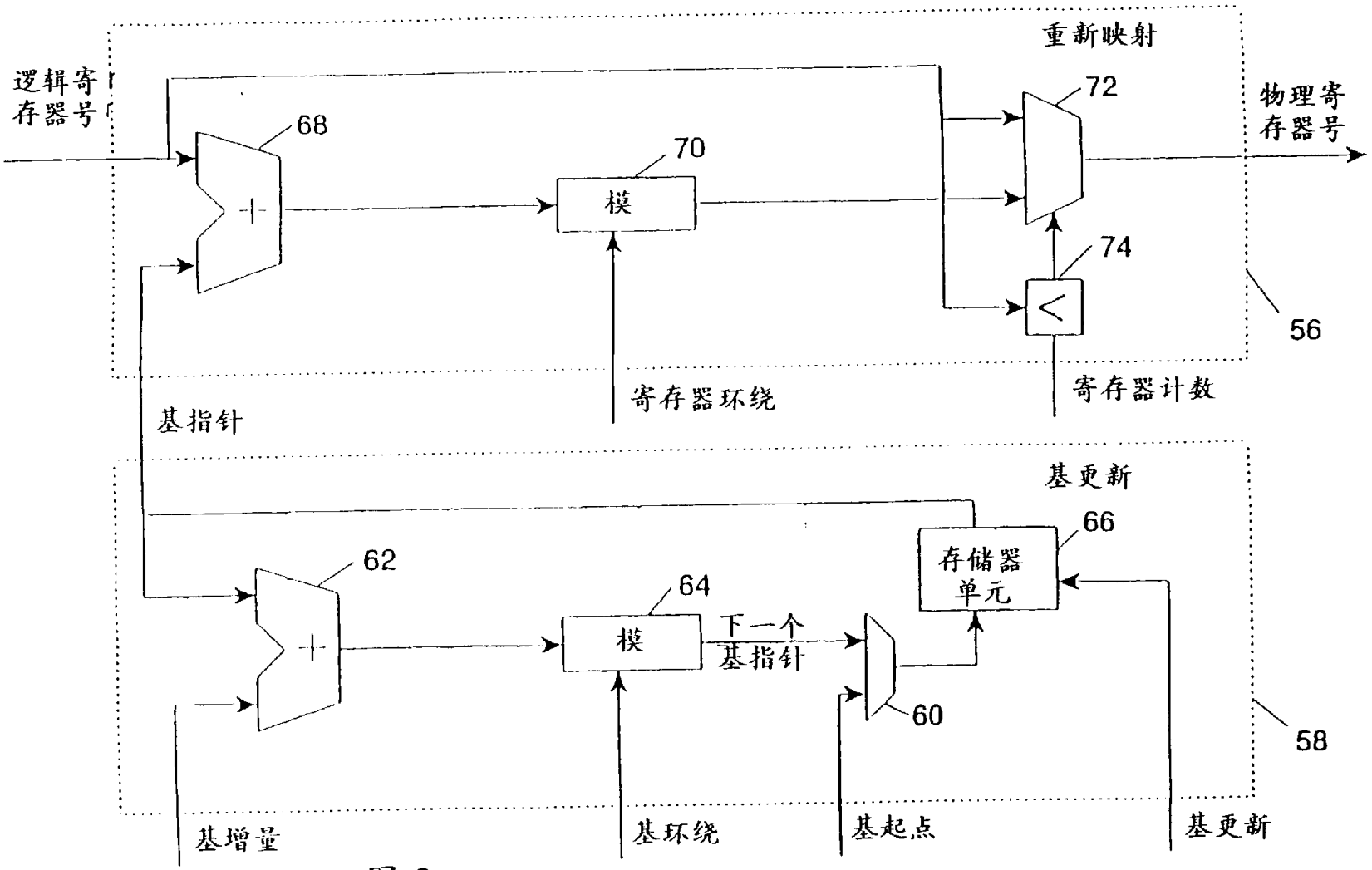

重新映射
逻辑寄存器号
物理寄存器号
68
70
72
74
模
寄存器环绕
寄存器计数
56
基指针
基更新
62
64
66
模
下一个基指针
存储器单元
60
58
基增量
基环绕
基起点
基更新

图 6

FIR（ 块过滤　算法 ）

	d0	d1	d2	d3	d4	d5
A0=	c0	c1	c2	c3	c4	c5
A1=		c0	c1	c2	c3	c4
A2=			c0	c1	c2	c3
A3=				c0	c1	c2

图 7