[19]中华人民共和国国家知识产权局

[51]Int. Cl⁷
G06F 15/78
G06F 9/318

[12] 发明专利申请公开说明书

[21] 申请号 99811315.8

[43]公开日 2001年10月24日 [11]公开号 CN 1319210A

[22]申请日 1999.9.10 [21]申请号 99811315.8
[30]优先权
[32]1998.9.23 [33]DE [31]19843640.8
[86]国际申请 PCT/DE99/02878 1999.9.10
[87]国际公布 WO00/17771 德 2000.3.30
[85]进入国家阶段日期 2001.3.23
[71]申请人 因芬尼昂技术股份公司
地址 德国慕尼黑
[72]发明人 R·阿诺 H·克莱维
C·西默斯
[74]专利代理机构 中国专利代理(香港)有限公司
代理人 程天正 张志醒

权利要求书4页 说明书31页 附图页数8页

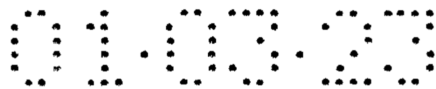
[54]发明名称 可配置的硬件块的配置方法

[57]摘要

讲述可配置的硬件块的不同配置方法。该方法的特征尤其在于被 用来配置硬件块的配置数据的生成。通过所讲述的配置数据生成，配 置数据生成本身和硬件块配置都可以利用该配置数据来简单、快速及 有效地实现。

ISSN1008-4274

知识产权出版社出版



权 利 要 求 书

1. 可配置的硬件块的配置方法，

其特征在于：

通过采用如下配置数据来实现硬件块的配置，即该配置数据是从需执行的程序的指令或指令序列的转换中得出的，而且，在转换该指令或指令序列时执行如下步骤，

－测定为执行相应指令所需要的、可配置硬件块的子单元(AUx, CU, DEMUX, MUXAx, MUXB)的类型，

－选出还没有在其它方面被占用的、事先所测定的类型的子单元，而且，若能找到这种子单元，则

－对设置在所选子单元周围的可配置的通信连接进行配置。

2. 如权利要求1所述的方法，

其特征在于：

所述转换从只具有一个入口点和一个出口点的指令块的第一个指令开始。

3. 如权利要求1或2所述的方法，

其特征在于：

所述转换在只具有一个入口点和一个出口点的指令块的最后一个指令的转换之后结束。

4. 如权利要求2或3所述的方法，

其特征在于：

所述的转换是以超块的方式实现的。

5. 如上述权利要求之一所述的方法，

其特征在于：

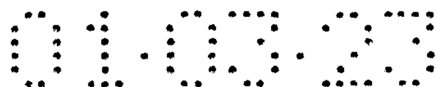
如果转换所需要的、硬件块的组件不可用或不再可用，则自动地终止所述的转换。

6. 如上述权利要求之一所述的方法，

其特征在于：

给硬件块的可按功能配置的子单元(AUx, CU, DEMUX, MUXAx, MUXB)分配一些虚拟单元，其中，所述的虚拟单元表示了一些功能，该功能可以通过不同的配置赋予相关的子单元。

7. 如权利要求6所述的方法，



其特征在于:

所有物理子单元(AUx, CU, DEMUX, MUXAx, MUXB)的所述虚拟单元被登录在一个表格或目录内。

8. 如权利要求7所述的方法,

5 其特征在于:

所述表格或目录的登录项包括如下有关信息,即给哪个物理子单元(AUx, CU, DEMUX, MUXAx, MUXB)分配相关的虚拟单元。

9. 如权利要求7或8所述的方法,

其特征在于:

10 所述表格或目录的登录项包括如下有关信息,即为了给物理子单元赋予由虚拟单元表示的功能,需要怎样来配置所分配的该物理子单元(AUx, CU, DEMUX, MUXAx, MUXB)。

10. 如上述权利要求之一所述的方法,

其特征在于:

15 通过寻找所需类型的虚拟子单元来选择为执行指令所需的子单元(AUx, CU, DEMUX, MUXAx, MUXB)。

11. 如权利要求10所述的方法,

其特征在于:

对于为使用而选出的所需类型的虚拟单元以及象该选出的虚拟单

20 元一样被分配给同一物理子单元(AUx, CU, DEMUX, MUXAx, MUXB)的虚拟单元,均由该方法负责使它们在接下来的转换中再也不能被选出来使用。

12. 如上述权利要求之一所述的方法,

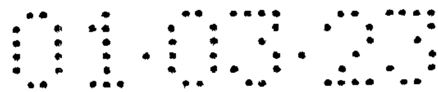
其特征在于:

25 在对设置于所选子单元(AUx, CU, DEMUX, MUXAx, MUXB)周围的可配置通信连接进行配置、以便把相关的子单元同由需转换的指令定义的数据源及/或信号源连接起来时,检验所述相关的数据源及/或信号源是否为事先已由硬件块的子单元之一写过的存储区。

13. 如权利要求12所述的方法,

30 其特征在于:

如果确定由需转换的指令所定义的数据源及/或信号源事先已被硬件块的子单元(AUx, CU, DEMUX, MUXAx, MUXB)之一写过,则将该子单元



用作数据源和/或信号源。

14. 如上述权利要求之一所述的方法，

其特征在于:

在对设置于所选子单元(AUx, CU, DEMUX, MUXAx, MUXB)周围的可配置通信连接进行配置、以便把相关的子单元同由需转换的指令定义的数据目标及/或信号目标连接起来时，检验所述相关的数据目标及/或信号目标是否也为硬件块的其它子单元所写的存储区。

15. 如权利要求 14 所述的方法，

其特征在于:

如果确定由需转换的指令所定义的数据目标及/或信号目标也为硬件块的其它子单元(AUx, CU, DEMUX, MUXAx, MUXB)所写的存储区，则将其它存储区用作数据目标和/或信号目标。

16. 如权利要求 15 所述的方法，

其特征在于:

对于表示同一数据目标和/或信号目标的存储区，为其执行在超标量处理器中所应用的寄存器重命名。

17. 如上述权利要求之一所述的方法，

其特征在于:

如果在需转换的指令中包含有恒量，则寻找包含该恒量的恒量存储区，然后，如果找到该恒量存储区，则将该恒量存储区用作数据源和/或信号源。

18. 如权利要求 17 所述的方法，

其特征在于:

如果所述的恒量没有存储在现有的恒量存储区中，则将该恒量存入一个新的恒量存储区，并把该新的恒量存储区用作数据源和/或信号源。

19. 尤其如上述权利要求之一所述的方法，

其特征在于:

在将指令转换成配置数据时，尝试构造包括有多个超块的伪超块。

20. 如权利要求 19 所述的方法，

其特征在于:

所述的伪超块通过采用 if 转换来形成。

21. 如权利要求 19 或 20 所述的方法，

其特征在于:

根据可能性按伪超块的方式将指令转换成配置数据。

5 22. 可配置的硬件块的配置方法，

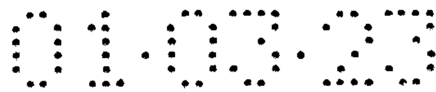
其特征在于:

通过采用由转换如下代码所产生的配置数据来实现硬件块的配置，即如果通过采用电路说明语言定义了可由所述可配置的硬件块实现的电路，则生成所述的代码。

10 23. 如权利要求 22 所述的方法，

其特征在于:

采用 VHDL 作为电路说明语言。



说　明　书

可配置的硬件块的配置方法

本发明涉及一种如权利要求 1 及 22 的前序部分所述的方法，也即一种可配置的硬件块的配置方法。

为了对所谓的>S<puter 的 s 单元进行配置，需要有这种方法。>S<puter 为一种程控单元，通过在处理指令的部分中采用可配置的硬件块，所述程控单元能够在每个处理器脉冲内执行多于一个的指令。

这种>S<puter 譬如曾在 EP 0 825 540 A1 中公开过。

>S<puter 的基本结构如图 11 所示，下面参考附图来对此进行讲述。

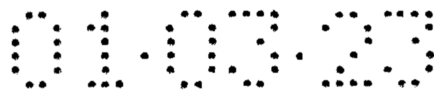
出于完整性的原因，需要指出的是，此处只部分地示出和讲述所述的>S<puter、尤其是其处理指令的部分（针对当前所考察的可配置的硬件块及其配置，只讲述对其有意义的部分）。

图 11 所示的>S<puter 包括一个预解码单元 1、指令缓冲器 2、解码重命名及装载单元 3、上文已提及的 s 语句变化单元（s 单元）4、数据高速缓冲器 5、以及存储器接口 6，其中，s 单元 4 包括有结构编程缓冲器(可编程结构缓冲器)41、具有可编程结构的功能单元 42、整型/地址指令缓冲器 43 和寄存器块（整型寄存器文件）44。

所述>S<puter 的特殊性尤其在于其 s 单元 4，准确地说是功能单元 42。该功能单元 42 是一种可结构化的硬件，它可根据>S<puter 所执行的指令或指令序列进行动态地配置，使得能够执行由指令或指令序列所规定的作用或操作。

由>S<puter 执行的指令（准确地说是表示该指令的代码数据）从图中未示出的存储器出发，经过存储器接口 6 到达预解码单元 1，并在此进行预解码；在此，譬如可以在所述的代码数据中插入一些信息，该信息可以使稍后在解码重命名及装载单元 3 中的解码变得更容易。然后，所述的代码数据经过指令缓冲器 2 到达该解码重命名及装载单元 3，并在此对由代码数据表示的指令的执行进行预处理。该预处理包括代码数据的解码、功能单元 42 的配置或结构化、整型寄存器文件 44 的初始化或管理、以及按需要进行配置的功能单元 42 的启动。

通过采用表示所需配置的配置数据来实现所述功能单元 42 的结



构化或配置，而且该配置数据从解码重命名及装载单元 3 被写入到可编程结构缓冲器 41 中。所述表示所需配置的配置数据是在解码重命名及装载单元 3 内创建的；但它们也可以以编码的形式而包含在所述的编码数据之中。

功能单元 42 被设计用来从寄存器文件 44 及/或数据高速缓冲器 5 中读取数据，并对读出的数据进行算术及/或逻辑处理，然后把表示该处理结果的数据写入寄存器文件 44 和/或数据高速缓冲器 5 之中。

在对寄存器文件 44 进行合适的初始化和对功能单元 42 进行合适的配置时，由所述功能单元 42 的工作来执行那些通过执行如下指令而导致的操作，即所述寄存器文件 44 的初始化和功能单元 42 的配置是基于该指令而产生的。

众所周知，对于通过执行指令所导致的作用而言，如果用相应配置的硬件（功能单元 42）来执行它，则要比由常规程控单元在“标准”的算术逻辑单元（ALU）中执行这些指令时快得多。这在如下情形尤其是这样，即所述的硬件（功能单元 42）作如此配置，使得通过其工作可实现与执行多个相继的指令（包括多个指令的宏指令）相一致的结果。

有关>S<puter 的构造、功能及作用方式以及在此所包含的可配置硬件的详细情况可以从所述的 EP 0 825 540 A1 中查取。

出于完整性的原因，此处应指出的是，并不是所有由>S<puter 执行指令时所导致的作用都能由功能单元 42 执行。尤其是诸如分支、转移、不操作、等待以及停止等被用于程序运行控制或监视流程控制的指令通常都是用常规的方式和方法来执行的。

不过，通过采用诸如功能单元 42 等可配置的硬件块，每时间单位通常可以比采用常规的程控单元要执行更多的、由执行指令所导致的作用，也就是说，每个处理器脉冲可以处理多于一个的指令。

当然，前提条件是所述的硬件块能快速地配置和有效地利用。

因此本发明的任务在于，对权利要求 1 的前序部分所述的方法作如下改进，使得任意应用的硬件块可以快速和简单地配置，并能有效地得到使用。

根据本发明，该任务由权利要求 1 的特征部分和权利要求 22 的特征部分所述的特征来解决。

为此规定,

－通过采用如下配置数据来实现硬件块的配置,即该配置数据是从需执行的程序的指令或指令序列的转换中得出的,而且,在转换该指令或指令序列时执行如下步骤(权利要求1的特征部分):

5 －测定为执行相应指令所需要的、可配置硬件块的子单元的类型,

－选出还没有在其它方面被占用的、事先所测定的类型的子单元,而且,若能找到这种子单元,则

－对设置在所选子单元周围的可配置的通信连接进行配置,

10 或者(权利要求22的特征部分):

－通过采用由转换如下代码所产生的配置数据来实现硬件块的配置,即如果通过采用电路说明语言定义了可由所述可配置的硬件块实现的电路,则生成所述的代码。

通过这种方案,在任何情况下都可以利用极小的费用对需配置的

15 硬件块进行所需的配置。在此,该配置能迅速地实现,并可优化利用所述硬件块的各部分;如此配置的硬件块能够非常有效地得到使用。

本发明的优选扩展方案由从属权利要求、下文的说明和附图给出。

下面参照附图并借助实施例来详细阐述本发明。

20 图1A-1D示出了统一指令格式的例子,该指令格式具有在本实施例中需要转换的指令,或者优选地在转换之前将需要转换的指令置为该指令格式,

图2示出了在按硬件实现所述的转换时所采用的查询表,

图3A和3B示出了从图2所示查询表中输出的数据格式,

25 图4示出了用于选择和配置为执行指令而需要的硬件块子单元的电路框图,

图5示出了为由图4所示电路所选出的子单元确定数据源和/或信号源以及数据目标和/或信号目标的电路框图,

图6示出了用于对需转换的指令中所包含的恒量进行处理的电路

30 框图,

图7示出了用于实现所谓的数据转发的电路框图,

图8示出了一种把配置数据写入到暂时比特流中的所谓纵横开

关，

图 9 示出了用于将所述的暂时比特流注入到主比特流中的装置，

图 10 示出了一种把指令转换成用于对硬件块进行预期配置的配置数据的全部装置，

图 11 示出了>S<puter 的原理结构，以及

图 12 示出了当前所示类型的硬件块的原理结构。

为便于理解，首先来讲述可通过下面讲述的方法进行配置的硬件块的原理结构。

该硬件块的原理结构如图 12 所示。所示的硬件块被设计用来根据其配置读出存于存储装置之内的数据，并对读出的数据进行算术及/或逻辑处理，然后把表示该处理结果的数据写入所述的存储装置；它譬如可以用作如图 11 所示的>S<puter 的功能单元 42。

对于由可配置的硬件块从其读出数据以及由该硬件块向其写入数据的存储装置，它们可以装设在所述硬件块之内或之外；在本实施例中，所述存储装置由图 11 所示的>S<puter 的寄存器文件 44 构成。该硬件块为一种位于存储装置输出和输入端之间的异步开关网络；硬件块的组件异步地相互进行耦合。

在硬件块投入运行之前，可优选地从该硬件块的外部对存储装置进行初始化；可以想见，也可由硬件块自己来促使或实现存储装置的初始化。

图 12 所示的硬件块具有一个或多个算术单元 AU1、AU2，一个或多个比较单元 CU，一个或多个第一类型的多路转换器 MUXA1、MUXA2、MUXA3，一个或多个第二类型的多路转换器 MUXB，以及一个或多个多路分解器 DEMUX。

在本实施例中，所述的算术单元 AU1、AU2 具有两个输入端、一个输出端和一个控制端。算术单元 AU1、AU2 负责对经其输入端输入的输入信号进行算术和/逻辑处理。由算术单元 AU1、AU2 执行的操作可以被固定，或可以专门地调整（配置）；该单元尤其包括有加法、减法、乘法、除法等算术运算，“与”连接、“或”连接、倒置、求补等逻辑连接，算术和逻辑变换运算，以及数据转接（把输入信号接通到输出端）。所述的算术单元 AU1、AU2 与诸如微处理器、微控制器等常规程控单元的算术/逻辑单元（ALU）是不同的；算术单元 AU1、AU2 所执

行的操作是有限的，这样其结构可以较为简单。通过算术单元 AU1、AU2 的控制端，可以规定相关的算术单元是否要执行为其设定的操作。对于其执行取决于存在一定条件的指令来说，这可以为其实现实际的转换。所述的条件譬如为某一标志的状态：如果标志被设置，则执行
5 相关算术单元所负责的任务（譬如加法），否则不执行（或返回）。这种在下文被称为“条件指令”的指令可以取消很难运用的条件转移指令；这在稍后还要详细讲述。

在所示的实施例中，比较单元 CU 具有两个输入端和一个输出端。该比较单元 CU 负责把其输入端上的信号或数据进行比较运算。由比较
10 单元 CU 执行的运算可以被固定，或可专门地进行调整（配置）；该单元譬如包括有大于、大于/等于、小于、小于/等于、等于、以及不等于等比较，而且还包括对真和假的检验。比较单元 CU 的输出端通过下文将详细讲述的多路分解器 DEMUX 而被接到算术单元 AU1、AU2 的控制端上。于是，算术单元 AU1、AU2 是否要执行为其设定需要执行的运算，
15 取决于在比较单元 CU 内所执行的操作的结果。

第一类型的多路转换器 MUXA1、MUXA2、MUXA3，第二类型的多路转换器 MUXB，以及多路分解器 DEMUX 被用来选择数据源及/或信号源和选择数据目标及/或信号目标。准确地说是：

- 多路转换器 MUXA1 用来选择输入到算术单元 AU1 输入端的数据
20 和/或信号的信源（在本实施例中可能的数据源和/或信号源为寄存器文件 44 及其它的算术单元），

- 多路转换器 MUXA2 用来选择输入到算术单元 AU2 输入端的数据和/或信号的信源（在本实施例中可能的数据源和/或信号源为寄存器文件 44 及其它的算术单元），

25 - 多路转换器 MUXA3 用来选择输入到比较单元 CUA 输入端的数据和/或信号的信源（在本实施例中可能的数据源和/或信号源为寄存器文件 44 及其它的算术单元），

- 多路转换器 MUXB 用来选择输入到寄存器文件的数据和/或信号的信源（在本实施例中可能的数据源和/或信号源为算术单元和/或寄
30 存器文件本身），

- 多路分解器 DEMUX 用来选择从比较单元 CUA 输出的数据和/或信号的目标（在本实施例中可能的数据目标和/或信号目标为算术单

元）。

第一类型的多路转换器具有多个输入端和两个输出端，第二类型的多路转换器具有多个输入端和一个输出端，而多路分解器具有一个输入端和多个输出端。

所述的多路转换器和多路分解器具有图 1 未示出的控制端，通过该控制端可以调整将哪些输入数据和/或输入信号接通到哪些输出端上。所述控制端的数目取决于各种分配组合所需要的数目；在 32 个输入端和两个输出端的情况下譬如需要 10 个控制端，以便将任意输入端上的信号和/或数据接通到任意的输出端上。在图 11 所示的>S<puter中采用硬件块作为功能单元 42 的情况下，控制信号端优选地被连接在可编程结构缓冲器 41 上，这样，写到该缓冲器内的配置数据基本上可以直接地用于多路转换器控制。在可编程结构缓冲器 41 内存储的配置数据优选地还包括用于规定如下单元相应功能的配置数据，即算术单元 AU1、AU2 和比较单元 CU。

利用算术单元 AU1、AU2，比较单元 CU，第一类型的多路转换器 MUXA1、MUXA2、MUXA3，第二类型的多路转换器 MUXB 以及多路分解器 DEMUX，硬件块可以读出存于存储装置（寄存器文件 44）之内的数据，并对读出的数据进行算术及/或逻辑处理，然后把表示该处理结果的数据写入所述的存储装置（寄存器文件 44）。

图 12 所示的、并参考其进行说明的硬件块只是被考虑用来阐述基本的结构。在实际中，所述的算术单元、比较单元、多路转换器以及多路分解器等可以装设比图 12 所示实施例多得多的数量。优选地，如此来选择硬件块的大小，使得在通常情况下，由稍后还要详细讲述的所谓超块所导致的全部操作都可一次性地编入到该硬件块之中。

装设在硬件块中的数据回路和/或信号回路可以通过单个的线路或通过总线来构成，其中，如果在硬件块的各个子单元或总线系统中可以对需要考虑多少和/或哪些总线进行配置，则证明是比较优选的。

硬件块可以基于指令或指令序列并按图 12 所示的类型进行配置。如果把指令或指令序列转换成相应的硬件块结构，则如此配置的硬件块可以作为顺序指令序列的运行单元而进行使用。这种硬件块配置的形式在下面也被称作结构过程的编程。

结构过程的编程的出发点可以是用诸如 C、C++等高级语言编写的

01.03.23

程序。该程序通过编译器进行翻译，由此获得的代码（优选地以超块的方式）被转换成结构信息，而需配置的硬件块则在该结构信息的基础上进行配置。稍后将详细讲述怎样理解超块。

显然，结构过程的编程的出发点也可以是用汇编语言编写的程序或其它类似的程序。编程的方式和方法（功能式，命令式，面向对象，...）同样是没有限制的。

采取如下做法被证明是有利的，即转换成结构信息的代码、亦即通过编译器或以其它方式及方法生成的机器指令只包括某些机器指令类型，即无条件指令、条件指令、谓词指令和 Loop(循环)指令。于是，通常可形成只具有一个入口点和一个出口点的非常长（尤其包含有许多指令）的指令块。生成尽可能长的、只具有一个入口点和一个出口点的指令块是很有意义的，因为属于一个或同一指令块的指令、并且也只有这些指令可以以一个单元的形式（作为由多个指令组成的宏指令）进行处理，而所述的单元可以转换成一个共同的硬件块结构，并一次性地得到执行。如果硬件块的配置总是严格地基于这种单元（而且硬件块大得足以进行该配置），那么，程序处理所需要的、硬件块的再结构化或再配置的数量可减到最小。其生成在目前令人满意的、也可通过上述指令群形成的这种指令块便是上文讲述的超块。

超块尤其还有一个特征在于，通过采用下文还要详细讲述的所谓 if（如果）转换，可以取消有条件的转移指令。

有关超块、其它指令块以及相关的主题可以参考：

－Wen-Mei W. Hwu 等人：“用于未来微处理器的编译技术”，IEEE 报告的特约文章，卷 83(12)，1995 年 12 月，微处理器特刊，第 1625 至 1640 页，

－Henk Neefs, Jan van Campenhout：“固定块长度结构的指令群结构所用的微结构”，第 8 届 IASTED 国际会议的关于并行及分散的计算和系统的报告，第 38～42 页，IASTED/ACTA 出版，1996，以及

－Richard H. Littin, J. A. David McWha, Murray W. Pearson, John G. Cleary：“基于块的执行和任务级并行性”，出自：John Morris(Ed.)，“计算机结构 98”，第三届大洋洲计算机结构会议的报告，ACAC'98，Perth, 1998, 2-3 月，澳大利亚计算机科学通信，卷 20，号 4，页 57～66，Springer，新加坡。

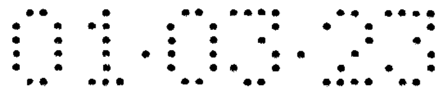
上文所述的无条件指令是指一些无条件地处理数据的指令，包括把数据从一个存储区拷贝到另一存储区（从一个寄存器拷贝到另一寄存器）。这种指令在下面被称为标准指令。它们包括用于得出新值的数据和用于拷贝寄存器内容的所谓Move（移动）指令之间的算术和逻辑连接。该指令的一般格式为：〈助记符〉〈目标寄存器〉,〈源寄存器1〉,〈源寄存器2〉。为了实现这种指令所规定的操作，通常需要硬件块的一个算术单元。

条件指令是指在出现一定条件时对数据进行处理的指令。由该指令执行的作用对应于由标准指令执行的作用，但其中相关作用的执行取决于预定的条件。如果满足条件，则执行由该指令规定的作用，否则不执行（于是相关的指令、譬如NOP指令起作用）。该指令在下文被称为条件指令。该指令的一般格式为：〈助记符〉p〈目标寄存器〉,〈源寄存器1〉,〈源寄存器2〉〈p标志〉，其中，由助记符结尾处的“p”发出在执行指令时对条件的依赖性的信令，而且所述条件由某一标志（譬如p标志）的某种状态来定义。为了实现由该指令规定的操作，通常需要硬件块的一个算术单元；为了对条件进行检验，需要一个比较单元，其输出端与所述算术单元的控制端相连。

谓词指令是指用于确定在条件指令中所采用的条件标志（例如p标志）的状态的指令。在此，所述的确定是在程序运行期间根据两个数据的比较来实现的。该指令在下文被称为pxx指令。该指令的一般格式：为pxx〈源寄存器1〉,〈源寄存器2〉,〈p标志〉,其中，xx规定了所述需要实现的比较操作，并用gt(大于)、ge(大于或等于)、eq(等于)、ne(不等于)、le(小于或等于)或lt(小于)来代替。该pxx指令可比作为普通的分支指令，并用来通过采用所谓的if(如果)转换(对此参见上述由Wen-Mei等人所著的论文)来替代所述的分支指令。

Loop指令是指位于超块结束处的用于循环重复的指令。如果满足指令中规定的条件，则由该指令促使跳回到相关超块的开始处；倘若不再满足该条件，则由它可以促使产生“就绪”信号。所述的条件是通过比较操作的某个结果来定义的。该指令的一般格式为:Loopxx〈源寄存器1〉,〈源寄存器2〉,其中，xx规定了需要执行的比较操作。

从上述指令类型的格式可以看出，总是采用寄存器作为数据源和/或信号源以及数据目标和/或信号目标。这在采用图12所示类型的硬



件块的情况下被证明是有利的，因为可以特别有效地访问所述的寄存器（寄存器文件 44）。但是从原理上讲，也可以允许指令的数据源和/或信号源以及数据目标和/或信号目标不是寄存器。

许多程序或至少该程序的大部分可以通过只采用所述的指令类型来进行描述，或者翻译成这种程序，因此这些程序可以全部放在图 12 所示类型的硬件块中执行。所以，在程控单元中使用这类硬件块可以大大提高其性能。但是，图 12 所示类型的硬件块也可以作为独立的设备而在程控单元之外使用，然后同样基于指令或指令流进行配置。

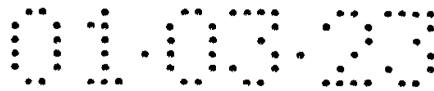
下面来讲述硬件块的配置，通过该配置可以执行由指令或指令序列预先规定的算术和/或逻辑操作或操作序列。

在本实施例中，所述的硬件块、准确地说是其子单元（算术单元，比较单元，多路转换器，多路分解器，...），以及各子单元之间的通信连接是通过表示所需配置的配置数据（配置比特）来配置的。据此，下面所讲述的配置方法的任务在于，根据硬件块配置所基于的指令或指令序列来生成或改变配置数据或包含该配置数据的比特流。

本实施例的出发点为只转换上述类型的指令，也即标准指令、条件指令、pxx 指令和 loopxx 指令；其它的指令必须在其它方面执行，譬如利用常规程控单元的执行单元来执行。

为了把可转换的指令转换成相应的硬件块结构，采取如下做法是比较有利的，即所述的指令总是具有图 1A（标准指令）、1B（条件指令）、1C（pxx 指令）及 1D（loopxx 指令）所示的示范格式，或者由解码器置成这样的格式。

尤其当硬件块的子单元可以被配置时，给这些（物理）子单元分配一些逻辑或虚拟的单元，其中，所述虚拟单元规定了物理子单元的不同功能。倘若物理子单元“第一算术单元 AU1”可以被配置，譬如就可给它分配一些虚拟单元，如加法器、减法器等等。一个虚拟单元被严格地分配给一个物理子单元，但一个物理子单元可以分配多个虚拟单元。优选地，所有的虚拟单元在一个表格或目录内进行管理。除了针对虚拟单元的信息外，相应的表格登录项还包括有如下有关信息，即各个虚拟单元被分配给哪个物理单元，以及为了给物理子单元赋予由虚拟单元表示的功能，必须通过哪些配置比特和必要时怎样来配置该物理子单元。



优选地，将整个超块转换成硬件块结构。

把指令转换成硬件块结构化信息主要分三个阶段来实现。

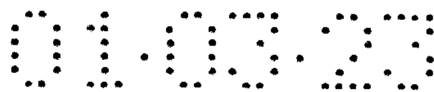
在第一阶段，首先测定需要哪种类型的虚拟单元（加法器，减法器，乘法器...）来执行需要转换的指令，以及该虚拟单元是否仍然可用。如果所需类型的虚拟单元是空闲的，则选出该虚拟单元或从中选出一个虚拟单元来执行相关的指令。然后实行配置或准备配置，并为选出的虚拟单元预备所分配的物理子单元。为了进行配置，简单地对分配给相关物理子单元的配置比特进行设置或复位；这是没有什么难度的，因为哪个物理单元分配了所选的虚拟单元、以及需要通过哪些配置比特和必要时需要怎样配置该物理子单元等信息是同所述的虚拟单元一起进行管理的。为选出的虚拟单元预备所分配的物理子单元是必要的，以便避免多次使用相关的物理子单元。在本实施例中，这是通过如下方式来实现的，即每次在为某个目的而分配给物理单元之后，便禁止相关物理子单元所分配的所有虚拟单元。

在 pxx 指令的情况下，可以按照硬件块的结构要求根据 p 标志选择一个完全确定的物理子单元(比较单元)。

在条件指令的情况下，如果某些指令只有利用某些标志才是可能的，也即在条件指令的分指令系统中没有完全的正交性，那么，p 标志只对虚拟/物理单元的选择起作用。

在硬件块配置的第二阶段，对前接和/或后接于所选的物理子单元之上的多路转换器进行配置，以便根据需转换的指令中的规定来调整数据源和/或信号源以及数据目标和/或信号目标。在理想情况下，所述多路转换器与需转换的指令的格式是相互匹配的，使得所述指令中对数据源和/或信号源以及数据目标和/或信号目标进行规定的部分可以不变地被接收作为配置该多路转换器的配置比特。如果这－出于一些原因总是这样－不可能或不理想，则对多路转换器进行配置的配置数据譬如可以从一个表格中查取，在该表格内存储了指令中用于对数据源和/或信号源以及数据目标和/或信号目标进行规定的部分同对多路转换器进行配置的配置比特之间的分配。优选地，为了建立通往某个数据源和/或信号源及/或通往某个数据目标和/或信号目标的通信连接，所需的配置对于所有多路转换器都是相同的。

倘若需执行的操作所基于的数据至少部分地由包含在指令代码中



的恒量组成，则需要进行特殊处理。于是，必须

-搜寻空闲的（恒量）寄存器，

-将该寄存器用作数据源和/或信号源，以及

-在硬件块投入运行之前，把在指令代码中所包含的恒量写到所选的寄存器中。

在本实施例中，预先检验所述的相关恒量是否已存储在（恒量）寄存器中。在此，如果结论是已存在包含该恒量的（恒量）寄存器，则将该已存在的（恒量）寄存器用作数据源和/或信号源。

另外，需要注意的是，所述需转换的指令具有许多不同的数据源和/或信号源以及数据目标和/或信号目标，而且/或者取决于多个条件，在这方面需要对单个指令进行特殊处理。

此外，用作数据目标和/或信号目标的寄存器被标记为已占用，因为在一个超块内不允许有第二次占用，而且必须通过所谓的（运行时间）寄存器重命名、也即一种从超标量结构得知的技术来进行避免。

在该（对所有指令通用的）第二阶段之后为各个指令类型加入特殊的分步骤，该分步骤是从相应的特殊性得出的。

另外，在条件指令的情况下，必须对检验是否存在条件的比较单元进行测定，并且通过所属的多路分解器将其输出信号接通到执行该操作的算术单元上。另外还需考虑该条件为何种类型。

在条件Move(移动)指令的情况下，还需负责使所述目标寄存器的内容在不执行指令时保持不变。

在所述硬件块配置的第二阶段之后，配置可以结束，并启动该硬件块。但是优选地，这只有在执行完下述第三阶段之后才施行。

在硬件块的该第三阶段中实现一种所谓的数据转发。在此，并不固执地采用指令中给出的数据源和/或信号源作为数据源和/或信号源，而是在可能范围内采用各个超块内的相关数据源和/或信号源已在其上预先写过的物理子单元。这在两方面被证明是有利的：一方面，由于可能需要较少的寄存器（如果不采用指令中给出的数据源和/或信号源作为数据源和/或信号源，则无须对其进行写操作，必要时可以完全将其取消），而另一方面，由于若从产生所需数据的子单元（譬如算术单元）读取该数据，则会比先将数据写入寄存器然后再从那儿读取要更早地使用该数据。该数据转发在所有指令中都可应用，而且通

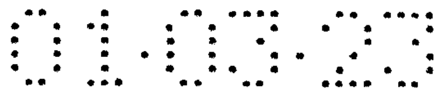
常被证明是巨大的优点。

上文简要讲述的方法也可以通过其软件及硬件的实现可能性、并用数学表示法来进行阐明。

首先来讲述采用类似于C++表示的软件实现。在本实施例中，针对硬件块配置数据的信息管理是通过分类来实现的。

在本实施例中，所述虚拟单元的类被定义如下:

```
class clVirtualUnit {
 private: unsigned int uiPhysicalPartNumber;
          unsigned int uiMnemonicType;
          BOOL bIsConfigurable;
          unsigned int uiConfBits;
          unsigned int uiConfBitsIndex;
          BOOL bTwoSourceRegs;
          unsigned int uiSrcMultiplexNumber[2];
          unsigned int uiSrcMultiplexIndex[2];
          BOOL bDestinationReg;
          unsigned int uiDestMultiplexNumber;
          unsigned int uiDestMultiplexIndex;
          BOOL bIsUsed;
          BOOL bSecondPIsused;
          BOOL bIsConstantRegiser;
          unsigned int uiConstantIndex;
          unsigned int uiConstantValue;

 public:  unsigned int uiGetPartNumber(void);
          unsigned int uiGetMnemonicType(void);
          BOOL bIsUnitConfigurable(void);
          unsigned int uiGetConfBits(void);
          unsigned int uiGetConfBitsIndex(void);
          BOOL bHasTwoSourceRegs(void);
          unsigned int uiGetSrcMultiplexNumber
                                   (unsigned int);
```



```
unsigned int uiGetSrcMultiplexIndex
                              (unsigned int);
BOOL bHasDestinationReg(void);
unsigned int uiGetDestMultiplexNumber(void);
unsigned int uiGetDestMultiplexIndex(void);
void vFreePart(void);
BOOL bMarkUsedPart(void);
BOOL bMarkSecondUsedFlag(void);
BOOL bGetIsUsed(void);
BOOL bGetIsUsedSecondFlag(void);
BOOL bIsConstantRegister(void);
BOOL bSetConstantValue(void);
unsigned int uiGetConstantValue(void);
unsigned int uiGetConstantIndex(void);
    }
```

在该类中所包含的数据或方法被用来模拟一种宏结构。

这些数据的意义为：

uiPhysicalPartNumber：该变量包含有硬件块内物理子单元的单值号码。

uiMnemonicType：该变量包含有编码形式的、属于相应虚拟单元的逻辑连接类型。

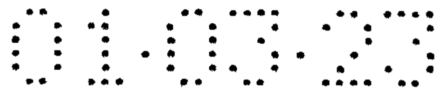
bIsConfigurable：该标志指示出是否必须配置所属的物理子单元，以便获得该虚拟单元。

uiConfBits：若bIsConfigurable==真，则在此存储所属的配置比特，以便正确地为该功能配置物理子单元。

uiConfBitsIndex：若bIsConfigurable==真，则在该位置存储用于在比特流中存储配置比特的指数。

bTwoSourceRegs：如果必须为相关的指令给出两个源寄存器，则该标志被置为“真”，否则被置为“假”。

uiSrcMultiplexNumber[2]：在两个源寄存器的情况下，在该变量中存储所属的多路转换器的物理号码，必要时只有具有指数0的变量有效。



uiSrcMultiplexIndex[2]：在此存储了用于源寄存器的多路转换器的指数。

bDestinationReg：如果必须为相关的指令给出目标寄存器（不是标志！），则该标志被置为“真”，否则被置为“假”。

5 uiDestMultiplexNumber：在此为目标寄存器存储所属的多路转换器的物理号码。

uiDestMultiplexIndex：在此为目标寄存器存储多路转换器的指数。

bIsUsed：在该标志内存储该虚拟子单元（以及由此同时还有物理

10 子单元）是否已被使用。该标志被置为“真”时，意味着该子单元不可再使用（在条件 Move 指令(movep)情况下除外）。

bSecondPIsUsed：对于 movep 指令的特殊情况，在该标志中存储了包括比较在内的 p 标志的两种使用。如果 bIsUsed 和 bSecondPIsUsed 被置为“真”，则禁止进一步使用把 movep 指令施加

15 到其上的动态多路器(AU)。

bIsConstantRegiser：该标志指示出物理子单元是对应于一个恒量寄存器(真)或不是这样(假)。

uiConstantIndex：在恒量寄存器的情况下，必须将所述可以存储和使用的恒量的值登录到比特流中。在该情形下，在此变量内存储位

20 于比特流之中的指数。

uiConstantValue：在恒量寄存器内存储的值被附加地存储在所述级的该变量之中，以便用于进一步比较。

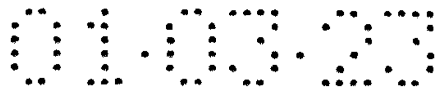
在该类的一个级内所出现的变量必须在启动配置的时间点上全部被占用。对此采用了此处没有明确列出的方法，而该方法被用在下文

25 阐述的可配置块分类或者说是 CB 分类的结构当中，以便把转换所需的全部信息写入到所述的级中，并同时把标志 bIsUsed 和 bSecondPIsUsed 置为“假”。于是在该级的服务时间内，只有两个通过预定的方法可赋予值“真”或“假”的标志发生变化，以及－在恒量寄存器的情况下－还有变量 uiConstantValue，该变量中临时存储

30 了用于进一步比较的寄存器的当前值。

上述定义的类中的各方法对虚拟单元意味着：

unsigned int uiGetPartNumber(void)：该方法允许对属于虚拟



子单元的物理子单元的号码进行读访问；该号码作为返回值返回。

unsigned int uiGetMnemonicType(void)：该方法允许对可以在虚拟单元中实现的助记符进行读访问。

BOOL bIsUnitConfigurable(void)：若必须对物理子单元进行配置，则该方法提供“真”。在该情况下，uiConfiBits 和 uiConfBitslndex 中的登录项有效，而且可利用下面的方法获得 uiGetConfBits( )和 uiGetConfBitsIndex()。另外，属于同一物理子单元的所有其它虚拟子单元同样必须被禁止。相反，对于返回值“假”的情况，虚拟单元和物理单元是恒等的。

unsigned int uiGetConfBits(void)：利用该方法读出配置比特，并作为返回值返回。该值只有在 bIsConfigurable 的值为“真”时才有效。

unsigned int uiGetConfBitsIndex(void)：利用该方法读出配置比特在比特流中的指数，并作为返回值返回。该值只有在 bIsConfigurable 的值为“真”时才有效。

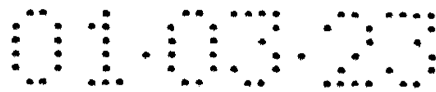
BOOL bHasTwoSourceRegs(void)：如果调用该方法的操作占用两个源寄存器，则该调用提供值“真”，并将其登录到相应的多路转换器中，否则提供值“假”。

unsigned int uiGetSrcMultiplexNumber(unsigned int)：该方法提供为源寄存器描述多路转换器的物理子单元的号码。调用参数为项数为 2 的数组中的指数，其中，如果标志 bHasTwoSourceRegs 的值为“真”，则指数 1 只提供有效值。

unsigned int uiGetSrcMultiplexIndex(unsigned int)：该方法提供用于登录到比特流中的指数，以便为源寄存器配置多路转换器。调用参数为项数为 2 的数组中的指数，其中，如果标志 bHasTwoSourceRegs 的值为“真”，则指数 1 只提供有效值。

BOOL bHasDestinationReg(void)：如果调用该方法的操作占用一个目标寄存器，则该调用提供值“真”，并将其登录到相应的多路转换器中，否则提供值“假”。

unsigned int uiGetDestMultiplexNumber(void)：该方法提供为目标寄存器描述多路转换器的物理子单元的号码。返回值只有在标志 bHasDestinationReg 的值为“真”时才有效。



unsigned int uiGetDestMultiplexIndex(void)：该方法提供用于登录到比特流中的指数，以便为目标寄存器配置多路转换器。返回值只有在标志 bHasDestinationReg 的值为“真”时才有效。

void vFreePart(void)：该方法通过将所有的占用标志置为“假”
5 值而清空所有的占用标志。

BOOL bMarkUsedPart(void)：通过该方法将占用标志 bIsUsed 置为“真”。若操作成功，则返回值为“真”，若该元素已被占用，则返回值为“假”。

BOOL bMarkSecondUsedFlag(void)：把第二个占用标志
10 bSecondPIsUsed 相应地置为“真”。若操作成功，则此处的返回值也为“真”，若该元素已被占用，则返回值为“假”。

BOOL bGetIsUsed(void)：该方法以返回值的形式提供变量 bIsUsed 的值。

BOOL bGetIsUsedSecondFlag(void)：该方法以返回值的形式提
15 供变量 bSecondPIsUsed 的值。

BOOL bIsConstantRegister(void)：若虚拟子单元对应于一个恒量寄存器，则该方法返回“真”，否则返回“假”。

BOOL bSetConstantValue(void)：如果所述的虚拟单元对应于一个恒量寄存器，且该寄存器此时还未被占用，则可以利用该方法把当
20 前的恒量值存储到变量 uiConstantValue 之中。

unsigned int uiGetConstantValue(void)：借助该方法返回所存储的恒量值。

unsigned int uiGetConstantIndex(void)：通过该方法获得为在此时存储所述恒量值而需要的比特流中的指数。

25 为了模拟硬件块(CB)还定义了一个第二类，该类包含有类 clVirtualUnit 的级和其它的变量及方法。为简化起见，假定元素的存储是在一个静态数组内进行的；当然也可以构想为一个链表。此处应注意的是，只为此处列出的分类讲述了一部分方法。

30
```
Class clCB
{
private: BITFIELD *clbitfield;
```

```
        class clVirtualUnit clArrayVirtualUnits
                                          [NUM_OF_PARTS];
public: clCB( );
        void vSetupBitfield(BITFIELD *);
5       void vFreeAll(void);
        BOOL bDoAllPhase_1_Parts
                    (unsigned int,BITFIELD *)
        BOOL bDoCommonPhase_2_Parts(unsigned int,
                                    BITFIELD *,
10                                  unsigned int,
                                    unsigned int,
                                    unsigned int);
        void vDataForwarding(unsigned int,unsigned int);
        void vCopyBitfield(BITFIELD *,BITFIELD *);
15      unsigned int uiGetMuxCode(unsigned int,
                                  unsigned int);
        unsigned int uiGetRegPartNumFromCode
                                  (unsigned int);
        unsinged int uiGetPartNumFromFlag
20                                (unsinged int);
        unsigned int uiGetIndexFromNum(unsigned int);
        unsigned int uiGetPartNumFromBitfield
                                  (unsigned int);
        void vSetBitfield(unsigned int,unsigned int,
25                                  unsigned int);
    };
```

所述类 clCB 的变量和方法的详细意义为:

BITFIELD *clBitfield: 该变量对应于用于 CB 的运行时间配置的、需生成的比特流。

30 class clVirtualUnit clArrayVirtualUnits[NUM_OF_PARTS]: 该类 clVirtualUnit 中数组形式的级内包含所有虚拟单元的所有信息, 并由此也包含所有物理子单元的所有信息。

clCB( ): 该结构被列出用来阐明所述类的任务是什么。在开始阶段，必须对比特区和类 clVirtualUnit 的包括在数组 clArrayVirtualUnits[ ]之内的所有级进行初始化。为了初始化类的各个级，尤其还包括对所有配置数据进行说明和对所有标志进行复位，以便在工作期间可以对必要的数据进行读访问。

void vSetupBitfield(BITFIELD *): 在该方法中给比特区提供所有的预占用。

void vFreeAll(void): 该方法被调用来清空数组 clArrayVirtualUnits[ ]的所有占用标志。

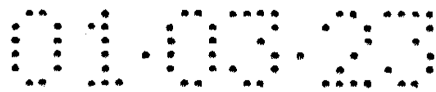
BOOL bDoAllPhase_1_Parts(unsigned int,BITFIELD *): 在该方法中综合了阶段 1 的所有部分。该方法在找到用于接收助记符的空闲子单元之后被调用，而且还包括将所有所属的虚拟单元标记为已占用、确定配置比特及在比特流中的指数、以及登录到暂时比特流。参数是虚拟单元数组中的指数和指向暂时比特流的指针。返回值“真”指示出阶段 1 是成功的，而“假”则指示失败（可能由于不足的网络资源）。

BOOL bDoCommonPhase_2_Parts(unsigned int, BITFIELD *, unsigned int, unsigned int, unsigned int): 该方法综合了所有指令组共用的方法。在此，还包括源寄存器和目标寄存器的登录，并把恒量看作为输入值。返回值“真”指示出阶段 1 是成功的，而“假”则指示失败。

void vDataForwarding(unsigned int,unsigned int): 在该方法中集成了数据转发的计算和所有所属的方法。该优选方案涉及到源寄存器，该源寄存器的物理号是在所述的参数中传输的。利用其它的方法来测定源寄存器是否为先前的目标寄存器。如果是，则从比特流中测出最后进行计算的 AU，并用其代替该寄存器进行登录。

void vCopyBitfield(BITFIELD *,BITFIELD *): 该方法用“或”把第二比特流中的登录项和第一比特流中的登录项连接起来，并将结果存储在第一比特流中。由此将所述暂时的中间结果存储在为稍后配置而计算的比特流中。

unsigned int uiGetMuxCode(unsigned int,unsigned int): 该方法计算出需要为多路转换器装入到比特流中的配置比特，以便以信



源的形式选出一个物理子单元。该方法的参数是所述多路转换器和信源单元的物理号。该方法对于配置是绝对必要的，因为在描述虚拟单元时存储的是登录项的指数，而不是登录项本身。对于整个网络而言，该方法可以以表格支持的换算形式、且必要时无需考虑多路转换器来实现，其原因是，在该情形下所有的多路转换器都是以统一的方式进行配置的。对部分网络而言，此时必须要有较高的费用，特别是不能实现结网。在成功的情况下返回值为所述的配置比特，而在不成功时为不可用的编码。

unsigned int uiGetRegPartNumFromCode(unsigned int)：该方法从指令的代码中计算出子单元的号码。当然这可以只为寄存器实现，其中在恒量的情况下，可在该方法中集成已讲述过的优选方案，由它来促使存储恒量和返回恒量寄存器的物理号。在成功情况下，返回值为子单元的号码，否则在不成功时的返回值为不采用的标识。

unsinged int uiGetPartNumFromFlag(unsinged int)：该方法负责把标志号换算成物理子单元的号码。调用参数是指令格式中的 p 区，返回值为所述的子单元号，或在失败时为一个特殊标识。

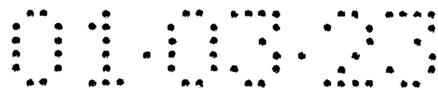
unsigned int uiGetIndexFromNum(unsigned int)：利用该方法（以参数的形式）为具有已知物理号的子单元计算和返回在比特流中的指数。该计算可以用表格的形式实现。

unsigned int uiGetPartNumFromBitfield(unsigned int)：该方法读出以参数形式传输的指数处的比特区中的登录项，并将获得的配置掩码换算成所述子单元的物理号，并将其作为结果返回。UiGetPartNumFromBitfield 被应用在数据转发之中，在此，把数据路径从先前的目标寄存器挪回到确定结果的子单元，由此可以提前使用该数据。

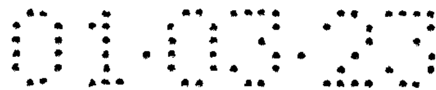
void vSetBitfield(unsigned int,unsigned int,unsigned int)：该方法利用三个参数进行调用：登录项的指数、登录项的长度及掩码。该调用对比特区相应位置的登录项起作用。

利用上面提到和讲述的变量和方法，可以为基于指令或指令序列的、图 12 所示类型（用于结构过程的编程）硬件块的配置方法提供下述的伪代码：

```
unsigned int k;
```



```
BITFIELD  *clTempBitfield;

//阶段 1: 确定用于接收逻辑连接的物理子单元.
//助记符位于 uiMem 之中

vSetupBitfield(clTempBitfield);

for(k=0;k<NUM_OF_PARTS;k++)
{
   if(clArrayVirtualUnits[k]::uiGetMnenomic( )==uiMem
      && clArrayVitualUnit[k]::bGetIsUsed( )==假)
   break;
}
if(k==NUM_OF_PARTS)//没有找到空闲的逻辑连接
  return(假);

//此时将该空闲的子单元标记为已占用，必要时还对配置进行确
//定，并在该情况下将其它所有的虚拟子单元也标为已占用. 将
//所有的掩码比特存储在暂时的比特流中.

If(bDoAllPhase_1_Parts(k,clTempBitfield)==假)
return(假);

//现在开始阶段 2: 为所有指令确定两个、必要时为一个源寄存
//器，并将其登录到比特流中. 如果有的话，相应地利用目标寄
//存器来实现. 指令中的相应编码在于变量 uiSourceReg1、
//uiSourceReg2 和 uiDestReg，其中，有时也可以在此把恒量识
//别为信源.

If(bDoPhase_2_CommonParts(k,clTempBitfield uiSourceReg1,
   uiSourceReg2,uiDestReg==假)
 return(假);
```



```
switch(uiMnemonicType)
{
 case BEDINGTER_BEFEHL      //确定p标志，为CU登录
 case MOVEP_BEFEHL:         //特别是第一登录
                            //也可以是第二登录
}

vDoDataForwarding (uiSourceReg1,uiSourceReg2);

//最后的作用：将所述暂时存储的比特流代码拷贝到原来的比特率中

vCopyBitfield(clBitfield,clTempBitfield);
return(真);
```

所述中央程序是为每个可翻译的指令进行调用的。若转换成功则返回值为“真”，否则为“假”。在后一种情形下，所述的指令必须保留在调用的程序内，因为它不是被插入的，而且可以装载用于执行的比特流。于是，通过资源的用尽来指示转换的结束，或通过一个不可翻译的、诸如分支指令的指令来获得。

正如上文所述，所述结构过程的编程既可以用软件方式实现，也可以用硬件方式来实现。下面参照图 2~10 来阐述一种按硬件实现的可能实施方案。在此，尝试使各个阶段尽可能并行地工作。

在软件实现方式中所进行的由表格支持的换算在硬件实现方式中是以所谓的查询表（LUT）来实现的。LUT 被设计用来为响应输入的数据而输出与该数据有关的数据。这种 LUT 譬如可以通过 EPROM 或其它的存储装置来构成。于是，输入的数据作为地址使用，而输出的数据是通过该地址进行存储的数据。

为阶段 1 采用如图 2 所示类型的 LUT。该 LUT 具有两个输入端(“地址”，“计数器地址”）和四个输出端（“代码”，“互补”，“增加计数”，“无入口”）。所述的两个输入端被用作 LUT 的寻址，其

中通过输入端（“地址”）输入的数据和/或信号取决于需翻译的代码，而通过另一输入端（“计数器地址”）输入的数据和/或信号为可通过输出端“增加计数”进行增加计数的计数器（计数器阵列）的计数状态。所述的输出端用于：输出翻译过的代码；输出信号以便增加生成“计数器地址”的计数器或计数器阵列的计数；如果不再有有效和空闲的登录项（“无入口”），便发出信号；以及输出为处理条件 Move 指令（movep）所需要的信号（“互补”），其中，翻译过的代码由配置比特、配置指数及子号组成。因此对于第一部分，查询表的登录项具有图 3 所示的格式。

所述的计数器（计数器阵列）是以标记装置（占用，写）的形式进行使用的，其中每个操作类型（加，减...）都有一个单独的计数器。所述计数器的状态给出了可以采取多少种可能性作为应答来执行所分配的操作类型。计数器阵列内的计数器的深度(Tiefe)取决于执行相关操作的可能性的数目。譬如，若存在三种加法可能性，则计数器的深度为 2 比特；但是，在由助记码和计数状态进行寻址的相应 LUT 中，第 4 个位置（计数状态 3）将为一个“无入口”编码，以便指示该操作的故障；这种 LUT 登录项如图 3B 所示。

在本实施例中，所说的计数器为具有异步复位和允许的二进制计数器。该类型的二进制计数器在本实施例中是按如下方式进行编码的；该表示是利用 DNF（析取范式）逻辑中所惯用的 DNF 格式来实现的。该类型的计数器在下文被称为第一类型的计数器。

```
BIT   b0,b1:OUT;
BIT   reset,enable:IN;
BIT   clock:IN;

b0=/b0*enable+b0*/enable;
b0.clk=clock;
bo.areset=reset;

b1=/b1*b0*enable+b1*/b0*enable+b1*/enable;
b1.clk=clock;
```

```
b1.areset=reset;
```

必须为条件指令实现一种与该计数器并行的存储器阵列，以便存储条件标志的代码。正如上文所述，这对于组成 Movep 指令是必须的。由于每个标志只能给出一个 CU 级（与 AU 相反，通常有多个标志，但所有这些标志都是用比特名称来区别的），因此所述的二进制计数器由两个比特组成，其中第一比特指示出第一占用，而第二比特则指示出互补占用。借助指令中的 p 标志来识别正确的 CU。

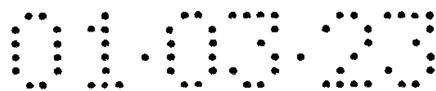
在本实施例中，用于条件指令的 2 比特二进制计数器是按如下方式进行编码的；该表示也是利用 DNF（析取范式）逻辑中所惯用的 DNF 格式来实现的。该类型的计数器在下文被称为第二类型的计数器。

```
BIT   b0,b1:OUT;
BIT   reset,enable:IN;
BIT   clock:IN;

b0=/b0*enable+b0;
b0.clk=clock;
bo.areset=reset;

b1=/b1*b0*enable+b1;
b1.clk=clock;
b1.areset=reset;
```

对于如下情况，即必须采取判断、且该判断需要在数据回路中进行转换，那么就为此集成一种特殊的逻辑。

于是，硬件块结构化的方法的第 1 阶段实现如图 4 所示。

对于除条件 Movep 指令外的所有指令，每个算术单元 AU 或每个比较单元 CU 都需要一个遵照上述第一类型计数器的类型的计数器级。这种计数器是可以满足要求的，因为只需要一个简单的占用信号。相反，Movep 语句需要有第二类型的计数器，该计数器在两个比特内发送部分(b0)和完全(b1)占用的信令。在第一源寄存器保持不变期间，两次



参考相同标志的条件 Movep 指令必须以倒置的形式（相对于第一参考）处理该标志，然后在相应的 AU 内被登录为第二信源。该方法可集成在 LUT 内；对非倒置的条件的参考由“无入口”信令来中断。

第 2 阶段包括对一些寄存器的确定，所述寄存器可以用作相关操作的数据源和/或信号源以及数据目标和/或信号目标。对于所有三种可能的寄存器，这是以并行的、且在很大程度上相同的形式来实现的。指令内相应寄存器的编码－如果相关区包含有效的登录项－是通过查询表被转换成掩码的，以便用于比特流和比特流中的指数。

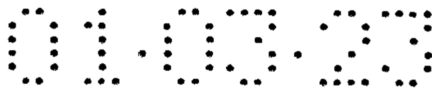
图 5 示出了对用作数据源和/或信号源以及数据目标和/或信号目标的寄存器进行确定和编码的电路图；在本实施例中，通过控制线“源寄存器 1”、“源寄存器 2”以及“目标寄存器”来识别实际（必须）转换哪些寄存器（参考图 4 和 5）。

源寄存器和目标寄存器的处理是不同的。在目标寄存器的情况下对登录项进行标记，以便能够识别第二占用（信号“无入口”）和触发数据的转发。这些信号对源寄存器是不采用的。在此，为比特流的登录项实现一种“直接”生成，但在恒量的情况下不生成该代码，并转移到下文所述的级上。

在图 5 中标记了哪些只对源寄存器和只对目标寄存器相关：用(*)标示的部分被确定只用于目标寄存器，而用(**)标示的部分被确定只用于源寄存器。

对于可能代替源寄存器而出现在编码内部的恒量，为其实现一个并行的方法，该方法将所述的恒量并行地与所有恒量寄存器内的内容进行比较，而且在不相同的情况下用该恒量占用下一空闲的寄存器（利用计数器实行指针管理），并且以编码的形式返回该寄存器，或者在相同的情况下以编码的形式返回包含该恒量的恒量寄存器的编码。

为此，所述的查询表作如此设计，使得在比较为肯定的情况下直接把相关寄存器的编码数字提供给比特区，而在不一致的情况下另外还将该恒量存储起来，并增加寄存器计数。“无入口”信号对于所有恒量已占用的情况是有效的，而且由它结束指令块的算法，因为资源已用尽。同时，需要注意的是，所述的恒量寄存器为（主）比特流的一部分，原因是该恒量寄存器在前面的周期内可能已被占用，并需要被用来装载指令块。



用于分配所述恒量寄存器的电路框图如附图6所示。

为源寄存器实行上文已多次讲述的数据转发。借助位于寄存器的占用标志内的、指示在该周期内该寄存器已是目标寄存器的登录项来判断：是把实际的源寄存器、还是把可以以目标寄存器的信源形式被测定的登录项作为新的信源而登录到比特流中。

对此适用的电路框图如图7所示。

如果网络内的所有信源都采取相同的编码，则在图7中可以取消通过LUT实现对新的信源进行再编码。可以为整个网络假定的这种情况将会导致：暂时比特流中用于（先前的）目标寄存器的信源登录项将作为用于当前操作的新信源编码而被登录，以代替在指令中被编码的源寄存器。在任何情况下，该选择都是通过由信号“数据转发”（参见图5）进行控制的多路转换器来实现的。

倘若所有的操作都成功（这可通过不出现“无入口”信令来识别），则在写脉冲期间将暂时比特流与现有的主比特流进行“或”逻辑连接，然后写回到该比特流中。

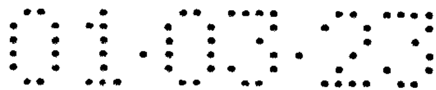
图8和9示出了把配置数据写入到暂时比特流和写入到主比特流中的框图。

从图8可以看出，将配置数据写入到暂时比特流中是通过所谓的纵横开关来实现的。纵横开关是大家都已知的，在此无需赘述。它把配置比特传送到暂时比特流中由配置指数所定义的位置，其中，所述纵横开关未被占用的输出端被置为预定的值（譬如“0”）。为了基于助记符对物理子单元进行选择、以及为了配置该物理子单元并把源寄存器和目标寄存器分配给该物理子单元，总是需要一个固有的、如图8所示的纵横开关。

把暂时比特流转换成主比特流（主比特流的叠加是通过纵横开关的输出端）是通过主比特流的输入端处的“或”门OR实现的（参见图9）。

上述的各部分可以按照图10组合成一种装置，该装置能够把由m比特、ks1比特、ks2比特、kd比特和p比特组成的指令（参见图1A至1D）转换成用于配置硬件块的配置数据，并将该数据写入到为配置所述硬件块而采用的比特流中。

最后，还是以数学表示法来给出上文所致力于的转换（结构过程

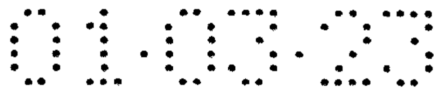


的编程）。

对此，首先必须达成一系列关于表示和映射的协定。它们为：

I^+	所有的指令群
I	所有与数据流相关的（适用于块的执行的）指令群
SR	包括 NO-SOURCE 表示在内的所有源寄存器群，恒量寄存器除外
CR	包括 NO_CONST 和 IS_CONST 的表示在内的所有恒量寄存器群
SR^+	$SR \cup CR$
DR	包括 NO_DEST 表示在内的所有目标寄存器群，谓词比特除外
PR	包括 NO_PRED 在内的所有谓词比特群
DR^+	$DR \cup PR$
RN	所有的寄存器群，$SR \cup CR \cup DR$
RN^+	包括谓词比特在内的所有寄存器群，$RN \cup PR$
List(pp)	4 元组形式(px$\in$PP，offset(偏移)$<k$，nbit$<k$，bitwert(比特值)$<2^k-1$)的比特流 B 的所有可能值群，必要时取决于 pp$\in$PP
Nbit	可能的比特值群(在 n 比特的数据宽度时：$0...2^n-1$)
B	用于配置所述结构且为比特流形式的、k 个二进制值的登录项群
OCC	所有的占用标记群（FREE(空闲)、WRITE(写)、READ(读)、READ_WRITE(读写)）
PP	所有的物理子单元群
PLNUM	逻辑单元的所有单值号码群
PL	CB 内的所有逻辑子单元群，由 11 元组构成（pl$\in I \cup RN^+$，plnum$\in$PLNUM，pp$\in$PP，occ$\in$OCC，source(信源)$\in$PLNUM，val$\in$Nbit，pbit$\in$PR，List(pp)，konfOffset$\leqslant k$，konfAnzahl$<k$，konfWert$<2^k-1$)

在下面的说明中利用了一些基本假设和功能，现首先来讲述它们。分量 occ(代表“现象”)内的标识可以选择四种值，用以标明状态‘未占用’（FREE）、‘读占用’（READ）、‘写占用’（WRITE）和‘读



写占用'（READ_WRITE）。在此，标识'读占用'在必要时不作进一步分析，但在本说明书内仍继续进行。

另外，为 RN 内的寄存器假定：对于这些子单元，所述的逻辑和物理表示是一致的。这意味着，与有些功能子单元（譬如表示为两个逻辑单元的可配置的加法/减法单元，但当然只可一次占用）相反，对寄存器是无需进行配置的，而且在寄存器号 rn∈RN 和逻辑子单元号 plnum∈PLNUM 之间存在一种注入式（但还不是双注入）的映射关系，该映射在下文用 rn2plnum( )来标示。该假定不适合目标比特形式的谓词指令。

在该前提条件下，将指令转换成用于对硬件块进行结构化的结构信息的过程如下：

1.在第 1 阶段，把包括所有由原始二进制格式组成的操作数在内的每个指令转化成说明 bi=(i∈I，sr1∈SR，cr1∈CR，n1∈Nbit，sr2∈SR，cr2∈CR，n2∈Nbit，dr∈DR，pr_source∈PR，pr_dest∈PR)。在该说明中，为恒量登录用于 cr1 或 cr2 的标识 IS_CONST，并以 n1/n2 登录该恒量值，其中在该情形下，相应的源寄存器获得标识 NO_SOURCE。相应地，为用于 dr 的谓词指令(譬如 pge...)采用 NO_DEST，而 pr_dest 由此便具有谓词比特的号。

为了改善可区别性，为谓词指令（譬如 movep）不把 pr_dest 设成相应的值，而是把 pr_source 设置成相应的值。

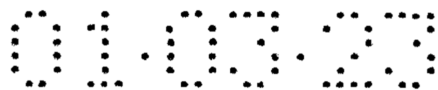
由具有 j∉I 的指令引发该转换的结束。

2. 在第 2 阶段，把 bi 中最大的五个登录项翻译成配置。之所以采取五个，是因为一些组合是相互排斥的。对此，为各部分区分如下：

为指令 bi→i∈I 和 bi→pr_dest==NO_PRED(无谓词语句)搜索第一元素 pl∈PL，该元素覆盖了该 i 并具有 occ==FREE。如果没有找到，则终止转换。

如果找到 pl，则用 occ==READ_WRITE 占用 PL 中映射同一物理元素 pp∈PP 的所有元素。pl 的配置借助所述元组中的信息而被登录在比特流 B 中。

若 bi→pr_source==NO_PRED，则为此不执行登录。否则按 p2∈PL 并利用 p2→pbit==bi→pr_source 进行搜索，其中必须 p2→occ==WRITE。对于该 p2，通过 List(p2→pp)来搜索 pl，并登录在所



述的比特流中，此外还把 p2→occ 置为 READ_WRITE。

为指令 bi→i∈I 和 bi→pr_dest！=NO_PRED(谓词语句)搜索第一元素 pl∈PL，该元素覆盖了该 i 并具有 occ==FREE。如果没有找到，则终止转换。

5 如果找到 pl，则用 occ==WRITE 占用 PL 中映射同一物理元素 pp∈PP 的所有元素。pl 的配置借助所述元组中的信息而被登录在比特流 B 中。

所有的指令 i∈I 都适用：对于适用！=NO_SOURCE 的 bi→sr1 和 bi→sr2，如果对属于 sr1/2 的 pl1/12∈PL、pl1→occ==FREE 及 pl2→

10 occ==FREE 合适的话，则通过 List(pl→pp)把相应的配置装入到比特流 B 中，同时在 pl1/12→source 时登录 pl→plnum(用于稍后的转发)。如果不是这种情况，则执行第 3 阶段（数据转发）。

对于源寄存器 bi→sr1 和 bi→sr2，如果它们！=NO_SOURCE，则在 PL 中把所属的 p31 及 p32∈PL（通过给定的功能 rn2plnum 获得）的

15 相应登录项 p31→occ 和 p32→occ 置为 READ(若它们事先！= WRITE 及 READ_WRITE)，否则便置为 READ_WRITE。

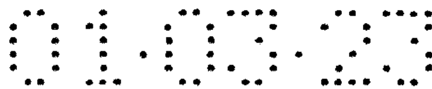
对于恒量寄存器 cr1 和 cr2，如果它们！=NO_CONST，则首先为所有的 p3∈PL 检验是否 p3→pp∈CR、p3→occ==READ_WRITE、以及 p3→val==bi→n1/2。如果是这种情况，则根据用于源寄存器的相应方法来

20 为 p3 实现登录。

如果该搜索没有成功，则必须搜索适合于 p4→pp∈CR 和 p4→occ==FREE 的 p3∈PL。假如找到，则把 bi→n1/2 登录到 p4→val，并设置 p4→occ=READ_WRITE，而且譬如在源寄存器中继续进行该登录。若搜索失败，便终止该转换。

25 为目标寄存器 dr 检验：对具有 p5→pp==dr 的相应登录项 p5 是否适合条件 p5→occ==FREE 或 READ。如果不是这种情况，则终止该转换，否则设置 p5→occ==WRITE 或 READ_WRITE，并将 List(p5→pp)中的登录项传输到比特流 B 中。为可能的数据转发而登录 p5→source=pl(赋值指令的逻辑元素)。

30 3. 对于在第 2 阶段具有所属元素 p6∈PL 的值、也即值 p6→occ==WRITE 或 READ_WRITE 的所有源寄存器 sr∈SR，通过如下方式为其实现数据转发，即在比特流 B 中不登录 List(p6)的值，而是登录



List(p6→source)的值。

通过上述的配置数据生成，可以在可配置的硬件块中执行标准程序的指令，所述的标准程序系指被设计在按 von-Neumann 原理进行工作的程控单元（常规的微处理器、微控制器）中执行的程序。

5 在此，所述转换的方法和方式可以实现：把指令转换成配置数据和通过采用该配置数据进行硬件块的配置都可以非常快地、简单而有效地实现。

通过所述的把指令转换成配置数据，可以产生一系列配置数据组，其中

10 －一方面，总是把尽可能多的指令转换成一个配置数据组，而且

－另一方面，总是只把如下数目的指令转换成一个配置数据组，即在硬件块（再）配置的时间点上可供使用的硬件块资源能够执行该数目的指令。

因此，在硬件块的再配置数目极小、且在使用配置数据前没有复

15 杂和有误差的检验的同时，可以确保最有效地使用硬件块。

如果使用的是本申请附图 12 所示类型的硬件块，且该硬件块可以通过相应的配置数据组分别进行复杂的再配置，则这在很大程度内都是适合的（但无疑不是唯一的）。

对于通过使用按权利要求所产生的配置数据组而配置的这种类型

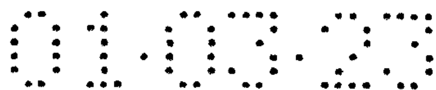
20 的硬件块，它可以并行地执行被转换成相关配置数据组的指令。如果所述的硬件块已经执行完被转换成相关配置数据组的指令，硬件块便优选地发出信号（譬如通过上述的“就绪”信号或利用其它的方式及方法），而且通过采用接下来的配置数据组对该硬件块进行再配置，由此在该硬件块内执行接下来的指令（为执行这些指令而需要执行的

25 操作）。被用来实现再配置的该接下来的配置数据组是从上述或类似实现的、对接下来的指令进行转换中产生的。如果执行这些该接下来的指令，则重新配置该硬件块。同时重复上述的过程。

利用这种方式，可以在可配置的硬件块内快速（特别是由于至少部分地并行执行指令而大大快于在常规程控单元内的情况）而简单地

30 执行如下的程序，即该程序是被设计用来在按 von-Neumann 原理工作的程控单元内执行的。

在上述把指令转换成配置数据的转换中，按超块方式进行转换被



视为力图争取的目标，也即在该转换中恰好把一个超块被转换成一个配置数据单元。

当然，如果能把多于一个的超块转换成配置数据组则会更好；这样总能够设定同时执行最大数目的指令。在这某些条件下甚至是可能的。

在图 12 所示类型的硬件块中（在能够执行上述谓词指令的硬件块中），尤其可以把超块序列 0、1 及 2

```
Hyperblock_0;
If(codition)
  Hyperblock_1;
else
  Hyperblock_2;
```

转换成一个单独的配置数据组。这是通过上述的 if 转换来实现的。在此，所示的条件可以转换为一个 p 标志，而且根据该条件所执行的指令在执行时与该 p 标志的值有关。在此譬如可以规定，如果 p 标志的值为 1，则执行在 if 条件转移（Hyperblock_1）内所包含的指令，而如果 p 标志的反值为 1，则执行在 else 条件转移（Hyperblock_2）内所包含的指令。于是，由三个超块 0、1 及 2 可以形成一个包含该超块的伪超块，该伪超块可被转换成一个单独的配置数据组。

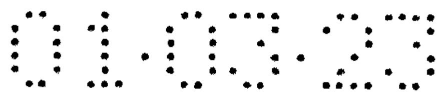
这种伪超块自身又可以包含一个或多个伪超块。此处的例子为如下序列：

```
Hyperblock_0;
If(codition)
  Pseudo-Hyperblock_1;
else
  Pseudo-Hyperblock_2;
```

在该情形下可以构成一个伪超块，且该伪超块包含有超块 0 和伪超块 1 及 2。

因此，在把指令转换成配置数据时，根据可能性在第一步骤中尝试构成伪超块。同时需要为此检查是否可建立伪超块，并在被考虑用来构成伪超块的程序部分中实现一个 if 转换。

在某些情况下，特别是当通过可配置的硬件块“只”应实现某一



个电路（譬如一个串行接口）时，采取如下做法被证明是比较有利的，即通过采取一种用譬如 VHDL 等电路说明语言实现的电路定义来配置硬件块。对此，首先用一种电路说明语言对想要通过硬件块实现的电路进行定义，然后把在此获得的代码转换成须被用来配置所述硬件块的配置数据（配置数据组或配置数据组序列），由此使该硬件块与利用它所实现的电路相一致。在此，该硬件块优选地作如此构成，使得利用它可以按照配置实现不同的电路，以及/或者可以象上文所讲述的那样能够在该硬件块内执行被转换成配置数据的指令。

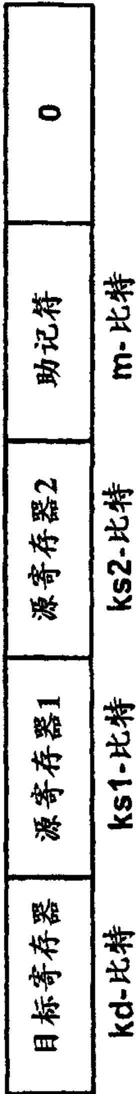
以上说明和实现可能性均涉及图 12 所示类型的硬件块。需指出的是，本发明并不局限于此。所述配置数据的生成也可被实现用于改进或扩展的硬件块。在此，所述配置数据的生成和通过采取该配置数据所进行的硬件块配置都可以快速、简单和有效地实现。不过此处的硬件块组件得到了最佳利用。这使硬件块实现了极为有效的工作。

参考符号清单

1	预解码单元
2	指令缓冲器
3	解码重命名及装载单元
4	s 单元
5	数据高速缓冲器
6	存储器接口
41	可编程结构缓冲器
42	具有可编程结构的功能单元
43	整型/地址指令缓冲器
44	整型寄存器文件
AUx	算术单元
CU	比较单元
DEMUX	多路分解器
MUXAx	第一类型的多路转换器
MUXB	第二类型的多路转换器

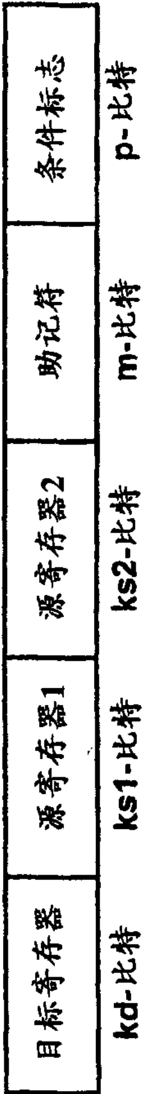
01.03.23

说明书附图



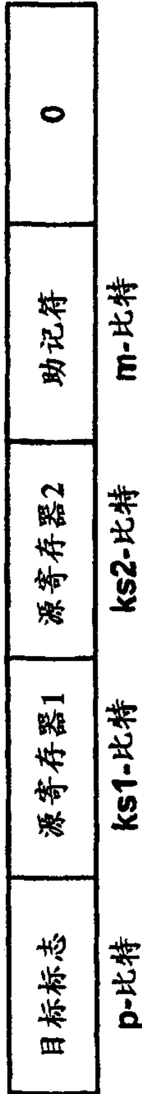
标准指令

图 1A



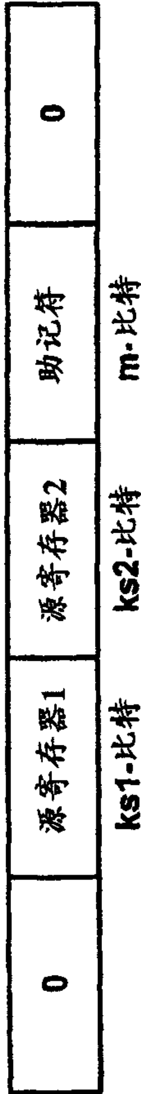
条件指令

图 1B



pxx-指令

图 1C



loopxx-指令

图 1D

01·03·23

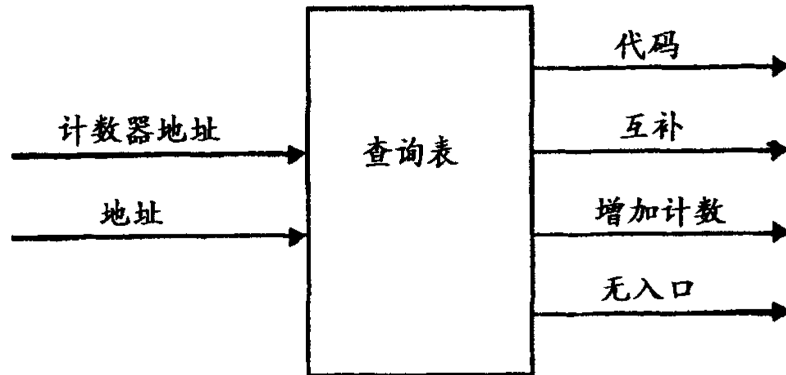


图 2

配置比特	配置指数	子号	互补	增加计数	无入口

图 3A

0	0	0	0	0	1

图 3B

01·03·23

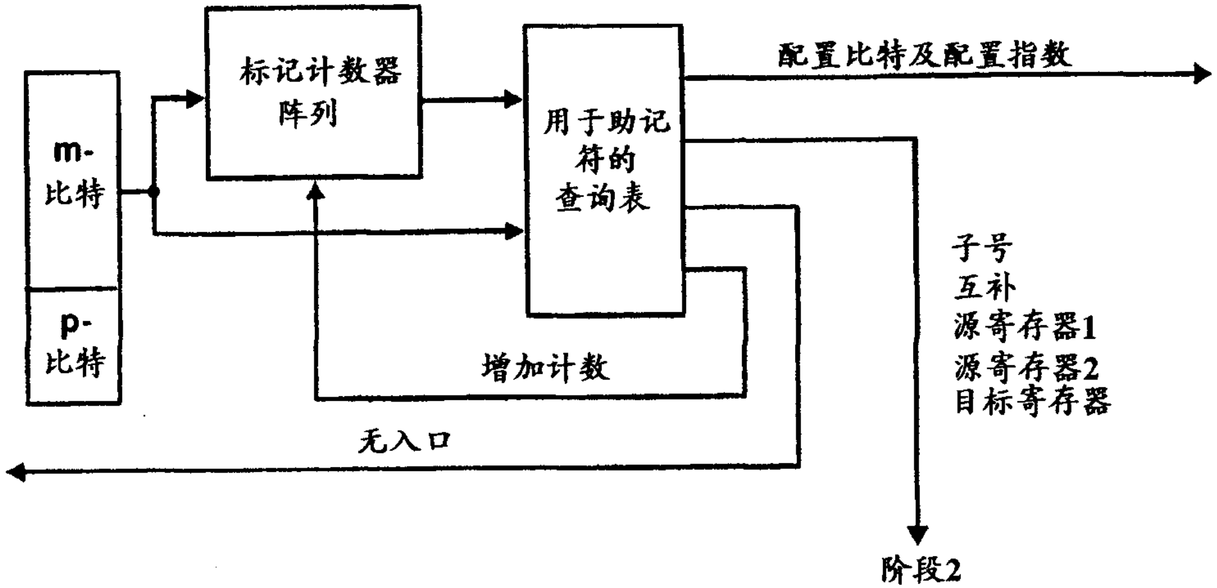


图 4

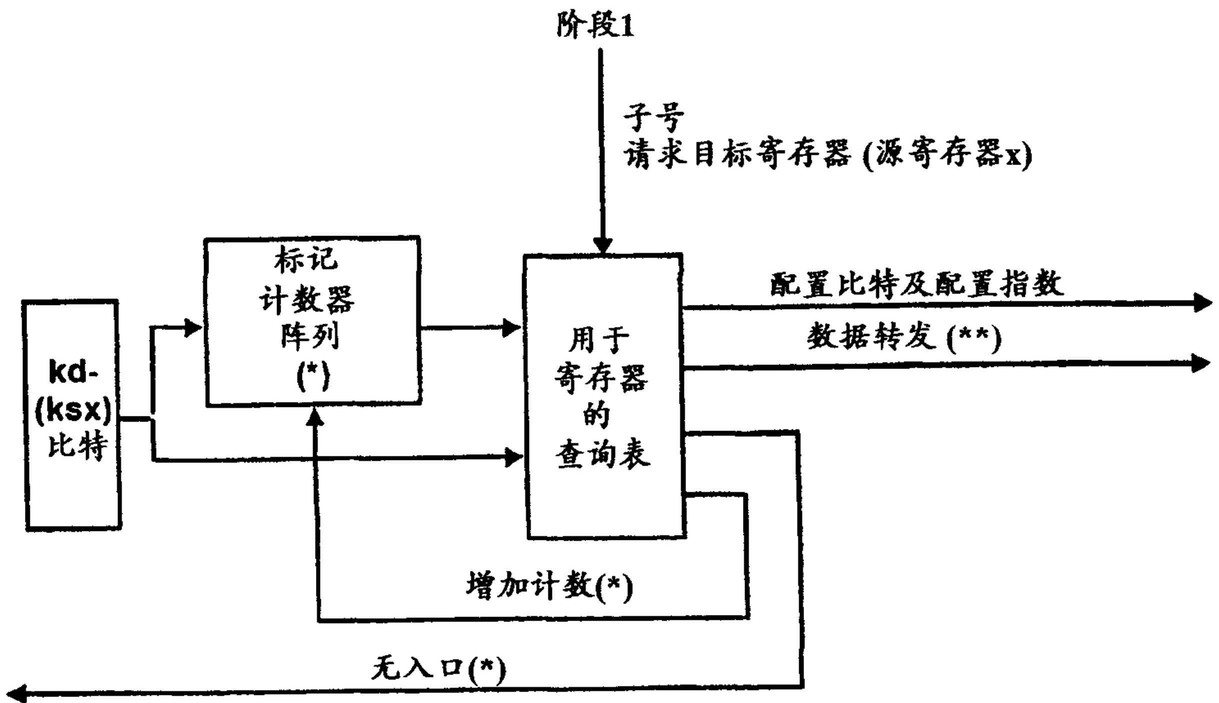


图 5

01·03·27

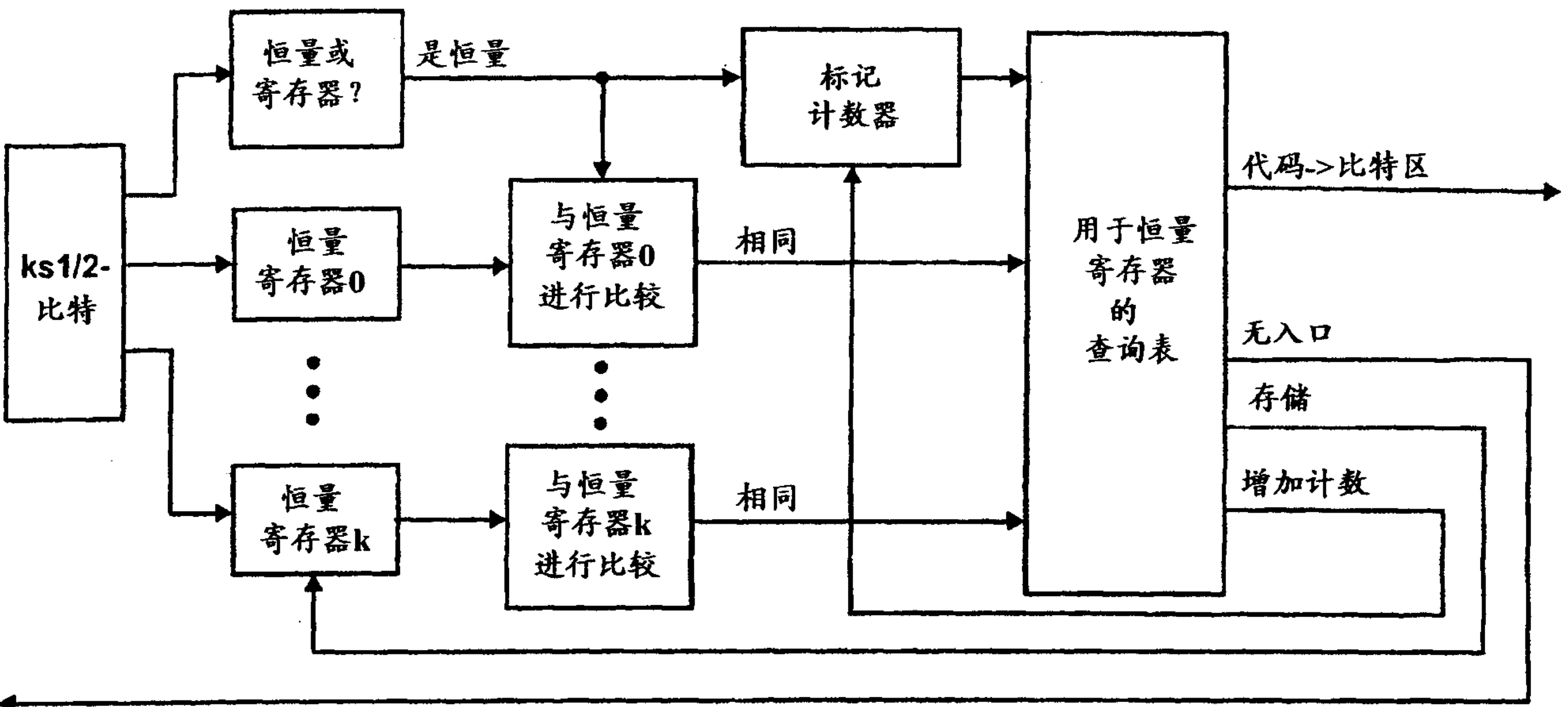


图 6

01·03·23

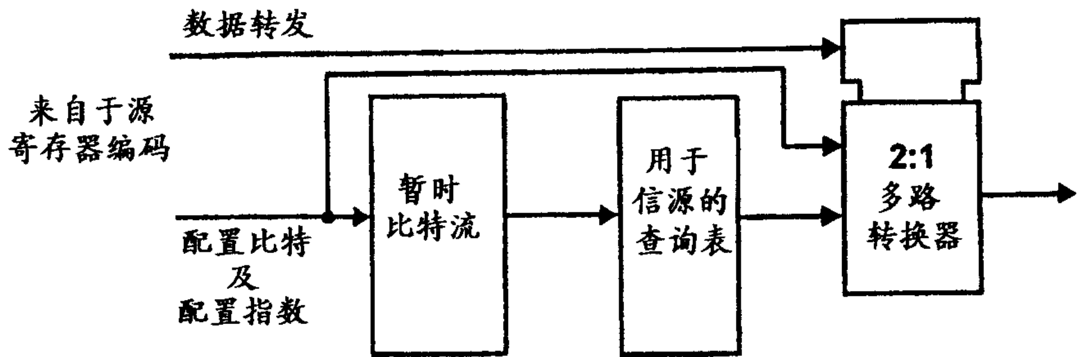


图 7

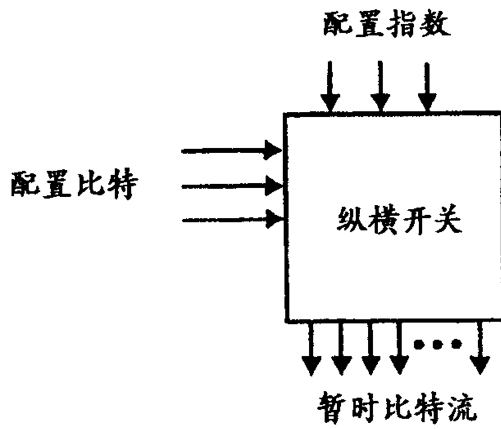


图 8

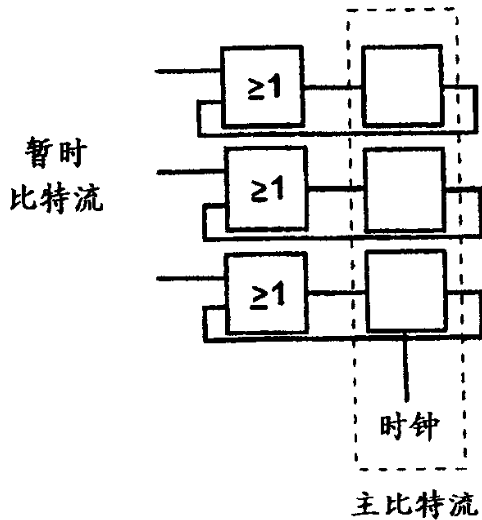


图 9

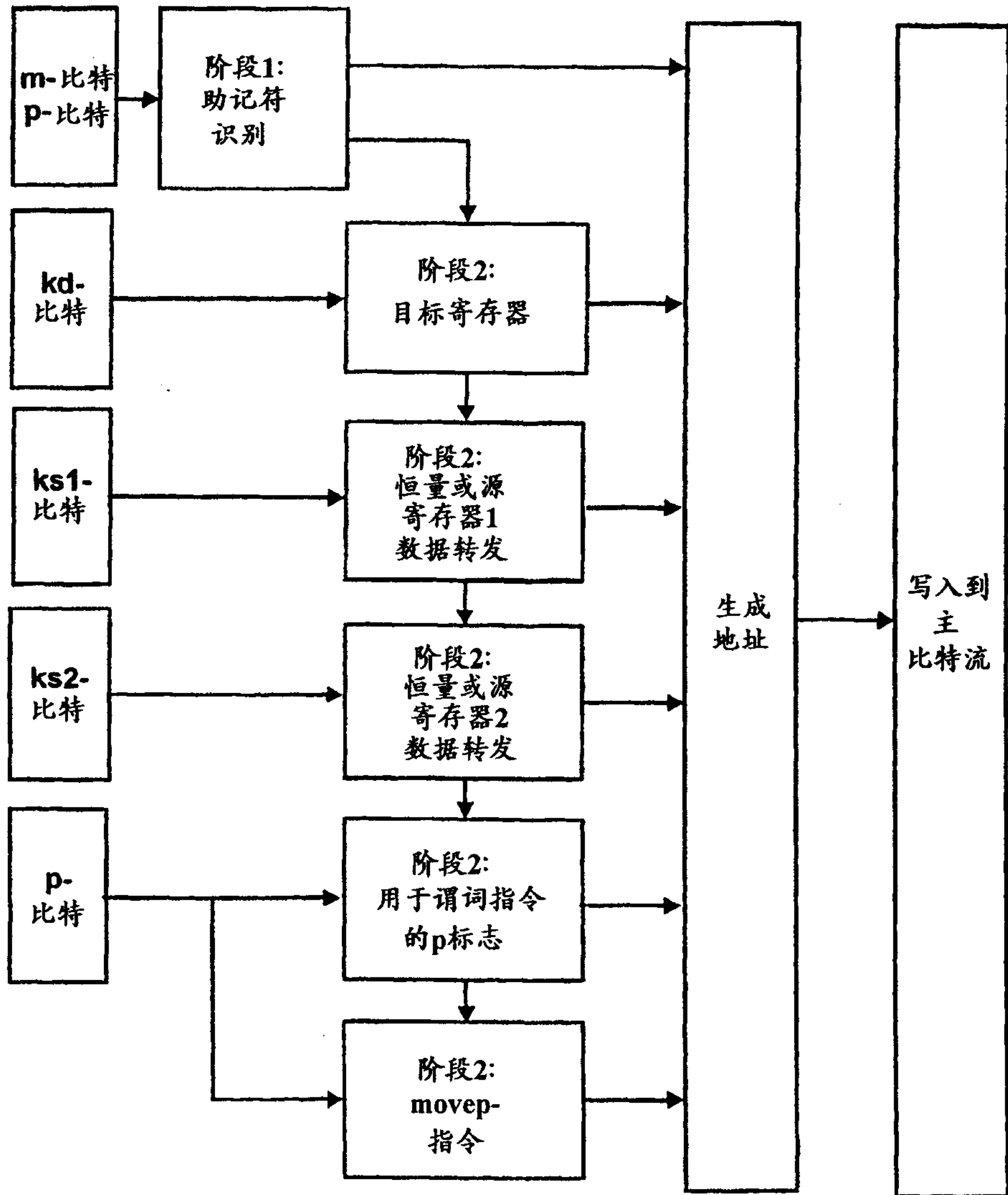


图 10

01·03·23

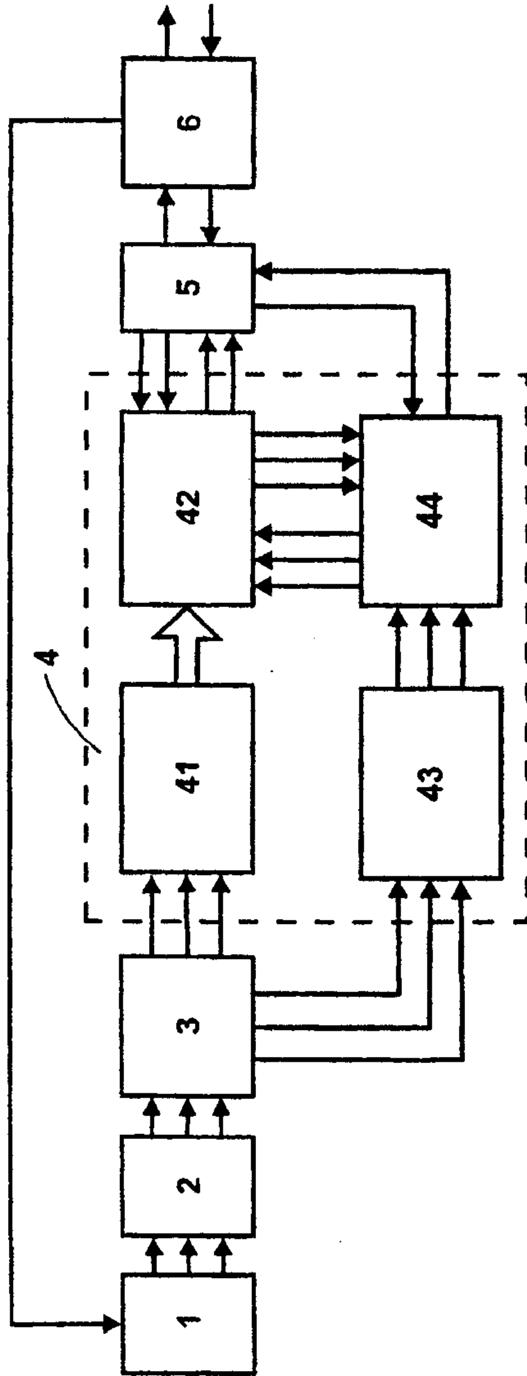


图 11

01.03.23

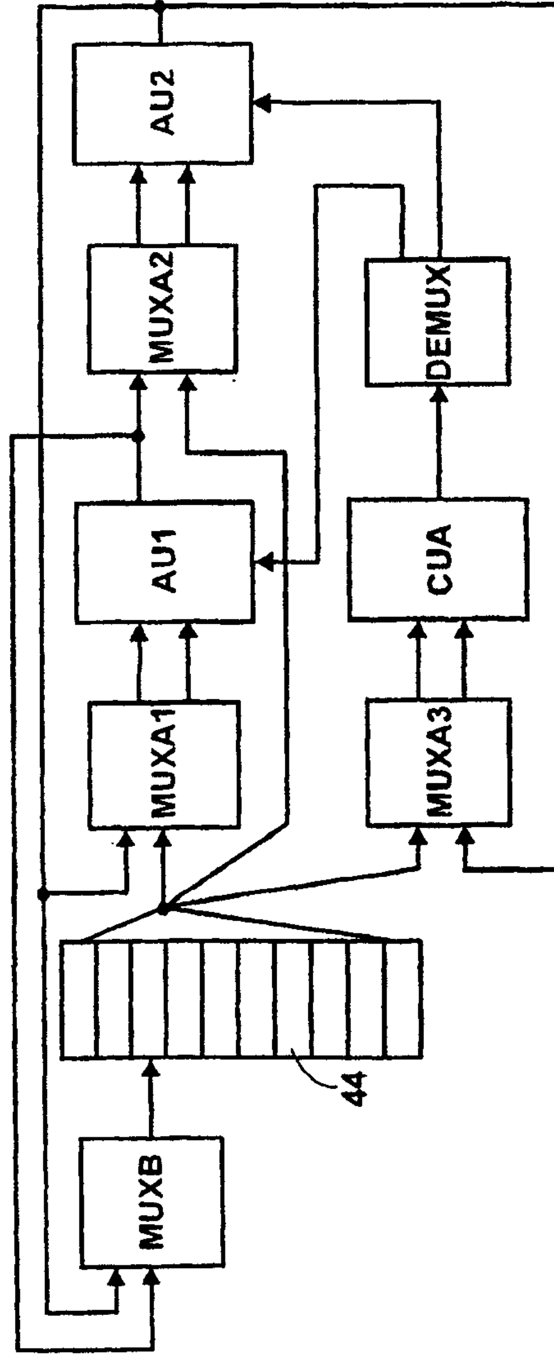


图 12