



[12] 发 明 专 利 说 明 书

[21] ZL 专利号 97180942.9

[43] 授权公告日 2003 年 3 月 12 日

[11] 授权公告号 CN 1103079C

[22] 申请日 1997. 12. 12 [21] 申请号 97180942.9

[30] 优先权

[32] 1996. 12. 23 [33] US [31] 08/772,686

[86] 国际申请 PCT/US97/22768 1997. 12. 12

[87] 国际公布 WO98/28689 英 1998. 7. 2

[85] 进入国家阶段日期 1999. 6. 23

[71] 专利权人 全斯美达有限公司

地址 美国加利福尼亚州

[72] 发明人 M·J·温 G·P·得索萨

[56] 参考文献

US4598402 1986. 07. 01 G06F11/10

US5239646 1993. 08. 24 G06F11/00

US5463767 1995. 10. 31 G06F11/00,7/02

审查员 马晓亚

[74] 专利代理机构 上海专利商标事务所

代理人 李 湘

权利要求书 7 页 说明书 55 页 附图 9 页

[54] 发明名称 用于高级微处理器的门控存储缓冲器

[57] 摘要

一种门控存储缓冲器, 包含: 按照顺序暂时存储除其它存储器内容以外所有在主处理器将目标指令序列翻译为主指令的翻译期间产生的存储器内容的装置; 如果执行翻译时未发生意外事件则按照顺序将翻译期间产生的存储器内容转移至存储器的装置; 指示最近的向同一存储器地址存储的存储器内容以响应对存储器地址的访问的装置; 以及如果执行翻译时未发生意外事件则按照顺序消除翻译期间产生的存储器内容的装置。

1. 一种对处理器指令序列执行期间产生的数据存储在存储器中进行控制的缓冲器，其特征在于，它包含：

多个存储单元，每个能够保存寻址至存储器的数据和数据地址；

当产生数据时将处理器操作产生的数据转移至存储单元直到指令序列完成执行为止的装置；

识别缓冲器内由未完成执行指令序列产生的数据的装置；

检测缓冲器内指向特定存储器地址的最近数据以响应存储器访问的装置；

在产生存储内容的指令序列执行完成而未产生意外事件或错误之后将存储单元内的数据转移至存储器的装置；以及

当产生存储内容的指令序列执行时产生意外事件或错误时消除存储单元内的数据的装置。

2. 如权利要求 1 所述的缓冲器，其特征在于，检测缓冲器内指向特定存储器地址的最近数据以响应存储器访问的装置包含缓冲器内指向特定存储器地址的每次最近可寻址数据量的装置。

3. 如权利要求 2 所述的缓冲器，其特征在于，识别缓冲器内由未完成执行指令序列产生的数据的装置包含将未完成执行的指令序列产生的数据从缓冲器内其它数据分离直到指令序列执行为止的装置。

4. 如权利要求 3 所述的缓冲器，其特征在于，将数据与缓冲器内其他数据分离直到已经执行了指令序列为止的装置是指出缓冲器内数据序列开始和结束位置的装置。

5. 如权利要求 4 所述的缓冲器，其特征在于，缓冲器内指向特定存储器地址以响应存储器访问的每一最近可寻址数据量的装置包含用寻址至同一存储器地址并由尚未执行指令序列产生的新数据代替缓冲器内数据序列开始与结束之间存储单元内旧数据的装置。

6. 如权利要求 4 所述的缓冲器，其特征在于，缓冲器内指向特定存储器地址以响应存储器访问的每次最近可寻址数据量的装置包含：

可寻址存储单元的第一指示器，用来将放入缓冲器内的数据识别为最近数据；

将正向存储单元内放入数据的存储器地址与存储单元内已有数据的存储器地址进行比较并去除存储器地址与存储单元内放入的数据存储器地址相同的数据的指示符的比较器；

可寻址存储单元的第二指示器，用来将存储单元内的数据识别为指令序列产生的最近数据以响应未产生错误或意外事件的指令序列执行；以及

将产生存储内容的指令序列的执行产生意外事件或错误时从缓冲器内消除的数据的存储器地址与识别未消除并由第二指示器识别为指令序列产生的最近数据的存储器地址进行比较的比较器，用来响应未产生错误或意外事件的指令序列的执行并作为最近数据写入第一指示器。

7. 如权利要求 2 所述的缓冲器，其特征在于，当产生存储内容的指令序列执行时产生意外事件或错误时消除存储单元内的数据的装置包含使数据无效以响应指令序列执行期间产生的错误或意外事件的指示器。

8. 如权利要求 3 所述的缓冲器，其特征在于，从缓冲器内其它数据中分离出数据直到指令序列完成执行为止的装置包括将数据序列放入所指缓冲器物理分离部分内以包含尚未执行指令数据的装置。

9. 如权利要求 8 所述的缓冲器，其特征在于，指示缓冲器内指向特定存储器地址的每次最近可寻址数据量以响应存储器访问的装置包含：

用寻址至同一存储器地址并由尚未执行指令序列产生的新数据代替指向包含未执行指令序列数据缓冲器的物理分离部分内存储单元上的旧数据的装置。

10. 如权利要求 8 所述的缓冲器，其特征在于，当产生数据的指令序列执行时未产生意外事件或错误时将存储单元内的数据转移至存储器的装置进一步包含指示数据由未发生错误的指令序列产生的装置。

11. 如权利要求 8 所述的缓冲器，其特征在于，在产生数据内容的指令序列执行后未产生意外事件或错误时将存储单元内的数据转移至存储器的装置进一步包含比较器，它比较未发生错误或意外事件的指令序列产生的缓冲器内数据的存储器地址并仅仅选择最近可寻址数据存储在任何存储器地址上。

12. 一种对处理器指令序列执行期间产生的数据存储于存储器中进行控制的方法，其特征在于，包含以下步骤：

当产生存储缓冲器分离部分的数据时按照顺序将与存储器地址相关的数据转移；

检测存储缓冲器中特定存储器地址的最近数据以响应对存储器地址的访问；
如果指令序列未产生意外事件或错误则将来自存储缓冲器分离部分的所有数据转移以抽取至存储器；以及

如果指令序列引起意外事件或错误则从存储缓冲器分离部分消除所有数据。

13. 如权利要求 12 所述的方法，其特征在于，当产生存储缓冲器分离部分的数据时按照顺序将与存储器地址相关的数据转移的步骤包含按照存储单元顺序将数据放入第一与最后存储单元之间的存储缓冲器部分内的步骤。

14. 如权利要求 13 所述的方法，其特征在于，如果指令序列未产生意外事件或错误则将来自存储缓冲器分离部分的所有数据转移以抽取至存储器的步骤包含移动第一存储单元指示以指定最后存储单元的步骤；以及

如果指令序列引起意外事件或错误则从存储缓冲器分离部分消除所有数据的步骤包含移动最后存储单元指示以指定第一存储单元的步骤。

15. 如权利要求 14 所述的方法，其特征在于，检测存储缓冲器内特定存储器地址最近数据以响应对存储器地址访问的步骤包含以下步骤：

当放入存储单元的数据是与该数据相关的存储器地址的最近数据时对任何可寻址数据放置第一指示；

当数据从存储缓冲器分离部分转移时，且该数据为从存储缓冲器分离部分转移的最近数据，则对任何可寻址数据放置第二指示；

当消除存储缓冲器分离部分内所有数据时对包含第二指示的任何可寻址数据放置第一指示；以及

比较存储器地址和保护在可寻址的存储单元中的任何数据的第一指示以响应保持在存储缓冲器中的数据的存储器地址的访问。

16. 如权利要求 12 所述的方法，其特征在于，包含下列步骤：

当产生存储缓冲器分离部分的数据时按照顺序将与存储器地址相关的数据转移的步骤包含：

在缓冲器物理分离部分放置数据序列；以及

指示存储缓冲器物理分离部分包含来自未完全执行指令序列的数据。

17. 如权利要求 16 所述的方法，其特征在于，检测存储缓冲器内特定存储器地址最近数据以响应对存储器地址访问的步骤包含以下步骤：

将每个分离的数据可寻址部分写入存储缓冲器的分离部分内，复盖在存储缓

冲器的分离部分内的同一存储器地址上的；以及

将存储缓冲器中所有数据的存储器地址与被访问的任何存储器地址比较以检测存储缓冲器内的最近数据。

18. 如权利要求 16 所述的方法，其特征在于，如果指令序列未产生意外事件或错误则将来自存储缓冲器分离部分的所有数据转移以抽取至存储器的步骤包含指示存储缓冲器物理分离部分包含来自已执行指令序列数据的步骤；以及

如果指令序列引起意外事件或错误则从存储缓冲器分离部分消除所有数据的步骤包含指示存储缓冲器物理分离部分包含无效数据。

19. 如权利要求 18 所述的方法，其特征在于，进一步包含从存储缓冲器的分离部分将所有数据抽取至存储器的步骤，为此它对每一存储器地址比较从分离部分转移的所有数据的地址并只将任何存储器地址的最近数据抽取至存储器。

20. 一种用于处理系统的装置，处理系统包括主处理器，它能够执行第一指令集以帮助运行不同指令集的指令，不同指令集由主处理器翻译为第一指令集，其特征在于，包含：

缓冲器，包含暂时存储所产生的存储器内容直到确定翻译指令序列在主处理器上不发生意外事件或错误地执行为止的分离区域；

检测向缓冲器内所保持的特定存储器地址存储的最近存储器内容以响应对存储器地址的访问的逻辑电路系统；

从分离部分去除暂时存储的存储器内容并在确定翻译指令序列在主处理器上能不发生意外事件或错误地执行时永久存储存储器内容的抽取电路系统；以及

当确定翻译指令序列在主处理器上能不发生意外事件或错误地执行时消除缓冲器分离部分内暂时存储的存储器内容的装置。

21. 如权利要求 20 所述的用于处理系统的装置，其特征在于：

缓冲器分离区域由指向第一和最后存储单元的指针定义；

检测向缓冲器内特定存储器地址存储的最近存储器内容以响应对存储器地址的访问的逻辑电路系统包含：

位于每个存储单元内的第一位置以指示最近存储器内容；

无论何时存储器内容放置在存储单元内在第一位置内放置指示并去除第一位置内同一存储器地址的旧存储器内容指示的电路系统；

位于每个存储单元内的第二位置以指示缓冲器分离部分以外的最近存储器

内容；

无论何时由抽取电路系统从分离部分去除存储器内容在第二位置内放置指示并去除第二位置内同一存储器地址的旧存储器内容指示的电路系统；以及
比较存储器地址并检测第一位置内存储器内容的指示的比较器。

22. 如权利要求 20 所述的用于处理系统的装置，其特征在于：

缓冲器分离区域由存储单元的物理分离部分所限定，该存储单元包含的指示表示该部分内的存储器内容已经由未完成执行的指令序列产生；

检测向缓冲器内所保持的特定存储器地址存储的最近存储器内容以响应对存储器地址的访问的逻辑电路系统包含：

将存储器内容覆盖分离部分已经存储的任何存储器内容的存储器地址检测器；

指定存储器内容放置在缓冲器任何物理分离部分内次序的电路系统；以及
比较存储器地址并检测缓冲器部分内任何存储单元上最旧存储器内容的比较器。

23. 一种门控存储缓冲器，其特征在于，包含：

暂时将在主处理器把目标指令序列翻译为主指令的翻译期间按顺序产生的所有存储器内容与其它存储器内容隔开的装置；

如果执行翻译时未发生意外事件则按照顺序将翻译期间产生的存储器内容转移至存储器的装置；

指示最近的向同一存储器地址存储的存储器内容以响应对存储器地址的访问的装置；以及

如果执行翻译时未发生意外事件则按照顺序消除翻译期间产生的存储器内容的装置。

24. 一种计算机，其特征在于，包含：

为执行主指令集指令而设计的主处理器；

将目标指令集的指令翻译为主指令集指令的主机软件；

存储器；以及

用于存储主机软件在主处理器上翻译目标指令序列时产生的数据的存储缓冲器，它包括：

多个存储单元，每个能够保存寻址至存储器的数据和数据地址；

当产生数据时将主处理器操作产生的数据转移至存储单元直到指令序列完成执行为止的装置；

识别缓冲器内由未完成执行指令序列产生的数据的装置；

检测缓冲器内指向特定存储器地址的最近数据以响应存储器访问的装置；

在产生存储内容的指令序列执行完成而未产生意外事件或错误之后将存储单元内的数据转移至存储器的装置；以及

当产生存储内容的指令序列执行时产生意外事件或错误时消除存储单元内的数据的装置。

25. 如权利要求 24 所述的计算机，其特征在于：

当数据产生时将处理器操作产生的数据转移至存储单元直到指令序列完成执行为止的装置包含将数据转移至缓冲器内顺序存储单元上的装置；以及

检测缓冲器内指向特定存储器地址的最近数据以响应存储器访问的装置包含缓冲器内指向特定存储器地址的每次最近可寻址数据量的装置。

26. 如权利要求 25 所述的计算机，其特征在于，识别缓冲器内由未完成执行指令序列产生的数据的装置包含将未完成执行的指令序列产生的数据从缓冲器内其它数据分离直到指令序列执行为止的装置。

27. 如权利要求 26 所述的计算机，其特征在于，将数据从缓冲器内数据分离直到指令序列已执行的装置是指示缓冲器内数据序列开始和结束的装置。

28. 如权利要求 27 所述的计算机，其特征在于，缓冲器内指向特定存储器地址的每次最近可寻址数据量以响应存储器访问的装置包含用寻址至同一存储器地址并由尚未执行指令序列产生的新数据代替缓冲器内数据序列开始与结束之间存储单元内旧数据的装置。

29. 如权利要求 27 所述的计算机，其特征在于，缓冲器内指向特定存储器地址的每次最近可寻址数据量以响应存储器访问的装置包含：

可寻址存储单元的第一指示器，用来将放入缓冲器内的数据识别为最近数据；

将存储单元内放入数据的存储器地址与存储单元内已有数据的存储器地址进行比较并去除存储器地址与存储单元内放入的数据存储器地址相同的数据的指示符的比较器；

可寻址存储单元的第二指示器，用来将存储单元内的数据识别为指令序列产

生的最近数据以响应未产生错误或意外事件的指令序列执行；以及

将产生存储内容的指令序列的执行产生意外事件或错误时从缓冲器内消除的数据的存储器地址与识别未消除并由第二指示器识别的由指令序列产生的最近数据的存储器地址进行比较的比较器，用来响应未产生错误或意外事件的指令序列的执行并作为最近数据写入第一指示器。

30. 如权利要求 24 所述的计算机，其特征在于，当指令序列执行时产生意外事件或错误时消除缓冲器内的数据的装置包含使数据无效以响应指令序列执行期间产生的错误或意外事件的指示器。

31. 如权利要求 26 所述的计算机，其特征在于，从缓冲器内其它数据中分离出数据直到指令序列完成执行为止的装置包括将数据序列放入所指缓冲器物理分离部分内以包含尚未执行的指令序列数据的装置。

32. 如权利要求 31 所述的计算机，其特征在于，指示缓冲器内指向特定存储器地址的每次最近可寻址数据量以响应存储器访问的装置包含：

用寻址至同一存储器地址并由尚未执行指令序列产生的新数据代替指向包含未执行指令序列数据缓冲器的物理分离部分内存储单元上的旧数据的装置。

33. 如权利要求 31 所述的计算机，其特征在于，当产生数据的指令序列执行时未产生意外事件或错误时将存储单元内的数据转移至存储器的装置进一步包含指示数据由未发生错误的指令序列产生的装置。

34. 如权利要求 31 所述的计算机，其特征在于，当产生数据的指令序列执行时未产生意外事件或错误时将存储单元内的数据转移至存储器的装置进一步包含比较器，它比较未发生错误或意外事件的指令序列产生的缓冲器内数据的存储器地址并仅仅选择最近可寻址数据存储在任何存储器地址上。

用于高级微处理器的门控存储缓冲器

发明领域

本发明涉及计算机系统，特别是涉及与主微处理器一起控制存储器存储操作的门控存储缓冲器，主微处理器执行从为不同处理器的执行而设计的程序翻译过来的程序。

背景技术

在计算机上运行着成千上万针对特定微处理器系列设计的应用程序，其中为利用“X86”系列微处理器(包括 Intel®8088、Intel 8086、Intel 80186 Intel 80286、i386、i486 以及随后发展的各种奔腾®系列微处理器，它们都是由位于加州 Santa Clara 的 Intel 公司设计和制造)的计算机(通常称为“IBM 兼容个人电脑”)而设计的程序数量最多，还有许多其他的程序设计来使来用其它系列处理器的计算机运转。由于运行在这些计算机上的应用程序非常多，所以能够为这类计算机(特别是为处理 X86 程序而设计的计算机)所用的微处理器拥有巨大的市场。微处理器市场不仅容量巨大而且获利丰厚。

虽然能够运行大量应用程序的微处理器市场容量巨大而且获利丰厚，但是设计新的有竞争力的微处理器却不那么容易。例如，虽然 X86 系列的微处理器已经问世多年并且在大多数销售和使用的计算机内都包含这些处理器，但是也只有少数成功的有竞争力的微处理器者能够运行 X86 程序。其中的原因是多方面的。

为了取得成功，微处理器必须能在不高于现有微处理器开销的前提下，与现有处理器一样快地运行针对现有系列处理器设计的所有程序(包括操作系统和已有程序)。此外，为了在经济上取得成功，新的微处理器必须至少在某一方面胜过现有的处理器，从而使买主有先于现有经考虑的处理器而选购新微处理器的理由。

要做到使微处理器运行速度与现有微处理器一样快是困难和昂贵的。处理器通过诸如加载、移位、相加、存储和类似的低层次操作之类的基本操作执行指令并且在执行应用程序提供的任何指令时仅仅响应这类基本操作。例如在诸如 X86

之类为运行复杂指令集计算机(CSIC)的指令而设计的处理器中, 指令可能指定待进行的过程具有较高的级别, 该处理器包含存储所谓微指令的只读存储器(ROM)。每条微指令包含一串基本指令, 依照顺序执行这些基本指令将得到高级CISC指令所命令的结果。典型的“A加B”CISC指令经过译码后形成在ROM内查找地址, 执行“A加B”指令功能的微指令就存储在ROM内。加载微指令后, 按照顺序执行其基本指令, 从而完成“A加B”指令的执行。在这类CSIC计算机中, 微指令内的基本操作在程序执行过程中绝对不可能发生变化。每条CISC指令只能通过指令译码、寻址和读取微指令以及按照微指令提供的顺序运行基本操作序列完成运行过程。每次运行微指令时都必须遵循同一顺序。

运行X86应用程序的目前水平的处理器利用大量的技术, 以在合理的性价比前提下提供最快的处理速度。任何采用已知硬件技术提高处理器速度的新处理器都必然会增加处理硬件的复杂度。这增加了硬件成本。

例如超标量微处理器(superscalar), 为了同时执行两个或更多的操作, 它使用多条处理通道, 对此需要一系列额外的要求。在最底层级别上, 简单的超标量微处理器将每条应用指令译码为执行应用指令功能的微指令。随后, 如果两条微指令不需要相同的硬件资源并且一条微指令的执行不依赖于待处理的另一条微指令的结果, 则简单超标量微处理器将这两条微指令安排为同时执行。

更加高级的超标量微处理器一般将每条应用指令译码为一系列的基本指令, 从而以最有效率的执行次序重新排序和安排这些基本指令的执行。这需要每条基本操作都能单独寻址和存取。为了完成重新排序, 处理器必须确保需要其它基本指令产生数据的基本指令在其他指令产生所需数据之后执行。这类超标量微处理器必须确保同时执行的两条基本指令不需要使用同一硬件资源。而且还必须在完成转移操作之前解决条件转移。

因此超标量微处理器需要大量的硬件来比较基本指令之间的相互关系, 重新排序和安排执行任何指令的基本指令的顺序。随着处理通道数的增加, 完成这些超标量加速技术的硬件数量和成本将以二次方的速度增加。所有这些硬件需求都增加所涉及电路系统的复杂性和成本。在处理微指令过程中, 当每次执行一条应用指令时, 超标量微处理器都必须使用比较复杂的寻址和读取硬件来读取每条基本指令, 都必须根据其它基本指令和硬件使用状态重新排序和重新安排这些基本指令, 并且随后还必须执行所有重新安排后的基本指令。由于每次执行一条应用

指令时都必须通过整个硬件序列，所以限制了超标量处理器能够执行指令的速度。

即使利用各种硬件技术提高了处理速度，由此带来的硬件复杂性也大大增加了这类微处理器的成本。例如，Intel i486 DX4 处理器采用大约 150 万个晶体管。但是如果要在诸如 Intel 奔腾®之类的基本超标量处理器中通过增加硬件来利用两条通道完成指令处理所需的依存度检查和执行安排，则需要 300 多万个晶体管。在 Intel Pentium Pro™ 微处理器中，为了能重新排序源自不同目标指令的基本指令，提供预测执行，完成寄存器更名和提供转移预测，增加到超过 600 多万个晶体管。由此可见，为提高运算速度而增加硬件已使最新的目前水平的微处理器的晶体管数量出乎寻常地增加。

即使采用这些已知的技术，但是由于现有微处理器制造商已经采用了绝大部分经济可行的技术来提高现有微处理器的运行速度，所以也不可能制造出更快的微处理器。这样，设计更快的处理器就成了一项非常困难和费钱的工作。

降低处理器成本也是非常困难的。如上所述，制造具有足够能力处理器的硬件加速技术非常昂贵。如果要设计一种新的处理器，必须拥有生产硬件的设备。由于芯片制造商一般不会投资小批量器件，所以难以获得这类设备。生产芯片制造设备所需的资本投资如此巨大，以致于超出了大多数公司力所能及的范围。

即使设计出的新处理器能够至少以竞争处理器同样快的速度运行所有针对处理器系列设计的应用程序，竞争处理器在价格中也包含了足够的利润，从而确保其能面对竞争对手作大幅度削价。

虽然通过增加硬件复杂度来设计富有竞争力的处理器是非常困难的，但是可以有另一种运行针对某一特定系列微处理器(目标微处理器)设计的应用程序(目标应用程序)的方法，它在另一种更快的微处理器(主微处理器)上用软件仿真目标微处理器。由于这仅仅需要增加一定形式的仿真软件以在更快的微处理器上运行应用程序，所以是一种成本日趋低廉的方法。仿真器软件将针对目标处理器系列编写的应用程序目标指令改写为能够被主微处理器执行的主指令(host instruction)。随后这些被改写的指令在较快主微处理器上操作系统的控制下运行。

有许多种不同的设计可以使目标应用程序运行在处理器速度快于目标计算机处理器的主计算机上。由于从理论上讲，精简指令集(RISC)微处理器更为简单

而且速度较其它类型处理器更快，所以利用仿真软件执行目标程序的主计算机都采用 RISC 微处理器。

但是即使利用仿真器软件的 RISC 计算机系统经常能够运行 X86(或其它)程序，它们的运行速度通常也明显慢于 X86 计算机系统运行同一程序的速度。而且这些仿真器程序常常无法运行所有或大量现有的目标程序。

仿真器程序无法象目标微处理器那样快地运行目标程序的原因相当复杂并且需要对不同的仿真操作有所了解。图 1 包括一系列的示意图，它们表示多种不同类型微处理器执行目标应用程序的不同的方式。

在图 1(a)中，示出诸如 Intel X86 微处理器之类的典型 CISC 微处理器运行目标应用程序该目标应用程序为目标微处理器运行而设计。如图所示，应用程序运行在利用 CISC 操作系统(例如 X86 计算机使用的 MS DOS、Windows 3.1、Windows NT 和 OS/2)的 CISC 处理器上，这些操作系统设计成提供访问计算机硬件的接口。典型情况是选择应用程序的指令，只通过操作系统提供的访问使用计算机设备。因此操作系统处理允许应用程序访问计算机存储器和各种输入/输出设备的操作。目标计算机包含操作系统能够识别的存储器和硬件，并且目标应用程序对操作系统的调用使操作系统设备驱动程序在目标计算机定义的设备上产生预期的操作。应用程序的指令在处理器上执行，它们被变换为可以被处理器执行的操作，这些操作由微代码或汇编为微代码的更加基本的操作具体实现。如上所述，每次执行复杂的目标指令时，指令都调用以微代码(或同一基本操作组)形式存储的同一子程序。总是执行同一子程序。如果处理器是超标量的，则执行目标指令的这些基本操作常常可以被处理器以上述方式，利用各种处理通道进行重新排序、重新安排和执行；但是子程序仍然被读取和执行。

在图 1(b)中，诸如用于苹果 Macintosh 计算机中的 PowerPC 之类的典型 RISC 微处理器运行着同一目标应用程序该目标应用程序是针对图 1(a)CISC 处理器运行而设计的。如图所示，目标应用程序运行于采用至少一部分目标操作系统的主处理器以响应目标应用程序产生的一部分调用。典型的是对目标操作系统应用类部分的调用，用来在显示器上提供图形接口和通常是应用类程序的小型实用程序。目标应用程序和目标操作系统的这些部分由诸如 SoftPC®之类的软件仿真器变换，它将目标应用程序和应用类目标操作系统配备的指令分解为主处理器及其主操作系统能够执行的指令。主操作系统提供了访问 RISC 计算机的存储器和输

入/输出硬件的接口。

但是主 RISC 处理器及在 RISC 计算机中与其相关的硬件设备通常与这样一些设备有相当大的差异，它们与针对目标应用程序而设计的处理器相关；并且目标应用程序提供的各种指令被设计为使目标操作系统的设备驱动程序在访问目标计算机各部分时协同工作。因此对于将目标应用程序指令变换为主操作系统能够利用的基本主指令的仿真程序而言，它必须在某种程度上将为操纵目标计算机内硬件设备而设计的操作与主系统硬件设备能够实现的操作联系起来。由于目标设备并不等同于主计算机的设备，所以通常需要由仿真器软件产生响应目标应用程序指令的虚拟设备来完成主系统无法实现的操作。有时候需要仿真器产生从这些虚拟设备通过主操作系统至主硬件设备的链接，这些主硬件设备虽然是真实存在，但是要由主操作系统按照不同的方式寻址。

由于众多的原因，以这种方式执行的目标程序速度较慢。首先，来自目标应用程序和目标操作系统的每条目标指令都必须由仿真器变换为主处理器使用的主基本功能。如果目标应用程序是针对诸如 X86 之类的 CSIC 机而设计的，则目标指令是长度可变的并且相当复杂的，因此将它们变换为主基本指令就相当费事。首先对源目标指令译码，并确定构成目标指令的基本主指令序列。随后确定每串基本主指令序列的地址，读取每串基本主指令序列，并按照顺序或者不按照顺序执行这些基本主指令。每次执行指令时都必须由仿真器将目标应用程序和操作系统指令变换为主处理器理解的主指令，这需要大量额外的步骤，因此减慢了仿真处理的速度。

其次，许多目标指令包括由特定硬件设备实施的操作的基准，这些特定硬件设备在目标计算机，硬件中起着特殊的作用，但在主计算机中却不存在。为了执行该类操作，仿真软件必须通过已有的主操作系统与主计算机的硬件设备实现软件连接或者配备虚拟硬件设备。用软件仿真另一种计算机硬件是非常困难的。仿真软件必须对主操作系统的每一个目标应用程序的调用产生虚拟设备；并且每一虚拟设备都必须向实际主设备提供调用。硬件设备仿真要求当目标指令使用设备时从存储器中读取表示指令所需虚拟设备的代码并进行运行以实现设备功能。解决问题的这些方法中的每一个在执行指令序列时都额外增加了一系列的操作。

仿真问题的复杂化是目标应用要解决各种意外事件的结果，为了使计算机系统运行，目标计算机的硬件和目标操作系统要对这些意外事件作出处理。当处理

目标计算机运行期间的意外事件时，一般必须通过调用微代码序列以完成保存发生意外事件时刻计算机状态的操作，必须检索正确的意外事件处理器(handle)，必须处理发生意外事件，并且必须找到程序继续运行下去的正确入口。有时候这要求程序返回至目标计算机在发生意外事件处的状态，而其它时候进入意外事件处理器提供的转移。在任一情况下，都必须在某种程度上对完成这些操作所需的目标计算机硬件和软件进行仿真处理。由于必须在发生此类意外事件时可以得到正确的目标状态以供适当执行，仿真器必须始终精确跟踪该状态以准确响应这些意外事件。在现有技术中，这要求每条指令的执行必须按照目标应用程序提供的顺序，因为只有这种方式能够保持正确的目标状态。

而且现有技术的仿真器出于其它原因，总是需要保持目标应用程序的执行顺序。目标指令可以有两种，一种影响存储器另一种影响存储器映象的输入/输出(I/O)设备。如果不试验着执行指令是无法知道操作是影响存储器还是存储器映象 I/O 设备的。当指令在存储器上运行时，可以作优化和重新排序并且这大大有助于提高系统运行速度。但是作用于 I/O 设备的操作常常必须按照精确的顺序进行，这些操作必须按照该顺序编程而不能省略任何步骤，否则可能对 I/O 设备的操作产生一些不利的影响。例如特殊的 I/O 操作可能会清除 I/O 寄存器的内容。如果操作不按照顺序进行从而清除了寄存器内仍然需要的值，则操作结果可能不同于目标指令所命令的那样。如果没有区分存储器和存储器映象 I/O 的装置，就需要将所有的指令都象处理作用于存储器映象 I/O 的指令那样处理。这大大限制了优化所能达到的性能。由于现有技术的仿真器缺少检测被寻址存储器性质的装置和从这类故障恢复的装置，所以需要顺序处理目标指令，就好象每条指令都作用于存储器映象 I/O 那样。这极大地制约了主指令的优化的可能性。

另一个限制现有技术仿真器优化主代码能力的问题源于自修改代码。如果目标指令被变换为主指令序列，而主指令序列又接着写回以改变了源目标指令，则主指令不再有效。结果仿真器必须不断地作检验以确定目标代码区域是否有存储。所有这些问题都使这种类型的仿真比在目标处理器上运行目标应用程序慢得多。

图 1(b)所示另一种类型的仿真软件实例在题为“Talisman:快速而精确的多计算机模拟”(R.C.Bedichek, 麻省理工学院计算机科学实验室)的文章中有所论述。这是更为完整的转换实例，它可以仿真完整的研究系统并运行研究目标操作

系统。Talisman 采用主 UNIX 操作系统。

在图 1(c)中示出了另一种仿真实例。在该实例中，用于苹果 Macintosh 计算机的 PowerPC 微处理器运行针对摩托罗拉 68000 系列 CISC 处理器设计的目标应用程序，后者用于早期的 Macintosh 计算机；这样做的目的是为了使苹果原有的程序能够在带 RISC 处理器的 Macintosh 计算机上运行。显而易见，目标应用程序使用至少部分目标操作系统运行于主处理器上以响应目标操作系统应用类部分。软件仿真器将目标应用程序和应用类目标操作系统程序配备的指令分解为主处理器及其主操作系统能够执行的指令。主操作系统提供了访问主计算机的存储器和输入/输出硬件的接口。

但是主 RISC 处理器及在主 RISC 计算机中与其相关的设备与配备摩托罗拉 CISC 处理器的设备有相当大的差异；并且将各种目标指令设计成为在访问目标计算机各部分时与目标 CSIC 操作系统协同工作。因此仿真程序必须将操纵目标计算机内硬件设备的操作与主系统硬件设备能够实现的操作联系起来。这需要由仿真器产生响应目标应用程序指令的软件虚拟设备并产生从这些虚拟设备通过主操作系统至主硬件设备的链接，这些主硬件设备虽然是真实存在，但是要由主操作系统按照不同的方式寻址。

由于与图 1(b)仿真运行得慢的同样的原因，以这种方式运行的目标程序运行速度较慢。首先，来自目标应用程序和目标操作系统的每条目标指令都必须经过指令读取才能变换；并且每次执行指令时从该指令导出的所有主基本功能都必须按照顺序运行。其次，仿真软件必须产生每个目标应用程序调用主操作系统的虚拟设备；并且每一虚拟设备都必须向实际主设备提供调用。第三，仿真器必须象处理直接作用于存储器映象 I/O 设备的指令那样保守地处理所有指令，否则就有发生无法恢复的意外事件的危险。最后，仿真器必须始终保持正确的目标状态并总是要在确定是否存储目标代码区域之前检查存储操作。所有这些要求都削弱了仿真器对运行在主处理器上代码作重大优化的能力并且使这种仿真的速度远远慢于目标应用程序在目标处理器上的运行速度。认为仿真速度低于现有处理器的四分之一是很好的。通常情况下，这种仿真软件在能够运行为另一种处理器设计的应用程序的能力只是有用而非基本用途的情况下，很难找到它的使用价值。

在图 1(d)中示出一种在主处理器上仿真目标应用程序的特殊方法，它为非常少数的一系列目标应用程序提供了较好的性能。目标应用程序为仿真器提供指

令，该仿真器将这些指令变换为主处理器和主操作系统的指令。主处理器为数字设备公司的 Alpha RISC 处理器，而主操作系统为微软的 NT。可以在这种系统上运行的唯一的目标应用程序是为采用 Windows WIN32s 兼容操作系统的目标 X86 处理器而设计的 32 位应用程序。由于主操作系统和目标操作系统几乎是相同的，它们设计成处理相同的指令，所以仿真器软件可以非常方便地变换指令，而且主操作系统也已设计为响应目标应用程序产生的相同调用，所以产生虚拟设备的数量大大减少。

虽然在技术上它是一种使目标应用程序运行于主处理器上的仿真系统，但却是一种非常特殊的情况。在这种情况下仿真软件运行于已经为运行相似应用程序而设计好的主操作系统上。它使得来自目标应用程序的调用能够更为简单地指向主处理器和主操作系统的正确的设备。更为重要的是，该系统只能运行 32 位 Windows 应用程序，其在所有 X86 应用程序中所占的数量不到 1%，而且该系统只能在一种操作系统上，即 Windows NT 上运行应用程序；而 X86 处理器可运行针对大量操作系统而设计的应用程序。因此就本说明书前面所表述的含义而言，这样的系统不具备兼容性。因此运行这类仿真器的处理器是不能认为是富有竞争力的 X86 处理器。

在图 1(e)中示出另一种仿真方法，它利用软件在识别不同指令集的计算机上运行为第一指令集编写的应用程序部分。这种形式的仿真软件一般由程序员使用，他将一个应用程序从一种计算机系统移植到另一种计算机上。典型的情况是目标应用程序针对除运行仿真器的主机以外的一些目标计算机而设计。仿真器软件分析目标指令，将这些指令翻译为可以在主机上运行的指令，并且将这些主指令存入高速缓冲存储器内供再次使用。这种动态翻译和高速缓存可以使应用程序部分运行得非常快。这种形式的仿真器一般与软件跟踪工具一起使用，以提供正在运行的目标程序的详细的运行信息。跟踪工具的输出又被用来驱动对跟踪信息进行分析的分析程序。

为了确定代码实际上是如何工作的，这种类型的仿真器与主机上的主操作系统协同运行，配备了主操作系统未提供的虚拟硬件，并且将设计应用软件的计算机的操作映象到主机硬件资源以执行正在运行的程序的操作。这种硬件的软件虚拟化和对主计算机的映象可能非常慢并且很不完善。

而且由于常常需要多条主指令以执行一条目标指令，所以可能产生包括故障

和陷井在内的意外事件，这需要目标操作系统的意外事件处理器，并且使主机在与目标指令边界无关的位置上终止处理主指令。当发生这种情况时，由于主处理器和存储器的状态不正确，所以无法正确处理意外事件。如果出现这种情况，仿真器必须停止并重新运行以跟踪产生意外事件的操作。因此虽然这种仿真器能够非常快地运行目标代码序列，但是却没有办法从这些意外事件中恢复过来，因此无法快速运行应用程序中任何相当大的一部分。

由于仿真器、跟踪器和相关的分析器所完成的功能是直接产生新的程序或者将旧程序移植到另一种机器上，因而就仿真器软件运行速度而言很少有定论，因此这不是这种形式仿真器特有的问题。即，程序员通常对仿真器产生的代码在主机上运行得有多快并不感兴趣，他们感兴趣的是仿真器是否能产生可在为其设计的机器上执行的代码并能在该机器上快速地运行的代码。因此除了编程目的以外，这种类型的仿真软件不能提供使第一指令集编写的应用程序能运行在不同类型的微处理器上的方法。这种仿真软件的实例在题为“Shade: A Fast Instruction-Set Simulator for Execution Profiling”(Cmelik 和 Keppel)的文章中有所论及。

因此最理想的是提供一种具有竞争力的微处理器，它比现有技术的微处理器更快更便宜，但是又与为现有技术微处理器运行各种可用操作系统而设计的目标应用程序完全兼容。

尤其，最理想的是提供一种电路系统，它提供的主处理器在处理主处理器存储内容时能够迅速从意外事件中恢复过来，从而提高了处理器的功能速度。

发明内容

因此本发明的目的是借助加速程序执行期间产生的存储器内容的装置提高微处理器的运算速度。

实现本发明的本自的和其他目的的装置和方法包括：暂时保存除其它存储器内容以外的所有存储器内容，它们在主处理器将目标指令翻译为主指令的翻译间隔期间依照顺序产生；如果翻译执行不产生意外事件则将翻译间隔期间按照顺序产生的存储器内容转移的电路系统；响应于存储器地址的存储器寻址指示哪一个至等同存储器地址的存储器内容是最新的电路系统；以及如果翻译执行不产生意外事件则将翻译间隔期间按照顺序产生的存储器内容消除的电路系统

通过以下附图对本发明的详细描述可以更好地理解本发明的各种目的和特

点，在附图中相同的单元采用相同的标记。

附图的简要说明

图 1(a)-(e)为按照现有技术设计的微处理器操作方式的示意图。

图 2 为微处理器框图，该处理器运行针对不同微处理器设计的应用程序。

图 3 为图 2 所示微处理器某一部分的示意图。

图 4 为寄存器文件的框图，该寄存器文件用于图 3 所示的微处理器。

图 5 为按照本发明设计的门控存储缓冲器的框图。

图 6(a)-(c)示出了用于现有技术各种微处理器和图 3 所示微处理器的指令。

图 7 示出了图 3 所示微处理器软件部分实现的方法。

图 8 示出了图 3 所示微处理器软件部分实现的另一种方法。

图 9 为改进的图 3 所示计算机系统的框图。

图 10 为图 3 所示微处理器某一部分的框图。

图 11 为图 3 微处理器中翻译旁视缓冲器的更为详细的框图。

图 12 为图 3 所示处理器中所用存储器的详细框图。

图 13 为按照本发明的门控存储缓冲器第一逻辑实施例的框图。

图 14 为按照本发明的门控存储缓冲器第二实施例的框图。

记号和术语

以下某些详细描述部分将借助对计算机存储器内数据位操作所作的符号化表示，这些描述和表达形式是数据处理领域内技术人员向同行表述其工作内容最为有效的方式。这些操作需要对物理量施行物理操作。虽然并非必要，但是通常情况下这些物理量采用可以存储、转换、组合、比较和其他处理方式的电学或磁学信号形式。为方便起见，主要是通用的原因，这些信号被称为位、值、单元、符号、字符、项、数字等。但是应该认识到，无论如何，所有这些以及相似的术语都与合适的物理量相联系并且仅仅是为方便表示这些物理量而采用的标记。

还有，所完成的控制通常用诸如加法或比较之类的术语表示，它们一般与操作人员所完成的思维操作有联系。在构成本发明的大多数操作中，操作人员的这种能力并不是必不可少或需要的；这些操作是机器操作。完成本发明操作的有用机器包括通用数字计算机或其它类似设备。在所有情况下，应考虑到操纵计算机

的操作方法与计算方法本身之间应视为是有区别的。本发明涉及操作计算机的方法和装置，这种计算机处理电学或其它(例如机械、化学)物理信号以产生其它所需的物理信号。

在下面的描述中，由于大多数的仿真器都运行 X86 应用程序，所以为了举例说明操作细节，在某些情况下将目标程序看作为在 X86 微处理器上执行而设计的程序。但是目标程序可以为在任意目标计算机系列上运行而设计。它们包括目标虚拟计算机，例如 Pcode 机、Postscript 机或者 Java 虚拟机。

实施发明的较佳方式

本发明有助于克服现有技术存在的问题并且提供了一种速度比现有技术的微处理器更快的微处理器，它能够运行所有(大量现有技术微处理器都可以运行的)操作系统下的所有软件，但是价格又低于现有技术的微处理器。

与采用更为复杂的硬件加快运算速度的做法不同，处理器是组合的一个部分，包括比现有技术微处理器简单得多的增强硬件处理部分(在本说明书中称为“词态主机(morph host)”；以及仿真软件部分(在本说明中称为“代码词态化软件(code morphing software”)，二者结合成为比已知的有竞争力的微处理器功能更强大的微处理器。词态主机是一种包含硬件增强的处理器，当发生意外事件或者错误时它立即恢复为目标计算机的状态，而代码词态化软件将目标程序的指令翻译为词态主机的词态主机指令，并在需要用正确的目标状态代替工作状态以响应意外事件或错误，从而进行正确的重新翻译。代码词态化软件还可以包括提高处理速度的各种过程。与所有速度极快的现有技术微处理器通过提供硬件来提高处理速度的做法不同，改进的微处理器借助代码词态化软件在可以选择的阶段运用大量的加速增强技术。假定在代码词态化软件中运用提速技术使得可以采用复杂程度较低但运行速度比现有技术微处理器更快而价格大为降低的硬件实现词态主机。作为比较，在一个用包含 Pentium Pro 微处理器四分之一数量门电路的词态主机实现的微处理器的实施例中，运行 X86 应用程序的速度要快于 Pentium Pro 微处理器或者其它能够处理这类应用程序的已知微处理器。

代码词态化软件采用了某些技术，这些技术以前只是由程序员在设计新软件或仿真新硬件时采用过。词态主机包含硬件增强部分，它特别适合于充分利用代码词态化软件提供的加速技术。这些硬件增强部分允许代码词态化软件在更宽的

指令范围内实现加速技术。这些硬件增强部分还允许代码词态化软件实现其它加速技术，这些技术在硬件处理器中不具备，且无法在硬件处理器内应用，除非花费巨额的代价。与现有技术微处理器执行本机指令集的情形相比，这些技术明显提高了微处理器的速度。

例如，与增强型词态主机结合的代码词态化软件可以采用重新排列与重新安排由目标指令序列产生的基本指令的技术而无需增加太多的电路。由于可以一起重新排序与重新安排大量目标指令，所以可以采用其它的优化技术来减少处理器执行目标指令组所需的步骤数，使其少于其它运行目标应用程序的微处理器所需的步骤数。

与增强型词态主机组合起来的代码词态化软件快速将目标指令翻译为词态主机的指令并且将这些主指令高速缓存入存储器数据结构(在本说明书中称为“翻译缓冲器”)。使用保存翻译指令的翻译缓冲器可以再次调用指令而无需在每次执行每条目标指令时，重新运行冗长的过程，确定需要哪一条基本指令来实现每条目标指令，对每条基本指令作寻址、读取基本指令、优化基本指令序列，为每条基本指令分配资源，对基本指令进行重新排序并执行包括每一时间执行的每一目标指令的每一基本指令序列的每个步骤。目标指令一旦被翻译出来，它就可以从翻译缓冲器内再次调用并执行而无需再三重复这些数不清的步骤。

现有仿真技术的主要问题是无法高性能地处理目标程序执行期间产生的意外事件。如果在运行直接指向目标操作系统的目标应用程序时产生意外事件，情况更是如此，此时为了正确处理意外事件和执行随后的指令，必须有发生意外事件时正确的目标状态供使用。因此迫使仿真器始终精确地跟踪目标状态并一直进行检查，以确定是否存储到目标代码区域。其它意外事件也会带来类似的问题。例如仿真器可以产生意外事件以检测已经被一些特定主功能替代的特定目标操作。特别是目标处理器的各种硬件操作可以由仿真器软件提供的软件操作代替。此外，执行由目标指令导出的主指令的主处理器也可能产生意外事件。所有这些意外事件都有可能在仿真器试图将目标指令变换为主指令时或者在主处理器上执行主翻译时发生。高效的仿真必须提供一些从这些意外事件有效恢复的方式并且能够正确处理意外事件。现有技术均无法对所有被仿真的软件做到这一点。

为了克服现有技术的这些不足，在增强型词态主机中作了大量的硬件改进。这些改进包括门控(gated)存储缓冲器和许多新增加的处理器寄存器。某些新增加

的寄存器可以采用寄存器重新命名以缓解需要同一硬件资源指令的问题。新增的寄存器还可以维护一组处理主指令的主机或工作寄存器和一组目标寄存器以保持目标处理器(用它产生目标应用程序)的正式状态。目标(或屏蔽)寄存器通过专用接口与与其等效的工作寄存器相连,专用接口使得称为“调拨”的操作将所有工作寄存器的内容快速传递至正式目标寄存器,并使得称为“返还”的操作将所有正式目标寄存器的内容快速传递回与其等效的工作寄存器。门控存储缓冲器存储硬件“门电路”的“未调拨”一侧的工作存储器状态变化,和硬件门电路“调拨”一侧的正式存储器状态变化,这些调拨的存储“排放”至主存储器。调拨操作将门电路为非调拨一侧的存储传递至门电路的调拨一侧。新增的正式寄存器和门控存储缓冲器使存储器状态和目标寄存器状态在一条或一组目标指令被翻译和运行无误之后一起更新。

这些更新由代码词态化软件选择以发生于完整的目标指令边界。因此,如果由主处理器运行构成目标指令系列翻译的基本主指令而未发生意外事件,则将这些指令产生的工作存储器存储和工作寄存器状态传递至正式存储器和正式目标寄存器。这样,如果在处理主指令时发生意外事件而此时又不在被翻译的一条或一组目标指令的边界上时,可以将最近更新(或调拨)的目标寄存器的原始状态再次调用至工作寄存器并可以转储门控存储缓冲器内未调拨的存储器存储结果。如果发生的意外事件是目标意外事件,则可以一次重新翻译一条引起目标意外事件的目标指令并象目标微处理器执行目标指令那样依照顺序执行目标指令。当正确无误地执行每条目标指令时,可以更新目标寄存器的状态;并且存储缓冲器内数据受到门电路的控制向存储器传送。随后,当运行主指令时再次发生意外事件时,目标计算机的正确状态由词态主机的目标寄存器和存储器保存;并且可以无延迟地正确地处理操作。由这种纠错翻译产生的每条新翻译结果可以高速缓存以供以后使用。因为是对一次性或偶尔发生的诸如页面出错事件而翻译或者交替转储的,这使得代码词态化软件与词态主机组合构成的微处理器能够比原先为其编写软件的处理器更快地执行指令。

应该指出的是,在利用本发明的微处理器执行目标程序的过程中,可能会发生许多不同类型的意外事件,它们需要不同方式处理。例如一些意外事件由产生意外事件的目标软件引起,该意外事件使用了目标操作系统意外事件处理器。使用这种意外事件处理器要求代码词态化软件包含仿真整个意外事件处理过程的

子程序，该以外事件处理过程包含任何由处理过程的目标计算机提供的硬件。这就需要代码词态化软件保存目标处理器的状态从而在处理完意外事件之后继续正确执行。某些意外事件，例如页面出错，需要在实现将过程翻译之前在新的存储器页面中读取数据，当意外事件处理之后它要求返回至被翻译过程的开始之处。其它意外事件实现了软件中硬件未提供的特殊操作，这些意外事件都要求意外事件处理器在处理完意外事件之后将操作返回翻译中的下一步骤。每种不同类型的意外事件都可以由本发明的微处理器有效地处理。

此外，有些意外事件由主机硬件产生并且检测各种主机和目标机状态。有些意外事件的表现如同在普通微处理器上一样，而其它的则被代码词态化软件用来检测各种预测的故障。在这些情况下，采用上述状态保存和恢复机制的代码词态化软件使目标状态恢复至其最近的正式状态并将产生和保存新的翻译结果(或者再次使用先前产生的安全的翻译结果)，这避免了无效预测。随后执行该翻译结果。

词态主机包含新增的硬件意外事件检测机构，它与上述返还和重新翻译方法结合可以作进一步的优化。例子为有一种装置，它将存储器从存储器映象 I/O 区分开来，另有一种装置，它通过保护地址或地址范围省却了存储器基准，由此可以在寄存器内保存目标变量。

在利用意外事件检测其它预测失效(例如操作是否影响存储器或存储器映象 I/O)时，通过借助不同的存储器操作和不同的优化产生的新的翻译结果来完成恢复工作。

图 2 为词态主机硬件的示意图，它正在运行与图 1(a)上 CISC 处理器相同的应用程序。如图所示，微处理器包含上述代码词态化软件部分和增强型硬件词态主机部分。目标应用程序将目标指令载入代码词态化软件以将其翻译成为词态主机能够执行的主机指令。与此同时，目标操作系统接收来自目标应用程序的调用并将调用转送至代码词态化软件。在微处理器的较佳实施例中，词态主机是一种超长指令字(VLIW)处理器，将它设计成有多条处理通道。图 6(c)示出了这种处理器的总体操作。

在图 6(a)-(c)中示出了适用于 CISC 处理器、RISC 处理器和 VLIW 处理器的指令。显然 CISC 指令是变长度的并且可以包含多条更基本的操作(例如加载和相加)。另一方面，RISC 指令是等长度的并且主要是基本操作。图示的 VLIW 处理

器的一条超长指令包括 CISC 和 RISC 指令的每个更为基本的操作(例如加载、存储、整数相加、比较、浮点乘法和转移)。如图 6(c)所示, 每条基本指令一起构成一条超长指令字, 所述基本指令与其它基本指令并行地载入 VLIW 处理器中多条分开的处理通道中的一条或者存储器内以供处理通道和存储器并行处理。并行操作的所有结果都被转送至多端口寄存器文件。

可作为词态主机基础的 VLIW 处理器是比上述其它处理器简单得多的处理器。它不包含检测结果依存性的电路或者重新排序、优化和重新安排基本指令的电路。与运行原先为其设计目标应用程序的处理器或者利用仿真程序运行目标应用程序的其它处理器相比, 这使得可以在较高的时钟频率有更快的处理速度。但是这并不局限于 VLIW 处理器, 诸如 RISC 处理器之类的任意类型的处理器都可以实现同样的效果。

图 2 所示微处理器的代码词态化软件包括翻译机部分, 它对目标应用程序的指令进行译码, 将目标指令转换为词态主机能够执行的基本主指令, 优化目标指令所需的操作, 将基本指令重新排序和重新安排为用于词态主机的 VLIW 指令(翻译)并执行主 VLIW 指令。图 7 为翻译机操作示意图, 它示出了代码词态化软件主循环操作。

为了加速包含代码词态化软件和增强词态主硬件的微处理器的操作速度, 如图 2 所示, 代码词态化软件包含翻译缓冲器。一个实施例的翻译缓冲器是一种可以在存储器内存储的软件数据结构; 在特殊的实施例中也可以采用硬件高速缓存。用翻译缓冲器存储主指令, 主指令概括了每一个目标指令的完整的翻译。显而易见, 一旦翻译出单条目标指令并对获得的主指令进行优化、重新排序和重新安排, 所得到的主指令就被存储在翻译缓冲器内。构成翻译结果的主指令随后由词态主机执行。如果主指令的执行不发生意外事件, 则无论何时需要实现由目标指令或指令的操作时都可以再次调用该翻译结果。

如图 7 所示, 由应用程序载入目标指令地址的微处理器词态化软件的典型操作是首先确定目标地址上的目标指令是否已经翻译。如果目标指令未被翻译, 则读取该目标指令以及随后的目标指令, 并且将它们译码、翻译并随后(可能是)优化、重新排序和重新安排为新的主翻译结果, 并由翻译机将它们存储在翻译缓冲器内。如下可见, 优化可以达到不同的程度。在本说明书中术语“优化”常常指的是加速处理的技术, 例如重新排序就是一种优化形式, 它使得处理更快, 因此

属于该术语的范畴。许多可能的优化方法在现有技术的编译器优化中都有描述，有些优化方法难以在类似现有技术的“超块”中完成，而在 VLIW 研究中出现。随后控制被转移至翻译以使增强词态主硬件重新开始执行。

在运行应用程序中当接着遇到特定的目标指令序列时，将在翻译缓冲器内找到主翻译指令并立即执行而无需翻译、优化、重新排序或者重新安排。利用下述高级技术，据估计每一百万次中只有一次需执行翻译。可在翻译缓冲器中找到所有目标指令(曾一旦全部经过翻译)的翻译结果。因此在第一次翻译之后，翻译所需的所有步骤(例如译码、读取基本指令，优化基本指令，将其重新安排为主翻译指令并在翻译缓冲器内存储)可以从所要求的处理中省略。由于为其编写目标指令的处理器在每次执行指令时都必须译码、读取、重新排序和重新安排每条指令，所以这样做大大减少了执行目标指令所需的工作量并提高了改进型微处理器的速度。

在省略所有这些现有技术处理器执行目标应用程序所需的步骤之后，本发明的微处理器克服了现有技术的问题，这些问题使得这类操作不能合理的速度运行。例如改进型微处理器的某些技术被用于上述仿真器以将应用程序移植到其它系统。但是由于在处理翻译指令时，引起调用各种系统意外事件处理器的意外事件发生在主处理器状态与处理同样指令的目标处理器无关的操作时刻，所以一些仿真器无法运行应用程序较长的部分。因此产生这类意外事件时目标处理器的状态是未知的。这样，目标机的正确状态就无法确定；操作必须停止，重新启动，并在意外事件可以处理和继续执行之前确认正确的状态，这使得无法以主机速度运行应用程序。

词态主机硬件包含了解决该问题的多个增强部分。图 3、4 和 5 示出了这些增强部分。为了在发生错误时确定寄存器的正确状态，增强硬件提供了一组正式目标寄存器来保存为其设计源应用程序的目标处理器的寄存器状态。目标寄存器可以包含在每个浮点单元、任何整数单元和其它执行单元内。这些正式寄存器连同增加的正常工作寄存器一起被加入词态主机使得包括寄存器重命名的一系列优化得以实现。增强硬件的一个实施例在整数单元内包含 64 个工作寄存器而在浮点单元内包含 32 个工作寄存器。实施例还包含一组增强目标寄存器，它们包含所有提供处理器状态的经常变化的目标处理器寄存器；这包括状态控制寄存器和其它控制仿真系统所需的寄存器。

应该注意的是，根据词态主机所用的增强处理硬件类型，翻译指令序列可包含构成从源应用程序来的多条目标指令的基本操作。例如 VLIW 微处理器可以如图 6(a)-(c)所示立即执行多条 CISC 指令或者 RISC 指令。无论词态主机类型如何，除了整体目标指令边界以外词态主机硬件的目标寄存器状态是不会改变的；并且随后所有的目标寄存器得到了更新。因此，如果微处理器正在执行目标指令或已经被翻译为一系列可以重新排序和重新安排为主翻译指令的基本指令，则当处理器开始执行已翻译的指令序列时，正式目标寄存器在第一条目标指令被寻址时保存由为其设计应用程序的目标处理器寄存器保存的值。但是在词态主机开始执行翻译指令之后，工作寄存器保存由执行到该点的翻译指令基本操作确定的值。因此尽管一些工作寄存器保存的是与正式目标寄存器内相同的值，但是其它工作寄存器内保存的值对于目标处理器毫无意义。在为了采用高级加速技术提供位定目标机更多的寄存器的实施例这尤其是如此。一旦开始执行经翻译的主指令，工作寄存器中的值是经翻译的主指令确定寄存器状态的任何值。如果执行一组经翻译的主指令而未产生意外事件，则在这组指令结束时确定的新工作寄存器值就被一起转送至正式目标寄存器(可能包括目标指令指针寄存器)。在本实施例的处理器，这种转移发生于主指令执行的外部在附加的流水线站内，从而不会降低词态主机的处理速度。

同样，图 5 所示实施例中的门控存储缓冲器在改进型微处理器的硬件中被用来控制数据转移至存储器。门控存储缓冲器包括多个单元，每个相当于一个存储单元，可保存存储器存储操作的地址和数据。可通过任意数量的不同的硬件配置(例如先进先出缓冲器)执行这些单元所示实施例利用随机存取存储器和三个专用工作寄存器执行。三个专用寄存器分别存储指向存储器存储队列首部的指针、指向门的指针和指向存储器存储队列尾部的指针。指针还存储在分立的寄存器内(图 5 中未画出)，该寄存器指定将数据读取至存储器的单元。存储单元内位于队列首部与门之间的存储器存储内容已经被调拨给存储器，而与其它存储单元相隔的存储单元内位于队列门与尾部之间的还未调拨给存储器。在主翻译指令执行期间产生的存储器存储内容由整数单元将按照词态主机执行主指令的顺序放入存储缓冲器但是在主指令中遇到调拨操作以前不允许写入存储器。因此随着翻译指令的执行，存储操作被放入队列。假定这些是第一次存储的内容因而在门控存储缓冲器内没有其它的存储内容，则首部指针和门指针将指向同一位置。随着执行每一

存储内容，它被放入队列中下一位置并且尾部指针指向下一个位置(在图中是向上)。这一直持续到执行调拨指令为止。这通常发生于完成一组目标指令翻译而未发生意外事件或者出现错误退出条件时。当词态主机正确无误地执行翻译指令时，执行期间产生的存储缓冲器内的存储器存储内容被一起转移通过存储缓冲器的门(给调拨的)并随后写入存储器。在所示实施例中，这是通过将保存尾部指针的寄存器内的值复制到保存门指针的寄存器内而完成的。

由此可见，寄存器状态从工作寄存器转送至正式目标寄存器和工作存储器内容转移至正式存储器是一起发生的并且只发生在整个目标指令之间的边界上对明确的调拨操作作出响应。

这使得微处理器可以从在增强词态主机执行指令过程中发生的目标意外事件中几乎无延迟地恢复过来。如果在运行任何翻译指令或指令期间产生目标意外事件，则由词态主机硬件或软件检测意外事件。为了响应检测到目标意外事件，代码词态化软件可以将正式寄存器内保存的任何值放回工作寄存器并使门控存储缓冲器内未调拨的存储器存储内容转储(一种称为“转储”的操作)。图 5 在门控存储缓冲器内的存储器存储内容的转储可以通过将保存门指针的寄存器内的值复制到保存尾部指针的寄存器内实现。

将目标寄存器的值放入工作寄存器可以使发生意外事件时的第一条正在运行的目标指令的地址放入工作指令指针寄存器内。从工作寄存器内目标处理器的正式状态开始，当发生意外事件时正在运行的目标指令将按照串行顺序重新翻译而不进行重新排序或其它优化。在将每条目标指令重新译码和翻译为新的主指令之后，代表目标指令的翻译后的主指令由词态主机执行并且可能引起意外事件或不引起意外事件。(如果词态主机是 VLIW 处理器以外的处理器，则主翻译指令的每条基本操作按照顺序执行。如果在主翻译指令执行时没有产生意外事件，则运行下一基本功能)。这一直延续到再发生意外事件为止或者已经翻译和执行单条目标指令为止。在一个实施例中，如果在执行目标指令翻译时没有产生意外事件，则工作寄存器的状态被转移至目标寄存器并且门控存储缓冲器内的数据被调拨从而可以转移至存储器。但是如果在翻译指令运行期间再次发生意外事件，则目标寄存器和存储器的状态未变化而是与发生意外事件时目标计算机内产生的状态一致。因此当产生目标意外事件时，意外事件将由目标操作系统正确处理。

同样，一旦指令系列在翻译时产生意外事件的第一条目标指令执行时未发生

意外事件，则目标指令指针指向下一目标指令。这第二条目标指令与第一条指令的处理方式一样，只作译码和重新翻译而不进行优化或重新排序。随着词态主机处理单条目标指令的每条主翻译指令，当目标寄存器和存储器的状态与目标计算机内发生的状态一致时将产生意外事件。因此意外事件得到立即而正确的处理。这些新的翻译指令可以存储在翻译缓冲器内作为目标应用程序内指令序列的正确翻译结果，并在指令再次执行时被再次调用。

完成与图 5 门控存储缓冲器同样结果的其它实施例包括这样的方案，它将存储内容直接转移至存储器而与此同时记录足够的数据以恢复目标计算机的状态。如果在意外事件或者错误需要返回时执行翻译结果。在这种情况下，任何在翻译和执行期间发生的存储器存储内容的影响都不得不逆转过来并且恢复翻译开始时存储器的状态；与此同时工作寄存器必须以上述方式接收正式目标寄存器内保存的数据。在实现这项操作的一个实施例中，维持一个分立的目标存储器保存原始的存储器状态，如果需要返回，则该状态被用来代替改写的存储器内容。在实现存储器返回的另一实施例中，在每条存储和存储器数据被替换时对它们进行登录并在需要返回时使存储过程逆向进行。

实现上面详述的门控存储缓冲器存在许多问题。例如，在微处理器运行期间数据可以经常地存储在同一存储器地址上。在这种存储数据实际传送至存储器之前的任意时刻，微处理器可能需要将数据用于其它操作。通常情况下，所应该用的数据是最新可得到的有效的数据。在典型的存储缓冲器中，转移至缓冲器内同一存储器地址的数据作为已在缓冲器内的数据的地址通常存储在旧数据的位置上。因此，无需搜索该地址上的有效数据就可以立即从缓冲器调用最新的数据。

但是由于缓冲器专门设计为保存还未完成操作的存储内容，所以上述门控存储器要做到上述情况是不可能的。在这些存储内容调拨入存储器之前，必须确定产生存储内容的指令序列是否正确执行。因此，通过将缓冲器内较近发生的存储器内容代替已经调入的较早存储器内容，将在完成产生新存储数据的主指令执行之前高效率地调拨新的存储数据而不会发生意外事件。这完全与门控存储缓冲器的用途背道而驰。

因此，在门控存储缓冲器的一个实施例中，新的存储器内容并不代替同一地址上旧的存储内容，而当它们产生时是按顺序存储在缓冲器的分立的部分或隔离的部分；而且在任意时刻，门控存储缓冲器内可能有多个存储内容存储至同一存

存储器地址,有些是调拨入存储器而其它一些还未调拨。门控存储缓冲器内每个存储位置包括一些存储位置内数据有效性的标记。在图 13 所示的逻辑布局实施例中,对于门控存储缓冲器存储位置上的每个数据字节(实施例中可寻址的最小数据量)提供了有效位(用 V 表示的列)。一般采用 64 位存储每个缓冲器存储位置上的数据,因此缓冲器内每个存储位置包括带自身有效位的 8 个单独字节。

为了区分保存在缓冲器内存储器地址上最近存储的数据,每个存储位置的每个字节提供了第一位(用 M 指定的列),它被用于指示该字节是存储器地址上最近的数据字节。每个存储位置包括比较电路系统,它测试存储器地址(包括字节地址)和每个新写入门控存储缓冲器的控制位。当新数据存储存储在存储缓冲器的存储位置时,新存储数据的每个字节包含所声明的最近存储位。如果新存储数据的存储器地址与门控存储缓冲器内有效数据的存储器地址一致,则比较电路系统根据字节确定这一点并去除先前最近存储的同一存储器地址上任何字节的最近位标记。这样,通过在特定存储器地址用其所声明的最近位检测数据可以满足将最近数据加载至存储器地址的需要。

在本实施例中显而易见的是,替代现有技术中在门控存储缓冲器的存储器地址上存储的旧数据,为了从该地址加载至微处理器寄存器,通过去除旧的最近位标记消除了旧数据。去除被访问存储器地址上字节的最近位标记使得该字节被忽略而用最近位标记的同一地址的字节从存储缓冲器内读取作加载操作。同时,存储位置上不再有最近位标记的数据保存在存储缓冲器内从而可以最终按照门控存储缓冲器内的顺序排序调拨入存储器。

由于为了以上述方式加速微处理器的性能,门控存储缓冲器内的数据受调拨和返回操作的支配,所以在返回操作期间未调拨存储的数据可以转储(消除)。该数据总是比保存在门控存储缓冲器的同一存储器地址上任何的调拨数据更新。因此如果调拨数据存在于门控存储缓冲器相同存储器地址的另一存储位置上作为被转储的未调拨数据,则在返回之后,存储器地址上的最新调拨数据在存储缓冲器的存储器地址上仍然是最新的并且应该这样标明。但是当在门控存储缓冲器内同一地址上,放置较新的未调拨数据(已经被转储)时,将去除该调拨数据的最新位标记。

为了确保存储器地址的最新数据在返回和调拨操作后保持正确的指定,每个字节不仅有一个最近位位置而且还有第二个“屏蔽最近位”位置(图 13 中用“S”

指定)。当通过将门位置移动至缓冲器尾部来调拨门控存储缓冲器内一个或多个存储位置上的数据时,存储在缓冲器内每个地址上的最近数据包含被标记的屏蔽最新位并且与此同时去除同一存储器地址上其它数据的屏蔽最近位标记。当这情况发生的时刻,新调拨的数据实际上是存储器地址上的最近数据,所以所需要的是将与被调拨的每个存储位置的每个字节相关的最近位复制到该字节的屏蔽最近位。当较新的数据被写入同一存储器地址的存储缓冲器时,被调拨数据的最近位被去除标记而屏蔽最近位仍然保留标记。随后,如果接着发生返回从而使得更近的调拨至该地址的数据被转储,则被调拨数据最近的屏蔽最近位被复制至调拨字节的最近位位置从而正确指示最近调拨数据实际上是最近的存储数据。

因此在上述实施例中,显而易见的是用来写入门控存储缓冲器的逻辑将新数据写入下一存储器位置并赋予写入该存储位置的每个字节以有效和最近位标记。逻辑比较写入缓冲器内存储位置的存储器地址的数据的存储器地址以确定该地址上的旧数据是否在缓冲器内。如果发现这样的数据,则去除该地址的旧数据的最近位标记。在调拨操作中,所有带最近位标记的数据包含屏蔽最近位标记;并且所有带屏蔽最近位标记的数据包含去除的位标记。在返回操作中,未调拨数据被转储并且对带屏蔽最近位的调拨数据作最近位标记。

图示的实施例可以利用新增的寄存器将指针存储到存储器存储位置。接下来将抽取该存储位置的数据主存储器,从而使得数据被依次抽取入存储器。随着将每个存储位置上的数据抽取至存储器,为了保持正确的抽取序列,该数据的有效位、最近位和阴影最近位被清除。为了通过利用逻辑比较所有存储到每个存储器地址上的调拨数据并只抽取存储器地址的最近调拨数据至存储器而忽略不是最近的调拨数据来缩短抽取至存储器的操作,也可以采用指向被抽取存储位置的新增寄存器。

图 14 示出了门控存储缓冲器的第二实施例。该实施例利用多组存储位置,每组保存足够数量的存储位置来保存目标指令序列典型的完整翻译产生的存储内容,一个以上的组被用来存储翻译指令。与图 13 的实施例一样,一组中的每个存储位置保存存储数据的多个字节、存储器地址和每个字节的有效位。每组存储位置可以指定为工作组(例如通过赋予组以位 W),其中当前存储了所产生的存储数据。工作组中的数据全部未被调拨,但是每组排列为相关的高速缓冲存储器寻址从而使新的写入特定存储器地址的存储数据覆盖已存在于工作组内的旧存

储数据并写入同一存储器地址。在所示实施例中是利用在每个存储位置上的比较器完成的。这样，在存储位置组的仅仅一个存储位置上有寻址至存储在组内的任何存储器地址的数据。因此如果数据需要在调拨前由主处理器加载，则可以通过测试组内存储器地址寻找。这省去了前面实施例中所用的最近位和屏蔽最近位。

如果执行目标指令序列而不产生意外事件，则工作组内该序列产生的数据存储内容被一起调拨。利用标记与存储位置特定组相关的调拨位(位 C)并去除该组的工作位(位 W)可以完成这种调拨(它等效于前述实施例中将门指针移向尾部指针的位置)。一旦调拨了一组存储位置，这些存储位置上的数据就保留在该组内直到被抽取至存储器。因此省去了保存首部、门和尾部指针以指定调拨或未调拨存储内容的电路系统而利用工作和调拨位获得同样的结果。

因此如果在执行翻译的目标指令序列时产生了需要返回的意外事件，则简单地消除存储在工作组内的数据(转储)。例如为此可以去除作为 W 位存储的工作指示而不去除调拨位来实现。实际上，去除工作指示等效于前面实施例中将尾部指针移动至门指针位置。

上述两个实施例的各种组合对于本领域内技术人员来说是显而易见的。例如利用允许只在门与尾部(数据还未调拨)之间的存储位置内用新数据存储内容覆盖旧数据的逻辑可以缩小第一实施例的规模。同样，可以利用逻辑检测同一地址的所有经调拨的存储内容并按照上面的表示利用抽取指针指向最近调拨数据的方式仅仅将最近(字节)写入存储器。

也可以将上述实施例的细节与典型的处理器高速缓冲存储器结合从而用单一电路完成两种用途。在这种情况下，需要提供手段来指示存储调拨的和未调拨的存储器内容的高速缓存以及其它控制指示符的指示线。

代码词态软件提供了一项新增的操作，它大大加快了被翻译的处理程序的速度。除了简单翻译指令、优化、重新排序、重新安排、高速缓存和执行每条翻译指令从而可以在需要执行该组指令时再次运行以外，翻译器还链接不同的翻译结果以在绝大多数情况下避免返回翻译过程的主循环。图 8 示出了完成链接过程的代码词态化软件翻译器部分所执行的步骤。对于本领域内的普通技术人员来说，显而易见的是这种链接操作基本上在大多数指令翻译时避免了返回主循环，这节约了开销。

为了示例的目的的运行的目标程序由 X86 指令组成。当翻译目标指令序列并

且重新排序和重新安排基本主指令时，可能在每条主翻译指令结束产生两条基本主指令。第一条是一基本指令，它更新目标处理器的指令指针（或其等效的）的值；该指令用来将下一目标指令的正确地址放入目标指令指针寄存器。该基本指令之后是一条转移指令，它包含转移的两种可能目标中的每一个的地址。处理转移指令的基本指令可以更新目标处理器的指令指针值，其方式是测试在条件代码寄存器内的转移条件代码，并随后确定控制转移的条件所指示的两个转移地址中一个是否存入翻译缓冲器内。第一次翻译目标指令序列时，主指令的两个转移目标都保存翻译机软件主循环的同一主处理器地址。

当主翻译指令完成翻译，存储在翻译缓冲器并第一次执行时，目标指令指针寄存器（作为其余目标寄存器）内的指令指针得到更新；并且操作转移返回主循环。在主循环内，翻译器软件在目标指令指针寄存器内查找指向下一目标指令的指令指针。随后对下一目标指令序列寻址。假定尚未对该目标指令序列进行过翻译，因此在翻译缓冲器内未驻留翻译指令，则从存储器读取下一组目标指令，对其译码、翻译、优化、重新排序、重新安排、高速缓存入翻译缓冲器并予以执行。由于第二组目标指令跟随在第一组目标指令之后，所以第一组目标指令的主翻译结束处的基本转移指令就被自动更新替代为第二组目标指令的主翻译指令地址成为控制转移的特定条件转移地址。

如果这样，则第二经翻译主指令循环返回到第一条经翻译的主指令，第二翻译指令结束处的转移操作包含主循环地址和第一条翻译指令的 X86 地址作为转移两种可能的目标。前述转移的更新指令指针基本操作测试状态并确定循环返回到第一翻译指令以及将目标指令指针更新为第一条翻译指令的 X86 地址。这使得翻译器在翻译缓冲器内查找以确定所寻找的 X86 地址是否在那里出现。第一条翻译指令的地址被找到，并且它在主机存储器空间内的值被第二条主翻译指令结束时转移中的 X86 地址上的值替换。随后第二主翻译指令被高速缓存和执行。这使得循环延续下去直到从第一条翻译指令转移到第二条翻译指令的条件失败，并且转移取道返回主循环。当发生这种情况时，第一条翻译主指令转移返回主循环，从而在翻译缓冲器内搜索目标指令指针指定的下一组目标指令，从高速缓存中读取主指令；或者在翻译缓冲器内找不到，则从存储器内读取该目标指令并进行翻译。当该经过翻译的主指令高速缓存入翻译缓冲器内时，其地址代替了结束循环的转移指令内的主循环地址。

这样，各种翻译主指令被互相链接起来，因此只有在这种链接不存在时才需要历经通过翻译器主循环的长路径。最终，主指令的转移指令内主循环参考地址几乎可以完全省略。当达到该条件时，在运行任何主指令之前读取目标指令、译码目标指令译码、读取构成目标指令的基本指令、优化这些基本操作、基本操作重新排序和这些基本操作重新安排所需的时间可以省去。因此与所有每次执行应用程序指令时必须执行这些步骤的每一步的所有现有技术微处理器相反，利用改进型微处理器在第一条翻译指令执行之后运行任意目标指令组所需的工作量大为减少。当每组经翻译的主指令与其它组经翻译的主指令都链接起来时，工作量可进一步减少。实际上，据估计在应用程序运行期间一百万条翻译指令需要执行的翻译次数不超过一次。

本领域内技术人员将会发现，由于为了避免再次翻译，被翻译的每组指令都被高速缓存，所以微处理器需要大量的翻译缓冲器。针对不同系统编程的应用程序功能而设计的翻译机将根据支持的缓冲存储器不同而有所不同。但是针对运行 X86 程序设计的微处理器实施例采用 2Mb 的随机存储器作为翻译缓冲器。

两种新增的硬件增强手段有助于提高本发明微处理器处理应用程序的速度。第一种手段是将异常/正常(A/N)保护位，与每条地址翻译指令一起存储在翻译指令旁视缓冲器(TLB)(参见图 3)中，在上述缓冲器中首先查找目标指令的物理地址。翻译指令内的目标存储器操作可以分为两类，一种是对存储器操作(正常操作)而另一种是对存储器映象 I/O 设备操作(异常操作)。

作用于存储器的正常存取以正常方式完成。当指令在存储器上操作时，指令的优化和重新排序是有益的并且大大加快了利用本发明微处理器的任何系统的操作。另一方面，作用于 I/O 设备的异常存取操作经常必须按照这些操作编程时的精确顺序进行而不能省略任何步骤，否则对 I/O 设备有不利的影响。例如某一特定的 I/O 操作可能是清除 I/O 寄存器；如果基本操作顺序出错，则操作结果可能会不同于目标指令命令的操作。由于没有区分存储器和存储器映象 I/O 的装置，所以要用作用于存储器映象 I/O 的翻译指令所用的保守假设处理所有的存储器。这大大限制了优化所能达到的性能。由于现有技术仿真器没有检测被寻址存储器性质预测失效的装置和从这类失效中恢复过来的装置，所以其性能受到限制。

在图 11 所示的微处理器实施例中，A/N 位是可以在翻译旁视缓冲器内设定

的位，以指示是存储器页面还是存储器映像 I/O。翻译旁视缓冲器存储用于存储器存取的页表项，每一项包括被存取的虚拟地址和可以用以找到数据的物理地址以及其它有关该项的信息。A/N 位属于其它信息部分并且表示物理地址是存储器地址还是存储器映像 I/O 地址。尽管作用于存储器的操作似乎属于存储器操作的翻译，但是实际上该操作的翻译是预测其作用于存储器。在一个实施例中，当代码词态化软件首先试图执行需要访问存储器或存储器映像 I/O 设备的指令翻译时，它实际上假定该存取是存储器存取。在另一实施例中，软件可能假定目标指令需要进行 I/O 存取。假定在此之前未对该地址进行过访问，则在翻译旁视缓冲器中将没有相应的页表项；并且在翻译旁视缓冲器中的存取失败。这种失败使得软件进行页表查找并将页表项填入翻译旁视缓冲器的存储位置以向虚拟地址提供正确的物理地址翻译结果。此后，软件将物理地址的 A/N 位输入翻译旁视缓冲器内。随后假定访问的是存储器地址，则试图进行再一次的存取。当试图进行存取时，通过将预先假定的存取类型(正常或异常)与 TLB 页表项内的 A/N 保护位比较，检验目标存储器的参考地址。当存取类型与 A/N 保护不一致时，发生意外事件。如果操作实际上作用于存储器，则在翻译期间可以正确应用上述优化、重新排序和重新安排技术。如果与 TLB 中 A/N 位的比较表明操作作用于 I/O 设备，则操作的执行导致意外事件；并且翻译器一次产生一条目标指令的新翻译结果而不进行优化、重新排序或者重新安排等。同样，如果翻译时错误地将实际上作用于存储器的操作假定为 I/O 操作，则操作的执行导致意外事件；并且利用优化、重新排序和重新安排技术重新翻译目标指令。这样处理器可以出乎寻常地提高性能。

本领域内的技术人员将会发现，利用 A/N 位来确定有关存储器访问还是存储器映像 I/O 设备访问的预测是否失败的技术也可以用来预测存储器映像地址的其它性质。例如可以利用这种正常/异常位来区分不同类型的存储器。本领域内技术人员还可以找到区分存储器性质的其它类似用途。

改进型微处理器预测最多的是翻译内是否发生目标意外事件。这使得与现有技术相比作了明显的优化。首先，目标状态不必在每条目标指令边界上更新而只需在发生在翻译边界的目标指令边界上更新。这省略了在每条目标指令边界上保存目标状态所需的指令。原先无法对安排和去除冗余操作所作的优化现在也变得可行起来。

改进的微处理器适于选择合适的翻译过程。按照上述翻译方法，尽管一组指令是作用于存储器的，但是仍然可以先进行翻译，然后当优化、重新排序和重新安排主指令时，利用翻译旁视缓冲器内提供的 A/N 位状态可能发现是 I/O 设备的地址。A/N 位与表示 I/O 操作的翻译指令地址进行比较后产生错误意外事件，启动软件初始返回储程序，引起未调拨的存储器存储内容转储并将目标寄存器内的值放回到工作寄存器。随后一次对一条目标指令进行翻译而不进行优化、重新排序或者重新安排。这种重新翻译适合于对 I/O 设备进行主翻译。

同样，也可能将存储器操作错误地翻译为 I/O 操作。所产生的错误可以用来引发正确的重新翻译，对指令进行优化、重新排序和重新安排以提供更快的操作。

现有技术仿真器对称为自修改代码的问题一直比较棘手。如果目标程序向包含目标指令的存储器写入内容，这将导致目标指令已有的翻译结果“过时”并不再有效。当这些存储器内容动态地发生时需要对它们进行检测。在现有技术中，这种检测需要借助对每一存储使用额外的指令完成。这个问题的范围要超出程序本身的修改。任何能够向存储器写入内容的机构，例如第二处理器或 DMA 设备，也可能引起这个问题。

现在经过改进的微处理器通过另外的增强词态主机的性能解决这个问题。可以用也存储在翻译旁视缓冲器内的翻译位(T 位)来表示已存在翻译结果的目标存储器页面。T 位可表示目标存储器特定页面包含已存在主翻译的目标指令，如果对这些目标指令重写则它们将变得过时。如果试图向存储器内被保护的页面写入内容，则翻译位的存在将引起意外事件，当代码词态化软件进行处理时将使相当的翻译结果无效或者从翻译缓冲器内去除。T 位还可以用来标记其它目标页面使翻译结果不被重写。

借助图 3 可以理解这一点，它示出了本发明微处理器总体功能单元的框图。当词态主机执行目标程序时，实际上它运行的是代码词态化软件的翻译器部分，它仅仅包括有效地运行在词态主机上的初始未翻译主指令。图中右边示出存储器，它被划分为包含翻译器和翻译缓冲器的主机部分和包含目标指令和数据(包括目标操作系统)的目标部分。词态主机硬件开始执行翻译器时从存储器内读取主指令并将其放入指令高速缓存内。翻译器指令产生对存储在存储器目标部分内的第一条目标指令的读取指令。执行目标读取指令使整数单元在正式目标指令指针寄

寄存器内查找目标指令的首地址。随后在存储器管理单元的翻译旁视缓冲器内访问首地址。存储器管理单元包括调度页面的硬件并提供 TLB 的存储器映象手段。假定 TLB 被正确地映象从而保存了目标存储器的正确页面的查找数据, 则将目标指令指针值翻译为目标指令的物理地址。此时, 对表示目标指令是否完成翻译的位(T 位)状态进行检测; 但是访问是读操作, 并且不会发生 T 位意外事件。还检测了表示是对存储器还是对存储器映象 I/O 访问的 A/N 位的状态。假定后面提及的位表示存储器位置, 则由于不存在翻译结果, 所以在目标存储器内访问目标指令。将目标指令和后续目标指令作为数据转移至词态主机计算单元并在指令高速缓存内存储的翻译器指令的控制下进行翻译。翻译器指令采用重新排序、优化和重新安排技术, 就好象处理作用于存储器的目标指令一样。随后将包含主指令序列的最终翻译结果存储在主存储器的翻译缓冲器内。翻译结果经门控存储缓冲器被直接转送至主存储器中的翻译缓冲器内。一旦将翻译结果存储在主存储器内之后, 翻译器转入随后执行的翻译。所执行的操作(以及后续操作)确定翻译是否对意外事件和存储器作出了正确的假设。在执行翻译之前, 对包含已翻译目标指令的目标页面的 T 位进行设定。该指示提醒指令已经翻译; 并且如果试图向目标地址写入内容, 则将导致意外事件, 可能使得翻译结果无效或者被除去。

如果试图向用 T 位标记的目标页面写入内容, 则产生意外事件并中止写入。在意外事件的响应确认对写入目标存储器地址有关的指令翻译是无效的或者受到保护的, 直到它们被适当更新之后, 写入操作才可以继续。由于没有有效的翻译起作用, 所以有些写入操作实际上不需要做。其它的写入操作则需要, 做一条或更多与寻址的目标存储器相关的翻译被适当标记或者去除。图 11 示出了翻译旁视缓冲器的实施例, 包含保存 T 位指示的每条页面项的存储位置。

新增的加强词态主机性能的硬件电路可以使正常存储在存储器内但经常用于操作执行的数据在执行单元寄存器内被复制(或者“别名”)以便节省数据从存储器内读取或存入存储器内的时间。例如如果存储器内的数据在代码序列执行期间经常被重复使用, 则在每次使用该数据时一般从存储器内将其调出并载入执行单元内的寄存器。为了减少这种频繁的存储器访问所需的时间, 在代码序列执行期间, 数据可以在代码序列开始时一次性地从存储器载入执行单元寄存器。该指定的寄存器在代码序列继续执行期间起着代替存储器的作用一旦这样, 每条将数据从指定的存储器地址载入寄存器的加载操作就变为寄存器。与寄存器之间的复

制操作，这使处理加快了许多；而且即使是复制操作，也可以经常通过进一步的优化而省略。

同样，在执行代码序列期间代码序列的执行常常需要将数据写入存储器地址。为了减少频繁地将存储器内容数据写入同一地址所需的时间，在执行代码序列期间，每次将数据写入存储器地址时，都可以将其转送至指定功能的执行单元寄存器以在代码序列继续执行期间代替存储器空间。一旦指定了执行单元寄存器，则每次数据更改都只需进行简单的寄存器到寄存器之间的传送操作，这比存入存储器地址要快得多。

为了完成这些别名操作，改进的微处理器提供了独特的技术，在图 10 所示的实施例中，词态主机被设计为响应针对指定存储器地址的“加载和保护”命令，在执行代码序列时频繁地使用该地址。词态主机对执行单元 110 内保存存储器数据的工作寄存器 111 进行分配并将存储器地址存入存储器控制单元的特殊寄存器 112 内。工作寄存器 111 可以是以此目的而分配的执行单元内许多寄存器中之一(例如图 4 所示为 8 个工作寄存器)。

当采用存储器别名避免从存储器地址将数据加载入执行单元时，存储器地址上的数据首先被载入寄存器 111 而存储器地址则被放入寄存器 112。随后，利用寄存器 111 内的数据加速执行代码序列。在此期间，通常需要从保存在寄存器 112 内的存储器地址加载数据的每条操作都被从寄存器 111 复制数据所取代。这种操作一直延续到代码序列执行结束(或者以某种其他方式终止)并且去除存储器空间保护为止。

同样，为了加快执行常常将数据从执行单元 110 存到同一存储器地址的代码序列，可以采用类似的别名处理。“加载和保护”命令使存储器地址放入寄存器 112 并将通常存储在该存储器地址上的数据转移至工作寄存器 111。例如在循环执行的计算中通常将一系列数值存储在同一存储器地址上的计算中，通过分配保存数据的寄存器 111 并在寄存器 112 内保存存储器地址可以使存储过程变为执行单元内部的寄存器至寄存器之间的数据转移。这种操作一直延续到代码序列执行结束(或者以某种其他方式终止)，存储空间被更新，并且去除存储器空间保护为止。

虽然这些别名技术都大大提高了某些代码序列的执行速度，但是这些减少存储器访问的操作会引起相当多的问题。如果主处理器操作的基本部分涉及目标指

令集与主指令集之间的指令翻译时情况尤其如此，所有这些问题都涉及到需要确保指令执行时使用的数据在使用时是有效的。

因存储在存储器地址上的数据与存储在执行单元寄存器内的数据不同而造成其中之一在某一时刻失效的例子很多。例如，如果工作寄存器 111 被用来保存代码序列执行期间经常从存储器加载到寄存器的数据，则可能在完成利用执行单元寄存器内数据的代码序列执行完成以前将指令写入存储器地址。在这种情况下，代码序列所使用的执行单元寄存器内数据将会过时并且必须更新。

作为另一个实例，如果工作寄存器被用来保存代码序列执行期间经常存入存储器地址上的数据，则可以在利用执行单元寄存器代替存储器的代码序列的执行完成前将一条指令写入存储器地址。如果主处理器处于只在代码序列执行结束时正常地更新存储器地址上数据的功能模式(回写模式)，则执行单元寄存器内的数据将会过时并且必须用写入存储器地址的数据更新。当然，如果主处理器处于每次将数据写入执行单元寄存器时正常更新存储器地址上数据的功能模式(写入模式)，则寄存器与存储器是一致的。

作为另一个实例，如果工作寄存器被用来保存代码序列执行期间经常存入存储器地址上的数据，则指令可以在完成将数据转移至寄存器 111 的代码序列之前从存储器地址上读取数据。如果主处理器处于只在代码序列执行结束时正常地更新存储器地址上数据的功能模式(回写模式)，则存储器内的数据将会过时并且在读取之前必须用执行单元寄存器内数据更新。如上例所述，如果主处理器处于每次将数据写入执行单元寄存器时正常地更新存储器地址上数据的功能模式(写入模式)，则寄存器与存储器是一致的。

另一种可能性是由于词态主机与代码词态化软件组合形成的微处理器适于重新排序并重新安排以加快执行速度，所以存储器和别名寄存器内保存的数据可能不一致。从下面代码序列的各种实例可见，一旦在执行单元寄存器内的存储器数据被赋予别名以用于代码序列的执行，则执行单元寄存器内数据就可以复制到其它寄存器内并且可能对指令进行重新排序和重新安排处理。如果有重新排序和重新安排处理，则代码序列中的指令可能被写入赋予别名的存储器地址从而在进一步使用之前必须更新执行单元寄存器内的数据。但是，如果执行单元寄存器 111 内刚刚过时的数据已经被复制到其它寄存器内并且利用这些寄存器的指令的代码序列已经变动，则被复制的寄存器内过时数据可能会在代码序列执行时使用。

因此可能发生第二级不一致。

在加载和存储操作后，未确认存储器地址上数据与寄存器 111 内数据一致之前对受保护的存储器地址保证不进行加载和存储操作，存储器控制单元内的比较器 113 与地址寄存器 112 相连。在翻译期间，比较器 113 从存储器接收加载地址并存储到指向存储器的门控存储缓冲器。如果加载或存储的存储器地址与寄存器 112(或者根据实施情况不同而是其它寄存器)内的地址进行比较，则根据模式的不同产生一定的意外事件。代码词态化软件响应意外事件，确保存储器地址与执行单元寄存器 111 保存同一正确数据。这可以纠正上述的不一致。

代码词态化软件作出响应的方式取决于特定的意外事件。如果是数据不一致，则在一个实施例中是返回翻译并重新执行，执行单元寄存器内没有任何的“别名”数据。这种解决方案可以纠正存储器与执行单元寄存器数据的不一致和存储器与在代码序列重新排序和重新安排之前就被从执行单元寄存器 111 复制到其它寄存器的数据的不一致。解决这个问题的其它可行方法是用最近的存储器数据更新执行单元寄存器或者用最近的加载数据更新存储器。

在存储器地址被赋予别名以避免从存取地址加载数据期间，由于执行单元寄存器 111 内的数据可能在新数据写入存储器地址时过时，所以比较器试图进行存储器地址的写入操作。在这种情况下，比较器 113 检测到写入受保护的存储器地址的意图；并且如果有这种意图就产生意外事件。意外事件使得在进一步使用寄存器数据之前将存储器内的数据写入寄存器 111 以更新寄存器，或者导致不使用执行单元寄存器的代码返回并执行以完成别名优化。这可能涉及目标代码的重新翻译。

在存储器地址被赋予别名以使顺序存储操作利用寄存器 111 表示存储器地址期间，可能有一个命令禁止存储数据到存储器地址的意外事件的产生该命令将电路系统置于一种模式（写入模式），在读模式中未检查别名就从寄存器 111 将数据存入存储器地址从而使寄存器的数据重复写入受保护的存储器地址。

另一方面，在存储器地址被赋予别名允许利用寄存器 111 表示存储器地址的存储操作期间，电路系统可以处于一种模式(回写模式)，应该模式中存储器地址上数据直到代码序列执行完毕或以其他方式终止时才更新。在这种模式下，写入存储器地址的指令可能需要更新保存在执行单元寄存器内的数据使之与新数据一致。另一方面，在这种模式下，试图读取存储器地址的操作要求产生意外事件，

从而可以在读取数据之前使存储器内保存的数据更新为与执行单元寄存器内新数据一致。

图 12 示出了包含比较器 120 的实施例的别名电路，比较器 120 用来检测和控制对受保护存储器的数据加载和存储操作。比较器 120 包括多个存储单元 122(图中只示出了一个)，诸如内容定址存储器的作为受保护存储器地址的输入项。例如可以有 8 个存储单元用于输入项，每个输入项包含足够多数量的位位置(例如 32 个)来存储存储器单元的物理地址、字节屏蔽和各种属性位。在属性位中有表示受保护存储器大小的位和表示存储器地址是正常还是异常的位。应该指出的是，比较器 120 中输入项的存储单元的每一个都与图 10 所示寄存器 112 等效，从而使得比较器 120 担负起图 10 的寄存器 112 和比较器 113 的功能。

别名电路还包括别名使能寄存器 124、屏蔽别名使能寄存器的寄存器 125、别名出错寄存器 126、存储别名电路使能指示(例如单个位)的寄存器 127 以及存储模式位的寄存器 128。

在操作中，受保护的物理地址连同字节屏蔽(表示哪些位置的字节受保护的位)一起存储在输入项的其中一个存储单元内。这种物理地址可以对 64 位的数据寻址，字节屏蔽的每个位表示该地址上数据的一个字节。受保护的特定输入项通过设定硬件使能寄存器 124 的特定位表示。在翻译期间当允许返回翻译时，寄存器 125 屏蔽调拨时刻寄存器 124 内的值。在所示实施例中，使能寄存器 124 和屏蔽使能寄存器实际上被分配为存储单元 122 的属性位。

当寄存器 127 指示别名使能时，根据寄存器 128 指示的设定模式的条件，比较器保存物理存储器地址和字节屏蔽并用来检测存储器加载和存储的存储器地址。如果模式设定为写入状态，则持续用执行单元寄存器保存的受保护存储器地址的数据更新存储器，因此从该存储器地址加载其它地址的数据一直是最新的数据并且无需检验。但是将数据存入存储器地址可能使执行单元寄存器 112 内的数据无效，因此必须测试这些存储数据。如果将数据存储在被保护的地址上并且字节屏蔽表明数据存储在被保护的地址的受保护字节上，则比较器产生别名意外事件以检测写入模式下的存储操作。

另一方面，如果模式设定为回写模式状态，则只有当别名硬件释放或者发生意外事件时才更新存储器地址。因此存储器地址上的数据可能过时，因此在别名硬件处于使能状态时必须检测加载和存储二种操作。为此，如果加载或存储指向

受保护地址并且其字节屏蔽表明被访问的数据位于比较器 120 保存的存储器地址的受保护字节上，则比较器产生别名意外事件。

两种模式下发生的意外事件都在别名出错寄存器 126 内设定合适的位以指定引起意外事件的地址。根据代码词态化软件的特定意外事件处理器，所产生的特定的意外事件可以修复或者返回以解决这种问题。以修复解决问题的使得最新的数据被放入作用于执行单元数据寄存器和存储器地址的特定字节上。这包括使能寄存器 124 的状态返回到寄存器 125 所保存的状态。

以上描述了采用别名检测硬件可以进行减少加载和存储操作的优化并且可以在加载和存储操作减少的基础上进行重新排序和重新安排操作。重新排序有利于在诸如超标量或 VLIW 之类并行执行资源的机器上更好地安排操作。

这种方法也可以在不减少加载和存储操作的情形下，根据加载和存储的情况安全地重新排序操作。这改善了安排性能并且对于不重复加载或存储操作的代码很有用。

本领域内的技术人员将会发现，微处理器可以通过电路形式与典型的计算机元件连接从而构成诸如图 9 所示的计算机。显而易见，当微处理器用于现代 X86 计算机时，它可以通过处理器总线与存储器和总线控制电路连接。存储器和总线控制电路安排或能提供对主存储器的访问，也提供了对与微处理器一起使用的高速缓存的访问。存储器和总线控制线路还提供了对诸如 PCI 或其它局部总线的访问，通过这些总线对 I/O 设备进行访问。特定的计算机系统取决于由本发明微处理器所替代的典型微处理器一起使用的电路。

为了描述处理器的操作和加快执行速度的方法，这里的实例是将 X86 目标代码的少量实例翻译为主基本指令。实例涉及了将 X86 目标指令翻译为词态主机指令的过程，包括本发明微处理器所作的优化、重新排序和重新安排各种步骤。通过以下描述的过程，本领域内技术人员将会理解利用目标处理器执行源指令所需的操作与主处理器上执行翻译所需操作之间的差别。

用 C 语言源代码编写的源指令描述了一个非常简单的循环操作。当每次循环后都减一的变量“n”大于“0”时，数值“c”存储在指针“*s”表示的地址内，每次循环之后该指针都增一。

Original C code

```
while( (n--)>0) {
    *s++=c
}
```

Win32 x86 instructions produced by a compiler compiling this C code.

```
mov    %ecx,[%ebp+0xc]      // load c from memory address into the %ecx
mov    %eax,[%ebp+0x8]      // load s from memory address into the %eax
mov    [%eax],%ecx          // store c into memory address s held in %eax
add    %eax,#4              // increment s by 4.
mov    [%ebp+0x8],%eax      // store (s + 4) back into memory
mov    %eax,[%ebp+0x10]     // load n from memory address into the %eax
lea    %ecx,[%eax-1]        // decrement n and store the result in %ecx
mov    [%ebp+0x10],%ecx     // store (n-1) into memory
and    %eax,%eax            // test n to set the condition codes
jg     .-0x1b               // branch to the top of this section if 'n>0'
```

在实例的第一部分, 执行用 C 语言语句定义的操作的每条 X86 汇编语言指令用汇编语言操作助记符列示, 后面跟着涉及特定基本操作的参数。每条指令的注释对操作作了解释。虽然所示的执行顺序可以由目标处理器改变, 但是每次执行目标 C 语言指令的循环时每条汇编语言指令都必须执行。因此, 如果循环执行 100 次, 则所示的每条指令也必须执行 100 次。

Shows each X86 instruction shown above followed by the host instructions necessary to implement the X86 instruction.

```
mov    %ecx,[%ebp+0xc]      // load c from memory address into ecx

add    R0,Rebp,0xc          ; form the memory address and put it in R0
ld      Recx,[R0]           ; load c from memory address in R0 into Recx

mov    %eax,[%ebp+0x8]      // load s from memory address into %eax

add    R2,Rebp,0x8          ; form the memory address and put it in R2
ld      Reax,[R2]           ; load s from memory address in R2 into Recx

mov    [%eax],%ecx          // store c into memory address s held in %eax
st      [Reax],Recx         ; store c into memory address s held in Reax

add    %eax,#4              // increment s by 4
add    Reax,Reax,4          ; increment S by 4

mov    [%ebp+0x8],%eax      // store (s + 4) back into memory

add    R5,Rebp,0x8          ; form the memory address and put it in R5
st      [R5],Reax           ; store (s + 4) back into memory
```

```

mov  %eax, [%ebp+0x10]    // load n from memory address into %eax

add   R7, Rebp, 0x10      ; form the memory address and put it in R7
ld    Reax, {R7}          ; load n from memory address into the Reax

lea   %ecx, [%eax-1]      // decrement n and store the result in %ecx

sub   Recx, Reax, 1        ; decrement n and store the result in Recx

mov   [%ebp+0x10], %ecx    // store (n - 1) into memory

add   R9, Rebp, 0x10      ; form the memory address and put it in R9

st    [R9], Recx          ; store (n - 1) into memory

and   %eax, %eax          // test n to set the condition codes

andcc R11, Reax, Reax      ; test n to set the condition codes

jg    .-0x1b              // branch to the top of this section if "n>0"

```

most instruction key:

```

ld = load          add = ADD          st = store
sub = subtract     jg = jump if condition codes indicate greater
andcc = and set the condition codes

```

下一实例描述了执行 C 语言指令的同一目标基本指令。但是在每条基本目标指令之后列出了在微处理器特定实施例中完成同一操作所需的基本主指令，其中词态主机为针对上述方式设计的 VLIW 处理器。应该注意的是，由正式目标寄存器屏蔽的主寄存器用 X86 寄存器名的前面加“R”表示，因此例如 Reax 表示与 EAX 正式目标寄存器相关的工作寄存器。

Adds host instructions necessary to perform X86 address computation and upper and lower segment limit checks.

```

mov  %ecx, [%ebp+0xc]      // load c

add   R0, Rebp, 0xc        ; form logical address into R0
chkl  R0, Rss_limit        ; Check the logical address against segment lower
limit
chku  R0, R_FFFFFFFF       ; Check the logical address against segment upper
limit
add   R1, R0, Rss_base     ; add the segment base to form the linear
address
ld    Recx, [R1]           ; load c from memory address in R1 into Recx

mov  %eax, [%ebp+0x8]      // load s

add   R2, Rebp, 0x8        ; form logical address into R0
chkl  R2, Rss_limit        ; Check the logical address against segment lower
limit
chku  R2, R_FFFFFFFF       ; Check the logical address against segment upper
limit
add   R3, R2, Rss_base     ; add the segment base to form the linear
address
ld    Reax, [R3]           ; load s from memory address in R3 into Ra

mov  [%eax], %ecx          // store c into [s]

chku  Reax, Rds_limit      ; Check the logical address against segment upper
limit
add   R4, Reax, Rds_base   ; add the segment base to form the linear
address
st    [R4], Recx          ; store c into memory address s

add  %eax, #4              // increment s by 4

addcc Reax, Reax, 4         ; increment s by 4

mov  [%ebp+0x8], %eax      // store (s + 4) to memory

add   R5, Rebp, 0x8        ; form logical address into R5
chkl  R5, Rss_limit        ; Check the logical address against segment lower
limit
chku  R5, R_FFFFFFFF       ; Check the logical address against segment upper
limit
add   R6, R5, Rss_base     ; add the segment base to form the linear
address
st    [R6], Reax          ; store (s + 4) to memory address in R6

mov  %eax, [%ebp+0x10]     // load n

add   R7, Rebp, 0x10       ; form logical address into R7
chkl  R7, Rss_limit        ; Check the logical address against segment lower
limit
chku  R7, R_FFFFFFFF       ; Check the logical address against segment upper
limit
add   R8, R7, Rss_base     ; add the segment base to form the linear
address
ld    Reax, [R8]           ; load n from memory address in R8 into Reax

lea  %ecx, [%eax-1]        // decrement n

sub   Recx, Reax, 1         ; decrement n
mov  [%ebp+0x10], %ecx      // store (n - 1)

```

```

add    R9, Rbp, 0x10          ; form logical address into R9
chkl   R9, Rss_limit          ; Check the logical address against segment lower
limit
chku   R9, R_FFFFFFFF        ; Check the logical address against segment upper
limit
add    R10, R9, Rss_base      ; add the segment base to form the linear
address
st     [R10], Rcx             ; store n-1 in Rcx into memory using address
in R10

and    %eax, %eax             // test n to set the condition codes

andcc  R11, Rax, Rax          ; test n to set the condition codes

```

下一实例表示每条基本目标指令的主基本指令加法，代码词态化软件可以利用其产生目标操作所需的地址。应该注意的是，主地址产生指令只是在采用代码词态化软件产生地址而非地址产生硬件来产生地址的微处理器实施例中才需要。在诸如 X86 微处理器之类的目标处理器中，地址是利用地址产生硬件产生的。在这种实施例中，无论何时产生地址，都完成了计算；并且还加入主基本指令以检查地址值从而确定计算的地址是否在合适的 X86 段界内。

Adds instructions to maintain the target X86 instruction pointer "eip" and the commit instructions that use the special morph host hardware to update X86 state.

```

mov  %ecx, [%ebp+0xc]           // load c

add  R0, Rebp, 0xc
chk1 R0, Rss_limit
chku R0, R_FFFFFFFF
add  R1, R0, Rss_base
ld   Recx, [R1]
add  Reip, Reip, 3              ; add X86 instruction length to eip in
Reip
commit                          ; commits working state to official state

mov  %eax, [%ebp+0x8]           // load s

add  R2, Rebp, 0x8
chk1 R2, Rss_limit
chku R2, R_FFFFFFFF
add  R3, R2, Rss_base
ld   Reax, [R3]
add  Reip, Reip, 3              ; add X86 instruction length to eip in
Reip
commit                          ; commits working state to official state

mov  [%eax], %ecx               // store c into {s}

chku Reax, Rds_limit
add  R4, Reax, Rds_base
st   [R4], Recx
add  Reip, Reip, 2              ; add X86 instruction length to eip in
Reip
commit                          ; commits working state to official state

add  %eax, #4                   // increment s by 4

addcc Reax, Reax, 4
add  Reip, Reip, 5              ; add X86 instruction length to eip in
Reip
commit                          ; commits working state to official state

mov  [%ebp+0x8], %eax           // store (s + 4)

add  R5, Rebp, 0x8
chk1 R5, Rss_limit
chku R5, R_FFFFFFFF
add  R6, R5, Rss_base
st   [R6], Reax
add  Reip, Reip, 3              ; add X86 instruction length to eip in
Reip
commit                          ; commits working state to official state

mov  %eax, [%ebp+0x10]          // load n

add  R7, Rebp, 0x10
chk1 R7, Rss_limit
chku R7, R_FFFFFFFF
add  R8, R7, Rss_base
ld   Reax, [R8]
add  Reip, Reip, 3              ; add X86 instruction length to eip in
Reip
commit                          ; commits working state to official state

```

```

    lea    %ecx, [%eax-1]                // decrement n

    sub    Recx, Reax, 1
    add    Reip, Reip, 3                ; add X86 instruction length to eip in
    Reip                                       ; commits working state to official state
    commit

    mov    [%ebp+0x10], %ecx            // store (n - 1)

    add    R9, Rebp, 0x10
    chkl   R9, Rss_limit
    chku   R9, R_FFFFFFFF
    add    R10, R9, Rss_base
    st     [R10], Recx
    add    Reip, Reip, 3                add X86 instruction length to eip in
    Reip                                       ; commits working state to official state
    commit

    and    %eax, %eax                  // test n

    andcc  R11, Reax, Reax
    add    Reip, Reip, 3                ; commits working state to official state
    commit

    jg     .-0x1b                      // branch "n>0"

    add    Rseq, Reip, Length(jg)
    ldc    Rtarg, EIP(target)
    selcc  Reip, Rseq, Rtarg
    commit                               ; commits working state to official state
    jg     mainloop, mainloop

Host Instruction key:
    commit = copy the contents of the working registers to the official
            target registers and send working stores to memory

```

本实例示出了每组基本主指令的两步相加，在执行了完成每条基本指令所需的主指令后更新正式目标寄存器并将门控存储缓冲器内未调拨的值调拨入存储器。显而易见，在每种情况下，目标指令的长度与工作指令指针寄存器(Reip)内

的值相加。随后执行调拨指令。在实施例中，调拨指令对屏蔽在相关正式目标寄存器的每一工作寄存器的当前值进行复制并将指定门控存储缓冲器位置的指针值从紧靠未调拨存储数据之前移动至紧靠这些存储数据之后从而将它们放入存储器。

显而易见的是上面最后示出的指令清单都是构成源目标汇编语言指令的主指令翻译所需的指令。如果翻译操作在该处停止，则基本主指令的数量将远远大于目标指令数量(大约是 6 倍)，并且执行时间将超过目标处理器的执行时间。但是此时尚未对指令进行过重新排序、优化和重新安排。

如果将运行指令但只是运行一次，则完成指令进一步重新排序和其它优化所需的时间可能超过此时执行翻译的时间。如果这样，一种实施例的微处理器将在此处停止翻译，存储翻译结果，随后执行翻译指令以确定是否发生意外事件或错误。在本实施例中，重新排序其它优化步骤仅仅发生在经确定特定的翻译操作将进行多次或者需要优化时。例如通过在每条翻译指令内放入主指令，对翻译执行进行计数并在计数值达到某一数值时产生意外事件(或转移)的方法来实现。意外事件(或转移)将操作转移至代码词态化软件，由其进行下述部分或所有优化以及任何适于该翻译操作的其它优化。第二种确定翻译指令执行次数和是否需要优化的方法是以一定的频度或者根据一些统计依据中断翻译指令的执行，并优化该时刻运行的任何翻译指令。这最终使得最经常运行的指令得到了优化。另一种方案是对每一条特定类型的主指令进行优化，例如产生循环的主指令或者运行次数可能最多的主指令。

Optimization

=====

Assumes 32 bit flat address space which allows the elimination of segment base additions and some limit checks.

Win32 uses Flat 32b segmentation

Record Assumptions:

Rss_base==0

Rss_limit==0

Rds_base==0

Rds_limit==FFFFFFF

SS and DS protection check

```

mov  %ecx,[%ebp+0xc]           // load c

add  R0,Rebp,0xc
chku R0,R_FFFFFFFF
ld   Recx,[R0]
add  Reip,Reip,3
commit

mov  %eax,[%ebp+0x8]           // load s

add  R2,Rebp,0x8
chku R2,R_FFFFFFFF
ld   Reax,[R2]
add  Reip,Reip,3
commit

mov  [%eax],%ecx               // store c into [s]

chku Reax,R_FFFFFFFF
st   [Reax],Recx
add  Reip,Reip,2
commit

add  %eax,#4                   // increment s by 4

addcc Reax,Reax,4
add  Reip,Reip,5
commit

mov  [%ebp+0x8],%eax           // store (s + 4)

add  R5,Rebp,0x8
chku R5,R_FFFFFFFF
st   [R5],Reax
add  Reip,Reip,3
commit

mov  %eax,[%ebp+0x10]          // load n

add  R7,Rebp,0x10
chku R7,R_FFFFFFFF
ld   Reax,[R7]
add  Reip,Reip,3
commit

```

```

    lea    %ecx,[%eax-1]                // decrement n

    sub    Rcx,Rcx,1
    add    Reip,Reip,3
    commit

    mov    [%ebp+0x10],%ecx             // store (n - 1)

    add    R9,Rebp,0x10
    chku   R9,R_FFFFFFFF
    st     {R9},Rcx
    add    Reip,Reip,3
    commit

    and    %eax,%eax                    // test n

    andcc  R11,Rcx,Rcx
    add    Reip,Reip,3
    commit

    jg     .-0x1b                        // branch "n>0"

    add    Rseq,Reip,Length(jg)
    ldc    Rtarg,EIP(target)
    selcc  Reip,Rseq,Rtarg
    commit
    jg     mainloop,mainloop

```

本实例示出了可以利用改进型微处理器实现的优化步骤的第一阶段。在优化阶段，就象代码词态化软件的许多其它操作一样，假定是有优化结果的。特定的优化操作假定作为为 X86 系列处理器普通存储器模型编写的目标应用程序作为 32 位程序将继续保持原样。将会注意到是，这种假设只针对 X86 系列而对其它被仿真的处理器系列并非必要。

如果假设成立，则在 X86 应用程序中所有的段都被映象至相同的地址空间。这使得可以减少 X86 分段处理所需的基本主指令。显而易见，段值开始时被设定为零。随后，数据基点也被设定为零，并且上限被设定为最大可用存储空间。随后在执行目标基本指令的每组基本主指令中，都省略了分段所需的对段基点值的检查和段基点地址的计算。这减少了用于需要寻址功能的每目标基本指令的两

应该注意的是，这种优化需要对应用程序是否采用 32 位普通存储器模型作出预测。如果不是这样的情况，则由于主循环要实现的目的地址的控制转移并检查出源地址假设与目的地址假设不匹配，所以将会发现错误。随后将进行新的翻译操作。这种技术非常普通并且被应用于各种分段操作和其它不经常变化的“模式化”情况，诸如调试、系统管理模式或“实”模式之类的“模式”。

Assume data addressed includes no bytes outside of computer memory limits which can only occur on unaligned page crossing memory references at the upper memory limit, and can be handled by special case software or hardware.

```

mov  %ecx, [%ebp+0xc]           // load c

add  R0, Rbp, 0xc
ld   Recx, {R0}
add  Reip, Reip, 3
commit

mov  %eax, [%ebp+0x8]           // load s

add  R2, Rbp, 0x8
ld   Reax, {R2}
add  Reip, Reip, 3

commit

mov  [%eax], %ecx               // store c into [s]

st   {Reax}, Recx
add  Reip, Reip, 2
commit

add  %eax, #4                   // increment s by 4

addcc Reax, Reax, 4
add  Reip, Reip, 5
commit

mov  [%ebp+0x8], %eax           // store (s + 4)

add  R5, Rbp, 0x8
st   {R5}, Reax
add  Reip, Reip, 3
commit

mov  %eax, [%ebp+0x10]          // load n

add  R7, Rbp, 0x10
ld   Reax, {R7}
add  Reip, Reip, 3
commit

```

```

lea    %ecx, [%eax-1]           // decrement n

sub     Recx, Reax, 1
add     Reip, Reip, 3
commit

mov     [%ebp+0x10], %ecx        // store (n - 1)

add     R9, Rebp, 0x10
st      [R9], Recx
add     Reip, Reip, 3
commit

and     %eax, %eax              // test n

andcc   R11, Reax, Reax
add     Reip, Reip, 3
commit

jg      .-0x1b                  // branch "n>0"

add     Rseq, Reip, Length(jg)
ldc     Rtarg, EIP(target)
selcc   Reip, Rseq, Rtarg
commit
jg      mainloop, mainloop

```

Host Instruction key:

selcc = Select one of the source registers and copy its contents to the destination register based on the condition codes.

上述实例示出了优化步骤的下一阶段，其中的预测翻译操作避免了存储上限边界的检查，这只需要对于穿越位于存储器地址空间顶部的存储器参考位置的未定位页面才需要。由硬件或软件定位工具检查这种假设的失效。这减少了对需要寻址的每条目标基本指令用另一主基本指令翻译的量。这种优化需要以前所作的假定，应用程序使用 32 位普通存储器模型并且预测到指令得到了定位。如果上述假设和预测都得不到满足，则当执行翻译指令时将会失效；并且需要重新翻译。

Detect and eliminate redundant address calculations. The example shows the code after eliminating the redundant operations.

```

mov     %ecx, [%ebp+0xc]         // load c

add     R0, Rebp, 0xc
ld      Recx, [R0]
add     Reip, Reip, 3

```

```

commit

mov  %eax, [%ebp+0x8]           // load s

add  R2, Rbp, 0x8
ld   Reax, [R2]
add  Reip, Reip, 3
commit

mov  [%eax], %ecx              // store c into [s]

st   [Reax], Recx
add  Reip, Reip, 2
commit

add  %eax, #4                  // increment s by 4

addcc Reax, Reax, 4
add  Reip, Reip, 5
commit

mov  [%ebp+0x8], %eax          // store (s + 4)

st   [R2], Reax
add  Reip, Reip, 3
commit

mov  %eax, [%ebp+0x10]         // load n

add  R7, Rbp, 0x10
ld   Reax, [R7]
add  Reip, Reip, 3
commit

lea  %ecx, [%eax-1]           // decrement n

sub  Recx, Reax, 1
add  Reip, Reip, 3
commit

mov  [%ebp+0x10], %ecx         // store (n - 1)

st   [R7], Recx
add  Reip, Reip, 3
commit

and  %eax, %eax               // test n

andcc R11, Reax, Reax
add  Reip, Reip, 3
commit

jg   .-0x1b                   // branch "n>0"

add  Rseq, Reip, Length(jg)
ldc  Rtarg, EIP(target)
selcc Reip, Rseq, Rtarg
commit
jg   mainloop, mainloop

```

本实例示出了下一步优化，其中省略了共用的主指令表达方式。具体而言，在翻译第二条目标基本指令时，工作寄存器 Rebp(表示 X86 处理器堆栈基点寄存器的工作寄存器)内的值与偏移值 0x8 相加并放入主工作寄存器 R2。将会注意到的是，除了将加法结果放入工作寄存器 R5 以外，与在前面的实例中，翻译目标基本指令 5 时进行的是同一操作。因此当主基本指令 5 执行期间工作寄存器 R5 内放入的值已经存在于工作寄存器 R2 中。这样在翻译目标基本指令 5 时可以省略主加法指令；将工作寄存器 R2 的值复制到工作寄存器 R5 内。同样，由于在翻译目标基本指令 6 时已经完成了将工作寄存器 Rebp 的值与偏移值 0x10 相加的步骤并且结果驻留在寄存器 R7 内，所以在翻译目标基本指令 8 时可以省略该步骤。应该注意的是这种优化不依赖于预测，因此不会失效或者需要重新翻译。

Assume that target exceptions will not occur within the translation so delay updating eip and target state.

```

mov  %ecx,[%ebp+0xc]           // load c

add  R0,Rebp,0xc
ld   Recx,[R0]

mov  %eax,[%ebp+0x8]           // load s

add  R2,Rebp,0x8
ld   Reax,[R2]

mov  [%eax],%ecx               // store c into {s}

st   {Reax},Recx

add  %eax,#4                   // increment s by 4

add  Reax,Reax,4

mov  [%ebp+0x8],%eax           // store {s + 4}

st   {R2},Reax

```

```

mov  %eax,[%ebp+0x10]           // load n

add   R7,Rebp,0x10
ld    Reax,[R7]

lea   %ecx,[%eax-1]             // decrement n

sub   Recx,Reax,1

mov   [%ebp+0x10],%ecx          // store (n - 1)

st    [R7],Recx

and   %eax,%eax                 // test n

andcc R11,Reax,Reax

jg    .-0x1b                    // branch "n>0"

add   Rseq,Reip,Length(block)
ldc   Rtarg,EIP(target)
selcc Reip,Rseq,Rtarg
commit
jg    mainloop,mainloop

```

上述实例示出了一种优化步骤，它预测构成整个翻译操作的基本目标指令的翻译能够在不发生意外事件的情况下完成。如果符合预测情况，则无需在执行一条目标基本指令的每一主基本指令序列结束时更新正式目标寄存器或者调拨存储缓冲器内的未调拨的存储结果。如果预测为真，则只需在目标基本指令序列结束时更新正式目标寄存器并且只需调拨一次存储内容。这可以省略两条执行每一基本目标指令的基本主指令，它们被一条更新正式目标寄存器并将未调拨存储内容调拨入存储器的主基本指令代替。

显而易见的是，其它的预测操作也极有可能是正确的预测。如果预测保持为真值，那么该步骤与现有仿真技术比就有极大的优势。它使所有执行目标基本指令序列的基本主指令归类为一个序列从而可以对所有主基本指令逐条优化。这有利于在得益于超长指令字技术的词态主机上并行运行大量的操作。由于有更多的优化选择，所以可以采用其它大量的优化。但是如果预测情况失真并且在执行循环时发生意外事件，则由于直到实际执行主指令序列后才发生调拨操作，所以正式目标寄存器和存储器保存的是目标基本指令序列开始时的正式目标状态。从意外事件恢复所需的操作是转储未调拨的存储内容，使正式寄存器返回工作寄存器并在序列开始时重新翻译目标基本指令。这种重新翻译一次翻译一条目标指令，

并且在代表每条目标基本指令的主序列翻译之后才更新正式状态。随后执行翻译。当该重新翻译过程中出现意外事件时，立即使用正式目标寄存器和存储器的正确目标状态来执行意外事件。

In summary:

```

add    R0,Rebp,0xc
ld     Recx,[R0]
add    R2,Rebp,0x8
ld     Reax,[R2]
st     [Reax],Recx
add    Reax,Reax,4
st     [R2],Reax
add    R7,Rebp,0x10
ld     Reax,[R7]           // Live out
sub    Recx,Reax,1         // Live out
st     [R7],Recx
andcc  R11,Reax,Reax

add    Rseq,Reip,Length(block)
ldc    Rtarg,EIP(target)
selcc  Reip,Rseq,Rtarg
commit
jg     mainloop,mainloop

```

The comment "Live Out" refers to the need to actually maintain Reax and Recx correctly prior to the commit. Otherwise further optimization might be possible.

=====

上面的总结示出了优化过程中处于该位置时保留的主基本指令序列。虽然本实例示出了目标指令指针(EIP)串列的维护，但是也可以使翻译时转移的指针 EIP 保持不一致，这将省去实例中本步骤和后续步骤中更新序列的指针 EIP。

Renaming to reduce register resource dependencies. This will allow subsequent scheduling to be more effective. From this point on, the original target X86 code is omitted as the relationship between individual target X86 instructions and host instructions becomes increasingly blurred.

```

add    R0,Rebp,0xc
ld     R1,[R0]
add    R2,Rebp,0x8
ld     R3,[R2]
st     [R3],R1
add    R4,R3,4
st     [R2],R4
add    R7,Rebp,0x10
ld     Reax,[R7]           // Live out
sub    Recx,Reax,1         // Live out
st     [R7],Recx
andcc  R11,Reax,Reax
add    Rseq,Reip,Length(block)
ldc    Rtarg,EIP(target)
selcc  Reip,Rseq,Rtarg
commit
jg     mainloop,mainloop

```

本实例示出了通常称为寄存器更名的下一个优化步骤，其中需要使用在主基本指令序列中用于一个以上的操作的工作寄存器的操作改为采用不同的未使用工作寄存器以防止两条主指令要求使用同一硬件。因此，例如在上述两实例中的第二条主基本指令采用代表正式目标寄存器 ECX 的工作寄存器 Recx。第十条主基本指令也使用工作寄存器 Recx。通过改变第二主基本指令中的操作从而使 R0 内地址所指向的值存储在工作寄存器 R1 中而不是寄存器 Recx 中，两条主指令就不会使用同一寄存器。同样，第四、第五和第六条主基本指令在前面实例中都使用工作寄存器 Reax；通过使第四主基本指令使用前面未使用的工作寄存器 R3 以替代工作寄存器 Reax 而第六主基本指令使用前面未使用的工作寄存器 R4 以替代寄存器 Reax，避免了它们使用同一硬件。

After the scheduling process which organizes the primitive host operations as multiple operations that can execute in the parallel on the host VLIW hardware. Each line shows the parallel operations that the VLIW machine executes, and the "&" indicates the parallelism.

```

add    R2,Rebp,0x8      & add R0,Rebp,0xc
nop                                & add R7,Rebp,0x10
ld     R3,[R2]          & add Rseq,Reip,Length(block)
ld     R1,[R0]          & add R4,R3,4
st     [R3],R1          & ldc Rtarg,EIP(target)
ld     Reax,[R7]        & nop
st     [R2],R4          & sub Recx,Reax,1
st     [R7],Recx        & andcc R11,Reax,Reax
selcc  Reip,Rseq,Rtarg  & jg  mainloop.mainloop & commit

```

Host Instruction key:
nop = no operation

上述实例示出了在词态主机上执行的主基本指令的安排。在该实例中，假定词态主机为 VLIW 处理器，除了用于协调代的词态化软件的硬件增强部分以外，词态主机还包括其它处理单元中的两个算术和逻辑(ALU)单元。第一行表示两个单独的相加指令，将它们安排在词态主机上一起运行。显而易见，在前述实例中它们是第三和第八条基本主指令。第二行包括 NOP 指令(无操作但是转向下一指令)和另一加法操作。NOP 指令表示即使在经过某些安排优化之后也不一定是两

条指令可以放在一起运行。在任何情况下，该实例示出了此时只剩下 9 组基本主指令来执行原先 10 条目标指令。

Resolve host branch targets and chain stored translations

```

add    R2, Rbp, 0x8      & add R0, Rbp, 0xc
nop                                & add R7, Rbp, 0x10
ld     R3, [R2]          & add Rseq, Reip, Length(block)
ld     R1, [R0]          & add R4, R3, 4
st     [R3], R1          & ldc Rtarg, EIP(target)
ld     Reax, [R7]        & nop
st     [R2], R4          & sub Rcx, Reax, 1
st     [R7], Rcx         & andcc R11, Reax, Reax
selcc  Reip, Rseq, Rtarg  & jg Sequential, Target & commit

```

本实例基本上表示同一组主基本指令，除了指令现在被存储于翻译缓冲器内并且执行一次或一次以上，由于最后的跳转(jg)指令现在指向将另一翻译指令序列链接起来的跳转地址。链接过程使指令序列跳出了翻译器主循环从而完成了序列的翻译。

Advanced Optimizations, Backward Code Motion:

This and subsequent examples start with the code prior to scheduling. This optimization first depends on detecting that the code is a loop. Then invariant operations can be moved out of the loop body and executed once before entering the loop body.

entry:

```

add    R0, Rbp, 0xc
add    R2, Rbp, 0x8
add    R7, Rbp, 0x10
add    Rseq, Reip, Length(block)
ldc    Rtarg, EIP(target)

```

Loop:

```

ld     R1, [R0]
ld     R3, [R2]
st     [R3], R1
add    R4, R3, 4
st     [R2], R4
ld     Reax, [R7]
sub    Rcx, Reax, 1
st     [R7], Rcx
andcc  R11, Reax, Reax
selcc  Reip, Rseq, Rtarg
commit
jg     mainloop, Loop

```

上述实例示出了通常只与大量重复的序列一起使用的高级优化步骤。该过程首先检测构成循环的翻译，并检查单条基本主指令以确定循环体内哪条指令产生

不变的结果。这些指令从循环内去除并且只执行一次，将值放入寄存器内；从此，存储在寄存器内的值被重复使用而不是重新运行指令。

Schedule the loop body after backward code motion. For example purposes, only code in the loop body is shown scheduled

Entry:

```
add    R0, Rbp, 0xc
add    R2, Rbp, 0x8
add    R7, Rbp, 0x10
add    Rseq, Reip, Length(block)
ldc    Rtarg, EIP(target)
```

Loop:

```
ld      R3, [R2]           & nop
ld      R1, [R0]           & add R4, R3, 4
st      [R3], R1           & nop
ld      Reax, [R7]         & nop
st      [R2], R4           & sub Rcx, Reax, 1
st      [R7], Rcx          & andcc R11, Reax, Reax
```

```
selcc Reip, Rseq, Rtarg    & jg Sequential, Loop    & commit
```

Host Instruction key:

ldc = load a 32-bit constant

当这些不重复的指令从循环中去除并且对序列进行安排进行执行，经安排的指令与上述实例中的指令相同。由此可见，在循环第一次迭代期间完成初始化指令但是只执行一次，此后在循环期间只执行所示 7 个时钟间隔内剩余的主基本指令。因此执行基本目标指令所需的，执行时间从 10 条指令间隔缩短为 7 个指令间隔。

显而易见，从循环中去除的步骤是地址产生步骤。这样在改进型微处理器内只需在循环开始时产生一次地址；即，地址产生只需执行一次。另一方面，X86 目标处理器的地址产生硬件必须在每次执行循环时产生地址。如果循环执行 100 次，则改进型微处理器只产生一次地址而目标处理器对每一地址要产生 100 次。

After Backward Code Motion:

Target:

```
add    R0, Rbp, 0xc
add    R2, Rbp, 0x8
add    R7, Rbp, 0x10
add    Rseq, Reip, Length(block)
ldc    Rtarg, EIP(target)
```

Loop:

```
ld      R1, [R0]
ld      R3, [R2]
st      [R3], R1
add     R4, R3, 4
st      [R2], R4
ld      Reax, [R7]          //Live out
```

```

sub    Recx,Reax,1      //Live out
st     [R7],Recx
andcc  R11,Reax,Reax
selcc  Reip,Rseq,Rtarg
commit
jg     mainloop,Loop
=====

```

Register Allocation:

This shows the use of register alias detection hardware of the morph host that allows variables to be safely moved from memory into registers. The starting point is the code after "backward code motion". This shows the optimization that can eliminate loads.

First the loads are performed. The address is protected by the alias hardware, such that should a store to the address occur, an "alias" exception is raised. The loads in the loop body are then replaced with copies. After the main body of the loop, the alias hardware is freed.

Entry:

```

add    R0,Rebp,0xc
add    R2,Rebp,0x8
add    R7,Rebp,0x10
add    Rseq,Reip,Length(block)
ldc    Rtarg,EIP(target)

ld     Rc,[R0]          ;First do the load of the variable from memory
prot   [R0],Alias1      ;Then protect the memory location from stores

ld     Rs,[R2]
prot   [R2],Alias2

ld     Rn,[R7]
prot   [R7],Alias3

```

Loop:

```

copy   R1,Rc
copy   R3,Rs
st     [R3],R1
add    R4,Rs,4

copy   Rs,R4
st     [R2],Rs,NoAliasCheck
copy   Reax,Rn          //Live out
sub    Recx,Reax,1      //Live out
copy   Rn,Recx
st     [R7],Rn,noAliasCheck
andcc  R11,Reax,Reax
selcc  Reip,Rseq,Rtarg
commit
jg     Epilog,Loop

```

Epilog:

```

FA     Alias1          Free the alias detection hardware
FA     Alias2          Free the alias detection hardware
FA     Alias3          Free the alias detection hardware
j      Sequential

```

```
Host Instruction key:
protect = protect address from loads      FA = free alias
copy = copy                               j = jump
```

本实例示出由本发明微处理器实现的更为高级的优化步骤。参见本实例前的第二个实例就会注意到前 3 条相加指令包括堆栈内的地址计算。这些地址在主操作序列执行期间不会变化。因此这些地址上存储的值可以从存储器内查找出来并装载入寄存器以供立即可得到而执行。显而易见，这是在主基本指令 6、8 和 10 中完成的。在指令 7、9 和 11 中，每个存储器地址被特殊的主别名硬件加上保护标记并且寄存器被表示为这些存储器地址的别名，所以如果试图改变数据就会引发意外事件。此时，每条涉及从堆栈存储器地址移动数据的加载操作都变为简单的寄存器至寄存器之间的复制操作，这比从存储器地址加载数据要快许多。应该注意的是，一旦循环执行到 $n=0$ ，则必须去除每个存储器地址上的保护从而可以使用别名寄存器。

Copy Propagation :

After using the alias hardware to turn loads within the loop body into copies, copy propagation allows the elimination of some copies.

Entry:

```
add    R0, Rbp, 0xc
add    R2, Rbp, 0x8
add    R7, Rbp, 0x10
add    Rseq, Reip, Length(block)
ldc    Rtarg, EIP(target)

ld     Rc, [R0]
prot   [R0], Alias1

ld     Rs, [R2]
prot   [R2], Alias2

ld     Recx, [R7]
prot   [R7], Alias3
```

Loop:

```
st     [Rs], Rc
add    Rs, Rs, 4
st     [R2], Rs, NoAliasCheck
copy   Reax, Recx           //Live out
sub    Recx, Reax, 1        //Live out
st     [R7], Recx, NoAliasCheck
andcc  R11, Reax, Reax
selcc  Reip, Rseq, Rtarg
```

```

commit
jg     Epilog, Loop

Epilog:
    FA    Alias1
    FA    Alias2
    FA    Alias3
    j      Sequential

```

本实例示出了下一优化步骤，其中大多数代替上一实例中优化的加载指令的复制指令并非必要并且可以省略，即，如果进行寄存器至寄存器之间的复制，则在复制数据的寄存器的操作前数据已经存在。如果是这样，可以在第一寄存器内访问数据而不是在复制数据的寄存器内，因此可以省略复制操作。显而易见，这省略了上一实例循环中所示的第一、第二、第五和第九基本主指令。此外，其它主基本指令中所用的寄存器也可以改为反映正确的寄存器数据内容。这样，例如当省略第一和第二复制指令时，第三存储指令必须从实际存在数据的工作寄存器 Rc(而不是寄存器 R1)复制数据并放入实际存在地址的工作寄存器 Rs(而非寄存器 R3)表示的地址上。

Example illustrating scheduling of the loop body only.

```

Entry:
    add    R0, Rbp, 0xc
    add    R2, Rbp, 0x8
    add    R7, Rbp, 0x10
    add    Rseq, Reip, Length(block)
    ldc    Rtarg, EIP(target)

    ld     Rc, {R0}
    prot   {R0}, Alias1

    ld     Rs, {R2}
    prot   {R2}, Alias2

    ld     Rcx, {R7}
    prot   {R7}, Alias3

Loop:
    st     {Rs}, Rc,          & add Rs, Rs, 4      & copy Reax, Rcx
    st     {R2}, Rs, NAC      & sub Rcx, Rcx, 1
    st     {R7}, Rcx, NAC     & andcc R11, Reax, Rcx
    selcc  Reip, Rseq, Rtarg  & jg Epilog, Loop & commit

Epilog:
    FA    Alias1
    FA    Alias2
    FA    Alias3
    j      Sequential

```

Host Instruction key:
NAC= No Alias Check

上述实例示出了安排后的主指令。将会看到，在执行该循环时，序列所需的时钟数要少于从源代码经原始译码而得到的基本目标指令执行所需的时钟数。因此除了所有其它加速技术以外，所运行联合的操作的总数也少于执行源目标代码所需的操作数量。

Store Elimination by use of the alias hardware.

Entry:

```

add    R0,Rebp,0xc
add    R2,Rebp,0x8
add    R7,Rebp,0x10
add    Rseq,Reip,Length(block)
ldc    Rtarg,EIP(target)
ld      Rc,{R0}
prot   [R0],Alias1      ;protect the address from loads and
stores

```

```

ld      Rs,{R2}
prot   [R2],Alias2      ;protect the address from loads and
stores

```

```

ld      Recx,{R7}
prot   [R7],Alias3      ;protect the address from loads and
stores

```

Loop:

```

st      [Rs],Rc,          & add Rs,Rs,4      & copy Reax,Recx
sub     Recx,Reax,1        & andcc R11,Reax,Reax
selcc   Reip,Rseq,Rtarg    & jg  Epilog,Loop & commit

```

Epilog:

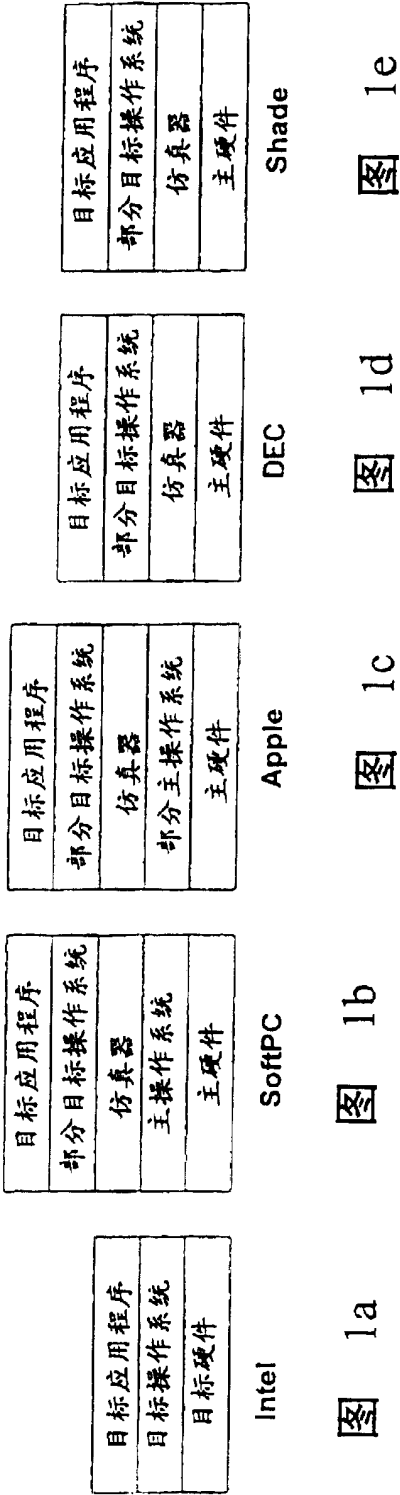
```

FA      Alias1
FA      Alias2
FA      Alias3
st      [R2],Rs           ;writeback the final value of Rs
st      [R7],Recx         ;writeback the final value of Recx
j       Sequential

```

本实例所示的最后优化用于别名硬件以节省存储。这节省了循环体内的存储并且只在循环结束部分施行。与最初 10 条目标指令相比，这将循环体内主指令的数量减少到 3 条。

虽然以上借助实施例描述了本发明，但是本领域内技术人员可以在不偏离本发明范围和精神的前提下对本发明作出各种修改和变动。例如虽然本发明描述的是 X86 处理器的仿真，但是它也可以应用于针对其它处理器结构设计的应用程序上，也可以用于运行在虚拟机上执行的程序，例如 P code、Postscript 或 Java 程序等。因此本发明由下面所附权利要求限定。



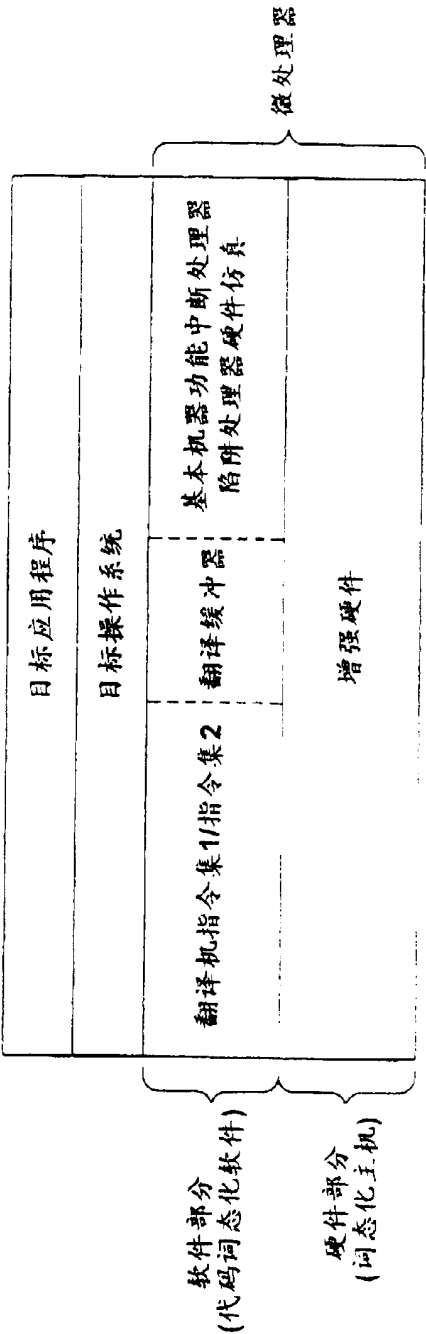


图 2

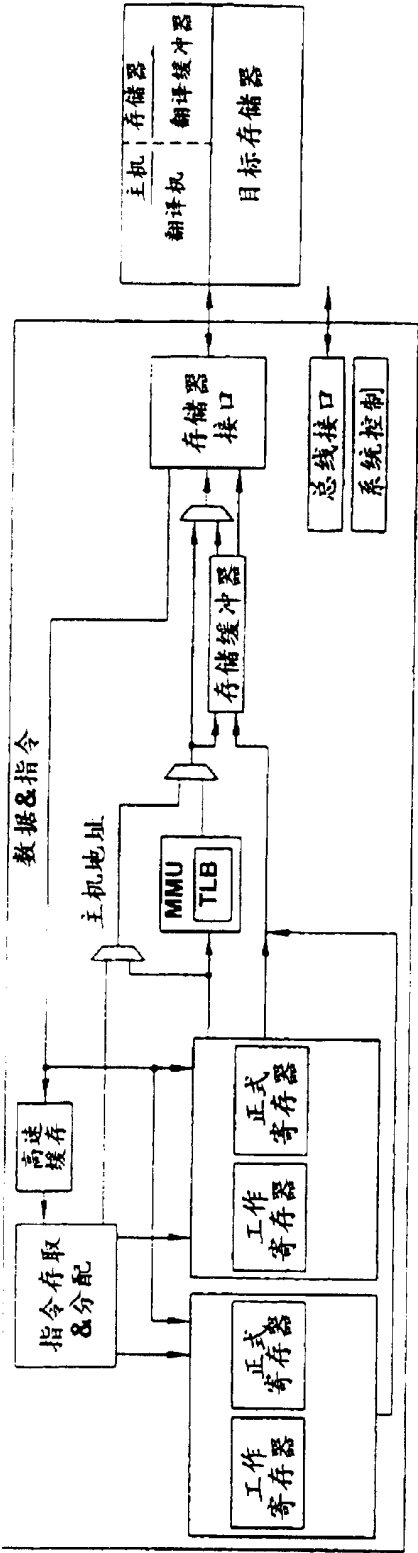


图 3

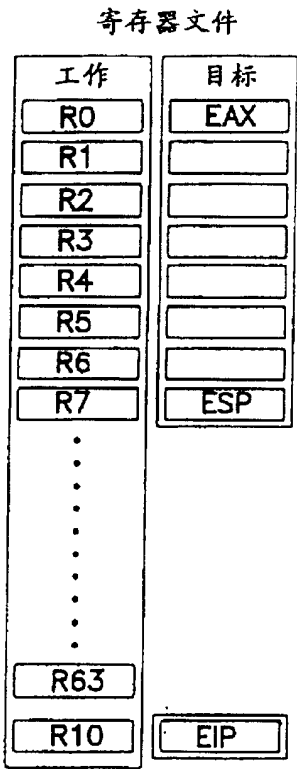


图 4

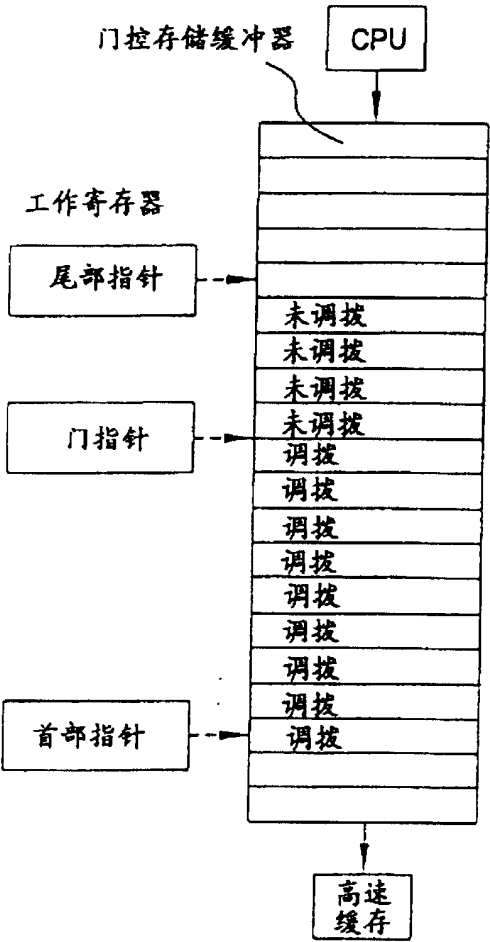


图 5

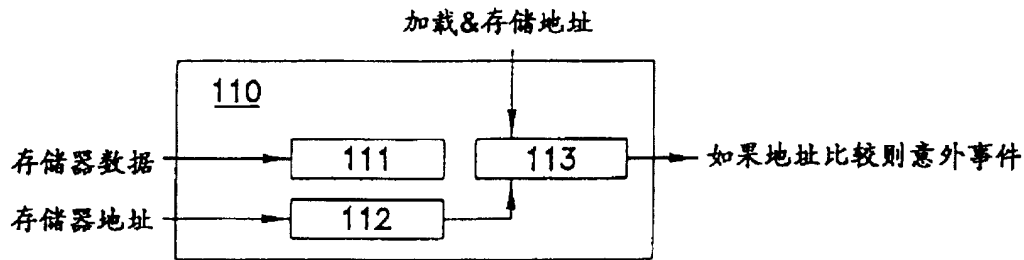


图 10

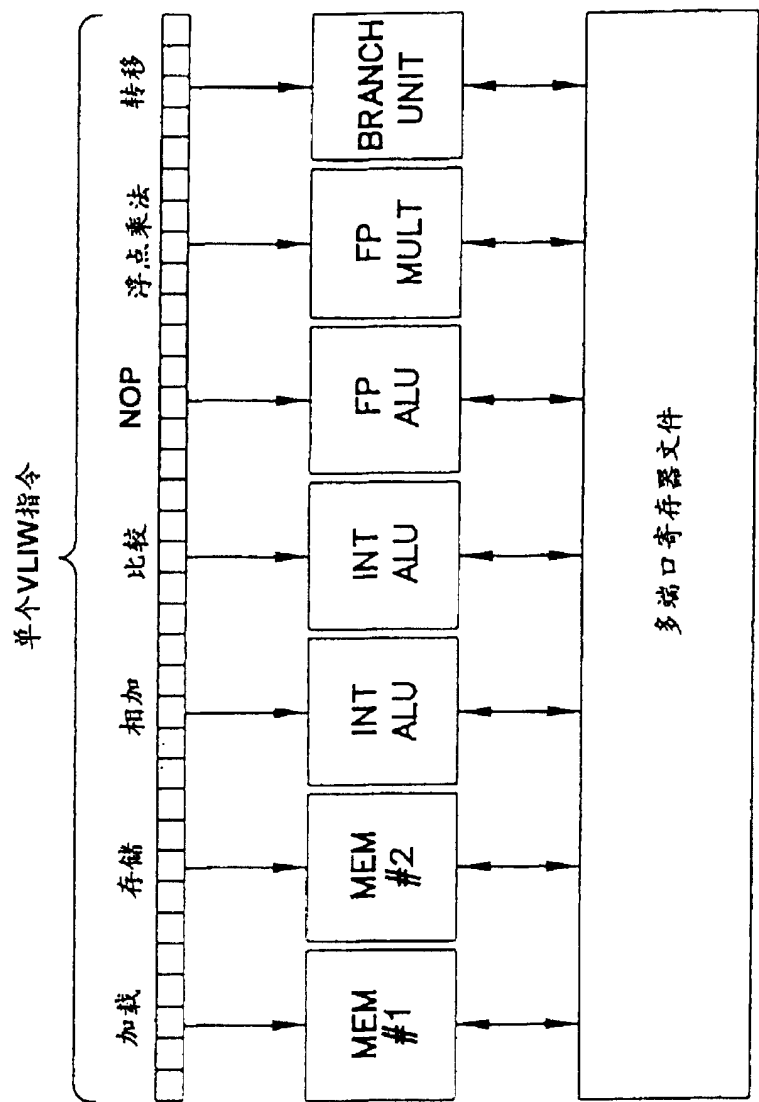


图 6c

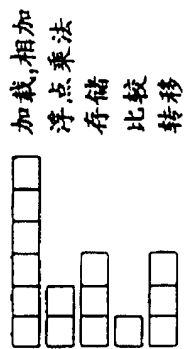


图 6a

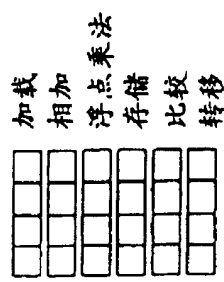


图 6b

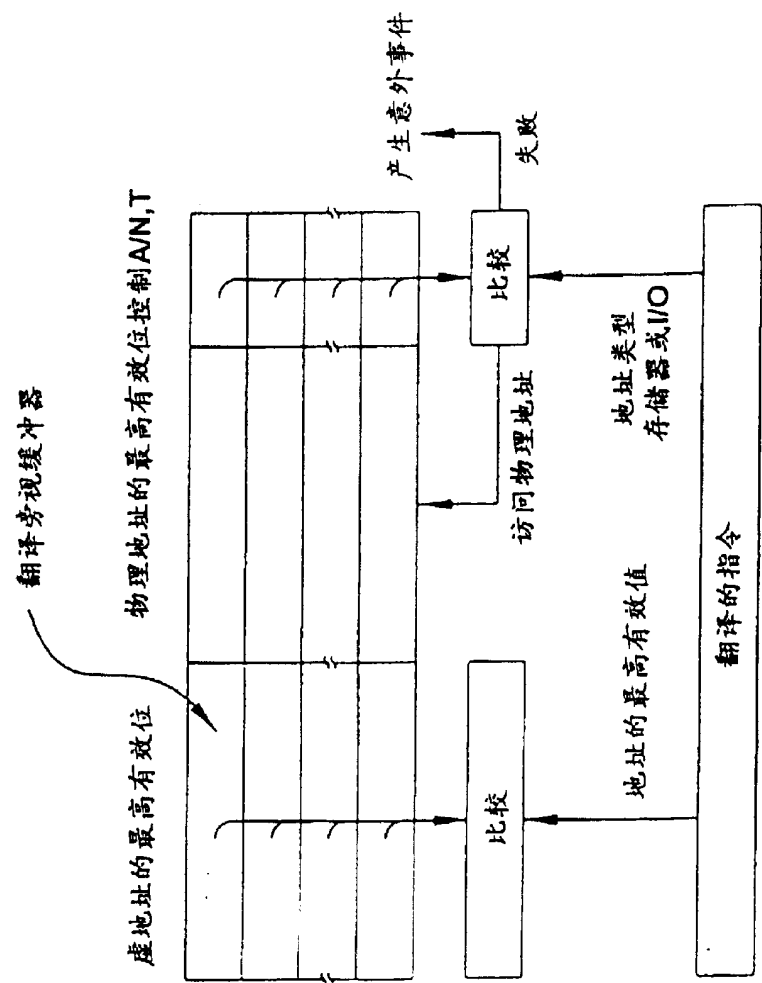


图 11

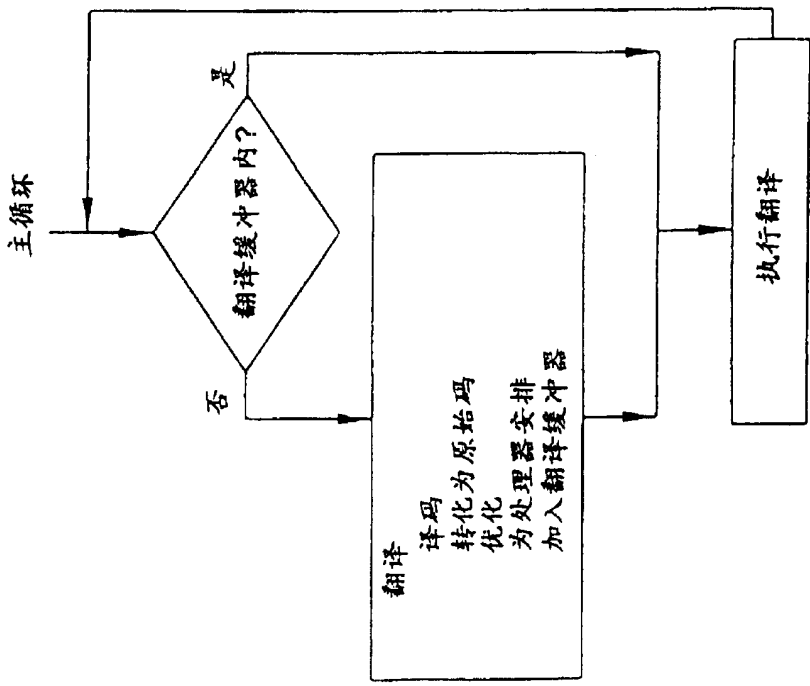


图 7

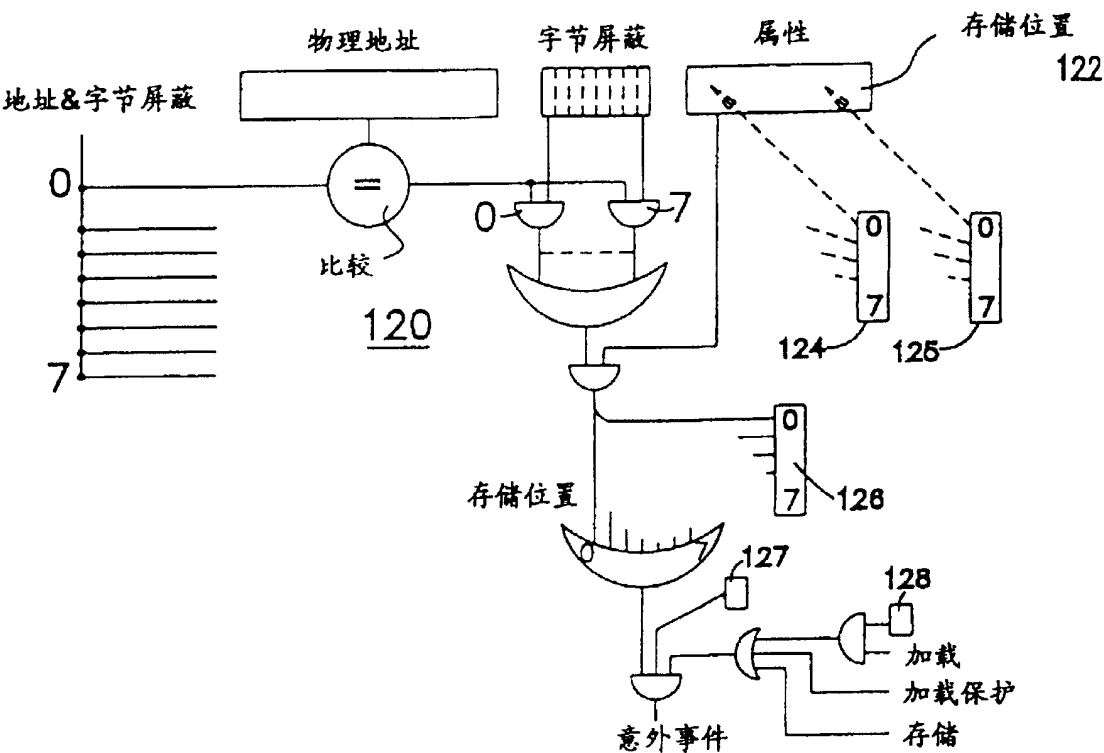


图 12

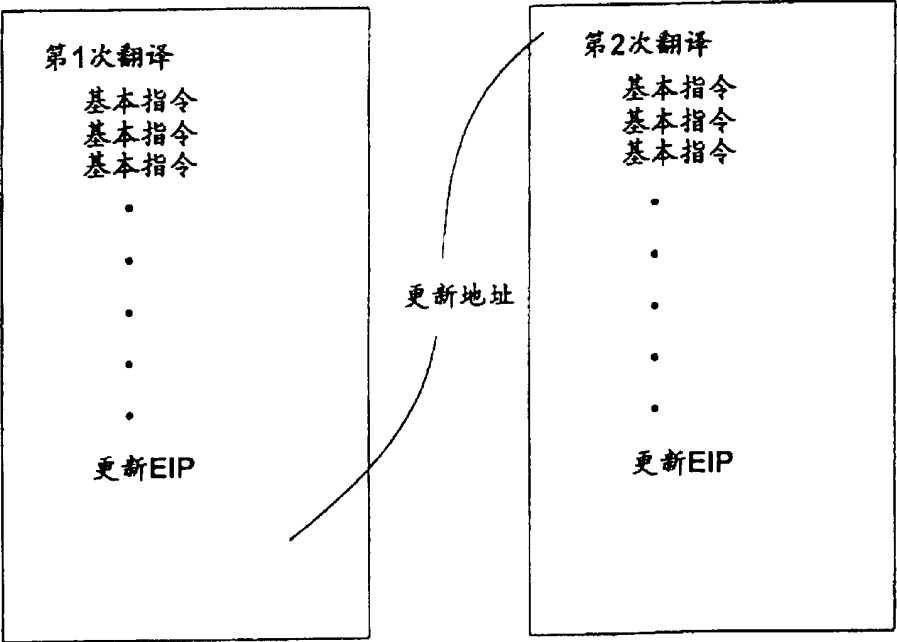


图 8

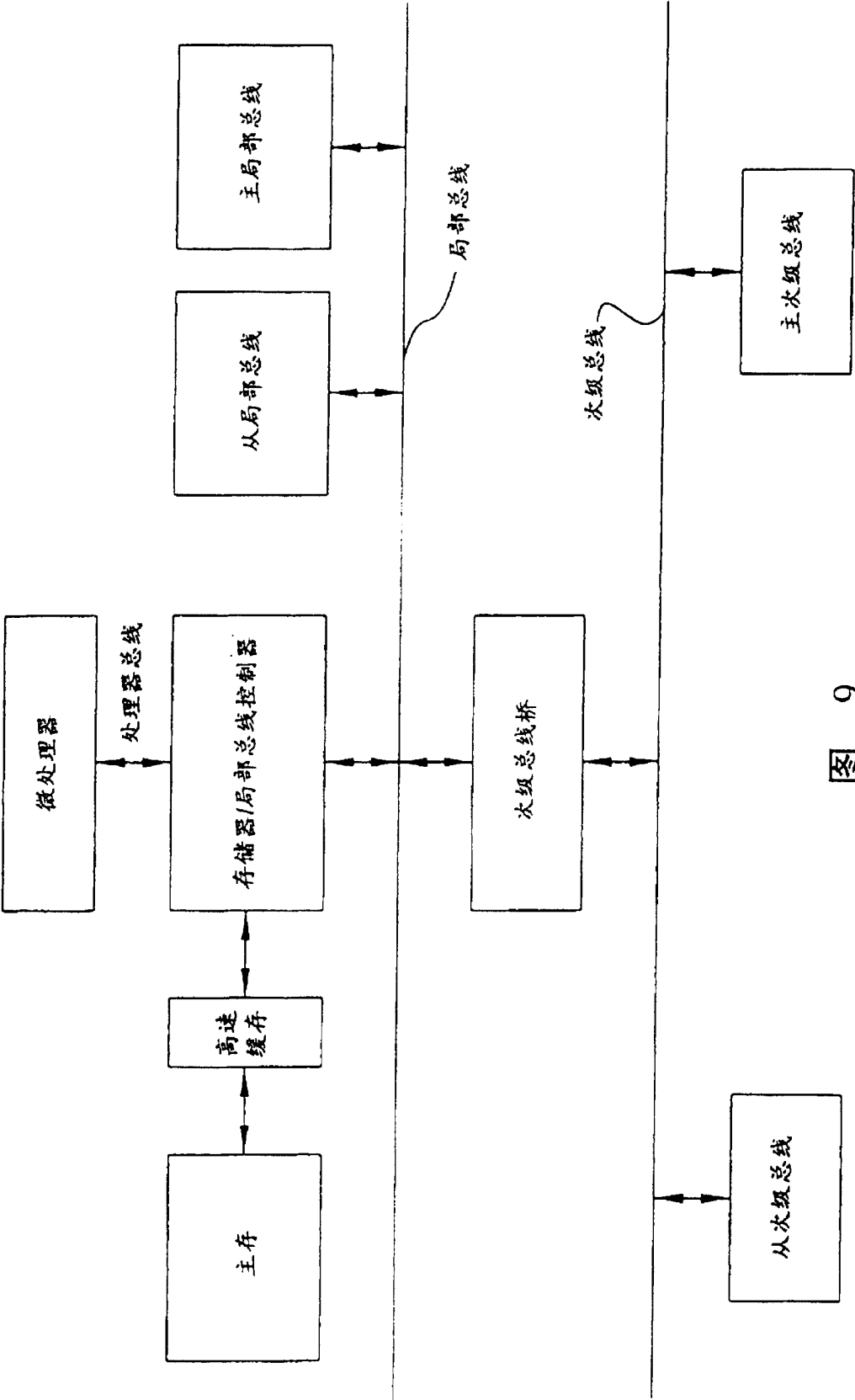


图 9

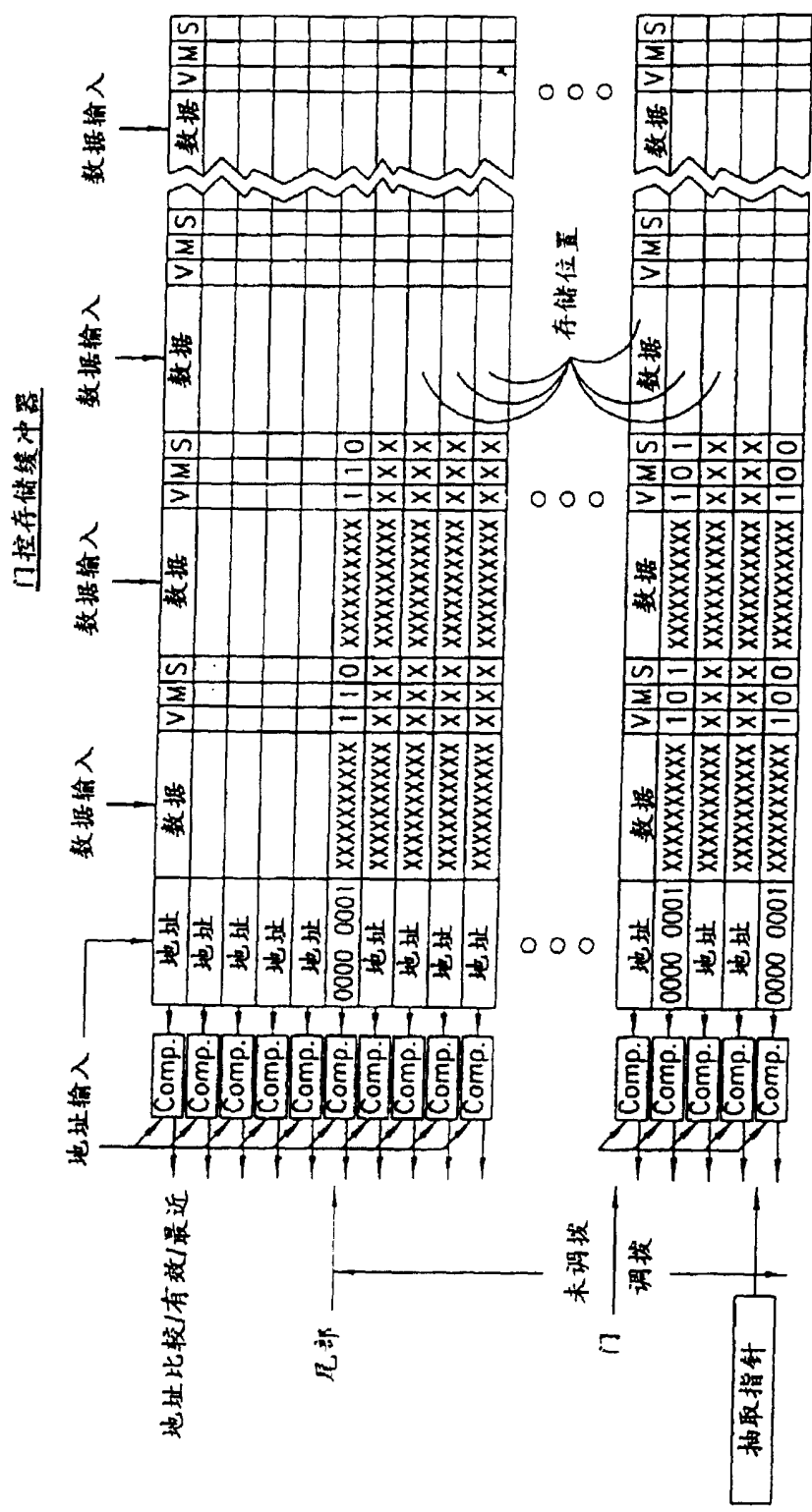


图 13

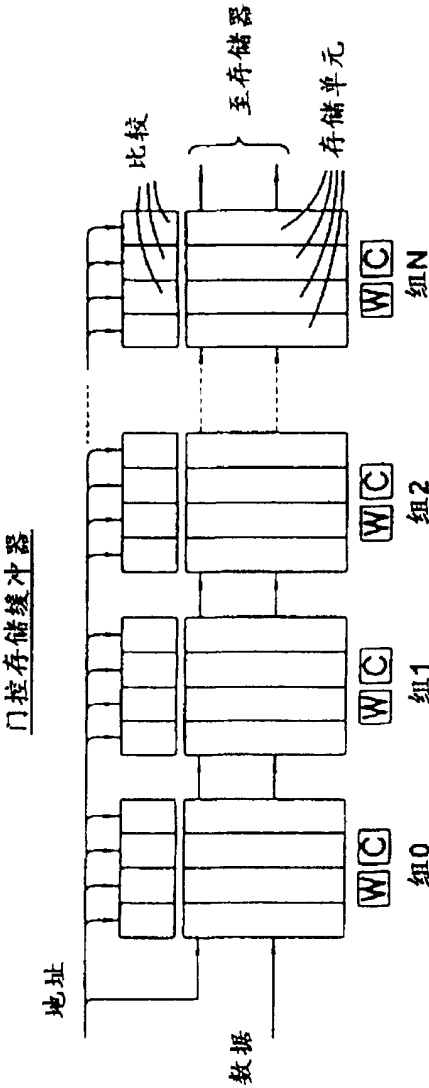


图 14