[19] 中华人民共和国国家知识产权局

[51] Int. Cl[7]
G06F 9/455
G06F 9/318

[12] 发明专利申请公开说明书

[21] 申请号 02811101.X

[43] 公开日 2004 年 8 月 18 日

[11] 公开号 CN 1522404A

[22] 申请日 2002.2.26 [21] 申请号 02811101.X
[30] 优先权
[32] 2001. 5.31 [33] GB [31] 0113197.8
[86] 国际申请 PCT/GB2002/000858 2002.2.26
[87] 国际公布 WO2002/097609 英 2002.12.5
[85] 进入国家阶段日期 2003.12.1
[71] 申请人 ARM 有限公司
地址 英国剑桥郡
[72] 发明人 E·C·内维尔 A·C·罗斯

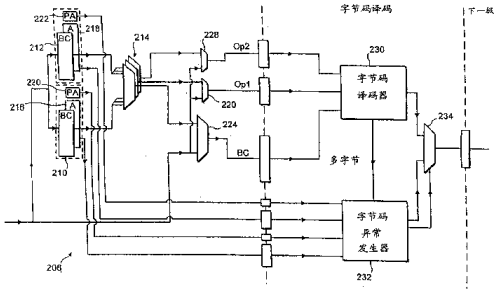
[74] 专利代理机构 中国专利代理(香港)有限公司
代理人 程天正 王 勇

权利要求书 8 页 说明书 26 页 附图 21 页

[54] 发明名称 在多指令集系统中对未处理操作的处理

[57] 摘要

诸如 Java 字节码之类的第一指令集的程序指令的未处理操作被检测。 不调用直接处理那个未处理操作的机制，相反，使用来自诸如 ARM 指令之类的第二指令集的一条或多条指令来仿真遭受所述未处理操作的那条指令。 如果第二指令集的这些指令也遭受到未处理操作，则在所述第二指令集中的、用于处理所述未处理操作的机制可以被调用来修补那个操作。 这一方法很好地适用于处理如下的可变长度指令的未处理操作，所述可变长度指令由具有本机固定指令集的处理器核心来加以解释。特别是，采用这种方式，可以分方便地处理预取中止和未处理浮点运算。



ISSN1008-4274

1.用于在第一指令集和一个或多他指令集的程序指令的控制下处理数据的装置，所述装置包括:

(i)未处理操作检测器，可操作地来检测所述第一指令集程序指令的未处理操作；和

(ii)未处理操作处理器，可操作地来在检测所述未处理操作时，触发使用所述一个或多个其他指令集中的至少一个指令集的一条或多条指令来仿真所述第一指令集的所述指令。

2.如权利要求1所述的装置，其中，所述未处理操作检测器是提取中止检测器，可操作地来检测指令字的未处理提取。

3.如权利要求1所述的装置，其中，所述未处理操作检测器是未处理浮点运算检测器，可操作地来检测未处理浮点运算。

4.如权利要求1、2和3中任一项所述的装置，其中，所述第一指令集是可变长度指令集，所述一个或多个其他指令集中的至少一个指令集是固定长度指令集。

5.如权利要求1、2和3中任一项所述的装置，其中，所述第一指令集是可变长度指令集，所述一个或多个其他指令集全部都是固定长度指令集。

6.如权利要求4和5中任一项所述的装置，包括:

指令提取器,可操作地来提取包含要被执行的程序指令的固定长度指令字，所述可变长度指令集的至少一些程序指令跨越了多于一个指令字。

7.如权利要求6所述的装置，其中，所述未处理操作检测器是提取中止检测器，可操作地来在检测如下指令字的未处理提取时，中止所述可变长度指令并触发使用一条或多条固定长度指令来仿真所述可变长度指令，其中所述指令字是包含可变长度指令的一部分的第二或后续指令字。

8.如权利要求6所述的装置，其中，所述未处理操作检测器是提取中止检测器，可操作地来在检测到如下指令字的未处理提取时，中止所述可变长度指令并且触发使用一条或多条固定长度指令来仿真所

述可变长度指令，在所述指令字中，所述可变长度指令被确定为跨过存储器页面边界。

9. 如权利要求 6 所述的装置，其中，所述未处理操作检测器是提取中止检测器，可操作地来在检测到如下指令字的未处理提取时，中止所述可变长度指令并且触发使用一条或多条固定长度指令来仿真所述可变长度指令，其中在所述指令字中，所述可变长度指令被确定为在存储器页面边界尾部的字节的固定数目内开始，其中，所述字节的固定数目小于由用于在第一指令集程序指令的控制下处理数据的所述装置所处理的最长的可变长度指令。

10. 如权利要求 3 所述的装置，其中，多个浮点运算可以在执行所述第一指令集的单条指令期间加以执行，所述多个浮点运算中的任一浮点运算都潜在地会引发未处理浮点运算，并且所述未处理操作检测器是未处理浮点运算检测器，可操作地来检测由所述多个浮点运算中的任一浮点运算所生成的未处理浮点运算。

11. 如权利要求 3 和 10 中任一项所述的装置，其中，当执行所述第一指令集中的、引发所述未处理浮点运算的指令时，所述未处理操作检测器并不立即检测未处理浮点运算，相反，在执行所述第一指令集的后续指令时才检测所述未处理浮点运算，所述未处理操作检测器可操作用来中止所述后续指令并触发使用一条或多条固定长度指令来仿真所述后续指令。

12. 如权利要求 11 所述的装置，其中，所述未处理操作检测器仅仅在所述后续指令又引发浮点运算时，才检测所述未处理浮点运算。

13. 如权利要求 11 和 12 中任一项所述的装置，还包括事件指示器，用于使所述未处理操作处理器能够确定未处理浮点运算已经发生。

14. 如权利要求 11-13 中任一项所述的装置，还包括操作指示器，用于允许确定引发未处理浮点运算的浮点运算。

15. 如权利要求 11-14 中任一项所述的装置，其中，在仿真所述第一指令集的所述指令之前，所述未处理操作处理器采用现有机制来处理所述未处理操作。

16. 如权利要求 15 所述的装置，其中，所述未处理操作处理器执

行固定长度浮点指令，所述固定长度浮点指令的执行具有采用现有机制来处理所述未处理操作的效果。

17. 如在先权利要求中任一项所述的装置，其中，所述第二指令集是执行所述程序指令的处理器核心的本机指令集。

18. 如在先权利要求中任一项所述的装置，其中，所述第一指令集是解释型指令集。

19. 如在先权利要求中任一项所述的装置，其中，所述第一指令集包括 Java 字节码指令。

20. 如在先权利要求中任一项所述的装置，还包括重启动逻辑，用于在所述未处理操作之后重新开始执行；其中

所述装置可被操作用来生成对应于所述多个指令集中一个指令集的指令的多组翻译器输出信号中的一组的序列，以便表示所述多个指令集的至少一条指令，每个序列都是这样的，不对输入变量做任何改变，直到所述序列中的最后操作被执行为止；和

在执行表示所述多个指令集的所述至少一条指令的操作序列期间，未处理操作发生之后：

(i) 如果所述未处理操作是在开始执行所述序列的最后操作之前发生的，则所述重启动逻辑重新启动所述序列中第一操作的执行；和

(ii) 如果所述未处理操作是在开始执行所述序列的最后操作之后发生的，则所述重启动逻辑重新启动在所述序列之后的下一个指令的执行。

21. 一种在第一指令集和一个或多他指令集的程序指令的控制下处理数据的方法，所述方法包括如下操作：

(i) 检测所述第一指令集的程序指令的未处理操作；和

(ii) 当检测到所述未处理操作时，触发使用所述一个或多个其他指令集的至少一个指令集的一条或多条指令来仿真所述第一指令集的所述指令。

22. 如权利要求 21 所述的方法，其中，所述检测步骤检测指令字的未处理提取。

23. 如权利要求 21 所述的方法，其中，所述检测步骤检测未处理

浮点运算。

24. 如权利要求 21、22 和 23 中任一项所述的方法，其中，所述第一指令集是可变长度指令集，所述一个或多个其他指令集中的至少一个指令集是固定长度指令集。

25. 如权利要求 21、22 和 23 中任一项所述的方法，其中，所述第一指令集是可变长度指令集，所述一个或多个其他指令集全部都是固定长度指令集。

26. 如权利要求 24 和 25 中任一项所述的方法，包括:

(i)提取包含要被执行的程序指令的固定长度指令字，所述可变长度指令集的至少一些程序指令跨越了多于一个指令字。

27. 如权利要求 26 所述的方法，其中，当检测到如下指令字的未处理提取时，中止所述可变长度指令并触发使用一条或多条固定长度指令来仿真所述可变长度指令，其中所述指令字是包含可变长度指令的一部分的第二或者后续指令字。

28. 如权利要求 26 所述的方法，其中，当检测到如下指令字的未处理提取时，中止所述可变长度指令并触发使用一条或多条固定长度指令来仿真所述可变长度指令，在所述指令字中，所述可变长度指令被确定为跨过存储页面边界。

29. 如权利要求 26 所述的方法，其中，当检测到如下指令字的未处理提取时，中止所述可变长度指令并且触发使用一条或多条固定长度指令来仿真所述可变长度指令，其中在所述指令字中，所述可变长度指令被确定为在存储器页面边界尾部的字节的固定数目内开始，其中，所述字节的固定数目小于所处理的最长的可变长度指令。

30. 如权利要求 23 所述的方法，其中，多个浮点运算可以在执行所述第一指令集的单条指令期间执行，所述多个浮点运算中的任一浮点运算都潜在地会引发未处理浮点运算，并且所述检测可操作地来检测由所述多个浮点运算中的任一浮点运算所生成的未处理浮点运算。

31. 如权利要求 23 和 30 中任一项所述的方法，其中，当执行所述第一指令集中的、引发所述未处理浮点运算的指令时，所述未处理操作检测器并不立即检测未处理浮点运算，相反，在执行所述第一指令集的后续指令时才检测所述未处理浮点运算，以便中止所述后续指令

并触发使用一条或多条固定长度指令来仿真所述后续指令。

32. 如权利要求 31 所述的方法，其中，所述检测仅仅在所述后续指令又引发浮点运算时，才检测所述未处理浮点运算。

33. 如权利要求 31 和 32 中任一项所述的方法，还包括事件指示器，用于使所述未处理操作处理器能够确定未处理浮点运算已经发生。

34. 如权利要求 31-33 中任一项所述的方法，还包括设置指示器以便允许确定引发未处理浮点运算的浮点运算。

35. 如权利要求 31-34 中任一项所述的方法，其中，在仿真所述第一指令集的所述指令之前，处理所述未处理操作采用现有机制来处理所述未处理操作。

36. 如权利要求 35 所述的方法，其中，处理所述未处理操作执行固定长度浮点指令，所述固定长度浮点指令的执行具有采用现有机制来处理所述未处理操作的效果。

37. 如权利要求 21-36 中任一项所述的方法，其中，所述第二指令集是执行所述程序指令的处理器核心的本机指令集。

38. 如权利要求 21-37 中任一项所述的方法，其中，所述第一指令集是解释型指令集。

39. 如权利要求 21-38 中任一项所述的方法，其中，所述第一指令集包括 Java 字节码指令。

40. 如权利要求 21-39 中任一项所述的方法，还包括:

生成对应于所述多个指令集中一个指令集的指令的多组翻译器输出信号的一组的序列，以便表示所述多个指令集的至少一条指令，每个序列都是这样的，不对输入变量做任何改变，直到所述序列中的最后操作被执行为止；和

在执行表示所述多个指令集的所述至少一条指令期间，未处理操作发生之后:

(i) 如果所述未处理操作是在开始执行所述序列的最后操作之前发生的，则所述重启动逻辑重新启动所述序列中第一操作的执行；和

(ii) 如果所述未处理操作是在开始执行所述序列的最后操作之后

发生的，则所述重启动逻辑重新启动在所述序列之后的下一个指令的执行。

41. 一种用于控制数据处理装置在第一指令集和一个或多他指令集的程序指令的控制下处理数据的计算机程序产品，所述计算机程序产品包括：

未处理操作处理器逻辑，可操作地来在检测到所述未处理操作时，触发使用所述一个或多个其他指令集的至少一个指令集的一条或多条指令来仿真一指令集中引发所述微处理操作的所述指令。

42. 如权利要求 41 所述的计算机程序产品，其中，所述未处理操作是指令字的未处理提取。

43. 如权利要求 41 所述的计算机程序产品，其中，所述未处理操作是一个或多个未处理浮点运算。

44. 如权利要求 41、42 和 43 中任一项所述的计算机程序产品，其中，所述第一指令集是可变长度指令集，所述一个或多个其他指令集中的至少一个指令集是固定长度指令集。

45. 如权利要求 41、42 和 43 中任一项所述的计算机程序产品，其中，所述第一指令集是可变长度指令集，所述一个或多个其他指令集全部都是固定长度指令集。

46. 如权利要求 44 和 45 中任一项所述的计算机程序产品，其中，包含要被执行的程序指令的固定长度指令字被提取，所述可变长度指令集的至少一些程序指令跨越了多于一个指令字。

47. 如权利要求 46 所述的计算机程序产品，其中，当检测到如下指令字的未处理提取时，中止所述可变长度指令并触发使用一条或多条固定长度指令来仿真所述可变长度指令，其中所述指令字是包含可变长度指令的一部分的第二或后续指令字。

48. 如权利要求 46 所述的计算机程序产品，其中，当在检测到如下指令字的未处理提取时，中止所述可变长度指令并且触发使用一条或多条固定长度指令来仿真所述可变长度指令，在所述指令字中，所述可变长度指令被确定为跨越存储器页面边界。

49. 如权利要求 46 所述的计算机程序产品，其中，当检测到如下

指令字的未处理提取时，中止所述可变长度指令并且触发使用一条或多条固定长度指令来仿真所述可变长度指令，其中在所述指令字中，所述可变长度指令被确定为在存储器页面边界尾部的字节的固定数目内开始，其中，所述字节的固定数目小于所处理的最长的可变长度指令。

50. 如权利要求 43 所述的计算机程序产品，其中，多个浮点运算可以在执行所述第一指令集的单条指令期间执行，所述多个浮点运算中的任一浮点运算都潜在地会引发未处理浮点运算，并且未处理浮点运算检测可操作地来检测由所述多个浮点运算中的任一浮点运算所生成的未处理浮点运算。

51. 如权利要求 43 和 50 中任一项所述的计算机程序产品，其中，当执行所述第一指令集中的、引发所述未处理浮点运算的指令时，未处理操作检测并不立即检测未处理浮点运算，相反，在执行所述第一指令集的后续指令时才检测所述未处理浮点运算，以便中止所述后续指令并触发使用一条或多条固定长度指令来仿真所述后续指令。

52. 如权利要求 51 所述的计算机程序产品，其中，检测未处理操作仅仅在所述后续指令还引发浮点运算时，才检测所述未处理浮点运算。

53. 如权利要求 51 和 52 中任一项所述的计算机程序产品，还包括事件指示器逻辑，用于使所述未处理操作处理器逻辑能够确定未处理浮点运算已经发生。

54. 如权利要求 51-53 中任一项所述的计算机程序产品 51，还包括操作指示器逻辑，用于允许确定引发未处理浮点运算的浮点运算。

55. 如权利要求 51-54 中任一项所述的计算机程序产品，其中，所述未处理操作处理器逻辑在仿真所述第一指令集的所述指令之前，采用现有机制来处理所述未处理操作。

56. 如权利要求 55 所述的计算机程序产品，其中所述未处理操作处理器逻辑执执行固定长度浮点指令，所述固定长度浮点指令的执行具有采用现有机制来处理所述未处理操作的效果。

57. 如权利要求 41-56 中任一项所述的计算机程序产品，其中，所述第二指令集是执行所述程序指令的处理器核心的本机指令集。

58.如权利要求41-57中任一项所述的计算机程序产品，其中，所述第一指令集是解释型指令集。

59.如权利要求41-58中任一项所述的计算机程序产品，其中，所述第一指令集包括Java字节码指令。

60.如权利要求41-59中任一项所述的计算机程序产品还包括重启动逻辑，用于在所述未处理操作之后重新开始执行；其中

所述数据处理装置可被操作用来生成对应于所述多个指令集中一个指令集的指令的多组翻译器输出信号的一组的序列，以便表示所述多个指令集的至少一条指令，每个序列都是这样的，不对输入变量做任何改变，直到所述序列中的最后操作被执行为止；和

在执行表示所述多个指令集的所述至少一条指令期间，发生未处理操作之后：

(i)如果所述未处理操作是在开始执行所述序列的最后操作之前发生的，则所述重启动逻辑重新启动所述序列中第一操作的执行；和

(ii)如果所述未处理操作是在开始执行所述序列的最后操作之后发生的，则所述重启动逻辑重新启动在所述序列之后的下一个指令的执行。

在多指令集系统中对未处理操作的处理

本发明涉及数据处理系统领域。更具体而言，本发明涉及在支持多指令集的系统中对未处理操作的处理。

提供支持多指令集的数据处理系统是众所周知的。这种系统的一个例子是由英格兰剑桥 ARM 有限公司所生产的支持 Thumb 指令的处理器。这些 启用 Thumb 指令的处理器支持 32 位 ARM 指令和 16 位 Thumb 指令的执行。

在数据处理系统中，程序指令不能够直接被所述数据处理系统处理的情况会时常发生。相应地，提供处理这种未处理操作的机制也是众所周知的。这种情形的例子是预取指令中止。众所周知的是，当在虚拟存储器系统中预取指令时，指令的加载可能会跨过页面边界，并且中止也会由于新的页面还没有被正确地映射到所述虚拟存储器系统中而发生。然后，就可以完成正确的映射并且重新发出指令预取。

这种情形的另一个例子是执行浮点指令。众所周知的是，在浮点运算执行期间，所述数据处理系统不能直接处理的情形有可能会发生。这对于与 IEEE 754 规范相兼容的浮点系统而言尤其如此。这种情形的例子是除以零的运算、涉及 NaN 的任何操作、涉及无穷大的任一操作或者涉及反向规格化数的某些操作。

问题出现的原因在于：当添加一个新指令集时，人们需要花费相当大的精力和相当可观的开发来确保用于所有可能发生的中止的合适中止机制是适当的。

当未处理的浮点运算发生时，一个特别的问题会发生。许多系统依靠检查指令流来确定还没有被处理过的浮点运算。使用新指令集，这些系统不得不被重写以便迎合所述新指令集。此外，当所述新指令集能够为新指令集的单条指令生成多个浮点运算时，问题也会出现。在这一情况下，让所述系统通过检查指令流来确定哪个浮点运算还没有被处理过也许是不可能的，因为单条指令可以引起不止一个未处理的浮点运算。

另一个问题出现在所述未处理浮点运算是不精确之时，也就是说，在执行生成浮点运算的指令之时，所述未处理浮点运算没有被检测，而相反在以后的某个时候被检测。这种情形发生的原因是由于许多浮点系统的并行性质。所述数据处理系统在碰到指定浮点运算的指令时，给浮点子系统发出浮点运算。一旦所述浮点运算已经被发送给所述浮点子系统，则所述主数据处理系统继续执行在指令流中的其他指令。在所述浮点子系统检测未处理浮点运算并且将未处理操作状态发信号通知给所述主数据处理系统之前，许多指令都可以被执行。在这一情况下，未处理浮点运算的原因都无法通过检查指令流来加以确定。众所周知的是，在像上述的情况下，包含了用于标识未处理浮点运算的寄存器的浮点系统，例如是由英格兰剑桥 ARM 有限公司所生产的向量浮点系统。

在许多系统中，让浮点子系统在任意点将未处理操作发信号通知主数据处理系统是不可能的，未处理操作仅仅在主数据处理系统与浮点子系统执行握手的、已经定义好的点才可以被发信号通知给主数据处理系统。典型地，这些握手仅仅发生在执行指定浮点运算的指令之时。在这一情况下，未处理浮点运算不能够被发信号通知给主数据处理系统，直到主数据处理系统执行指定浮点运算的另一条指令时为止。

引入那种可以结合不精确的未处理浮点运算执行每指令多个浮点运算的新指令集，导致很困难的问题或者使系统不可能来处理未处理浮点运算。所述系统无法确定哪条指令引发过未处理浮点运算，并且它还无法确定一旦所述未处理浮点运算已经被处理了，指令流中的执行应该从哪里继续。

从一方面来看，本发明提供了如下装置，所述装置用于在来自第一指令集和一个或者多个其他指令集的程序指令的控制之下来处理数据，所述装置包括：

未处理操作检测器，可操作地来检测所述第一指令集的程序指令的未处理操作；和

未处理操作处理器，可操作地来在检测所述未处理操作时，触发使用所述一个或多个其他指令集中的至少一个指令集的一条或者多条指令来仿真所述第一指令集的所述指令。

本发明认为上述问题可以通过如下安排而得到相当大的减少，即，安排所述系统来识别第一指令集中的未处理操作，而不必设法来修补所述情形并且重新执行所涉及到的第一指令集指令。取而代之，引起过所述未处理操作的第一指令集（新指令集）的指令被采用所述一个或多个其他指令集（例如，第二指令集）的一条或者多条指令加以仿真。根据发生了的未处理操作的类型，可能的情况是在仿真时，所述未处理操作将不会重新发生。可替换地，如果在使用所述一个或多个其他指令集的指令进行仿真时，未处理操作又发生了，则处理所述一个或多个其他指令集的这种未处理操作的现有机制可以被用来克服未处理操作。

所述未处理操作处理器可以发出指令以便清除数据处理系统的未处理操作状态。一些数据处理系统包括一个或者多个标志，所述标志用于记录数据处理系统是否处在未处理操作状态之中。在仿真所述第一指令集的所述指令之前，这种标志可能需要未处理操作处理器来清除。所述标志是否需要被清除取决于未处理操作的类型以及是否存在与那种未处理操作类型相关联的标志。

尽管本发明是属于通用型的，但是本发明却特别好地适用于如下系统，在所述系统中，第一指令集是可变长度指令集，所述一个或多个其他指令集则是固定长度指令集。这种组合是这样的：其中在固定指令集中不可能的新类型的未处理操作在可变长度指令集中却是可能的，并且其中，在当未处理操作自身结合可变长度第一指令集发生时，为处理在所述一个或多个其他指令集中的那些未处理操作而开发的机制并不是很容易地就适合于来处理所述未处理操作。

本发明在其中具有极强优势的特定情形是处理可变长度指令的提取中止。在这种情形下，可变长度指令可以跨越不止一个指令字，并且对于引起过所述未处理操作的那个指令字的提取，它也许能并不清除。这样就会在正确地处理所述未处理操作中引发困难。

本技术还特别好地适用于包含浮点系统或者子系统的系统，在所述系统中，第一指令集的单条指令可以生成多个浮点运算，或者发信号通知未处理浮点运算是不精确的。

本发明在其中具有极强优势的特定情形是在如下系统中，在所述系

统中，结合不精确地发信号通知未处理浮点运算，第一指令集的单条指令可以生成多个浮点运算。在这种情形下，让所述系统确定第一指令集的哪一条指令引发过未处理浮点运算也许是不可能的。

本发明通过触发使用所述一个或多个其他指令集的一条或多条指令来仿真第一指令集的如下指令来克服这一情形，所述指令在未处理浮点运算已被触发之时正在被执行。第一指令集中的、未处理浮点运算被触发之时正在被执行的指令可能是、也可能不是引发所述未处理浮点运算的那条指令。不管第一指令集中的、未处理浮点运算被触发之时正在被执行的指令是否引发过未处理浮点运算，对第一指令集中的、未处理浮点运算被触发之时正在被执行的指令的仿真，将能够使用于处理所述一个或多个其他指令集的这种未处理操作的现有机制被利用来克服所述未处理操作。

在使用所述一个或多个其他指令集的指令对第一指令集的所述指令进行仿真之前，如果存在一个未处理浮点运算标志，则所述未处理浮点运算处理器可能需要将其清除。

如果第一指令集中的、未处理浮点运算被触发之时正在被执行的指令是引发未处理浮点运算的那条指令，则在那条指令被仿真之时，相同的未处理浮点运算将再次发生，并且用于处理所述一个或多个其他指令集的这种未处理操作的现有机制将被用来克服所述未处理操作。

如果因为数据处理系统采用了不精确的未处理浮点运算检测，所以第一指令集中的、未处理浮点运算被触发之时正在被执行的指令不是引发未处理浮点运算的那条指令，则在那条指令被仿真之时，相同的未处理浮点运算将不会发生。在这一情况下，数据处理系统通常将使未处理浮点运算标志或等效机制来记录未处理浮点运算已经发生这一事实。此外，数据处理系统通常还将记录引发过未处理浮点运算的浮点运算。

清除用于记录未处理浮点运算已经发生这一事实的所述标志或者其他机制，通常将能够使用于处理所述一个或多个其他指令集的这种未处理操作的现有机制被利用来克服未处理操作。

在其他系统中，有必要测试所述标志或者其他机制，如果它指示未处理浮点运算已经被记录过了，则在清除所述标志或其他机制之前，

显式地利用所述一个或多个其他指令集的、用于处理这种未处理操作的机制。利用所述一个或多个其他指令集的、用于处理所述未处理操作的机制还可以自动地清除所述标志或其他机制，在这一情形下，不需要让未处理浮点运算处理器来显式地清除所述标志或者其他机制。

在另外的其他系统上，可以不必测试所述标志或其他机制。取而代之，利用所述一个或多个其他指令集的、用于处理所述未处理操作的机制可能就足够了。上述可以足够的原因在于，众所周知的是，凭借系统在未处理浮点运算处理器中执行代码这一事实，所述标志或者其他机制被置位，或者也可能是这种情况，即，所述机制的利用将涉及到测试所述标志或者其他机制以及可能后续对其进行的清除，它们都作为利用所述机制的整体的一部分。

不管在可以随不同数据处理系统而变化的未处理浮点处理器中所采用的确切技术是什么，在第一指令集中的、未处理浮点运算被触发之时正在被执行的指令不是引发未处理浮点运算的那条指令的情形下，在仿真第一指令集中的、引发过未处理浮点操的那条指令之前，所述未处理浮点运算将被弄清楚。在这一情况下，仿真第一指令集中的那条指令可能会、也可能不会引发未处理操作。然而，因为不知道它是否会引发未处理操作，所以，不可能简单地恢复执行，所以随着所述未处理浮点运算被重复地触发，并且未处理浮点运算处理器重复地重新执行所述指令，这样也许就会引发可能是无穷的循环。因此,第一指令集的指令必须要被仿真，并恢复执行后面的指令。

本技术还特别好地适用于如下的情形，在所述情形中，所述一个或多个其他指令集是处理器核心的本机指令集，所述处理器核心趋向于具有一组开发得很好的中止处理器，并且第一指令集是要求在以后的某个日子来为其添加支持的解释型指令集。这种可变长度解释型的第一指令集的特别例子是 Java 字节码指令。

从另一方面来看，本发明提供一种方法，用于在第一指令集和一个或多他其他指令集的程序指令的控制下处理数据，所述方法包括如下步骤:

检测所述第一指令集的程序指令的未处理操作；和

当检测所述未处理操作时,触发使用所述一个或多个其他指令集的

至少一个指令集的一条或多条指令来仿真所述第一指令集的所述指令。

从本发明的再一个方面来看，本发明还提供一种计算机程序产品，用于控制数据处理装置在第一指令集和一个或多他其他指令集的程序指令的控制下处理数据，所述计算机程序产品包括：

未处理操作处理器逻辑，可操作地在检测到未处理操作时，触发使用所述一个或多个其他指令集的至少一个指令集的一条或多条指令来仿真一个指令集中引起所述未处理操作的指令。

本发明除了将自身表示在装置和操作装置的方法方面，本发明还可以将自身表示成充当未处理操作处理器的计算机支持代码的形式。这种支持代码可以采用独立可记录的介质来加以分发，作为固件或采用某种其他方式被嵌入到嵌入式处理系统中。

本发明的实施例在以下将仅仅采用举例的方式并参考附图来加以描述，在所述附图中：

图 1 图示了一种并入字节码翻译硬件的数据处理系统；

图 2 示意性地图示了对字节码的软件指令解释；

图 3 是示意性地表示以序列终止指令结尾的代码片断在软件指令解释器之中操作的流程图；

图 4 是执行代替字节码的代码片断的例子；

图 5 图示了没有硬件字节码执行支持的数据处理系统的例子；

图 6 是图示当与图 5 的系统一起操作时软件指令解释器活动的流程图；

图 7 图示了在 Java 字节码和处理操作之间的映射；

图 8 图示了采用内容可编址存储器的形式的可编程翻译表；

图 9 图示了采用随机存取存储器形式的可编程翻译表；

图 10 是示意性地图示可编程翻译表的初始化和编程的流程图；

图 11 是示意性图示在系统之中执行 Java 字节码解释的处理流水线部分的方框图；

图 12 示意性地图示了跨越两个指令字和两个虚拟存储器页的可变长度指令；

图 13 示意性地图示了包括了用于处理图 12 所示预取中止类型的

机制的数据处理系统流水线部分;

图 14 给出了逻辑表达式，它是一种指定如何检测图 12 所示的预取中止类型的方式;

图 15 示意性地图示了用于中止处理和指令仿真的支持代码的装置;

图 16 是示意性地图示了用于处理可变长度字节码指令的预取中止所执行的处理的流程图;

图 17 图示了操作系统和受该操作系统控制的各种进程之间的关系;

图 18 图示了包括处理器核心和 Java 加速器的处理系统;

图 19 是示意性地图示正在控制 Java 加速器配置中的操作系统运行的流程图;

图 20 是示意性地图示 Java 虚拟机结合 Java 加速机制运行的流程图，其中，所述 Java 虚拟机在控制 Java 加速机制的配置中使用。

图 21 图示了并入图 1 中的字节码翻译硬件的数据处理系统，它还并入了浮点子系统;

图 22 图示了并入图 1 中的字节码翻译硬件和图 21 中的浮点子系统的数据处理系统，它还并入了浮点运算寄存器和未处理操作状态标志;

图 23 显示了为 Java 浮点指令所生成的 ARM 浮点指令;

图 24 显示了可以由 Java 加速硬件为 Java ‘dmul’ 和 ‘dcmpg’ 指令生成的 ARM 指令序列;

图 25 显示了当执行接着 ‘dcmpg’ 指令的 ‘dmul’ 指令时的操作序列，其中未处理浮点运算是通过执行由 Java 加速硬件为 Java ‘dmul’ 指令所生成的 ‘FCMPD’ 指令引发的，所示的操作序列适用于使用对应于图 22 的不精确的未处理操作检测的系统;

图 26 显示了在执行图 25 的 FMULD 指令之后的浮点运算寄存器的状态和未处理操作状态标志;

图 27 显示了当执行接着 ‘dcmpg’ 指令的 ‘dmul’ 指令时的操作序列，其中未处理浮点运算是通过执行由 Java 加速硬件为 Java ‘dcmpg’ 指令所生成的 ‘FCMPD’ 指令引发的，所表示的操作序列适用于使用对应于图 22 的不精确的未处理操作检测的系统;

图 28 显示了在执行图 27 的 FCMPD 指令之后的浮点运算寄存器的状态和未处理操作状态标志;

图 29 显示了当执行接着 'dcmpg' 指令的 'dmul' 指令时的操作序列，其中未处理浮点运算是通过执行由 Java 加速硬件为 Java 'dmul' 指令所生成的 'FMULD' 指令引发的，所表示的操作序列适用于使用对应于图 21 的精确的未处理操作检测的系统;

图 30 显示了当执行接着 'dcmpg' 指令的 'dmul' 指令时的操作序列，其中未处理浮点运算是通过执行由 Java 加速硬件为 Java 'dcmpg' 指令所生成的 'FCMPD' 指令引发的，所表示的操作序列适用于使用对应于图 21 的精确的未处理操作检测的系统.

图 1 图示了数据处理系统 2，它并入了处理器核心 4（诸如 ARM 处理器）以及字节码翻译硬件 6（也称为 Jazelle）。处理器核心 4 包括寄存器库 8、指令译码器 10 以及数据通路 12，所述处理器核心 4 用于对存储在寄存器库 8 的寄存器之中的数据值执行各种数据处理操作。所提供的寄存器 18 包括用于控制字节码翻译硬件 6 当前是启用还是禁止的标志 20。此外，所提供的寄存器 19 包括用于指示字节码翻译硬件当前是活动还是不活动的标志 21。换言之，标志 21 指示数据处理系统当前是执行 Java 字节码还是执行 ARM 指令。应该理解，在其它实施例中，寄存器 18 和 19 可以是包括了标志 20 和 21 这两者的单个寄存器。

在运行时，如果 Java 字节码处于执行中，并且字节码翻译硬件 6 是活动的，则 Java 字节码由字节码翻译硬件 6 接收，并且用于生成对应的 ARM 指令序列（在这一特定的没有限制的示例实施例中），或者生成至少表示 ARM 指令的处理器核心控制信号，然后它们被传递到处理器核心 4。因此，字节码翻译硬件 6 可以把单个 Java 字节码映射为可以被处理器核心 4 执行的对应的 ARM 指令序列。当字节码翻译硬件不活动时，它将被绕过并且可以将一般 ARM 指令供应到 ARM 指令译码器 10，以便根据其本机指令集控制处理器核心 4。应该理解，ARM 指令序列可以等同于 Thumb 指令序列和/或不同指令集中的指令的混合，而且这种替换被考虑和包括。

应该理解，字节码翻译硬件 6 仅仅可以对也许会碰到的可能的 Java 字节码的子集提供硬件翻译支持。某些 Java 字节码可能要求如此扩展和抽象的处理以至于在硬件中尝试将这些映射成对应的 ARM 指

令操作常常是低效率的。因此，当字节码翻译硬件 6 碰到这种没有硬件支持的字节码时，它将触发用 ARM 本机指令所写的软件指令解释器来执行由所述没有硬件支持的 Java 字节码指定的处理。

软件指令解释器可以被写入以便为所有可以被解释的可能 Java 字节码提供软件支持。如果字节码翻译硬件 6 存在并被启用，则只有那些没有硬件支持的 Java 字节码将正常地被提交到软件指令解释器之中的相关代码片段。然而，如果字节码翻译硬件 6 没有提供或者是禁止的（诸如在调试等期间），则所有的 Java 字节码都将被提交到软件指令解释器。

图 2 示意性地图示了软件指令解释器的活动。Java 字节码流 22 代表 Java 程序。这些 Java 字节码可以与操作数互相交错。因此，在执行给定的 Java 字节码之后，要执行的下一个 Java 字节码可能紧跟着出现在后面的字节位置，或者如果存在有插入的操作数字节，则可能是再后面的若干个字节位置。

如图 2 所示，遇到 Java 字节码 BC4，它得不到字节码翻译硬件 6 的支持。这将在字节码硬件翻译 6 之中触发如下异常，所述异常将引发使用字节码值 BC4 作为索引在指针表 24 中执行的查询，以便读取指向代码片段 26 的指针 P#4，所述代码片段 26 将执行得不到硬件支持的字节码 BC4 指定的处理。指针表的基地址值还可以存储在寄存器中。接着通过指向得不到支持的字节码 BC4 的 R14 进入所选定的代码片段。

如所图示的，因为存在 256 个可能的字节码值，所述指针表 24 包括 256 个指针。相似地，最大可提供 256 个 ARM 本机指令代码片段，用于执行由所有可能的 Java 字节码所指定的处理。（在两个字节码能够使用相同代码片段的情况下，可以少于 256 个）。为了提高处理速度，字节码翻译硬件 6 典型地为许多单个 Java 字节码提供硬件支持，在这一情况下，在软件指令解释器之中的对应代码片段将永远得不到使用，除非强制使用（诸如在调试期间或者在其它环境中（诸如在后面将要论述的预取中止等））。然而，因为这些通常是比较简单和比较短的代码片段，所以由于提供它们所产生的额外存储器开销就相对地小。此外，这种小的额外的存储器开销，要大于软件指令解释器的普遍性质以及在字节码翻译硬件不存在或者被禁止的情形下其处理所

有可能的 Java 字节码的能力所补偿的开销。

可以看出，图 2 的代码片段 26 中的每个都是以序列终止指令 BXJ 结尾的。这种序列终止指令 BXJ 的活动随着如图 3 所示的数据处理系统 2 的状态而变化。图 3 是采用高层原理的形式来图示由软件指令解释器之中的代码片段 26 所执行的处理的流程图。在步骤 28，正在被解释的 Java 字节码指定的操作被执行。在步骤 30，将要执行的下一个 Java 字节码从字节码流 22 中读取，并且在 Java 字节码流 22 之中对应下一个 Java 字节码的字节码指针被存储在寄存器库 8 中的寄存器（即 R14）之中。因此，对于图 2 的 Java 字节码 BC4，下一个 Java 字节码将是 BC5,并且指向 Java 字节码 BC5 的存储器单元的指针被加载到寄存器 R14。

在步骤 32，在指针表 24 之中对应于下一个 Java 字节码 BC5 的指针被从指针表 24 中读取，并且被存储在寄存器库 8 的寄存器即 R12 之中。

应该理解，图 3 图示了独立、顺序地被执行的步骤 28、30 以及 32。然而，根据已知的编程技术，步骤 30 和 32 的处理按照常规可以交织在步骤 28 的处理之中，以便利用在步骤 28 的处理过程中不用就会浪费的处理机会（周期）。因此，步骤 30 和 32 的处理可以使用相对小的执行速度开销而被实现。

步骤 34 使用被指定为操作数的寄存器 R14 来执行序列终止指令 BXJ。

在步骤 34 执行 BXJ 指令之前，系统的状态已经通过如下指针设置过了，所述指针是：指向 Java 字节码流 22 之中的下一个 Java 字节码的、正存储在寄存器 R14 之中的指针;以及指向对应于所述下一个 Java 字节码的代码片段、正存储在寄存器 R12 之中的指针。对特定寄存器的选择可以是变化的，并且没有、一个或者两个都被指定为针对序列终止指令的操作数或者是由所述体系结构所预定和定义的。

步骤 28、30、32 和 34 主要是软件步骤。在图 3 的步骤 34 之后的那些步骤主要是硬件步骤，并且是在没有独立地可标识的程序指令的情况下发生的。在步骤 36，硬件检测字节码翻译硬件 6 是否是活动的。它是通过读取字节码翻译硬件 6 存在和启用的寄存器标志值而进行的。用于确定活动的字节码翻译硬件 6 的存在的其它机制也是可能的。

如果字节码翻译硬件 6 存在的并且是启用的，则处理进行到步骤 38，在步骤 38，将控制与指定字节码指针的寄存器 R14 的内容一起传递给字节码翻译硬件 6，其中所述字节码指针指向字节码流 22 之中的、字节码翻译硬件 6 尝试将其作为它的下一个字节码来执行的字节码。所示的代码片段 26 的活动然后就终止。

可替换地，如果在步骤 36 的判定是这样的：没有字节码翻译硬件 6 或者字节码翻译硬件 6 是禁止的，则处理进行到步骤 40，在步骤 40，进行在本机 ARM 指令代码中的跳转，以便开始执行在软件指令解释器之中的、由存储在寄存器 R12 之中的地址所指向的代码片段。因此，对下一个代码片段的快速执行将被启动，从而得到在处理速度上的优势。

图 4 更加详细地图示了特定代码片段。这一特定例子是整数加法 Java 字节码，它的助记符为 iadd。

第一 ARM 本机指令使用寄存器 R14 中的已增量 1 的字节码指针，以便读取下一个字节码值（整数加指令没有任何跟随的字节码操作数，因此下一个字节码将紧跟着当前字节码）。寄存器 R14 中的字节码指针还以增量值被更新了。

第二和第三指令用来从堆栈中获取将要被相加的两个整数操作数值。

第四指令利用由于在寄存器 R0 上的寄存器互锁而不用就会浪费的处理周期，以便为存储在寄存器 R4 中的下一个字节码获取代码片段的地址值，并且将这一地址存储在寄存器 R12 之中。寄存器 Rexc 用于存储指向指针表 24 的开始的基指针。

第五指令执行由 Java 字节码指定的整数加。

第六指令将 Java 字节码的结果反向存储到堆栈中。

最后的指令是使用操作数 R12 所指定的序列终止指令 BXJ。如果要求软件解释，则寄存器 R12 存储对于软件解释下一个 Java 字节码所需要的 ARM 代码片段的地址。BXJ 指令的执行确定是否存在启用的字节码翻译硬件 6。如果它存在，则控制就同存储在寄存器 R14 之中的指定下一个字节码地址的操作数一起传递到这一字节码翻译硬件 6。如果活动的字节码翻译硬件 6 不存在，则开始执行由寄存器 R12 之中的地址值所指向的下一个字节码的代码片段。

图 5 示意性地图示了与图 1 相似的数据处理系统 42，例外的是，在这种情况下，不提供字节码翻译硬件 6。在这一系统中，标志 21 总是指示 ARM 指令处于执行之中，并且使用 BXJ 指令进入 Java 字节码的执行的尝试总是被当作好像字节码翻译硬件 6 是禁止的，标志 20 被忽略了的情况一样来处理。

图 6 图示了系统 42 在执行 Java 字节码时所执行的处理的流程图。这与图 3 的处理是相似的，原因在于，相同的软件解释器代码处于使用之中，除了在序列终止指令 BXJ 被执行时的情况之外，所以不可能有硬件字节码支持，并且由此处理总是这样继续，也就是跳转到执行由 R12 所指向的、作为下一个 Java 字节码的代码片段的代码片段。

应该理解，在这一情况之下的软件指令解释器是作为 ARM 本机指令被提供的。软件指令解释器（或者其它的支持代码）可以作为有其自身的版权的独立的计算机程序产品被提供。这一计算机程序产品可以通过记录介质（诸如软盘或者 CD）来发布或者可以通过网络链接动态地下载。在本发明特别好地适合于用在其中的嵌入式处理应用的情况下，软件指令解释器可以作为嵌入式系统中的只读存储器或者一些其它非易失性程序存储设备之中的固件而被提供。

图 7 图示了 Java 字节码和它们所指定的处理操作之间的关系。从图 7 可以看出，8 位 Java 字节码提供 256 个可能的不同字节码值。这些 Java 字节码中的第一个 203 从属于以 Java 标准所指定的、对应的处理操作（诸如前面所论述的 iadd）的固定式绑定。最后两个 Java 字节码即 254 和 255 在作为定义的实施的 Java 虚拟机规范之中进行论述。因此，Java 实施是随意分配固定式绑定给这些字节的。或者，Java 实施可以选择将这些作为可编程式绑定对待。Jazelle 为这些字节码指定固定式绑定。在包含的字节码值 203 和 253 之间，可编程式绑定可以按照用户的希望而指定。这些通常用于提供在字节码和处理操作之间的绑定，诸如在运行时被解析的快速形式字节码（参见《Java 虚拟机规范》，作者 Tim Lindholm 和 Frank Yellin，发行商 Addison Wesley，ISBN 0-201-63452-X）。

根据图 7 应该理解，尽管硬件加速的解释技术非常适合于处理固定式绑定，但是这些技术却不怎么适合于可编程式绑定。尽管有可能

使用软件解释技术来处理所有的可编程式绑定，诸如解释对应的代码片段所代表的相关字节码，但是由于在一些情况下可能成为性能关键字节码的内容而会是慢速的。

图 8 图示了可编程翻译表的一种形式。这一可编程翻译表 100 采用的是内容可寻址存储器的形式。要翻译的字节码被输入到 CAM 查询阵列 102。如果这一阵列 102 包含匹配的字节码条目，则生成命中，它引发指定要输出的值的对应操作，即:

如果在 CAM 表中存在匹配的字节码条目，则硬件使用指定操作的代码来确定要在硬件中执行的操作，执行上述操作并移动到下一个字节码;

如果在 CAM 表中不存在匹配的字节码条目，则字节码就被当作得不到硬件支持来处理并且调用其代码片段。

在这个例子中，指定操作的值是 4 位值，并且已经引起命中的 CAM 条目对应于字节码 bc6。根据图 7 可以看出，可能要进行这种可编程翻译的所有字节码的最高有效的两位是“1”，因此只有这些字节码的最低有效的 6 位需要被输入到阵列 102。

在这一例子的可编程翻译表 100 有 8 个条目。存在的条目数可以随着专用于这一任务所需要的硬件资源的量而变化。在一些例子中，可以只提供四个条目，不过在其它情况下 10 个条目也许是适当的。还有可能为每个可能的可编程式绑定字节码提供条目。

可以理解，如果可用的可编程映射资源首先被最关键的翻译占用，则不是很关键的翻译可能就接受软件解释。提供软件解释器结合可编程翻译表允许配置系统以及对生成的表进行编程，而不需要知道有多少表条目可以使用，因为如果表溢出，则所要求的翻译将被捕获（trap）并且由软件解释器来执行。

图 9 图示了可编程翻译表 104 的第二个例子。在这一例子中，翻译表采用随机存取存储器的形式来提供，所述随机存取存储器包括要输入到译码器 106 进行翻译的字节码,译码器 106 将字节码处理为到 4 位字的 RAM 阵列 108 的地址，每个字代表操作指定的代码。在这种情况下，指定操作的代码总是对应字节码。因此，这种类型的表使用额外的指定操作的代码，它指定“调用这一字节码的代码片段”。

图 10 是图示初始化和配置具有图 8 的例子的形式的可编程映射硬

件解释器的示意性的流程图。实际上，在这一流程图中所图示的活动的不同部分分别由软件初始化指令和对应于这些指令的硬件来执行。

在步骤 110，表初始化指令被执行，初始化指令用于清除全部现有的表条目，并且设置指向表中顶部条目的指针。接着，初始化代码可以执行从而使用诸如协处理器寄存器加载之类的程序指令将映射加载到翻译表中。这些表加载指令的不同形式可以随着特定情况和环境的变化而变化。可编程映射硬件解释器系统通过在步骤 112 接收程序指令值(诸如 Java 字节码)以及与其相关联的操作值来响应这些指令。在步骤 114，不被支持的操作捕获硬件检查:处于编程之中的操作值是否是得到那个可编程映射硬件解释器支持的操作值。不同的可编程映射硬件解释器可以支持不同的操作值组，因此可以为其提供它们自己的专用捕获硬件。捕获硬件可以相对简单些，假如特定系统例如知道它支持操作值 0，1，2，3，4，5，6，7，8，10 而不支持 9 的话。在步骤 114 的硬件比较器能够比较操作值与值 9 是否相等，并且如果检测到 9，则通过把处理转向步骤 116 而拒绝编程。

假设步骤 114 指示操作值是得到支持的，则步骤 118 检查来确定是否已经到达可编程映射表的尾部。如果可编程映射表已经满了，则处理再次进行到步骤 116 而不添加新的映射。将步骤 118 提供在硬件之中意味着：在硬件只是拒绝溢出条目的情况下，支持代码可以尝试来编程可编程映射表而不用知道有多少条目是可以使用的。因此，程序员应该将最关键的映射放在表的开始，从而编程来确保这些占用的时隙（slot）可以使用。对支持代码来说避免必须知道有多少可编程时隙可以使用意味着：单一组支持代码可以运行在多种平台上。

假设表具有空条目，则在步骤 120，将新映射写入到所述空条目，表指针然后在步骤 122 推进。

在步骤 116，系统对将要被编程到可编程映射表中的更多的程序指令值进行测试。步骤 116 通常是软件步骤，在初始化系统期间支持代码尝试对它所希望的数量的映射进行编程。

在初始化图 9 所示的 RAM 表的情况下，结合图 10 的上述的过程接着可以进行下列的修改:

在步骤 110，通过设置图 9 的阵列 108 中的所有表条目为“调用这一字节码的字节码片断”而不是通过设置图 8 的阵列 102 来清除表，

从而使每一表条目不匹配任何字节码;

在步骤 110，没有要初始化的翻译表指针;

步骤 118 不存在，因为没有翻译表指针;

步骤 120 变为“将操作值写到由程序指令值所指示的表条目”;

以及

步骤 122 不存在，因为没有翻译表指针。

图 11 图示了可以用于 Java 字节码解释的处理流水线的一部分。处理流水线 124 包括翻译级 126 和 Java 译码级 128。后续级 130 可以采用多种不同的形式，这要取决于特定的实施。

来自 Java 字节码流的字可以交替地加载到摆动缓冲器（swing buffer）132 的两个半部分。通常，多路复用器 133 选择来自摆动缓冲器 132 的当前字节码以及其操作数，并通过多路复用器 137 将它交付给锁存器 134.如果因为流水线已经被清空或者一些其它原因而使摆动缓冲器 132 为空，则多路复用器 135 选择直接来自 Java 字节码流的进入的字的正确的字节码并且将它交付给锁存器 134。

译码字节码的第一周期是通过第一周期译码器 146 而进行的，第一周期译码器 146 对锁存器 134 中的字节码进行动作。为了允许在硬件支持的字节码带有操作数的的情况，其它多路复用器选择来自摆动缓冲器 132 的操作数并且将它们交付到第一周期译码器 146。这些多路复用器未在图中示出，它们与多路复用器 133 是类似的。典型地，第一周期译码器 146 对操作数输入的松散定时要求要多于对字节码输入的，从而对于操作数就不要求与多路复用器 135 和 137 以及锁存器 134 所提供的旁路路径相似的旁路路径。

如果摆动缓冲器 132 包括的操作数字节对于锁存器 134 中的字节码是不充足的，则在充足的操作数字节可以使用之前，第一周期译码器 146 停止。

第一周期译码器 146 的输出是通过多路复用器 142 被传递到后续流水线级 130 的 ARM 指令（或者代表 ARM 指令的处理器核心控制的信号组）。第二输出是通过多路复用器 139 被写到锁存器 138 的指定操作的代码。指定操作的代码包括位 140，它指定这是否是单周期字节码。

在下一个周期，接着的字节码通过如前所述的第一周期译码器 146

来译码。如果位 140 指示单周期字节码，则那个字节码就被译码并且控制如前所述的后续流水线阶段 130。

如果位 140 相反指示的是多周期字节码，则第一周期译码器 146 就停止，并且多周期或者翻译译码器 144 就译码在锁存器 138 中的指定操作的代码，以便产生 ARM 指令（或者是代表 ARM 指令的处理器核心控制的信号组），多路复用器 142 将 ARM 指令传递到后续的流水线级 130 而不是传递到第一周期译码器 146 的对应输出。多周期或翻译译码器还产生通过多路复用器 139 被写到锁存器 138(同样地不写到第一周期译码器 146 的对应输出）的其它指定操作的代码。这个其他指定操作的代码也包括位 140,它指定这是否是为多周期字节码生成的最后的 ARM 指令。在位 140 指示最后的 ARM 指令已经产生之前，多周期或翻译译码器 144 继续如上面所描述的生成其它 ARM 指令，接着第一周期译码器 146 取消停止并为后面的字节码产生第一个 ARM 指令。

当锁存器 134 中的字节码需要翻译时，前述的过程采用三种方式加以修改。第一,字节码由多路复用器 133 从摆动缓冲器 132 中提取并且由字节码翻译器 136 进行翻译，产生通过多路复用器 139 写到锁存器 138 的指定操作的代码。这一指定操作的代码具有位 140，它被设置以便指示还没有为当前字节码产生最后的 ARM 指令，从而多路复用器 142 和多路复用器 139 在翻译的字节码的第一周期将选择多周期或翻译译码器 144 的输出而不选择第一周期译码 146 的输出。

第二，多周期或翻译译码器 144 生成要传递到后续流水线级 130 的所有 ARM 指令以及它们的要反向写到锁存器 138 的对应的其它指定操作的代码，而不是仅仅在第一周期之后按照常规为不要求翻译的字节码生成那些指令。

第三，如果字节码是通过多路复用器 135 被直接写到锁存器 134 的，并且因此不存在于摆动缓冲器 132 中，以及在前一个周期不可能得到字节码翻译器 136 的翻译，则第一周期译码器 146 发信号通知字节码翻译器 136：它必须重新启动并停止一个周期。这样就保证：当第一周译码器 146 取消停止时，锁存器 138 为翻译的字节码保持有效的指定操作的代码。

从图 11 可以看出，提供翻译流水线级启用可编程翻译步骤所要求的处理以便有效地被隐藏或叠合到流水线之中，因为缓冲了的指令可

以按照要求提前被翻译并流进流水线的其余部分。

在图 11 中可以看出，在这一例子实施例中，可以考虑固定映射硬件解释器主要由第一周期译码器 146 以及运行在如下模式中的多周期或翻译译码器 144 所构成，在所述模式下，其中多周期或翻译译码器 144 译码已经由第一周期译码器 146 进行译码的多周期字节码。在这一例子中的可编程映射硬件解释器可以考虑由字节码翻译器 136 和多周期或翻译译码器 144（在这一例子中它在翻译可编程字节码之后运行）来构成。固定映射硬件解释器以及可编程映射硬件解释器可以采用多种不同的方式来提供，并且从抽象的观点来看可以共享重要的公共硬件同时还保留它们不同的功能。所有这些不同的可能性是包括在这里所描述的技术之中的。

图 12 图示了跨越了虚拟存储器页边界 204 的两个 32 位指令字 200，202。它可以是 1kB 的页边界，不过其它的页大小是可能的。

第一指令字 200 位于完全（properly）映射在虚拟存储器系统的虚拟存储页之中。第二指令字 202 位于在这一级没有被映射到虚拟存储器系统的虚拟存储器页中。因此，2 字节的可变长度指令 206 具有在指令字 200 中的第一字节以及在指令字 202 中的第二字节，它将具有与其第二字节相关联的预取中止。常规预取中止处理机制例如仅仅支持对齐指令的指令字，这些机制不能够处理这一情况，并且能够例如尝试来检查和修复包括可变长度指令 206 的第一字节的指令字 200 的读取而不是集中于实际上导致所述中止的、包括可变长度指令字 206 的第二字节的指令字 202。

图 13 图示了数据处理系统中用于处理 Java 字节码的指令流水线 208 的一部分，它包括用于处理图 12 中所示类型预取中止的机制。指令缓冲器包括两个指令字寄存器 210 和 212，其中的每个存储 32 位的指令字。Java 字节码每个长度是 8 位，伴随有 0 个或者多个操作数值。一组多路复用器 214 根据当前 Java 字节码指针位置从指令字寄存器 210 和 212 中选择合适的字节，所述当前 Java 字节码指针位置指示要译码的当前 Java 字节码指令的第一字节的地址。

与指令字寄存器 210 和 212 中的每个相关联的是各个指令地址寄存器 216 和 218 以及预取中止标志寄存器 220 和 222。这些相关联的寄存器分别存储指令字所相关的指令地址以及当那个指令从存储器系

统中被读取时是否出现预取中止。这一信息与指令字本身一起沿着流水线传递，因为顺着流水线进一步往下，这一信息通常是需要的。

如果需要，多路复用器 224，226 和 228 允许绕过输入缓冲器装置。这种类型的操作在上面已经论述了。应该理解，指令流水线 208（为简洁起见）没有显示前述指令流水线的所有特征。相似地，前述的指令流水线也没有显示指令流水线 208 的所有技术特征。实际上，系统可以被提供为具有在两个图示的指令流水线中显示的技术特征的组合。

在指令流水线 208 的字节码译码级，字节码译码器 230 响应来自多路复用器 224 的至少一个 Java 字节码以及可选地响应来自多路复用器 226 和 228 的一个或两个操作数字节，以便生成映射的指令或对应的控制信号从而传递到流水线的其它级来执行对应于译码的 Java 字节码的处理。

如果已经发生了图 12 所示类型的预取中止，则尽管 Java 字节码本身可能是有效的，但是跟随其后的操作数值将是无效的并且正确的操作直到修复了预取中止时才会发生。字节码异常发生器 232 响应来自寄存器 216 和 218 的指令字地址以及响应来自寄存器 220 和 222 的预取中止标志，以便检测图 12 所示情况的类型的发生。如果字节码异常发生器 232 检测到了这种情况，然后它就强制多路复用器 234 向下一级发出由字节码异常发生器本身而不是由字节码译码器 230 所生成的指令或控制信号。通过触发执行仿真中止 Java 字节码的 ARM 32 位代码片段而不是允许硬件解释那个 Java 字节码，字节码异常发生器 232 对检测到图 12 情形下的预取中止作出响应。因此，经历过预取中止的可变长度 Java 指令 206 本身将不会得到执行，相反地，它会被 32 位 ARM 指令序列所替代。当加载一个或多个操作数字节时，用于仿真该指令的 ARM 指令有可能遭受数据中止，这些数据中止发生是因为与当那些字节最初作为第二指令字 202 的一部分被读取时所发生的预取中止的原因相同的原因，并且在执行 ARM 32 位代码片段期间还有可能发生其它的预取和数据中止。所有这些中止发生在 ARM 指令执行期间，因此通过现有的中止异常处理器例程将会得到正确的处理。

照这样，在读取字节码时发生的预取中止就被遏制了（即，不被传递到 ARM 核心）。相反，ARM 指令序列得到执行，并且与这些 ARM 指令一起发生的任何中止将使用现有的机制来处理，这样就跳过了存

在问题的字节码。在执行用于用中止来替换字节码的仿真的 ARM 指令之后，对字节码的执行可以恢复。

如果字节码本身碰到预取中止，则标记有预取中止的 ARM 指令就被传递到 ARM 流水线的其余部分。如果且当它到达了流水线的执行级，则它将引发预取中止异常：这是处理 ARM 指令的预取中止的非常标准的方式。

如果字节码没有碰到预取中止，但是它的一个或多个操作数碰到了（如图 12 所示），则调用那个字节码的软件代码片段。被传递到 ARM 流水线其余部分的、用于引发代码片段被调用的任何 ARM 指令将不作预取中止的标记，因此如果且当它们到达了流水线的执行级，ARM 指令将会正常地执行。

图 14 图示了可以由字节码异常发生器 232 用来检测图 12 所示情况类型的逻辑表达式类型。图 13 的摆动缓冲器的任一半个部分指示为“Half1”（块 210，216，220 构成半个部分，块 212，218，222 构成另外的半个部分，在图 13 中用这些元件周围的虚线指示），它当前保存第一指令字（图 12 的 200），用“Half2”指示摆动缓冲器的另外半个部分，它保存第二指令字（图 12 的 202）。假设 PA(Half1)意思是块 220 和 222 中的任一个的内容在 Half1 中，对于 Half2 也是一样。

那么，图 12 所示情况的指示符是：PA(Half1)是假，PA(Half2)是真，字节码加上其操作数跨越了摆动缓冲器的两个半个部分的边界。（事实是：页边界被标记了，仅仅是因为那是正常的要求，对于它来说两个 PA()值不相同是可能的）。

在优选设计（诸如摆动缓冲器半个部分中的每个部分存储一个字以及受硬件支持的字节码被限制在最多两个操作数）中，用于确定字节码加上其操作数是否跨越边界的公式如下：

（（操作数的数目 = 1）AND（bcaddr[1:0]=11））

OR（（操作数的数目 = 2）AND（bcaddr[1]=1））

其中：bcaddr 是字节码的地址。这样就可导出图 14 所示的逻辑表达式。

用于标识预取中止的其它技术是可以使用的，诸如在存储器页边界的预定距离之内开始的可变长指令。

图 15 示意性地图示了与 Java 字节码解释相关联的支持代码的结构。这与前述的图是相似的，但是在这一情况下，图示了包括指向由字节码异常事件触发的字节码异常处理代码片段的指针。因此，每个 Java 字节码具有仿真其操作的相关联的 ARM 代码片段。而且，可能发生的每个字节码异常都具有相关联的 ARM 异常处理代码部分。在所示情况下，字节码预取中止处理例程 236 被提供以便在检测到上述类型的预取中止时由字节码异常发生器 232 将它触发。这一中止处理代码 236 通过标识在触发过它的可变长度指令的开始处的字节码而进行工作，接着调用在代码片段集合中那个字节码对应的仿真代码片段。

图 16 是示意性地图示字节码异常发生器 232 的操作和后续处理的流程图。步骤 238 用于确定图 14 的表达式是否为真。如果表达式为假，则这一过程结束。

如果步骤 238 已经指示了图 12 所示情况的类型，则步骤 246 被执行，它触发将要由字节码异常发生器 232 启动的字节码预取中止异常。字节码异常发生器 232 可以仅仅触发执行 ARM 代码字节码预取中止处理器 236。中止处理器 236 在步骤 248 用于标识可变长度指令的开始字节码，然后在步骤 250 触发执行仿真那个标识的字节码的 ARM 指令的代码片段。

上述用于处理预取中止的机制在如下情况中可很好地工作，在这些情况下，有 4 个或稍少的操作数（即，总量是 5 个或稍少的字节），否则字节码及其操作数有可能溢出第二缓冲器。实际上，字节码（优选地，为其提供硬件加速机制）都具有 0，1，2 个操作数，而其余字节码在所有情况下都使用软件处理（主要是由于它们的复杂性）。

图 17 图示了用于控制多个用户态进程 302，304，306 以及 308 的操作系统 300。操作系统 300 在管态下运行而其它进程 302，304，306，308 在用户态下运行，用户态下所具有的对系统的配置控制参数的访问权要少于操作系统 300 在管态下所具有的。

如图 17 所示，进程 302 和 308 分别涉及到不同的 Java 虚拟机。这些 Java 虚拟机 320，308 中的每一个都具有其自己的由字节码翻译映射数据 310，312 以及配置寄存器数据 314，316 所构成的配置数据。实际上，可以理解，单一的 Java 加速硬件组被提供以便执行这两个进程 302，308，但是当这些不同的进程正在使用 Java 加速硬件时，它

们中的每个都要求使用它们相关联的配置数据 310，312，314，316 来配置 Java 加速硬件。因此，当操作系统 300 切换到使用 Java 加速硬件执行不同于使用了那个硬件的前一个进程的进程时，则 Java 加速硬件就应该被重新初始化和重新配置。操作系统 300 本身并不执行 Java 加速硬件的重新初始化和重新配置，但是指示它应该通过将与 Java 加速硬件相关联的配置无效指示符设置到无效状态来实现。

图 18 示意性地图示了包括带有本机指令集（例如 ARM 指令集）的处理器核心 320 以及相关联的 Java 加速硬件 322 的数据处理系统 318。存储器 324 存储可以采用 ARM 指令或 Java 字节码形式的计算机程序代码。在 Java 字节码的情况下，这些被传递到 Java 加速硬件 322，Java 加速硬件 322 用于将它们解释为然后可以由处理器核心 320 执行的 ARM 指令流（或者对应于 ARM 指令的控制信号）。Java 加速硬件 322 包括字节码翻译表 326，它需要为希望执行 Java 字节码的每个 Java 虚拟机进行编程。在 Java 加速硬件 322 中还提供有配置数据寄存器 328 和操作系统控制寄存器 330，以便控制其配置。包括在操作系统控制寄存器 330 之中的是采用标志 CV 形式的配置无效指示符，当它被置位时，指示 Java 加速硬件 322 的配置是有效的，当取消对其的置位时，指示它是无效的。

如果配置有效指示符对应于 Java 加速硬件 322 的、处于无效形式的配置数据，则尝试执行 Java 字节码时的 Java 加速硬件 322 对配置有效指示符作出响应，以便触发配置无效异常。配置无效异常处理器可以是采用上述为预取中止处理器提供 ARM 代码例程相似的方式提供的 ARM 代码例程。硬件机制提供在 Java 加速硬件 322 之中，用于在配置异常被触发时以及在新的有效配置数据实际上已经被写入位置之前，将配置有效指示符设置为指示配置数据是有效的形式。尽管在配置数据实际上已经被写入之前，采用这种方式设置配置有效指示符直觉上好像是计数器，但是这种方法在通过设置配置数据能够避免与进程交换局部方式一起出现的问题上具有相当明显的优势。然后，配置异常例程为 Java 虚拟机建立所需的配置数据，配置异常例程通过写入前述的字节码翻译表条目以及所需要的任何其它配置数据寄存器值 328 而与 Java 虚拟机保持一致。配置异常代码必须确保：在由 Java 加速硬件 322 接手任何其它的任务之前，对配置数据的写入必须完成。

图 19 示意性地图示了操作系统 300 的操作。在步骤 332，操作系统等待以检测进程切换。当检测到进程切换时，步骤 334 确定新进程是否使用 Java 加速硬件 322（如前述也称为 Jazelle）。如果不使用 Java 加速硬件 322，则处理进行到步骤 336，在该步骤，在进行到将执行转移到新进程的步骤 339 之前，将 Java 加速硬件 322 禁止。如果使用 Java 加速硬件 322，则处理进行到步骤 338，在该步骤，确定正被调用的新进程与由操作系统 300 记录的、存储的 Java 加速硬件 322 的当前所有者是否相同。如果所有者没有改变（即，新进程实际上与使用了 Java 加速硬件 322 的上一个进程相同），则处理进行到步骤 337，在该步骤，在进行到步骤 339 之前启用 Java 加速硬件 322。如果新进程不是所存储的当前所有者，则处理进行到步骤 340，在该步骤中，配置有效指示符被置位来指示 Java 加速硬件 322 的当前配置是无效的。这样就限制了操作系统用来管理这一配置变化的责任，对配置数据的实际更新作为任务就留给 Java 加速硬件 322 使用它自己的异常处理机制来操作。

在步骤 340 之后，步骤 342 用于在执行控制的转移被传递到步骤 337 然后再到步骤 339 之前将所存储的当前所有者更新为新进程。

图 20 图示了由 Java 加速硬件 322 所执行的操作。在步骤 344，Java 加速硬件 322 等待以便接收字节码来执行。当接收到字节码时，该硬件使用步骤 346 检查：配置有效指示符是否显示 Java 加速硬件 322 的配置是有效的。如果配置是有效的，则处理就进行到步骤 348，在该步骤执行所接收的字节码。

如果配置是无效的，则处理进行到步骤 350，在该步骤，Java 加速硬件 322 使用硬件机制来置位配置有效指示符以便显示配置是有效的。如果希望的话，这还可以由异常处理器中的程序指令来进行。步骤 352 用于触发配置无效异常。配置无效异常处理器可以提供为指向代码片段的指针表以及合适的代码片段的组合以便处理所涉及到的每个异常，诸如指令的软件仿真、预取中止（它们都已经在上面论述过了）、或者这一例子中的配置异常。

步骤 354 用于执行组成配置无效异常、以及用于将所要求的配置数据写到 Java 加速硬件 322 的 ARM 代码。这一 ARM 代码可以采用协处理器的寄存器写序列的形式以便增加可编程翻译表 326 以及其它配置

寄存器 330。在步骤 354 之后，步骤 356 反向跳转到 Java 字节码程序以便重新尝试执行原始字节码。

如果进程切换发生在步骤 354 或者 358 期间，则至此时所建立的配置有可能会被其它进程改为无效，并且配置有效指示符会被操作系统清零。在图 20 的过程中，这导致再次进入 344-346-350-352-354 循环，即导致要从开始重新尝试重新配置。当字节码最终实际上得到了执行时，则该配置被保证是有效的。

图 21 图示了图 1 所示的、还并入有浮点子系统的数据处理系统。当未处理浮点运算发生时，浮点子系统提供用于处理 ARM 代码中的未处理浮点运算的机制。

这种子系统的例子是英格兰剑桥 ARM 有限公司的 VFP 软件仿真器系统。在 VFP 软件仿真器系统的例子中，所有的浮点运算都被当作未处理浮点运算来处理，因为没有可用的硬件来执行浮点运算。因此，所有的浮点运算都是使用所提供的用于仿真 ARM 代码中 VFP 行为的机制进行处理的。

在这种系统的情况下，未处理浮点运算是精确的，也就是说，未处理浮点运算的检测点与未处理浮点运算的出现点相同。

图 22 图示了如图 1 和图 21 所示的、还并入了浮点运算寄存器和未处理运算状态标志的数据处理系统。

这种子系统的例子是英格兰剑桥 ARM 有限公司的 VFP 硬件系统。在 VFP 硬件系统的情况下，只有特定类型的浮点运算被当作未处理浮点运算来处理，其余的由 VFP 硬件处理。

可能从属于未处理浮点运算的运算类包括:

- 除零;
- 涉及 NaN 的运算;
- 涉及无穷大的运算;
- 涉及反向规格化数的运算;

在这种系统的情况下，未处理浮点运算可以是不精确的，也就是说，未处理浮点运算的检测点可以不必与未处理浮点运算的出现点相同。

未处理 VFP 运算出现在当 VFP 协处理器拒绝接受通常构成 ARM 指令流的一部分的 VFP 指令时，但是当存在图 1 所示的字节码翻译器时，

未处理 VFP 运算还可能是已经被翻译成 ARM 和 VFP 指令的组合的字节码所导致的结果。

在未处理 VFP 运算作为 ARM 指令流的一部分的情况下，用于处理未处理 VFP 运算的 ARM 机制将生成未定义的指令异常并执行安装在未定义的指令矢量上的未定义指令处理器。

在 VFP 软件仿真器系统的情况下，所有 VFP 运算都被当作未处理 VFP 运算并且应用相同的 ARM 机制，生成未定义的指令异常并且执行未定义的指令处理器。

当未处理 VFP 运算作为 ARM 指令流的一部分出现时，未定义的指令处理器能够通过检查指令流而知道：引发未处理 VFP 运算的指令是否的确是 VFP 指令，而不是一些其它类型的未定义的指令，并且当未定义的指令处理器在特权模式下执行时，它能够发出所需要的协处理器指令以便从 VFP 协处理器中提取它所需要的任何内部状态，然后在软件中完成所需要的指令。未定义的指令处理器将使用在 ARM 指令流和 VFP 的内部状态中都标识了的指令，以便处理未处理操作。

在许多 VFP 的实施中，引发未处理操作的指令可以不同于在检测到未处理操作时正在执行的指令。未处理操作可以由在较早前所发出的指令引发，并且与后续的 ARM 指令并行地执行，除了它碰到未处理状态之外。VFP 通过拒绝接受后面的 VFP 指令，并且强制进入 VFP 未定义指令处理器而发信号通知这种状态，所述 VFP 未定义指令处理器能够询问 VFP 以便找到未处理操作的原始原因。

当 Jazelle 被集成到包括 VFP 子系统的系统中时，下列应用：

Java 浮点指令是通过使用一组与 VFP 指令直接对应的信号在核心之中直接发出对应的 VFP 指令而得到翻译的。

如果 VFP 碰到未处理运算，它可以发信号通知未处理运算状态。

假如 ARM 指令流中的 VFP 指令发信号通知了不正确的操作，则 Jazelle 截取未处理运算信号，以防止它被送到核心以及防止未定义指令处理器按照常规执行。反之，Jazelle 生成由 Jazelle VM 支持代码处理的 VFP 异常。

当碰到这样的 Jazelle VFP 异常时，VM 支持代码会执行 VFP “不操作”指令，即，使 Jazelle 状态保持不变的任何 VFP 指令，诸如 FMRX Rd，FPSCR 指令。这样就使 VFP 硬件与支持代码保持同步并且结合未

处理操作状态标志完成由浮点运算寄存器所指示的任何 VFP 操作的操作，在这一例子中，当刚碰到未处理操作时，该未处理操作状态标志就应该被置位。一旦完成了操作，未处理操作状态标志就被清零。

该方法利用了这样的事实：由 Jazelle 发出的指令序列是可以重启动的，如在 2000 年 10 月 5 日所提交的共同待审的英国专利申请号为 0024402.0 的申请，在此全部引入以供参考。结合本发明的技术使用上述参考文献中所描述的技术允许重新启动引发生成 VFP 指令的指令，其中，该 VFP 指令引发未处理操作。

图 23 图示了用于每个 Java 浮点运算的、由 Java 字节码翻译器所发出的对应 VFP 指令。注意，仅仅显示了被发出的 VFP 指令，Java 字节码翻译器可以结合 VFP 指令发出附加的 ARM 指令。Jazelle 字节码翻译器也可以发出附加 VFP 加载和存储指令，以便加载和存储浮点值。

图 24 图示了指令序列或者对应于可以由 Jazelle 字节码翻译器为由其后跟着 'dcmpg' 字节码的 'dmul' 字节码组成的 Java 字节码序列发出的指令的信号。假如（dmul，dcmpg）字节码序列在当双精度寄存器 D0,D1 以及 D2 分别保存 Java 执行堆栈的从顶部起的第三、从顶部起的第二以及顶部元素时，以及字节码序列的整数结果被期望放在整数寄存器 R0 时被执行，所示的序列将会出现，。

图 25，27，29 以及 30 图示了未处理浮点运算发生在翻译的指令序列的各个点时的操作序列。图 25 和 29 图示了在未处理浮点运算是由 FMULD 指令引发时的操作序列。图 27 和 30 图示了未处理浮点运算是由 FCMPD 指令引发时的操作序列。图 25 和 27 图示了发信号通知未处理浮点运算是不精确时的操作序列。图 29 和 30 图示了发信号通知未处理浮点运算是精确时的操作序列。

可以看出，有四个可能的事件序列：

1）图 25：不精确未处理操作检测，发出信号通知未处理操作的 Java 字节码与引发未处理操作的 Java 字节码不相同。

2）图 27：不精确未处理操作检测，尽管系统使用不精确未处理操作检测的事实，但发出信号通知未处理操作的 Java 字节码与引发所述未处理操作的 Java 字节码却是相同。这是因为第二个 Java 字节码 'dcmpg' 为一个 Java 字节码发出了两个 VFP 指令，其中的第一个引发未处理操作，第二个发信号通知所述未处理操作。

3)图 29: 精确未处理操作检测，发出信号通知未处理操作的 Java 字节码与引发未处理操作的 Java 字节码相同。

4)图 30: 精确未处理操作检测，发出信号通知未处理操作的 Java 字节码与引发未处理操作的 Java 字节码相同，但是却不知道两个 VFP 指令中的哪个是由于执行实际上引发并发信号通知未处理操作的 'dcmpg' 字节码而被发出的。

上面提到的重启动技术和本发明的技术的组合允许正确地处理所有这些可能的事件序列。

图 26 和 28 图示了紧跟在引发了未处理操作之后的浮点运算寄存器和未处理操作状态标志的状态，分别对应图 25 和图 27 所示的操作序列。

应该参考如下: 均在 2000 年 10 月 5 日提交的共同待审的英国专利申请 0024399.8, 0024402.0, 0024404.6, 0024396.4, 在 2000 年 11 月 20 日提交的英国专利申请 0028249.1 和在 2000 年 12 月 7 日提交的美国专利申请 09/731,060(它们也描述了 Java 字节码解释系统)。这些共同待审的申请的公开内容在全部引入以供参考。

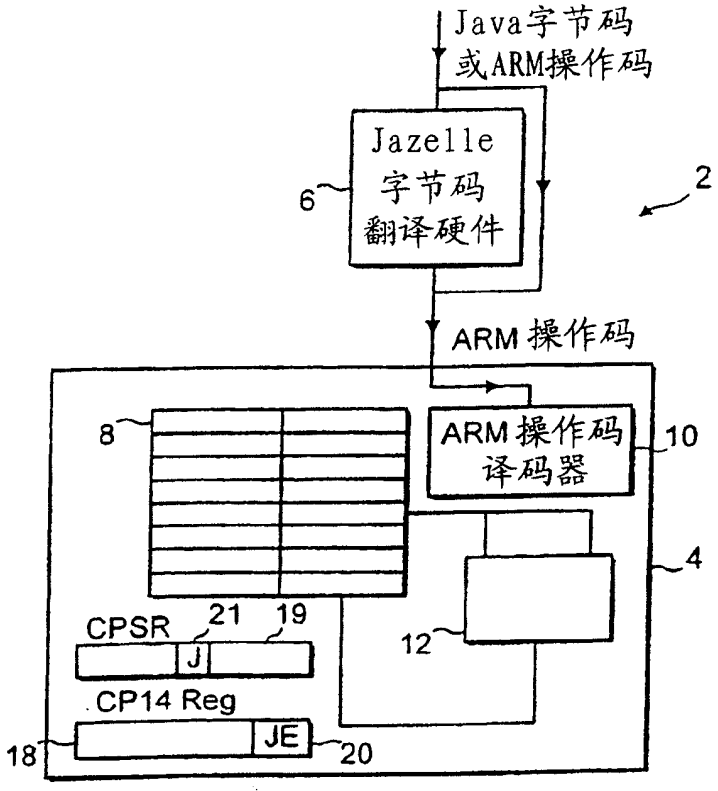

Java字节码
或ARM操作码
Jazelle
字节码
翻译硬件
6
2
ARM 操作码
8
ARM 操作码
译码器
10
4
12
CPSR
21
19
J
CP14 Reg
JE
18
20

图 1

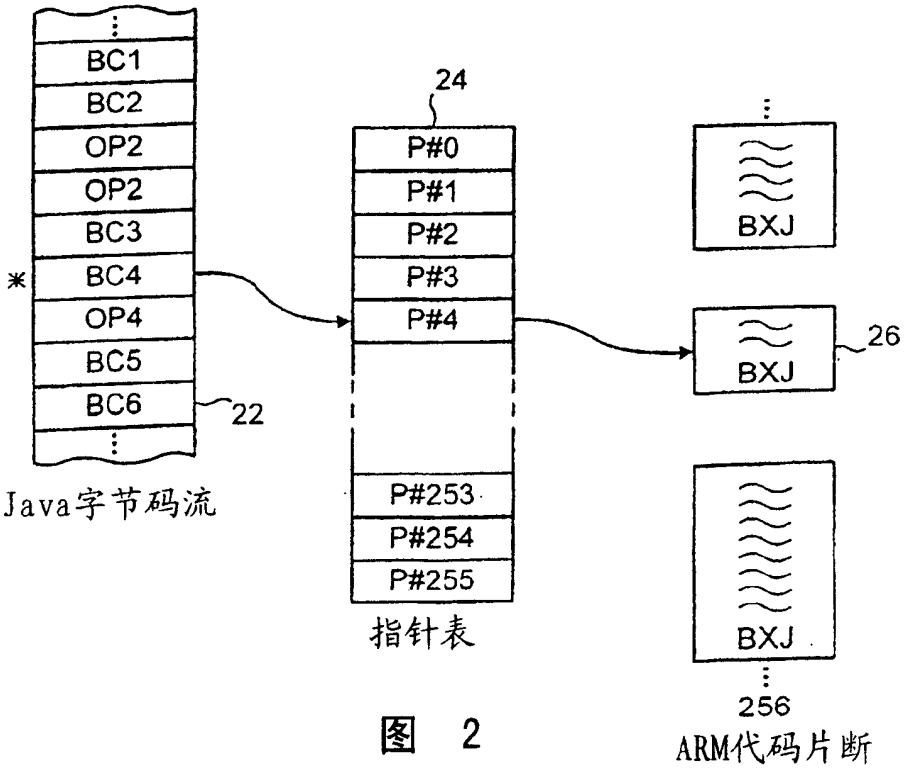

BC1
BC2
OP2
OP2
BC3
BC4
OP4
BC5
BC6
22
Java字节码流
24
P#0
P#1
P#2
P#3
P#4
P#253
P#254
P#255
指针表
BXJ
BXJ
26
BXJ
256
ARM代码片断

图 2

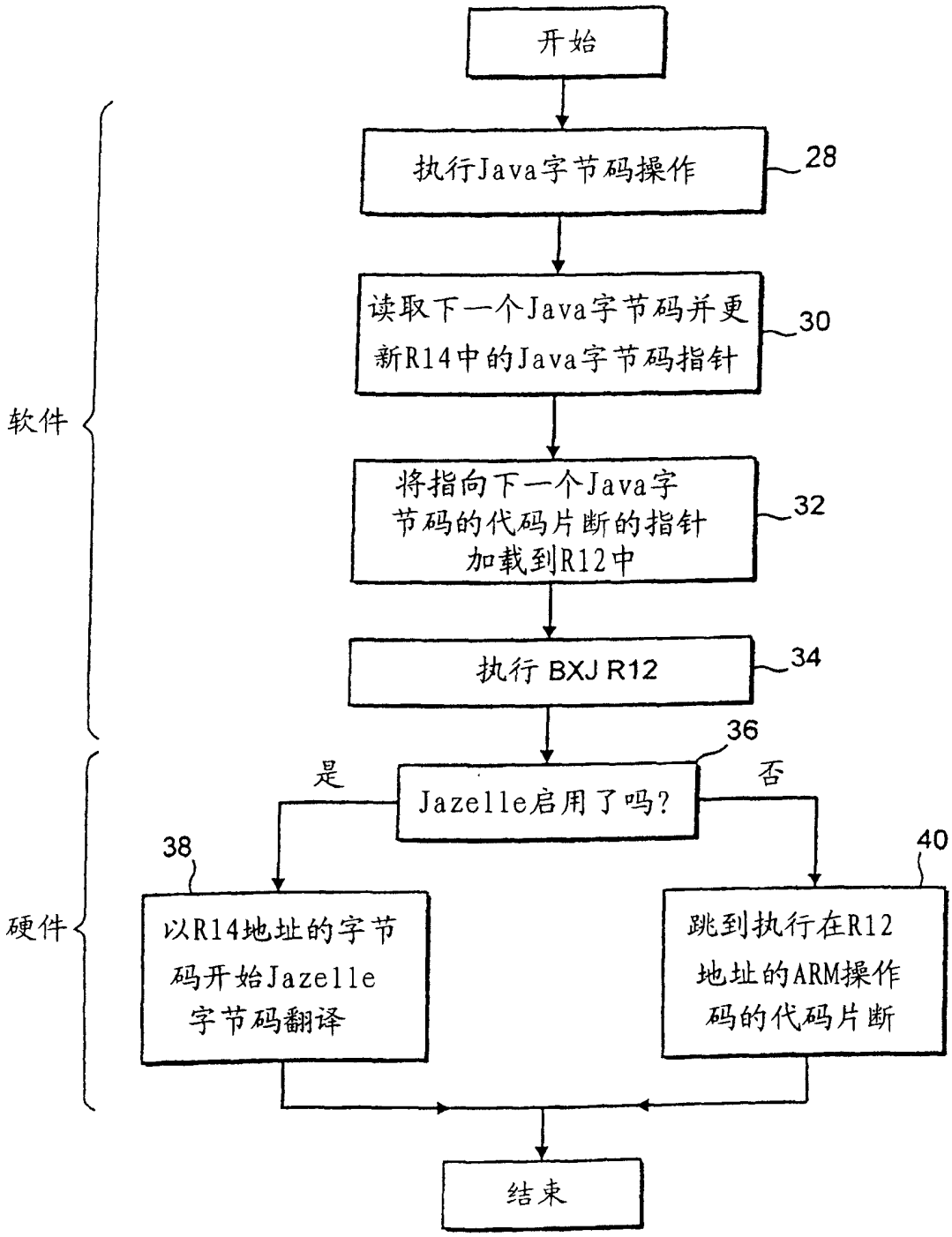

开始
执行Java字节码操作
28
读取下一个Java字节码并更新R14中的Java字节码指针
30
将指向下一个Java字节码的代码片断的指针加载到R12中
32
执行 BXJ R12
34
36
Jazelle启用了吗?
是
否
38
以R14地址的字节码开始Jazelle字节码翻译
40
跳到执行在R12地址的ARM操作码的代码片断
结束
软件
硬件

图 3

```
do_iadd
LDRB    R4, [R14, #1]! ———————— 加载下一个Java字节码并更新字节码指针
LDR     R1, [Rstack, #-4]! ———— 从堆栈中弹出第一操作数
LDR     R0, [Rstack, #-4]! ———— 从堆栈中弹出第二操作数
LDR     R12, [Rexc, R4, LSL #2] —— 获取下一个字节码的代码片断地址
ADD     R0, R0, R1 ———————— 执行整数加
STR     R0, [Rstack], #4 ———— 将结果压入堆栈
BXJ     R12 ———————————— 在硬件/软件中执行下一个字节码
```

图 4

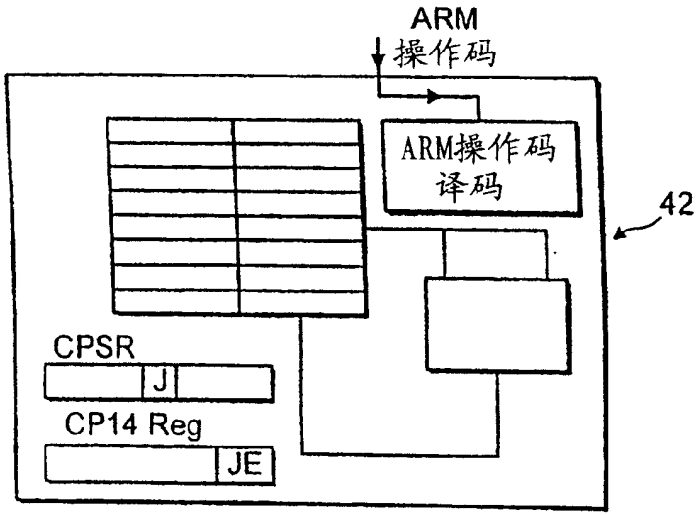


图 5

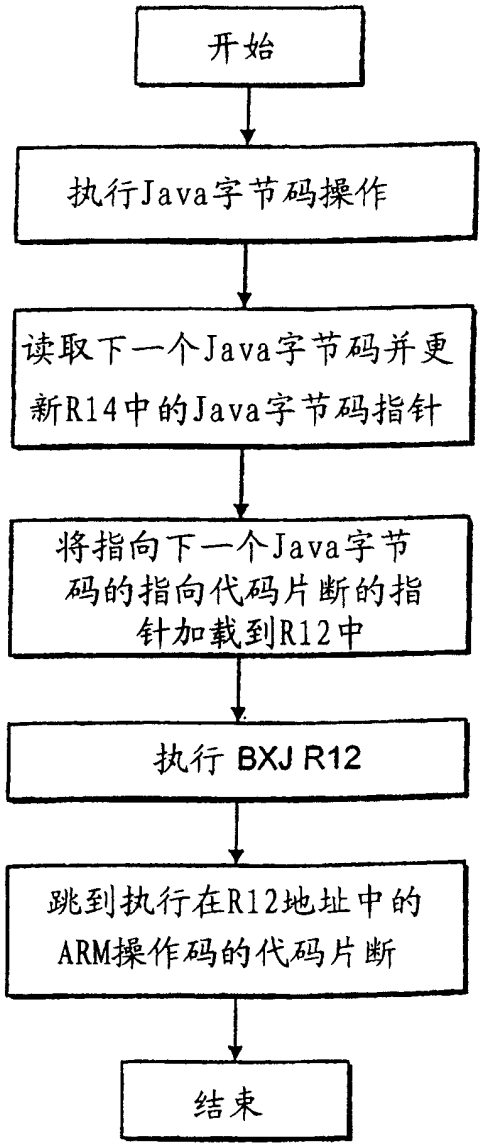

开始
执行Java字节码操作
读取下一个Java字节码并更新R14中的Java字节码指针
将指向下一个Java字节码的指向代码片断的指针加载到R12中
执行 BXJ R12
跳到执行在R12地址中的ARM操作码的代码片断
结束

图 6

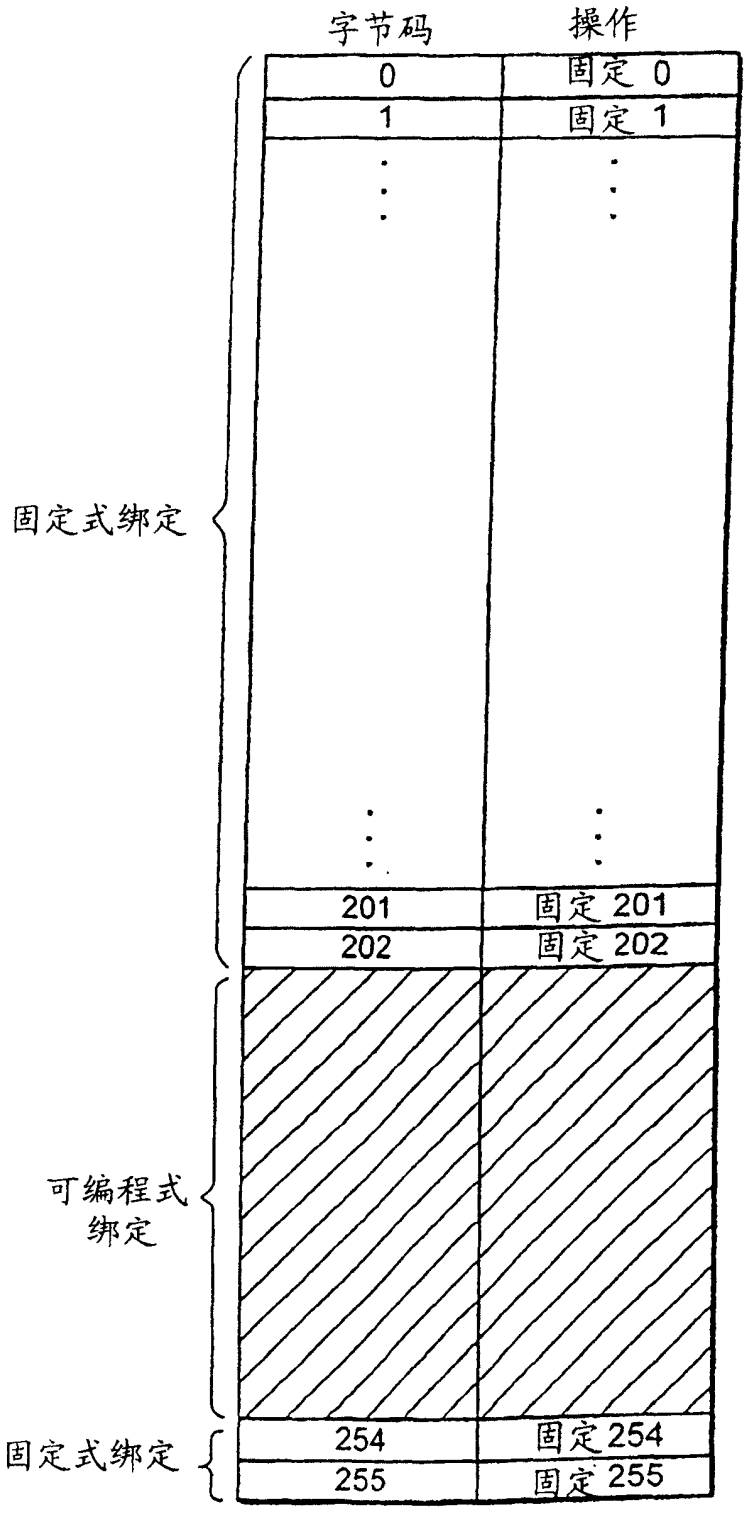

字节码
操作
0
固定 0
1
固定 1
201
固定 201
202
固定 202
254
固定254
255
固定 255
固定式绑定
可编程式绑定
固定式绑定

图 7

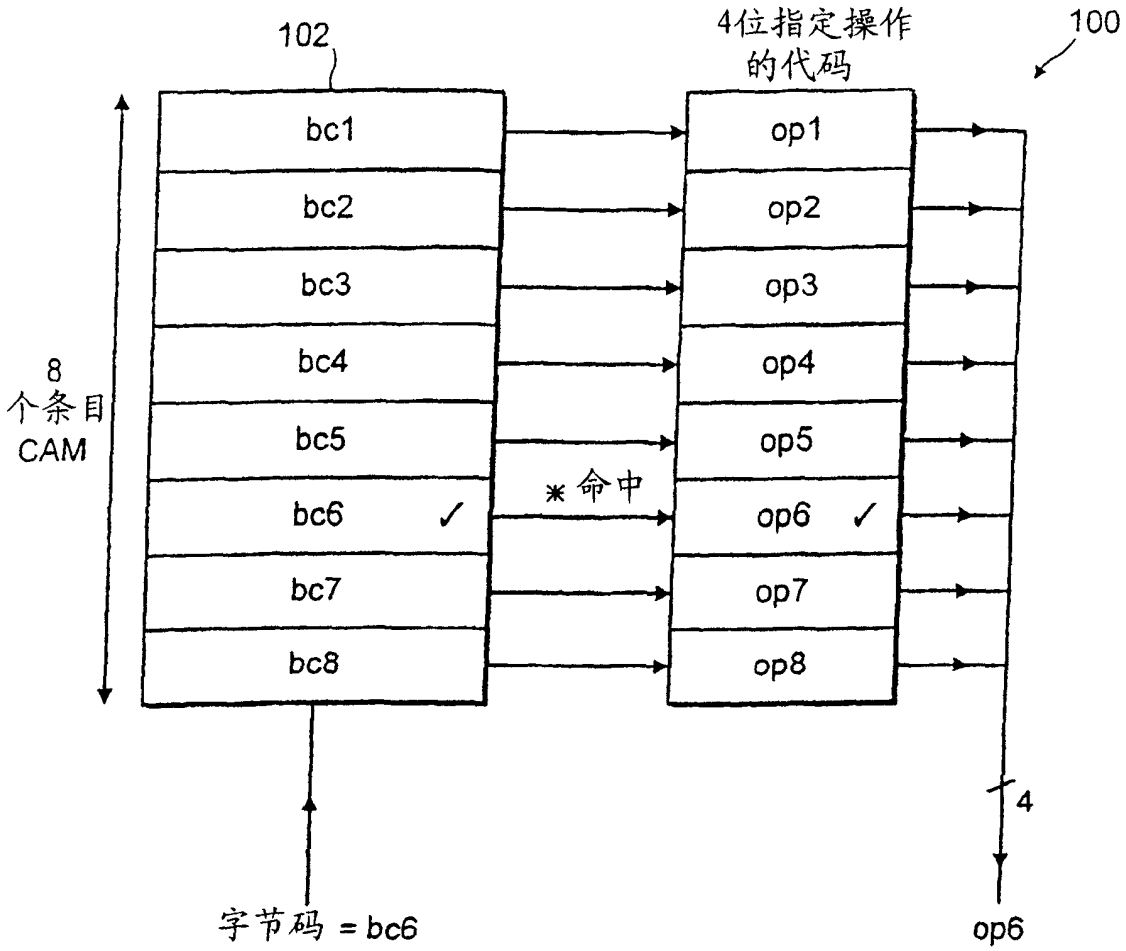

102
4位指定操作
的代码
100
bc1
bc2
bc3
bc4
bc5
bc6
bc7
bc8
op1
op2
op3
op4
op5
op6
op7
op8
8
个条目
CAM
✳命中
4
字节码 = bc6
op6

图 8

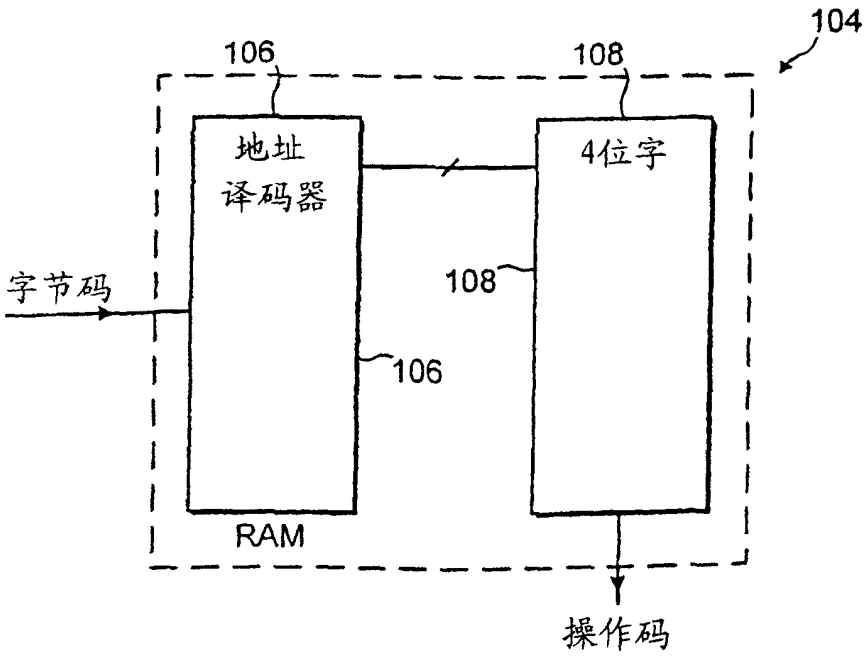

104
106
108
地址
译码器
4位字
字节码
108
106
RAM
操作码

图 9

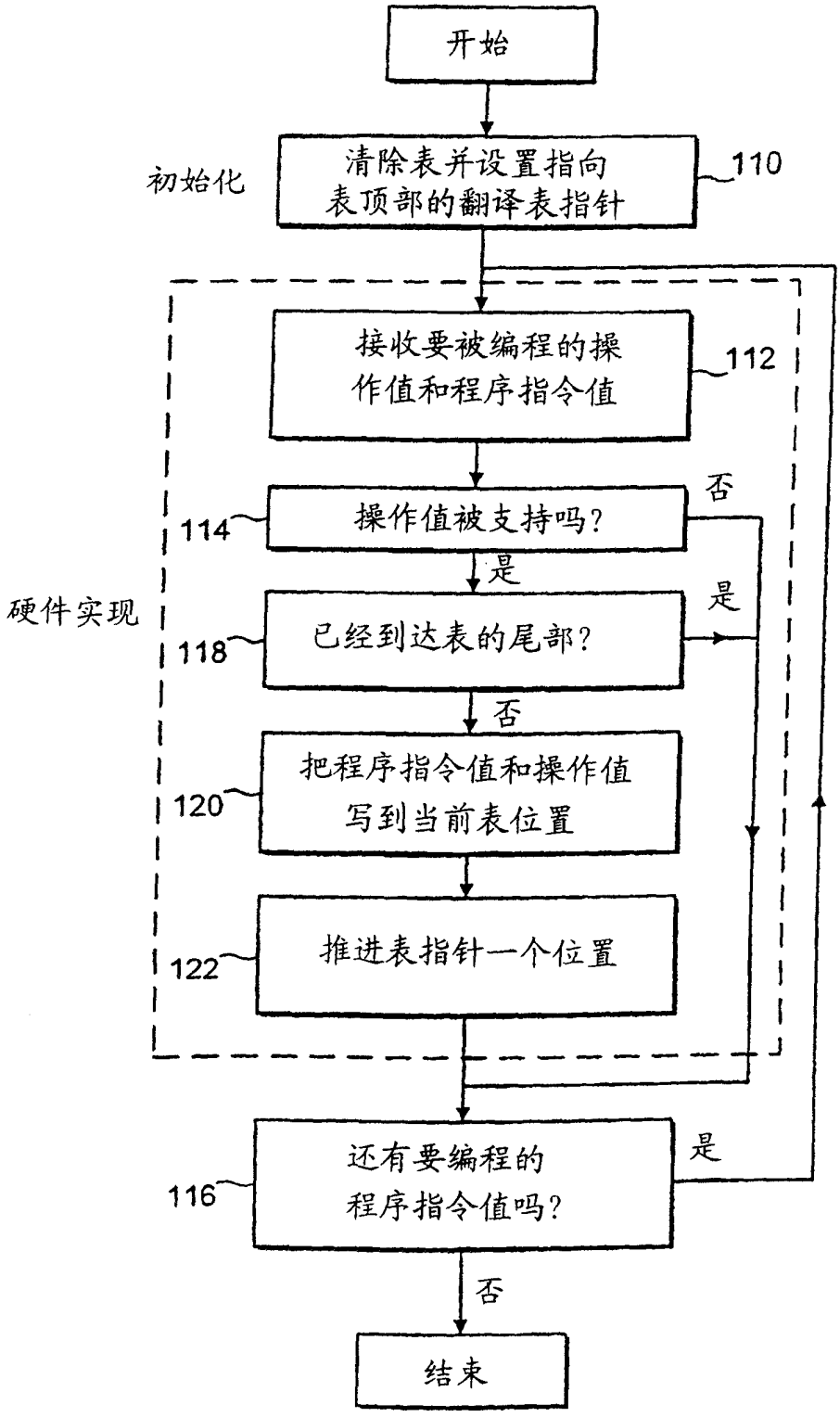

开始
初始化
清除表并设置指向表顶部的翻译表指针
110
接收要被编程的操作值和程序指令值
112
操作值被支持吗?
114
否
是
硬件实现
已经到达表的尾部?
118
是
否
把程序指令值和操作值写到当前表位置
120
推进表指针一个位置
122
还有要编程的程序指令值吗?
116
是
否
结束

图 10

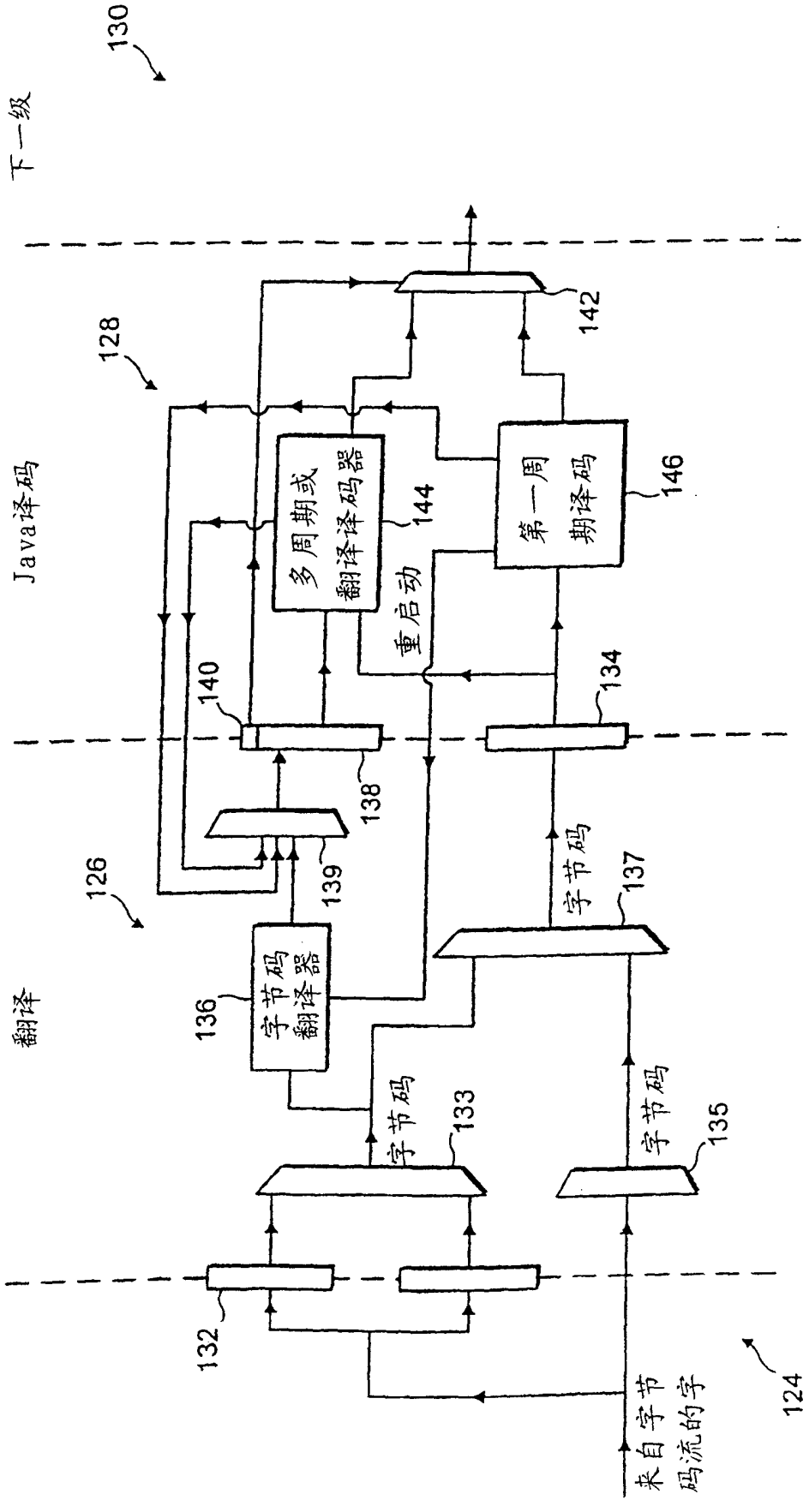

130
下一级
128
Java译码
126
翻译
142
多周期或翻译译码器
144
第一周期译码器
146
重启动
140
134
138
139
字节码翻译器
136
字节码
137
字节码
133
字节码
135
132
来自字节码流的字
124

图 11

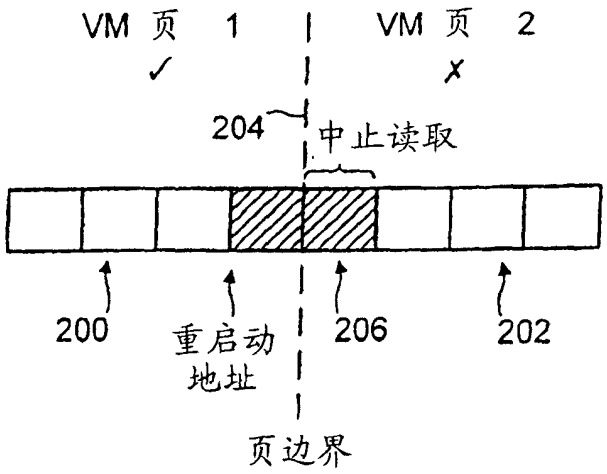


图 12

(PA (Half1) = 假) AND (PA (Half2) = 真)

AND

(((操作数数目 = 1) AND (bcadd [1:0] = 11))
OR ((操作数数目 = 2) AND (bcadd [1] = 1)))

图 14

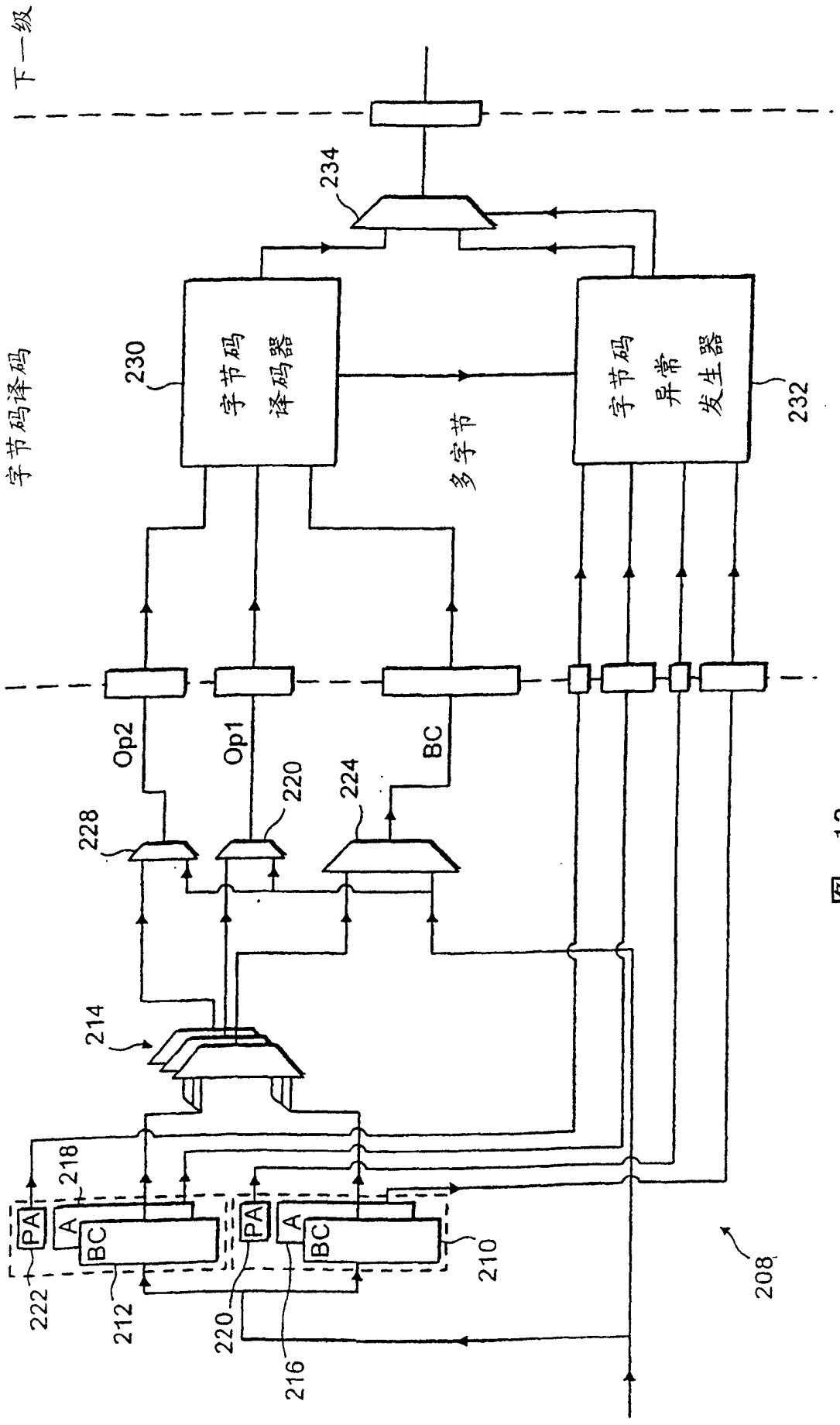

下一级
234
字节码译码
230
字节码译码器
多字节
字节码异常发生器
232
Op2
Op1
BC
220
224
228
214
218
PA
A
BC
222
212
220
216
210
208

图 13

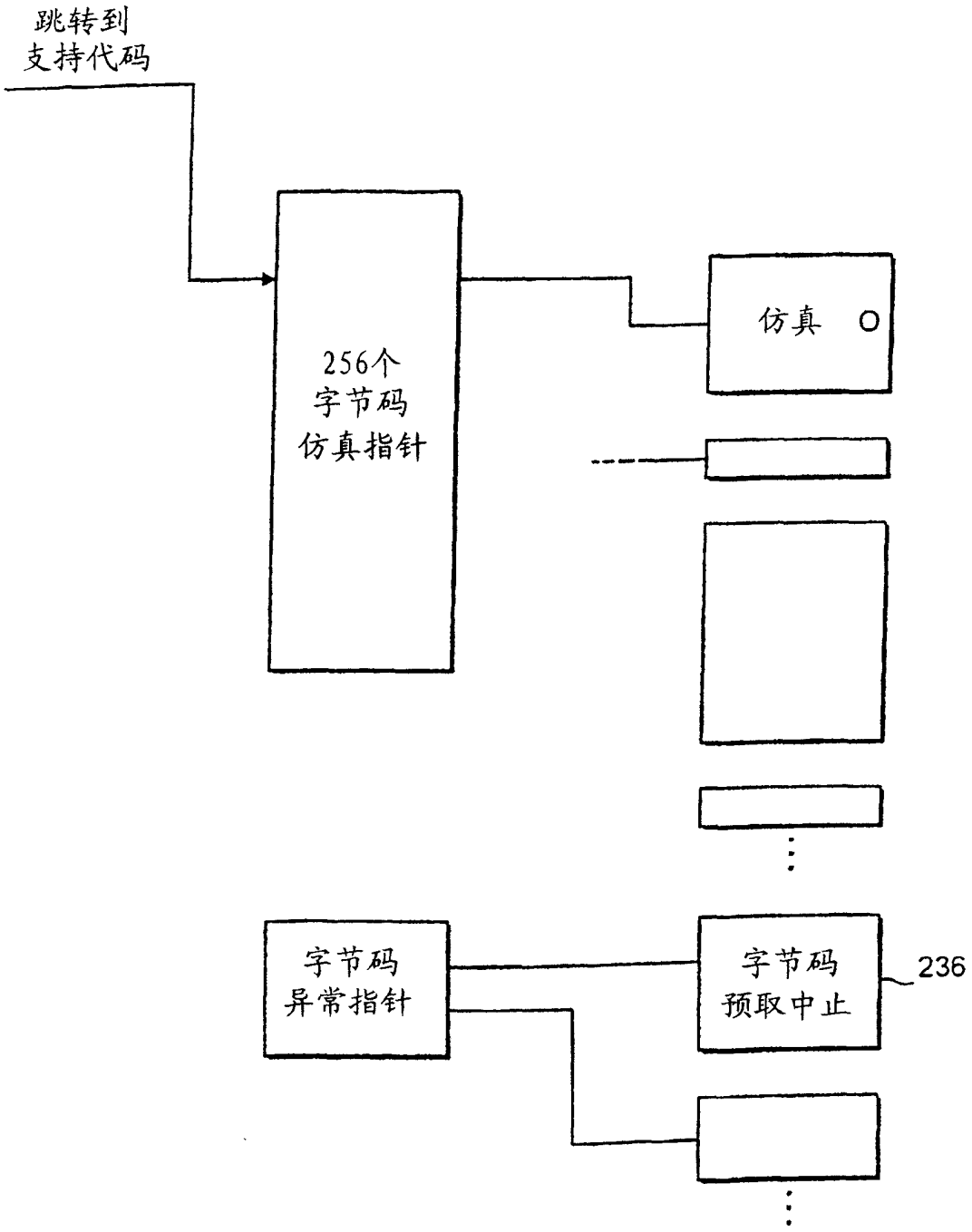


图 15

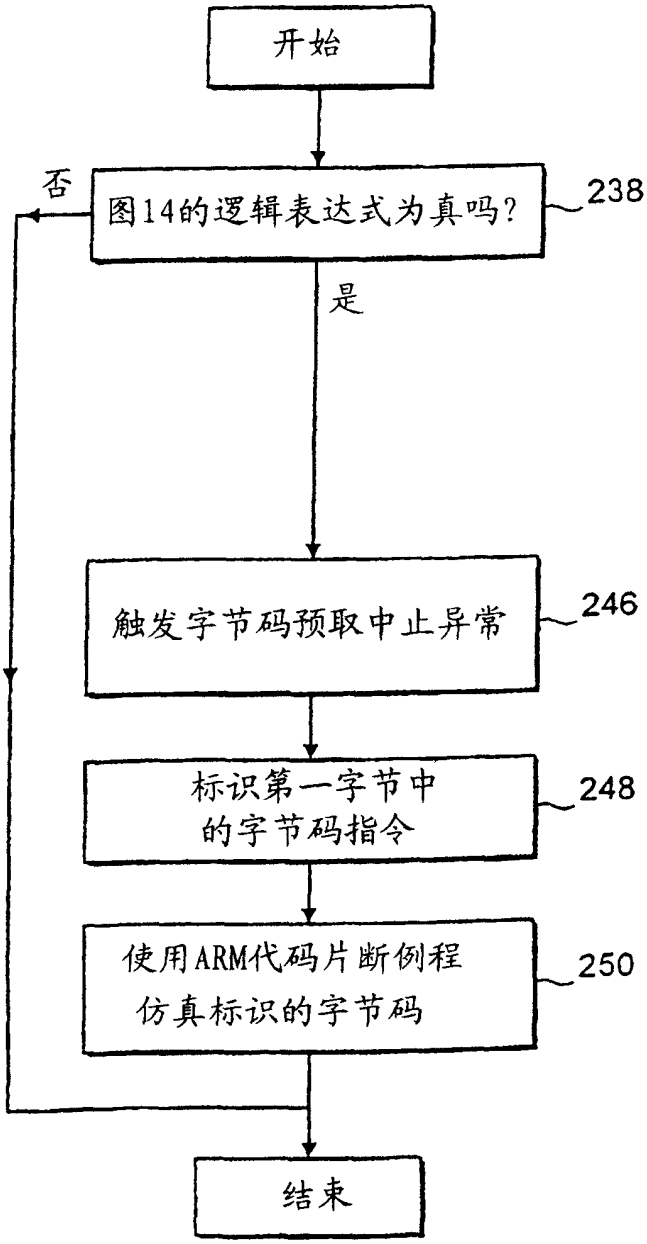

开始
否
图14的逻辑表达式为真吗?
238
是
触发字节码预取中止异常
246
标识第一字节中
的字节码指令
248
使用ARM代码片断例程
仿真标识的字节码
250
结束

图 16

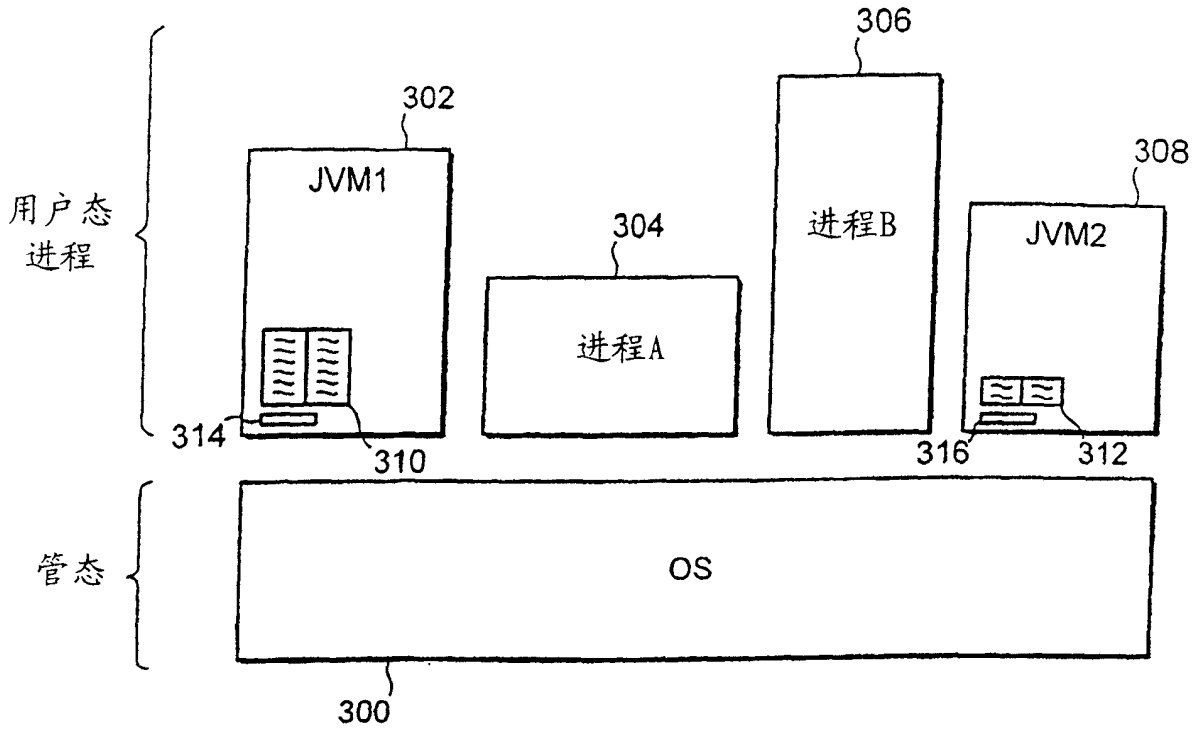

306
302
JVM1
308
用户态
进程
304
进程B
JVM2
进程A
314
310
316
312
管态
OS
300

图 17

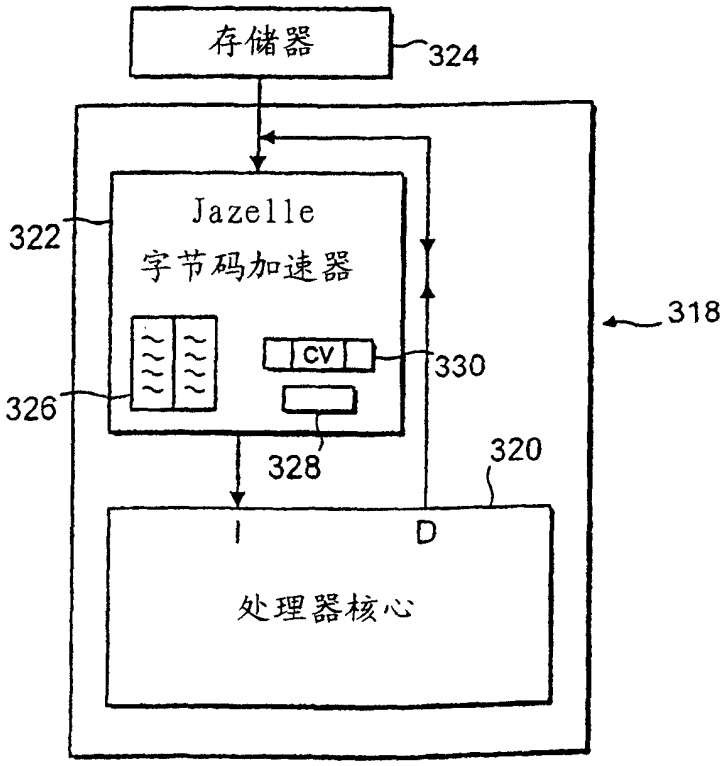

存储器
324
Jazelle
字节码加速器
322
318
CV
330
326
328
320
I
D
处理器核心

图 18

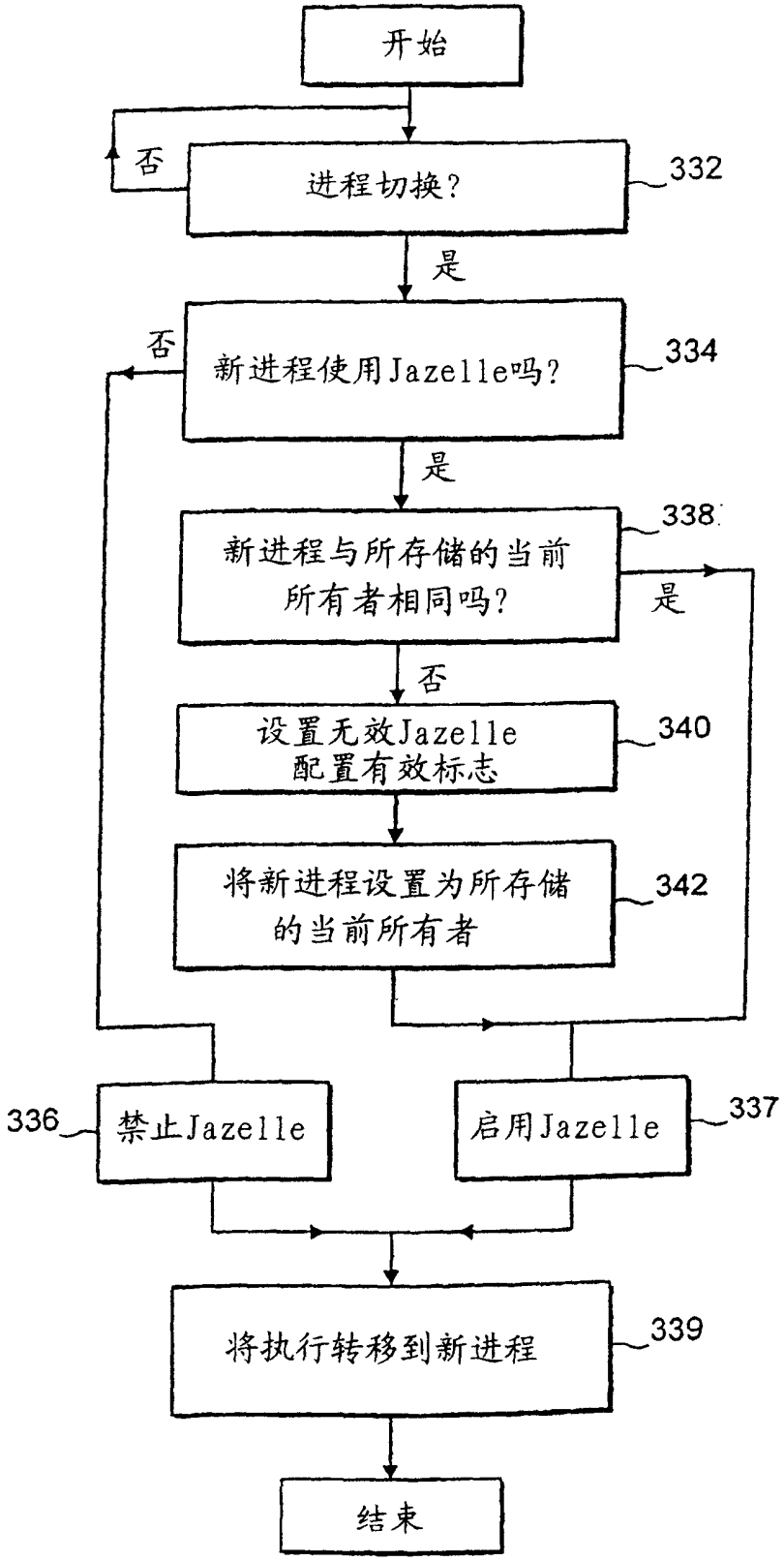

开始
否
进程切换?
332
是
否
新进程使用Jazelle吗?
334
是
新进程与所存储的当前所有者相同吗?
338
是
否
设置无效Jazelle配置有效标志
340
将新进程设置为所存储的当前所有者
342
336
禁止Jazelle
启用Jazelle
337
将执行转移到新进程
339
结束

图 19

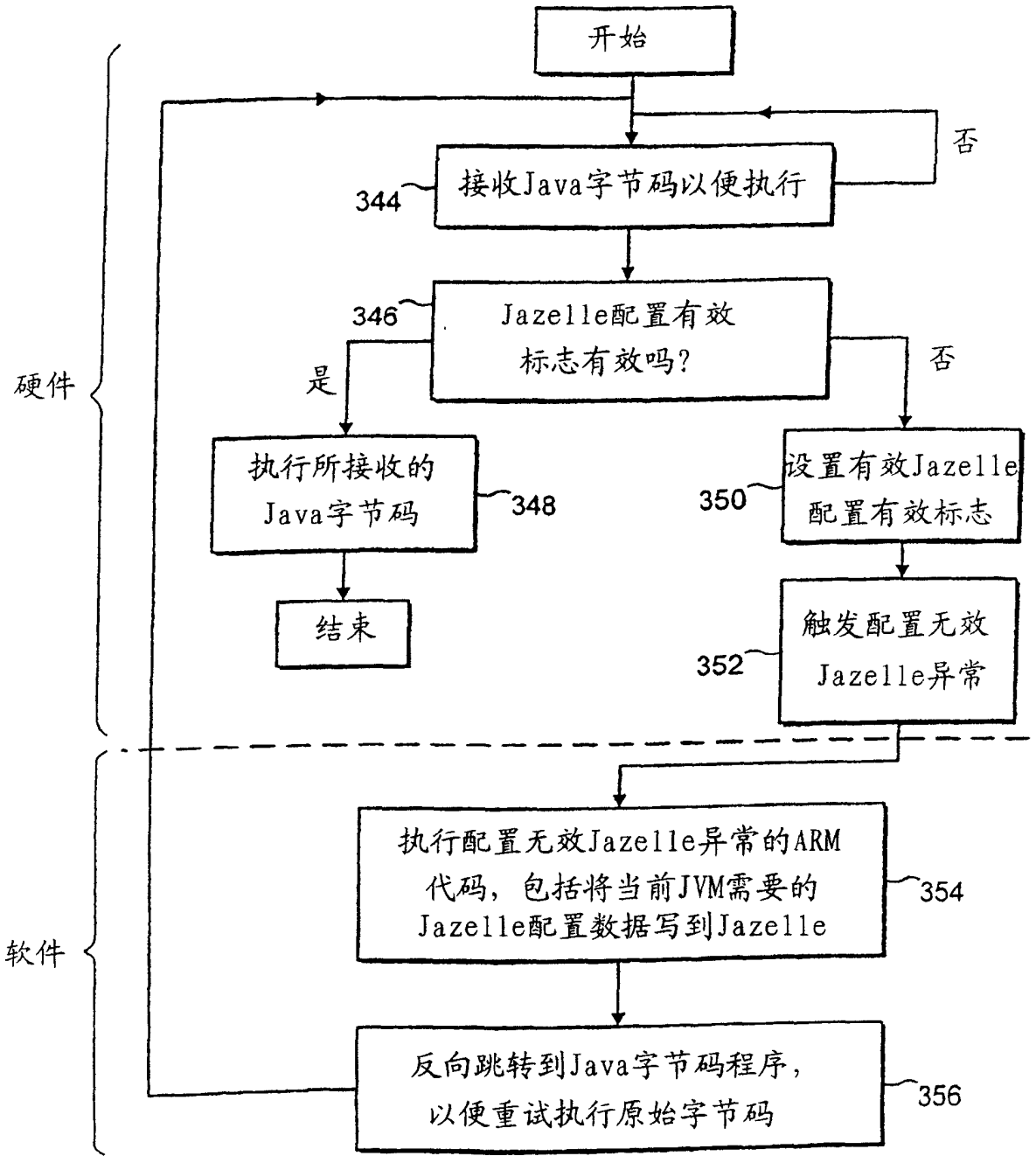

开始
否
344
接收Java字节码以便执行
346
Jazelle配置有效
标志有效吗?
是
否
执行所接收的
Java字节码
348
结束
350
设置有效Jazelle
配置有效标志
触发配置无效
Jazelle异常
352
硬件
软件
执行配置无效Jazelle异常的ARM
代码，包括将当前JVM需要的
Jazelle配置数据写到Jazelle
354
反向跳转到Java字节码程序，
以便重试执行原始字节码
356

图 20

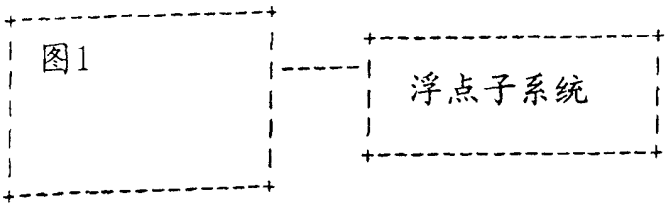


图 21

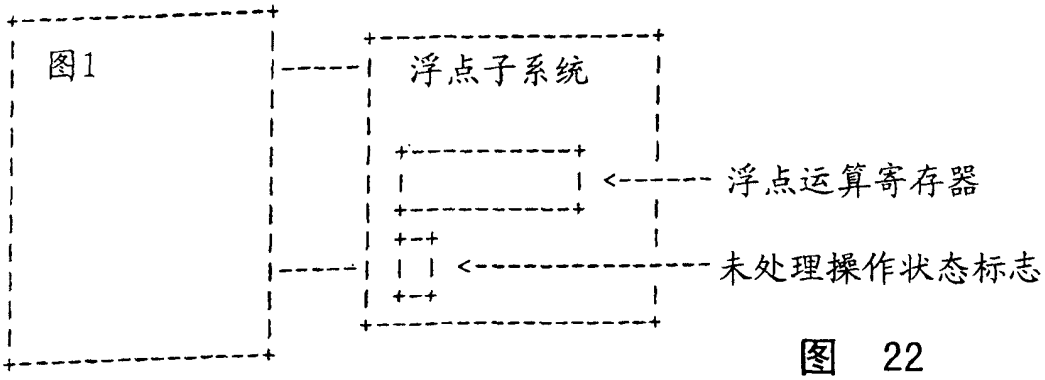


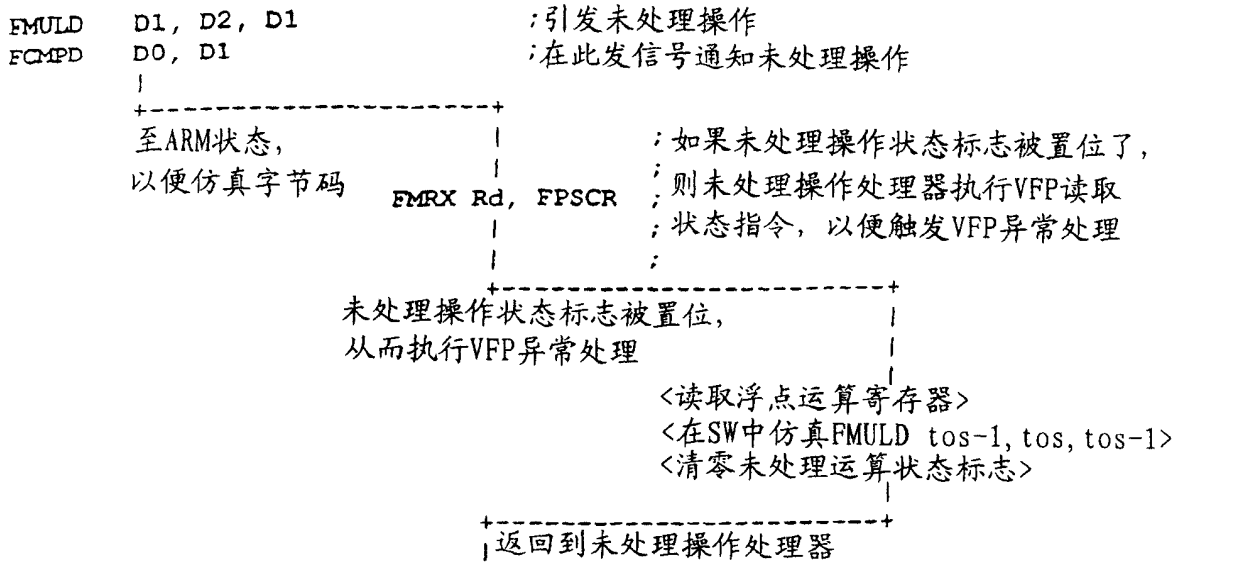
图 22

单精度		双精度	
fadd	FADDS Sd, Sn, Sm	dadd	FADDD Dd, Dn, Dm
fsub	FSUBS Sd, Sn, Sm	dsub	FSUBD Dd, Dn, Dm
fmul	FMULS Sd, Sn, Sm	dmul	FMULD Dd, Dn, Dm
fdiv	FDIVS Sd, Sn, Sm	ddiv	FDIVD Dd, Dn, Dm
frem	Not implemented in HW	drem	不在硬件中执行
fneg	FNEGS Sd, Sm	dneg	FNEGD Dd, Dm
f2d	FCVTDS Dd, Sm	d2f	FCVTSD Sd, Dm
f2i	FTOSIZS Sd, Sm	d2i	FTOSIZD Sd, Dm
f2l	Not implemented in HW	d2l	不在硬件中执行
i2f	FSITOS Sd, Sm	i2d	FSITOD Dd, Sm
l2f	Not implemented in HW	l2d	不在硬件中执行
fcmpl	FCMPS/FMSTAT	dcmpl	FCMPD/FMSTAT
fcmpg	FCMPS/FMSTAT	dcmpg	FCMPD/FMSTAT

图 23

```
dmul
        FMULD   D1, D2, D1
dcmpg
        FCMPD   D0, D1
        FMSTAT
        MVNMI   R0, #0
        MOVEQ   R0, #0
        MOVGT   R0, #1
<下一个Java字节码>
```

图 24



<这时,重试执行EMRX Rd,FPSCR指令,而不触发VFP异常处理>

<清理Java堆栈至存储器>

```
LDRB    R4, [R14]                ;加载触发了未处理操作的字节码
                                 ;
LDR     R12, [Rexc, R4, LSL #2]  ;获取代码片段的地址,以便仿真
                                 ;dcmpg指令。
BX      R12                      ;分支到代码片段.
                                 ;注意:使用BX而不是用BXJ,因为
                                 ;我们不希望用硬件来执行它
```

分支到dcmpg仿真代码

```
LDRB    R4, [R14, #1]!           ;加载下一个Java字节码
                                 ;并更新字节码指针
FLDD    D1, [Rstack, #-8]!       ;从堆栈中弹出第一操作数,
                                 ;1双精度数=2堆栈字。
FLDD    D0, [Rstack, #-8]!       ;从堆栈中弹出第二操作数
                                 ;1双精度数=2堆栈字
LDR     R12, [Rexc, R4, LSL #2]  ;获取下一个字节码的代码
                                 ;片段地址
FCMPD   D0, D1                   ;比较2个双精度数
FMSTAT                           ;读取比较结果,
MVNMI   R0, #0                   ;如果<,则结果=-1,
MOVEQ   R0, #0                   ;如果=,则结果=0,
MOVGT   R0, #1                   ;如果>,则结果=1
STR     R0, [Rstack], #4         ;将结果压入堆栈
BXJ     R12                      ;在硬件/软件中执行
                                 ;下一个字节码
```

图 25

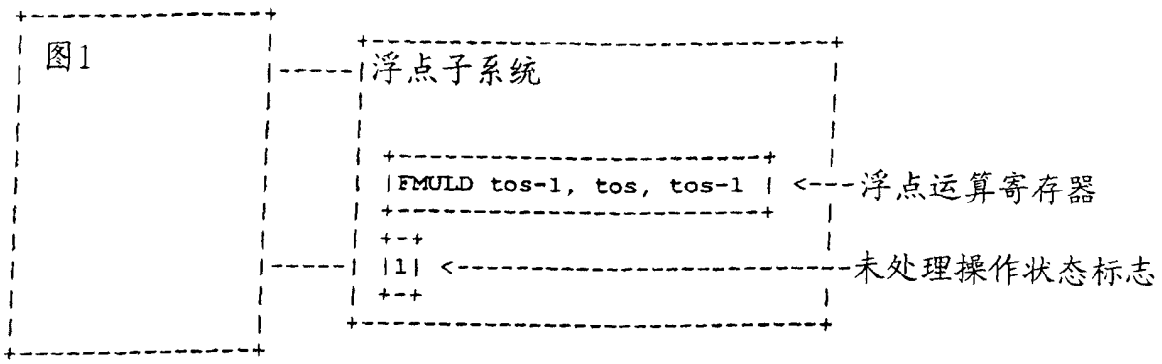


图1
浮点子系统
FMULD tos-1, tos, tos-1
浮点运算寄存器
1
未处理操作状态标志

图 26

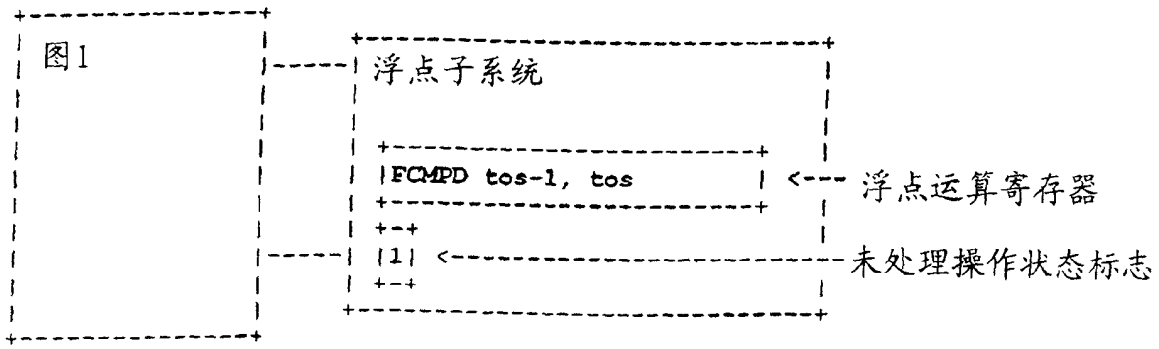


图1
浮点子系统
FCMPD tos-1, tos
浮点运算寄存器
1
未处理操作状态标志

图 28

```
FMULD   D1, D2, D1              ;正常执行
FCMPD   D0, D1                  ;引发未处理操作
FMSTAT                          ;此处发信号通知未处理操作
        |
        +----------------------+
至ARM状态,                     |          ;如果未处理操作状态标志被置位了,
以便仿真字节码          FMRX Rd, FPSCR     ;则未处理操作处理器执行VFP读取
                                |          ;状态指令, 以便触发VFP异常处理
                                |          ;
                                |          ;
                                +----------+-----------------------+
                  未处理操作状态标志被置位,                        |
                  从而执行VFP异常处理                              |
                                                                   |
                                          <读取浮点运算寄存器>      |
                                          <仿真FCMPD tos-1, tos>   |
                                          <清零未处理操作状态标志>  |
                                                                   |
                                +--------------------------+       |
                                | 返回到未处理操作处理器   |<------+
                                +--------------------------+
<此时, 重试执行FMRX Rd, FPSCR指令, 而不触发VFP异常>

<清理Java堆栈至存储器>

LDRB    R4, [R14]               ;加载触发了未处理操作的字节码
                                ;
LDR     R12, [Rexc, R4, LSL #2] ;获取代码片段的地址, 以便仿真
                                ;dcmpg指令。
BX      R12                     ;分支到代码片段.
        |                       ;注意: 使用BX而不使用BXJ, 因为
        |                       ;我们不希望在硬件中执行它
        |                       ;
        V
分支到dcmpg仿真代码
        |
        V

LDRB    R4, [R14, #1]!          ;加载下一个Java字节码
                                ;并更新字节码指针
FLDD    D1, [Rstack, #-8]!      ;从堆栈中弹出第一操作数,
                                ;1双精度数=2堆栈字。
FLDD    D0, [Rstack, #-8]!      ;从堆栈中弹出第二操作数
                                ;1双精度数=2堆栈字
LDR     R12, [Rexc, R4, LSL #2] ;获取下一个字节码的代码
                                ;片段地址
FMULD   D0, D0, D1              ;重试该乘法, 这会碰到与前面
                                ;所精确检测到的相同的问题,
                                ;但是执行ARM指令而不执行字
                                ;节码。VFP异常处理器能毫
                                ;无问题地处理该问题。
                                ;
                                ;
FSTD    D0, [Rstack], #8        ;压入结果, 2个字
BXJ     R12                     ;在硬件/软件中执行
                                ;下一个字节码 (dcmpg)
```

图 27

```
FMULD  D1, D2, D1                ;引发并发信号通知未处理操作
  |
  +-----------------------+
至ARM状态,                |      ;如果未处理操作状态标志被置位了,
以便仿真字节码            |      ;则未处理操作处理器执行VFP读取
                  FMRX Rd, FPSCR ;状态指令, 以便触发VFP异常处理.
                        |        ;因为通过精确地检测未处理操作,
                        |        ;没有未处理操作状态标志, 则将不
                        |        ;会触发VFP异常处理。
                        |        ;
                        |        ;
                        |        ;
                        |        ;
                        |        ;
<清理Java堆栈至存储器>
LDRB    R4, [R14]                ;加载触发了未处理操作的字节码
                                 ;
LDR     R12, [Rexc, R4, LSL #2]  ;获取代码片段的地址, 以便仿真
                                 ;'dmul'指令。
BX      R12                      ;分支到代码片段.
          |                      ;注意:使用BX而不使用BXJ,因为
          |                      ;我们不希望在硬件中执行它
          |                      ;
          V
分支到dmul仿真代码
          |
          V
LDRB    R4, [R14, #1]!           ;加载下一个Java字节码
                                 ;并更新字节码指针
FLDD    D1, [Rstack, #-8]!       ;从堆栈中弹出第一操作数,
                                 ;1双精度数=2堆栈字。
FLDD    D0, [Rstack, #-8]!       ;从堆栈中弹出第二操作数
                                 ;1双精度数=2堆栈字
LDR     R12, [Rexc, R4, LSL #2]  ;获取下一个字节码(dcmpg)
                                 ;的代码片段地址
FMULD   D0, D0, D1               ;重试该乘法, 这会碰到与前面
                                 ;所精确检测到的相同的问题,
                                 ;但是执行ARM指令而不执行字
                                 ;节。VFP异常处理器能毫
                                 ;无问题地处理该问题。
                                 ;
                                 ;
FSTD    D0, [Rstack], #8         ;压入结果, 2个字
BXJ     R12                      ;在硬件/软件中执行
                                 ;下一个字节码(dcmpg)
```

图 29

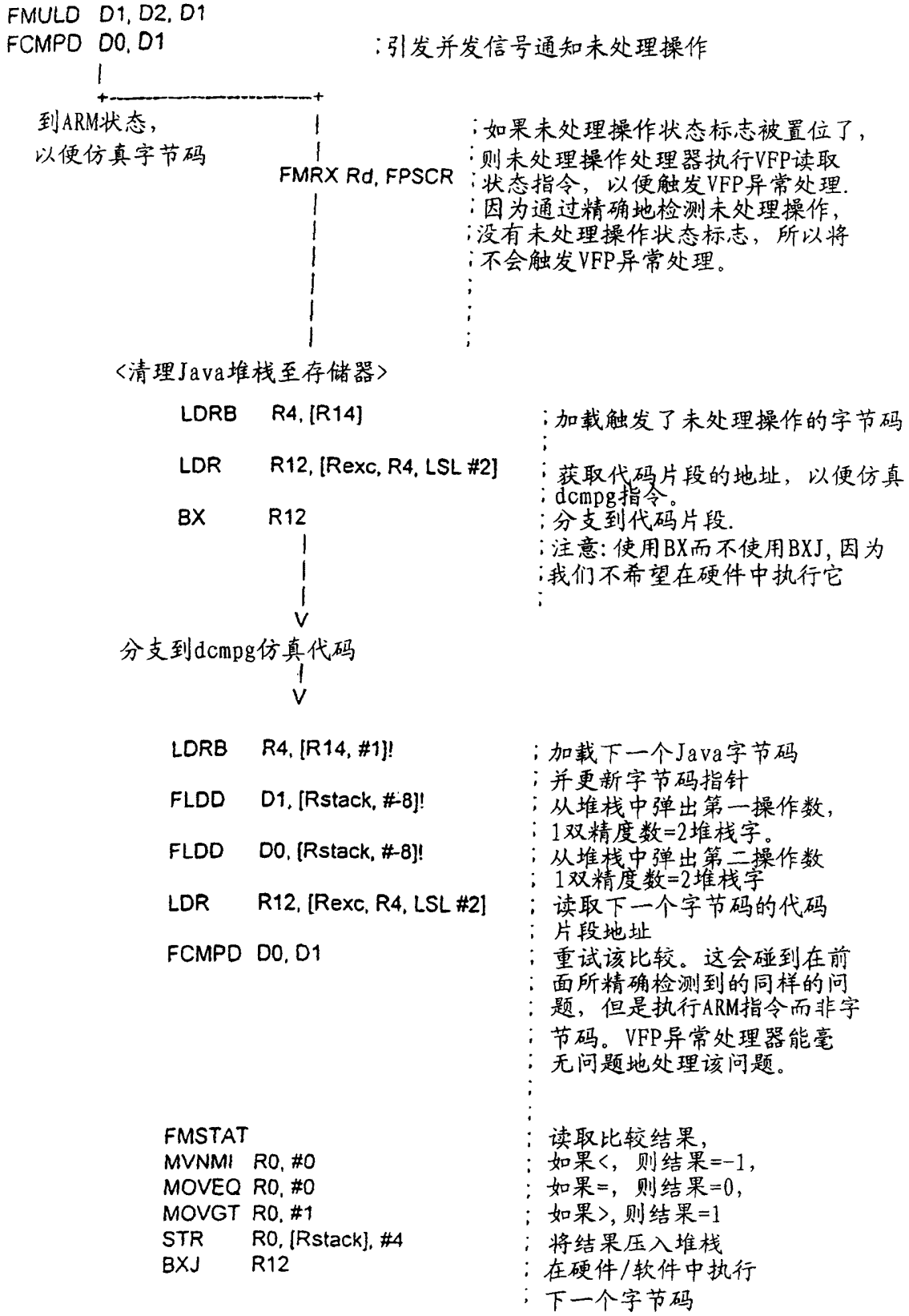

FMULD D1, D2, D1
FCMPD D0, D1
;引发并发信号通知未处理操作
到ARM状态，
以便仿真字节码
FMRX Rd, FPSCR
;如果未处理操作状态标志被置位了，
;则未处理操作处理器执行VFP读取
;状态指令，以便触发VFP异常处理.
;因为通过精确地检测未处理操作，
;没有未处理操作状态标志，所以将
;不会触发VFP异常处理。
<清理Java堆栈至存储器>
LDRB R4, [R14]
;加载触发了未处理操作的字节码
LDR R12, [Rexc, R4, LSL #2]
;获取代码片段的地址，以便仿真
;dcmpg指令。
BX R12
;分支到代码片段.
;注意:使用BX而不使用BXJ,因为
;我们不希望在硬件中执行它
分支到dcmpg仿真代码
LDRB R4, [R14, #1]!
;加载下一个Java字节码
;并更新字节码指针
FLDD D1, [Rstack, #-8]!
;从堆栈中弹出第一操作数，
;1双精度数=2堆栈字。
FLDD D0, [Rstack, #-8]!
;从堆栈中弹出第二操作数
;1双精度数=2堆栈字
LDR R12, [Rexc, R4, LSL #2]
;读取下一个字节码的代码
;片段地址
FCMPD D0, D1
;重试该比较。这会碰到在前
;面所精确检测到的同样的问
;题，但是执行ARM指令而非字
;节码。VFP异常处理器能毫
;无问题地处理该问题。
FMSTAT
;读取比较结果，
MVNMI R0, #0
;如果<，则结果=-1，
MOVEQ R0, #0
;如果=，则结果=0，
MOVGT R0, #1
;如果>,则结果=1
STR R0, [Rstack], #4
;将结果压入堆栈
BXJ R12
;在硬件/软件中执行
;下一个字节码

图 30