(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 107430507 B
(45)授权公告日 2021.08.27

(21)申请号 201680013705.4

(22)申请日 2016.01.21

(65)同一申请的已公布的文献号
申请公布号 CN 107430507 A

(43)申请公布日 2017.12.01

(30)优先权数据
62/110,840 2015.02.02 US
14/727,051 2015.06.01 US

(85)PCT国际申请进入国家阶段日
2017.09.04

(86)PCT国际申请的申请数据
PCT/US2016/014288 2016.01.21

(87)PCT国际申请的公布数据
WO2016/126433 EN 2016.08.11

(73)专利权人 优创半导体科技有限公司
地址 美国纽约州

(72)发明人 M·慕德吉尔 A·J·赫内
P·赫特利

(74)专利代理机构 北京泛华伟业知识产权代理有限公司 11280
代理人 王勇 李科

(51)Int.Cl.
G06F 9/30(2006.01)
G06F 9/38(2006.01)
H03M 7/40(2006.01)

(56)对比文件
US 2005/0028070 A1,2005.02.03
WO 2013/0095637 A1,2013.06.27
CN 105027109 A,2015.11.04

审查员 顾兰

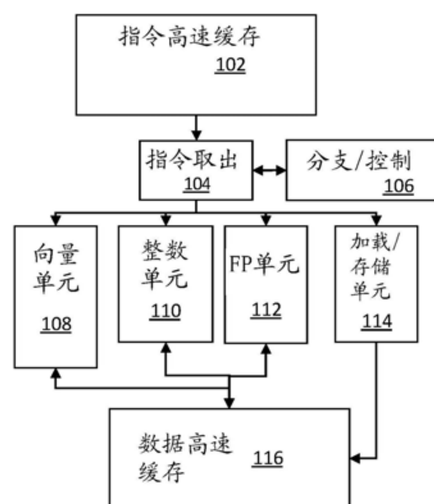
权利要求书5页 说明书47页 附图9页

(54)发明名称

配置成使用改变元素宽度的指令对可变长度向量进行操作的向量处理器

(57)摘要

公开了计算机处理器。计算机处理器可包括向量单元,向量单元包括向量寄存器文件,其包括保存变化数量的元素的至少一个寄存器。计算机处理器还可包括处理逻辑,其配置成使用产生具有与输入元素的宽度不同的宽度的元素的结果的一个或多个指令来对向量寄存器文件中的变化数量的元素进行操作。计算机处理器可被实现为单片集成电路。



CN 107430507 B

1.一种计算机处理器,包括:

向量寄存器文件,其包括用于保存变化数量的元素的至少一个寄存器;

向量长度寄存器文件,其包括第一向量长度寄存器以指定应用于所述至少一个寄存器中保存的元素的操作的数量,其中所述操作的数量独立于能够被填充到硬件实现中的所述至少一个寄存器内的元素的数量;以及

处理逻辑,其配置成执行可变长度向量指令,所述可变长度向量指令包括表示所述至少一个寄存器的第一标识符,和表示所述第一向量长度寄存器的第二标识符,以产生具有与所述变化数量的元素不同的位宽的元素的结果。

2.如权利要求1所述的计算机处理器,其中所述计算机处理器被实现为单片集成电路。

3.如权利要求1或2所述的计算机处理器,其中所述处理逻辑还配置成:

从所述向量寄存器文件读取两个向量寄存器;

将所述两个向量寄存器的内容作为位大小的输入整数元素的两个输入向量;

使第一向量寄存器的元素与第二寄存器的相应元素按元素相乘以产生输出整数元素的输出向量,每个输出整数元素的大小具有两倍于相应的输入整数元素的位的数量的大小;以及

将整数元素的输出向量存储到所述向量寄存器文件中的向量寄存器。

4.如权利要求3所述的计算机处理器,其中每个所述输入整数元素的大小是一个字节,且每个所述输出整数元素的大小是两个字节。

5.如权利要求3所述的计算机处理器,其中每个所述输入整数元素的大小是两个字节,且每个所述输出整数元素的大小是四个字节。

6.如权利要求3所述的计算机处理器,其中每个所述输入整数元素的大小是四个字节,且每个所述输出整数元素的大小是八个字节。

7.如权利要求1所述的计算机处理器,其中所述处理逻辑还配置成:

从所述向量寄存器文件读取向量寄存器;

将所述向量寄存器的内容作为较小位大小的输入整数元素的输入向量;

将输入整数元素的所述输入向量转换成较大位大小的输出整数元素的输出向量;以及

将输出整数元素的所述输出向量存储到所述向量寄存器文件中的向量寄存器。

8.如权利要求7所述的计算机处理器,其中为了从较小位大小转换成较大位大小,所述处理逻辑将零附加到每个所述输入整数元素的较高位。

9.如权利要求7或8所述的计算机处理器,其中从较小位大小转换成较大位大小包括将零附加到每个所述输入整数元素的较低位。

10.如权利要求7所述的计算机处理器,其中为了从较小位大小转换成较大位大小,所述处理逻辑将每个所述输入整数元素的最高位的副本附加到每个所述输入整数元素的较高位。

11.如权利要求7所述的计算机处理器,其中每个所述输入整数元素的大小是一个字节,且每个所述输出整数元素的大小是两个字节。

12.如权利要求7所述的计算机处理器,其中每个所述输入整数元素的大小是两个字节,且每个所述输出整数元素的大小是四个字节。

13.如权利要求7所述的计算机处理器,其中每个所述输入整数元素的大小是四个字

节，且每个所述输出整数元素的大小是八个字节。

14.如权利要求1所述的计算机处理器，其中所述处理逻辑还配置成：

从所述向量寄存器文件读取向量寄存器；

将所述向量寄存器的内容作为较大位大小的输入整数元素的输入向量；

将所述向量寄存器的内容转换成较小位大小的输出整数元素；以及

将较小位大小的输出整数元素的所产生的输出向量存储到所述向量寄存器文件中的向量寄存器。

15.如权利要求14所述的计算机处理器，其中为了从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素，所述处理逻辑丢弃每个所述输入整数元素的高位。

16.如权利要求14或15所述的计算机处理器，其中为了从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素，所述处理逻辑丢弃每个所述输入整数元素的低位。

17.如权利要求14所述的计算机处理器，其中为了从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素，所述处理逻辑还：

表示在所述较小位大小中的整数值；以及

如果原始整数值大于在所述较小位大小处可表示的最大数字，则将所述整数值转换成该可表示的最大数字；或者

如果所述原始整数值小于可表示的最小数字，则将所述整数值转换成该可表示的最小数字。

18.如权利要求14所述的计算机处理器，其中每个所述输入整数元素的位大小是两个字节，且每个所述输出整数元素的位大小是一个字节。

19.如权利要求14所述的计算机处理器，其中每个所述输入整数元素的位大小是四个字节，且每个所述输出整数元素的位大小是两个字节。

20.如权利要求14所述的计算机处理器，其中每个所述输入整数元素的位大小是八个字节，且每个所述输出整数元素的位大小是四个字节。

21.如权利要求1所述的计算机处理器，其中所述处理逻辑还配置成：

从所述向量寄存器文件读取向量寄存器；

将所述向量寄存器的内容作为较小位大小的输入浮点元素的输入向量；

将所述向量寄存器的输入浮点元素转换成较大位大小的输出浮点数的输出向量；以及

将较大位大小的输出浮点元素的所产生的输出向量存储到所述向量寄存器文件中的向量寄存器。

22.如权利要求21所述的计算机处理器，其中每个所述输入浮点元素的大小是两个字节，且每个所述输出浮点元素的大小是四个字节。

23.如权利要求21或22所述的计算机处理器，其中每个所述输入浮点元素的大小是四个字节，且每个所述输出浮点元素的大小是八个字节。

24.如权利要求1所述的计算机处理器，其中所述处理逻辑还配置成：

从所述向量寄存器文件读取向量寄存器；

将所述向量寄存器的内容作为较大位大小的输入浮点元素的输入向量；

将所述向量寄存器的内容转换成较小位大小的浮点元素；以及

将较小位大小的输出浮点元素的所产生的输出向量存储到所述向量寄存器文件中的向量寄存器。

25.如权利要求24所述的计算机处理器，其中每个所述输入浮点元素的位大小是四个字节，且每个所述输出浮点元素的位大小是两个字节。

26.如权利要求24或25所述的计算机处理器，其中每个所述输入浮点元素的位大小是八个字节，且每个所述输出浮点元素的位大小是四个字节。

27.如权利要求24所述的计算机处理器，还包括控制模式、舍入模式或饱和模式中的至少一个。

28.如权利要求27所述的计算机处理器，其中通过所述舍入模式下的舍入或者所述饱和模式下的饱和来从较大浮点数得到较小浮点数的值。

29.如权利要求27所述的计算机处理器，其中按照IEEE标准来定义所述舍入模式和饱和模式。

30.一种用于使用改变元素宽度的指令对可变长度向量进行操作的方法，包括：

由计算机处理器保存变化数量的元素，所述计算机处理器包括向量寄存器文件，所述向量寄存器文件包括至少一个寄存器；

在向量长度寄存器文件的第一向量长度寄存器中，指定应用于所述至少一个寄存器中保存的元素的操作的数量，其中所述操作的数量独立于能够被填充到硬件实现中的所述至少一个寄存器内的元素的数量；以及

由所述计算机处理器的处理逻辑执行可变长度向量指令，所述可变长度向量指令包括表示所述至少一个寄存器的第一标识符，和表示所述第一向量长度寄存器的第二标识符，以产生具有与所述变化数量的元素不同的位宽的元素的结果。

31.如权利要求30所述的方法，其中所述计算机处理器被实现为单片集成电路。

32.如权利要求30或31所述的方法，还包括：

由所述处理逻辑从所述向量寄存器文件读取两个向量寄存器；

将所述两个向量寄存器的内容作为位大小的输入整数元素的两个输入向量；

使第一向量寄存器的元素与第二寄存器的相应元素按元素相乘以产生输出整数元素的输出向量，每个输出整数元素的大小具有两倍于相应的输入整数元素的位的数量的大小；以及

将整数元素的输出向量存储到所述向量寄存器文件中的向量寄存器。

33.如权利要求32所述的方法，其中每个所述输入整数元素的大小是一个字节，且每个所述输出整数元素的大小是两个字节。

34.如权利要求32所述的方法，其中每个所述输入整数元素的大小是两个字节，且每个所述输出整数元素的大小是四个字节。

35.如权利要求32所述的方法，其中每个所述输入整数元素的大小是四个字节，且每个所述输出整数元素的大小是八个字节。

36.如权利要求30所述的方法，还包括：

由所述处理逻辑从所述向量寄存器文件读取向量寄存器；

将所述向量寄存器的内容作为较小位大小的输入整数元素的输入向量；

将输入整数元素的所述输入向量转换成较大位大小的输出整数元素的输出向量；以及

将输出整数元素的所述输出向量存储到所述向量寄存器文件中的向量寄存器。

37.如权利要求36所述的方法,其中从较小位大小转换成较大位大小包括将零附加到每个所述输入整数元素的较高位。

38.如权利要求36或37所述的方法,其中从较小位大小转换成较大位大小包括将零附加到每个所述输入整数元素的较低位。

39.如权利要求36所述的方法,其中从较小位大小转换成较大位大小包括将每个所述输入整数元素的最高位的副本附加到每个所述输入整数元素的较高位。

40.如权利要求36所述的方法,其中每个所述输入整数元素的大小是一个字节,且每个所述输出整数元素的大小是两个字节。

41.如权利要求36所述的方法,其中每个所述输入整数元素的大小是两个字节,且每个所述输出整数元素的大小是四个字节。

42.如权利要求36所述的方法,其中每个所述输入整数元素的大小是四个字节,且每个所述输出整数元素的大小是八个字节。

43.如权利要求30所述的方法,还包括:

由所述处理逻辑从所述向量寄存器文件读取向量寄存器;

将所述向量寄存器的内容作为较大位大小的输入整数元素的输入向量;

将所述向量寄存器的内容转换成较小位大小的输出整数元素;以及

将较小位大小的输出整数元素的所产生的输出向量存储到所述向量寄存器文件中的向量寄存器。

44.如权利要求43所述的方法,其中从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素包括丢弃每个所述输入整数元素的高位。

45.如权利要求43或44所述的方法,其中从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素包括丢弃每个所述输入整数元素的低位。

46.如权利要求43所述的方法,其中从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素包括:

表示在较小位大小中的整数值;

如果原始整数值大于在所述较小位大小处可表示的最大数字,则将所述整数值转换成该可表示的最大数字;或者

如果所述原始整数值小于可表示的最小数字,则将所述整数值转换成该可表示的最小数字。

47.如权利要求43所述的方法,其中每个所述输入整数元素的位大小是两个字节,且每个所述输出整数元素的位大小是一个字节。

48.如权利要求43所述的方法,其中每个所述输入整数元素的位大小是四个字节,且每个所述输出整数元素的位大小是两个字节。

49.如权利要求43所述的方法,其中每个所述输入整数元素的位大小是八个字节,且每个所述输出整数元素的位大小是四个字节。

50.如权利要求30所述的方法,还包括:

由所述处理逻辑从所述向量寄存器文件读取向量寄存器;

将所述向量寄存器的内容作为较小位大小的输入浮点元素的输入向量;

将所述向量寄存器的输入浮点元素转换成较大位大小的输出浮点数的输出向量;以及

将较大位大小的输出浮点元素的所产生的输出向量存储到所述向量寄存器文件中的向量寄存器。

51.如权利要求50所述的方法,其中每个所述输入浮点元素的大小是两个字节,且每个所述输出浮点元素的大小是四个字节。

52.如权利要求50或51所述的方法,其中每个所述输入浮点元素的大小是四个字节,且每个所述输出浮点元素的大小是八个字节。

53.如权利要求30所述的方法,还包括:

由所述处理逻辑从所述向量寄存器文件读取向量寄存器;

将所述向量寄存器的内容作为较大位大小的输入浮点元素的输入向量;

将所述向量寄存器的内容转换成较小位大小的浮点元素;以及

将较小位大小的输出浮点元素的所产生的输出向量存储到所述向量寄存器文件中的向量寄存器。

54.如权利要求53所述的方法,其中每个所述输入浮点元素的位大小是四个字节,且每个所述输出浮点元素的位大小是两个字节。

55.如权利要求53或54所述的方法,其中每个所述输入浮点元素的位大小是八个字节,且每个所述输出浮点元素的位大小是四个字节。

56.如权利要求53所述的方法,其中,所述计算机处理器还包括控制模式、舍入模式或饱和模式中的至少一个。

57.如权利要求56所述的方法,其中通过所述舍入模式下的舍入或者所述饱和模式下的饱和来从较大浮点数得到较小浮点数的值。

58.如权利要求56所述的方法,其中按照IEEE标准来定义所述舍入模式和饱和模式。

配置成使用改变元素宽度的指令对可变长度向量进行操作的向量处理器

[0001] 相关申请的交叉引用

[0002] 本申请要求2015年2月2日提交的美国临时专利申请62/110,840和2015年6月1日提交的美国实用新型专利申请号14/727,051的权益,这两个申请的公开内容通过引用被全部并入本文。

技术领域

[0003] 本公开的实施方式涉及向量处理器,且特别地涉及包括使用改变元素宽度的指令对可变长度向量操作的一个或多个指令的向量处理器的架构和实现。

背景技术

[0004] 向量指令是使用一个指令对一组值进行执行的指令。例如,在x86架构的流式SIMD扩展(SSE)指令ADDPS$xmm0,$xmm1(对封装的单精度浮点值求和)中,两个xmm寄存器每个保存4个单精度浮点值,它们被加到一起并存储在第一寄存器中的。这个行为等效于伪码序列:

[0005] for(i＝0;i<4;i++)

[0006] $xmm0[i]＝$xmm0[i]+$xmm1[i]

[0007] 该组值可来自寄存器、存储器或这两者的组合。保存通常预期由向量指令使用的几组值的寄存器被称为向量寄存器。在一组中的值的数量被称为向量长度。在一些例子中,向量长度也用于描述由向量指令执行的操作的数量。通常,在向量寄存器中的值的数量和在调用向量寄存器的相应向量指令中的操作的数量是相同的,但它们在某些情况中可以是不同的。

[0008] 包括向量指令的指令集架构被称为向量ISA或向量架构。实现向量ISA的处理器被称为向量处理器。

[0009] 向量ISA——其中所有向量指令从存储器读取它们的向量输入并写到存储器而不使用任何向量寄存器——被称为存储器到存储器向量或存储器-向量架构。

[0010] 向量ISA——其中所有向量指令除了加载或存储以外只使用向量寄存器而不访问存储器——被称为寄存器-向量架构。

[0011] 向量指令(例如上面的ADDPS)可隐含地指定固定数量的操作(在ADDPS指令的情况下是四个)。这些被称为固定长度向量指令。固定长度寄存器-向量指令的另一术语是SIMD(单指令多数据)指令。

[0012] 使用定制技术将先前时代的向量处理器实现在多个板上以提高性能。它们中的大部分以常常需要超级计算机的高性能计算机应用(例如天气预报)为目标。然而,技术发展使得单芯片微处理器能够在性能上超过这些多板实现,导致这些向量处理器逐步被淘汰。替代地,超级计算机变成将这些高性能微处理器中的多个组合在一起的多处理器。

[0013] 这些处理器的共同特性是,它们通常不与来自同一公司的较早的型号兼容,因为

指令集根据型号不同而改变。这种情况由这些处理器以问题领域为目标的事实所激发，其中在问题领域提取尽可能多的性能是关键的，并且人们愿意重写应用以这么做。但是，这种情况可能导致机器的实现细节在指令集中被暴露，且当机器实现细节根据型号不同而改变时，指令集可能改变。例如，可被指定的最大向量长度由向量寄存器在每个实现中可保存的元素的最大数量确定。

[0014] 当晶体管的密度继续上升时，向量处理器的第二潮流出现。到20世纪90年代后期时，通用微处理器已经达到通过增加它们可支持的标量功能单元的数量的收益递减点，尽管仍然有可用于支持更多的标量功能单元的芯片区域。同时，存在在这些微处理器上直接支持视频编码和解码的愿望。这两个趋势的汇合导致将各种固定长度向量扩展引入到现有通用架构——例如Intel x86的MMX、IBM PowerPC的Altivec/VMX和DEC Alpha的MVI。

[0015] 这些SIMD型架构使用具有固定字节长度(在MMX的情况下的8字节、对Alt ivec是16字节)的寄存器。寄存器一般被设计成保存可同时被操作的多个较小长度的元素。因此，MMX架构可保存2个4字节整数或4个2字节整数或8个1字节整数。指令PADDD/PADDW/PADDB可将两个寄存器的内容加在一起，将它们视为分别保存2个4字节/4个2字节/8个1字节值。

[0016] 随着技术进步，使向量寄存器保存额外的值变得可能。x86架构的MMX扩展后面是16字节SSE、32字节AVX2和64字节AVX3。在每个点处，额外的指令被引入以执行实质上相同的操作。

[0017] 在通用架构的实现的情况下，由于商业原因，不同的型号能够运行为较老的型号写的代码。因此，x86架构的较新实现可支持多个不同的向量寄存器宽度和对所有这些指令寄存器宽度操作的指令。

发明内容

[0018] 通过提供计算机处理器来解决上述问题并在本领域中实现技术解决方案。该计算机处理器包括向量单元，向量单元包括向量寄存器文件，其包括可保存变化数量的元素的至少一个向量寄存器。计算机处理器还可包括配置成使用产生具有与输入元素的宽度不同的宽度的元素的结果的一个或多个指令来对向量寄存器文件中的变化数量的元素进行操作的处理逻辑。计算机处理器可被实现为单片集成电路。

[0019] 在一个例子中，计算机处理器还可包括向量长度寄存器文件，其包括至少一个寄存器，其中向量长度寄存器文件的至少一个寄存器用于指定处理逻辑操作的元素的数量。

[0020] 在一个例子中，处理逻辑还可配置成：从向量寄存器文件读取两个向量寄存器；将两个向量寄存器的内容作为位大小的输入整数元素的两个输入向量；使第一向量寄存器的元素与第二寄存器的相应元素按元素相乘以产生输出整数元素的输出向量，每个输出整数元素的大小具有两倍于相应的输入整数元素的位的数量的大小；以及将整数元素的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中，每个输入整数元素的大小可以是一个字节，且每个输出整数元素的大小可以是两个字节。在一个例子中，每个输入整数元素的大小可以是两个字节，且每个输出整数元素的大小可以是四个字节。在一个例子中，每个输入整数元素的大小可以是四个字节，且每个输出整数元素的大小可以是八个字节。

[0021] 在一个例子中，处理逻辑还可配置成：从向量寄存器文件读取向量寄存器；将向量寄存器的内容作为较小位大小的输入整数元素的输入向量；将输入整数元素的输入向量转

换成较大位大小的输出整数元素的输出向量;以及将输出整数元素的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中,计算机处理器从较小位大小转换成较大位大小可包括计算机处理器将零附加到每个输入整数元素的较高位。在一个例子中,计算机处理器从较小位大小转换成较大位大小可包括计算机处理器将零附加到每个输入整数元素的较低位。在一个例子中,计算机处理器从较小位大小转换成较大位大小可包括计算机处理器将输入元素的最高位的副本附加到每个输入整数元素的整数元素的高位。

[0022] 在一个例子中,每个输入整数元素的大小可以是一个字节,且每个输出整数元素的大小可以是两个字节。在一个例子中,每个输入整数元素的大小可以是两个字节,且每个输出整数元素的大小可以是四个字节。在一个例子中,每个输入整数元素的大小可以是四个字节,且每个输出整数元素的大小可以是八个字节。

[0023] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件读取向量寄存器;将向量寄存器的内容作为较大位大小的输入整数元素的输入向量;将向量寄存器的内容转换成较小位大小的输出整数元素;以及将较小位大小的输出整数元素的所产生的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中,计算机处理器从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素可包括计算机处理器丢弃每个输入整数元素的高位。在一个例子中,计算机处理器从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素可包括计算机处理器丢弃每个输入整数元素的低位。

[0024] 在一个例子中,计算机处理器从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素可包括计算机处理器表示在较小位大小中的整数值,如果原始整数值大于在较小位大小处可表示的最大数字则将整数值转换成该可表示的最大数字,或者如果原始整数值小于可表示的最小数字则将整数值转换成该可表示的最小数字。

[0025] 在一个例子中,每个输入整数元素的位大小可以是两个字节,且每个输出整数元素的位大小可以是一个字节。在一个例子中,每个输入整数元素的位大小可以是四个字节,且每个输出整数元素的位大小可以是两个字节。在一个例子中,每个输入整数元素的位大小可以是八个字节,且每个输出整数元素的位大小可以是四个字节。

[0026] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件读取向量寄存器;将向量寄存器的内容作为较小位大小的输入浮点元素的输入向量;将向量寄存器的输入浮点元素转换成较大位大小的输出浮点元素的输出向量;以及将较大位大小的输出浮点元素的所产生的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中,每个输入浮点元素的大小可以是两个字节,且每个输出浮点元素的大小可以是四个字节。在一个例子中,每个输入浮点元素的大小可以是四个字节,且每个输出浮点元素的大小可以是八个字节。

[0027] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件读取向量寄存器;将向量寄存器的内容作为较大位大小的输入浮点元素的输入向量;将向量寄存器的内容转换成较小位大小的浮点元素;以及将较小位大小的输出浮点元素的所产生的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中,每个输入浮点元素的位大小可以是四个个字节,且每个输出浮点元素的位大小可以是两个字节。在一个例子中,每个输入浮点元素的位大小可以是八个字节,且每个输出浮点元素的位大小可以是四个字节。

[0028] 在一个例子中,计算机处理器还可配置成控制舍入和饱和模式。在一个例子中,可通过由舍入和饱和模式定义的舍入和饱和从较大浮点数得到较小浮点数的值。在一个例子

中,可按照IEEE标准来定义舍入和饱和模式。

[0029] 通过提供一种方法来解决上述问题并在本领域中实现技术解决方案。该方法可包括:由包括向量寄存器文件的计算机处理器的向量单元保存变化数量的元素,向量寄存器文件包括至少一个寄存器。该方法还可包括:由计算机处理器的处理逻辑使用产生具有与输入元素的宽度不同的宽度的元素的结果的一个或多个指令对在向量寄存器文件中的变化数量的元素进行操作。计算机处理器可被实现为单片集成电路。在一个例子中,计算机处理器还可包括向量长度寄存器文件,其包括至少一个寄存器,其中向量长度寄存器文件的至少一个寄存器用于指定处理逻辑操作的元素的数量。

[0030] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件读取两个向量寄存器;将两个向量寄存器的内容作为位大小的输入整数元素的两个输入向量;使第一向量寄存器的元素与第二寄存器的相应元素按元素相乘以产生输出整数元素的输出向量,每个输出整数元素的大小具有两倍于相应的输入整数元素的位的数量的大小;以及将整数元素的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中,每个输入整数元素的大小可以是一个字节,且每个输出整数元素的大小可以是两个字节。在一个例子中,每个输入整数元素的大小可以是两个字节,且每个输出整数元素的大小可以是四个字节。在一个例子中,每个输入整数元素的大小可以是四个字节,且每个输出整数元素的大小可以是八个字节。

[0031] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件读取向量寄存器;将向量寄存器的内容作为较小位大小的输入整数元素的输入向量;将输入整数元素的输入向量转换成较大位大小的输出整数元素的输出向量;以及将输出整数元素的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中,计算机处理器从较小位大小转换成较大位大小可包括计算机处理器将零附加到每个输入整数元素的较高位。在一个例子中,计算机处理器从较小位大小转换成较大位大小可包括计算机处理器将零附加到每个输入整数元素的较低位。在一个例子中,计算机处理器从较小位大小转换成较大位大小可包括计算机处理器将输入元素的最高位的副本附加到每个输入整数元素的整数元素的较高位。

[0032] 在一个例子中,每个输入整数元素的大小可以是一个字节,且每个输出整数元素的大小可以是两个字节。在一个例子中,每个输入整数元素的大小可以是两个字节,且每个输出整数元素的大小可以是四个字节。在一个例子中,每个输入整数元素的大小可以是四个字节,且每个输出整数元素的大小可以是八个字节。

[0033] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件读取向量寄存器;将向量寄存器的内容作为较大位大小的输入整数元素的输入向量;将向量寄存器的内容转换成较小位大小的输出整数元素;以及将较小位大小的输出整数元素的所产生的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中,计算机处理器从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素可包括计算机处理器丢弃每个输入整数元素的高位。在一个例子中,计算机处理器从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素可包括计算机处理器丢弃每个输入整数元素的低位。

[0034] 在一个例子中,计算机处理器从较大位大小的输入整数元素转换成较小位大小的相应输出整数元素可包括计算机处理器表示在较小位大小中的整数值,如果原始整数值大于在较小位大小处可表示的最大数字则将整数值转换成该可表示的最大数字,或者如果原始整数值小于可表示的最小数字则将整数值转换成该可表示的最小数字。

[0035] 在一个例子中，每个输入整数元素的位大小可以是两个字节，且每个输出整数元素的位大小可以是一个字节。在一个例子中，每个输入整数元素的位大小可以是四个字节，且每个输出整数元素的位大小可以是两个字节。在一个例子中，每个输入整数元素的位大小可以是八个字节，且每个输出整数元素的位大小可以是四个字节。

[0036] 在一个例子中，处理逻辑还可配置成：从向量寄存器文件读取向量寄存器；将向量寄存器的内容作为较小位大小的输入浮点元素的输入向量；将向量寄存器的输入浮点元素转换成较大位大小的输出浮点元素的输出向量；以及将较大位大小的输出浮点元素的所产生的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中，每个输入浮点元素的大小可以是两个字节，且每个输出浮点元素的大小可以是四个字节。在一个例子中，每个输入浮点元素的大小可以是四个字节，且每个输出浮点元素的大小可以是八个字节。

[0037] 在一个例子中，处理逻辑还可配置成：从向量寄存器文件读取向量寄存器；将向量寄存器的内容作为较大位大小的输入浮点元素的输入向量；将向量寄存器的内容转换成较小位大小的浮点元素；以及将较小位大小的输出浮点元素的所产生的输出向量存储到向量寄存器文件中的向量寄存器。在一个例子中，每个输入浮点元素的位大小可以是四个个字节，且每个输出浮点元素的位大小可以是两个字节。在一个例子中，每个输入浮点元素的位大小可以是八个字节，且每个输出浮点元素的位大小可以是四个字节。

[0038] 在一个例子中，计算机处理器还可配置成控制舍入和饱和模式。在一个例子中，可通过由舍入和饱和模式定义的舍入和饱和从较大浮点数得到较小浮点数的值。在一个例子中，可按照IEEE标准来定义舍入和饱和模式。

附图说明

[0039] 从结合下面的附图考虑的在下面提出的示例性实施方式的详细描述中可更容易理解本发明：

[0040] 图1是根据本公开的例子的处理器。

[0041] 图2是根据本公开的例子的向量单元。

[0042] 图3是根据本公开的例子的执行流水线。

[0043] 图4是根据本公开的例子的另一执行流水线。

[0044] 图5是根据本公开的例子的向量执行流水线。

[0045] 图6是根据本公开的例子的非重叠子向量发出。

[0046] 图7是根据本公开的例子的重叠子向量发出。

[0047] 图8是根据本公开的例子的指令发出结构。

[0048] 图9示出了根据本公开的例子的在执行指令序列时的重命名。

[0049] 图10示出了根据本公开的例子的寄存器文件组织。

[0050] 图11是根据本公开的例子的存储体。

[0051] 图12是根据本公开的例子的向量寄存器文件。

具体实施方式

[0052] 本公开的例子包括使用寄存器到寄存器可变长度向量指令的架构。这个架构被设计成允许可执行相同的指令但以不同的速率的变化的实现，允许对未来架构设计和不同的

价格/性能折衷的适应。本公开的例子包括支持数字信号处理和图形处理以及高性能计算的特征。

[0053] 本公开的例子被设计用于在现代无序超标量处理器的环境中的有效实现,所述实现与寄存器重命名和无序执行如何应用于可变长度向量寄存器架构的实现有关。

[0054] 本公开的例子包括适合于在现代通用微处理器的环境中实现的可变长度寄存器向量架构。在一个例子中,该架构可以:

[0055] ●允许指令指定具有比当前可在硬件中实现的值更大的值的向量长度,以及

[0056] ●使向量长度指定将被执行的操作的数量,而与可被填充到特定实现的寄存器内的元素的数量无关。

[0057] 本公开的例子允许后向兼容,即例子允许具有较大寄存器的实现执行具有针对具有较短寄存器的处理器优化的向量长度的指令,同时支持相同的指令集。

[0058] 在一些例子中,架构支持向量指令的精确异常(exception),并在特定的实现中引起不能由硬件直接执行的向量指令的异常。这些特征使得允许前向兼容变得可能。在这种情况下,如果针对比在特定实现上支持的更大的长度的向量优化指令,则本公开的实施方式可在硬件试图执行指令时抑制指令并然后在软件中仿真指令。

[0059] 向量长度与实际实现的分离给支持在不同速率下的操作的执行的各种实现提供了灵活性。通常,老式可变长度向量处理器每周期开始处理向量的一个元素,而SIMD固定长度向量实现一般同时开始处理所有元素。相反,本公开的实施方式允许不同的单元在不同的速率下处理操作。例如,一个或多个实施方式可选择一次开始多达16个ADD操作,使得向量长度38的加法将在三个时钟周期期间开始,在前两个时钟周期中有16个ADD操作,而在最后一个时钟周期中有剩余的六个ADD操作。同一实施方式可选择只实现一个除法单元,使得向量长度38的除法将在38个时钟周期期间开始,其中在每个周期中有一个除操作。

[0060] 老式向量处理器以高性能浮点处理为目标,而现代SIMD式架构是更通用的。本公开的实施方式包括指令,其以利用指令的可变长度性质和可用的较大寄存器大小的方式特别以数字信号处理和图形应用为目标。

[0061] 在一个实施方式中,向量架构可被实现为有序处理器。在另一实施方式中,可使用诸如无序发出和寄存器重命名的技术来实现向量架构以达到更好的性能。本公开示出可变长度向量处理器如何可以适应利用诸如无序发出和寄存器重命名的特征的架构,并利用这些特征来提高性能。

[0062] 向量指令的输入之一可以是指定操作的数量的寄存器。例如,可实现向量ISA,其中向量ADD指令是VADD$n0,$v1,$v2,$v3,具有下列行为:

[0063] for(i＝0;i<$n0;i++)

[0064] $v1[i]＝$v2[i]+$v3[i]

[0065] 没有对这种式样的向量指令的特定术语。原型向量处理器Cray-1使用向量计数寄存器来指定向量指令的长度,且默认地,Cray-1式向量指令与向量长度寄存器相关联。然而,在下面的讨论中,为了明确地与固定长度向量指令区分,从寄存器得到它们的长度的向量指令被称为可变长度向量指令。

[0066] 在一个实施方式中,相同的指令集架构可具有不同的实现。在一个可能的实现中,假设向量单元是在现代无序处理器中的几个单元之一,如在示出根据本公开的一个实施例

的处理器的图1中所示的。

[0067] 如图1所示的处理器可包括：

[0068] ●保存用于执行的指令的指令高速缓存102；

[0069] ●从指令高速缓存102取出指令的指令取出单元104；

[0070] ●基于被取出的指令和各种预测方法来控制从指令高速缓存102取出的指令的控制/分支执行单元106；

[0071] ●包括用于执行整数指令的整数单元110和用于浮点指令的浮点单元112的各种单元；

[0072] ●负责协调数据从存储器到与各种单元相关联的寄存器的移动的加载/存储单元114；

[0073] ●保存数据元素的数据高速缓存116；

[0074] ●向量单元108。

[0075] 向量单元108可包括如图2所示的各种电路块。图2示出根据本公开的一个实施方式的向量单元200。如图2所示的向量单元可包括：

[0076] ●寄存器重命名块202，其将架构式向量单元寄存器重命名为物理寄存器；

[0077] ●无序发出块204，其保存还没有完成的向量指令并负责发送这些指令用于执行。注意，可基于向量长度和那个指令可采用的功能单元的数量来重复地发出向量指令；

[0078] ●包括下列项的各种物理向量寄存器文件214：

[0079] ○向量寄存器文件206，其保存元素的向量；

[0080] ○向量长度寄存器文件208，用于指定由向量指令执行的操作的数量；

[0081] ○向量累加器寄存器文件210，其保存从操作（例如计算向量的总数）产生的标量值；

[0082] ○向量掩膜寄存器文件212，其保存单比特值并用于控制向量操作的执行；

[0083] ●各种功能块（如附图中所示）包括：

[0084] ○2个乘法器216

[0085] ○4个ALU 218；

[0086] ○单个除和/或平方根220；

[0087] ○用于搜索最小/最大值的搜索块222；

[0088] ○用于将向量元素求总和成单个值的归约块224。

[0089] 如图2所示的块的这个组合被选择以用于示意目的，且不是穷尽的。

[0090] 统一（Unity）向量架构

[0091] 统一ISA定义执行通常的控制、整数、浮点和存储器访问指令的标量指令。它还定义操纵指令的向量的指令的子集。在前一部分中所述的分类法中，“统一”是可变长度向量寄存器架构。

[0092] 本部分聚焦于从标量子集分离的统一ISA的向量子集。本公开的实施方式包括可通过改变指令的编码和寄存器的数量来适合于其它标量指令集的指令。

[0093] 在一个实施方式中，形成向量子集的统一ISA寄存器包括：

[0094] ●向量寄存器，其保存由向量指令操作的元素的集合；

[0095] ●向量计数寄存器，其用于指定向量指令的长度和元素计数被使用的其它地方；

[0096] ●向量掩膜寄存器，其保存单个比特的向量，用于在单个元素上控制向量指令的行为；

[0097] ●向量累加器寄存器，其在需要标量值作为向量指令的输入或输出时使用。

[0098] 在一个实施方式中，统一ISA的向量计数寄存器包括16位，且能够指定高达65535的向量长度。65535个双精度浮点数的向量将需要每寄存器几乎512K字节的存储器。这个大小在当前技术中实现是不切实际的。本公开的实施方式允许实现不同大小的向量寄存器的各种实现，适合于目标应用和流行技术的性价比。如果向量计数不能适应所实现的向量大小，则处理器可采用异常并仿真指令。

[0099] 指令

[0100] 一个统一向量指令(例如使两个向量相加的指令)被规定为vadd$n0,$v1,$v2,$v3。这个指令使来自两个源向量寄存器$v2、$v3的$n0个元素相加，并将结果存储在目标向量寄存器$v1中。这个指令的语义等效于下面的伪代码：

[0101] for(i＝0;i<$n0;i++)

[0102] $v1[i]＝$v2[i]+$v3[i]

[0103] 类型划分

[0104] 在一个实施方式中，统一向量ISA的向量的每个元素可隐含地被类型划分为下面的类型之一：

[0105] ●字节(byte):1字节整数

[0106] ●短(short):2字节整数

[0107] ●int:4字节整数

[0108] ●长(long):8字节整数

[0109] ●半(half):16位IEEE浮点

[0110] ●单(single):32位IEEE浮点

[0111] ●双(double):64位IEEE浮点

[0112] 对向量寄存器操作的所有指令指定它们预期作为输入的元素的类型和它们作为输出产生的元素的类型。它们包括向量寄存器加载和存储，以及在向量寄存器之间重新布置数据(而不实际对它们进行操作)的操作。它们还包括将一种类型的向量转换成其它类型的向量的指令。这些指令都指定了输入和输出类型。

[0113] 在汇编语言助记符中以及在本公开的描述中，对浮点元素操作的向量指令被添加有后缀“f”。所以Vadd使整数的向量相加，而vaddf对浮点元素的向量进行操作。整数/浮点元素的大小由另一后缀指示，对于整数为_b/_h/_i/_l以及对于浮点为_h/_s/_d。因此，vadd_b使1字节整数的向量相加，以及vaddf_s使单精度浮点值的向量相加。因为元素大小通常不改变指令的语义(不修改输入和输出的大小和类型)，大小后缀在本公开中通常被省略。

[0114] 如果向量寄存器元素通过第一指令被写为特定的类型并随后被预期某个其它类型的向量寄存器的第二指令读取，则通过第二指令读取的行为是未定义的。

[0115] 统一ISA规定对于每个实现，存在可保存并恢复所有向量寄存器的实现特定机制。

[0116] 对向量元素的类型划分的约束意味着统一ISA的实现可以自由地挑选各种向量元素类型的适当内部表示。例如，32位浮点数可存储在扩展格式中，具有25位带符号尾数、8位

指数以及指示NaN和无限大的位——总共35位。

[0117] 事实上,这些限制允许实现对同一向量寄存器使用不同的存储,取决于向量元素的类型。例如,每个向量寄存器可被映射到两个不同的寄存器阵列之一,一个由使用浮点类型的向量指令使用,而一个针对固定点类型。这将增加向量文件的总大小,但可通过允许整数和浮点向量操作指令的增加的重叠和/或通过允许在寄存器和功能单元之间的较近耦合来提高实现的性能,导致流水线阶段的减小的周期时间和/或数量。

[0118] 对保存和/或恢复所有向量寄存器的实现相关方式的存在的需要从下面的事实产生:在上下文切换期间,保存被换出的执行进程的向量寄存器,且换入将被执行的进程的向量寄存器。然而,上下文转换码不知道最后写到每个寄存器的元素的类型。因此,不可能使用标准指令而不违反它们预期元素具有适当的类型的约束。

[0119] 根据向量寄存器的实现,存在几种可能的可用机制。如果所有向量元素以与在存储器中相同的格式存储在寄存器中,则使用字节移动指令、抑制类型检查(如果有的话)是一个问题。如果基于类型而不同地存储向量元素,则实现可随时跟踪写到在实现特定(和可能非架构式)寄存器中的每个向量寄存器的最后一个类型。上下文转换码可使用那个信息使用适当类型的存储和加载指令来保存并接着随后恢复值。

[0120] 未写入的向量元素

[0121] 如果向量指令将N个元素写到向量寄存器且随后的向量指令从未被写到的那个向量寄存器读取元素,则行为是未定义的。在这种情况下,实现可选择采用异常,或返回未写入的元素的预先定义的值。

[0122] 在所有情况下,当执行读取未写入的值的指令时,行为需要是可再现的。这意味着如果同一指令序列重复地被执行,则它返回相同的结果。这即使在存在介入的中断或上下文切换时也是这样。

[0123] 针对这个需要的动机是使隔离正在处理器上被执行的代码中的故障变得可能。如果程序执行由于某个故障而读取未被到向量的先前写操作写入的向量元素的指令且正被读取的值是不一致的,则程序的执行的结果可根据不同的运行而改变,使确定准确的原因变得很难。

[0124] 浮点舍入模式/异常

[0125] 对标量操作定义的IEEE浮点舍入模式和异常通过将向量处理为一系列浮点操作来扩展到浮点操作的向量,其中每个浮点操作需要与IEEE浮点舍入模式兼容并适当地操纵五个IEEE浮点异常类型。

[0126] 对于五个浮点异常类型(无效、除以零、上溢、下溢、不精确),存在控制位和状态位。控制位用于在遇到异常时抑制或采取中断,而状态位用于记录未抑制的浮点异常的出现。

[0127] 当浮点向量指令执行一系列浮点操作以及该系列操作中的一个遇到将引起五个异常之一的输入操作数时,控制位被检查。如果控制位指示这个异常不使中断被采取,则相应于异常类型的状态位被设置到1。注意,如果在执行该系列操作时多个异常类型被遇到,且所有这些异常类型的控制位使得中断未被采取,则可在单个向量浮点指令的执行期间设置多个状态位。

[0128] 如果遇到异常且控制位被设置为使得异常应引起中断,则中断被采取,好像指令

不执行一样。足够的信息被记录，使得中断处理机可识别异常类型以及使中断被采取的操作在操作序列中的位置。

[0129] 对于完全的IEEE浮点支持，浮点向量指令支持所有4个IEEE舍入模式：-0、-∞、+∞、最近(下偶数优先)。舍入模式由控制寄存器指定，用于指定待应用的浮点舍入模式。当执行向量浮点指令时，基于控制寄存器算术一元指令来对操作序列中的每个的结果进行舍入。

[0130] 具有1个向量寄存器输入的算术向量指令VOP$n0,$v1,$v对从$v2写到$v1的元素执行$n0个一元操作。在伪代码中：

[0131] for(i＝0;i<$n0;i++)

[0132] $v1[i]＝OP$v2[i]

[0133] 基本整数一元指令是：

[0134] ●vabs:计算每个元素的绝对值

[0135] ●vneg:对每个元素取反

[0136] 基本浮点算术一元指令包括：

[0137] ●vabsf:计算每个元素的绝对值

[0138] ●vnegf:对每个元素取反

[0139] ●vrecf:计算每个元素的倒数

[0140] ●vrecef:计算可由软件完善到确切结果的每个元素的倒数的初始估计

[0141] ●vsqrtf:计算每个元素的平方根

[0142] ●vsqrtef:计算可由软件完善到确切结果的每个元素的平方根的初始

[0143] 估计

[0144] 注意，vabs/vabsf和vneg/vnegf指令等效于二元vdiff/vdiffs和vsub/vsubf二元指令(下面所述的)，第一输入包含所有零元素。

[0145] 算术二元指令

[0146] 具有2个向量寄存器输入的算术向量指令VOP$n0,$v1,$v2,$v3执行$n0个二元操作，获取来自$v2的一个输入和来自$v3的另一输入并写到$v1。在伪代码中：

[0147] for(i＝0;i<$n0;i++)

[0148] $v1[i]＝$v2[i]OP$v3[i]

[0149] 基本浮点算术二元指令包括：

[0150] ●vaddf:加上元素

[0151] ●vsubf:减去元素

[0152] ●vdifff:找到元素的绝对差

[0153] ●vmulf:使元素相乘

[0154] ●vminf:两个元素的最小值

[0155] ●vmaxf:两个元素的最大值

[0156] ●vdivf:浮点除

[0157] 基本整数算术二元指令包括：

[0158] ●vadd:加上元素

[0159] ●vsub:减去元素

[0160] ●vdiff:元素的绝对差

[0161] ●vmul:使元素相乘,保持结果的较低部分

[0162] ●vmin:两个元素的最小值,被处理为带符号的整数

[0163] ●vminu:两个元素的最小值,被处理为无符号的整数

[0164] ●vmax:两个元素的最大值,被处理为带符号的整数

[0165] ●vmaxu:两个元素的最大值,被处理为无符号的整数

[0166] ●vand:元素的位的逻辑与

[0167] ●vor:元素的位的逻辑或

[0168] ●vnand:元素的位的逻辑与非

[0169] ●vxor:元素的位的逻辑异或

[0170] ●vshl:将第一元素的值左移第二元素的相关3/4/5/6个较低位,取决于它是否是字节/短/int/长整数类型。

[0171] ●vshr:将第一元素的值右移第二元素的相关较低位,在0s中移动(逻辑右移)

[0172] ●vshra:将第一元素的值右移第二元素的相关较低位,在符号位中移动(算术右移)

[0173] 在一个实施方式中,统一ISA提供移位指令的字节变形。在这些指令中,第一变元是整数类型之一,而第二变元是字节元素的向量。这些指令包括对应于vshl、vshr、vshra的vshbl、vshbr和vshbra。

[0174] N个字节的整数相乘的结果通常可能需要2N个字节来表示。认识到此,实施方式包括向量相乘的几个不同的式样。基本指令vmul使两个N字节元素一起相乘并存储结果的较低N个字节。在一个实施方式中,统一ISA可包括使相乘结果升级到下一较大的整数类型并存储整个结果的向量指令。

[0175] 向量指令可包括:

[0176] ●v2mul:使两个元素相乘,将两个变元处理为带符号的整数。

[0177] ●v2mulus:使两个元素相乘,将第一变元处理为带符号的整数,而将第二变元处理无符号的整数。

[0178] ●v2muluu:使两个元素相乘,将两个元素都处理为无符号的整数。

[0179] 在一些实现中,不对长整数类型定义如上面讨论的这些向量指令。

[0180] 能够处理M个N字节元素的向量并产生M个2N字节元素的单个向量是固定向量架构不能做的事情。因为向量寄存器的大小是固定的,如果它们读取M*N个字节并产生2*M*N个字节,则它们需要:

[0181] ●使用两个单独的指令来处理向量,其中每个指令读取M/2个输入,或

[0182] ●使用写两个单独的向量寄存器的指令,每个寄存器具有M/2个输出。

[0183] 乘法和加法

[0184] 乘和加向量指令可包括三输入指令,例如VOP$n0,$v1,$v2,$v3,$v4指令,其执行$n0个乘和加操作、使来自$v2和$v3的输入相乘并将它与$v4组合、然后写到$v1。在伪代码中,VOP指令代表:

[0185] for(i=0;i<$n0;i++)

[0186] $v1[i]=±($v2[i]*$v3[i])±$v4[i]

[0187] 存在相应于相乘的结果可与第三变元组合的方式的乘和加指令的四个浮点变形。

[0188] ●vmuladdf:将相乘的结果加到第三变元

[0189] ●vmulsubf:将相乘的结果加到第三变元的负数

[0190] ●vnmuladdf:将相乘的结果的负数加到第三变元

[0191] ●vnmulsubf:将相乘的结果的负数加到第三变元的负数

[0192] 基于结果的类型是否与输入的类型相同或它是否是大小的两倍,存在两种类别的整数乘和加。

[0193] 在第一类别的四个指令中,所有三个输入都具有相同的长度。乘法的较低N个字节与第三变元组合,以及结果的N个字节被保存。第一类别的指令包括:

[0194] ●vmuladd:将相乘的结果加到第三变元

[0195] ●vmulsub:将相乘的结果加到第三变元的负数

[0196] ●vnmuladd:将相乘的结果的负数加到第三变元

[0197] ●vnmulsub:将相乘的结果的负数加到第三变元的负数

[0198] 在第二类别中,乘法的2个输入具有大小N,以及第三个输入具有大小2N。前两个变元在一起相乘作为带符号的整数,以及因而产生的2N个字节值被加到2N个字节的第3变元,2N字节的结果被存储。第二类别的指令包括:

[0199] ●v2muladd:将相乘的结果加到第三变元

[0200] ●v2mulsub:将相乘的结果加到第三变元的负数

[0201] ●v2nmuladd:将相乘的结果的负数加到第三变元

[0202] ●v2nmulsub:将相乘的结果的负数加到第三变元的负数

[0203] 归约

[0204] 归约操作将向量寄存器组合到存储在归约寄存器中的标量结果中。可选地,归约寄存器也可被加到结果。

[0205] 第一类别的归约指令包括向量和归约指令。浮点向量和归约指令vsumred0f$n0,$c1,$v2将$v2的$n0个元素加在一起,并将它们存储到累加器寄存器$c1。变形形式vsumredf $n0,$c1,$v2,$c3也将$c3加到结果。在伪代码中:

[0206]
```
sum = $c3 // or 0

for(i = 0; i< $n0; i++ )

    sum += $v2[i]

$c1 = sum
```

[0207] 在与输入类型相同的精度下计算浮点和。

[0208] 相反,指令的整数形式vsumred和vsumred0求结果的总和作为独立于向量$v2的元素的类型的64位数。

[0209] 第二类别的归约指令包括向量乘和指令。浮点乘归约指令是vmulred0f$n0,$c1,$

v2,$v3和vmulred0f$n0,$c1,$v2,$v3,$c4。这些指令的行为类似于和归约的行为，除了在求和之前两个向量寄存器的元素一起相乘以外。在伪代码中：

[0210]
```
sum = $c4 // or 0
for(i = 0; i< $n0; i++ )
    sum += $v2[i] * $v3[i]
$c1 = sum
```

[0211] 如在向量和归约的情况中的，在与向量元素相同的精度下计算浮点和。

[0212] 在乘归约的整数形式vmulred和vmulred0的情况下，两个向量的元素一起相乘作为带符号的整数，且双宽度积在64位中被加在一起。

[0213] 部分归约

[0214] 在一个实施方式中，统一ISA的指令可执行部分归约。这些指令可将向量的子集或者计算的结果的子集组合到具有较少元素的向量中。

[0215] 基本浮点组合指令是vsumnf_xN$n0,$v1,$v2，其中N是整数。这个指令求$v2的N个元素的组的总和，并将$n0/N个结果置于$v2中。这个指令的伪代码是：

[0216]
```
for(i = 0; i< $n0; i += N )
    sum = 0
    for( j = 0; j < N; j++ )
        sum += $v2[i+j]
    $v1[i/N] = sum
```

[0217] 在一个实施方式中，存在整数等效形式(vsumn)和使用为输入尺寸的两倍的结果来对整数值元素求和的等效形式(v2sumn)。

[0218] 在另一实施方式中，统一ISA可包括使两个向量的元素相乘并接着对积的组求和的另一部分归约向量指令。对浮点元素操作的版本是vdotf_xN$n0,$v1,$v2,$v3，其中在指令中的“点(dot)”是“点积”的缩写，因为操作类似于点积的操作。这个指令的伪代码是：

```
for(i = 0; i< $n0; i += N )

    sum = 0

[0219]    for( j = 0; j < N; j++ )

        sum += $v2[i+j]*$v3[i+j]

    $v3[i/N] = sum
```

[0220] 整数和双宽度整数等效形式是vdot和v2dot。

[0221] 复数指令

[0222] 在一个实施方式中,统一ISA的指令可包括对向量元素执行复数乘法的指令,其中向量被作为交替的实数值和虚数值的序列。例如,向量复数浮点乘法指令vxmulf$n0,$v1,$v2,$v3包括由下面的伪代码描述的行为:

```
for(i = 0; i< $n0; i += 2 )

    re2 = $v2[i+0]

    im2 = $v2[i+1]

[0223]
    re3 = $v3[i+0]

    im3 = $v3[i+1]

    $v1[i+0] = re2*re3 - im2*im3;

$v1[i+1] = re2*im3 + re3*im2;
```

[0224] 注意,$n0指定两倍于复数乘法数量的数量。原因是,当对复数乘法执行其它操作时,正常向量操作例如向量加法被使用。这些将实数和虚数ADD计数为单独的ADD。所以,为了加上N/2个复数值的向量,N的计数需要被指定。向量复数乘法指令使用N,使得同一向量计数可用于控制两种类型的操作。如果计数是奇数,向量复数乘法指令的行为是未定义的。

[0225] 使用浮点乘或带符号-带符号的整数乘的所有形式的向量指令具有它们的复数乘法等效形式。除了向量浮点乘法指令以外,这些包括向量整数乘(vxmul)和向量双宽度整数乘(v2xmul)、浮点乘加(vxmuladdf、vxmulsubf、vxnmuladdf、vxnmulsubf)、整数乘加(vxmuladd、vxmulsub、vxnmuladd、vxmulsub)和整数双宽度乘加(v2xmuladd、v2xmulsub、

v2xnmuladd、v2xnmulsub)。

[0226] 还有向量复数和归约及向量复数乘归约指令。这些指令以两个累加器为目标，一个用于实数而一个用于虚数和。例如，vxmulred0f$n0,$c1,$c2,$v3,$v4复数将向量寄存器$v3和$v4的元素对相乘，作为复数处理它们，并接着分别对实数和虚数值求和。实数和被写到$c0，以及虚数和被写到$c1。非零变形vxmulredf$n0,$c1,$c2,$v3,$v4在写回结果之前将$c0和$c1的原始内容加到和。这可由下面的伪代码表示：

```
resum = $c1 // or 0

imsum = $c2 // or 0

for(i = 0; i< $n0; i += 2 )
```

[0227]

```
    re3 = $v3[i+0]

    im3 = $v3[i+1]

    re4 = $v4[i+0]

    im4 = $v4[i+1]
    resum = re3*re4 - im3*im4;

    imsum = re3*im4 + re4*im3;
```

[0228]

```
$c1 = resum

$c2 = imsum
```

[0229] 向量复数和指令例如vxsumred0f$n0,$c1,$c2,$v3计算向量寄存器的交替元素的总和并将它们存储在两个指定的累加器中。存在归约指令的浮点(vxsumredf、vxsumred0f、vxmulredf、vxmulred0f)和整数等效形式(vxsumred、vxsumred0、vxmulred、xvmulred0)。

[0230] 部分归约指令也具有复数等效形式——vxsumnf、vxsumn、v2xsumn、vxdotf、vxdot、v2xdot。

[0231] 饱和

[0232] 在一个实施方式中，在统一ISA中的指令可包括固定点饱和向量指令。这些指令对整数型元素操作。这些固定点饱和指令中的每个可包括对整数型数据操作的等效非饱和向

量指令。通常,固定点饱和指令在超过由结果元素的类型可表示的值的范围的无限精度结果的处理中不同于它的非饱和等效形式。在非饱和操作的情况下,结果被截短。在饱和指令的情况下,结果被饱和到可表示的最正/负的值,取决于结果的符号。
[0233] 适合基本模式的指令是:
[0234] ●vadds:加饱和
[0235] ●vsubs:减饱和
[0236] ●vdiffs:绝对差饱和;注意这可以只饱和到最大的正值
[0237] ●vshls:左移饱和
[0238] ●vshbls:按字节左移饱和
[0239] 固定点乘法将两个输入处理为带符号的。固定点乘法包括额外的特性:两个元素的相乘的结果进一步乘以2。所以整数乘法和固定点乘法的结果可相差为2倍,即使无饱和发生。如果大小为N的固定点乘法的结果应存储回到大小N内,则乘法的较高N个字节被保存,与整数乘法不同,其中较低N个字节被保存。
[0240] 基于乘法的固定点指令包括:
[0241] ●vmuls:固定点乘饱和
[0242] ●vmuladds:固定点乘和加饱和
[0243] ●vmulsubs:固定点乘和减饱和
[0244] ●vnmuladds:使固定点乘和加饱和取反
[0245] ●vnmulsubs:使固定点乘和减饱和取反
[0246] ●v2muls:具有双宽度结果饱和的固定点乘
[0247] ●v2muladds:具有双宽度结果和加饱和的固定点乘
[0248] ●v2mulsubs:具有双宽度结果和减饱和的固定点乘
[0249] ●v2nmuladds:使具有双宽度结果和加饱和的固定点乘取反
[0250] ●v2nmulsubs:使具有双宽度结果和减饱和的固定点乘取反
[0251] 此外,复整数乘变形还包括饱和固定点变形:
[0252] ●vxmuls:固定点复数乘饱和
[0253] ●vxmuladds:固定点复数乘和加饱和
[0254] ●vxmulsubs:固定点复数乘和减饱和
[0255] ●vxnmuladds:使固定点复数乘和加饱和取反
[0256] ●vxnmulsubs:使固定点复数乘和减饱和取反
[0257] ●v2xmuls:具有双宽度结果饱和的固定点复数乘
[0258] ●v2xmuladds:具有双宽度结果和加饱和的固定点复数乘
[0259] ●v2xmulsubs:具有双宽度结果和减饱和的固定点复数乘
[0260] ●v2xnmuladds:使具有双宽度结果和加饱和的固定点复数乘取反
[0261] 整数归约操作也具有它们的固定点等效形式。在这种情况下,求和的结果在被写到累加器寄存器之前被饱和。这些指令是:
[0262] ●vsumred0s:计算元素的总和并减少饱和
[0263] ●vsumreds:用累加器计算元素的总和并减少饱和
[0264] ●vmulred0s:计算元素乘积的总和并减少饱和

[0265] ●vmulreds:用累加器计算元素乘积的总和并减少饱和

[0266] ●vxsumred0s:计算交替元素的总和并减少饱和

[0267] ●vxsumreds:用累加器计算交替元素的总和并减少饱和

[0268] ●vxmulred0s:计算复数元素乘积的总和并减少饱和

[0269] ●vxmulreds:用累加器计算复数元素乘积的总和并减少饱和

[0270] 部分归约操作也包括固定点等效形式。在这种情况下,使用固定点语义来执行乘法,且结果在被写回之前被饱和。

[0271] ●vsumns:计算向量元素的组的总和并使和饱和

[0272] ●v2sumns:计算具有双宽度结果的向量元素的组的总和并使和饱和

[0273] ●vdots:使用固定点语义使元素相乘,计算乘积的总和并使和饱和

[0274] ●v2dots:使用固定点语义使元素相乘,计算具有双宽度结果的乘积的总和并使和饱和

[0275] ●vxsumns:计算交替元素的组的总和并使和饱和

[0276] ●v2xsumns:计算具有双宽度结果的交替元素的组的总和并使和饱和

[0277] ●vxdots:使用复数固定点语义使元素相乘,计算乘积的总和并使和饱和

[0278] ●v2dxots:使用复数固定点语义使元素相乘,计算具有双宽度结果的乘积的总和并使和饱和

[0279] 转换

[0280] 向量的浮点元素可通过使用vunpackf$n0,$v1,$v2指令来转换成下一较高的大小。因此,vunpack_f$n0,$v1,$v2可将在$v2中的$n0个单精度值转换成双精度值,并将结果存储在$v1中。

[0281] 向量的浮点元素可通过使用vpackf$n0,$v1,$v2指令来转换成下一较小的大小。因此,vunpack_f$n0,$v1,$v2可将在$v2中的$n0个单精度值转换成半精度值,并将结果存储在$v1中。

[0282] 向量的整数元素可通过使用vunpack指令来转换成下一较高的大小。这个指令具有几个变形,包括:

[0283] ●符号扩展

[0284] ●零扩展

[0285] ●放置在上半部分中,用零填充右边

[0286] 例如,当拆开字节0xff时,三个选项可分别导致半字0x00ff、0xffff、0xff00。

[0287] 向量的整数元素可通过使用vpack指令来转换成下一较低的大小。这也具有几个变形,包括:

[0288] ●使用下半部分

[0289] ●使下半部分饱和

[0290] ●使用上半部分

[0291] ●固定点打包;这涉及采用上半部分,如果下半部分的最高位被置位则递增1,如果必要则饱和。

[0292] 例如,当将半字0xabf0打包时,四个选项产生0xf0、0x80、0xab、0xac的结果。

[0293] 整数值的向量可使用指令vfcvti/vcvtif转换成等效浮点值的向量或从等效浮点

值的向量转换。这些指令指定整数和浮点元素的类型。因此,vfcvti_w_s将四字节整数转成32位单精度浮点数,而vicvtf_d_w将双精度浮点元素转换成四字节整数。

[0294] 标量/向量移动

[0295] 向量可使用累加器寄存器的内容使用向量广播指令vbrdc$n0,$v1,$c2被初始化到整数值元素并使用浮点等效形式vbrdcf被初始化到浮点元素。这个指令的伪代码是:

[0296] for(i=0;i<$n0;i++)

[0297] $v1[i]=$c2

[0298] 可使用向量附加指令vappendc$n0,$v1,$c2将单个整数元素插入向量内。浮点等效形式是vappendcf。这些指令将元素$c2附加在$v1的位置$n0处,并接着使$n0递增。这个指令的伪代码是:

[0299] $v1[$n0]=$c2

[0300] $n0=$n0+1

[0301] $n0的递增允许重复地插入向量内而不必显式地改变计数寄存器。

[0302] 当使用vappendc/vbrdc指令将整数元素拷贝到向量内时,整数元素被截短。vapppendcs/vbrdcs指令在写入之前使值饱和到指定的整数元素宽度。

[0303] 可从使用veleminc$n0,$c1,$v2和velemdec$n0,$c1,$v2指令将单个整数元素从向量移动到累加器。veleminc指令将在$n0处的$v2中的带符号的整数元素拷贝到累加器$c1,并接着使$n0递增。velemdec指令使$n0递减,并接着将在递减位置处的值从$v2拷贝到$c1内。伪代码是:

[0304] //veleminc

[0305] $c1=$v[$n0]

[0306] $n0++

[0307] //velemdec

[0308] $c1=$v[$n0-1]

[0309] $n0—

[0310] 以这种方式将操作指定为允许在开始部分或末尾处开始将连续的值从向量寄存器移动到累加器。

[0311] 浮点等效形式是velemincf和velemdecf。

[0312] 数据重新布置

[0313] 存在重新布置数据的各种指令,一些使用仅仅一个向量,其它使用两个。这些指令的整数和浮点变形的行为是相同的,除了在被移动的值的大小方面以外。

[0314] 向量交错指令vilv_xN$n0,$v1,$v2,$v3及其浮点等效形式vilvf通过从$v2和$v3交替地选择N个元素来创建$n0个元素的结果向量$v1。$n0可以是2*N的倍数。这些指令的行为由下面的伪代码捕获:

```
for(i = 0; i< $n0; i += N )
```

[0315]
```
    for( j = 0; j < N; j++ )
        $v1[2*i+j] += $v2[i+j]
for( j = 0; j < N; j++ )
```

[0316]
```
    $v1[2*i+N+j] += $v3[i+j]
```

[0317] 本公开的实施方式还提供执行包括vodd_xN$n0,$v1,$v2和veven_xN$n0,$v1,$v2连同它们的浮点等效形式voddf&vevenf的逆操作的向量指令。这些指令提取N个元素的奇数和偶数组。这可被展示在下面的伪代码中：

[0318]
```
// veven
for(i = 0; i< $n0; i+= N )
    for( j = 0; j < N; j++ )
        $v1[i+j] = $v2[2*i+j]
// odd
for(i = 0; i< $n0; i+= N )
    for( j = 0; j < N; j++ )
        $v1[i+j] = $v2[2*i+j+N]
```

[0319] 在一个实施方式中,veven和vodd指令用于分离向量的元素的交替组,并接着使用vilv将它们一起放置回。

[0320] 可使用vtail$n0,$v1,$v2,$n3指令来提取向量的尾部元素。这个指令(及其浮点等效形式vtailf)从向量$v2提取在$n3开始的$n0元素并将它们放置在$v1内。下面的伪代码详细显示操作：

[0321] for(i＝0;i<$n0;i++)

[0322] $v1[i]＝$v2[i+$n3]

[0323] 可使用vconcat$n0,$v1,$v2,$v3,$n4指令及其浮点等效形式vconcatf来链接两

个向量。在这个指令中,$v2的前$n4个元素与$v3的相应元素组合以形成包含被写到$v1的$n0个元素的新向量。下面的伪代码详细显示操作:

[0324]
```
for(i = 0; i< $n4; i++ )

    $v1[i] = $v2[i]

for(i = $n4, j = 0; i< $n0; i++, j++ )

    $v1[i] = $v3[j]
```

[0325] 可使用vswap_xN$n0,$v1,$v2来重新布置在单个向量中的元素的组。这个指令将$n0个元素分成N个元素的组,并接着交换这些组并将它们写到$v1。如果$n0不是2*N的倍数,这个指令的行为是未定义的。伪代码是:

[0326]
```
for(i=0; i<$n0; i += 2*N )

    for( j = 0; j < N; j++ )

            v2[i+j] = v1[i+j+N]

            v2[i+j+N] = v1[i+j]
```

[0327] 掩模集

[0328] 向量掩模寄存器通常可被考虑为单个位的向量。

[0329] 向量掩模寄存器可由各种向量比较指令的结果设置。例如,指令vcmps_cond$n0,$m1,$v2,$v3比较向量的元素,假设元素是适当大小的带符号整数,并接着存储0/1,假设条件是假/真。这个指令的伪代码是:

[0330] for(i=0;i<$n0;i++)

[0331] $m1[i]=($v2[i]cond$v3[i])?1:0;

[0332] 条件cond可以是“等于”、“不等于”、“大于”、“小于”、“大于或等于”和“小于或等于”之一。

[0333] 类似的指令vcmpu比较整数元素的数量作为无符号的值。它包括相同的6个条件。

[0334] 浮点指令vcmpf比较浮点的向量。此外,它接受另外两个条件:检查是否任一输入是NaN,以及检查是否没有一个输入是NaN。

[0335] 实施方式还可包括指令vclassf_class$n0,$m1,$v2,其测试向量$v2的每个元素的浮点类别,并基于它是否是由在指令中的类别字段指定的多个类别的成员来设置相应的$m1。浮点数可以是:

[0336] ●Not-a-Number(NaN):这可以进一步分成安静NaN(qNaN)或发信号NaN(sNaN)

[0337] ●无限大

[0338] ●零

[0339] ●不正常(subnormal)

[0340] ●正常(其是所有其它情况)

[0341] 类别指令修饰符可允许指令也对类别的组合例如正常+零、正常+零+不正常等进行测试。

[0342] 实施方式可包括同时允许符号和类别的测试的两个指令。vclasspsf_class如上检查类别成员,且也检查符号位。只有当指令在被测试的类别中且是正的时,掩模位才被设置到1,除了被测试的类别是NaN的情况以外。对于这个异常情况,只有当NaN是发信号NaN(sNaN)时,条件才为真。vclassnqf_class检查负/qNaN。

[0343] vmop_op$n0,$m1,$m2,$m3使用按位操作op来组合向量掩模寄存器$m2和$m3的$n0个位,并将所产生的位存储回到$m1内。这个指令的伪代码是:

[0344] for(i=0;i<$n0;i++)

[0345] $m1[i]=$m2[i]op$3[i]

[0346] Op是“与”、“与非”、“或”、“或非”、“异或”、“异或非”、“与-补数”或“与非-补数”之一。

[0347] 掩模数据

[0348] 有在掩模位的控制下移动向量寄存器的内容的屏蔽数据操作的几种样式。

[0349] 向量选择指令vself$n0,$v1,$v2,$v3,$m4及其整数配对物vsel查看掩模寄存器的$n0个位的每个,根据位的值从$v2或$v3选择相应的元素,并将它存储在$v1中的那个位置上。它的伪代码是:

[0350] for(i=0;i<$n0;i++)

[0351] $v1[i]=($m4[i]==0)?$v2[i]:$v3[i]

[0352] 向量刺穿指令vpunct_tf$n0,$v1,$v2,$m3——其中tf是0或1之一——跨过向量掩模$m3的$n0个整数元素,且如果它与tf相同,则将$v2的相应元素附加到结果向量$v0。所附加的元素的数量被写回到$n0。浮点等效形式是vpunctf。可使用伪代码来描述它们的行为:

[0353]
```
j = 0
for(i = 0; i< $n0; i++ )
    if( $m3[i] == tf )
        $v1[j] = $v2[i]
        j++
$n0 = j
```

[0354] 另一指令是向量vmix指令vmix$n0,$v1,$v2,$v3,$m4。在vmix指令及其浮点配对物vmixf中,$m4的$n0个位被检查。基于它的值,$v2的下一未读取的值或$v3的下一未使用的值被挑选并加到$v1。行为由下面的伪代码捕获：

[0355]

```
j = 0
k = 0
for(i = 0; i< $n0; i++ )
    if( $m4[i] == 0)
    $v1[i] = $v2[j]
    j++
```

[0356]

```
else
    $v1[i] = $v3[k]
    k++
```

[0357] vpunct和vmix指令的组合可用于使用向量有条件地有效地实现循环。例如,下面的伪代码段示出一个实现：

[0358]

```
for(i = 0; i< N; i++ )
    if( M[i] )
        X[i] = computationF( A[i] )
    else
        X[i] = computationG( A[i] )
```

[0359] 在一个实施方式中,向量穿刺指令用于对真和假情况将输入阵列A的元素分成向量,并接着使用vmix指令来将来自真和假情况的结果组合成输出阵列X。这是相同的,好像代码段被重写为：

[0360]

```
// vpunct_1 等效序列
j = 0
for(i = 0; i< N; i++ )
    if( M[i] )
        A1[j++] = A[j]
N1 = j

// vpunct_0 等效序列
```

```
j = 0
for(i = 0; i< N; i++ )
    if( !M[i] )
        A0[j++] = A[j]
N0 = j

//有希望的，现在这些循环可被向量化
for(i = 0; i< N1; i++ )
    X1[i] = computationF( A1[i] )
for(i = 0; i< N0; i++ )
    X0[i] = computationG( A0[i] )

// vmix 等效序列
j = 0
k = 0
for(i = 0; i< N; i++ )
    if( M[i] )
        X[i] = X1[j++]
    else
        X[i] = X0[k++]
```

[0361]

[0362] 这些类型的变换非常有用的一种情况是在实现发散图像内核时。

[0363] 搜索

[0364] 向量搜索指令vsearchmax$n0,$n1,$c2,$v3和vsearchmin$n0、$n1、$c2、$v3可以

搜索在向量内的$v3中的$n0个元素的最大或最小值元素，并将那个值存储在$c2中且将最大或最小值元素的相应位置存储在$n1中。下面的伪代码是针对vsearchmax：

[0365]
```
max = $v3[0]
pos = 0
for(i = 1; i< $n0; i++ )
    if( $v3[i] > max )
        max = $v3[i]
        pos = i
```

[0366] 实施方式还包括这些指令的无符号和浮点等效形式，包括例如vsearchmaxu、vsearchminu、vsearchmaxf和vsearchminf。

[0367] 数字信号处理

[0368] 上面所述各种指令(例如固定点饱和指令)在DSP(数字信号处理)应用中是有用的。本部分描述主要动机是加速DSP操作的执行的指令。

[0369] 在DSP中的一组公共操作是过滤和关联，包括具有在元素的较长向量内的各种位置的短向量的重复的点积。在典型滤波器的情况下，点积在连续位置处被执行，如在下面的代码段中所示的。

[0370]
```
for(i = 0; i< N; i++ )
    sum = 0;
    for( j = 0; j < TAPS; j++ )
        sum += coeff[j]*in[i+j];
    out[i] = sum;
```

[0371] 统一ISA使用vfilt_xN$n0,$v1,$v2,$v3,$n4指令(及其浮点和饱和固定点等效形式vfiltf和vfilts)来使此加速。在vfilt_xN中，$v3是$n4个元素的向量，其使用在开始部分开始并以大小N的步长前进的$v2的$n4个元素来计算$n0个点积。这个行为由下面的伪代码描述：

[0372]
```
for(i = 0; i< $n0; i++ )
    sum = 0;
    for( j=0; j < $n4; j++ )
        sum += $v3[j] * $v2[N*i+j]
   $v1[i] = sum
```

[0373] 统一ISA加速的另一公共DSP操作是radix-2FFT(快速傅里叶变换)。radix-2FFT包括多个阶段，内核可被表示为如在下面的代码段中：

[0374] for(i＝0;i<2^M;i++)

[0375] X[i]＝A[i]+W[i]*A[i+2^M]；

[0376] X[i+2^N]＝A[i]-W[i]*A[i+2^M]；

[0377] 注意，X、A和W在这个例子中都是复数，且乘法是复数-复数乘法。W被称为旋转因子。

[0378] 通过使用vfftf_xN$n0,$v1,$v2,$v3或通过使用它的整数和固定点饱和等效形式vfft和vffts来使这个内核加速。这些指令将$v2的$n0个元素分成2*N个复数的组，即4*N个元素，其中元素的对代表实部和虚部。使用$v3的前2*N个元素作为旋转因子来在每个组上执行大小log2(N)的radix-2FFT，且结果被写到$v1。这个操作的伪代码是：

[0379]
```
for(i = 0; i< $n0; i += 4*N )
    for( j = 0; j < 2*N; j+= 2 )
        wre,wim = $v3[j], $v3[j+1]
        are,aim = $v2[i+j+2*N], $v2[i+j+2N];
```

[0380]
```
re = wre*are - wim*aim
im = wre*aim + wim*are
$v1[i+j]        = $v1[i+j] + re
$v1[i+j+1]      = $v1[i+j+1] + im
$v1[i+j+2N]     = $v[i+j] - re
$v1[i+j+2N+1] = $v1[i+j+1] - im
```

[0381] 如果$n0不是4N的倍数，指令的行为是未定义的。在vffts中的乘法使用固定点语义来完成。通常，N可以被限制到2的幂。

[0382] 当在上面的例子中2M是1时，W[0]＝1，且乘法是不必要的。vfft2f $n0,$v1,$v2指令及其配对物vfft2和vfft2s可利用这个事实。这个指令的伪代码是：

```
for(i = 0; i< $n0; i += 4 )
    $v1[i+0] = $v2[i+0] + $v2[i+2]
    $v1[i+1] = $v2[i+1] + $v2[i+3]
    $v1[i+2] = $v2[i+0] - $v2[i+2]
    $v1[i+3] = $v2[i+1] - $v2[i+3]
```

[0383]

[0384] 在FFT之后(或之前)，数据可能需要被重新记录在被称为位反转的模式中。在长度M的位反转中，如果我们将值的索引写出，如在位置bM-1bM-2…b1bo处的，则它可移动到位置b0b1…bM-2bM-1，即到它的位被反转并因此名称被反转的索引。vswapbr_M_xN $n0,$v1,$v2指令及其浮点等效形式vswapbrf可用于完成这个变换。它们将$v2的元素分组成大小N的组，然后将组布置到N*2M的簇内。在这些簇内的组基于位反转被交换。如果$n0不是N*2M的倍数，这些指令的行为是未定义的。伪代码是：

[0385]

```
for(i = 0; i< $n0; i += N*2^M )
    for( j = 0; j < 2^M; j++ )
for( k = 0; k < N; k++ )
```

[0386]

```
  $v1[i + j*N + k] = $v2[ i + bitreverse_M(j)*N + k ]
```

[0387] 图形

[0388] 图形代码涉及主要在小4x4矩阵和4-向量上且主要使用单精度浮点的向量和矩阵算术。在前面的章节中讨论了可计算4个向量(vdot_x4)的点积的指令。

[0389] vmulmmf_xN $n0,$v1,$v2,$v3指令假设$v2和$v3是乘在一起的NxN正方形矩阵的集合，且结果存储在$v1中。$n0可以是N2的倍数。伪代码是：

```
for(i = 0; i< $n0; i += N²)
    for( j = 0; j < N; j++ )
        for( k = 0; k < N; k++ )
```

[0390]

```
            sum = 0;
            for( l = 0; l < N; l++ )
                sum += $v2[i+j*N+l] + $v3[i+l*N+k]
        $v1[i+j*N+k] = sum;
```

[0391] vmulmvf_ord_xN$n0,$v1,$v2,$v3指令假设$v2是N个向量的集合，且$v3是NxN矩阵。这个指令使每个向量与矩阵相乘，并将$n0个结果写到$v1内。$n0可以是N的倍数。矩阵的顺序ord可被指定为列主要的或行主要的。矩阵的顺序代表矩阵的元素如何存储在向量$v3中。对于行顺序，伪代码是：

```
for(i = 0; i< $n0; i += N )
    for( j = 0; j < N; j++ )
```

[0392]

```
        sum = 0;
        for( k = 0; k < N; k++ )
            sum += $v2[i+k] + $v3[i+j*N+k]
```

[0393] $v1[i+j] = sum;

[0394] 对于列顺序，伪代码可读取$v3[i+j+N*k]。

[0395] 频繁地出现的另一向量操作是向量标准化。在向量标准化中，每个N-向量被按比例缩放，使得它的长度是1。这使用vnormalf_xN$n0,$v1,$v2指令来完成。这将$v2的$n0个元素分成N的组并除它们。伪代码是：

[0396]
```
for(i = 0; i< $n0; i += N )
    sum = 0;
    for( k = 0; k < N; k++ )
        sum += $v2[i+k] * $v2[i+k]
    scale = 1/√sum
    for( k = 0; k < N; k++ )
        $v1[i+k] += $v2[i+k] * scale
```

[0397] 这可能是计算起来相当昂贵的操作。因为精确精度通常在图形代码中是不需要的，一些实施方式可使用向量标准化近似的指令vnormalaf，其像vnormalf指令一样，除了它产生对实际标准化值的近似以外。

[0398] 在统一ISA中被定义来支持图形的另一操作是将数据重新布置在向量中以实现正方形矩阵的矩阵转置。指令vswaptrf_xN$n0,$v1,$v2将$v2的$n0个元素处理为NxN正方形矩阵的集合。这些矩阵中的每个然后被转置且结果被写到$v1。对此的伪代码是：

[0399]
```
for(i = 0; i< $n0; i += N^2 )
    for( j = 0; j < N; j++ )
        for( k = 0; k < N; k++ )
            $v1[i+j*N+k] = v2[i+k*N+j]
```

[0400] 在这部分中所述的指令的整数等效形式是vmulmm、vmulmv、vnormal、vnormala和vswaptr。

[0401] 向量加载/存储

[0402] 向量使用向量加载和存储从存储器移动到向量寄存器并返回。有指定存储器地址的不同方式。在统一ISA中，基本地址由被表示为$a的地址寄存器提供。当使在这里所述的指令适应于另一架构时，机制——基本地址通过该机制而产生——可能需要改变以适应那个架构的特性。

[0403] 基本加载/存储将存储器的连续块移动到向量寄存器/从向量寄存器移动到存储器的连续块。指令ldv$n0,$v1,$a2和stv$n0,$v1,$a2从在$a2开始的存储器的连续块移动$v1中的$n0个整数元素/将$v1中的$n0个整数元素移动到在$a2开始的存储器的连续块。使用ldvf/stvf指令来移动浮点向量。ldv操作的伪代码是：

[0404] for(i=0;i<$n0;i++)

[0405] $v1[i]=*($a2+i*SIZE)//SIZE=元素的字节数

[0406] 跨过的加载/存储移动存储器的非连续块。在这里，基于在指令ldstv$n0,$v1,$

a2,$n3和ststv$n0,$v1,$a2,$n3中的$n3中指定的步幅来计算每个元素的地址。ststdv指令的伪代码是:

[0407] for(i=0;i<$n0;i++)

[0408] *($a2+i*SIZE*$n3)=$v1[i]

[0409] 带索引的加载/存储ldixv$n0,$v1,$a2,$v3和stixv$n0,$v1,$a2,$v3使用向量$v3的元素来将偏移提供到基本地址$a2内以提供从存储器移动/移动到存储器的$v1的$n0个元素的每个的地址。ldixv指令的伪代码是:

[0410] for(i=0;i<$n0;i++)

[0411] $v1[i]=*($a2+$v3[i]*SIZE)

[0412] 向量偏移元素可具有任何类型。根据统一ISA的实施方式,向量偏移元素被假设具有半字精度。

[0413] 跨过的和带索引的加载/存储可具有浮点等效形式ldstvf、ststvf、ldixvf和stixvf。

[0414] 变形

[0415] 长度寄存器

[0416] 本公开的实施方式可使用显式长度寄存器$n0。每个向量指令可包括显式长度字段。可选地,可使用选定编码方案,其中长度寄存器有时或总是从在指令中的其它信息得到,包括例如:

[0417] ●目标寄存器

[0418] ●任一源寄存器

[0419] ●指令操作码

[0420] 可选地,确切的一个长度寄存器可被选择为使所有指令使用那个长度寄存器。其它计数寄存器可用于处理指令,例如vtail,其中多个计数是需要的。

[0421] 另一可选形式是有活动长度寄存器的概念。在这个模型中,存在指令(比如vactive$n0),其使得向量计数寄存器$n0之一作为对所有随后的向量指令的隐式输入,直到vactive指令的下一次执行为止。注意,这个指令将更多的状态引入处理器内,以跟踪当前活动的计数寄存器。

[0422] 类型划分

[0423] 在已描述的架构中,规定了由所有向量指令读取的和产生的向量的元素的类型。如果元素的输入类型不匹配由向量指令预期的类型,则结果是未定义的。

[0424] 新颖的可选模型依赖于动态地跟踪向量元素的类型;每当向量被写入时,元素的类型被记录。在这个模型中,当向量指令被执行时,每个输入的所记录的类型与预期类型比较。如果它们不是相同的,则使用某组规则将输入隐式地转换成预期输入类型。例如,当假设被执行的指令是vaddf_d(即向量加双精度)时,如果输入之一是单精度浮点元素的向量,则这些单精度浮点元素可在被使用之前转换成双精度。

[0425] 在这个新颖的模型中,转换规则可以是固定的,或可以或许通过使用控制寄存器来动态地被配置。

[0426] 又一可选模型对大部分向量指令免除类型划分。在这个模型中,向量指令通常不指定任何类型。替代地,当它执行时,它检查它的输入的所记录的类型并推断出操作和输出

类型。在这个模型中考虑如果例如vadd指令被执行则发生什么:

[0427] ●它读取它的输入

[0428] ●如果两个输入都是浮点向量,则它将执行浮点加操作;如果它们是整数,则它将执行整数加操作。

[0429] ●输出向量类型被设置为输入的类型。

[0430] 如果输入的类型不是相同的,则基于架构定义,因而产生的行为可以是未定义的,或它可导致值的隐式转换。

[0431] 在这个方法中,一些指令可能仍然需要指定向量的输出类型,包括没有足够的信息来推断出输出类型的那些指令(例如向量加载)和显式地改变向量的类型的那些指令。

[0432] 标量寄存器

[0433] 本公开到目前为止规定了向量单元可能特有的累加器和计数寄存器的使用。但是,可使用其它标量寄存器类型来代替这些寄存器。例如,通用/整数寄存器可用于规定计数,且浮点寄存器可用作浮点归约操作的目标。

[0434] 计数寄存器VS常数

[0435] 在各种指令(例如在vilv中的group counts_xN等)中,常数值被规定为指令的一部分。可选地,可使用计数寄存器。因此,代替vilv_xN$n0,$v1,$v2,$v3,可使用vilv$n0,$v1,$v2,$v3,$n4。当$n4包含值N时,vilv$n0,$v1,$v2,$v3,$n4提供相同的行为。

[0436] 其它

[0437] 所描述的指令可以用与上面所述的不同的方式被编码,包括:

[0438] ●在一些情况下,描述可使得各种字段的范围(例如group counts_xNinvilv,vsumn,vdot)未定义。架构可选择值的变化的子集。

[0439] ●在其它情况例如vpunct中,单个寄存器(例如在vpunct的情况中的$n0)被选择为输入和输出。可选的编码可为了这个目的而使用不同的寄存器,所以代替vpunct_tf$n0,$v1,$v2,$m3,指令可以是vpunct_tf$n0,$n1,$v2,$v3,$m4,$n1是输出。

[0440] ●其它方向也是可能的。例如,虽然使用vmuladd$n0,$v1,$v2,$v3,$v4,$v1是输出以及$v4作为加法器输入,也可使用vmuladd$n0,$v1,$v2,$v3,其中$v1用作加法器输入并用作输出目标。

[0441] ●本公开显式地列举了由指令使用的所有寄存器。例如,在vxmulred$n0,$c1,$c2,$v3,$v4中,显式地指明两个目标累加器寄存器$c1和$c2。在一些实施方式中,可使用寄存器对,其中只有一个寄存器被指明而另一寄存器被隐式地推导出。在另一实施方式中,目标是固定的,使得指令总是写到相同的两个累加器。

[0442] ●在指令——其中不同长度和/或其它计数的向量被涉及——中,向量计数寄存器指定一些值,而其它计数被推导出。可能有编码,其中不同的计数被显式地指定。例如在vsumn_xN$n0,$v1,$v2指令中,$n0指定输入$v2的长度,且输出$v1的长度被推导出为$n0/N。可选地,$n0可用于指定输出$v1的长度,并推导出输入的长度为$n0*N。

[0443] 指令发出

[0444] 这部分涉及与实现可变长度长向量-寄存器架构有关的问题,例如特别与指令发出有关的统一指令集架构,即来自可得到的指令的一个或多个指令的选择和用于执行的选定指令的调度。该问题是在各种实现样式的环境中。考虑的样式的范围从简单的有序、非重

命名、非流水线式实现到更面向性能的无序、重命名、深层流水线式实现。

[0445] 长向量架构的一个定义特性是在向量中的元素的数量，且因此，对那个向量执行的所需的操作的数量超过可执行那些操作的可用功能单元的数量。在每个周期中，只有向量操作的子集可开始被处理。因此，向量指令通常可能需要多个周期来完成。

[0446] 标量指令发出

[0447] 考虑在图3中所示的6阶段处理器执行流水线。处理器流水线的实现可通过组合各种阶段(例如解码和发出)而更简单，或更复杂(例如多个指令取出和执行阶段)。实际流水线依赖于进程技术、期望功率/面积/性能和正被实现的指令集架构。

[0448] 考虑执行两个操作的情况：mul$r0,$r1,$r2和add$r0,$r4,$0。第一指令：

[0449] ●在RF(寄存器取出)阶段中从寄存器文件读寄存器$r1和$r2

[0450] ●在EX(执行)阶段使这些值相乘

[0451] ●在WB(写回)阶段中将结果写回到寄存器$r0

[0452] 第二指令读乘法的结果和寄存器$r4，使这些值相加，并将结果写回到$r0。

[0453] 在朴素处理器实现中，ADD指令不能发出，直到乘法的内容已经被写到寄存器文件为止。这意味着ADD指令不能在乘法之后立即执行。替代地，ADD指令需要等待两个额外的周期。可选地，如果寄存器文件是直写寄存器文件或如果它包括旁路逻辑，则ADD的寄存器文件可与mul的WB重叠，且add持续仅仅一个周期。如果处理器包括绕过执行阶段的结果的逻辑，使得下一指令可使用它，则ADD指令可与mul指令背靠背地发出。

[0454] 指令发出逻辑204负责协调指令的发出，使得它们成功地执行。这包括确保适当的值被读取，如上所示。它也确保更多的指令不试图访问不被支持的硬件资源。一些例子是：

[0455] ●具有1个乘法器和1个加法器的处理器。指令发出逻辑可确保2个加法不被发送用于执行。

[0456] ●具有不是完全流水线且只可以每隔一个周期接受指令的4阶段乘法器的处理器。指令发出逻辑可确保2个乘法不被发出用于背靠背地执行。

[0457] ●具有在寄存器文件上的1个写端口的处理器，但多个功能单元具有变化数量的执行阶段。指令发出逻辑可确保指令被发送用于执行，使得没有两个指令在同一周期中完成。

[0458] 指令发出可通常分成两个类别：有序和无序。在有序指令发出中，指令被发送用于以它们从存储器被取出的顺序执行。如果指令不能被发出，则所有随后的指令被阻止发出，直到那个指令被发出为止。在无序指令发出中，维护指令池。指令从这个池被选取并被发出用于执行。如果指令不能被发出，但随后的指令可被发出，则可能随后的指令首先被执行。

[0459] 在图4中示出处理器流水线的另一组织。这个流水线组织通常存在于无序处理器中。指令从寄存器文件读取它们的输入值，并进入发出结构。发出结构是用于保存指令的存储器的一部分，而发出逻辑是用于从存储在发出结构中的用于发送以执行的指令中选择的逻辑电路。在一个实施方式中，发出结构和发出逻辑都是发出块204的一部分，如图2所示。注意，一些输入值可能在寄存器文件中不是可用的，因为产生它们的指令还没有完成执行或本身在发出结构中等待。发出逻辑检查在发出结构中的指令池，并从所有它们的输入值可用并且在给定可用资源的情况下可被执行的指令中选取用于执行的一个或多个指令。当指令执行完成时，执行结果被写回到寄存器文件。这些结果也被发送到发出结构，其中等待

这些结果的任何指令复制这些结果，且现在可准备发出。

[0460] 可变长度长向量寄存器指令

[0461] 向量寄存器是能够保存多个值(例如多个单精度浮点数或多个字节)的寄存器。这些寄存器可被识别为$vN。向量指令是指定在向量寄存器上的操作的指令，通常将同一操作应用于向量的各个元素。可变长度指令是一个指令，该指令指定可使用寄存器(通常被称为向量长度寄存器并被表示为$nN)来处理的元素的数量。例如，指令vadd_f$n0,$v1,$v2,$v3具有类似于下面的伪代码的行为：

[0462] for(j＝0;j<$n0;j++)

[0463] $v1[j]＝$v2[j]+$v3[j]

[0464] 长向量指令是可花费多个执行周期的指令。例如，如果向量可保持多达64个浮点数且只存在8浮点加法器，则向量指令的执行可能花费至少8个周期来执行最大长度向量。然而，可能$n0指定小得多的向量。在那种情况下，可能使用更少的执行周期(包括0，如果$n0是0)来完成指令。

[0465] 向量指令发出

[0466] 可能以类似于标量指令发出的方式实现向量指令发出，输入向量寄存器连同向量长度寄存器一起被全部读取，且执行单元在多个周期上执行，对输入的子集操作，产生相应的结果，并接着将结果写回到输出寄存器。在这个方法中，存在对可识别子集的状态机或定序器的需要，子集应在每个周期中由执行单元读取和写入。这个定序器可能总是运行最大数量的周期，忽略向量长度，或优选地可基于向量长度寄存器的内容运行多个周期。行为被捕获在用于vadd_f$n0,$v1,$v2,$v3的执行的下面的伪代码中：

[0467]

```
// RF 阶段
for( j = 0; j < VECTOR_LENGTH; j++ )
    in0[j] = $v2[j];
for( j = 0; j < VECTOR_LENGTH; j++ )
    in1[j] = $v3[j];
// multiple EX stages
for( j =0; j < $n0; j += NUM_ADDERS )
    for( k = 0; k < NUM_ADDERS; k++ )
        out[j+k] = in0[j+k] + in1[j+k]
// WB 阶段
for( j = 0; j < VECTOR_LENGTH; j++ )
   $v1[j] = out[j];
```

[0468] 这种方法具有缺点:它要求整个寄存器文件被读取,中间输入和输出流水线寄存器非常宽,因为它们必须匹配寄存器文件的大小,且必须具有非常宽的复用器来选择执行单元可操作于的输入的子集。

[0469] 可选的方法是在执行阶段所需的每个周期中读取并写入向量文件的那些子集。这种方法用下面的伪代码示出:

[0470]

```
for( j = 0; j < $n0; j += NUM_ADDERS )
    // RF 阶段
    for( k = 0; k < NUM_ADDERS; k++ )
        in0[k] = $v2[j+k]
```

[0471]

```
for( k = 0; k < NUM_ADDERS; k++ )
    in1[k] = $v3[j+k];
// EX 阶段
for( k = 0; k < NUM_ADDERS; k++ )
    out[k] = in0[k] + in1[k];
// WB 阶段
for( k = 0; k < NUM_ADDERS; k++ )
    $v3[k] = out[k]
```

[0472] 这个示例性伪代码省略几个细节,包括流水线重叠以及处理$n0不是NUM_ADDERS的倍数的情况,该处理可以用适当的方式实现。

[0473] 这种方法也可包括定序器。在一个实施方式中,定序器可以是独立的。在可选实施方式中,定序器可合并到指令发出逻辑内。当被合并时,指令发出和排序的所有细节被合并到一个块内。

[0474] 用于有效地执行向量指令的输入之一是向量长度寄存器的内容。向量长度寄存器的内容暗示对在发出之前读向量长度寄存器的需要。然而,由于效率原因,期望在发出之后读向量长度寄存器(否则,向量长度寄存器的内容需要被保存在发出结构中)。因此,向量单元的优选流水线组织是与图5所示的组织类似的组织。

[0475] 子向量指令发出

[0476] 如上面提到的,指令不能被发出,直到它的输入值是可用的为止(或将是可用的,经由旁路)。在长向量执行中,在每个执行周期期间,向量指令一次只对向量的一个子集操作。因此,执行不依赖于整个向量可用,而仅仅是子集。对于较高性能实现,指令发出逻辑在子向量级处理向量指令是优选的。

[0477] 考虑向量单元具有两组执行单元——8个乘法器和8个加法器——的情况。假设指令vmul_f$n0,$v0,$v1,$v2和vadd_f$n0,$v3,$v4,$v0需要被执行,以及向量长度是相同的且向量乘的结果由向量加使用。此外,假设两种类型的执行单元花费1个执行周期,以及$n0是32。如果指令发出逻辑在开始vadd之前等待$v0的所有元素被写入,如图6所示,则它可花费12个周期来执行。

[0478] 可选地,如果指令发出只等待$v0的适当子向量,则它可部分地使vadd的执行与如图7所示的vmul重叠,导致9个周期的总时间。

[0479] 如果指令发出逻辑是无序的,则子向量发出可适合于允许指令的子向量被发出用于相对于彼此无序地执行。这可包括属于单个指令的不同子向量的重新排序。

[0480] 在一个实施方式中,向量指令可分成子指令,其中每个子指令处理特定的子向量并将该子指令中的每个插入到发出结构内,以由指令发出逻辑独立地处理。这可影响特别

包括指令完成和撤回逻辑的其余流水线。

[0481] 实现

[0482] 实现基于设计目标，其包括在不同的向量指令之间重叠执行的能力和小程度的无序。特别是，目标应用不需要通过将指令分成子指令而实现的极端性能。替代地，在任一时间只有一个子向量是用于执行的候选项，通常是最小未执行的子向量。

[0483] 在实现中，所有寄存器被重命名，移除了在向量寄存器文件中的写-写冲突的可能性。此外，假设向量寄存器文件具有12个256字节的物理寄存器，这256个字节被组织成8个32字节的子向量。

[0484] 如图8所示的指令发出结构维护以指令取出顺序排序的潜在向量指令的时隙的队列。当指令被添加到发出结构时，它被分配在队列的尾部处的下一可用时隙。流水线的前面的阶段被建立以保证总是有可用时隙。如果没有可用时隙，则指令取出可停止。当指令被添加时，有效位被设置到1，且队列的尾部被递增。At-Count被设置到0。

[0485] 在指令最初被插入指令发出队列内之前，它读向量长度寄存器。如果在那个向量长度寄存器中的值不是有效的，则指令不被认为是准备好用于调度并被标记为等待。每当值被写回到向量长度寄存器时，在使等待位被设置的发出时隙中，将寄存器数量与每个指令中的向量长度寄存器数量进行比较。如果它们匹配，则值被拷贝到那个寄存器的时隙内。

[0486] 当长度变得可用时，剩余计数被设置到所需的执行周期的数量。这通常是长度/8。因此，这是那个指令的剩余的子向量的数量的计数。当剩余计数达到零时，从队列移除指令，且队列被压缩。在将那个子向量的值写到寄存器文件时，向指令完成/撤回逻辑通知那个指令的完成。

[0487] 当指令插入到发出结构内时，相应于每个子向量的位掩模被初始化。在我们的情况中，位向量是96位长。这个位掩模对应于子向量，其需要是可用的，以便使指令针对第一子向量被发出。通常，对于N-输入指令，它使N个位被设置，相应于对每个输入寄存器的第一子向量读取。然而，可能有一些指令，其读取比来自一个变元的子向量大的某个数量的值，并接着针对来自下一变元的子向量重复地应用它们。在那种情况下，可设置对应于第一变元的子向量的多个位和第二变元的第一子向量的1位。

[0488] 发出逻辑维护不可用的子向量的类似位掩模。这与重命名逻辑和寄存器-文件写回逻辑协作来工作。每当物理向量寄存器被分配到指令时，相应于那个向量寄存器的所有8个子向量被标记为不可用的(即被设置到1)。当指令写回到子向量时，相应的位被清零。

[0489] 在每个周期，发出逻辑检查时隙中的所有指令。对于每个时隙，它检查是否：

[0490] ●时隙是有效的

[0491] ●向量长度不是等待的

[0492] ●向量计数不是零

[0493] ●子向量读位掩模以及不可用的子向量位掩模的按位与是全零

[0494] 它选择队列中的满足这些条件的最老的指令(即最接近头部的指令)用于执行。

[0495] 当指令被发送用于执行时，它的计数被递减，且子向量读掩模被调节。通常，这涉及将一些位移动1。

[0496] 当向量计数为零且从队列移除指令时，它的输出向量寄存器的不可用的位都被清零。这照顾了向量长度使得不是所有子向量都被写入的情况。

[0497] 向量寄存器实现

[0498] 这部分考虑与实现可变长度长向量-寄存器架构有关的问题，例如特别是与寄存器文件有关的统一指令集架构。这部分在各种实现样式的环境中检查这些问题。考虑的样式的范围从简单的有序、非重命名、非流水线式实现到更面向性能的无序、重命名、深层流水线式实现。

[0499] 长向量架构的一个定义特性是在向量中的元素的数量以及因此对那个向量执行的所需的操作的数量超过可执行那些操作的可用功能单元的数量。在每个周期中，只有向量操作的子集可开始被处理。因此，向量指令通常可能需要多个周期来完成。

[0500] 约定

[0501] 假设有特定于向量单元的4种寄存器，包括：

[0502] ●被写为$vN的向量寄存器，其包含元素的向量，例如字节或浮点单精度数；

[0503] ●被写为$nN的向量计数寄存器，其包含将被操作的元素的数量的计数；

[0504] ●被写为$mN的向量掩模寄存器，其包含通常被用作向量比较的输出或调节向量指令的按元素的行为的单个位的向量；以及

[0505] ●被写为$cN的向量累加器寄存器，其包含用于保存向量归约操作(例如点积)的标量结果并在其它情况(其中标量输入/输出是向量操作所需要的)中使用的向量。

[0506] 检查在有和没有重命名的情况下寄存器文件的实现。在直接实现(即没有重命名的实现)中，在架构式和物理寄存器之间有一对一映射。在重命名实现中，这两者是不同的。在期望区分开相同类型的架构式寄存器和物理寄存器的环境中，小写体和大写体被使用。因此，在具有重命名的一种实现中，如果向量寄存器编号3映射到物理寄存器11，则我们应分别使用$v3和$V11。直接

[0507] 在缺乏重命名的情况下，有在架构式和物理寄存器之间的1对1映射。因此，指令例如vaddf$n0,$v0,$v1,$v2可以读物理寄存器$V1和$V2并写入物理寄存器$V0。这部分检查在处理直接实现中的异常时产生的问题。

[0508] 不精确异常

[0509] 考虑在向量指令的执行期间的一系列操作之一的执行可引起异常的情况，例如在执行向量浮点除法时当使两个元素相除时除以零，或当在向量浮点加法期间使两个元素相加时的上溢/下溢。

[0510] 一个情景是没有向量输出寄存器的元素可在异常被写入之前被写入。这可想到的是深层流水线操作的情况，例如浮点除法。如果假设20阶段浮点除法流水线(其在第一阶段中检测除以零)、4个除法单元和最大64元素的向量，则当在向量中的最后一个元素的除以零异常被检测到时，向量的第一个元素仍然离完成有几个阶段，所以整个指令可在输出寄存器被修改之前被中止。

[0511] 另一更可能的情景是异常出现在向量指令的一些输出已经被写入但不是全部输出已经被写入的点处的情景。在那种情况下，在异常出现的点处，输出寄存器包括由向量指令产生的新值和老值的混合。这种情况被称为不精确异常，其中在异常的点处的处理器的状态是在指令的执行和它的完成之间的中间状态。可使用任何适当的方法来处理这样的异常，包括用于解决异常并重新开始的方法。

[0512] 当同一寄存器用作输入并用作输出，并且存在即使在输出被重写之后也需要输入

值的可能性时，在处理精确异常时出现一种特殊情况。考虑统一ISA指令vmulmvf$n0,$v1,$v2,$v3。这个指令假设它的第二输入$v3保持N乘N矩阵以及它的第一输入$v2保持多个N-向量。它重复地使在$v2中的每个N-向量与在$v3中的N乘N向量相乘，并将结果写到$v1。现在，如果同一寄存器用于输出和矩阵，如在vmulmvf$no,$v1,$v2,$v1中的，矩阵可被结果重写。

[0513] 一种解决方案是有在开始计算之前将矩阵缓存在内部寄存器中的实现，并使所有矩阵-向量乘法针对矩阵输入使用那个内部缓冲寄存器。这个解决方案的问题是，如果中间计算之一采用异常，此时，$v1的原始值已经被(部分地)重写。为了能够在修正指令之后重新开始，我们必须能够恢复原始矩阵，可想到地通过暴露内部缓冲寄存器。

[0514] 另一解决方案是全局地(使得没有输出寄存器可以与输入相同)或在选定情况下(所以允许vmulmvf的输出与向量输入但不是矩阵输入相同)禁止这样的指令。

[0515] 通常，允许不精确异常具有几个不希望的后果，包括：

[0516] ●非常复杂的异常恢复代码

[0517] ●内部状态的添加以帮助恢复

[0518] ●实现特定内部状态的暴露

[0519] 精确异常

[0520] 相反，在具有精确异常的处理器上，在由指令引起的异常之后进入异常处理机的点处，处理器状态看起来好像指令从未被执行一样。

[0521] 在长向量指令的上下文中，用于实现精确异常的一种方法是使指令将任何结果临时写到中间寄存器，直到所有元素被处理为止，并接着将它们拷贝到输出寄存器。如果异常出现在中间，则拷贝被丢弃，留下具有原始内容的输出寄存器。

[0522] 可选地，我们可将输出寄存器的原始内容拷贝到内部寄存器。如果异常出现，则内部寄存器被拷贝回到内部寄存器。如果异常继续，则内部寄存器内容可被丢弃。将输出寄存器的原始内容拷贝到内部寄存器可在指令的执行之前批量地或当输出寄存器的内容被重写时更懒散地被完成。如果使用第二种方法，则如果异常出现则只有输出寄存器的重写部分从内部寄存器被拷贝回。

[0523] 在向量指令中的不精确异常的一个优点在我们想要在处理异常之后重新完成向量指令的情况下产生。使用精确异常，必须在第一元素处开始并重做任何以前完成的工作。使用不精确异常，在处理异常之后，通常可能在引起异常的元素处开始，不再需要重做所有以前的工作。这可潜在地是大的节省。

[0524] 重命名

[0525] 重命名向量寄存器

[0526] 本公开的实施方式可使寄存器重命名的技术适应于长向量寄存器。在寄存器重命名中，架构式寄存器(即在指令中被命名的寄存器)被映射到物理寄存器。存在比架构式寄存器更多的物理寄存器。在指令被执行之前，它的输入寄存器被映射到物理寄存器，且它的输出寄存器被映射到新的未使用的物理寄存器。物理寄存器可根据已知的技术来再循环以提供自由的物理寄存器。

[0527] 作为例子，假设架构式和物理寄存器具有1-1关系，使得$v0映射到$V0,$v1到$V1等。此外，假设有8个架构式寄存器和12个物理寄存器。图9示出在执行一系列指令之前和之后的映射。首先，执行指令vadd$n0,$v0,$v1,$v2，将架构式寄存器$v1映射到物理寄存器$

V1以及$v2到$V2,并将新寄存器分配到$v0,比如$V8。所以,被执行的指令等效于vadd$n0,$V8,$V1,$V2(忽略对$n0的任何重命名)。接着,执行指令vmul$n0,$v1,$v2,$v0。现在,$v2仍然被映射到$V2,但架构式$v0实际上是物理$V8。$v1的新寄存器被分配,比如$V9。被执行的指令是vmul$n0,$V9,$V2,$V8。最后,执行vsub$n0,$v0,$v1,$v0。在这一点处$v0和$v1被映射到$V8和$V9,以及$v0现在被新分配新寄存器,比如$V10,导致有效指令vsub$n0,$V10,$V9,$V8。

[0528] 异常处理

[0529] 显然,使用重命名,得到精确异常实现起来是容易的。使用重命名,输出寄存器是物理地不同于输入寄存器的未使用的寄存器,即使它具有相同的架构式寄存器名称。当在执行向量指令时出现异常时,可通过将映射回滚到它在异常出现之前的值来得到原始状态。

[0530] 在当前寄存器分配方案中,当异常出现时,新输出寄存器被释放,因为结果不再是有用的。然而在向量寄存器的情况下,部分完成的结果可能是有价值的,其或者在调试中用于帮助诊断异常的原因,或者用于避免必须在异常时重新启动之后重新计算这些部分结果。在一个实施方式中,这些部分结果被保留以将它们暴露于异常处理机制。这可以用几种方式之一完成:

[0531] ●将输出寄存器的内容拷贝到不同的寄存器,可能是专用非重命名寄存器,并且然后继续进行以照常释放寄存器

[0532] ●防止寄存器被立即释放,并提供用于使异常处理代码访问它的内容的手段。完成此的一种方式是有另一架构式寄存器名称,映射逻辑可将该名称映射到这个寄存器。也存在一旦异常处理和恢复逻辑使用它的内容被完成就释放寄存器的机制。

[0533] 名称重用

[0534] 在当前寄存器分配方案中,每个输出寄存器被分配新的未使用的物理寄存器。然而,存在这不是合乎需要的情况。例如,考虑统一ISA指令vappendc$n0,$v1,$c2,其将$c2的内容插到在位置$n0处的向量寄存器$v1内,使得$v1是输入和输出。如果$v1的输出物理寄存器不同于输入物理寄存器,则输入物理寄存器的元素(而不是在$n0处的元素)被拷贝到输出物理寄存器。这可能是相当昂贵的。为了修改在向量中的一个元素,有效地拷贝向量是合乎需要的。

[0535] 本公开的实施方式包括重命名机制,其并不总是创建用于输出寄存器的新映射,但替代地保留用于一些指令的映射。在长向量寄存器实现的上下文中,这对指令是有用的,其中输出寄存器也是输入寄存器,以及只有输入的值的子集被修改。

[0536] 寄存器组

[0537] 分段

[0538] 实现大寄存器向量所需的总存储器将是相当大的。例如,如果有八个物理向量寄存器,每个具有256个字节,则所需存储器是4K字节。然而,假设向量指令是在多个周期期间完成,则不是向量寄存器的所有内容都被立刻需要。

[0539] 在一个实施方式中,将每个寄存器分成段,使得向量指令一般一次处理最多相当于一段的数据。例如,256字节寄存器可以每个被分成8段,每段32字节。单精度浮点加法指令vaddf_s$n0,$v0,$v1,$v2可读取在$v1和$v2的第一段中的8个单精度数,将它们加到一

起，并将结果写回到$v0的第一段。然后第二段等类似地被处理。如果$n0小于最大值(在这种情况下是64)，则读所有段可能是不必要的。例如，如果$n0是8，则只有$v1和$v2的第一段需要被读取和处理。

[0540] 如果功能单元的数量匹配处理完整的段所需的数量，则处理可读取输入的段，并开始功能单元。如果功能单元的数量小于那个数量，则可能需要多个周期来消耗全部段。实现可以：

[0541] ●重新读取段，选择不同的子集，直到整个段被处理为止，和/或

[0542] ●将段缓存在内部寄存器中，并重复地从那个内部寄存器读取，直到它完全被处理为止。

[0543] 如果功能单元的数量超过处理完整的段所需的数量且保持它们忙碌是需要的，则实现可能需要读和写多个段。

[0544] 必须读/写多个段的另一情况是在对来自同一寄存器的非连续元素同时操作的那些向量指令的情况下，其中非连续元素可能来自寄存器的不同段。例如，在统一ISA中，指令vfftf_s_x8$n0,$v1,$v2,$v3指定实现在$v2的8个元素上执行radix-2FFT，其中元素是复数单精度浮点数。在这种情况下，为了执行第一FFT，指令可能需要同时读取字节[0…7]和字节[64…71]。因此，为了执行这个操作，指令需要读取两个32字节段。

[0545] 存储器阵列

[0546] 实现这些寄存器文件的一种方式是使用存储器阵列。存储器阵列可包括多个行，每行包含某个数量的位。当一行被寻址时，那行的位可被读取和写入。在一行中的位的数量被称为阵列的宽度。

[0547] 存储器阵列可允许多个行在同一周期中被读取和/或写入。支持在同一周期中N个同时访问的阵列被称为N端口阵列。一些端口可被限制为进行读取或写入。例如，所谓的“双(dual)端口阵列”可允许在同一周期中至多一行被读取以及一行被写入。所以，“双端口阵列”包括一个读端口和一个写端口。“两(two)端口阵列”相反具有可用于读取或写入的两个端口。因此，“两端口阵列”可在同一周期中进行两个读取或两个写入或读取和写入。当端口的数量增加同时保持行的数量和宽度不变时，阵列的大小增加且性能变低。

[0548] 在存储器阵列的宽度和它的功率/性能之间存在折衷。在某个设计点处，对于特定数量的行，存在可被实现的最大宽度。多个较小的存储器阵列可被组合在一起且并行地被访问以构建看起来更宽的存储器阵列。为了讨论的目的，假设可能直接地或通过组合较小的阵列来构建期望宽度的阵列。

[0549] 组织

[0550] 在一个实施方式中，寄存器文件使用具有每行一段的存储阵列，每行实际上是一段宽。因此，使用具有12个寄存器(其具有4个32字节的段)的寄存器文件，我们可将它实现为具有48行且每行256个位(48x32字节)的存储器阵列。

[0551] 可选的实现技术是使用多个存储器阵列。因此，上面的例子可使用2个存储器阵列来实现，每个存储器阵列包含24行，或者3个具有16行的存储器阵列。由于这些阵列中的每个包含较少的行，因此多个存储器阵列提供更快的速度和更低的功率的优点。

[0552] 读端口

[0553] 如果指令集架构包括可读取三个段并写入1个段的指令，例如向量乘加vmuladd$

n0,$v1,$v2,$v3,则对于完全的性能,单个阵列实现可能需要3-读取、1-写端口寄存器文件。这种向量指令的感兴趣的特性是,指令读取所有三个寄存器的相同段。指令可以首先读取所有三个寄存器的第一段,然后第二段,等等。寄存器的第i段被分布到不同的阵列使得没有多于两个寄存器具有在同一阵列中的同一段的多存储体(multi-bank)实现可将读端口要求削减到二。这可能最低限度地需要阵列的数量是寄存器的数量的一半。

[0554] 本公开的实施方式可组织48个段,如图10所示。符号$vN.I意指向量寄存器N的第I段。每列相应于单独的存储器阵列,且每行相应于在那个阵列内的行。行的数量是段的数量的两倍。

[0555] 如图10所示,在任何列中只有每个段的两个实例。所以,为了读取对vmulred的输入,需要至多2个读端口。这假设所有输入寄存器是不同的。如果指令指定同一寄存器两次(或三次),例如vmulred$n0,$v1,$v9,$v9,$v3,则硬件可识别这种情况只访问两个寄存器(或一个)并根据需要复制它。

[0556] 简单的寄存器文件实现可能要求寄存器文件需要3个读端口的另一情况是当指令需要访问寄存器的多个段以及另一寄存器的某个其它段时(如在上面的vfft例子中的)。

[0557] 再次,多个阵列可用于将读端口要求减小到二。只要寄存器的所有段在不同的阵列中,寄存器的不同段就可在同一周期中被读取,而不增加寄存器端口的数量。这要求存储体(bank)的数量至少与段的数量一样大。注意,在上面的例子中,这个条件被满足。

[0558] 在FFT情况下,将被同时访问的段是二的幂。假设存在八个段且根据FFT的大小,指令可同时访问段I和段I+1、I+2或I+4。只要有至少三个存储体且存储器阵列的数量不是2的幂,就可能布置段,使得该2的幂的访问模式从不需要对同一寄存器的同一存储体的不同行的两个访问。

[0559] 写端口

[0560] 也许可能组合写端口与读端口,并通过以几种方式之一实现寄存器文件来减小在存储器阵列中的端口的总数量。

[0561] 在一个实施方式中,通过停止对所有写的指令执行出现来实现寄存器文件,使得当指令将要写时,它通过当前或任何其它指令来阻止任何寄存器读,并将读延迟一个周期。

[0562] 这种方法的变形是当存储器阵列具有2个读端口和单个读/写端口时。在那种情况下,停止只在试图与写同时发出3输入指令时出现。在那种情况下,3输入指令被延迟。对2输入指令没有影响。

[0563] 本公开的实施方式可组合停止方法与多个存储体。在这种情况下,存储器可具有1个读端口和1个读/写端口。如果指令试图同时从同一存储体读2行,则值被写到那个存储体,指令可被延迟一个周期。

[0564] 也可能通过使寄存器重命名逻辑检查对应于输入的物理寄存器并接着分配与它们没有冲突的物理寄存器来对具有寄存器重命名的实现控制写端口冲突。

[0565] 对于4个段中12个寄存器的例子,假设段/存储体布局如在该例子中的,并假设为了这个例子的目的,所有指令花费一个周期。对于指令例如vmuladd,这意味着3个输入的段1在输出的段0被写入的同时被读取。可以有至多一个阵列,其端口都用于读。如果输出寄存器被分配,使得它的第0个段不在那个端口中,则可能没有冲突。假设指令是vmulred$n0,$v7,$v0,$v1,$v3以及$v0、$v1、$v3被映射到$V0、$V1和$V3。$V0和$V3每个周期从存储器阵

列读取。在第二周期中，它们从存储器1读取。只要寄存器重命名逻辑不将$v0映射到$V6或$V9，在写和读之间就没有冲突。

[0566] 特性

[0567] 类型

[0568] 如果最后写到寄存器的元素的类型不匹配指令预期的类型，则架构可规定指令的行为是未定义的。

[0569] 一个选择是忽略这种情况，并解释寄存器的内容，好像它们是预期类型的元素一样。这个选择可导致较不昂贵的硬件，但使检测在程序中的某些类型的错误变得更难。

[0570] 另一选择是检测这个失配并优选地通过引起异常来采取行动。在这种情况下，类型信息与每个寄存器存储在一起。为了上下文切换的目的，这个信息可以是通过代码可访问的，该代码保存并恢复寄存器状态，使得它也可保存并恢复类型信息。

[0571] 长度

[0572] 如果最后写到寄存器的元素的数量小于将由指令读取的元素的数量，则架构可规定指令的行为是未定义的。

[0573] 一个选择是检测这个失配并优选地通过引起异常来采取正确的行动。在这种情况下，长度信息与每个寄存器存储在一起。为了上下文切换的目的，这个信息可以是通过代码可访问的，该代码保存并恢复寄存器状态，使得它也可保存并恢复长度信息。

[0574] 一种可选的方法是忽略长度失配并使用寄存器的内容。特别是，在由最后一个写入寄存器的指令写入的元素之外的元素的值可取决于由前面的写入寄存器的指令写入的值。在这些值由除了当前程序以外的指令写入的情况下，这些值可能不是可预测的，导致不可接受地可变的结果。

[0575] 另一可选方法是每当寄存器被写入时使指令重写整个寄存器，将不被指令写入的元素设置到默认值。这个默认值可以是NaN或某个其它预定值，或它可以是配置寄存器的内容。

[0576] 可选地，长度与每个寄存器存储在一起(如上所述)，使用所存储的长度来检测在被写入的元素的范围之外进行读取的企图。此时，被返回的值可以是默认值。

[0577] 混合方法是将向量寄存器分成某个数量的区段，并优选地通过使用位掩模来跟踪由指令写入的该数量的区段。当向量指令写到向量寄存器时，被写入的区段的数量被记录。如果最后一个区段仅部分地被写入，则那个区段的尾部元素被设置到默认值。

[0578] 注意，在这种方法中，上下文转换码并不需要保存任何长度寄存器。当读取寄存器时，上下文转换码可试图读整个寄存器。这包括最后写到这个寄存器的元素和可能包含默认值的一些尾部元素。然后，当向量寄存器被恢复时，它的全部可被重写，但尾部元素的同一集合可包含默认值。

[0579] 变形

[0580] 虽然在本公开中讨论了具有至多3个输入和1个输出的示例性指令，但是可以没有限制地将相同的技术应用于具有多个输入和输出的指令。类似地，虽然本公开描述了通过多存储体(multi-banking)将端口要求从3读+1写端口削减到向量寄存器文件中的1读+1读/写端口，但类似的技术可用于在其它环境中实现节省。

[0581] 实现

[0582] 实现是基于设计目标的，该设计目标包括对在不同向量指令之间的重叠执行和小程度的无序的需要，支持精确异常并最小化实现成本。

[0583] 一个实施方式可包括将8个架构式寄存器命名到12个物理寄存器的重命名寄存器文件的实现。物理寄存器是256个字节长，被分成8个子向量，每个32字节。使用6个存储体来实现寄存器，存储体从较小的阵列被构建，使得它们实际上具有大小16x512个字节。

[0584] 在一个实施方式中，不在每个寄存器上保持类型或长度标签。但是，支持返回寄存器文件的未定义部分的默认值。

[0585] 存储体使用四个16x128b阵列来构建，这些阵列具有被共同连接的各种控件以及不同的数据信号。这在图11中示出。这些存储体中的每个包括读/写端口A和只读端口B。端口A的读启用(RENA)、端口A的写启用(WENA)、端口A的行地址(ADRA)、端口B的读启用(RENB)、端口B的写启用(WENB)、端口B的行地址(ADRB)和时钟(CLK)是所有存储器阵列所共有的。端口A的写数据输入(DA)和端口A和B的两个读数据输出(QA,QB)分布在所有存储器阵列上，如图11所示。

[0586] 组合由这些阵列形成的六个存储体，如图12所示。解码逻辑每个周期可接收针对子向量的多达3个读和1个写请求，导致对读和写启用不同的子向量。解码逻辑可将这些请求转换成读和写启用以及在6个存储体中的每个上的2个端口的地址。因此，多达4个存储体可在一个周期内被激活。3个输出中的每个使用512字节宽7到1mux连接到6个存储体，加上默认值，如下所述。

[0587] 用于映射在寄存器R中的子向量N的逻辑是：

[0588]

```
if( R < 6 )
    bank = (R*8+N)%6
    addr = (R*8+N)/6
else
    bank = (R*8+N+1)%6
    addr = (R*8+N)/6 + 8
```

[0589] 此外，实施方式提供了未定义的子向量的默认值。存在用于指示相应的子向量是否具有有效的内容的96个位的阵列。它使用触发器来实现，如果相应的子向量在操作期间被最后写入，触发器被设置到1。它在芯片复位时且当寄存器在重命名期间被首次分配时被清零。当子向量被写入时，它的位被设置。当子向量被读取且它的位是0时，所提供的值从默认值寄存器被读取。这是64位寄存器，其内容可被复制到512个位。

[0590] 实现变形

[0591] 实现可在确切的流水线组织和所选取的技术等的细粒度微选择中不同于彼此。然而，这个部分检查在实现的本质中的较粗的变形，如上面的部分所述的，包括可能的实现变形，例如：

[0592] ●不同的向量寄存器大小。

[0593] ●对不同类型的不同向量寄存器,可能具有不同数量的元素,

[0594] ●功能单元的不同混合,

[0595] ●不同的未实现的向量指令(例如完全在软件中实现的指令),

[0596] ●部分实现(例如一些指令变形在硬件中实现,但其它引起中断并在软

[0597] 件中实现)

[0598] 向量单元较少实现

[0599] 极低成本实现的一个可能性是省略向量单元的可能性,每当向量指令被选择用于执行时使得发生中断。在这种情况下,专门在软件中实现向量单元,且经由仿真来执行所有向量指令。

[0600] 多线程实现

[0601] 存在很多样式的多线程实现,包括轮询调度、超线程、对称多线程等。在向量单元的环境中,多线程的技术适用:

[0602] ●存在用于由处理器支持的每个线程的架构式寄存器文件的拷贝;

[0603] ●在有序多线程处理器中,有用于每个线程的单独寄存器文件;

[0604] ●在具有重命名的多线程处理器中,每个线程可具有它自己的用于重命名的寄存器的池,或可以有公共池,且寄存器从那个池被重命名;

[0605] ●所有线程共享使用用于执行它们的向量指令的相同的功能单元

[0606] 非对称多线程

[0607] 本公开的实施方式包括非对称多线程,这是多线程的版本,其中不是所有线程都同样地访问在处理器中的资源。在具有向量处理器的实现的特定环境中,这将意味着只有一些线程将访问向量单元。

[0608] 在向量单元中,相当大的区域由向量寄存器文件消耗。在多线程实现中,每个额外的线程增加所需的寄存器的数量,使寄存器文件的区域增长。较大的区域增加成本并可影响周期时间。

[0609] 此外,不是所有线程都需要向量单元。例如,在有双向多线程处理器的情况下,其中一个线程处理I/O中断且另一线程执行数据处理,中断处理线程不需要任何向量处理。在这种情况下,只有一个线程访问向量单元就可以了。因此,如果实施方式包括多线程处理器,其中一些线程访问向量单元而其它线程不访问,则它只需要实现足够的向量寄存器状态以满足访问的线程的要求,从而节省存储器区域。例如,如果有3线程处理器,其中只有1个线程被允许访问向量单元,则实现只需要有用于仅仅1个线程的向量寄存器,而不是3个线程,导致大的节省。如果结果是一次只一个程序需要来使用向量单元,则这个节省不会导致性能损失。

[0610] 静态VS动态非对称多线程

[0611] 在静态多线程的情况下,某些硬件线程访问向量单元,而其它线程不访问。硬件线程的分配由硬件固定。如果在不能访问向量单元的线程上运行的程序想要开始执行向量指令,则它被从处理器中换出,并接着被换回到能够进行向量访问的线程上。

[0612] 在动态多线程的情况下,处理器可配置成允许针对向量单元的不同线程,使得在任一时间,只有所有线程的子集能够访问向量单元,但那个子集可被改变。在这种特定的情

况下，如果在不能访问向量单元的线程上运行的程序想要开始执行向量指令，则硬件可被重新配置以允许那个线程访问向量单元。

[0613] 在动态多线程的情况下，程序可被分配以直接访问向量单元，如果没有其它程序正在使用向量单元的话，或者，可使它等待，直到当正在使用向量单元的程序释放该向量单元时其变得可用为止，或某个其它线程可被强制释放其向量单元。

[0614] 通常，当程序释放向量单元时，这类似于在上下文切换期间被换出。由程序使用的向量寄存器通常由操作系统保存在那个程序所特有的区域中。当程序获取向量单元时，这类似于在上下文切换期间被换入。向量寄存器通常由操作系统从那个程序所特有的区域加载。

[0615] 一般化

[0616] 本部分描述应用于向量单元的非对称多线程，其中只有线程的子集被允许同时访问向量单元，从而允许实现只对那些线程保持向量状态，与对所有线程保持向量状态不同。考虑到向量寄存器的大小和向量寄存器文件的区域，这导致相当大的节省。

[0617] 然而，这个想法具有更一般的可应用性。例如在任何多线程处理器中，可能允许线程非对称地访问处理器资源，并从而节省使所有线程能够访问所有资源的成本。例如在用于嵌入式处理器的多线程实现中，对浮点单元进行相同的事情——比如有4个线程，但只有一个访问浮点单元且因此只有一组浮点寄存器——可能是有意义的。

[0618] 在一个例子中，提供向量(计算机)处理器100。计算机处理器可包括向量单元108、200，其包括向量寄存器文件206，该向量寄存器文件206包括可保存变化数量的元素的至少一个向量寄存器。计算机处理器100还可包括配置成使用产生具有与输入元素的宽度不同的宽度的元素的结果的一个或多个指令来对向量寄存器文件206中的变化数量的元素进行操作的处理逻辑。计算机处理器100可被实现为单片集成电路。

[0619] 在一个例子中，计算机处理器100还可包括向量长度寄存器文件208，其包括至少一个寄存器，其中向量长度寄存器文件208的至少一个寄存器可用于指定处理逻辑操作的元素的数量。

[0620] 在一个例子中，处理逻辑还可配置成：从向量寄存器文件206读取两个向量寄存器；将两个向量寄存器的内容作为位大小的输入整数元素的两个输入向量；使第一向量寄存器的元素与第二寄存器的相应元素按元素相乘以产生输出整数元素的输出向量，每个输出整数元素的大小具有两倍于相应的输入整数元素的位的数量的大小；以及将整数元素的输出向量存储到向量寄存器文件206中的向量寄存器。在一个例子中，每个输入整数元素的大小可以是一个字节，且每个输出整数元素的大小可以是两个字节。在一个例子中，每个输入整数元素的大小可以是两个字节，且每个输出整数元素的大小可以是四个字节。在一个例子中，每个输入整数元素的大小可以是四个字节，且每个输出整数元素的大小可以是八个字节。

[0621] 在一个例子中，处理逻辑还可配置成：从向量寄存器文件206读取向量寄存器；将向量寄存器的内容作为较小位大小的输入整数元素的输入向量；将输入整数元素的输入向量转换成较大位大小的输出整数元素的输出向量；以及将输出整数元素的输出向量存储到向量寄存器文件206中的向量寄存器。在一个例子中，计算机处理器100从较小位大小转换到较大位大小可包括计算机处理器将零附加到每个输入整数元素的较高位。在一个例子

中,计算机处理器100从较小位大小转换到较大位大小可包括计算机处理器将零附加到每个输入整数元素的较低位。在一个例子中,计算机处理器100从较小位大小转换到较大位大小可包括计算机处理器100将输入元素的最高位的副本附加到每个输入整数元素的整数元素的较高位。

[0622] 在一个例子中,每个输入整数元素的大小可以是一个字节,且每个输出整数元素的大小可以是两个字节。在一个例子中,每个输入整数元素的大小可以是两个字节,且每个输出整数元素的大小可以是四个字节。在一个例子中,每个输入整数元素的大小可以是四个字节,且每个输出整数元素的大小可以是八个字节。

[0623] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件206读取向量寄存器;将向量寄存器的内容作为较大位大小的输入整数元素的输入向量;将向量寄存器的内容转换成较小位大小的输出整数元素;以及将较小位大小的输出整数元素的所产生的输出向量存储到向量寄存器文件206中的向量寄存器。在一个例子中,计算机处理器100从较大位大小的输入整数元素转换到较小位大小的相应输出整数元素可包括计算机处理器100丢弃每个输入整数元素的高位。在一个例子中,计算机处理器100从较大位大小的输入整数元素转换到较小位大小的相应输出整数元素可包括计算机处理器100丢弃每个输入整数元素的低位。

[0624] 在一个例子中,计算机处理器100从较大位大小的输入整数元素转换到较小位大小的相应输出整数元素可包括计算机处理器100表示在较小位大小中的整数值,如果原始整数值大于在较小位大小处可表示的最大数字,则将整数值转换成该可表示的最大数字,或者如果原始整数值小于可表示的最小数字,则将整数值转换成可表示的最小数字。

[0625] 在一个例子中,每个输入整数元素的位大小可以是两个字节,且每个输出整数元素的位大小可以是一个字节。在一个例子中,每个输入整数元素的位大小可以是四个字节,且每个输出整数元素的位大小可以是两个字节。在一个例子中,每个输入整数元素的位大小可以是八个字节,且每个输出整数元素的位大小可以是四个字节。

[0626] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件206读取向量寄存器;将向量寄存器的内容作为较小位大小的输入浮点元素的输入向量;将向量寄存器的输入浮点元素转换成较大位大小的输出浮点数的输出向量;以及将较大位大小的输出浮点元素的所产生的输出向量存储到向量寄存器文件206中的向量寄存器。在一个例子中,每个输入浮点元素的大小可以是两个字节,且每个输出浮点元素的大小可以是四个字节。在一个例子中,每个输入浮点元素的大小可以是四个字节,且每个输出浮点元素的大小可以是八个字节。

[0627] 在一个例子中,处理逻辑还可配置成:从向量寄存器文件206读取向量寄存器;将向量寄存器的内容作为较大位大小的输入浮点元素的输入向量;将向量寄存器的内容转换成较小位大小的浮点元素;以及将较小位大小的输出浮点元素的所产生的输出向量存储到向量寄存器文件206中的向量寄存器。在一个例子中,每个输入浮点元素的位大小可以是四个字节,且每个输出浮点元素的位大小可以是两个字节。在一个例子中,每个输入浮点元素的位大小可以是八个字节,且每个输出浮点元素的位大小可以是四个字节。

[0628] 在一个例子中,计算机处理器100还可配置成控制舍入和饱和模式。在一个例子中,可通过由舍入和饱和模式定义的舍入和饱和从较大浮点数得到较小浮点数的值。在一个例子中,可按照IEEE标准来定义舍入和饱和模式。

[0629] 在前面的描述中，阐述了很多细节。然而，对受益于本公开的本领域中的普通技术人员而言，明显的是，本公开可在没有这些特定细节的情况下实施。在一些实例中，以方框图形式而非详细地示出公知的结构和设备，以避免使本公开模糊。

[0630] 从对在计算机存储器中的数据位的操作的算法和符号表示方面提出详细描述的一些部分。这些算法描述和表示是由在数据处理领域中的技术人员使用来最有效地将他们的工作的实质传达给本领域中的其他技术人员的手段。算法在这里且通常被设想为导致期望结果的自身一致的步骤序列。步骤是那些需要对物理量的物理操纵的步骤。通常，虽然不是必须，这些量采取能够被存储、传送、组合、比较和以其他方式操纵的电或磁信号的形式。主要为了普遍使用的原因，将这些信号称为位、值、元素、符号、字符、项、数字等有时证明是方便的。

[0631] 然而应牢记在心，所有这些和类似的术语应与适当的物理量相关联且仅仅是应用于这些量的方便标签。除非另外特别声明，如从下面的讨论明显的，应认识到，在整个说明书中，利用术语例如“分段”、“分析”、“确定”、“启用”、“识别”、“修改”等的讨论指计算机系统或类似的电子计算设备的动作和处理，其将被表示为在计算机系统的寄存器和存储器内的物理(例如电子)量的数据操纵并变换成类似地被表示为在计算机系统存储器或寄存器或其它这样的信息存储、传输或显示设备内的物理量的其它数据。

[0632] 本公开还涉及用于执行本文的操作的装置。该装置可为了所需目的而被特别构造，或它可包括由存储在计算机中的计算机程序选择性激活或重新配置的通用计算机。这样的计算机程序可存储在计算机可读存储介质中，例如但不限于任何类型的盘(包括软盘、光盘、CD-ROM和磁光盘)、只读存储器(ROM)、随机存取存储器(RAM)、EPROM、EEPROM、磁卡或光卡或适合于存储电子指令的任何类型的介质。

[0633] 词“例子”或“示例性”在本文用于意指用作例子、实例或示例。在本文被描述为“例子”或“示例性”的任何方面或设计不必然被解释为相对于其它方面或设计是优选的或有利的。更确切地，词“例子”或“示例性”的使用意欲以具体方式提出概念。如在本说明书中使用的，术语“或”意指包含性的“或”而不是排他的“或”。也就是说，除非另有规定或从上下文是清楚的，“X包括A或B”意指任一自然包括性的排列。也就是说，如果X包括A、X包括B或X包括A和B，则“X包括A或B”在任一前述实例下被满足。此外，如在本申请和所附权利要求中使用的冠词“a”和“an”应通常被解释为意指“一个或多个”，除非另有规定或从上下文是清楚的应指向单数形式。而且，术语“实施方式”或“一个实施方式”或“实现”或“一个实现”的使用始终并不意欲意指同一实施方式或实现，除非被描述为这样。

[0634] 在整个这个说明书中对“一个实施方式”或“实施方式”的提及意指结合该实施方式所述的特定特征、结构或特性被包括在至少一个实施方式中。因此，短语“在一个实施方式中”或“在实施方式中”在整个这个说明书中的不同地方中的出现并不一定都指同一实施方式。此外，术语“或”意欲意指包含性的“或”而不是排他的“或”。

[0635] 应理解，上面的描述意欲为是示例性的而不是限制性的。在阅读和理解上面的描述后，很多其它实现将对本领域中的技术人员是明显的。因此应参考所附权利要求连同这样的权利要求享有的等效形式的完全范围来确定本公开的范围。

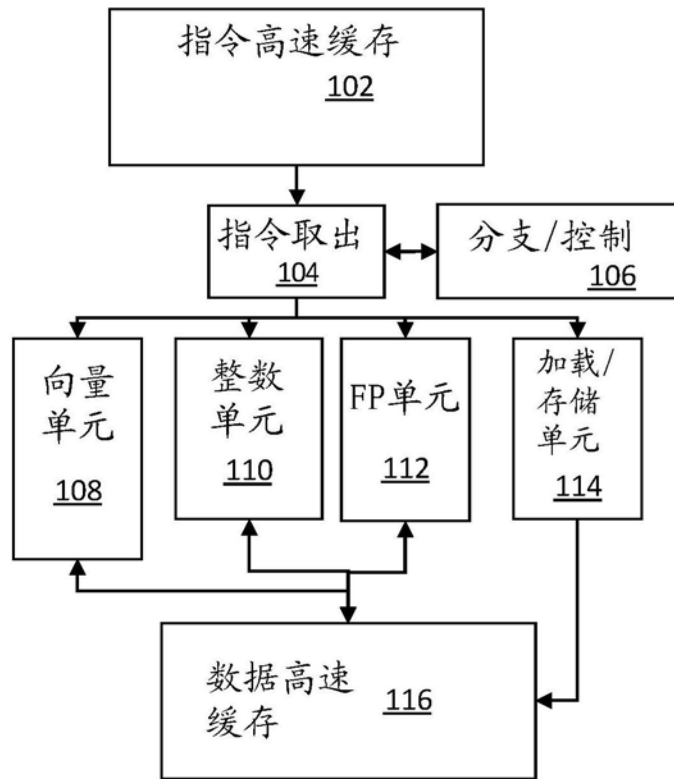


图1

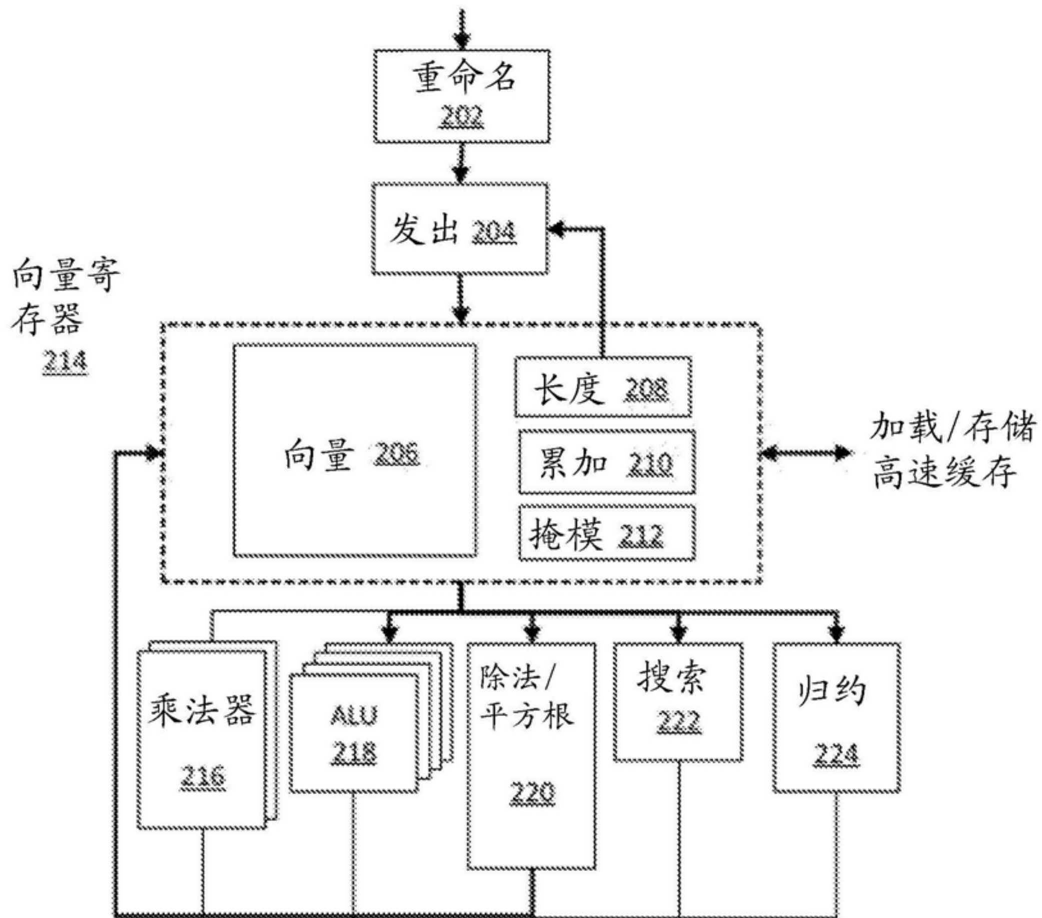


图2

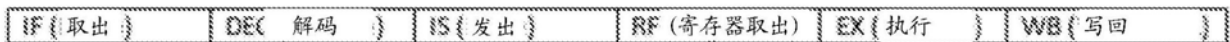


图3

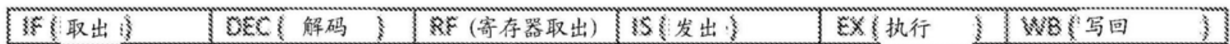


图4

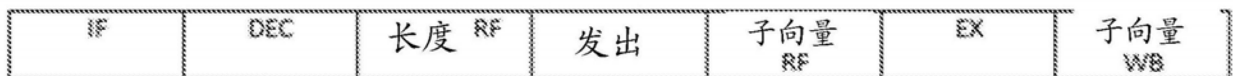


图5

	0	1	2	3	4	5	6	7	8	9	10	11
vmul 0:7	RF	EX	WB									
vmul 8:15		RF	EX	WB								
vmul 16:23			RF	EX	WB							
vmul 24:31				RF	EX	WB						
vadd 0:7							RF	EX	WB			
vadd 8:15								RF	EX	WB		
vadd 16:23									RF	EX	WB	
vadd 24:31										RF	EX	WB

图6

	0	1	2	3	4	5	6	7	8
vmul 0:7	RF	EX	WB						
vmul 8:15		RF	EX	WB					
vmul 16:23			RF	EX	WB				
vmul 24:31				RF	EX	WB			
vadd 0:7				RF	EX	WB			
vadd 8:15					RF	EX	WB		
vadd 16:23						RF	EX	WB	
vadd 24:31							RF	EX	WB

图7

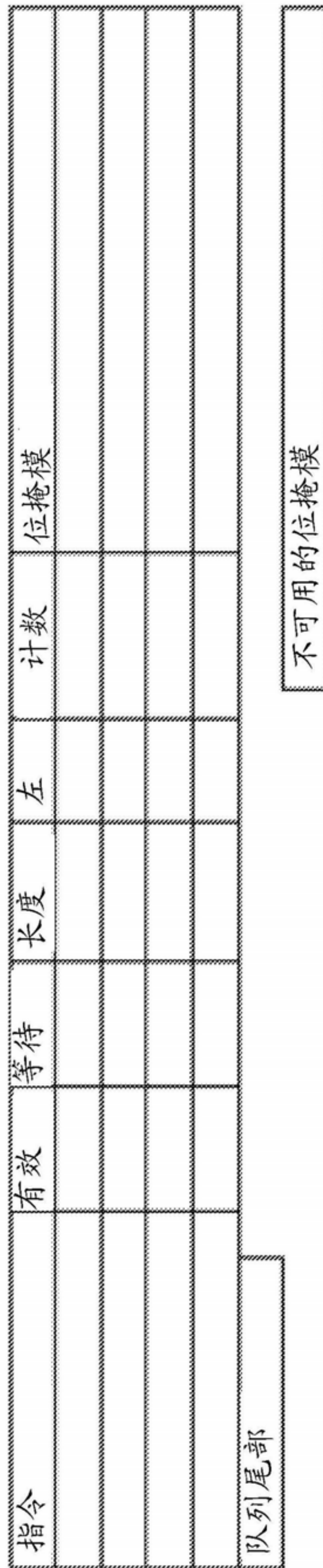


图8

架构式	映射 $v0	映射 $v1	映射 $v2	…	有效的
初始	$V0	$V1	$V2		
vadd $n0,$v0,$v1,$v2	$V8				vadd $n0,$V8,$V1,$V2
	$V8	$V1	$V2		
vmul $n0,$v1,$v2,$v0		$V9			vmul $n0,$V9,$V2,$V8
	$V8	$V9	$V2		
vsub $n0,$v0,$v1,$v0	$V10				vsub $n0,$V10,$V9,V8
	$V10	$V9	$V2		

图9

$v0.0	$v0.1	$v0.2	$v0.3	$v1.0	$v1.1
$v1.2	$v1.3	$v2.0	$v2.1	$v2.2	$v2.3
$v3.0	$v3.1	$v3.2	$v3.3	$v4.0	$v4.1
$v4.2	$v4.3	$v5.0	$v5.1	$v5.2	$v5.3
$v8.3	$v6.0	$v6.1	$v6.2	$v6.3	$v7.0
$v7.1	$v7.2	$v7.3	$v8.0	$v8.1	$v8.2
$v11.3	$v9.0	$v9.1	$v9.2	$v9.3	$v10.0
$v10.1	$v10.2	$v10.3	$v11.0	$v11.1	$v11.2

图10

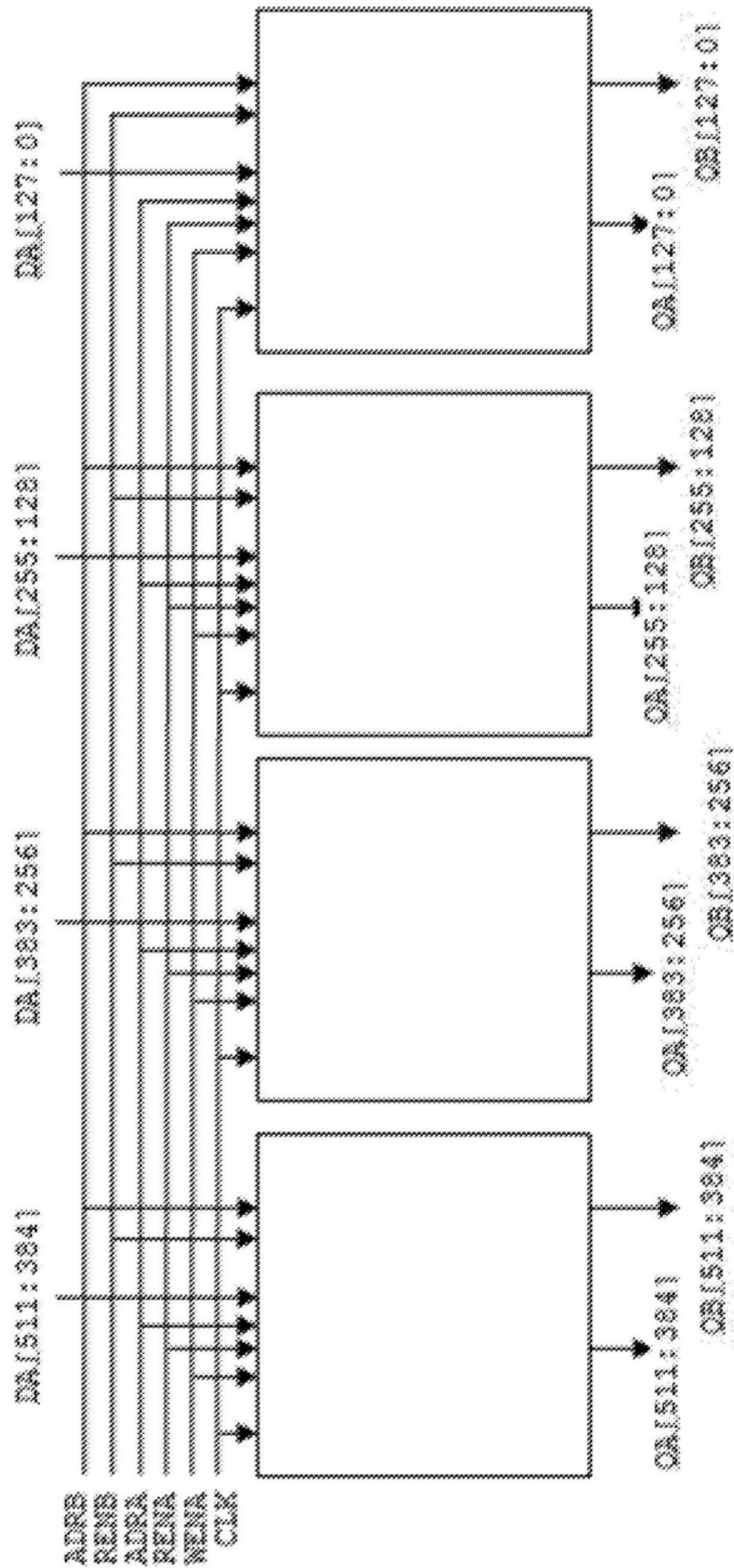


图11

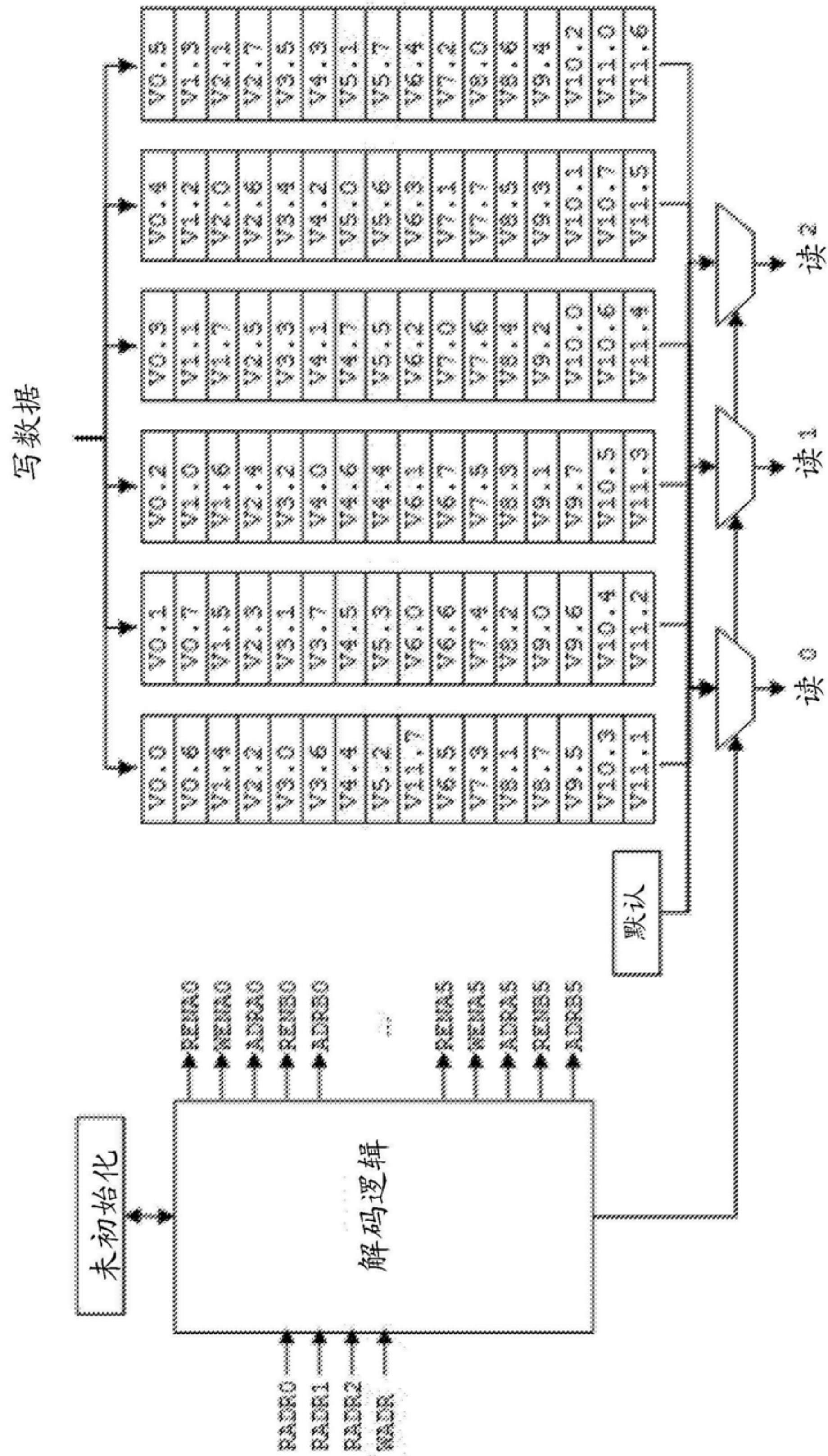


图12