RISC-V其实是反潮流，但是……

半导体行业观察 昨天

来源：内容由半导体行业观察（ID：icbank）编译自「Erik Engheim」，谢谢。

在1980年代，超级计算机的外观如下图所示。而Cray的半圆形则是80年代超级计算机的代名词。那就是一台超级计算机的样子。

1980年代的Cray超级计算机。

在一篇写RISC-V的文章里，我们提到过去的超级计算，这两者之间有什么关系？主要是因为，我们提到的Cray计算机，也被称为矢量处理机——一种已经被抛弃的“老古董”。

然而，RISC-V却将Cray风格的矢量处理重新带回来，并认为它应该替代SIMD（单指令多数据），这是否是一个异端？

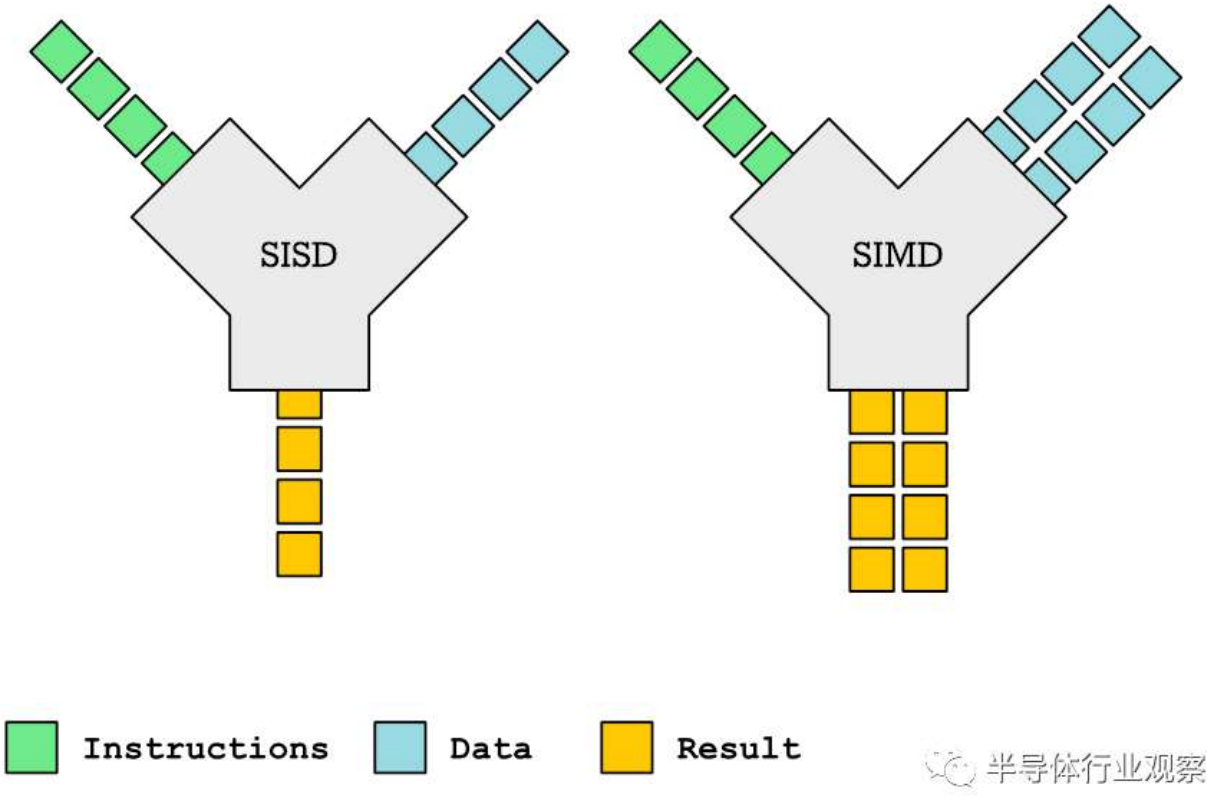
这样大胆而又不同的策略肯定需要一些解释。为什么RISC-V设计师会采用与竞争对手x86，ARM，MIPS等完全不同的方法？

像往常一样，我们需要绕道而行，以解释这些技术究竟是什么以及它们有何不同。尽管SIMD指令排在最后，但我相信从SIMD开始更容易掌握矢量处理指令。

什么是SIMD（单指令多数据）？

无论是基于x86还是基于ARM的大多数微处理器，在其中都提供了我们所谓的SIMD指令。您可能听说过MMX，SSE，AVX-2和AVX-512。而ARM有自己的高级SIMD和SVE。

这些指令允许您执行的操作是将相同的操作应用于多个元素。我们可以将它与SISD（单指令单数据）进行对比，后者仅在单个元素之间执行操作。下图是对此的简单说明：



单指令单数据（*SISD*）与单指令多数据（*SIMD*）

我们可以编写一些简单的代码来说明差异，以下是SISD的示例。我们也可以称其为标量（单个值）上的操作：

```
3 + 4 = 7
4 * 8 = 32
```

SIMD是关于向量（多个值）的操作：

```
[3，2，1] + [1,2,2] = [4，4，3]
[3，2，1]-[1,2,2] = [2，0，-1]
```

让我更详细地了解一些用于SISD的伪汇编代码（pseudo assembly code）。在这种情况下，我们要添加两个数组，每个数组包含两个元素。每个元素都是32位整数。一个从地址14开始，另一个从地址24开始：

```
load r1，14
load r2，24
add r3，r1，r2; r3←r1 + r2
```

```
load r1，18
load r2，28
add r4，r1，r2; r4←r1 + r2
```

使用SIMD，我们可以加载多个值并执行多个加法：

```
vload.32 v1、14 vload.32
v2、24
vadd.i32 v3，v1，v2; v3←v1 + v2
```

通常将向量和SIMD指令加上前缀v以将它们与标量指令分开。约定各不相同，但这是受ARM启发的，.32后缀表示我们要加载多个32位值。假设我们的向量寄存器v1和v2是64位，则意味着每次load两个元素。

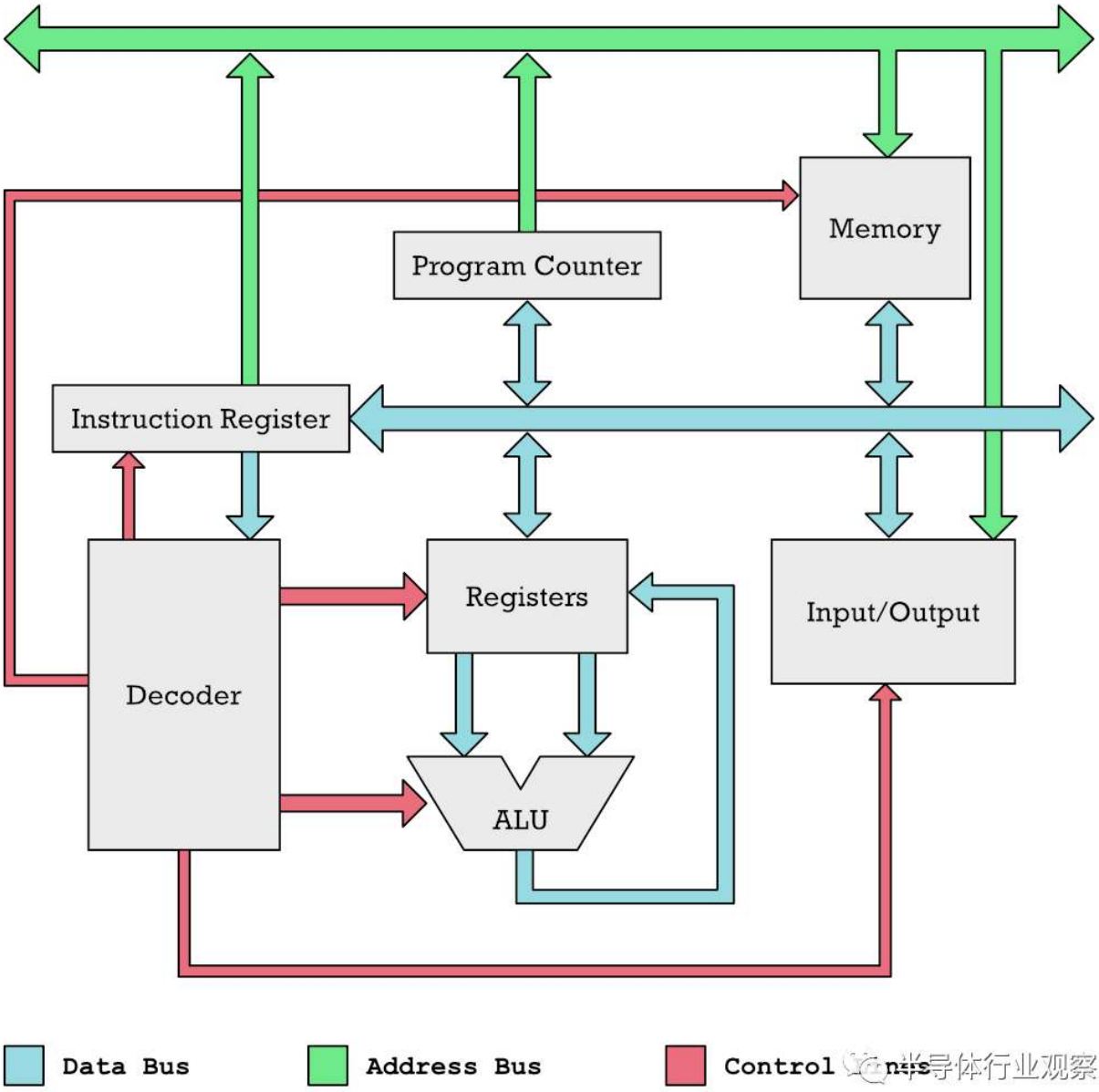
该vadd指令的.i32后缀表示我们要添加32位带符号整数。我们本来可以用来.u32表示无符号整数。

当然，这是一个完全不现实的示例，因为没有人会对这几个元素使用SIMD。更现实的是，我们将对16个元素进行操作。

SIMD如何工作?

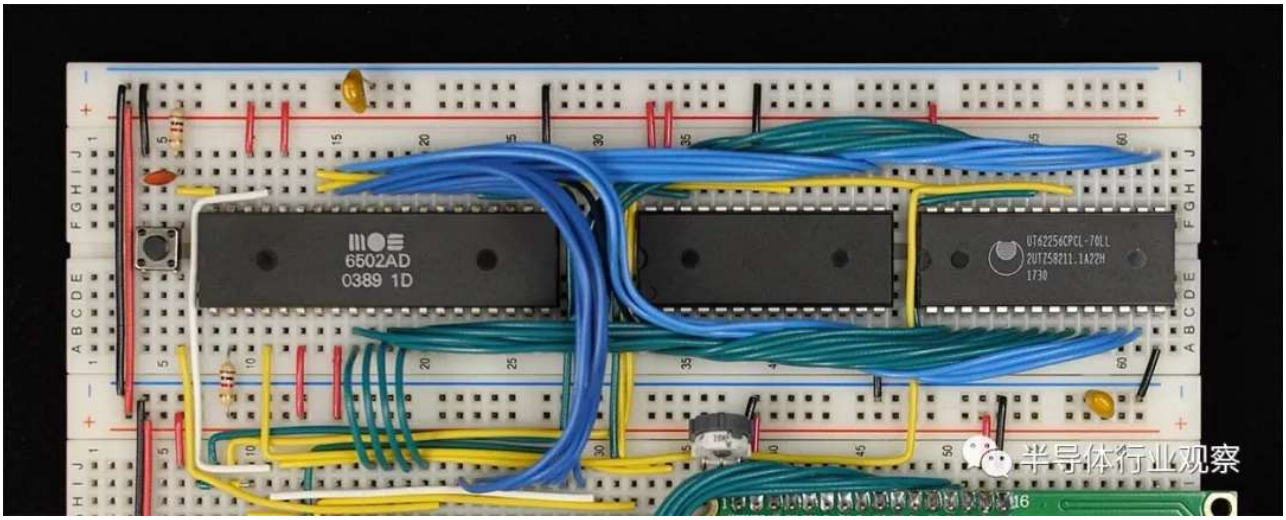
我们对SIMD指令的工作方式进行了高级描述。但是实际上，它们是如何在CPU级别处理的？执行SIMD指令时，CPU内部发生了什么？

在下面，您可以看到RISC微处理器的简化图。



一个简单的RISC微处理器的示意图。

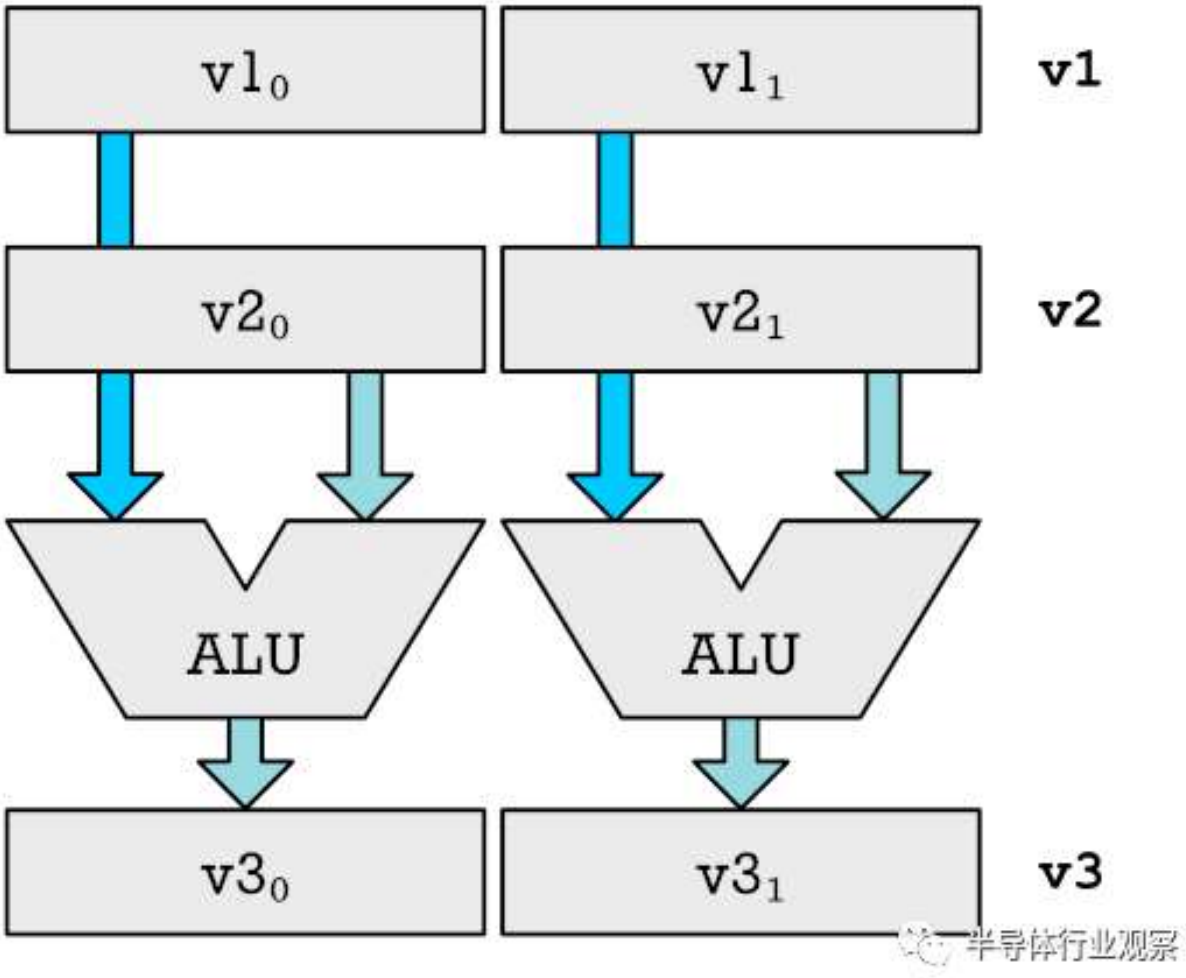
您可以将彩色条视为将数据推入CPU的不同部分的管道。我们在这里的主要兴趣是蓝色的东西，它们推动了我们操作的数据以及通过系统的指令。绿色管道是存储单元的地址位置。



Ben Eater在面包板上构建的6502计算机。彩色线是数据和地址总线以及控制线。

在一个简单的微处理器中，您只有一个算术逻辑单元（ALU）。这样的处理器的一个例子是在Commodore 64中使用的6502。ALU类似于CPU的计算器。它可以加减数字，它使用两个数字作为输入，然后将它们相加或相减，然后将输出到底部。输入来自寄存器，输出返回到寄存器（具有您要操作的保持编号的内存单元）。

要将我们的CPU变成可以同时处理数十个数字的执行SIMD的怪物，我们需要进行一些更改。以下是升级的简化示例，该升级允许同时将两个数字相加。请注意，我们仅显示与寄存器和ALU相关的部分。

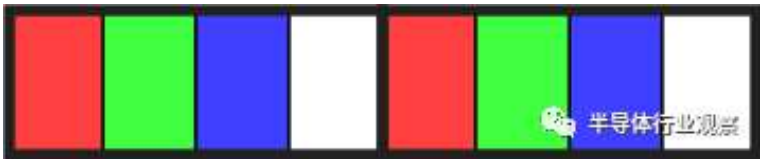


如何使用多个ALU允许执行SIMD。

v1，v2而v3就是我们所说的向量寄存器。它们分为不同的部分，显示为$v1_0$和$v1_1$。我们可以将向量的每个部分或元素输入到单独的ALU中。这使我们可以同时执行多个添加。对于真正的CPU，我们不只是添加一个额外的ALU。我们加一打。实际上，我们变得更加疯狂，我们添加了十二个乘法器和其他功能单元，它们能够执行CPU的所有不同操作。对于非常简单的CPU，您没有乘法器，因为您可以通过重复的加法和移位（加和减数字）来模拟乘法。

我们如何获得SIMD

那么这些SIMD指令是如何产生的呢？快速的图像处理的需求是起点。图像中的每个像素由四个8位值（RGBA）组成，需要将其视为单独的数字。为数百万个像素分别添加这些值很慢。SIMD指令是提高此类任务性能的明显方法。



每个像素由四个分量组成：红色，绿色，蓝色和Alpha值。每个都是一个字节，应分别计算。如果32位寄存器是具有4个组件的向量寄存器，则可以执行此操作。

SIMD还用于GPU内部，因为它们会添加位置向量，相乘矩阵。复合像素颜色值等。

SIMD的好处

虽然很难并行执行代码，但是，当处理诸如图像，几何，机器学习和大量科学计算之类的事情时，对数据的多个元素执行相同的操作相当简单。

换而言之，SIMD为我们提供了一种轻松加快这些计算速度的方法。如果可以只执行一条指令就可以加8个数字，那么基本上可以实现8倍的加速。因此，多年来x86和ARM微处理器堆积在大量SIMD指令上就不足为奇了。

GPU基本上包含执行大量SIMD计算的核心存储区。这就是大大提高了图形性能的原因，也是为什么科学代码越来越多地使用GPU的原因。

但是，如果SIMD如此出色，为什么RISC-V放弃它并进行向量处理呢？更具体地说，他们没有添加SIMD指令集扩展，而是添加了Vector指令集扩展。

SIMD指令存在的问题

RISC-V设计师David Patterson和Andrew Waterman写了一篇文章：SIMD指令被认为有害。

这是一本有趣的文章，但是它比我在这里更深入地介绍了技术。Patterson和Waterman描述了问题：

就像阿片类药物一样，SIMD的起点足够纯净。架构师将现有的64位寄存器和ALU分为许多8位，16位或32位块，然后对其并行进行计算。操作码提供数据宽度和操作。数据传输只是单个64位寄存器的加载和存储。谁会反对呢？

但这是一种推托：

自1978年以来，IA-32指令集已从80条增加到大约1400条，主要是由SIMD推动的。

因此，x86和ARM的规范和手册非常庞大。相反，您可以在一张双面纸上获得所有最重要的RISC-V指令的概述。这对于那些用硅制造芯片的人以及那些制造汇编器和编译器的人有影响，对SIMD指令的支持通常会在以后添加。

RISC-V的设计者希望有一个实用的CPU指令集，该指令集可用于长时间教学。在RISC-V到来之前，他们使用的是在商业界不再受追捧的MIPS，因为学术界不希望其教学是基于行业的潮流和炒作。大学强调教学知识的持久性。这就是为什么他们更愿意讲授数据结构和算法，而不是说如何使用调试工具或IDE。

因此，SIMD的发展是站不住脚的。每隔几年就会有新的说明。没有什么是非常耐用的。因此，Patterson 和Waterman认为：

向量架构是一种较旧的，更优雅的利用数据级并行性的替代方法。向量计算机从主存储器中收集对象，并将其放入顺序的长向量寄存器中。

回到Cray样式的矢量处理？

因此，RISC-V设计人员使用矢量指令而不是SIMD指令创建了扩展。但是，如果这样好得多，为什么它没有更早发生，为什么矢量处理在过去就不受欢迎了？
在回答任何一个问题之前，我们需要实际了解什么是向量处理。

向量与SIMD处理

理解差异的最好方法是查看一些C / C ++代码。在SIMD中，向量是固定大小的，并被视为固定长度类型，如下所示：

```
struct Vec3 {
  int x0;
  int x1;
  int x2;
};

struct Vec4 {
  int x0;
  int x1;
  int x2;
  int x4;
};
```

这意味着矢量加法函数处理的是固定长度：

```
Vec3 vadd3（Vec3 v1，Vec3 v2）{
  return  Vec3（v1.x0 + v2.x0，
           v1.x1 + v2.x1，
           v1.x2 + v2.x2）；
}
```

我们能想到的Vec3，Vec4并vadd3为现有的硬件。但是，开发人员需要更高级别的功能，并且可以组合以下操作以创建更多通用功能：

```
void vadd（int v1 []，int v2 []，int n，int v3 []）{
   int i = 0;
   while（i <n）{
      u = Vec3（v1 [i]，v1 [i + 1]，v1 [i + 3]）；
      v = Vec3（v2 [i]，v2 [i + 1]，v2 [i + 3]）；

      w = vadd3（u，v）；//efficient vector operation

      v3 [i] = w.x0;
      v3 [i + 1] = w.x1;
      v3 [i + 2] = w.x2;
      i+ = 3;
```

```
  }
}
```

您可以将其视为伪代码（pseudo-code）。要了解的一点是，您可以在处理较小固定长度向量的函数上构建功能来处理任何长度的向量。

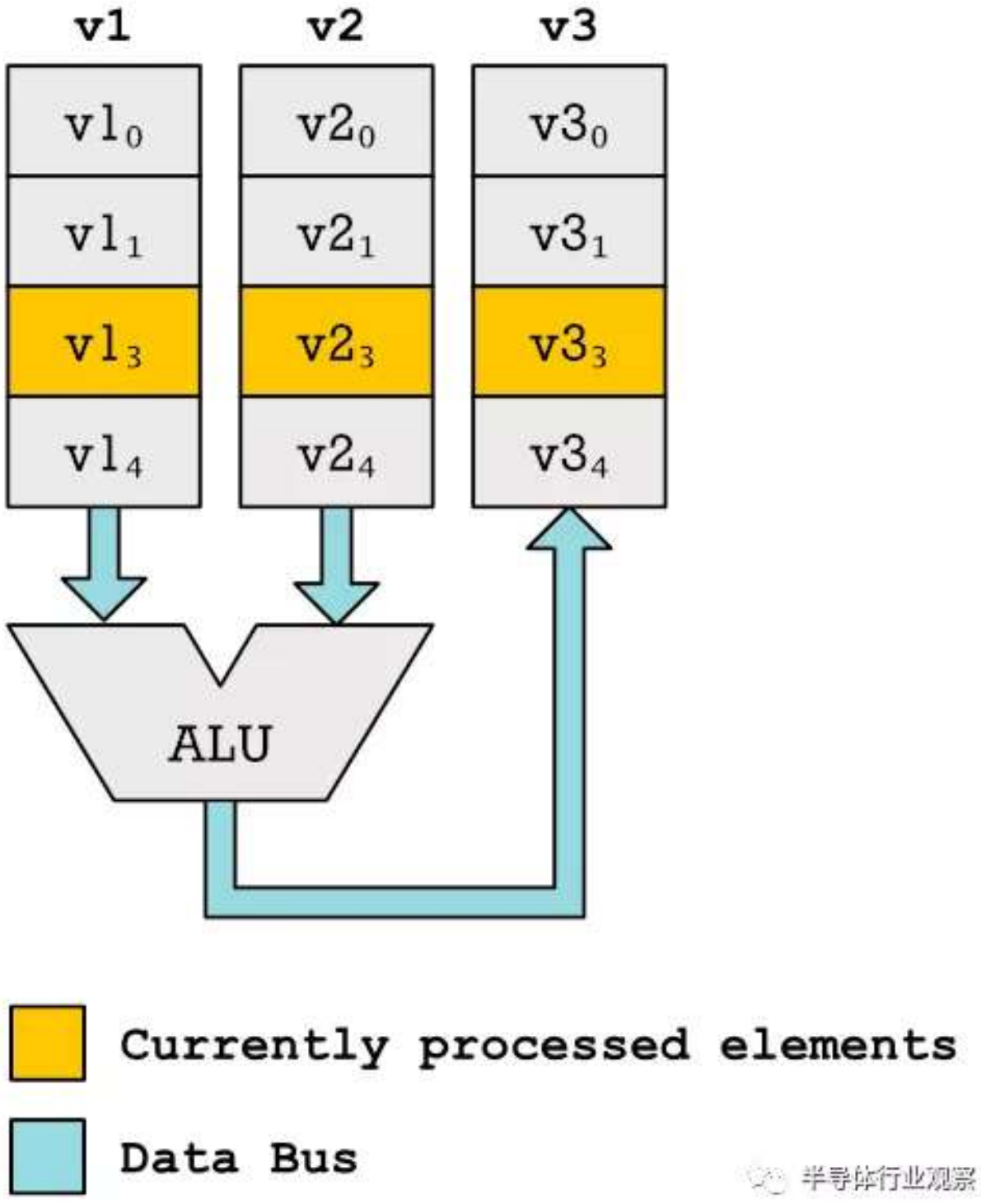
像使用老式Cray超级计算机一样进行矢量处理，这实际上是RISC-V人士提出的，就是将诸如vadd硬件之类的功能。

那这实际上是什么意思呢？

向量处理在硬件中的实现

这意味着在内部，我们仍然可以在某些固定宽度矢量上运行SIMD单元。但这不是汇编程序员所看到的。相反，就像vadd汇编代码一样，指令也不限于特定的向量长度。程序员可以将特殊的状态和控制寄存器（CSR）设置为他或她正在操作的向量的长度。这有点类似于如何vadd使用n参数指定向量的长度。

相反，我们得到了一些很长的向量。比SIMD指令使用的向量寄存器长得多。可能有数百个合适的元素。像我们对SIMD样式矢量寄存器所做的那样，为这些元素的每一个创建ALU和乘法器是不切实际的。



向量单元按顺序处理元素。当前处理的元素将突出显示。在这里，我们将单对元素。但实际上，我们使用多个ALU并按顺序处理多个元素。

相反，当CPU读取一个vadd函数时会发生什么，就像我们在伪代码示例中所做的那样，它开始遍历这些大寄存器。这是一个代码示例：

```
vsetlen r1 , 16, 120 ; 120 element vector. Each element 16-bit
vload v1, 14 ; Load 120 elements start at address 14
```

```
vload v2, 134 ; Load elements starting at address 123
vadd v3, v1, v2 ; Add all 120 elements.
vstore v3, 254 ; Store result at address 254
```

在执行操作之前，必须通过设置向量中元素的数量以及每个元素的大小和类型来配置向量处理器。在此示例中，我将其简化。我们一直在处理带符号整数。但是在实际系统中，您必须能够指定要处理的是浮点数以及带符号的还是无符号的整数。

vload做什么操作取决于配置。在这种情况下，我们将加载120个元素，每个元素距离内存16位宽。

vadd遍历v1andv2向量寄存器中的所有120个元素。将每个元素相加并将结果写入寄存器 v3。为了更好地了解它是如何工作的，让我们讨论一下所涉及的时钟周期数。

时钟周期是微处理器执行一项简单任务所需要的时间。解码指令可能需要一个时钟周期，一个指令要添加两个数字。诸如乘法之类的更复杂的操作可能需要多个时钟周期。

因为我们有四个ALU，所以我们可以在每个时钟周期执行四个加法运算。这意味着vadd需要30个时钟周期才能完成：

```
120/4 = 30
```

这听起来可能不太好。为什么不使用直接循环并执行这些SIMD指令的汇编程序直接执行相同操作。为什么要在硬件中实施必须迭代执行的操作？

向量处理的好处

一个主要的好处是我们需要小得多的程序。我们不需要编写具有多个加载，比较和循环的程序。

Patterson和Waterman在他们的文章“ SIMD指令被认为有害”中提供了一个示例程序进行比较。这是他们对程序大小差异的观察。

MIPS-32 MSA和IA-32 AVX2的代码的三分之二至四分之三是SIMD开销，用于为主SIMD循环准备数据或在n不等于n的倍数时处理边缘元素。SIMD寄存器中的浮点数。

但是更重要的是，对于矢量指令，您不必继续重复解码相同的指令。执行重复的条件分支等。在代码示例中，Patterson和Waterman使用它们来表示，与使用矢量指令的RISC-V版本相比，SIMD程序需要执行的指令多10至20倍。

原因是SIMD循环每次迭代仅处理2到4个元素。在矢量代码中，假定硬件支持具有64个元素的矢量寄存器。因此，每次迭代处理了64个元素的批处理，从而减少了需要迭代的次数。

可以在运行时查询最大向量长度，因此不需要对64个元素的大批量大小进行硬编码。

解码较少的指令会减少功耗，因为解码和获取会消耗大量功耗。

此外，我们实现了所有好的界面设计应努力实现的目标：隐藏实现细节。为什么这么重要？看看USB插头吗？当USB标准无需物理改变插头即可提高性能时，我们会喜欢它。

从USB-1到USB-3，我们避免更改接口。我们可以使用相同的电缆。同样，Vector扩展使我们可以在进行内部改进时保持相同的界面。

随着芯片技术的改进，您可以使用更多的晶体管。您可以使用它来添加更多的ALU和乘数，以并行处理更多的矢量元素。对于具有SIMD指令的CPU，这意味着您现在可以处理几百条针对新向量长度的新指令。我们也必须重新编译程序才能处理这些新添加的长向量，以提高性能。

但使用RISC-V则不需要这样，因为代码看起来一样。唯一的改变是，vadd它将以更少的周期完成，因为它具有更大的SIMD单元，从而可以在每个时钟周期内处理更大数量的元素。

如果矢量机如此厉害，为什么会被抛弃?

我想David Patterson在他先前的著作中没有很好地解释这个问题——为什么Cray矢量处理机基本上已经淘汰了。

卡车还是赛车?

要了解原因，我们需要了解权衡。如果您想将最大数量的货物从A运到B，则基本上可以采用两种方式。使用赛车以少量货物来回快速行驶。

或者，您可以使用大型的缓慢移动的卡车，该卡车可以拖运大量货物但移动缓慢。

快速传送少量数据或缓慢传送大量数据?

大多数通用软件是由赛车提供。通用程序不容易序列化。他们需要什么数据取决于执行的指令，有各种各样的条件分支和对内存的随机访问要考虑在内。每次访问仓库（内存）时，您根本无法拿起很多货物（数据），因为您不知道接下来需要什么。

因此，通用微处理器往往具有较大的快速存储器高速缓存，因此CPU可以在需要时快速获得所需的信息（与汽车类似）。

相反，矢量处理器的工作方式与GPU非常相似。他们没有处理通用程序。通常，它们用于科学软件，例如天气模拟，在其中您需要大量可以并行处理的数据。GPU同样可以并行处理大量像素或坐标。

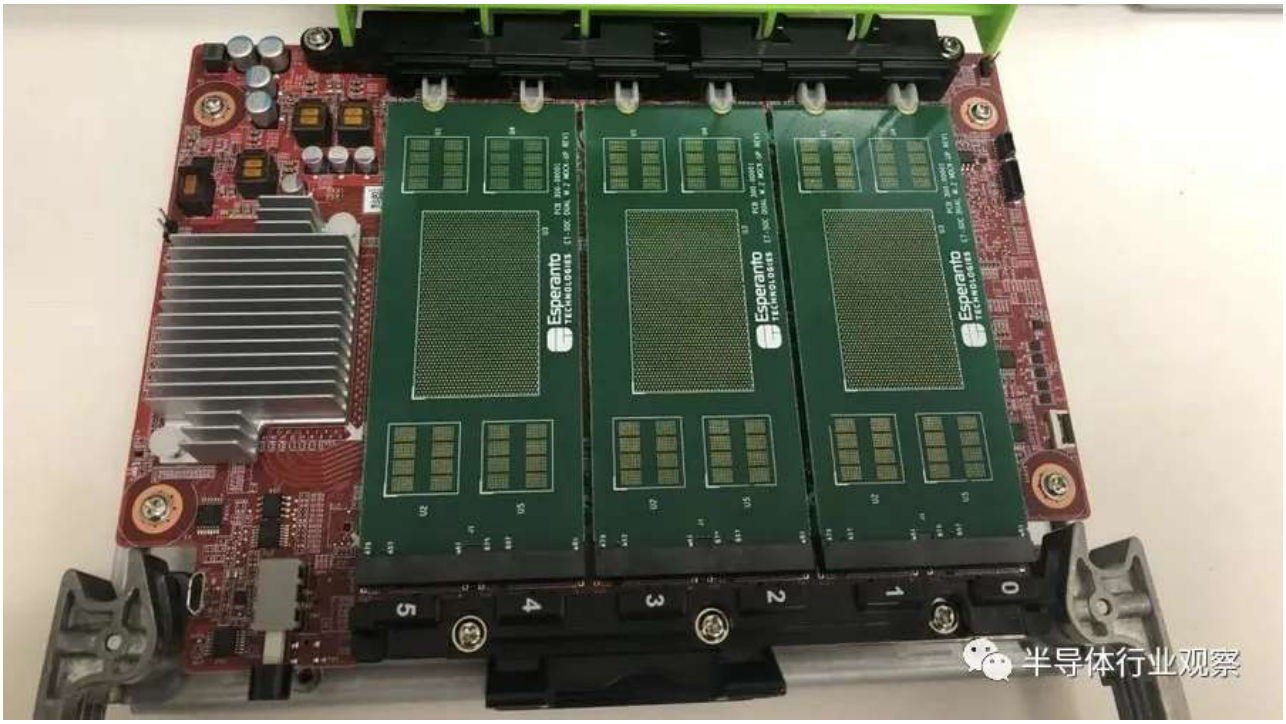
因此，您无需快速移动，因为每次都可以拾取大量数据。因此，GPU和矢量机通常具有较低的时钟频率和较小的缓存。相反，他们的内存系统设置为并行获取大量数据。换句话说，它们像卡车一样移动货物来移动数据。一次很多，但是很慢。

向量机实际上在pipelines中具有数据，因为可以预测下一个数据是什么。

向量处理器很小

当然，您可以提高矢量处理器的时钟频率，并为它们提供大量的高速缓存，但是，当您获得更多更好的选择时，又有什么意义呢？您不用花在缓存上的所有晶体管，就可以用来扩展并行处理更多元素的能力。此外，瓦特使用率和健康水平也不随时钟频率线性增长。它增长更快。因此，要降低热预算，必须降低时钟频率。

如果查看Esperanto Technologies的ET-SoC-1解决方案，您会发现所有这些折衷考虑在内。就晶体管数量而言，它们的SoC大小与Apple的M1 SoC相同。然而，矢量处理内核所需的硅要少得多，因为我们的目标不是高单线程性能。M1 Firestorm核心是怪兽，因为它们使用了许多晶体管来实现乱序执行（OoOE），分支预测，深层流水线和许多其他功能，以使单线程性能让你大跌眼镜。



板载6个*ET-SoC-1*芯片。图片：*Enterpriseai*

相比之下，ET-SoC-1可以容纳1000个以上实现Vector指令扩展的RISC-V CPU内核。这是因为矢量处理器可以做得非常小：

- 缓存需求最少。
- 它们是有序的，因此您可以通过不实现复杂的OoOE控制器逻辑来节省大量芯片。
- 较低的时钟频率简化了很多。

因此，如果您可以将问题描述为对大向量的运算，那么通过进行向量机设计，使用相同数量的晶体管，您可以获得一些疯狂的性能提升。

向量处理器吸引了通用计算

但是这里有一个关键：如果无法以这种方式表示您的程序，那么您就陷入了一个痛苦的世界。执行不能在大向量上运行的常规桌面软件将获得可怕的性能。为什么？

您的时钟频率低。您没有OoOE，并且您的缓存很小。因此，每条需要获取一些数据的指令都必须等待很长时间。这是CrY的问题。它们根本无法用于通用计算。由于使用其他传统CPU的其他市场价格便宜，而且Cray计算机上运行的许多软件都可以通过在多核计算机上运行，使用群集或其他方法来很好地完成。

当常规计算机开始需要矢量处理时，这是用于多媒体应用程序的。图像处理之类的东西。在这种情况下，您通常使用小的短向量。然后，SIMD指令是显而易见的简单解决方案。他们非常直接地进行设置。只需添加一些向量寄存器和操作即可。矢量指令需要更多的思考和计划。您需要设置矢量长度和元素类型的方法。在程序之间切换时，大的向量对于保存和恢复是不切实际的。无论如何，这些程序不需要长向量。

因此，与SIMD相比，矢量扩展最初没有明显的优势。由于矢量处理对于通用计算而言不是很好，因此Esperanto 公司开发的ET-SoC-1例如具有四个用于通用计算的RISC-V核心，称为ET-Maxion。这些更像是M1 Firestorm核心：

- 更大的缓存
- 智能分支预测器
- 多指令解码器
- 乱序执行

这些将运行操作系统并将工作任务调度到带有矢量扩展的较小的RISC-V内核（ET-minion）。这可能是架构选择的类型，我们将看到更多：混合使用具有不同强度的不同类型的核心。

通用计算不能真正受益于拥有大量内核。但是，对于特殊任务，使用非常规内核要比用于通用计算的大型内核好得多。

例如，矢量处理器可以很好地完成所有这些任务：

- 机器学习
- 压缩图像，压缩文件等
- 密码学
- 多媒体：音频，视频
- 演讲和手写
- 联网。奇偶校验，校验和
- 数据库。哈希/联接

因此，给定X晶体管数量的预算，要加快这些任务的执行速度，最好选择矢量处理器设计，而不是增加更多的通用内核。

在这两种情况下，我都在谈论苹果如何通过使用专用协处理器来从其M1芯片中提高速度。这就是这个意思。原则上，我们可以简单地将向量扩展名添加到任何RISC-V通用CPU中。但是您可以选择通过以下方式来定制向量处理的方法：删除大型缓存，将无序执行单元踢到路边，降低时钟频率，使用更简单的分支预测器以及扩大内存访问范围（读取更多数据）一次插入数据管道）。

如果这样做的话，您将获得功耗更低得多的小得多的芯片，与用于快速通用计算的胖芯片相比，矢量处理的性能可能更好。由于它体积小，功耗低，因此可以有很多。

实际上，这只是验证我先前制作的专用协处理器案例的另一种方法。

为什么矢量处理器再次出现问题

因此，这使我们可以完整地回到我们一直以来一直试图回答的内容。为什么矢量指令现在有意义，但过去却被放弃了。

该问题已得到部分回答。SIMD方法使我们陷入困境。但是更重要的是，我们的计算机现在正在执行更多各种各样的任务。特别是机器学习已经变得非常庞大。这已成为数据中心的主要重点。苹果并非没有理由在其iPad和iPhone苹果硅芯片上添加了神经引擎。

Google并非毫无理由地使用Tensor处理单元（TPU）在人群中提供了更高速度的机器学习。由于深度学习的兴起，处理非常大的阵列又重新投入了业务。

因此，我认为RISC-V将使用Vector扩展而不是SIMD扩展是非常明智的举动。SIMD诞生于一个世界，在多媒体环境中主要需要短向量。我们已经不在那个世界上了。向量扩展使用一块石头杀死两只鸟：

- 矢量指令不会使ISA膨胀。我们不需要继续添加新的。
- 未来的证据更多。
- 添加更多的ALU，乘法器和其他功能单元后，无需重新编译。

- 它们是机器学习应用程序的绝佳选择。

对矢量指令的批评

当然，并不是每个人都对我对矢量指令的热情满怀。我们得看一些批评。经常看到的一种指责是整个系统更加复杂，SIMD更加容易。人们认为矢量扩展将使芯片膨胀。

坦率地说，这不是批评，我们应该认真对待。Esperanto 已经证明，他们可以使用RISC-V向量扩展来制造小型高效芯片。

David Patterson本人并不是该领域的新手。他知道自己在做什么。他不仅是第一个RISC处理器背后的关键架构师之一，而且还积极参与了1990年代另一个鲜为人知的项目，即IRAM项目。

这在许多方面都是RISC的替代方法，后者采用了矢量处理方法。实际上，与发明原始的RISC相比，最后从事矢量处理的RISC-V人士可能会受到更大的影响。Patterson和其他人开始真正从他们的IRAM项目中相信矢量处理的强大功能和优雅。因此，V在RISC-V实际上代表两个5和载体。RISC-V从一开始就被认为是用于矢量处理的体系结构。

IRAM项目非常有趣，因为它预示了苹果M1芯片后来发生的许多事情。在广泛讨论如何在SoC上使用DRAM及其优势。提示认为统一内存。

当被问及使用向量处理指令的复杂性和难度时，David Patterson写道：没那么难。我们很早以前在IRAM项目中通过并行执行较小的数据类型来做到这一点。许多人都在使用这种矢量架构样式来构建RISC-V处理器。

在向量函数之间传递数据

但是，这也许是更严重的批评。您可以轻松创建固定长度的具体矢量类型和元素类型，并可以在高级语言中使用。因此，您可以创建一系列函数，这些函数可以获取通过向量寄存器传递参数数据。因此，我们可以组合多个功能，其中所有数据都使用矢量寄存器在它们之间传递。

例如，如果函数看起来像这样，我们可以很容易地与其他带有Vec3参数的函数一起使用：

```
Vec3 vadd3（Vec3 v1，Vec3 v2）{
  return Vec3（v1.x0 + v2.x0，
           v1.x1 + v2.x1，
           v1.x2 + v2.x2）；
}
```

如果矢量扩展名要求您将数据作为数组或指针传递给内存，则要困难得多。这意味着两个向量函数之间的数据将始终必须通过抛出主存储器来交换数据。这肯定会减慢速度。这是争论的重点：

您提倡的方法不能完全做到这一点。您无法为采用或返回千字节数据的函数发明合理的调用约定。当前的体系结构都在这些向量寄存器中传递参数和返回值，因为它们的计数和大小是ISA的一部分，即稳定且为编译器所知。

但是，我可以看到一些解决方法。例如，使用GPU编程，人们使用可以任意长度的CUDA阵列。这些实际上只是包装处理程序到图形内存中的数组。我不明白为什么向量寄存器不能以相同的方式工作。您只需使用特殊的数组类型作为这些向量函数的参数。

如果这不可能，那么“ Just in Time Compilation”实际上可能是一个不错的选择。例如，使用Julia编程语言，即时编译器通常会消除不同函数调用之间的接口。可以想象一个JIT以这种方式将几个基于向量的功能合并在一起，以避免在所有处理完成之前将结果写到内存中。

但老实说，这是我希望看到的。

为什么不使用GPU？

我看到的最后一个批评是，如果需要对长向量进行运算，则应该只使用GPU。这里的想法是，如果您需要对大数据块进行操作，那么您还可能会承受将大数据块传输到GPU进行处理然后读取结果的性能损失。

有两种看待批评的方法。如果向量扩展名仅用于较臃肿的通用CPU，它将有一些优点。无论如何，为快速处理标量数据而优化的CPU都将在向量处理方面具有劣势。

但是，我们不能假设这一点。正如我们从Esperanto中所看到的，设计专门的RISC-V处理器（例如ET-Minion）是完全有效的，ET-Minion是用于快速矢量处理的定制模式。

而且，如果我们相信Esperanto公司的主张，他们的解决方案将胜过基于GPU的机器学习解决方案。

我通常会在很多RISC-V批评中看到这个问题。它常常会遗漏标记，因为它假定我们一直在谈论用于运行Windows，Linux或macOS的通用处理器。但是，RISC-V应该适用于从微控制器，协处理器到超级计算机的任何事物。

矢量扩展对于通用CPU仍然有意义，仅因为它隐藏了矢量处理的实现细节。SIMD不能做的事情引起了很多ISA膨胀。

通用指令集的好处

即使专用处理器更适合长矢量，这并不意味着通用CPU和专用内核不能同时是具有矢量扩展的RISC-V处理器。

Playstation 3新颖的单元体系结构的问题之一是通用PowerPC CPU无法与功能更有限的协处理器共享指令集。实际上，PS3架构与Esperanto ET-SoC-1有很多共同点。PS3不是使用ET-Maxion内核作为通用CPU，而是使用PowerPC CPU来运行操作系统。PS3使用协同处理元素（SPE）代替了ET-Minion核心来处理面向矢量的主要工作量。

这是早期尝试做M1今天正在做的事情以及ET-SoC-1将来可能做的事情的尝试。

尽管写了无数关于PS3失败的页面，但提到的一个原因是中央处理器和SPE没有共享指令集。

对于开发人员而言，首先在主PowerPC CPU上进行矢量处理代码的开发和故障排除可能会容易得多。因此，在通用CPU上具有RISC-V向量扩展名对开发人员和故障排除很有用。

★ 点击文末【阅读原文】，可查看本文原链接。

今天是《半导体行业观察》为您分享的第2545内容，欢迎关注。

推荐阅读

★半导体行业的一个新机会
★疯狂的半导体2020，总融资超500亿元！
★2020年上市的半导体企业盘点

点击阅读原文，可查看本文 原文链接！

阅读原文　阅读 8528

分享　收藏　赞 18　在看 19

分享此内容的人还喜欢

为什么大家都看好RISC-V

半导体行业观察　赞 11

光子AI协处理芯片，为传统加速计算卡装上“涡轮增压”

半导体行业观察　赞 36

电子皮肤-具有传感器内自适应机器学习功能的用于手势识别的可穿戴生物传感系统

WEST可穿戴电子　阅读 1546

写下你的留言

精选留言

明奇　3

科普好文。顶一下。