RISC结构

——当代计算机发展的最新技术

何德书　田　云　周明德　编

電子工業出版社

RISC 结构

——当代计算机发展的最新技术

何德书　田云　周明德　编

电子工业出版社

(京)新登字 055 号

内容提要

RISC(Reduced Instruction Set Computer)是当今计算机发展的最新技术,RISC 结构的计算机在性能价格比上远远超出传统结构的计算机,本书的目的在于把 RISC 这一新技术介绍给我国广大计算机技术人员。本书首先简要介绍了 RISC 的概念、特征及实现方法,然后以 MIPS R2000 为例具体地介绍了 RISC 的处理器、协处理器的结构和指令系统以及 RISC 的内存管理。附录中还列出了 R2000 和 R2010 的指令集及使用。本书可作为广大计算机人员的参考书,也可作为大专院校本科生、研究生的教科书或学习参考书。

JS459/18

RISC 结构

——当代计算机发展的最新技术

何德书 田云 周明德 编

责任编辑 潘海

*

电子工业出版社出版(北京市万寿路)

电子工业出版社发行 各地新华书店经售

北京顺义李史山胶印厂印刷

*

开本:787×1092 毫米 1/16 印张:12.75 字数:326 千字

1992 年 1 月第 1 版 1992 年 1 月第 1 次印刷

印数: 5 000 册 定价:8.60 元

ISBN7-5053-1524-2/TP・282

前 言

传统的计算机经不断的改型换代，其指令集已很庞大，结构也越来越复杂。但大多数复杂的指令和为它们而设置的复杂结构不是必要的，因为对计算机实际运行的统计表明，简单指令以绝对优势占用着 CPU 及其它资源，复杂的指令很少用到。这种结论和观点，引导人们去研究“精简指令集计算机”(RISC，Reduced Instruction Set Computer)。

当代优化编译器和半导体存储器的发展，为 RISC 结构提供了最先进的工艺。于是在 80 年代，RISC 开始兴起，许多 RISC 结构的芯片(如 R2000，R3000，Spare，88000，88100 等)和计算机系统陆续走向市场。

RISC 结构的主要特点有：

1. 采用精简的指令集。RISC 结构采用精简的，长短划一的指令集，使大多数常用的操作获得了最大的效率。某些在传统结构中要多周期指令实现的操作，在 RISC 结构中，通过机器语言编程，就代之以多条单周期指令了。精简的指令集大大改善了处理器的性能，并推动和实现了 RISC 的设计。

2. 多级的指令流水线。由于所有的指令都化为能在一个机器周期内完成的单周期指令，处理器便采用了较大的流水线深度因子，同时并行处理多条指令，加快了总的执行速度。

3. 装入和存数(Load/Store)结构。在这种结构中，只有装入(load)和存数(Store)指令才去访问内存，所有的其它操作只访问保存在处理器寄存器中的操作数。

4. 用优化编译技术处理装入指令和分支指令的延迟。优化编译技术虽不是专门面向 RISC 结构的，但优化编译器依赖于 RISC 结构完成其出色的优化工作，RISC 结构又依赖于编译器得到它们更完善的性能。

5. 对操作系统的支持。RISC 结构对操作系统的支持是以计算所能获得的性能增益为依据，并以避免不必要的复杂性、以及简化和合理设计操作系统最频繁的操作为原则。这同 CISC(复杂指令集计算机)结构给操作系统提供大量的、面面俱到的支持的原则是不一样的。

本书从下述几个方面解释了这种“恰到好处”的支持：

- 虚拟存储系统
- 模式和保护
- 中断和陷阱
- 特殊功能指令

6. 总体设计的简化使机器的布局更加合理，使得设计者可以集中精力去优化那些剩下的，为数不多的，但又很关键的处理器特性。十分简化的结构使芯片上面积资源紧张的状态得以缓解，一些对性能至关重要的结构，象大的寄存器元件，转换查找缓存(TLBs)，协处理器和乘除单元都可以装在同一块芯片之上。这些附加的资源又使处理器增加了很大的性能优势。

由于 RISC 结构的上述特点，它可以用与微机差不多的代价，制造出性能比微机优越得多的工作站和小型机，在科研设计部门和高等院校等有着广阔的应用领域。预计在 90 年代，采用 RISC 结构的工作站将会和 80 年代的微机一样，风靡国内外的计算机市场。编写本书的目的，在于使读者对 RISC 结构的基本原理，设计思想，体系结构和性能有一个全面的了解，以便更好地跟上这一新技术发展的潮流。

本书第一章介绍 RISC 结构的概念。有关 RISC 结构的基本原理、设计思想和设计过程都可以从中学到。

第二章对 R2000 的特点、寄存器、指令集、编程模型、系统控制协处理器、内存管理系统以及流水线结构等都作了简要介绍。

第三章专门介绍 R2000 的指令集(摘要),包括装入和存数指令、计算指令、转移和分支指令、特殊指令,还介绍了 R2000 指令流水线和延迟的指令时间段等等。

第四章介绍 R2000 的内存管理系统。介绍虚拟内存的管理,介绍使用在片的转换查找缓存器 TLBs 的内存管理单元(MMU)及 CP0(系统控制协处理器)的工作过程。

第五章介绍 R2000 处理器处理异常的过程,以及在处理异常过程中 CP0 寄存器的工作。

第六章为 R2010(与 R2000 配合使用的浮点运算加速器)的概述。

第七章为浮点运算加速器指令集概要及指令流水线的介绍。附录 B 详述了这些指令。

第八章介绍浮点异常处理。

附录 A 为 R2000 处理器指令集。

附录 B 为 R2010 浮点指令集。

附录 C 讲解机器语言编程。通过机器语言编程,可以将在传统结构中要多周期指令实现的操作化为多条 RISC 精简指令集中的单周期指令。

附录 D 为 MIPS 汇编语言总论。

附录 E 简介 IEEE 浮点标准兼容性问题。

本书正文的全部,及附录中的 C 和 E 由何德书同志翻译,附录中的 A,B,D 由田云同志翻译,最后由周明德同志全面审校。

由于时间紧促,错误在所难免,故望有识之士不吝赐教。

编　者

1991.4.1

此为试读,需要完整PDF请访问: www.ertongbook.com

目　录

此为试读,需要完整PDF请访问: www.ertongbook.com

第一章　RISC 结构：概述

MIPS R2000 RISC 结构与传统结构的计算机相比，在性能价格比上已显示出引人注目的优势。这种优势产生于它的开发手段，这种开发手段需要在各学科之间——包括 VLSI，CPU 组织，系统级结构，操作系统和编译器的实现等——进行优化考虑。在优化处理中的各种权衡和比较，代表了并确实成了 RISC 设计的本质。

虽然本书大部分篇章用来叙述 R2000 的结构，但在本章，我们将描述那些在总体上表征 RISC 结构的基本概念。

1.1　概述的范围

RISC 设计是一个仍处在幼年期的方法学，在它努力壮大的过程中，忍受着常见的成长的痛苦。由于问题的复杂性以及它的不断变化的状态，更为深刻和全面的分析已经超出了本书的范围。由于设计方法本身的特点，使得我们很难用一两句话来说清楚 RISC 究竟是什么东西。RISC 的设计方法，包含着各种权衡比较，包含着在软件和硬件，硅片面积和编译技术，元件加工工艺和系统软件需求之间的折衷和协调，等等。因此，本章只打算简单地描述一下 RISC 的概念和它们的实现，为的是较好地理解和体会 MIPS R2000 处理器的结构。

1.2　什么是 RISC？

至今，越来越复杂的处理器一直是计算机结构的发展的主流。由于计算机市场不能不顾及现已存在的大量软件，复杂指令集计算机(CISC，Complex Instruction Set Computer)的结构随着越来越多的宏代码和功能越来越强的操作而向前发展，其目的是为高级语言和操作系统提供更多的支持。而半导体技术的进步使得制造更为复杂的集成电路成为可能。工艺越先进，制造出来的超大规模集成电路(VLSI)越复杂，计算机的结构也将变得越复杂，这象是不言而喻的了。

然而，近年来，简精指令集计算机(RISC，Reduced Instruction Set Computer)结构将硬件、固件和软件之间的复合状态做了不可思议的分割。RISC 的概念产生于对软件在实际的运行中是怎样使用处理器这个资源的统计学分析。对系统内核和由优化编译器产生的目标模块的动态度量向我们揭示了最简单的指令占有压倒的优势这一事实，即使在 CISC 结构的机器的代码中亦是如此。复杂的指令很少用到，因为微码(microcode)很少提供支持各种高级语言和系统环境所需要的精巧程序。因此，RISC 设计淘汰了微码程序而将机器的低级控制交给了软件。

这种方法并不新鲜，只是近年来它得到了更普遍的应用。这应当归功于高级语言的流行，能在"微码"级上优化的编译器的开发，以及半导体存储器和组件的引人注目的发展。用更快的 RAM 组成的指令超高速缓存(Cache)来取代机器的微码 ROM，在当前是完全可以办到的。机器的控制驻留在指令的超高速缓存中，这种安排也是很巧妙的。由系统和编译器生成的代码产生的指令流使得高级软件的需求和硬件功能之间配合得非常默契。

请注意，减少或简化指令集并不是在此描述的结构概念的主要目标，它只是一种从先进的工艺中获取最高级的性能的技巧的副产品，所以"精简指令集计算机"一词有一点令人误解：它改善了性

能，这种性能正在推动和实现着 RISC 的设计。因此，现在就让我们从定义性能开始吧！

1.2.1 定义性能

处理器的性能可以定义为完成一个特定的任务（如程序，算法，基准测试）所需要的时间，并且可以用三个因子的乘积来表示：

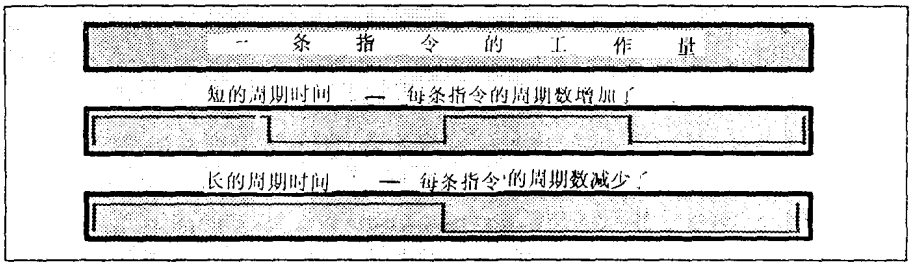
每个任务的完成时间 $= C \times T \times I$

其中，C= 每个指令的周期数，T= 每个周期的时间（时钟速度），I= 任务的指令数

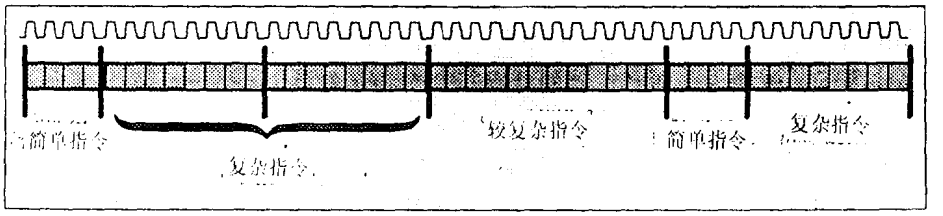
减少这三个因子中的任何一个，就可以达到改善性能的目的。RISC 类型的设计，是通过将头两个因子减少到最低的办法，来提高性能的。可是，每条指令的周期数和每个周期的时间这两个因子的减少，却导致了每个任务的指令数这个因子的增加：这种趋势已经成了影响 RISC 的最为首要的难题。然而，优化编译器和其它技术的应用，缓和了这种趋势。下面的章节将分别讨论这三个与性能相关的因子和用在 RISC 设计中的减少每一个因子的典型技术。

1.3 每条指令的时间(Time per Instruction)

执行一条指令所需要的时间是上节导出的等式中的头两个因子（C 和 T）的乘积。这两个因子是互补的：时钟速度的增加（减少了每一个周期的时间）减少了每个周期内能够完成的工作量。这样一来，快的时钟速率（短的周期时间）导致了执行一条指令所需的周期数的增加，请看下面的说明：



对于大多数处理器，无论用短的周期时间和多的指令周期，还是用长的周期时间和少的指令周期，差别都不大，因为每条指令的总时间（每个周期的时间乘以每条指令的周期数）才是真正有意义的。周期时间典型的选取方法是这样的：使得每一个最简单的操作（或子操作）正好能在一个周期内完成，而其它比较复杂的操作则允许其在多个周期内完成。这样，在典型的 CISC 处理器中的指令流可能是这样的：



在上面的例子中间，执行最简的指令需要 4 个周期，而执行复杂一点的指令则需要 8 到 12 个周期。这种方法看起来对于时间的利用比较合理：简单的指令执行起来很快，而复杂一点的指令则需要多一点的时间。每一条指令恰好消耗了它所需要的时间总量——不多也不少。这种技术有着一种非常有害的回流，因此使它不适用于 RISC 类的设计：它使得指令流水线的使用大大复杂了。

指令流水线是一项用于减少指令周期因子的基本技术，如果采用的指令集中，包含的指令的周期长短不一，就会在很大程度上抵消掉流水线所能提供的增益。指令流水线的优越性和它们的使用对指令集设计产生的影响将在以下的章节中讨论。

1.4 每条指令的周期(Cycles per Instruction)

如果每一条指令要执行的任务是简单而又明了的话，那么执行每一条指令所需要的时间可以被压缩，周期数可以减少。RISC 设计的目标是实现一个机器周期执行一条指令的速度。接近这个目标的技术包括：

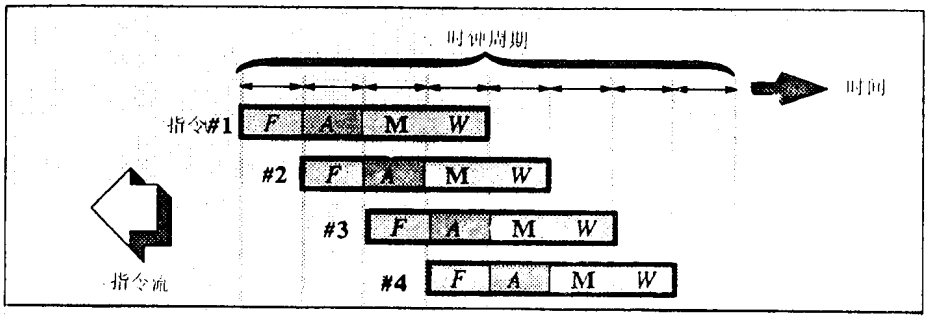
- 指令流水线
- 取/存结构
- 延迟的装入指令
- 延迟的分支指令

1.4.1 指令流水线(Instruction Pipelines)

减少执行一条指令所需的周期数的方法是重叠执行多条指令。指令流水线采用这样的工作方式：将每条指令的执行划分为几个离散的部分，然后同时执行多条指令。例如，一条指令的执行可以划分为右图所示的四个部分：

周期 #1	周期 #2	周期 #3	周期 #4
取指令	运算操作	内存存取	写结果
(F)	(A)	(M)	(W)

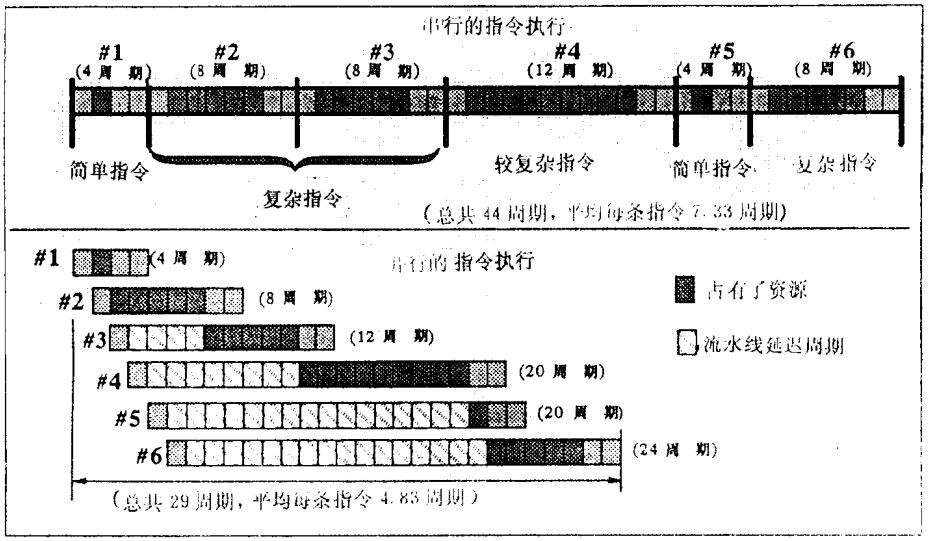
在这个例子中，执行一条指令需要 4 个时钟周期。因此，一个指令流水线，利用一个等于其流水线深度的因子，来减少其指令周期数是有潜在的可能性的。例如，在下面一张图中，每一条指令仍然要求 4 个时钟周期的总执行时间。但是如果用一个四级指令流水线的话，在每一个时钟周期的头上，都可以开始执行一条新的指令。这样一来，等效的执行速度就变成了每条指令一个周期了。



指令流水线技术可以比作一条装配线——指令象是被加工的产品一样，从一道工序流到下一道工序，一直到它执行完退出为止，正好比一辆汽车沿着一条装配线向前移动。试与流水线相比较，微码的方法是让所有的工作全由一个普通的单元来完成，在每一个独立的任务中，它(微码)显得能力较差。

在前面我们说过，流水线可以引用一个和流水线深度相等的因子，所以它有减少指令周期数的潜在可能。将这种潜在的可能变为现实，要求流水线始终充满有用指令并且没有任何东西阻碍指令通过这根流水线。这些需求给结构增加了一定的负担。例如，在串行执行指令流的较早的例子中，

每一条指令可以要求不同数量的时钟周期。下面一张图说明了如何把指令流看成是它沿着一个流水线流动。



在这个例子中，涂上同一灰度黑影的格子，表示要求使用同一个资源(如算术运算器、移位器或寄存器)的指令周期。这些资源的竞争，阻滞了流水线中指令的流动。对于许多指令来说，等待这些资源时无形中插入了若干延迟周期。在这个例子中，流水线技术减少了每条指令的周期数的平均值，但这样的收益由于不得不插入延迟周期而大大地降低了。

实际上，长短不一的执行时间所引起的副作用，比前面的例子所指出的还要坏得多。一个指令流水线的管理要求合理而有效地处理那些能够完全打断指令流的事件，如分支、异常或中断。如果指令流包含有各种不同长度的指令，并且正常的周期与延迟混杂不清的话，那么流水线的管理将会变得非常复杂。此外，这样一种混合的，复杂的指令流使得编译器几乎无法为指令做出减少和限制上面提到的那种延迟的安排。于是，RISC 设计的第一步是定义一个指令集——在此集中，所有的或大多数指令的执行所要求的周期数是一致的，在理想情况下，达到一个指令只占一个周期的执行速度。

1.4.2 装入 / 存数结构(Load /Store Architecture)

指令流水线的讨论，使我们明白了每条指令如何被分割成一系列离散的部分，以及处理器怎样并行地执行多条语句。为使这项技术更有效地发挥，每条指令的每一个子部分的执行时间应该是大致相等的。若某个子部分要求的时间分外长，那么，全部周期都应当加长；或者对于较长的操作再增加几个周期。

执行与内存中的操作数有关的操作指令，不是要求增加周期时间，就是要求增加指令的周期数，二者必取其一。因为这些指令要计算操作数的地址，将所要求的操作数从内存中读出，计算得出结果，再把操作结果送回内存，所以它们的执行时间就长得多。为了消除这些指令的负作用，RISC 设计实现了一种装入和存数(Load/Store)结构。在这种结构中，只有装入(load)和存数(store)指令才去访问内存，而所有其它的操作只访问保存在处理器寄存器中的操作数。这种方法的优点在于：

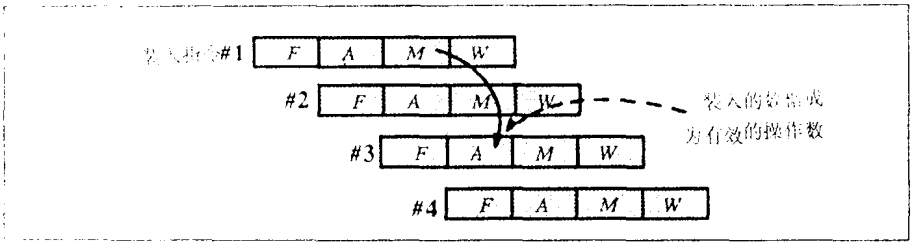
- 减少访问内存的次数降低了对内存带宽的需求。
- 将所有的操作限制于只针对寄存器，帮助了指令集的简化。

·取消了内存操作，使得编译器优化寄存器的分配更加容易——这种特性减少了内存的存取，同时也减少了“每一个任务的指令数”这个因子。

所有这些因素都有助于 RISC 设计实现它的每个周期执行一个指令的目标。尽管如此，这两类指令——装入指令(load)和存数指令(store)仍然阻碍着 RISC 设计目标的实现。下面的章节将讨论 RISC 设计是如何克服这些类型的指令所带来的障碍的。

1.4.3 延迟的装入指令(Delayed Load Instructions)

装入指令将其它指令的后续操作所需的操作数从内存读至寄存器。因为一般来说，内存操作比处理器的时钟速率慢得多，因此在利用指令流水线的处理器内，这个被取的操作数，对于跟在它后面的指令，并不会立即生效。这种数据和指令间的时序关系，在下面这张图上可以看得很清楚：



大家可以看到：#1 指令要取的操作数，在#2 指令的“A”周期内是不会有效和可用的。处理好这种关系的办法就是在取出的数据成为有效之前，在 #2 指令的执行过程中插入必要的附加时钟周期，以“拉长”这条“指令流水线”。这种方法显然引入了增加“每条指令的周期数”因子的时间延迟。

在许多 RISC 设计中用来处理这种数据相关性的技术就是使得编译器承认和感受到这样一种事实：所有的装入指令本身非有一段等待时间或装入延迟不可。在上面的说明中，取数的延迟和等待时间就等于一条指令(的间隔)。直接跟在装入指令后面的那条指令就被说成是在一个“装入延迟时间段”内。如果在这个时间段中的指令并不需要这次取出的数据，那么这流水线也就不需要延迟了。

如果软件看到这个装入延迟的存在，那么编译程序会巧妙安排这些指令，以确保在取数指令和处在装入延时段中的指令之间不存在这种数据的相互依赖关系。这种确保无数据相互依赖关系存在的最简单办法，就是插入一个 NOP(无操作)指令来填充这个时间段：

```
Load        R1,A
Load        R1,B
NOP             <——这就是填充延时段的指令
Add         R3,R1,R2
```

在这种情况下，虽不需要硬件控制的流水线等待，但是用 NOP 指令来填充这个延时段仍然不能很有效地利用这个流水线流，因为 NOP 指令增加代码的长度，并且做了没有用的工作。(然而，事实上 这种技术对于性能没有更多的副作用，特别是在这延迟不止一个周期的情况下。)

处理这种数据相关状态的更有效的方法是在这个装入延时段中填充一个有用的指令。一个好的优化编译器常常会作到这一点，特别是当这个装入延迟仅仅是一条指令时。下图说明了一个编译器如何通过重新排列指令的办法来处理潜在的数据相关性的。

对于 C := A + B; F := D 两种计算的程序编排

此为试读，需要完整PDF请访问：www.ertongbook.com