# 精简指令集计算机

15.1 指令执行特性

* 操作
* 操作数
* 过程调用
* 推论

15.2 大寄存器组方案的使用

* 寄存器窗口
* 全局变量
* 大寄存器组与高速缓存的对比

15.3 基于编译器的寄存器优化

15.4 精简指令集体系结构

* 采用CISC的理由
* 精简指令集体系结构特征
* CISC 与 RISC特征对比

15.5 RISC流水线技术

* 使用规整指令的流水线技术
* 流水线的优化

15.6 MIPS R4000

* 指令集
* 指令流水线

15.7 SPARC

* SPARC寄存器组
* 指令集
* 指令格式

15.8 RISC与CISC的争论

15.9 关键词、思考题和习题

|  |
| --- |
| **学习目标**  学习本章之后，你应该可以：   * 对推动RISC方法发展的指令执行特性提供概括性的研究结果。 * 总结了RISC机器的主要特点。 * 了解使用大型寄存器文件的设计和性能含义。 * 理解使用基于编译器的寄存器优化来提高性能。 * 讨论RISC体系结构对流水线设计和性能的影响。 * 列出并解释RISC机器上流水线优化的关键方法。 |

自从1950年左右存储程序计算机发展以来，在计算机组织和体系结构领域几乎没有真正的创新。以下是自计算机诞生以来的一些主要进展：

* **系统概念**（family concept）：1964年IBM在其System/360机器上引入此概念，之后不久又在了DEC的PDP-8使用了此概念。系统概念将机器的架构与它的具体实现分离开来。以不同的价格/性能特征提供的一组计算机，对用户来说具有同样结构，它们被称为一个系列。性能和结构方面的差异在于同样结构的不同实现。
* **微程序式控制器**（micro programmed control unit）：它是Wilkes于1951年首先提出的，并在1964年被IBM引入到它的S/360生产线。微程序设计使控制器的设计和实现变得更容易，并提供了对系统概念的支持。
* **高速缓存存储器**（cache memory）：商品化使用首先是在1968年的IBM S/360 型号 85机器上实现的。在存储器层次结构中插入cache这个层次，极大地改善了系统性能。
* **流水**（pipelining）：将并行性引入机器指令程序顺序本性的一种方式，例子是指令流水和向量处理。
* **多个处理器**（multiple processors）：这一类包含了几种不同组织和目标机器。
* **精简指令集计算机**（reduced instruction set computer, RISC）结构：这是本章的重点。

RISC架构在其刚出现时与处理器架构的传统趋势大相径庭。RISC结构分析将我们带入计算机组织和体系结构热点的讨论。

虽然，已有不同的团体以各种方式定义和设计了RISC系统，然而有些关键点是大多数设计都采用的：

* 通过大量的通用寄存器和（或）使用编译器技术来优化寄存器的使用。
* 一个有限且简单的指令集。
* 强调指令流水的优化。

表15-1比较了RISC和非RISC系统。

本章以简要综述指令集的几个研究结果开始，然后考察上述的三个主题，最后介绍RISC设计中的两个实例。

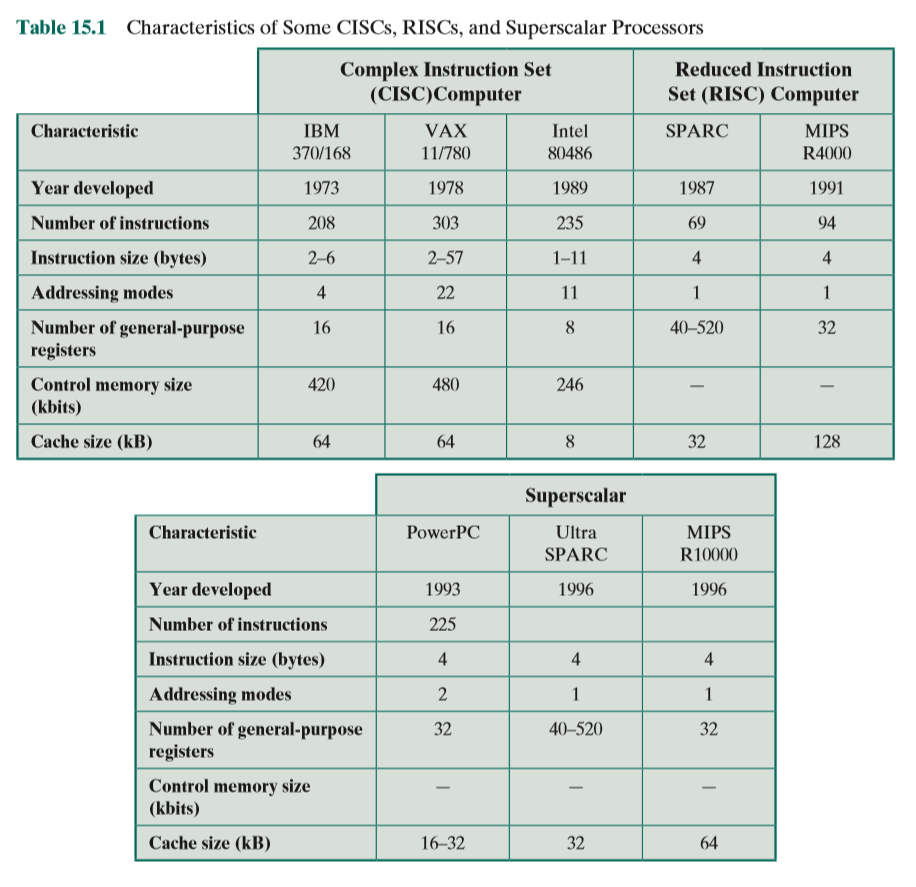
## 15.1 指令执行特征

计算机发展的最易见形式之一是编程语言。随着硬件成本的下降，软件的成本相对上升。另一方面，编程人员的长期缺乏也驱使软件成本在绝对意义上上升。因此，一个系统生存期的主要成本是软件而不是硬件。除成本和不便利之外，还有不可靠因素：不论是系统程序还是应用程序，运行多年之后虽经不断修正仍继续出现新的故障。

研究人员和业界对此的响应是，开发出了功能更强、更复杂的高级程序设计语言。这些高级语言（high-level languages, HLL）：（1）允许编程人员能更简明地表示算法；（2）允许编译器处理在编程人员的算法表示中不重要的细节；（3）通常自然地支持结构化编程和（或）面向对象设计。

然而，这种解决方法又提出了一个称为语义差距（semantic gap）的问题，即HLL中提供的操作与计算机硬件结构提供的操作间的差异。这种差距被认为是执行的低效、过长的机器程序和编译器复杂性的缘由。设计者试图以结构的改进来减小这个差距。关键的做法包括大的指令集、众多的寻址方式和硬件实现的各种HLL语句。后者的一个例子是VAX机上的CASE机器指令。这种复杂指令集希望：

* 使编译器编写者得到任务变得容易。
* 提高执行效率，因为复杂操作序列能以微代码实现。
* 提供更复杂更精致的HLL支持。



与此同时，确定HLL程序生成的机器指令执行的特征和样式，这样的研究已进行多年。研究结果促使设计人员寻找一种截然不同的方法：使支持HLL的硬件结构更简单而不是更复杂。

因此，为理解主张RISC的理由，我们先简单回顾一下指令执行特征。所关注的涉及计算的方面如下所示：

* **执行的操作**：这些操作确定了CPU机器与存储器相互作用所能完成的功能。
* **所用的操作数**：操作数类型和它们使用的频度，确定了存储它们的存储器组织和访问它们的寻址方式。
* **执行顺序** ：这确定了控制和流水线的组织。

下面，我们总结几个有关高级语言研究的报告，所有这些都使动态测量结果。动态测量是通过运行程序，并统计所出现的某个特征的次数，或者某个特征为真的次数，来收集得到的。相对地，静态测量只是在源程序文本上进行统计，这不会给出很有用的性能信息，因为它们没有对每条语句的执行次数加权。

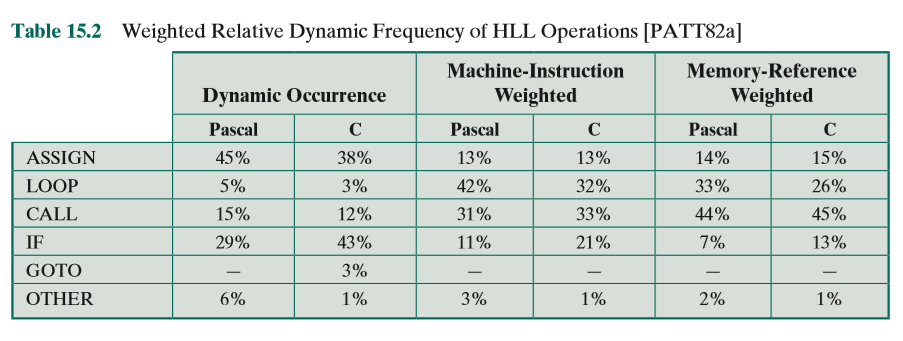
### 15.1.1 操作

已有不少研究分析了HLL程序的行为。第4章讨论过，表4-7包括了这些研究的重要结论。在混合的语言和应用的研究中，结论也具有相当的趋同性。复制语句在程序中很显著，这暗示简单的数据传送非常重要。条件语句亦在程序中占有优势，这些语句（IF、LOOP）是用一些比较和分支的机器指令来实现的。这表明指令集的顺序控制机制是关键。

这些研究成果对机器指令集设计人员具有指导意义，指出了什么类型语句出现最频繁，因而应以一种“优化”形式来支持它们。然而，这些成果未揭示什么样的语句在一个典型程序的执行中占用了最大时间。也就是说，对于一个给定的编译后的机器语言程序，源语言中的什么语句可能占有最多的机器语言指令执行次数。

为寻找这种潜在的规律性，研究者曾在VAX、PDP-11和Motorola 68000上编译Patterson程序[PATT82a]，其描述可参见附录4A，以确定每类语句的平均机器指令数和平均存储器访问数。表15-2的第2、第3两列是程序中各类HLL语句出现的相对频度。这些数据是通过观察程序运行得到的，故它们是动态频度统计。将这两列数据乘以编译器为各语句产生的机器指令数，再将乘积规范化就得到表中第4、第5两列数据，这样它们是机器指令加权后各类HLL语句的相对出现频度。类似地，将第2、第3两类数据乘以各语句引起的存储器相对访问次数，就得到第6、第7两列数据。第4~7列数据提供了执行各类语句所花费时间度量的一种替代测量值。此结果指出，过程调用/返回是典型HLL程序中最耗时的操作。

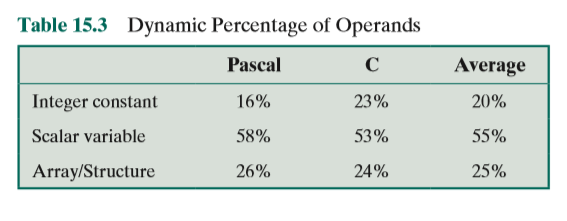
读者应清楚地了解表15-2 中的含义。该表指出当HLL程序被编译到典型的当代指令集结构时，HLL中各类语句的相对分量。某些其他结构肯定会产生不同的结果。不过，这个表给出的结果是以当代复杂指令集计算机（CISC）结构为代表的。于是，它们能为寻找支持HLL的更有效方式提供指导。



### 15.1.2 操作数

虽然操作数类型的出现率也是一个重要课题，但在这方面所做工作很少。操作数有几个方面是比较重要的。

前面提到过的Patterson研究报告[PATT82a]也查看了各类变量的动态出现频度（见表15-3）。Pascal和C程序的结论是一致的：主要使用的是简单标量变量，而且80%以上的标量是局部标量。于是，程序中大量访问的是标量，而且它们是高度局部化的。



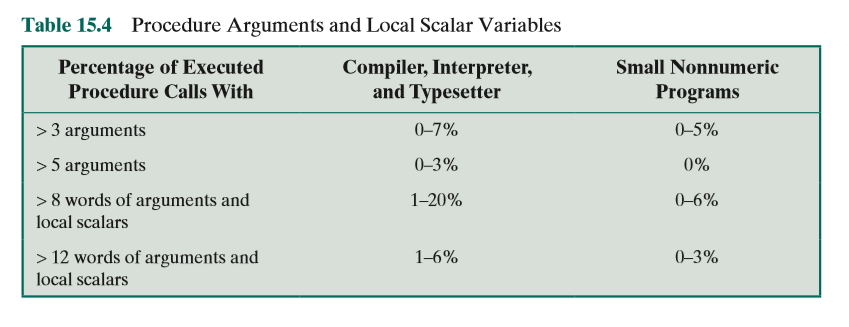
Patterson研究以独立于底层结构的方式考察了HLL程序的动态行为，正如前面讨论的，有必要针对实际结构来更深入地考察程序行为。一项研究[LUND77]动态地考察了DEC-10指令，平均的统计数据显示，每条指令访问0.5个存储器操作数和1.4个寄存器操作数。[HUCK83]报告对于运行在S/370、PDP-11和VAX机上的C、Pascal和FORTRAN程序也有类似的结果。当然，这种状况很大程度上取决于体系结构和编译器，但它们说明了操作数存取的频度。

这些后来的研究显示，因为操作数存取如此频繁，所以采用快速存取的结构将起重要作用。Patterson研究揭示，优化的主选方向应是对局部标量变量的存储和访问。

### 15.1.3 过程调用

我们已经看到，过程调用和返回是HLL程序的一个重要部分。表15-2指出，它们是编译后的HLL程序中最耗时的操作。于是，考虑高效地实现这些操作的方式将是有益的。其中两个方面有显著意义：过程使用的参数及变量的数量和嵌套的深度。

在Tanenbaum的研究[TANE78]中指出，98%的动态调用过程中传送的参数少于6个，而且其中92%使用少于6个局部标量变量。Berkeley的RISC小组报告了类似的结果[KATE83]，如表15-4所示。这些结论表明，每个过程调用所需字的数目是不多的。前面介绍的一些研究报告曾指出，操作数访问的绝大部分是对局部标量变量的访问。这些研究报告又指出，这些访问实际上只是限定在少数变量上。



Berkeley小组也研究了HLL程序中过程调用和返回的样式。他们发现很少出现这种情况：一系列长的不被打断的调用后面跟着一系列相应的返回。确切地说，他们发现程序保持在相当窄的过程调用窗口区域内。这已在图4-21中说明过。这些成果进一步证实了操作数访问是“高度局部化”的这一结论。

### 15.1.4 推论

一些研究组考察了上述这些报告的结果后认为：试图让指令集结构更接近HLL并不是一个有效的策略。相反，通过优化典型HLL程序中最耗时操作的性能，能更好地支持HLL。

由不少研究者的工作可发现，总的来说，RISC结构特征通常体现在以下三点：首先是使用大量的寄存器，这样可以优化操作数的访问。正如前面讨论所示，每个HLL指令都有几次操作数访问，并且传送（赋值）语句在程序中占有很高的份额。这些再结合局部性和标量访问的主导性，表明可以通过采取更多寄存器访问的方式，来降低内存访问次数，从而提高性能。由于这些访问的局部性，一个可扩展的寄存器组看起来是符合实际的。

其次，要精心谨慎地设计指令流水线。由于条件分支和过程调用指令的高比例，一个过于简单的指令流水线将是低效的。因为它本身可能表现出大量的指令被预取但却永不执行。

最后，给出了一个由高性能原语组成的指令集。指令应该具有可预测的开销(以执行时间、代码大小度量，并且越来越多地以能量耗散度量)，并与高性能实现一致(与可预测的执行时间开销相协调)。

## 15.2 大寄存器组方案的使用

15.1 节概述的研究成果指出了对操作数快速存取的要求。我们已经看到，在HLL程序中有大量的赋值语句，其中多数是简单的A←B 的形式。还有，每个HLL语句都有一定数量的操作数访问。若再考虑到大多数访问的是局部标量，则侧重于寄存器访问应是推荐使用的。

采用寄存器的理由是，寄存器是比主存和cache还要快的最快可用存储装置。寄存器组从物理上讲是小的，通常是与ALU和控制器在同一芯片上，并且它们使用比主存和cache地址还要短的地址。于是，需要一种策略能使最频繁访问的操作数保持在寄存器中，并减少“寄存器-存储器”操作。

有两种基本途径可实现这个目标，一种是基于软件，另一种是基于硬件。软件方法是依赖编译器是寄存器的使用率最大化。编译器将试图把寄存器分配给那些在一给定时间期内使用最多的变量。这种方法要求使用复杂的程序分析算法。硬件方法是简单地装备更多的寄存器，是的更多的变量能更长时间地保持在寄存器中。

本届将讨论硬件方法，这种方法是Berkeley RISC小组首先提出的 [ PATT82a ]，用于最初的RISC商业产品Pyramid中[RRAGA83]，如今也用于流行的SPARC架构中。

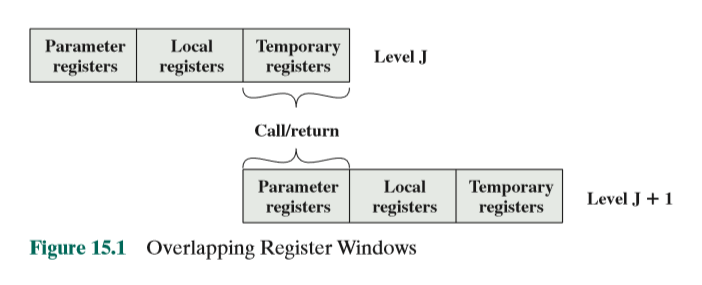
### 15.2.1 寄存器窗口

就表面上来判断，使用一大组寄存器应能减少对存储器访问的需求。因此，设计的任务就是以这样的方式组织寄存器来实现这个目标。

因为大多数操作数引用是局部标量，一种明显的方法是使用寄存器来保存它们，或许再将少量寄存器保留给全局变量。问题是这个“局部的”定义是随着每次过程调用和返回而改变的。每次调用时，寄存器中的局部变量必须被送到存储器保存，以使这些寄存器能由调用程序再次使用。而且，还需要传送过程调用的参数。返回时，父程序的变量必须恢复（加载回寄存器），并且结果也要返回到父程序。

解决方案基于15.1节中报告的另外两个结论。第一，一个典型的过程只使用少数传送参数和局部变量（表15-4）。第二，过程调用的深度仅限定在一个相对窄的范围内（图4-21）。为利用这些性质，使用了多个小寄存器组，每个小组分配给一个不同的过程。过程调用时自动地切换处理器以使用不同的但大小固定的寄存器窗口，而不再在存储器保存寄存器内容。相邻过程的窗口是重叠的，以允许参数传递。

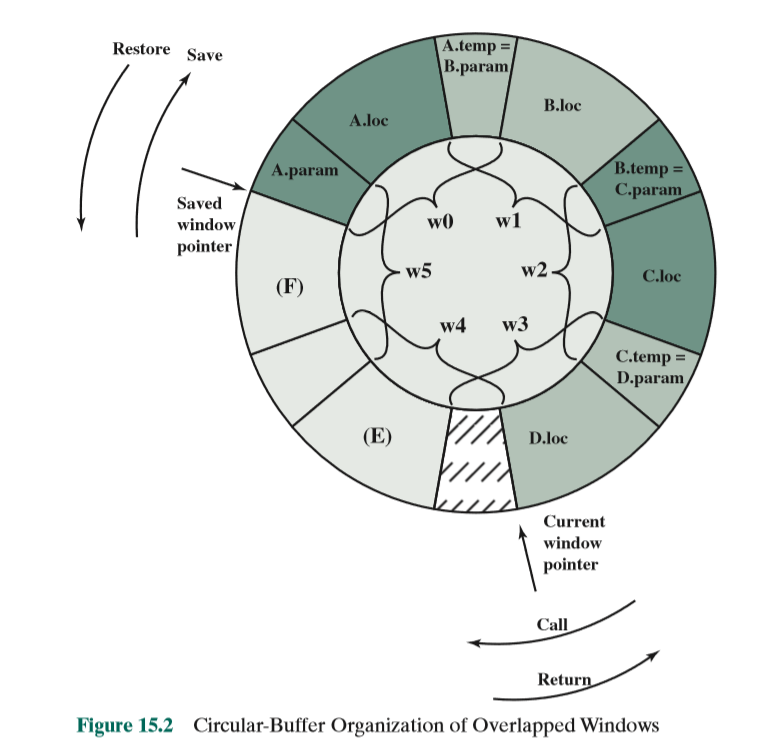
这个概念如图15-1所示。任何时刻，只有一个寄存器窗口是可见和可寻址的，就像它是唯一的一组寄存器一样（例如，地址0到N-1）。窗口分为三个固定大小区域。参数寄存器用来保存从调用当前过程的过程向下传递的参数和将被返回的结果。局部寄存器用于局部变量，这由编译器指派。临时寄存器用于当前过程与下一级过程（被当前过程调用的过程）交换参数和结果。某一级的临时寄存器与下一级的参数寄存器是物理同一的，这种重叠准许不用实际移动数据就能传递参数。记住，除了重叠的情况之外，两个不同级的寄存器窗口在物理上是完全不同的。也就是说，第J级的参数和局部寄存器与第J + 1级的局部和临时寄存器是不相交的。



为管理任何可能的调用和返回的模式，**寄存器窗口**的数目应该是不受限制的。不过这是不可能的，替代的方法是，寄存器窗口只用于保持少数最近过程的调用。 更早的过程调用必须保存到存储器中，当嵌套深度减少时再恢复。于是，寄存器组的实际组织是一个由重叠窗口组成的环形缓冲器。这种方法的两个值得一提的例子是Sun的SPARC架构，15.7节将有其描述，以及Intel公司Itanium处理器中使用的IA-64架构。

图15-2说明了这种环形组织形式，它描述的是一个6窗口的环形缓冲器。缓冲器填充深度为4（A调用B; B调用C; C调用D），而过程D处于活动状态。当前窗口指针（current-window pointer, CWP）指向当前活动过程的窗口。机器指令的寄存器引用是一个对此指针的位移，以此来确定实际使用的物理寄存器。 保存窗口指针（saved-window pointer, SWP）标识最近保存在存储器的窗口。如果当前过程D又调用过程E，则E的初始参数放在D窗口的临时寄存器中（图中w3和w4的重叠部分），CWP前进一个窗口。

如果过程E又调用过程F，则以目前的缓冲区状态，此调用不能立即执行。这是因为F的窗口重叠了A的窗口。如果F开始对其临时寄存器装入数据，准备一个调用，就会改写A的参数寄存器（A.in）。因此，当CWP递增（模6）以使其等于SWP时，一个中断就会发生，并保存A的窗口。保存时只需要保存窗口的前两部分（A.in和A.loc）。然后，SWP递增，现在可以调用对F过程了。返回时也会发生类似的中断。例如，在F过程完成之后逐级地返回，当B返回到A时，CWP递减变成等于SWP。这将引起中断，导致A窗口恢复。



由此可见，N个窗口的寄存器组仅能用于N-1个过程的调用。N值不需要很大，如附录4A中所述，[TAMI83]研究报告指出仅有1％的过程调用和返回需要8个窗口。Berkeley RISC计算机使用8个窗口，每个窗口有16个寄存器。Pyramid计算机采用16个窗口，每个窗口有32个寄存器。

### 15.2.2 全局变量

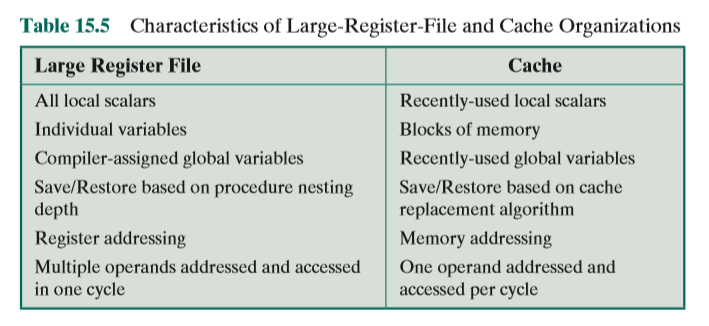
刚才介绍的窗口策略为在寄存器中存储局部标量变量提供了一种有效的组织形式。然而，这种策略没有解决存储全局变量的需求。全局变量由多个过程所使用，解决它有两种方法。首先，由编译器为高级程序设计语言（HLL）中声明的全局变量指派存储器位置，所有引用这些变量的机器指令都将使用存储器引用的操作数。无论从硬件还是软件（编译器）的角度来看，这种方法都是直截了当的。但是，对于频繁访问的全局变量而言，这种策略是低效的。

替代的方法是，CPU中包含有一组全局寄存器。这些寄存器的数量是固定的，并可供所有过程使用。一种统一编号的方法能用来简化指令格式。例如，寄存器引用号0~7指的是唯一一组全局寄存器，对寄存器8~31的访问指的是当前窗口中的具体寄存器。对于分立的寄存器寻址而言，这会增加硬件负担。此外，链接器必须决定哪些全局变量应分配给全局寄存器。

### 15.2.3 大寄存器组与高速缓存的对比

组织成窗口的寄存器组，充当一个小的快速缓冲器，用于保存可能多次使用的所有变量的一个子集。 从这个角度来看，寄存器组的作用很像一个高速缓冲存储器，尽管其速度要快得多。由此产生的问题是，使用cache和小型的传统寄存器组，哪种更简单、更好。

表15-5比较了两种方法的特征。基于窗口的寄存器组保存最近N-1个过程调用的所有局部标量变量（除少有的窗口上溢情况之外）。cache是有选择地保持最近使用过的标量变量。寄存器组节省了时间，因为它保留了所有局部标量变量。而cache可以更有效地利用空间，因为它能对动态变化的情况做出反应。此外，cache通常是将所有的存储器调用，包括指令和其他数据类型一样地对待。于是，使用cache做到其他方面的节省是可能的，而寄存器组则不行。



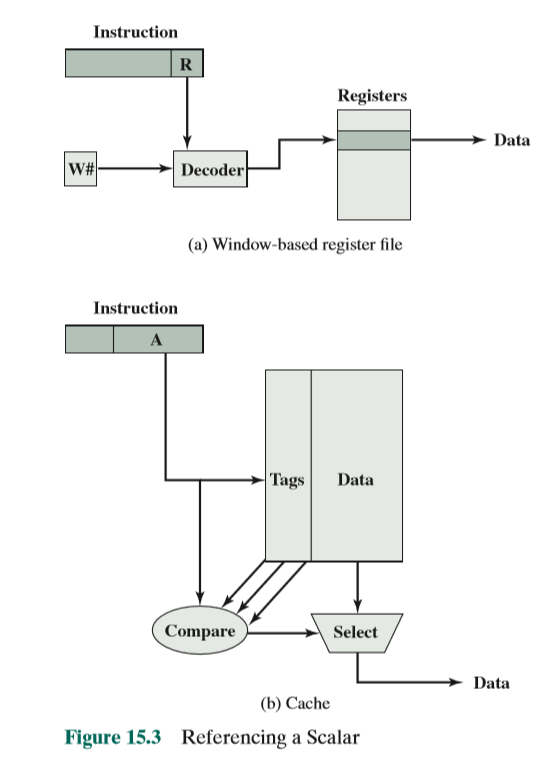
寄存器组在空间利用方面可能比较低效，因为并非所有过程都使用分配给它们的全部窗口空间。另外，cache承受另一种的低效：数据是成块读入cache的。寄存器组仅包含有用的变量，而cache读入一个数据块，但其中部分甚至更多的数据将不被使用。

cache能处理全局变量和局部变量。通常有很多全局标量，但其中只有少数被频繁使用[KATE83]。cache将动态地发现这些变量并保持它们。如果基于窗口的寄存器组补充有全局寄存器，则它也能保持某些全局标量。然而，当程序模块单独编译时，编译器不可能将全局值分配给寄存器；链接器必须执行此任务。

使用寄存器组，寄存器和存储器之间的数据传送由过程嵌套深度所确定。因为这个深度通常是在一个窄的范围内摆动，所以存储器的使用相对不太频繁。大多数cache是一种组关联结构，组的容量较小。因此，存在其他数据或指令可能会排挤走那些要频繁使用的变量的危险。

讨论至此，在大的基于窗口的寄存器组与cache之间的选择仍不明确。然而，有一个特征能说明寄存器方法占有明显优势——基于cache的系统是明显较慢的。这种区别表现在两种方法的寻址开销总量上。

图15-3说明了这种区别。为访问基于窗口寄存器组中的局部标量，要使用一个“虚拟”寄存器号和一个窗口号。这些通过一个相对简单的译码器来选择某一个具体的寄存器。要访问cache存储器中的一个位置，必须生成全宽度的地址。这种操作的复杂性取决于寻址方式。在一个组关联cache中，地址的一部分用于读取等同于组长度的几个字和标记（tag）。地址的另一部分用于与标签进行比较，以选择一个所读的字。应该清楚的是，即使cache与寄存器组一样快，cache的存取时间也要更长。因此，从性能的角度来看，基于窗口的寄存器组对于本地标量来说是更优化的。通过加入专门的指令cache，可以实现进一步的性能改进。



## 15.3 基于编译器的寄存器优化

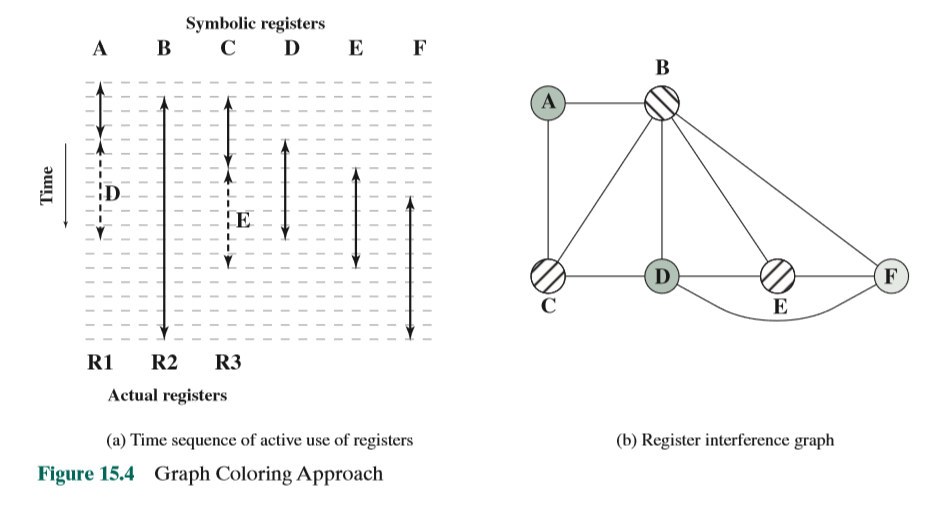
我们现在假设目标RISC机器上只有少量寄存器可用（如16~32个）。在这种情况下，优化寄存器的使用是编译器的责任。当然，用高级语言编写的程序没有对寄存器的显式引用（尽管C语言有register关键字）。相反，程序中的量是以符号来表示的。编译器的目标就是，在寄存器而不是主存储器中保持操作数尽可能多的计算，并减少装载和保存操作。

通常，所采用的方法如下所述。准备驻留在寄存器中的每个程序量先被指派到一个符号的或虚拟的寄存器，然后编译器将这些未限定数量的符号寄存器映射到固定数量的真实寄存器上。那些使用不重叠的符号寄存器可以共享相同的真实寄存器。若在程序具体运行的某个期间，需要处理的量多于真实寄存器数目，则某些量被指派到存储器位置上。装载和保存指令能把要计算的两暂时放置到寄存器中。

优化任务的本质是，判定在程序的任何给定时间点，什么样的量应指派到寄存器中。在RISC编译器中普遍使用一种称为图着色（graph coloring）的技术，这是一种由拓扑学借用过来的技术[CHAI82，CHOW86，COUT86，CHOW90]。

图着色的做法是这样的。对于一个由结点和边组成的给定图，为结点指定颜色，使得相邻结点不同色，并使得颜色数量最少。此问题以如下方式转换成编译器问题。首先，分析程序并构成一个寄存器相关图。图的结点是符号寄存器，若两个符号寄存器同时“生存”于同一程序段，则相应的两个结点用一条边连接起来以指示它们相关。尝试用n种颜色给图着色，其中n是真实寄存器的数量。使用相同颜色的结点可以分配给同一个寄存器。如果此过程不能完全成功，则这些不能着色的结点必须放入存储器中，并且必须使用装载和保存操作给它们开辟寄存器空间。

图15-4是该过程的一个简单示例。假定程序有6个符号寄存器将被编译到3个实际寄存器上。图15-4a表示每个符号寄存器有效使用的时间顺序。虚水平线表示连续的指令执行。图15-4b表示寄存器相关图（使用阴影和条纹代替颜色）。这里指出了一种使用三种颜色给图着色的可能方法。因为符号寄存器A和D不相干，所以编译可以将这两者指派到物理寄存器R1上。类似地，符号寄存器C和E可以指派到寄存器R3上。一个符号寄存器F未能着色，必须使用装载和保存操作来处理。



通常，在使用大量的寄存器和基于编译器的寄存器优化之间有一个权衡考虑的问题。例如，[BRAD91a]是在一个具有类似于Motorola 88000和MIPS R2000特色的RISC结构的模型机上所做的研究报告。研究人员选取了不同的寄存器数目（16~128），既考虑到所有寄存器都作为通用寄存器使用，也考虑到将寄存器分成整数寄存器和浮点寄存器的使用。他们的研究表明，如果只有相当简单的寄存器优化，那么使用超过64个寄存器也没什么好处。使用相当复杂的寄存器优化技术，当使用多于32个寄存器时，带来的性能改善也是不明显的。最后，他们注意到，只有少量的寄存器（例如16个）时，具有共享寄存器组织的机器比具有分立寄存器组织的机器执行得更快。从[HUGU91]可以得出类似的结论，该报告主要关注使用少量寄存器的优化问题，而不是将大寄存器组的使用与优化进行比较。

## 15.4 精简指令集体系结构

在本节中，我们将介绍精简指令集架构的一般特征和及其发展的原因。具体示例将在本章后面介绍。首先，我们讨论当代复杂指令集架构的发展原因。

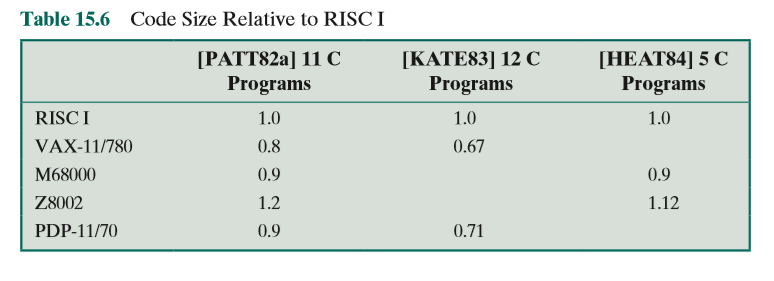
### 15.4.1 采用CISC的理由

我们已指出过朝更丰富指令集发展的趋势，其中包括更多的指令和更复杂的指令。有两个基本理由推动了这一趋势：要求简化编译器和改善性能。这两个理由之下的根本性原因是，大部分程序员已转移到高级语言（HLL）上，厂商试图设计能对HLL提供更好支持的机器。

本章并不是说CISC设计人员选错了方向。的确，技术还在发展，处理器架构存在着广泛的类型而非简单的两种类别，因此要做一个。 因此想做一个黑白分明的评定是不太可能的。于是，下面的解释仅仅是为了指出CISC方法的某些潜在缺陷，并提供对RISC发展动因的一些理解。

采用CISC的第一个理由——简化编译器，看起来是很明了的，但事实并非如此。编译器编写人员的任务是构建一个编译器，从而为HLL程序生成良好（快速，小巧，或两者兼有）的机器指令序列（即编译器在上下文中查看各个HLL语句）。若有类似于HLL语句的机器指令，那么任务就简单多了。但RISC研究人员对这个理由提出了异议（[HENN82]，[RADI83]，[PATT82b]）。他们发现复杂指令通常难以利用，因为编译器必须找到严格满足限制的情况。像优化生成的代码，以减小代码长度，减少指令执行数目和增强流水这样的任务，使用复杂指令集也是非常困难的。作为这个观点的证据，本章前面所引用的一些研究报告曾指出，编译后程序中的大多数指令都是相当简单的。

前面提到的采用CISC的另一个理由是，CISC可生成更小、更快的程序。我们考察这个主张的两个方面：程序会更小并且执行得更快。



小程序有两个优点。由于程序占用的内存较少，因此可以节省资源。但今天的存储器价格已经如此低，这个潜在的优点不再令人信服。 更重要的是，较小的程序能改善性能，这表现在三个方面：首先，较少的指令意味着待取的指令字节较少；其次，在内存分页环境下，较小的程序占用较少的页，减少了缺页中断；第三，更多的指令适合cache。

这种推理的问题在于，CISC程序小于相应的RISC程序，然而这一点远不像看上去那么肯定。多数情况下，以符号机器指令表示的CISC程序是会“短”些（即较少的指令），但它们占用的存储器位数可能不会明显更“小”。表15-6列出了来自三项研究的结果，比较了几类机器上编译后C程序的大小，包括精简指令集架构的RISC I。注意，CISC比RISC没有或只有少许的节省。另一点也是值得注意的，VAX具有比PDP-11复杂得多的CISC架构，然而与后者相比节省的成本非常低。这些结果也得到了IBM研究人员[RADI83]的证实，他们发现IBM 801（一台RISC机器）生成的代码是IBM S/370上代码大小的0.9倍。 该研究使用了一套PL/I程序。

有几个理由可以解释这些令人惊奇的结果。我们曾指出过，CISC上的编译器有偏爱简单指令的倾向，结果使得复杂指令所提供的简洁性很少能发挥作用。此外，CISC上有更多指令，因此需要更长的操作码，从而产生更长的指令。最后，RISC强调寄存器而不是存储器的访问，前者需要指令的位数更少。最后一种效应的例子稍后进行讨论。

因此，期望CISC能产生较小的程序并带有其他优点，是不太现实的。增加指令集复杂性的第二个动因是它的指令执行可能会更快。看起来这个观点是言之有理的，一个复杂的HLL操作，作为一条机器指令将会比作为一串更原始的指令执行得更快。但是，因为实际情况是偏向使用较为简单指令，因此上述想法不见得成立。为适应丰富的指令集，整个控制器必须做得更复杂，而且微程序控制存储也必须做得更大。不论哪种因素都增加了简单指令的执行时间。

实际上，一些研究人员已经发现，加速复杂功能的执行不在于复杂的机器指令是多么强有力，而在于它们驻留在高速控制存储中[RADI83]。该控制存储实际上起到指令cache的作用。因此，硬件结构研究者面临的任务便是，确定什么样的子程序或函数使用得最频繁，然后将它们指派到控制存储中，通过微代码实现它们。然而结果并不那么令人鼓舞。于是，在S/390系统上，诸如翻译（translate）和扩展精度的浮点除法（Extended-Precision-Floating-Point-Divide）之类的指令驻留在高速存储中，而涉及建立过程调用或初始中断处理程序这样重要的指令序列反而放在较慢的主存中。

于是，朝更加复杂指令集方向发展是否合适，远不是那么清楚，这导致了几个研究组朝相反的方向探索。

### 15.4.2 精简指令集体系结构特征

虽然精简指令集结构可能采取不同的方法，但某些特征对它们都是共同的：

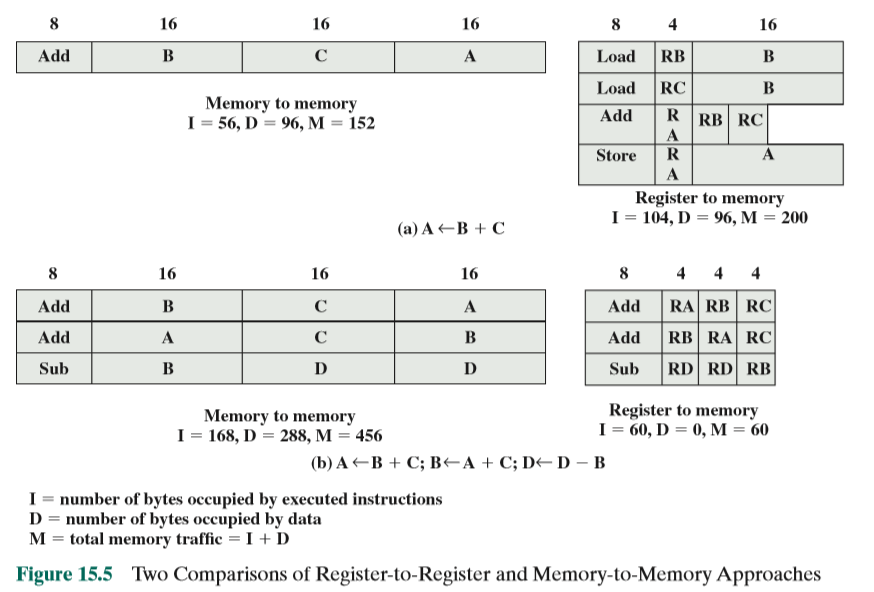
* 每周期一条指令
* 寄存器到寄存器的操作
* 简单的寻址方式
* 简单的指令格式

下面我们简要介绍这些特征，稍后再给出具体的例子。

第一个特性是，**每机器周期一个机器指令**。机器周期（machine cycle）定义为由寄存器取两个操作数，完成一个ALU操作，然后将结写入寄存器中所用的时间。因此，RISC机器指令不会比CISC机器上的微指令复杂，执行大约也是一样快（在第四部分中讨论过）。简单的单周期指令很少或没有对微代码的需求，机器指令能以硬布线方式实现。这样的指令应当比其他机器上的类似指令执行得更快，因为在指令执行期间它不必去访问微程序控制存储器。

第二个特征是，大多数操作应是**寄存器到寄存器**的，只以简单的LOAD和STORE操作访问存储器。这个设计功能简化了指令集，进而也简化了控制器。例如，一个RISC指令集可以仅包括一条或两条ADD指令（如整数加、带进位加），而VAX有25种不同的ADD指令。这种结构的另一个好处是：它鼓励寄存器的优化使用，使频繁存取的操作数保留在高速存储中。

这种寄存器到寄存器操作的强调，对于RISC设计是特别值得注意的。虽然现代CISC机器也提供这样的指令，但它们还包括存储器到存储器和混合的寄存器/存储器操作。对这些不同设计方法的比较是在20世纪70年代完成的，在RISC出现之前。图15-5a说明了所采取的方法。评价原型结构只在于比较程序的大小和传输的存储器位数。这样的结果曾使一位研究人员建议，未来的架构应不含有任何寄存器[MYER78]。人们奇怪他为什么这么想，那个时候（1978年）RISC机器Pyramid已经面市，它含有不少于528个寄存器！



这些研究遗漏的是，对少量局部标量的频繁访问，和以大量的寄存器或优化的编译器能使大多数操作数长时间地保持在寄存器中。于是，图15-5b可能是更公平的比较。

第三个特征是使用简单的寻址方式。几乎所有RISC指令都使用简单的寄存器寻址方式。其他几种寻址方式，如偏移寻址和PC相对寻址，也可能会包括进来。其他更为复杂的寻址方式可由这些简单方式用软件来合成。同样，该设计特征简化了指令集和控制器。

最后一个共同特征是使用简单的指令格式，而且通常仅使用一种或少数几种格式。指令长度固定并且在字边界上对齐。字段位置，特别是操作码字段位置是固定的。这个设计特点有多个优点。对于固定的字段，操作码的译码和寄存器操作数的访问能同时出现。简化了格式也就简化了控制器。因为以字长单位来取指令和数据，取指令也就被优化了。这还意味着，单一指令不会跨越内存分页的边界。

将这些特征综合在一起进行评估，就能确定RISC方法的潜在优势。这也有相当数量的“间接证据”。首先，能开发出更有效的优化编译器。利用更原始的指令，对于无循环的传送功能、有效地重组代码、最大化寄存器的使用等，都会有更多的机会，甚至能在编译时求解复杂指令的作用。例如，S/390移动字符（move characters, MVC）指令能将字符串从一个位置传送到另一位置上。每当执行该指令时，传送将取决于串的长度、是否或在什么方向上位置有重叠，以及排列的特征是什么。在大多数情况下，这些在编译时都将是已知的。于是，编译器能为这种操作生成一个优化的原始指令序列。

其次，前面已指出，编译器生成的大多数指令从任何方面讲都是相对简单的指令。专门为这些指令来构造，控制器看起来是有道理的，并且很少或根本不使用微代码来执行它们，要比相应的CISC快得多。

另外，与指令流水的使用有关。RISC研究者们发现，精简指令集能更有效地应用指令流水技术。我们即将更详细地考察这一点。

最后一点，也是不太重要的一点是，RISC处理器能更好地响应中断，因为中断是相当于在基本操作之间检查的。使用复杂指令的结构要么将中断限定在指令边界上，要么定义专门的中断点，并为重启动一条指令实现一种结构。

精简指令集结构的改善性能很强，但人们仍然有理由为CISC争辩。已有几个这方面的研究，但不是在技术和功能方面可比的机器上进行的。而且，大多数的研究还没有将精简指令集的效应与大寄存器组的效应分开。然而，上述的“间接证据”仍是具有启发性的。

### 15.4.3 CISC与RISC特征对比

在对RISC机器的最初热情之后，人们越来越认识到：

（1）RISC设计包括某些CISC特色会有好处；

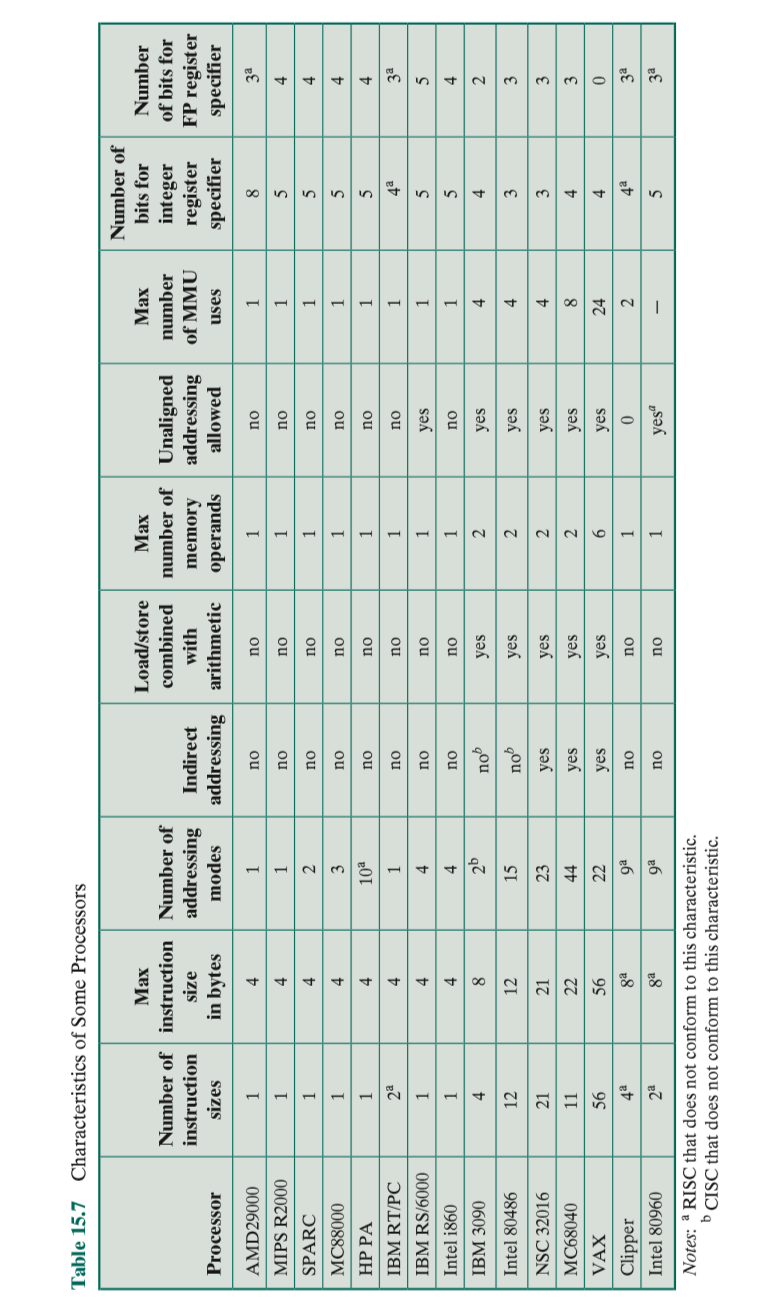
（2）CISC设计包括某些RISC特色也会有益。

结果是，最近的RISC设计，以PowerPC为代表，不再是“纯”RISC了。而最近的CISC设计以Pentium II和其后的型号为代表，也结合了某些RISC特性。

[MASH95]给出了一个令人感兴趣的比较，提供了对这些观点的某些见识。表15-7列出了几种处理器，并对几个特征进行了比较。作为这个比较的目的，可以列出如下一些被认为是典型的RISC特征：

（1）单一的指令长度；

（2）经典的指令长度是4字节；



（3）较少的寻址方式，一般少于5种。不过这个数目难以限定。表中未计入寄存器和立即数的方式，而带有不同位移大小的不同格式却分别予以统计了。

（4）无间接寻址。间接寻址要求先进行一次存储器访问来得到操作数的存储器地址。

（5）装载/保存操作不与算术操作混在一起（例如，由存储器加或加到存储器）。

（6）每条指令不会有多于要给存储器操作数。

（7）对装载/保存操作，不支持数据的任意对齐。

（8）对指令中的数据地址，最大化存储管理单元（Memory Management Unit, MMU）的使用。

（9）整数寄存器指定符的位数等于5或更多。这意味着，至少有32个整数寄存器能被显式地引用。

（10）浮点寄存器指定符的位数等于4或更多。这意味着，至少有16个浮点寄存器也能被显式地引用。

第1至3项指出了指令解码的复杂性；第4至8项表明流水线操作的难易程度，特别是在存在虚拟内存要求的情况下；第9项和第10项与能否充分利用编译器有关。

在表中，前八个处理器显然是RISC架构，后面的五个处理器显然是CISC架构，而最后两个处理器通常被认为是RISC架构，但实际上具有许多CISC的特性。

## 15.5 RISC 流水线技术

### 15.5.1 使用规整指令的流水线技术

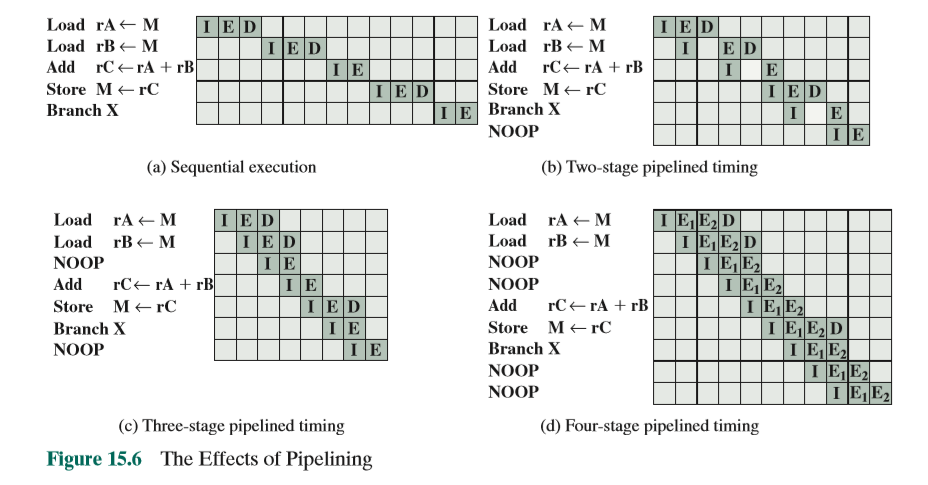
正如12.4节中所讨论过的，指令流水线技术经常被使用以提高性能。让我们再在RISC结构前提下考虑这个问题。大多数指令是寄存器到寄存器的，并且指令周期有如下两个阶段：

* I: 取指令。
* E：执行。带寄存器输入和输出，完成一个ALU操作。

对于装载和保存操作，需要三个阶段。

* I：取指令。
* E：执行（计算存储器地址）
* D：存储（寄存器到存储器或存储器到寄存器操作）

图15-6a描述了不使用流水线技术的一个指令序列的操作顺序。很清楚，这个过程有所浪费。即使一个很简单的流水线技术都能实质性地改善其性能。图15-6b表示一种两段流水处理策略，在此流水线处理方式中，两个不同指令的I和E同时完成。流水线中有两个阶段是指令获取阶段和执行指令的执行/存储阶段，包括“寄存器-存储器”和“存储器-寄存器”操作。因此，我们看到第二条指令的指令获取阶段可以与执行/存储阶段的第一部分并行执行。但是，必须把第二条指令的执行/存储阶段延迟到第一条指令清除流水线的第二个阶段。这种策略能产生串行策略两倍的执行速率。有两个问题妨碍了这种最大速率的达到。第一个问题是，如果假定使用单端口存储器，那么每个阶段只准一次存储器访问，这就要求在某些指令的执行中插入等待状态。其次，分支指令中断顺序执行流程。为了以最小的电路来适应这种情况，编译器或汇编器可以将NOOP指令插入到指令流中。第二个问题是，一条分支指令能打断顺序的执行流。为了以尽量少的电路来应对这种情况，可通过编译器或汇编器将NOOP指令插入指令流中。



通过允许每个阶段有两次存储器访问，能进一步改善流水线性能。这产生了如图15-6c所示的序列。现在，能重叠执行多达3条指令，改善的比例最大可达到3。同样，分支指令使加速不能达到最大允许值。还有，注意数据相关性也有影响。若一个指令需要某个操作数，而该操作数会由前面指令所更新，则需要一个延迟。同样，这能用插入NOOP来实现。

至此，若3个阶段有大致相等的期间，所讨论的流水线就能很好地工作。因为E阶段通常涉及一个ALU操作，它可能会更长一些。这种情况下，我们能将它分成两个子阶段。

* ​：寄存器组读。
* ：ALU操作和寄存器写。

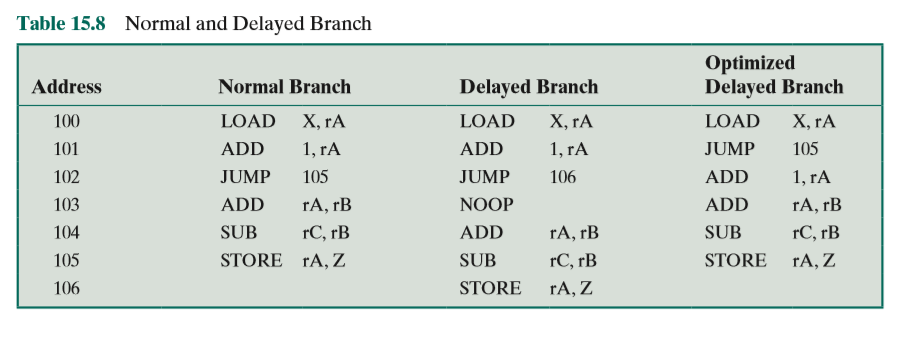
由于RISC指令集的简单性和规整性，设计3个或4个阶段的流水线很容易。图15-6d表示使用4段流水的结果。多达4条指令能同时进行，最大可能的加速比是4。请再次注意，考虑到因数据和分支的延迟而插入NOOP指令。

### 15.5.2 流水线的优化

由于RISC指令集的简单性和规整性，硬件设计人员更容易实现简单、快速的流水线。指令的执行时间没有多少变动，并且可以对流水线进行改进以反映这些变动。然而，我们已看到，数据的相关性和分支指令会降低整体的执行速率。

#### 1. 延迟分支

为抵消这些相关性带来的性能损失，开发人员采用了代码重组（code reorganization）技术。提高流水线效率的一种方式是延迟分支（delayed branch）。它利用了分支指令直到下面一条指令之后才产生影响这一特点，在分支指令之后安排一条有用指令来替代仅为延迟的空操作。分支指令之后的指令位置被称为延迟槽（delay slot）。这种奇特的处理可由表15-8说明。表中的第一列，我们看到的是以符号指令表述的正常的机器语言程序。在102处的分支指令执行之后，将要执行的下一条指令在105处。为规范流水线，在分支指令之后插入了一条NOOP指令。然而，若将101处和102处的指令进行交换，则能实现性能的提升。



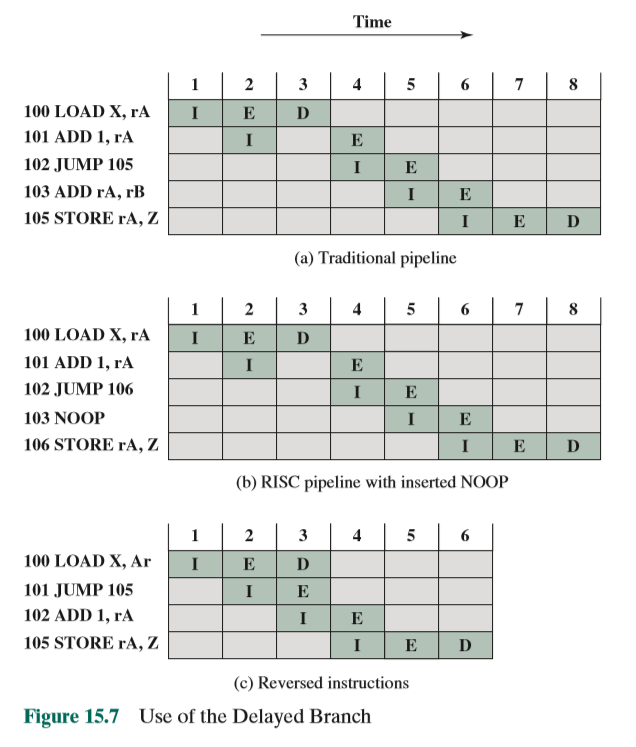


图15-7表示了上述处理的结果。图15-7a展示的是传统流水线方案，这种方案已在第14章讨论过（参见图14-11和图14-12)。在 时间4取来JUMP指令。在时间5，此JUMP指令在执行，与此同时指令103（ADD指令）已被取来。因为JUMP的出现，修改了程序计数器，流水线必须清除指令103。在时间6，JUMP的目标，即指令105被取来。图15-7b展示的是典型RISC组织来处理同样的流水线，时序相同。只不过由于插入NOOP指令， 我们不再需要清除流水线的专门电路了；此NOOP简单地被执行，无任何影响。图15-7c展示的是延迟分支方法的使用。此JUMP指令在时间2取来，在ADD指令之前，ADD指令在时间3取来。然而注意，在执行JUMP指令之前取出ADD指令有可能改变程序计数器。于是在时间4，ADD指令在执行，同时指令105被取来。这样，既保持程序的原语义，又使指令的总执行时间减少了两个时钟周期。

对于无条件跳转、调用和返回，都能成功地进行这种交换。然而，不能盲目地将其施加到条件分支指令上。若分支所测试的条件会被前面这条指令所修改，则编译器应避开这种交换，而插入NOOP指令。否则，编译器能在分支指令之后插入有用指令。以Berkeley RISC和IBM 801系统二者的经验来看，大多数的条件分支指令都能以这种交换方式得到优化（[PATT82a]，[RADI83]）。

#### 2. 延迟加载

延迟加载可用于LOAD指令。在LOAD指令上，作为加载目标的寄存器被处理器锁定。处理器随后继续执行指令流，直到其执行到需要该寄存器的指令，此时处理器空闲直到加载完成。若编译器可以重新排列指令，则在加载操作处于流水线中时，仍可以完成有用的工作，从而提高效率。

#### 3. 循环展开

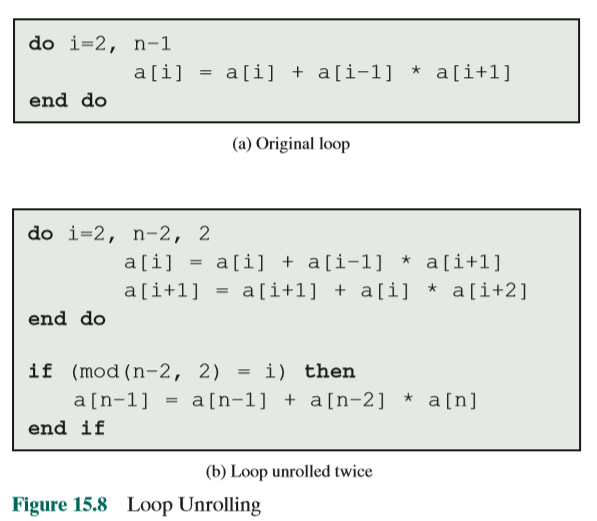
另一种提高指令并行性的编译技术是循环展开（loop unrolling）[BACO94]。它把一个循环的循环体复制若干次，其次数被称为展开因子（u），从而以步长u来执行循环，而不是步长1。

循环展开时通过以下方式来提高性能的：

* 降低循环开销。
* 通过提升流水线性能来提高指令并行性。
* 提高寄存器、数据高速缓存或页表快速缓存（TLB）的局部性。

图15-8用例子描述了这3种性能改进的效果。循环开销降低为原来的一半，因为在循环结尾的测试和分支之前，一次执行了两个迭代。指令并行性的提高是因为第二个赋值的执行可以与第一个赋值的结果保存和循环变量的更新同时进行。如果数组元素是被赋值到寄存器中，寄存器的局部性可获得提升，因为a[i]a[i]a[i]和a[i+1]a[i+1]a[i+1]在展开后的循环体中被用了两次，把每次循环迭代中的内存装载次数从3次降低到了2次。

最后应当指出，指令流水线的设计不应与其他适用于系统的优化技术隔离开进行。例如，[BRAD91b]指出，流水的指令调度策略应与寄存器的动态分配一起考虑，以提高效率。



## 15.6 MIPS R4000

最早商品化的一种RISC芯片组是MIPS Technology公司开发的。此系统受到一个在斯坦福研制的一个实验系统（也叫MIPS）的启发[HENN84]。本节我们介绍MIPS R4000。它实质上与MIPS设计的早先产品：R2000、R3000具有相同的体系结构和指令集。最显著的不同是R4000使用的是64位而不再是32位位宽，用于所有内部和外部数据路径和地址、寄存器以及ALU。

使用64位要比32位结构有几个好处。它允许更大的地址空间：大到能使操作系统将比太字节还大的文件直接映射到虚拟存储器，使存取变得很容易。现在普遍使用的1TB或更大的磁盘驱动器，32位机器的4GB地址空间变成了一种限制。另外，64位能允许R4000处理像IEEE双精度浮点数这样的数据，以及处理字符串数据时能一次处理多达8个的字符。

R4000处理器芯片分成两个部分，一部分含有CPU，另一部分含有存储管理单元。CPU是个 很简单的结构，其设计思想是，尽可能使指令执行逻辑简单，留出空间用于增强性能的逻辑 （例如，完整的存储管理单元）。

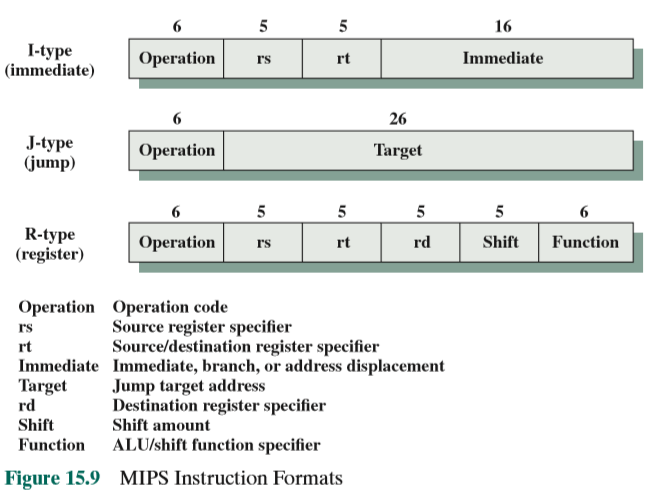
处理器支持32个64位寄存器。它还提供多达128 KB的高速cache，一半用于指令，一半用于数据。这种相对大的cache（IBM 3090提供128~256KB的cache）使系统能保持更多的程序代码和数据在处理器内，从而减轻了主存总线的负荷，也避免了对大寄存器组及其配套窗口逻辑的需求。

### 15.6.1 指令集

所有MIPS R系列指令都以单一32位字格式来编码。所有数据操作都是寄存器到寄存器，仅纯装载/保存操作有存储器访问。

R4000没使用条件码。若一条指令产生某个条件，其相应的标志存于一个通用寄存器中。这就避免了专门用于处理条件码的逻辑，因为它们影响流水线机制和编译器对指令的重排序。流水线已经实现了处理寄存器值相关的机制。而且，映射到寄存器组的条件在分配与再使川上，与存于寄存器其他值都一样可由编译器在编译时间优化。

与大多数RISC类和机器一样，R4000使用单一的32位指令长度。这既简化了取指令和译码，又简化了取指令与虚拟存储管理单元的相互作用（即指令不穿越字或页的边界）。它的三种指令格式（见图15-9）共享操作码和寄存器引用的公共格式，简化了指令译码，可在编译时间以简单指令的合成实现更复杂指令的效果。



只有最简单的和最经常使用的存储器寻址方式是以硬件实现的。所有的存储器引用由一个32位寄存器和一个16位的对此寄存器基址的偏移量组成。例如，“装载字”指令是如下形式：

lw r2,128(r3) /\*装载一个字到寄存器r2，该字是寄存器r3的值加上偏移量128

32个通用寄存器的任意一个都可被用作基址寄存器。有一个寄存器，r0，所包含的值总是0。

编译器使用多条机器指令的合成，来实现普通机器中的典型寻址方式。下面给出了一个来自[CHOW87]的例子。其中使用了lui （load upper immediate，装载上部立即数）指令，这条指令将16位立即数装入寄存器高半部，低半部全置为0。考虑如下使用一个32位立即数作为参数的汇编语言指令。

lw r2,#imm(r4) /\* 装载一个字到寄存器r2，该字的地址是寄存器r4的值加上32位的立即数偏移量#imm

上面这条指令将被编译为下列MIPS指令。

lui r1,#imm-hi /\* 装载偏移量#imm的高16位#imm-hi到寄存器r1

add r1,r1,r4 /\* 把寄存器r1中的#imm-hi与r4的值相加，结果保存回寄存器r1

lw r2,#imm-lo(r1) /\* #imm-lo是偏移量#imm的低16位

### 15.6.2 指令流水线

以其简化的指令集结构，MIPS能实现很有效的流水。考察MIPS流水线的演变情况是有指导意义的，因为它在大体上说明了RISC流水技术的改进。

最初实验RISC系统和第一代商品化的RISC处理器，实现了大约每系统时钟周期1条指令的执行速度。为改善这种性能，两类处理器：超标量和超级流水线体系结构，已发展到能每时钟周期执行多条指令。从本质上讲，超标量体系结构（super scalar architecture）复制每个流水线段， 使得流水线的同一阶段可以同时处理两条或多条指令。超级流水线体系结构（super pipelined architecture）是使用更多更细致的流水阶段。对这种更多的段，更多的指令能同时处于流水线中，从而提高并行度。

这两种方法都有限制。对超标量流水线技术，不同流水线中指令间的相关性会减慢系统。还有，为协调这些相关性也要求一些辅助逻辑。对超级流水线，指令由一个阶段传送到下一阶段的开销也会增加。

第16章将专门研究超标量体系结构。MIPS R4000是一个基于RISC的超级流水线体系结构的好例子。

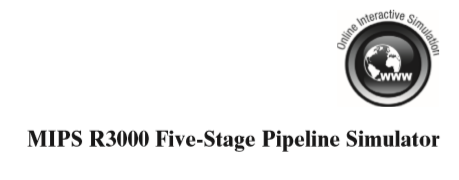
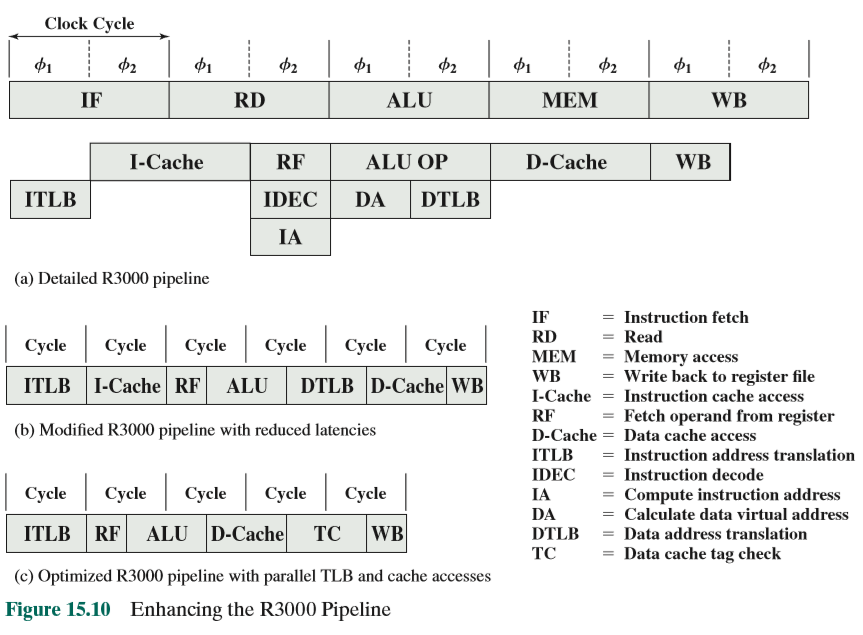


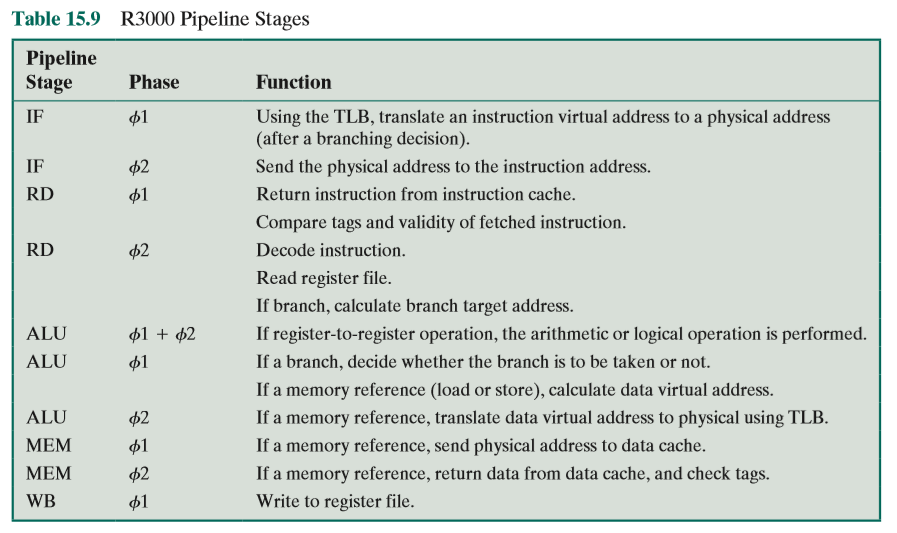
图15-10a显示了R3000的指令流水线，指令在流水线中每时钟周期前进一步。MIPS编译器能重排序指令．在70％~90％的情况下能填充分支延迟槽。所有指令都流经如下5个流水阶段：

* 取指令
* 由寄存器组取源操作数
* ALU运算或数据操作数地址生成
* 数据存储器访问
* 写回到寄存器组

正如图15-10a所显示的，这里不仅有流水线的并行性，也有单一指令执行内的并行性。 60ns的时钟周期分成两个30ns的相位。外部指令和数据的cache存取操作，每项操作都需要 60ns，主要的内部操作（OP、DA、IA）也需要同样的时间。指令译码是一个较简单操作，只要求一个30ns相位，能与同一指令的寄存器读取重叠。一个分支指令的地址计算亦与指令译码和寄存器读取重叠，于是一条分支指令i可提供指令i+2访问ICACHE的地址。类似地，指令i可装载会被指令i+1的操作紧接着使用的操作数，与此同时，一个ALU/shift的结果能直接传递给指令i+1而没有任何延迟。这种指令间的紧耦合有利于高效流水。

每个时钟周期分成的两个相位分别标为和。每个相位所完成的功能总结见表15-9。



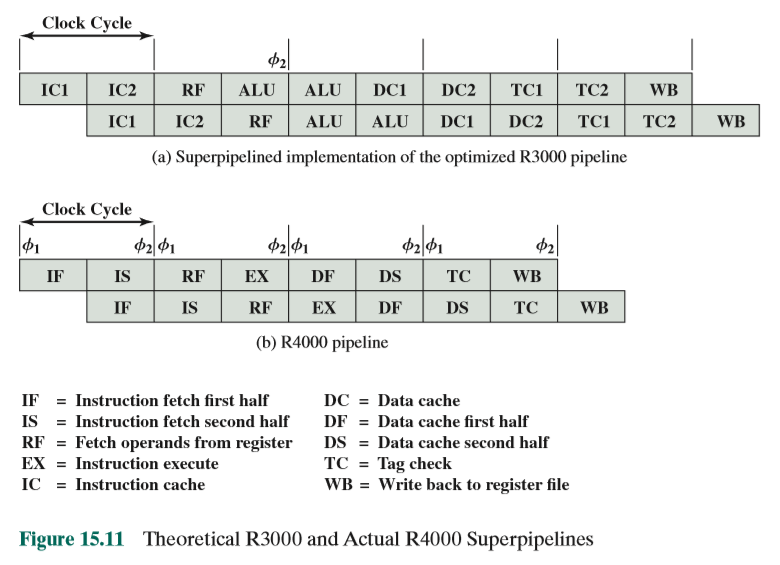


R4000比R3000又有几点技术改进。使用更先进的技术，使时钟周期缩短到原来的一半，即 30ns，寄存器的存取时间也缩短到原来的一半。另外，芯片的密度更大，指令和数据cache能集成到芯片上。在最后考察R4000之前，让我们先考虑R3000应如何修改以使用R4000技术提高性能。

图15-10b表示第一步。注意此图中的周期已是图15-10a中周期的一半长。因为指令和数据cache在处理器同一芯片上，它们的存取时间也是原来的一半长，故它们的流水阶段仍占据一个时钟周期。同样，因为寄存器组存取的加速，寄存器读和写仍占据时钟周期的一半。

因为R4000 cache在芯片上，虚拟地址到物理地址的转换会延迟cache访问。可通过虚拟地址索引的cache，从而使cache存取和地址转换的并行，以缩短延迟。图15-10c表示以这种改进而优化的R3000流水线。因为事件的密集，数据cache标记的检查放在cache存取之后的下一周期来完成。此检查确定数据是否在cache中。

在超级流水线式系统中，通过插入流水线的寄存器，每个流水阶段被细分开来，使得原有硬件在每周期可被多次使用。基本上，超流水线的每个阶段是以基本时钟频率的几倍来操作，倍数取决于超流水程度。R4000所具有的速度和密度，准许它有级别为2的超级流水线。图15-11a表示使用这种超级流水线的优化的 R3000。注意，它基本上同于图15-10c所示的动态结构。



为了进一步改进R4000，在其上设计了一个更大的专门的加法器，这使它能以两倍的速率来执行ALU操作。另外的改进允许装载和保存以两倍的速率来执行。改进后的流水线示于图15-11b。

R4000有8个流水段，意味着多达8条指令能同时在流水线中。流水以每时钟周期两段的速率向前推进。8个流水段如下所述。

* **取指令的前一半**（instruction fetch first half）：虚拟地址提交给指令cache和转换后备缓冲器（TLB）。
* **取指令的后一半**（instruction fetch second half）：指令cache送出指令和TLB生成物理地址。
* **寄存器组**（register file）：三个动作并行出现。
  + 指令被译码，并对联锁条件进行检查（即这条指令取决于前面指令的执行结果）。
  + 进行指令cache标记（tag）检查。
  + 由寄存器组读取操作数。
* **指令执行**（instruction execute）：可能出现下列三个动作之一。
  + 若指令是寄存器到寄存器的操作，ALU完成此算术或逻辑操作。
  + 若指令是装载/保存指令，计算数据的虚拟地址。
  + 若指令是分支指令，计算转移目标的虚拟地址并检查转移条件。
* **数据cache前一半**（data cache first half）：虚拟地址提交给数据cache和TLB。
* **数据cache后一半**（data cache second half）：数据cache输出数据，TLB生成物理地址。
* **标记检查**（tag check）：为装载/保存完成cache标记检查。
* **写回**（write back）：指令结果写回到寄存器组。

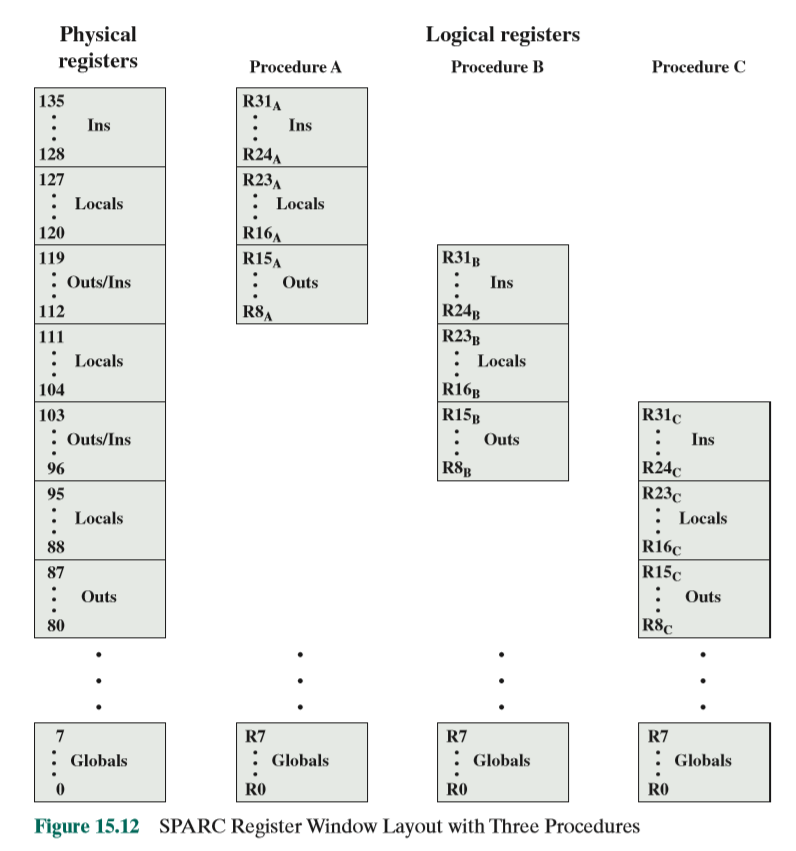
## 15.7 SPARC

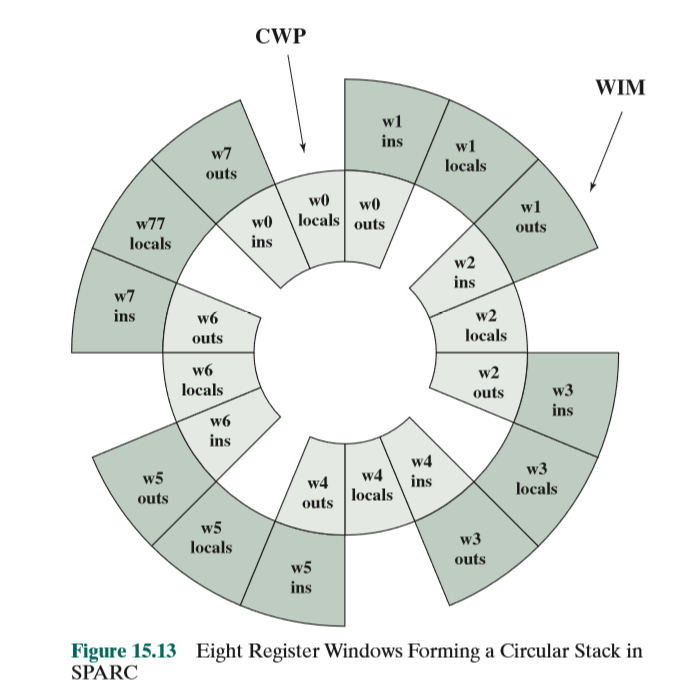
SPARC（scalable processor architecture，可扩展处理器体系结构）是指一种由Sun Microsystems 公司定义的处理器结构。Sun已开发了自己的SPARC实现，而且也许可其他厂商生产SPARC兼容机。SPARC结构的开发从Berkeley的RISC I机器得到了许多启发，它的指令集和寄存器组织也紧密基于Berkeley RISC模型之上。

### 15.7.1 SPARC寄存器组

与Berkeley RISC一样，SPARC也使用了寄存器窗口。每个窗口由24个寄存器组成；总的窗口数是2到32个，实际数目取决于具体实现。图15-12展示了一个8窗口的实现，总共使用了136个物理寄存器。正如15.2节讨论中所指出的，这是一个合理的窗口数。物理寄存器0到7，是所有过程共享的全局寄存器。每个过程见到逻辑寄存器号从0到31。逻辑寄存器24到31 标记为输入（ins），是与调用过程（父过程）共享的。逻辑寄存器8到15标记为输出（outs），是与被调用过程（子过程）共享的。这两部分与其他窗口重叠。逻辑寄存器16到23标记为局部的，是本过程使用的局部寄存器，既不与其它过程共享，也不与其他窗口重叠。再次，如同15.1节讨论中所指出的，8个寄存器用于参数传递，在大多数情况下，应该就足够了（见表15-4）。

图15-13是寄存器重叠使用的另一种视图。调用过程将要传送的参数放入它的outs寄存器中，被调用过程将这同一组物理寄存器看作它的ins寄存器。处理器维护一个指向当前执行过程窗口的指针，称为当前窗口指针（current window pointer, CWP）。CWP 位于处理器状态寄存器（PSR）中，此寄存器中还有一个窗口无效屏蔽WIM（window invalid mask），它指示哪个窗口无效。





使用SPARC的寄存器结构，过程调用通常没必要保存和恢复寄存器。因为编译器只需关心以有效方式为过程分配局部寄存器，而无需关心过程间的寄存器分配，故编译器大大简化了。

### 15.7.2 指令集

大多数SPARC指令只使用寄存器操作数。寄存器到寄存器指令有三个操作数，并能表示成：

其中， ​和​是寄存器，或者是寄存器，或者是一个13位立即数。寄存器零（​）已被硬布线为0值。这种指令形式非常适合于具有高比例的局部标量和常数的程序。

可由ALU实现操作的指令分成如下几组：

* 整数加法（带或不带进位）。
* 整数减法（带或不带借位）。
* 按位的布尔运算AND、 OR, XOR及其取反操作。
* 逻辑左移、逻辑右移和算术右移。

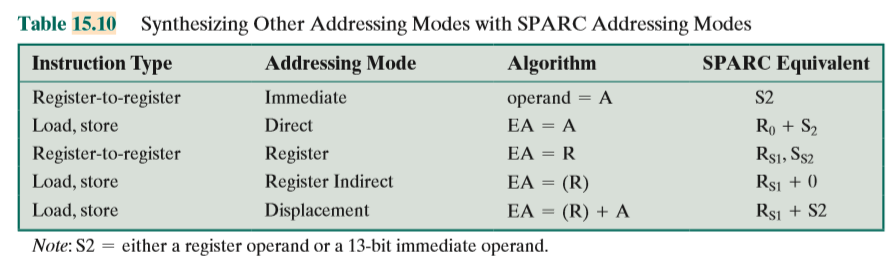
除移位指令之外，所有这些指令都可选择设置4个条件代码：零（ZERO）、负（NEGATIVE）、上溢（OVERFLOW）、进位（CARRY）。带符号整数以32位的2的补码形式表示。

只有简单的装载和保存指令访问存储器，并区分对字（32位）、双字、半字、字节的装载或保存。半字或字节的装载指令还可按有符号数和无符号数区别对待。对有符号数，符号位被扩展填充32位目的寄存器的高位。对无符号数，32位目的寄存器的高位将被0填充。

除寄存器外，唯一可用的寻址方式是“偏址”方式，即操作数的有效地址（EA）是基址和偏移量之和。基址来自寄存器，偏移量是立即数，也可能来自于寄存器，即可表示为：

或

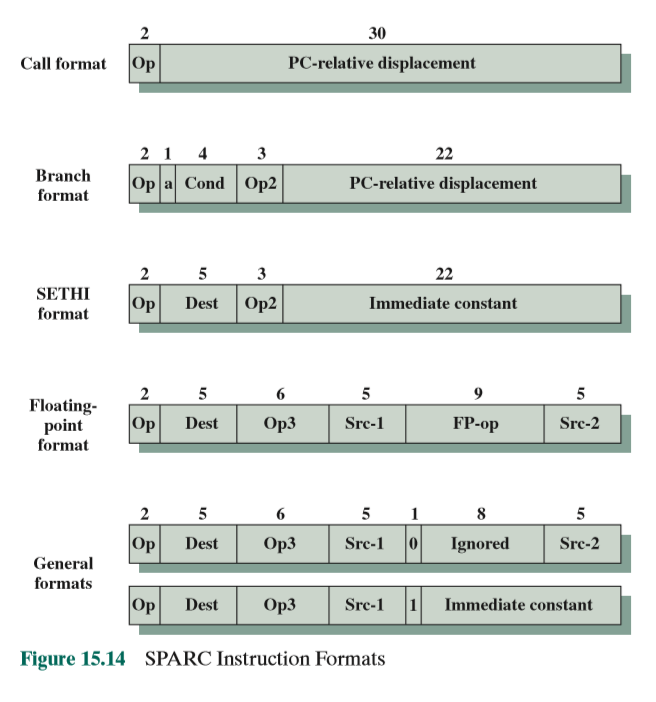
这取决于第二个操作数是立即数还是寄存器引用。为完成装载或保存，指令周期需添加一个额外相位，在第2步使用ALU完成存储器地址计算，在第3步装载或保存。这种单一寻址方式非常灵活，能综合形成其他寻址方式，如表15-10所示。



将SPARC的寻址能力与MIPS的寻址能力进行对比是有益的。MIPS使用16位偏移，SPARC 使用13位偏移。另外，MIPS不准许一个地址由两个寄存器的内容构成，而SPARC准许。

### 15.7.3 指令格式

与MIPS R4000一样，SPARC使用了一组简单的32位指令格式（见图15-14）。所有指令都以2位操作码开始，某些指令在指令格式中的其他位置还有操作码。对于Call指令，一个30位的立即数用右方加两个零位的方法扩展成32位，构成以2的补码格式表示的PC相对地址。指令对齐在32位边界上，故这种地址形式满足寻址需要了。



分支指令包括一个4位条件字段，它对应于4个标准条件码，因此可以测试这些条件的任何组合。22位PC相对地址以右方加两位0而扩展，形成24位的2的补码形式的相对地址。分支指令的一个不寻常特点是它的注销(Annul)位。若此注销位未置位时，则分支指令的直接后继指令总是被执行，不论是否发生转移。这是在许多RISC机器中都可以找到的典型延迟分支法策略，已在15.5节介绍过了（见图15-7）。但是，若注销位置位时，则仅当发生转移时才执行此分支指令之后的指令。当转移实际未发生时，处理器会注销已经取到流水线中的延迟槽指令。这个注销位很有用，因为它使编译器填充条件分支指令之后的延迟槽变得很容易。转移目标指令总放在延迟槽内，因为如果不发生转移，能自动注销此指令。采用这种办法的理由是，条件分支多半是要发生转移的。

SETHI指令是一条用于装载或保存32位值的特殊指令，这对于装载或保存地址或大的常数是一个很有用的特征。SETHI指令以它的22位立即数设置寄存器的高22位，以0填充寄存器的低10位。一条普通指令格式的指令能指定多达13位的立即数，这样的指令能用于填注寄存器剩余的低10位。装载或保存指令也可以用来实现直接寻址方式。假设要从内存位置K取一个值装入寄存器，我们可以使用如下SPRAC指令：

sethi %hi(k),%r8 ;将位置K的高22位地址装入寄存器r8

ld [%r8 + %lo(K)],%r8 ;将位置K的内容装入寄存器r8

宏%hi和%lo用于定义由位置的相应地址位组成的立即数。SETHI的这种使用类似于MIPS上的LUI指令的使用。

浮点指令格式用于浮点运算。它要求两个源寄存器和一个目的寄存器。

最后，包括装载、保存、算术和逻辑运算等所有其他指令，都使用图15-14最下方所列的两种普通指令格式之一。一种格式使用两个源寄存器和一个目的寄存器；另一种使用一个源寄存器，一个13位立即数和一个目的寄存器。

## 15.8 RISC与CISC的争论

多年来，计算机组织和体系结构朝着增加CPU复杂性方向发展：更多的指令，更多的寻址方式，更多专用寄存器等。RISC结构表示了与这种趋势背后根本原理的彻底决裂。显然，RISC 系统的出现和赞美RISC优点文章的发表，遭到了参与CISC体系结构设计人员的反对。

在评价RISC方法功过方面所做的工作可分成两类。

* **定量化**（quantitative）：试图比较采用了可对比技术的RISC和CISC机器上的程序大小及程序执行速度。
* **定性化**（qualitative）：考察诸如高级语言支持和VLSI资源的优化使用等设计出发点。

[PATT82b, HEAT84, PATT84]这些在RISC系统上所做的研究已完成定量评价的大量工作。正如已看到的，他们原则上赞同RISC方法。其他一些人考察了设计问题，对RISC路线还是抱着一些怀疑[COLW85a, FLYN87, DAVI87]。在试图进行这类比较时，存在几个问题 [SERI26]。

* 没有这样成对的RISC和CISC机器，它们在生存期、技术水平、成本、门电路的复杂性、编译器的精致性、操作系统支持等各方面都是可比较的。
* 不存在一组正式的测试程序。实际性能总是随所用测试程序不同而改变。
* 难于将硬件效应与编译器编写技巧的效应分开。
* 对RISC的分析比较，大都是在模型机上完成的而不是在商品机上。更何况作为RISC广告的大多数商品机又都具有RISC和CISC的混合特征。于是，与标榜为纯的CISC商品机（例如VAX、Pentium）进行公平比较是困难的。

定性评估几乎都是通过定义来完成的，因而是主观观点。几个研究者已将他们的注意力转到这种评估[COI, W85a, wALL85]上，但结论仍存在模糊性并遭到批驳[PATT85b]，当然， 也有反批驳[COLW85b]。

近几年来，RISC与CISC的争论在很大程度上已平息下来。这是因为已出现技术的逐渐交融。随着芯片密度和硬件速度的提高，RISC系统已变得更复杂。与此同时，在达到极限性能的努力中，CISC设计已关注的一些特点通常与RISC相联系，如增加通用寄存器数量和更加强调指令流水线设计等。

## 15.9 关键词、思考题和习题

### 关键词

|  |  |  |
| --- | --- | --- |
| complex instruction set computer (CISC): 复杂指令集计算机 | high-level language (HLL): 高级语言 | register window: 寄存器窗口 |
| delayed branch: 延迟分支 | reduced instruction set computer (RISC): 精简指令集计算机 | SPARC: 可扩展处理器体系结构 |
| delayed load: 延迟装载 | register file: 寄存器组 |  |

### 思考题

1. RISC组织的典型特征是什么？
2. 简要说明RISC机器上用于减少寄存器-存储器操作的两种基本方法。
3. 若用一个环形寄存器缓冲器来管理嵌套过程的局部变量，请描述管理全局变量的两种办法。
4. 循环展开是如何改善系统性能的？
5. 请定义术语机器周期。

### 习题

1. 考虑图4-20中的调用-返回样式，以如下窗口尺寸将出现多少上溢和下溢（它们每个都引起一个寄存器保存/回复）：

（a）12？（b）17？（c）15？

1. 在讨论图15.2时曾说过，仅窗口的前两部分需要保存或恢复。为什么没必要保存临时寄存器？
2. 我们希望确定一给定程序的执行时间，它使用15.5节讨论过的各种流水策略。令：

*N*=已执行指令数

*D*=存储器访问次数

*J*=转移指令数

对于简单的顺序策略（见图15-6a），执行时间是2*N* + *D*个阶段。求出两段、三段和四段流水的执行时间公式。

1. 重新组织图15-6d中的代码顺序以减少NOOP的数量。
2. 考虑如下的高级语言代码段：

for I in 1 ... 100 loop

S S + Q(I).VAL

end loop;

假定Q是一个32字节记录的数组，VAL字段是每个记录的前4个字节。使用x86代码，能将这程序段编译成：

MOV ECX,1 ;使用寄存器ECX，初始值为1

LP: IMUL EAX,ECX,32 ;得到位移量在EAX中

MOV EBX,Q[EAX] ;将VAL字段装入EBX

ADD S,EBX ;加到S中

INC ECX ;递增I

CMP ECX,101 ;与101比较

JNE LP ;循环得到I=100

这个程序使用了IMUL指令，它将第二个操作数乘以第三个操作数中的立即值，乘积放入第一个操作数中（参见习题10.13）。一个RISC拥护者证明，一个灵巧的编译器能取消如IMUL这样不必要的复杂指令。通过重写一个不使用IMUL指令的x86程序来提供此证明。

1. 考虑如下循环：

S:= 0;

**for** K:=1 **to** 100 **do**

S := S - K;

将这些语句翻译成通常的汇编语言，直截了当的做法可以是这样:

LD R1,0 ;S值保持在R1中

LD R2,1 ;K值保持在R2中

LP SUB R1,R1,R2 ;S:=S – K

BEQ R2,100,EXIT ;若K=100，则结束

ADD R2,R2,1 ;否则，递增K

JMP LP ;回到循环开始处

RISC机器的编译器将在这段代码中引入延迟槽，于是处理器能使用延迟分支机制。 JMP指令易于处理，因为该指令后面总是跟着一条SUB指令，因此我们可以简单地将SUB指令的副本放入JMP之后的延迟槽。 BEQ指令处理就有些困难，我们不能让代码就这样运行，否则ADD指令会执行太多次。于是，需要NOP指令。请给出使用延迟分支法的最终代码。

1. 为提高流水效率，RISC机器可将符号寄存器映射到实际寄存器，并重排指令顺序。这就提出了一个有趣的问题：这两个操作有没有先后次序。考虑如下程序段：

LD SR1.A ;A装入符号寄存器1

LD SR2,B ;B装入符号寄存器2

ADD SR3,SR1,SR2 ;SR1、SR2两内容相加，以及存入符号寄存器3

LD SR4,C

LD SR5,D

ADD SR6,SR4,SR5

1. 先进行寄存器映射，后进行指令重排序，使用了多少机器寄存器？有流水性能的任何改进吗？
2. 仍以原程序开始，现在是先做指令重排序，后做寄存器映射，使用了多少机器寄存器？有流水性能的任何改进吗？
3. 请在表15-7中加入下面两个处理器：

（a）Pentium II （b）ARM

1. 多数情况下．未列为MIPS指令集一部分的普通机器指令能以单个MIPS指令来合成。请表示出如下的各MIPS指令序列。

（a）寄存器到寄存器的传送 （b）递增，递减

（c）求补 （d）求负 （e）清除

1. 一个SPARC实现中有K个寄存器窗口，它的物理寄存器数目N是多少？
2. SPARC缺乏几条CISC机器上普遍有的指令，其中某些指令可使用寄存器R0（它的值总为0)或常数操作数来模拟而成。这些被模拟的指令称为伪指令（pseudo instruction)，并被SPARC编译器所承 认。请表示如何模拟出如下伪指令，每个只使用单一SPARC指令。所有这些伪指令中，src和dst分别指的是源寄存器和目的寄存器（提示：保存到R0对R0无影响）。

（a）MOV src, dst （b）COMPARE src1, src2（c）TEST src1

（d）NOT dst（e）NEG dst（f）INC dst（g）DEC dst

（h）CLR dst（i）NOP

1. 考虑如下程序段：

if K > 10

L := K + 1

else

L := K – 1

将这些语句翻译后，能以如下形式进入SPARC编译器：

sethi %hi(K),%r8 ;

ld [%r8+%lo(K)],%r8 ;

cmp %r8,10 ;

ble L1 ;

nop

sethi %hi(K),%r9

ld [%r9+%lo(K)],%r9 ;

inc %r9

sethi %hi(L),%r10

st %r9,[%r10+%lo(L)];

b L2

nop

L1: sethi %hi(K),%r11

ld [%r11+%lo(K)],%r12 ;

dec %r12

sethi %hi(L), %r13

st %r12,[%r13+%lo(L)] ;

L2:

在每个分支指令之后包含一个nop指令，因此这些代码准许以延迟分支法来运行。

1. 与RISC机器无关的标准编译器优化通常可以有效地对前面的代码执行两次翻译，先翻译成汇编语言，然后对它进行优化处理。 请注意，上述汇编语言程序中两个装载指令是不必要的，并且如果保存指令挪移到程序中另一位置，则两次保存可合并成一次。请写出完成这些修改之后的程序。
2. 如果编译器现在能够完成针对SPARC的特有优化，请考虑使用设置注销位的ble指令（表示为ble,a L1），并将其他有用的指令移入它之后的延迟槽内，而取代nop指令。写出这一改动之后的程序。
3. 现在还有两条不必要的指令，请将它们移走，写出最终优化的汇编语言程序。