

第5章 处理器

在关键领域，任何细节都不能忽视。

——法国谚语

5.1 概述

在第4章中，我们看到一台计算机的性能由三个关键因素决定：指令数目、时钟周期和每条指令所需时钟周期数(CPI)。我们在第2、3章学习的编译器和指令集系统决定了一个程序所需的指令数目。而处理器的实现方式则决定了时钟周期长度和每条指令所需的时钟周期数。在本章中，我们为MIPS指令系统的两种不同实现方式建立数据通路和控制单元。

本章阐述了实现一个处理器所涉及的原理和技术。本节对整章内容进行了高度抽象和简单概括，后面的各节分别介绍如何建立数据通路，如何设计一个能够执行简化MIPS指令集的处理器，最后引出实现类似IA-32的复杂指令集所必需的概念。

对于有兴趣理解指令高层解释和它对程序性能影响的读者，本章开始的一节提供了理解这些概念的足够背景，同时介绍了流水线的基本概念，具体细节在下一章6.1节中给出。

对于渴望了解如何用硬件实现各种指令的读者而言，5.3节和5.4节提供了所需的补充材料。而且这两节对于理解第6章中有关流水线的所有内容也提供了足够的知识基础。对硬件设计有兴趣的读者而言，这两节应当做深入阅读。

本章各节覆盖了现代硬件实现的一般方法，也包括非常复杂的处理器(例如Intel Pentium系列处理器)。这些章节还解释了有限状态控制的基本原理及其不同实现方法，包括微程序实现方式。那些有兴趣深入了解处理器及其性能的读者会发现5.4节和5.5节很有用。5.7节(见CD■)介绍微程序设计，这是一种用来实现复杂控制机制的方法(比如IA-32处理器中所使用的)；在5.8节(见CD■)中可以看到如何使用硬件设计语言和CAD工具来进行硬件实现。

5.1.1 MIPS基本实现

首先来看我们要实现的核心MIPS指令集系统的一个子集：

- 存储访问指令 load word(lw)和 store word(sw)
- 算术逻辑指令 add、sub、and、or 和 slt
- 跳转指令 branch equal(beq)和 jump(j)

这个子集没有包括任何整数指令(比如我们没有包括移位、乘法除法指令)，也不包括任何浮点指令。但是，在建立数据通道和设计控制部分时用到的关键原理都会得以体现。其余未包括的指令的实现方式也是类似的。

在讨论此实现时，我们将会看到指令集如何在多个方面对具体实现方式产生的影响，以及不同实现策略的是如何影响时钟速率和机器CPI值。许多在第4章介绍到的关键的设计原理，如加快常用操作的速度和简单来自于规整的指导思想，在这个过程中都将得到体现。并且，在本章及下一章中用于实现MIPS子集的大多数概念与设计其他的多数计算机的基本思想是一致的，这包括从高性能服务器到通用微处理器到嵌入式处理器等，而后者将越来越广泛地应用在从VCR到汽车等各种产品中。

5.1.2 处理器实现概述

第2、3章介绍了MIPS的核心指令，包括整数算术逻辑指令、存储访问指令及分支指令。这些指令的实现过程大致相同，而与具体的指令类型无关。每条指令执行的前两个步骤是一样的：

- 根据程序计数器(PC)从内存中取出指令，PC指向存放该指令的地址。

- 通过指令字段的内容，选择读取一个或两个寄存器。对于取字指令，只需读取一个寄存器，而其他大多数指令则要求读取两个寄存器。

这两步之后，为完成指令而进行的操作将取决于具体的指令类型。幸运的是，对三种指令类型(存储访问、算术逻辑、分支)的每一种而言，其动作大致相同，而与具体操作码无关。

即使是不同类型的指令，也有一定的共性。例如，所有类型的指令在读取寄存器后，都要使用算术逻辑单元(arithmetic-logical unit, ALU)。存储访问指令用ALU计算地址，算术逻辑指令用来执行运算，分支指令用ALU进行比较。可以看出，指令的简洁和规整使许多指令的执行很相似，因而简化了实现过程。

使用ALU之后，不同类型指令需要进行不同的操作。存储访问指令需要对存储单元进行读出或写入而访问存储器。算术逻辑指令需要将ALU产生的数据写回寄存器中。而分支指令会根据比较的结果，决定是否需要更改下条指令的地址；否则将PC加4(自增4)以指向下一条指令的地址。

图5-1给出了MIPS实现的高层视图，重点描述了不同的功能单元和它们之间的连接方式。虽然图5-1指出了经过处理器中的大部分数据通路，但是这张图省略了指令执行过程的两个重要方面。

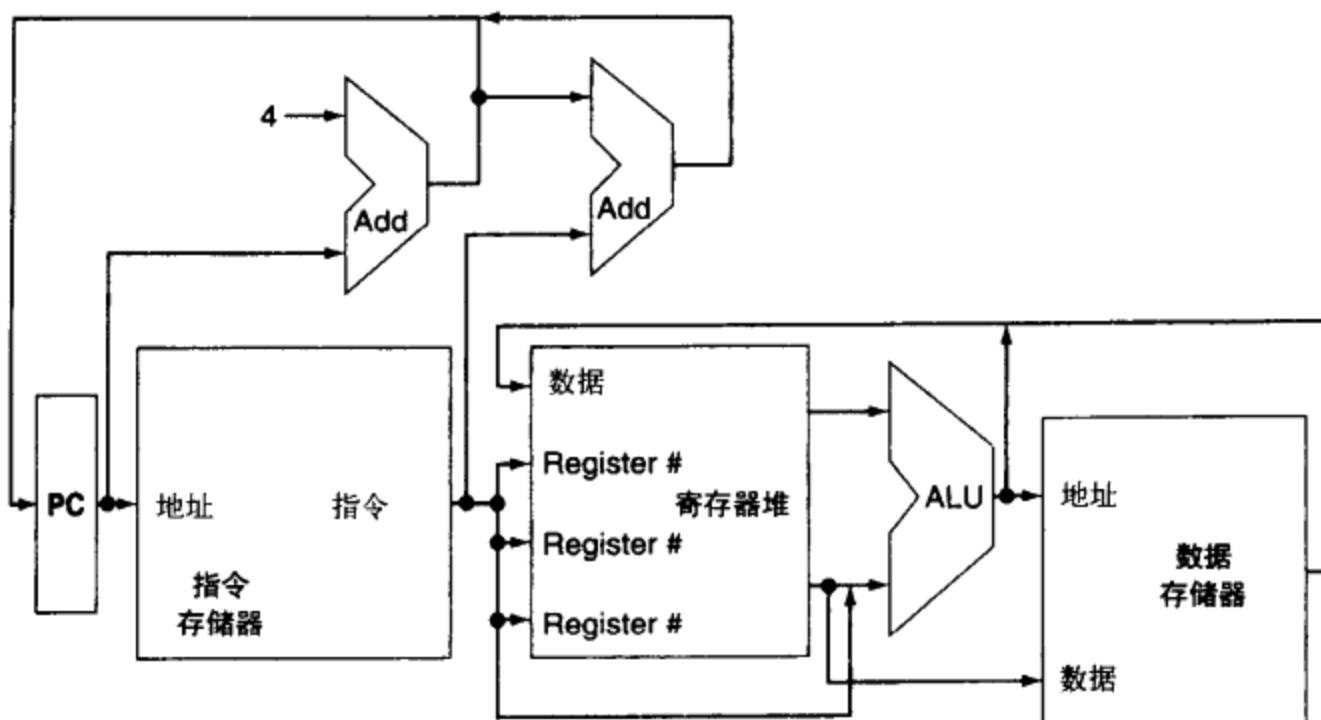


图5-1 MIPS子集的实现的抽象视图，包括主要的功能单元和它们之间的联系

[所有的指令通过程序计数器(PC)给指令存储器提供指令地址而开始执行。取指令后，由指令字段的内容指明指令的寄存器操作数。一旦取得寄存器操作数后，它们就可以用于计算存储器地址(用于存取数据)，计算算术结果(用于整数的算术逻辑指令)，或比较(用于分支指令)。若是算术逻辑指令，ALU的结果必须写入一个寄存器。若是存取数据操作，ALU的结果就作为地址，要么向寄存器中存储从存储器读出的数据或向寄存器写入存储器的内容。来自ALU或存储器的结果被写回寄存器堆。分支指令要求将ALU的输出作为下条被执行指令的地址，要么等于PC和分支偏移量相加，要么等于加法器将PC+4。连接功能单元的粗线表示总线，由多个信号构成。箭头帮助读者知道信息如何流动。因为信号线可能相交，在信号线相交的地方用圆点显式表示出来]

第一, 图 5-1 显示某些单元的数据来源于两个不同的源头。例如, 写入 PC 的值可能从两个加法器中的任一个得到, 还有写入寄存器堆(register file)的数据可能来源于 ALU 或者内存。实际上, 这些数据线不可能简单地连在一起; 必须增加器件用来从多个数据源中选择其中一个传输给目的单元。虽然这个选择设备称为数据选择器更合适, 它还是通常称为多路复用器(multiplexor)。多路复用器根据控制线的设置从多个输入中进行选择, 在附录 B 中有关于多路复用器的详细描述。控制线主要基于待执行的指令信息进行设置。

第二, 图中有些单元必须根据指令的类型进行控制。例如, 数据存储器在执行加载(load)指令时进行读操作, 在执行存储(store)指令时进行写操作。在执行加载和算术-逻辑指令时, 寄存器堆必须进行写操作。当然, 像在第 3 章看到的, ALU 也必须执行几种不同操作中的一个(附录 B 中阐述了 ALU 逻辑设计的细节)。类似多路选择器, 这些操作由控制线信号决定, 而控制线则根据指令码的不同域设置。

图 5-2 显示了图 5-1 对应的数据通路, 添加了三个必需的多路复用器和主要功能单元的控制线。一个以指令作为输入的控制单元用来决定如何设置功能单元和两个多路复用器的控制线。第三个复用器根据 ALU 的 zero 输出决定是将 PC 加 4 还是将分支目的地址写入 PC, 这是用来执行 beq 指令的比较操作。MIPS 指令集的规则和简单意味着可以使用简单的译码过程来判定如何设置控制线。

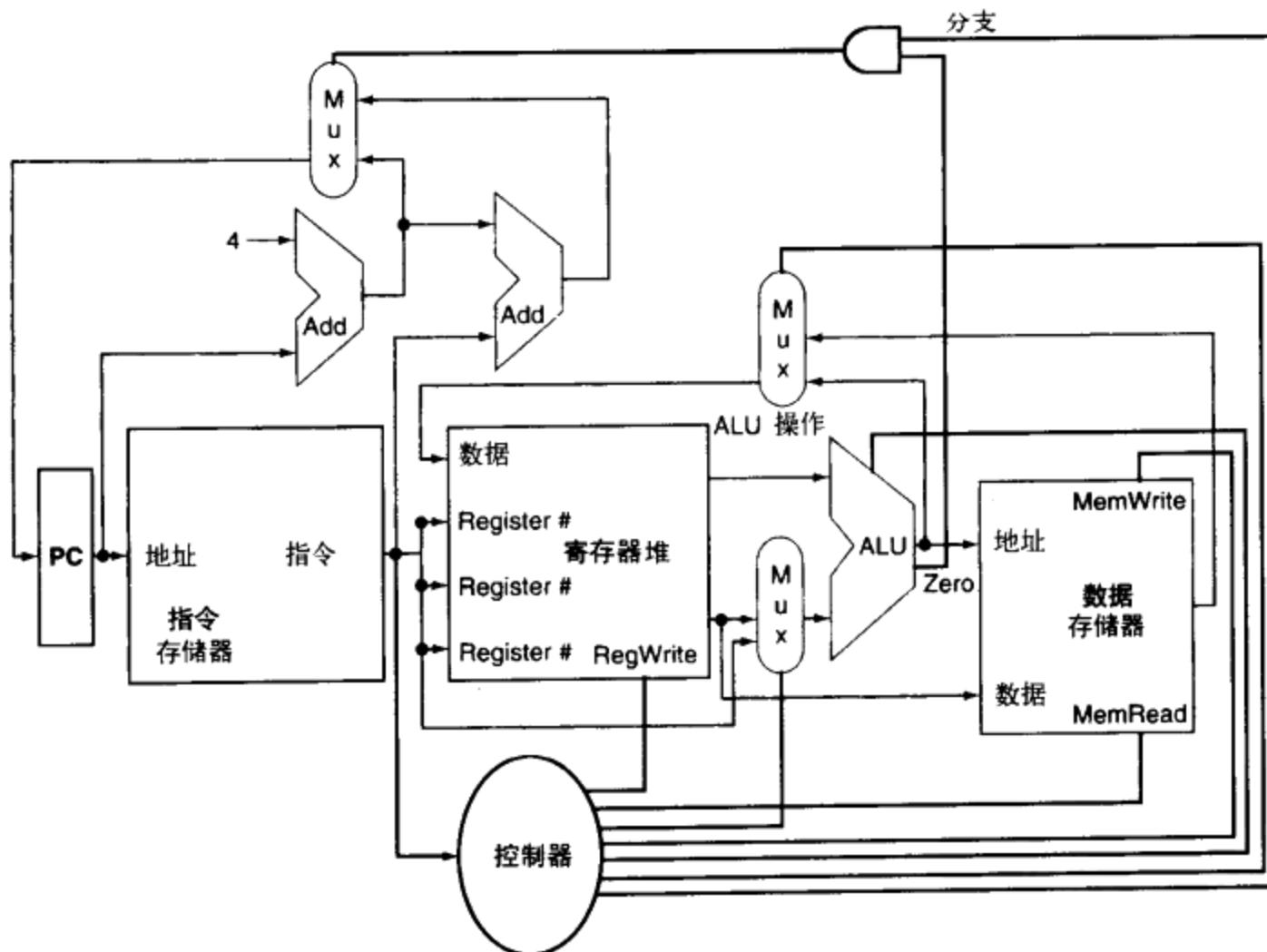


图 5-2 包含必需的多路复用器和控制线的 MIPS 子集的基本实现

[最顶端的多路复用器控制将哪个值写入 PC(PC + 4 或者分支目标地址); 多路复用器由一个“与”门控制, 一个输入是 ALU 的 Zero 输出, 另一个输入是表示分支指令的控制信号。输出返回寄存器堆的多路复用器用来控制 ALU 的输出(如果是算术-逻辑指令)或者数据内存的输出(如果是 load 指令)写入寄存器堆。最下面的多路复用器用来决定第二个 ALU 的输入是来源于寄存器(对于非立即型算术-逻辑指令)还是指令的偏移量字段(对于立即型操作、load 或 store、或者分支指令)。增加的控制线直接决定 ALU 执行的操作, 数据内存应该读还是写, 寄存器是否应该执行写操作。控制线用粗线表示以便于区分]

本章的其他部分将具体考察这幅视图中的各个细节。这需要加入更多的功能部件，以及增加功能单元间的连接数量，当然还要有控制单元以控制不同类型指令需要执行的操作。5.3 节和 5.4 节介绍了根据图 5-1 和图 5-2 的一般形式，每条指令使用单个长时钟周期的简单实现方案。在最初设计中，每条指令在一个时钟周期的边沿开始执行，在下一个时钟周期的边沿执行结束。

这种方式虽然容易理解，但并不实际。因为这比不同类型指令耗费不同数量短时钟周期的方法慢。在这个简单机器的控制设计完成之后，我们将介绍每条指令使用多个时钟周期来实现。在 5.5 节~5.8 节中，我们将讨论更高级的控制概念、异常处理方式和硬件设计语言的用途，届时会涉及多周期设计。

自测

本节概念描述的单周期数据通路必须将指令和数据内存独立分开，这是因为：

1. 在 MIPS 中指令和数据的格式不同，因此需要不同的内存。
2. 独立的内存设计价格更低。
3. 处理器的操作在一个时钟周期内完成，不可能在一个周期内使用单端口内存进行两次不同的访问。

5.2 逻辑设计规则

在考虑计算机的设计时，必须决定机器的逻辑实现如何操作以及机器的时钟。本节将讨论一些本章经常使用的有关数字逻辑的关键思想。如果你欠缺数字逻辑方面的知识，那么在继续学习之前，看一看■附录 B 将有所帮助。

MIPS 实现中的功能单元由两种不同类型的逻辑单元组成：计算数据值的组合单元和包含状态的时序单元。操作数据值的单元是组合式 (combinational) 的，也就是说它们的输出只依赖于当前的输入。给出相同的输入，组合逻辑单元总是产生相同的输出。在第 3 章和 ■附录 B 中讨论的图 5-1 中的 ALU 就是组合逻辑单元。因为 ALU 中没有内部状态，对于给定的一组输入，ALU 总是生成相同的输出。

本设计中其他的单元不是组合逻辑式的，它们包含某种状态 (state)。如果一个单元拥有内部存储功能，它就会包含状态。因此称这些单元为状态单元[⊖]，因为关机后，可以通过给这些状态单元设置为其各自的值而重启计算机。并且，如果保存并恢复了状态单元的值，机器就好像没有断电一样继续运行。因此，这些状态单元完全刻画了机器的特性。图 5-1 中，指令和数据存储器还有寄存器都属于状态单元。

一个状态单元至少有两个输入和一个输出。这两个输入分别为待写入该单元的数据值和决定何时写入的时钟信号。状态单元的输出提供了在先前某个时钟周期写入该单元的数据值。比如，逻辑上最简单的状态单元是一个 D 触发器 (参见附录 B)，它有两个输入 (一个数据值和一个时钟) 和一个输出。除了触发器，MIPS 的实现中还用了另外两种状态单元：存储器和寄存器，这些在图 5-1 中都已给出。时钟用于决定状态单元何时被写入；状态单元随时可读。

包含状态的逻辑部件又称为时序 (sequential) 逻辑电路，因为他们的输出由输入和内部状态共同决定。比如，代表寄存器的功能单元的输出取决于所提供的寄存器数和以前写入寄存器的内容。组合单元和时序单元的有关操作及结构都在 ■附录 B 中有详细论述。

本书使用已有效的信号 (asserted) 表示一个逻辑高的信号，有效信号 (assert) 是指应该设置为逻辑高的信号，无效信号 (deassert) 和已无效信号 (deasserted) 表示逻辑低。

[⊖] 状态单元 (state element) 一个存储单元。

时钟同步方法

时钟同步方法[○]规定了信号可以读出和写入的时刻。规定信号读写的时间很重要，因为若一个信号同时被读出和写入，则所读出的信号可能是写入前的值，也可能是新写入的值，甚至是两者的混合值！显然，计算机的设计中不能允许这样的不确定性。定时方法即是为避免这种情况而设计。

为简单起见，假定采用边沿触发[○]的时钟同步方法，它意味着时序逻辑单元中存储的所有值都只允许在时钟跳变的边沿时改变。也就是说，状态单元都只在时钟跳变的边沿改变其存储内容。因为只有状态单元能存储数据值，所有的组合逻辑都必须从状态单元集合接收输入，并将输出写入状态单元集合中。其输入为从前某时钟周期写入的数据，其输出可供以后的某个时钟周期使用。

图 5-3 描述了一个组合逻辑模块及与其相连的 2 个状态单元，组合逻辑模块的操作在一个时钟周期内完成：所有信号在一个时钟周期内从状态单元 1 经组合逻辑到达状态单元 2，信号到达状态单元 2 所需的时间定义了时钟周期的长度。

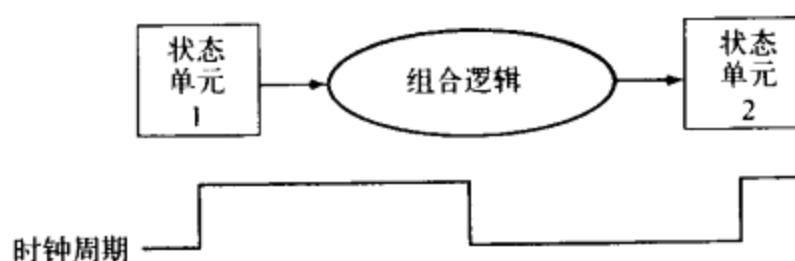


图 5-3 组合逻辑、状态单元和时钟周期的紧密联系

[在一个同步的数字系统中，时钟信号决定了数值何时写入状态单元的内部存储。在有效的时钟边沿导致状态更新之前，状态单元的输入信号必须达到稳定值(即信号在时钟边沿后不变)。假定所有状态单元，包括存储器都是边沿触发的]

为简单起见，当某个状态单元在每个有效的时钟边沿都进行写入操作时，我们不画出它的写控制信号[○]。相反，若某个状态单元不是每个周期都进行更新，那么它就要有一个显式的写控制信号。写控制信号和时钟信号都是输入信号，并且仅当写控制信号有效且时钟边沿到来时，状态单元才改变状态。

使用边沿触发的方法，如图 5-4 所示，可以在一个时钟周期内读出一个寄存器的值，并经过一些组合逻辑，再次写入该寄存器。选择在时钟的上升沿还是在下降沿进行写操作无关紧要，因为组合逻辑的输入只有在所规定的时钟边沿才可能发生变化。使用边沿触发定时方法，不存在同一个时钟周期内的反馈，图 5-4 所示的逻辑可以正确工作。在附录 B 中，简要介绍了其他的一些时序限制(如建立和保持时间)和一些定时方法。



图 5-4 一种边沿触发方法，允许状态单元在同一个时钟周期内被读出和写入，但不会产生竞争而出现在中间数据值

[当然，时钟周期仍然要足够长，使得当有效的时钟边沿到来时输入已经稳定。状态单元的更新由时钟边沿触发，所以不可能在一个时钟周期之内出现反馈。如果有反馈，这个设计就不能正常工作。本章和下一章的设计都以边沿触发的定时方法和类似于本图显示的结构为基础]

- 时钟同步方法(clocking methodology) 确定数据相对时钟而言，何时是有效和稳定的方法。
- 边沿触发时钟同步(edge-triggered clocking) 一种时钟同步机制，其中所有状态变化都发生在时钟边沿。
- 控制信号(control signal) 用于多路复用器的选择或指导功能单元操作的信号；与数据信号相比，控制信号包括功能单元所要执行的操作的具体信息。

几乎所有这些状态和逻辑单元的输入和输出都为 32 位, 因为处理器处理的大多数数据的宽度为 32 位。若某单元的输入或输出不是 32 位, 会被特别指出。图示中用粗线表示总线(bus), 即宽度为 1 位以上的信号。有时要把几根总线合起来构成一个更宽的总线; 比如, 可能将 2 根 16 位总线合成一根 32 位总线。在这种情况下, 总线标注将作出相应说明。另外还加上箭头以指明单元间数据传输的方向。最后, 用不同粗细区分数据信号和控制信号; 这两者的差别将在本章的后面更清楚地体现。

自测

判断正误: 因为在一个时钟周期内寄存器堆既可以读也可以写, 任何使用边沿触发写操作的 MIPS 数据通路一定在寄存器堆中拥有多于一个的备份。

5.3 数据通路的建立

开始设计数据通路时, 比较合理的方法是先看看每种 MIPS 指令执行时所需的主要部件。先来确定每条指令需要什么数据通路部件^①, 再用这些部件为每种指令类型建立其数据通路。在指出数据通路部件的同时, 确定它们对应的控制信号。

首先需要的部件是一个存储程序指令的地方: 一个存储单元用来存储程序的指令, 并根据所给地址提供指令, 如图 5-5 所示。当前指令的地址也必须存放在一个状态单元中, 称之为程序计数器(PC)^②, 也在图 5-5 中示出。最后, 需要一个加法器增加 PC 的值以指向下一条指令的地址。这个加法器是一个组合单元, 可以用第 3 章设计的 ALU 实现, 其设计方法详见附录 B, 只需将控制线路设为总是进行加法操作。如图 5-5 所示, 这样的 ALU 将被加上“Add”标记, 以表明它只是一个加法器而不能再进行其他 ALU 操作。

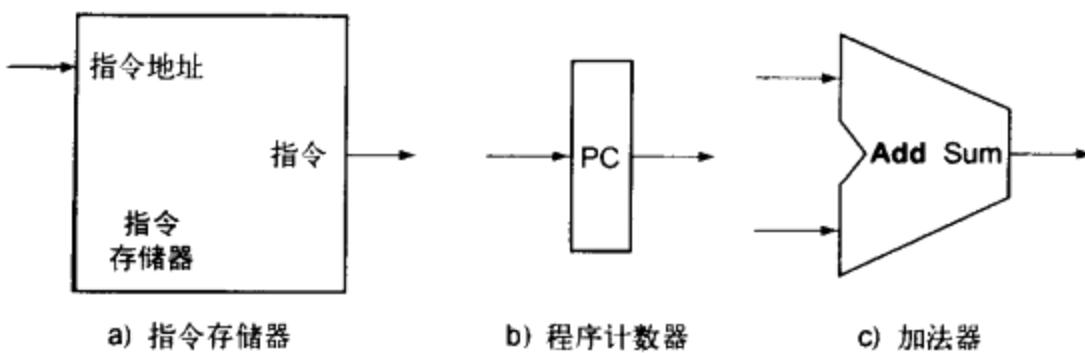


图 5-5 存取指令需要两个状态单元, 计算下一条指令的地址需要一个加法器

[两个状态单元分别是指令存储器和程序计数器。指令存储器只需提供读权限, 因为数据通路不会改写指令。由于指令存储器是只读的, 可将它视为组合逻辑: 任意时刻的输出都反映了输入的地址处的内容, 而不需要读控制信号。(在加载程序时需要写入指令存储器, 但是这很容易实现, 所以为简单起见忽略它。) 程序计数器是一个 32 位的寄存器, 它在每个时钟周期末都会被写入, 所以不需要写控制信号。加法器是被设置为只进行加法的 ALU, 它将输入的两个 32 位数相加, 将结果输出]

要执行任何一条指令, 首先要从存储器中将指令取出。为准备执行下一条指令, 也必须把程序计数器加到指向下一条指令, 即向后移动 4 个字节。此时所需的取指令以及增加 PC 以获得下一时序指令的数据通路, 如图 5-6 所示, 使用了图 5-5 中的 3 个部件。

现在讨论 R 型指令(参见图 2-7)。这种指令读两个寄存器, 对它们的内容进行 ALU 操作, 再

^① 数据通路部件(datapath element) 处理器中用来操作或保存数据的功能单元。在 MIPS 的实现中, 数据通路部件包括指令和数据存储器、寄存器堆、算术逻辑单元(ALU)以及加法器。

^② 程序计数器(program counter, PC) 保存程序正在执行的指令地址的寄存器。

写回结果。这类指令称为 R 型指令或算术逻辑指令(因为它们进行算术或逻辑运算)。这个指令集合包括第 2 章介绍的 add、sub、and、or 和 slt 指令。回忆一下，这类指令的典型形式是 add \$t1, \$t2, \$t3, 它将读 \$t2 和 \$t3，并将结果写到 \$t1。

处理器的 32 个通用寄存器位于一个叫做寄存器堆^①的结构中。一个寄存器堆就是一个寄存器集合，其中的寄存器都可通过指定相应的寄存器序号来进行读写。寄存器堆包含了计算机的寄存器状态。另外，还需要一个 ALU 来对从寄存器读出的数值进行运算。

由于 R 型指令有 3 个寄存器操作数，对每条指令，都要从寄存器堆读出 2 个数据字，再写入一个数据。为从寄存器中读出一个数据字，寄存器堆需要一个输入信号指定要读的寄存器号和输出信号指示从寄存器堆读出的结果。为写入一个数据字，寄存器堆要有两个输入：一个指定要写的寄存器序号，另一个提供要写的数据。寄存器堆总是根据输入的寄存器序号输出相应的寄存器内容，而写操作由写控制信号控制，在写操作发生的时钟边沿，写控制信号必须是已有效的。这样，一共需要 4 个输入(3 个寄存器序号和 1 个数据)和 2 个输出(2 个数据)，如图 5-7 所示。输入的寄存器序号为 5 位，可指示 32 个寄存器中的某一个($32 = 2^5$)，一条数据输入总线和两条数据输出总线宽度均为 32 位。

图 5-7 中的 ALU 有两个 32 位输入和一个 32 位输出，还有一个结果为 0 时的 1 位信号。ALU 由附录 B 中详细说明的 4 位信号控制。在需要知道如何设置控制信号时，我们还会简要回顾 ALU 控制。

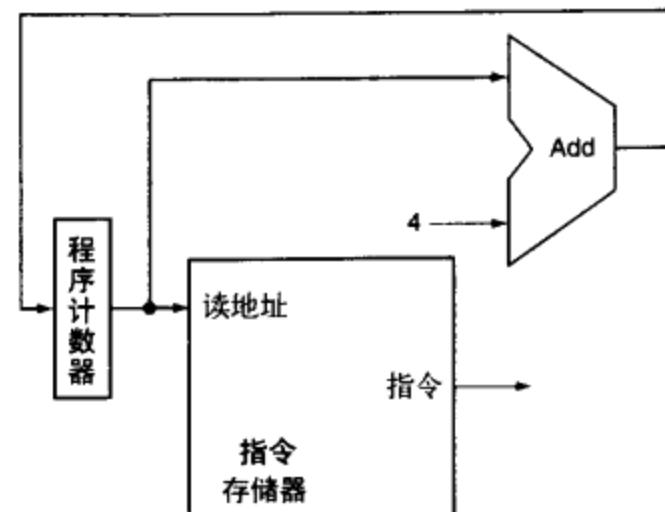


图 5-6 用于取指和程序计数器自增的数据通路的一部分

[取出的指令被数据通路的其他部分所使用]

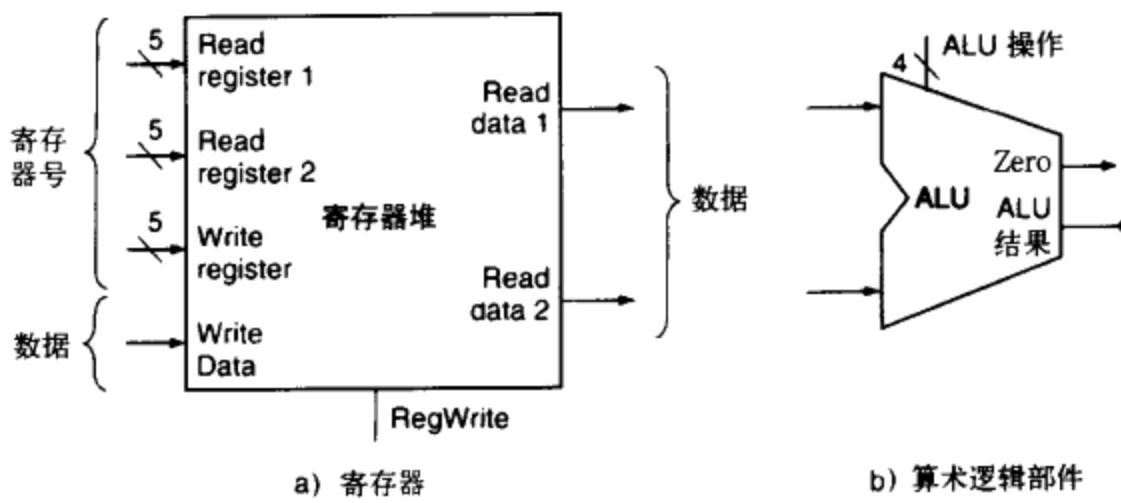


图 5-7 实现 R 型指令的 ALU 操作所需的两个状态单元为寄存器堆和 ALU

[寄存器堆包括了所有的寄存器，有两个读端口和一个写端口。多端口寄存器堆的设计在附录 B 的 B.8 节讨论。寄存器堆的输出通常对应于读寄存器的输入；不需要其他的控制输入。相反，写寄存器必须明确激活写控制信号。记住写操作是边沿触发的，所以所有的写操作的输入(比如，要写的值、寄存器号、写控制信号)必须在时钟边沿有效。因为寄存器堆的写入是边沿触发的，可以在同一时钟周期内读出和写入同一寄存器：读操作将读出以前的时钟周期写入的内容，而写入的值在下一时钟周期才可读。寄存器号的输入都是 5 位的，数据线为 32 位。采用附录 B 的 ALU 设计，ALU 将进行的操作由 4 位 ALU 操作信号控制。使用 ALU 的零输出信号实现分支控制。溢出信号到 5.6 节讲述异常时才会用到，在此之前将它忽略]

^① 寄存器堆(register file) 由一组寄存器组成的状态单元，可以通过寄存器编号对其读写。

下面考虑 MIPS 取字和存字指令，其一般形式为：

`lw $t1, offset_value($t2)` 或 `sw $t1, offset_value($t2)`

在这类指令中，通过将基址寄存器 `$t2` 的内容与指令中的 16 位有符号偏移字段相加，得到存储器地址。如果是存数指令，要从寄存器堆中的 `$t1` 中读出要存储的数据；如果是取数指令，则要将从存储器中取出的数据存入寄存器堆中指定的寄存器 `$t1` 中。所以，图 5-7 中的寄存器堆和 ALU 都将被用到。

另外，还需要一个单元将指令中的 16 位的偏移字段符号扩展^①为 32 位的有符号值，以及一个存储读出和写入的数据的存储单元。存储单元在存数指令时被写入；所以它有读、写控制信号，输入地址和要被写入存储器的输入数据。图 5-8 中给出了这两个单元。

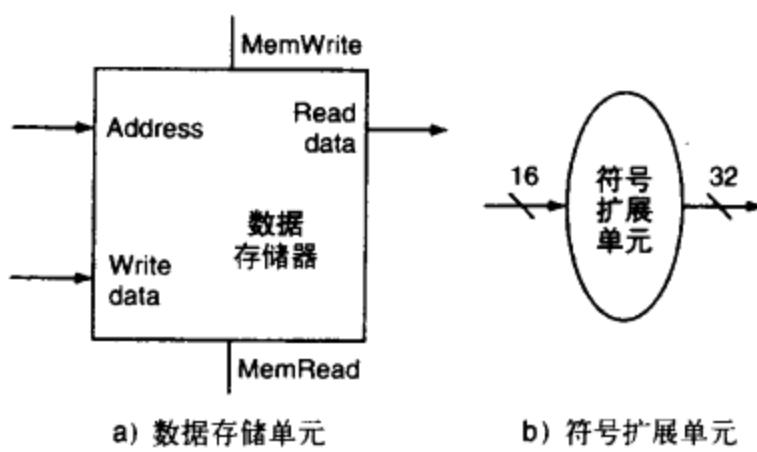


图 5-8 除了寄存器堆和图 5-7 的 ALU，存数和取数指令所需的两个单元是数据存储单元和符号扩展单元

[存储单元是一个状态单元，两个输入分别为地址和所写数据，一个输出为读出的内容。读、写控制信号都是独立的，尽管任意给定时钟只能激活其中一个。存储单元需要一个读信号，因为与寄存器堆不同，从存储单元中读无效地址的值会导致问题发生，这将在第 7 章中讨论。符号扩展单元有一个 16 位的输入，符号扩展为 32 位后输出(参见第 3 章)。假定数据存储器的写是边沿触发的。标准的存储芯片实际上有一个写的使能信号用于写操作。虽然写使能信号不是边沿触发的，边沿触发的设计可以很容易地应用于真正的存储芯片。关于真正的芯片如何工作，参见附录 B 的 B.8]

`beq` 指令有 3 个操作数，其中 2 个为寄存器，用于比较是否相等，另一个是 16 位偏移量，用以计算相对于分支指令地址的分支目标地址^②。指令格式为 `beq $t1, $t2, offset`。要实现这条指令，必须计算分支目标地址，这是通过把指令的符号扩展偏移字段加到 PC 完成的。分支指令(参见第 2 章)的定义中有两个需要注意的细节：

- 指令集系统规定计算分支地址时使用的基地址，是分支指令的下一条指令的地址。因为在取指数据通路中计算了 $PC + 4$ (下条指令的地址)，所以用这个值作为计算分支目标地址时的基地址较易于实现。
- 系统还规定偏移量字段左移 2 位以指示以字为单位的偏移量，这样偏移量的有效范围就扩大了 4 倍。

为了适应后面这种复杂情况，需要把偏移量字段移动 2 位。

除了计算分支目标地址，还必须确定是顺序执行下一条指令，还是去执行分支目标地址处的

① 符号扩展(sign-extend) 为了增加数据项的大小，将原来数据项的高端符号位复制到长度更大的目标数据项的高端符号位。

② 分支目标地址(branch target address) 在分支指令中定义的地址，如果分支成功该地址就成为新的程序计数器(PC)值。在 MIPS 体系结构中，分支目标由指令偏移量字段的值与紧随分支指令的地址之和给定。

指令。当转移条件为真(如, 操作数相等)时, 分支目标地址成为新的 PC, 就是实现了分支跳转^①。若操作数不等, 增值后的 PC 将取代当前 PC (就像其他一般指令一样); 这时就是分支不跳转^②。

所以, 分支数据通路需要进行两个操作: 计算分支目标地址和比较寄存器的内容。(将很快看到, 分支指令还需要改变数据通路的取指部分。)由于处理分支的复杂性, 我们在图 5-9 中解释了处理分支的目标数据通路段的结构。为计算分支目标地址, 分支数据通路包含了一个如图 5-8 所示的符号扩充单元, 和一个加法器。为进行比较, 要由图 5-7 所示的寄存器堆提供两个寄存器操作数(但不需向寄存器堆写入数据)。另外, 比较可由附录 B 设计的 ALU 完成。因为 ALU 提供一个指示结果是否为 0 的输出信号, 可以把两个寄存器操作数作为输入, 并将 ALU 控制设置为作减法。若 ALU 输出的零信号有效, 则可知两操作数相等。尽管零输出信号始终指示结果是否为 0, 但只用它来实现分支时的等值测试。稍后将讨论将 ALU 用于数据通路时, 怎样连接它的控制信号。

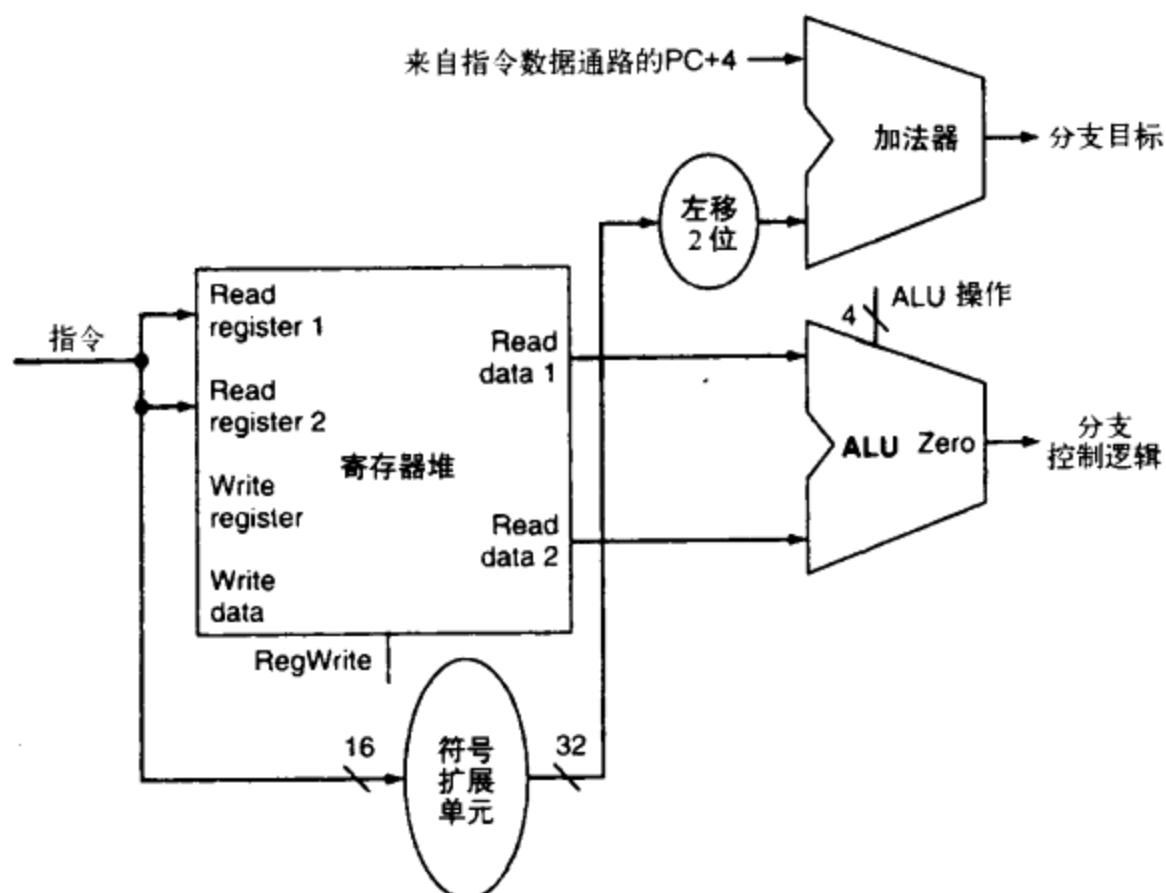


图 5-9 分支指令的数据通路通过 ALU 计算分支条件是否成立, 而另外的加法器将增值后的 PC 值与符号扩展后左移两位的指令低 16 位(分支偏移量)相加, 得到分支的目标地址

[标有左移两位的单元只是输入到输出之间一条简单的信号线路, 它给符号扩展后的偏移量字段的低位加上 00_{two} ; 因为所“移动”的距离是固定的, 所以并不需要真正的移位电路。我们知道偏移量是从 16 位扩展而来的, 所以移位只会丢掉“符号位”。控制逻辑根据 ALU 的零输出决定是用增值的 PC 还是分支目标地址来取代当前的 PC]

跳转指令将偏移地址的低 26 位左移两位后, 以之代替 PC 的低 28 位。移位通过给偏移量后面加上两个 0 实现(如第 2 章所述)。

- ① 分支跳转(branch taken) 如果分支条件满足且程序计数器(PC)的值是分支目标, 则称分支跳转。所有无条件分支指令都是跳转分支。
- ② 分支不跳转(branch not taken) 分支条件不满足并且程序计数器(PC)的值为分支指令的后续地址, 则称分支不跳转。

细节:在MIPS指令集中,分支指令是延迟^①的,即无论分支条件是否满足,它之后的那条指令总被执行。条件不满足时,情况与一般的分支指令相同;条件满足时,延迟的分支指令先执行它下面的那条指令,然后再跳转到指定的分支目标地址。将分支指令设计为延迟的,其动机来自流水线对分支指令的影响(参见6.6节)。为简单起见,在本章中忽略延迟的分支指令,而实现非延迟的beq指令。

创建单条数据通路

现在已经检视过单个指令类需要的数据通路部件,我们可以将它们组合到单个数据通路并加上控制来完成实现。最简单的数据通路会尝试在一个时钟周期执行所有指令。这意味着对每条指令,数据通路资源不能重复使用,因此任何要使用不止一次的单元都要复制。因此我们需要一个数据存储器外的指令存储器。尽管一些功能单元需要复制,许多单元仍然可以由不同的指令流共享。

要在两个不同的指令类间共享数据通路单元,需要允许多个到功能单元输入端的连接,使用多路复用器和控制信号在多个输入间选择。

例题 构建数据通路

算术逻辑(或R类型)指令的操作和内存指令十分相似。关键的区别如下:

- 算术逻辑指令通过来自两个寄存器的输入使用ALU进行计算。内存指令也能使用ALU进行地址计算,尽管第二个输入是符号扩展的16位偏移字段。
- 存储到目标寄存器的值来自ALU(大型指令)或者存储器(load指令)。

说明如何为存储引用指令和算术逻辑指令的操作部分构建数据通路,其中只使用一个寄存器堆和一个ALU来处理两类指令,加上任意必需的多路复用器。

解 要创建只有单个寄存器堆和ALU的数据通路,我们必须为第二个ALU输入以及存入寄存器堆的数据支持两个不同的源。因此,一个复用器放在ALU输入处,另一个放在寄存器堆的数据输入处。图5-10示出了这个组合数据通路的操作部分。

现在,通过增加指令取的数据通路(见图5-6),R类型和内存指令的数据通路(见图5-10)以及分支的数据通路(见图5-9),我们可以组合各个部分得到MIPS体系结构的简单数据通路。图5-11

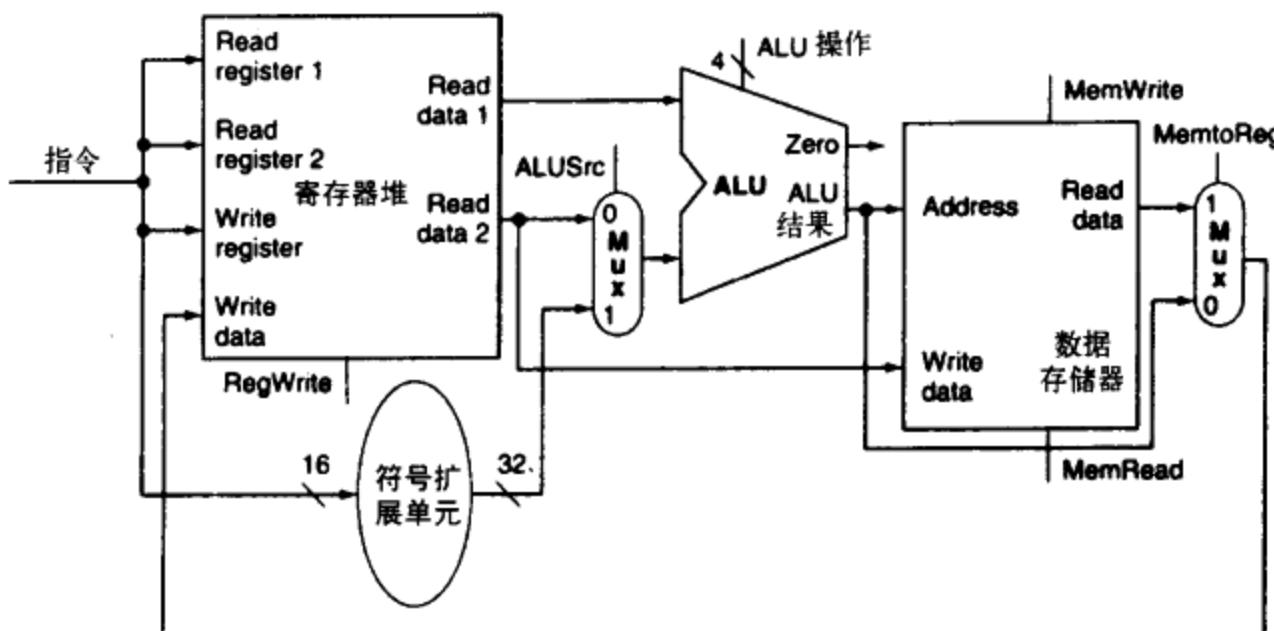


图 5-10 内存指令与 R 类指令的数据通路

[此例表明怎样通过增加乘法器来把图 5-7 和图 5-8 的片段组装成单数据通路。如上例所述需要两个乘法器]

① 延迟分支(delayed branch) 紧接着分支指令的指令总是能够被执行,不管分支条件是否满足。

示出了这样的组合后的数据通路。分支指令使用主 ALU 进行寄存器操作数比较，因此必须用图 5-9 的加法器计算分支目标地址。还要一个额外的复用器，用于选择将顺序指令地址($PC + 4$)或分支目标地址写入 PC。

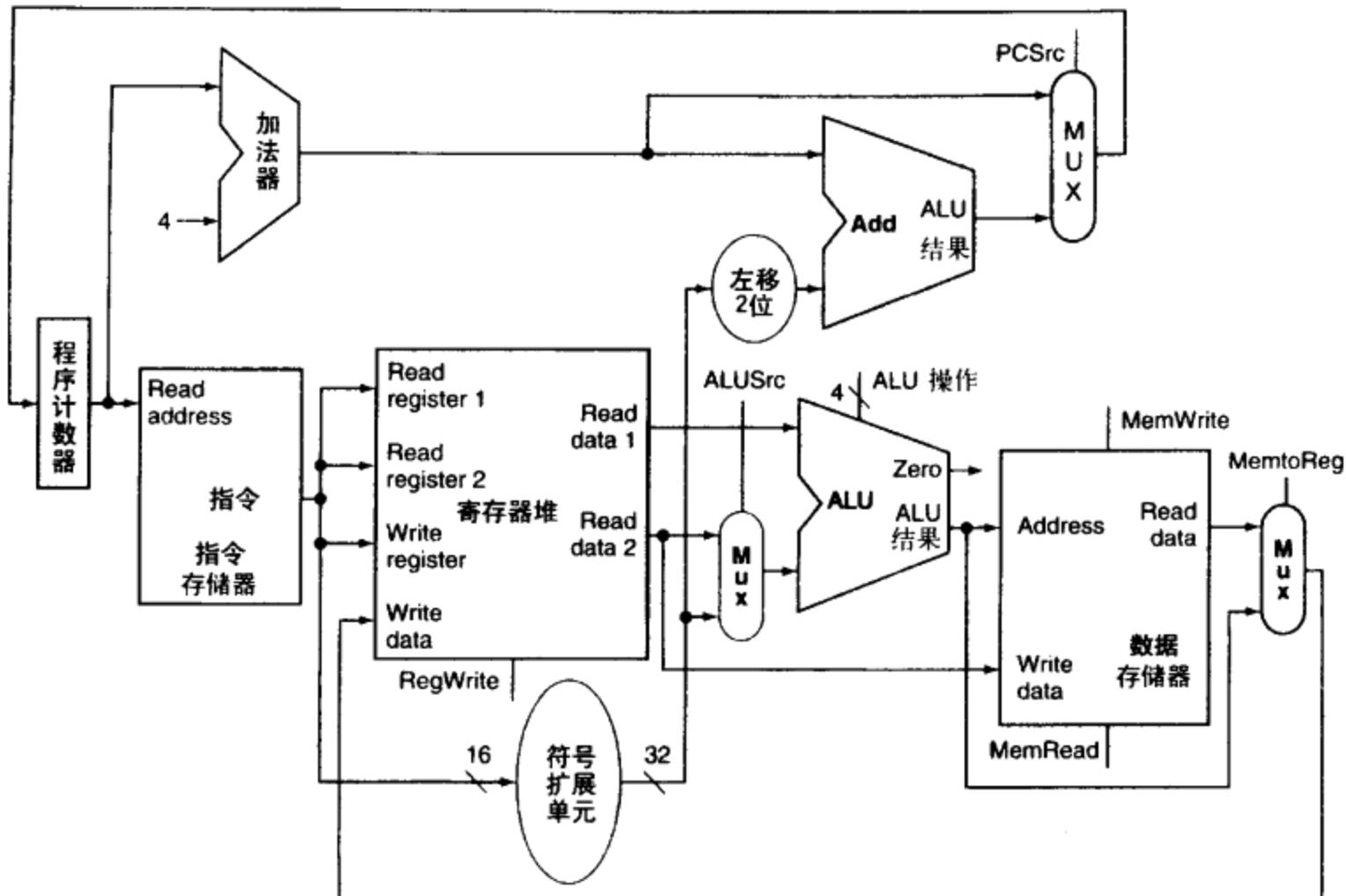


图 5-11 不同类型指令需要的单元组成的 MIPS 体系简单数据通路

[这个数据通路可以在单时钟周期内执行基本指令(取字/存字, ALU 操作, 分支指令)。为了包括分支指令执行需要额外的多路复用器。对跳转指令的支持以后再增加]

现在完成了简单数据通路，可以加上控制单元了。控制单元必须能接收输入并对每个状态量、选择器控制和 ALU 控制生成写信号。在设计控制单元时，最好先设计 ALU 控制，因为它的变化形式比较多。

自测

下列关于取指令的说法哪一个正确？

- 应该设置 MemtoReg，以将数据从内存送往寄存器堆。
- 应该设置 MemtoReg，以将正确的寄存器目标送往寄存器堆。
- 我们不必考虑 MemtoReg 的设置。

5.4 一个简单的实现方案

本节将讨论被认为是最简单的 MIPS 子集实现方案。通过把上一节中数据通路的各部分集成起来，并加入必要的控制线路，可得到这个简单的数据通路及其控制。这个简单的实现方案包括了取字(lw)，存字(sw)，等值分支(beq) 和算术逻辑指令 add、sub、and、or 和小于则置位。稍后还会将它扩充为包括跳转指令(j)。

5.4.1 ALU 的控制

在附录 B 中可以看到 ALU 有 4 个控制输入。这些位未被编码，所以 16 种可能的输入组合

中只有 6 种可能用于这个子集。■附录 B 中的 MIPS ALU 显示了下面 6 种组合：

根据指令的不同，ALU 将实现右边前五种功能中的某一种。(MIPS 指令集中的其他部分需要异或。)对于取字和存字指令，ALU 进行加法运算获取存储地址。对于 R 型指令，根据指令低六位的功能字段(功能)(参见第 2 章)，确定 ALU 执行五种操作中的一种(与、或、减、加、小于则置 1)。对等值分支指令，ALU 做减法操作。

可以用一个小的控制单元生成 4 位的 ALU 控制输入，这个功能单元的输入为指令的功能字段和一个占 2 位的称为 ALUOp 的控制字段。ALUOp 指明要进行的操作是存/取数需要的加法(00)，beq 需要的减法(01)，还是由指令的功能字段决定(10)。该 ALU 控制单元输出 4 位信号，即前面介绍的 6 种 4 位组合之一，直接对 ALU 进行控制。

图 5-12 说明了怎样根据 2 位的 ALUOp 和 6 位的功能码来设置 ALU 的控制输入。为更加完整，对 ALUOp 各位与指令操作码之间的关系也进行了说明。在本章的后面将会看到怎样由主控制单元生成 ALUOp。

ALU 控制输入	功 能
0000	与
0001	或
0010	加
0110	减
0111	小于则置 1
1100	异或

指令操作码	ALUOp	指令操作	功能字段	期望的 ALU 动作	ALU 控制输入
LW	00	取字	XXXXXX	加	0010
SW	00	存字	XXXXXX	加	0010
相等分支	01	相等分支	XXXXXX	减	0110
R 型	10	加	100 000	加	0010
R 型	10	减	100 010	减	0110
R 型	10	与	100 100	与	0000
R 型	10	或	100 101	或	0001
R 型	10	小于置 1	101 010	小于置 1	0111

图 5-12 如何根据 ALUOp 控制位和 R-型指令的不同功能码设置 ALU 控制位

[操作码在第一列，决定了 ALUOp 的各位设置。所有的编码以二进制给出。注意当 ALUOp 码为 00 或 01 时，期望的 ALU 动作不依赖于功能码字段；这时，称我们“不关心”功能码的值，功能字段写为XXXXXX。当 ALUOp 值为 10 时，功能码用于设置 ALU 的控制输入]

这种多层解码的手法(即，主控制单元生成 ALUOp 位作为 ALU 控制的输入，ALU 控制再生成真正控制 ALU 单元的信号)是一种常用的实现技术。使用多层控制可以减小主控制单元的规模。使用多个小控制单元还可能提高控制单元的速度。这种优化是很重要的，因为控制单元的性能通常很关键。

把 2 位的 ALUOp 字段和 6 位的功能字段映射为 3 位的 ALU 操作控制位，有多种不同方法。因为功能字段的 64 种可能取值中只有很小一部分有意义，并且仅当 ALUOp 位取值为 10 时才使用功能字段，可以用一个小逻辑单元去识别可能取的值，以生成正确的 ALU 控制位。

为设计这个逻辑单元，有必要为 ALUOp 位和功能码字段的有意义的组合生成一张真值表，如图 5-13 所示；这个真值表说明了 3 位的 ALU 控制信号怎样根据这两个输入字段得到。由于完整的真值表很大($2^8 = 256$ 项)，并且对其中许多种输入组合的 ALU 控制的值都不关心，只列出了使 ALU 控制有确定取值的真值表项。在本章中，均只列出了那些必须有效的真值表项，而省略那些恒为 0 的项或无关项。(这样做的缺点在■附录 C 的 C.2 节中讨论。)

ALUOp		功能字段							操作
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	0110	
1	X	X	X	0	0	0	0	0010	
1	X	X	X	0	0	1	0	0110	
1	X	X	X	0	1	0	0	0000	
1	X	X	X	0	1	0	1	0001	
1	X	X	X	1	0	1	0	0111	

图 5-13 三个 ALU 控制位(称为 Operation)的真值表

[输入为 ALUOp 和功能码字段。在此只列出了 ALU 控制有效的项，也包括一些无关项。比如，ALUOp 不用 11 编码，则真值表可包含 1X 和 X1 项，而不是 10 和 01 项。同样，当使用功能字段时，指令的前两位(F4 和 F5)总是 10，所以它们是无关项，在真值表中以 XX 替代]

由于在许多情况下对某些输入的取值无关，为了使真值表简练，也列出无关项^①。真值表中的无关项(通过在输入列以“X”表示)表明，输出与该列对应的输入取值无关。例如，ALUOp 位取 00 时，如图 5-13 的表的第一行，无论功能码取何值，ALU 控制总是被设为 010。这时，真值表中此行的功能码就为无关项。在后面，还会有另一种无关项的例子。若你不熟悉无关项的概念，可参见附录 B 中更多的讨论。

真值表建好之后，可以进行优化并转化成门电路。这是一个完全机械的过程。所以将此过程及其结果放在附录 C 的 C.2 节中讨论，在此省略。

5.4.2 主控制单元的设计

在设计完以功能码和 2 位信号作为控制输入的 ALU 后，现在来看看控制的其他部分。开始之前，首先看看一条指令的各个字段和如图 5-11 所示的数据通路所需的控制线路。为了理解怎样将指令的各个字段与数据通路相连，需要复习一下三种指令类型的格式：R-型指令、分支指令和存/取指令。它们的格式如图 5-14 所示。

遵循以下几条指令格式的主要规则：

- op 字段，亦称操作码^②，总是为 31:26 位。用 Op[5:0] 来表示。
- 对于 R-型指令、分支指令和存数指令，要读取的两个寄存器为 rs 和 rt 字段，分别为 25:21 位和 20:16 位。
- 存取数指令的基址寄存器字段在 25:21 位(rs 字段)。
- 等值分支指令、存数/取数指令的 16 位偏移量在 15:0 位。
- 有两个地方存放目标寄存器。对存数指令为 20:16 位(rt 字段)，对 R-型指令为 15:11 位(rd 字段)。所以需要一个多路复用器，以指示要写的寄存器序号在指令的哪个字段中。

根据这些信息，可以给简单的数据通路加上指令标记和一个额外的多路复用器(给寄存器堆的写寄存器号输入)。图 5-15 加上了这些，以及 ALU 控制模块、状态单元的写信号、数据存储器的读信号和多路复用器的控制信号。由于所有多路复用器都为双输入，它们各需要一根控制线。

^① 无关项(don't-care term) 逻辑函数中的一种，它的输出与任何输入的值都没有关系。无关项可以用不同的方式定义。

^② 操作码(opcode) 指明指令的操作和格式的字段。

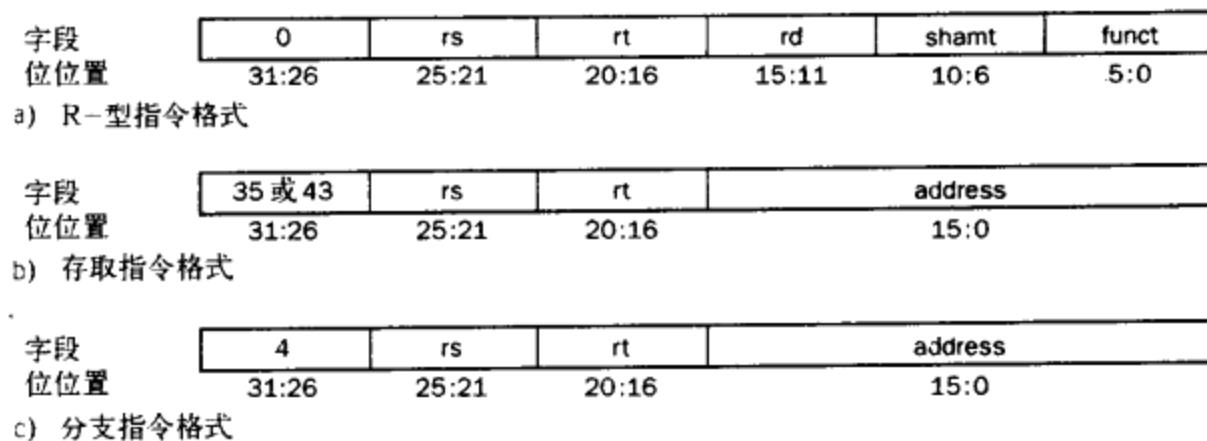


图 5-14 三种指令类型(R 型、存/取数、分支)使用两种不同的指令格式

[跳转指令使用另一种格式，很快会讲到。a)R 型指令的格式，所有操作码为 0。这些指令有 3 个寄存器操作数：rs、rt 和 rd。rs 和 rt 字段为源，rd 字段为目的。功能字段指出 ALU 功能，由前面设计的 ALU 控制解码。这里实现的 R 型指令有 add、sub、and、or 和 slt。shamt 字段只用于移位指令；本章中不予考虑。b)取数(操作码 = 35_{ten})和存数(操作码 = 43_{ten})指令的格式。rs 寄存器作为基址寄存器与 16 位的地址字段相加以得到存储地址。对取数指令，rt 是存放取出的值的目的寄存器。对存数指令，rt 是要存入存储器的数据所在的源寄存器。c)等值分支指令(操作码 = 4)的格式。rs 和 rt 是源寄存器，用于比较是否相等。16 位地址字段进行符号扩展、移位后与 PC 相加以得到分支目标地址]

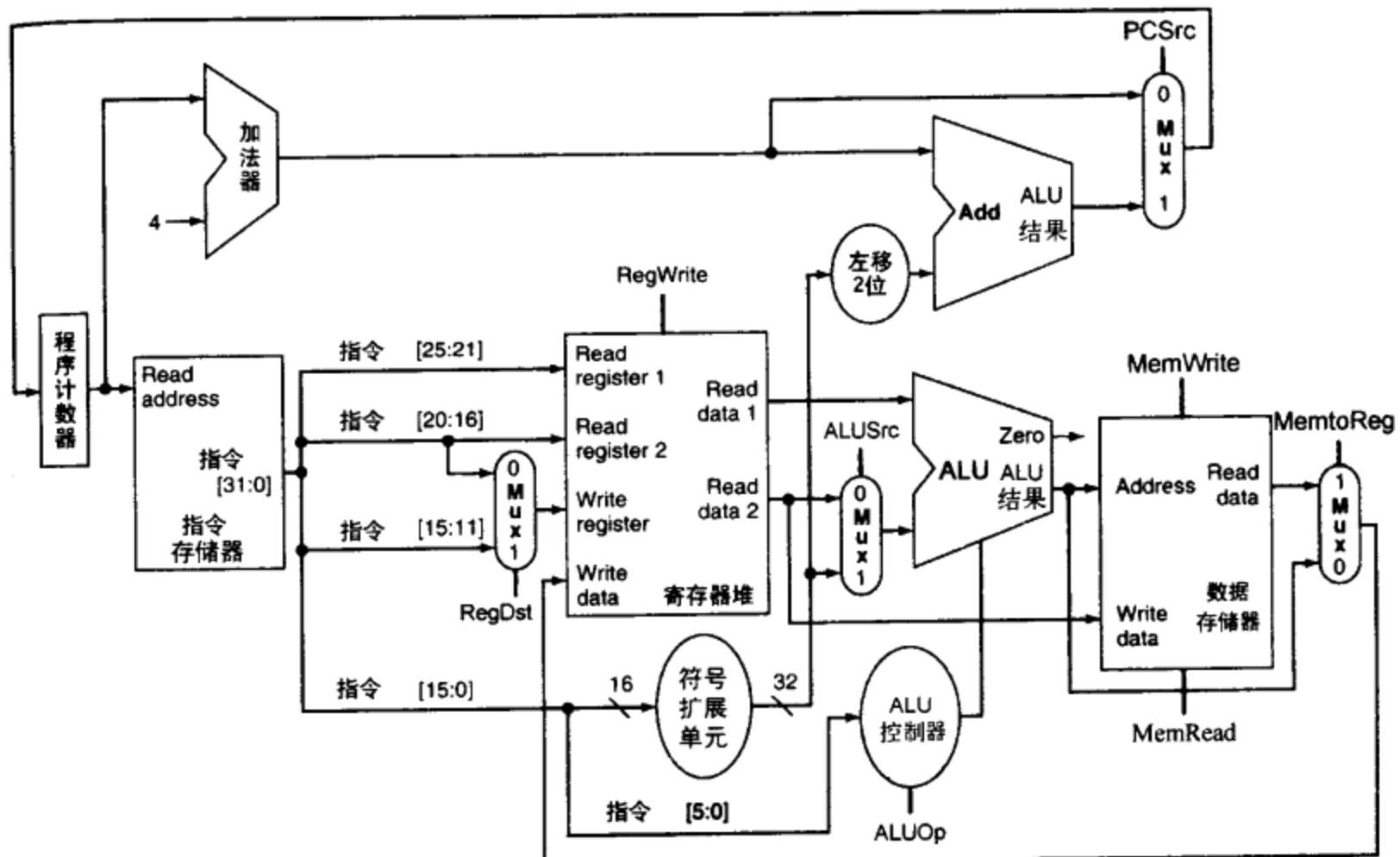


图 5-15 图 5-12 的数据通路，所有需要的多路复用器和控制路线都已标出

[控制线用粗线标出。还加入了 ALU 控制模块。PC 不需要写控制，因为它在每个时钟周期末被一次写入；分支控制逻辑决定给 PC 增值还是写入分支目标地址]

图 5-15 给出了 7 条 1 位的控制线和 2 位的 ALUOp 控制信号的工作原理。我们已经定义了 ALUOp 控制信号如何工作，在确定指令执行过程中如何设置这些控制信号之前，最好非正式地定义一下其他 7 条控制信号如何工作。图 5-16 说明了这 7 条控制线的功能。

信号名	失效时作用	有效时作用
RegDst	对于寄存器写的目的寄存器序号来自于 rt 字段(20:16 位)	对于寄存器写的目的寄存器序号来自于 rd 字段(15:11 位)
RegWrite	无	写寄存器的寄存器输入由读数据的输入值写入
ALUSrc	第二个 ALU 操作数来自第二个寄存器堆输出(读数据 2)	第二个 ALU 操作数是经过符号扩展的指令的低 16 位
PCSrc	PC 的值由加法器的输出替换为 PC + 4	PC 值由加法器输出的分支目的地址替换
MemRead	无	地址输入指定的数据存储器内容放置到读数据输出上
MemWrite	无	地址输入指定的数据存储器内容被写数据输入的值替换
MemtoReg	送往寄存器写数据输入的值来自于 ALU	送往寄存器写数据输入的值来自于数据存储器

图 5-16 七个控制信号的作用

[当 2 路多路复用器的一位控制有效时，多路复用器选择第 1 个输入。否则，若控制为无效状态，多路复用器选择第 0 个输入。请牢记所有的状态单元都以时钟信号作为一个默认的输入，且时钟信号用于写控制中。时钟信号从来不在状态单元之外通过任何门电路，因为这样可能导致时序问题(■附录 B 中有此问题的进一步讨论。)]

知道了每个控制信号的功能，再来看看它们该如何设置。除 PCSrc 控制线外，所有控制信号都可由控制单元只根据指令的操作码字段来确定。仅当指令为相等时分支(由控制单元确定)且用于等值比较的 ALU 的零输出为有效时，PCSrc 控制线才会有效。为生成 PCSrc 信号，需将一个来自控制单元称为分支的信号与 ALU 的零输出信号相“与”。

现在，这 9 个控制信号(图 5-16 的 7 个和 ALUOp 的 2 个)的状态可根据控制单元的 6 个输入信号(即操作码)来设置。图 5-17 给出了有控制单元和控制信号的数据通路。

在为控制单元建立一组方程或一个真值表之前，应该先非正式地定义一下控制功能。由于控制线的状态只由操作码决定，定义在每种操作码值下各控制信号应取 0、1 或是不关心("X")。图 5-18 定义了对应于每种操作码的控制信号的状态；这是根据图 5-12、图 5-16 和图 5-17 直接得来的。

数据通路的操作

根据图 5-16 和图 5-18 包含的信息，可以设计出控制单元的逻辑，但在此之前先看看每条指令是怎样使用数据通路的。接下来的几个图说明了 3 种类型的指令在数据通路上的流动。在图中，有效的控制信号和数据通路部件着重标出。要注意的是，对于多路复用器其控制为 0 时，虽然其控制线路没有着重标出，它也有明确的动作。对于多位控制信号，只要其中任何信号有效，就将被着重标出。

图 5-19 给出了 R-型指令的数据通路操作，如 add \$t1, \$t2, \$t3。将一条指令的执行分为若干步，讨论与每一步相关的数据通路部分。这样比将整个数据通路视为一整块组合逻辑简单一些。R 型指令的执行分为 4 步：

- 1) 从指令存储器中取出指令，PC 增值。
- 2) 寄存器 \$t2 和 \$t3 的内容从寄存器堆中读出。在这一步，主控制单元还计算各控制线应被设置的状态。
- 3) ALU 根据功能码(指令的 5:0 位为指令功能字段)确定 ALU 功能，对从寄存器堆读出的数据进行操作。

4) ALU 的结果被写入寄存器堆, 目标寄存器(\$t1)根据指令的 15:11 位选择。

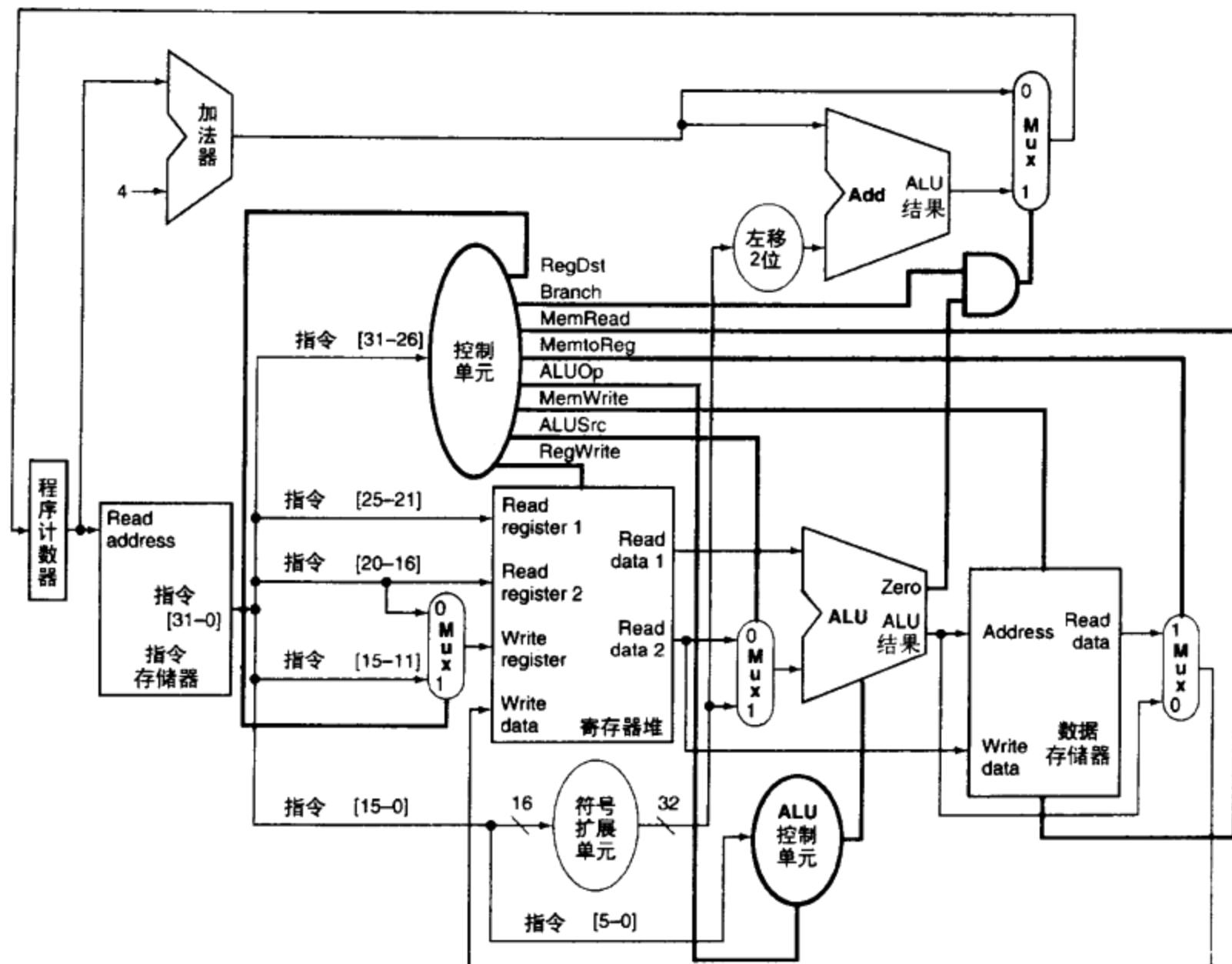


图 5-17 有控制单元的简单数据通路

[控制单元的输入为指令的 6 位操作码字段。控制单元的输出包括 3 个控制多路复用器的 1 位信号(RegDst、ALUSrc 和 MemtoReg), 3 个控制寄存器堆和数据存储器读写的信号(RegWrite、MemRead 和 MemWrite), 一位决定是否可以分支的信号(Branch), 和一个 ALU 的 2 位控制信号(ALUOp)。分支控制信号与 ALU 的零输出一起送入一个与门; 其输出控制着下一个 PC 的选择。注意现在 PCSrc 是一个间接信号, 而不是从控制单元直接得来。所以在后面的图中不用这个信号名称]

可以用和图 5-19 相似的方式描述取字指令如 `lw $t1, offset ($t2)` 的执行。图 5-20 给出了取字时有效的功能单元和控制线路。可将取字指令的执行设想分为 5 步(与将 R 型指令的执行分为 4 步同理):

- 1) 从指令存储器取指, PC 增值。
- 2) 从寄存器堆读出寄存器 \$t2 的值。
- 3) ALU 将从寄存器堆读出的值与符号扩展后的指令低 16 位(offset)值相加。
- 4) 将 ALU 的结果作为数据存储器的地址。
- 5) 存储单元的数据写入寄存器堆; 目标寄存器由指令的 20:16 位(\$t1)指出。

最后, 以同样方式说明相等则分支指令如 `beq $t1, $t2, offset` 的执行。它的操作很像 R 型指令, 但 ALU 的输出用于决定 PC 是增值为 $PC + 4$ 还是置为分支目标地址。图 5-21 给出了执行的 4 步:

指令	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R型	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

图 5-18 控制线路的设置完全取决于指令的操作码字段

[表的第一行对应于 R 型指令 (add、sub、and、or、slt)。对所有这些指令，源寄存器字段都为 rs 和 rt，目的寄存器字段为 rd；这决定了 ALUSrc 和 RegDst 信号如何设置。并且，R 型指令写寄存器 (RegWrite=1) 但是不读写数据存储器。当分支控制信号为 0 时，PC 无条件地由 PC+4 取代；否则，如果 ALU 的零输出也为高，则 PC 由分支目标地址取代。R 型指令的 ALUOp 字段被设为 10，指出 ALU 的控制应该由功能分支生成。这张表的第 2、3 行给出了 lw、sw 指令的控制信号的设置。ALUSrc 和 ALUOp 字段被设为进行地址计算。MemRead 和 MemWrite 被设为进行存储访问。最后，RegDst 和 RegWrite 被设为在取数指令中将结果存入寄存器 rt 中。分支指令与 R 型操作相似，因为它将寄存器 rs 和 rt 寄存器送入 ALU。分支指令的 ALUOp 字段被设为做减法 (ALU 控制 = 01)，以进行等值的测试。注意 RegWrite=0 时 MemtoReg 字段的设置无关紧要：因为寄存器没有被写入，寄存器写数据端口的数据值不被使用。所以，本表最后两行 MemtoReg 的值由于无关而用 X 取代。RegWrite=0 时，RegDst 的值也可用 X 取代。这种无关项必须由设计者加入，因为它依赖于对数据通路工作原理的了解]

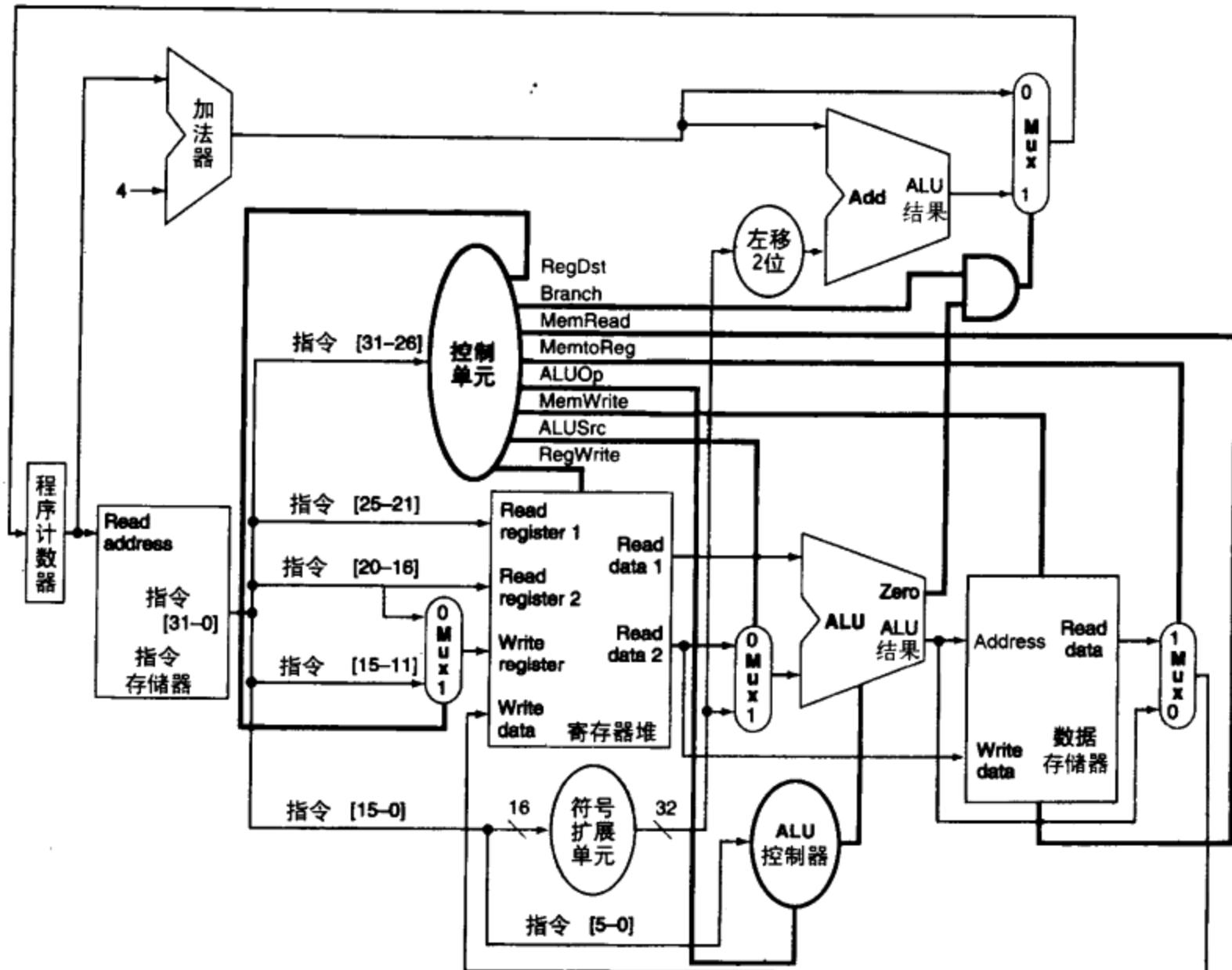


图 5-19 R 型指令例如 add \$t1, \$t2, \$t3 操作的数据通路

[活跃的控制线、数据通路单元和连接用粗线标出]

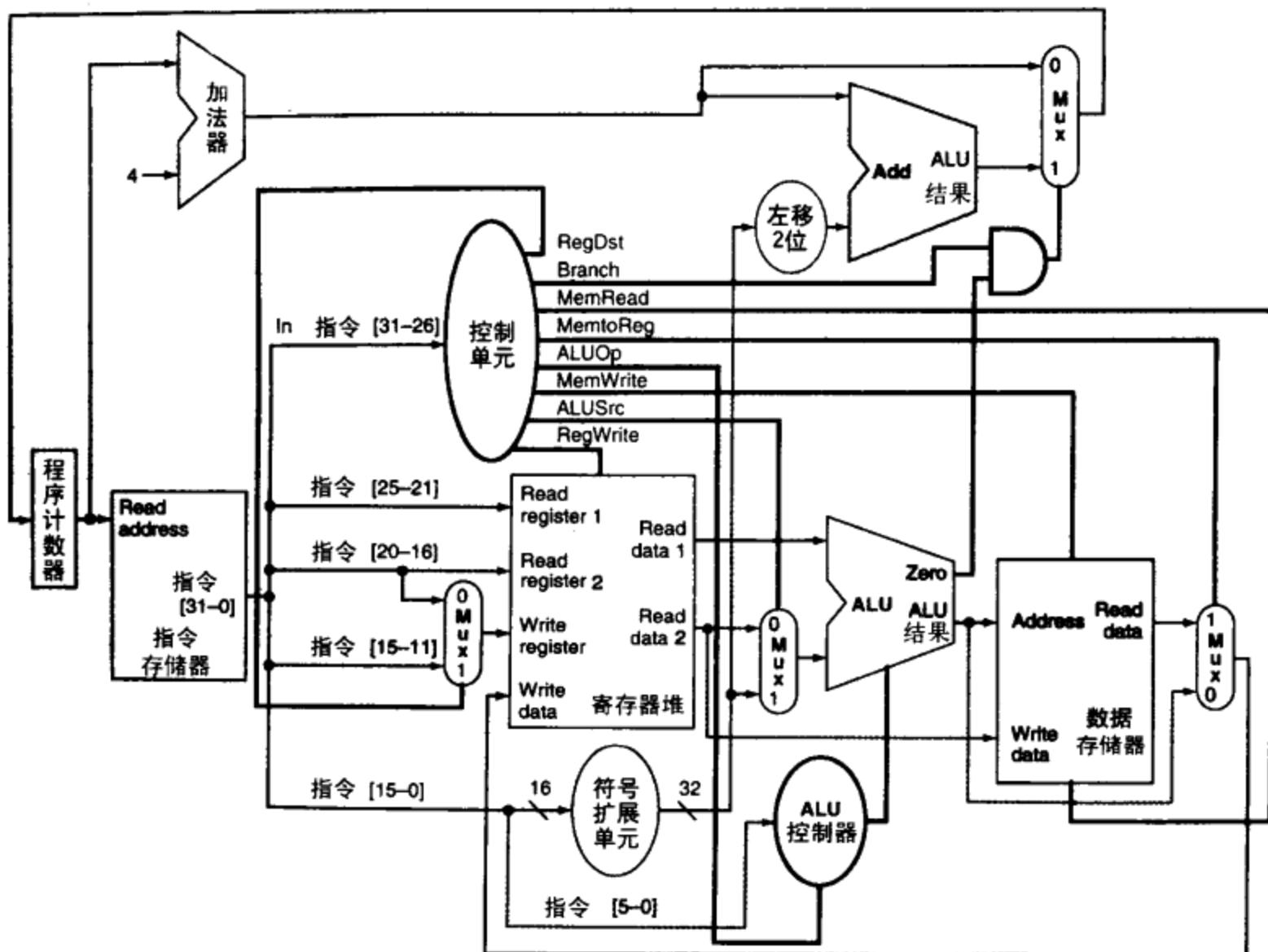


图 5-20 取数指令操作的数据通路

[活跃的控制线、数据通路单元和连接用粗线标出。存数指令的操作与此很类似。主要的区别在于存储器控制将指明要进行写而不是读操作，读出的第二个寄存器的值将作为所存储的数据，并且将不会有将数据存储器的内容写入寄存器堆的动作]

- 1) 从指令存储器取指，PC 增值。
- 2) 从寄存器堆读出寄存器 \$t1 和 \$t2 的值。
- 3) ALU 将从寄存器堆读出的两数相减。PC+4 的值与符号扩展并与左移 2 位后的指令低 16 位值(offset)相加；结果即分支目标地址。
- 4) 根据 ALU 的零输出决定哪个加法器的结果存入 PC 中。

在下面一节，将讨论真正顺序操作的机器，即每一步都占用一个单独的时钟周期。

控制的结束

看完指令的各步操作步骤后，现在继续来讨论控制的实现。控制的功能可由图 5-18 精确定义。其输出为控制线路，输入为 6 位操作码字段，Op[5:0]。这样，可以为每个输出建立一张真值表。因此，根据操作码的二进制编码，可以为每个输出建立一个真值表。

根据这些信息，可以把控制单元包括所有输出以及以操作码位作为输入的逻辑综合描述在一张大的真值表上，如图 5-22 所示。它完整地描述了控制功能，可以自动地转换为门电路实现。在附录 C 的 C.2 节描述了这最后一步。

现在再加上跳转指令，看看怎样将基本的数据通路和控制扩展，以实现指令集中的其他指令。

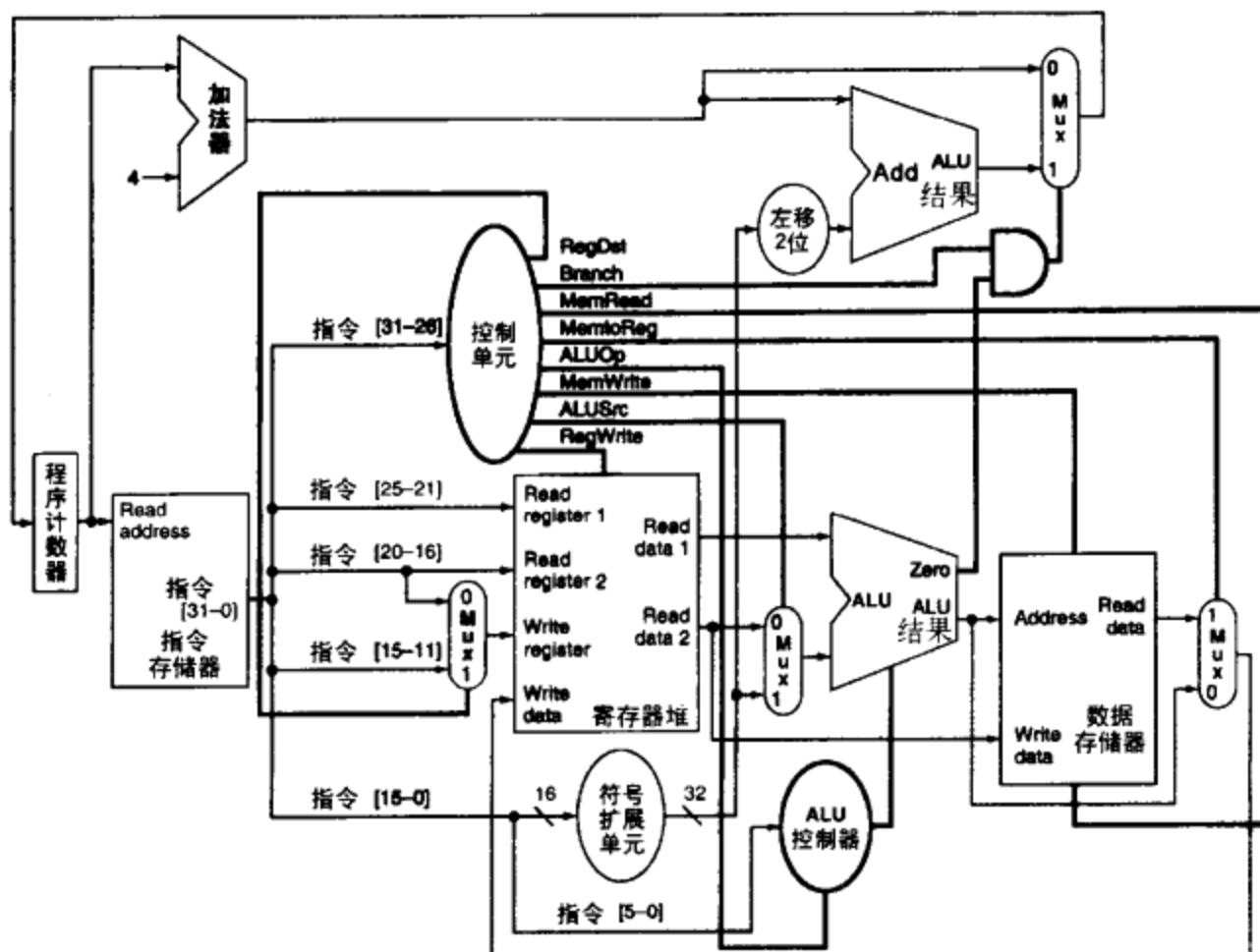


图 5-21 相等则分支指令的数据通路

[活跃的控制线、数据通路单元和连接用粗线标出。在用寄存器堆和 ALU 进行比较操作之后，ALU 的零输出将用于在两种可能的下一 PC 中选择其一]

输入或输出	信号名	R型	lw	sw	beq
输入	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
输出	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

图 5-22 简单的单周期实现^①的控制功能完全由此真值表说明

[表的上半部分为对应于决定控制输出的设置的四个操作码的输入信号组合。(前面讲过 Op[5:0]对应于指令的 31:26 位，即操作码字段。)表的下半部分为输出。这样，RegWrite 输出在两种组合的输入时被激活。如果只考虑这张表中的四个操作码，可以通过用输入部分的无关项简化真值表。比如，可以由表达式 $\overline{Op5} \cdot \overline{Op2}$ 确定一条 R 型指令，因为这已经足够将 R 型指令与 lw、sw 和 beq 指令区分开。不用这种简化，是因为其他 MIPS 操作码用于完全的实现中]

① 单周期实现(single-cycle implementation) 也称为单时钟周期实现，是指指令在一个时钟周期内执行。

例题 跳转的实现

图 5-17 给出了许多第 2 章提到的指令的实现。有一类指令没有给出，即跳转指令。请对图 5-17 的数据通路和控制进行扩展，以包含跳转指令。请指出新控制线路的状态设置。

解 跳转指令有点像分支指令，但它以不同的方式计算目标 PC，且是无条件的。与分支指令一样，跳转地址的低两位恒为 00_{two}。这个 32 位地址的次低 26 位来自指令的 26 位立即数字段，如图 5-23 所示。新的 PC 值的高 4 位来自跳转指令的 PC 加 4。所以，实现跳转指令即将如下各位连接为 PC：

- 当前 PC + 4 的最高 4 位(即顺序下条指令地址的 31:28 位)
- 跳转指令的 26 位立即数字段
- 低位 00_{two}



图 5-23 跳转指令的格式(操作码 = 2)

[跳转指令的目标地址由当前的 PC + 4 的高 4 位与跳转指令的 26 位地址字段和加入的低两位 00 连接而成]

图 5-24 给出了在图 5-17 基础上加上的跳转指令的控制。为了在 PC + 4、分支目标 PC 和跳转

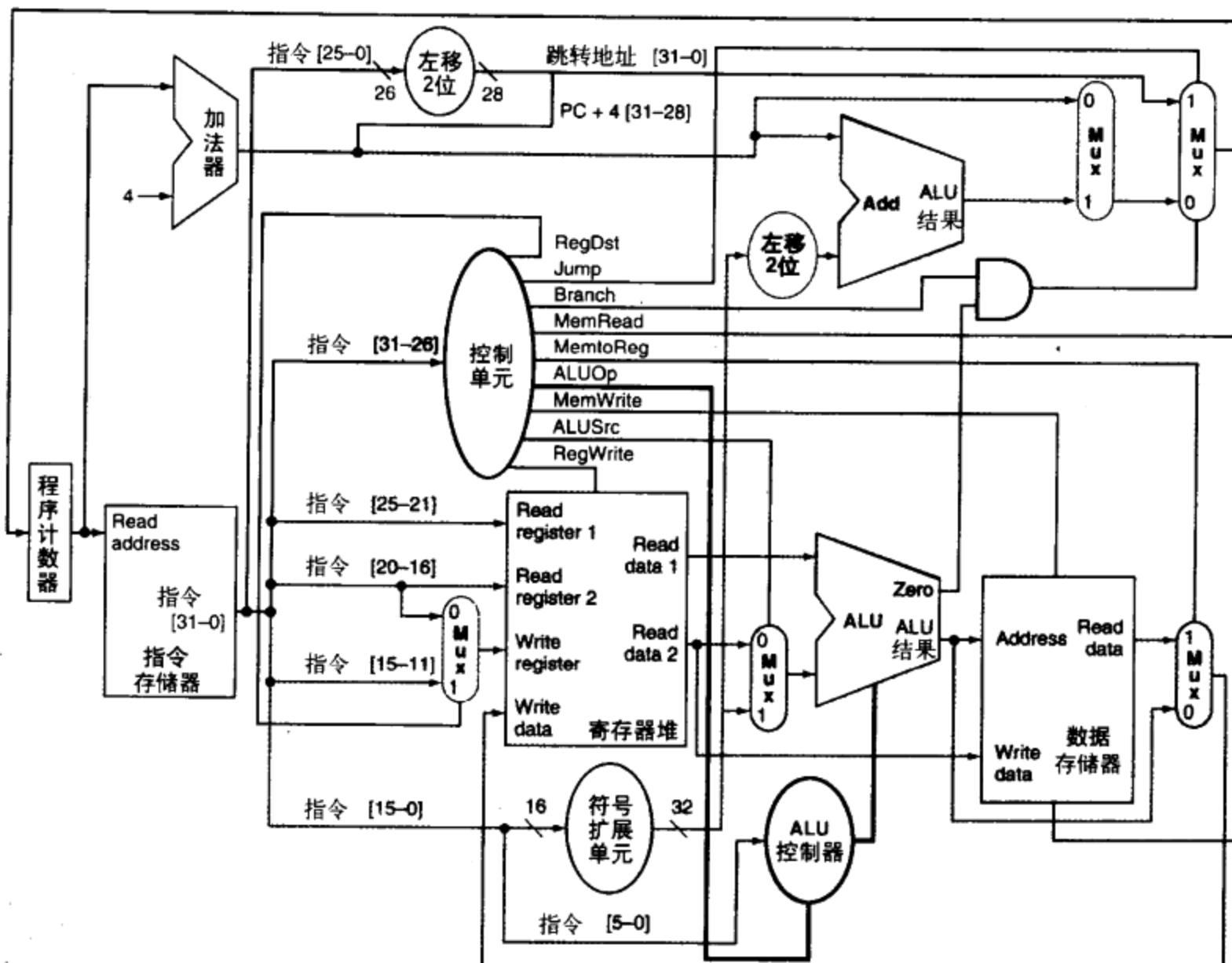


图 5-24 扩展简单的控制和数据通路以处理跳转指令

[在此加入一个多路复用器(右上角)以在跳转目的地址与分支目的地址或顺序递增的下一条指令地址之间进行选择。这个多路复用器由跳转控制信号控制。将跳转指令的低 26 位左移 2 位，其实是加入低两位 00 作为低位，再将 PC + 4 的高 4 位作为高位连接起来，即得到 32 位的跳转目的地址]

目标 PC 中选择新 PC 值的来源，还加上了一个多路复用器。这个多路复用器需要一个控制信号，称为跳转(Jump)。只有当操作码为 2，即指令为跳转指令时，该控制信号才有效。

5.4.3 为什么不使用单周期实现方式

虽然单周期设计也可以正确地运作，但现代设计中并不采取这种方式，因为它的效率太低。究其原因，是在单周期设计中，时钟周期对所有指令等长，即 CPI(参见第 4 章)为 1。当然，时钟周期要由计算机中可能的最长路径决定。这条路径几乎肯定是一条取数指令，它顺序使用 5 个功能单元：指令存储器、寄存器堆、ALU、数据存储器、寄存器堆。虽然 CPI 为 1，单周期实现方式的总体性能并不见得很好，因为某些指令类型本来可在更短时钟周期内完成。

例题 单周期计算机的性能

假设在这种实现方式中，主要功能单元的操作时间如下：

- 存储单元：200 ps
- ALU 和加法器：100 ps
- 寄存器堆(读或写)：50 ps

假设多路复用器、控制单元、访问 PC、符号扩展单元和线路没有延迟，下面实现方法中哪个更快？快多少？

1) 每条指令在一个固定长度的时钟周期内完成。

2) 每条指令在一个可变长度的时钟周期内完成，时钟周期长度仅为指令所需。(这样的方法并不可行，但它能显示出所有指令都在同样长度的时钟周期完成的缺点。)

为比较性能，假设指令的组成比例为：25% 取数，10% 存数，45% ALU 指令，15% 分支指令，5% 跳转。

解 首先来比较 CPU 执行时间，在第 4 章提到

$$\text{CPU 执行时间} = \text{指令数目} \times \text{CPI} \times \text{时钟周期长度}$$

因为 CPI 肯定为 1，上式可简化为

$$\text{CPU 执行时间} = \text{指令数目} \times \text{时钟周期长度}$$

由于两种方案的指令数目和 CPI 都相等，只需找出它们的时钟周期长度即可。不同指令类型的关键路径如下：

指令类型	指令类型用到的功能单元				
	取指令	访问寄存器	ALU	访问寄存器	
R型	取指令	访问寄存器	ALU	访问寄存器	
取字	取指令	访问寄存器	ALU	访问存储器	访问寄存器
存字	取指令	访问寄存器	ALU	访问存储器	
转移	取指令	访问寄存器	ALU		
跳转	取指令				

根据这些关键路径，可计算出各指令类型要求的时间长度：

指令类型	指令存储器	读寄存器	ALU 操作	数据存储器	写寄存器	总共
R型	200	50	100	0	50	400 ps
取字	200	50	100	200	50	600 ps
存字	200	50	100	200		550 ps
分支	200	50	100	0		350 ps
跳转	200					200 ps

对于所有指令用一个固定时钟周期的计算机，时钟周期长度将由最长的指令决定，在此为 600 ps。（这只是一个近似的计时，因为计时模型很简单。实际上，现代数字系统的计时很复杂，通常可以从一个时钟周期中借一点时间给下一个时钟周期使用。）

对时钟周期长度可变的计算机，其长度将在 200 ps 至 600 ps 之间变化。根据上面的信息和指令频度分布情况，可得出这种计算机的平均时钟周期长度。

于是，一条指令的平均执行时间为：

$$\text{CPU 时钟周期} = 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% = 447.5 \text{ ps}$$

由于可变时钟周期的实现方法中平均时钟周期较短，它当然较快。来计算一下性能比：

$$\begin{aligned}\frac{\text{CPU 性能}_{\text{可变时钟}}}{\text{CPU 性能}_{\text{固定时钟}}} &= \frac{\text{CPU 执行时间}_{\text{固定时钟}}}{\text{CPU 执行时间}_{\text{可变时钟}}} \\ &= \left(\frac{\text{IC} \times \text{CPU 时钟周期}_{\text{固定时钟}}}{\text{IC} \times \text{CPU 时钟周期}_{\text{可变时钟}}} \right) = \frac{\text{CPU 时钟周期}_{\text{固定时钟}}}{\text{CPU 时钟周期}_{\text{可变时钟}}} \\ &= \frac{600}{447.5} = 1.34\end{aligned}$$

可见可变时钟周期的实现方式快了 1.34 倍。遗憾的是，为每个指令类型实现一个可变的时钟周期非常困难，而且所导致的额外开销可能得不偿失。在下一节中，将介绍另一种方法，它通过在更短的时钟周期做较少的工作，改变每种指令类型所用的时钟周期数。 ■

使用固定时钟周期的单周期设计的代价是很大的，但对于小指令集可能可以接受。历史上看，具有非常简单的指令集的早期计算机的确使用了这项技术。可是若要实现浮点单元和包含更复杂指令的指令集，这样的单周期设计根本不能胜任。下面给出 CD 中 ■ For More Practice 练习 5.4 中的一个例子。

因为必须假定时钟周期等于所有指令中最坏情况下的延迟，因此在不改进最坏时钟周期时间情况下，不能使用减少一般操作延迟的技术。所以，单周期的实现方法违反了关键的设计原则，即加速完成常用操作。另外，在单周期实现中，一个时钟周期内每个功能单元只能使用一次。因此，某些功能单元必须有副本，这就增加了实现的代价。从性能和硬件成本来说单周期设计都是低效的！

为避免这些困难，可以使用具有更短时钟周期的实现技术——时钟周期由基本功能单元延时决定——并且每条指令需要多个时钟周期完成。下一节就将讨论这种替代实现方案。在第 6 章中，将要介绍另一种实现技术，称为流水线，流水线使用类似单周期数据通路的通路，但是更有效率。通过重叠执行多条指令，流水线获得更高的效率，提高了硬件利用率并且提高了系统性能。只需对处理器使用的高层概念感兴趣的读者，掌握本节的内容足以阅读第 6 章的概述小节，并且可以理解流水线处理器的基本功能。要是想要理解硬件实际如何实现控制，还要继续学习下面的章节！

自测

参考图 5-22 中的控制信号。图中任何控制信号都能被另一个的非替换吗？如果可以，是否可以在不添加非门的情况下，使用一个信号作为另一个控制信号。

5.5 多周期实现方案

在前面的一个例子中，曾根据所需进行的功能单元的操作，将一条指令的执行分解为一系列步骤。可以用这些步骤来生成一个 **多周期实现**^① 方案。在这个方案中，指令的每一步将占用一个时钟周期。一个功能单元可以在一条指令的执行过程中使用多次，只要是在不同周期中。这种共享可减少所需硬件数量。允许指令的执行占用不同周期数和功能单元可在单指令执行过程中共享，是多周期设计的主要优点。图 5-25 给出了多周期数据通路的大致轮廓。和图 5-11 的单周期数据通路相比，可发现以下区别：

① 多周期实现(multicycle implementation) 也称为多时钟周期实现，是指一条指令的执行要经过多个时钟周期。

- 指令和数据使用同一个存储器单元。
- 只有一个 ALU，而不是一个 ALU 和两个加法器。
- 每个主要的功能单元都加上了一个或多个寄存器存储输出值，以便在后面的时钟周期中使用。

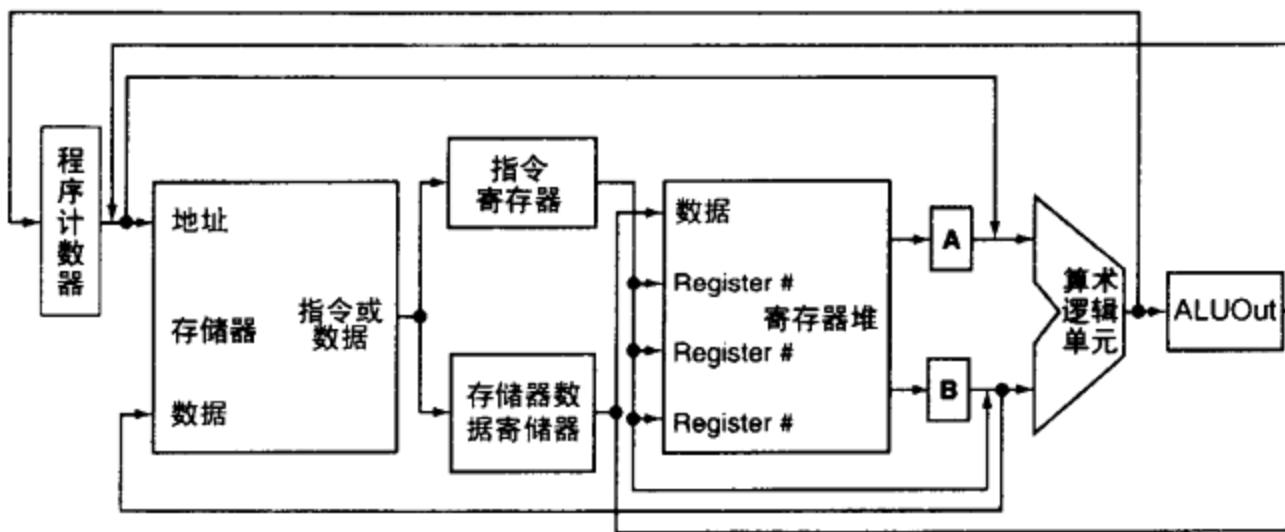


图 5-25 多周期数据通路的高层视图

[这张图画出了数据通路的关键部件：共享的存储单元、一个各指令共享的 ALU 和各共享单元之间的联系。共享功能单元的使用要求增加或扩展多路复用器，以及用于在同一条指令不同时钟周期间存储数据的临时寄存器。附加的寄存器有指令寄存器(IR)、存储器数据寄存器(MDR)、A、B 和 ALUOut]

一个时钟周期结束时，在以后的周期中将要用到的所有数据都必须存储在状态单元中。以后时钟周期中随后的指令将要用到的数据被存入一个程序员可见的状态单元中(即寄存器堆、PC 或存储器)。相反，在以后的周期中同一指令要用到的数据必须存入这些附加寄存器中。

这样，附加寄存器究竟放在哪里将由两个因素决定：哪些组合单元适合在单个周期中使用，以及哪些数据将在指令执行过程中后面的各个周期中使用。在这个多周期设计中，假设一个时钟周期内最多只能完成下列操作之一：一次访存、一次寄存堆访问(2 次读或 1 次写)或一个 ALU 操作。于是这三个功能单元的任何一个(存储器、寄存器堆或 ALU)产生的数据必须存储在一个临时寄存器中，以供后面的周期使用。如果没有被缓存，则可能出现周期竞争，导致使用不正确的值。

为满足这些要求，加入下面的临时寄存器：

- 指令寄存器(IR)和存储器数据寄存器(MDR)分别用于存储从存储器读取的指令和数据的输出。使用两个独立的寄存器，是因为在同一周期中两者的值都要被用到，稍后会进一步说明。
- 寄存器 A、B，用于存储从寄存器堆中读出的寄存器操作数。
- 寄存器 ALUOut，用于存储 ALU 的输出。

除 IR 外，所有寄存器只在相邻两时钟周期之间存储数据，所以它们不需要写控制信号。IR 需保存指令至其执行结束，所以需要一个写控制信号。这个区别将在研究了每条指令的各个时钟周期后更加清楚。

由于一些功能单元要为不同目的所共享，所以需要添加多路复用器，并扩展已有的多路复用器。比如，由于一个存储器既用于存储指令也用于存储数据，就需要一个多路复用器来为一个存储地址选择数据来源，即 PC(用于存取指令)还是 ALUOut(用于存取数据)。

用一个 ALU 代替单周期数据通路中的三个 ALU，意味着这一个 ALU 必须能接收原来三个 ALU 的所有输入。处理这些输入使数据通路有以下变化：

- 1) 给 ALU 的第一个输入添加了一个多路复用器，它在寄存器 A 和 PC 间进行选择。
- 2) ALU 第二个输入的多路复用器由 2 路改为 4 路，其新加的两个输入为常数 4(用于 PC 增值)和符号扩展及移位后的偏移量字段(用于计算分支地址)。

图 5-26 详细画出了加上了这些多路复用器后的数据通路。通过引入一些寄存器和多路复用器得以将存储单元数目从 2 减为 1，并取消了 2 个加法器。由于寄存器和多路复用器都很小，这样做可以大大降低硬件成本。

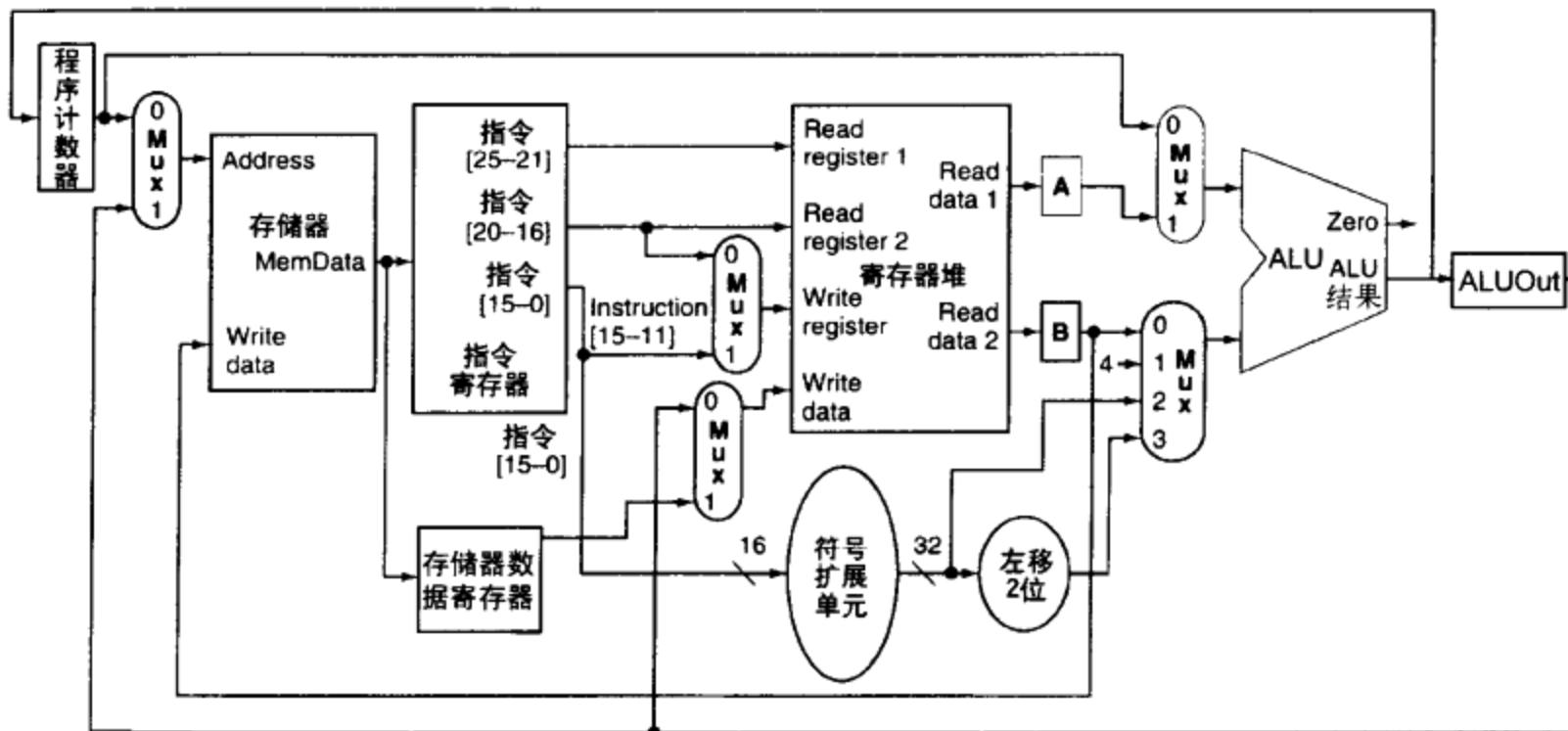


图 5-26 MIPS 的多周期数据通路处理基本指令

[虽然这条数据通路支持 PC 的普通增值，为了分支和跳转指令还需要增加一些连接和一个多路复用器；将在后面很快加入这些部件。比起单周期的数据通路，增加了一些寄存器(IR、MDR、A、B、ALUOut)，一个用于存储地址的多路复用器，一个用于 ALU 最高输入的多路复用器以及将 ALU 底部输入的多路复用器扩展为 4 个输入。这些小的增加可以去掉两个加法器和一个存储单元]

图 5-26 的数据通路用多个时钟周期完成一条指令，所以需要一套不同的控制信号。程序员可见的状态单元(PC、存储器和寄存器)和 IR 需要写控制信号。存储器还需一个读信号。单周期数据通路的 ALU 控制单元(参见图 5-13 和附录 C)也可在此用于控制 ALU。最后，每个 2 输入的多路复用器都需要一根控制线，4 输入的多路复用器需要 2 根控制线。图 5-27 给出了图 5-26 中的数据通路加上这些控制线后的情况。

为支持分支和跳转指令，多周期数据通路还要添加部件；完成这些添加后，再看看指令的执行顺序如何确定，以及如何生成数据通路的控制。

对于跳转和分支指令，PC 有三种可能来源：

- 1) ALU 的输出，取指时为 $PC + 4$ 。这个数应直接写入 PC。
- 2) 寄存器 ALUOut，存储的是计算出的分支目标地址。
- 3) 指令寄存器(IR)的低 26 位左移 2 位与 PC 增值后的高 4 位相连，这是跳转指令时 PC 的来源。

在单周期控制的实现中可以看出，PC 的写入可以是条件的或无条件的。在普通的增值和跳转时，PC 的写入是无条件的。对于条件分支指令，只有在两个指定寄存器相等时 ALUOut 的值才会取代增值的 PC。所以，控制部分需两条独立的 PC 写控制信号，称为“PCWrite”和“PCWrite-Cond”。

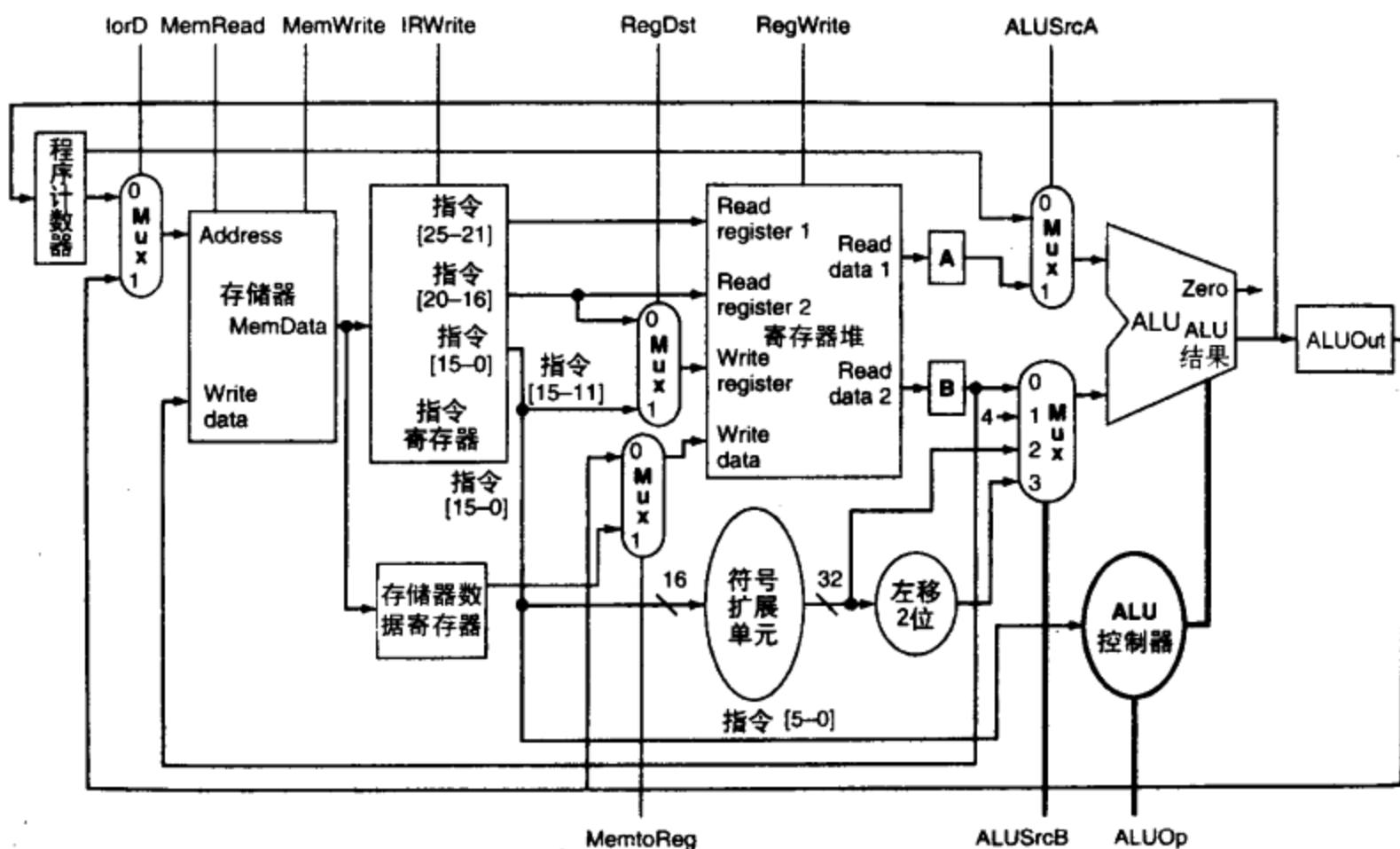


图 5-27 图 5-26 的数据通路加上控制线路

[ALUOp 和 ALUSrcB 信号为 2 位控制信号，其他控制信号为 1 位。寄存器 A 和 B 都不需要写控制信号，因为它们的内容只在写入的下一个周期读出。加入了存储数据寄存器，以便使取数时存储从存储器读出的数据。从存储器读出的数据不能直接写入寄存器堆，因为单时钟周期内不够访问存储器外加寄存器堆写。MemRead 信号被移到存储器单元的顶部以简化本图。分支指令的全部数据通路和控制线路将很快在后面加入]

这两个控制信号需要与 PC 写控制相连。和单周期数据通路一样，我们将使用一些门电路，从 PCWrite、PCWriteCond 和用于检验 beq 的两个寄存器操作数是否相等的 ALU 零输出信号生成 PC 写控制信号。为了决定在条件分支中 PC 是否被写入，将 ALU 的零输出信号与 PCWriteCond 相与，然后将与门的输出结果与非条件的 PC 写信号 (PCWrite) 相或。或门的输出直接作为 PC 的写控制信号。

图 5-28 为完整的多周期数据通路和控制单元，包括为实现更新 PC 而添加的控制信号和多路复用器。

在讨论各指令的执行步骤前，让我们先大致看看所有控制信号的作用（就像在图 5-16 一样）。图 5-29 说明了各个控制信号被设置为有效或无效状态时所起的作用。

细节：为了减少连接功能单元用的信号线，设计者可以使用共享总线。共享总线是连接多个单元的一组线；一般情况下，它有多个向总线传输数据的源，也有多个总线数据值的读者。就像减少数据通路的功能单元一样，可以通过共享总线来减少连接多个单元的总线数目。比如，ALU 有 6 个输入来源；然而任一时刻只需要其中两个。这时就可以用一对总线来传输送给 ALU 的数据值。设计者可以使用一根共享总线并保证任意时刻总线只有一个驱动源，而无须在 ALU 前面加一个大的多路复用器。虽然这样可以节省信号线，却需要同样数目的控制线路以控制总线传输什么数据。使用这样的总线结构的主要缺点是可能会对性能产生影响，因为总线不会像点对点连接那么快。

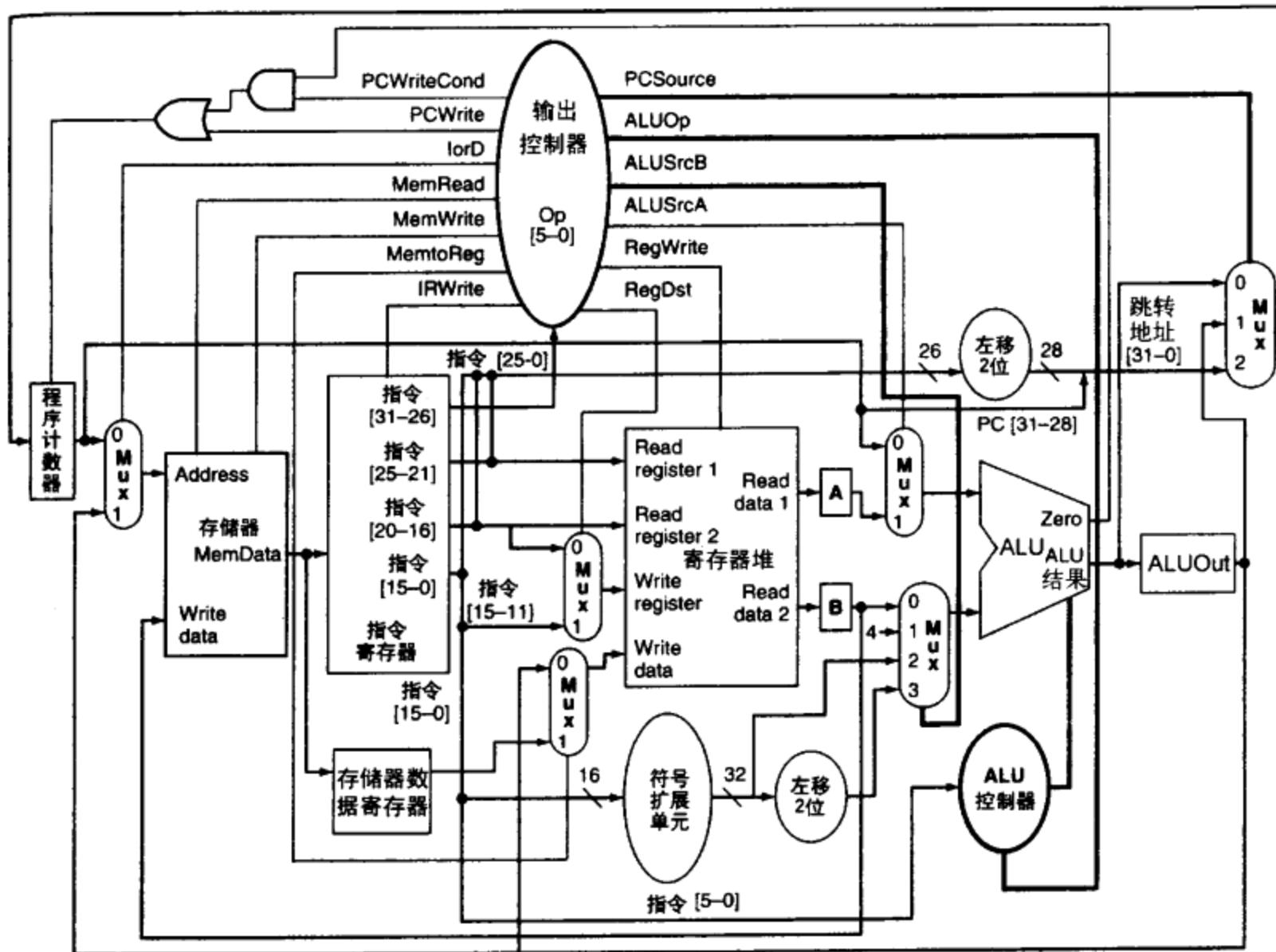


图 5-28 多周期实现的完整的数据通路和必要的控制线路

[图 5-27 的控制线路被加到控制单元，并且包含了改变 PC 所需的控制和数据通路单元。相对于图 5-27 而言，主要增加的包括一个用于选择新 PC 值来源的多路复用器(右上角)；用于合并 PC 写信号(左上角)的两个门；和控制信号 PCSOURCE、PCWRITE、PCWRITCOND。PCWRITCOND 信号用来判定条件分支是否成立。这也包括对跳转指令的支持]

1 位控制信号动作

信号名	无效时作用	有效时作用
RegDst	写寄存器在寄存器堆中的目的编号来自于 rd 字段	写寄存器在寄存器堆中的目的编号来自于 rd 字段
RegWrite	无	写数据输入的值写入到被写寄存器号选择的通用寄存器
ALUSrcA	ALU 第一个操作数是 PC	ALU 第一个操作数来自寄存器 A
MemRead	无	将地址输入指定位置的存储器内容放到存储器数据输出上
MemWrite	无	将地址输入指定位置的存储器内容替换为写数据输入的值
MemtoReg	送往寄存器堆写数据输入的值来自于 ALUOut	送往寄存器堆写数据输入的值来自于 MDR
IorD	PC 提供地址给内存单元	ALUOut 提供地址给存储器单元
IRWrite	无	存储器的输出写入 IR
PCWrite	无	写 PC；源由 PCSOURCE 控制
PCWRITCOND	无	如果 ALU 的 Zero 输出也被激活，写 PC

图 5-29 设置图 5-28 的控制信号所产生的动作

2 位控制信号动作		
信号名	值(二进制)	作用
ALUOp	00	ALU 执行加操作
	01	ALU 执行减操作
	10	指令的功能字段决定 ALU 操作
ALUSrcB	00	ALU 的第二个输入来自寄存器 B
	01	ALU 的第二个输入是常数 4
	10	ALU 的第二个输入是经过符号扩展的 IR 的低 16 位
	11	ALU 的第二个输入是经过符号扩展的 IR 的低 16 位左移 2 位的值
PCSource	00	ALU 的输出($PC + 4$)送去写入 PC
	01	ALUOut 的内容(分支目标地址)送去写入 PC
	10	跳转目标地址($IR[25: 0]$ 左移 2 位后与 $PC + 4[31: 28]$ 连接)送去写入 PC

图 5-29 (续)

[上表描述 1 位控制信号，下表描述 2 位控制信号。只有那些影响多路复用器的控制线路设为无效时有动作。这类似于图 5-16 的单周期数据通路，但增加了一些新的控制线路(IRWrite、PCWrite、PCWriteCond、ALUSrcB 和 PCSource)，去掉了一些不用的或被取代的线路(PCSrc、分支和跳转)]

5.5.1 将指令的执行分到各个时钟周期

图 5-28 给出了数据通路，现在应该看看多周期指令执行在每个时钟周期内的动作，因为这决定了可能需要附加什么控制信号，以及它们的设置。将指令的执行分解到各个时钟周期，目的是最大化性能。可以把任意指令的执行分解为一系列步骤，每步一个时钟周期，基本上长度相当。比如，规定每一步最多只能包含一次 ALU 操作，或一次寄存器堆访问，或一次存储器访问。在此规定下，时钟周期应当至少长于这些操作中最长的操作所花费的时间。

前面讲过，每个时钟周期结束时，要在后面的周期中用到的所有数据必须存入一个寄存器，可以是一个主状态单元(如 PC、寄存器堆或存储器)，一个在每个时钟周期写入的临时寄存器(如 A、B、MDR 或 ALUOut)或一个有写控制的临时寄存器(如 IR)。并且，由于我们的设计基于边缘触发型，可以继续读取寄存器的当前值；新的值直到下一个时钟周期到来时才可能出现。

在单周期数据通路中，每条指令的执行使用一套数据通路单元。许多数据单元顺序操作，以某个其他单元的输出作为自己的输入。一些数据通路单元并行操作；比如，PC 的增值和指令的读取同时进行。多周期数据通路的情况类似。同一步内列出的所有操作在一个时钟周期内并行完成，随后各步的操作在不同周期内顺序完成。而一次 ALU 操作、一次存储器访问和一次寄存器堆访问的限制也决定了一步内最多可以完成的内容。

注意：需要把读写 PC 或一个独立的寄存器与读写寄存器堆区分开来。前者的读写只占一个时钟周期的一部分，而寄存器堆的读写额外需要一个周期。此区别的原因是相对于单独的寄存器而言，寄存器堆有额外的控制和访问开销。为了缩短时钟周期，访问寄存器堆需要多个时钟周期完成。

可能的指令执行步骤及动作如下。每条 MIPS 指令需要其中 3 到 5 步：

步骤 1：取指

从存储器中取出指令并计算下条指令的地址：

```
IR <= Memory[PC];
PC <= PC + 4;
```

操作：将 PC 作为地址传送给存储器，进行读取，将指令写入并存储在指令寄存器 (IR)。同时，将 PC 加 4。我们使用 Verilog 符号“ \leftarrow ”，此符号表明计算右边的式子然后赋值，它有效说明了时钟周期期间硬件执行的情况。

为实现这一步，需要将控制信号 MemRead 和 IRWrite 设为有效，将 IorD 置 0 以选择 PC 作为地址来源。为实现 PC 加 4，要将 ALUSrcA 信号置 0(将 PC 送往 ALU)，ALUSrcB 信号置 01(将 4 送往 ALU)，ALUOp 置 00(使 ALU 进行加法)。最后，还要将 PC 源设置为 00 并且设置 PCWrite，以把增值后的指令地址存入 PC。PC 的增值和指令存储器的访问可以并行进行。新 PC 值直到下一个时钟周期才可见。(增值后的 PC 也要存入 ALUOut，但这个动作没有什么影响。)

步骤 2：指令译码和读取寄存器

在上一步和这一步，还不知道指令的具体内容是什么，所以只能进行那些针对所有指令都需要进行的操作(如第一步中的取指)或没有坏影响的操作。于是，在这一步中可以读取指令字段 rs 和 rt 指定的寄存器的内容，因为即使不必要这样做也没有什么害处。从寄存器堆读出的数据值可能在后面的步骤中要用到，所以我们将它们以寄存器堆中读出并将它们存入临时寄存器 A 和 B 中。

还要用 ALU 计算分支目标地址，这样做也没有什么害处。因为若发现指令不是分支指令则忽略计算结果即可。计算出的分支目标地址存于 ALUOut 中。

尽早执行这些“良性”操作的好处是减少指令执行所用的时钟周期数。指令格式的规整便于这些“良性”操作的尽早完成。比如，若指令有两个寄存器输入，则它们总是位于 rs 和 rt 字段；若指令是分支指令，则其偏移量总是低 16 位：

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend (IR[15-0]) << 2);
```

操作：访问寄存器堆，读取寄存器 rs 和 rt 的内容并将结果存入寄存器 A 和 B。由于寄存器 A 和 B 在每个时钟周期都被重新写入，所以可以在每个时钟周期读出寄存器堆的内容存入寄存器 A、B。这一步中还计算分支目标地址并存入 ALUOut，若指令为分支指令则下一个时钟周期将用到此 ALUOut。这个操作要求给 ALUSrcA 置 0(以将 PC 送入 ALU)，ALUSrcB 置 11(以将符号扩展并移位后的偏移量送入 ALU)，ALUOp 置 00(以使 ALU 作加法)。读取寄存器堆和计算分支目标地址并行执行。

在这个时钟周期以后，就可以根据指令内容来决定要进行的具体操作了。

步骤 3：指令的执行，存储地址的计算或分支的完成

这是数据通路操作中由指令类型决定的第一个时钟周期。在所有情况下，ALU 对前面准备好的操作数进行操作，根据指令类型执行四种功能之一种。根据指令的类型说明要进行的操作：

存储器访问：

```
ALUOut <= A + sign-extend (IR[15:0]);
```

操作：ALU 将操作数相加，得到存储地址。这要求将 ALUSrcA 置 1(使 ALU 的第一个输入为寄存器 A)，ALUSrcB 置 10(使符号扩展单元的输出作为 ALU 的第二个输入)。需将 ALUOp 信号置为 00(以使 ALU 作加法运算)。

算术逻辑指令(R型)：

```
ALUOut <= A op B;
```

操作：ALU 对前面周期中从寄存器堆读出的数据进行功能码所指定的操作。这要求将 ALUSrcA 置 1，ALUSrcB 置 00(使寄存器 A 和 B 作为 ALU 的输入)。ALUOp 信号需置为 10(以由

功能字段决定 ALU 控制信号的设置)。

分支:

```
if (A == B) PC <= ALUOut;
```

操作: ALU 将前面步骤中从寄存器堆读出的两寄存器进行相等比较。ALU 的零信号输出用来决定是否分支。这要求将 ALUSrcA 置 1, ALUSrcB 置 00(使寄存器堆的输出成为 ALU 的输入)。ALUOp 信号需置为 01(使 ALU 做减法)以进行等值的测试。若 ALU 的零输出信号有效, 则需将 PCWriteCond 信号置有效, 以修改 PC。通过将 PCSOURCE 置 01, 写入 PC 的内容将来自 ALUOut, 而 ALUOut 存储的正是前面周期中计算出来的分支目标地址。对于条件分支指令, 实际上有两次写 PC: 一次来自 ALU 的输出(在指令译码/读取寄存器时), 一次来自 ALUOut(在分支完成时)。最后写入 PC 的内容用于下一次取指。

跳转:

```
# [x, y]是字段 x 和 y 连接的 Verilog 标记  
PC <= {PC[31:28], (IR[25:0], 2'b00)};
```

操作: PC 由跳转目的地址取代。PCSOURCE 设置为指示将跳转地址直接作为 PC 的输入, PCWrite 置为有效, 以将跳转地址写入 PC。

步骤 4: 存储器的访问或 R 型指令的完成

在这一步中, 存或取指令访问存储器, 算术逻辑指令写结果。当数据值从存储器中取出后, 被存入存储器的数据寄存器 (MDR), 它将用于下一个时钟周期。

存储访问指令:

```
MDR <= Memory[ALUOut];
```

或

```
Memory[ALUOut] <= B;
```

操作: 若为取数指令, 从存储器中读出一个数据字并写入 MDR。若为存数指令, 则将数据写回存储器。在这两种情况下, 所用的存储器地址都是在前面的步骤中计算出并存于 ALUOut 中的地址。对于存数指令, 源操作数存于寄存器 B 中。(实际上 B 被读取了两次, 一次在第 2 步, 一次在第 3 步。幸好, 两次读取的数据值相同, 因为寄存器号——存于 IR 中, 用于读取寄存器堆——没有变。)信号 MemRead(用于取数)或 MemWrite(用于存数)将置为有效。另外, 对于存取指令, 信号 IorD 被置为 1, 以强迫存储器地址来自 ALU 而非 PC。由于 MDR 在每个时钟周期被写入, 不需设置另外的控制信号。

算术逻辑指令(R 型):

```
Reg[IR[15:11]] <= ALUOut;
```

操作: 将 ALUOut 的内容, 即前面周期中的 ALU 的操作结果, 写入结果寄存器。信号 RegDst 必须置为 1, 以使 rd 字段(15: 11 位)在寄存器堆中用于选择要写入的寄存器。RegWrite 必须有效, 同时 MemtoReg 必须为 0, 以使 ALU 的输出而非存储器的数据输出被写入。

步骤 5: 读取存储器完成

在这一步中, 写入来自存储器的值, 完成取数指令。

取数指令:

```
Reg[IR[20:16]] <= MDR;
```

操作: 将在前面周期中存入 MDR 的数据写入寄存器堆, 为此, 将 MemtoReg 置 1(以写入来自存储器的结果), 使 RegWrite 有效(以引起写操作), 将 RegDst 置 0 以选择 rt(20:16 位)字段为

寄存器号。

以上五步概括如图 5-30。这五步可以决定每个时钟周期里控制信号该做些什么。

步骤名	R型指令动作	存储器访问指令动作	分支动作	跳转动作
取指		$IR \leq Memory[PC]$ $PC \leq PC + 4$		
指令译码/取寄存器		$A \leq Reg[IR[25:21]]$ $B \leq Reg[IR[20:16]]$ $ALUOut \leq PC + (sign-extend(IR[15:0]) \ll 2)$		
执行、地址计算、分支/跳转完成	$ALUOut \leq A op B$	$ALUOut \leq A + sign-extend(IR[15:0])$	if ($A == B$) $PC \leq ALUOut$	$PC \leq \{PC[31:28], (IR[25:0], 2'b00)\}$
存储器访问或 R 型指令完成	$Reg[IR[15:11]] \leq ALUOut$	Load: $MDR \leq Memory[ALUOut]$ or Store: $Memory[ALUOut] \leq B$		
存储器读操作完成		Load: $Reg[IR[20:16]] \leq MDR$		

图 5-30 所有类型指令执行步骤的总结

[指令分 3 到 5 步执行。前两步与指令类型无关。之后，指令根据类型不同，还需要 1~3 步完成。存储器访问或存储器读完成步骤为空的项说明有些指令需要较少的周期完成。在一个多周期实现中，当前指令一完成下一条指令就马上开始，所以这些周期不会空闲或浪费。如前所述，实际上每个周期都读寄存器堆，但是只要 IR 不变，从寄存器堆读出的内容就是一样的。特别地，对于分支或 R 型指令，在指令解码阶段读入寄存器 B 的值，与在执行阶段写入 B 的值，以及随后存字指令中存储器访问阶段使用的值都是一样的]

5.5.2 控制单元的定义

现在已经确定了控制信号的内容及设置的时间，就可以实现控制单元了。为单周期数据通路设计控制单元，使用了一组真值表，根据指令类型确定控制信号的设置。而多周期数据通路的控制更为复杂，因为其指令是分一系列步骤执行的。多周期数据通路的控制部分必须确定每一步及下一步要设置的信号。

在本小节及■ 5.7 节中，将研究两种不同的控制技术。第一种技术基于有限状态机，通常以图形表示。第二种技术称为微程序[⊖]，通过编程实现控制。这两种描述控制的形式都允许其具体实现——门电路、ROM、PLA 的使用由 CAD 系统合成。在本章中，将集中讨论这两种方式下控制的设计及描述。

■ 5.8 节给出了如何使用硬件设计语言设计现代处理器，并且包括多周期数据通路和有限状态控制的例子。在现代数字系统设计中，将硬件描述变为实际门电路的最后步骤由逻辑和数据通路合成工具处理。附录 C 中通过将多周期控制单元转化为详细硬件实现过程显示了处理器如何工作。无需阅读■ 5.8 节和■ 附录 C，就可以通过本节掌握控制的关键思想。但是，如果想要进行实际的硬件设计，5.9 节是有用的，并且■ 附录 C 给出了门电路级的具体实现。

根据这个实现方案和每个状态需要一个时钟周期的知识，可以计算出一套典型混合指令的 CPI。

例题 多周期 CPU 的 CPI

使用图 3-26 所示的 SPECINT2000 混合指令，假设每个多周期 CPU 状态需一个时钟周期，CPI 为多少？

⊖ 微程序(microprogram) 将控制用符号化形式表示为一组微指令，并在一个简单的微机器上执行。

解 混合指令为 25% 取数据字(1% 取字节 + 24% 取字), 10% 存数据字(1% 存字节 + 9% 存字), 11% 分支(6% beq, 5% bne), 2% 跳转(1% jal + 1% jr), 52% ALU(假定所有剩下的混合指令都是 ALU 指令)。从图 5-30 可得, 各指令类型所需时钟周期数为:

- 取数据字: 5
- 存数据字: 4
- ALU 指令: 4
- 分支: 3
- 跳转: 3

CPI 计算如下:

$$\begin{aligned} \text{CPI} &= \frac{\text{CPU 时钟周期数}}{\text{指令数}} = \frac{\sum (\text{指令数}_i \times \text{CPI}_i)}{\text{指令数}} \\ &= \sum \frac{\text{指令数}_i}{\text{指令数}} \times \text{CPI}_i \end{aligned}$$

比值

$$\frac{\text{指令数}_i}{\text{指令数}}$$

即为指令类型 i 的指令频度。于是可得

$$\text{CPI} = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

这个 CPI 要优于所有指令使用相同周期数 5.0 时的最坏 CPI。当然, 设计开销可能会降低或增加这种差异。多周期设计可能也有更好的性价比, 因为在数据通路中需要更少的独立组件。■

描述多周期控制的第一种方法是**有限状态机**^①。一个有限状态机由一组状态及状态间的转换规则组成。转换规则由一个**后继状态函数**^②确定, 它将现有状态和输入映射到一个新的状态。当用一个有限状态机描述控制时, 每个状态还表示一组输出, 当机器在此状态时, 这些输出有效。有限状态机的实现方式一般假设所有的输出都不会被明确地设为有效或无效。同样地, 当一个信号没有被明确地设为有效时, 就是无效状态, 而非无关状态, 这是数据通路正常运作的前提。比如, 仅当寄存器堆中某一项要被写入时, RegWrite 信号才应是有效的; 当它未被明确设为有效时, 一定是无效的。

多路复用器的控制稍有不同, 因为它们选择一个为 0 或为 1 的输入。这样, 在有限状态机中, 经常要确定好所关心的多路复用器的控制。当逻辑实现有限状态机时, 可将控制置 0 设为默认值, 这样它们不需任何门电路。在■附录 B 中有一个简单的有限状态机的例子, 如果你对有限状态机的概念不甚熟悉, 可在继续学习之前仔细研读■附录 B。

有限状态机的控制基本上对应于 5.5.1 节的五个执行步骤; 有限状态机的每一个状态占用一个时钟周期。有限状态机由几个部分组成。由于所有指令的前两个执行步骤都一样, 所有指令的有限状态机的前两个状态也都一样。第 3~5 步不一样, 具体由操作码决定。在某类型指令的最后一步执行完后, 有限状态机将回到原始状态, 开始取下一条指令。

图 5-31 抽象地表示了一个有限状态机。为了进一步研究有限状态机的细节, 首先要扩充取指令和解码部分, 然后介绍不同指令类型各自对应的状态(和动作)。

① 有限状态机(finite state machine) 一个逻辑函数序列由下列元素构成: 一组输入, 输出, 将当前状态和输入映射到一个新状态的下一状态函数, 以及将当前状态和可能的输入映射为一组有效输出的输出函数。

② 后继状态函数(next-state function) 一个组合函数: 给定输入和当前状态, 决定有限状态机的下一个状态。

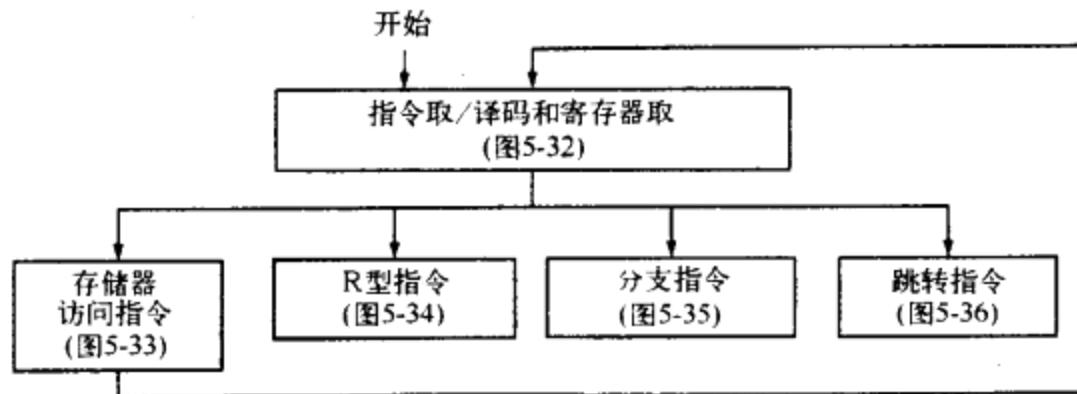


图 5-31 有限状态机控制的高层视图

[前面的步骤与指令类型无关；然后进行一系列由指令的操作码决定的操作，以完成各种类型指令的执行。完成该类型指令所需操作后，控制转回取新的指令。这张图中的每一个方框表示一个或多个状态。标有开始的弧线指出要取第一条指令开始时的状态]

图 5-32 以传统的图形方式表示了有限状态机的前两个状态。为便于说明，给各状态随机标号。0 状态对应第一步，是有限状态机的初始状态。

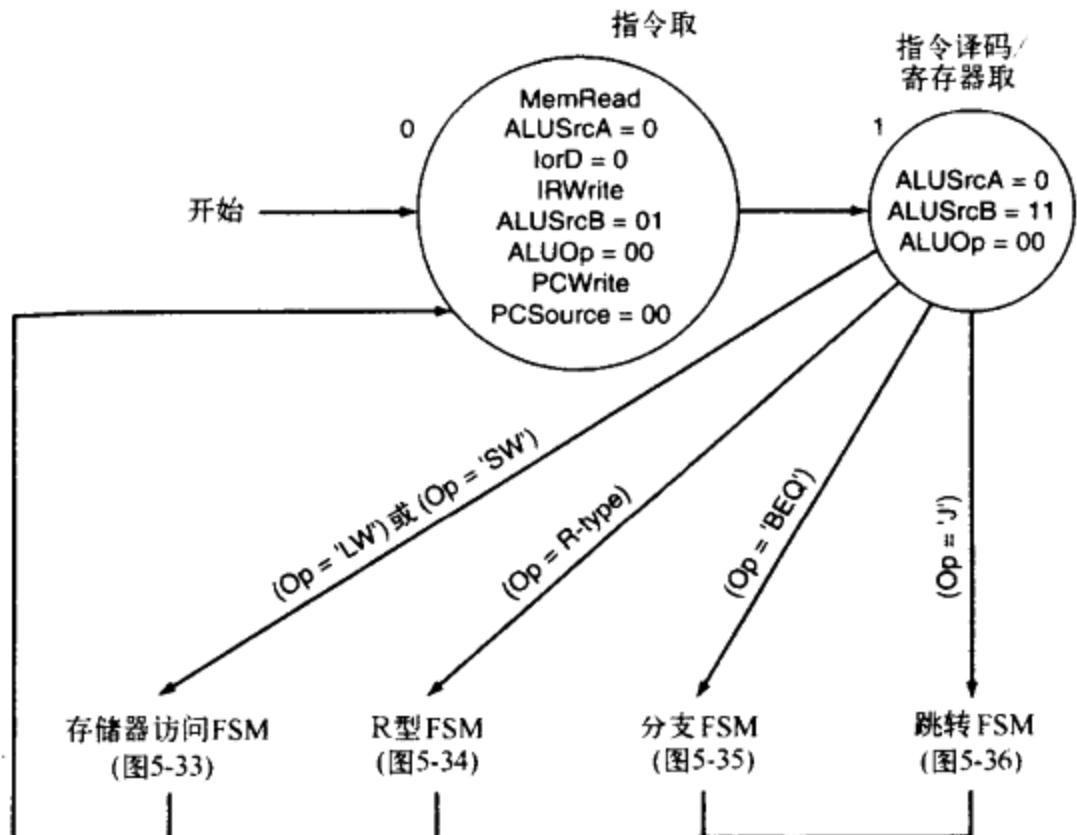


图 5-32 所有指令的取指和译码部分都是相同的

[这些状态对应于图 5-31 的概括的抽象有限状态机的上面那个方框。在第一个状态，激活两个信号 (MemRead 和 IRWrite) 使存储器读一条指令并将它写入指令寄存器，并设置 IorD 为 0 以选择 PC 为地址来源。设置信号 ALUSrcA、ALUSrcB、ALUOp、PCWrite、PCSource 以计算 PC + 4 并存入 PC。(它也会存入 ALUOut，但不会从那里被使用。) 在下一状态，将 ALUSrcB 设为 11(使低 16 位符号扩展并移位后的 IR 送入 ALU)，ALUSrcA 设为 0，ALUOp 设为 00，以计算分支目标地址；将结果存于每个周期都写入的 ALUOut 寄存器中。根据在当前状态中得知的指令类型，在 4 个后续状态中进行选择。控制单元的输入称为 Op，用于决定选择哪条弧线。请记住，所有不是显式标明为有效的信号都是无效的；这点对于控制写操作的信号特别重要。对于多路复用器控制，没有特别设置的表示无关多路复用器的设置]

各状态中有效的信号在表示该状态的圆圈中写出。状态之间的弧线指向下一个状态，当有多个可能的下一状态时，弧线上标注了选择此下一状态的条件。在状态 1 之后，有效的信号取决于指令类型。所以，有限状态机从状态 1 伸出 4 条弧线，对应于 4 个指令类型：存储访问、R 型、等

值分支和跳转。这个根据指令向不同状态分支的过程称为解码，因为后继状态的选择，即下面的动作，由指令类型决定。

图 5-33 给出了有限状态机用于实现存储访问指令的部分。对于这种指令，取指或读取寄存器后的第一个状态计算存储地址(状态 2)。为计算存储地址，ALU 的输入多路复用器需设置为接受寄存器 A 为第一个输入，符号扩展后的偏移量字段为第二个输入；其结果写入 ALUOut 寄存器中。存储地址计算之后，应读/写存储器；这需要两个不同的状态。若指令的操作码为 `LW`，则状态 3(对应于存储访问步骤)进行存储器读(`MemRead`被激活)。存储器的输出通常写入 MDR。若是 `SW`，状态 5 进行存储器写(`MemWrite`被激活)。在状态 3 和 5 中，`IorD` 信号被设为 1 以保证存储器地址来自 ALU(存数指令并不需要这样，因为写地址使用存储器的不同输入)。写存储后，`sw` 指令执行完毕，后继状态为状态 0。若是取数指令，则需要另一状态(状态 4)将存储器的结果写入寄存器堆。设置多路复用器控制信号 `MemtoReg = 1` 和 `RegDst = 0` 将使从 MDR 中取出的数值写入寄存器堆，并以 `rt` 作为寄存器号。这一步对应于存储器读完成步骤，其下一状态为状态 0。

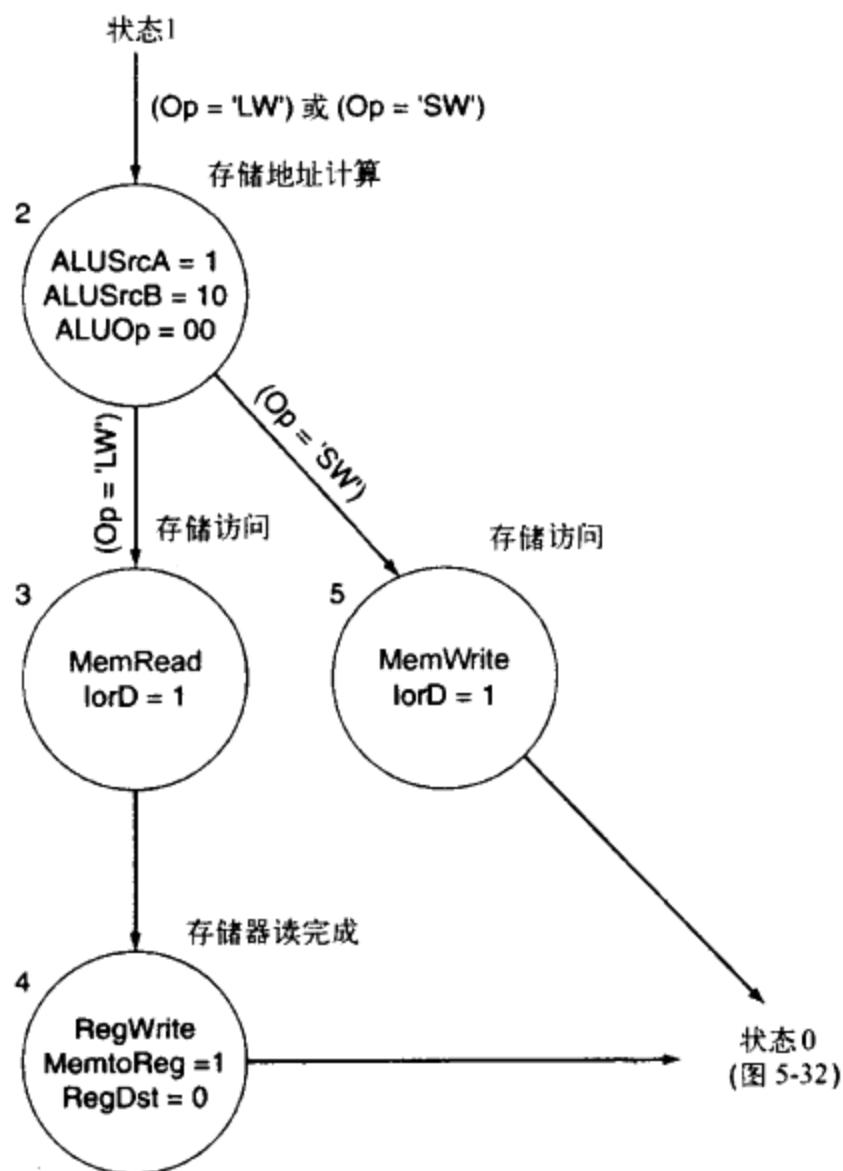


图 5-33 控制存储访问指令的有限状态机有四个状态

[这些状态对应于图 5-31 中标有“存储访问指令”的方框。计算存储地址之后，存和取指令分别需要一个序列。信号 `ALUSrcA`、`ALUSrcB` 和 `ALUOp` 的设置是为了在状态 2 进行存储地址的计算。取指令需要多一个状态以将来自 MDR 的结果(结果在状态 3 时写入)写入寄存器堆]

为实现 R 型指令，需要对应于步骤 3(执行)和步骤 4(R 型指令完成)的两个状态。图 5-34 给出了有限状态机的两个状态部分。状态 6 激活 `ALUSrcA` 并置 `ALUSrcB` 为 00；这使得从寄存器堆中读取的两个寄存器作为 ALU 的输入。置 `ALUOp` 为 10 使 ALU 控制单元根据指令的功能字段

设置 ALU 的控制信号。在状态 7 中, RegWrite 被激活以写寄存器堆, RegDst 被激活以使 rd 字段作为目的寄存器号, MemtoReg 无效以选择 ALUOut 作为源值写入寄存器堆。

对于分支指令, 只需多加一个状态, 因为指令执行在第 3 步就完成了。在这一状态中, 必须设置控制信号使 ALU 对寄存器 A 和 B 的内容做比较, 也要设置信号使 ALUOut 寄存器中的内容有条件地写入 PC。为进行比较, 需激活 ALUSrcA 并置 ALUSrcB 为 00, 置 ALUOp 为 01(强制进行减法)。(只使用 ALU 的零输出信号, 而不用减法的结果。)为控制 PC 的写入, 需激活 PCWriteCond 并置 PCSource=01, 这样若 ALU 的零输出信号有效, ALUOut 寄存器的内容(为状态 1 计算出的分支目标地址, 见图 5-32)将写入 PC。图 5-35 给出这一状态。

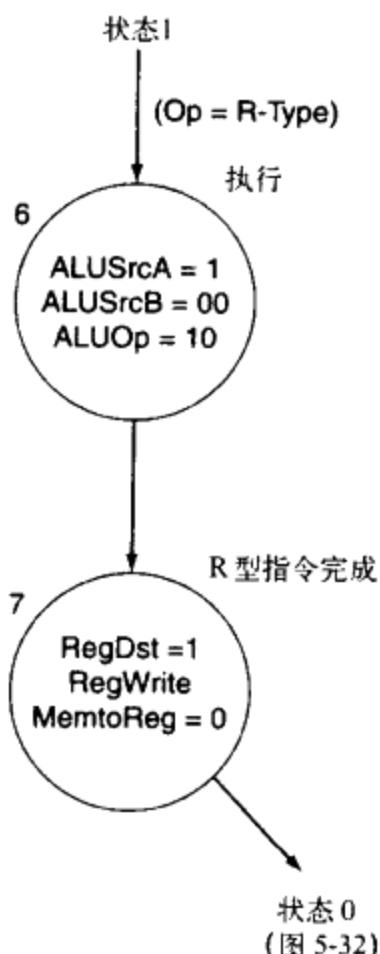


图 5-34 R 型指令可以由简单的两个状态的有限状态机实现

[这些状态对应于图 5-31 中标有“R 型指令”的方框。第一个状态使 ALU 进行操作, 第二个状态使 ALU 的结果(在 ALUOut 中)写入寄存器堆。在状态 7 激活的三个信号使 ALUOut 的内容写入寄存器堆中由指令寄存器的 rd 字段标识的寄存器中]

最后一类是跳转指令; 同分支指令一样, 它也只需一个状态(如图 5-36 所示)完成其执行。在这一状态中, PCWrite 信号被激活以写 PC。通过置 PCSource 为 10, 所写的内容将为指令寄存器的低 26 位加上最低两位 0 和高 4 位 PC 值。

现在, 我们可以将有限状态机的这些部分合在一起, 组成控制单元如图 5-38 所示。在每一状态中写出了被激活的信号。后继状态取决于指令的操作码位, 所以弧线上标出相应的指令操作码。

一个有限状态机可由一个包含当前状态的临时寄存器和一个决定数据通路信号设置和下一状态的组合逻辑模块实现。图 5-37 给出了这种实现可能的样子。■附录 C 详细描述了用这种结

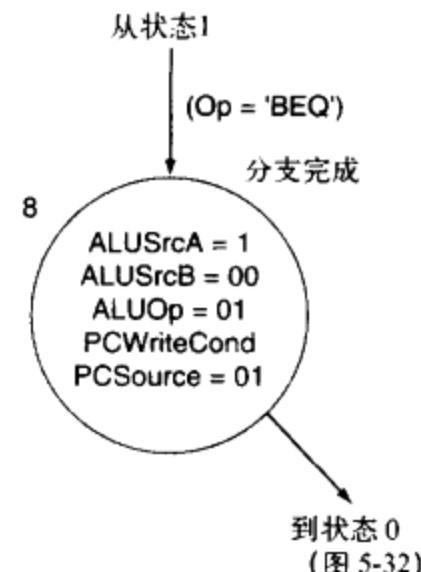


图 5-35 分支指令只需要一个状态

[被激活的前三个输出(ALUSrcA、ALUSrcB 和 ALUOp)使 ALU 对寄存器进行比较, 若分支条件为真, PCSouce 和 PCWriteCond 信号进行条件写。注意不使用写入 ALUOut 的值; 而只用 ALU 的零输出信号。分支目标地址从 ALUOut 得到, 它在第一个状态结束时存入 ALUOut 中]

构实现的有限状态机。在 C.3 节, 图 5-38 的有限状态机的组合控制逻辑分别由一个 ROM(只读存储器)和一个 PLA(可编程逻辑阵列)实现。(附录 B 中有对这些逻辑单元的描述。)在本章的下一部分, 将以另外的方式描述控制。两种技术只是同样控制信息的不同表示方式。

第 6 章中主要内容流水线几乎总被用来加速指令的执行。对于简单指令, 流水线能够比多周期设计获得更高时钟频率和比单周期设计获得更高单周期 CPI。但是, 大多数流水线处理器中, 某些指令的执行比单个时钟周期长, 需要多时钟周期控制。浮点指令就是普遍的例子。在 IA-32 体系结构中有许多需要使用多周期控制的例子。

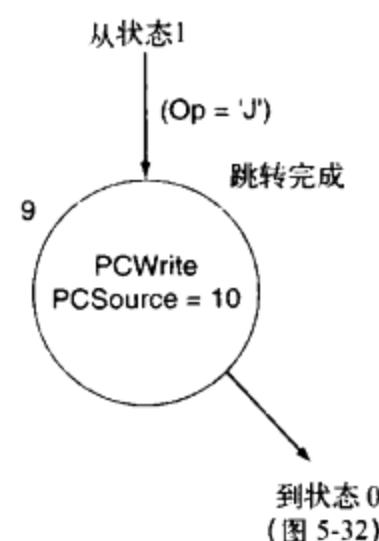


图 5-36 跳转指令只需要一个状态, 它激活两个控制信号, 以将指令寄存器的低 26 位左移 2 位并与本指令 PC 的高 4 位相连后写入 PC

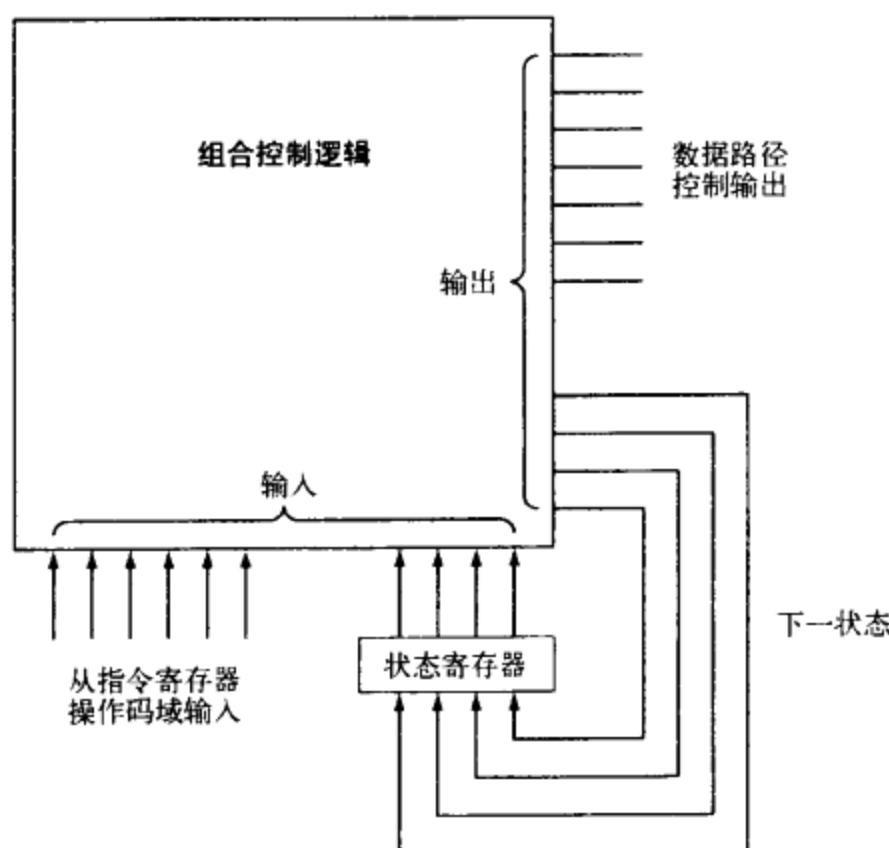


图 5-37 有限状态机的控制器通常由一个组合逻辑模块和一个保存当前状态的寄存器组成

[组合逻辑的输出为下一状态号和在当前状态中应该激活的控制信号。组合逻辑的输入为当前状态和所有可能决定后继状态的输入信号, 在当前情况下, 输入即为指令寄存器的操作码位。注意在本章使用的有限状态机中, 输出仅取决于当前状态, 而不取决于输入信号。下面的细节部分对此进行了更详细的解释]

细节: 图 5-37 的有限状态机称为 Moore 机, 以 Edward Moore 命名。其显著特点是其输出只依赖于当前状态。对一个 Moore 机, 标有组合控制逻辑的模块可分为两部分。一部分有控制输出和只有状态输入, 另一部分只有后继状态输出。

另一种有限状态机称为 Mealy 机, 以 George Mealy 命名。Mealy 机允许输入和当前状态共同决定输出。Moore 机在速度和控制单元大小方面有潜在的实现优势。速度优势是因为在时钟周期早期需要的控制输出与输入无关, 只取决于当前状态。在附录 C

中，当有限状态机的实现具体到逻辑门一级时，大小优势便显而易见。Moore 机的潜在缺陷是它可能需要额外的状态。比如，当两个状态序列间只有一个状态不同时，Mealy 机可通过使输出取决于输入而将状态合并。

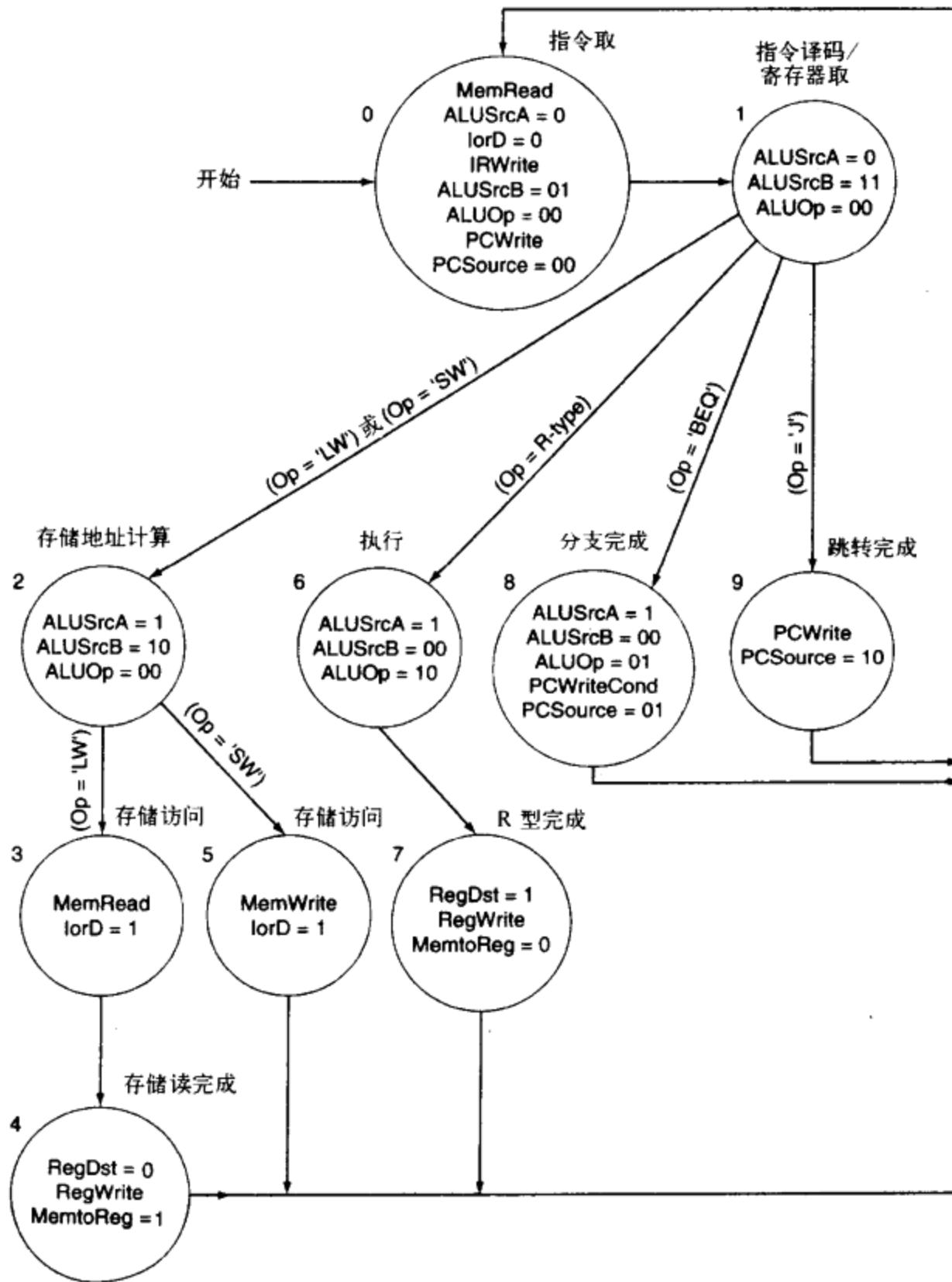


图 5-38 图 5-28 中数据通路的完整的有限状态机控制

[弧线上的标注是在确定后继状态时进行测试的条件；当后继状态是无条件的时候没有标注。结点中的标注为该状态中激活的输出信号；如果正确的操作要求，通常就要指明多路复用器控制信号的设置。这样，在某些状态下一个多路复用器的控制可能设为 0]

理解程序性能

对于给定时钟频率的处理器，两个代码段的相对性能由 CPI 的结果和执行每个代码段的指令

数决定。正如这里所见，即使简单的处理器，不同指令也可能有不同的 CPI。在随后两章中，引入流水线和高速缓存后不同指令的 CPI 可能会更不同。虽然硬件设计者可以控制许多影响 CPI 的因素，但是程序员、编译器和软件系统决定了最终执行哪些指令，从而决定了程序的有效 CPI 是多少。程序员要想提高性能必须理解 CPI 的作用和影响 CPI 的因素。

自测

1. 对错与否：由于跳转指令不依赖寄存器的值或正在计算的分支目标地址，因此可以在第二个状态间完成分支，而不是等到第三个状态。
2. 对错与否或者是否可能：能否使用 PCSrc[0]取代 PCWriteCond 控制信号。

5.6 异常

控制设计是处理器设计中最具挑战性的一个方面：它最难达到正确，也最难以提高速度。控制中最难的部分之一是实现异常^①和中断^②——除分支、跳转以外改变正常指令执行顺序的事件。一个异常是一个来自处理器内部的意外事件；算术溢出就是一个异常的实例。一个中断也是一件导致控制流意外改变的事件，但它来自处理器外部。在第 8 章中将会看到，I/O 部件通过中断与处理器进行通信。

许多体系结构和著作者不区分中断和异常，通常用老术语中断概括这两种事件。遵循 MIPS 的习惯，术语异常指控制流中任何意外的改变，而无论其产生原因是来自处理器内部还是外部；术语中断则只用于由外部引起的事件。Intel IA-32 体系用中断指代这两种意外事件。

中断本来是为处理如算术溢出之类的意外事件和指示 I/O 部件请求服务而创建的。同样的基本机制被扩展，也用于处理内部产生的异常。这里有一些例子，说明某些情况是处理器内部还是外部产生的：

事件类型	来 源	MIPS 术语
I/O 部件的请求	外部	中断
从用户程序调用操作系统	内部	异常
算术溢出	内部	异常
使用未定义的指令	内部	异常
硬件错误	两者都可能	异常或中断

许多对异常支持的要求都来自导致意外发生的特殊情况。在第 7 章讨论存储级别和第 8 章讨论 I/O 时，将回到这个话题，以能更清楚地理解要求额外的异常机制的动机。在这一节中讨论了检测两种异常的控制实现，这两种异常由我们前面已经涉及的指令集部分和其实现方式所产生。

检测异常条件并采取适当举措，这经常处于一个机器的关键时间路径上，这条路径决定了时钟周期的长度，即机器性能。如果在控制单元的设计中没有对异常的充分考虑，那么在复杂的实现中加入异常支持可能导致性能的明显下降，并且使正确的设计变得更为困难。

5.6.1 异常是怎样处理的

这里的实现中可能产生的两种异常是未定义指令的执行和算术溢出。异常发生时机器必须

① 异常(exception) 也称中断。是一个打断程序运行的突发事件；它被用来检测溢出等。

② 中断(interrupt) 来自处理器外的异常。(有些体系结构中也用中断一词表示所有的异常现象。)

进行的基本操作是：在异常程序计数器(EPC)中保存出错指令的地址，并把控制权转交给某特定地址处的操作系统。

操作系统可采取适当的行动，如给用户程序提供一些服务，对溢出情况进行事先定义好的操作，或者终止程序的执行并报告错误。在完成处理异常所需动作后，操作系统可以终止程序，也可以继续程序的执行，此时由 EPC 决定重新开始执行的地方。在第 7 章将更详细讨论重新开始执行的问题。

为处理异常，操作系统必须知道引起异常的是哪条指令，和异常的原因。主要有两种方法用于表示产生异常的原因。MIPS 体系使用的方法是设置一个状态寄存器(称为原因寄存器(cause register))，其中有一个字段用于记录异常产生的原因。

另一种方法是使用向量中断^②。在向量中断中，控制权被转移到由异常原因决定的地址处。例如，为处理前面的两种异常，可如下定义两种异常向量地址：

异常类型	异常向量地址(十六进制)
未定义的指令	C000 0000 _{hex}
算术溢出	C000 0020 _{hex}

操作系统通过初始地址可以知道产生异常的原因。地址由 32 个字节即 8 条指令表示，操作系统必须记录异常起因，并可以在这个指令序列之内进行一些有限的处理。当异常不是向量型的，所有异常使用同一入口点，操作系统通过解码状态寄存器以寻找异常的原因。

通过给基本实现加上一些额外的寄存器和控制信号，再稍微扩充一下有限状态机，就可以进行对异常的处理。如果要实现的是 MIPS 体系结构使用的异常系统。(实现异常向量也不难。) 这需要给数据通路加上两个附加寄存器：

- EPC：一个 32 位寄存器，用于存储受影响的指令地址。(向量式异常也需要这样一个寄存器。)
- Cause：一个记录异常原因的寄存器。MIPS 体系结构中这个寄存器是 32 位的，虽然其中一些位现在还没有用。假定用寄存器的低位对前面提到的两种异常的可能原因进行编码：未定义指令 = 0，算术溢出 = 1。

需要加入两个控制信号称为 EPCWrite 和 CauseWrite，使 EPC 和 Cause 寄存器被写入。另外，需要一位控制信号 IntCause 对 Cause 寄存器的低位进行正确的设置；最后，需要将异常地址，即操作系统处理异常的入口点，写入 PC；在 MIPS 体系中地址为 8000 0180_{hex}(MIPS 的 SPIM 模拟器使用 8000 0080_{hex})。现在，在 PC 的人口接有一个由 PCSOURCE 信号(参见图 5-28)控制的三路复用器。也可以改用一个四路复用器，另一个输入恒为 8000 0180_{hex}。当 PCSOURCE 为 11_{two} 时就选择了这个值写入 PC。

由于在每条指令的第一个周期 PC 增值，所以不能将 PC 值写入 EPC，这时 PC 值为指令地址加 4。然而可以用 ALU 将 PC 减 4，再把 ALU 的输出写入 EPC。这并不需要额外的控制信号或通路，因为可以用 ALU 做减法，且常数 4 已经是 ALU 的一个可选的输入。所以 EPC 的写数据端口与 ALU 的输出相连。图 5-39 画出了加上实现异常处理后的多周期数据通路。

用图 5-39 所示数据通路，处理各种异常所需的操作各占一个状态。对每种情况，该状态设置 Cause(原因)寄存器，计算并存储原始 PC 的值至 EPC 寄存器，而将处理异常的地址写入 PC。所

② 向量中断(vectored interrupt) 用来进行控制转换的中断地址根据例外的原因决定。

以,为处理这里考虑的两种异常,只需加入两个状态。在加入这两个状态前必须决定如何检查异常,因为这些检查控制连到新状态的弧。

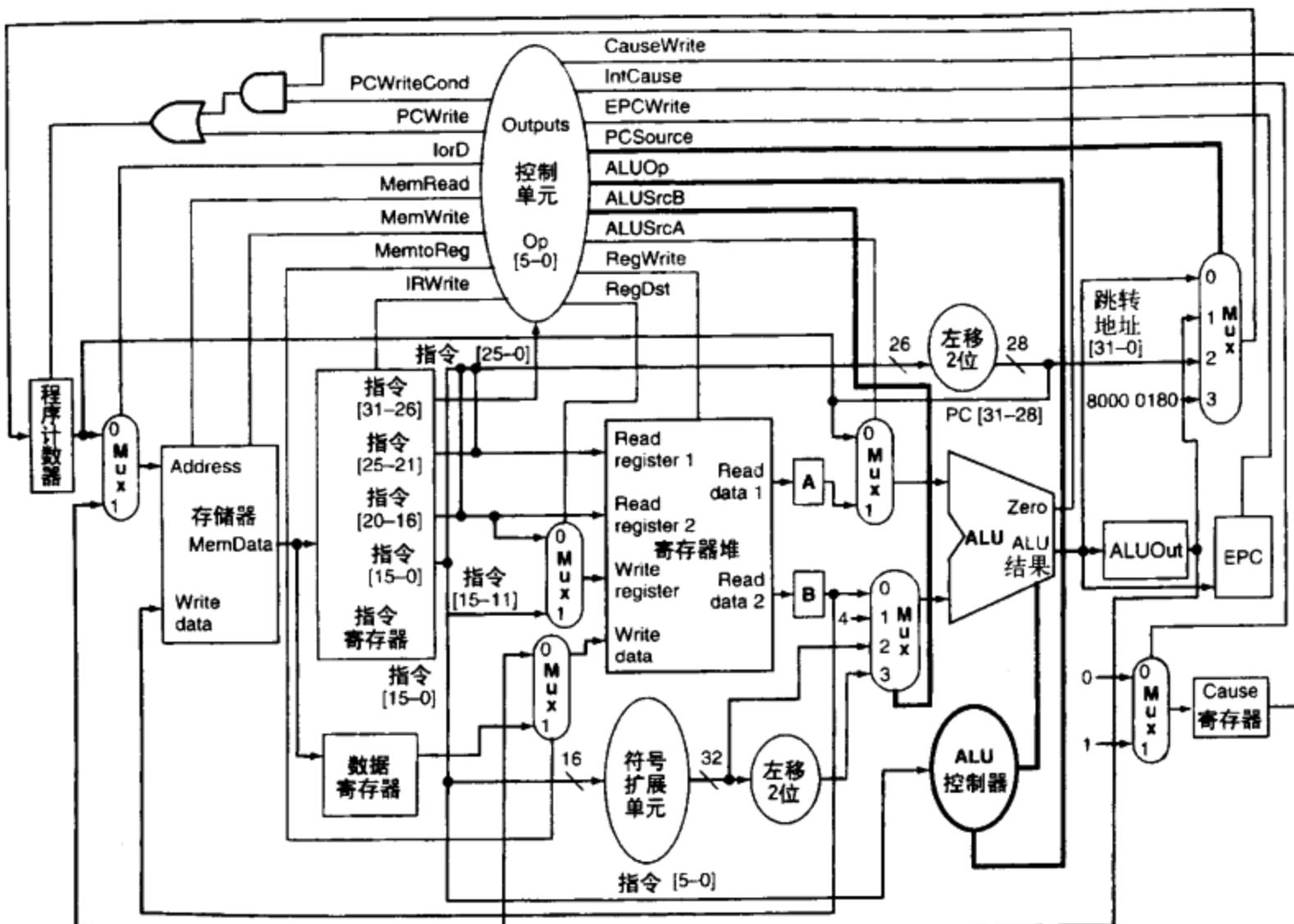


图 5-39 加入了异常处理的多周期数据通路

[所作的增加包括 Cause 和 EPC 寄存器,一个控制 Cause 寄存器输入的多路复用器,一个控制写入 PC 的值的多路复用器的扩展,和用于所加的多路复用器和寄存器的控制线路。为了简化起见,本图中没有显示 ALU 溢出信号,这个信号需要存储在寄存器中的一位,并且作为控制单元的额外输入(图 5-40 中显示如何使用这个信号)]

5.6.2 控制部件如何检测异常

现在要设计一种方法去检测异常,并将控制权转交给异常状态中适当的一个。图 5-40 给出了两种新状态(10 和 11),以及这两个状态与其他有限状态控制的连接。对两种可能的异常分别进行检测:

- 未定义指令:当操作码值的状态 1 没有符合定义的后继状态时,就认为发现了这种异常。为处理这种异常,将除 lw、sw、0(R型)、j 和 beq 以外的操作码值的后继状态定义为 10。给不符合从状态 1 射出至新状态 10 的弧线上所标的任何操作码的操作码字段标符号 *other*,以表示这种异常情况。
- 算术溢出:■附录B讲述了 ALU 中检测溢出的逻辑,并提供了一个称为 Overflow 的信号作为 ALU 的输出。在图 5-40 中,修改过的有限状态机用这个信号指明状态 7 的一个增加的可能的后继状态(11)。

图 5-40 完整地描述了这个有两种异常的 MIPS 子集的控制。请牢记,实际机器的控制设计的挑战性在于处理指令与其他引起异常的事件之间的多种相互影响,同时要求控制逻辑又小又快。可能存在的复杂的相互影响使控制单元在硬件设计中最具挑战性。

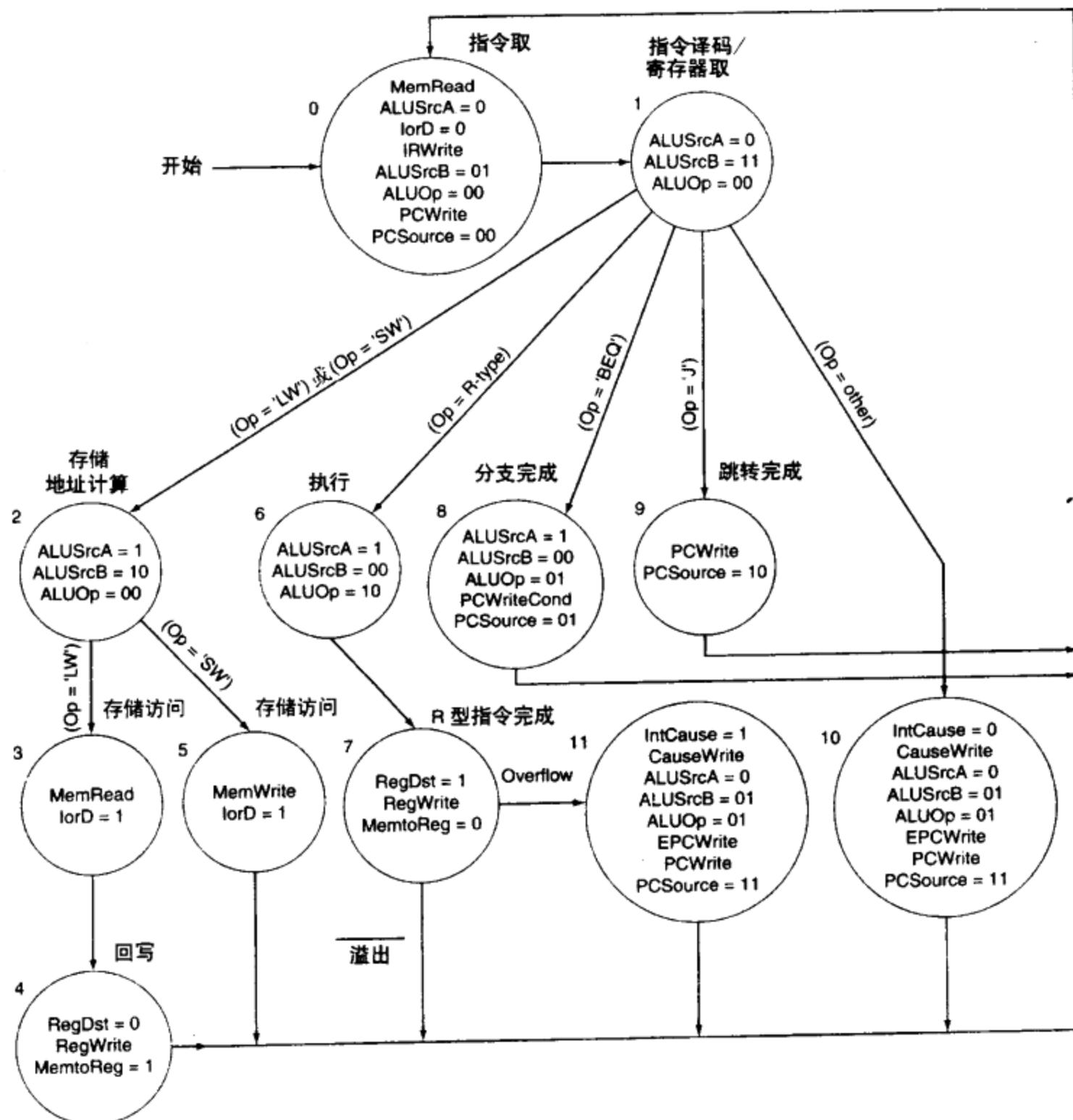


图 5-40 这是加入了处理检测异常部分后的有限状态机

[状态 10 和 11 来是对异常生成相应控制的新的状态。出自状态 1 的标有 (Op = other) 的弧线指明了当输入与操作码 lw、sw、0(R型)、j 或 beq 都不匹配时的后继状态。出自状态 7 的标有溢出的弧线指明了当 ALU 出现溢出时应采取的动作]

细节：如果仔细观察图 5-40 的有限状态机，你会发现处理异常的方式可能出现一些问题。例如，对算术溢出，引起溢出的指令仍完成结果的写入，因为溢出分支是在写完成状态中的。不过，系统可以将引起异常的指令定义为无效；MIPS 指令集系统就是这样做的。在第 7 章中，将看到某些类型的异常要求禁止指令改变机器状态，这样处理异常就变得复杂并且限制了机器的性能。

自测

5.5.2 节自测中提到的有关 PCSource 的优化对于图 5-40 中异常的扩展控制是否有效？为什么？

■ 5.7 微程序设计：简化控制设计

微程序设计是设计复杂控制单元的一种技术。它使用非常简单的硬件机构，然后编程实现更复杂的指令集。当前，微程序设计用来实现复杂指令集的某些部分，例如 Pentium 和其他特殊用途处理器。本节收录在光盘中，其中解释了微程序的基本概念，显示了如何用微程序来实现 MIPS 的多周期控制。

■ 5.8 使用硬件描述语言进行数字设计概述

现代数字设计使用硬件描述语言和现代计算机辅助合成工具完成，可以使用库和逻辑合成器将设计描述转化成详细的硬件设计。全书都是关于这些语言和它们在数字设计中的运用。本节收录在光盘中，给出了简要介绍并且阐述了如何使用 Verilog 硬件设计语言从行为角度(适合通过硬件合成的形式)来描述 MIPS 多周期控制。

5.9 实例：近期的 Pentium 处理器的实现结构

本章讲述的数据通路和控制单元的构造技术对每个计算机都至为重要。然而，所有的近期计算机都不仅应用本章介绍的技术还使用流水线。流水线是下一章的主要内容，它通过重叠多条指令的执行来改善机器的性能，使吞吐率达到几乎每周期一条指令(就像单周期实现)，其时钟周期的长度则由单个功能单元的延迟，而非一条指令的整个执行通路(如多周期设计)决定。最后的不采用流水线的 Intel IA-32 处理器是 1985 年生产的 80386；第一个 MIPS 处理器 R2000 也是在 1985 年生产的，它便采用了流水技术。

最近的 Intel IA-32 处理器(奔腾 II、III 和 4)采用了越来越复杂的流水线方法。然而，这些处理器仍然面临着为复杂的 IA-32 指令集(参见第 2 章)实现控制电路这一挑战。这些现代处理器的基本功能单元和数据通路，虽然比本章所描述的复杂得多，但具有与本章相同的基本功能和相似的控制信号类型。所以其控制单元的设计基于与本章所用到的相同的原理。

5.9.1 实现更复杂体系结构的挑战

与 MIPS 架构不同，IA-32 体系结构的指令非常复杂，可能用到几十个、甚至上百个时钟周期来执行。例如，串传送指令(MOVS)要求计算并修改两个不同的存储地址，并且存取一个字节串。IA-32 系统繁多而复杂的寻址模式会使得即使像 MIPS 一样简单的指令结构的实现也变得复杂。幸运的是，多周期数据通路的结构，可适应 IA-32 内在指令要进行的多样的工作。这种适应性来自两种能力：

1) 一个多周期数据通路允许指令占用不定数目的时钟周期。IA-32 中类似 MIPS 系统的简单指令可占 3、4 个时钟周期，而复杂的指令可占用几十个时钟周期。

2) 一个多周期数据通路可在一条指令内多次使用数据通路部件。这点对于处理 IA-32 体系结构更复杂的寻址方式和实现 IA-32 更复杂的操作很关键。若没有这种能力，数据通路必须进行扩展，以在不重用部件的前提下处理复杂指令的需求，而这几乎是不可能实现的。例如，一个不重用部件的单周期数据通路若用于 IA-32，则需要多个数据存储器和非常多的 ALU。

使用多周期数据通路和微程序控制器^①为实现 IA-32 指令集提供了一个框架。然而，挑战性的任务是构造高性能的实现，这要求处理来自不同指令的多样化要求。简而言之，一个高性能的实现要求保证简单指令的快速执行，而复杂指令系统的负担主要由复杂的、不频繁使用的指令承担。

① 微程序控制(microprogrammed control) 使用微代码而不是有限状态机来表示控制的方法。

为达到这一目的，486 之后 IA-32 系统的每个 Intel 实现都综合使用硬连线控制^①来处理简单指令和微代码^②控制来处理更复杂指令。那些在数据通路中可一遍执行完的指令——复杂程度类似于 MIPS 指令——由硬连线控制生成控制信息，在数据通路中一次执行完成，只占用少量时钟周期。那些需多次数据通路执行和顺序复杂的指令由微代码控制器处理，占用更多时钟周期，多次通过数据通路以完成其执行。这种方法的好处是使设计者可以以少数周期完成简单指令，而不用为最复杂又常用的指令建造非常复杂的数据通路。

5.9.2 Pentium 4 的实现结构

最新的奔腾处理器都可以通过使用称为超标量^③的高级流水线技术实现每个时钟周期执行多条指令。下一章将讲述超标量处理器的工作原理。现在要理解的是单周期执行多条指令要求复制一些数据通路资源。简单地看，处理器可以有多条数据通路，每一条为处理某一类型的指令：即存数取数、ALU 操作和分支指令。这样，处理器可在同一时钟周期内执行一条分支指令，一条 ALU 操作和一条存/取数指令。奔腾 III、4 允许一个时钟周期执行多达 3 条 IA-32 指令。

实际上，奔腾 III、4 的数据通路执行类似于 MIPS 的简单的微指令^④（用 Intel 术语即微操作^⑤）。这些微指令是完全自含的操作，初始为 70 位宽。实现这些微指令的数据通路的控制完全是硬连线的。在整数数据通路中，控制的最后一级将 3 条微指令扩展为 120 条控制线，而浮点数数据通路中将 275 条扩展为超过 400 条控制线，后者对应到奔腾 4 中含的新的 SSE2 指令。将微指令扩展为控制线的最后步骤与单周期数据通路或 ALU 的控制的形成很相似。

(怎样实现 IA-32 指令和微指令间的转换呢?)在早期 Pentium 处理器实现中(也就是 Pentium Pro、Pentium II 和 Pentium III)，指令译码单元同一时间查看多达 3 条 IA-32 指令，每个时钟周期由一些 PLA 译码成 6 条微指令。在 Pentium 4 中采用了更高的时钟频率，原先的解决方案已经不够了，需要一种全新的方法生成微指令。

Pentium 4 中采用的解决方案中包括一种微指令的踪迹缓存^⑥技术，IA-32 的程序计数器可以访问这个缓存。踪迹缓存是指令缓存的更复杂形式，具体细节在第 7 章中解释。现在，可以把踪迹缓存看作保存实现某个 IA-32 指令的微程序缓存(buffer)。当访问要执行的下一条指令地址的踪迹缓存时，发生下列事件中的一种：

- IA-32 指令译码后的内容在踪迹缓存中。这时，从踪迹缓存中最多可以产生 3 条微指令。这 3 条微指令代表从 1 至 3 条 IA-32 指令。根据匹配几条微指令序列，IA-32 PC 前进 1 至 3 条指令。
- IA-32 指令译码后的内容在踪迹缓存中，但是需要超过 4 条微指令实现。对于这种复杂 IA-32 指令，使用微代码 ROM 处理；控制单元将控制转给 ROM 中保存的微程序。微程序生

-
- ① 硬连线控制(hardwired control) 有限状态机控制的实现，通常使用可编程逻辑阵列(Programmable logic array, PLA)或 PLA 与随机逻辑的组合。
 - ② 微代码(microcode) 对处理器进行控制的微指令集合。
 - ③ 超标量(superscalar) 一种高级流水线技术能够使处理器每时钟周期执行多条指令。
 - ④ 微指令(microinstruction) 使用底层指令表示控制，每一条微指令声明一组控制信号使其在给定时间激活，并且指明下一条要执行的微指令。
 - ⑤ 微操作(micro-operations) 在最近的 Pentium 处理器上实现的直接由硬件执行类 RISC 指令。
 - ⑥ 踪迹缓存(trace cache) 一个指令缓存，存有给定地址开始的一指令序列；在最近的 Pentium 实现中，踪迹缓存保存的是微操作而不是 IA-32 指令。

成微指令，直到完成复杂的 IA-32 指令。微代码 ROM 提供超过 8000 条微指令和一些 IA-32 指令共享的微指令序列。完成之后将控制返回，继续从踪迹缓存中取指。

- IA-32 指令的译码不在踪迹缓存中。这时使用 IA-32 的译码单元对指令进行译码。如果译码产生的微指令数为 4 或者更少，译码得到的微指令放入踪迹缓存，这条指令下次执行时就可以直接从缓存中得到。否则，使用微代码 ROM 完成微指令序列。

踪迹缓存产生的 1 至 3 条微指令送入 Pentium 4 微指令流水线，具体细节在第 6 章最后给出。Pentium 4 使用简单硬连线控制和简单数据通路处理微指令，结合指令译码的踪迹缓存，获得令人吃惊的时钟频率，近似于实现更简单指令集的处理器。而其，对简单指令使用直接硬连线控制结合对复杂指令使用微代码控制的方法使得 Pentium 4 能够高速执行 IA-32 指令集中简单、高频率使用的指令，得到较低而且非常有竞争力的 CPI。

理解程序性能

不考虑内存系统，虽然 Pentium 4 的大部分性能依赖微操作流水线的效率，但是前端的 IA-32 指令译码可以显著影响性能。特别是根据译码器结构，使用需要 4 个或更少微操作的简单 IA-32 指令，避免使用微代码调度[⊖]可以带来更好的性能。由于这种实现策略(与 Pentium III 类似)，编译器开发者和汇编语言程序员应该尽量使用简单 IA-32 指令序列，而不是复杂的指令。

5.10 谬误和陷阱

陷阱：用微代码实现一条复杂指令可能并不比使用一系列简单指令快。

大多数有庞大且复杂指令集的机器在实现中，至少部分使用了存于 ROM 中的微代码。令人惊讶的是，在这样的机器上，有时执行一系列独立的简单指令和执行某条指令的微代码序列一样快，甚至更快。

为什么会这样呢？微代码曾经有过这样的优势：与从内存中取指令相比，从速度更快的高速的专用存储中读取微操作码要快得多。由于高速缓存在 1968 年开始使用，微代码在取指时间上不再有这样的优势。然而，在计算中微代码因使用内部临时寄存器仍有优势，这样的寄存器在用于缺乏通用寄存器的机器中时很有效。微代码的缺点是在建造机器之前必须选好算法，并在下一代体系结构出现之前不能更改。而另一方面，程序中的指令却可以在机器生命中的任何时刻得益于算法的改进。所以微代码序列在所有可能的操作组合中可能并非最优。

IA-32 实现中一条这样的指令例子是串传送指令(MOVS)，使用在第 2 章讨论过的重复前缀。这条指令通常比一个每次移动一个字的循环慢，正如前面的“谬误和陷阱”所示。

另一个例子涉及 LOOP 指令，它对一个寄存器进行减值，若减后非 0 则分支到一特定标号处。这些指令是为固定重复次数的循环(如许多 for 循环)中，在最后一层进行分支而设计的。这样一条指令，除包含了一些额外工作外，好处是使在流水的机器中分支指令的潜在损失最小化(在下

⊖ 调度(dispatch) 微程序控制单元的一个操作，操作根据微指令的一个或多个字段选择下一个微指令，一般建立一个包含目的微指令地址的表，再根据微指令的域对表进行索引。调度表通常使用 ROM 或可编程逻辑阵列(PLA) 实现。调度一词也用在动态调度处理器中，指将一个指令发送到某个队列的过程。

一章讨论分支时将看到)。

遗憾的是，在所有近期 Intel IA-32 实现中，LOOP 指令总是慢于由简单的独立指令组成的微代码序列(假设小的代码规模不作为一个考虑因素)。这样，以提高速度为核心的优化编译器从不生成 LOOP 指令。而造成的结果便是不鼓励在将来的机器中加快 LOOP 的执行，因为它很少被用到！

谬误：如果控制存储器有空，新增指令不需付出代价。

微程序方式的一个好处是在 ROM 中实现的控制存储不太昂贵，并且随着晶体管预算的增加，额外的 ROM 几乎免费。有一个比喻：就像盖一幢房子，在快盖好的时候，发现还剩有足够的地方和材料来增盖一个房间。但这个房间并非免费的，因为劳动力和房子的维修都要开销。添加“免费”指令的企图只可能发生在指令集没有固定，一般即机器的第一代模型时。由于二进制程序的向上兼容性很重要，机器以后的所有模型都必须包含这些所谓的免费指令，即使在后来空间成为首要因素。

在设计 80286 时，给指令集添加了许多指令。更多可用的硅资源和微程序实现的使用，使这样的添加看起来不需付出代价。最大的添加可能是一个一般不用的复杂的保护机制，但在新的实现中还必须包含。添加的原因是认为有对此机制的需求，并希望加强微处理器系统以提供与更大的计算机同样的功能。同样地，还添加了许多十进制指令以提供以字节为单位的十进制数运算。现在这些指令很少用到，因为使用 32 位二进制算术运算加上与十进制间的转换要快的多。和保护机制一样，在新的处理器中必须实现十进制指令，即使它们很少用到。

5.11 结论

在这一章看到，处理器的数据通路和控制的设计，可以从指令集系统和对基本技术特征的理解开始。在 5.3 节，我们介绍了在体系结构和决定用单周期实现的基础上，如何构造 MIPS 处理器的数据通路。当然，一些基础性的技术，如数据通路中哪些部件可用，以及单周期实现是否有意义等，也影响了许多设计的决策。同理，在 5.5 节，通过使用将时钟周期分为多个步骤的思想，经过修改我们得到了多周期数据通路。在两种情况下，顶层结构——一个单周期还是多周期机器——与指令集一起，决定了数据通路设计中的许多特征。

重点

开始设计的时候可以用多种方式描述控制。时序控制的选择和逻辑表示的确定相互无关；然后可以用几种结构化逻辑设计技巧之一进行控制的实现。图 5-41 为多种控制的描述方法，以及几种从描述到结构化逻辑电路的实现方式。

同样地，控制主要由指令集系统、组织方式和数据通路的设计来共同决定。在单周期组织结构中，这三者主要定义了控制信号的设置方法。在多周期设计中，根据指令集系统进行指令的执行在各时钟周期中的分解方式，以及数据通路的设计，共同定义了对控制设计的要求。

控制是计算机设计中最富挑战性的一个方面。一个主要原因是控制的设计要求理解处理器所有部件的操作。为帮助设计，讨论了两种描述控制的方式：有限状态图和微程序。这些描述可以从实现细节抽象出控制的特征。这样的抽象是对付计算机设计的复杂性的主要方法。

具体描述出控制后，可以将它映射到实际的硬件。控制的实现细节取决于控制的结构和实现它所使用的基础技术。控制的实现与技术有关，而它会随时间变化，这也是为什么对控制的特征进行抽象有好处的原因之一。

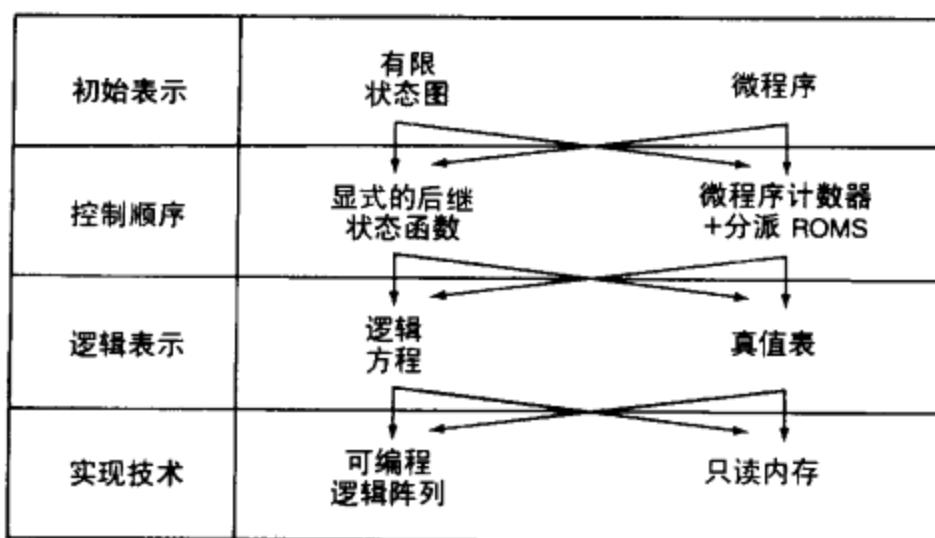


图 5-41 几种描述和实现控制的方法

[箭头表示了可能的设计线路：所有从原始的表示到最后的实现技术的通路都是可行的。传统上，“硬连线控制”指使用左边的技术，而“微程序控制”指使用右边的技术]

■ 5.12 历史回顾和深入阅读

微程序设计的兴起和它对指令集设计与计算机发展的影响，是电子计算机头几十年中有趣的相互作用之一。本书附赠光盘中的历史回顾主要介绍这方面的故事。

5.13 习题

- 5.1 [6] <§ 5.2> 实现下列功能单元，需要使用组合逻辑、时序逻辑，还是两者结合使用？
- 多路复用器
 - 比较器
 - 递增运算器/递减运算器
 - 桶形移位器(barrel shifter)
 - 带有移位器和加法器的乘法器
 - 寄存器
 - 存储器
 - ALU(单周期或者多周期数据通路的 ALU)
 - 先行加法器(carry look-ahead adder)
 - 锁存器
 - 通用有限状态机(FSM)
- 5.2 [10] <§ 5.4> 考虑一个恒 0 错误(即无论某信号应为何值，它总是为 0)，对图 5-17 的单周期数据通路中多路复用器的影响。哪些指令仍可以正常工作？并解释原因。分别考虑下面几种错误：RegWrite = 0, ALUop0 = 0, ALUop1 = 0, Branch = 0, MemRead = 0, MemWrite = 0。
- 5.3 [5] <§ 5.4> 与 5.2 题类似，但考虑恒 1 错误(即信号总是为 1)。
- 5.4 [5] <§ 5.4> ■ For More Practice：带有浮点数的单周期数据通路。
- 5.5 [5] <§ 5.4> ■ For More Practice：带有浮点数的单周期数据通路。
- 5.6 [10] <§ 5.4> ■ For More Practice：带有浮点数的单周期数据通路。
- 5.7 [2~3 个月] <§ § 5.1~5.4> 使用标准元件构建本章所描述的单周期机器。
- 5.8 [15] <§ 5.4> 希望给本章描述的单周期数据通路加入 jr(根据寄存器中地址跳转的跳转寄存器)指令。给图 5-17 的单周期数据通路加入必要的数据通路和控制信号，并在图 5-18 上进行必要的添加。为加快速度可以复制这些图。

- 5.9 [10] <§ 5.4> 与习题 5.8 类似, 但这次要加入的是 2.5 节讲述的 sll(逻辑左移)指令。
- 5.10 [15] <§ 5.4> 与习题 5.8 类似, 但这次要加入的是 2.9 节讲述的 lui(取立即数到寄存器高位)指令。
- 5.11 [20] <§ 5.4> 与习题 5.8 类似, 但这次要加入的是 lw(取字)指令的变形。(即将两寄存器相加以得到要取数据的地址), 在从内存取字之后递增索引寄存器。这个指令(l_inc)类似下面两条指令:

```
lw    $rs,L($rt)
addi $rt,$rt,1
```

- 5.12 [5] <§ 5.4> 请解释: 为什么修改单周期实现方式以实现习题 5.12 中的交换指令就必须修改寄存器堆?
- 5.13 [7] <§ 5.4> 考虑图 5-17 中的单周期数据通路。有一个朋友建议将 MemtoReg 信号去掉以修改这个单周期数据通路。以 MemtoReg 作为输入的多路复用器将改用 MemRead 控制信号作为输入来取代 ALUSrc 或者 MemRead 控制信号。这个朋友的修改可以正常工作吗? 这两个信号(ALUSrc 和 MemRead)可以相互替代吗? 为什么?
- 5.14 [10] <§ 5.4> MIPS 选择简化它的指令集。通过使用 MIPS 指令实现复杂指令的方法是将复杂指令译码为多条简单的 MIPS 指令。给出 MIPS 如何实现下面的指令: swap \$rs, \$rt, 将寄存器 \$rs 和 \$rt 中内容交换。考虑可以有个寄存器中的内容被破坏和没有这样的寄存器两种情况。

如果硬件实现这条指令将增加一条指令 10% 时钟周期, swap 指令在指令流中占多少比例时用硬件实现才真正有益?

- 5.15 [5] <§ 5.4> ■ For More Practice: 控制多路复用器中错误的影响
- 5.16 [5] <§ 5.4> ■ For More Practice: 控制多路复用器中错误的影响
- 5.17 [5] <§ 5.5> ■ For More Practice: 控制多路复用器中错误的影响
- 5.18 [5] <§ 5.5> ■ For More Practice: 控制多路复用器中错误的影响
- 5.19 [15] <§ 5.4> ■ For More Practice: 给数据通路增加指令
- 5.20 [15] <§ 5.4> ■ For More Practice: 给数据通路增加指令
- 5.21 [8] <§ 5.4> ■ For More Practice: 给数据通路增加指令
- 5.22 [8] <§ 5.4> ■ For More Practice: 给数据通路增加指令
- 5.23 [5] <§ 5.4> ■ For More Practice: 给数据通路增加指令
- 5.24 [10] <§ 5.4> ■ For More Practice: 数据通路控制信号
- 5.25 [10] <§ 5.4> ■ For More Practice: 数据通路控制信号
- 5.26 [15] <§ 5.4> ■ For More Practice: 修改数据通路和控制
- 5.27 [8] <§ 5.4> 重做习题 5.14, 但是指令换为递增取数: l_incr \$rt, Address(\$rs)。
- 5.28 [5] <§ 5.4> “关键路径”是指机器中最长的可能路径, 介绍见 5.4 节。根据你对单周期实现的理解, 哪些功能单元能够容忍更多延时(也就是不再关键路径上), 哪些单元能够从硬件优化中获得好处。使用 5.4 节中的数据量化答案(“例题: 单周期机器性能”)。

- 5.29 [5] <§ 5.5> 类似习题 5.2, 这次考虑恒 0 错误对图 5-28 中的多周期数据通路的影响。考虑下列错误:
- RegWrite = 0
 - MemRead = 0
 - MemWrite = 0
 - IRWrite = 0
 - PCWrite = 0
 - PCWriteCond = 0.

- 5.30 [5] <§ 5.5> 类似习题 5.29, 但是考虑恒 1 错误(信号总是 1)。
- 5.31 [15] <§ 5.4, 5.5> 类似习题 5.13 但更一般化。判定是否单周期实现中的任一控制信号都能取

消并且用另一个已有的控制信号或其反信号代替。之所以存在这样的冗余，是因为只有一个很小的指令集，如果实现了更多指令这种冗余它就会消失(或很难找到)。

5.32 [15] <§ 5.5> 希望给本章中的多周期数据通路加上 lui(取上一个立即数)指令。这条指令在第3章有所阐述。以图5-28的多周期数据通路相同的结构，并在图5-38的有限状态机上进行任何必要的修改。可以参考5.5.1节的执行步骤，并考虑一下为执行新的指令需要进行哪些操作。请说明实现这条新指令需要多少个周期。

5.33 [15] <§ 5.5> 修改习题5.32中的实现，使得执行时间减少1个周期。给图5-28所示的多周期数据通路加上所有必要的数据通路和控制信号。可以复印已有的图以方便修改。必须假定在第一个阶段完成前(第二个周期结束)不知道执行的是什么指令。请说明新的指令在修改后的数据通路和有限状态机上执行需要多少个周期。

5.34 [20] <§ 5.5> 本题类似习题5.32，要实现的指令换成ldi(取立即数)，从紧跟指令地址的内存地址中取出32位立即数。

5.35 [15] <§ 5.5> 考虑将多周期实现中的寄存器堆改为只有一个读端口。(用图)描述此修改所带来的数据通路的其他必需的修改。修改有限状态机以说明指令怎样工作，给出新的数据通路。

5.36 [15] <§ 5.5> 控制处理器性能的两个重要参数：时钟周期时间和每条指令时钟周期数。在微处理器的设计过程中始终存在这两个参数的权衡。某些设计者倾向于提高处理器的主频，但是以大的CPI为代价。另一些设计者遵循不同的思路，以降低主频换取减少CPI。

考虑下面的机器，使用图3-26中给出的SPEC CPUint2000数据，比较各自的性能。

M1：1GHz时钟频率的第5章介绍的多周期数据通路。

M2：类似第5章的多周期数据通路，但是寄存器更新所用的时钟周期与读内存或ALU计算相同。因此在图5-38中，状态6、7和状态3、4是可组合的。这台机器的时钟频率为3.2GHz，因为寄存器更新过程增加了关键通路的长度。

M3：类似M2，但是有效地址计算与内存访问在同一时钟周期完成。因此，状态2、3和4是可组合的，同样状态2和5、6和7也是可组合的。因为寄存器的更新增加了关键路径的长度，这台机器的时钟频率为2.8GHz。

算出哪台机器最快。是否存在不同的指令集合可以使另外一台机器更快，如果是请举出例子。

5.37 [20] <§ 5.5> 你在C³(Creative计算机公司)的朋友认为多周期数据通路中决定时钟周期长度的关键路径为存取数据(而非指令)的存储器访问。这使得他们最新的MIPS 30000时钟频率为4.8GHz，而非目标时钟频率所定的5.6GHz。然而C³的Clara有办法解决这个问题。若所有的存储访问被分至两个时钟周期，机器就可达到目标的运行速度。

使用第3章(图3-26)的SPEC CPUint 2000指令集，计算双周期存储访问的机器比单周期存储访问的4.8GHz的机器快多少。假设所有跳转和分支指令使用的周期数相同，设置指令和算术立即数指令由R-型指令实现。考虑进一步将取指令划分成两个时钟周期是否能够将时钟频率提高到6.4GHz，为什么？

5.38 [20] <§ 5.5> 试想有一条MIPS指令bcmp，比较两个内存地址中的字块。假设指令要求第一个数据块地址存于\$t1寄存器，第二个数据块地址存于\$t2，要比较的字数存于\$t3(\$t3≥0)。假设指令能将结果(第一个不匹配的地址或者完全匹配时的0)放在\$t1和/or \$t2中。还假设这些寄存器及\$t4和\$t5寄存器的内容在指令执行中可被破坏(这样寄存器就可作为指令执行时的临时寄存器)。

写出实现块比较的MIPS汇编语言程序。要进行两个100个字块的比较，需要多少条指令？用多周期实现中指令的CPI计算，进行100个字块比较需要多少个时钟周期？

5.39 [2~3个月] <§ 5.1~5.5> 用标准器件构造本章中的多周期机器。

5.40 [15] <§ 5.5> ■ For More Practice：为数据通路增加指令

5.41 [15] <§ 5.5> ■ For More Practice：为数据通路增加指令

5.42 [15] <§ 5.5> ■ For More Practice：为数据通路增加指令

- 5.43 [15] <§ 5.5> ■ **For More Practice:** 为数据通路增加指令
- 5.44 [15] <§ 5.5> ■ **For More Practice:** 为数据通路增加指令
- 5.45 [20] <§ 5.5> ■ **For More Practice:** 为数据通路增加指令
- 5.46 [10] <§ 5.5> ■ **For More Practice:** 为数据通路增加指令
- 5.47 [15] <§ § 5.1~5.5> ■ **For More Practice:** 比较处理器性能
- 5.48 [20] <§ 5.5> ■ **For More Practice:** 实现 MIPS 指令
- 5.49 [30] <§ 5.6> 为本章介绍的多周期数据通路增加 eret(异常返回)指令, 其主要任务是将发生异常或错误自陷时的地址重新存入 PC。假定如果处理器处理错误自陷, 则 PC 从寄存器 EPC(ErrorPC)中取地址。否则处理器处理异常, EPC 从 PC 中取地址。假定在发生错误自陷时, 原因寄存器中有 1 位编码表示发生了自陷, 并将 PC 的内容保存在 EPC 寄存器中。为了提供自陷/异常调用和返回, 在图 5-39 所示的多周期数据通路中增加必要的数据通路和控制信号, 并修改图 5-40 中的有限状态机来实现 eret 指令。可以复印图来方便修改。
- 5.50 [6] <§ 5.6> 当处理器发生不可预测的事件时, 控制流将发生改变以对异常进行处理。发生异常的原因以及导致异常的指令怎样用 MIPS 机器的硬件表示? 给出两个处理器能够在处理完异常后重新执行指令的例子, 并给出两个导致程序终止的异常例子。
- 5.51 [6] <§ 5.6> 异常检测是异常处理的重要部分。试着在图 5-28 的多周期数据通路中标出能够检测出异常的周期。
- 考虑下面的几种异常:
- 除 0 异常(假定在一个周期中使用同一个 ALU 做除法运算, 并且这个 ALU 能为其余的控制部件所识别)
 - 溢出异常
 - 无效异常
 - 外部中断
 - 无效指令内存地址
 - 无效数据内存地址
- 5.52 [15] <§ 5.6> ■ **For More Practice:** 为数据通路增加指令
- 5.53 [30] <§ 5.7> 微代码用于给指令集加入更强的指令; 考虑一下这样做的好处。用多周期数据通路和微代码设计一种习题 5.38 中 bcmp 指令的实现方案。为了更有效地实现 bcmp 指令, 可能需要对数据通路进行一些修改。描述你的修改和 bcmp 指令如何实现。在数据通路中加入内部寄存器对支持 bcmp 指令有好处吗? 估算用硬件实现指令(而非习题 5.38 中的软件方法)可能达到的性能改善, 并解释这种改善的原因。
- 5.54 [30] <§ 5.7> ■ **For More Practice:** 微代码
- 5.55 [30] <§ 5.7> ■ **For More Practice:** 微代码
- 5.56 [5] <§ 5.7> ■ **For More Practice:** 微代码
- 5.57 [30] <§ 5.8> 根据你的习题 5.53 的解法, 修改 Figure 5.7.1 的 MIPS 微指令格式, 并写出 bcmp 指令完整的微程序。详细说明你是怎样扩展微代码, 以支持微代码内部更为复杂的控制结构(如循环)的。对 bcmp 指令的支持改变了微代码的大小吗? 微指令格式的修改对 bcmp 以外的其他指令有影响吗?
- 5.58 [5] <§ 5.8> A、B 寄存器的值由下面 Verilog 初始化代码给定:

```
reg A,B;
initial begin
    A = 1;
    B = 2;
end
```

分析下面两段 Verilog 描述，并比较变量 A 和 B 的结果以及每个例子中完成的操作。

```
a)    always @(negedge clock) begin
      A = B;
      B = A;
    end
b)    always @(negedge clock) begin
      A <= B;
      B <= A;
    end
```

5.59 [15] <§ 5.4, 5.8> 用组合 Verilog 写 ALUControl 模块，使用下面形式作为基础：

```
module ALUControl (ALUOp, FuncCode, ALUCt1);
  input ALUOp[1:0], FuncCode[5:0];
  output ALUCt1[3:0];
  ...
endmodule
```

5.60 [1 周] <§ 5.3, 5.4, 5.8> 用一种硬件仿真语言，如 Verilog，实现单周期方案的一个功能仿真器。如果可以，用已有的器件库构造你的仿真器，如果器件中含有时序信息，计算该实现的时钟周期。

5.61 [2~4 小时] <§ 4.7, 5.5, 5.8, 5.8> 通过增加无符号 MIPS 乘法指令扩展■ 5.8 节中的多周期 Verilog 描述；假定这个指令使用 MIPS ALU 和移位、加法操作实现。

5.62 [2~4 小时] <§ 4.7, 5.5, 5.8, 5.9> 通过增加无符号 MIPS 除法指令扩展■ 5.8 节中的多周期 Verilog 描述；假定这个指令使用 MIPS ALU 以及一次一位(one-bit-at-a-time)算法实现。

5.63 [1 周] <§ 5.5, 5.8> 用一种硬件仿真语言，如 Verilog，实现 PowerPC 处理器设计中使用的多周期方案的一个功能仿真器。如果可以，用已有的器件库构造你的仿真器。如果器件中含有时序信息，计算该实现的时钟周期。

类似 MIPS, PowerPC 每条指令 32 位。假定指令集支持下列指令格式：

R型

	op		Rd		Rt		Rs		0		Func		RC	
0	5 6	10 11	15 16	20 21	22	30 31								

取数/存数&立即

	op		Rd		Rt		Address				
0	5 6	10 11	15 16				31				

分支条件

	op		0		BI		BD		AA LK
0	5 6	10 11	15 16				29 30	31	

跳转

op	Address	[AA LK]
0 5 6 10 11 15 16	29 30 31	

RC-reg
[LT GT EQ OV]
0 1 2 3

Func 字段(22: 30): 类似 MIPS, 表示功能码。

RC 位(31): 如果置(1), 则更新 RC-reg 控制位, 表示指令(所有 R 型)结果。

AA(30): 1 表示给定地址是绝对地址; 0 表示相对地址。

LK: 如果为 1, 更新 LNKR(链接寄存器), 在后面实现子程序返回时要用到。

BI: 分支条件编码(例如: beg \rightarrow BI = 2, blt \rightarrow BI = 0 等)。

BD: 分支相对目标地址。

简化版本的 PowerPC 实现, 只要能够实现下列指令即可:

```

add:      add $Rd, $Rt, $Rs      ($Rd <- $Rt + $Rs)
          addi $Rd, $Rt, #n      ($Rd <- $Rt + #n)
subtract: sub $Rd, $Rt, $Rs      ($Rd <- $Rt - $Rs)
          subi $Rd, $Rt, #n      ($Rd <- $Rt - #n)
load:     lw $Rd, Addr($Rt)    ($Rd <- Memory[$Rt + Addr])
store:   sw $Rd, Addr($Rt)    (Memory[$Rt + Addr] <- $Rd)
AND, OR:  and/or $Rd, $Rt, $Rs    ($Rd <- $Rt AND/OR $Rs)
          andi/ori $Rd, $Rt, #n    ($Rd <- $Rt AND/OR #n)
Jump:     jmp Addr    (PC <- Addr)
Branch conditional: Beq Addr   (CR[2]==1? PC<- PC+BD : PC <- PC+4)
subroutine call: jal Addr   (LNKR <- PC+4; PC<- Addr)
subroutine restore: Ret     (PC <- LNKR)

```

- 5.64 [讨论] <§ 5.7, 5.10, 5.11> 假说: 若一种体系结构的第一个实现采用了微程序, 它会影响其指令集体体系结构。这是为什么? 你可以找出一种只使用微代码的体系结构吗? 为什么? 什么机器从来不使用微代码? 为什么? 你认为在设计指令集体体系结构时设计师打算实现的控制是怎样的?
- 5.65 [讨论] <§ 5.7, 5.12> Wilkes 发明微程序主要是为了简化控制的构造。1980 年之后计算机辅助设计软件的爆炸性增长, 其目的也是为了简化控制的构造。这时的控制设计容易了很多。你能从工具和实际设计中找出证据, 来证实或反驳这种假说吗?
- 5.66 [讨论] <§ 5.12> MIPS 指令与 MIPS 微指令有许多相似之处。是什么使得编译器生成 MIPS 微代码比生成 MIPS 指令代码困难呢? 对微体系结构进行怎样的修改可以使微代码在这种应用中更有用?

自测题参考答案

§ 5.1:3。

§ 5.2: 错。

§ 5.3:A。

§ 5.4: 对, MemtoReg 和 RegDst 是互为非的。对, 可以仅使用另一个信号并且翻转多路复用器的输入顺序即可!

§ 5.5:1. 错。2. 可能：如果在不关心 PCSource[0]信号的情况下（大多数情况下如此），将该信号总是设为 0，那么这个信号就等同于 PCWriteCond。

§ 5.6：不行，因为状态 11 虽然以前没有使用过，但现在它的值被使用了！

§ 5.7：55 个条目的四个表，不要忘了最初的分配！

§ 5.8：1.0, 1, 1, X, 0.2。不行，因为不是所有的路径都被分配了状态。

现实世界中的计算机：助力残障人士

问题：克服残障人士面临的困难。

解决方案：使用机器人、传感器还有计算机控制来代替或补充受损肢体和器官。

右图显示了为一名在灭火中负伤的消防队员开发的系统。在乳胶手指内部的传感器直接感受冷、热温度，再通过他的假肢内的电子接口刺激上臂神经末梢，通过神经系统将冷、热信息传递到脑部。这套价值 3000 美元的系统使这名消防员的假手能够感觉到压力和重量，使其自从 1986 年失去手臂以后，第一次能拿起一罐苏打水而不会将罐子压扁或从手中滑落。这套系统的主要功能装备是一个电子接口，它能够将信号传递给 Whitten 的上臂神经末梢，然后再将信息传递到大脑。

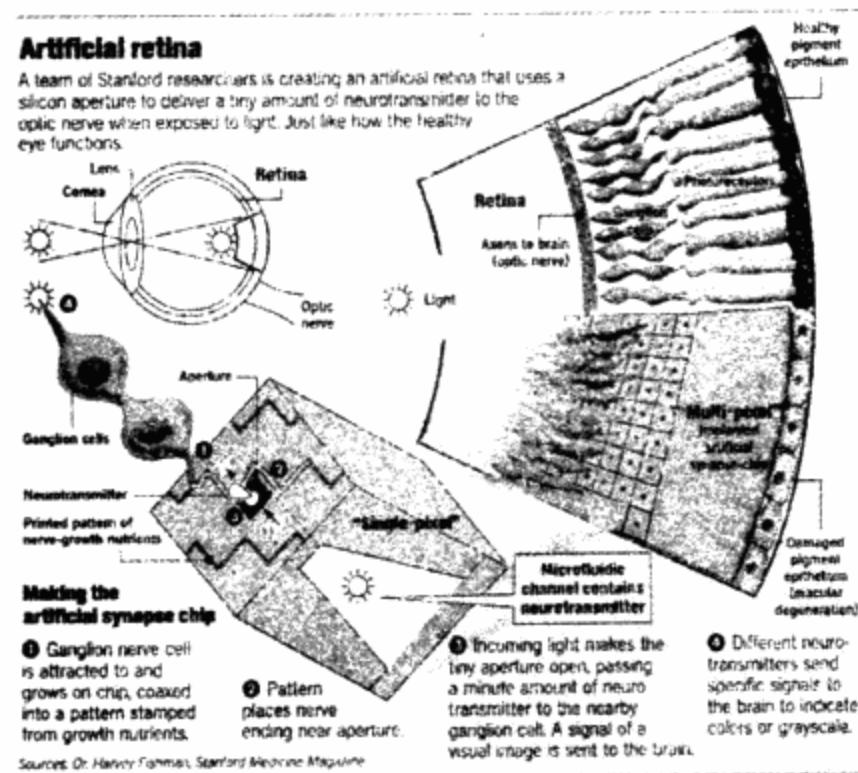
斯坦福大学的 Harvey Fishman 和 Mark

Peterman 在信息技术领域已取得一些进展，这些进展有望帮助因年龄增长造成的失明。他们的方法是将眼睛的图像接收系统断开，而将数码相机的信号直接连接到视觉系统中。他们已经开发了一个到视觉系统的神经接口，称为人工突触芯片。这里面主要的挑战是将电子信号转换为细胞间通信的化学信号。这个芯片附着在细胞上，对于细胞来说人工突触就是一个用硅制作的小洞。这个洞连接到神经递质的收集器。当电场施加到这个芯片上时，神经递质从这个小洞泵出，以刺激周围的细胞。在 2003 年，他们已经在一块芯片上设计了 1 厘米规格的 4 个人工突触。

尽管这项工作还处在早期阶段，但却不仅仅限于研究与眼睛有关的问题。正如 Fishman 所言：“任何需要神经连接之处，我们都有可能对它进行重连接。”



消防员 Ken Whitten 自豪地展示他的新型仿生手臂



使用人工突触芯片的人造视网膜。摘自《The San Francisco Chronicle》，2004 年 1 月 5 日

§ 5.5:1. 错。2. 可能：如果在不关心 PCSource[0]信号的情况下（大多数情况下如此），将该

要了解更多，请在■图书馆(Library)中参阅下列参考资料：

- Rick Smolan and Jennifer Erwitt, *One Digital Day: How the Microchip Is Changing Our World*, Times Publishing, 1998
- Peterman et al, "The artificial synapse chip: A flexible retinal interface based on directed retinal cell growth and neurotransmitter stimulation," *Artificial Organs*: 27(11), November 18, 2003

第6章 利用流水线提高性能

随着时间的推移，每个事物都按固有的规律发展变化；在旧的事物逝去的同时，新的事物又产生了。

Robert Herrick 在 1648 年圣诞前夜庆典上的讲话

6.1 流水线概述

永远不要浪费时间。

美国谚语

流水线^①是一种可以将多条指令的执行过程相互重叠的实现技巧，目前它是提高处理器处理速度的关键。

本节将通过一个类比对流水线的概念及其相关问题进行概述。如果你只是想对流水线技术有一个大概的了解，可以集中精力看完本节，然后直接跳到 6.9~6.10 节学习现代处理器中的高级流水线技巧，如 Pentium III 和 Pentium 4 处理器。如果你想对基于流水线的计算机进行深入剖析，6.2~6.8 节将具体论述本节涉及的内容。

任何一个经常光顾洗衣店的人都会不自觉地采用流水线技术。非流水线方式的洗衣过程将包括如下几个步骤：

- 1) 把一批脏衣服放入洗衣机里清洗。
- 2) 洗衣机洗完后，把衣服取出放入烘干机中。
- 3) 烘干后，将衣服从烘干机中取出，放在桌子上叠起来。
- 4) 衣服叠好以后，请你的室友帮忙把桌子上的衣服拿走。

当你的室友把这批洗干净了的衣服从桌子上全都拿走以后，再开始洗下一批脏衣服。

采用流水线的方法将节省大量的时间，如图 6-1 所示。当把第一批脏衣服从洗衣机里取出放入烘干机之后，你就可以把第二批脏衣服放入洗衣机里进行清洗了。当第一批衣服被烘干之后，就可以将它们折叠起来，同时把洗净的下一批湿衣服放入烘干机中，同时再将下一批脏衣服放入洗衣机里清洗。下一步工作就是让你的室友把第一批衣服从桌子上拿走，而你开始折叠第二批衣服，这时烘干机中放的是第三批衣服，而且你就可以把第四批衣服放入洗衣机清洗了。这样，所有的洗衣步骤(即流水线的各级)在同时操作。一旦每一级操作都有各自独立的工作资源时，我们就可以采用流水线方式来快速完成任务了。

流水线的奇妙之处在于，对于单独的一批衣服来说，从它进洗衣机到烘干机，再到折叠、收拾，整个过程的总的处理时间并没有缩短；而在有多批任务时流水线之所以快的原因是所有的工作都是在并行进行的，因此单位时间内能够完成的工作量就大大增加了。在不改变单批次衣物处理时间的前提下，流水线提高了整个洗衣系统的吞吐率。所以，虽然流水线并不能减少单批衣物的处理时延，但是由于吞吐率的提高，如果我们有足够的工作可以做的话，那么处理这些任务将花费更少的时间。

如果流水线的各级所花费的时间相同并且任务很多，那么流水线化所带来的加速比将大致等于流水线的级数。在上面的例子中，流水线有四级：洗衣，烘干衣服，叠衣服，收拾衣服。所以这

① 流水线(pipelining) 是一种可将多条指令的执行过程相互重叠的实现技巧，很像生产流水线。

个流水线洗衣系统相对未流水线化的系统而言加速比最高为 4：处理 20 份衣服所花的时间大致是处理一份衣服所花时间的 5 倍，而如果非流水线化处理 20 份衣服将花费处理单份衣物所花时间的 20 倍。图 6-1 所示情况中，流水线化版本只快了大约 2.3 倍，这是由于我们只画出了处理 4 份衣物的情况。在该图的流水线化版本中，开始和结束的时候流水线都只是部分被占用；这种占用率递增和递减的现象将对性能产生负面影响，对于任务数相对于流水线级数而言不够大的情况尤其明显。当任务数远大于流水线级数 4，那么流水线各级将始终保持工作状态，同时吞吐率也会非常接近 4。

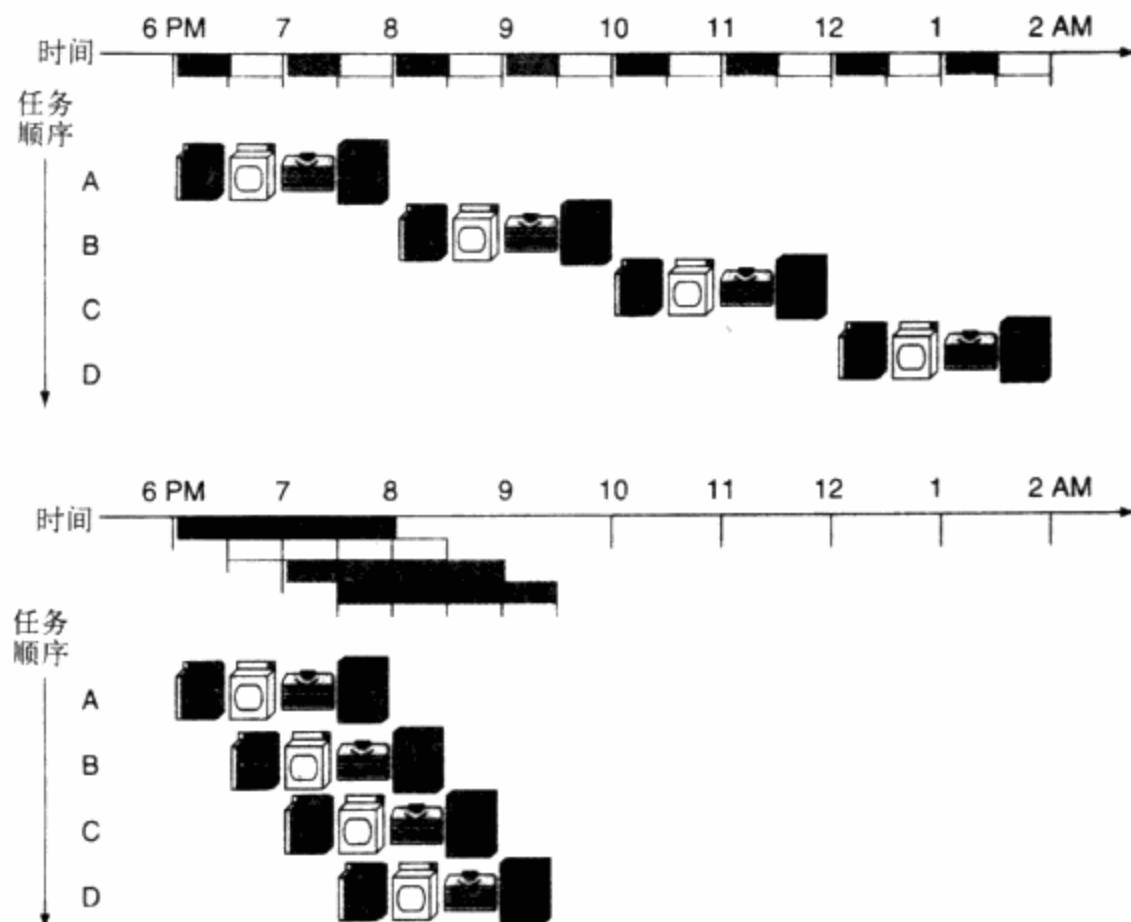


图 6-1 以洗衣店为例来类比流水线的运作过程

[安妮、布莱恩、凯西和唐每个人都有一些脏衣服要清洗、烘干、折叠及收拾。洗衣机、烘干机、“折叠机”和“整理机”每个都需要 30 分钟来完成各自的任务。顺序化的洗衣策略将花费 8 个小时的时间来洗完四批衣服，而流水线化的洗涤策略只需花费 3.5 小时。图中采用在这个二维时间轴上显示 4 道工作副本的方法来表示同一时间不同的处理步骤对应的流水线级，而事实上每种洗衣设备我们都只有一台]

同样的原理也可以应用到处理器中，即采用流水线方式执行处理器指令。通常，一条 MIPS 指令包含如下五个处理步骤：

- 1) 从存储器中读取指令。
- 2) 指令译码的同时读取寄存器。MIPS 的指令格式允许指令译码和读取寄存器同时进行。
- 3) 执行操作或计算地址。
- 4) 在数据存储器中读取操作数。
- 5) 将结果写回寄存器。

因此，本章讨论的 MIPS 流水线有五个处理步骤。正如流水线能加速洗衣店的工作一样，下面的例子将说明流水线如何加快指令的总执行时间。

例题 单周期指令模型性能与流水线模型性能

为了使问题具体化，我们首先假设一个流水线结构。在本例以及本章其余内容中，我们将只考虑以下的 8 条指令：取字(load word, lw)；存储字(store word, sw)；加(add, add)；减(subtract, sub)；与

(and, and); 或(or, or); 小于则置位(set-less-than, slt)和相等则分支(branch-on-equal, beq)。

本例将比较流水线化指令执行与单周期指令执行的平均执行时间，其中在单周期模型中所有指令的执行都花费一个时钟周期。本例中，主要功能单元的操作时间为：内存访问花费 200 ps，ALU 操作花费 200 ps，寄存器文件的读写各自花费 100 ps。(在第 5 章中我们已经说明在单周期模型中每一条指令都恰恰只花费一个时钟周期，因此，时钟周期必须延长以满足最慢的指令。)

解 8 条指令中每一条指令所需要的执行时间如图 6-2 所示。单周期模型中必须考虑最慢的指令，在图 6-2 中是 lw，因此，每一条指令所花费的执行时间为 800 ps。与图 6-1 类似，图 6-3 比较了三条取字指令非流水线与流水线方式的执行过程，其中在非流水线模型中，第一条与第四条指令之间的时间差是 $3 \times 800 \text{ ps} = 2400 \text{ ps}$ 。

指令类型	取指令	读寄存器	ALU 计算	访问数据	写回寄存器	总时间
取字(lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
存储字(sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R 型(add、sub、and、or、slt)	200 ps	100 ps	200 ps		100 ps	600 ps
分支(beq)	200 ps	100 ps	200 ps			500 ps

图 6-2 由各操作组件所花费时间计算出来的各条指令的执行时间

[这里我们假设乘法器、控制单元、PC 访问和符号扩展单元都没有延时]

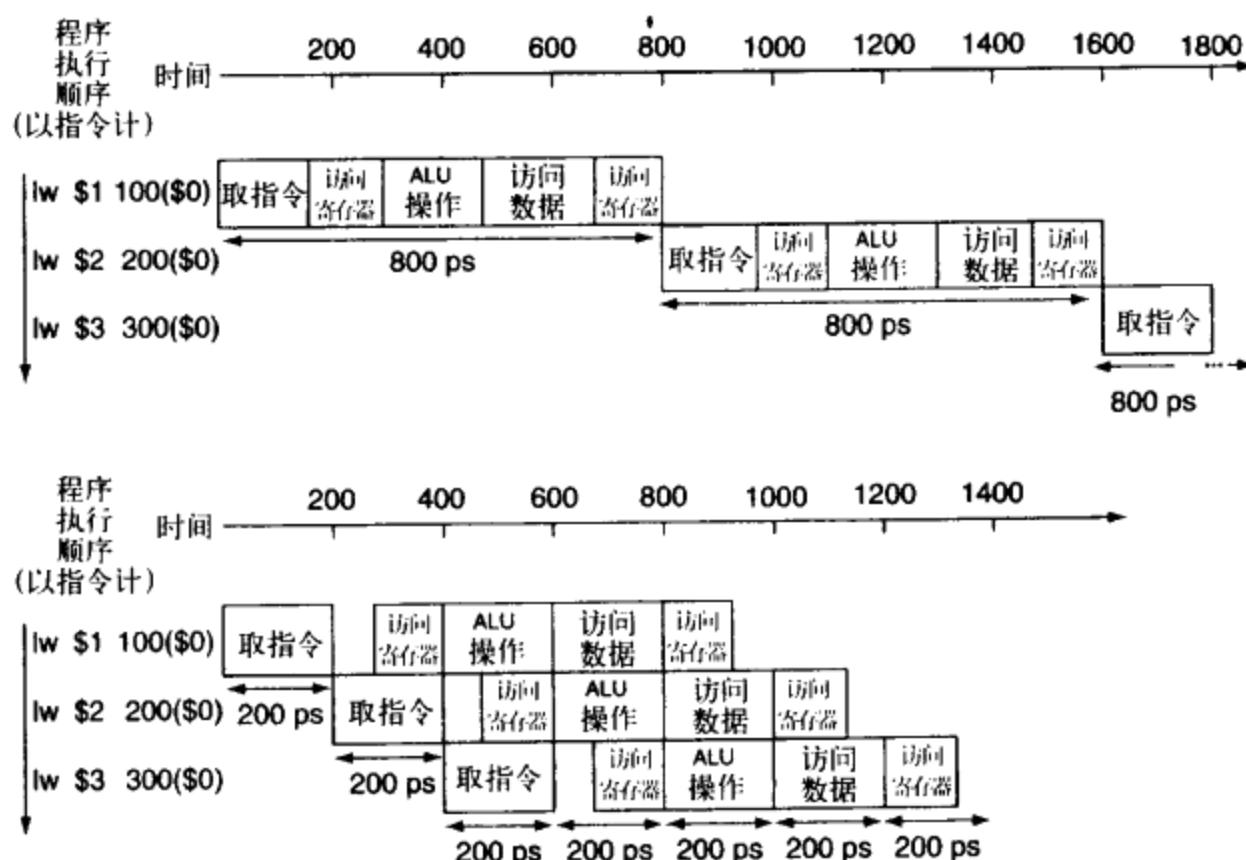


图 6-3 单周期、非流水线的指令执行过程(上图)与流水线的指令执行过程(下图)

[两者采用相同的硬件组件，各组件的处理时间如图 6-2 所示。在此种情况下，指令的执行速度提高了 4 倍，即从 800 ps 降到了 200 ps。将本图与图 6-1 比较。在洗衣服的例子中，我们假设所有的处理过程的完成时间都是相等的。如果烘干机运行得最慢，那么就把烘干的时间设定为各步骤需要的处理时间。计算机的流水线过程处理时间也是受限于最慢的处理资源，即 ALU 操作和内存访问。同时我们假设对寄存器堆的写操作发生在时钟周期的前半段，对寄存器堆的读操作发生在时钟周期的后半段，并且本章所有内容都遵循这个假设]

流水线的每级都只花费一个时钟周期的时间，所以时钟周期必须足够长以满足最慢的操作执

行需要。正像在单周期模型中虽然有些指令的执行只需要 500 ps，但它必须选择 800 ps 作为时钟周期一样，流水线执行模型的时钟周期也必须选择最坏情况所对应的 200 ps，尽管有些级可以达到 100 ps。即使这样，流水线仍能将性能提高四倍：第一条与第四条指令之间的时间差缩短为 3×200 ps，即 600 ps。 ■

我们可以把上面讨论的流水线模型所能获得的性能加速比归纳为一个公式。如果流水线的各级分配均匀，那么在流水线处理器上单个指令执行时间为（在理想情况下）：

$$\text{指令执行时间}_{\text{流水线化}} = \frac{\text{指令执行时间}_{\text{非流水线化}}}{\text{流水线的级数}}$$

即在理想的情况下，流水线所带来的加速比与流水线的级数大致相同；一个五级流水线所能获得的加速比大致为 5。

这个公式说明一个五级流水线相对 800 ps 的非流水线执行时间有 5 倍的提高，也就是相当于 160 ps 的时钟周期。然而正如例子所示，各级之间并非完全平衡。此外，流水线还会引入一些常规的额外开销，我们稍后会对此详细论述。所以，在流水线处理器中，每一条指令的执行时间最终会超过这个最小的可能值，从而流水线能够获得的加速比也就小于流水线的级数。

此外，即使我们在前面的分析中断言能将三条指令的执行速度提高四倍，但本例子三条指令的执行中并没有反映出来，它实际获得的加速比为 2400 ps/1400 ps。为什么实际的加速比小了许多呢？这是因为指令的数目不够多。如果增加执行指令的数目将会发生什么呢？我们首先将前面的图中的指令增加到 1 000 003 条，也就是说，再在上面的流水线例子中加入 1 000 000 条指令；每一条指令都将会使整个的执行时间增加 200 ps。那么整体执行时间就变成 $1 000 000 \times 200$ ps + 1400 ps，即 200 001 400 ps。在未采用流水线的例子中，我们也加入 1 000 000 条指令，每条指令的执行时间是 800 ps，因此整个的执行时间为 $1 000 000 \times 800$ ps + 2400 ps，即 800 002 400 ps。在这些理想条件下，非流水线处理器与流水线处理器的实际执行时间的比值就非常接近与两者指令平均执行时间的比值，即：

$$\frac{800\ 002\ 400\ \text{ps}}{200\ 001\ 400\ \text{ps}} \approx 4.00 \approx \frac{800\ \text{ps}}{200\ \text{ps}}$$

流水线所带来的性能提升是通过提高指令的吞吐率来实现的，而不是通过减小单条指令的执行时间来实现。由于实际程序都会执行上亿条指令，因此指令的吞吐率是一个很重要的参数。

6.1.1 针对流水线结构的指令集设计

尽管上面的例子只是对流水线的最简单的说明，我们也能通过它讨论 MIPS 指令集的设计，而 MIPS 指令集就是专为指令执行的流水线化而设计的。

首先，所有的 MIPS 指令都有相同的长度。这一限制将简化流水线的第一级指令取与第二级指令译码的过程。在诸如 IA-32 之类的指令集中，指令的长度并不相同，从 1 字节到 17 字节不等，这样将会给流水线的执行带来相当大的挑战性。我们在第 5 章中已经介绍了：现在所有对 IA-32 体系结构的实现实际上都是将 IA-32 指令翻译为简单的微操作，这些微操作与 MIPS 指令非常类似。我们在 6.10 节中将看到，Pentium 4 处理器实际上是将各个微操作进行流水线处理，而不是原始的 IA-32 指令。

第二，MIPS 只有很少的几种指令格式，并且每条指令中的源寄存器字段都固定不变。这种指令结构的对称性意味着在流水线的第二级中，也就是在硬件确定指令类型的同时，就可以开始读取寄存器堆。如果 MIPS 指令格式是非对称的，我们就需要将第二级一分为二，而流水线级数也就相应增加为 6。我们不久将看到较长流水线的缺点。

第三，MIPS 中对内存的操作仅出现在存取操作中。这一限制意味着可以在指令执行级计算

内存地址，就可以接着在下一个步骤访问内存。如果我们允许可以直接操作内存中的操作数，就像在 IA-32 中那样，那么第 3 级与第 4 级将被扩展为地址计算、内存访问和执行三个步骤。

第四，正如第 2 章讨论的，内存中的所有操作数必须对齐。因此，我们不需要担心一条数据传输指令需要两次内存访问的情况；所请求的数据可以在单个流水线步骤内完成在处理器和内存之间的传输。

6.1.2 流水线冒险

流水线有这样一种情况，在下一个时钟周期中下一条指令不能执行。这种情况称为冒险 (hazard)，我们将其分为三类。

结构冒险

第一种冒险叫做结构冒险^①。即硬件不支持多条指令在同一个时钟周期内执行。在洗衣店例子中，如果用洗衣-烘干机代替独立的洗衣机与烘干机，或者如果我的室友正在做其他的事情而不能帮助我将衣服收拾好，都会发生结构冒险。如果发生上述情况，那我们精心构筑起来的流水线就会被破坏。

正如我们在上面所说的那样，MIPS 指令集是专为流水线设计的，这使得设计者在设计流水线时能够很容易地避免结构冒险。假设图 6-3 的流水线结构只有一个内存而不是两个内存。如果有第四个指令的话我们将会发现，在某一个时钟周期，第一条指令在访问内存的同时第四条指令将在同一内存中预取指令。如果没有两个内存的话，流水线就会发生结构冒险。

数据冒险

在一个操作必须等待另一操作完成后才能进行时，流水线必须停顿，我们称这种情况为数据冒险^②。假设你正在折叠衣服时遇到一只袜子，但是很遗憾你没有找到与它对应的另外一只。一种解决方法是回到你房间找回那只丢失的袜子。显然，当你在寻找那只袜子时，烘干机又产生出新的需要叠起来的衣服，而那些刚洗好的衣服也必须等待使用烘干机。

在一个计算机流水线中，处于流水线内部的各条指令之间的数据相关关系会导致数据冒险，而这种情况在洗衣服的例子中是不存在的。例如，假设我们有一条加法指令，它之后紧跟着一条减法指令，而这条减法指令要使用加法指令的和 (\$s0)：

```
add    $s0, $t0, $t1  
sub    $t2, $s0, $t3
```

在不做任何干涉的情况下，这一数据冒险会严重地阻碍流水线。加法指令直到第五步才能写回它的结果，这就意味着我们必须在流水线当中引入三个气泡。

虽然我们可以试图依靠编译器来避免这种数据冒险，但实际上很难获得令人满意的效果。这种相关关系的发生过于频繁，同时会导致相当大的延迟，以致于不可能完全指望编译器来为我们摆脱该困境。

我们有一种最基本的解决方法，这种方法主要基于以下观察，即不需要等待指令执行完成就能解决数据冒险。对于上述的代码段，一旦 ALU 生成了加法运算的结果，我们就可将它用作减法运算的一个输入项。通过添加额外的硬件，我们能从内部资源中提前得到所缺少的运算项，这

① 结构冒险(structural hazard) 是指一条本来在某个时钟周期计划要执行的指令，由于硬件缺乏对当前要执行的所有指令的支持，而不能执行的情况。

② 数据冒险(data hazard) 也称为流水线数据冒险，是指由于指令执行所需的数据暂时不可用而造成将要执行的指令不能在原定时钟周期内执行的情况。

个过程称为数据定向或者数据旁路^①。

例题 两条指令间的数据转发

对于上述的两条指令，说明如何将流水线各级连接起来才能实现数据转发。图 6-4 描述了流水线的五级的数据通路。按照图 6-1 中的洗衣店流水线所示，把每条指令的数据通路排成一列。

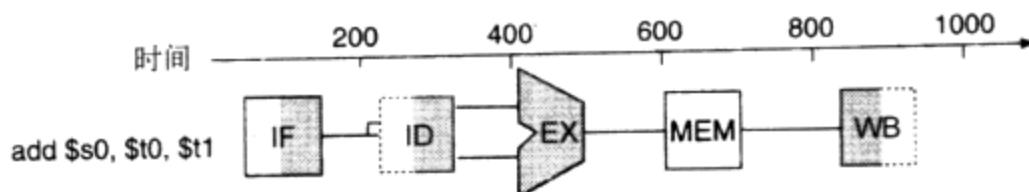


图 6-4 指令流水线的图形表示，其基本思想与图 6-1 中的洗衣店流水线类似

[在本图以及本章所有内容中，我们使用图形符号来代表流水线各级使用的物理资源。这些符号在五个流水线级中所代表的意义分别是：IF 表示指令的预取，其外方框表示指令的内存；ID 表示指令的译码或寄存器堆的读取，外边的虚线方框表示要读取的寄存器堆；EX 表示指令的执行，其外面的框表示 ALU；MEM 代表内存访问，包围它的方框代表数据内存；WB 表示写回，包围它的虚线方框代表被写回的寄存器堆。阴影表示该资源被指令所使用。因此，MEM 没有阴影，因为 add 指令在这一步并不读取数据内存。上图各级(IF, ID, WB)右半边的阴影表示在该级中读取寄存器堆或内存，左半边的阴影表示在该级中写入寄存器堆或内存。因此，由于第二步需要读取寄存器堆，ID 的右半边有阴影，而由于第五步中需要写入寄存器堆，WB 的左半边有阴影]

解 图 6-5 显示了数据定向，通过这条旁路，在 add 指令通过执行阶段后 \$s0 的值将作为 sub 指令在执行阶段的输入。 ■

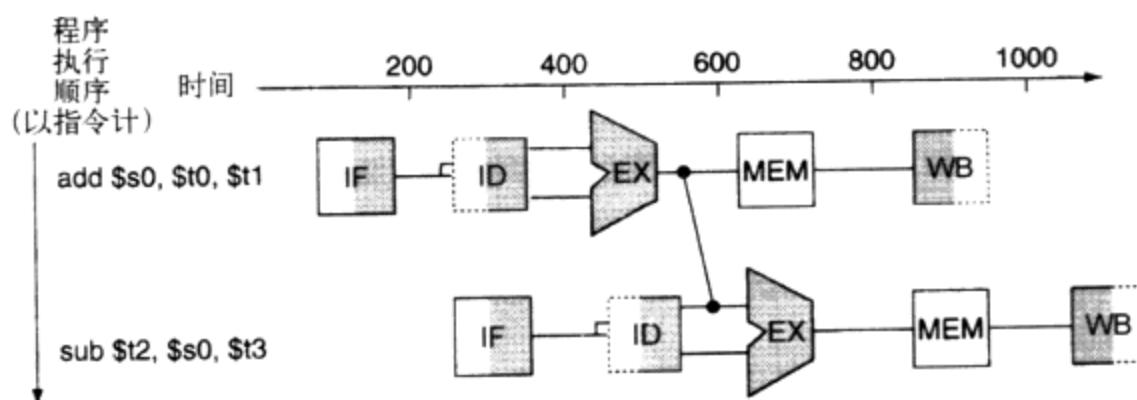


图 6-5 数据转发的图形表示

[图中的连线表示从 add 指令 EX 阶段的输出到 sub 指令 EX 阶段的输入的转发路径，它替换掉了 sub 指令第二步从寄存器 \$s0 读取的值]

在图中的各个事件中，只有当定向的目标阶段在时序上晚于定向的源阶段，数据旁路才有效。例如，第一条指令内存访问阶段的输出不能通过数据旁路作为下一条指令执行阶段的输入，因为这将意味着时间的倒流。

定向可以很好地工作，其具体内容将在 6.4 节中详细介绍。然而它并不能够避免所有的流水线阻塞。例如，假设第一条指令不是 add 而是载入 \$s0 寄存器的内容，我们可以通过图 6-5 所示的结构想象一下，由于数据间的相关性，所需要的数据只有在前一条指令完成流水线的第 4 级后才有效，这对于 sub 指令的第 3 级的输入请求来说就太迟了。因此，如图 6-6 所示，即使使用了转

^① 数据定向(forwarding) 也称数据旁路(bypassing)是一种解决数据冒险问题的方法，使用内部的数据缓存直接提供缺少的数据，而不是等待该数据到达程序员可见的寄存器或者内存才去使用它。

发机制，在加载-使用型数据冒险^①情况下，流水线仍然不得不阻塞一步。这幅图还引入了流水线中一个重要的概念，我们称之为流水线阻塞^②，它还有一个昵称：气泡(bubble)。我们将会遇到流水线中其他原因产生的气泡。为了解决这类棘手问题，在6.5节我们将介绍一些具体方法，包括硬件的检测及阻塞，以及将载入延迟当做分支延迟来处理的软件方法。

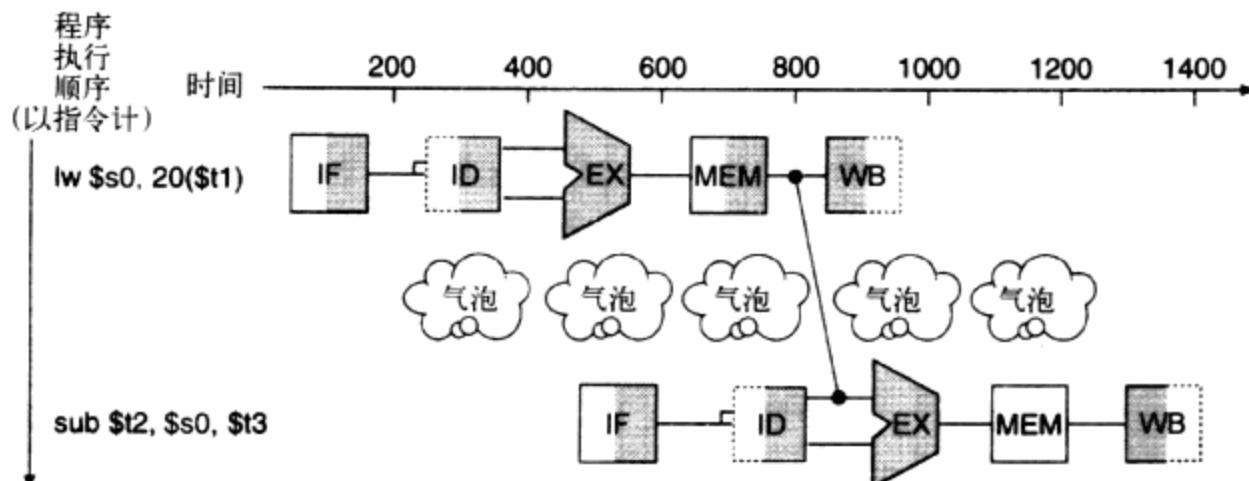


图 6-6 即使使用了转发机制，当一条 R 型指令之前紧接着便是一条产生该指令所需数据的取指令时，仍然会产生一次阻塞

[如果不引入一次阻塞的话，从内存访问步骤的输出到执行步骤的输入之间的路径在时间上将是倒着的，这显然是不可能的。这张图仅仅是一个示意图，这是由于我们在对 sub 指令取指令和译码之前是不知道应当引入必要的阻塞的。在 6.5 节中我们将详细描述冒险发生时的真实情景]

例题 重新排列代码以避免流水线阻塞

考虑下面的 C 语言代码：

```
A = B + E;
C = B + F;
```

下面是上述代码段翻译为 MIPS 指令集得到的代码段，我们假设所有变量都在内存中并由 \$t0 进行寻址：

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add    $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add    $t5, $t1,$t4
sw      $t5, 16($t0)
```

找出上述代码段中的冒险关系，并对上述指令进行重新排序以防止流水线阻塞现象。

解 两条 add 指令都会产生冒险，这是由于它们各自对应的 lw 指令之后紧接着就是这两条指令本身。同时我们可以注意到数据转发消除了其他几个潜在的数据冒险关系，包括第一个 add 指令与第一个 lw 指令之间的冒险，以及所有存储指令引起的数据冒险。通过将第三个 lw 指令前提就能消除前面提到的那两个数据冒险：

```
lw      $t1, 0($t0)
lw      $t2, 4($t1)
lw      $t4, 8($t0)
add    $t3, $t1,$t2
```

- ① 加载-使用型数据冒险(load-use data hazard) 是一种数据冒险的特殊形式，指当某个时刻被请求的数据尚未产生(其值尚无效)的情况。
- ② 流水线阻塞(pipeline stall) 也称气泡。指流水线为了解决冒险情况而引入的停顿。

```

sw      $t3, 12($t0)
add    $t5, $t1,$t4
sw      $t5, 16($t0)

```

在一个流水线处理器上，经过重新排序的指令序列将比原序列早两个周期完成。 ■

数据转发的设计让我们注意到另一个关于 MIPS 结构设计的要点。每条 MIPS 指令至多输出一个结果，并在接近流水线的末端完成对这个结果的写回。如果为每条指令转发的结果多于一个，或者指令在执行过程中采用较早写回策略的话，我们将更难实现数据转发。

细节：“数据转发”术语命名的根源是转发的过程其实是由一条较早执行的指令将结果传送给另一条较晚的指令。而术语“数据旁路”则是由于这个过程实际上是将计算结果不通过寄存器堆直接提供给需要的计算单元。

控制冒险

第三种冒险叫做控制冒险^①。这种冒险会在下面的情况下出现：处理器需要根据一条指令的结果做出决策，此时其他的指令可能仍在执行过程中。

假设洗衣店的店员们接到了一个令人高兴的任务：为一个足球队清洗队服。由于衣服非常脏，我们需要确定清洗剂的用量，还要了解水的温度设置是否能够将衣服清洗干净，但同时要保证不能过量以避免过度磨损衣物。在洗衣店流水线中，店员只有等到第二步烘干衣服以后才能确定是否需要改变洗衣机的设置。在这种情况下应该怎么办呢？

有两种办法可以解决洗衣店的控制冒险，同样的方法也可以应用到计算机中。

阻塞：在第一批被烘干之前按串行的方式操作，并且重复这一过程直到找到正确的洗衣机设置为止。这种保守的方法当然可以保证正常工作，但它的速度比较慢。

计算机中对应的决策任务就是分支指令。我们注意到必须在读取分支指令的每一个周期继续取指令。但是由于流水线刚从内存读取了一条分支指令，它不可能知道下面应该取哪条指令，就像洗衣店例子一样，一种可能的解决方法是当我们取到一条分支指令的时候立即将流水线阻塞下来，直到完成该分支指令为止，这时我们就知道具体应该取内存中的哪条指令了。

下面我们假定设计了足够多的硬件，以至于我们可以在流水线的第二级完成寄存器取值、分支地址计算和更新 PC 的操作(详见 6.6 节)。即便我们使用这些额外的硬件，含有条件分支指令的流水线仍然如图 6-7 所示。如果分支失败(也就是输出为 FALSE)，那么将执行 lw 指令，则这条指令将被阻塞一个周期，也就是 200 ps。

例题 “分支阻塞”对性能的影响

估计分支带来的阻塞对指令消耗的平均时钟周期数(clock cycles per instruction, CPI)的影响。假定其他所有指令的 CPI 值均为 1。

解 图 3-26 显示在 SPECint2000 中条件分支指令所占比例为 13%。由于其他指令的 CPI 都为 1，而分支指令要增加一个时钟周期的阻塞，因此相比理想状态而言 CPI 平均值为 1.13，也就是说速度也下降了 1.13 倍。值得注意的是，我们仅仅计入了分支指令，而跳转指令还有可能进一步引入阻塞。 ■

如果不能在流水线第二级中解决分支问题(对于较长的流水线而言这很正常)，那么再对分支

^① 控制冒险(control hazard) 又称分支冒险(branch hazard)，是指在某个周期内应当被执行的指令由于当前取得的指令不是这条真正需要的指令而不能被执行的情况；也就是说，这时指令流中的地址并不是流水线真正取得的地址。

指令采取阻塞将导致更大的速度下降。对大多数计算机而言，这种阻塞方法的代价太大。因此也就产生了另外一种消除控制冒险的方法：

预测：如果你自信有正确的洗衣设置来洗涤那些队服的话，那么就可以在第一批衣服烘干的同时清洗第二批衣服。这种做法在预测正确的时候不会降低流水线的速度。但是一旦预测错误，就不得不重说预测失误所洗的衣服。

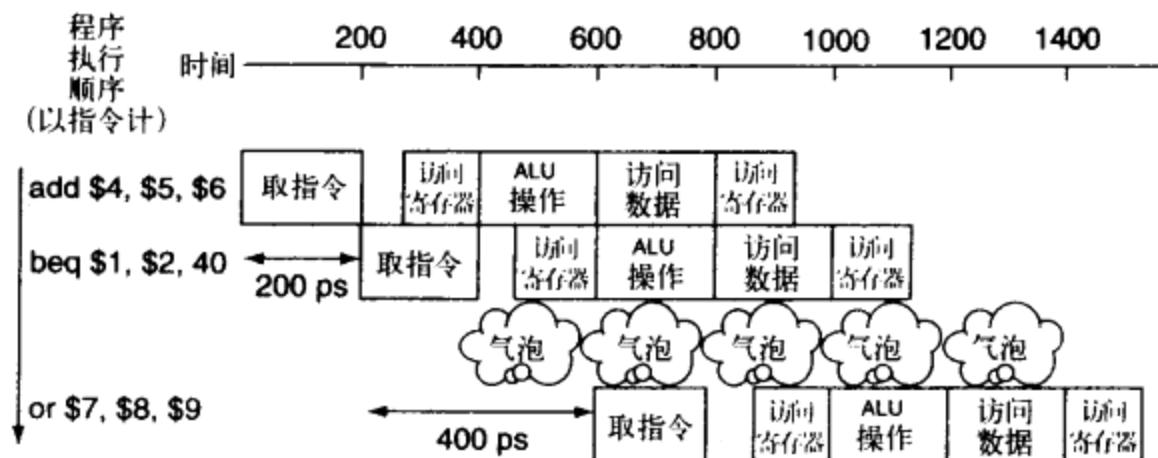


图 6-7 在每个条件分支语句之后阻塞流水线以解决控制冒险

[在分支之后，有一个长度为 1 级的流水线阻塞，即气泡。在 6.6 节我们可以看到，在实际中产生一个阻塞的过程要稍复杂。而阻塞流水线所带来的性能损失与插入一个气泡是相同的]

计算机的确是采用预测方法来处理分支的。一种简单的预测方法就是总是预测分支未执行[○]。当你预测正确(即分支失败)的时候，流水线会全速进行处理。只有当分支成功时流水线才会阻塞。图 6-8 给出了这样一个例子。

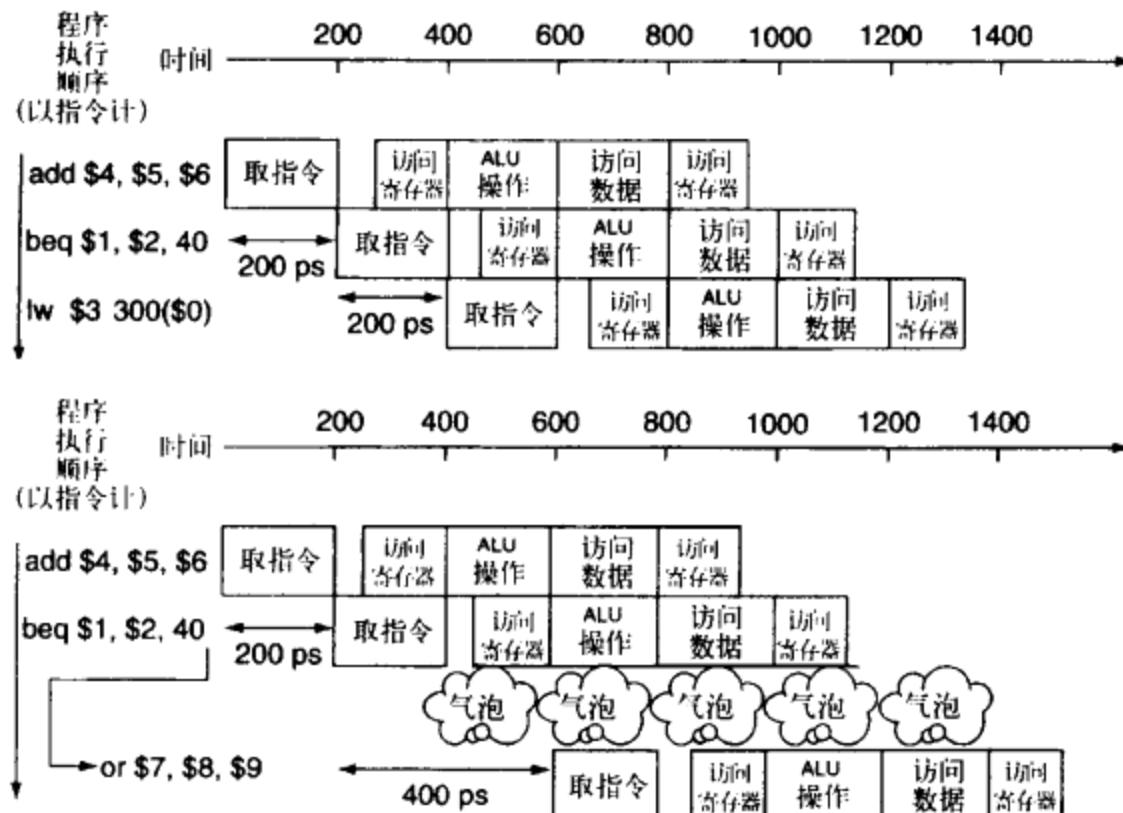


图 6-8 预测分支未执行是避免流水线控制冒险的一种解决方法

[上图显示的是分支没有被执行的流水线；下图显示的是分支发生了的流水线。正如我们在图 6-7 中所说的一样，插入气泡的方式是实际情况的简化版本，至少对于紧跟分支后的那个时钟周期是这样的。具体情况见 6.6 节]

○ 分支未执行(untaken branch) 指那些输出结果导致继续执行下一条指令的分支语句。被执行的分支则是结果导致控制转向分支目标的分支语句。

一种更复杂的分支预测[○]方法是预测一些分支成功而预测另外一些分支失败。类比上面的洗衣店的例子，深色队服即主场比赛队服使用一种洗衣设置，而浅色即客场比赛队服则使用另一种设置。以计算机为例，循环体的底部分支总是会分支回到循环体的顶部。由于这些分支总是被执行并且总是向前跳转，因此我们可以预测前向跳转的指令总是会被执行。

这种分支预测的僵硬方法依赖于一成不变的行为，也没有考虑每条分支指令各自的特点。动态硬件预测器与这种方法截然不同，它根据每一个分支的行为进行预测，在整个的程序生命期内对某一条分支指令的预测结果还有可能改变。类比洗衣店的例子，如果使用动态预测方法，店员将会通过对制服肮脏程度的观察推测一个洗衣设置，并根据本次预测的成功与否调整下一次的预测行为。动态分支预测的一种比较普遍的实现方法是保存每个分支的历史记录，然后利用分支的近期记录来预测未来可能的行为。我们将会看到，动态分支预测中达到 90% 的正确率可能记录的历史类型和数量所带来的开销是巨大的（见 6.6 节）。当预测错误时，流水线控制机制必须确保错误预测分支下的指令不会生效，并且在正确的分支地址处重新启动流水线。在类比的洗衣店例子中，我们必须停止放入下一批衣物，以便重洗预测错误的那批衣物。

如同其他解决控制冒险的方法一样，较长的流水线会恶化预测的性能，它将增加错误预测的代价。控制冒险的解决办法在 6.6 节中将有更加详细的介绍。

细节：还有一种解决控制冒险的方法，即延迟决定（delayed decision）。类比洗衣店的例子，每当要决定如何洗衣服时，就在第一批足球队队服洗好之后将一批不是足球队队服的衣物放进洗衣机里，同时等待足球队的制服被烘干。只要有足够多不需决定洗衣设置的脏衣服，这种方法就能很好地工作。

在计算机中这种方法被称为延迟分支，这种方法在 MIPS 系统结构中也得到了实际应用。在延迟分支策略中，分支指令的下一条指令将总是被执行，而在该指令之后再开始执行分支。延迟分支并没有暴露给 MIPS 汇编语言编写者，这是由于汇编器会自动排列指令，以使得分支的行为达到程序员的要求。MIPS 软件会在延迟分支指令的后面紧跟着放置一条不受该分支影响的指令。发生了的分支会改变这条安全指令之后的指令的地址。在我们的例子中，图 6-7 中分支前的指令 add 不影响分支，因此可以放置在分支指令后面以完全屏蔽分支指令造成的延迟。因为延迟分支对于分支延迟较短的情况比较有用，所以没有什么处理器采用多于一个周期的延迟分支。对于较长的分支延迟而言，我们惯常使用硬件分支预测器。

6.1.3 流水线概述的总结

流水线是一种用于开发顺序指令流中指令间并行性的技术。与其他加速技术不同（参见■第 9 章），其优势在于它对程序员是透明的。

在本章的以下几节中，我们通过 MIPS 的指令子集 lw、sw、add、sub、and、or、slt 和 beq（见第 5 章）及其简化的流水线方式介绍关于流水线的一些基本概念。然后讨论引入流水线所带来的一些问题以及在一些典型情况下所能够达到的性能。

如果你想将精力集中在软件方面并了解流水线的大致性能，那么我们相信在完成了本节的学习以后，你已经具有足够的背景知识并可直接跳到 6.9 节，去了解一些流水线的高级概念，如超标量流水线与动态调度，在 6.10 节我们将对 Pentium 4 微处理器流水线进行分析。

○ 分支预测（branch prediction） 一种用来解决控制冒险（分支冒险）的方法，预测分支指令的输出结果并根据这个结果继续计算过程，而不是等待确定的实际输出结果。

反之，如果你想深入了解流水线的实现技术以及如何处理冒险现象，你可以接着阅读后面的各节，在6.2节我们将对一个流水线化的数据通路的设计进行分析；6.3节介绍流水线数据通路的基本控制。基于对这些知识的理解，我们在6.4节将对数据转发的具体实现进行阐述，在6.5节对流水线阻塞的设计进行分析。之后你可以阅读6.6节以详细了解分支冒险的处理方法，以及6.8节中异常的处理方法。

重点

流水线增加了同时执行的指令的数目，提高了处理器的吞吐率。流水线并不能减少单一指令的执行时间，我们称这个时间为时延[⊖]（或延迟）。例如一个五级流水线仍然需要五个周期来完成一条指令。用第4章的术语来描述：流水线提高了指令的吞吐率（throughput）而不是减少了单条指令的执行时间或时延。

对流水线的设计者们来说指令集既可能将事物简单化，也可能将事物复杂化。流水线设计者必须解决结构冒险、控制冒险和数据冒险。而分支预测、转发和流水线阻塞机制能够在保证得到正确结果的前提下提高计算机的性能。

理解程序性能

除了内存系统外，流水线所进行的有效计算是决定处理器CPI值的最重要的因素。我们将在6.9节看到，如何衡量一个现代的多发射流水线处理器的性能是一项复杂的任务，仅有对简单流水线处理器的理解是远远不够的。然而，无论在简单还是复杂流水线中，对各种冒险情况（结构、数据和控制冒险）的理解都十分重要。

对于现代的流水线而言，结构冒险总是与浮点运算单元相关，这是由于这些单元不能被有效地流水线化，而控制冒险更多出现在整数程序当中，这些程序往往含有更多的分支指令以及更多难以预测的分支。无论对于整数程序还是浮点程序而言，数据冒险都可能成为性能的瓶颈。浮点程序中的数据冒险相对而言较容易处理，这是由于浮点程序的分支指令更少，访问模式也更规则，这都为编译器调度指令避免冒险提供了空间。对于整数程序而言，由于其访问模式更不规则，同时更多地使用了指针，所以优化的难度更高。我们将在6.9节介绍更高级的编译器技术和硬件技术，它们通过指令调度来解决数据依赖问题。

自测

对于下面的各段代码，阐述它们是否会阻塞，如果会阻塞那么是否可以通过数据转发得到解决。

代码序列1	代码序列2	代码序列3
lw \$t0, 0(\$t0) add \$t1, \$t0, \$t0	add \$t1, \$t0, \$t0 addi \$t2, \$t0, #5 addi \$t4, \$t1, #5	addi \$t1, \$t0, #1 addi \$t2, \$t0, #2 addi \$t3, \$t0, #2 addi \$t3, \$t0, #4 addi \$t5, \$t0, #5

⊖ 时延(latency) 流水线的级数或者顺序执行过程中两条指令之间的流水线级数。

6.2 流水线的数据通路

这里面看起来东西挺多，其实不然。

Tallulah Bankhead, remark to Alexander Wollcott, 1922

图 6-9 是摘自第 5 章的一个单时钟周期的数据通路。将指令划分为五个阶段意味着这是一个五级流水线，同时意味着在任何一个单时钟周期内，最多有五条指令会位于流水线中。因此我们必须把数据通路也分为五个部分，每一部分用与它相对应的指令执行阶段来命名。

- 1) IF: 取指令。
- 2) ID: 指令译码，读寄存器堆。
- 3) EX: 指令执行或地址计算。
- 4) MEM: 数据内存访问。
- 5) WB: 写回。

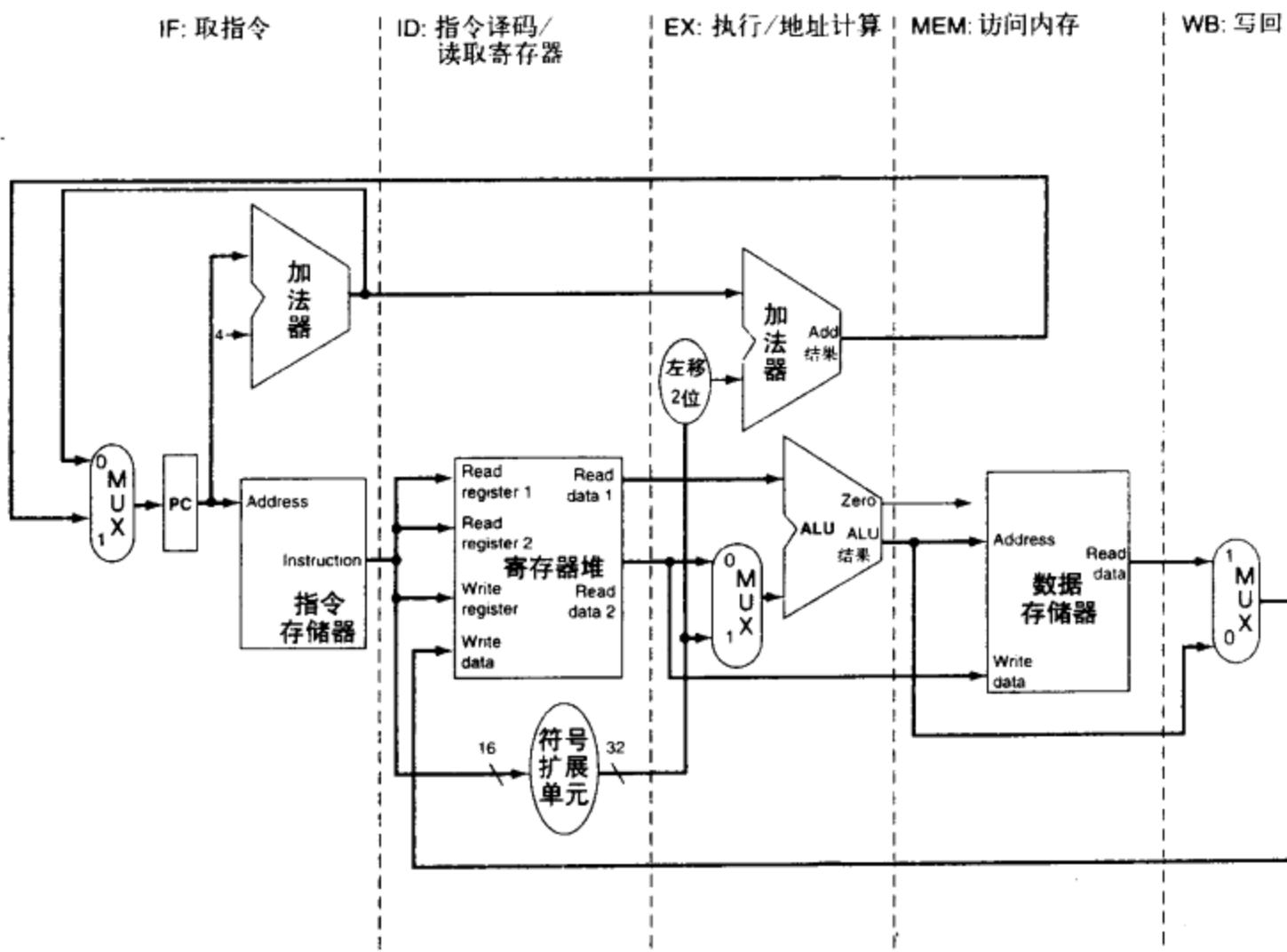


图 6-9 单周期的数据通路(与图 5-17 类似)

[指令在流水线中的各个步骤都可以与图中所示的数据通路一一对应起来。惟一的例外是 PC 的更新与写回步骤(如图中上下两条线所示)，它们发送 ALU 结果或内存数据到左边的寄存器堆中(通常我们用粗线条代表控制，而在本图中的都是数据线)]

在图 6-9 中，图中的五个部分与数据通路大致对应；指令与数据随着执行的过程从左到右一次在这五个阶段中流过。正如在洗衣店中，衣服沿着一条工作线依次被清洗、烘干和整理，而不会反向移动。

然而，在这个从左到右的指令流中有两个例外：

- 写回阶段，它把结果写回到数据通路中间的寄存器堆中

■ 选择 PC 的下一个值，在 PC 与 MEM 阶段的分支地址间进行选择

从右向左的数据流并不会影响当前的指令；流水线中只有当前指令之后的指令才会受到这种反向运动数据的影响。需要注意的是第一个从右向左的箭头会导致数据冒险，而第二个会导致控制冒险。

一种表示流水线的数据通路的执行方法是假定每一条指令都有它独立的数据通路，然后把这些数据通路放在同一时间轴上表示它们之间的关系。图 6-10 在同一时间轴表示了图 6-3 中指令执行过程中各自的数据通路。我们仍然使用图 6-9 中的格式来表现图 6-10 中所示的关系。

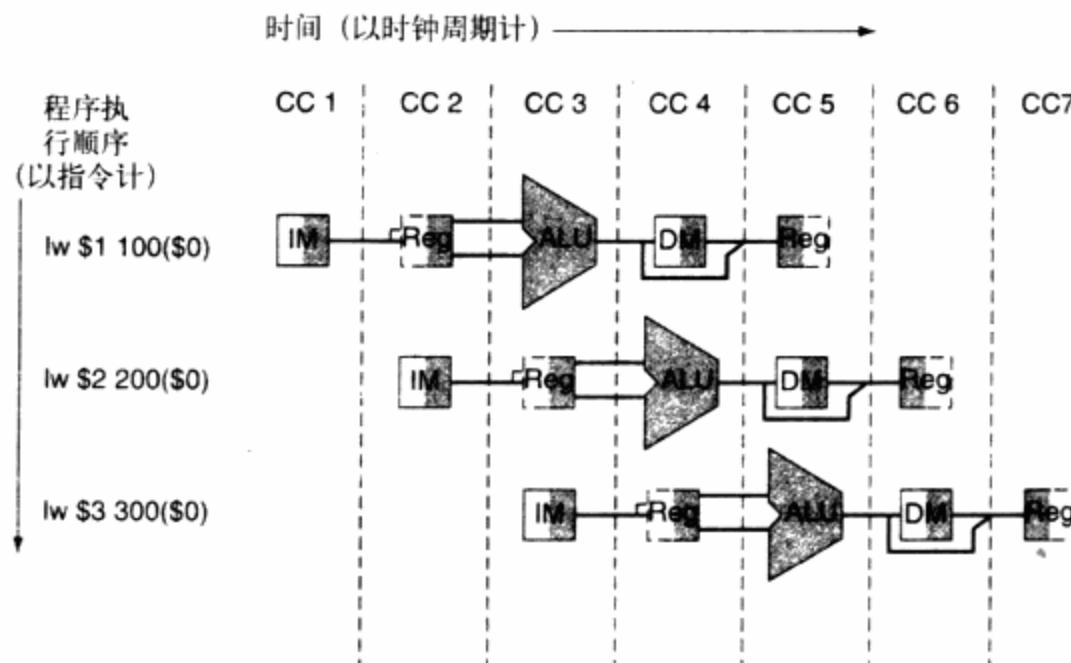


图 6-10 指令按图 6-9 中的单时钟周期数据通路执行(假定以流水线的方式执行)

[与图 6-4~图 6-6 类似，本图假设每一条指令有它独立的数据通路，并根据使用情况将相应的部分涂上阴影。与那些图不同的是，流水线的每一步都用该步骤使用的实际资源部件标示，分别对应图 6-9 中数据通路的相应部分。IM 表示指令内存与指令预取阶段的 PC；Reg 表示指令译码/寄存器堆读取阶段(ID)的寄存器堆和信号合成器，依此类推。为了保持正确的时序，这种格式的数据通路把寄存器堆从逻辑上划分为两个部分：寄存器读取(ID)阶段的寄存器读，和写回(WB)时的寄存器写。这种双重的应用在图中表示为：在 ID 阶段当它没有被写入的时候，将没有阴影的寄存器堆的左半部分用虚线表示，而在 WB 阶段当它没有被读时，则将没有阴影的右边部分用虚线表示。与以前一样，我们假设在时钟周期的前半部分写寄存器堆而在时钟周期的后半部分读寄存器堆]

图 6-10 看起来好像是三条指令需要三条数据通路。在第 5 章我们增加了保存数据的寄存器，从而使得在指令的执行过程中可以共享部分数据通路；在这里我们使用同样的技术共享多条数据通路。例如，在图 6-10 中指令内存只在每条指令的五个步骤中的一步中用到，因此我们允许它在其他四步中被其他的指令共享。

为了在其他四步中为各条指令保持各自的值，从指令内存中读出的数据必须保存在寄存器中。同样的方法应用到每个流水线步骤中，我们需要在图 6-9 中各级间有分割线的地方都加入寄存器(这种变化与第 5 章中在从单时钟周期到多时钟周期数据通路变化时增加寄存器的方法类似)。再回到洗衣店类比例子中，这时我们可以用篮子在两个步骤之间存放下一步的衣服。

图 6-11 描述了流水线的数据通路，其中流水线寄存器加灰底重点表示。在每个时钟周期中所有指令都会从一个流水线寄存器推进到另一个流水线寄存器中。我们用被这些寄存器分开的两个阶段来命名它们，如 IF 和 ID 阶段之间的流水线寄存器叫做 IF/ID。

需要注意的是在写回阶段的后面没有流水线寄存器。任何指令都会更新机器中的一些状态，如寄存器堆、内存或者是 PC 等，因此各个流水线寄存器堆对于更新后的状态来说是多余的。例如，一条取指令会把它的结果放入 32 个寄存器中的一个中，以后任何需要此数据的指令只需要读

取相应的寄存器就可以了。

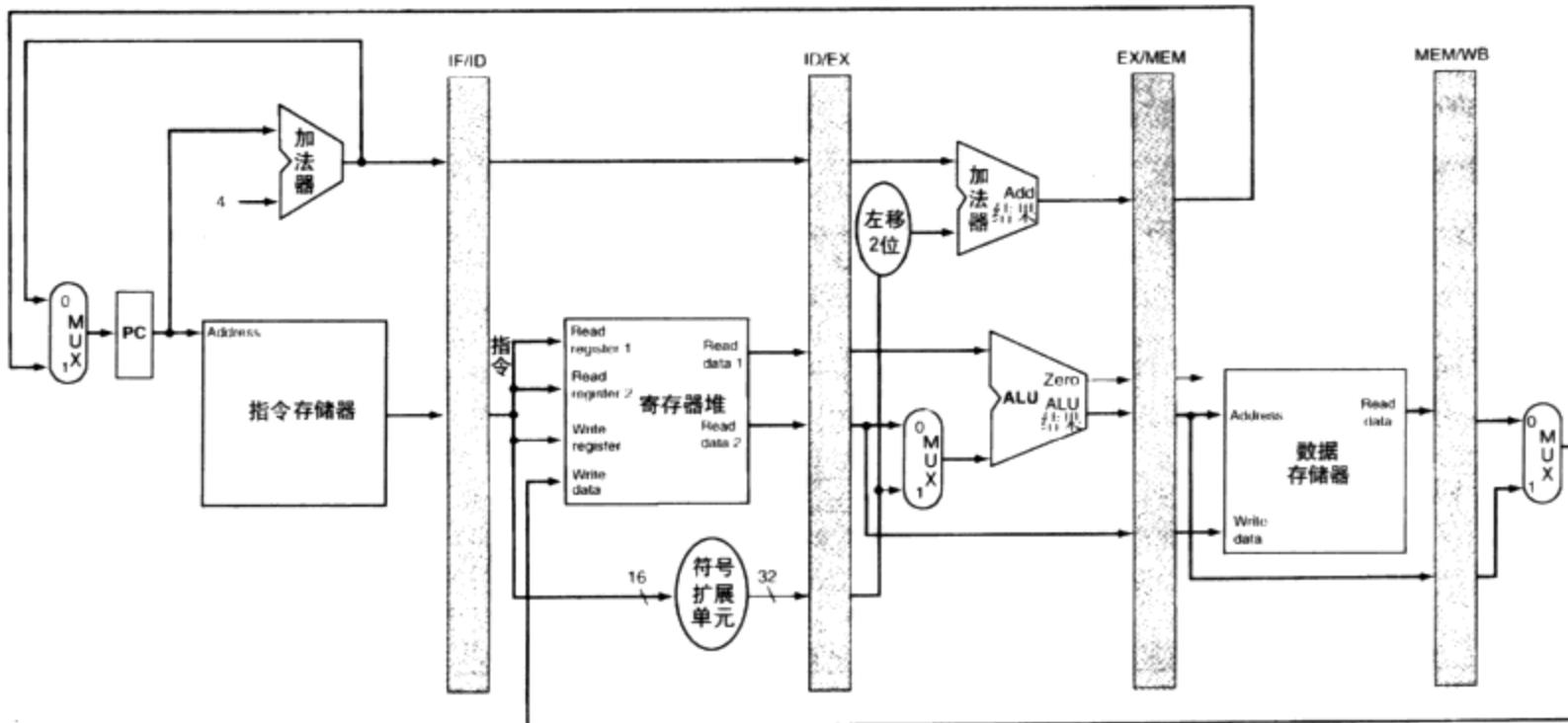


图 6-11 图 6-9 中数据通路的流水线版本

[流水线寄存器(图中阴影部分所示)将流水线的各部分分开。它们分别以被分开的各步骤为标志;例如,第一个被标为 *IF/ID* 是因为它将指令的预取与指令的译码阶段分开的缘故。为了存储所有穿过它的由线条代表的数据,寄存器的位宽必须足够大。例如,因为 *IF/ID* 寄存器必须同时保存 32 位的从内存中提取出来的指令及 32 位的 PC 增量的地址,它的位宽必须是 64 位。我们将在本章中渐渐增加寄存器的位宽的,目前另外三个流水线寄存器分别包含 128 位、97 位和 64 位]

当然,每条指令都会更新 PC:或者将其递增,或者将它设为分支目的地址。所以 PC 可以看成是流水线中的一个寄存器:它用于流水线的 *IF* 段。但是,与图 6-11 中那些用阴影标出的寄存器不同,PC 属于在指令集层次可见的寄存器;在异常发生时,PC 的内容必须被保存下来,而流水线寄存器的内容则可以直接被丢弃掉。类比洗衣店的例子,你可以将 PC 想象为洗衣开始前盛着脏衣服的那个篮子!

为了描述流水线的工作原理,本章将使用一系列图示来表示这些顺序的操作。这些额外的内容可能需要一定的时间去理解。但千万不要害怕;这些图片实际上比它们看上去要容易理解,因为你可以对比它们以观察每一个时钟周期内所发生的变化。6.4 节和 6.5 节将阐明当流水线中的指令间发生数据冒险时的具体情况;现在暂请忽略这一点。

图 6-12~图 6-14 是第一级图示,它们表示了一条取(load)指令在通过五级流水线时数据通路是如何工作的,在图中我们高亮标出数据通路。我们首先讲解 load 指令的原因是因为 load 指令在各级中始终是活动的。在图 6-4~图 6-11 中,当寄存器或者是内存被读取时我们在图中用阴影标出它们的右半部分;而当它们被写入的时候,我们在图中用阴影标出其左半部分。我们把每一幅图中活动的流水线步骤用指令的缩写 *lw* 标出。这五个步骤的情况如下:

- 1) 取指令:图 6-12 的上半部分表示指令使用 PC 中的地址从内存中读取数据,然后将数据放入 *IF/ID* 流水线寄存器中(*IF/ID* 流水线寄存器与图 5-26 中的指令寄存器类似)。PC 中的地址加 4 然后存入 PC 中为下一个时钟周期做准备。增加后的地址同时也存入 *IF/ID* 流水线寄存器中以备以后的指令使用,如 *beq*。计算机并不知道所预取的类型,所以必须为可能的任何指令做准备,并顺着流水线传递所有可能有用的信息。

- 2) 指令译码与读取寄存器堆:图 6-12 的下半部分表示的是 *IF/ID* 流水线寄存器的指令部分,

它提供 16 位的立即数字段(可扩展为带符号的 32 位数), 并读取两个寄存器的寄存器序号。然后将这三个值以及递增的 PC 地址存入 ID/EX 流水线寄存器中。这里我们也必须传递后面时钟周期的指令可能用到的所有信息。

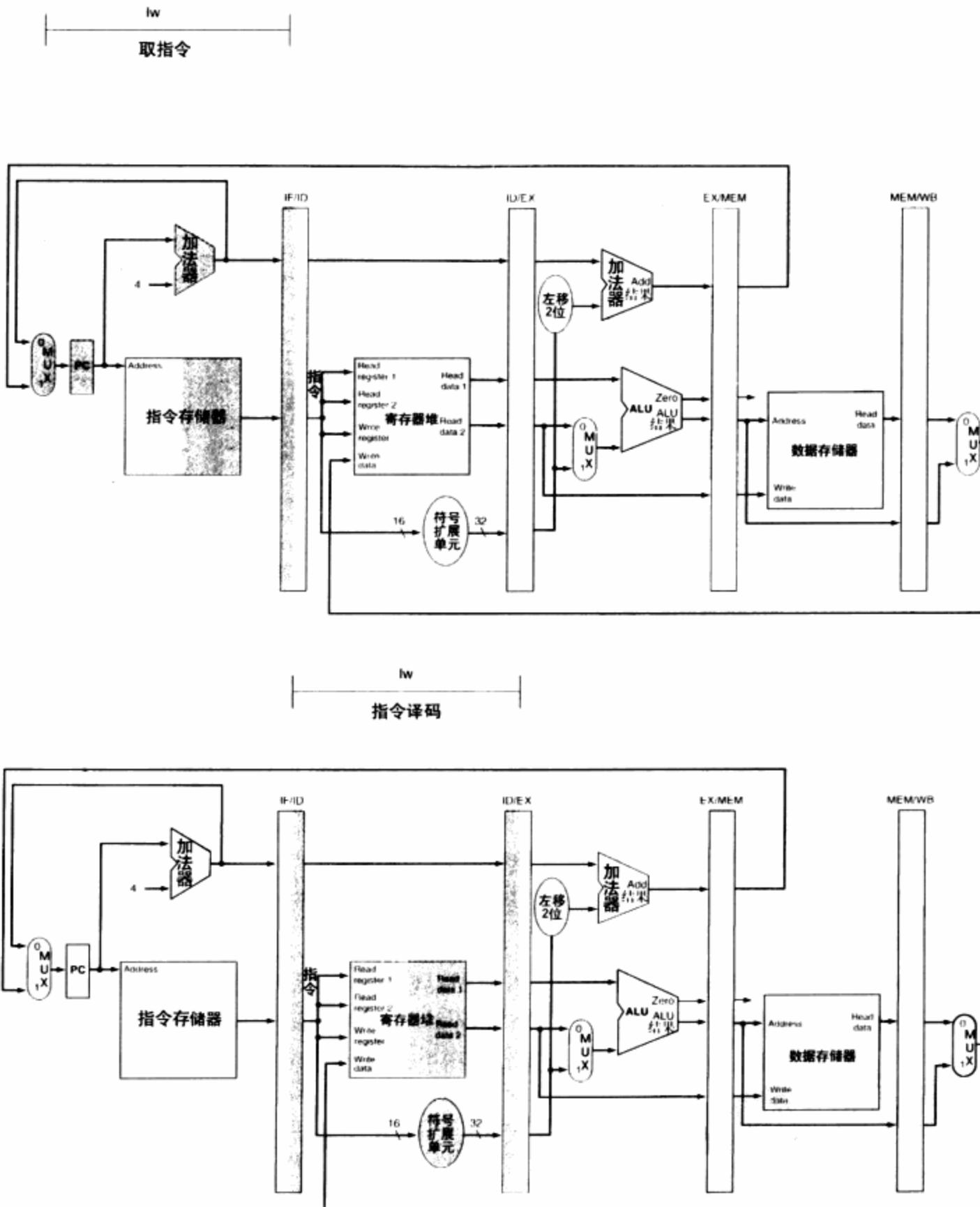


图 6-12 IF 与 ID: 指令在流水线中的第一步与第二步, 图 6-11 中数据通路的活动部分用阴影标出

[图中阴影部分的表示约定与图 6-4 中相同。与第 5 章相同, 读写寄存器并不发生冲突, 因为寄存器内容的变化只有在时钟周期的边缘发生。虽然 load 指令只需要第二个步骤中最上面的那个寄存器, 但由于处理器并不知道当前是哪一条指令正在被译码, 因此它把 16 位的常量进行符号扩展, 并把两个寄存器中的值都读入 ID/EX 流水线寄存器中。我们并不需要所有这三个操作数, 但是全部保留三个操作数能简化对它们的控制]

3) 执行或者地址计算: 图 6-13 表示 load 指令从 ID/EX 流水线寄存器中读取寄存器 1 中的内容以及扩展后带符号的立即数, 并用 ALU 将它们相加。加法的和被放入到 EX/MEM 流水线指令寄存器中。

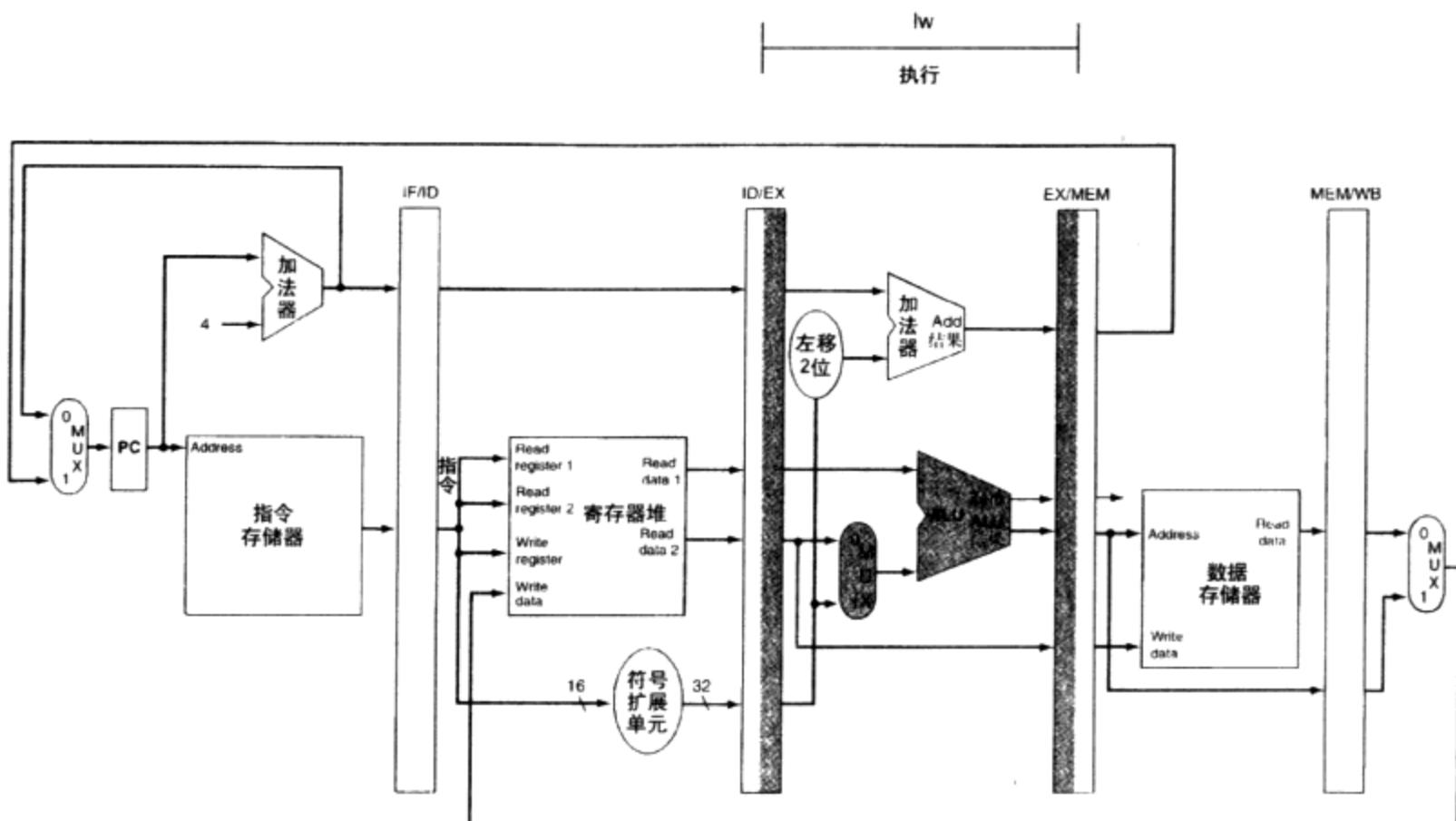


图 6-13 EX: load 指令在流水线中的第三步, 图中用阴影表示出
图 6-11 中的数据通路在流水线本级中的部分

[寄存器与经过符号扩展的立即数相加, 其和放入 EX/MEM 流水线寄存器中]

4) 内存访问: 图 6-14 的上半部分表示 load 指令使用从 EX/MEM 流水线寄存器中得到的地址读取数据内存, 并将数据载入 MEM/WB 流水线寄存器当中。

5) 写回: 图 6-14 的下图表示了最后一个步骤: 从 MEM/WB 流水线寄存器中读取数据并将它写入图中间的寄存器堆中。

通过对 load 指令的整个执行过程的分析, 我们可以确定在任何时刻, 哪些数据将被下一个流水线步骤使用到, 哪些数据就需要写入流水线寄存器中。纵观 store 指令的执行过程, 我们会发现除了要将有用的信息传递到后面的流水线步骤外, 指令的执行还有一定的相似性。下面是 store 指令的五个流水线级步骤:

1) 取指令: 利用 PC 中的地址从内存中读出指令, 然后将指令放入 IF/ID 流水线寄存器中。这个步骤发生在指令被确认之前, 所以图 6-12 中的上图的内容既适用于 load 指令也适用于 store 指令。

2) 指令译码与读取寄存器堆: IF/ID 流水线寄存器中的指令支持读取两个寄存器的寄存器号和经过符号扩展的 16 位立即数。这三个 32 位的数都存放在 ID/EX 流水线寄存器中。图 6-12 中的下图同时也可描述了 store 指令的第二个流水线步骤。由于此时无法得知要执行的指令的类型, 因此这两个步骤在所有的指令执行过程中都会被执行。

3) 指令执行或地址计算: 图 6-15 描述了流水线的第三个步骤; 有效地址存放在 EX/MEM 流水线寄存器中。

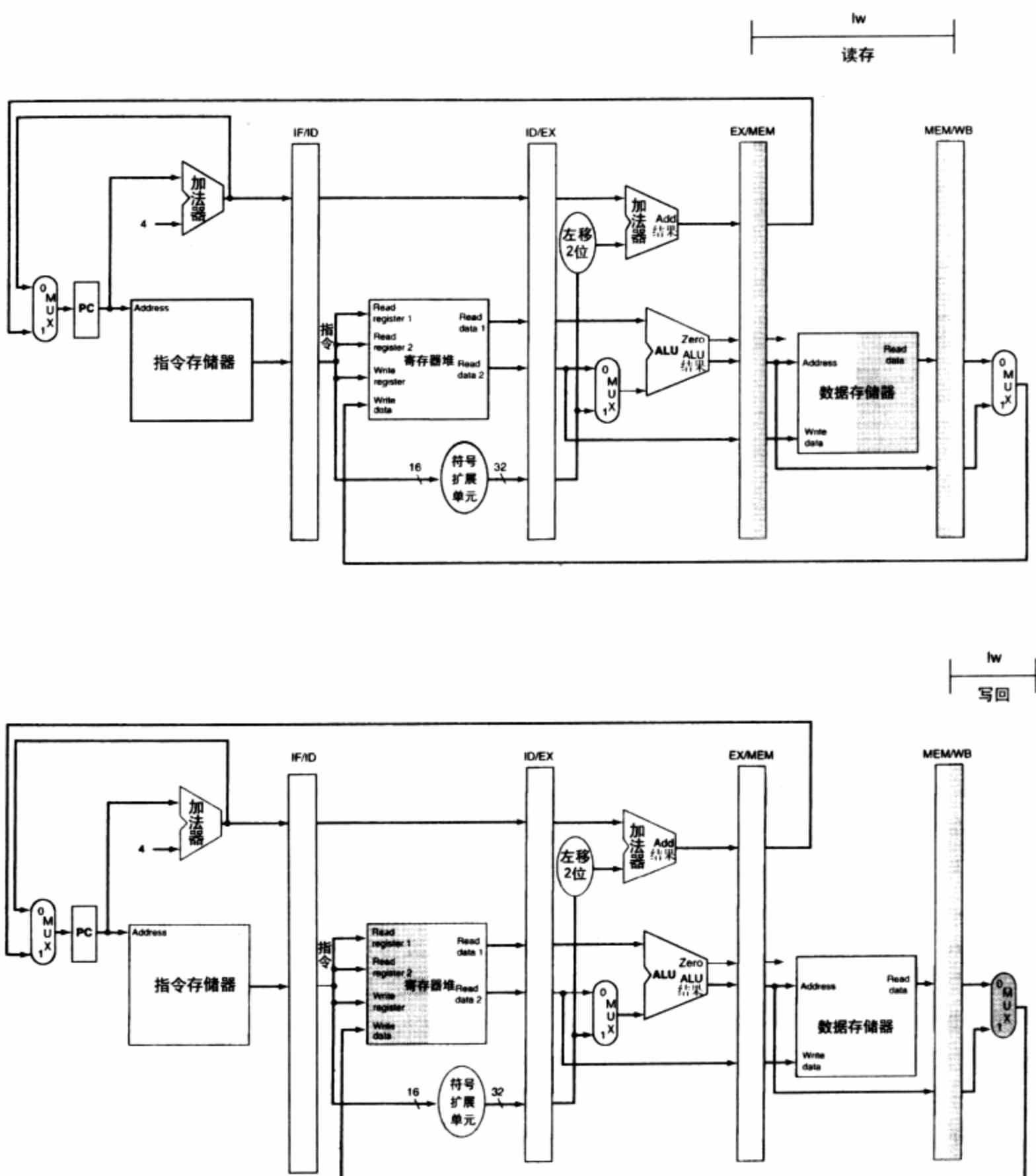


图 6-14 MEM 与 WB: load 指令在流水线中的第四与第五步，并且用阴影表示出图 6-11 中的数据通路在这两个流水线级中的部分

[利用 EX/MEM 流水线寄存器中的地址读取数据内存，并将读取的数据放入到 MEM/WB 流水线寄存器中。然后从 MEM/WB 流水线寄存器中读取数据并将数据写入数据通路中间的寄存器堆中]

4) 内存访问：图 6-16 的上图描述的是数据写入内存的过程。需要注意的是，需要存入寄存器中的数据在较早的流水线步骤中已经读出并存放在 ID/EX 中。在 MEM 阶段惟一一种获得这个数据的方法就是把数据放入 EX 步骤中的 EX/MEM 流水线寄存器中，这一过程与将有效地址放入 EX/MEM 中类似。

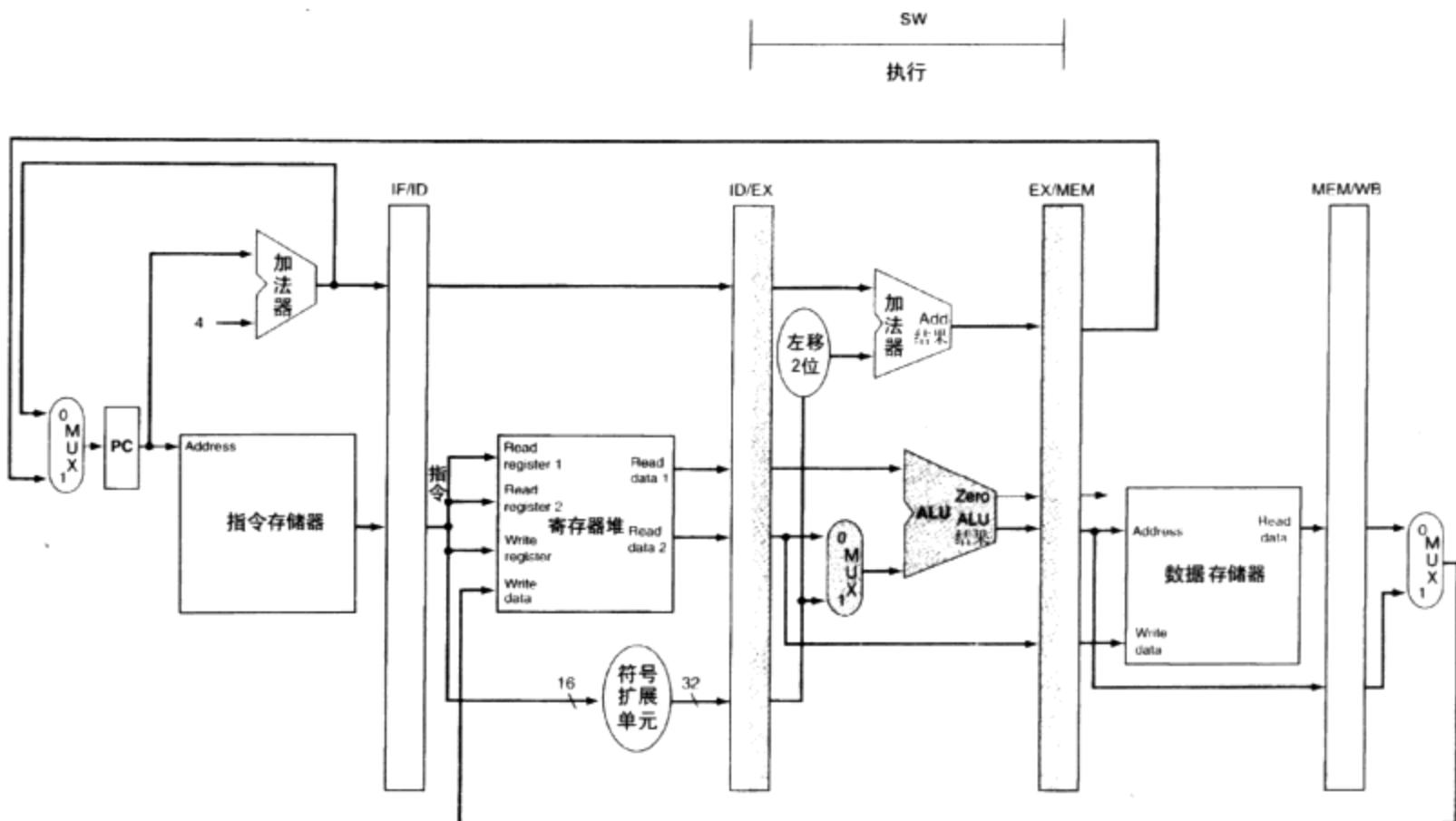


图 6-15 EX: store 指令在流水线中的第三步

[与图 6-13 中 load 指令的第三个流水线步不同的是，第二个寄存器中的数据被载入 EX/MEM 流水线寄存器中，并被用于下一个步骤。虽然总是将第二个寄存器中的数据载入 EX/MEM 流水线寄存器中并不会产生什么不良的影响，但为了使流水线更易于理解，我们仍然只在 store 指令中才写第二个寄存器的内容]

5) 写回：图 6-16 中的下图描述了 store 指令的最后一个流水线步骤。store 指令在写回步骤中不做任何事情。由于 store 指令后的每一条指令都已经进入流水线中，所以我们无法加速这些指令。因此，任何一条指令都必须经过流水线中的每一个步骤，即使在这个步骤中它实际上什么都没有做，这是因为后面的指令已经按照最大的速率在流水线中进行处理。

store 指令再次说明在流水线中为了从上一级向下一级传递信息，必须将信息放入流水线寄存器中；否则当下一条指令进入流水线的本级时这些信息将会丢失。在 store 指令中，我们需要将一个寄存器中的内容在 ID 步骤中读出然后在 MEM 步骤放入内存中。这些数据首先放在 ID/EX 流水线寄存器然后被传送到 EX/MEM 流水线寄存器中。

load 与 store 的执行过程还表明了另一个重要特性：数据通路中的每一个逻辑元件，如指令内存、寄存器读取端口、ALU、数据内存以及寄存器写入端口，都只能在流水线的单级中使用，否则就会产生结构冒险(请查阅 6.1.2 节)。因此，这些元件以及它们的控制单元同时只能与流水线的单级相关联。

现在我们可以揭示 load 指令设计中的一个 bug 了。你发现了这个 bug 吗？在 load 指令执行的最后一个阶段哪一个寄存器内容改变了呢？更确切地说，哪一条指令提供了写入寄存器编号呢？在 IF/ID 流水线寄存器中的指令提供了写入寄存器编号，但是很显然这条指令已是本条 load 指令之后的某条指令。

因此，我们要在 load 指令中保存目的寄存器号。就像 store 指令为了 MEM 步骤使用的需要将寄存器的内容从 ID/EX 转送到 EX/MEM 流水线寄存器中去一样，为了 WB 步骤中的使用的需要，load 指令必须把寄存器号从 ID/EX 经过 EX/MEM 传送到 MEM/WB 流水线寄存器中。换一个角度来考虑传输寄存器号的必要性：为了共享流水线的数据通路，我们需要在 IF 步时保存读取的指令，因此每一个流水线寄存器都要保存当前步和以下流水线各步所需的部分指令信息。

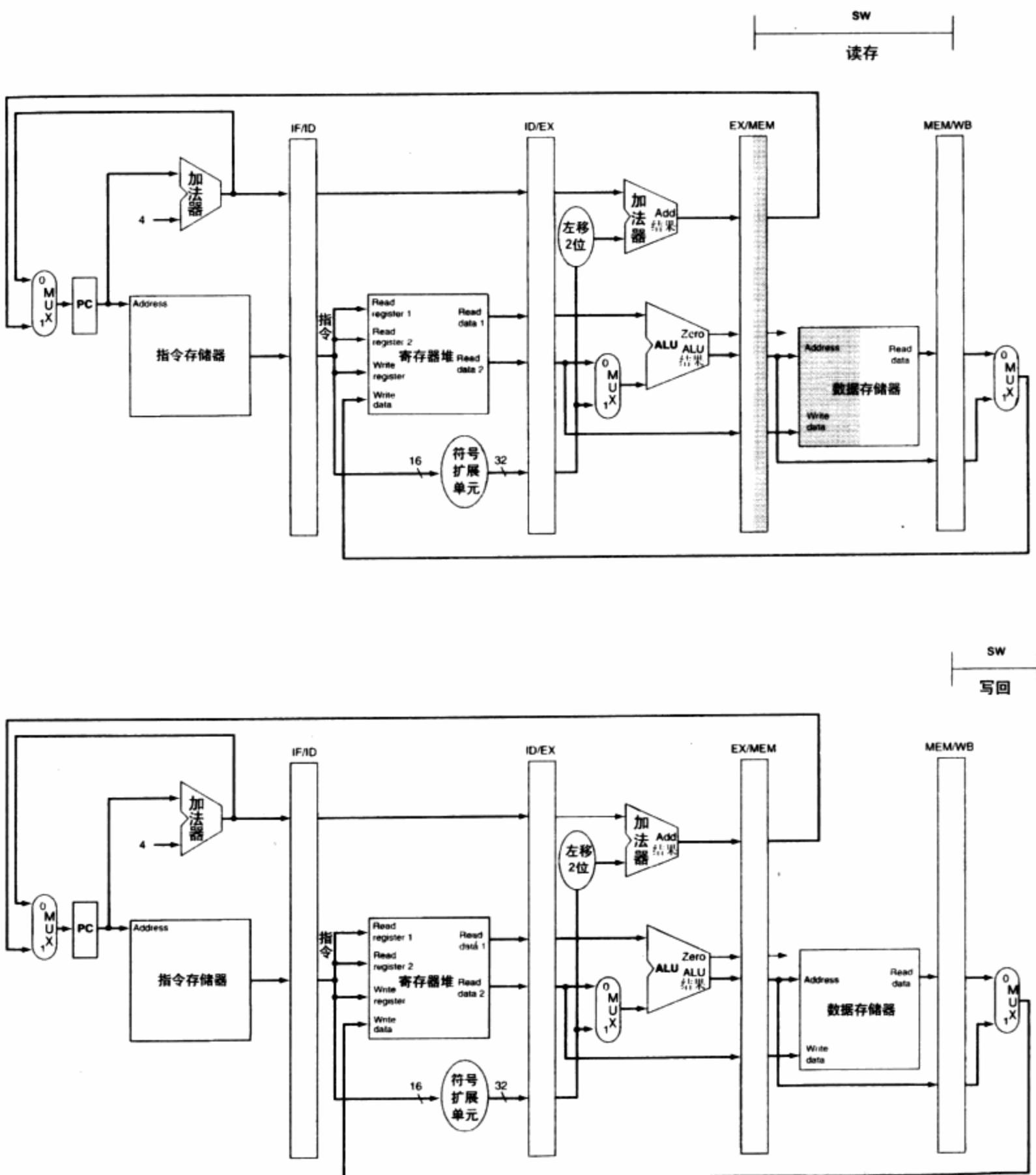


图 6-16 MEM 和 WB: store 指令在流水线中的第四和第五步

[在第四步中，为了存储数据将数据写入数据内存中。需要注意的是数据来自于 EX/MEM 流水线寄存器，并且 MEM/WB 流水线寄存器并没有改变。一旦数据写入内存，store 指令就没有什么可做的了，所以在步骤 5 中 store 指令并不做任何处理]

图 6-17 描述了正确的数据通路，首先将写入寄存器编号传送到 ID/EX 寄存器，然后再送到 EX/MEM 寄存器，最后送到 MEM/WB 寄存器。在 WB 步使用寄存器号以确定要写入哪一个寄存器。图 6-18 是一个正确的简单的数据通路图，图中用阴影标出了从图 6-12~图 6-14 的 load 字指令在所有五个流水线步骤中要使用的硬件。请参阅 6.6 节相关内容以了解如何使分支指令按期望的方式工作。

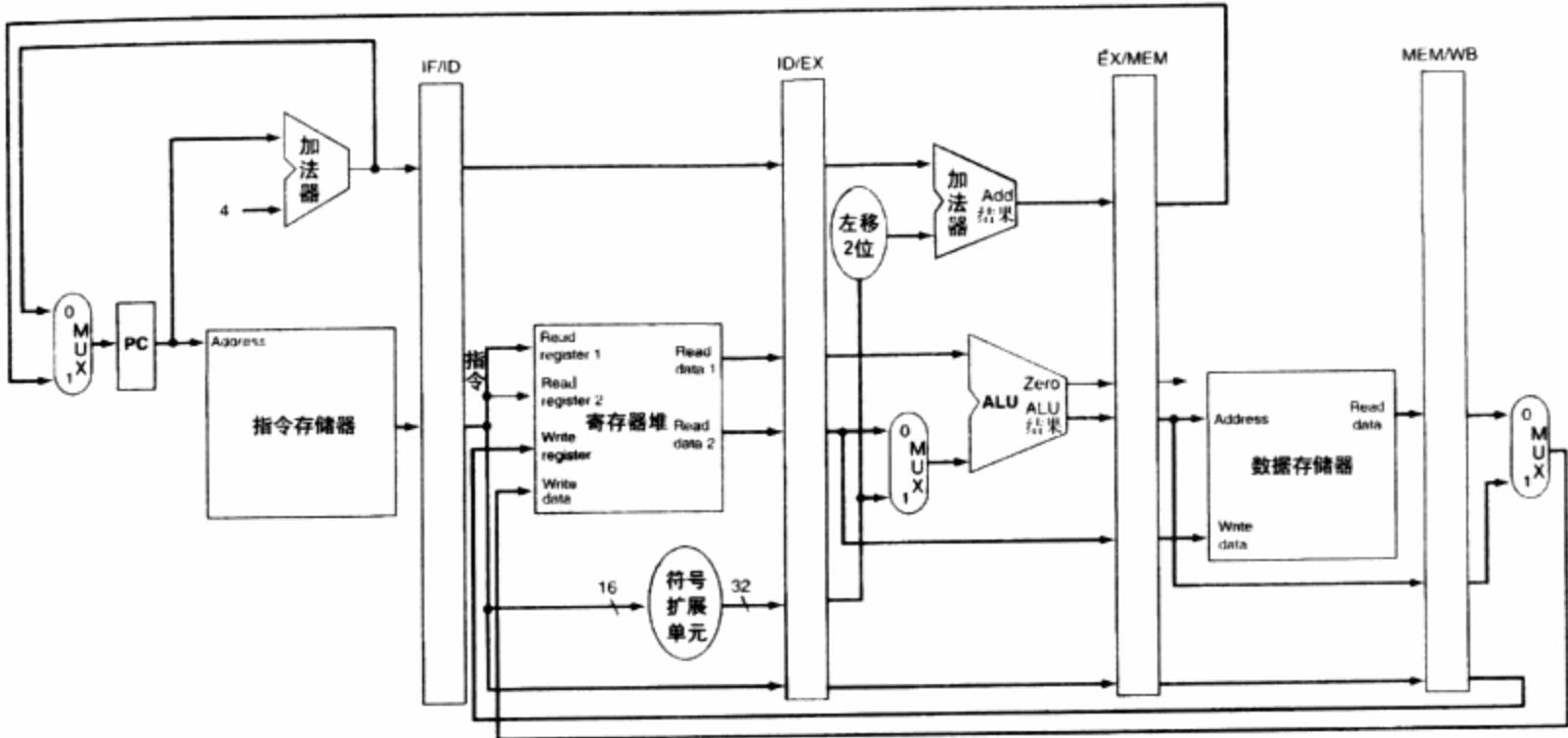


图 6-17 正确处理 load 指令的流水线数据通路

[写寄存器号与寄存器数据一起从 MEM/WB 流水线寄存器中得到。通过在最后的三个流水线步上分别增加了 5 位，寄存器号就能从 ID 流水线步一直传送到 MEM/WB 流水线步。这条新的通路在图中用粗线标出]

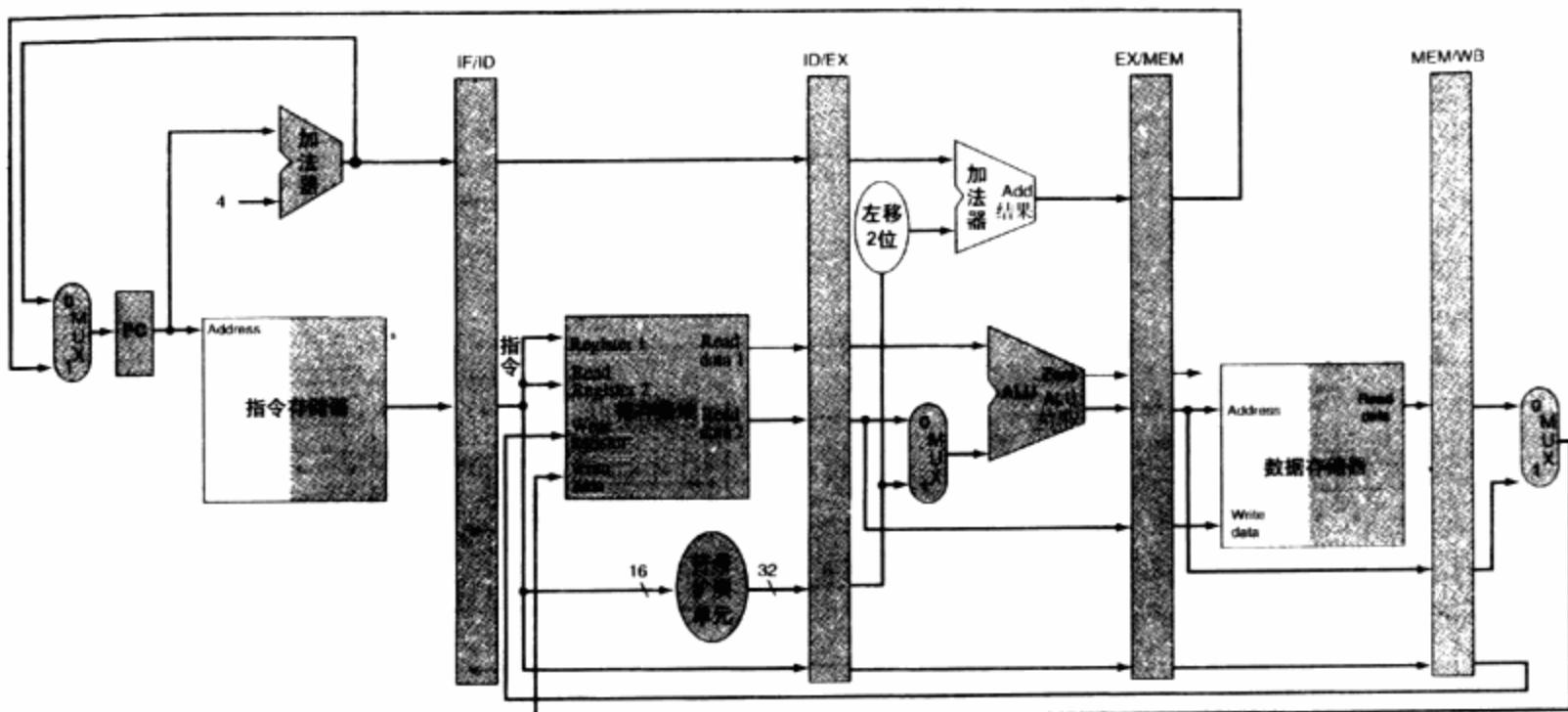


图 6-18 图 6-17 中在 load 指令所有的五个流水线步中都用到的数据通路部分

用图形的方式表示流水线

流水线技术比较难以理解，因为在每一个时钟周期内同时会有很多指令在一个数据通路中执行。为了辅助我们理解流水线的原理，有两种基本的表示流水线的图形化方法，即多时钟周期的流水线图(如图 6-10 所示)和单时钟周期的流水线图(如图 6-12~图 6-16 所示)。多周期流水线图虽然简单但是忽略了一些细节。我们以下面五条指令构成的指令序列为为例对其作一说明。

```

lw    $10, 20($1)
sub   $11, $2, $3
add   $12, $3, $4
lw    $13, 24($1)
add   $14, $5, $6

```

图 6-19 所示为这些指令对应的多时钟周期流水线图。与图 6-1 中洗衣店流水线表示方法类似，从左到右时间递增，从上到下指令递增。沿着指令轴方向我们可以观察某一时钟周期流水线的各级，它们分别为各条指令所占据。这些格式化的数据通路分别表示流水线中的五级，我们同样也可以用一个小方块写上流水线的名字来清晰地表示出流水线。图 6-20 所示为一个更为传统的多时钟周期流水线图的表示方法。需要提醒我们注意的是图 6-19 中描述的是每步所使用的实际资源，而图 6-20 描述的则是各步的名字。我们用多时钟周期流水线图描述流水线的整体情况。

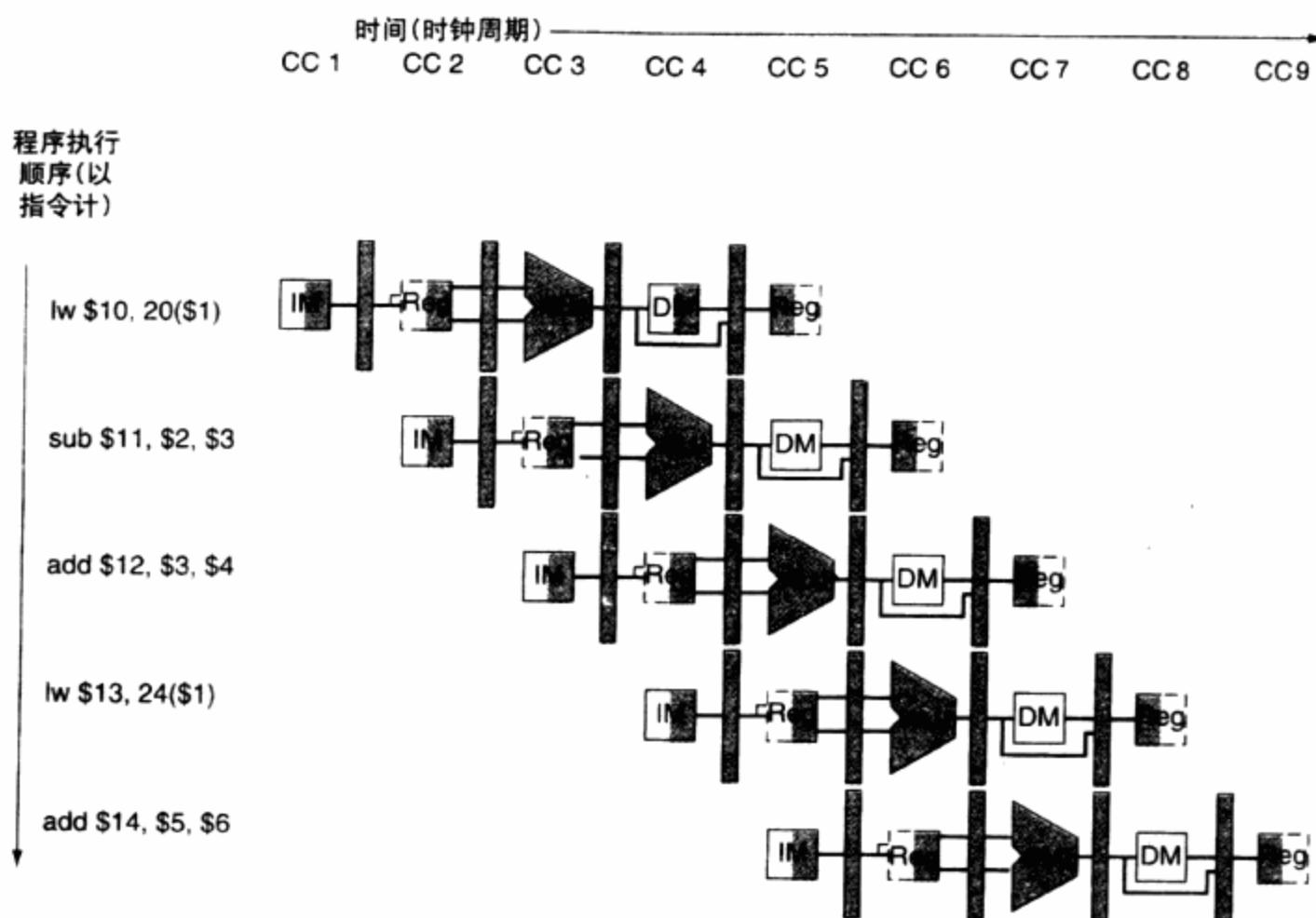


图 6-19 五条指令序列对应的多时钟周期流水线图

[这种表示流水线的方式在同一张图中画出了指令完整的执行过程。指令按照执行次序从上向下依次排列，始终周期则由左到右来表示。与图 6-4 不同，我们在此图中画出了流水线各级之间的流水线寄存器。图 6-20 是传统表现方式下的流水线图]

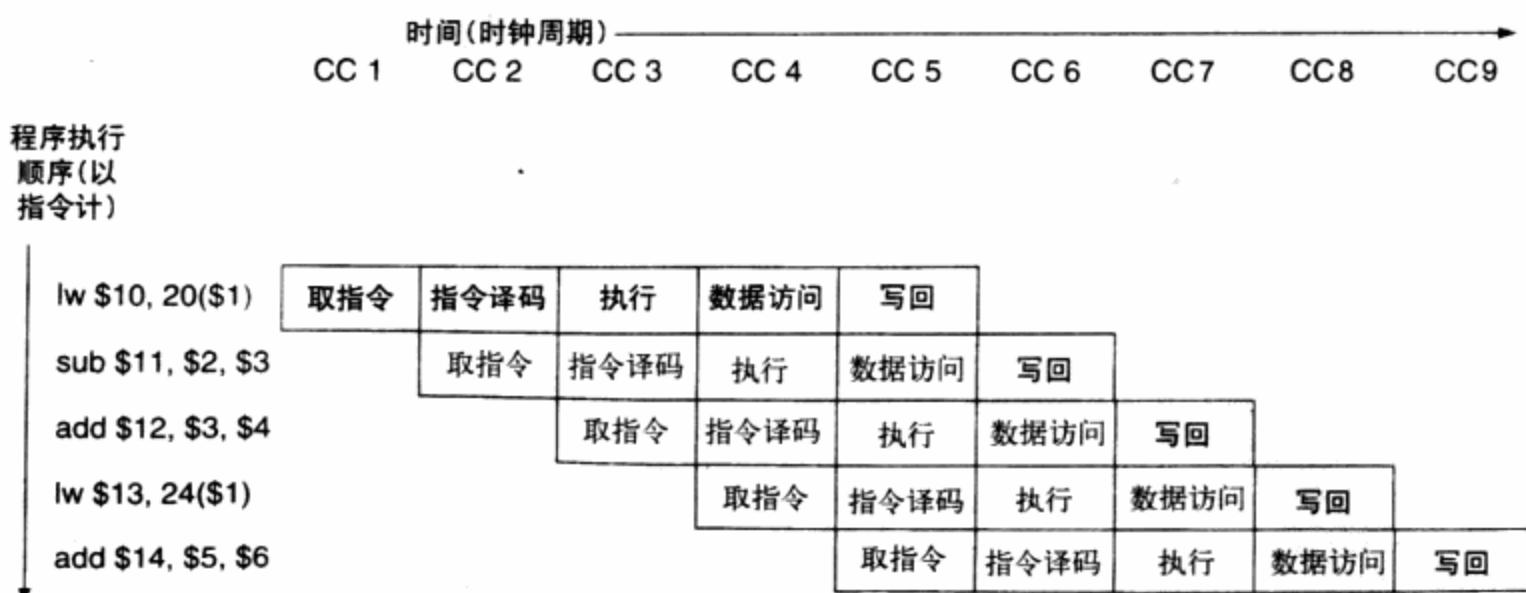


图 6-20 图 6-19 中传统表示方式下该指令序列对应的多周期流水线图

单时钟周期流水线图表示的是在同一个时钟周期内整个数据通路的状态。而所有五个在流水线中的指令都在流水线各级上作相应的标示。这种流水线表示图描述了在每一个时钟周期内流水线中所发生的细节事件。我们通常使用一组单时钟周期流水线图来表示在一系列时钟周期内流水线的操作。单时钟周期图通过一套多时钟周期图来表达垂直切片，表明了在指定的时钟周期内流水线中每条指令对数据通路的使用。比如，图 6-21 概括了图 6-19~图 6-20 所示的第五个时钟周期的单周期流水线图。显然，单周期流水线图展现了更多的细节，但其所占的篇幅也相当大。本书光盘中对应的“For More Practice”小节里面有这两条指令对应的单周期流水线图，同时它还包含一些练习，要求你为其他指令序列列出其对应的流水线图。

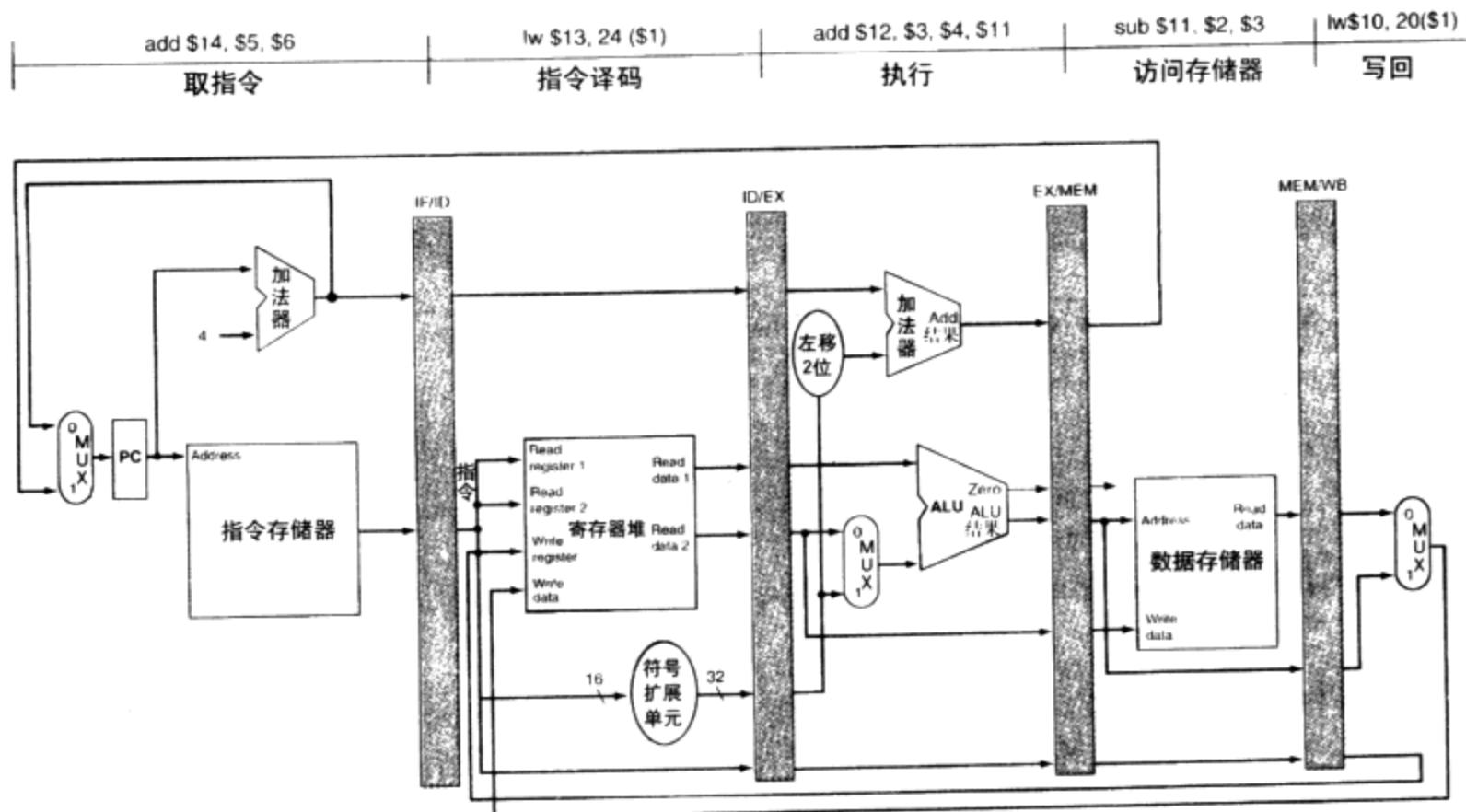


图 6-21 图 6-19 和图 6-20 中第五个时钟周期对应的单周期流水线图

[我们可以观察到，单周期图对应的是多周期图从竖直方向上分割得到的那一部分]

自测

曾经有一个学生指出并不是所有的指令在流水线的各级中都处于活动状态，因此一组学生对五级流水线的效率问题展开了讨论。在不考虑冒险的前提下，他们提出了下面 5 个论点，哪些是正确的？

- 1) 在任何情况下，允许跳转指令、分支指令和 ALU 指令用更少的级数完成，而不是 load 指令的 5 级，这样做总可以提高流水线的性能。
- 2) 因为流水线的吞吐量是由时钟周期决定的，所以试图让一部分指令在较少的周期内完成不会有什么效果；每条指令所占的流水线的级数会影响时延，而非吞吐量。
- 3) 允许跳转指令、分支指令和 ALU 操作在较少周期内完成只会在没有 load 和 store 指令存在于流水线中的前提下有用，所以这样做收效甚微。
- 4) 你不可能让 ALU 指令在较少的时钟周期内完成，因为它们必须写回结果，但是分支和跳转指令可以在较少的周期内完成，所以对此仍有一些改进的空间。
- 5) 我们应当尝试将流水线的级数增多，而非试图减少执行指令所需要的周期数，这样虽然指令需要更多个周期才能完成，但是时钟周期变短了。这可能会带来一些性能上的改进。

6.3 流水线中的控制

计算机之间的不同在于控制系统的差别，在 6600 型计算机中尤其是这样。

James Thornton, *Design of a Computer: The Control Data 6600*, 1970

5.4 节介绍了在简单的数据通路中如何加入控制电路，本节将介绍如何针对基于流水线的数据通路完成同样的工作。首先我们在诸多限制的条件下通过一个简单的设计方案了解流水控制，然后在 6.4~6.8 节中逐步去掉这些限制进一步展示实际情况中的冒险。

我们首先要做的工作就是标出已有的数据通路上的控制线(如图 6-22 所示)。我们尽量借用图 5-17 中简单数据通路中控制电路的实现方法。具体来讲：我们将使用相同的 ALU 控制逻辑、分支逻辑、目的寄存器号多路复用器和控制线。在图 5-12、图 5-16 以及图 5-18 中已经给出了这些功能部件的定义。为了使本节内容更易于理解，图 6-23~图 6-25 重新阐释了这些关键信息。

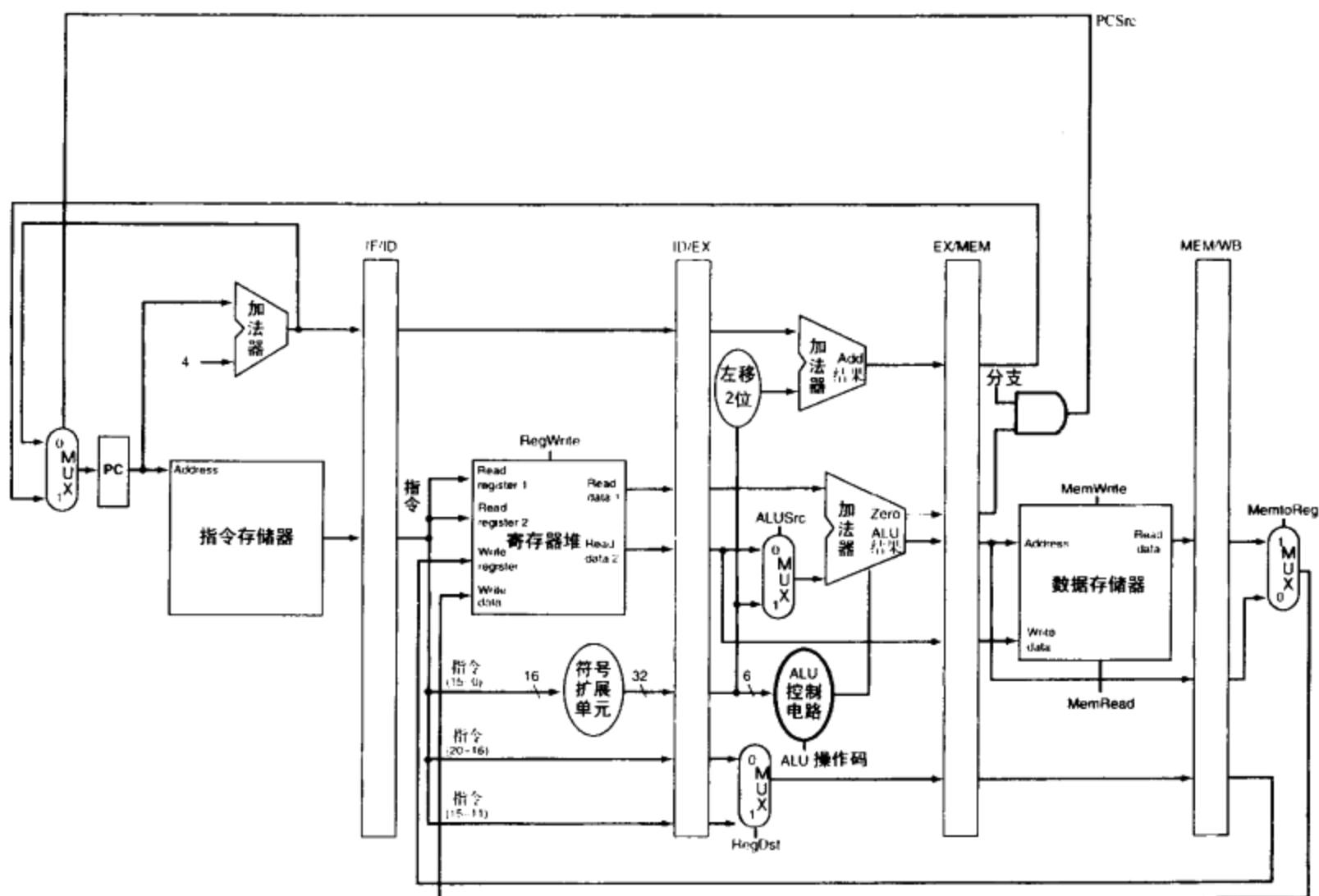


图 6-22 标明控制信号的流水线数据通路(见图 6-17)

[这个数据通路采用了与第 5 章中相同的 PC 源控制逻辑、寄存器目标号和 ALU 控制。需要注意的是，这时在 EX 流水线步骤中指令需要一个六位的功能字段(即功能码)作为 ALU 控制的输入，因此这六位数也必须存放 在 ID/EX 流水线寄存器中。这六位数是指令中的立即数字段的低六位有效位，由于在对立即数进行符号化扩 展时这六位数并没有发生变化，因此 ID/EX 流水线寄存器可以在立即数字段中获得这六位数]

与第 5 章中讨论的单时钟周期实现方法一样，我们假定在每一个时钟周期内都会写 PC，因此就不需要单独的 PC 写信号。同样的道理，流水线寄存器(IF/ID、ID/EX、EX/MEM 和 MEM/WB)也不需要单独的写信号，因为在每个周期它们都会被写入一次。

指令操作码	ALU 操作码	指令功能	功能码	ALU 动作	ALU 控制信号
LW	00	加载一个字	XXXXXX	加法	0010
SW	00	存储一个字	XXXXXX	加法	0010
相等时跳转	01	相等时跳转	XXXXXX	减法	0110
R型	10	加法	100000	加法	0010
R型	10	减法	100010	减法	0110
R型	10	与操作	100100	与操作	0000
R型	10	或操作	100101	或操作	0001
R型	10	小于时置位	101010	小于时置位	0111

图 6-23 图 5-12 的复本

[本图描述了如何根据 ALU 操作码控制位和不同的 R 型指令的功能码设置 ALU 控制位的值]

信号名称	设为 0 时的效果	设为 1 时的效果
RegDst	写入寄存器的目标编号来自 rt 字段(20:16 位)	写入寄存器的目标号来自 rd 字段(15:11 位)
RegWrite	无	将写入数据的输入写入至写入寄存器输入对应的寄存器
ALUSrc	第二个 ALU 操作数来自第二个寄存器堆的输出(读出数据 2)	第二个 ALU 操作数为已符号化扩展的指令的低 16 位
PCSrc	PC 的值替换为计算 PC + 4 的加法器的输出	PC 的值置为计算分支目标地址的加法器的输出
MemRead	无	输入地址对应的数据内存的内容放置到读出数据的输出
MemWrite	无	输入地址对应的数据内存的内容替换为写入数据的输入
MemtoReg	ALU 提供寄存器写数据的输入值	数据内存提供寄存器写数据的输入值

图 6-24 图 5-16 的复本

[图中定义了七个控制信号的功能。ALU 控制线(ALUOp)已经在图 6-23 中的第二列中定义。当一个二路复用器的一个控制位置为 1 时，复用器选择与 1 对应的输入，否则，如果这个控制位置为 0 时，那么复用器就选择与 0 对应的输入。需要提醒注意的是 PCSrc 是由图 6-22 中的一个与门控制。如果分支信号与 ALU 为零信号线都置 1，则 PCSrc = 1；否则它的值将为 0。控制只在 beq 指令中才置分支信号为 1；其他任何时候 PCSrc 都将设置为 0]

指令	执行/地址计算级控制信号				访存级控制信号			写回级控制信号	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R型	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

图 6-25 控制线的值与图 5-18 中的相同，但它们根据流水线的最后三个阶段分成了三组

为了说明对流水线的控制，我们所要做的只是在每一个流水线步骤中都相应设定控制信号的值。由于每一条控制线与流水线中单级内的某个活动的器件相关联，因此根据流水线的五个步骤我们可以将控制线分为五组：

1) 取指令：读取指令内存和写 PC 的控制信号总是确定的，因此在取指令阶段没有什么特别需要控制的东西。

2) 指令译码/读取寄存器堆：与上一步类似，在每个时钟周期内这个步骤所完成的工作都是相同的，因此不需要设置控制线。

3) 指令执行/地址计算：被设置的信号有 RegDst、ALUOp 和 ALUSrc(参见图 6-23 和图 6-24)。根据这些信号选择结果寄存器，确定 ALU 的操作，同时读取数据 2 或经过符号扩展后得到的立即数。

4) 内存访问：这一步设置的控制线有 Branch、MemRead 和 MemWrite，这些控制信号分别由相等分支、取和存指令设定。除非控制电路指示这是一条分支指令同时 ALU 输出为 0，将选择线性地址中的下一条指令作为 PCSrc 信号。

5) 写回：两条控制线分别是 MemtoReg 和 RegWrite，其中前者决定是将 ALU 结果还是将内存数据传送到寄存器堆，后者记录所要写入的数据。

由于流水线方式的数据通路不改变控制线的意义，因此我们可以使用与以前相同的控制值。图 6-25 就与第 5 章具有相同的控制值，与之不同的是现在 9 条控制线按流水线步骤进行了分组。

实现控制就是为每一条指令的每一个步骤中的 9 条控制线设置成相应的值。其最简单的实现方法就是扩展流水线寄存器使之包含这些控制信息。

由于控制线从 EX 步骤开始，因此可以在指令译码阶段创建控制信息。图 6-26 描述了当指令在流水线中传递时这些控制信号的相应使用方法，这一点与图 6-17 中执行 load 指令时目标寄存器号在流水线中的传递过程类似。图 6-27 描述了带有扩展流水线寄存器并且将控制线连接到相应的流水线步骤的完整的数据通路。

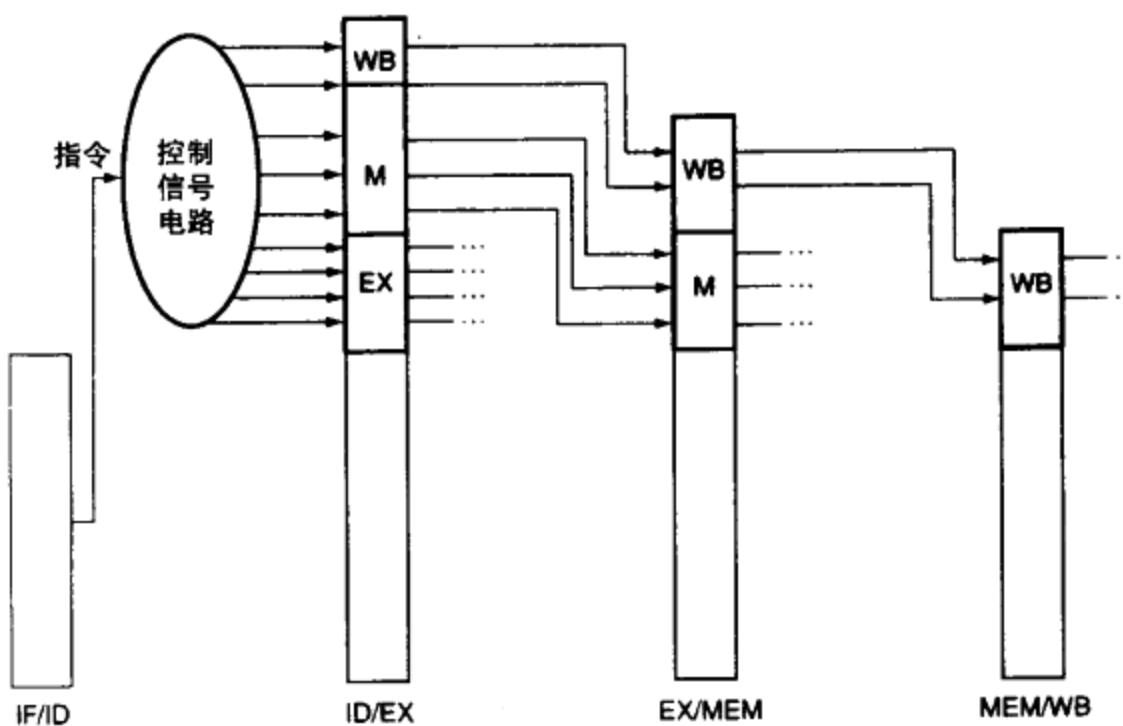


图 6-26 流水线最后三级的控制线

[需要提醒注意的是 9 条控制线中有 4 条用于 EX 阶段，而剩下的 5 条控制线被传递到扩展的保存控制线的 EX/MEM 流水线寄存器中；传递来的 5 条控制线中有 3 条用于 MEM 阶段，剩下的 2 条传递到 MEM/WB 并用于 WB 阶段]

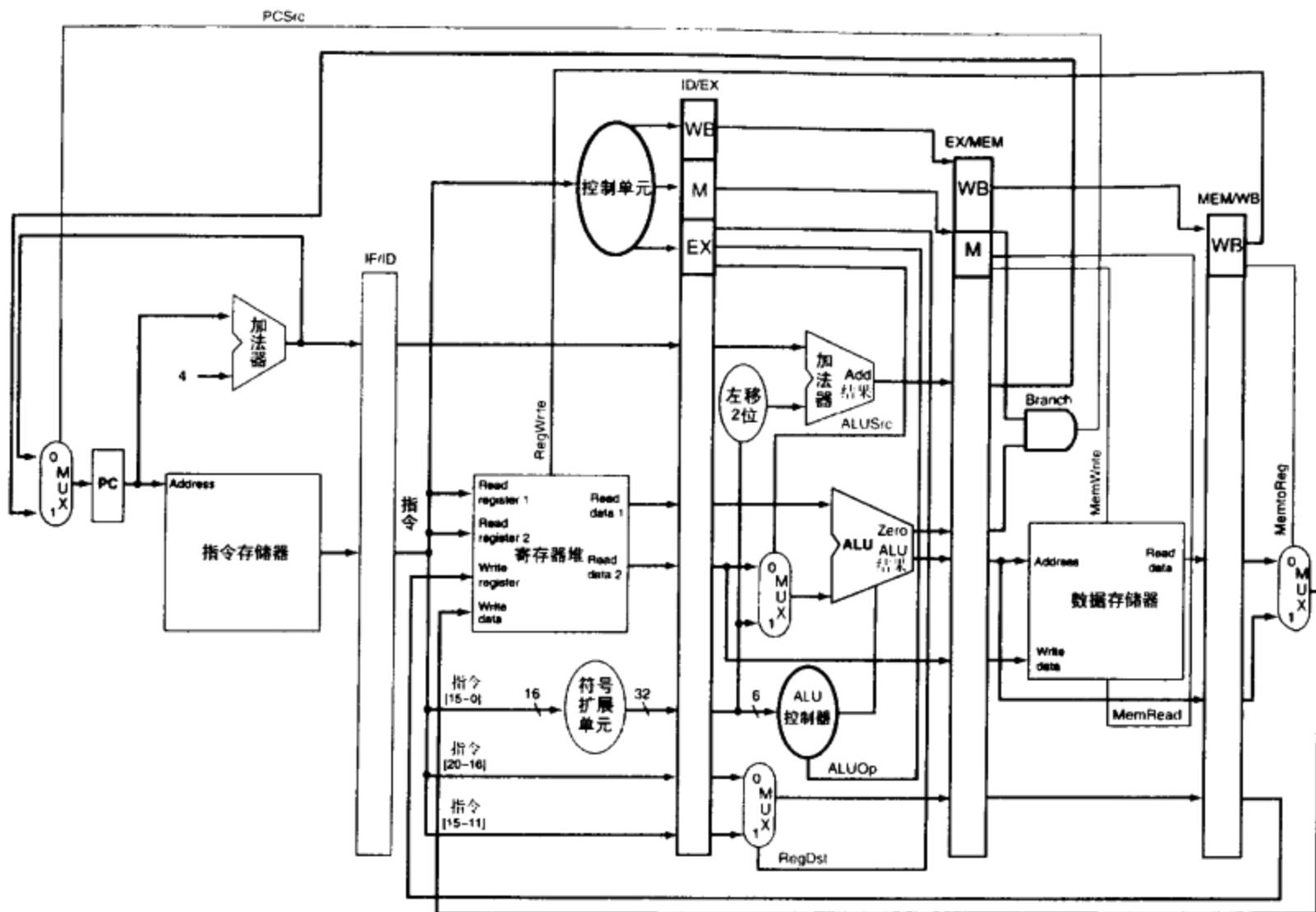


图 6-27 图 6-22 中的流水线数据通路、本图中已将控制信号连接到流水线寄存器的控制部分

[流水线最后三级的控制信号在指令的译码阶段生成，然后放入 ID/EX 流水线寄存器。流水线的每级都使用相应的控制线，并将剩余的控制线传递到流水线的下一级]

6.4 数据冒险与转发

你是什么意思？为什么一定要建？这是一个旁路，你总是会需要旁路的。

Douglas Adams, *Hitchhikers Guide to the Galaxy*, 1979

上一节的例子介绍了流水线的强大功能以及硬件如何通过流水线的方式执行任务。本节我们避开这些光环看看在流水线在实际程序中的情况。图 6-19~图 6-21 中的各指令之间是相互独立的；其中任何一条指令都没有用到任何其他指令的计算结果。然而，在 6.1 节中我们就已经发现数据冒险是影响流水线执行的一个主要障碍。

让我们看一看下面这个存在着许多相关性的指令序列，依赖关系都以粗体标出：

```

sub $2, $1,$3      # Register $2 written by sub
and $12,$2,$5      # 1st operand($2) depends on sub
or  $13,$6,$2       # 2nd operand($2) depends on sub
add $14,$2,$2       # 1st($2) & 2nd($2) depend on sub
sw  $15,100($2)     # Base ($2) depends on sub
    
```

后四条指令都依赖于第一条指令得到的寄存器 \$2 的结果。如果寄存器 \$2 在 sub 指令执行之前的值为 10 而在 sub 指令执行之后的值为 -20，程序员的意图是后四条指令访问到的寄存器 \$2 的值为 -20。

这个指令序列在流水线中是如何执行的呢？图 6-28 用多时钟周期流水线图表示了这几条指

令的执行过程。为了在当前流水线中表示这个指令序列的执行过程，图 6-28 的上部给出了寄存器 \$2 中在每个时钟周期开始时的值，它的值在第五个周期内改变，这是由于 sub 指令写回了它的执行结果。

一类潜在的冒险可以通过设计相应的寄存器堆的硬件解决：但如果一个寄存器在同一时钟周期内同时读和写时会发生什么呢？这时我们假设写寄存器操作发生在时钟周期的前半段而读寄存器操作发生在时钟周期的后半段，因此读操作将读取到最新写入的内容。大多数寄存器堆的实现方法与我们的假设一致的，而且在这种假设条件下不会发生数据冒险。

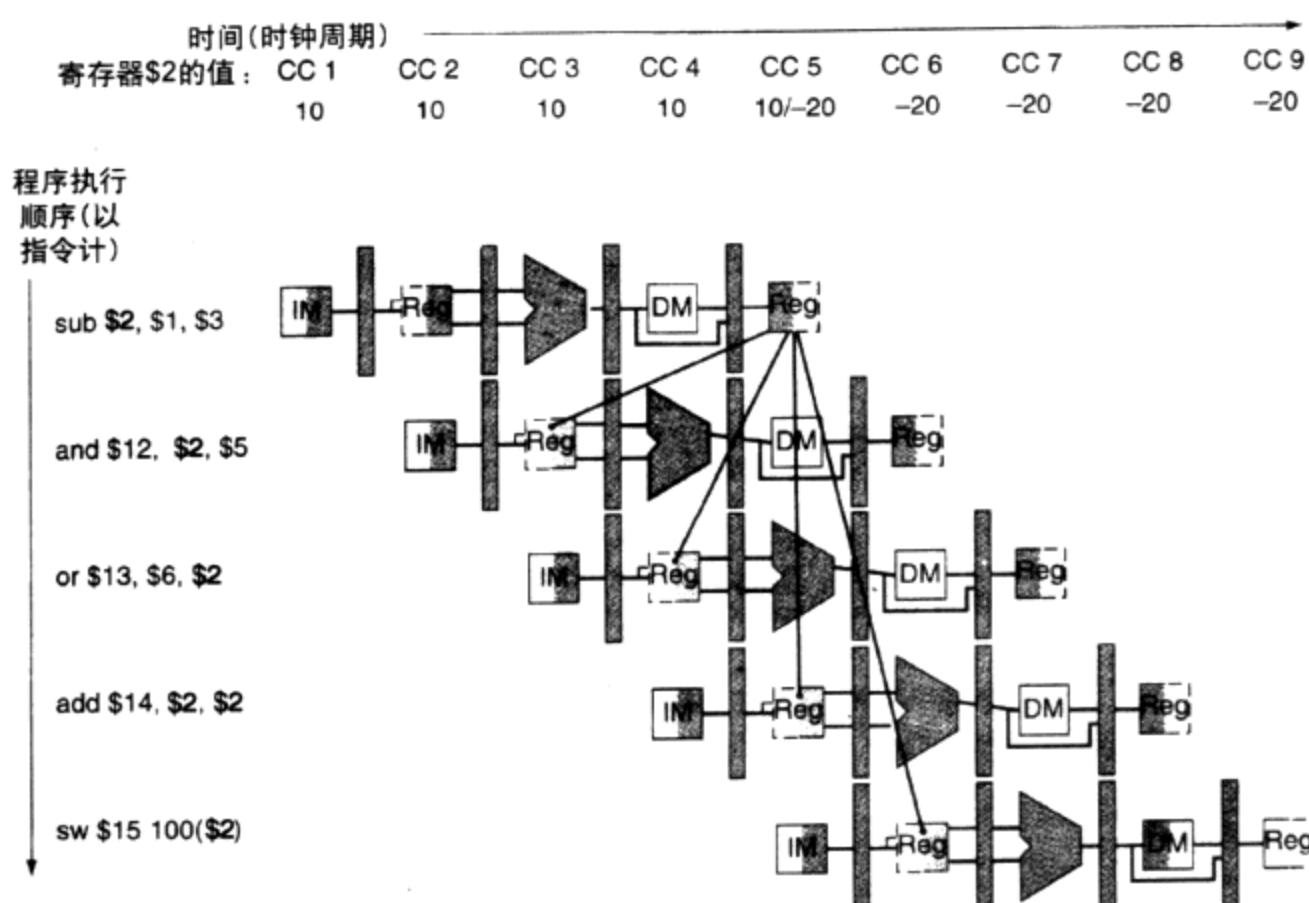


图 6-28 使用简化的数据通路流水线图表示具有相关性的五条指令序列中的相关性

[所有的依赖行为在图中都标出来了，图顶部的“CC *i*”表示第 *i* 个时钟周期。指令序列第一条指令写寄存器 \$2，后四条指令读寄存器 \$2。寄存器 \$2 在第 5 个时钟周期被写入，所以在此之前它的值都是无效的。(寄存器堆必须读取这一时钟周期前半段写入的寄存器的值。)图中上部的数据通路中到下部的数据通路中的线表示的是数据依赖。那些导致时间后退的依赖就是流水线数据冒险]

图 6-28 表明如果不是在第 5 个时钟周期或其后再读寄存器 \$2，读操作得到的寄存器值就不会是 sub 指令的结果。因此，指令 add 和 sw 将得到正确结果 -20；而指令 and 和 or 将得到错误结果 10！使用这种流水线图，我们可以说当一条依赖关系的方向与时间轴方向相反就会发生数据冒险。

但是观察图 6-28：sub 指令的结果到底是什么时候产生的？该结果于 sub 指令到达 EX 段结尾、也就是第三个周期末被产生。该结果又是在什么时候被 and 和 or 指令请求的呢？是在这两条指令进入 EX 段的时候，分别对应第四个和第五个周期。所以，只要在这个结果产生之后，直接将它转发给(forward)需要它的指令，这样这些指令就可以执行了，而不需要先写回寄存器堆而后再将它读取出来。

如何将新产生的数据直接通过旁路传递给其他指令呢？为了简化讨论，本节后续部分只考虑如何直接传送 EX 段产生的数据，该数据可能是 ALU 运算的结果，也有可能是地址计算的结果。这意味着当一条指令在其 EX 段请求某个数据，这个数据恰好是前面某条指令在 WB 段要写回的数值时，实际上只是将该数据作为 ALU 的输入。

通过给流水线寄存器字段显式地命名，我们可以有另一种更为准确的标定依赖关系的方法。比如，“ID/EX.RegisterRs”可以用来标明 ID/EX 段的流水线寄存器中的数据；也就是寄存器堆第一个读端口输出的数据。这个名称的前半部分，也就是点号左面的部分，表示的是流水线寄存器的名称；后半部分是该流水线寄存器中对应域的名称。使用这种标定方法，代码中的两个冒险关系可以表示为：

- 1a. EX/MEM. RegisterRd = ID/EX. RegisterRs
- 1b. EX/MEM. RegisterRd = ID/EX. RegisterRt
- 2a. MEM/WB. RegisterRd = ID/EX. RegisterRs
- 2b. MEM/WB. RegisterRd = ID/EX. RegisterRt

第一个冒险关系是 6.4 节第 1 个代码片断中 `sub $2, $1, $3` 指令执行结果和指令 `and $12, $2, $5` 的第一个源操作数之间关于寄存器 `$2` 的依赖关系。我们可以在 `and` 指令在 EX 段而前一条指令在 MEM 阶段时检测出这个依赖关系，所以冒险关系 1a 表示为：

$$\text{EX/MEM. RegisterRd} = \text{ID/EX. RegisterRs} = \$2$$

例题 相关性检测

将 6.4 节第 1 个指令序列中的依赖进行分类：

```
sub    $2,    $1, $3  # Register $2 set by sub
and    $12,   $2, $5  # 1st operand($2) set by sub
or     $13,   $6, $2  # 2nd operand($2) set by sub
add    $14,   $2, $2  # 1st($2) & 2nd($2) set by sub
sw     $15, 100($2) # Index($2) set by sub
```

解 如上所述，`sub-and` 是一个 1a 类冒险。其余的冒险分别是：

■ `sub-or` 是一种 2b 类冒险：

$$\text{MEM/WB. RegisterRd} = \text{ID/EX. RegisterRt} = \$2$$

- `sub-add` 上的两个依赖都不是冒险，因为在 `add` 的 ID 阶段寄存器堆能够提供相应的数据。
- `sub` 指令和 `sw` 指令之间也不存在数据冒险，因为 `sw` 指令在 `sub` 指令写寄存器 `$2` 之后读取 `$2`。

由于一些指令不写寄存器，所以这种解决冒险的方法是不正确的；因为它在一些不必要的时候也会转发。一种简单的解决方法是检测 `RegWrite` 信号是否是活动的：即通过检测流水线寄存器在 EX 和 MEM 阶段的 WB 控制域以确定 `RegWrite` 是否被设置。而且，MIPS 要求在每次使用 `$0` 作为操作数时都必须提供一个值为 0 的操作数。在一条流水线中的指令的目标寄存器是 `$0` 的情况下（如 `sll $0, $1, 2`），就要避免转发可能的非零结果值。不转发将 `$0` 作为目标寄存器的结果使得汇编程序员和编译器不用考虑 `$0` 作为目标寄存器的情况。因此，只要将 `EX/MEM. RegisterRd ≠ 0` 加入第一种冒险条件，将 `MEM/WB. RegisterRd ≠ 0` 加入第二种冒险条件，上面给定的冒险控制策略就能正确工作。

前面介绍了检测冒险的方法，问题已经解决了一半，我们仍需要转发正确的数据的策略。

图 6-29 描述了图 6-28 的指令序列中流水线寄存器和 ALU 的输入之间的相关性。与图 6-28 不同的是，这里依赖开始于一个流水线寄存器而不是等待 WB 阶段写寄存器堆。由于流水线寄存器保存了需要转发的数据，因此后面的指令能够获得相应的数据。

如果可以从任何流水线寄存器中，而不仅仅从 ID/EX 中得到 ALU 的输入，那么我们就一定能正确转发相应的数据。通过在 ALU 的输入中加入多路复用器和正确的控制策略，我们就可以

得到存在数据相关的情况下仍然能全速运行的流水线。

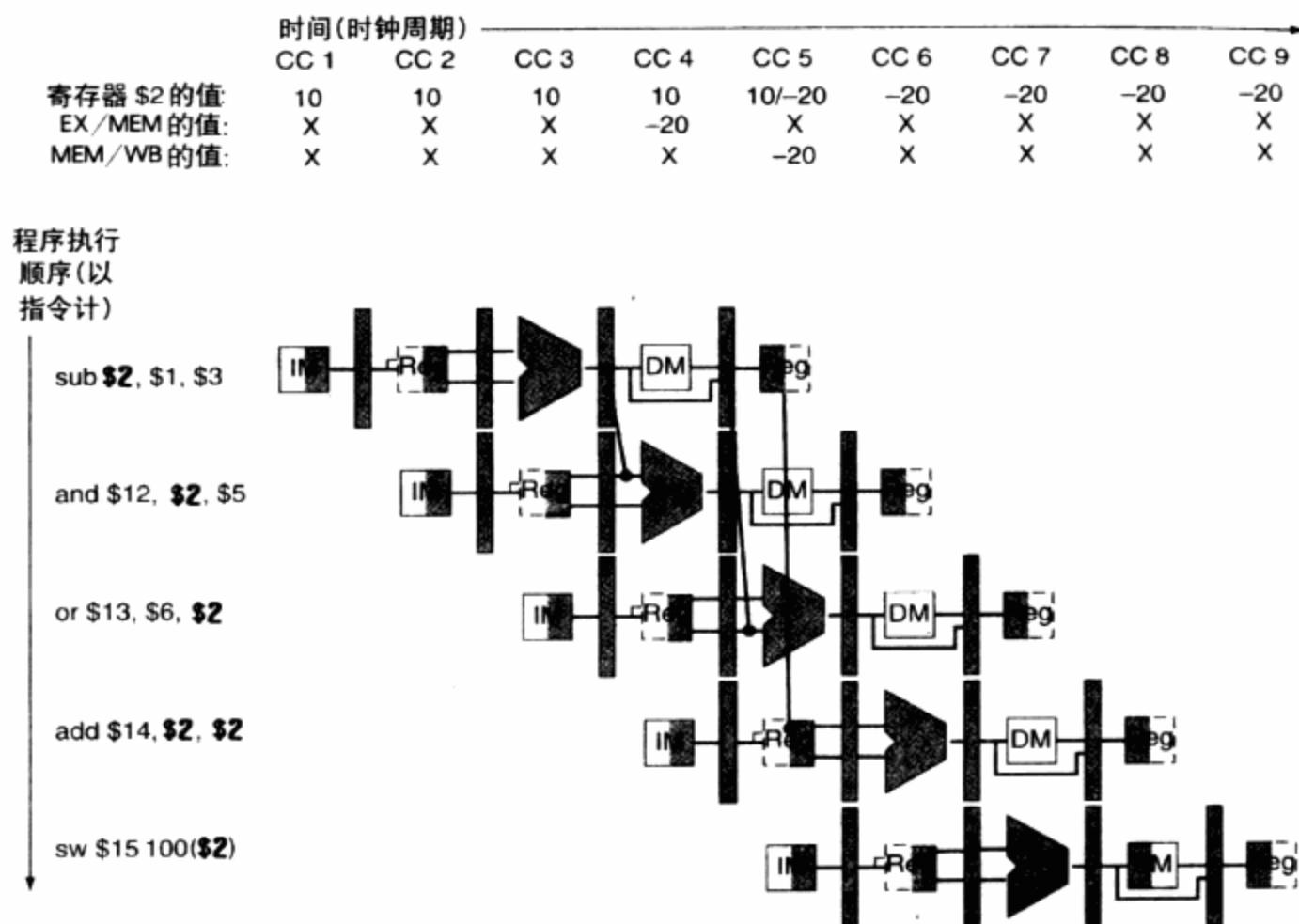


图 6-29 随着时间的推移流水线寄存器间的相关性，通过将流水线寄存器中找到的结果进行转发就有可能为 and 指令和 or 指令提供所需的 ALU 的输入

[流水线寄存器存有相应的值，因此在数据写入寄存器堆之前就已经有效。我们假定寄存器堆在同一个时钟周期内转发要读写的数据，因此 add 指令不用阻塞，但它的值不是来自流水线寄存器而是来自寄存器堆。寄存器堆“转发”是为什么在第 5 个时钟周期的开始寄存器 \$2 中的值是 10 而在周期结束时它的值却是 -20 的原因。即在这一时钟周期里读操作读取到的值是写操作写入的值]

现在，我们假设需要转发的指令只有四个 R 型指令：add、sub、and 和 or。图 6-30 给出了在加入转发机制前后 ALU 和流水线寄存器的示意图。图 6-31 表示的是在选择寄存器堆值和某一转发的数值间进行选择的 ALU 多路复用器的控制线的值。它要么选择寄存器堆值，要么选择被前进的一个值。

由于 ALU 转发多路复用器在 EX 步骤中产生，因此转发控制也在这一步骤中完成。因此，我们必须通过 ID/EX 流水线寄存器从 ID 步骤中传递操作数寄存器号，以决定是否转发相应的值。我们已经有了 rt 字段(20-16 位)。在转发前，ID/EX 寄存器不需要包括保存 rs 域的空间，因此 rs(25-21 位)被加入 ID/EX 中。

下面我们将给出检测冒险的条件以及解决冒险的控制信号：

1) EX 冒险：

```

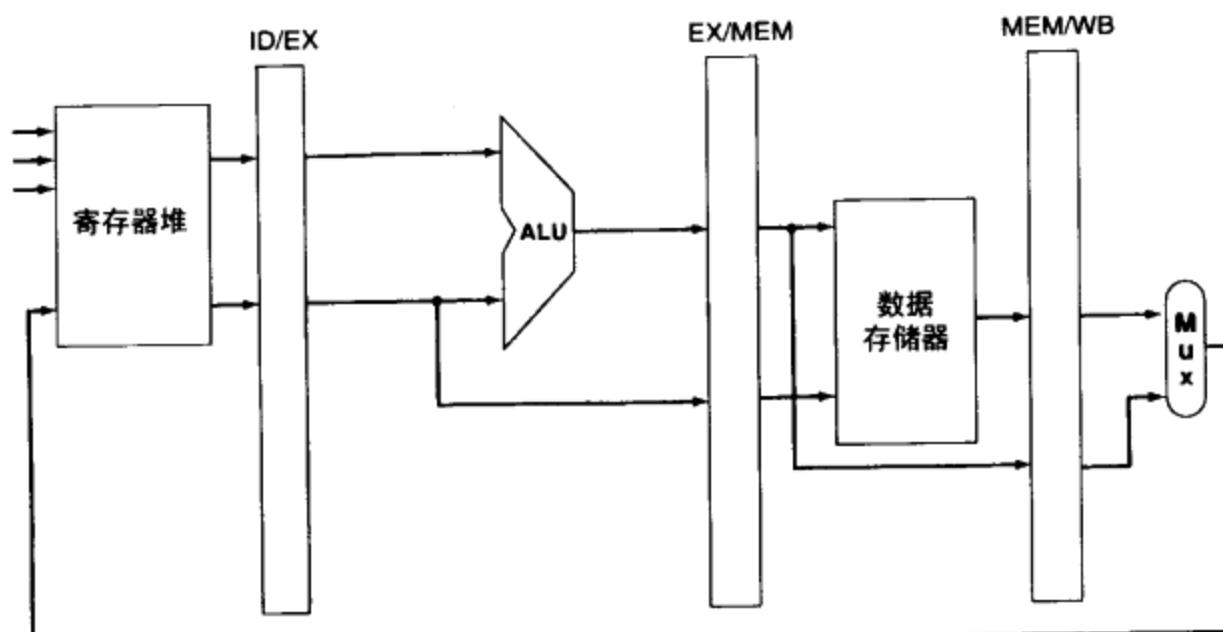
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

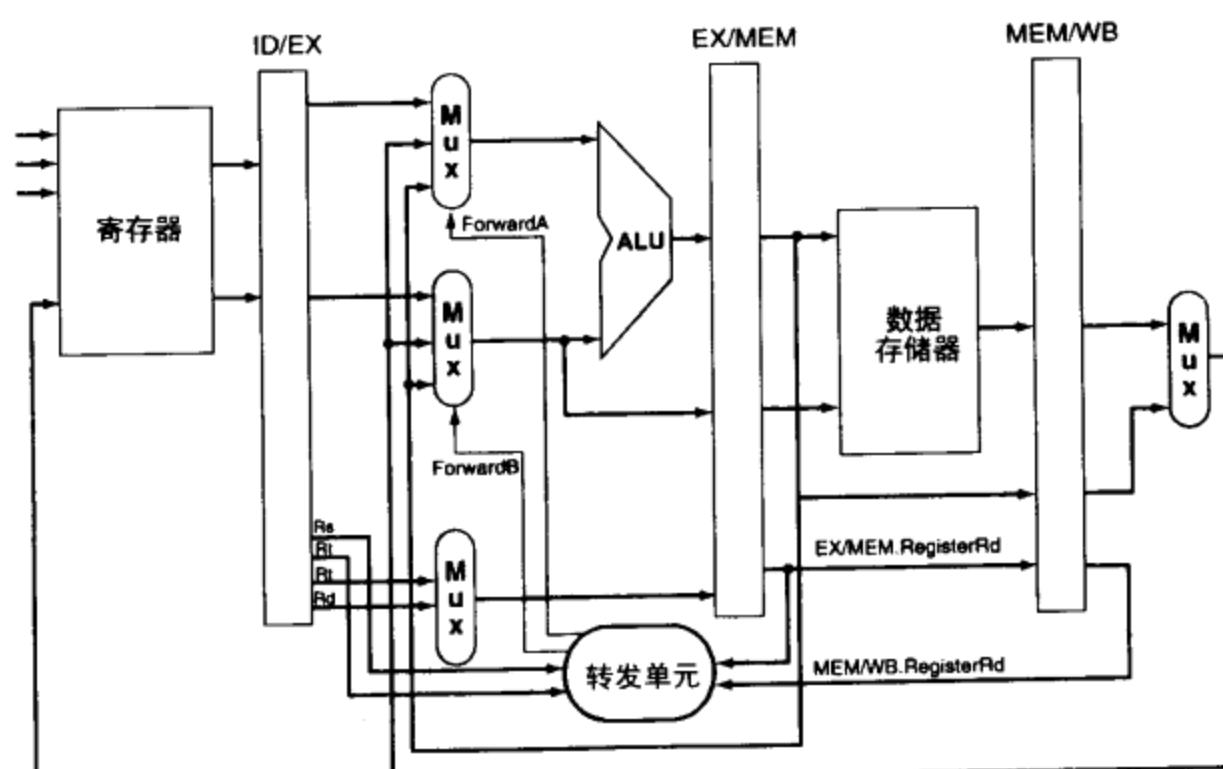
```

这种情况下将前一条指令的结果转发到任何一个 ALU 输入中。如果前一条指令要写寄存器

堆而且要写的寄存器号与 A 或 B ALU 输入的要读的寄存器号一致，只要不是寄存器 0，那么就调整多路复用器从流水线寄存器 EX/MEM 中读取数值。



a. 无数据转发



b. 有数据转发

图 6-30 上图是加入转发机制前的 ALU 和流水线寄存器。下图中使用多路复用器加入了转发路径，图中标示了转发单元

[本图只是一个示意图，图中没有标示诸如信号扩展硬件之类的有关全部数据通路的一些细节。需要注意的是 ID/EX.RegisterRt 字段在图中出现了两次，一次连接到 mux，一次连接到转发单元，但它本身只是一个信号。在前面的讨论中，忽略了存储值到存储指令的转发]

2) MEM 冒险：

```

if (MLM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
  
```

多路选择器信号线	源	解 释
ForwardA = 00	ID/EX	第一个 ALU 操作数来自寄存器堆
ForwardA = 10	EX/MEM	第一个 ALU 操作数由上一个 ALU 运算结果转发获得
ForwardA = 01	MEM/WB	第一个 ALU 操作数从数据内存或者前面的 ALU 结果中转发获得
ForwardB = 00	ID/EX	第二个 ALU 操作数来自寄存器堆
ForwardB = 10	EX/MEM	第二个 ALU 操作数由上一个 ALU 运算结果转发获得
ForwardB = 01	MEM/WB	第二个 ALU 操作数由数据内存或者前面的 ALU 结果中转发获得

图 6-31 图 6-30 中多路复用器的控制值

[作为 ALU 另一个输入的带符号的立即数将在本节的说明中详细解释]

正如上面所提到的，在 WB 阶段不会发生冒险，这是由于我们假设如果 ID 阶段指令读取的寄存器与 WB 阶段指令写入的寄存器是同一寄存器的话，就由寄存器堆提供正确的结果。这样，寄存器堆实现了另外一种形式的前进，但这种转发只发生在寄存器堆内部。

复杂的潜在数据冒险发生在 WB 阶段的指令结果、MEM 阶段的指令结果和 ALU 阶段的指令的源操作数之间。例如，在一个寄存器堆中对一系列的数字进行求和运算时，一系列连续的指令将会读/写同一寄存器：

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
...
```

在这种情况下，由于 MEM 阶段中的结果更新，因此更新的结果是由 MEM 步骤转发得到。这样，对 MEM 冒险的控制策略为(额外加入的部分用高亮表示)：

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

图 6-32 给出了支持转发所必需的硬件。

细节：转发还可以帮助解决 store 指令依赖其他指令导致的冒险。由于 store 指令在 MEM 阶段只使用一个数据值，所以转发十分容易。但考虑到 store 指令后紧跟着的是 load 指令，为了提高内存-内存拷贝的速度，我们就需要加入更多的转发硬件。如果我们重画图 6-29，并分别使用 lw 和 sw 指令代替 sub 和 and 指令，我们发现这时也有可能避免一次阻塞，因为 load 指令的 MEM/WB 寄存器中存在的数据能够及时地提供给 store 指令在 MEM 阶段使用。为了实现这个功能我们可能需要在内存访问阶段加入转发。避免这类冒险导致的数据通路的变化的问题留作习题。

图 6-32 中省略了 load 和 store 指令所需的输入到 ALU 的带符号的立即数。由于中央控制决定在寄存器和立即数之间选择，而且转发单元选择流水线寄存器作为 ALU 的寄存器输入，因此最简单的解决方法就是加入一个 2:1 的多路复用器，由它在 ForwardB 多路复用器输出与带符号的立即数之间进行选择。图 6-33 所示为改动后的电路。需要注意的是这种方法与我们在第 5 章中介绍的方法并不相同，后者扩展由线 ALUSrcB 控

制的多路复用器使之包括立即数输入。而这种方法通过将转发多路复用器输出(在执行 store 指令时它包含要存储的数据)连接到 EX/MEM 流水线寄存器同时也解决了 store 指令中的转发。

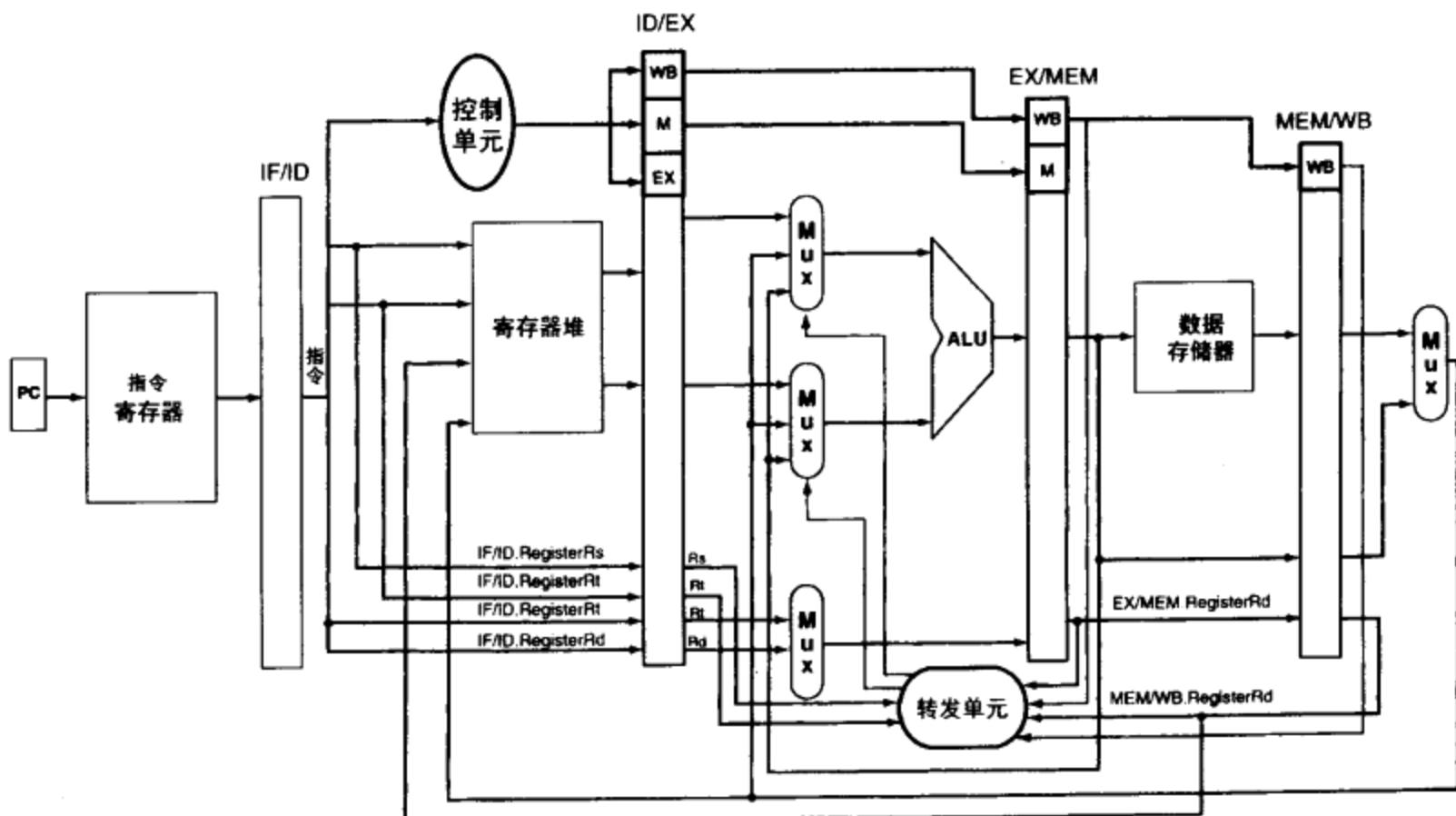


图 6-32 通过转发解决冒险的数据通路

[与图 6-27 的数据通路相比, 本图在 ALU 的输入部分加入了多路复用器。为了使本图表述更加清楚, 图中忽略了完整的数据通路的一些细节, 如分支硬件和符号扩展硬件]

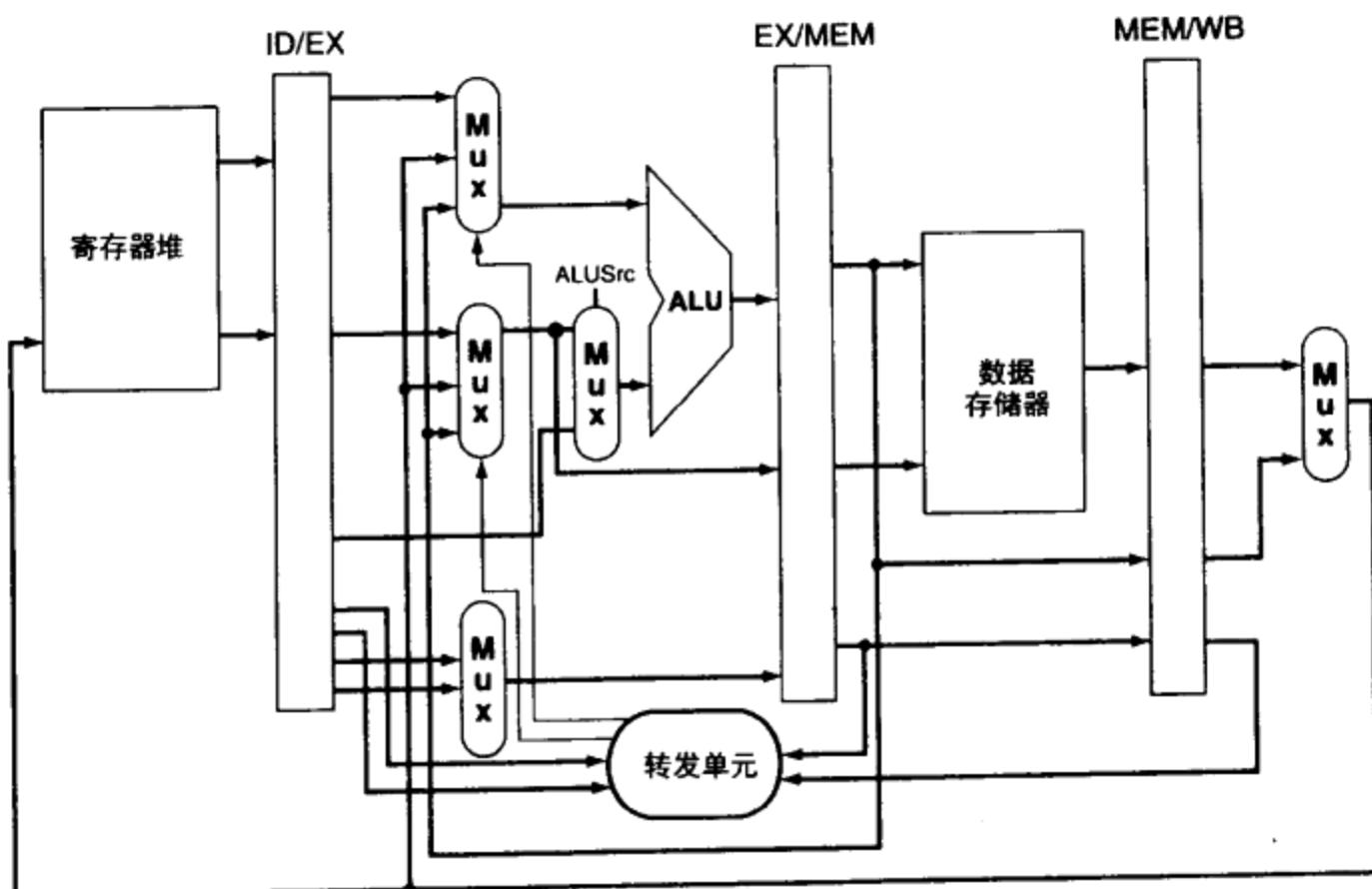


图 6-33 图 6-30 的数据通路的一个特写, 在图中加入了一个 2:1 的多路选择器, 用以选择带符号立即数作为 ALU 输入

6.5 数据冒险与阻塞

如果开始你没能够取得成功，那么请重新定义什么叫成功。

——佚名

如 6.1 节所述，当一条指令试图读取一个寄存器而它前一条指令是一条 load 指令，并且该 load 指令写入的是同一个寄存器时，定向转发的方法就无法解决问题。图 6-34 说明了这个问题。数据仍然是在第四个时钟周期从内存中读出，而 ALU 继续执行后续的指令。当 load 指令后紧跟着一个需要读取它的结果的指令时必须采用相应的机制阻塞流水线。

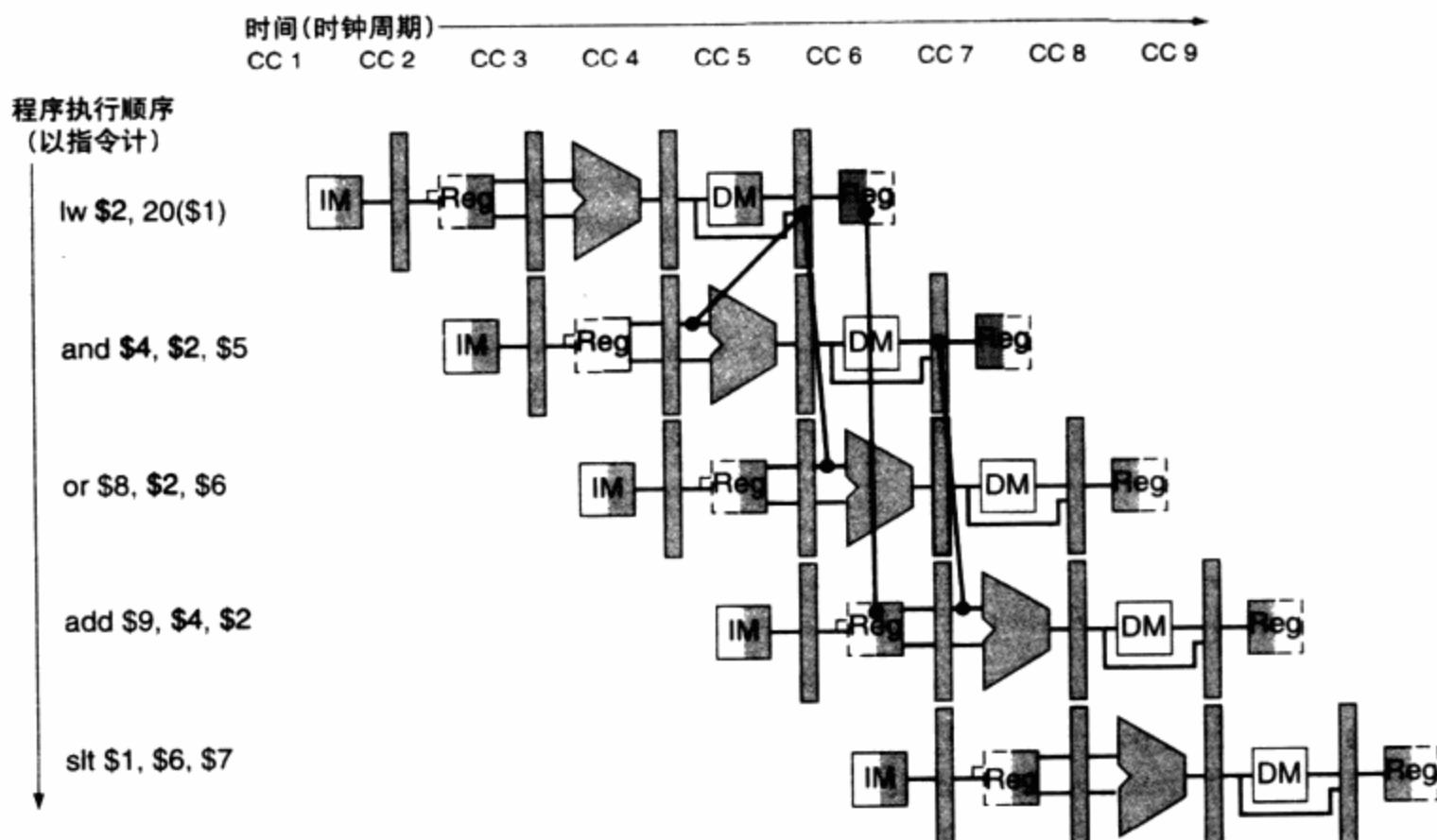


图 6-34 指令的一条流水线序列

[由于 load 指令和紧随其后的指令(and)之间的依赖关系在时间上是回溯的，所以这种冒险不可能通过转发来解决。因此，这类指令组合必然在冒险检测单元导致一次阻塞]

因此，除了一个转发单元以外，我们还需要一个冒险检测单元。它在 ID 阶段工作，从而在 load 指令与紧随其后利用它的结果的指令间插入阻塞。这个冒险检测单元检测 load 指令，它的控制满足如下的单一条件：

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline
```

因为惟一的一条读取数据内存的指令就是 load，因此第一行条件检查指令是否是一条 load 指令。后面的两行是检测在 EX 阶段的 load 指令的目的寄存器是否与在 ID 阶段的指令的某一个源寄存器相匹配。如果条件成立，指令将阻塞一个时钟周期。经过这一个周期的阻塞，转发逻辑就可以处理依赖并继续执行进程了。（如果没有转发，那么图 6-34 中的指令将需要另一个阻塞周期。）

如果处于 ID 阶段的指令被阻塞，那么处于 IF 阶段的指令也必须阻塞；否则，已经预取的指令就会丢失。防止这两条指令继续执行的方法保持 PC 寄存器和 IF/ID 流水线寄存器不变。如果

这些寄存器内容保持不变，在 IF 阶段的指令将继续使用相同的 PC 预取指令，而在 ID 阶段的寄存器将继续使用在 IF/ID 流水线寄存器中的相同的指令字段进行读取。再回到我们熟悉的洗衣店的类比例子中，这一过程就好像是你重新打开洗衣机洗相同的衣服而让烘干机继续空转一样。当然，类似于烘干机，EX 段后半部分开始的流水线始终是在做些什么事情的；它所做的就是执行没有任何效果的指令：nop^①指令。

nop 指令的效果与流水线气泡(bubble)相同，我们如何将 nop 指令插入流水线呢？在图 6-25 中，通过将 EX、MEM 和 WB 段的所有 9 个控制信号置零，我们可以产生一条 nop 指令。在 ID 段探测到冒险条件，我们可以通过将 ID/EX 段流水线寄存器的 EX、MEM 和 WB 控制字段置零来插入一个空隙。这些控制信号将传递到流水线后面的各级：由于控制信号均为零，所以不会对任何寄存器和存储器进行写入操作。

图 6-35 描述了硬件中发生的真实情况：和 and 指令相关的流水线执行槽被转换成空操作，而所有以 and 指令开始的指令都被延后一个周期。由于冒险条件满足，指令 and 和 or 将被迫在时钟周期 4 中重复时钟周期 3 中所做的内容，即指令 and 读寄存器并进行译码，指令 or 在指令存储器中重新取。这种重复的工作就像阻塞一样，但它的效果是拉伸了指令 and 和 or，并且延迟了第二个 add 指令的预取。就像洗衣流水线中的空气泡那样，一个延迟气泡把它后面的所有工作都进行了延迟，并且沿着指令流水线前进直到它在结尾处退出为止。

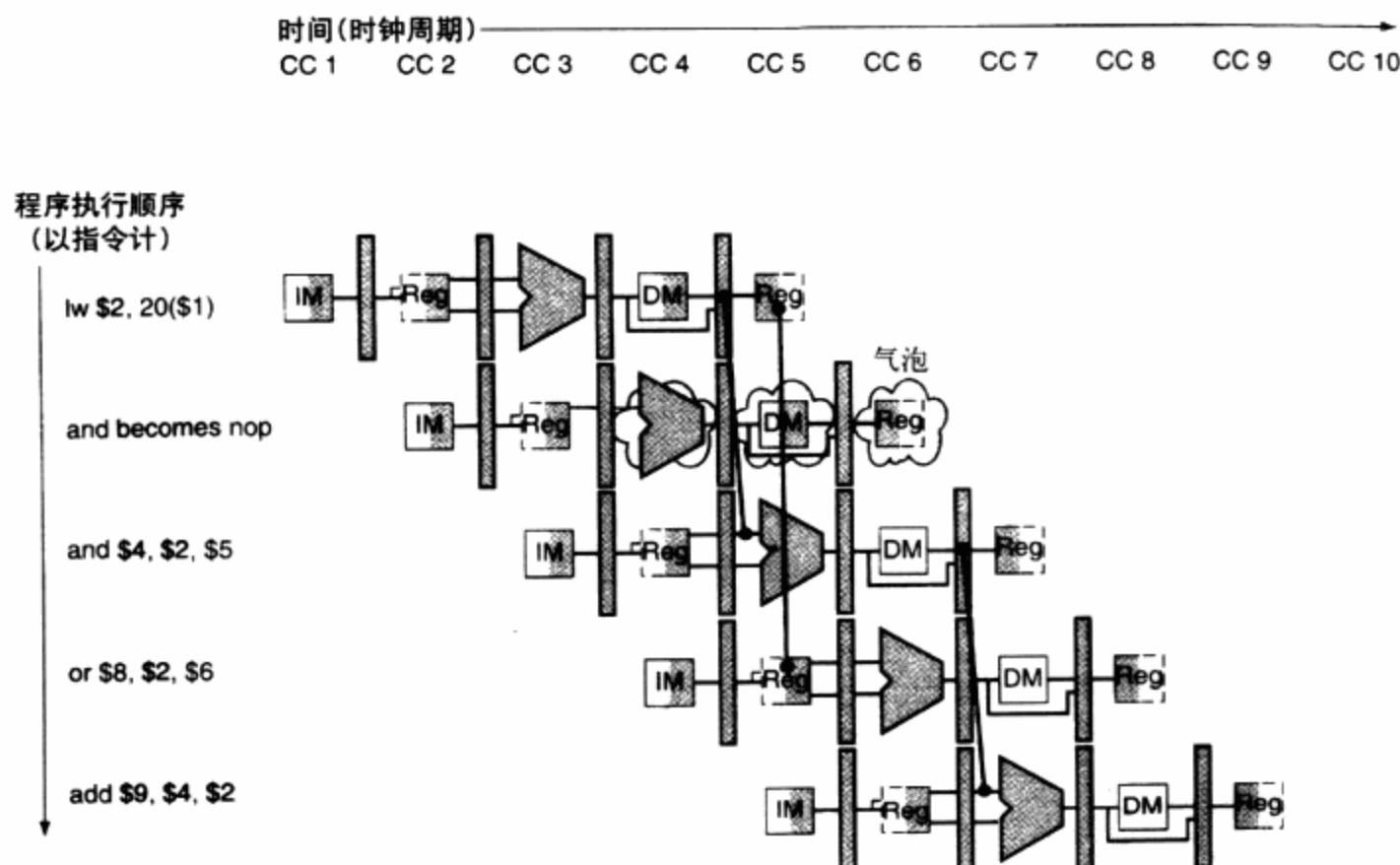


图 6-35 在正在执行的流水线中插入停顿的方法

[通过将 and 指令变换为一个 nop 指令，我们在第四个周期开始时插入了一个流水线气泡。我们注意到 and 指令在第二、三周期分别进行了取指和译码，但是它的执行一直推迟（相对于第四个周期而言）到第五个周期才进行。同样的 or 指令虽然在第三个周期进行了取指，但是它的 IF 级直至第五个周期才完成（本应在第四个周期完成）。在插入流水线气泡之后，原来在时间轴上与时间轴反向的依赖关系现在都转化为与之同一方向了，这样就消除了数据冒险]

^① nop 一条不改变状态的指令。

图 6-36 标注出了冒险检测单元和转发单元的流水线连接。和前面的介绍一样，转发单元控制 ALU 多路复用器，从而用相应的流水线寄存器的值代替通用寄存器的值。冒险检测单元控制 PC 和 IF/ID 寄存器的写入以及在实际控制值与全 0 中进行选择的多路复用器。如果上面的 load 指令冒险条件为真，冒险检测单元就阻塞并清除所有的控制字段。在本书光盘的 **For More Practice** 部分中有对应的单周期流水线图。

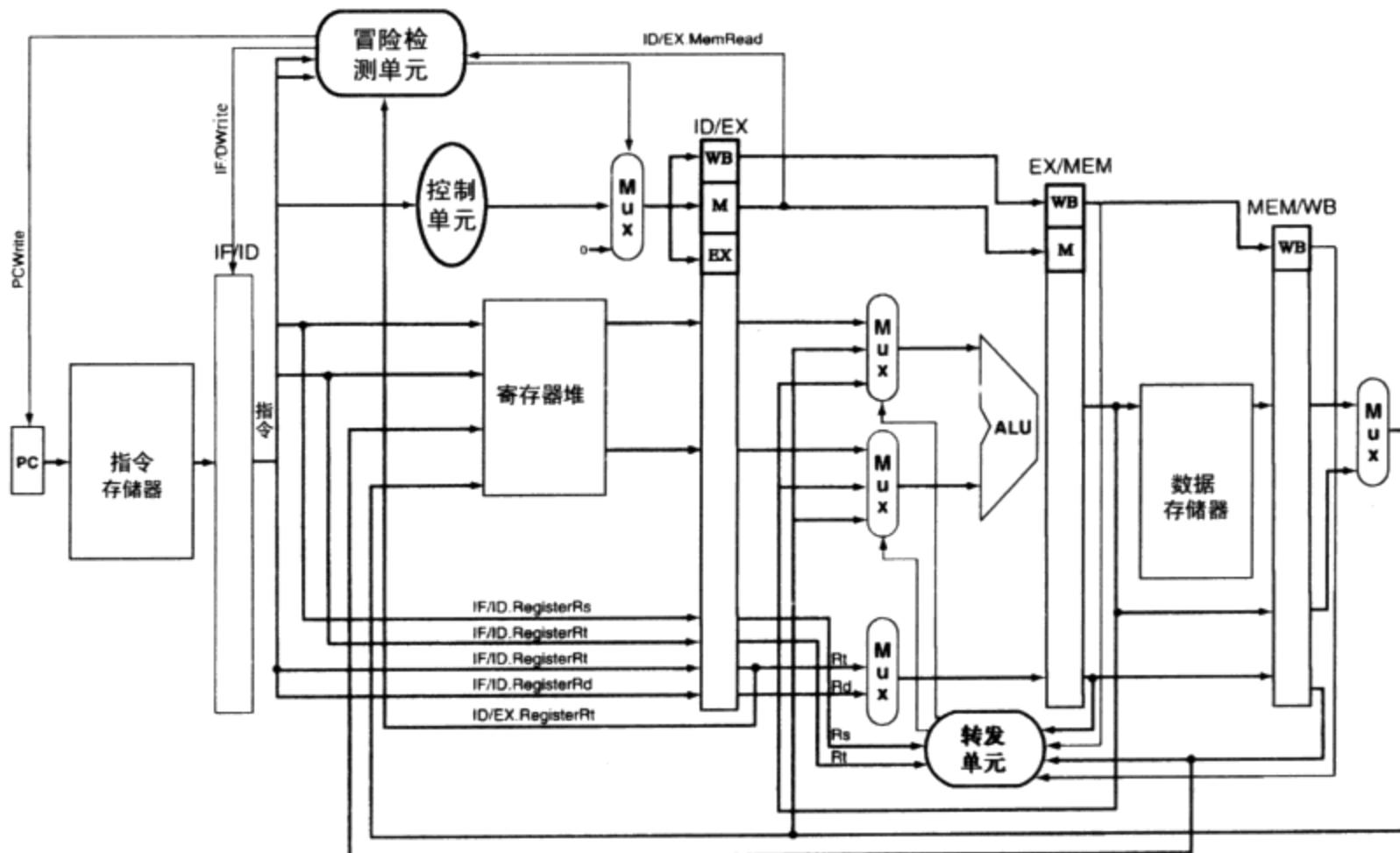


图 6-36 流水线控制概述，图中有两个转发多路复用器、冒险检测单元和转发单元

[虽然简化了 ID 和 EX 阶段(省略了经过符号化扩展的立即数和分支逻辑)，但本图说明了转发硬件需求的本质]

重点

虽然使用硬件的方法解决冒险依赖以保证指令的正确执行可能依赖于编译器，也可能不依赖于编译器，但为了获得最好的效果，编译器必须了解流水线。否则，无法预料的阻塞会降低编译代码的执行效率。

细节：重新考虑前面提到的为了避免写寄存器和内存将控制线置为 0，事实上只需要将信号 RegWrite 和 MemWrite 置为 0，而不用关心其他控制信号。

6.6 分支冒险

在对抗邪恶的斗争中，一击中的者非常少见，往往万里挑一。

Henry David Thoreau, *Walden*, 1854

截至目前，我们只把冒险的概念局限在算术运算和数据传输中。但正如 6.1 节中所提到的那样，还有一类冒险就是包含分支的流水线冒险。图 6-37 描述了一个指令序列，同时说明了在流水线中何时会发生分支。每个时钟周期内都要取指令，这样才能维持流水线的运行，但在我们的设计中必须等到 MEM 流水线阶段才能够确定是否执行分支。如 6.1 节所述，与我们前面讨论的数

据冒险相对应，这种为了确定预取正确的指令导致的延迟叫做控制冒险或分支冒险。

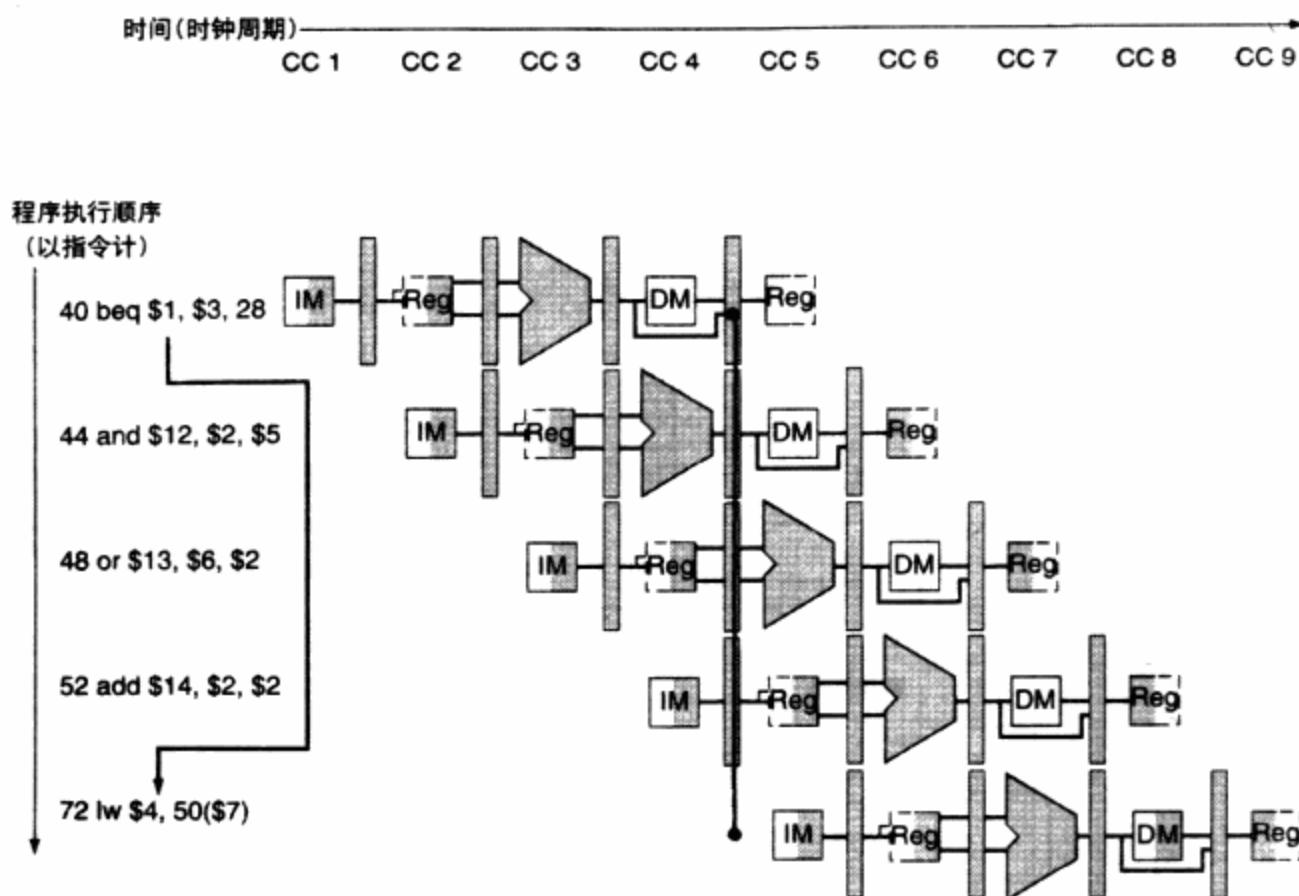


图 6-37 分支指令对流水线的影响

[指令左边的数字(40, 44, ...)是指令的地址。由于分支指令在 MEM 阶段(上面 beq 指令对应的时钟周期 4)决定是否执行分支, 分支后面的三条指令将被预取和执行。如果不加干涉的话, 这三条指令将在 beq 指令执行到地址 72 执行 lw 之前就开始执行(图 6-7 通过引入额外的硬件从而将控制冒险减少到一个时钟周期; 本图使用没有经过优化的数据通路)]

因为控制冒险相对易于理解, 它们出现的频率也比数据冒险小得多, 而且与采用转发的方法就能有效解决数据冒险相比, 我们还没有同样奏效的方法能够解决分支冒险问题, 因此, 这一节关于控制冒险讨论的篇幅要比前一节的数据冒险要短得多。这一节我们将介绍两种解决控制冒险的方案, 以及一种对这些方案进行优化的方法。

6.6.1 假定分支不发生

如 6.1 节所述, 阻塞直到分支发生完毕的速度实在太慢。一种比较普遍的提高分支阻塞速度的方法是假设分支不发生, 并继续执行顺序的指令流。如果分支发生的话, 就丢弃已经预取并译码的指令。指令的执行将沿着分支目标继续。如果分支不发生的可能性是 50%, 同时丢弃指令的代价很少的话, 那么这种优化方法可以将控制冒险的代价减半。

为了丢弃指令, 我们只需要将最初的控制信号置为 0 就可以了, 这一点与阻塞解决 load 指令的数据冒险一样。其不同之处在于当分支到达 MEM 阶段时必须改变分别在 IF、ID 和 EX 阶段的三条指令; 而对于 load 指令的阻塞只需要将 ID 阶段的控制值置为 0 并将其从流水线中退出即可。分支冒险中的丢弃指令意味着必须能够将流水线的 IF、ID 和 EX 阶段的指令都清除^①掉(flush)。

6.6.2 缩短分支的延迟

一种提高分支效率的方法是降低执行分支的代价。截至现在我们都假设直到 MEM 阶段才能确定分支结构要执行的下一条指令的 PC。而如果我们能在流水线中提早分支指令的执行过程,

^① 清除(指令)(flush (instructions)) 丢弃流水线中的指令(往往出于不可预料的事件)。

那么我们就能减少需要清除的指令数。MIPS 结构支持单周期快速分支指令，这些指令的流水线较短，开销也较小。许多设计者观察到许多分支指令的结果仅仅依赖于一些简单的测试操作（比如：是否相等，或者符号的正负等），这些测试操作并不需要完整的 ALU 操作，而很可能用几个门电路就可以完成。当需要一个复杂的分支决策操作时，另有一条指令通过使用 ALU 来完成所需要的比较操作，这类似于为分支指令提供条件码。

提前分支指令的决策过程需要提前完成两个操作：计算分支的目的地址和判断分支指令的跳转条件。前者较为简单：我们已经有 PC 值和 IF/ID 流水线寄存器中的指令立即数，我们只需将分支加法器从 EX 段移到 ID 段；当然，我们将针对所有指令都执行分支目的地址的计算过程，但是我们只在需要它的时候才会用到。

提前确定分支跳转条件的难度较大。对于相等分支指令而言，我们在 ID 段比较读取寄存器得到的两个值以判断它们是否相等。我们可以将它们对应的各位都做异或操作，然后把各位的结果取或操作来判断是否相等。将分支条件判断提早到 ID 段意味着我们须对电路加入额外的硬件以支持数据旁路和检查冒险条件，这样做是因为有可能某条分支指令依赖于某个流水线中的指令产生的结果，只有加入额外电路才能对分支指令提供数据旁路服务。比如：为了支持相等/不等分支指令，我们必须将结果通过数据旁路提供给 ID 段中测试数据是否相等的电路。这里面有两个使事情变得更加复杂的因素：

1) 在 ID 段中，我们必须对指令进行译码以决定是否我们真的需要将数据取给比较电路，而后进行比较操作，这样才能在这条指令是分支指令的前提下设置 PC 为跳转的目的地址。数据旁路原来由 ALU 相关的电路负责，我们引入 ID 段中的比较电路势必同时引入另一套数据旁路电路。注意到分支指令的源操作数可能来自于 ALU/MEM 或者 MEM/WB 流水线寄存器。

2) 由于我们在 ID 段就计算出了分支指令的比较结果，但是同时这个结果又会在后面的某个周期被重新生成，所以有发生数据冒险的可能，这可能会最终引发流水线的停顿。比如，如果分支指令的前一条指令是 ALU 指令，而它的结果用于了该分支指令的比较操作，我们就必须停顿流水线，因为 ALU 指令的 EX 阶段发生在分支指令的 ID 阶段之后。

尽管有上述的困难，将分支指令的执行过程提前到 ID 段的确有一定的改善，因为在分支结果是跳转的条件下，它将分支的开销减少到了一条指令（也就是取指令得到的那条指令）。习题中有如何实现数据旁路和冒险探测电路的具体细节。

为了在 IF 阶段清除一条指令，我们加入了一条称作 IF.Flush 的控制线，即将 IF/ID 流水线寄存器的指令字段置为 0。清空寄存器的结果是将预取到的指令转变成为 nop 指令，后者不执行任何会改变状态的操作。

例题 流水线分支

假定流水线为不发生的分支进行了优化，并且分支的执行提前到了流水线的 ID 阶段，说明下面的指令序列在分支发生时的执行情况：

```

36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40 + 4
+ 7 * 4 = 72
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw $4, 50($7)

```

解 图 6-38 描述了分支产生时指令序列的执行情况。与图 6-37 不同，这时在一个发生的分支上只有一个流水线气泡。 ■

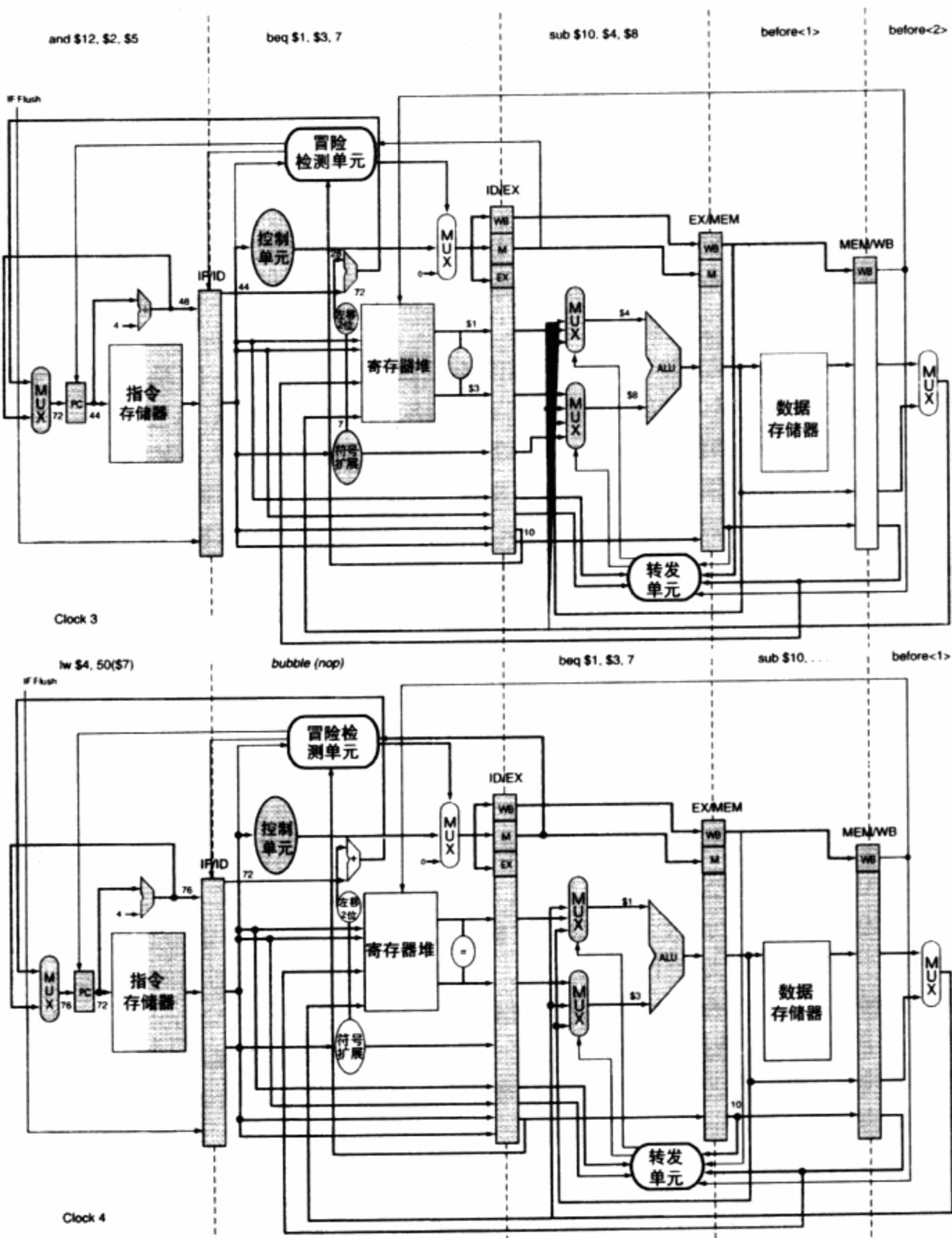


图 6-38 在第三个时钟周期 ID 阶段确定分支发生，因此地址 72 被选为下一个 PC 地址同时将为下一个时钟周期预取的指令置为 0

[时钟周期 4 的图描述了地址为 72 的指令被预取，并且分支发生的后果是在流水线中产生了一个气泡或者一条 nop 指令。(由于 nop 实际上是指令 s11 \$0, \$0, 0, 所以时钟周期 4 的 ID 阶段是否应该标出还有待商榷)]

6.6.3 动态分支预测

假设分支不发生是一种简单的分支预测方法。这种方法预测分支不发生，如果预测错误就清空流水线。对于简单的 5 段流水线而言，若辅以编译器的分支预测支持，这很可能会是一种合理的方案。对于较深的流水线而言，若以时钟周期数来衡量的话，分支的开销会变大。同样地，对

于多发射处理器，分支开销从丢弃的指令数的角度而言也在增大。综合这两个因素，对于一个高性能处理的流水线而言，简单的静态的分支预测机制可能会对性能造成过多的浪费。如 6.1 节所述，如果有更多的硬件支持，我们就可能在程序执行时实现一些其他的分支预测方法。

一种策略是查找指令的地址观察上一次执行该指令时分支是否发生，如果上次执行时分支发生就从上次分支发生的地方开始取指令。这种技术称为动态分支预测^①。

动态分支预测的一种实现方法是采用分支预测缓存或分支历史表^②。分支预测缓存是一个用分支指令的地址低位寻址的小型记忆体。其中记录了分支最近是否执行。

分支预测缓存器是一种最简单的缓存；其实我们并不知道预测的是否是当前的分支指令——该执行结果可能是别的某条分支指令的执行结果。但这并不影响这种方法的正确性。预测只是一种假设正确的提示，在这个基础上，指令预取沿着预测的方向进行。如果这种假设错误，不正确的预测指令会被删除，预测位将取非，并返回原来位置，继续按照正确的指令顺序取指并执行。

这种简单的使用一个预测位的预测方法具有性能上的缺陷：即使一个分支几乎总是发生，但它仍然可能有多个(而不是一次)分支不发生从而导致预测错误。下面的例子说明了这种问题。

例题 循环与预测

让我们看一个循环分支，它在一行上分支发生九次，而不是只发生一次。假设分支的预测位保存在预测缓存中，这种分支预测的正确率是多少？

解 静态预测方法会在第一次和最后一次的循环迭代时预测错误。由于分支在一行上发生了九次，因此预测位在最后一次循环时被设为分支发生，而且这次预测错误是不可避免的。而在第一次循环时发生预测错误是因为预测位在循环的上一次重复时被前一个执行设置为不执行(在那次退出的循环中分支并没有发生)。因此这个预测方法在 90% 的时间中分支发生的情况的预测的正确性只有 80%(两次错误预测，八次正确预测)。 ■

在理想的情况下，在这种高度规则的分支结构中预测的正确性与发生分支的频率相匹配。为了弥补这一缺陷，经常使用两位预测位的方案。在两位的方案中，必须发生两次预测错误才改变预测。图 6-39 给出了有限状态机的两位预测方案。

分支预测缓冲可以使用在 IF 流水线阶段指令地址能够访问的小容量的专用缓冲实现。如果指令预测分支发生，那么一旦获得新的 PC 就从该目标地址开始预取指令(如 6.6.2 节所述，在 ID 阶段就可以获得 PC)，否则就顺序预取指令并继续执行。如果预测的结果是错误的，就按照图 6-39 说明的方法改变预测位。

细节：如 6.1 节所述，通过重新定义分支，在五级流水线中我们将控制冒险转化为一种可用的特性。一个被延迟的分支总会执行下一条指令，但分支后的第二条指令会受到分支的影响。

编译器和汇编器都会试图把总在分支后执行的那条指令放入分支延迟槽^③中。这些软件的作用就是使后续的指令有效并且有用。图 6-40 给出了三种调度分支延迟槽的方法。

延迟的分支调度的限制在于(1)被调度到分支延迟槽中的指令的限制(2)我们在编译时对分支发生与否的预测能力。

延迟分支对五级流水线每个时钟周期发射一条指令而言是一个简单且有效的解决方

① 动态分支预测(dynamic branch prediction) 在运行时根据运行时信息进行分支预测。

② 分支预测缓存(branch prediction buffer)又称分支历史表(branch history table) 一个用分支指令地址的低位寻址的小型记忆体，其中包含一位或多位，记录了分支指令最近是否执行。

③ 分支延迟槽(branch delay slot) 在 MIPS 体系中由不影响分支的指令构成的紧跟延迟的分支指令的槽。

案。随着处理器向更长的流水线以及单周期多指令的方向发展(参见6.9节),分支延迟时间变得更长而一个单延迟槽实际上并没有多大作用。而且,随着单芯片上的晶体管的数目的增加,动态预测器已经越来越普遍了。

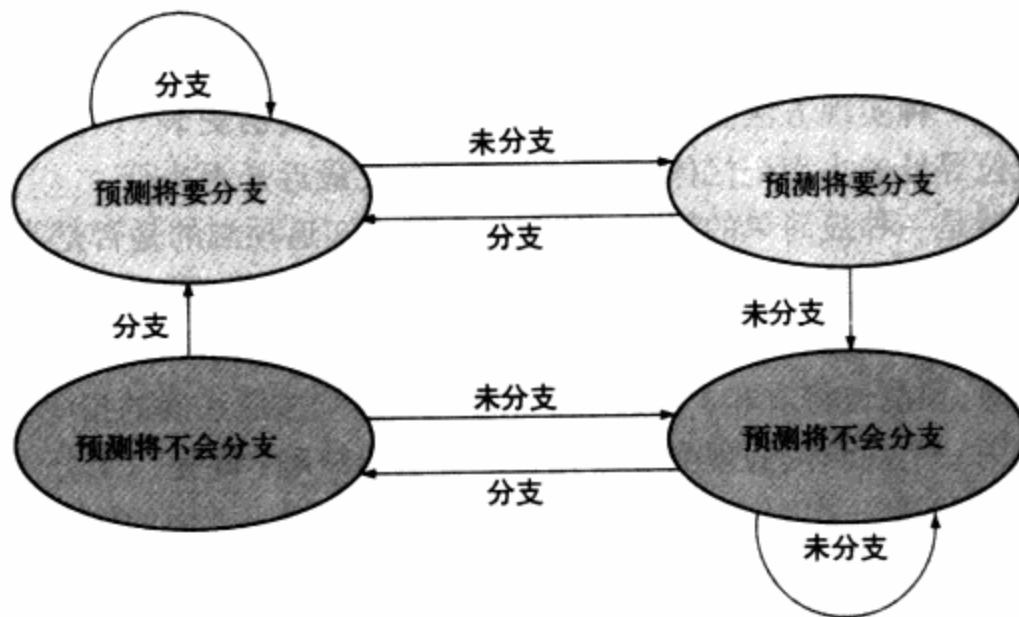


图 6-39 两位预测方案的状态图

[之所以使用两位预测位而非一位预测位,是因为在分支经常发生或经常不发生的情况下(大多数分支都是这样)只会发生一次预测错误。两位数据在系统中表示成四种状态。这种基于两位的方案属于基于计数器的预测器:计数器在预测正确的情况下递增,反之则递减,其所覆盖范围的中心点将被作为判断是否跳转的分水岭]

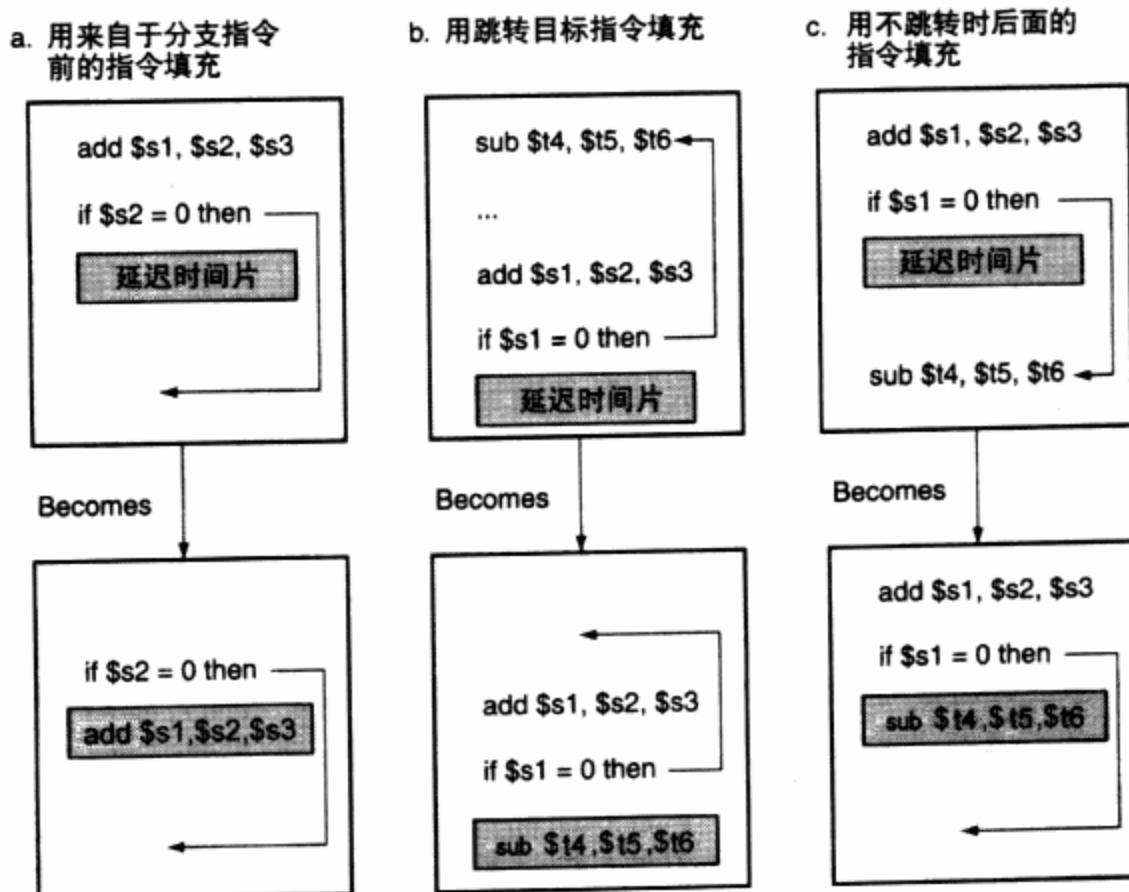


图 6-40 分支延迟时间片的调度

[每一对方框中的上面一个表示调度前的代码;下面一个表示调度后的代码。在方案(a)中,延迟时间片通过插入分支之前的一条不相关的指令实现。这是一种最佳的选择。当方案(a)无法实现时,就使用方案(b)和(c)。在(b)和(c)的代码序列中,分支条件中使用 \$s1 从而阻止了将 add 指令(它的结果寄存器是 \$s1)移入分支延迟。在(b)和(c)的代码序列中,分支延迟时间片是依照分支的目标地址调度的;由于目标指令可以通过其他的通路访问到,时间片。(b)中的分支延迟时间片是按照分支的目标地址调度的;由于目标指令可以通过其他的通路访问到,通常需要将它们进行复制。当分支发生的可能性比较大时,一般选择方案(b),如循环分支。否则就采用(c)预测分支不发生时,不跳转时用后面的指令进行调度。为了使(b)和(c)中的优化合法,当分支走向非预测方向的时候,指令 sub 必须能够“正常”的执行。“正常”意味着虽然有些工作是多余的,但程序必须依然能够正确执行。例如,当分支将跳转到非预测方向时 \$t4 是一个未被使用的临时寄存器就是这种情况]

细节：分支预测器虽然能预测分支发生与否，但是仍需要计算分支的跳转目的地址。在五级流水线中，这个计算将消耗一个周期，这意味着跳转的分支将有一个周期的开销。延迟的分支指令是一种消除这种开销的方法。另一种方式是用一个缓冲器(cache)记录跳转目标程序计数器或目标指令，这需要引入一个**分支目标缓冲**^①。

细节：两位动态分支预测机制只使用到了单个分支指令的信息。一些研究人员发现如果同时使用当前分支指令的局部和全局的运行结果，在使用相同多的预测器位的前提下，我们可以获得更高的预测精度。这类预测称为**相关预测**^②。一个典型的相关预测中与每条分支指令对应的有两个两位预测单元，以及基于分支运行结果(即分支发生与否)选择合适的分支预测单元。所以，全局分支指令行为可以看作是对预测结果的判定额外加入了一个寻址机制。

在分支预测方面最近的一个创新是锦标赛预测。**锦标赛预测**^③使用了多种预测单元，并记录它们谁产生的结果最好。一个典型的锦标赛预测器可能为每一个分支地址都保留两个预测结果：一个是基于局部信息的预测结果，另一个则基于全局的分支行为。另外一个选择子决定每次的预测究竟采用哪个预测单元。这个选择子可以按照1位或者2位预测器的机制工作，以能选择精度较好的预测单元。许多现代的高级微处理器都采用了类似的复杂预测单元。

6.6.4 流水线小结

截至目前，我们介绍了单时钟周期、多时钟周期和流水线三种指令执行模型。流水线控制尽力在一个时钟周期完成一条指令，这一点与单时钟周期模型一样，但实际上与多时钟周期模型一样，它的时钟周期要短得多。让我们再回顾一下比较单周期和多周期处理器的例子。

例题 比较几种控制方案的性能

使用SPECint2000的指令比例(见5.4.3节和5.5.2节的例子)，比较单周期、多周期和流水线控制方式的性能。假定所有的单元都有5.4.3节例子中的时钟周期。对于流水线执行方式，假定一半的load指令之后紧跟着使用该load执行结果的指令；假定分支错误预测的开销为1个周期；假定1/4的分支指令预测错误；假定跳转指令总会产生一个周期的延时，也就是消耗两个时钟周期。忽略其他的冒险问题。

解 从5.4.3节的例子(单周期处理器的性能)，我们可以得到功能单元的延时：

- 访存时间为200 ps
- ALU操作时间为100 ps
- 寄存器文件读写延时为50 ps

对于单周期数据通路，这将产生时钟周期

$$200 + 50 + 100 + 200 + 50 = 600 \text{ ps}$$

5.5.2节的例子中(多周期处理器的CPI)，指令分布如下：

- 25% load指令

① 分支目标缓冲(branch target buffer) 一个用于记录分支目的PC或目的指令的缓存。它的常用组织方式是采用高速缓存的结构，因而相对于简单的预测缓冲器而言它的开销更大。

② 相关预测(correlating predictor) 一种使用分支指令的局部行为特性和最近执行的分支的全局信息的分支预测器。

③ 锦标赛分支预测(tournament branch predictor) 一种使用多种分支预测器并选择某个行为最好的预测器的预测结果的预测机制。

- 10% store 指令
- 11% 分支指令
- 2% 跳转指令
- 52% ALU 计算指令

此外，5.5.2 节例子中指出：多周期处理器的 CPI 值为 4.12。多周期的数据通路和流水线的时钟周期必须与功能单元最长的时间延时相同，即为 200 ps。

对于基于流水线的设计而言，当不存在 load-use 型依赖关系的时候 load 指令执行时间为 1 个周期，如果存在该依赖关系则执行时间为 2 个周期。所以，load 指令执行的时钟周期数平均为 1.5 个。store 指令和 ALU 指令执行时间都是 1 个周期。分支指令如果正确预测则需 1 个周期，错误预测情况下需 2 个周期。所以分支指令平均消耗的周期数为 1.25 个。跳转指令需要 2 个时钟周期。所以，平均的 CPI 值为：

$$1.5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1.25 \times 11\% + 2 \times 2\% = 1.17$$

下面让我们比较三种设计中平均每条指令所消耗的时间。对于单周期设计而言，是定长的 600 ps。对于多周期设计而言，是： $200 \times 4.12 = 824$ ps。对于流水线设计而言，是： $1.17 \times 200 = 234$ ps。与其他任何一种设计相比，都至少快一倍以上。

聪明的读者会发现访存操作消耗了较长的时钟周期，它是流水线设计和多周期设计共同的瓶颈所在。将访存分解为两个周期，可以将时钟周期降低为 100 ps，这对两种方案都带来了性能上的改进。我们将在练习中详细探讨这个问题。 ■

本章从洗衣店的例子出发，说明了流水线原理在日常生活中的应用。通过类比，我们从单周期数据通路开始，逐渐引入流水线寄存器、数据旁路、如何检测数据冒险、分支预测和如何清除流水线中的指令等内容，一步步地讲述了指令流水线的原理。图 6-41 是最终得到的数据通路和控制电路。

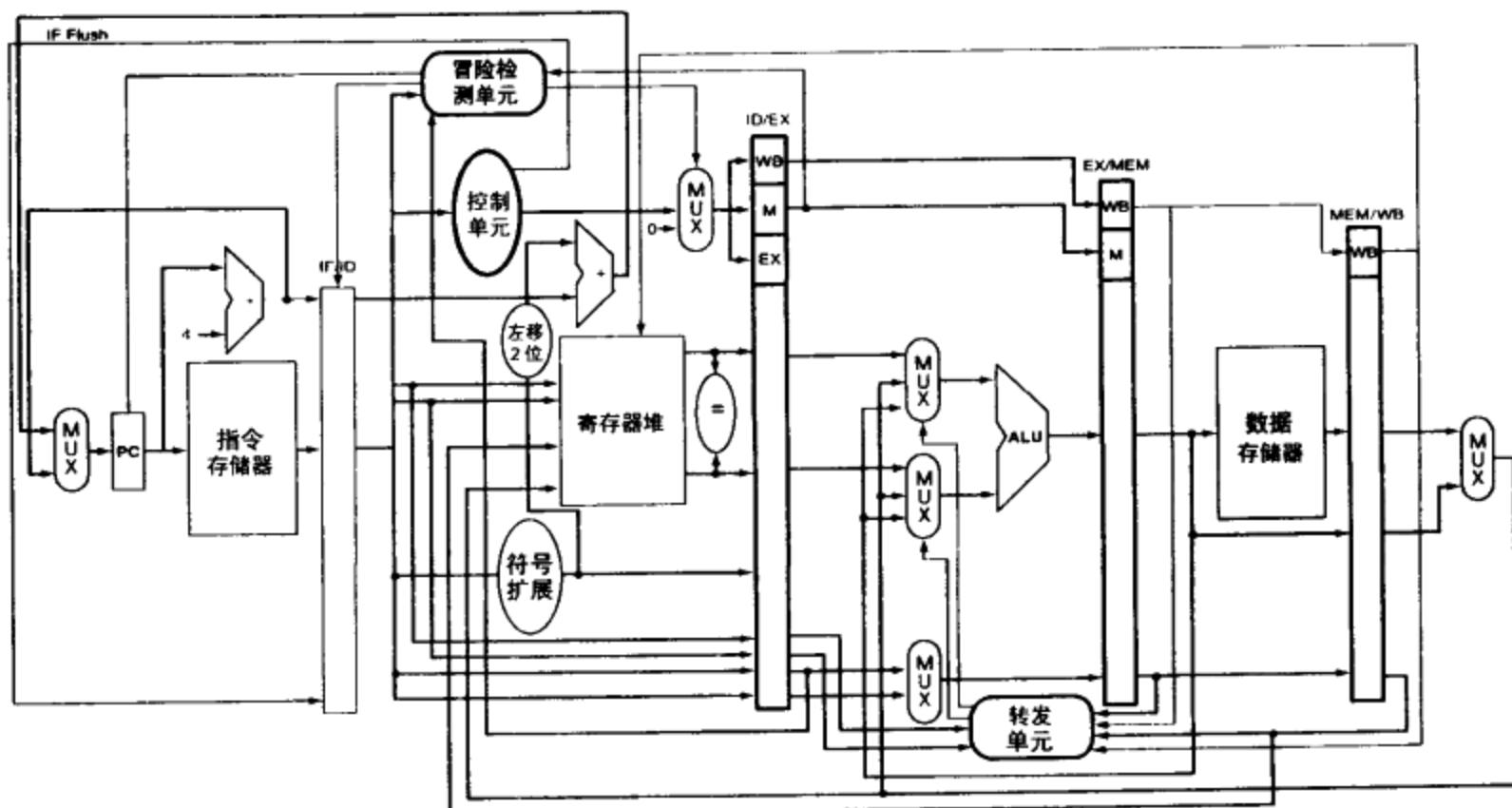


图 6-41 本章最终完成的数据通路和控制电路

自测

考虑三种分支预测方法：预测分支不执行，预测分支执行，动态分支预测。假设它们错误预测的开销都为 2 个周期，而正确预测是均无开销。同时假定动态分支预测的精度为 90%。那么对下面三种分支指令而言，最好的预测器各是哪一种？

1. 以 5% 的可能性跳转的分支
2. 以 95% 的可能性跳转的分支
3. 以 70% 的可能性跳转的分支

■ 6.7 使用硬件描述语言对流水线进行建模

本节出现在本书光盘上，其中提供了用 Verilog 语言描述的 MIPS 机器的 5 级流水线模型。最初的模型忽略了冒险的情况，后面的模型进而考虑了数据旁路、数据冒险和分支冒险。

6.8 异常

制造具有自动中断程序功能的计算机并不是一件容易的事情，因为在中断发生时有很多处于不同执行状态的指令。

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962

另一种形式的控制冒险是异常。例如，假设指令：

```
add $1,$2,$1
```

产生了一个算术溢出。因为我们不希望这个无效值影响其他的寄存器和内存区，因此我们需要在这条指令之后立即将控制传递给对应的异常处理例程。

如同我们在上一节所需要进行的处理一样，我们必须清除流水线中指令 add 后面的所有指令并且从一个新的地址开始预取指令。我们可以采用与发生分支时同样的机制，但是这里是由于异常导致了控制线的重置。

我们已经知道如何通过将 IF 阶段的指令转换成指令 nop 清除指令的方法处理分支预测的错误。为了清除 ID 阶段的指令，我们使用 ID 阶段已有的多路复用器，将控制信号清零以产生阻塞。一个称作 ID.Flush 的新的控制信号与来自冒险检测单元的阻塞信号进行与(or)操作，从而在 ID 阶段清除指令。为了清除 EX 阶段的指令，我们使用了一个称作 EX.Flush 的新的信号，用它控制新的多路复用器将控制线清零。为了从算术溢出的异常地址 8000 0180_{hex} 开始预取指令，我们只要加入一个额外的输入到 PC 多路复用器，由它将 8000 0180_{hex} 传递到 PC。图 6-42 具体描述了这种变化。

这个例子指出了异常存在的一个问题：如果我们不在指令的中间停止执行，程序员将无法看到导致溢出的寄存器 \$1 中的原始值，因为它将作为指令 add 的目的寄存器被冲掉了。由于经过认真的计划，我们决定异常溢出在 EX 阶段被检测出来；因此我们可以使用 EX.Flush 信号来避免在 EX 阶段的指令在它的 WB 阶段的写入结果。许多异常情况要求我们最终完成那条产生异常的指令，就像它没有产生异常一样。要实现这点最简单的方法是将这条指令消除掉，进而在异常被处理的地方将这条指令重新执行一次。

异常处理的最后一步将导致异常的指令的地址保存到异常程序计数器 (Exception Program Counter, EPC) 中，这等同于我们在第 5 章中所做的工作。实际上，我们保存的是原始地址 + 4，因此异常处理例程必须先从保存的地址中减去 4。图 6-42 给出了一个数据通路，其中包括分支硬件以及为处理异常所进行的必要调整。

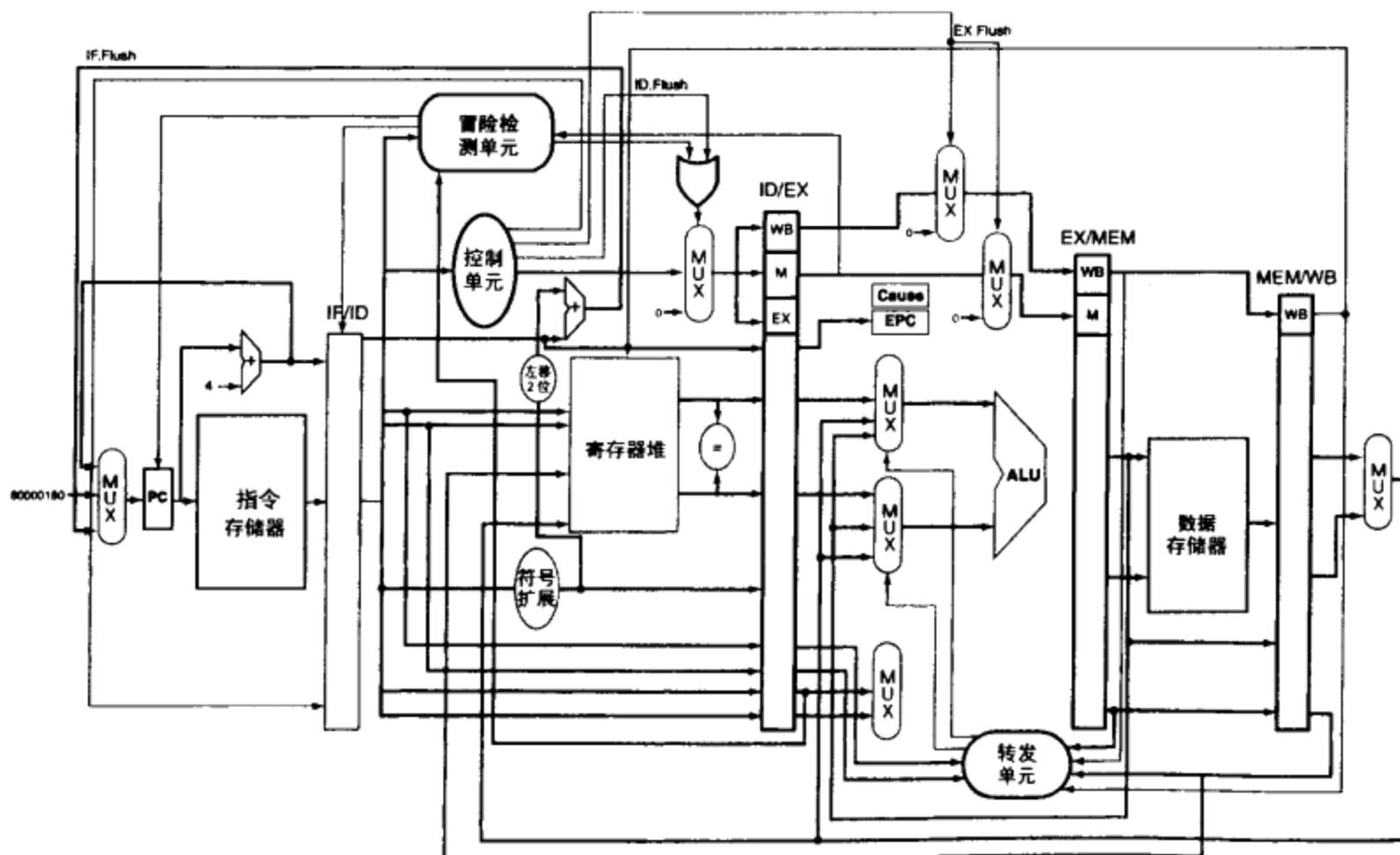


图 6-42 带有异常处理控制的数据通路

[本图中主要加入了一个新的输入，它的值为 $8000\ 0180_{hex}$ ，它作为产生一个新 PC 值的多路复用器的输入；还有一个记录异常发生原因的原因寄存器；以及一个保存导致异常的指令的地址的异常 PC 寄存器。输入到多路复用器的 $8000\ 0180_{hex}$ 是在发生异常时开始预取指令的地址。尽管图中未示出 ALU 溢出信号，其实它也是控制单元的一个输入]

例题 流水线计算机中的异常

给定如下指令序列

```

40hex    sub    $11, $2, $4
44hex    and    $12, $2, $5
48hex    or     $13, $2, $6
4Chex    add    $1, $2, $1
50hex    slt    $15, $6, $7
54hex    lw     $16, 50($7)
...

```

假设在异常发生时执行的指令开始部分是：

```

40000040hex    sw      $25, 1000($0)
40000044hex    sw      $26, 1004($0)
...

```

说明在指令 add 发生溢出时流水线的执行情况。

解 图 6-43 从指令 add 的 EX 阶段开始描述了整个过程。在指令 add 的 EX 阶段检测到发生溢出， $4000\ 0040_{hex}$ 被强制写入 PC。在第七个时钟周期指令 add 及其后面的指令都被清除，处理异常的代码的第一条指令被预取。需要注意 add 后面的指令的地址 $4C_{hex} + 4 = 50_{hex}$ 被保存。 ■

第 5 章中给出了一些其他导致异常的原因：

- I/O 设备请求
- 用户程序调用操作系统服务

- 使用未定义指令
- 硬件错误

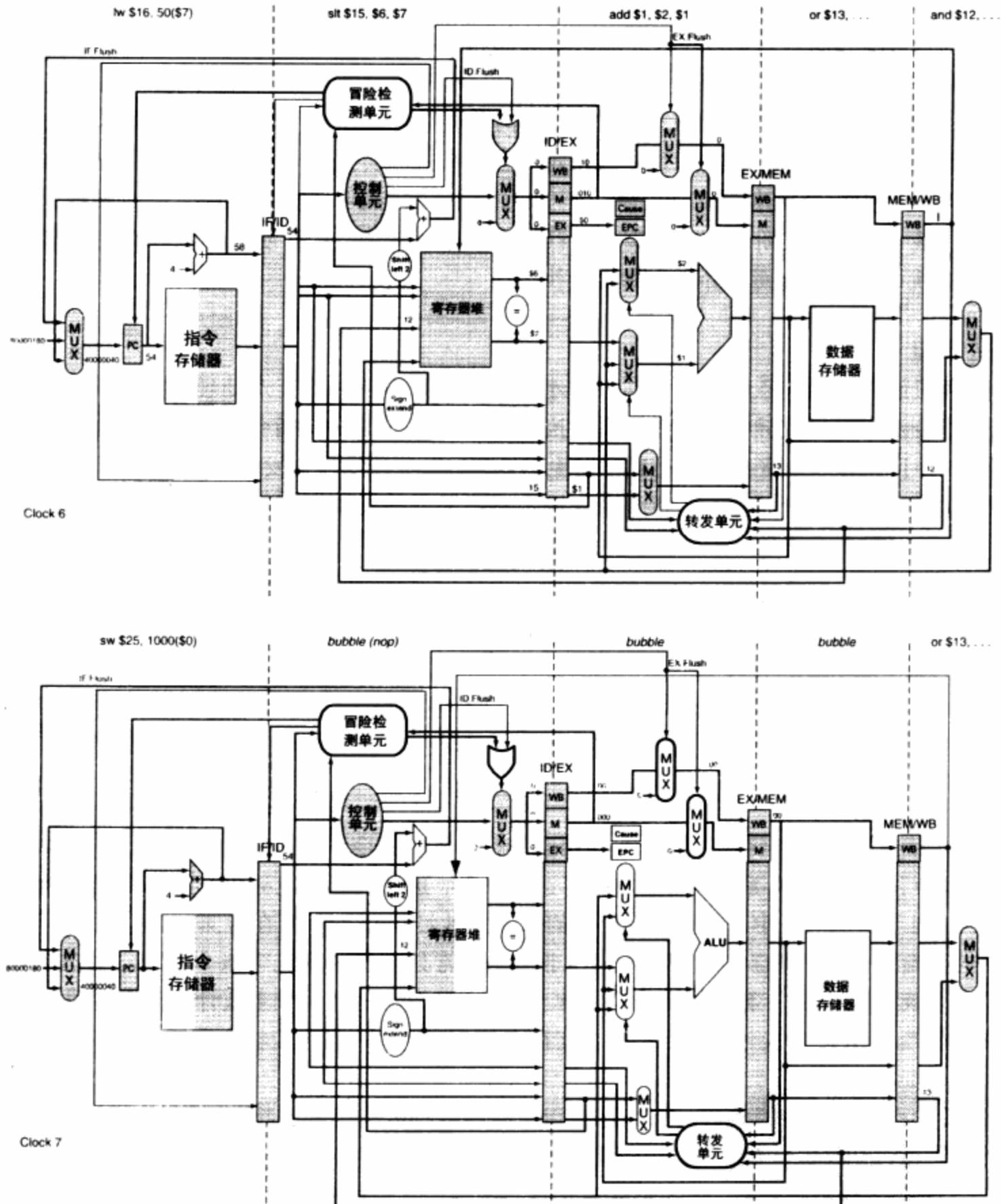


图 6-43 由于 add 指令算术溢出导致的异常

[溢出在第六个时钟周期的 EX 阶段被检测到，因此将 add 后面的指令的地址($4C + 4 = 50_{hex}$)保存到 EPC 寄存器。溢出导致在该周期后所有的 Flush 信号都设置为 1，并清除 add 的控制值(置为 0)。时钟周期 7 将流水线中的指令转换成 nop，同时开始从地址 $4000\ 0040_{hex}$ 预取异常处理例程的第一条指令 `sw $25, 1000($0)`。需要注意的是位于指令 and 前的 add 指令和 or 指令仍然会执行完毕。虽然图中没有画出 ALU 溢出信号，但它也是控制单元的一个输入]

在任一个时钟周期流水线中都有五条活动的指令，因此确定导致异常的指令是我们要解决的

问题。而且，在一个时钟周期里可能同时发生多个异常。为了容易确定哪种异常应该优先服务，通常的解决方法是对异常划分优先级；这种策略在流水线处理器上也能够很好的工作。在大多数MIPS实现中，硬件对异常进行排序从而使得最先发生异常的指令被中断。

I/O设备请求与硬件错误并不与特定的指令相关，因此它们在中断流水线的实现方法上具有一定的灵活性。因此，将此机制用于其他异常也同样有效。

EPC捕捉被中断的指令的地址，同时MIPS的原因寄存器在一个时钟周期内记录下所有可能的异常，因此异常处理软件必须将指令和特定的异常相关联。关联它们的一个重要的线索是了解一类异常可能在哪一个流水线阶段发生。例如，未定义的指令异常会在ID阶段被发现，而调用操作系统异常发生在EX阶段。异常集中保存在原因寄存器中，从而使得一旦前面的异常被服务之后，硬件就可以按照后面的异常中断指令执行。

软硬件接口

计算机硬件与操作系统必须协同工作才可能按照我们期望的方式处理异常。硬件协定一般暂停指令流中的导致异常的指令，同时执行完该指令前的所有指令，而清除该指令后的所有指令，并且设置一个寄存器描述异常发生的原因，保存导致异常发生的指令的地址，然后跳转到预先确定的地址开始执行。操作系统协定查看异常的发生原因并采取相应的操作。对于一个没有定义的指令异常、硬件错误异常或者算术溢出异常，操作系统通常杀掉执行的程序并返回原因描述。对于I/O设备请求异常或者操作系统服务调用异常，操作系统保存程序的当前状态，执行期望的任务，然后重新载入程序继续运行。对于这种I/O设备请求的情况，我们通常也选择暂时离开当前发送I/O请求的任务而转到另一个任务，因为在I/O尚未完成时该请求任务很有可能不能继续执行。这就是为什么保存每个任务状态是如此重要的原因了。最重要同时也是最常用的异常是处理分页失误和TLB异常的情况；第7章详细叙述了这些异常和它们的处理方法。

在流水线计算机中要将每一个异常与相应导致异常的指令联系起来的难度很大，因此一些计算机设计者们在一些非关键的应用中放松了这种要求，这种机器一般称为具有**非精确中断或非精确异常**^①。在上面的例子中，尽管导致异常的指令的地址是4C_{hex}，但在检测到异常后的下一个时钟周期的开始时PC的值通常为58_{hex}。具有非精确异常处理的计算机可能会将58_{hex}放入EPC中，而让操作系统确定是哪一条指令导致了异常。MIPS以及当前的大量主流计算机都提供**精确中断或精确异常**^②。（这其中的一个原因是为了支持虚拟内存，这一点我们将在第7章看到。）

自测

MIPS的设计者们希望整数乘法与除法指令与其他整数指令并行执行。由于乘法与除法需要多个时钟周期，一组学生为能否实现精确异常而争辩。以下论点中哪些是完全正确的？

1. 有可能实现精确异常，因为乘除能在其后的指令之后提交异常。
2. 由于乘和除一旦开始便不能提交异常，因而对所有异常的计时显然是精确的。所以实现精确异常毫无意义。

① 非精确中断(imprecise interrupt) 也称为**非精确异常**(imprecise exception)。在基于流水线的计算机中，中断和异常是不与直接产生中断或异常的指令精确相关联的。

② 精确中断(precise interrupt) 也称为**精确异常**(precise exception)。在流水线计算机中，异常或中断与其产生者始终能精确地关联起来。

3. 乘和除能否提交异常无关紧要。当其他指令提交异常时乘除仍就能执行而不结束，这一事实使实现精确异常成为可能。

4. 尽管乘或除仍将执行，它们被确保即刻结束。当这发生时，所有为乘除之后的指令提交的异常都将是精确的。

6.9 高级话题：如何提高性能

首先提醒你，6.9节和6.10节将简单介绍流水线中一些高级而吸引人的问题。如果你想了解更多的细节，请参阅相关的高级参考书：《计算机系统结构：量化研究方法（第3版）》，本书接下来十几页的内容在该书中将用超过200页的篇幅予以论述。

流水线开发了指令流内部隐藏着的并行性。这种并行性被称为**指令级并行^①**（ILP）。主要存在着两种增加指令级并行的策略。第一种是增加流水线的级数以能将更多指令的执行过程重叠起来。使用前面的洗衣店的类比实例，假设洗衣过程比其他的过程相比所消耗的时间要更长，我们可以将传统的洗衣机分成三个机器分别进行清洗、漂洗和旋转洗涤。这样原来的4级流水线就变成了6级流水线。无论对处理器还是洗衣机，为了获得最大的加速比，我们需要调整流水线中的其他步骤，从而使各个步骤具有相同的长度，这点适用于洗衣店，同样适用于处理器。由于更多的操作在并行执行，所以并行度的开发程度更高。由于时钟周期的缩短，性能有了显著的提高。

第二种策略是设计更多的内部元件从而能够在一个流水线的每级中发射多条指令。通常我们称这种技术为**多发射^②**。例如，一个多发射的洗衣店可能用3台洗衣机和3台烘干机来代替我们家用的一台洗衣机和一台烘干机。当然这时你需要招募更多的助手，从而在相同时间内能整理并保存以前三倍的衣服。接下来的工作就是保证机器一直工作并将负载传递给下一个流水线阶段。

在一个流水线阶段中发射多条指令使得指令的执行效率能够超过时钟频率，也就是说，达到CPI值小于1。有时我们使用它的倒数IPC，即一个周期内完成的指令数（instructions per clock cycle），特别是CPI值小于1的时候！因此，一个运行于6GHz的四路的多发射微处理器在峰值情况下每秒可以执行240亿条指令，也就可以达到最理想的0.25的CPI值，即IPC为4。如果假设它是5级流水线的话，任何一个时刻处理器内部都同时存在着20条指令。当今的高端微处理器试图在每周期发射3~8条指令。但如果指令流中的指令间存在依赖关系或不符合特定的标准，那么将面临很多限制条件。

有两种实现多发射的方法，它们之间的主要区别在于编译器和硬件之间的分工。由于分工主要在于某些决定是在编译期（即编译时）决定还是在运行期（即执行时）动态决定，所以这两种方法有时称为**静态多发射^③**和**动态多发射^④**。我们将会看到，这两种方法都有别名，这些其他叫法虽然不够准确，使用却很广泛。

对于多发射流水线而言有两个主要任务：

1) 将指令打包到**发射槽^⑤**中：处理器如何确定每个周期能发射多少条以及何种指令呢？在大多数基于静态发射的处理器中，这个工作至少有一部分是由编译器完成的；在动态发射处理器

① 指令级并行(instruction-level parallelism) 指令之间存在的并行性。

② 多发射(multiple issue) 一种单个周期内发射多条指令的技术。

③ 静态多发射(static multiple issue) 实现处理器多发射的一种方法，决策主要在运行前由编译器静态地做出。

④ 动态多发射(dynamic multiple issue) 实现处理器多发射的一种方法，许多决策都在运行时由处理器硬件做出。

⑤ 发射槽(issue slots) 每个时钟内能够发射指令的那些具体位置，可以类比为短跑比赛中各道的起点。

中，尽管编译器通常都要尝试合适的顺序调度指令以达到较好的指令序列来提高发射速率，打包的工作主要仍是由处理器动态完成的。

2) 处理数据冒险和控制冒险：在静态发射处理器中，部分或者所有的数据和控制冒险及其后序处理都是静态地由编译器完成的。与此相反，大多数动态发射处理器都会在运行期通过硬件技巧处理至少几种冒险的情形。

虽然我把这两种称为不同的方法，但是实际中某种方法的具体技巧往往借鉴于另一种，因而没有一个可以称得上完全纯粹。

6.9.1 推测

推测技术是开发指令级并行度(ILP)的一个重要方法。编译器和处理器利用推测技术[⊖]对要执行的指令的属性进行“猜测”，这样才能使得可能依赖于被推测指令的那些指令提早执行。比如，我们可以推测一条分支指令的输出结果，这样我们才能提早执行本分支后面的指令。或者我们也可以推测一条 load 指令和它之前的那条 store 指令引用的不是同一个地址，这样 load 指令就可能可以在 store 指令执行前先得到执行。推测技术的难点在于它并不总是对的。所以，任何推测机制都必须包含推测正确与否的监测方法和回滚机制，以消除错误推测所带来的影响。在任何采用推测技术的处理器中，实现恢复机制都会增添额外的复杂度。

推测可以由编译器或硬件来做。例如，编译器可以用推测来重排指令，将一条指令移出分支或将读写调换次序。处理器硬件能用本节后面我们将讨论的技术在运行时执行同样的转换。

不同推测机制的恢复方式都大相径庭。软件的推测恢复机制中，通常编译器会插入检查推测精度的额外指令，并提供一个修复程序以备错误推测发生时使用。而在硬件的推测恢复机制中，处理器通常将推测得到的结果保留在缓存中，直至对推测的正确性做出判断。如果推测正确，那么将完成推测后执行的指令，或者将它们的结果写回寄存器，或者写回内存。如果推测错误，硬件将清除缓存中的内容，并重新开始执行正确的指令序列。

推测技术引入了另一个可能的问题：对某些指令推测执行可能会引发异常，这些异常之前并不存在。比如：一条 load 指令由于推测而被触发开始执行，但是当推测错误时它引用的是非法地址。结果就是一个本不该发生的异常发生了。如果这条 load 指令不是推测执行的，这个异常本是应该发生的，那么情况就更复杂了！对于基于编译器的推测机制而言，可以通过加入特殊的推测支持语句以忽略这些异常，直至有足够的信息能判断是否该执行该语句。对于基于硬件的推测机制而言，仅需将这些异常暂时缓存下来，直至推测结果已知，并且导致这个异常发生的语句要完成执行的时候再触发异常；此时将开始正常的异常处理过程。

如果正确使用推测机制，性能会得到提升，而不慎使用会对性能产生负面影响。所以人们投入了大量精力讨论何时应当采用推测执行机制。在下面的几节中，我们将观察静态发射和动态发射的推测执行技术。

6.9.2 静态多发射

静态多发射处理器利用编译器的辅助对指令进行打包和处理各种冒险情况。在静态发射处理器中，你可以把某个周期内发射的指令看成一条含有多个操作的长指令，我们称之为发射包[⊖]。这种观点不仅仅是一种类比。由于静态多发射处理器通常会限制每个周期内能发射的指令其类型，所以将发射包想象为一条在某些固定字段内包含多个操作的长指令是很合理的。这种观点直

[⊖] 推测技术(speculation) 一种编译器或者处理器猜测指令结果以消除依赖关系并继续执行后续指令的技术。

[⊖] 发射包(issue packet) 同一个周期内发射的指令组成的集合；它可能由编译器静态地生成，也有可能被处理器动态生成。

接引出了此类方法的最初名字：超长指令字(Very Long Instruction Word, VLIW)。Intel IA-64 采用了这种方法，Intel 自己称之为显式并行性处理机(Explicitly Parallel Instruction Computer, EPIC)。安腾(Itanium)和安腾 2 处理器分别在 2000 年和 2002 年问世，它们是 IA-64 架构的第一个具体实现。

大多数静态发射处理器同样依赖于编译器来负责一部分处理数据和处理控制冒险情况。编译器的任务可能包括通过静态分支推测和代码调度减少或防止冒险现象。

在介绍如何将这些技术使用在更高级处理器之前，让我们分析一个 MIPS 静态发射处理器。通过这个简单例子我们分析一些基本观点，然后讨论 Intel IA-64 结构的特点。

一个实例：MIPS ISA 指令集下的静态多发射

为了让大家对静态多发射产生一个感性认识，我们现在来考虑一个简单的 2 发射 MIPS 处理器，同时发射的两条指令可以是一条整数 ALU 操作或分支指令，另一条可以是 load 或 store。这类似于一些嵌入式 MIPS 处理器。每周期发射两条指令需要每周期预取并译码 64 位的指令。在许多静态多发射处理器以及几乎所有的 VLIW 处理器中，都会对同时被发射的两条指令的放置情况进行限制，以简化译码和指令发射。所以我们要求两条指令成对放在一个 64 位对齐的内存区域中，并将 ALU 指令或分支指令放在前面。此外，如果这一对指令中的任一条不能被使用，那么我们用 nop 指令替代。这样指令总是成对发射，可能在一个时间片内有一个 nop 指令。图 6-44 描述了指令成对进入流水线的情况。

指令类型	流水线阶段							
	IF	ID	EX	MEM	WB			
ALU 或分支指令	IF	ID	EX	MEM	WB			
load 或 store 指令	IF	ID	EX	MEM	WB			
ALU 或分支指令		IF	ID	EX	MEM	WB		
load 或 store 指令		IF	ID	EX	MEM	WB		
ALU 或分支指令			IF	ID	EX	MEM	WB	
load 或 store 指令			IF	ID	EX	MEM	WB	
ALU 或分支指令				IF	ID	EX	MEM	WB
load 或 store 指令				IF	ID	EX	MEM	WB

图 6-44 静态二发射流水线的处理过程

[一条 ALU 指令和一条数据传输指令在同一个时间发射。这里我们假设使用了与单发射相同的五级流水线结构。尽管这并不一定必要，但仍有一定的优点。特别是，在流水线末端进行寄存器堆写操作降低了处理异常和实现精确异常模型的难度。在多发射处理器中，实现它们的难度越来越大]

静态多发射处理器之间的不同点在于它们处理潜在的数据与控制冒险的方式不同。在某些设计中，完全由编译器负责去除所有的冒险，调度代码并插入 nop 指令，使得代码执行过程中不需要检测冒险条件，也不需要实现基于硬件的流水线阻塞操作。而在另一些设计中，通过硬件检测数据冒险并在连续的两个指令匣之间插入停顿，而编译器则负责消除所有指令对内部的依赖关系。即便这样，由于单个指令的依赖关系所产生的冒险通常会迫使整个指令匣停顿下来。软件(编译器)应当处理所有的冒险情况，还是仅仅试图降低不同发射槽之间的冒险性？我们再次从“包含多个操作的单个长指令”的观点来看待问题。在下面的例子中我们假设使用第二种方法。

为了并行提交 ALU 和数据传输操作，第一个需要增加的硬件(除了通常的冒险检测和阻塞逻辑外)是额外的寄存器堆端口(参见图 6-45)。在一个时钟周期内可能需要为 ALU 操作读两个寄

存器并同时为 store 指令读两个寄存器，还有为 ALU 操作写一个端口和为 load 指令写一个端口。由于在执行 ALU 操作时 ALU 被占用，因此需要一个独立的加法器来计算数据传输所需的有效地址。如果没有这些额外的资源，我们的二发射流水线就会由于结构冒险而被阻塞。

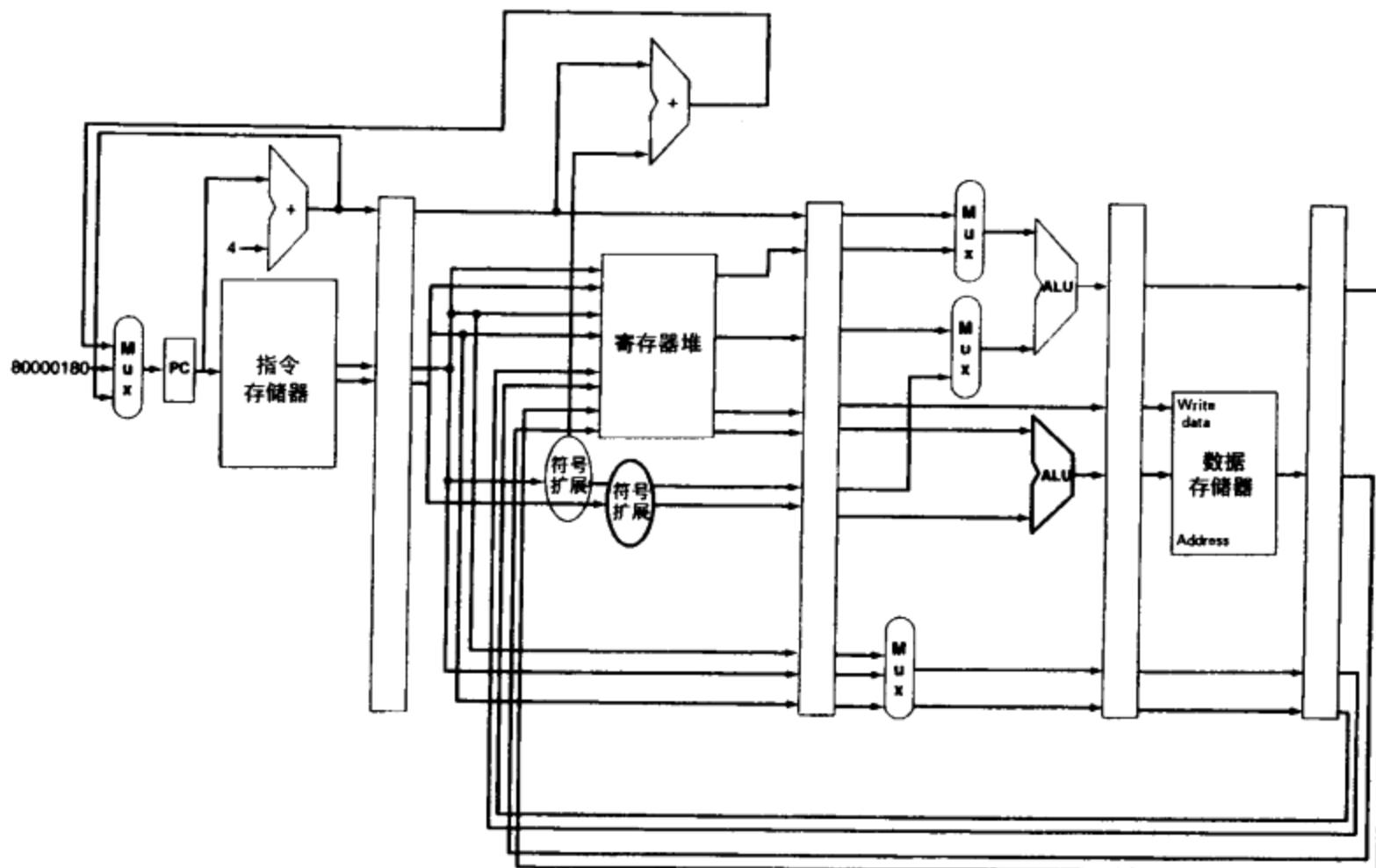


图 6-45 一个静态二发射数据通路

[由于要完成二发射而引入的额外部分在图中以粗线表示，其中包括：有指令内存中的另 32 位、寄存器堆的两个新的读端口和一个新的写端口以及一个新的 ALU。假设下面的 ALU 计算数据传输的地址，上面的 ALU 处理其他问题]

很明显，这个二发射的处理器有潜力将性能提高为原来的两倍。然而这意味着同时被执行的指令数是原来的两倍，这会增加潜在的由数据和控制冒险导致的性能损失。比如在我们的五级流水线中，load 指令的使用延时为一个时钟周期，这可能导致使用 load 执行结果的某条指令阻塞。在二发射五级流水线中，load 指令的结果不能为下一个周期所使用。这意味着下面的两条指令如果要使用 load 的结果，则都必须阻塞。此外，ALU 指令在简单的五级流水线中没有使用延时但现在却有一个周期使用延时，所以其结果不能供与其配对的 load 或 store 指令所使用。为了更有效地使用多发射处理器开发程序并行度，我们需要更强大的编译器或更高级的硬件调度技巧，在静态多发射机器中，这属于编译器的工作。

例题 简单的多发射代码调度

以下的循环代码在 MIPS 二发射流水线中是如何调度的？

```

Loop:    lw      $t0, 0($s1)      # $t0=array element
         addu   $t0,$t0,$s2      # add scalar in $s2
         sw      $t0, 0($s1)      # store result
         addi   $s1,$s1,-4       # decrement pointer
         bne   $s1,$zero,Loop    # branch $s1!=0

```

为尽量避免流水线阻塞请重新排列这些指令。假设分支是可预测的，所以控制冒险可由硬件处理。

解 前三条指令具有数据相关性，后两条指令也具有数据相关性。图 6-46 给出了对这些指令的最佳调度方法。需要注意的是只有一对指令有两个发射槽，每一次循环需要 4 个时钟周期；并且在第 4 个时钟周期中执行 5 条指令。我们实际达到的 CPI 为 0.8，而最好情况下的 CPI 值为 0.5，也就是说 IPC 分别为 1.25 和 2.0，前者的结果是令人失望的。我们可以看到在计算 CPI 或 IPC 的时候我们未计人任何有用的 nop 指令，若计人 nop 指令，虽然在 CPI 值上会有所改观，但是真实性能却不变。■

	ALU 或分支指令	数据传输指令	时钟周期
Loop:		lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4		2
	addu \$t0, \$t0, \$s2		3
	bne \$s1, \$zero, Loop	sw \$t0, 0(\$s1)	4

图 6-46 二发射 MIPS 流水线上的代码调度

[空的发射槽对应 nop 指令]

循环展开^①是一种开发程序中循环部分性能的重要的编译技术，通过循环展开我们将得到多个循环体，因而可以将不存在于一次循环中的指令一起调度。

例题 多发射流水线的循环展开

看看在上例中循环展开和调度是怎样工作的。为简单起见，假定循环索引是 4 的倍数。

解 为了不产生延迟地调度循环，我们需要产生四个循环体的拷贝。在展开以后，循环就包含了指令 lw、add、sw、addi 和 bne 中每个指令的 4 个副本。展开后的代码调度方法如图 6-47 所示。

在循环展开的过程中，编译器将引入一些额外的寄存器 (\$t1, \$t2, \$t3)。这个过程称为**寄存器重命名^②**，是用来消除非真实的数据依赖关系而引入的。这些依赖关系虽然不是真正的数据依赖关系，但是仍会导致潜在的冒险并进一步阻碍编译器使其不能灵活地调度代码。考虑一下如果我们只使用 \$t0 进行展开会怎样？这样的话将会有不断的 lw \$t0, 0(\$\$s1), addu \$t0, \$t0, \$s2 和 sw \$t0, 4(\$s1) 的指令串出现，虽然这些指令都用到了 \$t0，但是它们并不两两相互依赖——没有数据从某对指令流向另一对指令。我们称这种情况为**反依赖关系或者名字依赖关系^③**，它们不是真正的数据依赖关系。

在循环展开的过程中对寄存器进行重命名可以使编译器能够移动这些相互独立的指令，以能更有效地进行代码调度。重命名机制在保留真实依赖关系的同时消除了名字依赖关系。

我们注意到循环中的 14 条指令中的 12 条都是成对执行的。四次循环会消耗 8 个周期，也就是每次循环 2 个周期，这时 CPI 值为 $8/14 = 0.57$ 。通过循环展开和代码调度，二发射使我们得到了两倍的改善，其中一部分归功于循环控制指令的消除，一部分归功于二发射机制。我们为性能提升所付出的代价是使用了 4 个临时寄存器，而原来我们只使用了 1 个，同时我们的代码大小显

- ① 循环展开(loop unrolling) 一种增进访问向量数组的循环的性能的技术，通过展开得到循环体的多个副本，从而使得原来不存在于同一次循环的指令能一起调度。
- ② 寄存器重命名(register renaming) 通过编译器或者硬件手段，将寄存器重新命名，以消除反向依赖。
- ③ 反依赖关系(antidependence) 又称名字依赖关系(name dependence)，由于重用名字(通常是寄存器名)引入的一种执行次序，与两个变量使用同一个数值的真实依赖不同。

著增加了。

	ALU 或分支指令	数据传输指令	时钟周期
Loop:	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)	1
		lw \$t1, 12(\$s1)	2
	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1)	3
	addu \$t1, \$t1, \$s2	lw \$t3, 4(\$s1)	4
	addu \$t2, \$t2, \$s2	sw \$t0, 16(\$s1)	5
	addu \$t3, \$t3, \$s2	sw \$t1, 12(\$s1)	6
		sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

图 6-47 图 6-46 中代码在静态二发射 MIPS 流水线上的循环展开和调度代码的方法

[空格表示 nop 指令。由于第一对指令将 \$s1 减去 16，所以载入的地址是 \$s1 的原始值，然后再用这个地址依次减去 4、8 和 12]

Intel IA-64 架构

IA-64 架构是一个寄存器-寄存器型的 RISC 风格的指令集，它类似于 64 位 MIPS 架构（称为 MIPS-64），但却独有特性：它支持编译器驱动下显式地开发指令级并行性。Intel 称之为显式并行指令计算机（Explicitly Parallel Instruction Computer, EPIC）。IA-64 和 MIPS 架构之间的主要区别在于：

1) IA-64 比 MIPS 有更多的寄存器：它含有 128 个整数寄存器和 128 个浮点寄存器，8 个专用于分支的寄存器和 64 个 1 位的谓词寄存器。另外，IA-64 支持类似原来 Berkeley RISC 和 Sun SPARC 架构风格的寄存器窗口。

2) 在 IA-64 架构下，指令被组织在指令包中，指令包的格式固定，其中明确规定了指令间的依赖关系。

3) IA-64 架构包含特殊的指令和设计，以支持推测执行和消除分支，这样便提高了可用的指令级并行度。

IA-64 架构的设计目标是获得 VLIW 中单条指令中多余操作之间的隐含并行性以及操作域的固定格式，同时又保持了比通常的 VLIW 更高的灵活性。在 IA-64 架构中，达到这种灵活性是通过两个概念来实现的：指令组与指令包。

一个指令组^①指的是一个连续的、相互之间不存在寄存器级数据依赖关系的指令序列。一个指令组内部的所有指令完全可以并行执行，只要我们有足够的硬件并且可以保证内存操作的依赖关系得到保证。一个指令组的长度任意，但是编译器必须显式地标明各个指令组之间的边界。这个边界可以通过在指令组之间加入一个停止标记^②来实现。

在 IA-64 中，将指令编码并形成指令包(bundle)，每个指令包长度为 128 位。每一个指令包由一个 5 位长的模板字段和三个 41 位长的指令组成。为了简化指令译码和发射过程，模板字段指明了三条指令各需要五类功能部件中的哪一类。这五类功能部件包括：整数 ALU，非整数 ALU（包括移位和多媒体操作），访存部件，浮点功能部件，以及分支功能部件。

① 指令组(instruction group) 在 IA-64 中一系列连续而相互之间不存在寄存器数据依赖关系的指令序列。

② 停止标记(stop) 在相互无关的指令与相关的指令之间的一个显式的标志位。

五位长的模板字段不仅描述了指令包中的三条操作所需要的部件类型，同样还表明了停止标记的位置。所以指令包格式只能表示出一部分指令和停止标记的组合。

为了开发更多的指令级并行性，IA-64 提供了许多谓词和推测执行的支持(参考本部分后文的细节)。谓词化[○]是一种通过将指令的执行与否与谓词联系起来以消除分支指令的技术，而不是将指令是否执行与某条分支直接关联起来。我们前面已经介绍了，分支指令会限制指令级并行度的开发，因为它会限制代码的走向。循环展开可以很好地消除循环分支，但是循环体内部的分支，比如一个循环体中包含一个 *if-then-else* 语句，它们是不能通过展开而消除掉的。然而谓词化却可以消除这类分支，因而就能更灵活地开发并行度。

比如，我们有下列的代码序列：

```
if (p) lstatement 1l else lstatement 2l
```

使用通常的编译方法，这段代码的编译结果将含有两条分支指令：一条用于跳转到 *else* 部分，而另一条用于在 *statement 1* 之后跳过 *else* 部分。通过使用谓词化，这段代码将被编译为：

```
(p) statement 1  
(~p) statement 2
```

其中(*condition*)部分表明其所指的代码仅在 *condition* 为真时才被执行，否则这条代码将被转化为一个 *nop*。很明显，谓词化可以用来实现另一种消除分支的方法：推测执行。

IA-64 架构为谓词化提供了丰富支持：IA-64 中几乎所有的指令都能通过与某个谓词寄存器(其可用指令字段的低六位标识)相关联而被谓词化。谓词化的一个可能的结果就是一个条件分支指令可被转化为一个被谓词寄存器限制的绝对分支指令！

IA-64 指令集是所有通过编译器开发指令集并行度的指令集中最复杂的。Intel Itanium 与 Itanium2 处理器实现了 IA-64 指令集。图 6-48 是这些处理器的概览。

处理器	每周期最多发射的指令数	功能部件	每周期最多执行的操作数	最高频率	晶体管数(百万)	功耗(瓦)	SPEC int2000	SPEC fp2000
Itanium	6	4 整数/多媒体 2 访存 3 分支 2 浮点	9	0.8 GHz	25	130	379	701
Itanium 2	6	6 整数/多媒体 4 访存 3 分支 2 浮点	11	1.5 GHz	221	130	810	1427

图 6-48 Itanium 与 Itanium2 处理器概览，它们是 IA-64 架构的 Intel 最早的两个实现

[除了包含更多的功能部件并拥有更高的主频之外，Itanium2 还包含片上的三级高速缓存，而 Itanium 只有非片上的三级高速缓存]

细节：IA-64 架构中对推测执行的支持还包含对控制推测执行的独立支持，它可以推迟推测执行的指令(包含访存指令)所产生的异常，并支持 load 的推测执行。通过将推测执行的 load 指令标记 **poison 标签**[○]，可以推迟该指令所产生的异常。当一个标记了 poison 标签的指令所产生的结果被另一条指令所使用时，该指令也会被标记 poison 标签。当软件

○ 谓词化(predication) 将指令是否执行与谓词相关联，而不是与某条分支指令相关联的技术。

○ Poison 标签(poison) 标识推测执行的 load 指令产生异常，或者标识使用了标有 poison 标签指令的指令。

探测到执行过程已经不再是推测的情况下，就可以探测被标记 poison 标签指令的结果了。

在 IA-64 中，我们同样可以推测地进行访存操作：将 load 指令提前到 store 指令之前执行，而前者可能与后者之间存在依赖关系。这可以通过使用一条高阶 load 指令来完成。高阶 load 指令^①在正常运行之余，会通过一个特殊的表来跟踪处理器所进行的访存操作。所有的 store 指令都会在表中与之对应的项进行标记。这样我们还需要另一条指令，在 load 指令变为非推测状态之后对表中的项进行检查。如果发生了冲突，那么负责检测的指令需要设定一个恢复程序以重新执行这条 load 指令和其他所有依赖于它的指令，之后才能继续执行；如果未发生与之冲突的 store 指令，那么将清除表项以标识 load 指令不再是推测执行状态。

6.9.3 动态多发射处理器

动态多发射处理器通常也称为超标量^②处理器，或简称超标量。在最简单的超标量处理器中，指令顺序发射，每个周期处理器决定是发射 1 条或多条指令，或者干脆不发射任何指令。很显然，在这种处理器上要达到较好的性能必须依赖编译器以调度各条指令，错开依赖关系以求达到较高的发射速率。即便使用编译器来完成指令调度，简单的超标量与 VLIW 处理器之间仍有显著不同：代码不管是否经过了调度，都须通过硬件保证执行的正确性。进一步说，编译得到的代码应当始终能正确执行，而与发射速率以及处理器的流水线结构无关。在某些 VLIW 的设计中情况并非如此，当把代码从一台机器拿到另一台机器上运行时，可能还需要重新编译；在其他一些静态发射处理器上，代码可以在不同的处理器实现上无缝执行，但有时效果可能很差以至于不得不重新编译。

许多超标量将动态发射的框架拓展为动态流水线调度^③。在动态流水线调度策略中，下面一个周期内将执行的指令是动态选择出来的，指令选择的约束条件仅为冒险和依赖关系。让我们从一个简单的数据冒险的例子出发来进行说明。考虑下面的代码：

```
lw      $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slti   $t5, $s4, 20
```

虽然 sub 指令已经满足执行的条件，但是它必须等待 lw 和 addu 指令先结束才行，如果内存很慢那么将意味着 sub 会等待很多个周期。（第 7 章解释了高速缓存系统，以及为什么有时访存操作会很慢的原因。）动态流水线调度可以部分或者完全地避免这种冒险现象。

动态流水线调度

当等待解决阻塞时，动态流水线调度会到以前阻塞的地方后寻找指令执行。通常流水线分为 3 个主要单元：一个取指令和指令发射单元，多个功能单元（在 2004 年高端设计中有 10 个或更多）和一个指令完成单元^④。图 6-49 描述了这个模型。第一个单元用于取指令，将指令译码，并将它们送到相应功能单元执行。每个功能单元都有自己的缓冲器，称作保留站^⑤，它们用来保存操作数和操作指令。（下一节中我们将讨论许多当前存储器正使用的替代保留站的一种选择）。当缓冲中包含了所有的操作数，并且功能单元已经就绪，结果就被计算出来。当完成结果时，它就被

- ① 高阶 load 指令(advanced load) 在 IA-64 中能够使加载失效的一种支持检查内存操作别名情况的推测型 load 指令。
- ② 超标量(superscalar) 一种高级流水线技术，可以使每个周期处理器能执行的指令数多于一条。
- ③ 动态流水线调度(dynamic pipeline scheduling) 对指令进行重新排序以避免处理器阻塞的硬件支持。
- ④ 指令完成单元(commit unit) 在动态或乱序执行的流水线中决定何时可以将操作执行结果写回程序员可见寄存器或者内存中的功能部件。
- ⑤ 保留站(reservation station) 功能单元中的一个用于暂时存储操作数和操作类型的缓冲器。

发送到等待特殊结果的储存站以及指令完成单元，而指令完成单元确定何时能够安全地将结果放入寄存器堆或内存(对 store 指令)中。指令完成单元中的缓冲器通常称为重排序缓冲区^②，它也可以用来提供操作数，其工作方式大致类似于旁路逻辑在静态调度流水线中的工作方式。一旦结果写回寄存器堆，使可以从寄存器堆中直接取得操作数，就像一般流水线中取得操作数的方式一样。

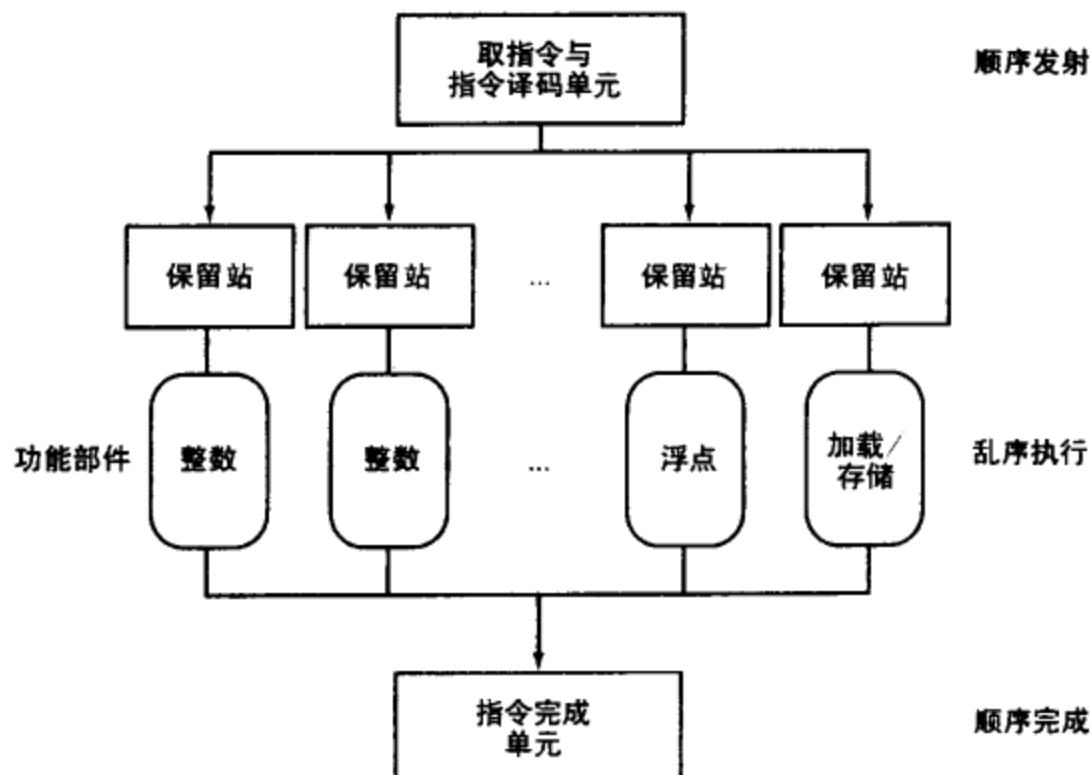


图 6-49 动态调度流水线中的三个主要单元

[最后一步，即用于更新机器状态的步骤通常也被称为指令退出或指令毕业]

将操作数缓存在保留站中并且将结果放在重排序缓冲器中的方式，其实提供了一种形式的寄存器重命名机制，这类似于我们前面 6.9.2 节第 2 道例题的循环展开例子中编译器所做的工作。为了在概念上分析其工作方式，考虑下面的几步：

1) 当指令发射时，如果它的操作数在寄存器堆中或重排序缓冲区中，那么操作数可以立即拷贝到储存站中，在除非所有操作数和执行单元可用时它们都会被缓冲在那里。对于被发射的指令而言，此时其操作数所对应的寄存器复本已不需要，所以如果发生了对某个源寄存器的写请求，该寄存器可以被覆写。

2) 如果操作数不存在于寄存器堆或重排序缓冲区中，那么它一定会在某个时刻被某个功能单元以计算结果的形式输出。硬件将帮助定位产生这个结果的功能单元。当该单元最终完成执行的时候，结果将被直接从功能单元旁路寄存器中复制到保留站中。

这几步可以有效地利用重排序缓冲区和保留站以实现寄存器重命名。

从概念上讲，我们可以把动态调度流水线想象为对程序数据流结构的分析过程，就像我们在第 2 章中在编译器例子里面分析数据流那样。处理器在不违背程序原有的数据流顺序的前提下以某种顺序执行各条指令。为了使程序表现得像是在一条简单的顺序流水线上执行的情况一样，取指令和译码单元必须能够顺序发射指令，以记录程序中的依赖关系，而指令完成单元也必须按照程序顺序将结果写回寄存器和内存。这种保守的方案称为顺序完成。所以当异常发生时，计算机可以指向最

^② 重排序缓冲区(reorder buffer) 在动态调度的处理器中用于暂时存储执行结果的缓冲器，等到可以确认能够安全地将结果写回寄存器或内存中时再将其写回。

后执行的那条指令，而只有这条导致异常的指令之前的所有指令对寄存器状态进行了修改。虽然处理器的前端(取指和发射)和后端(完成)按照顺序操作指令，各功能单元可以随时开始执行过程，只要能获得所需数据。当前所有的动态调度流水线都采用顺序完成，虽然并不一定非得这样。

动态调度通常与基于硬件的推测机制相结合共同工作，特别是分支指令。通过对分支指令的跳转方向进行推测，动态调度处理器可以在推测方向上进行取指和执行。由于指令是顺序完成^①的，我们将在分支指令以及所有推测执行的指令完成之前得知推测的正确与否。一个推测执行的动态调度流水线同样可以对 load 指令的目的地址进行推测，以实现 load 指令和 store 指令的重排序以提高性能，同时利用重排序机制对不正确的推测进行补救。在下一节中我们将研究 Pentium 4 处理器的动态调度流水线设计与推测机制。

细节：指令完成单元控制寄存器堆和内存的更新。一些动态调度处理器在执行过程中即刻更新寄存器，而使用额外寄存器来实现重命名功能并保存寄存器的早期备份直到更新寄存器的指令不再是预测性的。其他处理器通常把结果缓存在称为重排序缓冲器的结构中，而实际的寄存器文件的更新作为指令完成的一部分在后来更新。在指令完成之前，往内存的写必须缓存在存储缓冲器(见第 7 章)或重排序缓冲器。指令完成单元允许存操作从缓冲器往内存写，当缓冲器有合法地址和数据时，或者存储不再依赖预测分支。

细节：完成单元控制更新寄存器堆和内存。一些动态调度流水线处理器在执行时立即更新寄存器堆。另外一些计算机则保留寄存器堆的一个备份，而实际对寄存器堆的更新是作为提交的一部分在后来完成的。存入内存前必须被缓冲至到完成时写入缓冲区(store buffer)(参见第 7 章)或重排序缓冲区中。当缓冲区中保存了有效的地址和数据，并且 store 指令不再依赖于推测的分支时，完成单元允许 store 指令从缓冲区向内存进行写入。

细节：非阻塞缓存(nonblocking cache)在缓存访问失效时能够继续提供缓存访问服务(参见第 7 章)，内存访问能够从非阻塞的缓存受益。为了使指令在缓存缺失时能够执行，乱序执行^②的处理器需要非阻塞的缓存支持。

软硬件接口

既然编译器同样可以依据数据依赖关系调度代码，那么你可能会问，为什么还需要超标量处理器来进行动态调度？这里面主要有三个原因。首先，并不是所有的阻塞都是事先知道的或者确定的。尤其是高速缓存缺失会导致不可预测阻塞(参见第 7 章)。动态调度使得处理器将这些阻塞通过调度并执行其他无关的指令的方式得以掩盖。

第二，如果处理器采用动态分支预测预测分支的结果，那么由于这些信息依赖于预测和分支指令的真实执行状况，而编译期无法得知指令的正确顺序。采用动态预测机制而不采用动态调度机制，会极大程度地限制可以开发的指令级并行度。

第三，由于流水线延时和发射宽度依处理器具体实现而大相径庭，所以编译代码顺序的最佳方法也并不固定。比如，调度一系列存在着相互依赖关系的指令的具体方式将与发射宽度和延时存在着密切关系。流水线的结构同样会影响循环展开的深度，才能避免可能的阻塞并指导编译器进行寄存器重命名的过程。动态调度使得硬件将这些细节屏蔽起来。因此，用

① 顺序完成(in-order commit) 流水线中的指令按照取指的顺序对程序员可见状态进行修改以完成指令的执行过程。

② 乱序执行(out-of-order execution) 在基于流水线的执行过程中，一条由于某种原因阻塞的指令不会造成后面的指令等待的过程。

户和软件发行商就不用针对同一指令集的不同处理器的架构发行相应的软件了。同样地，以前的代码也会从更新的处理器上获得好处而不用重新编译。

重点

流水线和多发射执行都提高了指令吞吐量的峰值并致力于开发指令级并行度。然而，由于处理器有些时候必须等待依赖关系明确化之后才能继续工作，所以程序中的数据和控制依赖关系往往限制了可达到性能的上限。基于软件的并行性开发方法依赖于编译器来寻找并尽量减少这些依赖关系所造成的不良后果，而基于硬件的方法则主要依赖于流水线和发射机制。推测执行可以由硬件，也可由编译器来完成，它可以增加可用的并行度，但是使用的时候我们也必须小心，因为错误的推测更会降低性能。

理解程序性能

现代的高性能微处理器可以在同一周期内发射多条指令；遗憾的是，持续这样的高发射速率是相当困难的。比如，虽然我们有一个周期可以发射4~6条指令的处理器，很少应用程序能保持每周期发射多于两条指令。这里面主要有两个原因。

首先，由于使用了流水线，主要的性能瓶颈在于那些不能立即满足的依赖关系，这就限制了指令之间的并行度因此也就限制了发射速率。虽然对于真正的数据依赖关系而言没什么好的解决方法，但是通常情况下硬件或编译器都不能确切知道依赖关系是否存在，因而也就只能保守地假设依赖关系存在。比如，使用了指针的程序由于有更多的内存别名问题，往往有更大的存在隐式依赖关系的可能。反之，数组访问由于有更大的规则性因而编译器可以推测出没有依赖关系存在的情况。同样地，不能在编译期或运行期被准确预测的分支指令同样会限制可供开发的并行度。通常其他指令级可用的指令级并行性总是存在，但是由于并行度较为分散(有时可能存在于上千条指令之间)，编译器和硬件往往显得力不从心。

其次，内存系统导致的消耗同样会使得流水线难以满负荷运转(这是第7章的主题)。一些内存访问引起的阻塞可以被掩盖掉，但是有限的指令级并行度同样会使阻塞被掩盖的程度有所下降。

自测

说明下列开发指令级并行度的技术或组件主要是基于硬件还是基于软件。对于某些项来讲答案是都有可能的。

1. 分支预测
2. 多发射
3. 超长指令字(VLIW)
4. 超标量
5. 动态调度
6. 乱序执行
7. 推测机制
8. 显式并行(EPIC)
9. 重排序缓冲区
10. 寄存器重命名

11. 谓词化

6.10 实例：Pentium 4 处理器的流水线

在上一章中，我们讨论了 Pentium 4 处理器如何对 IA-32 指令取指以及如何将其译码为微操作。这些微操作之后将被具有复杂动态调度功能和推测机制的流水线所执行，这条流水线可以维持一个周期执行 3 条微操作的速率。本节将着重讲述微操作流水线。Pentium 4 采用了深流水线设计和多发射，以达到较低的 CPI 值和较高的主频。

当我们着手设计一个复杂的基于动态调度的处理器时，它所涉及的各个方面，诸如功能单元的设计，高速缓存和寄存器堆的设计，指令发射和流水线整体控制等问题都是相互关联、互相影响的，这就使得将数据通路从流水线中提取出来的工作显得异常复杂。正因为如此，许多工程师和研究人员采用了微结构^⑦这一术语来指处理器内部结构的具体细节。图 6-50 所示为 Pentium 4 处理器的微结构，主要为用于完成微操作的部件。

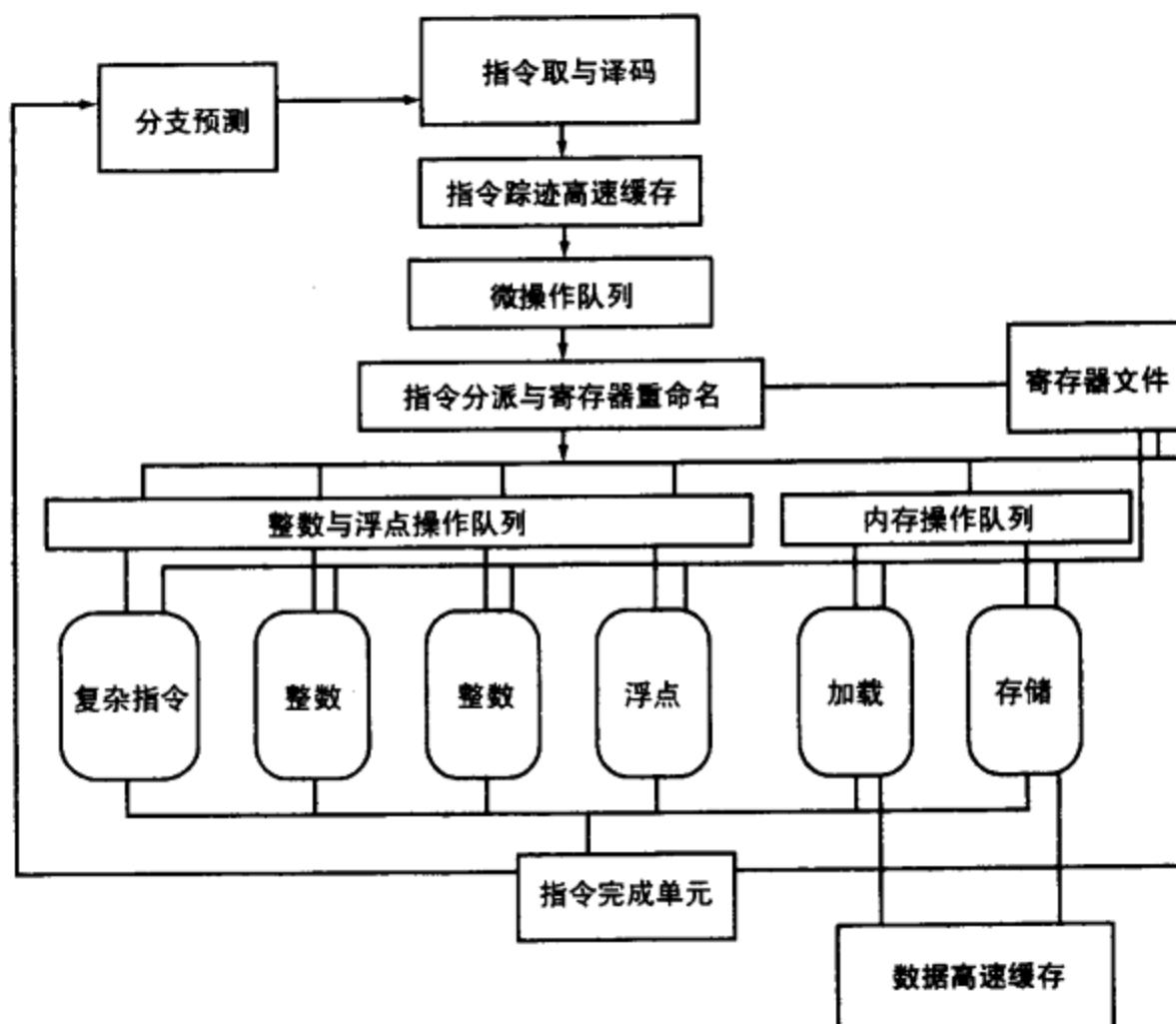


图 6-50 Intel Pentium 4 处理器的微结构

[指令队列允许同时存在于流水线中的微操作数目达到 126 条，其中可以有 48 条 load 操作，24 条 store 操作。实际上有 7 个功能单元，而不是图中的 6 个。这是由于浮点运算(FP)单元包含一个专门用于移动浮点数的器件。load 和 store 单元通常被分为两部分，第一部分用于目的地址的计算，而第二部分则负责具体的访存操作。我们在第 5 章中已经提到，Pentium 4 采用了一种特殊的缓存，称为踪迹缓存(trace cache)，其中记录着 IA-32 指令经过初步译码得到的微操作。踪迹缓存的功用在第 7 章将详细介绍。浮点运算功能单元也负责 MMX 多媒体和 SSE2 指令。在功能单元之间存在着一个复杂的旁路网络；由于流水线是动态而非静态的，旁路是通过标记结果和追踪源操作数而完成的，以便在指令结果生成时能在请求结果的队列中找到匹配。据估计 Intel 将在 2004 年发布新版本的 Pentium 4 处理器，可能会对微结构进行进一步的改进]

^⑦ 微结构(microarchitecture) 处理器内部的组织方式，其中包括主要的功能单元，它们之间是如何互连起来的，以及流水线控制部分。

另一种看待 Pentium 4 的方式是观察一条典型的指令是如何在流水线的各级中流过的。图 6-51 所示为流水线结构，以及各级所消耗的时钟数；当然，指令消耗的时钟数由于动态调度以及各个微操作类型不同等原因，很可能不是定值。

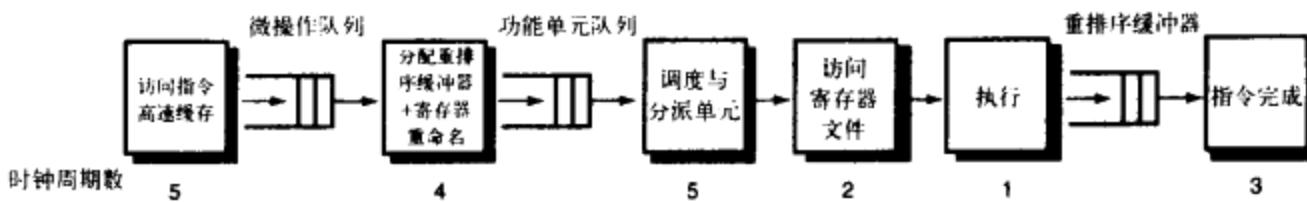


图 6-51 Pentium 4 流水线示意图，所示为一条典型的指令流过流水线各级所消耗的时钟周期数
[主要的指令缓冲区也在图中示出]

Pentium 4，以及早些时候生产的 Pentium III 和 Pentium Pro，都选择使用将 IA-32 指令译码为微操作，并在流水线中使用采用预测机制并含有多个功能单元的流水线来执行这些微操作。实际上，这些处理器的基本结构都大致相同，所有的处理器每个周期都能最多完成 3 个微操作。Pentium 4 比 Pentium III 的性能要高，主要在于有以下优点：

- 1) 流水线的深度大约增长了一倍(Pentium 4 大约为 20 个周期，而 Pentium III 大约为 10 个周期)，所以在同样的工艺下频率可以提高几乎两倍。
- 2) 功能单元要更多(7 比 5)。
- 3) 流水线可以同时支持更多的操作(126 比 40)。
- 4) Pentium 4 使用了 Trace Cache(参见第 7 章)，以及精度更高的分支预测器(4K 项比 512 项)
- 5) 其他内存子系统方面的改进，我们将在第 7 章详述。

细节：为了消除反依赖关系同时建立错误预测恢复机制，Pentium 4 同时采用了重排序和寄存器重命名。寄存器重命名机制显式地将处理器中的**体系结构可见寄存器**^①(IA-32 中有 8 个)映射到另一个更大的物理寄存器集合上(Pentium 4 中有 128 个)。Pentium 4 处理器利用寄存器重命名来消除反依赖关系。寄存器重命名要求处理器中维护一个**体系结构可见寄存器**到物理寄存器的映射表，此中记录了哪一个物理寄存器是当前某个**体系结构可见寄存器**最近一次映射所对应的寄存器。由于会将重命名的过程一一记录下来，寄存器重命名提供了另一种错误预测恢复机制：只要将映射过程反向执行，直到完成错误预测之后执行的第一条指令即可。这会使处理器的状态恢复到最后一条正确指令执行结束后的状态，保证了**体系结构可见寄存器**到物理寄存器之间映射关系的正确性。

理解程序性能

Pentium 4 采用了深流水线(平均每条指令的流水线级数为 20 个以上)与多发射的混合设计方案以达到高性能。通过将指令间短程依赖关系的延时维持在较低的水平(对于 ALU 操作而言是 0 个周期，load 指令为 2 个周期)，减小了数据依赖关系的影响。在这个处理器上，对性能能产生最大影响的潜在瓶颈在何处呢？下面的列表中列出了几个潜在的影响性能的问题，其中后三个适用于所有的高性能流水线处理器。

- 使用那些不能映射为 3 条或 3 条以下微操作的复杂 IA-32 指令。

^① 体系结构可见寄存器(architectural register) 通过指令集暴露出来的程序可见寄存器，比如，在 MIPS 指令之中包含有 32 个整数寄存器和 16 个浮点寄存器。

- 难于预测的分支，这会造成误预测下的阻塞和错误推测下重启的开销。
 - 较低的指令局部性，这会使得踪迹缓存不能有效地工作。
 - 较长的依赖关系，通常会由延时较大的指令或产生数据高速缓存缺失的指令产生，这会造成流水线的阻塞。
- 访存(参见第7章)可能会导致处理器延迟，进而对性能产生负面影响。

自测

下面的论述正确与否？

1. Pentium 4 比 Pentium III 在一个周期内能发射的指令数要多。
2. Pentium 4 的多发射流水线可以直接执行 IA-32 指令。
3. Pentium 4 采用了动态调度，但是未使用推测机制。
4. Pentium 4 微结构含有远远多于 IA-32 指令集寄存器数目的物理寄存器。
5. Pentium 4 处理器的流水线级数比 Pentium III 要少。
6. Pentium 4 中的踪迹缓存完全等同于一般的指令缓存。

6.11 谬误和陷阱

谬误：流水线是一种简单的结构。

本书证明了正确执行流水线必须非常谨慎。我们的一本高级教程的第一版尽管经过上百人审校过，而且曾经在 18 个大学的课堂上使用过，它仍然有一个流水线方面的错误。直到有人根据该书制造计算机时才发现了这个错误。如果用 Verilog 去描述一条如书中所述的基于 Pentium 4 处理器的流水线，需要几千行的代码，这个事实也指出了其复杂性。所以必须谨慎小心！

谬误：流水线概念的实现与技术无关。

当片上的晶体管的数量和速度决定五级流水线是最好的解决方案时，延迟的分支(参见 6.6.3 节的细节部分)就是一种简单的控制冒险的方法。但对于长流水线、超标量执行和动态分支预测，延迟分支的方法就成为多余的了。在 20 世纪 90 年代初期动态流水线调度需要耗费很多的资源，并且无法得到很好的性能，但随着晶体管的预算持续加倍和逻辑电路变得比内存更快，多个功能单元和动态流水线更加实用。今天，所有的高端处理器都使用多发射，而且大部分处理器选择使用更激进的预测方法。

陷阱：没有认识到指令集的设计反过来会影响流水线。

很多流水线引入的困难是由于指令集的复杂性造成的，例如：

- 指令长度和指令运行时间变化太大会导致流水线各阶段的不平衡，而且它们还会使一条设计流水线里面的冒险检测在指令集层面上严重复杂化。这个问题最初是在 20 世纪 80 年代后期，通过使用今天的 Pentium 4 处理器使用的微流水线配置，在 DEC VAX 8500 上得到克服。当然，介于微操作和实际指令之间通信的高层次转化和维持仍然存在。
- 复杂的寻址方法可能引起很多不同的问题。更新寄存器的寻址方法，如更新寻址(参见第 3 章)，会使冒险检测复杂化。而需要多次访存的寻址方法将使流水线的控制复杂化，并且使保持流水线平稳流动比较困难。

大概最好的例子是 DEC Alpha 和 DEC NVAX。通过比较可以看到，Alpha 的新指令集使得它的性能是 DEC NVAX 性能的两倍以上。另一个例子是 Bhandarkar 和 Clark[1991]使用 SPEC 基准测试通过计算时钟周期测试比较了 MIPS M/2000 和 DEC VAX 8700；他们得到了如下结论，尽管 MIPS M/2000 执行了更多的指令，但是 VAX 平均执行时间 2.7 倍于 MIPS 的时钟周期，所以总体

上 MIPS 更快一些。

6.12 结论

智慧十分之九体现在恰当的时机。

美国谚语

流水线提高了每一条指令的平均指令时间。根据你是单时钟周期数据通路还是多时钟周期数据通路描述，平均指令时间的缩短可以理解为减小了时钟周期的长短或者减小了执行每条指令的时钟周期数(CPI)。对简单的单时钟周期数据通路，流水线可以表示为减少了简单数据通路的时钟周期长短。而多发射，相比较而言，则明确地关注于减少 CPI(或者增加 IPC)。图 6-52 说明了对于第 5 章和第 6 章中的微体系结构 CPI 和时钟速率的影响。性能向右上方增长，因为它是一个 IPC 的和时钟速率的产品，而对于一个给定的指令集，时钟速率决定了性能。

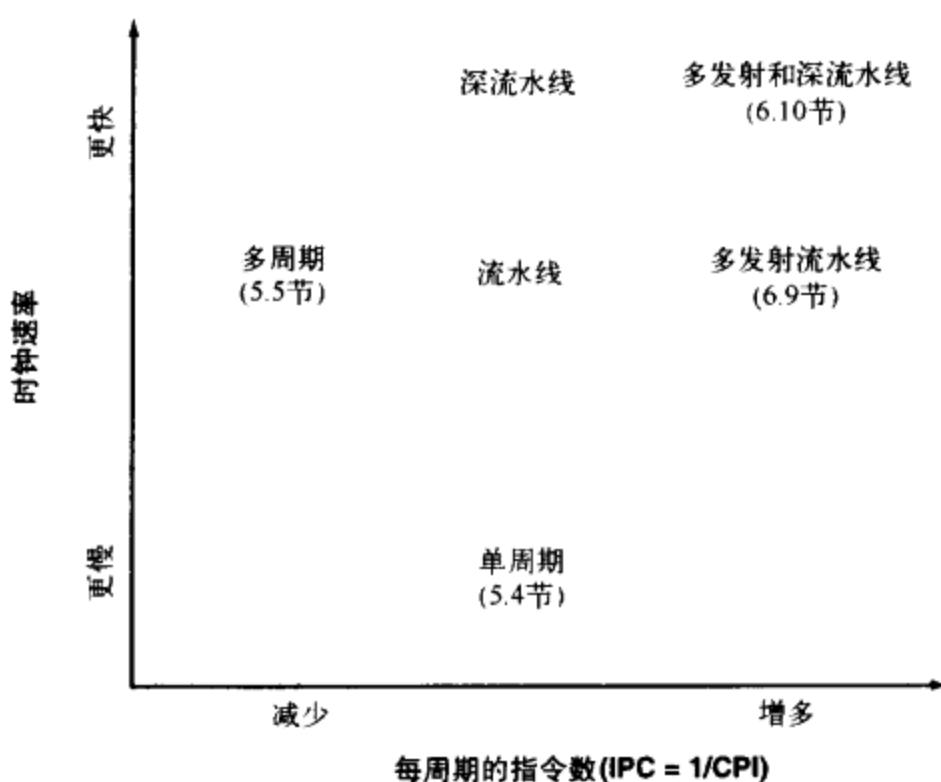


图 6-52 第 5 章介绍的简单数据通路(单时钟周期)和多时钟周期数据通路
以及第 6 章介绍的流水线执行模式的性能

[请记住 CPU 性能是一个以 IPC 乘以时钟速率为参数的函数，因此向右上方移动意味着性能的提高。虽然简单数据通路每个周期的指令数(指令吞吐量)稍大一些，但是流水线数据通路每个周期的指令数实际上与它也非常接近，而且它能达到与多周期数据通路同样快的时钟速率]

流水线提高了吞吐量，但不能提高指令的内在的执行时间(即延时)；指令延时在长度上与多周期方法类似。不同的是多周期方法在指令执行时使用多个相同的硬件，而流水线在每个时钟周期使用专用的硬件执行指令。同样地，多发射添加了额外的允许多指令在每个时钟周期内开始的数据通路硬件，但是却增加了延时。图 6-53 根据共享的硬件数和指令延时[⊖]重新描述了图 6-52 中的数据通路。

流水线和多发射均尝试着开发指令级并行。数据和控制依赖的存在会很有害，是究竟可以开发出多少并行的主要限制。在硬件和软件上的调度和推测执行，是用于降低依赖性能影响的主要技术。

[⊖] 指令延时(instruction latency) 执行一条指令所真正花费的时间。

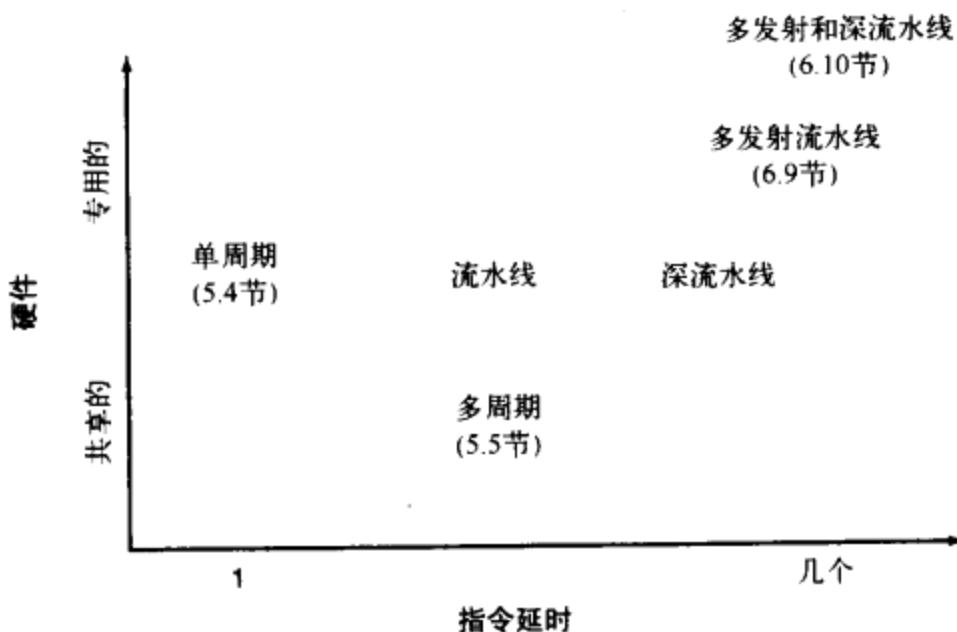


图 6-53 图 6-52 中的数据通路的基本关系

[注意 x 轴使用的是指令的延时，它决定了保持流水线满的难易程度。由于单条指令的执行时间并没有缩短，流水线数据通路的显示是按照多时钟周期的指令延时来的；流水线提高的是指令的吞吐率]

20世纪90年代中期我们转向更长的流水线，多指令发射和动态调度的使用，这些技术帮助我们维持了从20世纪80年代初期以来就以每年60%的速度增长的处理器性能在今天继续增长下去。过去，提高处理器性能的选择好像就是最高的时钟速率或者最复杂的超标量处理器。正如我们所看到的，Pentium 4处理器就是通过结合上述两种方法来获得高性能的。

随着数据处理的显著进步，Amdahl定律预言了系统中的其他部件会成为瓶颈。这个瓶颈就是下一章要讨论的——内存系统。

除了在单处理器系统上开发指令级的并行性外，另一种开发并行性的方法是采用多处理器，它能够开发更粗粒度上的并行性。有关并行处理的内容，我们将在第9章（见光盘 Chapter 9）中讨论。

■ 6.13 历史回顾和深入阅读

这一部分存放在光盘当中，讨论了第一条流水线处理器的历史，最早的超标量，乱序和推测技术的发展，还有同时期的编译器技术的重要发展。

6.14 习题

- 6.1 [5] <§ 6.1>如果 ALU 操作的时间可以被缩短 25%（如图 6-2 描述的那样）：
 - a. 请问它会影响我们从流水线上获得的加速比吗？如果会影响，请问会影响多少？如果不会影响，请说明理由。
 - b. 如果 ALU 操作现在需要多花费 25% 的时间，会怎么样？
- 6.2 [10] <§ 6.1>一个计算机设计师需要设计一个新的微处理器的流水线。她现在有一个 10^6 条指令的核心程序例子。每条指令需要花费 100 ps 执行完成。
 - a. 如果在一个没有流水线的处理器上执行这个核心程序，需要花费多少时间？
 - b. 当前的微处理器拥有 20 级流水线，假设均是十分理想的流水线。对比于没有流水线的处理器，它可以达到多少的加速比？
 - c. 现实的流水线都不是理想的，因为应用流水线会导致流水线的每一级增加一些开销。这些开销会影响指令延时吗？会影响指令吞吐量吗？还是都会影响。

6.3 [5] <§ 6.1> 请使用与图 6-5 类似的图，说明执行下列四条指令所需的转发路径。

```
add $3, $4, $6
sub $5, $3, $2
lw $7, 100($5)
add $8, $7, $2
```

6.4 [10] <§ 6.1> 找出下面代码中的所有数据相关。哪些相关可以通过转发解决的数据冒险？哪些相关可以导致阻塞的数据冒险？

```
add $3, $4, $2
sub $5, $3, $1
lw $6, 200($3)
add $7, $3, $6
```

6.5 [5] <§ 6.1> ■ For More Practice: 延迟分支

6.6 [10] <§ 6.2> 使用图 6-22 作为指导，使用彩笔标示出 sw 指令在五级流水线的每个阶段哪部分数据通路是活动的，哪部分是非活动的。建议你将图 6-22 复印 5 份回答这个问题。另外要确认包括一个图例来解释你的颜色安排。

6.7 [5] <§ 6.2> ■ For More Practice: 通过画图来理解流水线

6.8 [5] <§ 6.2> ■ For More Practice: 通过画图来理解流水线

6.9 [15] <§ 6.2> ■ For More Practice: 通过画图来理解流水线

6.10 [5] <§ 6.2> ■ For More Practice: 流水线寄存器

6.11 [15] <§ 6.2> ■ For More Practice: 流水线浮点

6.12 [15] <§ 6.3> 图 6-37 和图 6-35 是两种流水线的绘图方法。确信你已经理解了两种流水线画法之间的关系，请使用图 6-37 的形式画出图 6-31~图 6-35 中的信息。请突出标示图中数据通路的活动部分。

6.13 [20] <§ 6.3> 图 6-14-10 与 ■ For More Practice 部分的图 6-14-7 相似，但其中的指令没有确认，请你尽可能多地确定 5 级流水线步骤中的 5 条指令的情况。如果不能确定某条指令的某个字段，请说明原因。参考图 3-18 和图 A-10-2，考虑在哪些位置的机器指令容易被译码成为汇编语言。参考图 6-26~图 6-28，考虑在哪些位置的指令容易获得控制信号的值。你可以仔细分析图 6-14-5 到图 6-14-9 以帮助理解如何表示控制值集(例如，在一个周期中最左边的位在下一个周期将成为最上边的位)。例如，图 6-14-6 中减法指令的 EX 阶段的控制值 1100 是在第 3 个周期时的 ID 阶段计算得到的，而它在第 4 个周期将变成 3 个独立的值，即 RegDst(1)、ALUOp(10) 和 ALUSrc(0)。

6.14 [40] <§ 6.3> 如下的代码片断被图 6-30 的流水线执行。

```
lw $5, 40($2)
add $6, $3, $2
or $7, $2, $1
and $8, $4, $3
sub $9, $2, $1
```

在第 5 个周期，这些指令被执行之前，处理器的状态如下：

a. PC 中 sub_instruction 的地址是 100_{ten} 。

b. 每个寄存器的初始值为 10_{ten} 加上寄存器号(例如：\$8 寄存器的初始值是 18_{ten})。

c. 访问的每个内存字的初始值为 1000_{ten} 加上这个字的字节地址(例如：内存[8]的初始值为 1008_{ten})。

确定在第 5 个时钟周期时，四个流水线寄存器每个域的值。

6.15 [20] <§ 6.3> ■ For More Practice: 标注流水线的控制图表

6.16 [20] <§ 6.4> ■ For More Practice: 举例说明转发图表

6.17 [5] <§ 6.4, 6.5> 考虑在图 6-36 中的流水线数据通路中执行以下代码：

```
add $2, $3, $1
sub $4, $3, $5
```

```

add    $5,  $3,  $7
add    $7,  $6,  $1
add    $8,  $2,  $6

```

说明在第 5 个时钟周期末，哪些寄存器被读取，哪些寄存器将被写入？

- 6.18 [5]<§ 6.4, 6.5>考虑习题 6.17 中的程序，解释在第 5 个时钟周期时转发单元所起的作用。如果有对照之处，请解释其不同之处。
- 6.19 [5]<§ 6.4, 6.5>考虑习题 6.17 中的程序，解释在第 5 个时钟周期时，冒险探测单元所起的作用。请解释其不同之处。
- 6.20 [20]<§ 6.4, 6.5>■ For More Practice: 内存的转发
- 6.21 [5]<§ 6.5>我们有一个包含 10^3 条形如“lw, add, lw, add, ...”指令的程序。其中 add 指令只依赖于它之前的那条 lw 指令，同样 lw 指令也只依赖于它之前的那条 add 指令。如果这个程序被图 6-36 的流水线数据通路执行：
- 实际的 CPI 是多少？
 - 如果没有转发，实际的 CPI 是多少？
- 6.22 [5]<§ 6.4, 6.5>考虑在图 6-36 的流水线数据通路上执行如下的代码：

```

lw    $4, 100($2)
sub   $6, $4, $3
add   $2, $3, $5

```

执行这段代码需要几个时钟周期？仿照图 6-34，画一张图描述需要解决的相关性。仿照图 6-35，画一张图描述在流水线中指令为了解决确定的问题(加入阻塞或者转发)实际上是怎样执行的。

- 6.23 [15]<§ 6.5>列举图 6-36 中转发单元所有的输入和输出。并给出每一个输入和输出的名字、占据的位数以及主要的用法。
- 6.24 [20]<§ 6.5>■ For More Practice: 举例说明转发和阻塞图表
- 6.25 [20]<§ 6.5>■ For More Practice: 如果将转发移到 ID 阶段的影响
- 6.26 [15]<§ 6.2~6.5>■ For More Practice: 流水线上的访问内存模式的影响
- 6.27 [10]<§ 6.2~6.5>■ For More Practice: 流水线内存操作数算术操作的影响
- 6.28 [30]<§ 6.5, 附录 C>■ For More Practice: 转发单元硬件设计
- 6.29 [1 周] <§ 6.4, 6.5>请使用本书提供的仿真器统计 C 程序的数据冒险(这个 C 程序可以是指令，也可以是软件)。这里你需要编写一个子程序，该子程序传递指令并模拟本章讨论的五级流水线。你的程序收集以下的统计信息：
- 执行指令的数量
 - 没有被转发解决的数据冒险的数量和被转发解决的数据数量
 - 如果你使用的 MIPS C 语言编译器使用 nop 指令避免冒险，统计 nop 指令的数量
- 假设内存的访问需要 1 个时钟周期，计算执行一条指令占用的平均时钟周期数。将软件插入的 nop 指令看作是阻塞，因此在计算 CPI 时需要从执行指令的数量中减去这些 nop 指令的数量。
- 6.30 [7]<§ 6.4, 6.5>在例子中，我们可以看到访问内存相比于使用 ALU 需要更长的时间，而这正限制了多周期设计的性能优点。假设内存访问需要 2 个时钟周期。请找出单周期和多周期设计的相关性能。在接下来的一些练习中，我们将这个扩充到需要更多工作的流水线设计上！
- 6.31 [10]<§ 6.6>■ For More Practice: 编写条件转移的代码。
- 6.32 [10]<§ 6.6>■ For More Practice: 条件转移的性能优点。
- 6.33 [20]<§ 6.2~6.6>在 6.6.4 节例子中，我们可以看到访问内存相比于使用 CPU 需要更长的时间，而这正限制了多周期和流水线设计的性能优点。假设内存访问需要 2 个时钟周期。画出改进的流水线。列举出所有可能的新的转发位置和所有可能的新的冒险以及他们的长度。
- 6.34 [20]<§ 6.2~6.6>为了比较单周期和多周期，使用练习 6.33 中的调整过的流水线重做 6.6.4 节

的例子。为了分支，假设同样的预测准确性，但是适当地提高处罚。为了 load，假设后续指令以 1/2, 1/4, 1/8, 1/16 等概率依赖于 load 操作。也就是说，一条 load 指令后的第二条指令有 25% 的可能性使用 load 操作的结果作为它自己的资源之一。忽略其他数据冒险，用调整过的流水线找出流水线设计和单周期设计的相关性能。

- 6.35 [10]<§ 6.4~6.6>正如在 6.6.2 节指出的那样，将分支比较移到 ID 阶段可以产生一个转发和冒险的可能性，而这是不能够被转发所解决的。给出一个能够显示需要的可能转发路径和必须要被探测到的冒险情形的代码序列，同时考虑只有两个操作数中的一个。如果没有转发，这个情形的数量应该等于冒险的最大长度。
- 6.36 [15]<§ 6.6>我们有一个包含有五个条件分支的程序核。这个程序核将要被执行几千次。下面是程序核一次执行后每个分支的结果(T 代表跳过，N 代表没有跳过)。

分支 1:T-T-T

分支 2:N-N-N-N

分支 3:T-N-T-N-T-N

分支 4:T-T-T-N-T

分支 5:T-T-N-T-T-N-T

假设每次程序核执行时每个分支的行为都是保持一致的。为了动态规划，假设每个分支都有它自己的预测缓冲器并且在每次执行之前每个缓冲都被初始化到同一状态。列举出如下的分支预测规划的预测情况：

- 总是跳过
- 总是不跳过
- 一位的预测器，初始化为预测跳过
- 两位的预测器，初始化为弱预测跳过

请问，预测的精确度是多少？

- 6.37 [10]<§ 6.4~6.6>简略绘制所有的分支输入的转发路径，并且给出它们什么时候必须被激活(像我们在 6.4 节所做的一样)。
- 6.38 [10]<§ 6.4~6.6>像我们在 6.4 节所做的一样，写出能够检测出任何分支来源上的冒险的逻辑。
- 6.39 [10]<§ 6.4~6.6>6.1.2 节的第 2 个例子显示了如何通过转发和阻塞一个 load 指令后的使用，进而来最大化我们的流水线数据通路的性能。重写下面的代码使得在这个数据通路上的性能最小化，那也就是说，记录这些指令，这样这个序列会花费最多的时钟周期来执行，同时又可以得到同样的结果。

```
lw    $2, 100($6)
lw    $3, 200($7)
add   $4, $2, $3
add   $6, $3, $5
sub   $8, $4, $6
lw    $7, 300($8)
beq   $7, $8, Loop
```

- 6.40 [20]<§ 6.6>在图 6-54 的流水线数据通路中，是否能同时清除和阻塞指令？如果能，它们是否会产生冲突和/或协作？如果产生协作，它们是如何协作的？如果产生冲突，它们应该具有什么样的优先级？为了保证它们之间必要的优先级，需要对这个数据通路做出什么简单改变？你可以使用下面的程序代码帮助你回答这个问题。

```
beq $1, $2, TARGET  # assume that the branch is taken
lw  $3, 40($4)
add $2, $3, $4
sw  $2, 40($4)
TARGET: or  $1, $1, $2
```

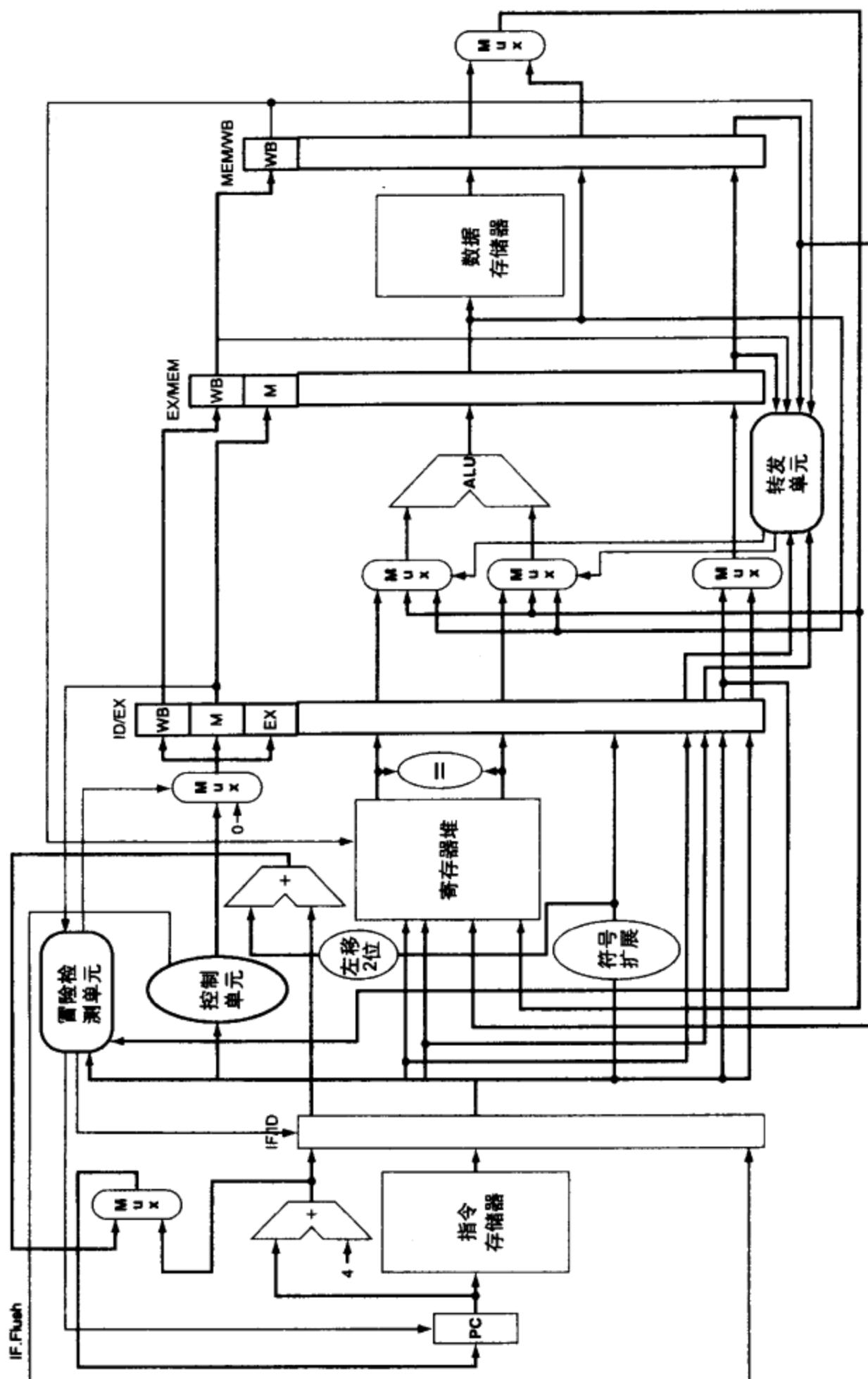


图 6-54 分支的数据通路，包括清除跟随分支的指令的硬件

[最优化的是将分支决定从流水线的第四阶段移到第二阶段；在那个时候将会有一条跟随分支的指令在流水线内。通过将 IF/ID 流水线寄存器的值清零，控制线 IF.Flush 会将取指令转化成什么都不做的 nop 指令。虽然清除线是从图中的控制单元引出的，但是实际上它却是来自与能够决定分支操作是否跳过的硬件，并且用 ID 阶段的寄存器右侧的等号来标记。转发的交换和路径必须也要添加到这个阶段，但是为了简化该图，这些东西都没有在图上显示]

- 6.41 [15]<§ 6.4, 6.7>在 6.7-4~6.7-5 页的图 6-7-2 中的描述转发的 Verilog 代码没有考虑转发作为一个结果，它的值是被一条 SW 指令存储的。请将这点在 Verilog 代码中添加上。
- 6.42 [5]<§ 6.5, 6.7>在 6.7-6~6.7-7 页的图 6-7-3 中的描述阻塞的 Verilog 代码没有考虑转发作为一个结果，它的值在地址计算中使用。请将这点在 Verilog 代码中添加上。
- 6.43 [15]<§ 6.6, 6.7>在 6.7-6~6.7-7 页的图 6-7-3 上的描述执行分支冒险探测和阻塞的 Verilog 代码没有探测 BEQ 指令的两个源寄存器的数据冒险的可能性。扩展 6.7-6~6.7-7 页的图 6-7-3 上的 Verilog 代码，使其能够处理分支操作数的数据冒险。写出所有在 ID 阶段，用于完成分支的转发和阻塞逻辑。
- 6.44 [10]<§ 6.6, 6.7>重写图 6-7-3 上的 Verilog 代码，使其可以执行延迟分支策略。
- 6.45 [20]<§ 6.6, 6.7>重写图 6-7-3 上的 Verilog 代码，使其可以执行一个分支目标缓冲。假设这个缓冲被如下定义的一个模块所使用：

```
module PredictPC (currentPC,nextPC,miss,update,destination);
    input currentPC,
          update, // true if previous prediction was unavailable or incorrect
          destination; / used with update to correct a prediction
    output nextPC, // returns the next PC if prediction is accurate
          miss; // true means no prediction in buffer
endmodule;
```

确认你已经包括了所有的三种可能：一个正确的预测，一个在缓冲中的缺失(那就是说发生缺失)，和一个不正确的预测。在最后的两种情况里，你必须也要更新预测。

- 6.46 [一个月]<§ 5.4, 6.3~6.8>如果你对 Verilog 或 ViewLogic 等模拟系统有所了解，请首先设计第 5 章中的单循环数据通路及控制。然后在你的设计中加入本章介绍的流水线结构。为了保证你的设计能正确工作，采用单步的方式执行 MIPS 程序。
- 6.47 [10]<§ 6.9>如下的代码已经被展开了一次，但是还没有被调度。假设循环索引是 2 的倍数(例如：\$10 是 8 的倍数)：

```
Loop:   lw    $2,  0($10)
        sub   $4,  $2, $3
        sw    $4,  0($10)
        lw    $5,  4($10)
        sub   $6,  $5, $3
        sw    $6,  4($10)
        addi  $10, $10, 8
        bne   $10, $30, Loop
```

调度这段代码，让其快速执行在标准的 MIPS 流水线上(假设它支持 addi 指令)。假设初始值 \$10 是 0，\$30 是 400，并且那个分支被 MEM 阶段就被解决了。请分析这段被调度的代码相比于原始的没有被调度的代码怎样？

- 6.48 [20]<§ 6.9>本题与习题 6.47 类似，不过在这里指令被展开了两次(即生成了指令的 3 份拷贝)。但是，循环的次数并不能确认是 3 的倍数，请给出能保证代码正确执行的方法。(提示：考虑在循环的开始或结尾加入一些代码，并注意这些代码不执行的情况。)
- 6.49 [20]<§ 6.9>使用习题 6.47 的代码，展开这段代码四次，然后在书中 6.9.2 节中描述的以静态的多发射版本的 MIPS 处理器调度它。你可以假设循环执行 4 的倍数那么多次。
- 6.50 [10]<§ 6.1~6.9>技术向着更小的特征尺寸方向发展，线路相应地就变慢了(相对于逻辑)。由于收缩的特征尺寸和时钟速率的增加，逻辑变快了，线路上的延迟消耗了更多的时钟周期。那也就是为什么 Pentium 4 处理器有一些流水线阶段专门沿着线路从流水线的一部分传输数据到另一个部分。添加处理线路延迟的流水阶段的缺点是什么？
- 6.51 [30]<§ 6.10>新处理器的发布速度比新教科书的出版速度还要快。为了使你的教科书跟上时代步伐，调查一些这个领域的最新发展，然后写一页总结附在 6.10 节的后面。可以使用万维网搜索一些

关于最近的 Intel 或 AMD 处理器的特性作为开始。

自测题参考答案

§ 6.1: 1. 阻塞 LW 指令的结果。2. 旁路 ADD 指令的结果。3. 不要求任何的阻塞和旁路指令。

§ 6.2: 第二句和第五句是正确的；其余的都是错误的。

§ 6.6: 1. 预测没有被跳过。2. 预测被跳过。3. 动态预测。

§ 6.7, ■ 6.7-3 页: 第一句和第三句都是正确的。

§ 6.7, ■ 6.7-7 页: 只有句子# 3 是完全正确的。

§ 6.8: 只有# 4 是完全正确的。# 2 部分正确。

§ 6.9: 推测: 全部; 记录缓冲: 硬件; 寄存器重命名: 全部; 乱序执行: 硬件; 预测: 软件; 分支预测: 全部; VLIW: 软件; 超标量: 硬件; EPIC: 全部, 因为有一个实际的硬件支持; 多发射: 全部; 动态规划: 硬件。

§ 6.10: 所有的叙述都是错误的。

现实世界中的计算机: 无网守的大众通信方式

问题: 为社会提供传统大众媒体所不能提供的新闻和视点信息的资源。

解决方案: 使用因特网和 WWW 以选择并发表非传统和全局性的新闻信息。

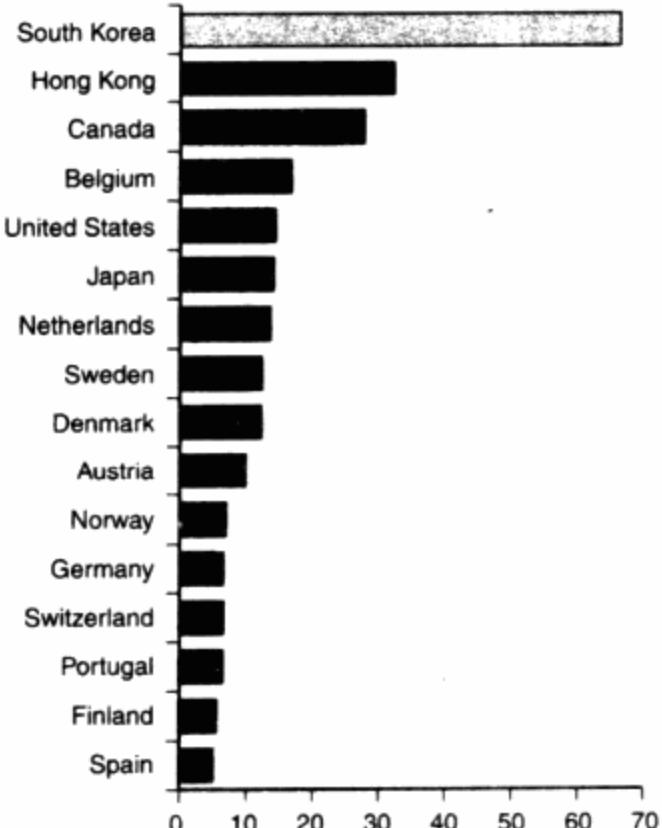
因特网有望满足大众获得未经过传统媒体(比如: 电视、报纸和杂志)处理过的信息的需求。为了对未来有所预见, 我们首先来考察一下有广泛、高速因特网接入的国家。

其中一个国家就是南韩。在 2002 年, 68% 的南韩家庭有宽带接入, 而与之相比美国只有 15%, 西欧只有 8%。(宽带的速度通常为数字订阅者环路或者电缆速度, 大约在 300~1000 Kbps)这个现象的主要原因在于南韩的 70% 家庭都位于大城市中, 同时大约有一半人都居住在公寓里。所以, 南韩的电信业可以迅速完成向 90% 的家庭提供宽带服务的普及工作。

高速接入的广泛使用对南韩社会有何影响呢? 因特网新闻网站变得非常流行, OhMyNews 网站是一个例子, 在那里任何一篇文章只要被查证属实之后任何人都可以对其发文评论。

很多人认为因特网新闻服务对 2002 年南韩总统竞选产生了影响。首先, 它吸引了很多年轻人参与投票。第二, 获胜的候选者倡导了与热衷因特网新闻服务的群体更贴近的政治。他们同心协力战胜了那些主要的传统媒体所大相称颂的竞选对手。

Google News 是另一个超脱于一个国家传统大众媒体的非传统新闻来源的例子。它通过搜索各个国际性新闻服务源以获得新闻主题, 然后将它们总结、按照受关注程度排序并将它们显示出来。这样将哪些信息放在头版将不再由本地的新闻编辑决定了, 这完全取决于世界范围的媒体信息。另外, 通过提供其他许多国家的新闻链接, 它能使得读者能获得一个国际性的视角, 而非以前的狭隘观点。同时每天它都会多次更新, 较于日报优点不言自明。下面的图中比较了 *New York Times* 的头版和 Google News 的首页在同一天各自显示的内容。



2002 年家庭接入宽带网络的比例(按照国家排序)。来源: The Yankee Group, 波士顿

这些技术对社会产生的广泛影响提醒我们计算机工程师，我们对我们所在的社团有自己的责任。我们必须对诸如隐私、安全、自由言论等问题有足够的认识，以使得新技术革新能使价值观念更加稳固，而不是受其所害。

New York Times Front Page	Google News
<p>Judge Rules Out a Death Penalty for 9/11 Suspect Rebuke for Justice Dept.</p> <p>Poll Shows Drop in Confidence on Bush Skill in Handling Crises Country on Wrong Track, Says Solid Majority</p> <p>Revised Admission for High Schools City Says Students Will Get First Preference</p> <p>No Illicit Arms Found in Iraq, U.S. Inspector Tells Congress</p> <p>U.S. Practice How to Down Hijacked Jets</p> <p>Coetzee, Writer of Apartheid as Bleak Mirror, Wins Nobel</p> <p>Sexual Accusations Lead to an Apology by Schwarzenegger</p> <p>Interim Chief Accepts Stock Exchange Shift</p> <p>Yankees Even with Twins Agency Warns of Fake Drugs Limbaugh Fallback Position</p>	<p>Top Stories</p> <p>More than 1000 rally behind Schwarzenegger AP-5 minutes ago</p> <p>Maria Shriver defends husband CNN Can accusations hurt Arnold's campaign?</p> <p>KESQ</p> <p>Bush: Hussein 'A Danger to the World' ABC news-5 hours ago</p> <p>Bush Stands By Decision Voice of America</p> <p>Hunt for weapons yields no evidence The Canberra Times</p> <p>and 1252 related</p> <p>World Stories</p> <p>Defiant UN chief announces rival blueprint for Iraq The Times (UK)-2 hours ago</p> <p>France, Russia Assail US Draft on Iraq Reuters</p> <p>and 598 related</p> <p>and 782 related</p>

New York Times 与 Google News 的对比图(2003 年 10 月 3 日下午 6 点)

[报纸的头版头条必须在重要信息和国家新闻、地方新闻、体育新闻之间做权衡。Google News 的每一个头条都对应了更多的新闻，并且它们来自于全球，并且还包含读者进一步了解所必需的链接。Google 提供的新闻一天更新多次因而更加及时]

要了解更多，请在图书馆(Library)中参阅下列参考资料

- “Seriously wired,” *The Economist*, April 17, 2003.
- OhMyNews, www.ohmynews.com
- Google News, www.news.google.com