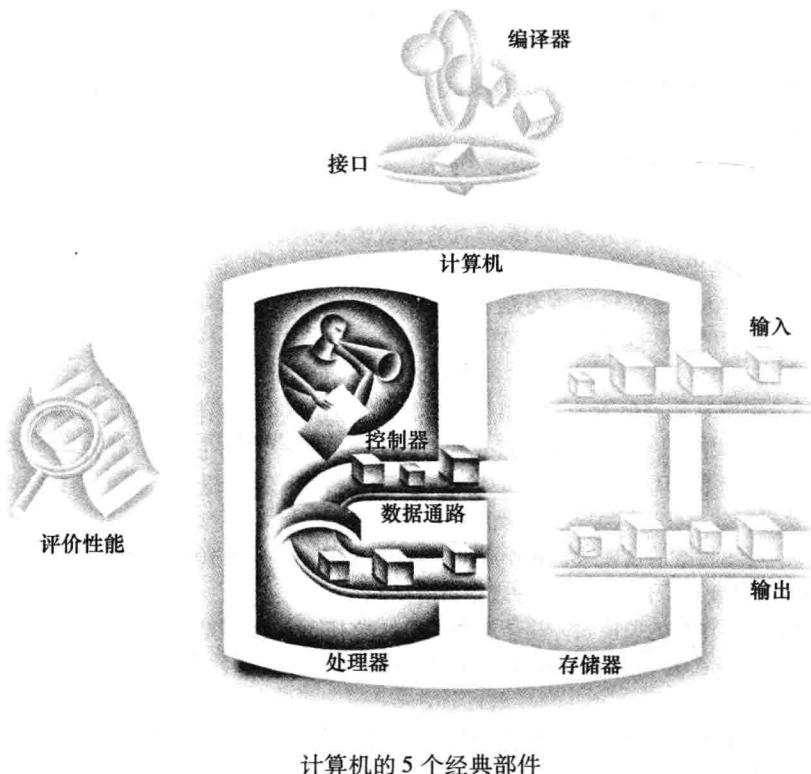


# 处 理 器



在关键问题上，没有什么细节是小事。

——法 国 谚 语

## 4.1 引言

在第1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、时钟周期长度和每条指令所需时钟周期数（CPI）。我们在第2章阐明编译器和指令集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 CPI。在本章中，我们为 MIPS 指令集的两种不同实现方式分别建立数据通路和控制单元。

本章包含了实现一个处理器所需的原理与技术。先从一个高度抽象和简化的概述开始，再建立数据通路，并进一步构建一个简单的处理器以实现像 MIPS 这样的指令集。本章的其余部分还包括：一个更实际的流水化的 MIPS 实现，以及实现更复杂的指令集（如 x86）时所需要的概念。

对理解指令的高层解释及其对程序性能的影响感兴趣的读者，4.5 节给出了流水线的基本概念。4.10 节介绍了最近的趋势。4.11 节描述了最新的 Intel Core i7 和 ARM Cortex-A8 体系结构。4.12 节展示了如何通过指令级并行将 3.8 节的矩阵乘法性能提高两倍以上。这几节为在高

层次理解流水线概念提供了必要的背景知识。

对希望能更深入地理解处理器及其性能的读者，4.3节、4.4节和4.6节很有用。对如何建立一个处理器感兴趣的读者可以阅读4.2节、4.7节、4.8节和4.9节。对现代硬件设计感兴趣的读者，可以参考4.13节，其中介绍了实现硬件时使用的硬件设计语言与CAD工具，以及如何使用硬件设计语言来描述一个流水化的实现。它对于理解流水化硬件执行的细节有很大帮助。

## 一个基本的MIPS实现

我们将要设计的实现方式包含了MIPS指令集的一个核心子集：

- 存储器访问指令：取字（lw）和存字（sw）。
- 算术逻辑指令：加法（add）、减法（sub）、与运算（AND）、或运算（OR）和小于则设置（slt）。
- 分支指令：相等则分支（beq）和跳转（j），我们放到最后实现。

这个子集没有包含所有的整数指令（如不包含乘、除指令和移位指令），也没有包含任何浮点指令。然而，使用该子集可以说明在建立数据通路和控制单元时的关键原理，并可以在此基础上实现其他指令。

242  
244

在学习此实现方式时，我们将会有机会看到指令集如何决定实现方式的多个方面，以及实现策略如何影响计算机的时钟速度和CPI。在第1章介绍的许多关键设计原理，如“简单源于规整”的指导思想，都将体现出来。并且，本章中用于实现MIPS子集的大多数概念与很多计算机的基本构造思想是一致的，包括从高性能服务器到通用微处理器、嵌入式处理器等各式各样的计算机。

### 实现方式概述

在第2章中，我们学习了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。实现每条指令的前两步是一样的：

- 1) 程序计数器（PC）指向指令所在的存储单元，并从中取出指令。
- 2) 通过指令字段内容，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，而其他大多数指令要求读取两个寄存器。

这两步之后，为完成指令而进行的步骤则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，与具体指令无关。MIPS指令集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。

例如，除跳转指令外的所有指令在读取寄存器后，都要使用算术逻辑单元（ALU）。存储访问指令用ALU计算地址，算术逻辑指令用ALU执行运算，分支指令用ALU进行比较。在使用ALU之后，完成不同指令所需的动作就有所不同了。存储访问指令需要访问内存以便读取和存储数据。算术逻辑指令或取数指令将来自ALU或存储器的数据写入寄存器。对分支指令，我们需要根据比较的结果决定是否改变下一条指令地址；如果不修改下一条指令地址，则下一条指令地址默认是当前指令地址+4。

图4-1给出了一种MIPS实现的抽象视图，图中主要描述了不同的功能单元及其互连关系。尽管该图给出了绝大多数数据在处理器中的流动方式，但它仍然忽略了指令执行过程中的两个重要方面。

首先，在图4-1中的许多位置，某个单元的数据可能来自于两个不同的单元。例如，写入

245

PC 的值可能来自两个加法器中的一个，写入寄存器堆的数据可能来自 ALU 或数据存储器，ALU 的第二个输入可能来自寄存器或指令中的立即数字段。实际上，不能简单地直接将这些数据线连在一起，必须增加一个逻辑单元用以从不同的数据来源中选择一个送给目标单元。这个选择过程通常是由一个叫多选器（multiplexor）的逻辑单元完成的，尽管该单元叫数据选择器可能更合适。附录 B 详细描述了多选器根据控制信号选择不同输入的过程。控制信号主要由当前执行指令中包含的信息决定。

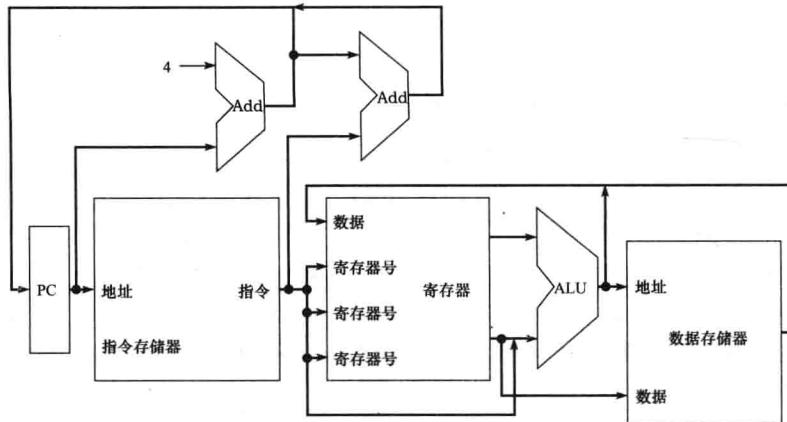


图 4-1 一个 MIPS 子集实现的抽象视图，描述了主要功能单元及其连接。所有指令都始于使用程序计数器获得指令在指令存储器中的地址。在取到指令后，指令所使用的寄存器操作数由指令中的对应字段决定。在取到寄存器操作数之后，可以用来计算存储器地址（对于存取类指令），或者计算算术运算结果（对于整数算术逻辑类指令），或者进行比较（对于分支类指令）。如果是算术逻辑类指令，ALU 的结果必须写回寄存器；如果是存取类指令，ALU 的结果可作为读写存储器的地址。ALU 或存储器的结果可写回寄存器堆。分支操作需要使用 ALU 的输出来决定下一个指令地址，下一个指令地址可能来自 ALU（在 ALU 中完成 PC 值与分支偏移量相加），也可能来自加法器（当前 PC 值加 4）。连接功能单元的粗线表示总线，其中包含多个信号。箭头用来指示信息流动的方向。因为信号线在图上可能相交，所以在相交信号线确实相连时用一个黑点来表示

246

其次，图 4-1 中的许多单元的控制依赖于当前执行指令的类型。例如，存取指令读写数据存储器，取数指令和算术逻辑指令写入寄存器堆。很显然，ALU 根据不同的指令执行不同的操作。（附录 B 给出了 ALU 的设计细节）。类似于多选器，这些操作都由控制信号确定，而控制信号是由指令的某些字段所决定的。

图 4-2 在图 4-1 的基础上增加了三个必需的多选器和主要功能单元的控制信号。图中还增加了一个控制单元（control unit），它以指令为输入，决定功能单元和两个多选器的控制信号。第三个选择器用来决定是将  $PC + 4$  还是分支目的地址写入 PC，该选择器在执行 beq 指令时，根据 ALU 进行比较时设置的 Zero 标志位选择写入 PC 的数值。MIPS 指令集的简单与规整使得只需简单的译码即可生成控制信号。

在本章后面的部分，我们将会为图 4-2 加入更多的细节，包括更多的功能单元和单元间的连接，并增强控制单元功能以控制不同类型的指令执行过程。4.3 节和 4.4 节描述了每条指令使用一个时钟周期的简单实现方式，它将遵循图 4-1 和图 4-2 的一般形式。在第一个设计中，每条指令在一个时钟沿开始执行，然后在下一个时钟沿完成执行。

尽管这种方法易于理解，但是并不实际，因为时钟周期必须设置为足够容纳执行时间最长的指令。在设计完这种简单计算机的控制后，我们将会讨论一种流水的实现方式及其带来的复杂性和异常。

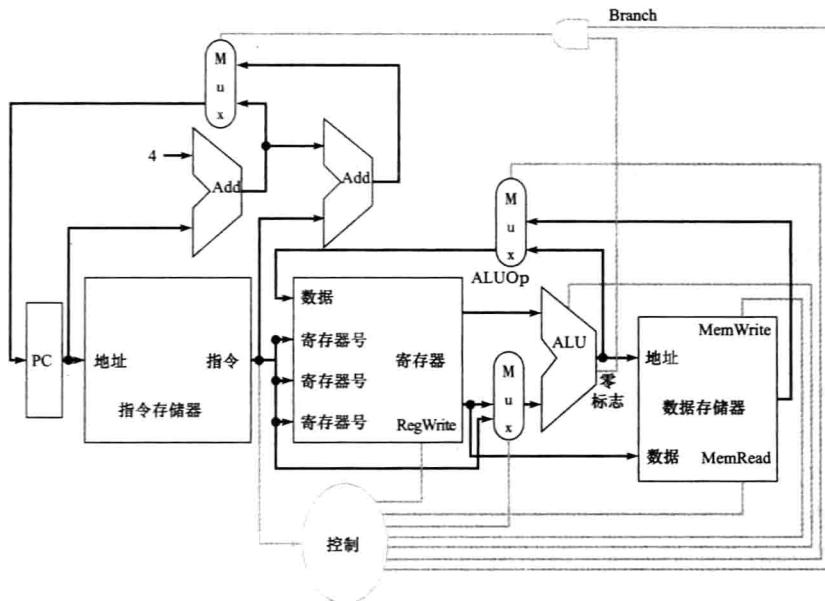


图 4-2 一个 MIPS 子集的基本实现，其中包含了必要的多选器和控制信号。最上面的多选器控制写入 PC 的值 ( $PC + 4$  或分支目的地址)，该多选器由一个门控制，该门将 ALU 的零输出与一个指示是否为分支指令的信号相“与”。中间输出到寄存器堆的多选器，用来选择将被写入寄存器堆中的是 ALU 的输出（算术逻辑指令时）还是数据存储器的输出（取数指令时）。最后，最下面的多选器决定 ALU 的第二个输入是来自寄存器堆（算术逻辑指令或分支指令时）还是指令的偏移量字段（存取指令时）。新增的控制信号直接控制 ALU 的操作、数据存储器的读写和寄存器堆的写入等。控制信号在图中用灰色线标识出来。

247

## 01 小测验

图 4-1 和图 4-2 包含了本章开始给出的计算机五大经典部件中的哪几个？

## 4.2 逻辑设计的一般方法

在考虑计算机的设计时，必须决定机器的逻辑实现以及机器时钟。本节将继续讨论一些本章经常用到的数字逻辑的关键思想。如果你缺乏数字逻辑方面的知识，那么在继续学习之前，阅读附录 B 将有所帮助。

MIPS 实现中的数据通路功能部件包括两种不同的逻辑单元：处理数据值的单元和存储状态的单元。处理数据值的单元都是组合单元（combinational element），它们的输出只取决于当前的输入。当输入相同时，组合单元产生的输出也相同。出现在图 4-1 中并在附录 B 中详细论述的 ALU 就是组合单元。因为其没有内部存储功能，当给定一组输入时总是产生同样的输出。

设计中的其他单元是非组合单元，而是包含状态的。如果一个单元带有内部存储功能，它就包含状态，称之为**状态单元**（state element），因为关机后重启计算机，通过恢复状态单元的原值，计算机可继续运行如同没有断电一样。也就是说，这些状态单元完全描述了计算机的状态。图 4-1 中的指令存储器、数据存储器和寄存器都是状态单元。

- 组合单元：一个操作单元，如与门或 ALU。
  - 状态单元：一个存储单元，如寄存器或存储器。

248

一个状态单元至少有两个输入和一个输出。两个必要的输入为要写入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前一个时钟信号写入单元的数据值。例如，逻辑上最简单的一种状态单元是 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 4-1 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被称为时序的（sequential），因为它们的输出由输入和内部状态共同决定。例如，代表寄存器的功能单元的输出取决于所提供的寄存器号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它们的结构都在附录 B 中有详细论述。

### 时钟方法

**时钟方法**（clocking methodology）规定了信号可以读出和写入的时间。规定信号读写的时间非常重要，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。显然，这种不确定性在计算机的设计中是不允许的。时钟方法即是为避免这种情况而提出的。

为简单起见，我们假定采用边沿触发的时钟（edge-triggered clocking）方法，即在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（见图 4-3）。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中。其输入为之前某时钟周期写入的数据，其输出可供之后某时钟周期使用。

- 时钟方法：用来确定数据相对于时钟何时稳定和有效的方法。
- 边沿触发的时钟：一种所有的状态改变发生于时钟沿的时钟机制。

249

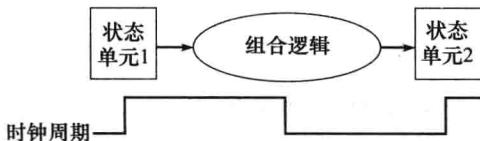


图 4-3 组合逻辑、状态单元和时钟周期的关系。在一个同步的数字系统中，时钟信号决定了数值何时写入状态单元。在有效的时钟边沿导致状态变化之前，状态单元的输入信号必须达到稳定（也就是说，状态单元的值保持不变，直到时钟沿到来）。本章假定所有状态单元（包括存储器）都是上升沿触发的，即这些信号都是在时钟的上升沿发生变化。

图 4-3 描述了一个组合逻辑单元及与其相连的两个状态单元。组合逻辑单元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 所需的时间决定了时钟周期的长度。

为简单起见，若某状态单元在每个有效的时钟边沿都进行写入操作，则可忽略写控制信号（control signal）。相反，若某状态单元不是每个周期都进行修改，那么它就需要一个写控制信号。写控制信号和时钟信号都是输入信号，只有时钟边沿到来并且当写控制信号有效时，状态单元才改变状态。

我们将使用术语**有效**（asserted）表示信号为逻辑高，**无效**（deasserted）表示信号为逻辑低。另外，我们之所以要使用术语“有效”和“无效”，是因为在进行硬件实现时，数字 1 有时表示逻辑高，有时表示逻辑低。

- 控制信号：用来决定多选器选择或指示功能单元操作的信号；它与数据信号相对应，数据信号包含由功能单元操作的信息。
- 有效：信号为逻辑高或真。
- 无效：信号为逻辑低或假。

使用如图 4-4 所示的一种边沿触发的方法，可以在一个时钟周期内读出一个寄存器的值，然后使之经过一些组合逻辑，同时将新值写入该寄存器。选择在时钟的上升沿（从低到高）还是下降沿（从高到低）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。本书我们使用时钟的上升沿。这种边沿触发时钟方法在一个时钟周期内不会出现反馈，图 4-4 中的逻辑可以正确地工作。在附录 B 中，还介绍了其他的一些时序约束（如建立和保持时间）和一些时序方法。

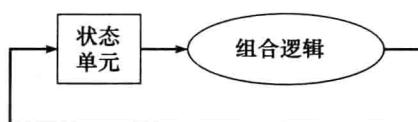


图 4-4 一种边沿触发方法，支持状态单元在同一个时钟周期内同时读写而不会因竞争而出现中间数据。当然，必须保证时钟周期足够长，以使得当有效的时钟边沿到来时输入已经稳定。状态单元的改变由时钟边沿触发，所以不可能在一个时钟周期之内出现反馈。如果有反馈，这个设计就不能正常工作。本章和下一章的设计都采用边沿触发的时钟方法，结构与本图类似。

对 32 位 MIPS 体系结构而言，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，我们会特别指出。图示中用粗线表示总线，即宽度为 1 位以上的信号。有时要把几根总线合起来构成更宽的总线，例如可能将两根 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，灰色线表示的控制信号将其与数据信号区分开来，两者的差别将随本章的进展愈趋明显。

250

### 01 小测验

是非判断：由于寄存器堆在一个时钟周期内既要写入又要读出，所以任何使用边沿触发方式写入的 MIPS 数据通路中必须包含至少一份寄存器堆的备份。

### 01 精解 还有一种 64 位版本的 MIPS 系统结构，其中绝大多数数据通路都是 64 位宽。

## 4.3 建立数据通路

设计数据通路比较合理的方法是首先分析执行每种 MIPS 指令时需要哪些主要部件。下面先来看看每条指令需要什么数据通路部件 (datapath element)。在指出数据通路部件的同时，我们也会指出它们的控制信号。我们将从自底向上开始，使用抽象来对此进行说明。

图 4-5a 给出了我们需要的第一个元素：一个存储单元，它用于存储程序的指令，并在给定地址时提供指令。图 4-5b 展示了程序计数器 (program counter, PC)，在第 2 章曾经出现过，用于保存当前指令的地址。最后，我们需要一个加法器来计算 PC 的值以指向下条指令的地址。这个加法器是一个组合单元，可以用附录 B 中设计的 ALU 实现，只需将其中的控制信号设为总是进行加法操作即可。如图 4-5c 所示，我们将给这样的 ALU 加上 “Add” 标记，以表明它成为了一个加法器而不能再进行其他 ALU 操作。

- **数据通路部件：**一个用来操作或保存处理器中数据的单元。在 MIPS 实现中，数据通路部件包括指令存储器、数据存储器、寄存器堆、ALU 和加法器。
- **程序计数器：**存放下一条将要被执行指令的地址的寄存器。

要执行任何一条指令，首先要从存储单元中将指令取出。为准备执行下一条指令，也必须增加程序计数器使其指向下一条指令，即向后移动 4 字节。此时的数据通路如图 4-6 所示，使

用了图 4-5 中的 3 个部件，它可以取指令并能自增 PC 以获得下一条指令的地址。

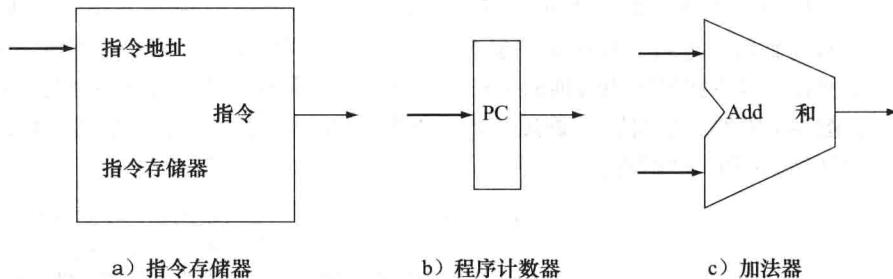


图 4-5 存取指令需要的两个状态单元，以及计算下一条指令地址所需要的加法器。两个状态单元分别是指令存储器和程序计数器。因为数据通路没有写指令，所以指令存储器只提供读访问。因为指令存储器是只读的，我们将它视为组合逻辑：任意时刻的输出都反映了输入地址处的内容，而不需要读控制信号。（在装载程序时需要写入指令存储器，但是这很容易实现，所以为了简单起见我们将其忽略。）程序计数器是一个 32 位的寄存器，它在每个时钟周期末都会被写入，所以不需要写控制信号。加法器采用只进行加法的 ALU，它将输入的两个 32 位数相加，将结果输出

现在讨论 R 型指令（参见图 2-20）。这类指令读两个寄存器，对它们的内容进行 ALU 操作，再写回结果。我们将这类指令称为 R 型指令或算术逻辑指令（因为它们执行算术或逻辑运算）。这个指令集合包括第 2 章介绍的 add、sub、AND、OR 和 slt 指令。251 回忆一下，此类指令的典型形式是 add \$t1, \$t2, \$t3，它将读取 \$t2 和 \$t3，并将结果写回 \$t1。

处理器的 32 个通用寄存器位于一个叫作寄存器堆（register file）的结构中。寄存器堆即寄存器集合，其中的寄存器都可通过指定相应的寄存器号来进行读写。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。

② 寄存器堆：包含一系列寄存器的状态单元，可以通过提供寄存器号进行读写。

由于 R 型指令有 3 个寄存器操作数，每条指令都要从寄存器堆读出两个数据字，再写入一个数据字。为读出一个数据字，寄存器堆需要输入一个要读的寄存器号和一个从寄存器堆读出结果的输出指示。为写入一个数据字，寄存器堆要有两个输入：一个提供要写的寄存器号（register number），另一个提供要写的数据（data）。寄存器堆总是根据输入的寄存器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，写控制信号必须是有效的。这样，我们一共需要 4 个输入（3 个寄存器号和 1 个数据）和两个输出（两个数据），如图 4-7a 所示。输入的寄存器号为 5 位，可指示 32 个寄存器中的某一个 ( $32 = 2^5$ )，而一条数据输入总线和两条数据输出总线宽度均为 32 位。

图 4-7b 所示为 ALU，该 ALU 有两个 32 位输入、一个 32 位输出，还有一个 1 位输出指示

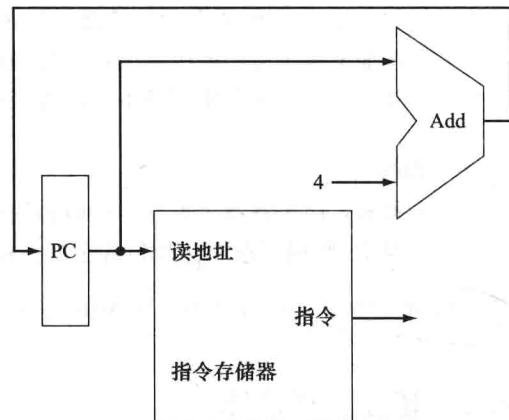


图 4-6 用于取指和程序计数器自增的数据通路部分。取出的指令被数据通路的其他部分使用

其结果是否为 0。ALU 的 4 位控制信号在附录 B 中有详细的描述。在需要了解如何设置 ALU 控制信号时，我们将进行简要的回顾。

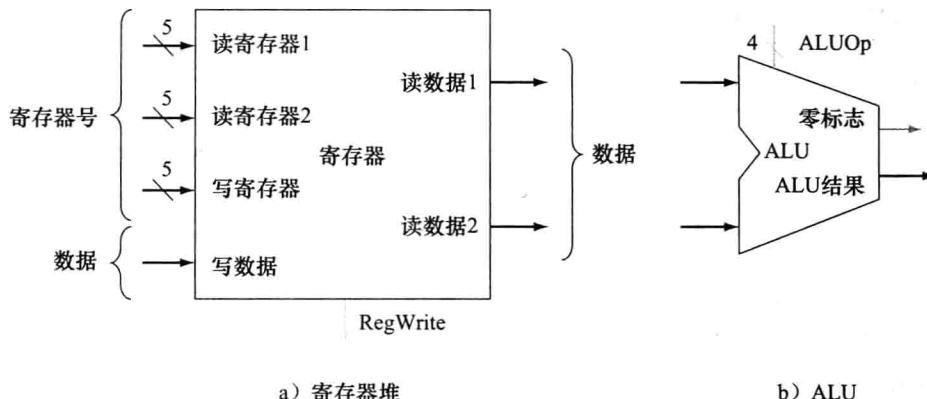


图 4-7 实现 R 型指令的 ALU 操作所需的两个单元——寄存器堆和 ALU。寄存器堆包括了所有的寄存器，有两个读端口和一个写端口。多端口寄存器堆的设计在附录 B 的 B.8 节讨论。寄存器堆的读输出总是对应于读寄存器号，不需要其他的控制信号。但是写寄存器必须明确使能写控制信号。注意写操作是边沿触发的，所以所有的写操作的输入（要写的内容、寄存器号、写控制信号）必须在时钟边沿有效。因为寄存器堆的写入是边沿触发的，故可以在同一时钟周期内读出和写入同一寄存器：读操作将读出以前写入的内容，而写入的内容在下一时钟周期才可读。寄存器号的输入都是 5 位的，数据线为 32 位。若采用附录 B 中的 ALU 设计，则 ALU 的操作可由 4 位 ALU 操作信号控制。我们使用 ALU 的零检测输出信号实现分支指令。溢出信号在 4.9 节讲述异常时才会用到，在此之前我们先忽略它。

下面考虑 MIPS 的存取指令，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 `$t2` 的内容与指令中的 16 位带符号偏移地址相加，得到存储器地址。如果是存储指令，要从寄存器 `$t1` 中读出要存储的数据；如果是取数指令，则要将从存储器中读出的数据存入指定的寄存器 `$t1` 中。所以，图 4-7 中的寄存器堆和 ALU 都会用到。

另外，还需要一个单元将 16 位的偏移地址符号扩展（sign-extend）为 32 位的带符号值，以及一个保存读出或写入数据的存储单元。数据存储单元在存储指令时被写入，所以它有读写控制信号、地址输入和写入存储器的数据输入。

② 符号扩展：为增加数据项的长度，将原数据项的最高位复制到新数据项多出来的高位。

`beq` 指令有 3 个操作数，其中两个为寄存器，用于比较是否相等，另一个是 16 位偏移量，用于计算相对于分支指令所在地址的分支目标地址（branch target address）。它的指令格式为 `beq $t1,$t2,offset`。为了实现该指令，我们必须将 PC 值与符号扩展后的指令偏移量字段相加以得到分支目标地址。分支指令（见第 2 章）的定义中有两个需要注意的地方：

- 指令集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。原因是我们在取指通路中计算了  $PC + 4$ （下一条指令的地址），用这个值作为计算分支目标地址时的基地址比较容易实现。
- 系统结构还规定偏移量左移 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。

为了处理后面这种情况，我们需要把偏移量左移 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

的指令。当分支条件为真（例如，操作数相等）时，分支目标地址成为新的PC，我们就说分支发生（branch taken）了。若操作数不等，自增后的PC将取代当前PC（就像其他普通指令一样），这时就说分支未发生（branch not taken）。

- ② 分支目标地址：该地址指定了一个分支，如果分支发生，那么它将成为新的程序计数器（PC）。在MIPS架构中，指令偏移域与分支指令的下一条指令地址之和组成本分支目标。
- ③ 分支发生：分支条件满足而PC变为分支目标地址的分支。所有的无条件跳转都是发生的分支。
- ④ 分支未发生：分支条件不满足而PC变为分支指令的下一条指令地址。

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操作数。（很快我们还将讲到，分支指令也影响数据通路的取指部分。）图4-9为分支数据通路。为计算分支目标地址，分支数据通路包含了一个如图4-8所示的符号扩展单元和一个加法器。为了进行比较，要由图4-7a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，比较可由在附录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，故可以把两个寄存器数作为输入，并将ALU设置为减法。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，但我们只用它来实现分支时的等值测试。稍后将详细介绍将ALU用于数据通路时，怎样连接它的控制信号。

253  
254

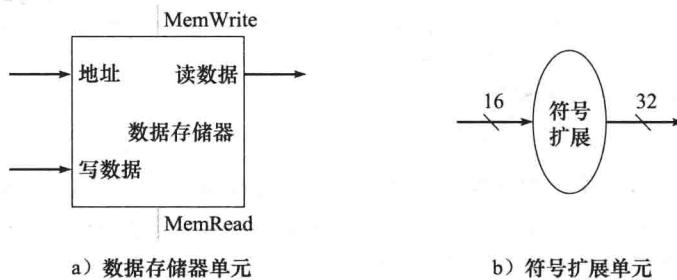


图4-8 除了图4-7中的寄存器堆和ALU，存储指令和取数指令还需要两个单元——数据存储器单元和符号扩展单元。数据存储器单元是一个状态单元，两个输入为地址和所写数据，一个输出为读出的数据。读、写控制信号都是独立的，尽管任意时钟只能激活其中一个。不像寄存器，存储器单元需要一个读控制信号，因为读一个无效地址可能会出问题，我们在第5章会看到这种情况。符号扩展单元有一个16位的输入，符号扩展为32位后输出（见第2章）。假定数据存储器的写是边沿触发的。标准的存储芯片实际上有一个写使能信号用于写操作。尽管标准存储芯片的写使能信号不是边沿触发的，我们的边沿触发的设计可以很容易地应用于真正的存储芯片。关于存储器芯片工作细节的讨论，见附录B的B.8节

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）。

**01 精解** 在实际MIPS指令集中，分支指令是“延迟的”，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目标地址。将分支指令设计为延迟的原因是减轻流水线对分支指令的影响（见4.8节）。为简单起见，本章仅实现非延迟的beq指令。

⑤ 延迟的分支：不管分支条件是否满足，分支指令之后的那条指令总被执行的一种分支。

255

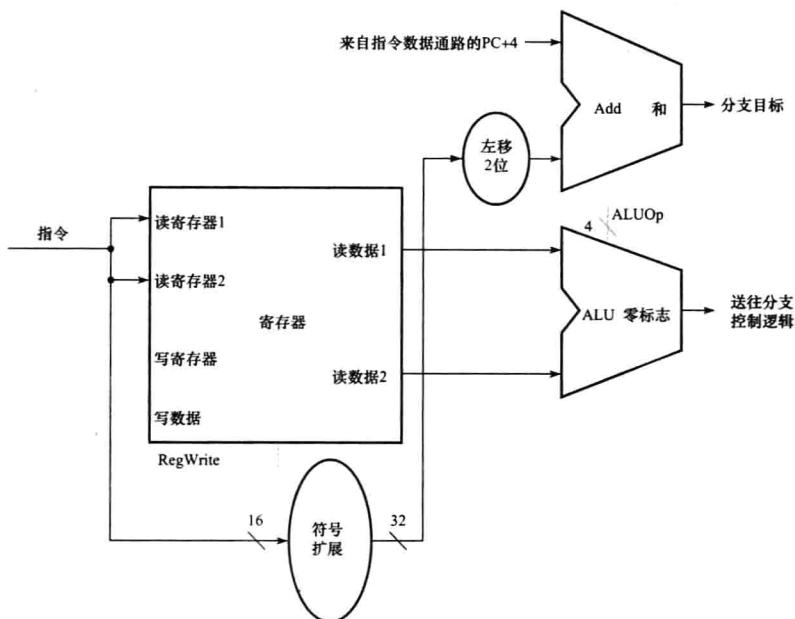


图 4-9 在分支指令的数据通路中，用 ALU 计算分支条件是否成立，用另外的加法器将自增后的 PC 值与符号扩展后左移两位的指令低 16 位（分支偏移量）相加，得到分支目标地址。标有“左移两位”的单元只是输入到输出之间一条简单数据通路，它给符号扩展后的偏移量字段的低位加上两个 0（二进制）。因为“移动”的距离是固定的，所以并不需要真正的移位电路。我们知道偏移量是从 16 位扩展而来的，所以移位只会丢掉“符号位”。控制逻辑根据 ALU 的零输出决定是用自增的 PC 还是分支目标地址来取代当前的 PC

## 创建一个简单的数据通路

我们已经讨论了不同指令类型所需要的数据通路单元，可以把它们连在一起并加上控制来完成一个最简单的 MIPS 子集实现方案。这个最简单的数据通路中，每个时钟周期执行一条完整的指令。这意味着每条指令执行过程中任何数据通路单元都只能被用一次，如果需要使用多次则必须将该数据通路单元复制多份。所以我们除了需要一个指令存储器外，还需要一个数据存储器。尽管有的功能单元需要复制，但在执行不同指令时，很多功能单元也可以被共享。

为了在两种不同类型的指令间共享数据通路单元，我们需要让功能单元有多个输入，而使用多选器和控制信号来从多个输入中进行选择。

256

### 01 例题·建立一个数据通路

算术逻辑指令（或 R 型指令）的数据通路与访存指令的数据通路很相似。它们的主要区别为：

- 算术逻辑指令使用 ALU，并且其输入来自两个寄存器。存储指令也使用 ALU 来进行地址计算，但 ALU 的第二个输入是对指令中 16 位偏移地址进行符号扩展后的值。
- 存入目标寄存器中的值来自于 ALU（对 R 型指令而言）或者存储器（对取数操作而言）。

试设计存储指令和算术逻辑指令操作部分的数据通路，只能使用一个寄存器堆和一个 ALU，可增加必要的多选器。

### 01 答案

为了只用一个 ALU 和一个寄存器堆来创建一个数据通路，ALU 的第二个输入和要存入寄存器堆的数据都需要两个不同的来源。所以，要在 ALU 的输入和寄存器堆的输入数据处各加

入一个多选器。图 4-10 给出了合并后的数据通路。

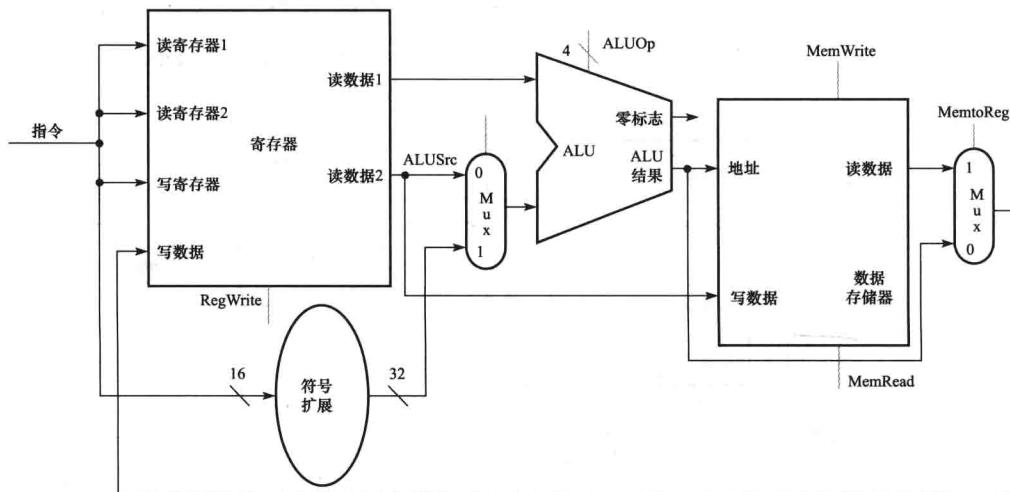


图 4-10 访存指令和 R 型指令数据通路的合并。这个例子说明了如何通过加入多选器将图 4-7 和图 4-8 合并成一个数据通路，其中增加了两个多选器  $\square$

现在，加上图 4-6 的取指数据通路、图 4-9 的分支数据通路、图 4-10 的 R 型指令和访存指令数据通路，我们可以把所有部件合并在一起建立一个简单的 MIPS 体系结构数据通路，如图 4-11 所示。由于分支指令用主 ALU 对寄存器操作数进行比较，所以还需要图 4-9 中的加法器完成分支目标地址的计算。此外还增加了一个多选器，用于选择是将顺序的指令地址 (PC + 4) 还是分支目标地址写入 PC。

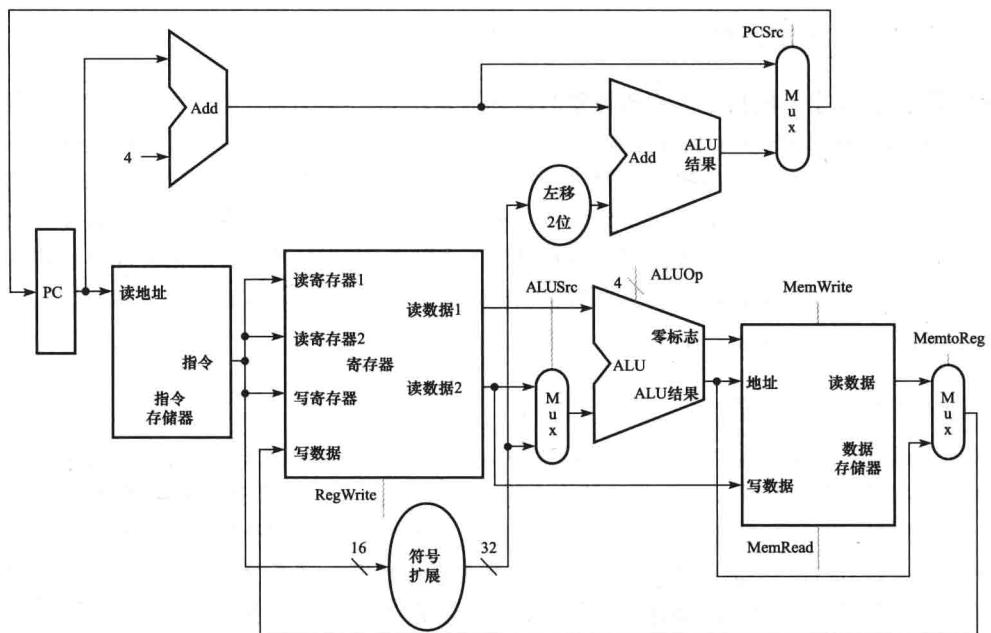


图 4-11 将不同类型指令所需的功能单元合并在一起实现的一个简单的核心 MIPS 系统结构数据通路。图中的部件来自图 4-6、图 4-9 和图 4-10。这个数据通路可以在一个时钟周期内完成基本的指令（存取字、ALU 操作和分支）。为了支持分支指令还增加了一个额外的多选器。对跳转指令的支持将在以后增加

在完成这个简单的数据通路后，可以加上控制单元。控制单元必须能够接收输入，能够产生每个状态单元的写信号、每个多选器的选择信号和 ALU 的控制信号。由于 ALU 的控制比较特殊，因此最好先设计 ALU，随后再设计控制单元的其他部分。

### 01 小测验

- I. 对取数指令来说，以下哪个是正确的？参考图 4-10。
- MemtoReg 信号线应该被设置为将存储器中的数据发送至寄存器堆。
  - MemtoReg 信号线应该被设置为将正确的目标寄存器的数据发送至寄存器堆。
  - 对取数指令而言，MemtoReg 信号线的设置无关紧要。
- II. 本节描述的单周期数据通路必须有独立的指令存储器和数据存储器，因为：
- MIPS 中指令与数据的格式是不同的，所以需要不同的存储器。
  - 使用独立的存储器会比较便宜。
  - 因为处理器在一个周期内只能操作每个部件一次，而在一个周期内不可能对一个单端口存储器进行两次存取。

257  
258

## 4.4 一个简单的实现机制

在本节中，我们将学习如何实现最简单的 MIPS 子集。我们用上一节的数据通路和增加一个简单的控制单元来实现一个 MIPS 体系结构。这一结构实现了取字 (lw)、存字 (sw)、相等则分支 (beq) 和算术逻辑指令加法 (add)、减法 (sub)、与运算 (AND)、或运算 (OR) 和小于则置位 (set on Less than)，后面我们还将实现跳转指令 (j)。

### 4.4.1 ALU 控制

附录 B 中描述的 MIPS ALU 在 4 位控制信号上定义了 6 种有效的输入组合：

ALU 控制信号	功能	ALU 控制信号	功能
0000	与	0110	减
0001	或	0111	小于则置位
0010	加	1100	或非

根据指令类型的不同，ALU 将执行上述 5 种功能中的一种。（或非操作在我们目前实现的 MIPS 子集中暂时没有使用。）对于取字和存储字指令，ALU 用加法计算存储器地址。对于 R 型指令，根据指令低 6 位的 funct 字段（见第 2 章），ALU 执行 5 种操作中的一种（与、或、减、加、小于则置位）。对相等则分支指令，由 ALU 执行减法操作。

259

使用一个小的控制单元即可生成 4 位的 ALU 控制信号，其输入为指令的 funct 字段和 2 位的 ALUOp 字段。ALUOp 指明要进行的操作是存取指令需要的加法 (00)、beq 需要的减法 (01)，还是由指令的 funct 字段 (10) 决定。该 ALU 控制单元输出 4 位信号，即前面介绍的 4 位控制信号，直接对 ALU 进行控制。

图 4-12 说明了怎样根据 2 位的 ALUOp 和 6 位的 funct 功能字段生成 ALU 的控制信号。在本章的后面将会看到怎样由主控制单元生成 ALUOp。

这种多级译码的方法（主控制单元生成 ALUOp 作为 ALU 控制单元的输入，再由 ALU 控制单元生成真正控制 ALU 的信号）是一种常用的实现方式。使用多级译码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对时钟周期非常关键。

指令操作码	ALUOp	指令操作	funct 字段	ALU 动作	ALU 控制信号
取字	00	取字	XXXXXX	加	0010
存字	00	存储字	XXXXXX	加	0010
相等则分支	01	相等分支	XXXXXX	减	0110
R 类型	10	加	100000	加	0010
R 类型	10	减	100010	减	0110
R 类型	10	与	100100	与	0000
R 类型	10	或	100101	或	0001
R 类型	10	小于则置位	101010	小于则置位	0111

图 4-12 如何根据 ALUOp 控制位和 R 型指令的 funct 字段设置 ALU 的控制信号。第一列是操作码，操作码决定了 ALUOp 位。所有的编码以二进制给出。注意，当 ALUOp 为 00 或 01 时，ALU 的动作不依赖于 funct 字段，故功能字段记为 XXXXXX。当 ALUOp 为 10 时，funct 字段用于设置 ALU 的控制信号。详情见附录 B

260 有多种不同方法把 2 位的 ALUOp 和 6 位的 funct 字段映射为 4 位 ALU 控制信号。因为 funct 功能字段的 64 种可能取值中只有很小一部分有意义，并且只有当 ALUOp 取值为 10 时才使用功能字段，我们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 控制信号。

为设计这个逻辑单元，有必要为 ALUOp 和 funct 字段有意义地组合生成一张真值表，如图 4-13 所示。该真值表（truth table）说明了如何根据两个输入字段得到 4 位的 ALU 控制信号。由于完整的真值表很大 ( $2^8 = 256$  项)，我们并不关心所有的输入组合，只列出了使 ALU 控制信号有效的部分表项，而忽略那些恒为 0 或无关的项。在本章中，我们将一直采用这样的方式表示真值表（这样做的缺点在附录 D 的 D.2 节中讨论）。

由于在许多情况下对某些输入的取值并不关心，为了简化真值表，我们也列出无关项（don't-care term）。真值表中的无关项（在输入列中用 X 表示）表明，输出与该列对应的输入取值无关。如图 4-13 的第一列所示，当 ALUOp 取 00 时，无论 funct 字段取何值，ALU 控制总为 0010。这时，真值表中此行的 funct 字段就是无关项。在后面，还会有另一种无关项的例子。无关项的概念在附录 B 中有更多的讨论。

- ② 真值表：逻辑操作的一种表示方法，即列出输入的所有情况和每种情况下的输出。
- ② 无关项：逻辑函数的一个元素，表示输出与该输入取值无关。无关项可以用不同的方式指定。

真值表建好以后，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 D 中的 D.2 节讨论。

ALUOp		Funct 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 4-13 4 位 ALU 控制信号（称为操作）的真值表。该真值表的输入为 ALUOp 和 funct 字段。在此只列出了 ALU 控制有效的项，也包括一些无关项。例如，ALUOp 不使用编码 11，故真值表包含 1X 和 X1 项，而不是 10 和 01 项。同样，当使用 funct 字段时指令的前两位（F4 和 F5）总是 10，所以它们是无关项，在真值表中用 XX 代替

#### 4.4.2 主控制单元的设计

我们已经描述了如何使用 funct 和 2 位信号作为输入来进行 ALU 控制单元的设计，现在来看看控制的其他部分。在开始之前，首先看一条指令的各个字段和图 4-11 所示的数据通路所需的控制信号。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，如图 4-14 所示。

261

字段	0	rs	rt	rd	shamt	funct
比特位位置	31:26	25:21	20:16	15:11	10:6	5:0
a) R型指令						
字段	35 or 43	rs	rt	rd	shamt	地址
比特位位置	31:26	25:21	20:16	15:11	10:6	5:0
b) 存取指令						
字段	4	rs	rt	rd	shamt	地址
比特位位置	31:26	25:21	20:16	15:11	10:6	5:0
c) 分支指令						

图 4-14 三种指令类型（R 型、存取和分支）使用的两种指令格式。后面我们马上会讲到，跳转指令使用另一种格式。a) R 型指令的格式，操作码为 0，寄存器操作数有 3 个：rs、rt 和 rd。rs 和 rt 字段为源操作数，rd 字段为目的操作数。funct 字段指出 ALU 功能，由前面设计的 ALU 控制单元译码。我们实现的 R 型指令有 add、sub、AND、OR 和 slt。shamt 字段只用于移位指令，本章中暂不考虑。b) 取数指令（操作码 =  $35_{10}$ ）和存储指令（操作码 =  $43_{10}$ ）的格式。rs 寄存器作为基址与 16 位的地址字段相加以得到访存地址。对取数指令，rt 是取出数据的目的寄存器。对存储指令，rt 是要存入存储器的数据所在的寄存器。c) 相等则分支指令（操作码 = 4）的格式。rs 和 rt 是源寄存器，用于比较是否相等。16 位地址进行符号扩展、移位后与 PC + 4 相加以得到分支目标地址

MIPS 的指令格式遵循以下规则：

- Op 字段，第 2 章亦称操作码（opcode），总是为 31:26 位。我们将用  $Op[5:0]$  来表示。
- 对于 R 型指令、分支指令和存取指令，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。
- 存取指令的基址寄存器在 25:21 位中（rs 字段）。
- 相等则分支指令、存取指令的 16 位偏移量在 15:0 位中。
- 有两个地方存放目标寄存器。对取数指令为 20:16 位（rt 字段），对 R 型指令为 15:11 位（rd 字段）。所以我们需要一个多选器，以指示要写的寄存器号在哪个字段中。

#### 操作码：指示指令操作和格式的字段。

从第 2 章得到的第一个设计原则——简单源于规整——在这里就体现出来了。

262

根据上述信息，可以给简单的数据通路加上指令标记并增加一个多选器（用于选择寄存器堆的写寄存器号），图 4-15 展示了这些增加的部件和 ALU 控制块、状态单元的写信号、数据存储器的读信号和多路选择器的控制信号。由于所有的多路选择器都是两个输入端，因此每个多路选择器都需要一条单独的控制线。

图 4-15 给出了 7 个 1 位控制信号和 2 位 ALUOp 控制信号。我们已经说明了 ALUOp 控制信号如何工作，在继续说明指令执行过程中如何设置这些控制信号之前，最好定义一下其他 7 条控制信号如何工作。图 4-16 说明了这 7 个控制信号的功能。

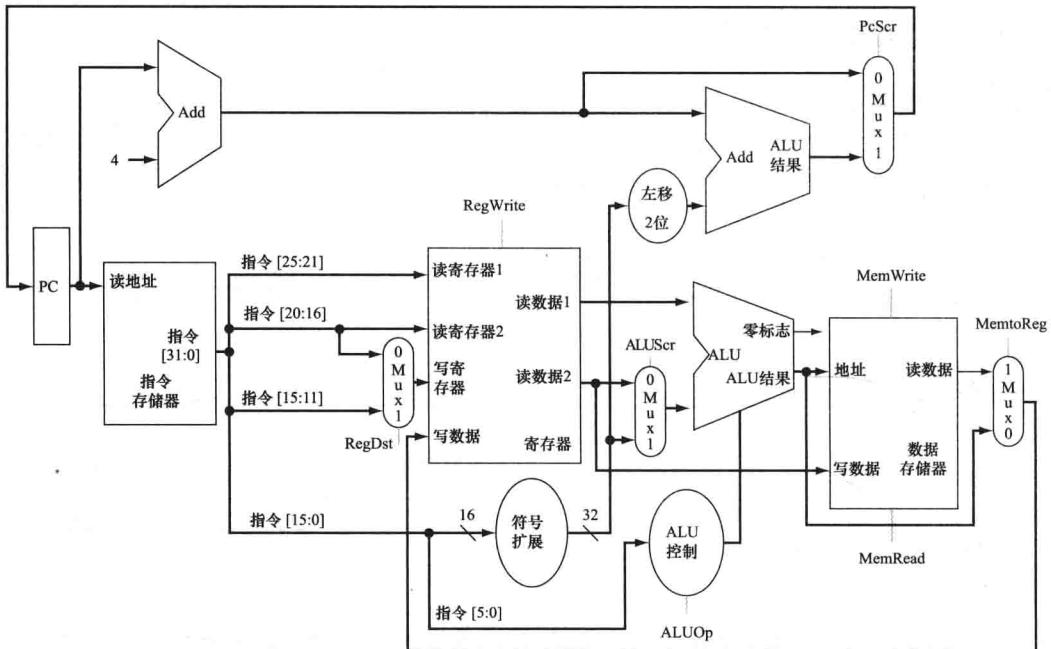


图 4-15 在图 4-11 的数据通路上增加了所有必需的多选器，并标识出了所有的控制信号。控制信号以灰色线表示。还增加了 ALU 控制单元。PC 不需要写控制，因为它在每个时钟周期末都被写入一次。分支控制逻辑决定给 PC 自增还是写入分支目标地址

控制信号名	无效时的含义	有效时的含义
RegDst	写寄存器的目标寄存器号来自 n 字段（位 20: 16）	写寄存器的目标寄存器号来自 rd 字段（位 15:11）
RegWrite	无	寄存器堆写使能有效
ALUSrc	第二个 ALU 操作数来自寄存器堆的第二个输出（读数据 2）	第二个 ALU 操作数为指令低 16 位的符号扩展
PCSrc	PC 由 $PC + 4$ 取代	PC 由分支目标地址取代
MemRead	无	数据存储器读使能有效
MemWrite	无	将写入数据输入端的数据写入到用地址指定存储器单元中取
MemtoReg	写入寄存器的数据来自 ALU	写入寄存器的数据来自数据存储器

图 4-16 7 个控制信号的作用。当两路多选器的控制信号有效时，选择第 1 个输入，否则选择第 0 个输入。需要注意的是，所有状态单元都有一个默认输入——时钟信号，且用于写操作的控制。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 中对此问题有进一步的讨论。）

了解了每个控制信号的功能之后，再来看看它们如何设置。除 PCSrc 控制信号外，所有控制信号都可由控制单元只根据指令的操作码来确定。而 PCSrc 信号有效的条件是指令为相等则分支（由控制单元确定），且用于等值比较的 ALU 的零输出有效。为生成 PCSrc 信号，需将一个来自控制单元称为“Branch”（分支）的信号与 ALU 的零输出信号相“与”。

现在，这 9 位控制信号（图 4-16 的 7 位和 2 位 ALUOp）的状态可根据控制单元的 6 位输入信号（操作码位 31:26）来设置。图 4-17 给出了包含控制单元和控制信号的数

据通路。

在设计控制单元的计算公式或真值表之前，这里定义一下控制功能。由于控制信号的状态仅由操作码决定，我们需要定义在每种操作码下每个控制信号的取值：0、1 或任意值 X。根据图 4-12、图 4-16 和图 4-17，图 4-18 定义了对应于每种操作码的控制信号状态。

## 1. 数据通路的操作

根据图 4-16 和图 4-18 包含的信息，可以设计出控制单元逻辑，但在此之前先分析一下每条指令是如何使用数据通路的。接下来的一些图说明了 3 种类型的指令在数据通路上的执行过程。在这些图中，有效的控制信号和数据通路部件已着重标出。需要注意的是，对于多选器其控制为 0 时，即使其控制信号没有着重标出，它也有相应的动作。对于多位信号，只要其中任何信号有效，就将其着重标出。

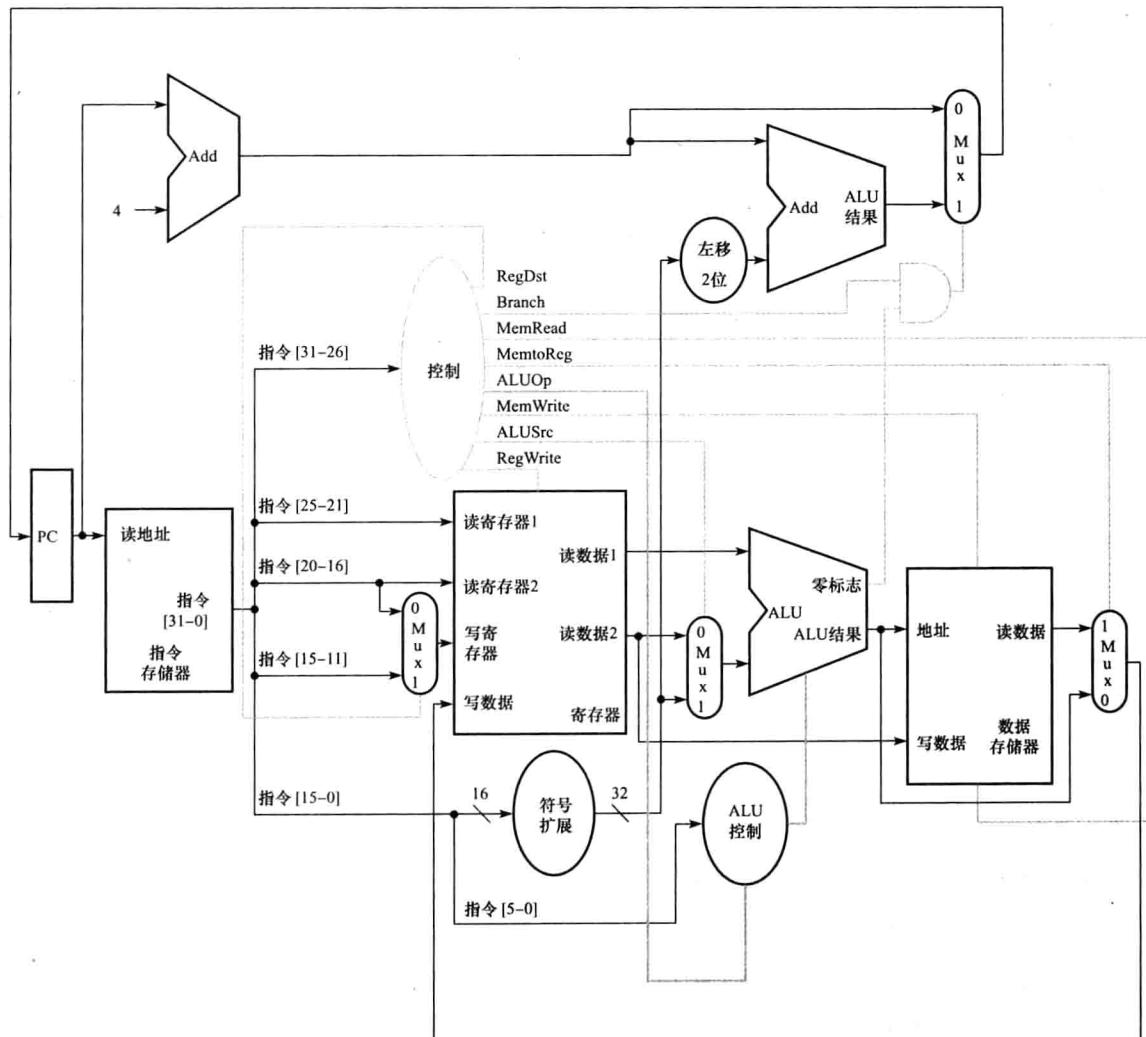


图 4-17 包含控制单元的简单数据通路。控制单元的输入为指令的 6 位操作码。控制单元的输出包括 3 个控制多选器的 1 位信号 (RegDst、ALUSrc 和 MemtoReg)，3 个控制寄存器堆和存储器读写的信号 (RegWrite、MemRead 和 MemWrite)，一个决定是否可以转移的 1 位信号 (Branch)，和一个 ALU 的 2 位控制信号 (ALUOp)。分支控制信号与 ALU 的零输出一起送入一个与门，其输出控制下一个 PC 的选择。注意 PCSrc 是一个间接信号，而不是从控制单元直接得来。所以在图中我们没有标出这个信号名称

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 4-18 按指令操作码设置的控制信号。表的第一行对应于 R 型指令 (add、sub、AND、OR 和 slt)：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 Branch 控制信号为 0 时，PC 无条件地由 PC + 4 取代；反之，如果 ALU 的零输出也为高，则 PC 由分支目标地址取代。当 R 型指令的 ALUOp 为 10 时，ALU 的控制信号应由 funct 字段生成。本表的第二行和第三行给出了 lw、sw 指令的控制信号设置：ALUSrc 和 ALUOp 被设为进行地址计算；MemRead 和 MemWrite 被设为进行存储器访问；最后，RegDst 和 RegWrite 被设为在装载指令中将结果存入寄存器 rt 中。分支指令与 R 型指令相似，因为它将寄存器 rs 和 rt 送入 ALU：分支指令的 ALUOp 字段被设为进行减法 (ALUOp = 01)，以进行等值的测试。注意，RegWrite = 0 时 MemtoReg 的设置无关紧要——因为寄存器没有被写入，寄存器写端口的数据值不被使用，所以最后两行 MemtoReg 的值由于不被关心而用 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 4-19 给出了执行 R 型指令 (如 add \$t1, \$t2, \$t3) 时的数据通路操作。尽管一切都发生在一个时钟周期内，但我们可以考虑分 4 步来执行指令，具体如下：

- 1) 从指令存储器中取出指令，PC 自增。
- 2) 从寄存器堆中读出寄存器 \$t2 和 \$t3。同时，主控制单元计算出各控制信号的状态。
- 3) ALU 根据 funct 字段 (指令的 5:0 位) 确定 ALU 的功能，对从寄存器堆读出的数据进行操作。
- 4) 将 ALU 的结果写入寄存器堆，根据指令的 15:11 位选择目标寄存器 (\$t1)。

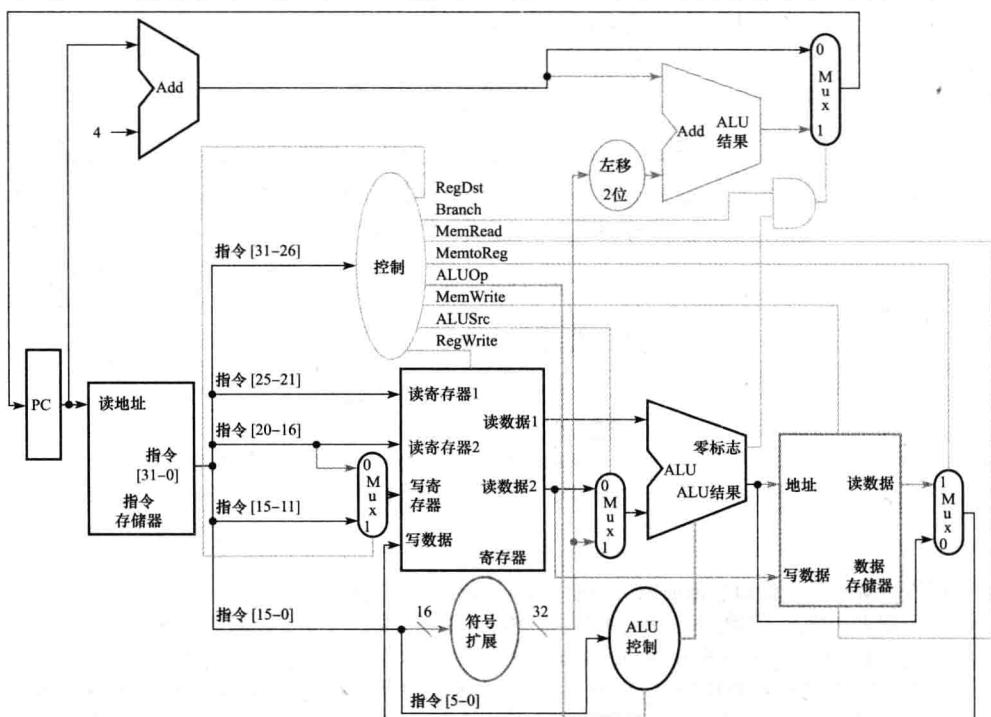


图 4-19 执行 R 型指令 (如 add \$t1, \$t2, \$t3) 时数据通路的操作。操作中用到的控制信号、数据通路单元和连接均用灰色线显示

我们可以用和图 4-19 类似的方式描述取数指令（如 `lw $t1, offset($t2)`）的执行。图 4-20 给出了取数时有效的功能单元和控制信号。可以考虑将取数指令的执行分为 5 步（与将 R 型指令的执行分为 4 步类似）：

- 1) 从指令存储器取指，PC 自增。
- 2) 从寄存器堆读出寄存器  $\$t_2$  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 位值 (`offset`) 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 ( $\$t_1$ ) 指出。

264  
266

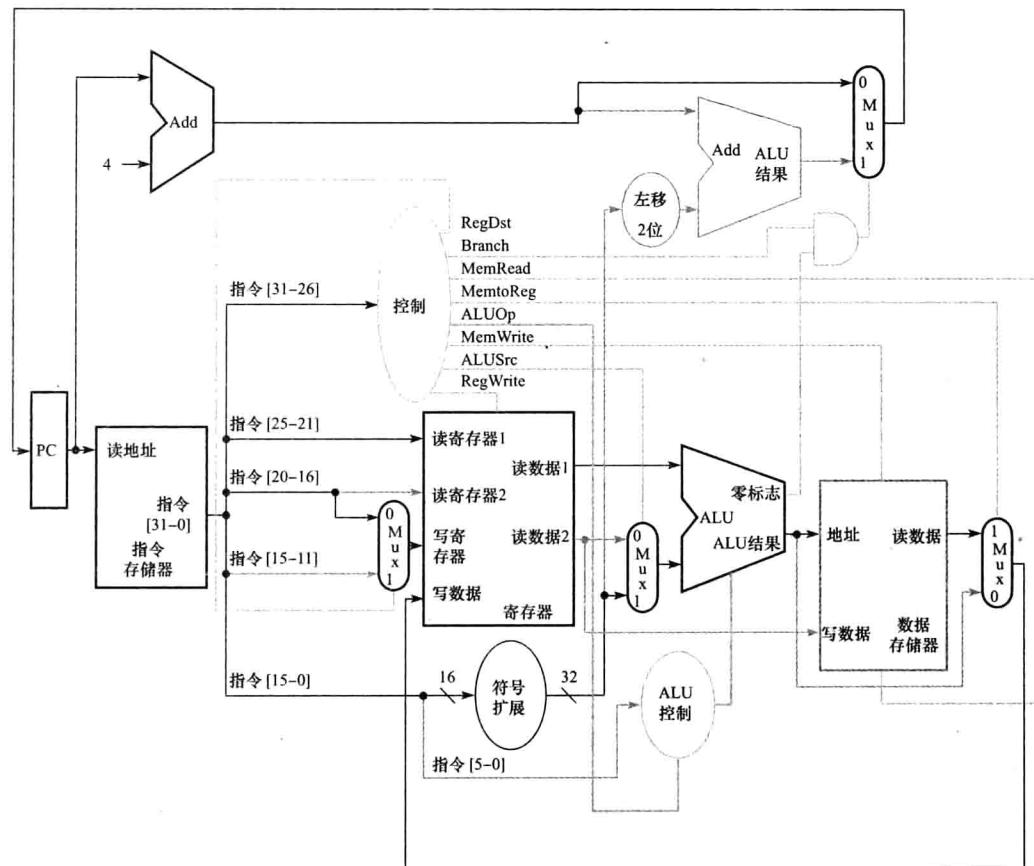


图 4-20 执行取数指令时数据通路的操作。操作中用到的控制信号、数据通路单元和连接用灰色线显示。存储指令的操作与此类似。主要区别在于数据存储器的控制将指明要进行写操作，而不是读操作，读出的第二个寄存器的值将作为要存储的数据，并且不会有将数据存储器的内容写入寄存器的操作。

最后，以同样方式说明相等则分支指令（如 `beq $t1, $t2, offset`）的执行过程。它的操作类似于 R 型指令，但 ALU 的零输出用于决定 PC 自增为  $PC + 4$  还是置为分支目标地址。图 4-21 给出了执行的 4 步：

- 1) 从指令存储器中取指，PC 自增。
- 2) 从寄存器堆读出寄存器  $\$t_1$  和  $\$t_2$  的值。
- 3) ALU 将从寄存器堆读出的两数相减。 $PC + 4$  的值与符号扩展并左移 2 位后的指令低 16 位 (`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。

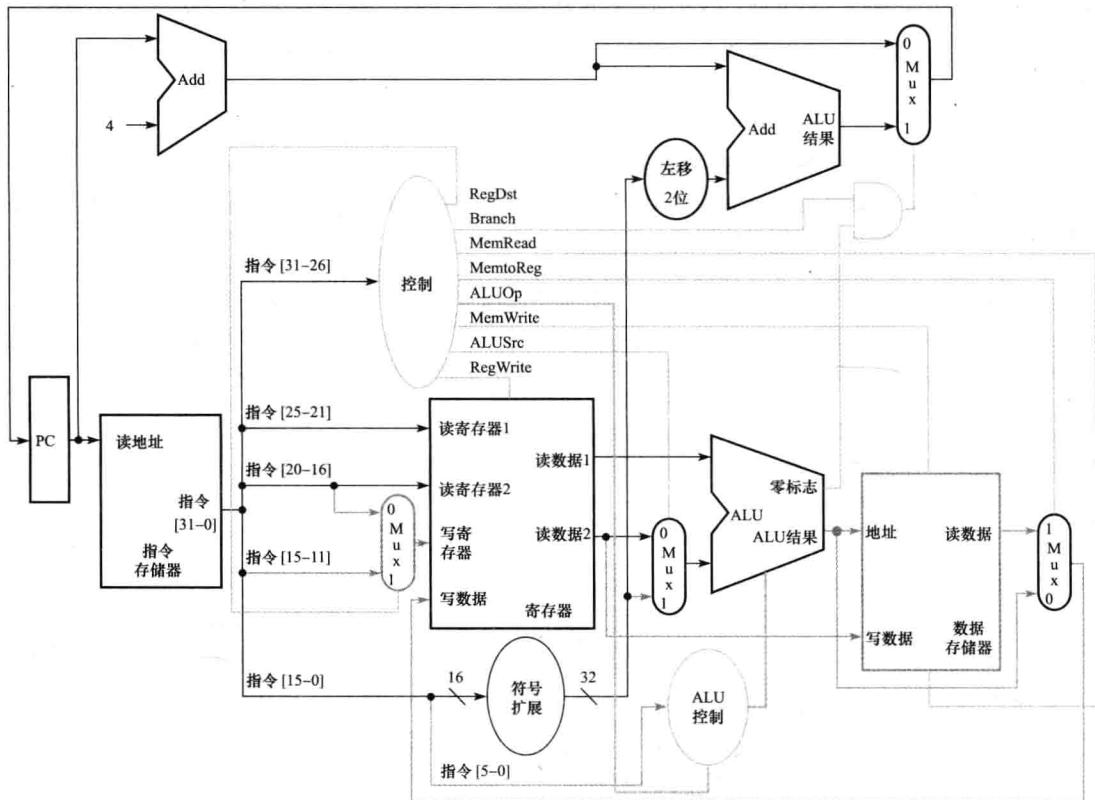


图 4-21 执行相等则分支指令时数据通路的操作。控制信号、数据通路单元和连接使用灰色线显示。在使用寄存器堆和 ALU 进行比较操作之后，ALU 的零输出用于在两种可能的 PC 中选择其一。

## 2. 控制的结束

讨论过指令的操作之后，现在继续讨论控制单元的实现。控制单元的功能可由图 4-18 精确定义，其输入为 6 位操作码  $Op[5:0]$ ，输出为控制信号。这样，可以基于操作码的二进制编码为每个输出建立一张真值表。

根据这些信息，可以把控制单元（包括所有输出的逻辑综合）描述在一张大的真值表中，如图 4-22 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，附录 D 的 D.2 节对此进行了描述。

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

图 4-22 简单的单周期实现的控制功能真值表。表的上半部分为输入，其包括操作码（对应于指令的 31:26 位的  $Op[5:0]$ ）的 4 种组合。表的下半部分为 4 种组合的输出。因此，RegWrite 对于两种不同的输入组合是有效的。如果只考虑这张表中的 4 个操作码，则可以用输入部分的无关项简化真值表。例如，可以由表达式  $Op5 \cdot Op2$  确定是否为 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。之所以不用这种简化，是因为在 MIPS 指令集的完整实现中会用到其他操作码。

输入或输出	信号名	R型	lw	sw	beq
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 4-22 (续)

既然我们已经有了包含 MIPS 核心指令集中绝大多数指令的单周期实现 (single-cycle implementation)，在此基础之上我们再加上跳转指令，看看怎样通过扩展基本数据通路和控制通路，来实现指令集中的其他指令。

- 单周期实现：也被称为单时钟周期实现 (single clock cycle implementation)，即一个时钟周期执行一条指令的实现机制。虽然它很容易理解，但现实中，由于它太慢而不实用。

### 01 例题·跳转的实现

图 4-17 给出了第 2 章中提到的许多指令的实现，但没有给出跳转指令的实现。请对图 4-17 的数据通路和控制通路进行扩展，从而支持跳转指令，并给出控制信号的设置方式。

### 01 答案

跳转指令类似于分支指令，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，跳转地址的最低两位恒为  $00_2$ 。32 位跳转地址的次低 26 位来自指令的 26 位立即数，如图 4-23 所示。跳转地址的高 4 位来自于跳转指令的  $PC + 4$ 。也就是说，实现跳转指令即将下面 3 个部分拼接为跳转地址：

- 当前  $PC + 4$  的高 4 位（下条指令地址的 31:28 位）。
- 跳转指令的 26 位立即数字段。
- 低位  $00_2$ 。

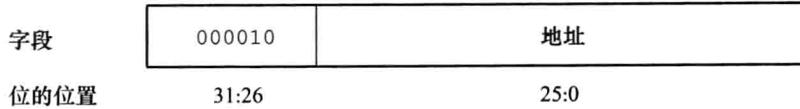


图 4-23 跳转指令的格式（操作码 = 2）。跳转指令的目的地址由当前  $PC + 4$  的高 4 位与跳转指令中的 26 位地址连接，再将 00 作为最低两位形成

图 4-24 所示为在图 4-17 基础上增加了对跳转指令的支持。为了在  $PC + 4$ 、分支目标 PC 和跳转目标 PC 中选择新 PC 值的来源，加上了一个多选器。这个多选器需要一个控制信号 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。

□

270

### 4.4.3 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工作，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，这样时钟周期要由执行时间最长的那条指令决定。这条指令几乎可以肯定是取数指令，它依次使用了 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（见第 1 章），单周期实现方式的总体性能并不一定很好，因为时钟周期实在是太长了。

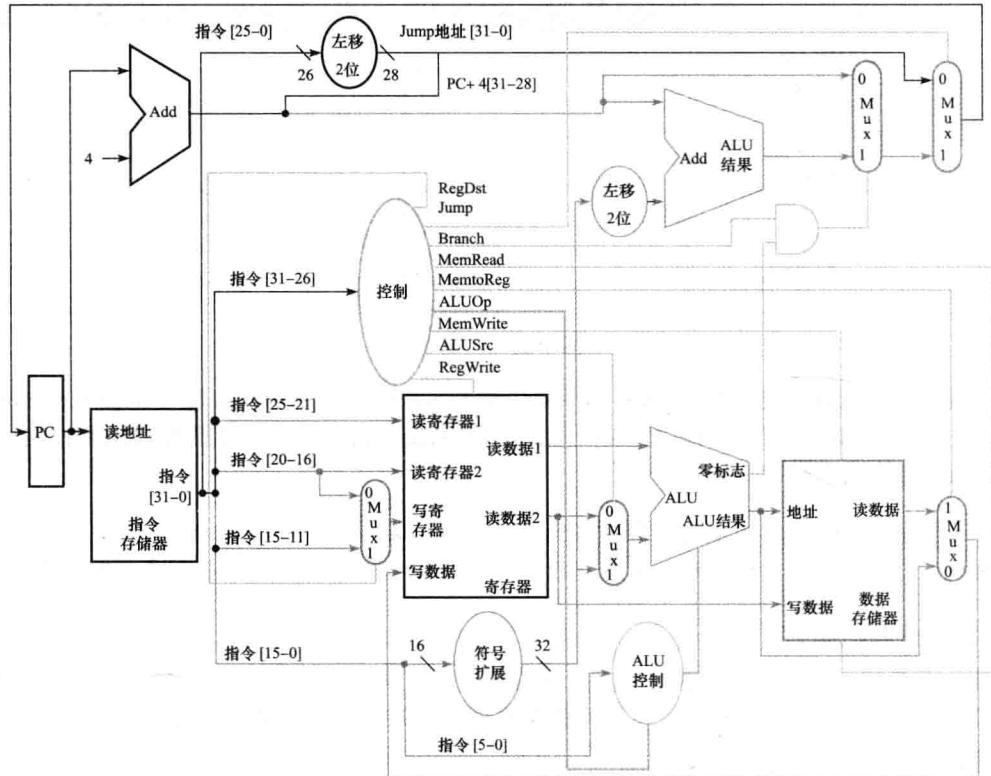


图 4-24 扩展后能处理跳转指令的简单控制和数据通路。加入了一个多选器（右上角）用来选择分支目标地址、跳转目标地址和下一指令地址三者之一。该多选器由 Jump 信号控制。跳转目标地址通过将 jump 指令中低 26 位地址左移两位，从而高效地增加 00 作为低位，然后将 PC + 4 的高 4 位作为高位，从而产生 32 位地址。

使用单周期设计的代价虽然很大，但对于小指令集来说，或许是可以接受的。事实上，早期具有简单指令集的计算机就曾经采用过这种实现方式。然而，若要实现包含浮点或更复杂指令的指令集，这样的单周期设计根本不能胜任。

因为时钟周期必须满足所有指令中最坏的情况，故不能使用那些缩短常用指令执行时间而不改善最坏情况的实现技术。这样，单周期实现方式违背了第 1 章中加速大概率事件这一设计原则。

在下一节，我们将看到一种称为流水线的实现技术，使用与单周期类似的数据通路，但效率更高。从方法来说，流水线是通过重叠多条指令的执行来提高效率的。

### 01 小测验

观察图 4-22 中的控制信号，你能在其基础上进行整合吗？其中是否有控制信号可以被其他控制信号取反来替代（提示：将无关项考虑进去）？如果有，不加反向器是否可以直接用一个控制信号替代另一个呢？

## 4.5 流水线概述

绝对不要浪费时间。

——美国谚语

流水线（pipelining）是一种实现多条指令重叠执行的技术。目前，流水线技术广泛应用。

## ② 流水线：一种实现多条指令重叠执行的技术，与生产流水线类似。

本节对流水线的概念及其相关问题进行了概述。如果只是想对流水线技术有一个大致的了解，可以详细看完本节，然后直接跳到 4.10 节和 4.11 节学习在最近的处理器（Intel Core i7 和 ARM Cortex-A8）中所使用的高级流水线技术。如果想深入了解基于流水线技术的计算机，4.6~4.9 节给出了相关细节。

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

- 1) 把一批脏衣服放入洗衣机里清洗。
- 2) 洗衣机洗完后，把衣服取出并放入烘干机中。
- 3) 烘干衣服后，将衣服从烘干机中取出，然后放在桌子上叠起来。
- 4) 叠好衣服后，请你的室友帮忙把桌子上的衣服收好。

当你的室友把这批干净衣服从桌子上拿走后，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 4-25 所示。当把第一批脏衣服从洗衣机里取出放入烘干机之后，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。接着让你的室友把第一批衣服从桌子上收好，而你开始叠第二批衣服，这时烘干机中放的是第三批衣服，同时可以把第四批脏衣服放入洗衣机清洗了。这样，所有的洗衣步骤（流水线的步骤）都在同时操作。只要在每一个操作步骤中都有独立的工作单元时，我们就可以采用流水线的方式来快速完成任务了。

272

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在并行地进行。因此，单位时间内能够完成的工作量就大大地增加了。流水线实际上是改善了洗衣系统的吞吐率。虽然洗每一件衣服的时间没有缩短，但如果有很多衣服要洗，吞吐率的改善就减少了完成整个工作的时间。

如果所有的步骤所需的时间一样，并且有足够的工作可做，那么从流水线得到的速度提高倍数等于流水线中步骤的数目，洗衣房的例子是 4 倍：清洗、烘干、折叠和收拾。采用流水线方式工作与非流水线方式工作的洗衣房相比在速度上提高了 4 倍：前者洗完 20 批衣服所需的时间是洗完一批衣服所需时间的 5 倍，而后者洗完 20 批衣服所需的时间是洗完一批衣服的 20 倍。在图 4-25 中，流水线方式只将处理速度提高了 2.3 倍的原因是图中只显示了清洗 4 批衣服的处理过程。注意图 4-25 中的流水线版本在开始和结束阶段的负载情况，可以看出其流水线未完全充满。当任务数量相对于流水线级数不是很大时，突然启动和逐渐结束会影响流水线的性能。在本例中，如果任务数量远大于 4，那么绝大多数时候流水线都将是充满的，这时吞吐率的提升就非常接近于 4 倍。

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指令。通常，一个 MIPS 指令包含如下 5 个处理步骤：

- 1) 从指令存储器中读取指令。
- 2) 指令译码的同时读取寄存器。MIPS 的指令格式允许同时进行指令译码和读寄存器。
- 3) 执行操作或计算地址。
- 4) 从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 流水线具有 5 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体执行时间。

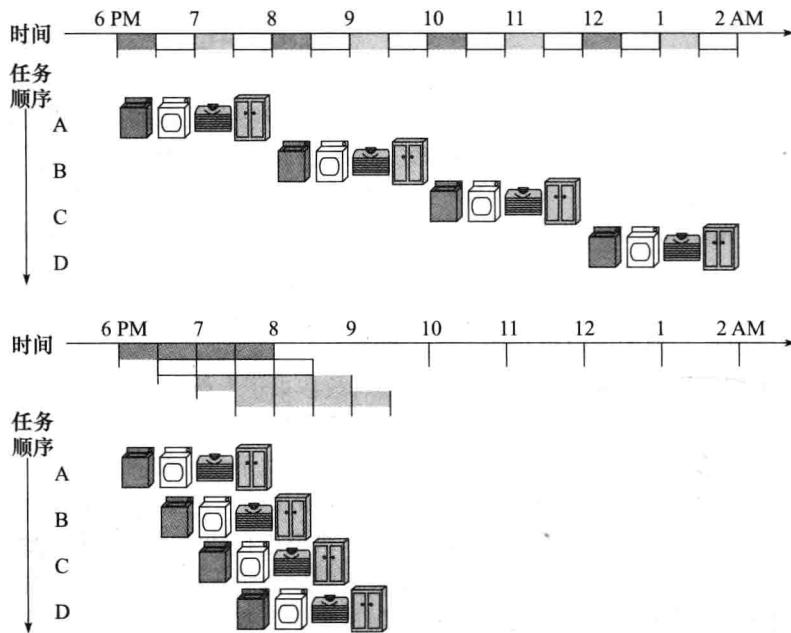


图 4-25 以洗衣店为例类比流水线的动作过程。安妮、布朗、凯西和唐每个人都有一些脏衣服要清洗、烘干、折叠及收拾。洗衣机、烘干机、“折叠机”和“收拾机”每个都需要 30 分钟来完成各自的任务。顺序的洗涤方法将花费 8 小时的时间洗完 4 批衣服，而流水线的洗涤方法只需要花费 3.5 小时。图中在二维时间线中通过资源的 4 次复制给出了不同工作负载的流水级，事实上每种资源只有一个

### 01 例题·单周期指令模型与流水线性能

为了使问题具体化，我们首先创建一个流水线结构。在本例以及本章剩余的部分中，我们将只考虑以下 8 条指令：取字 (lw)、存字 (sw)、加 (add)、减 (sub)、与 (AND)、或 (OR)、小于则置位 (slt) 和相等则分支 (beq)。

本例将比较流水线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。假设主要功能单元的操作时间为存储器访问：200ps；ALU 操作：200ps；寄存器堆的读写：100ps。在单周期模型中，每一条指令都只花费一个时钟周期，因此，时钟周期必须满足最慢的指令。

273  
l  
274

### 01 答案

8 条指令中每一条指令所需要的执行时间如图 4-26 所示。单周期模型的设计必须考虑到最慢的指令，在图 4-26 中是 lw，因此，每一条指令所需要的执行时间为 800ps。与图 4-25 类似，图 4-27 比较了三条取数指令非流水线与流水线方式的执行过程，其中在非流水线方式中，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 操作	数据存取	写寄存器	总时间
取字 (lw)	200ps	100ps	200ps	200ps	100ps	800ps
存字 (sw)	200ps	100ps	200ps	200ps		700ps
R 型 (add, sub, AND, OR, slt)	200ps	100ps	200ps		100ps	600ps
分支 (beq)	200ps	100ps	200ps			500ps

图 4-26 根据各功能单元所需时间计算出来的每条指令的总执行时间。假设多选器、控制单元、PC 访问和符号扩展单元都没有延时

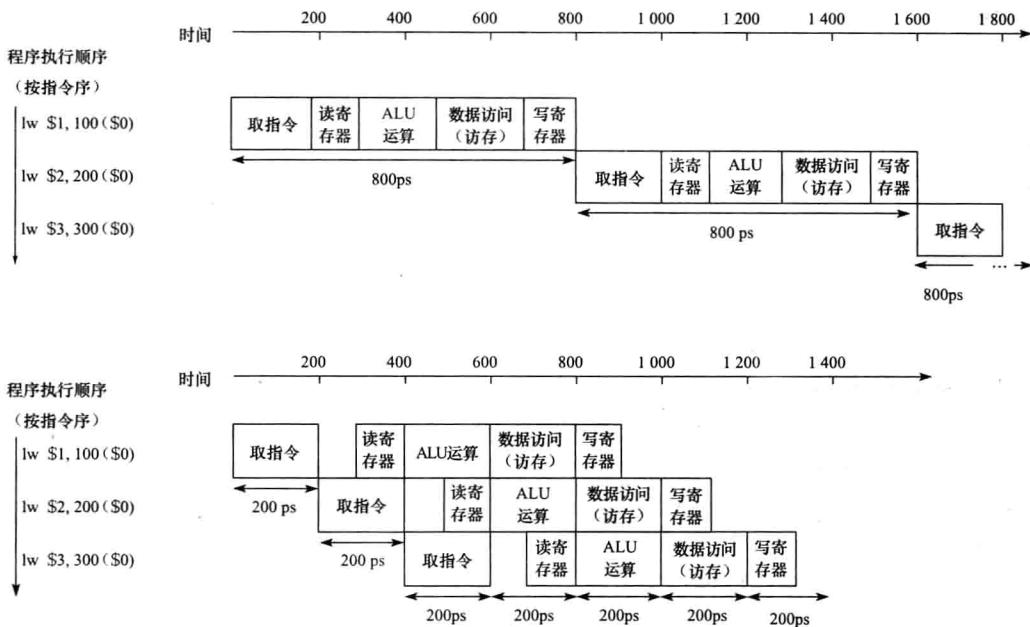


图 4-27 单周期、非流水线的指令执行过程（上图）与流水线的指令执行过程（下图）。两者采用相同的功能单元，各功能单元的处理时间如图 4-26 所示。在这种情况下，指令的执行速度提高了 4 倍，即从 800ps 降到了 200ps。将本图与图 4-25 比较。在洗衣服的例子中，我们假设所有步骤需要的处理时间都是相等的。如果烘干机运行得最慢，那么就把烘干的时间设定为一个步骤需要的处理时间。计算机流水线的处理时间也受限于最慢的处理步骤，即 ALU 操作和存储器访问。同时我们假设对寄存器堆的写操作发生在时钟周期的前半段，对寄存器堆的读操作发生在时钟周期的后半段，本章后面将一直遵循这个假设 □

所有的流水级 (pipeline stage) 都只花费一个时钟周期的时间，因此，时钟周期必须能够满足最慢操作的执行需要。这就像在单周期模型中虽然有些快的指令的执行只需要 500ps，但它必须选择在最坏情况下的 800ps 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下的 200ps 而不是有些步骤可以达到的 100ps。流水线能够将性能提高 4 倍：第一与第四条指令之间的时间差距缩短为  $3 \times 200\text{ps} = 600\text{ps}$ 。

我们可以把上面讨论的流水线模型能够获得的性能加速比归纳成一个公式。如果流水线各阶段操作平衡，那么在流水线机器上的指令执行时间为（在理想情况下）：

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

即在理想情况且有大量指令的情况下，流水线所带来的加速比与流水线的级数近似相同。例如一个 5 级流水线能获得的加速比接近于 5。

这个公式说明一个 5 级流水线在 800ps 的非流水线执行时间的基础上获得接近 5 倍的速度提高，即相当于 160ps 的时钟周期。然而，在例子中显示，各级间并不是完全平衡的。另外，流水线引入了一些开销，开销的来源问题稍后会更加清楚。所以，在流水线机器中每一条指令的执行时间会超过这个最小的可能值，因此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指令的执行速度提高 4 倍，但在本例中并没有反映出来，它实际获得的加速比为 2400ps/1400ps，这是因为执行指令的数量不够多。如果增加执行指令的数目将会发生什么呢？我们首先将前面图中的指令增加到 1 000 003 条，也就是说，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 200ps，因此，整个的执行时间就变成  $1 000 000 \times 200\text{ps} + 1400\text{ps}$ ，即 200 001 400ps。在非流水线的例子中，我们也加入 1 000 000 条指令，每条指令的执行时间是 800ps，因此整个的执行时

275  
276

间为  $1\ 000\ 000 \times 800\text{ps} + 2\ 400\text{ps}$ , 即  $800\ 002\ 400\text{ps}$ 。在这些条件下, 非流水线程序与流水线程序的实际执行时间的比值就非常接近于两者指令平均执行时间的比值, 即为

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

流水线所带来的性能提高是通过增加指令的吞吐率, 而不是减少单条指令的执行时间实现的。由于实际程序都会执行成千上万条指令, 因此, 指令的吞吐率是一个很重要的参数。

#### 4.5.1 面向流水线的指令集设计

尽管上面的例子只对流水线进行了最简单的说明, 我们也能够通过它讨论面向流水线执行的 MIPS 指令集的设计。

第一, 所有的 MIPS 指令的长度都是相同的。这一限制简化了流水线的第一级取指与第二级译码。在诸如 x86 之类的指令集中, 指令的长度并不相同, 从 1 字节到 15 字节不等, 这样将会给流水线的执行带来更大的挑战。最近的 x86 体系结构实现实际上是将 x86 指令转化成类似 MIPS 指令的简单操作, 然后再将这些简单操作进行流水, 而不是直接对原始的 x86 指令流水! (见 4.10 节)

第二, MIPS 只有很少的几种指令格式, 并且每一条指令中的源寄存器字段位置都是相同的。这种对称性意味着流水线的第二级在确定取指类型的同时就能够开始读寄存器堆。如果 MIPS 的指令格式是非对称的, 我们就需要将第二级一分为二, 从而使得流水线的级数变为 6。稍后我们将看到长流水线的缺点。

第三, MIPS 中的存储器操作数仅出现在存取指令中。这一限制意味着可以利用执行级计算存储器地址, 然后可以接着在下一级访问存储器。如果可以直接操作内存中的操作数 (就像在 x86 中那样), 那么第三级与第四级将会扩展为地址计算、存储访问和执行阶段。

第四, 如第 2 章所述, 所有操作数必须在存储器中对齐。因此, 我们不需要担心一个数据传输指令需要访问两次存储器的情况, 所请求的数据可以在一级流水线内在处理器与存储器之间完成传输。

#### 4.5.2 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)。我们将介绍三种流水线冒险。

##### 1. 结构冒险

第一种冒险叫作结构冒险 (structural hazard)。即硬件不支持多条指令在同一时钟周期执行。在洗衣店的例子中, 如果用洗衣烘干一体机代替独立的洗衣机与烘干机, 或者如果你的室友正在做其他的事情而不能帮助你将衣服收拾好, 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受到破坏。

277

结构冒险: 因缺乏硬件支持而导致指令不能在预定的时钟周期内执行的情况。

正如我们在上面所说的那样, MIPS 的指令集是为流水线设计的。因此, 它就使得设计者在设计流水线时能够非常容易地避免结构冒险。假设图 4-27 的流水线结构只有一个存储器而不是两个存储器, 那么如果有第四条指令的话, 第一条指令在访问存储器的同时第四条指令将会在同一存储器中预取指令, 流水线就会发生结构冒险。

##### 2. 数据冒险

数据冒险 (data hazard) 发生在由于一条指令必须等待另一条指令的完成而造成流水线暂停的情况下。假设你在折叠衣服时发现有一只短袜找不到与之配对的另一只。你可能做的是下

楼到你的房间，在衣橱中找，看是否能找到另一只。很明显，当你在找的时候，已经烘干且正需要折叠的衣服以及已经洗完且正需要烘干的衣服不得不搁置一边。

- 数据冒险：也称为流水线数据冒险（pipeline data hazard），即因无法提供指令执行所需数据而导致指令不能在预定的时钟周期内执行的情况。

在计算机流水线中，数据冒险是由于一条指令依赖于更早的一条还在流水线中的指令造成的（这是一种在洗衣店例子中不存在的情况）。例如，假设有一条加法指令，它之后紧跟着一条减法指令，而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，这就意味着在流水线中浪费了三个时钟周期。

虽然可以试图通过编译器来避免这种数据冒险的发生，但实际上这种努力很难令人满意。因为这种冒险的发生过于频繁而且导致的延迟太长，因此不可能指望编译器把我们从这种困境当中解脱出去。

一种最基本的解决方法是基于以下发现：在解决数据冒险问题之前不需要等待指令的执行结束。对于上述的代码序列，一旦 ALU 生成了加法运算的结果，就可以将它用作减法运算的一个输入项。从内部资源中直接提前得到缺少的运算项的过程称为前推（forwarding）或者旁路（bypassing）。

- 前推：也称为旁路。一种解决数据冒险的方法，具体做法是从内部寄存器而非程序员可见的寄存器或存储器中提前取出数据。

### 01 例题·两条指令间的旁路

对于上述的两条指令，说明如何使用旁路将流水线各级连接起来。图 4-28 描述了流水线的五级。与图 4-25 中的洗衣店流水线类似，每条指令的数据通路排成一行。

278

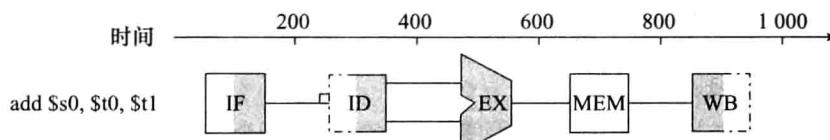


图 4-28 指令流水线的图形表示，其与图 4-25 中的洗衣店流水线类似。本图以及本章均使用图形符号来代表流水线各级使用的物理资源。这些符号在五级流水线中所代表的意义分别是：IF 表示取指阶段，其外方框表示指令的存储器；ID 表示指令的译码或寄存器堆的读取阶段，外边的虚线方框表示要读取的寄存器堆；EX 表示指令的执行阶段，外边的图符表示 ALU；MEM 表示存储器访问阶段，包围它的方框代表数据存储器；WB 表示写回阶段，包围它的虚线方框代表被写回的寄存器堆。阴影表示该资源被指令所使用。因为 add 指令在这一步并不读取数据存储器，所以 MEM 没有阴影。寄存器堆或存储器右半边的阴影表示它们在此步骤中被读取，左半边的阴影表示它们在此步骤中被写入。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。

### 01 答案

图 4-29 表示了把 add 指令执行后的 \$s0 中的值作为 sub 指令执行的输入的旁路连接。□

在图 4-29 中，只有当目标步骤在时间上晚于源步骤时旁路的路径才有效。例如，从前一条指令存储器访问的输出至下一条指令执行的输入就不能实现旁路，因为那样的话将意味着时间的倒流。

旁路可以工作得很好，其具体内容将在 4.7 节详细介绍。然而它并不能够避免所有流水线阻塞的发生。例如，假设第一条指令不是 add 而是装载 \$s0 寄存器的内容，正如图 4-29 所描

279

述的那样，由于数据间的依赖，所需要的数据只有在前一条指令流水线的第四级完成之后才能生效，这对于 sub 指令的第三级输入来说就太迟了。因此，如图 4-30 所示，即使采用了旁路机制，在遇到取数 - 使用型数据冒险（load-use data hazard）时，流水线不得不阻塞一个步骤。图中显示了一个重要的流水线概念，正式的叫法是流水线阻塞（pipeline stall），但是它经常被称为气泡（bubble）。我们经常会在流水线中看到阻塞的发生。4.7 节将给出处理这种复杂情况的方法，即采用硬件上检测阻塞和软件上重新安排代码顺序等方法来避免装载 - 使用型数据冒险。

- ② 取数 - 使用型数据冒险：一类特殊的数据冒险，指当装载指令要取的数还没取回来时其他指令就需要使用的情况。
- ③ 流水线阻塞：也称为气泡。为了解决冒险而实施的一种阻塞。

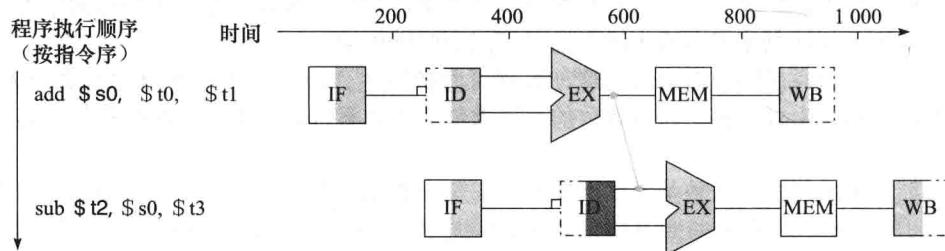


图 4-29 旁路的图形表示。图中的连接表示从 add 指令的 EX 操作输出到 sub 指令的 EX 操作输入的旁路路径，从而替换掉在 sub 的第二步从寄存器 \$s0 读取的值

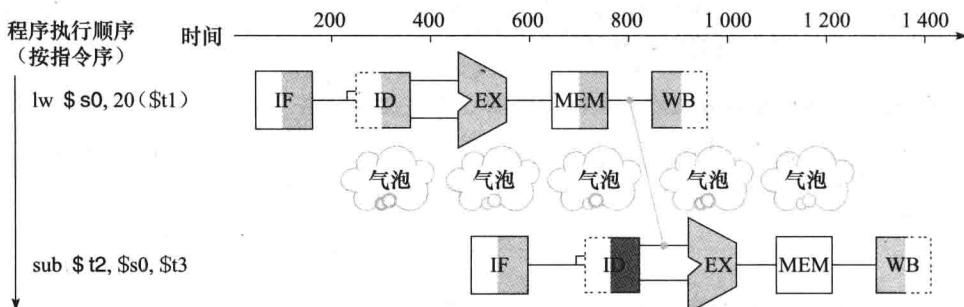


图 4-30 当一条 R 型指令之后紧跟着一条需要使用其结果的装载指令时，即使使用了旁路机制，仍然会产生一次阻塞。如果不进行一次阻塞，从存储器访问的输出到执行级的输入之间的路径在时间上将是倒着的，这显然是不可能的。事实上，这仅是一个示意图，因为直到减法指令取指和译码之后，我们才知道是否需要阻塞。4.7 节详细介绍了这种冒险情况

### 01 例题·重新安排代码以避免流水线阻塞

考虑下面这段 C 代码：

```
a = b + e;
c = b + f;
```

下面是这段 C 代码对应的 MIPS 指令，假设所有的变量都在存储器中，且以 \$t0 为基址进行寻址：

```
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)
```

试找出上述代码段中存在的冒险并试着重新安排指令顺序以避免流水线阻塞。

### 01 答案

两条 add 指令都存在冒险，因为它们都依赖于上一条 lw 指令。注意，通过旁路可以消除一些潜在的冒险，包括第一条 add 指令对第一条 lw 指令的依赖和 sw 指令导致的冒险。而将第 3 条 lw 指令上移到第 3 条指令的位置则可以进一步消除所有冒险：

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

在一个具有旁路功能的流水线处理器中，执行这个重排序后的指令序列要比上面那个指令序列快 2 个时钟周期。□

在前面介绍了设计适应流水线的 MIPS 体系结构指令集的 4 个原则，由旁路可以得到设计 MIPS 体系结构指令集的另一个原则。即每条 MIPS 指令最多只写一个结果并且在流水线的最后一级执行。如果每条指令要写多个结果或写在流水线更早阶段进行则旁路设计要复杂得多。

**01 精解** “前推”这个名称来源于将结果从前面的指令直接发送到后面的指令的思想。“旁路”这个名称来源于把寄存器堆中的结果直接传递到需要的单元中。

### 3. 控制冒险

第三种冒险叫作控制冒险（control hazard）。这种冒险会在下面的情况下出现：决策依赖于一条指令的结果，而其他指令正在执行中。

● 控制冒险：也称为分支冒险（branch hazard）。因为取到的指令并不是所需要的（或者说指令地址的变化并不是流水线所预期的）而导致指令不能在预定的时钟周期内执行。

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以及设置水温以保证能够将衣服清洗干净，但同时要保证清洗剂的用量不能过大，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（stall）：在第一批衣服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。

计算机中的决策就是分支指令。注意，在取分支指令之后，紧跟着就会取下一条指令。但是流水线并不知道下一条真正要执行的指令在哪里，因为它才刚刚从指令存储器中把分支指令给取出来！跟洗衣店的例子一样，一种可能的解决方法是取分支指令后立即阻塞流水线，直到流水线确定分支指令的结果并知道下一条真正要执行的指令在哪为止。

假设可以加入足够多的硬件使得在流水线的第二级能测试寄存器、计算分支地址并更新 PC（详情见 4.8 节）。通过使用这些额外的硬件，包含条件分支的流水线执行情况如图 4-31 所示。如果分支指令的判定条件失败，即分支不执行，则要执行 lw 指令，它在启动之前被阻塞一个 200ps 的额外时钟周期。

### 01 例题·阻塞对分支性能的影响

评价分支阻塞对单位指令时钟周期数（CPI）的影响。假设其他所有指令的 CPI 都为 1。

### 01 答案

第3章的图3-27说明在SPECint2006中，分支指令约占执行指令的17%。由于其他指令的CPI都为1，而分支指令阻塞要多一个时钟周期，因此平均CPI为1.17。与理想的情况相比，现在的速度下降了1.17倍。□

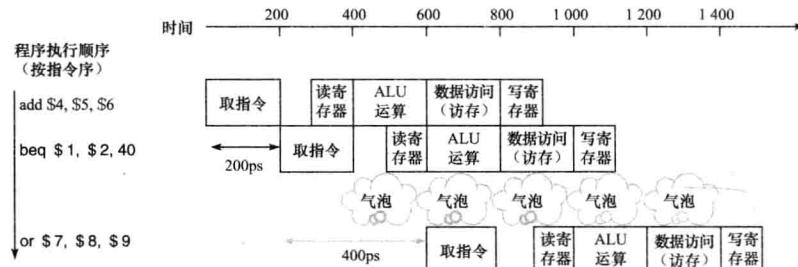


图4-31 在每一个条件分支上阻塞是避免流水线控制冒险的一种解决方法。这个例子假设分支发生，并且分支目标地址处是一条OR指令。分支指令之后会插入一个周期的流水线阻塞，或者叫气泡。事实上，产生阻塞的过程有些复杂，我们会在4.8节说明这一点。这种方法对性能的影响与插入一个气泡是一样的

282

如果不能在第二级解决分支问题（这种情况在较长的流水线中经常发生），那么分支结构上的阻塞将导致更大的速度下降。对很多计算机来说，这种阻塞的方法代价太大，因此也就产生了另外一种消除控制冒险的方法（该方法使用第1章中提到的伟大思想来应对控制冒险）：

**预测 (predict):** 如果你有自信正确地设置洗衣设备来洗涤那些队服（可以预测它的正确工作条件），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，就不得不将已经洗过的队服重新洗一遍。

计算机的确是采用预测的方法来处理分支的。一种简单的预测方法就是总预测分支未发生。当预测正确（分支未发生）的时候，流水线会全速地执行。只有当分支发生时流水线才会阻塞。图4-32给出了这样一个例子。

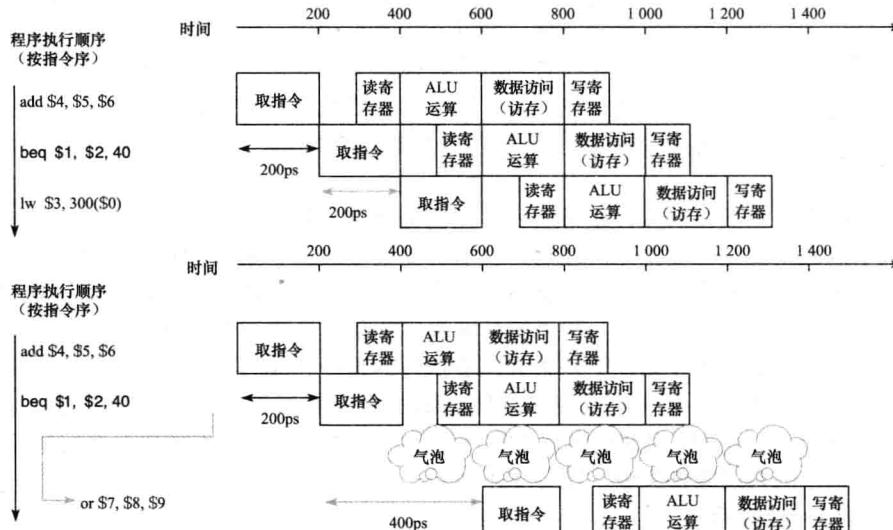


图4-32 预测分支未发生是一种避免流水线控制冒险的解决方法。上图显示的是分支未发生的流水线，下图显示的是分支发生了的流水线。正如我们在图4-31中提到的那样，这种插入气泡的方式是一种简化的表示方法，至少对紧跟分支指令的下一个时钟周期而言是这样。4.8节将给出其中的细节

283

一种更加成熟的分支预测（branch prediction）方法是预测一些分支发生而预测另一些分支不发生。如在上面洗衣店的例子中，夜晚和主场比赛的队服使用一个洗衣设备设置，而白天或客场比赛的队服则使用另一个设置。在计算机程序中，循环体底部的分支总是会跳回到循环体的顶部。在此种情况下，由于分支总是发生并且向前跳转，因此我们可以预测分支会跳转到前面的某一地址处。

- 分支预测：一种解决分支冒险的方法。它预测分支结果并立即沿预测方向执行，而不是等真正的分支结果确定后才开始执行。

这种分支预测的方法依赖于始终不变的行为，它没有考虑特定分支指令的特点。动态硬件预测器与这种方法截然不同，它的预测取决于每一条分支的行为，并且在整个程序生命期内可能改变分支的预测结果。用洗衣店的例子来说，使用动态预测方法，店员将会观察衣服脏的程度并预测一个洗衣设备的设置，然后在本次预测成功的基础上调整下一次的预测行为。

计算机中动态预测方法的一种比较普遍的实现方式是保存每次分支的历史记录，然后利用这个历史记录来预测。稍后我们将看到，历史记录的数量和类型足够多时，这种动态分支预测的方式能够达到 90% 的正确率（见 4.8 节）。当预测错误时，流水线控制必须确保被错误预测的分支后面的指令执行不会生效，并且必须在正确的分支地址处重新开始启动流水线。在洗衣店的例子中，我们必须停止接受新的任务，从而可以重新执行错误预测的任务。

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，并会提高错误预测的代价。控制冒险的解决办法在 4.8 节中将有更加详细的描述。

**01 精解** 还有一种解决控制冒险的方法，即延迟决定（delayed decision）。与洗衣店的例子类比，每当要决定如何洗衣服时，就将一批非足球队的衣服放进洗衣机里，同时等待足球队的制服被烘干。只要有足够多不需要决策的脏衣服，这种方法就很有效。

在计算机中这种方法被称为延迟分支（delayed branch），在 MIPS 体系结构中也得到了实际应用。延迟分支顺序执行下一条指令，在一条指令延迟之后再开始执行分支。由于编译器会自动排列指令，使得分支的行为达到程序员的要求，因此这个过程对 MIPS 的汇编程序员是透明的。MIPS 编译器会在延迟分支指令的后面紧跟着放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地址。在我们的例子中，图 4-31 中分支前的 add 指令不影响分支，所以可以把它移到分支之后以完全隐藏分支延迟。因为只有当分支延迟较短时，延迟分支才有效，所以没有处理器使用超过一个时钟周期的延迟分支。对更长的分支延迟，一般都使用硬件分支预测器。

284

#### 4.5.3 对流水线概述的小结

流水线是一种在顺序指令流中利用指令间并行性的技术。与多处理器编程相比，其优势在于它对程序员是不可见的。

在以下几节中，我们首先使用 4.4 节单周期实现方式的 MIPS 指令子集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流水线在一些典型情况下所能获得的性能提升。

如果想了解更多软件和流水线对性能的意义，并且你已经具有足够的背景知识，可以直接跳到 4.10 节。4.10 节介绍了一些高级流水线概念，如超标量、动态调度等。4.11 节介绍了一些最新的微处理器流水线。

反之，如果你想深入了解流水线的实现方式和如何处理冒险现象，可以接着阅读后面的几节。4.6 节介绍了一个流水线的数据通路和基本控制设计。在 4.6 节的基础上，你可以在 4.7

节中学习旁路和阻塞的实现。紧接着4.8节介绍了处理分支冒险的方法。而4.9节则介绍了异常是如何处理的。

### 01 小测验

对下面每个指令序列，说明哪个必须阻塞，哪个只使用旁路就可以避免阻塞，而哪个既不需要阻塞也不需要旁路就可以执行。

指令序列1	指令序列2	指令序列3
<pre>lw \$t0,0(\$t0) add \$t1,\$t0,\$t0</pre>	<pre>add \$t1,\$t0,\$t0 addi \$t2,\$t0,#5 addi \$t4,\$t1,#5</pre>	<pre>addi \$t1,\$t0,#1 addi \$t2,\$t0,#2 addi \$t3,\$t0,#2 addi \$t3,\$t0,#4 addi \$t5,\$t0,#5</pre>

**01 理解程序性能** 除了存储系统以外，流水线的有效运作是决定处理器 CPI 乃至其性能最重要的因素。正如我们将在4.10节看到的那样，理解现代多发射流水线处理器的性能是一项复杂的任务，相对简单流水线处理器而言需要理解更多的问题。不管怎样，结构冒险、数据冒险和控制冒险在简单流水线处理器和更复杂的流水线处理器中都是非常重要的。

对现代流水线而言，结构冒险经常出现在浮点单元附近，浮点单元是一个几乎不可能完全流水的地方。与之相比，控制冒险一般出现在整数程序中，因为其中分支出现的概率更高，也更难预测。数据冒险在整数和浮点程序中都可能成为性能瓶颈。一般来说，浮点程序中的数据冒险更容易处理，因为低的分支出现频率和规则的存储器存取使得编译器有更大的空间调度指令以避免冒险。与之相比，在整数程序中涉及大量的指针，存储器的存取更不规则，做这样的优化就要困难一些。正如我们将在4.10节看到的那样，有很多编译器和基于硬件的技术通过调度来减少数据间的依赖。

285

**01 重点** 流水线增加了同时执行的指令数目以及指令开始和结束的速率。流水线并不能够减少单一指令的执行时间，也称为延迟（latency）。例如，一个五级流水线仍然需要5个时钟周期来完成一条指令。用第1章的术语来描述就是流水线提高了指令的吞吐率而不是减少了单条指令的执行时间或延迟。

② 延迟：流水线的级数或者顺序执行过程中两条指令间的级数。

对流水线的设计者来说，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、旁路和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。

## 4.6 流水线数据通路及其控制

看起来东西很多，其实不然。

——Tallulah Bankhead, remark to Alexander Woollcott, 1922

图4-33是摘自4.4节的一个单时钟周期的数据通路。将指令划分为5个阶段意味着一个流水线采用5级，也就意味着在任何一个单时钟周期内，最多会执行5条指令。因此必须把数据通路分为5个部分，每一部分用与之对应的指令执行阶段来命名。

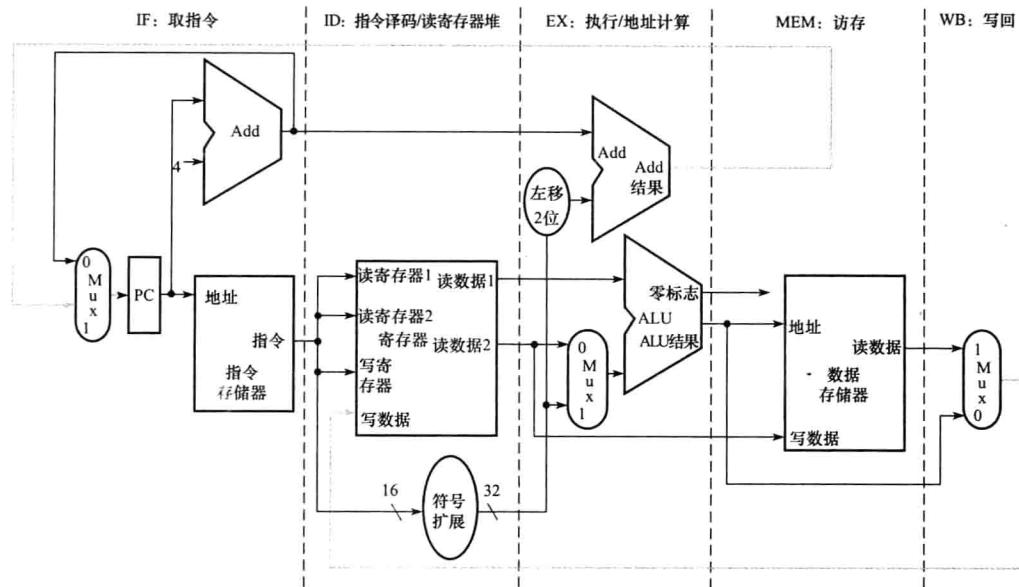


图 4-33 4.4 节中的单时钟周期数据通路（与图 4-17 类似）。图中自左至右把指令的每一步映射到数据通路中。PC 更新与写回过程是唯一的例外（图中用灰色线表示），其发送 ALU 结果或存储器数据到左边的寄存器堆中。（我们通常使用灰色线表示控制，但在这里表示数据线。）

- 1) IF: 取指令。
- 2) ID: 指令译码, 读寄存器堆。
- 3) EX: 执行或计算地址。
- 4) MEM: 数据存储器访问。
- 5) WB: 写回。

图 4-33 的 5 个部分大致与数据通路相符：指令与数据随着执行过程从左到右依次通过五级流水线。正如洗衣店的例子一样，衣服沿着一条工作线依次完成清洗、烘干和整理，而不会反向移动。

286

然而，在从左到右的指令流中有两个例外：

- 写回阶段，它把结果写回数据通路中间的寄存器堆中。
- 选择 PC 的下一个值时，需在自增的 PC 和 MEM 级的分支地址间进行选择。

这两个从右向左的数据流不会影响当前指令；只有当前指令以后的指令才会受到这种数据反向活动的影响。需要注意的是，第一个例外会导致数据冒险，而第二个会导致控制冒险。

一种表示流水线数据通路的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出它们之间的关系。图 4-34 在同一时间轴上表示了图 4-27 中指令执行过程中各自的数据通路（我们仍然使用图 4-33 中的格式来表示图 4-34 中的关系）。

从表面来看，图 4-34 中的三条指令似乎需要三条数据通路。事实上，通过增加保存中间数据的寄存器，使得在指令执行过程中可以共享部分数据通路。

例如，如图 4-34 所示，指令存储器只在每条指令的 5 个步骤中的一步中用到，因此我们允许它在其他 4 步中被其他的指令共享。为了在其他 4 步中保持指令的值，从指令存储器中读出的数据必须保存在寄存器中。将同样的方法应用到每个流水线级中，我们需要在图 4-33 中各级间有分割线的地方都加入寄存器。再回到洗衣店的例子中，这里可以用篮子在两个步骤间存放下一步的衣服。

图 4-35 描述了流水线的数据通路，其中流水线寄存器用灰色表示。在每个时钟周期中所

有指令都会从一个流水线寄存器传递到另一个流水线寄存器中。寄存器以被该寄存器分开的两个阶段来命名，如 IF 和 ID 之间的流水线寄存器叫作 IF/ID。

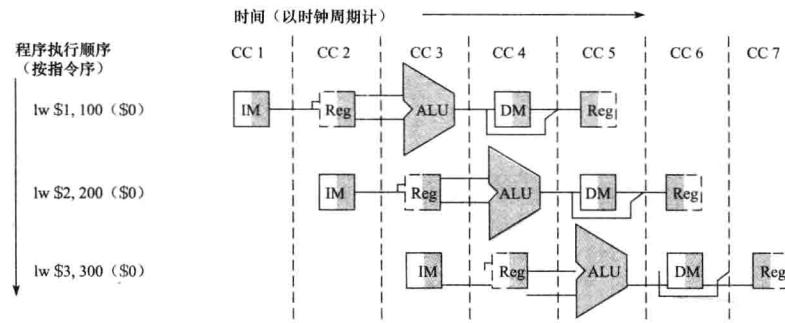


图 4-34 按图 4-33 中的单时钟周期数据通路执行的指令（假定以流水线方式执行）。与图 4-28 到图 4-30 类似，本图假设每一条指令有它独立的数据通路，并根据使用情况将相应的部分涂上阴影。与这些图不同的是，流水线的每一级都用该级使用的物理资源标示，分别对应图 4-33 中数据通路的相应部分。IM 表示指令存储器与取指令阶段的 PC，Reg 表示指令译码/寄存器堆读取阶段 (ID) 的寄存器堆和符号扩展单元，等等。为了保持正确的时序，这种形式的数据通路把寄存器堆从逻辑上划分为两个部分：寄存器读取 (ID) 阶段的寄存器读和写回 (WB) 阶段的寄存器写。这种复用在图中表示为：在 ID 级当寄存器堆没有被写入时，将没有阴影的寄存器堆的左半部分用虚线表示；而在 WB 级，当寄存器堆没有被读取时，将没有阴影的右边部分用虚线表示。与以前一样，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

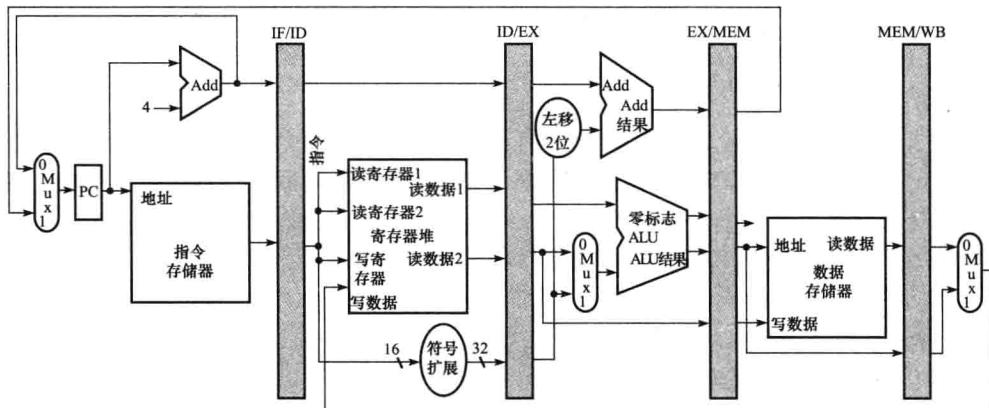


图 4-35 图 4-33 数据通路的流水线版本。流水线寄存器（以灰色标识）将流水线的各部分分开。例如，IF/ID 将取指令和指令译码阶段分开。为了存储所有穿过它的数据（用线条表示），寄存器的宽度必须足够大。例如，因为 IF/ID 寄存器必须同时保存从存储器中提取出来的 32 位指令及 32 位 PC 自增地址，所以它的宽度必须是 64 位。我们将在本章中逐渐增加寄存器宽度，目前另外三个流水线寄存器的宽度分别是 128 位、97 位和 64 位

需要注意的是，在写回阶段的后面没有流水线寄存器。所有指令都会更新机器中的某些状态，如寄存器堆、存储器或 PC 等，因此各个流水线寄存器对于更新后的状态来说是多余的。例如，装载指令会把它的结果放入 32 个寄存器中的某一个，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。

当然，每条指令都会更新 PC，不管是自增还是设置为分支目的地址。PC 可以看成一个流水线寄存器：给流水线的 IF 级提供数据。不像图 4-35 中那些灰色的流水线寄存器，PC 是可见

体系结构寄存器的一部分，发生异常时必须保存它的内容，而那些流水线寄存器的内容可被丢弃。用洗衣店的例子来说，你可以把 PC 看成洗涤步骤之前装脏衣服的篮子。

为了描述流水线的工作方式，本章将使用一系列图片来表示这些顺序的操作。这些内容需要一定时间去理解，但不要害怕，这些图片实际上比它们看上去要容易理解，因为可以对比观察每一个时钟周期内所发生的变化。4.7 节将介绍流水线指令间发生数据冒险的情况，这里暂时忽略。

图 4-36 ~ 图 4-38 表示了装载指令在通过流水线的五级时数据通路的活动部分。先讨论装载指令是因为它完全使用了流水线的五级。正如图 4-28 ~ 图 4-30 所显示的那样，当寄存器或存储器被读取时，图 4-35 中活动的数据通路部件用灰色表示。在图中用阴影表示其右半部分；而当它们被写入时，用阴影来表示其左半部分。

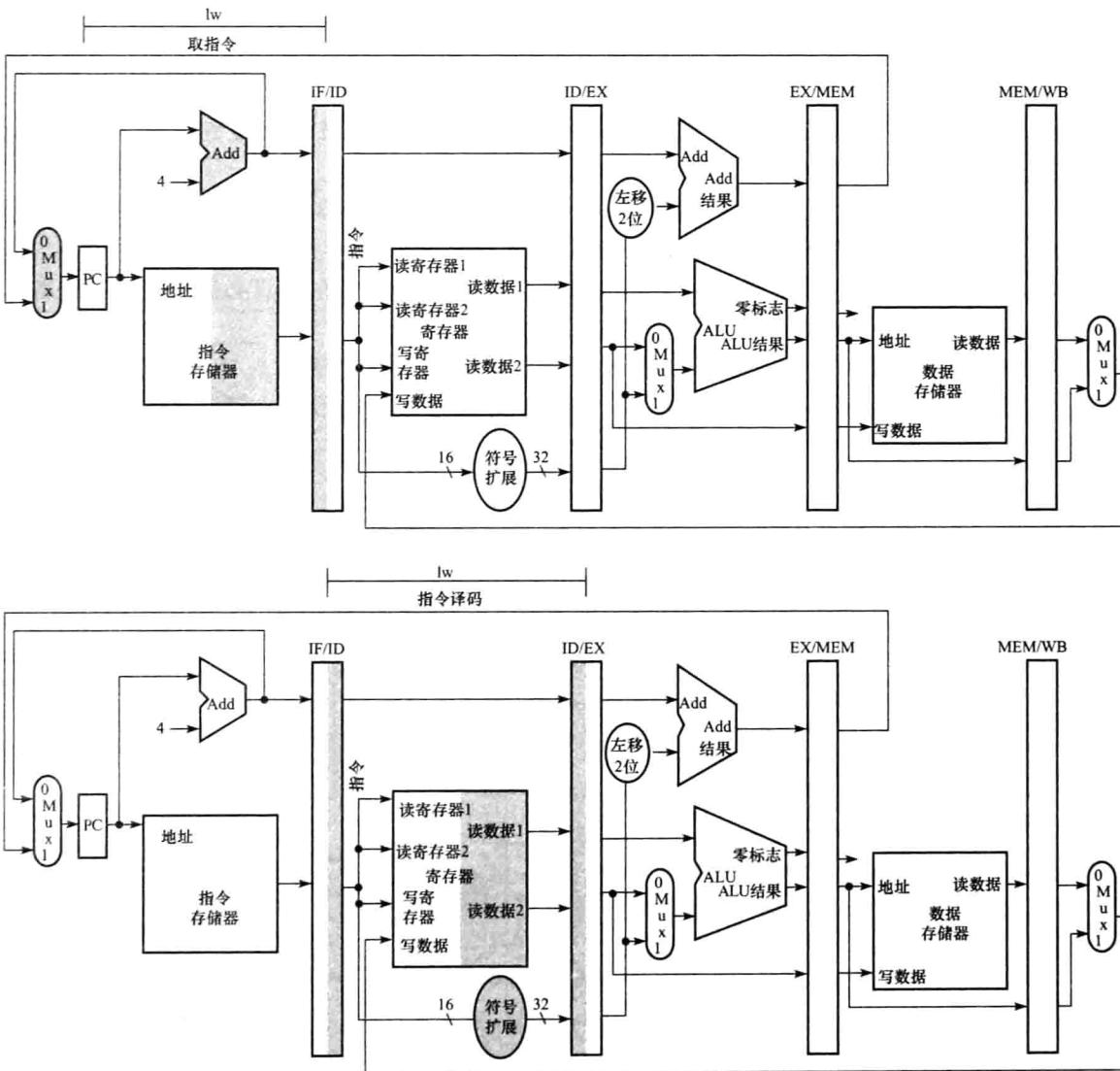


图 4-36 IF 和 ID：一条指令在流水线中的第一、二步。这种灰色的表示方法与图 4-28 相同。正如 4.2 节中介绍的那样，读写寄存器并不会发生冲突，因为寄存器内容的变化只在时钟的边缘发生。虽然 `lw` 指令只需要第二级中寄存器 1 的值，但由于处理器并不知道当前是哪一条指令正在被译码，因此它把符号扩展后的 16 位常量及两个寄存器的值都读入 ID/EX 流水线寄存器中。我们并不一定需要所有这三个操作数，但是保留全部三个操作数能简化控制。

我们把每一幅图中活动的流水线级用指令的缩写 `lw` 和流水级名称标出。具体情况如下：

1) 取指令：图 4-36 的顶端表示指令使用 PC 中的地址从存储器中读取数据，然后将数据放入 IF/ID 流水线寄存器中。PC 地址加 4 然后写回 PC 以便为下个时钟周期做好准备。增加后的地址同时也存入了 IF/ID 流水线寄存器中以备后面的指令使用（如 `beq`）。计算机并不知道所取指令的类型，所以必须考虑到所有可能的指令，并沿流水线传递所有可能有用的信息。

2) 指令译码与寄存器堆的读取：图 4-36 的底部显示的是 IF/ID 流水线寄存器的指令部分，其中包括一个 16 位的立即数（可扩展为带符号的 32 位数）和两个寄存器号（用于读取寄存器）。这三个值和自增的 PC 地址一起存入 ID/EX 流水线寄存器中。这里同样必须传递后面指令可能需要的所有信息。

3) 执行或者地址计算：图 4-37 表示装载指令从 ID/EX 流水线寄存器中读取由寄存器 1 传过来的值以及经符号扩展后的立即数，并用 ALU 将它们相加，和值存入 EX/MEM 流水线寄存器中。

4) 存储器访问：图 4-38 的顶端表示装载指令使用从 EX/MEM 流水线寄存器中得到的地址读取数据存储器，并将数据存入 MEM/WB 流水线寄存器中。

5) 写回：图 4-38 的底部表示了最后一个步骤，即从 MEM/WB 流水线寄存器中读取数据并将它写回图中部的寄存器堆。

290  
291

对装载指令整个过程的描述表明任何后面的流水线级可能用到的信息必须通过流水线寄存器传递。存储指令也是如此。下面是存储指令的 5 个执行步骤：

1) 取指令：利用 PC 中的地址从存储器中读出指令，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译码之前，所以图 4-36 中顶端部分既适用于装载指令也适用于存储指令。

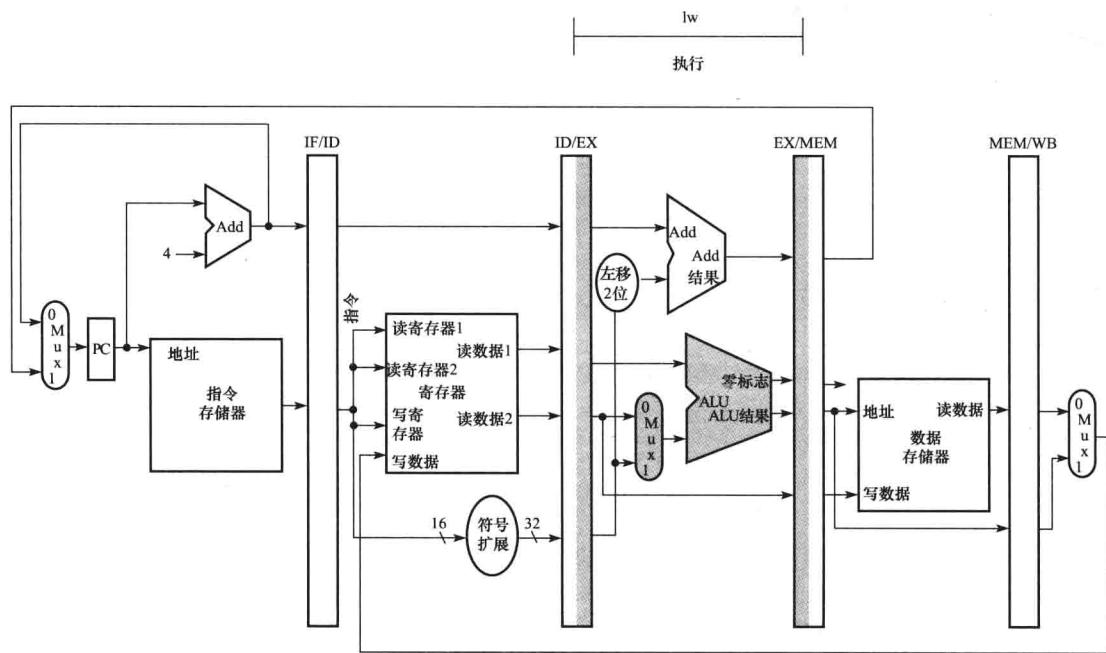


图 4-37 EX: `lw` 指令在流水线中的第三步，图 4-35 中活动的数据通路部件用灰色表示。将寄存器的值与经过符号扩展的立即数相加，其和放入 EX/MEM 流水线寄存器中

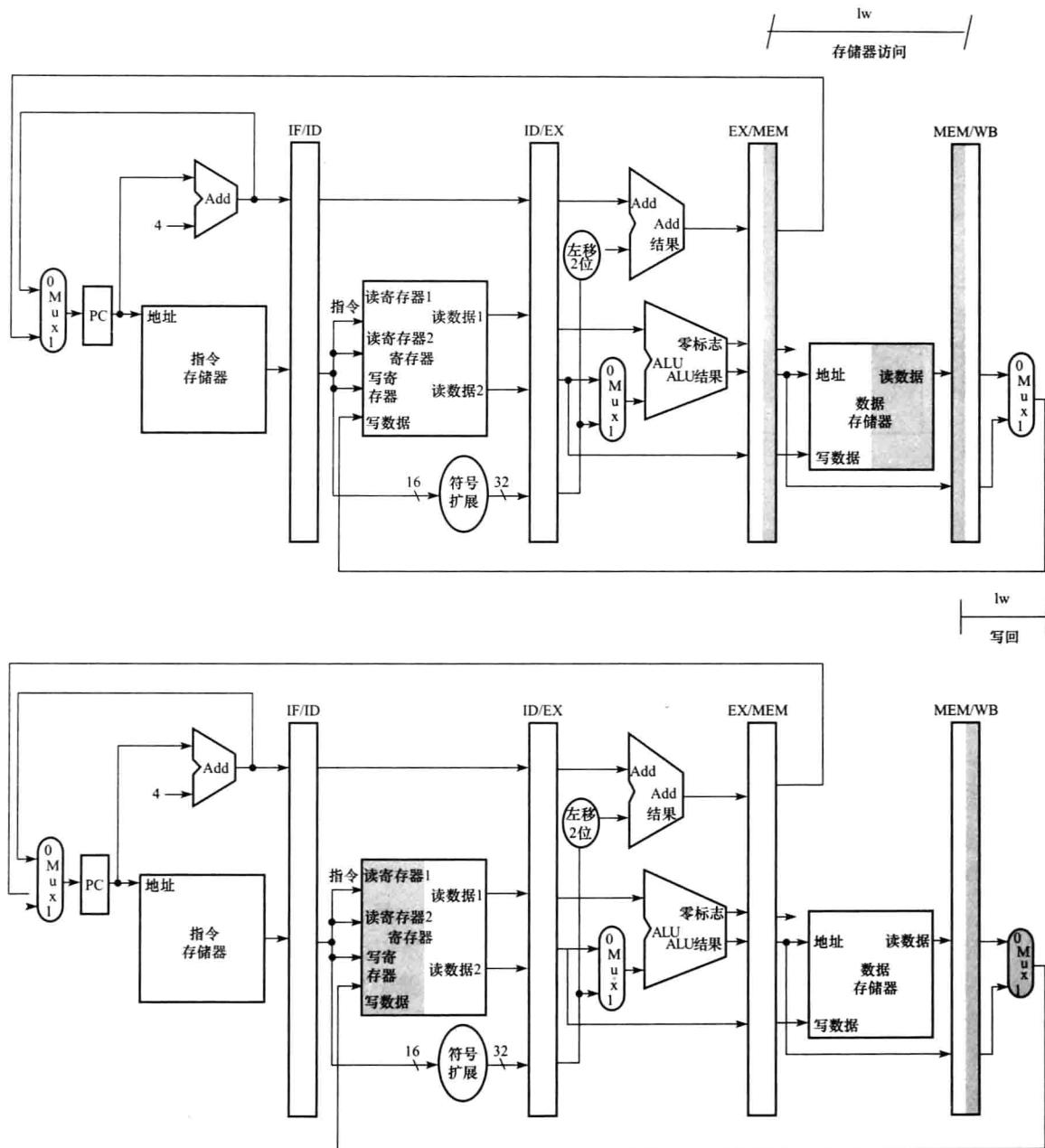


图 4-38 MEM 和 WB: lw 指令在流水线中的第四步和第五步, 图 4-35 中活动的数据通路部件用灰色表示。利用 EX/MEM 流水线寄存器中包含的地址读取数据存储器, 并将读取的数据放入 MEM/WB 流水线寄存器中, 然后从 MEM/WB 流水线寄存器中读取数据写回数据通路中部的寄存器堆。请注意: 这里有一个错误, 将在后面的图 4-41 中修复

2) 指令译码与寄存器堆的读取: IF/ID 流水线寄存器中的指令包括用于读取寄存器的两个寄存器号和用于符号扩展的 16 位立即数。读出的两个寄存器值和符号扩展后的 32 位立即数都存放在 ID/EX 流水线寄存器中。图 4-36 中的底部同时也可描述装载指令的第二个流水级。由于此时并不知道要执行的指令类型, 因此所有指令都执行这两个步骤。

3) 指令执行或地址计算: 图 4-39 描述了 sw 指令在流水线中的第三步, 有效地址存放在 EX/MEM 流水线寄存器中。

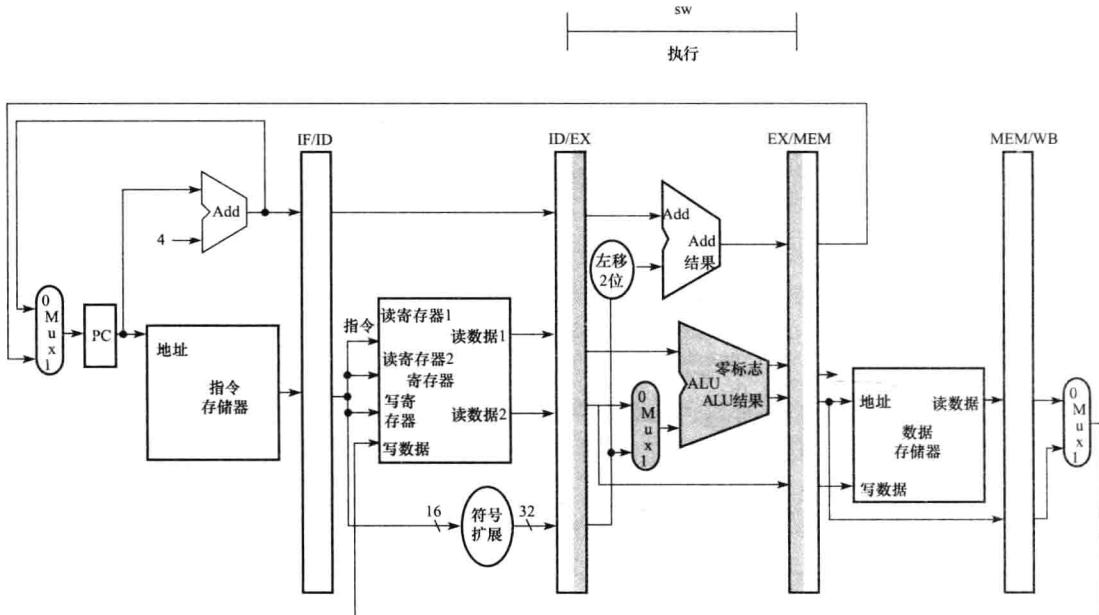


图 4-39 EX: sw 指令在流水线中的第三步。与图 4-37 中装载指令的第三个流水级不同的是，第二个寄存器中的数据被装入 EX/MEM 流水线寄存器中，并被用于下个流水级。虽然总是将第二个寄存器中的数据装入 EX/MEM 流水线寄存器中并不会产生什么不良影响，但为了使流水线更易于理解，我们只在存储指令中才写第二个寄存器的内容

4) 存储器访问：图 4-40 的顶端描述的是数据写入存储器的过程。值得注意的是，需要写入存储器的数据在较早的流水级中已经读出并存放在 ID/EX 中。在 MEM 级唯一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。

5) 写回：图 4-40 的底部描述了存储指令的最后一步。存储指令在写回步骤中不做任何事情。由于存储指令后的每一条指令都已经进入流水线中，所以无法加速这些指令。因此，任何一条指令都必须经过流水线的每一个步骤，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。

存储指令再次说明在流水线中为了从前面的流水级向后面的流水级传递信息，必须将信息放入流水线寄存器中，否则当下一条指令进入该流水级时这些信息将会丢失。在存储指令中，需要将一个寄存器中的内容在 ID 级读出然后在 MEM 级写入存储器。这些数据首先放在 ID/EX 流水线寄存器中，然后传送到 EX/MEM 流水线寄存器中。

取数指令与存储指令的执行过程还表明了另一个重要特性，即数据通路中的每一个逻辑单元（如指令存储器、寄存器读取端口、ALU、数据存储器以及寄存器写入端口）都只能在一个流水级中使用，否则就会产生结构冒险（见 4.5 节）。所以这些单元及其控制可以和一个流水级相联系。

现在我们可以修复图 4-38 中装载指令设计的错误了。你发现这个错误了吗？在装载指令执行的最后一级写回了哪个寄存器呢？更确切地说，哪条指令提供了写寄存器号呢？在 IF/ID 流水线寄存器中的指令提供了写寄存器号，但是很显然现在这条指令已经是装载指令之后的指令了！

因此，我们要在装载指令中保存目的寄存器号。就像存储指令为了 MEM 的需要将寄存器的内容从 ID/EX 传送到 EX/MEM 中一样，为了 WB 级使用的需要，装载指令必须把寄存器号从 ID/EX 经过 EX/MEM 传送到 MEM/WB 中。从另一个角度来考虑寄存器号的传递，为了共享流水线的数据通路，我们需要在 IF 中保存读取的指令，因此每一个流水线寄存器都要保存当

前和后续流水级所需的部分指令。

图 4-41 给出了修正后的数据通路。首先将写寄存器号传送到 ID/EX 寄存器，然后送到 EX/MEM 寄存器，最后送到 MEM/WB 寄存器。在 WB 级使用寄存器号指定了要写入的寄存器。图 4-42 是一个简单的数据通路图，它标出了从图 4-36 到图 4-38 装载指令在所有 5 个流水级中要使用的硬件。阅读 4.8 节可以了解如何使分支指令按期望的方式工作。

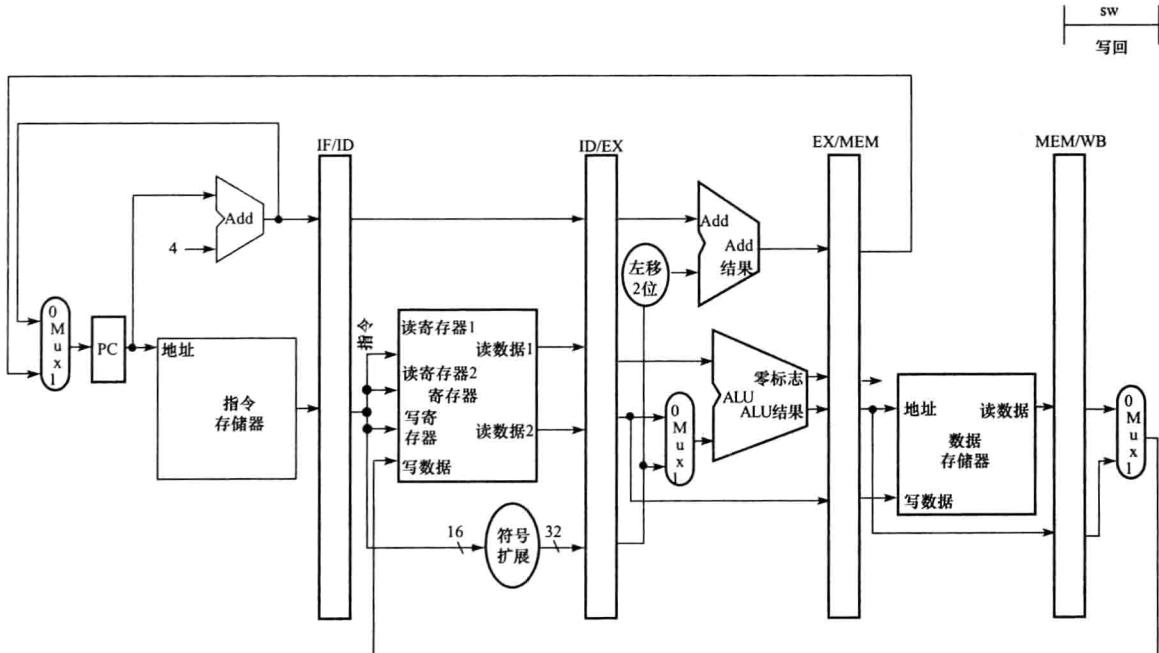
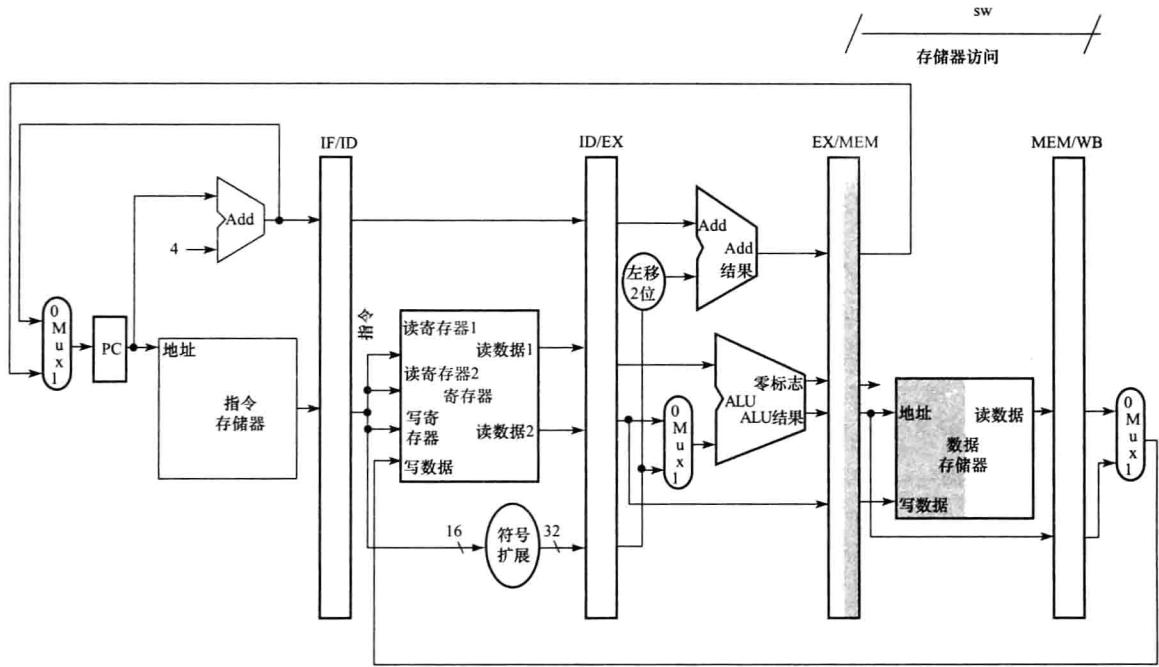


图 4-40 MEM 和 WB: sw 指令在流水线中的第四步和第五步。第四步将数据写入数据存储器中，写入数据来自于 EX/MEM 流水线寄存器。MEM/WB 流水线寄存器没有改变。一旦数据写入存储器，存储指令就没有什么可做的了，所以在第五步中存储指令并不做任何处理

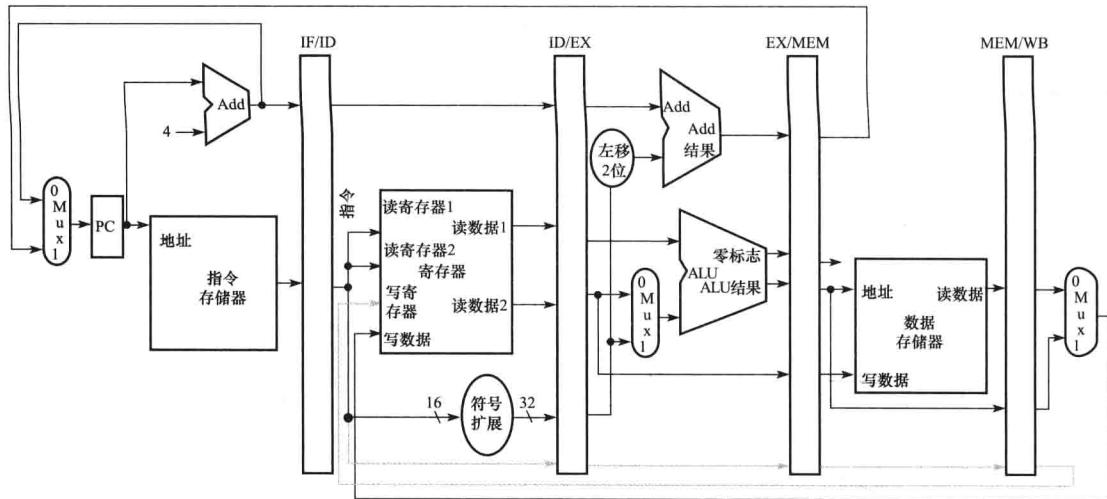


图 4-41 可正确执行装载指令的流水线数据通路。写寄存器号与数据一起从 MEM/WB 流水线寄存器中得到。通过在最后的三个流水线寄存器上分别增加 5 位，寄存器号就能从 ID 流水级一直传送到 MEM/WB 流水线寄存器。新的路径以灰色线标识

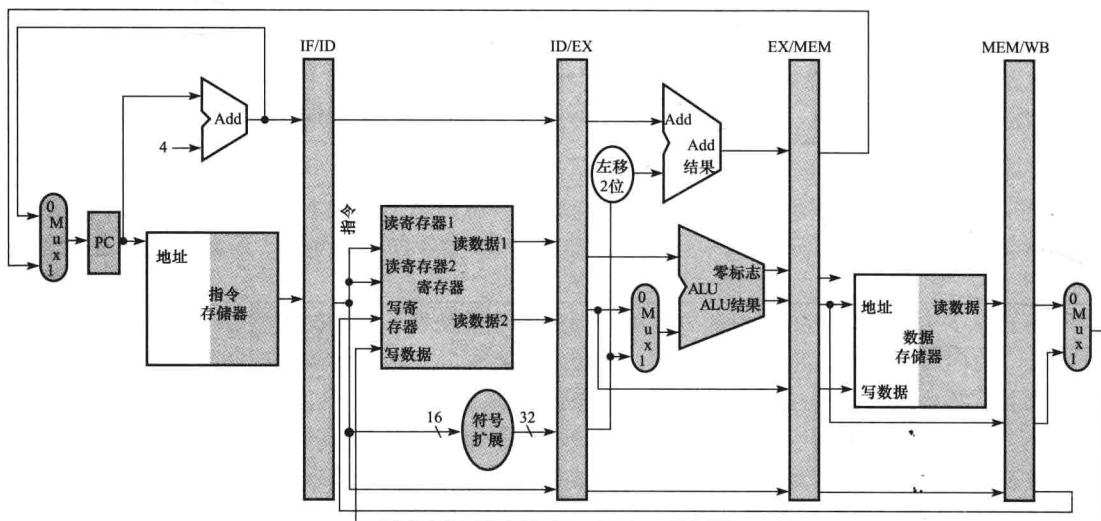


图 4-42 图 4-41 中指令的五级流水线中用到的全部数据通路

#### 4.6.1 图形化表示的流水线

295  
296

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。为了帮助理解流水线，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（见图 4-34）和单时钟周期的流水线图（见图 4-36~图 4-40）。多时钟周期图虽然简单但不包括所有的细节。下面以这 5 条指令构成的指令序列为为例进行说明：

```

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

```

图 4-43 表示的是该指令序列的多时钟周期流水线图。与图 4-25 中洗衣店流水线的表示方法类似，时间从左到右前进，指令从上到下前进。沿着指令轴分别表示各流水级以及所占据的时钟周期。这些程式化的数据通路用图形的方式展示了流水线的 5 个级别，但用方框来命名每个流水线等级也是很好的表示方法。图 4-44 给出了一个更加传统的多时钟周期流水线图的表示方法。需要注意的是，图 4-43 中描述的是每个步骤中使用的物理资源，而图 4-44 描述的是每个步骤的名称。

单时钟周期流水线图表示的是在一个时钟周期内整个数据通路的状态，通常所有 5 个流水级中的指令都在各流水级上做相应的标志。这种流水线图描述了在每一个时钟周期内流水线中所发生事件的细节。通常，可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的流水线操作，而使用多时钟周期流水线图对流水线总体进行全局描述。（如果你对图 4-43 的细节感兴趣，可参考 4.13 节中对单时钟周期图的描述）。从多时钟周期图中抽出一个时钟周期就表示了单时钟周期图流水线的状态，其中显示了流水线中每条指令对数据通路的使用。例如，图 4-45 的单时钟周期图对应的就是图 4-43 和图 4-44 的第 5 个时钟周期。很明显，单时钟周期图可以表现更多的细节，但表示同样多时钟周期时所占空间要比多时钟周期图大得多。本章后面的练习会要求你根据其他的指令序列画出对应的流水线图。

297

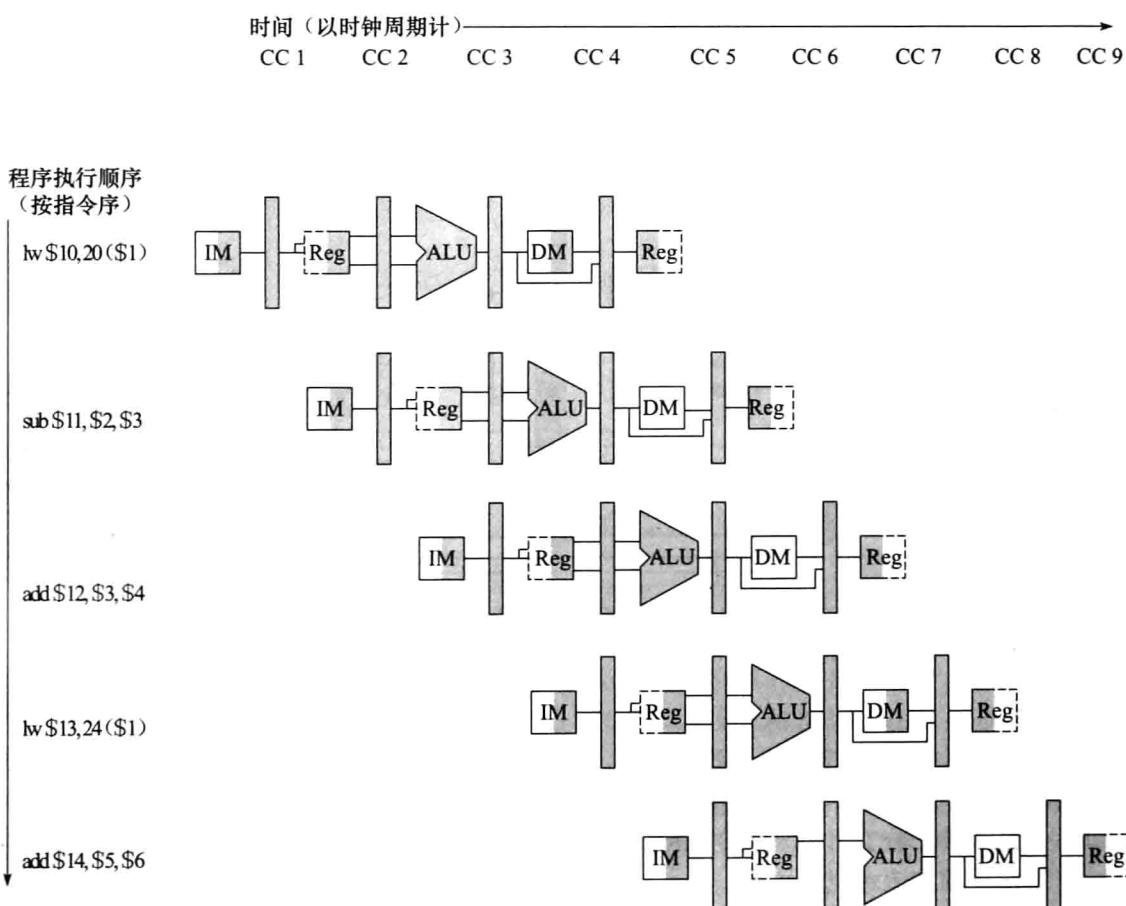


图 4-43 5 条指令的多时钟周期流水线图。此种流水线图在一幅图中表示了指令序列的完整执行过程。指令从上到下按照执行的顺序被排列，时钟周期从左向右前进。与图 4-28 流水线表示方法不同的是，本图给出了每一级的流水线寄存器。图 4-44 给出了这种图更为传统的表示方法

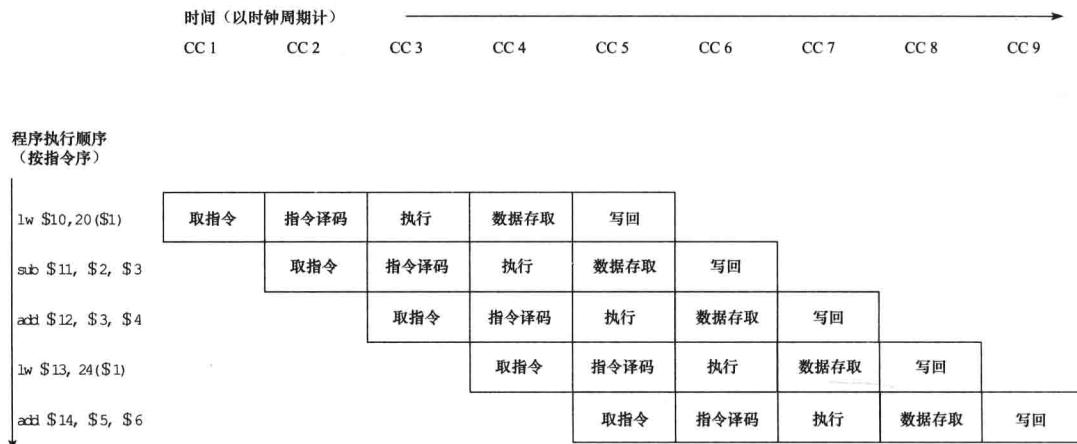


图 4-44 相对图 4-43 更为传统的多时钟周期流水线图

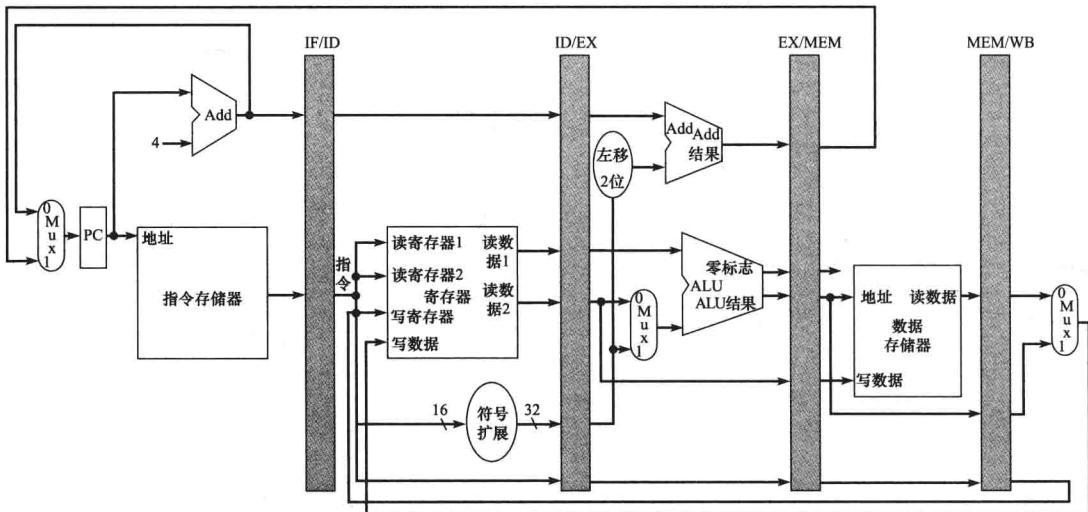
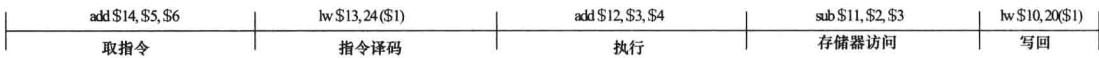


图 4-45 对应图 4-43 和图 4-44 的流水线第 5 个时钟周期的单时钟周期流水线。从图中可以看出，单时钟周期图就是从多时钟周期图中抽出的一列

### 01 小测验

几个学生在讨论五级流水线的效率问题。有一个学生指出并非所有流水级中的指令都是活动的。在忽略冒险的情况下，他们作出了以下几个断言，其中哪一个是正确的？

- 1) 允许跳转、分支、ALU 指令使用比 5 级（装载指令需要的级数）更少的级数将在所有情况下增加流水线的性能。
- 2) 允许一些指令使用更少的级数并不能提高性能，因为吞吐率是由时钟周期决定的。每条指令所需的流水线级数仅影响它的延迟时间，而不影响吞吐率。
- 3) 不可能减少 ALU 指令所需的时钟周期数，因为它们需要写回结果。不过分支和跳转指令是可以减少时钟周期数的，因此存在改善性能的机会。
- 4) 相对于尝试减少指令所需的时钟周期数，我们可以延长流水线的级数，虽然每条指令

花费更多的时钟周期数，但时钟周期的长度变短了，这样才能提高性能。

#### 4.6.2 流水线控制

相对以前的任何计算机，6600型计算机的控制系统是大不相同的。

——James Thornton,《Design of a Computer: The Control Data 6600》, 1970

4.3节介绍了在单周期数据通路加入控制的方法，下面我们将介绍在采用流水线的数据通路中如何加入控制。首先我们在带有诸多限制条件下通过一个简单设计方案了解流水线控制。

我们首先要做的工作就是标识已有数据通路上的控制信号，如图4-46所示。我们尽量借用图4-17中简单数据通路的控制方法，特别是使用相同的ALU控制逻辑、分支逻辑、目的寄存器号多选器和控制信号。尽管图4-12、图4-16以及图4-18中已给出了这些功能单元的定义，为了使下面的内容更易于理解，图4-47~图4-49重新对其进行了解释。

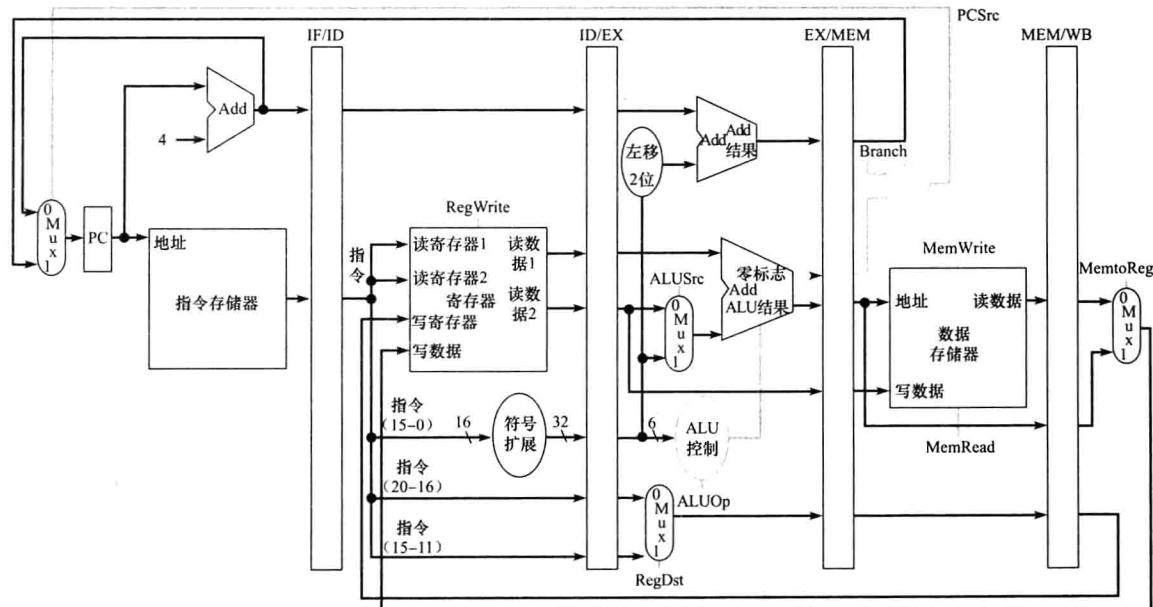


图4-46 在图4-41上增加了控制信号的流水线数据通路。这个数据通路采用了与4.4节中相同的PC源控制逻辑、寄存器目标号和ALU控制。需要注意的是，这时在EX流水级中指令需要一个6位的功能字段（功能码）作为ALU控制的输入，所以该6位字段必须存放在ID/EX流水线寄存器中。而该6位字段是指令中立即数的低6位，由于在对立即数进行符号扩展时低6位没有发生变化，所以ID/EX流水线寄存器可以从立即数中获得这6位数。

与单时钟周期实现方法一样，我们假定在每个时钟周期内都会写PC，因此就不需要单独的PC写信号。同理，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也会写入一次。

为了详细说明流水线的控制问题，我们只需要在每一个流水级中都设置相应的控制信号。由于每一个控制信号只与某个流水级中的某个功能单元相关，因此我们可以根据流水线的5级将控制信号分成5组：

- 1) 取指令：读指令存储器和写PC的控制信号总是有效的，因此在取指阶段没有特别需要控制的内容。
- 2) 指令译码/寄存器堆读：与第一步类似，在每个时钟周期内本阶段所做的工作都是完全相同的，因此不需要设置控制信号。
- 3) 指令执行/地址计算：控制信号有RegDst、ALUOp和ALUSrc（见图4-47和图4-48）。

根据这些信号选择结果寄存器、ALU 操作，并为 ALU 读取数据 2 或符号扩展后的立即数。

4) 存储器访问：这一步的控制信号有 Branch、MemRead 和 MemWrite。这些控制信号分别由相等则分支、装载指令和存储指令设置。除非控制电路确定是一条分支指令并且 ALU 结果为 0，否则将选择线性地址中的下一条指令作为图 4-48 中的 PCSrc 信号。

5) 写回：控制信号有 MemtoReg 和 RegWrite，其中前者决定是将 ALU 结果还是将存储器数据传送到寄存器堆，后者决定是否写入寄存器堆。

由于采用流水线方式的数据通路并不改变控制信号的意义，因此可以使用与简单数据通路相同的控制信号。图 4-49 就与 4.4 节具有相同的控制信号，只是这 9 个控制信号按流水级进行了分组。  
[302]

指令操作码	ALUOp	指令操作	功能码	ALU 操作	ALU 控制信号
LW	00	取字	XXXXXX	加	0010
SW	00	存字	XXXXXX	加	0010
相等则分支	01	相等则分支	XXXXXX	减	0110
R 型	10	加	100000	加	0010
R 型	10	减	100010	减	0110
R 型	10	与	100100	与	0000
R 型	10	或	100101	或	0001
R 型	10	小于则置位	101010	小于则置位	0111

图 4-47 图 4-12 的副本。本图描述了如何根据 ALUOp 控制位和不同 R 型指令的功能码设置 ALU 控制信号的值

信号名	置无效时的效果 (0)	置有效时的效果 (1)
RegDst	写入寄存器的目标号来自 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	写入寄存器的源寄存器设置为输入的写入数据
ALUSrc	第二个 ALU 操作数来自第二个寄存器堆的输出 (读数据 2)	第二个 ALU 操作数是指令低 16 位的符号扩展
PCSrc	PC 被 PC + 4 替代	PC 被分支目标地址替代
MemRead	无	输入地址对应的数据存储器的内容为读数据的输出
MemWrite	无	输入地址对应的数据存储器的内容替换为写数据的输入
MemtoReg	ALU 提供寄存器写数据的输入	数据存储器提供寄存器写数据的输入

图 4-48 图 4-16 的副本。图中定义了 7 个控制信号的功能。ALUOp 已经在图 4-47 的第二列中定义。当一个二路多选器的控制位有效时，多选器选择 1 对应输入；否则，如果控制位无效，多选器选择 0 对应输入。注意 PCSrc 是由图 4-46 的一个与门控制的。如果分支信号与 ALU 的零信号都有效，则 PCSrc 为 1，否则为 0。控制单元仅在 beq 指令中才设置分支信号有效，其他时候 PCSrc 都会为 0

指令	执行/地址计算阶段的控制信号				存储器存取阶段的控制信号			写回阶段的控制信号	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
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 4-49 按流水线最后三级分为三组的控制信号，其值与图 4-18 相同

实现控制就是为每一条指令的每一个步骤中的 9 个控制信号设置合适的值，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号。

由于控制从 EX 级开始，因此可以在指令译码阶段创建控制信号。图 4-50 描述了当指令在

流水线中传递时控制信号的使用方法，这一点与图 4-41 中执行装载指令时目的寄存器号在流水线中的传递过程类似。图 4-51 描述了带有扩展流水线寄存器且将控制信号连接到相应流水级的完整数据通路。（如果你想知道更多的细节，4.13 节给出了更多 MIPS 代码在流水线硬件中执行的单时钟周期流水线图。）

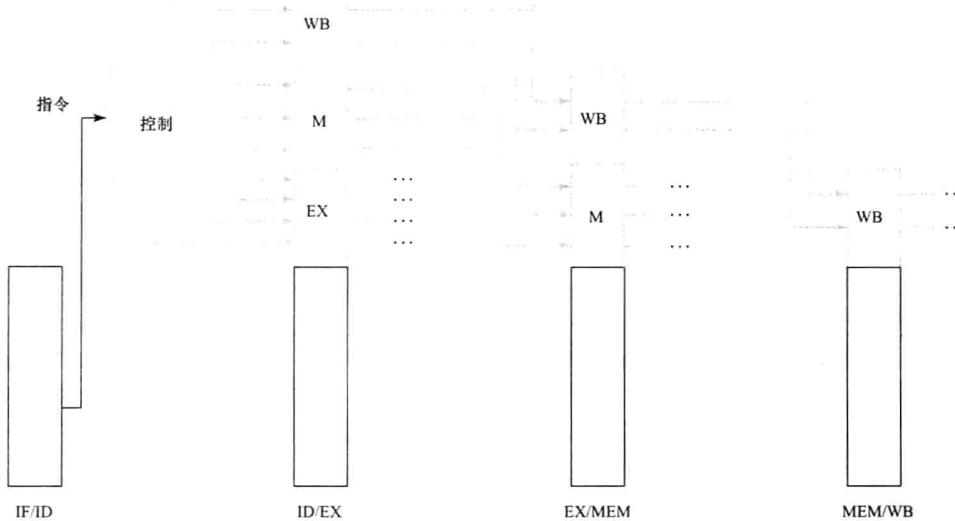


图 4-50 流水线最后三级的控制信号。需要注意的是，9 个控制信号中有 4 个用于 EX 级，而剩下的 5 个控制信号被传递到扩展的保存控制信号的 EX/MEM 流水线寄存器中；传递来的 5 个控制信号中有 3 个用于 MEM 级，剩下的 2 个传递到 MEM/WB 并用于 WB 级

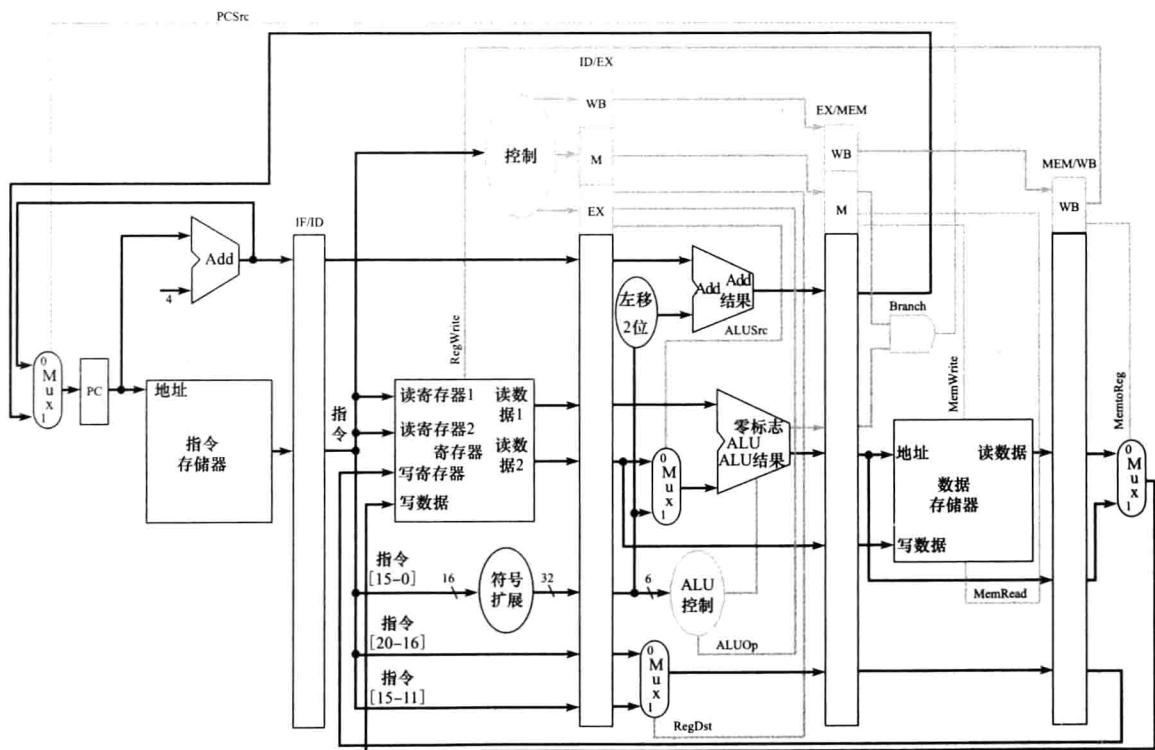


图 4-51 图 4-46 中的流水线数据通路，已将控制信号连接到流水线寄存器的控制部分。流水线最后三级的控制信号是在指令译码阶段创建的，随后放入 ID/EX 流水线寄存器。每个流水级使用相应的控制信号，并将剩余的控制信号传递到下个流水级

## 4.7 数据冒险：旁路与阻塞

这是什么意思，为什么要构建它？这是旁路，你必须构建旁路。

——Douglas Adams, 《The Hitchhiker's Guide to the Galaxy》, 1979

上节的例子介绍了流水线的强大功能以及硬件如何以流水线的方式执行任务。本节我们避开这些光环，看看流水线在实际程序中的情况。图 4-43 ~ 图 4-45 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 4.5 节中我们就已经发现数据冒险是影响流水线执行的主要障碍之一。

让我们分析下面这个带有许多相关性的指令序列（依赖关系以粗体标出）：

```
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 4 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，程序员认为后 4 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 4-52 用多时钟周期流水线图进行了表示。为了在当前流水线中表示这个指令序列的执行过程，图 4-52 的顶部给出了寄存器 \$2 中的值，可以看出寄存器 \$2 的值在第 5 个时钟周期的中间发生改变，也就是 sub 指令写结果的时候。

最后一个潜在的冒险可以通过设计相应的寄存器堆硬件解决。当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？这里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是一致的，而且在这种假设条件下不会发生数据冒险。

图 4-52 表明如果在第 5 个时钟周期之前读寄存器 \$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 可得到正确结果 -20，而指令 AND 和 OR 将得到错误结果 10。使用这种风格的流水线图，当一条依赖关系的方向与时间轴相反时，该问题就变得很明显。

正如 4.5 节所提到的那样，sub 指令在 EX 级（第 3 个时钟周期）的末尾就可以得到需要的结果。那么 AND 指令和 OR 指令什么时候真正需要该数据呢？应该是在 AND 指令和 OR 指令的 EX 级开始前，分别是第 4 个和第 5 个时钟周期。所以只要我们在刚得到数据时就将其旁路给所需的单元而不是等待其可以从寄存器堆中读出来，就可以无阻塞地执行这两条指令了。

旁路到底是怎样工作的呢？在本节下面的部分，为了简化讨论，我们仅考虑如何直接传送 EX 级产生的数据，该数据可能是 ALU 运算的结果，也可能是地址计算的结果。这意味着如果一条指令试图在 EX 级使用前面一条指令在 WB 级才写入寄存器堆的数据时，我们需要提前将数据送到 ALU 的输入端。

一种更精确的表示相关性的方法是使用流水线寄存器字段。例如，ID/EX.RegisterRs 表示一个需要流水线寄存器 ID/EX 获得的源寄存器号，其值可在流水线寄存器中 ID/EX 中找到。这个名称的第一部分，即点号的左边，表示流水线寄存器的名称；第二部分表示寄存器中字段的名称。使用这种表示方法，4 个冒险条件分别是：

- 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

303  
304

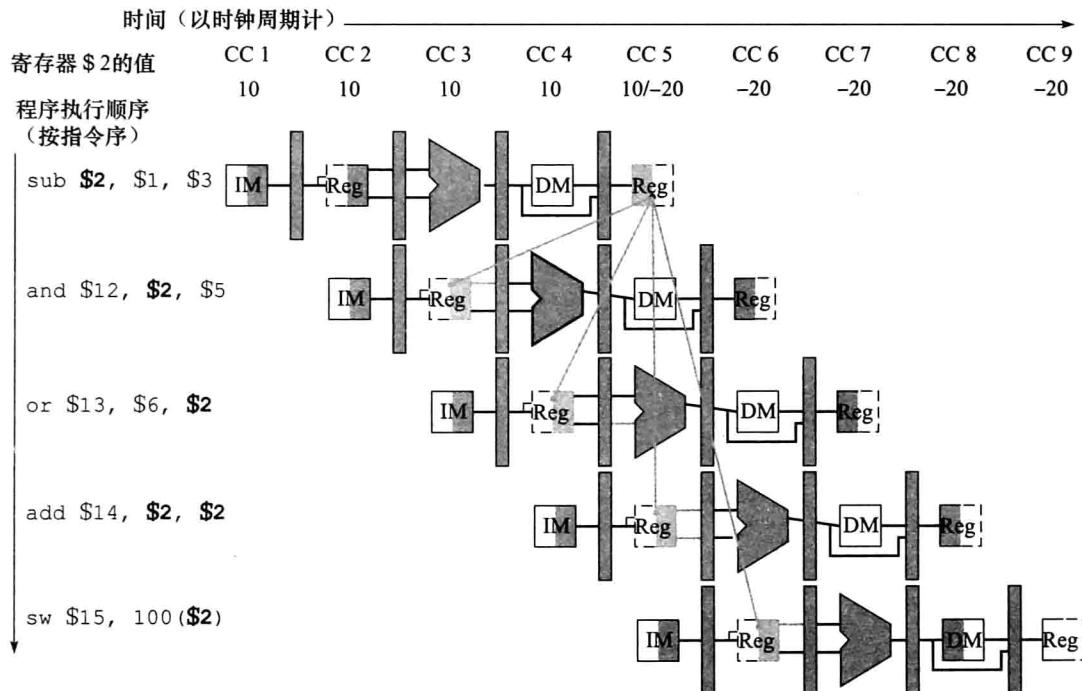


图 4-52 使用简化数据通路表示一个由 5 条指令组成的指令序列中的流水线相关情况来说明相关问题。所有的相关都用灰色标记出来，顶部的“CC 1”表示第 1 个时钟周期。指令序列中第一条指令写寄存器 \$2，后 4 条指令读寄存器 \$2。寄存器 \$2 在第 5 个时钟周期被写入，所以在此之前它的值都是无效的。（当这样的写操作发生时，一个时钟周期中寄存器的读操作返回该周期前半段写入的值。）数据相关性用数据通路中从顶部到底部的灰线表示。那些导致时间后退的依赖就是流水线数据冒险

305

本节开始给出的指令序列的第一个冒险发生在 sub \$2, \$1, \$3 的结果和 and \$12, \$2, \$5 的第一个读操作数之间。这个冒险在 and 指令处于 EX 级而 sub 指令处于 MEM 级时就能检测出来，这就是冒险 1a：

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

### 01 例题·相关性检测

将前面指令序列中的相关性进行分类：

```
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
```

306

### 01 答案

如上所述，sub-and 是一个 1a 类冒险。其余的冒险分别是：

- sub-or 是一个 2b 类冒险：

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

- sub-add 上的两个相关性都不是冒险，因为在 add 的 ID 级寄存器堆已能提供相应的数据。
- 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 后才读取 \$2。□

但是，直接采用总是旁路的方式解决冒险是不正确的，因为某些指令可能不写回寄存器，

就会产生一些不必要的旁路。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 级的 WB 控制字段以确定 RegWrite 是否被有效。而且，MIPS 要求 \$0 始终为 0，这就需要在目标寄存器是 \$0 的情况下（如 `sll $0, $1, 2`），必须避免把 \$0 按非零结果旁路，从而使得汇编程序员和编译器不必考虑 \$0 作为目标寄存器的情况。因此，需要在第一类冒险条件中加入附加条件 EX/MEM. RegisterRd ≠ 0，在第二类冒险条件中加入附加条件 MEM/WB. RegisterRd ≠ 0。

至此，我们介绍了检测冒险的方法，问题已经解决了一半，但仍然需要解决旁路数据策略的问题。

图 4-53 描述了图 4-52 的指令序列中流水线寄存器和 ALU 输入间的相关性。与图 4-52 不同的是，这里的相关性开始于一个流水线寄存器而不是等待 WB 级写操作的寄存器堆。由于流水线寄存器保存了需要旁路的数据，因此后面的指令能够获得相应的数据。

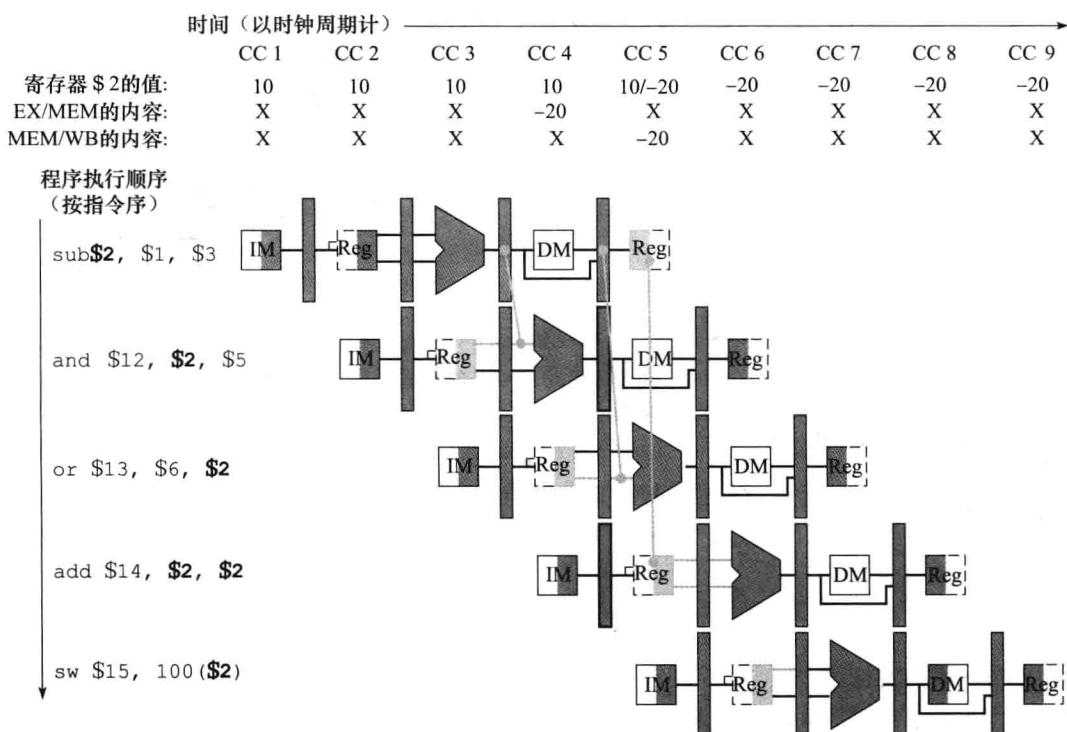
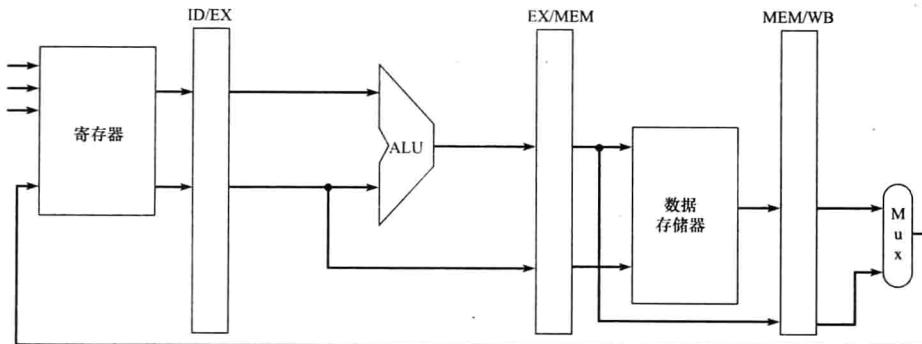


图 4-53 流水线寄存器间的相关随着时间移动，所以通过旁路流水线寄存器中保存的结果就有可能提供 AND 指令和 OR 指令所需的 ALU 输入。流水线寄存器存有相应的值，在数据写入寄存器堆之前就已经有效了。如果寄存器堆可在同一时钟周期内旁路要读写的数据，add 指令就不用阻塞了。这种寄存器堆的旁路的值不是来自于流水线寄存器而是来自寄存器堆。它使得寄存器 \$2 中的值在第 5 个时钟周期的开始是 10，而在周期结束时是 -20，即在这一时钟周期里读操作读到的值是写操作写入的值。在本节下面的部分，我们将处理所有的旁路（除了存储指令要存的数值之外）

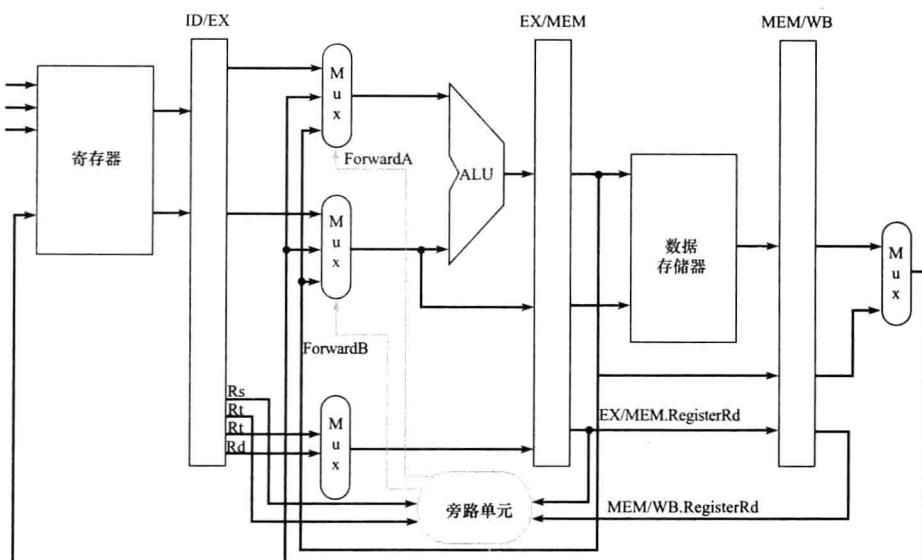
如果可以从任何流水线寄存器而不仅仅从 ID/EX 中得到 ALU 的输入，那么就可以旁路所需的数据。通过在 ALU 的输入中加入多选器和正确的控制策略，就可以在存在相关性的情况下仍然能够全速运行流水线。

现在，假设需要旁路的指令只有 4 个 R 型指令：add、sub、AND 和 OR。图 4-54 给出了在加入旁路机制前后，ALU 和流水线寄存器的示意图。图 4-55 给出了在寄存器堆值和某一旁路的数值间进行选择的 ALU 多选器控制信号的值。

因为 ALU 旁路多选器在 EX 中，所以旁路控制也在这一级中完成。因此，我们必须通过 ID/EX 流水线寄存器从 ID 级中获得操作数寄存器号，以决定是否旁路相应的值。我们已经有了 rt 字段（20~16 位）。在支持旁路前，ID/EX 流水线寄存器未保存 rs 字段。因此，为支持旁路，rs(25~21 位) 被加入 ID/EX 流水线寄存器中。



a) 未使用旁路



b) 使用旁路

图 4-54 加入旁路机制前后的 ALU 和流水线寄存器。下图使用多选器增加了旁路路径，并标识了旁路单元。新硬件用灰色表示。本图只是一个示意图，没有标识诸如符号扩展硬件之类的细节。需要注意的是，尽管 ID/EX. RegisterRt 字段在图中标识了两次，一根连接到多选器，一根连接到旁路单元，但实际上它是一个信号。如前所述，这里还忽略了旁路存储指令中数据的情况。还有点要注意的是，这一机制也适用于 slt 指令

307  
309

多选器控制	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 运算结果旁路获得
ForwardA = 01	MEM/WB	第一个 ALU 操作数从数据存储器或者前面的 ALU 结果中旁路获得
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 运算结果旁路获得
ForwardB = 01	MEM/WB	第二个 ALU 操作数从数据存储器或者前面的 ALU 结果旁路获得

图 4-55 图 4-54 中旁路多选器的控制信号。作为 ALU 另一个输入的带符号立即数将在本节的“精解”部分中解释

下面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
```

注意，EX/MEM.RegisterRd 字段是 ALU 指令（来自 Rd 字段）或装载指令（来自 Rt 字段）的目标寄存器号。

这种情况是将前一条指令的结果旁路到任何一个 ALU 输入中。如果前一条指令要写寄存器堆且要写的寄存器号与 ALU 输入要读的寄存器号（A 或 B）一致（只要不是寄存器 0），那么就调整多选器从流水线寄存器 EX/MEM 中读取数值。

2) MEM 冒险：

```
if (MEM/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
```

如上所述，在 WB 级不会发生冒险，这是因为我们假设在 ID 级指令读取的寄存器与 WB 级指令写入的寄存器是同一寄存器时，就由寄存器堆提供正确的结果。这样，寄存器堆实现了另一种形式的旁路，但这种旁路只发生在寄存器堆内部。

更为复杂的潜在数据冒险发生在 WB 级的指令结果、MEM 级的指令结果和 ALU 级的指令源操作数之间。例如，在一个寄存器中对多个数字进行求和运算时，一系列连续的指令将会读写到同一寄存器：

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

310

在这种情况下，由于 MEM 级的结果是最新的，因而结果是由 MEM 级旁路得到。这样，对 MEM 冒险的控制策略为（额外加入的条件采用粗体表示）：

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.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 not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

图 4-56 给出了为了支持旁路 EX 级结果所增加的必要硬件设备。注意，图中 EX/MEM.RegisterRd 字段是一条 ALU 指令（来自 Rd 字段）或装载指令（来自 Rt 字段）的目标寄存器。

4.13 节给出了两段 MIPS 代码，其中存在需要使用旁路解决的冒险，你可以使用单时钟周期流水线图对这些例子进行深入分析。

**01 精解** 旁路还可以帮助解决因存储指令依赖其他指令而导致的冒险。由于存储指令在

MEM 级只使用一个数据，所以旁路应当容易。但在 MIPS 架构中，由于存储器之间的复制很频繁，必须考虑复制时存储指令后紧跟着的是装载指令的情况。为了提高复制的速度，我们需要加入更多的旁路硬件。如果我们重画图 4-53，并分别使用 `lw` 和 `sw` 指令代替 `sub` 和 `AND` 指令，我们将发现这时也可能避免一次阻塞，只要装载指令的 MEM/WB 寄存器中存在的数据能够及时地提供给存储指令在 MEM 级使用。为了实现这个功能，我们需要在存储器访问阶段加入旁路。我们将如何对其修改作为练习留给读者。

此外，图 4-56 中省略了装载指令和存储指令所需的输入到 ALU 的带符号立即数。由于中央控制决定如何在寄存器和立即数之间进行选择，而且旁路单元选择流水线寄存器作为 ALU 的一个寄存器输入，因此最简单的解决方法就是加入一个 2:1 的多选器，由它在 ForwardB 多选器的输出和带符号立即数之间进行选择。图 4-57 描述了这种变化。

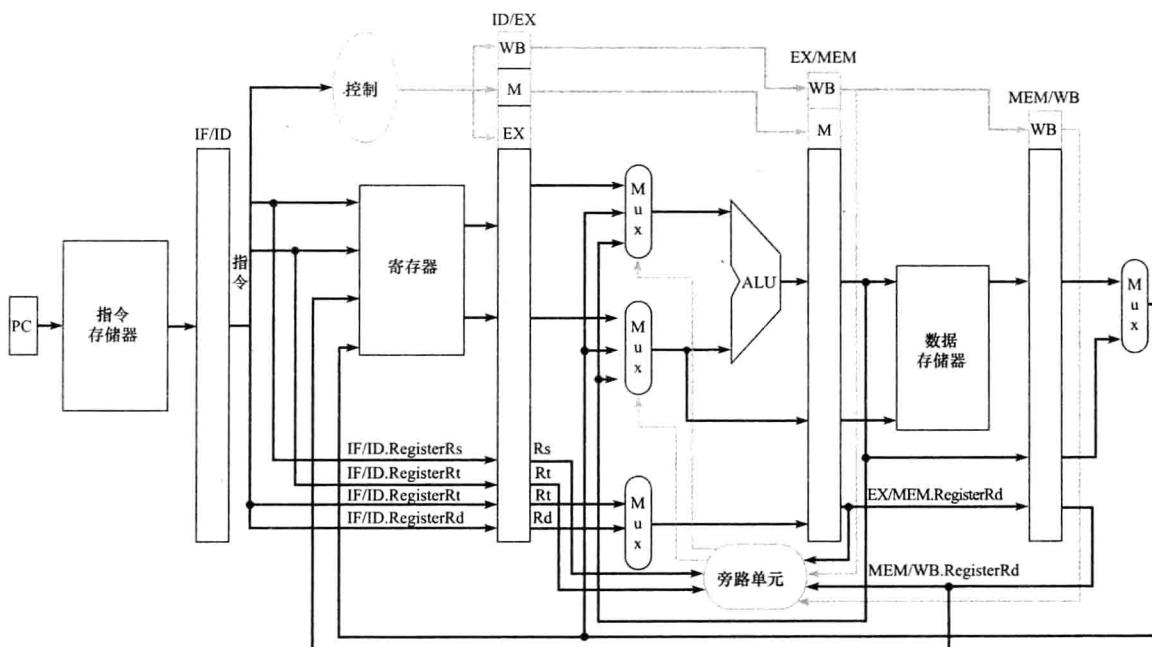


图 4-56 通过旁路解决冒险的数据通路。与图 4-51 的数据通路相比，本图在 ALU 的输入部分加入了多选器。为了使表述更加清楚，图中忽略了完整数据通路中的一些细节，如分支硬件和符号扩展硬件等。

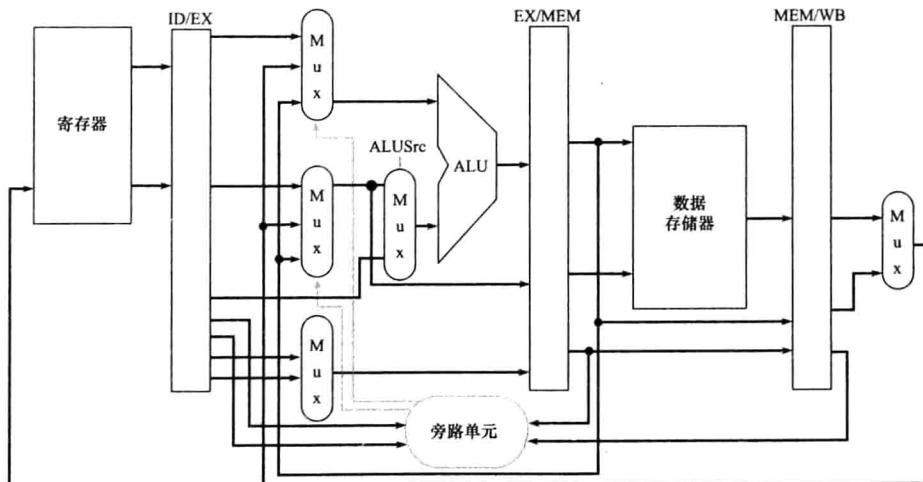


图 4-57 在图 4-54 中的数据通路加入了一个 2:1 的多选器，用以选择带符号立即数作为 ALU 的输入。

## 数据冒险与阻塞

如果你第一次没有成功，那就重新定义成功是什么。

——佚名

如4.5节所述，当一条指令试图读取一个由前一条装载指令读入的寄存器时，就无法使用旁路解决冒险了。图4-58说明了这个问题。当ALU正在执行后续指令的操作时，数据仍然是在第4个时钟周期从内存中读出的。所以，当装载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。

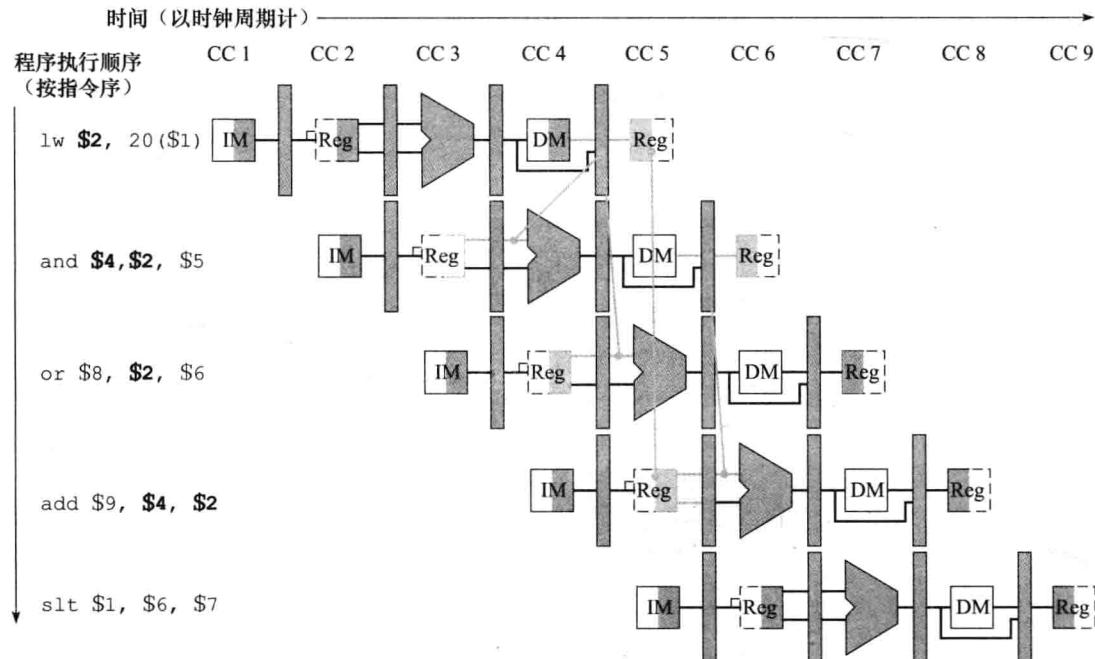


图4-58 一个指令序列的流水线图。由于装载指令和紧随其后的and指令之间的相关性在时间上是回溯的，这种冒险不可能通过旁路来解决。因此，这类指令组合导致冒险检测单元产生阻塞

313

因此，除了一个旁路单元以外，还需要一个冒险检测单元。它工作在ID级，从而可以在装载指令与紧随其后需要它的结果的指令间插入阻塞。这个冒险检测单元检测装载指令，它的控制满足如下条件：

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

因为读取数据存储器的指令一定是装载指令，所以第一行条件检查指令是否是一条装载指令。后面的两行是检测在EX级的装载指令的目的寄存器是否与在ID级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，旁路逻辑就可以处理相关性并继续执行程序了（如果没有采用旁路，那么图4-58中的指令还需要阻塞一个周期）。

如果处于ID级的指令被阻塞，那么处于IF级的指令也必须被阻塞，否则，已经取到的指令就会丢失。防止这两条指令继续执行的方法是保持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级的指令将继续使用相同的PC取指令，而在ID级将继续使用IF/ID流水线寄存器中的相同的指令字段读寄存器堆。再回到我们熟悉的洗衣店的例子

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，就像烘干机一样，从 EX 开始的流水线后半部分必须“空转”，它们执行的指令必须不产生任何效果，即空指令（nop）。

### ② 空指令：一种不进行任何操作或不改变任何状态的指令。

那我们怎么在流水线中插入空指令（就像气泡一样）呢？从图 4-49 中我们知道，在 EX、MEM 和 WB 级如果将所有 9 个控制信号都清除（置为 0），就会产生一个“什么都不做”的指令，即空指令。通过识别 ID 级的冒险，可以在流水线中插入一个气泡，方法是把 ID/EX 流水线寄存器的 EX、MEM 和 WB 级的控制信号都置为 0。这些控制信号在每个时钟周期都向前传递，但不会产生不良后果，因为如果控制信号都是 0，那么所有寄存器和存储器都不进行写操作。

图 4-59 描述了该指令序列的运行过程：与 AND 指令相关的流水线执行槽被插入一条空指令，这样从 AND 开始的所有指令都被延迟一个时钟周期。就像水管中的气泡，一个阻塞的气泡会延缓后面所有指令的执行，同时在每个时钟周期，气泡也沿着流水线向后推进一级，直到它退出流水线为止。在这个例子中，冒险强迫指令 AND 和 OR 在第 4 个时钟周期重复第 3 个时钟周期所做的内容，即指令 AND 读存储器并进行译码，指令 OR 从指令存储器中取指令。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 指令的取数操作。

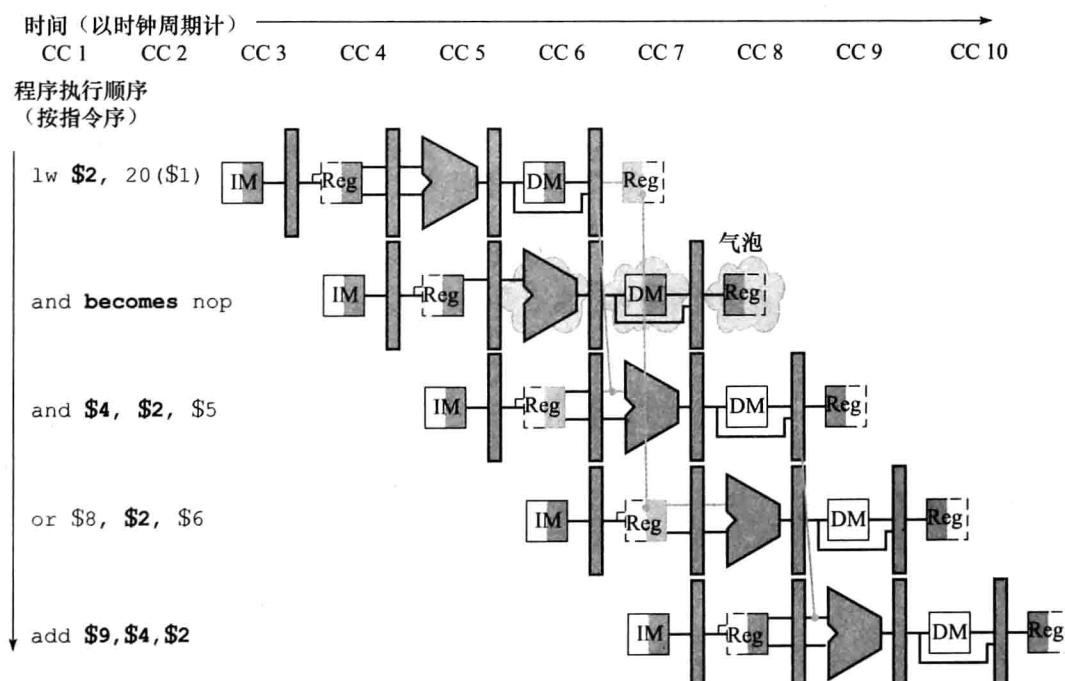


图 4-59 在流水线中插入阻塞的方法。在第 4 个时钟周期中，通过将 and 指令变成 nop 插入了一个气泡。注意，and 指令的 IF 和 ID 级在第 2 个和第 3 个时钟周期，但它的 EX 级被推迟到第 5 个时钟周期（不阻塞的话应该在第 4 个时钟周期）。与此类似，OR 指令的 IF 级在第 3 个时钟周期，但它的 ID 级被推迟到第 5 个时钟周期（不阻塞的话应该在第 4 个时钟周期）。在插入气泡后，所有的相关性沿时间前进，冒险不再发生

图 4-60 给出了冒险检测单元和旁路单元的流水线连接。和前面的介绍一样，旁路单元控制 ALU 多选器，从而可以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 流水线寄存器的写入，以及在实际控制信号与全 0 中进行选择的多选器。如果上面的取

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。如果你想了解更多细节，4.13节给出了一段 MIPS 代码，其中含有会导致阻塞的冒险，并附带了对应的单时钟周期流水线图。

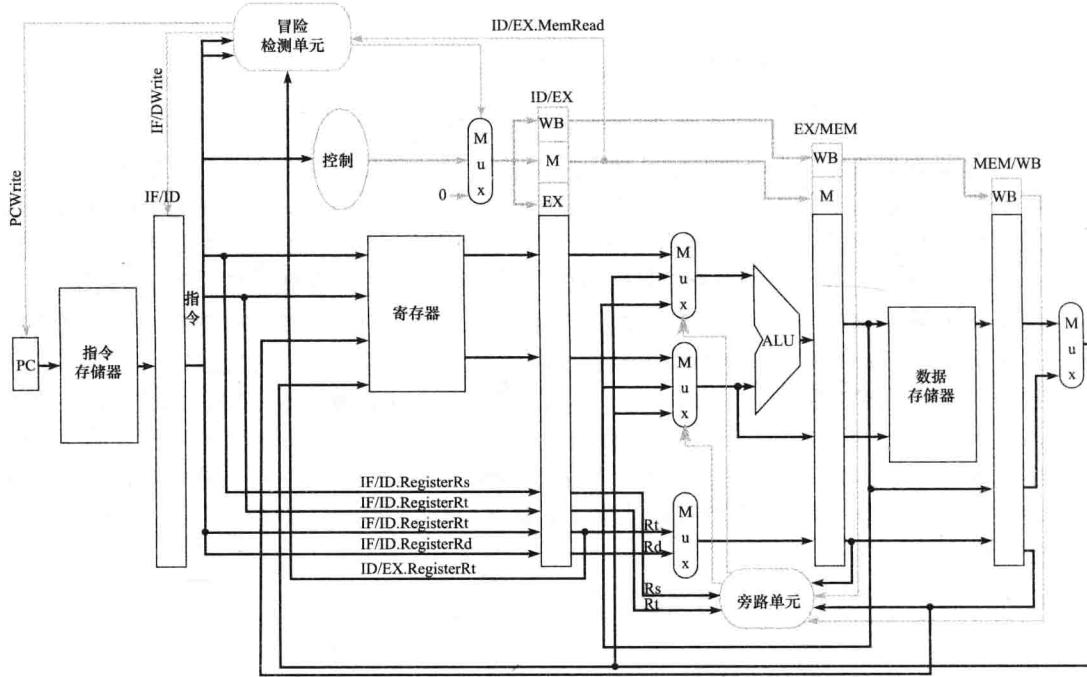


图 4-60 流水线控制概述，其中包括两个旁路多选器、一个冒险检测单元和一个旁路单元虽然简化了 ID 和 EX 级（省略了经过符号扩展的立即数和分支逻辑），但本图说明了旁路的基本硬件支持

**① 重点** 尽管编译器通常依赖于硬件解决冒险相关性以保证指令正确执行，但为了获得最好的效果，编译器的设计者必须了解流水线。否则，未预料到的阻塞会降低编译代码的执行效率。

**① 精解** 前面提到为了避免写寄存器或存储器而将所有的控制信号都置为 0。事实上，只需将信号 RegWrite 和 MemWrite 置为 0，而不用关心其他控制信号。

## 4.8 控制冒险

即使对邪恶从侧面进行上千次攻击，也比不上从根源上进行一次攻击。

——Henry David Thoreau, 《Walden》, 1854

315  
316

直到现在，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 4.5 节中所提到的那样，还有一类包含分支的流水线冒险。图 4-61 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。为了维持流水线的运行，每个时钟周期都必须取指，但在我们的设计中必须等到 MEM 级才能确定是否执行分支。如 4.5 节所述，与前面讨论的数据冒险相对应，这种为了确保预取正确指令而导致的延迟叫作控制冒险（control hazard）或分支冒险（branch hazard）。

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要小得多，而且与采用旁路就能有效地解决数据冒险相比，还没有有效的方法能够解决控制冒险。因此，这一节关于控制冒险的讨论要比前一节的数据冒险短得多。本节将介绍两种解决控制冒险的方案，并进行了优化。

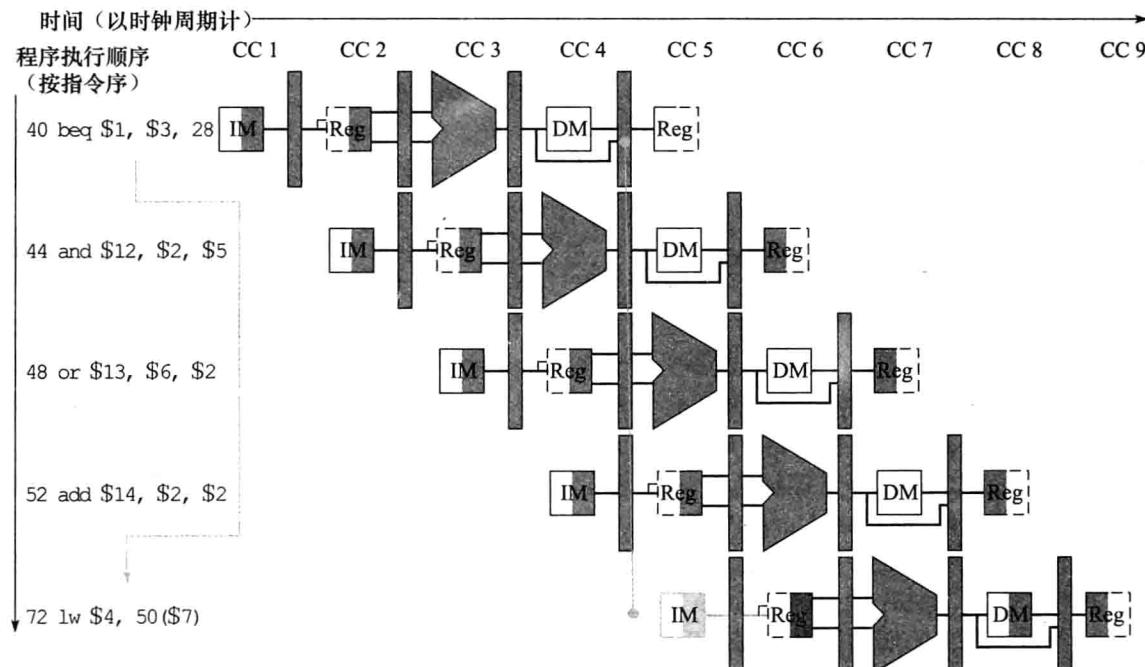


图 4-61 分支指令对流水线的影响。指令左边的数字（40, 44, …）表示指令的地址。由于分支指令在 MEM 级（beq 指令对应于时钟周期 4）才能决定是否执行分支，分支后面三条指令都将被取回并执行。如果不加干涉，这三条指令将在 beq 指令跳转到地址 72 执行 lw 之前就开始执行了（图 4-31 通过引入额外的硬件从而将控制冒险减少到一个时钟周期，本图使用的是没有经过优化的数据通路）

#### 4.8.1 假定分支不发生

如 4.5 节所述，采用阻塞直到分支判断完毕来处理控制冒险的速度实在太慢。一种比较普遍的提高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，就丢弃已经读取并译码的指令，并按分支目标继续执行。如果分支不发生的可能性是 50%，同时丢弃指令的代价很小，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，只需要将最初的控制信号置为 0 即可，这一点与阻塞解决装载指令的数据冒险类似。其不同之处在于当分支到达 MEM 级时必须分别改变在 IF、ID 和 EX 级的三条指令的控制信号，而对于装载指令的阻塞只需要将 ID 级的控制信号置为 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 级的指令都清除（flush）。

○ 清除：因发生了意外而丢弃流水线中的指令。

#### 4.8.2 缩短分支的延迟

一种提高分支效率的方法是缩短分支的执行时间。直到现在，我们都假设在 MEM 级才能确定分支结构要执行的下一条指令的 PC。确定分支目标地址的时间越早，需要清除的指令就越少。MIPS 体系结构是面向支持快速的单周期分支设计的。设计者注意到许多分支仅仅需要简单的判断（如相等或正负），这些简单的判断并不需要完整的 ALU 操作而仅使用简单的一些逻辑门就足够了。如果分支条件更复杂，一般有一条单独的指令使用 ALU 来进行比较——这种情况类似于第 2 章中提到的分支条件码。

为了将分支决策提前，需要提前两个动作：计算分支目标地址和判断分支条件。分支目标

地址的计算是比较简单的。我们在 IF/ID 流水线寄存器中已经有了 PC 的值和立即数字段，所以只需要将分支地址计算从 EX 级移到 ID 级就可以了。当然，尽管分支目标地址对所有指令都会计算，但仅在需要时才会使用。

判断分支条件比较复杂。为了判断分支的执行条件，需要比较从 ID 级取到的两个寄存器的值是否相等。判断相等的方法可以是先将对应的位进行异或操作，然后将结果按位进行或操作。为了把分支条件判断提前到 ID 级，还需要额外的旁路和冒险检测硬件，因为分支条件的判断可能依赖于还在流水线中的结果。例如，为了实现相等则分支（或不等则分支），我们需要旁路结果至 ID 级进行相等检测。这里有两个比较复杂的因素：

317  
318 1) 在 ID 级指令译码后，决定是否需要将所需数据旁路到相等检测单元进行相等检测。如果是分支指令，就可以把 PC 替换为分支目标地址。旁路分支指令的操作数以前是由 ALU 旁路单元来完成的，但 ID 级相等检测单元的引入需要一个新的旁路单元。必须注意的是，需要旁路的分支指令源操作数可能来自 ALU/MEM 或 MEM/WB 流水线寄存器。

2) 因为 ID 级进行分支比较所需的数据可能在后面才能产生，因此有可能会发生数据冒险，这样就需要阻塞流水线。例如，如果分支指令前刚好是一条 ALU 指令，而这条 ALU 指令的结果恰是分支指令比较所需要的，那么必然产生阻塞，因为 ALU 指令的 EX 级将在分支指令的 ID 级后发生。再举一个例子，如果分支指令前刚好是一条装载指令，而装载指令的结果恰是分支指令判断所需要的，则必须产生两个阻塞，因为装载指令的结果将在装载指令的 MEM 级结束时产生，但在分支指令的 ID 级开始时就会用到。

尽管有这些困难，将分支执行提前到 ID 级依然是值得的，因为它将分支预测错误的代价减小到只有一条指令，就是分支执行时正在取的那条指令。下面的例题对旁路路径和检测冒险的实现细节进行了讨论。

为了在 IF 级清除指令，我们加入了一条称为 IF. Flush 的控制信号，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空指令，该指令不作任何操作，不改变机器状态。

### 01 例题·流水线分支

假定流水线对分支不发生进行了优化，并且分支的执行提前到流水线的 ID 级。试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)
```

### 01 答案

图 4-62 描述了分支产生时指令序列的执行情况。与图 4-61 不同，这里在一个发生的分支上只有一个流水线气泡。□

### 4.8.3 动态分支预测

假设分支不发生是一种粗略的分支预测方法。在这种情况下，我们总是预测分支不发生，如果预测错误就清空流水线。对简单的五级流水线而言，这种方法结合基于编译器的预测就已经足够了。从时钟周期数的角度来说，使用更深的流水线时分支代价将增加。类似地，以丢弃的指令数来计算，对多发射（见 4.10 节）的分支代价也将增加。这种组合意味着在一个高性

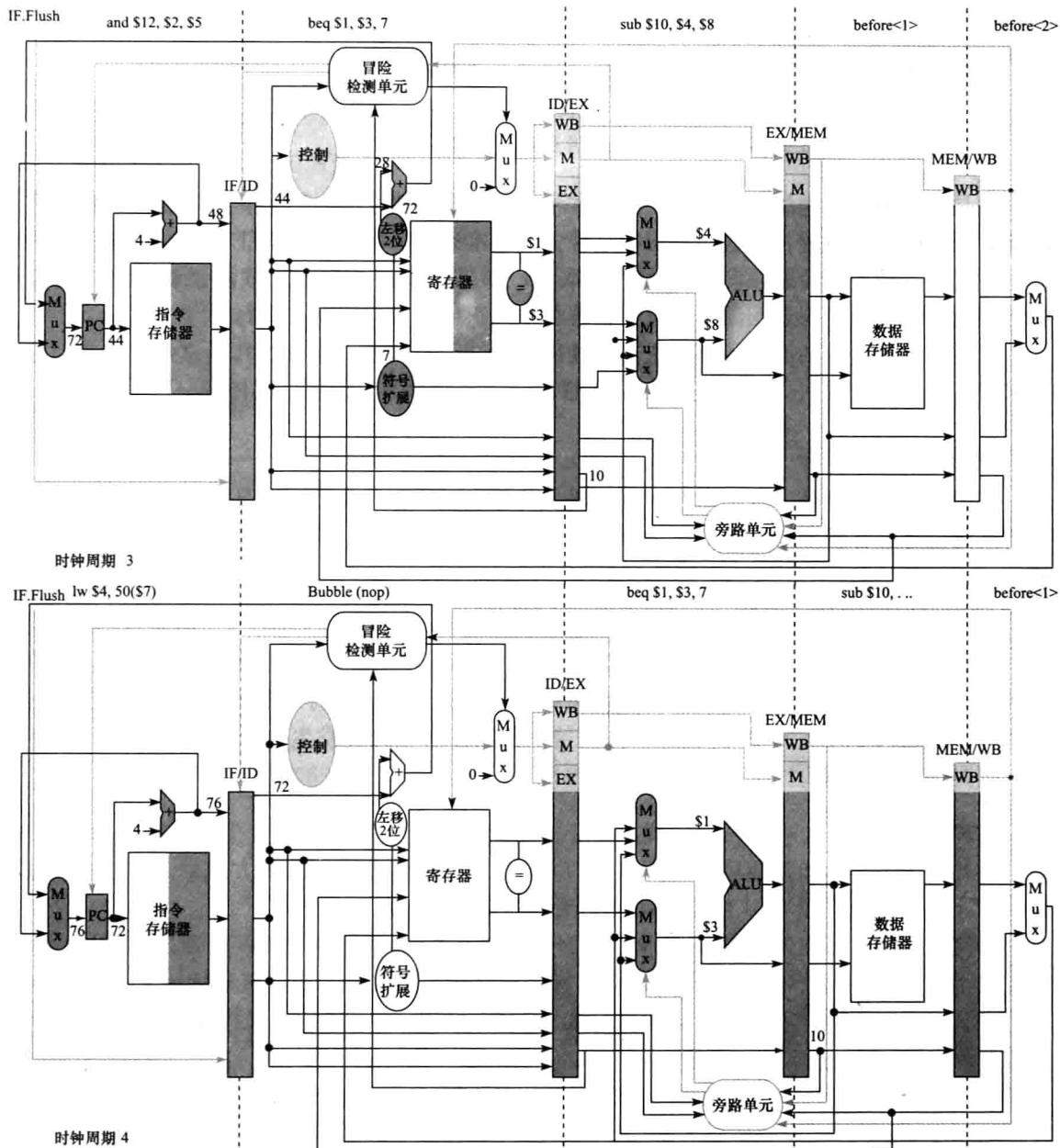


图 4-62 在第三个时钟周期 ID 级确定分支发生，因此地址 72 被选为下一个 PC 地址，同时将为下一个时钟周期预取的指令置为 0。时钟周期 4 的图描述了地址为 72 的指令被取回，并且分支发生的后果是在流水线中产生了一个气泡或者一条空指令（由于空指令实际上是 `sll $0, $0, 0`，所以时钟周期 4 的 ID 级是否应该标出还有待商榷）

能的流水线设计中，简单的静态预测机制将可能浪费大量的性能。如 4.5 节所述，如果有更多的硬件支持，我们就可能实现一些其他的分支预测方法。

一种策略是通过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新的指令。这种技术称为动态分支预测（dynamic branch prediction）。

这种策略的一种实现方法就是采用分支预测缓存（branch prediction buffer）或分支历史记录表（branch history table）。分支预测缓存是一小块按照分支指令的地址低位索引的存储器区，

其中包括一位或多位数据用以说明最近是否发生过分支。

- ② 动态分支预测：根据运行信息在运行中进行分支预测。
- ③ 分支预测缓存：也称为分支历史记录表。一小块按照分支指令的低位地址索引的存储器区，其中包括一位或多位数据用以说明最近是否发生过分支。

这是最简单的一类缓存，我们实际上并不知道预测是否正确，而且它还可能由其他具有相同地址低位的分支设置，但这并不影响这种方法的准确率。预测只是对正确分支方向的一种假设，在这个基础上，沿着预测的方向进行取指。如果这种假设错误，预测错误的指令将被删除，预测位将取反，并返回原来的位置，继续按照正确的方向取指并执行。

使用一位预测位的简单预测方法具有性能上的缺陷：即使一个分支几乎总是发生，但它一旦未发生就将导致两次（而不是一次）预测错误。下面的例子说明了这种情况。

### 01 例题·循环与预测

我们来看一个循环分支，它在一行代码上的分支发生了9次，而不是发生了一次。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？

### 01 答案

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9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。而在第一次迭代时发生预测错误是因为预测位在循环的上一次迭代时被前一个执行设置为不执行（在那次退出的迭代中分支并没有发生）。因此这个预测方法在分支发生90%的情况下预测的正确性只有80%（两次错误预测，8次正确预测）。 □

320  
321

在理想的情况下，在这种高度规律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。在一个两位预测位的方案中，再次发生预测错误时才改变预测。图4-63给出了两位预测位的有限状态机。

分支预测缓存可以使用在IF级指令地址能够访问的小容量专用缓存实现。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取指（如4.8.3节所述，在ID级就可以获得PC），否则就顺序取指并继续执行。如果预测的结果是错误的，就按照图4-63说明的方法改变预测位。

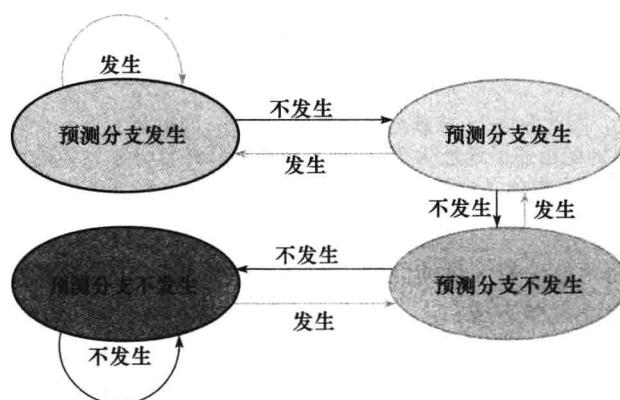


图4-63 两位预测位机制的状态图。通过使用两位（不是一位）预测位，在分支经常发生或经常不发生的情况下（大多数分支都是这样）只会发生一次预测错误。两位数据在系统中可以表示4种状态。这种两位方案是基于计数器预测方法的一个应用。基于计数器的预测方法是当预测成功时计数器加1，预测失败时计数器减1，然后使用计数器表示范围的中点作为分支与不分支的分界点。

### 01 精解

如4.5节所述，在五级流水线中，通过重新定义分支，我们可以将控制冒险转化为一种可用的特性。延迟分支可执行下一条指令，但分支指令后的第二条指令仍将受到分支的影响。

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时间槽 (branch delay slot)。这些软件的作用就是使后续的指令有效并且有用。图 4-64 给出了三种调度分支延迟时间槽的方法。

延迟分支调度的限制在于：对能够被调度到分支延迟时间槽中的指令的限制；在编译时对分支发生与否的预测能力。

对每个时钟周期发射一条指令的五级流水线处理器而言，延迟分支是一种简单有效的方法。随着处理器向更深流水线以及单周期多指令的方向发展（见 4.10 节），分支延迟变得更长，单延迟时间片实际上并没有多大作用。所以，与开销大但更灵活的动态预测方法相比，延迟分支技术已经失去了吸引力。同时，根据摩尔定律使动态预测的成本相对更低。

- ② 分支延迟时间槽：紧跟延迟分支指令的时间片。在 MIPS 体系结构中，用不影响分支的一条指令填充到该时间片中。

322

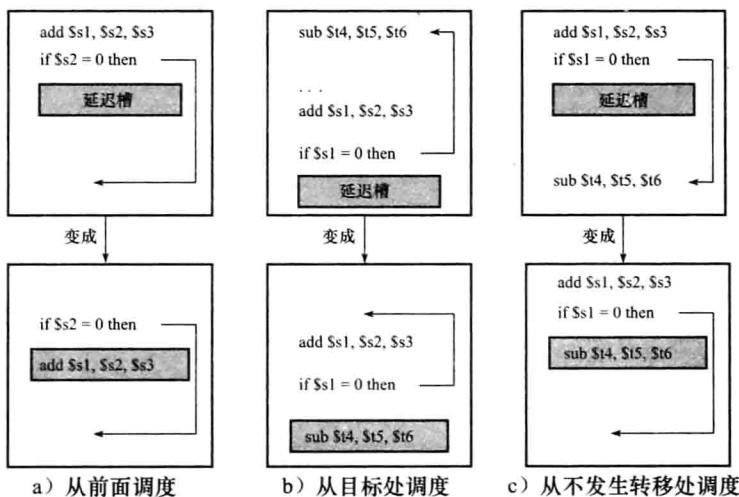


图 4-64 分支延迟时间槽的调度。每一对方框中的上面一个表示调度前的代码，下面一个表示调度后的代码。在方案 a 中，延迟时间片通过插入分支之前的一条与分支无关的指令实现，这是一种最佳的选择。当方案 a 无法实现时，就使用方案 b 和方案 c。在方案 b 和方案 c 的代码序列中，分支条件中使用了 \$1，因而不能将 add 指令（它的目的寄存器是 \$1）移入分支延迟时间槽。方案 b 中的分支延迟时间槽是按照分支目标地址调度的。由于目标指令可以通过其他路径访问到，通常需要将它们进行复制。当分支发生的可能性比较大时，一般选择方案 b，如循环分支。最后，也可能采用方案 c 预测分支不发生的下一条指令进行调度。为了使方案 b 和方案 c 中的优化合法，sub 指令必须在分支预测错误时也能“正常”执行。“正常”意味着虽然有些工作是多余的，但程序依然能够正确执行。例如，当分支预测错误且 \$t4 是未被使用的临时寄存器时，就是这种情况

323

**01 精解** 分支预测器告诉我们分支是否会发生，但我们依然需要计算分支目标地址。在五级流水线中，计算分支目标地址需要一个时钟周期，即分支发生将需要一个时钟周期的开销。延迟分支是消除这个开销的一种方法。另一种方法是使用分支目标缓存 (branch target buffer) 保存分支目标地址或分支目标指令。

两位的动态预测机制仅使用某个特定分支的信息。研究人员发现，在使用相同数量的预测位的情况下，同时使用局部分支和最近执行分支的全局行为信息，能够产生更高的预测精度。这种预测器称为相关预测器 (correlating predictor)。一个典型的相关预测器为每个分支提供两个两位的预测器，其选择依据是上次分支执行的结果（分支发生与否）。这样，全局分支行为可以被看成是在预测查找表中加入额外的索引位。

最新的分支预测方法是竞争预测器。竞争预测器 (tournament branch predictor) 对每

一个分支使用多个预测器，并记录哪个预测器的预测结果最好。目前竞争预测器的预测是最准确的。典型的竞争预测器对每个分支地址有两个预测：一个基于局部信息，一个基于全局分支行为。有一个选择器用于选择哪个预测器的预测结果，其操作类似于一位或两位的预测器，而两位预测器的精度更高。一些最新的微处理器使用了这种预测器。

- ② 分支目标缓存：一种用于缓存分支目标地址或分支目标指令的结构，其一般形式为带标志位的 cache，因而其硬件开销大于简单的分支预测缓存器。
- ③ 相关预测器：综合考虑特定分支的局部行为和最近执行分支的全局行为的分支预测器。
- ④ 竞争预测器：具有多种预测机制的分支预测器，其带有一个选择器，对给定分支可选择其中一个作为预测结果。

### 01 精解 一种减少条件分支数量的方法是加入条件移动指令（conditional move instruction）。

不同于条件分支指令改变 PC 值，条件移动指令将根据条件改变移动的目的寄存器。如果条件不成立，条件移动指令就相当于一条 nop 指令。例如，某版本的 MIPS 体系结构指令集包含 movn（move if not zero）和 movz（move if zero）两条指令。例如，movn \$8, \$11, \$4，如果寄存器 \$4 的值为非零，该指令复制寄存器 \$11 的内容至寄存器 \$8；否则，该指令什么也不做。

ARMv7 指令集在绝大多数指令中都有条件字段。因此，ARM 程序一般比 MIPS 程序的条件分支要少一些。

## 4.8.4 流水线小结

我们从洗衣店的例子开始，介绍了日常生活中的流水线原理。用这个例子类比，逐步解释了指令的流水化，即在单周期数据通路的基础上逐步增加流水线寄存器、旁路路径、数据冒险检测、分支预测和异常时指令的清除。图 4-65 给出了最终的数据通路及控制。现在我们已经准备好处理另一种控制冒险：异常。

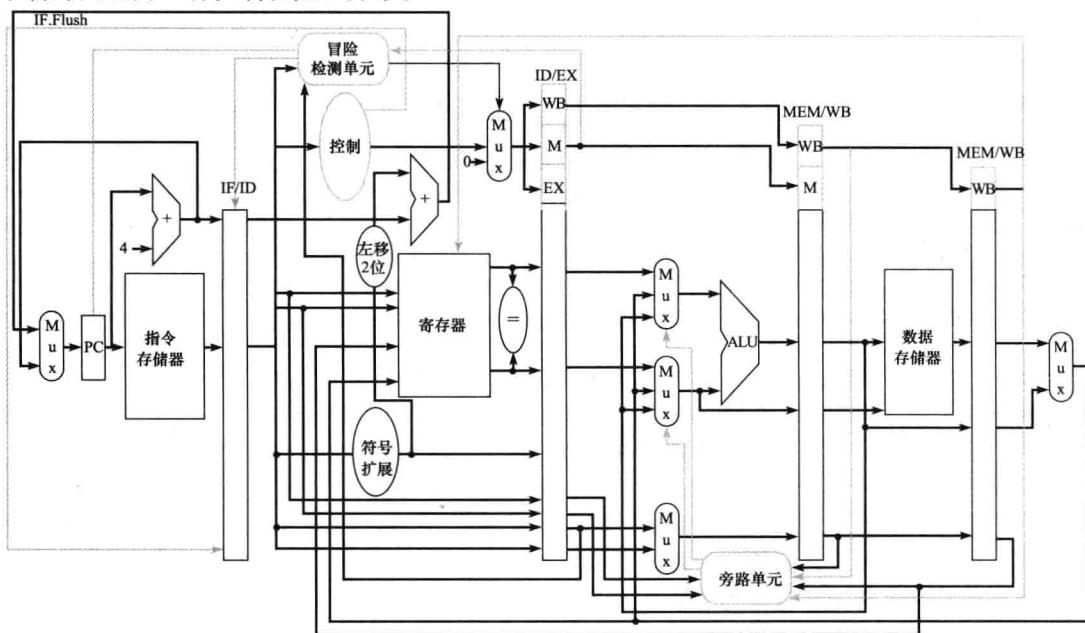


图 4-65 本章最终的数据通路与控制。注意，这是一个概略图，没有覆盖到数据通路的所有细节，如图 4-57 中的 ALUsrc 多选器和图 4-51 中的多选器控制都没有标识出来

### 01 小测验

考虑三种分支预测机制：预测分支不发生、预测分支发生和动态分支预测。假定它们在预测正确时无开销，预测错误时开销为两个时钟周期，动态预测器的平均准确率为 90%。在此情况下，对下面的分支而言哪种预测器是最好的选择？

1. 分支发生概率为 5%。
2. 分支发生概率为 95%。
3. 分支发生概率为 70%。

324

## 4.9 异常

使一台计算机具有自动程序中断能力并非一件简单的事，因为中断发生时处于不同执行阶段的指令数量可能非常多。

——Fred Brooks, Jr., 《Planning a Computer System: Project Stretch》, 1962

控制是处理器设计中最具挑战性的一个方面：它最难达到正确，也最难提高速度。控制中最难的部分之一是实现异常（exception）和中断（interrupt）——除分支以外改变正常指令执行顺序的事件。异常和中断最初是用来处理来自处理器内部的意外事件，如算术溢出。在第 5 章中我们将看到，它们也可用于 I/O 部件与处理器的通信。

- 异常：也称为中断，指打断程序正常执行的突发事件，用于检测溢出等。
- 中断：来自处理器外部的异常。（某些体系结构也用“中断”一词表示所有的异常。）

许多体系结构和作者不区分中断和异常，统称为中断，如 Intel x86。我们遵循 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，术语中断指由外部引起的事件。下面 5 个例子说明了在处理器内部或外部的事件情况。

事件类型	来源	对应的 MIPS 术语
I/O 设备请求	外部	中断
用户程序进行操作系统调用	内部	异常
算术溢出	内部	异常
使用未定义的指令	内部	异常
硬件故障	内部或外部	异常或中断

导致异常发生的不同情况对异常处理的支持提出了诸多要求。在第 5 章中我们将再次讨论这个话题，从而更加清楚地理解这一问题。本节讨论两种异常的检测机制，这两种异常由我们讨论过的指令集及其实现方式产生。

检测异常情况并采取适当举措，通常处于处理器的关键路径上。该路径决定了时钟周期的长度以及处理器性能。如果在控制单元的设计中没有充分考虑异常，那么在复杂实现中加入异常支持会明显降低性能，并使正确的设计更加复杂。

### 4.9.1 MIPS 体系结构中的异常处理

目前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。在接下来的部分，我们将使用 add \$1, \$2, \$1 指令作为算术溢出类型异常的例子。异常发生时处理器必须进行的基本操作是：在异常程序计数器（Exception Program Counter, EPC）中保存出错指令的地址，并把控制权转交给操作系统的特定地址。

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，

325  
326

也可以继续执行程序，此时由 EPC 决定重新开始执行的地方。在第 5 章将更详细地讨论重新开始执行的问题。

为了处理异常，操作系统除了要知道是哪条指令引起异常之外，还必须知道引起异常的原因。主要有两种方法用于表示产生异常的原因。MIPS 使用的方法是设置一个状态寄存器（称为 Cause 寄存器），其中有一个字段用于记录异常产生的原因。<sup>②</sup>

另一种方法是使用向量中断（vectored interrupt）。在向量中断中，控制权被转移到由异常原因决定的地址处。<sup>③</sup>例如，为处理前面的两种异常，可定义如下的两个异常向量地址：

- ② 向量中断：由异常原因决定中断控制转移地址的中断。

异常类型	异常向量地址（十六进制）
未定义指令	8000 0000 <sub>16</sub>
算术溢出	8000 0180 <sub>16</sub>

操作系统根据引起异常的地址得知导致异常的原因。地址由 32 字节或 8 条指令进行区分，并且操作系统必须记录异常的原因，并依此顺序执行一些有限的处理。当出现的异常不属于向量异常时，单个入口点供所有异常使用，并且操作系统对状态寄存器进行译码以便找到原因。

通过给基本的实现加上一些额外的寄存器和控制信号，并将控制进行一些扩展，就可以处理异常。假定我们实现的是 MIPS 体系结构的异常处理系统，统一入口地址为 8000 0180<sub>16</sub>（事实上，实现向量异常也不难），需要给当前的 MIPS 实现加上两个寄存器：

- EPC：32 位寄存器，用于保存发生异常的指令地址（向量中断也需要这样一个寄存器）。
- Cause：记录异常原因的寄存器。在 MIPS 体系结构中它是 32 位的，虽然其中一些位现在还没有用到。假定使用一个 5 位的字段对前面两种异常原因进行编码：未定义指令 = 10，算术溢出 = 12。

#### 4.9.2 在流水线实现中的异常

在流水线实现中，异常可被视作另一种形式的控制冒险。例如，假设指令 add 产生了一个算术溢出。正如上一节对分支发生的处理，我们必须清除流水线中 add 指令后的一系列指令并从新的地址开始取指。我们将使用与之相同的机制，不过这次是由异常重置控制信号。  
327

在处理分支预测错误时，我们已经知道如何通过将 IF 级的指令转换成 nop 指令来清除指令。为了清除 ID 级的指令，我们使用 ID 级已有的多选器，将控制信号清零以产生阻塞。一个称为 ID. Flush 的新控制信号与冒险检测单元的阻塞信号相或，可以在 ID 级进行清除。为了清除 EX 级的指令，我们使用一个称为 EX. Flush 的新信号，用它控制新的多选器将控制信号清零。为了从地址 8000 0180<sub>16</sub>（MIPS 异常地址）开始取指令，只要简单地加入一个额外的输入到 PC 的多选器，由它将 8000 0180<sub>16</sub> 传递到 PC。图 4-66 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，即如果不在指令执行期间中止指令的执行，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 的目标寄存器被冲掉。这一问题可以通过下面的方法解决：异常溢出在 EX 级检测出来，可用 EX. Flush 信号避免 EX 级的指令在 WB 级写回结果。许多异常需要我们能够最终正常执行引起异常的指令。做到这一点最简单的方法是先清除这条指令，然后在异常处理完后再重新执行这条指令。

异常处理的最后一步是将导致异常的指令的地址保存到 EPC 中。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 4-66 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。

② 所有异常使用同一入口地址，操作系统根据状态寄存器确定异常原因。——译者注

③ 操作系统通过异常向量地址得知异常原因。——译者注

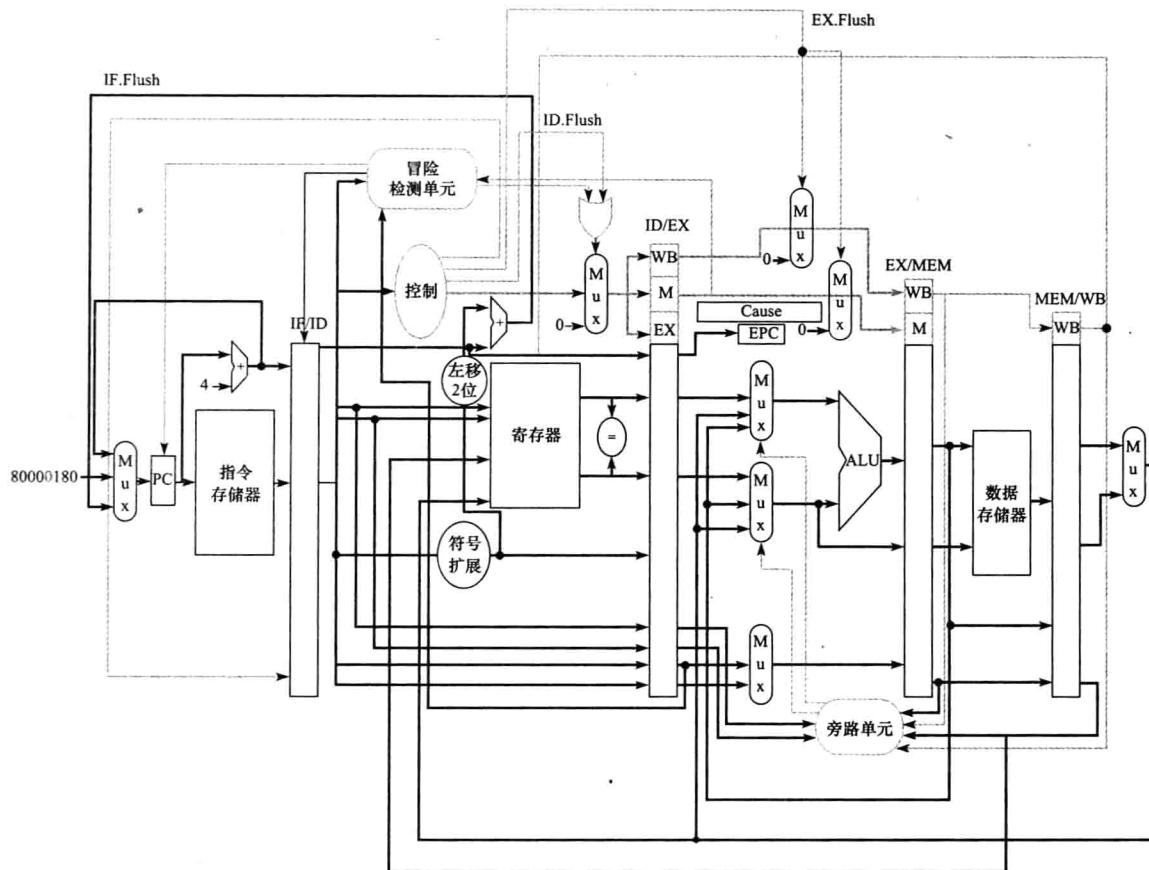


图 4-66 处理异常的数据通路与控制。主要增加了以下部分：在 PC 多选器中增加了一个新的输入  $8000\ 0180_{16}$ 、一个记录异常发生原因的 Cause 寄存器以及一个保存导致异常的指令地址的 EPC 寄存器。 $8000\ 0180_{16}$  是发生异常时开始取指令的地址。尽管图中没有表示出 ALU 溢出信号，但它也是控制单元的一个输入

### 01 例题·流水线计算机中的异常

给出以下指令序列：

```

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)
...

```

假定异常处理程序的开始部分如下：

```

80000180hex sw $26, 1000($0)
80000184hex sw $27, 1004($0)
...

```

给出 add 指令发生溢出异常时流水线的情况。

### 01 答案

图 4-67 给出了从 add 指令的 EX 级开始发生的情况。溢出在 EX 级被检测到， $8000\ 0180_{16}$  被强制放入 PC。在第 7 个时钟周期，add 指令及其后面的指令被清除，并且异常代码的第一条指令被取出。注意，保存的地址是 add 指令下一条指令的地址 ( $4C_{16} + 4 = 50_{16}$ )。

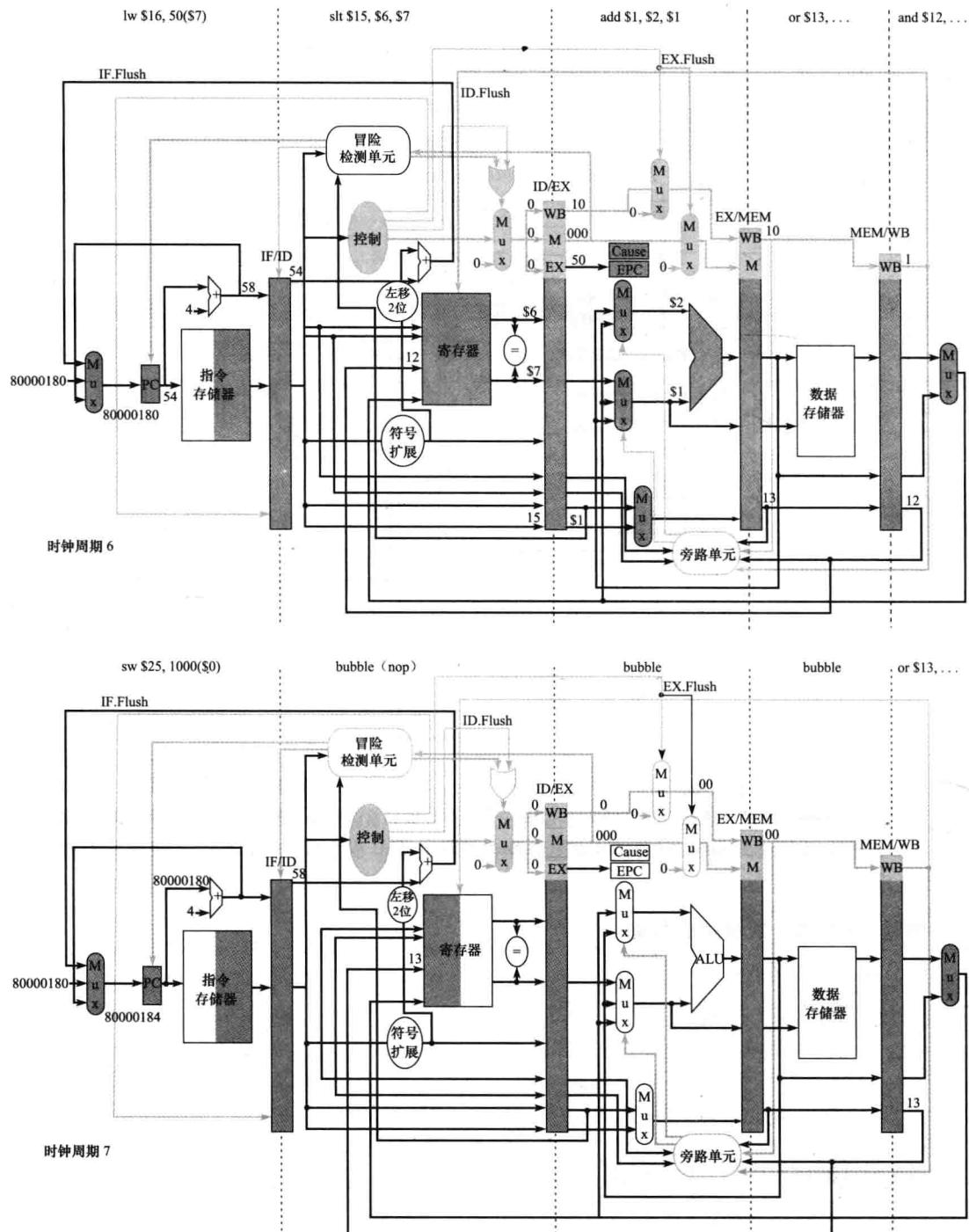


图 4-67 add 指令算术溢出导致的异常。溢出在第 6 个时钟周期的 EX 级检测到，因此将 add 后面的指令地址 ( $4C + 5 = 50_{16}$ ) 保存到 EPC 寄存器。溢出导致在该周期后面所有的 Flush 信号都设置为 1，并置 add 的控制信号为无效（置为 0）。时钟周期 7 显示了流水线中转化为气泡的指令和取异常处理程序的第一条指令 sw \$25, 1000 (\$0)（从指令地址  $8000\ 0180_{16}$  处取得）。需要注意的是，位于 add 指令前的 AND 指令和 OR 指令仍然会执行完毕。虽然图中没有画出 ALU 溢出信号，但它也是控制单元的一个输入

在前面我们曾提到 5 个异常的例子，在第 5 章我们还会看到其他的例子。任何时钟周期流水线中都有 5 条活动的指令，问题是如何确定到底是哪条指令引起了异常。而且，一个时钟周期内还可能发生多个异常。通常的解决方法是对异常划分优先级，这样多个异常同时发生时就知道先处理哪个。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 设备请求与硬件故障并不与特定的指令相关，因此它们在流水线中断时机的实现上具有一定的灵活性。因此，用于其他异常的机制在这里也可以很好地工作。

EPC 捕捉中断指令的地址，而 MIPS 的 Cause 寄存器在一个时钟周期内记录下所有可能的异常，因此异常处理软件判断出该指令发生了何种异常。一个重要的判断依据是某一类异常可能在哪一个流水线阶段发生。例如，未定义的指令异常发生在 ID 级，而调用操作系统异常发生在 EX 级。如果在 Cause 寄存器中保存有多个异常，当优先级最高的异常处理之后，会继续导致硬件中断，从而处理后面的异常。

**01 硬件/软件接口** 硬件与操作系统必须协同工作才能按照我们期望的方式处理异常。硬件一般暂停指令流中导致异常的指令，同时执行完该指令前的所有指令，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。操作系统则查看异常发生的原因并采取相应的操作。对于一个未定义指令异常、硬件错误异常或算术溢出异常，操作系统通常终止执行的程序并返回原因描述。对于 I/O 设备请求或操作系统服务调用，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。在 I/O 设备请求的情况下，我们可能需要在继续执行发出 I/O 设备请求的任务前先运行另一个任务，因为该任务一般在 I/O 完成之后才能继续执行。这就是保存和恢复任务状态如此重要的原因。一个最重要且频繁出现的异常是页缺失与 TLB 异常。第 5 章描述了更多关于这些异常及其处理的细节。

**01 精解** 在流水线计算机中将每一个异常与导致异常的相应指令对应起来的难度很大，因此一些计算机设计者在一些非关键情况下放松了这种要求，这种处理器一般称为具有非精确中断 (imprecise interrupt) 或者非精确异常 (imprecise exception)。在上面的例子中，尽管导致异常的指令地址是  $4C_{16}$ ，但在检测到异常后下一个时钟周期开始时 PC 的值通常为  $58_{16}$ 。具有非精确异常处理的处理器可能会将  $58_{16}$  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处理器都提供精确中断 (precise interrupt) 或精确异常 (precise exception) (我们将在第 5 章中看到，原因之一是为了支持虚拟存储器)。331

- ② 非精确中断：也称为非精确异常。流水线计算机中的中断或异常不与导致中断或异常的指令精确地关联。
- ② 精确中断：也称为精确异常。流水线计算机中的中断或异常与导致中断或异常的指令精确地关联。

**01 精解** 尽管 MIPS 对绝大多数异常使用  $8000\ 0180_{16}$  作为异常入口地址，但为了提高性能，对 TLB 缺失异常使用  $8000\ 0000_{16}$  作为异常入口地址（参见第 5 章）。

### 01 小测验

在下面的指令序列中会首先识别哪个异常？

1. add \$1, \$2, \$1 #算术溢出
2. XXX \$1, \$2, \$1 #未定义指令
3. sub \$1, \$2, \$1 #硬件错误

## 4.10 指令级并行

首先说明一下，本节是对一些高级主题的概述。如果你希望了解更多的细节，可以参考另一本教材：《Computer Architecture: A Quantitative Approach》第5版。本节大约13页的内容在该书中扩充到近200页（含附录）。

流水线挖掘了指令间潜在的并行性。这种并行性被称为**指令级并行**（instruction-level parallelism, ILP）。有两种方法可以增加潜在的指令级并行程度。第一种是增加流水线的深度以重叠更多的指令。还是用洗衣店的例子来说明，假设洗衣机周期比其他机器的周期要长，我们可以把洗衣机划分成三个机器，分别完成原洗衣机洗、漂、甩三个功能。这样我们就将四级流水线变成了六级流水线。为了达到完全的加速效果，我们需要重新平衡其他步骤使得它们的长度相同，在处理器和洗衣店中都是这样。因为更多的操作被重叠，有更多的并行性被挖掘出来。因为时钟周期缩短的缘故，性能会得到潜在的增强。

另一种方法是复制计算机内部部件的数量，使得每个流水级可以启动多条指令。这种技术一般被称为**多发射**（multiple issue）。一个多发射的洗衣店会把原有的一台洗衣机和烘干机替换为三台洗衣机和三台烘干机。还需要雇用更多的洗衣工来折叠和存储三倍于原来的衣服。这种方法的缺点是需要额外的工作让所有机器同时运转并将负载传到下个流水级。

- ② 指令级并行：指令间的并行性。
- ③ 多发射：一种单时钟周期内发射多条指令的机制。

332 每个阶段同时启动多条指令允许指令执行速率超过时钟速率，换句话说，就是CPI小于1。正如第1章介绍的，有时候使用IPC，即每时钟周期执行的指令数作为度量会更方便。例如，一个4GHz四路多发射微处理器能以每秒160亿指令的峰值速率执行，其最好情况下的CPI达到0.25，IPC达到4。假设是五级流水线，这个处理器任何时刻都可能有20条指令在同时执行。现在的高端微处理器尝试在每个时钟周期发射3~6条指令，甚至中端设计的目标都是峰值IPC为2。然而，一般来说存在很多约束，例如哪些类型的指令可以同时执行，当相关性增加时会发生什么新的问题等。

实现一个多发射处理器主要有两种方式，其区别是将主要工作分给编译器来做还是硬件来做。由于不同的实现方式将导致某些决策是静态进行的（在编译时）还是动态进行的（在执行时），所以这两种方式有时也被称为**静态多发射**（static multiple issue）和**动态多发射**（dynamic multiple issue）。正如我们即将见到的，两种方式还有其他别名。

多发射流水线必须处理以下两个问题：

1) 往**发射槽**（issue slot）中发射多条指令：处理器如何确定在给定的时钟周期发射多少条指令以及发射何种指令呢？在大多数静态发射处理器中，这个过程至少有很大一部分是由编译器处理的。而在动态发射处理器中，这个问题一般是由处理器在运行时处理的，尽管编译器也会尽其所能通过调整指令顺序加以协助。

2) 处理数据冒险和控制冒险：在静态发射处理器中，部分甚至全部的数据冒险和控制冒险是由编译器静态处理的。相反，绝大多数的动态发射处理器通过硬件技术在执行时至少消除某些类别的冒险。

尽管这里我们把它们看成两种不同的方法，实际上这两种方法经常借用对方的技术，没有哪一种方法可以称得上是完全独立的。

- ④ 静态多发射：实现多发射处理器的一种方法，其中决策是在执行前的编译阶段做出的。
- ⑤ 动态多发射：实现多发射处理器的一种方法，其中决策是由处理器在执行阶段做出的。
- ⑥ 发射槽：在给定时钟周期内能够发射指令的位置，可以类比于短跑比赛中的起点位置。

#### 4.10.1 推测的概念

推测是一种寻找和挖掘更大 ILP 的最重要的方法。推测 (speculation) 是一种为了使依赖于被推测指令的其他指令可以执行，而允许编译器或处理器“猜测”指令结果的方法。例如，我们可以推测分支指令的结果，这样分支后的其他指令就可以提前执行了。另一个例子是假设 load 指令前有一条 store 指令，我们可以推测它们不对同一存储器地址进行访问，这样就可以把 load 指令提到 store 指令前执行。推测的问题在于可能会猜错。所以，任何推测技术必须包含一种机制，它能检查推测的正确性并在推测错误时能取消根据原推测结果执行指令的影响。实现这种回卷能力增加了额外的复杂性。

333

- 推测：一种编译器或处理器推测指令结果以消除执行其他指令对该结果依赖的方法。

推测可以由编译器或硬件来完成。例如，编译器可以利用推测对指令进行重排序，将一条指令移过分支，也可将 load 指令与 store 指令交换。使用本节后面讨论的技术，处理器硬件可以在运行时实现同样的变换。

推测错误时的恢复机制对软硬件是非常不同的。对软件来说，编译器经常插入额外的指令检查推测的正确性并提供专门的修复例程供推测错误时使用。对硬件来说，处理器经常缓存推测的结果直至推测的结果得到确认。如果推测是正确的，缓存的结果写回寄存器堆和存储器。如果推测是错误的，硬件将清除缓存并重新执行正确的指令序列。

推测还可能导致另一个问题：对某些指令的推测会导致原本不存在的异常发生。例如，假设推测执行一条装载指令，但是在推测错误的情况下，该指令所使用的地址是非法的。结果，一个本不应该发生的异常发生了。这个问题之所以复杂是因为，如果这条装载指令本来不是推测执行，那么该异常必然发生。在基于编译器的推测中，这类问题的处理方法是加入额外的推测支持，使得这样的异常暂时被忽略，直至可以确定异常会发生为止。在基于硬件的推测中，异常被简单地缓存起来，直到导致异常的指令确定会执行。在异常真正发生时，就会执行正常的异常处理程序。

推测在设计正确时能改善性能，而不慎使用可能降低性能，所以需要做大量的工作来决定何时采用推测更为合适。在本节的后半部分，我们将介绍静态和动态的推测技术。

#### 4.10.2 静态多发射处理器

所有的静态多发射处理器都使用编译器来帮助封装多条指令并处理冒险。在一个静态发射处理器中，可以在给定时钟周期内发射多条指令，也称为发射包 (issue packet)。发射包可被视为一条完成多个操作的长指令。这种看法不仅是为了类比。因为静态多发射处理器一般对一个时钟周期内能发射的多条指令有所限制，因此把发射包看成允许同时进行很多操作的一条指令是可行的。这种观点引出了这种方法的最初名字：超长指令字 (Very Long Instruction Word, VLIW)。

334

- 发射包：在一个时钟周期内发射的多条指令的集合。这个包可以由编译器静态生成，也可以由处理器动态生成。
- 超长指令字：一类可以同时启动多个操作的指令集，其中操作在单个指令中相互独立，并且一般都有独立的操作码字段。

绝大多数静态多发射处理器也依赖编译器处理数据冒险和控制冒险。编译器的任务可能包括静态分支预测和代码调度，以减少冒险或阻止所有的冒险。在描述更先进的处理器中所采用的技术之前，先来看一个简单的静态多发射 MIPS 处理器的例子。

### 一个例子：MIPS 指令集的静态多发射

为了感受一下静态多发射，我们考查一个简单的双发射 MIPS 处理器，其中一条指令可以是整型 ALU 操作或分支，另一条指令可以是装载指令或存储指令。在某些嵌入式 MIPS 处理器中就是这么设计的。每个时钟周期发射两条指令意味着需要取回和译码 64 位的指令。在许多静态多发射处理器中，甚至是所有的 VLIW 处理器中，严格限制了可同时发射指令的所处位置以简化译码和发射过程。因此，我们要求两条指令成对地放在一个 64 位对齐的内存区域中，并且 ALU 指令或分支指令必须放在前面。此外，如果找不到另一条与之可以同时发射的指令，就用 nop 指令代替它。这样，指令总是可以成对发射，当然其中可能有一条 nop 指令。图 4-68 给出了指令成对在流水线中运行的情况。

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

图 4-68 静态双发射流水线。ALU 指令与数据传输指令同时发射。这里我们假设使用与单发射相同的五级流水线。尽管这并非严格的要求，但这样做确实会带来一些好处。特别是使寄存器堆的写操作位于流水线的最后，可以简化异常处理和降低实现精确异常的难度，这些问题在多发射处理器中将变得更加难以处理

静态多发射处理器之间的不同在于处理潜在的数据冒险和控制冒险的方式。在有的设计中，编译器负责避免所有的冒险，它通过调度指令和插入 no-ops 等方法使得代码在执行时完全不需要冒险检测和硬件产生阻塞。在另外一些设计中，硬件检测数据冒险并在两个发射包间产生阻塞，而编译器只负责避免一个指令对中两条指令之间的依赖。尽管如此，冒险仍会使包含依赖指令的整个发射包阻塞。不管是软件必须处理所有的冒险还是只负责减少不同发射包之间的冒险，都会增加一次完成多个操作的长指令的情况。在这个例子中，我们假定使用第二种方法。

为了并行发射一个 ALU 操作和数据传输操作，首先需要增加一些硬件：除了通常的冒险检测和阻塞逻辑之外，还有寄存器堆的额外端口（见图 4-69）。在一个时钟周期内，我们需要为 ALU 操作读两个以上寄存器，为存储操作读两个以上寄存器，为 ALU 操作写一个端口，为装载操作写一个端口。因为 ALU 要用来进行 ALU 操作，所以需要一个额外的加法器来为数据传输计算有效地址。如果没有这些额外的硬件资源，我们的双发射流水线将不可避免地遭遇结构冒险。

显然，双发射处理器最多能将性能提高两倍。事实上，为了达到这一点，需要双发射流水线中重叠的指令数翻倍。额外的重叠使数据冒险和控制冒险带来的相对性能损失也增加了。例如，在我们简单的五级流水线中，装载指令有一个时钟周期的使用延迟（use latency），以防止一条指令无阻塞地使用其结果。在一个双发射五级流水线中，装载指令的结果不能在下个时钟周期使用。这意味着下两条指令不能无阻塞地使用装载的结果。而且，原本在简单的五级流水线中没有使用延迟的 ALU 指令现在有一个周期的使用延迟，因为其结果不能在与其配对的存储指令或装载指令中使用。为了有效地挖掘多发射处理器中潜在的并行性，需要使用更高级的编译器或硬件调度技术，其中静态多发射对编译器有更高的要求。

- 使用延迟：在装载指令与可以无阻塞使用其结果的指令间相隔的时钟周期数。

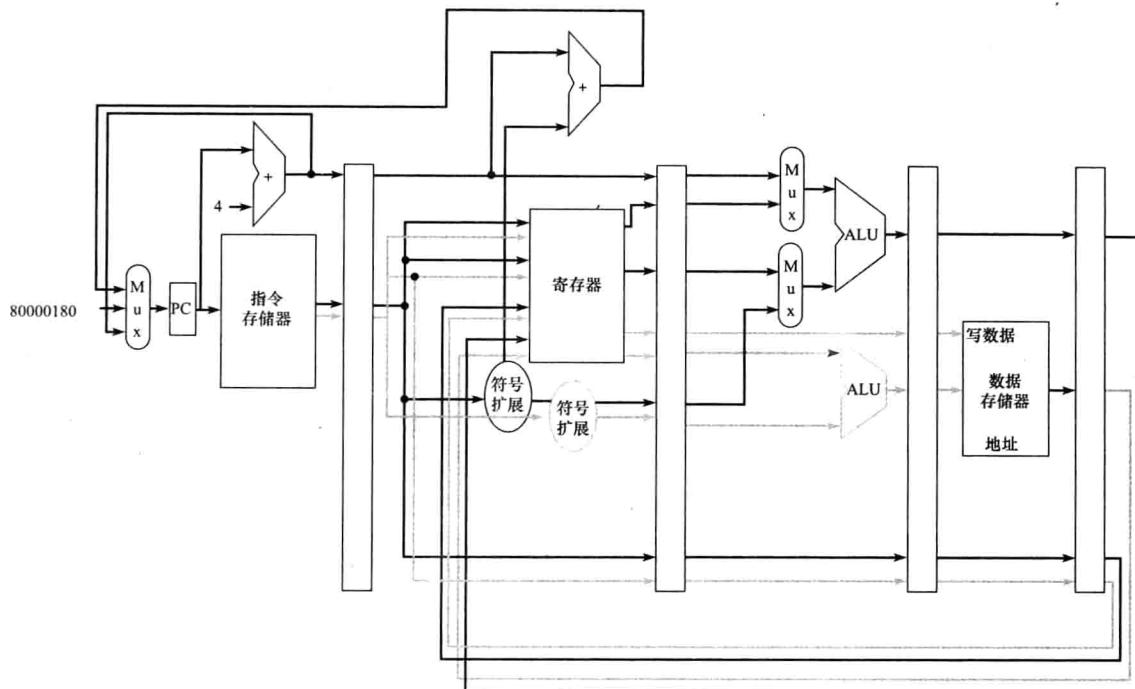


图 4-69 一个静态双发射的数据通路。双发射所需的额外硬件用灰色线显示，主要包括：来自指令存储器的额外 32 位输出，寄存器堆多出的两个读端口和一个写端口，还有一个额外的 ALU。这里假设下面那个 ALU 处理数据传输时的地址计算，而上面那个 ALU 处理所有的其他操作

### 01 例题·简单的多发射代码调度

在一个 MIPS 静态双发射流水线中，下面这个循环将如何调度？

```
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
```

对该指令序列进行重排序，以尽可能地避免流水线阻塞。假设分支是可预测的，即控制冒险由硬件处理。

### 01 答案

前三条指令间存在数据相关性，最后两条指令间也是如此。图 4-70 给出了这些指令的最佳调度方式。注意，只有一对指令同时使用了两个发射槽。每次循环需要花费 4 个时钟周期。在 4 个时钟周期内执行 5 条指令，与最好情况下 0.5 的 CPI 和 2.0 的 IPC 相比，CPI 只有 0.8 而 IPC 只有 1.25。注意，在计算 CPI 或 IPC 时，我们没有把执行的 nop 指令也算到有效的指令中去。如果算进去能提高 CPI，但并不能提高真实的性能。

	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 \$t0,4(\$s1)	4

图 4-70 在双发射 MIPS 流水线中调度的代码。空白槽中是 nop 指令

有一种重要的从循环中获得更多性能的编译技术叫循环展开 (loop unrolling)。循环展开时循环体会被复制多份。循环展开后，通过重叠不同循环体中的指令可以获得更高的指令级并行 (ILP)。

- 循环展开：一种从存取数组的循环中获取更多性能的技术，其中循环体会被复制多份并且不同循环体中的指令可能会调度到一起。

### 01 例题·多发射流水线中的循环展开

试着对上面的例子进行循环展开和调度，看其效果如何。为简单起见，假设循环起始地址与 32 位内存边界对齐。

#### 01 答案

为了无延迟地调度循环，我们需要把循环体复制 4 份。在展开和消除不必要的循环开销指令后，将得到 4 个备份，每份包含 lw 指令、add 指令和 sw 指令，还有 addi 指令和 bne 指令各一条。图 4-71 给出了展开并调度后的代码。

	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 4-71 对图 4-70 中的代码进行循环展开并在一个静态双发射 MIPS 流水线中调度后的代码。空白槽中是 nop 指令。因为循环中的第一条指令将 \$s1 寄存器中的值减 16，而装载指令的地址又是 \$s1 寄存器中的原值，所以这个地址依次减 4、减 8、减 12

在循环展开过程中，编译器引入了几个临时寄存器（\$t1、\$t2、\$t3）。这个过程被称为寄存器重命名 (register renaming)，目的是消除一些虚假的数据依赖，这些虚假的数据依赖可能导致潜在的冒险或妨碍编译器灵活地调度代码。考虑一下如果只使用 \$t0 展开的代码是什么样子的。在指令 sw t0, 4 (\$s1) 后面会有多对 lw \$t0, 0 (\$\$s1) 指令和 addu \$t0, \$t0, \$s2 指令。如果不管 \$t0 寄存器的使用，那么这些指令应该是完全无关的，即一个指令序列与下一个指令序列之间没有任何数据流动。这就是反相关 (antidependence)，也被称为名字相关 (name dependence)，即只是因为重用寄存器名引起的相关，而并非一个真实的数据相关。

- 寄存器重命名：由编译器或硬件对寄存器进行重命名以消除反相关。
- 反相关：也被称为名字相关，因为寄存器名的重用导致的相关，并非由两条指令中使用同一个值导致的真正相关。

在循环展开中的寄存器，重命名使得编译器移动不相关的那个指令，从而得到更好的代码。重命名的过程消除了名字相关，同时保留了真正的相关。

注意，既然循环中 14 条指令中的 12 条以指令对的形式被执行，4 次循环将花费 8 个时钟周期，即每次循环 2 个时钟周期，CPI 为  $8/14 = 0.57$ 。双发射加上循环展开与调度使得性能提高了将近两倍，这一方面是因为减少了循环控制指令，另一方面是因为双发射的缘故。这种性能提高的代价是使用了 4 个而非一个临时寄存器，同时代码长度也增长了很多。

#### 4.10.3 动态多发射处理器

动态多发射处理器通常也称为超标量处理器，或简称超标量（superscalar）。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0条、1条，还是多条指令。显然，在这种处理器上要达到较好的性能仍然依赖编译器对指令的调度，通过错过依赖关系以达到较高的指令发射速率。尽管使用了编译器进行调度，这种简单的超标量处理器与VLIW处理器仍有显著不同。在超标量处理器中，不管代码是否经过调度，都是由硬件来保证执行的正确性。并且，编译得到的代码应当始终正确执行，而与指令发射速率和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，当把代码从一个处理器移到另一个处理器上运行时，可能需要重新编译。在其他一些静态发射处理器上，代码可以在不同的处理器上实现正确运行，但效果可能很差以至于不得不进行更加有效的编译。

许多超标量处理器扩展了基本的动态发射决策，将动态流水线调度（dynamic pipeline scheduling）也包含在内。动态流水线调度选择某个时钟周期内将执行的指令，约束条件是尽量不产生冒险和阻塞。让我们从一个简单数据冒险的例子出发来进行说明。考虑下面的指令序列：

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

即使sub指令准备好执行，它也必须等待lw和addu指令先结束才行。如果内存很慢（第5章解释了有时访存操作会很慢的原因，即高速缓存缺失），sub指令可能会等待很多个时钟周期。动态流水线调度可以部分或者完全避免这种冒险。

- 超标量：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超过一条。
- 动态流水线调度：对指令进行重排序以避免阻塞的硬件支持。

#### 动态流水线调度

动态流水线调度选择下一条要执行的指令，可能的话会重排指令以避免阻塞。在这种处理器中，流水线被划分为3个主要单元：取指与发射单元、多个功能单元（在2013年的高端处理器中有一打或更多）和一个提交单元（commit unit）。图4-72描述了这个模型。第一个单元取指并译码，然后将每条指令发送到相应功能单元执行。每个功能单元都有自己的缓冲区（称为保留站（reservation station）），用来保存操作数和操作（下一节我们将讨论许多最新处理器中使用的保留站的替代选择）。当缓冲区中包含了所有的操作数，并且功能单元就绪时，结果就被计算出来。结果得到后，被发送到等待该结果的保留站和提交单元。提交单元缓存这个结果，在确定安全时，再将这个结果写回寄存器堆或存储器（对存储指令）。提交单元中的缓冲区通常称为重排序缓冲区（reorder buffer），它也可以用来提供操作数，其工作方式类似于静态调度流水线中的旁路逻辑。一旦结果写回寄存器堆，其可以从寄存器堆中直接被取出，和一般的流水线完全一样。

- 提交单元：位于动态流水线和乱序流水线中的一个单元，用以决定何时可以安全地将操作结果送至程序员可见的寄存器和存储器。
- 保留站：功能单元的缓冲区，用来保存操作数和操作。
- 重排序缓冲区：动态调度处理器中用于暂时保存执行结果的缓冲区，等到安全时才将其中的结果写回寄存器或存储器。

将操作数缓存在保留站中并将结果放在重排序缓冲区中，实际上提供了一种寄存器重命名

机制，类似于前面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，考虑如下几个步骤：

340 1) 发射指令时，它先被复制到合适功能单元的保留站。如果它的操作数在寄存器堆中或重排序缓冲区中可用，那么操作数立即被复制到保留站中。除非所有的操作数和执行单元可用，否则指令一直缓存在保留站中。如果指令已经被发射，那么其操作数对应的寄存器堆副本不再需要，如果此时发生了对该寄存器的写请求，其值可以被覆盖。

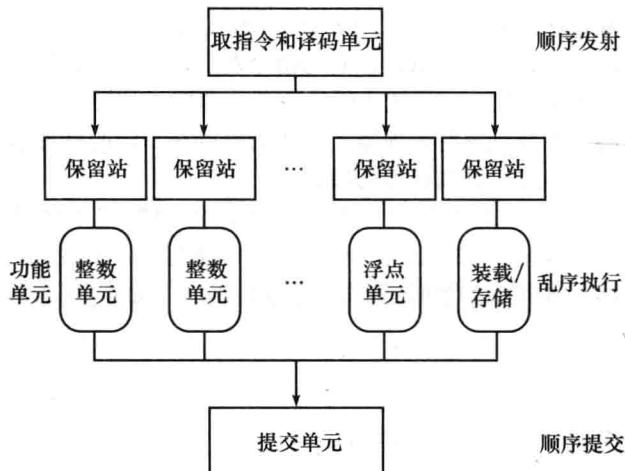


图 4-72 动态调度流水线中的三个主要单元。最后一个更新状态的步骤也被称为提交

2) 如果操作数不在寄存器堆或重排序缓冲区中，那么它应该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。当该单元计算出结果时，这个结果将直接从功能单元复制到保留站，而跳过寄存器堆。

上面这两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，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。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。这种执行方式被称为**乱序执行**（out-of-order execution），因为执行指令的顺序可以与取指的顺序不同。

为了使程序表现得像是在一条简单的顺序流水线上执行，取指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。而提交单元也必须按照程序顺序将结果写回寄存器堆和存储器。这种保守的方案称为**顺序提交**（in-order commit）。所以当异常发生时，处理器可以找到最后执行的那条指令，而只有这条导致异常的指令之前的指令才能对寄存器状态进行修改。虽然处理器的前端（取指和发射）和后端（提交）按照顺序操作指令，各功能单元可以在获得所需数据的条件下随时开始执行过程。目前所有的动态调度流水线都采用顺序提交。

- **乱序执行**：流水线执行的一种情况，即执行的指令被阻塞时不会导致后面的指令等待。
- **顺序提交**：流水线执行的结果以取指顺序写回程序员可见寄存器的一种提交方式。

动态调度经常与基于硬件的推测机制相结合，特别是对分支指令的推测。通过对分支指令的方向进行推测，动态调度处理器可以在推测方向上进行取指和执行。由于指令是顺序提交的，我们可以在分支指令及所有推测执行的指令提交前知道推测是否正确。一个推测执行的动态调度流水线同样可以对装载指令的目的地址进行推测、对存取指令进行重排序和利用提交单元避免错误的推测。在下一节中我们将讨论 Intel Core i7 处理器的动态调度流水线设计与推测机制。

**01 理解程序性能** 既然编译器可以根据数据依赖关系调度代码，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。

第一，并不是所有的阻塞都是可以事先知道的。尤其是 cache 缺失（参见第 5 章）会导致不可预测的阻塞。动态调度使得处理器在一些指令阻塞时，可以调度其他指令继续执行，以掩盖阻塞。

第二，如果处理器采用动态分支预测推测分支的结果，那么由于这些信息依赖于预测和分支指令的真实执行情况，编译器无法得知指令的精确顺序。采用动态推测而不使用动态调度，会极大地限制可开发的指令级并行度（ILP）。

第三，由于流水线延迟和发射宽度根据处理器的具体实现的不同有很大的差别，所以最佳的编译代码顺序也并不固定。例如，调度一个相互依赖的指令序列的具体方式与发射宽度和延迟存在着密切关系。流水线的结构同样会影响循环展开的尝试，才能避免可能的阻塞。它还会影响基于编译器的寄存器重命名的过程。动态调度使得硬件将这些细节隐藏起来。因此，用户和软件发行商就不用针对同一指令集的不同处理器发行相应的软件了。同样，以前的代码也能从更新的处理器上获得好处而不用重新编译。

**01 重点** 流水线和多发射都提高了指令的吞吐率并致力于开发指令级并行。然而，由于处理器有时必须等待依赖关系明确后才能继续工作，所以程序中的数据相关和控制相关往往限制了可达性能的上限。基于软件的指令级并行开发主要依赖于编译器来寻找依赖关系，并尽量减少这些依赖关系可能造成的不良后果。基于硬件的指令级并行开发主要依赖于流水线和多发射机制。推测执行可以由硬件或编译器完成，它可以增加指令行并行。但是使用时必须小心，因为错误的推测可能会降低性能。

**01 硬件/软件接口** 现代的高性能微处理器可以在一个时钟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。例如，尽管我们有一个单时钟周期可以发射 4~6 条指令的处理器，只有很少的应用程序能保持每周期发射两条以上指令。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解决的相关性，这就限制了指令间的并行度，因此也就限制了发射速率。虽然对于真正的数据相关而言没有什么好的解决方法，但是一般情况下硬件或编译器对于相关是否确实存在都不知道，因而也就只能保守地假设相关存在了。例如，使用了指令的程序由于有更多的内存别名问题，往往存在隐式相关的可能性更大。反之，对于数组访问而言，由于有更大的规则性，使得编译器可以推测出没有相关存在的情况。同样，不能在编译期或运行期被准确预测的分支同样会限制指令级并行的开发。一般来说，指令级并行总是有开发的空间的，但是因为并行度较为分散（有时可能存在于上千条指令之间），编译器和硬件往往会显得力不从心。

其次，存储器层次（这是第 5 章的主题）中的缺失同样会使流水线难以满负荷运转。尽管一些访存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。

#### 4.10.4 能耗效率与高级流水线

通过动态多发射和推测执行开发指令级并行的负面问题是功耗效率。每项发明都成功地将更多的晶体管转化为性能，但是这种转化往往极其缺乏效率。因为功耗墙的原因，最新的处理器是单片多核式的，而非其前辈的深流水线或贪婪式推测。

尽管简单的处理器没有复杂的处理器那么快，但是在同样的能耗下却能得到更高的性能。所以当设计的约束更多来自能耗而非晶体管数量时，简单的处理器能在单芯片上获得更高的性能。

图 4-73 给出了一些处理器的流水线级数、发射宽度、推测级别、时钟频率、每芯片的核数和功耗等。注意从单核发展到多核时流水线级数和功耗的减少。

**343** **精解** 提交单元负责寄存器堆和存储器的更新。一些动态调度处理器在执行过程中即时更新寄存器堆，而使用额外的寄存器来实现重命名功能并保存之前寄存器的副本直到更新该寄存器的指令不再是靠推测得出的。其他处理器通常把结果缓存在重排序缓冲器中，由提交单元在随后更新寄存器堆。在指令提交之前，写内存的数据必须先缓存在存储缓冲器（见第 5 章）或重排序缓冲器中。提交单元允许缓冲器在地址和数据有效时并且 store 操作不依赖于预测的分支时写内存。

**精解** 非阻塞 cache (nonblocking cache) 在 cache 缺失（参见第 5 章）时能够继续提供 cache 访问服务。为了使指令在 cache 缺失时能继续执行，乱序执行处理器需要非阻塞 cache 的支持。

处理器	年份	时钟频率	流水线级数	发射宽度	乱序/推测	核数目/片	功耗
Intel 486	1989	25MHz	5	1	No	1	5 W
Intel Pentium	1993	66MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2 000MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3 600MHz	31	3	Yes	1	103 W
Intel Core	2006	2 930MHz	14	4	Yes	2	75 W
Intel Core i5 Nehalem	2010	3 300MHz	14	4	Yes	1	87 W
Intel Core i5 Ivy Bridge	2012	3 400MHz	14	4	Yes	8	77 W

图 4-73 Intel 和微处理器的流水线复杂度、核数和功耗的发展情况。其中，Pentium 4 的流水线级数没有包括提交级，如果加上提交级，Pentium 4 的流水线级数会更深一些

### **01 小测验**

说明下列开发指令级并行度的技术或单元主要是基于硬件还是基于软件。对某些项来说两者都有可能。

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. 重排序缓冲区
9. 寄存器重命名

## 4.11 实例：ARM Cortex-A8 和 Intel Core i7 流水线

**344** 图 4-74 给出了本节将要考察的两个处理，它们的目标是后 PC 时代的标志性产品。

处理器	ARM A8	Intel Core i7 920
市场	个人移动设备	服务器, 云计算
功耗	2W	130W
时钟频率	1GHz	2.66GHz
核/芯片	1	4
浮点	无	有
多发射	动态	动态
峰值指令数/周期	2	4
流水线级数	14	14
流水线调度	静态顺序	动态乱序猜测执行
分支预测	2 级	2 级
1 级 cache/核	32KiB 指令, 32KiB 数据	32KiB 指令, 32KiB 数据
2 级 cache/核	128 ~ 1 024KiB	256KiB
3 级 cache (共享)		2 ~ 8MiB

图 4-74 ARM Cortex-A8 和 Intel Core i7 920 处理器的特点

#### 4.11.1 ARM Cortex-A8

ARM Cortex-A8 处理器主频为 1GHz，具有 14 级流水线。它采用动态多发射技术，每个时钟周期可以发射两条指令，其流水线为静态顺序流水线，指令发射、执行和提交顺序执行。流水线包含取指令、指令译码和执行三个阶段。图 4-75 给出了流水线的整体情况。

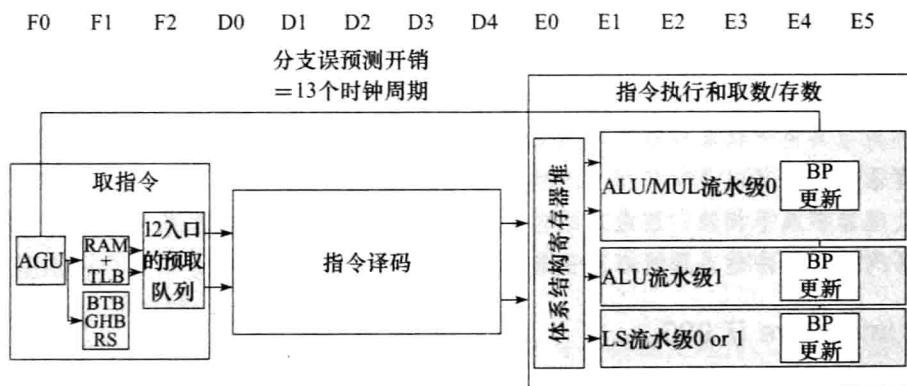


图 4-75 A8 的流水线。前端的三个流水级为 12 入口的指令预取缓存取入指令。地址产生单元 (AGU) 使用一个分支目标缓存 (BTB)、全局历史缓存 (GHB) 和返回栈 (RS) 进行分支预测以保持预取队列满状态。指令译码有 5 个流水级，指令执行有 6 个流水级

前端的三个流水级每次取入两条指令，从而使得能够保存放 12 条指令的预取缓存维持满状态。它使用了一个两级的分支预测器，其中包含了一个 512 入口的分支目标缓存、4096 入口的全局历史缓存和一个用于预测未来返回操作的 8 入口的返回栈。当分支预测错误时清空流水线，导致 13 个时钟周期的误预测开销。

译码阶段的 5 个流水级确定一对指令间是否存在可导致顺序执行的相关，并且确定将指令送往哪条执行流水线。

指令执行阶段的 6 个流水级提供了一条用于 load 和 store 指令的流水线、两条算术操作流水线，其中只有第一对可以处理乘法。指令对中的任何一条指令都可以发射到 load-store 流水线。三条流水线间在执行级具有全旁路机制。

345

图 4-76 给出了 A8 在使用从 SPEC2000 基准程序中衍生出来的一个小版本程序集的 CPI。虽然理想的 CPI 是 0.5，但是这里最好情形只有 1.4，一般情形是 2.0，而最差情形是 5.2。在一般情形下，80% 的流水线阻塞是源于冒险，20% 源于存储层次。流水线阻塞是由分支误预测、结构冒险和指令对间的数据相关造成的。由于 A8 是静态流水线，使用编译器来尽可能避免结构冒险和数据相关。

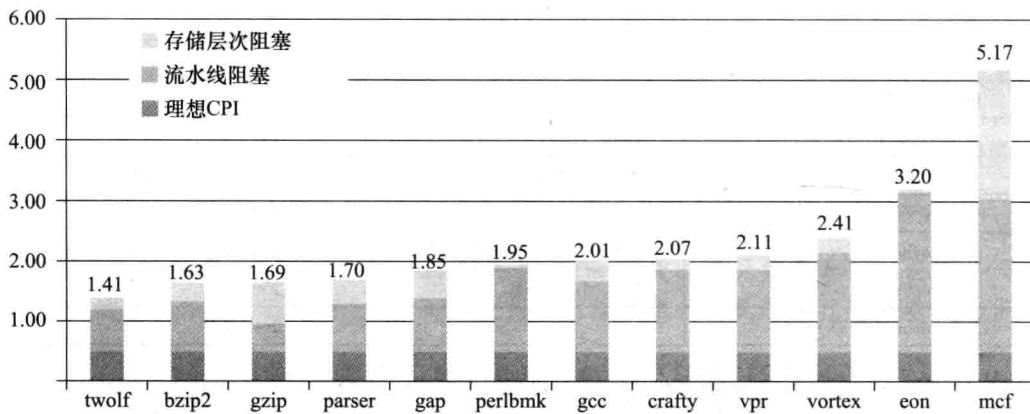


图 4-76 Minnespec 基准测试程序在 ARM Cortex-A8 上的 CPI，Minnespec 是 SPEC2000 的一个简化版本。这些基准程序使用更小的输入集使运行时间减少几个数量级。小的输入集极大地低估了存储层次对 CPI 的影响（见第 5 章）

**01 精解** Cortex-A8 是一个支持 ARMv7 指令集的可配置内核。它以 IP (Intellectual Property, 知识产权) 核方式交付使用。IP 核是嵌入式、个人移动设备和相关市场中用于交付的主要形式；数十亿的 ARM 和 MIPS 处理器都是由这种 IP 核产生的。

注意 IP 核与 Intel i7 多核计算机中的核不同。一个 IP 核（自身可能就是多核）的设计目标是与其他逻辑集成在一起（因此是一个芯片的“核心”）形成一个对某种应用优化的处理器，这里其他逻辑包括专用处理器（例如视频编解码器）、I/O 接口和存储器接口。虽然处理器核几乎相同，但最终的芯片可能有许多不同。一个参数就是 L2 cache 的容量，它在不同的应用中的差别可以高达 8 倍。

#### 4.11.2 Intel Core i7 920

x86 采用了复杂的流水线技术，在其 14 级流水线中综合使用了动态多发射、乱序执行和推测执行的动态流水线调度技术。正如第 2 章中提到的，处理器依然面临着实现复杂 x86 指令集的挑战。Intel 取入 x86 指令，将其翻译为类 MIPS 指令，Intel 称之为“微操作”。微操作由复杂的基于推测执行的动态调度流水线执行，该流水线每个时钟周期最多可执行 6 个微操作。本节集中讨论微操作流水线。

当我们考虑复杂的动态调度处理器的设计时，功能单元、cache 和寄存器堆、指令发射和整个流水线控制的设计将混在一起，使得把数据通路和流水线分开变得很困难。因此，许多工程师和研究人员使用术语 **微体系结构** (microarchitecture) 来描述处理器内部体系结构的细节。

② **微体系结构：**处理器的组织，包括主要的功能单元及它们的互连关系与流水线控制。

Intel Core i7 使用重排序缓冲区和寄存器重命名技术来解决反相关和推测错误。寄存器重命名技术显式地将处理器中的 **体系结构寄存器** (architectural register)（在 64 位版本的 x86 体系

346

结构中是 16 个)重命名为一组更大的物理寄存器集合。Core i7 使用寄存器重命名技术来消除反相关。寄存器重命名需要处理器维护体系结构寄存器和物理寄存器之间的映射关系,要能指出哪个物理寄存器才是某个体系结构寄存器的最新备份。通过跟踪已经发生的重命名,寄存器重命名提供了另一种推测错误时的恢复方法:简单地撤销所有第一条推测错误指令后建立的所有映射。这会使处理器的状态返回到最后一条正确执行的指令处,并保持结构寄存器与物理寄存器之间的正确映射关系。

- ② 体系结构寄存器:处理器中的可见寄存器。如在 MIPS 中,有 32 个整数寄存器和 16 个浮点寄存器是可见的。

图 4-77 给出了 Core i7 的整体组合和流水线结构。下面是一条 x86 指令的执行需要经历的 8 个阶段。

1) 取指令——处理器使用一个多级分支目标缓冲器在速度和预测准确性方面做平衡。另外还有一个返回地址栈用于加速函数返回。误预测将导致 15 个周期的开销。取指部件使用预测地址从指令 cache 中取入 16 字节。

2) 这 16 字节放入预译码指令缓冲器——预译码阶段将这 16 字节转换为独立的 x86 指令。因为 x86 指令长度可以是 1~15 字节不等,预译码操作必须扫描多个字节以确定指令长度,所以预译码操作非常复杂。每条 x8 指令放入一个 18 入口的指令队列。

3) 微操作译码——每条 x86 指令被翻译为微操作。有三个译码器将 x86 指令直接翻译为一个微操作。而对于具有复杂语法功能的 x86 指令,则使用一个微代码引擎产生一个微操作序列;它可以在每个周期生成 4 个微操作直到必需的微操作序列生成为止。这些微操作按照 x86 指令顺序放入 28 入口的微操作缓冲器。

4) 微操作缓冲器执行循环流检测——如果有一个小的指令序列(少于 28 条指令或长度小于 256 字节)包含一个循环,循环流检测器将识别该循环,并直接从缓冲器中发射微操作,从而减少了指令预取和指令译码。

5) 执行基本指令发射——在将微操作发射到保留站之前,在寄存器表中查找寄存器位置、对寄存器进行重命名、分配重排序缓冲器入口、从寄存器或重排序缓冲中取结果。

6) i7 使用一个被 6 个功能单元共享的 36 入口的集中式保留站。在每个周期内最多可以向功能单元分派 6 个微操作。

7) 各个功能单元执行微操作,执行结果不但送往寄存器提交部件,在已知指令将不再预测的情况下更新寄存器状态,还可送往任何一个等待的保留站。重排序缓冲中与指令对应的入口标记为完成。

8) 当前面的一条或多条指令已经被标记为完成,则执行寄存器提交部件中未决的写操作,指令从重排序缓冲器中移走。

**01 精解** 第二步和第四步中的硬件能够将操作进行合并,从而减少需要执行的微操作的数量。第二步中的宏操作合并进行 x86 指令的合并,例如将比较后面紧跟一个分支合并成一个操作。第四步中的微操作合并将 load/ALU 操作和 ALU/store 之类的微操作对进行合并,并将它们发射到一个保留站中(在这里它们依旧可以独立发射),从而提高了缓冲器的利用率。在研究微操作合并和宏操作合并在内的 Intel 核体系结构时,Bird 等 [2007] 发现微操作合并对性能影响很小,而宏操作合并似乎对整数性能有适度的正面影响,对浮点性能影响很小。

347

348

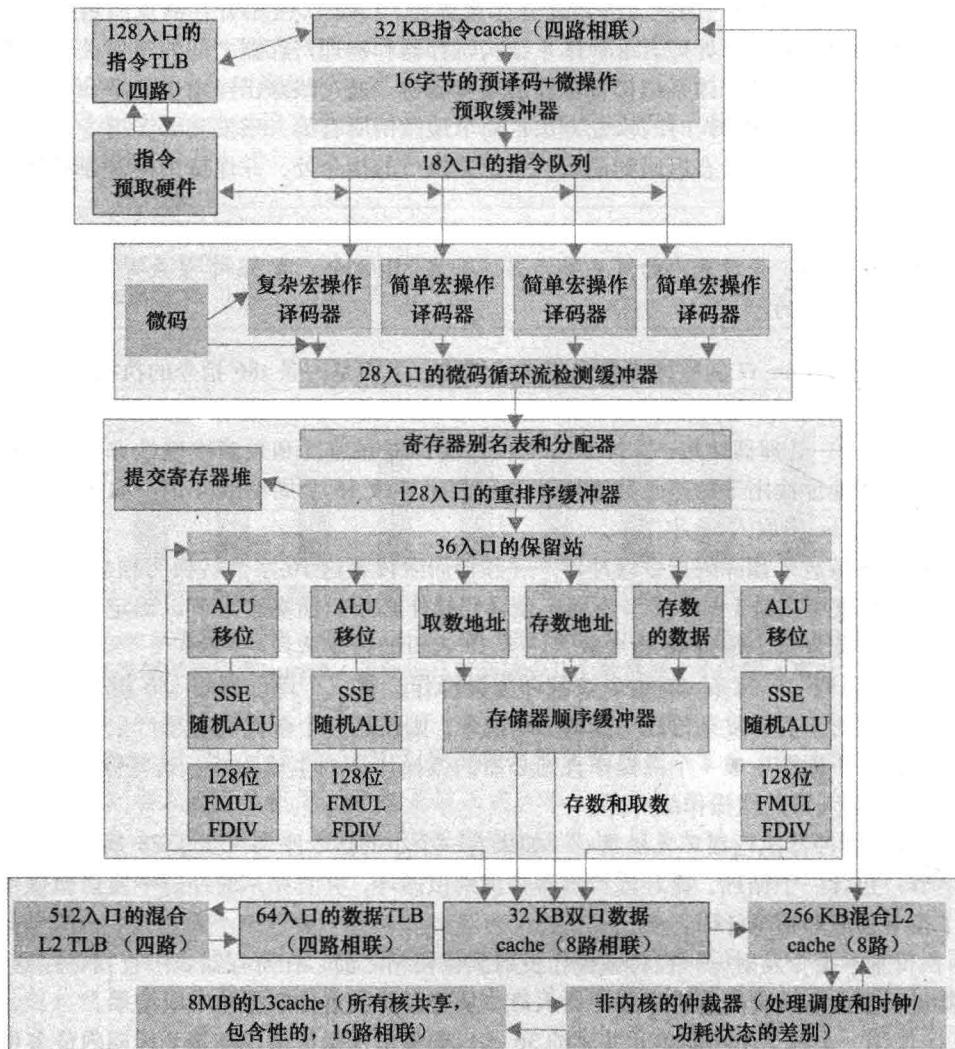


图 4-77 包含存储部件的 Core i7 流水线结构。流水线总深度为 14 级，误预测的代价是 17 个时钟周期。该设计可以缓存 48 个 load 操作和 32 个 store 操作。6 个相关部件在每个时钟周期可以开始执行一个 RISC 操作

#### 4.11.3 Intel Core i7 920 的性能

图 4-78 给出了每个 SPEC2006 基准程序在 Intel Core i7 上的 CPI。虽然理想的 CPI 是 0.25，但是最佳情况是 0.44，平均情况是 0.79，而最差情况只有 2.67。

在动态乱序执行流水线中，很难区分流水线阻塞和访存阻塞，但是我们却能有效地区分分支预测和推测执行。图 4-79 显示了分支误预测的比例和与所有的微操作分派相关的工作中未提交（即它们的结果被取消）的比例（使用分派进入流水线的微操作数量衡量）。误预测的最小值、平均值和最大值分别是 0%、2% 和 10%。对于浪费的工作，分别是 1%、18% 和 39%。

浪费的工作在有些情况下与误预测率相匹配，例如 gobmk 和 astar。在许多情况下，例如 mcf，浪费的工作的比例相对要大于误预测比例。这种差别可能来源于存储器行为。只要在访存时保留站有足够的空间，mcf 具有很高的数据 cache 缺失率，在误预测时刻分派许多条指令。当最

终确定推测执行许多指令中的一个分支是误预测时，与所有这些指令相关的微操作将被清除。

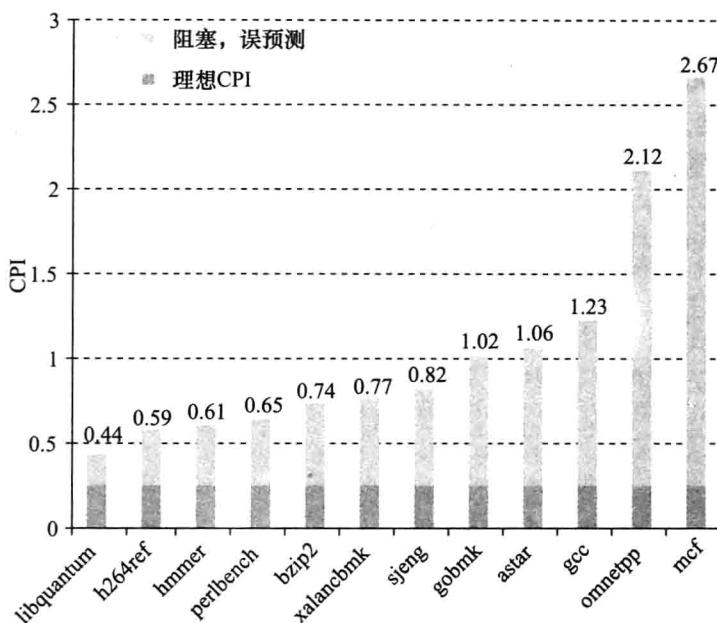


图 4-78 在 Intel Core i7 920 上运行 SPEC2006 整数基准程序的 CPI

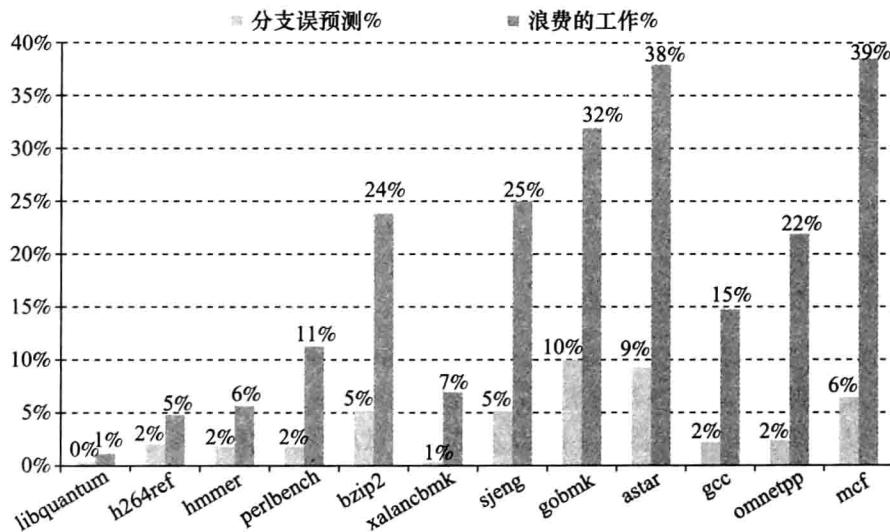


图 4-79 在 Intel Core i7 920 上运行 SPEC2006 整数基准程序时，分支误预测的比例和无效的推测所浪费的工作的比例

**Q1 理解程序性能** Intel Core i7 同时使用一个 14 级的流水线和激进的多发射来获取高性能。在保持背对背操作低延迟的同时，也消除了数据依赖的影响。对运行在这个处理器上的程序而言，最严重的潜在性能瓶颈在哪里呢？下面的列表包含了一些潜在的性能问题，最后三个问题在任何高性能流水线处理器中都会以某种形式出现。

- 使用了不能映射成几条简单微操作的 x86 指令。
- 难于预测的分支，会导致预测错误时的阻塞和推测失败时的重启。
- 长依赖——典型情况是执行时间很长的指令或存储器层次——这会导致阻塞。
- 存储器访问延迟增大（见第 5 章）将导致的处理器阻塞。

## 4.12 运行更快：指令级并行和矩阵乘法

回到第 3 章的 DGEMM 的例子，我们可以看到通过循环展开使得多发射乱序执行处理器有更多的指令用于调度，从而对指令级并行有影响。图 4-80 给出了图 3-23 中程序的循环展开版本，图 3-23 包含了使用 C 循环体的特点生成的 AVX 指令。

```

1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm ( int n, double* A, double* B, double* C )
5 {
6     for ( int i = 0; i < n; i+=UNROLL*4 )
7         for ( int j = 0; j < n; j++ ) {
8             __m256d c[4];
9             for ( int x = 0; x < UNROLL; x++ )
10                 c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12             for( int k = 0; k < n; k++ )
13             {
14                 __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                 for (int x = 0; x < UNROLL; x++)
16                     c[x] = _mm256_add_pd(c[x],
17                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18             }
19
20             for ( int x = 0; x < UNROLL; x++ )
21                 _mm256_store_pd(C+i+x*4+j*n, c[x]);
22         }
23     }

```

图 4-80 使用 C 循环体特点为 x86 生成 AVX 子字并行指令的优化的 DGEMM C 语言版本（图 3-23）和使用循环展开开发指令级并行。图 4-81 给出了使用编译器产生的内循环的汇编语言，将 3 个 for 循环体进行展开以显示指令级并行

与前面的图 4-71 中的循环展开的例子一样，我们将循环展开 4 次（在 C 代码中我们使用常数 UNROLL 来控制我们希望展开的次数）。与图 3-23 中手工将 C 循环中每个循环体复制 4 份不同，我们可以依赖于 gcc 编译器的 -O3 优化选项来做展开。我们将每个循环体使用一个简单的 for 循环包起来形成 4 个迭代（第 9、14 和 20 行），将图 3-23 中的向量 c0 使用一个 4 元素数组 c[ ]（第 8、10、16 和 21 行）替换。

图 4-81 给出了展开后的汇编语言代码。正如所期望的一样，图 3-24 中的每条 AVX 指令在图 4-81 中有 4 个版本，只有一个例外。因为在循环中，我们可以反复使用寄存器 %ymm0 中的 B 元素的 4 个副本，所以我们只需要 vbroadcastd 指令的一个副本。因此，图 3-24 中的 5 条 AVX 指令变成了图 4-81 中的 17。另外，虽然常数和地址根据循环展开进行变化，7 条整数指令在两种情况下没有变化。所以，即使循环展开了 4 次，循环体中的指令数目只是翻倍：由 12 条变为 24 条。

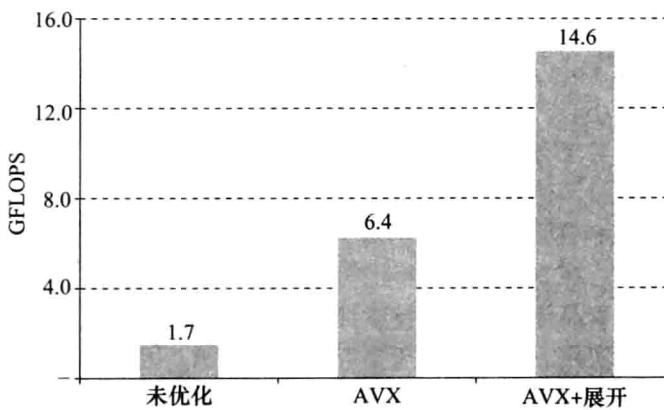
图 4-82 给出了使用 DGEMM 计算  $32 \times 32$  的矩阵从未优化 AVX 到使用循环展开的 AVX 时性能的提升情况。可以看出，循环展开使得性能增加了一倍以上，由 6.4GFLOPS 增加到了 14.6GFLOPS。相对于图 3-21 的未优化的 DGEMM，子字并行和指令级并行共同作用获得了 8.8 倍的加速比。

```

1  vmovapd (%r11),%ymm4      # Load 4 elements of C into %ymm4
2  mov    %rbx,%rax          # register %rax = %rbx
3  xor    %ecx,%ecx          # register %ecx = 0
4  vmovapd 0x20(%r11),%ymm3  # Load 4 elements of C into %ymm3
5  vmovapd 0x40(%r11),%ymm2  # Load 4 elements of C into %ymm2
6  vmovapd 0x60(%r11),%ymm1  # Load 4 elements of C into %ymm1
7  vbroadcastsd (%rcx,%r9,1),%ymm0  # Make 4 copies of B element
8  add    $0x8,%rcx          # register %rcx = %rcx + 8
9  vmulpd (%rax),%ymm0,%ymm5  # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4   # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3   # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0 # Parallel mul %ymm1,4 A elements
15 add    %r8,%rax           # register %rax = %rax + %r8
16 cmp    %r10,%rcx          # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2   # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1   # Parallel add %ymm0, %ymm1
19 jne    68 <dgemm+0x68>    # jump if not %r8 != %rax
20 add    $0x1,%esi           # register %esi = %esi + 1
21 vmovapd %ymm4,(%r11)       # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)   # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)   # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)   # Store %ymm1 into 4 C elements

```

图 4-81 对图 4-80 中循环展开后的 C 代码编译产生的嵌套循环体的 x86 汇编语言

图 4-82 三个用于计算  $32 \times 32$  矩阵的 DGE MM 版本的性能。相对于图 3-21 中未优化的代码，子字并行和指令级并行将性能提高了近 9 倍

**01 精解** 如 3.8 节中精解所提到的，这些结果是在 Turbo 模式关闭情况下获得的。如果将 Turbo 模式打开，与第 3 章相同，时钟频率提高了  $3.3/2.6 = 1.27$  倍，未优化的 DGE MM 性能将提升为 2.1GFLOPS，AVX 性能将提升为 8.1GFLOPS，循环展开后的 AVS 性能将提升为 18.6 GFLOPS。如 3.8 节所述，当一个八核芯片中只使用一个核时 Turbo 模式会很好的工作。

**01 精解** 虽然寄存器`%ymm5`在第 9~17 行被重用，但是由于 Intel Core i7 流水线对寄存器进行了重命名，因此没有流水线阻塞。

### 01 小测验

判断下列表述的正误。

1. Intel Core i7 使用多发射流水线直接执行 x86 指令。
2. A8 和 Core i7 都使用动态多发射。
3. Core i7 微体系结构中的寄存器比 x86 所要求的更多。
4. Intel Core i7 的流水线级数比早期 Intel Pentium 4 Prescott 的一半还少（见图 4-73）。

## 4.13 高级主题：通过硬件设计语言描述和建模流水线来介绍数字设计以及更多流水线示例

现代数字设计是用硬件描述语言和现代的计算机辅助综合工具完成的，其中综合工具能使用库和逻辑综合将描述转化为具体的硬件设计。关于这些语言和它们在数字设计中的使用有相关书籍说明。本节（在配套网站上）仅进行简单的介绍，并展示如何用一种硬件设计语言（Verilog）分别从行为级和可综合级描述 MIPS 控制。接着还提供了用 Verilog 描述的 MIPS 五级流水线行为级模型。最初的模型忽略了冒险，随后增加的部分着重于支持旁路、数据冒险和分支冒险所做的改变。

我们接着提供了大量使用单时钟周期图形化流水线表示的示意图，以帮助读者更好地理解执行一连串 MIPS 指令时流水线的工作细节。

353  
354

## 4.14 谬误与陷阱

**谬误：流水线是一种简单的结构。**

本书证明了正确设计流水线必须非常谨慎。我们的另一本教程《Computer Architecture: A Quantitative Approach》的第 1 版尽管经过了上百人的校对，并且曾经在 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。直到有人根据该书设计处理器时才发现了这个错误。用 Verilog 来描述一个如 Intel Core i7 的流水线需要几千行代码，从中可以看出流水线的复杂性，因此设计流水线必须非常小心。

**谬误：流水线概念的实现与工艺无关。**

当芯片上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，延迟分支（见 4.3 节的精解）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，延迟分支就成为多余的方法了。在 20 世纪 90 年代初期，动态流水线调度需要耗费大量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存储器更快，多个功能单元和动态流水线变得更加实用。当今，由于要考虑功耗问题，因此不能采用太激进的设计。

**陷阱：没有考虑指令集的设计反过来会影响流水线。**

许多流水线中遇到的困难都是由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各流水级的不均衡，从而阻碍了某个流水级的运行，而且它们还会使冒险检测的实现复杂化。这个问题已经解决，最初在 20 世纪 80 年代后期的 DEC VAX 8500 中，采用了微操作和微流水线的方案，正如今天的 Intel Core i7 所采用的一样。当然，在微操作和实际指令间的转化和一致性维护上，开销依然是存在的。
- 复杂的寻址模式可能引起很多问题。更新寄存器的寻址模式会使冒险的检测复杂化。而需

要多次访问存储器的寻址模式会使流水线的控制复杂化，并且难以保持流水线平稳流动。

- 最好的例子大概是 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。另一个例子是，Bhandarkar 和 Clark [1991] 使用 SPEC 基准测试程序比较了 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 的平均时钟周期数是 MIPS 的 2.7 倍，所以总体上 MIPS 更快一些。

355

## 4.15 本章小结

智慧十之八九体现在恰当的时机。

——美国谚语

在这一章我们看到，处理器的数据通路和控制通路的设计，可以从指令集系统和对工艺基本特性的理解开始。在 4.3 节，我们看到了在指令集体系结构确定和决定使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，背后的工艺也影响许多设计决策，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。

流水线提高了吞吐率，但不能提高指令的内在执行时间（指令延迟（instruction latency））；对某些指令而言，指令延迟与单周期实现的延迟类似。多发射增加了额外的允许每个时钟周期发射多条指令的数据通路硬件，但是却增加了有效延迟。为了减少简单的单周期实现数据通路的时钟周期，提出了流水线技术。相比之下，多发射关注于减少每条指令的时钟周期数（CPI）。

② 指令延迟：执行一条指令所真正花费的时间。

流水线和多发射都试着开发指令级并行。开发更高指令级并行的主要限制因素是存在数据相关和控制相关。在软硬件上都使用预测来调度和推测，是降低相关带来影响的主要手段。

我们展示了将 DGEMM 的循环展开 4 次来开发指令级并行，利用 Core i7 的乱序执行机制可使性能提升一倍以上。

20 世纪 90 年代中期我们开始使用更长的流水线、多发射和动态调度，这些技术帮助我们维持了从 20 世纪 80 年代早期以来每年 60% 的处理器性能增长速度。正如第 1 章中所提到的，这些微处理器依旧使用顺序执行程序模型，但是它们最终会遇到功耗墙。因此，工业界被迫转向在更粗粒度上开发并行性的多处理器（这是第 6 章的主题）。这种趋势也迫使设计者们对 20 世纪 90 年代中期一些发明的功耗 - 性能含义重新进行评价，其结果是在最新的微体系结构中使用了更简单而不是复杂的流水线。

为了维持通过并行处理器带来的计算性能提高，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主题——存储器层次。

356

## 4.16 历史观点和拓展阅读

这一部分放在配套网站中，讨论了第一个流水线处理器、最早的超标量处理器、乱序执行与推测执行技术的发展以及同时期编译器技术的发展。

## 4.17 练习题

4.1 考虑如下指令：

指令：AND Rd, Rs, Rt

解释:  $\text{Reg}[\text{Rd}] = \text{Reg}[\text{Rs}] \text{ AND } \text{Reg}[\text{Rt}]$

**4.1.1** [5] <4.1> 对上述指令而言, 图 4-2 中的控制单元将产生哪些控制信号?

**4.1.2** [5] <4.1> 对上述指令而言, 将用到哪些功能单元?

**4.1.3** [10] <4.1> 哪些功能单元会产生输出, 但输出不会被以上指令用到? 对以上指令而言, 哪些功能单元不产生任何输出?

**4.2** 图 4-2 中基本的单周期 MIPS 实现仅能实现某些指令。可以在这个指令集中加入新的指令, 但决定是否加入取决于给处理器的数据通路和控制通路增加的成本和复杂度。对下面的新指令而言, 试回答下列 3 个问题:

指令: LWI Rt, Rd (Rs)

解释:  $\text{Reg}[\text{Rt}] = \text{Mem}[\text{Reg}[\text{Rd}] + \text{Reg}[\text{Rs}]]$

**4.2.1** [10] <4.1> 对上述指令而言, 哪些已有的单元还可以被使用?

**4.2.2** [10] <4.1> 对上述指令而言, 还需要增加哪些功能单元?

**4.2.3** [10] <4.1> 为了支持这些指令, 需要在控制单元增加哪些信号?

**4.3** 当处理器设计者考虑改进处理器数据通路时, 往往要考虑性能与成本的折中。假设我们从图 4-2 的数据通路出发, 其中指令存储器、加法器、多选器、ALU、寄存器堆、数据存储器和控制单元的延迟分别为 400ps、100ps、30ps、120ps、200ps、350ps 和 100ps, 相应的成本分别为 1000、30、10、100、200、2000 和 500。

考虑给 ALU 增加一个乘法, 这将使 ALU 的延时增加 300ps, 同时 ALU 的成本增加 600。这样做的结果是需要执行的指令减少了 5%, 主要是由于不再需要模拟 MUL 指令。

**4.3.1** [10] <4.1> 改进前后的时钟周期分别是多少?

**4.3.2** [10] <4.1> 改进后将获得多大的加速比?

**4.3.3** [10] <4.1> 比较改进前后的性价比。

**4.4** 本练习题中的题目假定在实现一个处理器的数据通路时, 逻辑模块的延时如下:

I-Mem	Add	Mux	ALU	Regs	D-Mem	Sign-Extend	Shift-Left-2
200ps	70ps	20ps	90ps	90ps	250ps	15ps	10ps

**4.4.1** [10] <4.3> 如果处理器只需做连续取指这一件事 (见图 4-6), 那么时钟周期是多少?

**4.4.2** [10] <4.3> 考虑一个与图 4-11 类似的数据通路, 但是假设处理器只需处理无条件相对跳转指令, 那么时钟周期是多少?

**4.4.3** [10] <4.3> 重做练习题 4.4.2, 但这次假设只需处理有条件相对跳转指令。

本练习题中剩下的三个问题是关于 Shift-Left-2 数据通路单元的:

**4.4.4** [10] <4.3> 哪些类型的指令需要该单元?

**4.4.5** [20] <4.3> 对哪些类型的指令而言, 该单元位于关键路径上?

**4.4.6** [10] <4.3> 假设仅需支持 beq 指令和 add 指令, 讨论该单元的延迟变化对处理器时钟周期的影响。假定其他单元的延迟不变。

**4.5** 本练习题的问题中, 假定没有流水线阻塞, 各种类型所占的比例如下:

add	addi	not	beq	lw	sw
20%	20%	0%	25%	25%	10%

**4.5.1** [10] <4.3> 数据存储器平均用了多少时钟周期?

**4.5.2** [10] <4.3> 符号扩展电路的输入平均用了多少时钟周期? 在未用到该输入的其他时间, 符号扩展电路在做什么?

**4.6** 在制造硅芯片时, 材料 (例如, 硅) 的缺陷和制造错误会导致电路失效。一个非常普遍的问题是一根线上的信号会对相邻线上的信号产生影响, 这被称为串扰。有一类串扰问题是这样的, 某些线上的信号为常值 (如电源线), 该线附近的线也被固定为 0 (stuck-at-0) 或 1 (stuck-at-1)。下

面的问题中的缺陷发生在图 4-24 中寄存器堆的写寄存器输入端的第 0 位。

- 4.6.1** [10] <4.3, 4.4> 假设这样测试处理器的缺陷：先给 PC、寄存器堆、数据和指令存储器中设置一些值（可以自己选择），执行一条指令，然后读出 PC、寄存器堆和存储器中的值；最后检查这些值以判断处理器中是否存在缺陷。你能设计一个方案检查该信号上是否有固定为 0 缺陷吗？
- 4.6.2** [10] <4.3, 4.4> 重做练习题 4.6.1，这次检查固定为 1 缺陷。你能只设计一个测试方案同时检查固定为 0 缺陷和固定为 1 缺陷吗？如果可以，请解释如何实现；如果不能，请说明理由。
- 4.6.3** [60] <4.3, 4.4> 如果我们知道一个处理器在该信号上有一个固定为 1 缺陷，它还能用吗？为了使这个处理器仍然可用，我们必须将原来能在正常 MIPS 处理器上运行的程序作一些变换，使之可以在这个处理器上运行。假设指令存储器和数据存储器都很大，足够容纳变换后的程序。提示：将因为该缺陷不能用的指令替换为一系列能用的指令，这一系列指令与原指令功能相同。
- 4.6.4** [10] <4.3, 4.4> 重做练习题 4.6.1，这次检测控制信号 MemRead 是否存在该缺陷？如果 RegDst 控制信号为 0 时，MemRead 控制信号为 0，则有缺陷，否则无缺陷。
- 4.6.5** [10] <4.3, 4.4> 重做练习题 4.6.4，这次检测控制信号 Jump 是否存在该缺陷？如果 RegDst 控制信号为 0 时 Jump 控制信号为 0，则有缺陷，否则无缺陷。
- 4.7** 本练习题将讨论单周期数据通路中指令的执行细节。本练习题中的问题基于单周期处理器取入如下的指令字后的一个时钟周期：

101011000110001000000000000010100

假定数据存储器是全 0 且处理器的寄存器在取入以上指令字的时钟周期的开始时的内容如下：

r0	r1	r2	r3	r4	r5	r6	r8	r12	r31
0	-1	2	-3	-4	10	6	8	2	-16

- 4.7.1** [5] <4.4> 对该指令字而言，符号扩展单元和左移两位单元（图 4-24 的左上角）的输出是什么？
- 4.7.2** [10] <4.4> 对该指令字而言，ALU 控制单元的输入是什么？
- 4.7.3** [10] <4.4> 该指令执行后的新 PC 值是什么？在图 4-24 中灰线决定该新 PC 值的路径。
- 4.7.4** [10] <4.4> 对给定的指令字和寄存器堆初值，给出每个多选器数据输出的值。
- 4.7.5** [10] <4.4> 给出 ALU 和两个加法器数据输入的值。
- 4.7.6** [10] <4.4> 给出寄存器堆所有输入信号的值。
- 4.8** 本练习题讨论流水线对处理器时钟周期的影响。下面给出了数据通路中不同阶段的延迟情况：

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

另外，假定处理器执行各种指令的比率如下面所示：

alu	beq	lw	sw
45%	20%	20%	15%

- 4.8.1** [5] <4.5> 流水线处理器与非流水线处理器的时钟周期分别是多少？
- 4.8.2** [10] <4.5> lw 指令在流水线处理器和非流水线处理器中的总延迟分别是多少？
- 4.8.3** [10] <4.5> 如果可以将原流水线数据通路的一级划分为两级，每级的延迟是原级的一半，那么你会选择哪一级进行划分？划分后处理器的时钟周期为多少？
- 4.8.4** [10] <4.5> 假设没有阻塞和冒险，数据存储器的利用率是多少（占总周期数的百分比）？
- 4.8.5** [10] <4.5> 假设没有阻塞和冒险，寄存器堆的写寄存器端口的利用率是多少？
- 4.8.6** [30] <4.5> 假设一种多周期的处理器设计，其中每条指令需要多个时钟周期完成，但上一条指令完成前不取下一条指令。在这种设计中，指令仅经过其所需的阶段（例如，存储指令仅需 4 个时钟周期，因为其不需要 WB 阶段）。比较单周期设计、多周期设计和流水线设计三者的时钟周期和总执行时间。

- 4.9** 本练习题讨论数据相关如何影响4.5节中基本五级流水线的运行。结合下面的指令序列完成问题：

```
or r1,r2,r3
or r2,r1,r4
or r1,r1,r2
```

另外，假定每种相关旁路的周期时间如下：

无旁路	全旁路	ALU-ALU 旁路
250ps	300ps	290ps

- 4.9.1** [10] <4.5> 指出指令序列中存在的相关及其类型。
- 4.9.2** [10] <4.5> 假设该流水线处理器没有旁路，指出指令序列中存在的冒险并加入nop指令以消除冒险。
- 4.9.3** [10] <4.5> 假设该流水线处理器中有充分的旁路。指出指令序列中存在的冒险并加入nop指令以消除冒险。
- 4.9.4** [10] <4.5> 该指令序列在无旁路和有充分的旁路时，总执行时间分别是多少？后者相对于前者的加速比是多少？
- 4.9.5** [10] <4.5> 如果仅有ALU至ALU的旁路（没有从MEM到EX的旁路），如何加入nop指令以消除可能的冒险？
- 4.9.6** [10] <4.5> 该指令序列在仅有ALU至ALU的旁路时，总执行时间分别是多少？与无旁路的情况相比，加速比是多少？

- 361** **4.10** 本练习题将考查资源冒险、控制冒险和指令集体系结构设计如何影响流水线的执行。本练习题中的问题涉及下面的MIPS代码片段

```
sw r16,12(r6)
lw r16,8(r6)
beq r5,r4,Label # Assume r5!=r4
add r5,r1,r4
slt r5,r15,r4
```

假定每个流水级的延迟如下：

IF	ID	EX	MEM	WB
200ps	120ps	150ps	190ps	100ps

- 4.10.1** [10] <4.5> 假设所有的分支都被正确预测（控制冒险完全被消除）且没有使用延迟时间槽，并且只有一个存储器（既存储指令又存储数据）。如果一个时钟周期内同时取指和取数据就会发生结构冒险。为保证前进，该冒险必须始终以有利于取数指令的方式解决。该指令序列在仅有一个存储器的五级流水线中执行的总时间是多少？我们知道插入nop指令可以消除数据冒险，可以用同样的方法消除这里的结构冒险吗？为什么？
- 4.10.2** [20] <4.5> 假设所有的分支都被正确预测（控制冒险完全被消除）且没有使用延迟时间槽。如果我们改变存取指令的格式，仅使用寄存器（不含偏移地址）进行寻址，那么这些指令不再需要使用ALU。结果是MEM级和EX级可以重叠成一级，整个流水线也就成为四级。改变该代码以适应上述ISA的改变。假设这个改变不影响时钟周期，对该指令序列而言，这个改变造成的加速比是多少？
- 4.10.3** [10] <4.5> 假设在分支时进行阻塞且没有使用延迟时间槽，那么在ID级确定分支方向相对于在EX级确定分支方向的加速比是多少？
- 4.10.4** [10] <4.5> 在给定的流水级延迟下，重做练习题4.10.2，考虑可能的时钟周期变化。如果EX级和MEM级重叠起来，它们的大部分工作可以并行执行。这样重叠后的EX/MEM级的延迟是原来两级的较大者，不能并行工作时延迟还要再加20ps。

**4.10.5** [10] <4.5> 在给定的流水级延迟下，重做练习题 4.10.3，考虑可能的时钟周期变化。假设分支方向判断从 EX 级移到 ID 级时，ID 级的延迟增加 50%，而 EX 级的延迟减少 10ps。

**4.10.6** [10] <4.5> 假设在分支时进行阻塞且没有使用延迟时间槽，如果 beq 指令的地址计算移到 MEM 级，时钟周期将变为多少？该指令序列的总执行时间将变为多少？加速比是多少？假设分支方向判断从 EX 级移到 MEM 级时，EX 级的延迟减少 20ps，而 MEM 级的延迟不变。

**4.11** 考虑下面的循环：

```
loop:lw r1,0(r1)
      and r1,r1,r2
      lw r1,0(r1)
      lw r1,0(r1)
      beq r1,r0,loop
```

假定使用了完美的分支预测（没有控制冒险导致的阻塞），没有延迟槽，流水线支持完全旁路。另外假定循环结束前该循环迭代了很多次。

**4.11.1** [10] <4.6> 画出循环第三次执行的流水线图，从取出循环的首条指令开始至取出下次循环的首条指令（不包括该次取指）结束。给出这段时间内流水线中的所有指令（不仅仅是第三次循环中的那些指令）。

**4.11.2** [10] <4.6> 在这段时间内有百分之多少五级流水线都在做有用的工作？

**4.12** 本练习题讨论流水线处理器中旁路的成本/复杂度/性能折中。参考图 4-45 的流水线数据通路，假设指令中有部分存在 RAW (read after write, 写后读) 数据相关。RAW 数据相关根据生成结果的流水级 (EX 或 MEM) 和使用结果的流水级 (1st 意味着生成结果后的第一条指令，2nd 意味着生成结果后的第二条指令) 确认。假设在时钟周期的前半部分写寄存器，在后半部分读寄存器，这样 “EX to 3rd” 和 “MEM to 3rd” 相关不会产生数据冒险。最后假设无数据冒险时处理器的 CPI 为 1。

仅 EX to 1 <sup>st</sup>	仅 MEM to 1 <sup>st</sup>	仅 EX to 2 <sup>nd</sup>	仅 MEM to 2 <sup>nd</sup>	EX to 1 <sup>st</sup> 和 MEM to 2 <sup>nd</sup>	其他 RAW 相关
5%	20%	5%	10%	10%	10%

假定各级流水线延迟如下。其中 EX 级给出了不同旁路情况下的延迟。

IF	ID	EX (无旁路)	EX (全旁路)	EX (仅有 EX/MEM 的旁路)	EX (仅有 (MEM/WB 的旁路)	MEM	WB
150ps	100ps	120ps	150ps	140ps	130ps	120ps	100ps

**4.12.1** [10] <4.7> 如果不使用旁路，会有百分之多少的时钟周期因为数据冒险阻塞？

**4.12.2** [5] <4.7> 如果使用完全的旁路（旁路所有可以旁路的结果），会有百分之多少的时钟周期因为数据冒险阻塞？

**4.12.3** [10] <4.7> 假设不能提供三输入多选器（对完全的旁路是必需的），我们必须确定从 EX/MEM 流水线寄存器旁路（旁路下一个周期）还是从 MEM/WB 流水线寄存器旁路（旁路 2 个周期）更好？哪种方法会产生更少的数据阻塞？

**4.12.4** [10] <4.7> 对给定的冒险概念和流水级延迟，完全的旁路相对于无旁路的加速比是多少？

**4.12.5** [10] <4.7> 如果加入能消除所有数据冒险的时间旅行旁路，其相对于具有旁路机制的处理器的加速比是多少？假设在 EX 级完全旁路的基础上加入这个还没发明的时间旅行旁路的代价是增加 100ps 的延迟。

**4.12.6** [20] <4.7> 重做练习题 4.12.3，这次问哪种方法会产生更小的 CPI。

**4.13** 本练习题讨论旁路、冒险检测和指令集设计之间的关系。根据下面的指令序列回答下列问题。假设其在一个五级流水线数据通路上执行。

```

add r5,r2,r1
lw r3,4(r5)
lw r2,0(r2)
or r3,r5,r3
sw r3,0(r5)

```

- 364 4.13.1 [5] <4.7> 如果没有旁路或冒险检测电路, 请插入 nop 指令以保证正确执行。
- 4.13.2 [10] <4.7> 重做练习题 4.13.1, 这次仅当通过改变或重排序指令都不能避免冒险时才插入 nop 指令。假设可以使用寄存器 R7 作为临时寄存器。
- 4.13.3 [10] <4.7> 如果处理器中存在旁路, 但忘了实现冒险检测单元 (以为实现了), 代码执行时会发生什么情况?
- 4.13.4 [20] <4.7> 如果存在旁路, 在执行指令序列的前 5 个时钟周期, 通过图 4-60 中的冒险检测和旁路单元指出每个周期中哪些信号是有效的。
- 4.13.5 [10] <4.7> 如果没有旁路, 对图 4-60 中的冒险检测单元来说还需要哪些新的输入输出信号? 以该指令序列为列, 说明为什么需要这些信号。
- 4.13.6 [20] <4.7> 对练习题 4.13.5 中新的冒险检测单元来说, 给出执行时前 5 个时钟周期中每个周期的输出信号 (使有效)。
- 4.14 本练习题讨论流水线处理器的延迟时间槽、控制冒险和分支执行之间的关系。假设下列 MIPS 代码在一个五级流水线、有完全旁路和预测分支总发生的处理器上运行。试根据下面的两个序列回答下列问题。

```

lw r2,0(r1)
label1: beq r2,r0,label2 # not taken once, then taken
        lw r3,0(r2)
        beq r3,r0,label1 # taken
        add r1,r3,r1
label2: sw r1,0(r2)

```

365

- 4.14.1 [10] <4.8> 画出该指令序列的流水线执行图, 假设没有延迟时间槽而且在 EX 级执行分支。
- 4.14.2 [10] <4.8> 重做练习题 4.14.1, 但是假设使用了延迟时间槽。给定的代码中, 跟在分支之后的指令是该分支的延迟槽指令。
- 4.14.3 [20] <4.8> 不同于需要一个 ALU 操作, 另一种提前确定分支方向的方法是使用 “bez rd, label” 和 “bne rd, label” 之类的条件分支指令, 该类条件分支指令根据寄存器值是否为零决定是否分支。变换原指令序列以使用该类条件分支指令而非 beq 指令。假设寄存器 \$8<sup>⊖</sup> 是临时寄存器, 并且可以使用 R 型指令 seq (set if equal, 相等置位)。
- 4.8 节说明了如何把分支执行提前到 ID 级以减少控制冒险。这个方法需要在 ID 级增加一个专用的比较器, 如图 4-62 所示。但是这个方法增加了 ID 级的延迟, 并且需要额外的旁路逻辑和冒险检测。
- 4.14.4 [10] <4.8> 以指令序列中第一条分支指令为例, 说明图 4-62 中为支持在 ID 级执行分支应加入的冒险检测逻辑。该逻辑需要检测什么类型的冒险?
- 4.14.5 [10] <4.8> 对给定的指令序列, 把执行分支移到 ID 级带来的加速比是多少? 为什么? 假设 ID 级进行的额外比较不影响时钟周期。
- 4.14.6 [10] <4.8> 以指令序列中第一条分支指令为例, 说明为支持在 ID 级执行分支应加入的旁路。比较新旁路单元与图 4-62 中旁路单元的复杂度。
- 4.15 一个好的分支预测器有多重要取决于条件分支指令的频率, 它与分支预测器的精度共同决定误预测分支导致的阻塞时间长短。在本练习题中, 假设指令的动态执行频度如下:

R型	beq	jmp	lw	sw
40%	25%	5%	25%	5%

⊖ 原书为 R8, 有误。——译者注

假定分支预测器的精度如下：

分支总发生	分支总不发生	2位预测器
45%	55%	85%

4.15.1 [10] <4.8> 误预测分支导致的阻塞将增加 CPI。对分支总发生预测器而言，误预测分支将导致 CPI 增加多少？假设分支方向在 EX 级确定，没有数据冒险且不使用延迟时间槽。

4.15.2 [10] <4.8> 重做练习题 4.15.1，这次改为分支总不发生预测器。

4.15.3 [10] <4.8> 重做练习题 4.15.1，这次改为 2 位分支预测器。

4.15.4 [10] <4.8> 对 2 位分支预测器而言，将一半分支指令用 ALU 指令替代（一条 ALU 指令替代一条分支指令）将获得的加速比是多少？假设被正确预测的分支指令和被不正确预测的分支指令被取代的概率相同。

4.15.5 [10] <4.8> 对 2 位分支预测器而言，将一半分支指令用 ALU 指令替代（两条 ALU 指令替代一条分支指令）将获得的加速比是多少？假设被正确预测的分支指令和被不正确预测的分支指令被取代的概率相同。

4.15.6 [10] <4.8> 有些分支是非常容易预测的。假设 80% 的分支指令都是非常容易预测的循环返回分支，那么 2 位分支预测器对剩下的 20% 分支指令的预测精度是多少？

4.16 本练习题讨论不同分支预测器对给定分支模式（如循环）的预测精度。给定的分支模式为：T, NT, T, T, NT。

4.16.1 [5] <4.8> 对该分支模式，分支总发生预测器与分支总不发生预测器的准确率分别是多少？

4.16.2 [5] <4.8> 对该分支模式的前 4 个分支而言，2 位分支预测器的准确率是多少？假设预测器的初始状态与图 4-63 左下角状态相同（预测未发生）。

4.16.3 [10] <4.8> 如果该分支模式一直重复下去，2 位分支预测器的准确率是多少？

4.16.4 [30] <4.8> 如果该分支模式一直重复下去，设计一个能取得最高准确率的预测器。这个预测器必须是一个时序电路，有一个输出表示预测结果（1 表示发生，0 表示未发生），除了时钟和指示当前指令是条件分支指令的信号外没有其他输入。

4.16.5 [10] <4.8> 如果有一个分支模式与该分支模式完全相反且一直重复下去，那么在练习题 4.16.4 中你设计的预测器对这个分支的准确率是多少？

4.16.6 [20] <4.8> 重做练习题 4.16.4，这次你的预测器最终（可能需要一个热身过程）可以同时完美地预测该分支模式及完全相反的分支模式（假设分支模式一直重复下去）。这个预测器应该有一个输入告诉它真实的分支结果。提示：这个输入可以帮助预测器判断是两个分支模式中的哪一个。

4.17 本练习题讨论异常处理对流水线设计的影响。根据下面两条指令回答前三个问题：

指令 1	指令 2
BNE R1, R2, Label	LW R1, 0(R1)

4.17.1 [5] <4.9> 每条指令分别可能产生什么异常？对每个可能产生的异常，指出其将在哪个流水线级被检测到。

4.17.2 [10] <4.9> 如果每个异常都有独立的处理程序地址，流水线应该怎样设计才能处理异常？假设设计处理器时已知每个异常处理程序的地址。

4.17.3 [10] <4.9> 如果第二条指令紧跟第一条指令从表中取出，试说明第一条指令发生异常（见练习题 4.17.1）时流水线的运行情况。给出从第一条指令取指开始到异常处理程序第一条指令完成时的流水线运行图。

4.17.4 [20] <4.9> 在向量异常处理中，异常处理程序地址表在数据存储器的一个固定位置。改变流水线的实现以支持向量异常处理。重做练习题 4.17.3，这次使用支持向量异常处理的流水线。

**4.17.5** [15] <4.9> 我们想要在仅有一个固定处理器地址的处理器上模拟向量异常处理（见练习题4.17.4），写出相应的程序。提示：这段程序应识别异常类型，从异常向量表中获得正确地址，然后跳转到该异常处理程序处。

**4.18** 本练习题比较单发射和双发射处理器的性能，并考虑对双发射处理器进行程序优化。根据下表的C代码分别回答下列问题。

```
for(i=0;i!=j;i+=2)
    b[i]=a[i]-a[i+1];
```

在编写MIPS代码时，假设变量被保存在寄存器中，如下表所示，除了空闲的寄存器，其余寄存器都被用来保存变量的值，因此不能再用作其他用途。

i	j	a	b	c	空闲
R5	R6	R1	R2	R3	R10, R11, R12

**4.18.1** [10] <4.10> 将这段C代码翻译成MIPS代码。这种翻译必须是直接的，不允许对代码进行重排序以达到更好的性能。

**4.18.2** [10] <4.10> 如果循环仅执行两次后就退出，画出练习题4.18.1中MIPS代码在图4-69的双发射处理器中执行的流水线图。假设处理器能进行完美的分支预测，并且一个周期能取任意两条指令（不仅仅是连续的两条指令）。

**4.18.3** [10] <4.10> 重排序练习题4.18.1中的MIPS代码，以在图4-69的双发射静态调度处理器上获得更好的性能。  
368

**4.18.4** [10] <4.10> 重做练习题4.18.2，但这次使用练习题4.18.3中的MIPS代码。

**4.18.5** [10] <4.10> 从单发射处理器到图4-69的双发射处理器，性能的加速比是多少？在单发射和双发射处理器分别运行练习题4.18.1的代码，假设循环执行1 000 000次。与练习题4.18.2相同，假设处理器能进行完美的分支预测，并且一个双发射处理器在同一个周期能取任意两条指令。

**4.18.6** [10] <4.10> 重做练习题4.18.5，这次假设双发射处理器中一条指令可以是任意类型的，而另一条指令必须是非存取指令。

**4.19** 本练习题讨论性能与功耗的关系。假设数据通路各部件（指令存储器、寄存器、数据存储器）的功耗如下表所示，其他部件的功耗可以忽略。

指令存储器	一次读寄存器	写寄存器	读数据存储器	写数据存储器
140pJ	70pJ	60pJ	140pJ	120pJ

假定数据通路上的部件延迟如下表所示，其他功能部件的延迟可以忽略。

指令存储器	控制	寄存器读或写	ALU	数据存储读或写
200ps	150ps	90ps	90ps	250ps

**4.19.1** [10] <4.3, 4.6, 4.14> 在单周期设计和五级流水线设计中执行一条加法指令的功耗分别是多少？

**4.19.2** [10] <4.6, 4.14> 功耗消耗最大的MIPS指令是哪一条？执行这条指令的功耗是多少？

**4.19.3** [10] <4.6, 4.14> 如果功耗是最重要的约束，应该怎样设计流水线？在这种流水线下执行一条lw指令时的功耗减小的比例是多少？

**4.19.4** [10] <4.6, 4.14> 如果像练习题4.19.3中那样设计流水线，其对性能会造成多大的影响？

**4.19.5** [10] <4.6, 4.14> 我们可以去掉MemRead控制信号，即每个周期都读数据存储器（MemRead恒为1）。解释为什么去掉该控制信号后处理器依然能正常工作。它对时钟频率和功耗又有什么影响？

**4.19.6** [10] <4.6, 4.14> 如果一个单元的空闲功耗仅为正常工作时的10%，每个周期指令存储器的功耗是多少？指令存储器消耗的功耗中有多少是空闲功耗？  
369

**01 小测验答案**

- 4.1 控制器、数据通路、存储器。少了输入和输出。
- 4.2 错。边沿触发状态单元可以同时进行读写。
- 4.3 I. a; II. c。
- 4.4 是，Branch 与 ALUOp0 是相同的。而且，MemtoReg 和 RegDst 是相反的，不需要额外的反相器。仅使用另外一个信号，并翻转多路选择器的输入即可。
- 4.5 I. 因为 lw 的结果而阻塞；II. 旁路第一个 add 的结果写入 \$t1；III. 不需要阻塞或旁路。
- 4.6 2 和 4 正确，其余错误。
- 4.8 1. 预测不发生；2. 预测发生；3. 动态预测。
- 4.9 第一条指令，因为在逻辑上它最先执行。
- 4.10 1. 都有；2. 都有；3. 软件；4. 硬件；5. 硬件；6. 硬件；7. 都有；8. 硬件；9. 都有。
- 4.11 前两个错误，后两个正确。