数据冒险

由于一个步骤必须等待另一个步骤完成而导致的流水线停顿叫作**数据冒险**（data hazard）。假设你在叠衣服时发现一只袜子找不到与之匹配的另一只。一种可能的策略是跑到房间，在衣橱中找，看是否能找到另一只。显然，当你在找袜子时，完成烘干准备被折叠的衣服和那些已经洗完准备去烘干的衣服，不得不停顿等待。

数据冒险：也称为流水线数据冒险，因无法提供指令执行所需数据而导致指令不能在预期的时钟周期内执行。

在计算机流水线中，数据冒险源于一条指令依赖于前面一条尚在流水线中的指令（这种关系在洗衣例子中并不存在）。例如，假设有一条加法指令，它后面紧跟着一条使用加法的和的减法指令（x19）：

```
add  x19, x0, x1
sub  x2, x19, x3
```

在不做任何干预的情况下，这一数据冒险会严重地阻碍流水线。add 指令直到第五个阶段才写结果，这将浪费三个时钟周期。

尽管可以尝试通过编译器来消除这些冒险，但结果并不令人满意。这些依赖经常发生，并且导致的延迟太长，所以不可能指望编译器将我们从这个困境中解救出来。

一种基本的解决方案是基于以下发现：不需要等待指令完成就可以尝试解决数据冒险。对于上面的代码序列，一旦 ALU 计算出加法的和，就可将其作为减法的输入。向内部资源添加额外的硬件以尽快找到缺少的运算项的方法，称为**前递**（forwarding）或**旁路**（bypassing）。

前递或旁路：一种解决数据冒险的方法，提前从内部缓冲中取到数据，而不是等到数据到达程序员可见的寄存器或存储器。

例题 | 两条指令的前递

对于上面的两条指令，说明前递将连接哪些流水级。图 4-26 表示流水线五个阶段的数据通路。与图 4-23 中的洗衣例子的流水线类似，每条指令的数据通路排成一行。

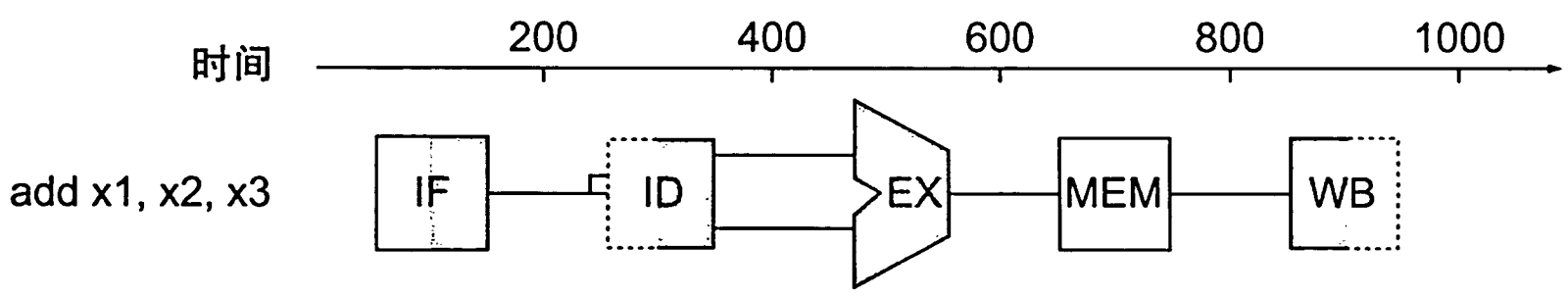


图 4-26　指令流水线的图形表示，本质上与图 4-23 的洗衣流水线类似。本图及本章使用图形符号和流水线阶段的缩写来表示物理资源。五个阶段的图形符号分别为：IF 表示取指令阶段，方框表示指令存储器；ID 表示指令译码 / 读寄存器阶段，虚线框表示正在被读的寄存器堆；EX 表示执行阶段，图中图形表示 ALU；MEM 表示存储器访问阶段，方框表示数据存储器；WB 表示写回阶段，虚线表示被写入的寄存器堆。阴影表示该单元被指令使用。因为 add 指令不访问数据存储器，所以 MEM 没有阴影。寄存器堆或存储器右半部分为阴影表示该阶段它们被读，而左半部分为阴影表示该阶段它们被写。因此，ID 的右半部分在第二阶段为阴影，因为寄存器堆被读，而 WB 的左半部分在第五阶段为阴影，因为寄存器堆被写

答案 | 图 4-27 所示为将 add 指令的执行阶段后的 x1 中的值，前递给 sub 指令作为执行阶段的输入。

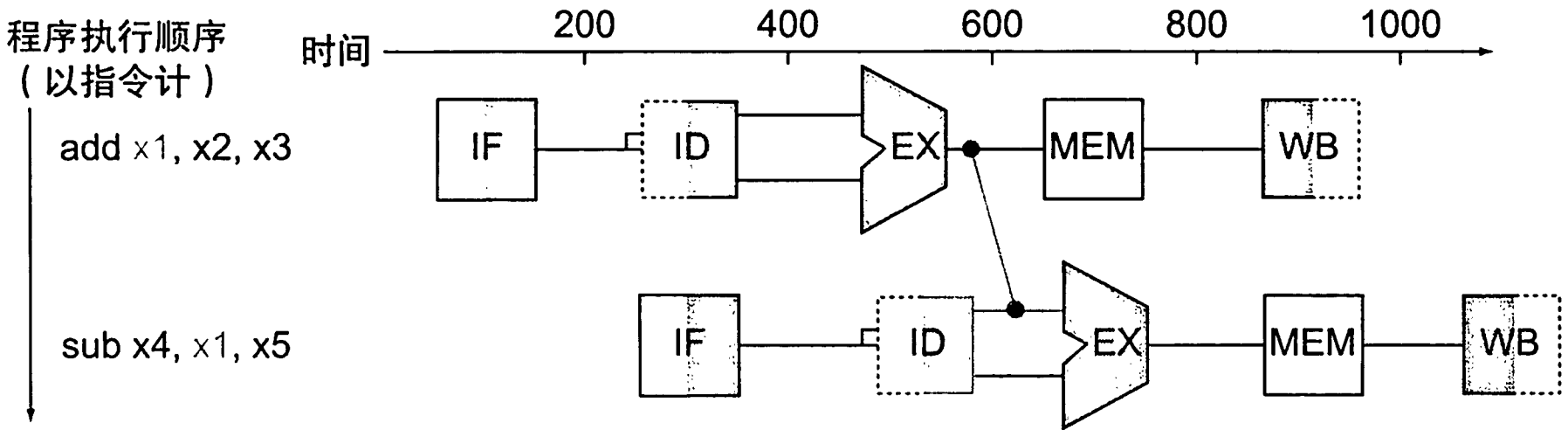


图 4-27 前递的图形表示。灰色线表示前递的路径，将 add 指令 EX 阶段的输出前递到 sub 指令 EX 阶段的输入，替换 sub 指令在第二阶段读出的寄存器 x1 的值

在图 4-27 中，仅当目标阶段在时间上晚于源阶段时，前递路径才有效。例如，从第一条指令存储器访问阶段的输出到下一条指令执行阶段的输入不可能存在有效前递路径，否则意味着时间倒流。

前递的效果很好（详见 4.7 节），但不能避免所有的流水线停顿。例如，假设第一条指令是 load x1 而不是加法指令，正如图 4-27 所述，在第一个指令的第四个阶段之后，sub 指令所需的数据才可用，这对于 sub 指令第三个阶段的输入来说太迟了。因此，即使使用前递，流水线也不得不停顿一个阶段来处理**载入－使用型数据冒险**（load-use data hazard），如图 4-28 所示。该图包含流水线的一个重要概念，正式叫法是**流水线停顿**（pipeline stall），但通常俗称为**气泡**（bubble）。我们经常看到流水线中发生停顿。4.7 节介绍如何处理这种类似本例的复杂情况，即使用硬件检测和停顿，或由软件对代码进行重新排序以尽量避免载入－使用型流水线停顿。

载入－使用型数据冒险：一种特定形式的数据冒险，指当载入指令要取的数据还没取回时，其他指令就需要该数据的情况。

流水线停顿：也称为气泡，为了解决冒险而实施的一种阻塞。

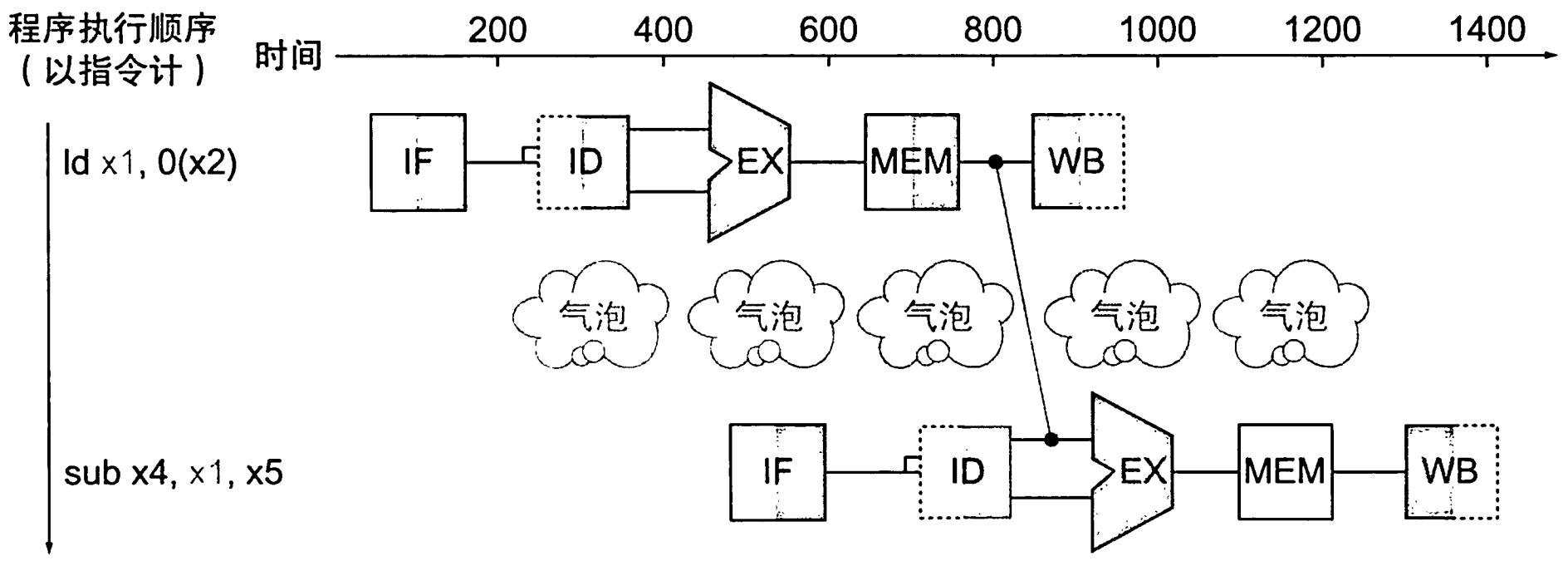


图 4-28 当一条 load 指令之后紧跟着一条需要使用其结果的 R 型指令时，即使使用前递也需要停顿。如果不停顿，从存储器访问阶段的输出到执行阶段的输入这条路径意味着时间倒流，这是不可能的。该图实际是一个示意图，因为直到 sub 指令被取出并译码后才知道是否需要停顿。4.7 节介绍了发生冒险时的细节

例题 | 重排代码以避免流水线停顿

考虑以下 C 语言代码段：

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

下面是这个代码段生成的RISC-V代码，假设所有变量都在存储器中，并且以x31作为基址，加偏移后即可访问这些变量：

```
ld      x1, 0(x31)   // Load b
ld      x2, 8(x31)   // Load e
add     x3, x1, x2   // b + e
sd      x3, 24(x31)  // Store a
ld      x4, 16(x31)  // Load f
add     x5, x1, x4   // b + f
sd      x5, 32(x31)  // Store c
```

试找出上述代码段中的冒险并重新排列指令以避免流水线停顿。

答案 两条add指令都有冒险，因为它们分别依赖于上一条ld指令。请注意，前递消除了其他几种潜在冒险，包括第一条add指令对第一条ld指令的依赖，以及sd指令带来的冒险。把第三条ld指令提前为第三条指令可以消除这两个冒险：

```
ld    x1, 0(x31)
ld    x2, 8(x31)
ld    x4, 16(x31)
add   x3, x1, x2
sd    x3, 24(x31)
add   x5, x1, x4
sd    x5, 32(x31)
```

在具有前递的流水线处理器上，执行重新排序的指令序列将比原始版本快两个时钟周期。

除4.5.1节提到的RISCV的三个特点之外，前递引出了RISC-V体系结构的另一个特点。即每条RISC-V指令最多写一个结果，并在流水线的最后一个阶段执行写操作。如果每条指令有多个结果要前递，或者需要在指令执行的更早阶段写入结果，前递设计会复杂得多。

详细阐述 “前递”这个名称来源于将结果从前面一条指令直接传递给后面一条指令的思想。“旁路”这个名称来源于将结果绕过寄存器堆，直接传递给需要它的单元的思想。

控制冒险

第三种冒险称为**控制冒险**，出现在以下情况：需要根据一条指令的结果做出决定，而其他指令正在执行。

控制冒险：也称为分支冒险，由于取到的指令并不是所需要的，或者指令地址的流向不是流水线所预期的，导致正确的指令无法在正确的时钟周期内执行。

假设洗衣店的工作人员接到一个令人高兴的任务：清洁足球队队服。根据衣服的污浊程度，需要确定清洗剂的用量和水温设置是否合适，以致能洗净衣物又不会由于清洗剂过量而磨损衣物。在洗衣流水线中，必须等到第二步结束，检查已经烘干的衣服，才知道是否需要改变洗衣机设置。这种情况该怎么办？

有两种办法可以解决洗衣问题中的控制冒险，也适用于计算机中的相同问题，以下是第一种办法。

停顿：第一批衣物被烘干之前，按顺序操作，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有效，但速度很慢。

计算机中相同的问题是条件分支指令。请注意，在取出分支指令后，紧跟着在下一个时钟周期就会取下一条指令。但是流水线并不知道下一条指令应该是什么，因为它刚刚从存

储器中取出分支指令！就像洗衣问题一样，一种可能的解决方案是在取出分支指令后立即停顿，一直等到流水线确定分支指令的结果并知道要从哪个地址取下一条指令为止。

假设加入足够多的额外硬件，使得在流水线第二个阶段能够完成测试寄存器、计算分支目标地址和更新 PC（详见 4.8 节）。通过这些硬件资源，包含条件分支指令的流水线如图 4-29 所示。如果分支指令的条件不成立，要执行的指令在开始执行之前需额外停顿一个时钟周期（200ps）。

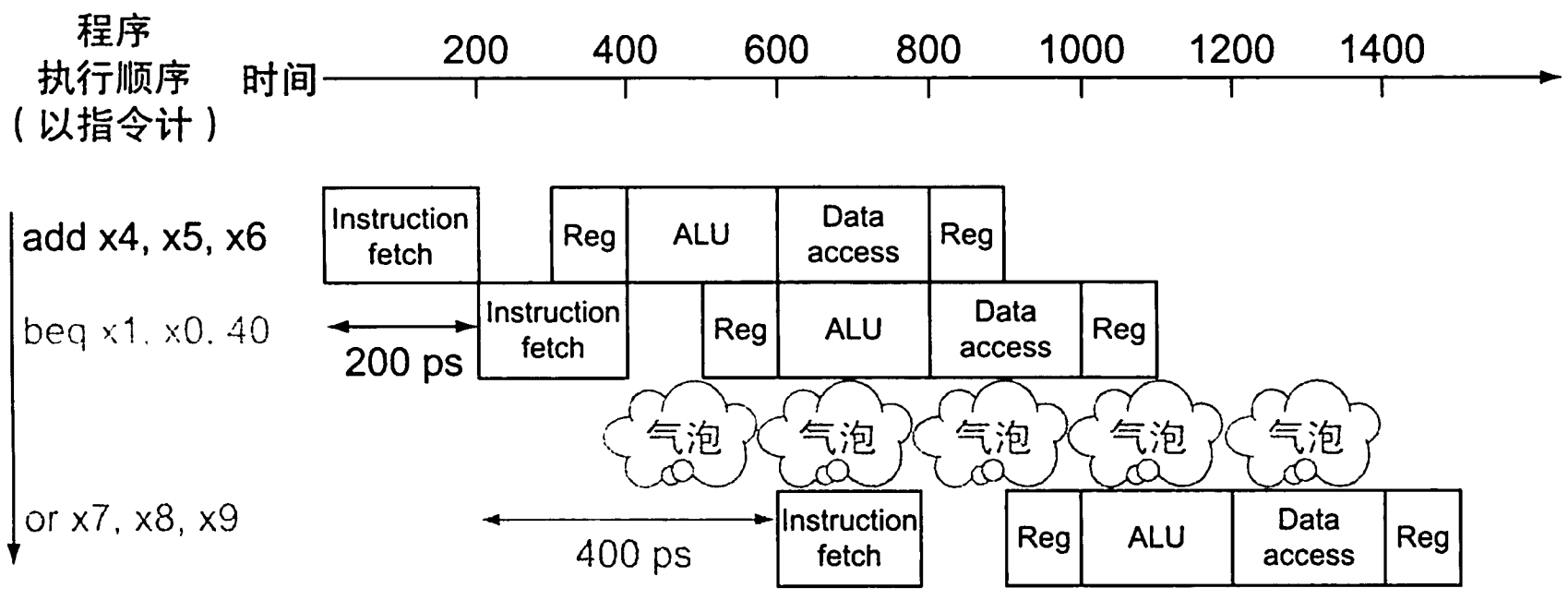


图 4-29 每遇到条件分支指令就停顿以避免控制冒险的流水线。本例假定条件分支指令发生跳转，并且分支目标地址处的指令是 or 指令。分支指令后会插入一个周期的停顿或气泡。我们将在 4.8 节中看到，实际中产生一次停顿的过程要更复杂。这种方法对性能的影响与插入一个气泡是一样的

例题 | 分支指令停顿的性能

估计对分支指令带来的停顿对指令时钟周期数（CPI）的影响。假设其他指令的 CPI 均为 1。

答案 | 图 3-28 表明在 SPEC int2006 中，条件分支指令占执行指令的 17%。由于其他指令的 CPI 为 1，而条件分支指令由于停顿多一个时钟周期，所以平均 CPI 为 1.17，与理想情况相比，速度下降了 1.17 倍。

对较长的流水线而言，通常无法在第二阶段解决分支指令的问题，那么如果每个条件分支指令都停顿，将导致更严重的速度下降。对大多数计算机来说，这种方法的代价太大，根据第 1 章中的伟大思想，由此产生了解决控制冒险的第二个方法：

预测：如果你确定清洗队服的设置是正确的，就预测它可以工作，那么在等待第一批衣服被烘干的同时清洗第二批衣服。

如果预测正确，这个方法不会减慢流水线。但是如果预测错误，就需要重新清洗做预测时所清洗的那些衣服。

计算机确实采用预测来处理条件分支。一种简单的方法是总是预测条件分支指令不发生跳转。如果预测正确，流水线将全速前进。只有条件分支指令发生跳转时，流水线才会停顿。图 4-30 给出了这样一个例子。

更成熟的**分支预测**是预测一些条件分支指令发生跳转，而另一些不发生跳转。在洗衣的类比中，夜晚和主场比赛的队服采用一种洗衣设置，而白天和客场比赛的队服则采用另一种设置。在计算机程序中，循环底部是条件分支指令，并会跳转回到循环的顶部。由

分支预测：一种解决分支冒险的方法。它预测分支的结果并沿预测方向执行，而不是等分支结果确定后才开始执行。

于它们很可能发生分支并且向回跳转，所以可以预测发生分支并跳到靠前的地址处。

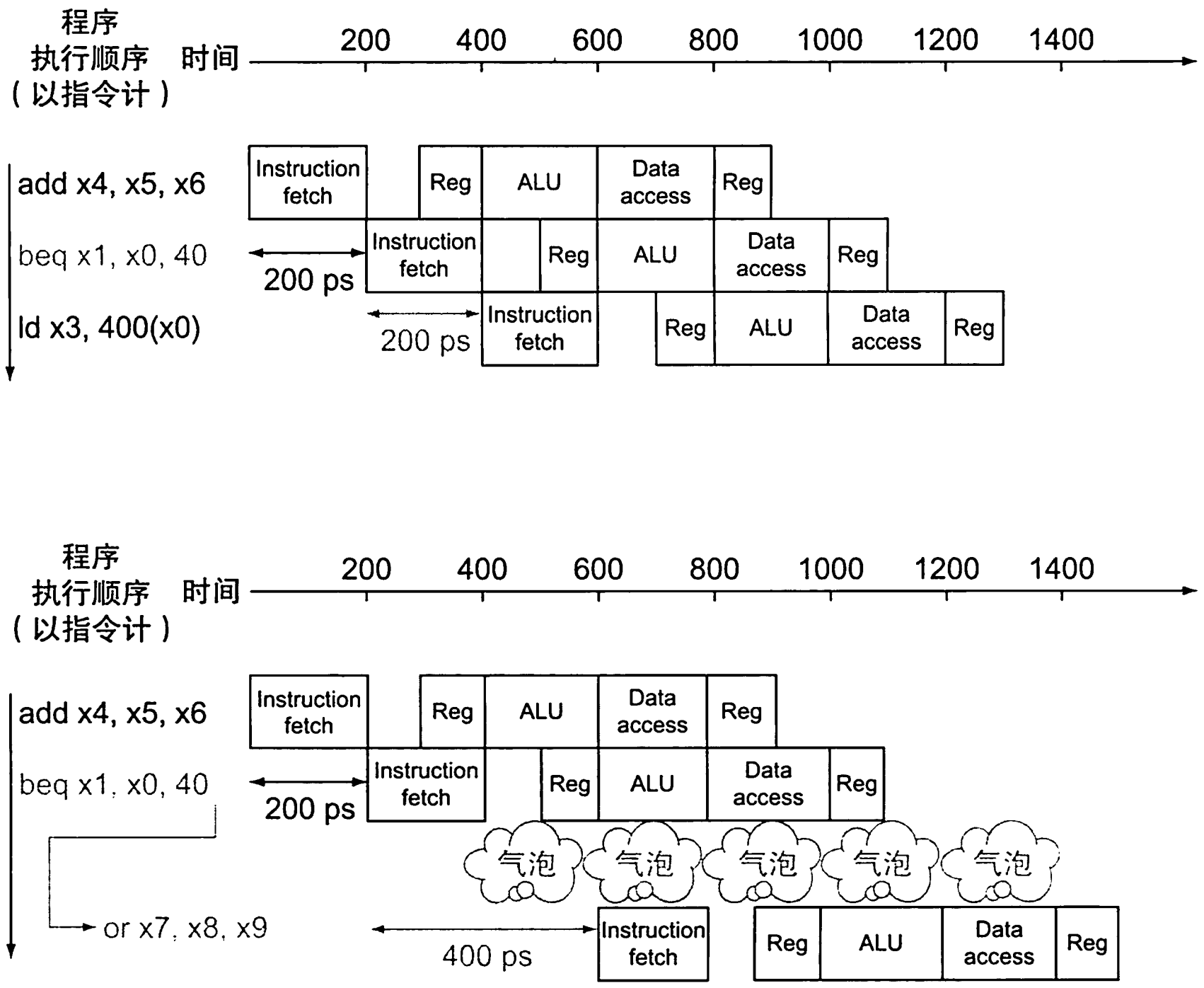


图 4-30　预测分支不发生是一种控制冒险的解决方案。上图表示分支不发生时的流水线，下图表示分支发生时的流水线。正如图 4-29 说明的那样，这种插入气泡的方式实际是一种简化的方法，至少对紧跟分支指令的下一个时钟周期而言是这样。4.8 节将介绍其中细节

这种分支预测方法依赖于始终不变的行为，没有考虑到特定分支指令的特点。与之形成鲜明对比的是，动态硬件预测器根据每个条件分支指令的行为进行预测，并在程序生命周期内可能改变条件分支的预测结果。对于洗衣例子，使用动态预测方法，一名店员查看队服的污浊程度并预测洗衣设置，同时根据最近的成功预测调整下一次的预测。

动态预测的一种常用实现方法是保存每个条件分支是否发生分支的历史记录，然后根据最近的过去行为来预测未来。正如我们将看到的，历史记录的数量和类型足够多时，动态分支预测器的正确率超过 90%（见 4.8 节）。当预测错误时，流水线控制必须确保预测错误的条件分支指令之后的指令执行不会生效，并且必须从正确的分支地址处重新启动流水线。在洗衣例子中，必须停止接受新的任务，以便可以重新启动预测错误的任务。

如同其他解决冒险的方案一样，较长的流水线会恶化预测的性能，并增加预测错误的代价。4.8 节更详细地介绍了控制冒险的解决方案。

详细阐述　控制冒险的第三种解决方法称为延迟决定（delayed decision）。在洗衣例子中，每当需要做出有关洗衣的决定时，只需在等待足球队服被烘干的同时，向洗衣机中放入一批非足球队服的衣服。只要有足够多不受决定影响的脏衣服，这个方案就可以正常工作。

在计算机中这种方法被称为延迟转移，也就是 MIPS 架构实际使用的解决方案。延迟转移顺序执行下一条指令，并在该指令后执行分支。由于汇编器可以自动排序指令，使得分支指令的行为达到程序员的期望，所以这个过程对 MIPS 汇编语言程序员来说不可见。MIPS

软件会在延迟转移指令的后面放一条不受该分支影响的指令，并且发生转移的分支指令会改变这条安全指令后的指令地址。在图 4-29 中，分支指令前的 add 指令不影响转移，所以可以把它移动到分支指令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，所以很少有处理器使用超过一个时钟周期的延迟转移。对于较长的分支延迟，通常使用基于硬件的分支预测。

4.5.3 总结

流水线技术是一种在顺序指令流中开发指令间并行性的技术。与多处理器编程相比，其优点在于它对程序员是不可见的。

在接下来的几节中，我们将通过 4.4 节 RISC-V 指令子集的单周期实现及其简化的流水线实现，来介绍流水线的相关概念。接着着眼于流水线所带来的问题以及流水线在典型情况下可获得的性能提升。

如果想了解更多软件和流水线对性能的影响，你现在有足够的背景可以跳到 4.10 节。4.10 节介绍高级流水线的概念，如超标量和动态调度。4.11 节介绍最新微处理器的流水线。

或者，如果想深入了解如何实现流水线和如何处理冒险，可以继续阅读后面几节。4.6 节介绍流水线数据通路和基本控制的设计。在此基础上，你可以在 4.7 节中学习前递和停顿的实现。紧接着 4.8 节介绍控制冒险的解决方案，4.9 节中介绍如何处理例外。

理解程序性能 除存储系统以外，流水线的有效操作是决定处理器 CPI 及其性能的最重要因素。正如我们将在 4.10 节看到的那样，现代多发射流水线处理器的性能是复杂的，因为它不仅仅存在简单流水线处理器中出现的问题。不管怎样，结构冒险、数据冒险和控制冒险在简单的流水线和更复杂的流水线处理器中都很重要。

对于现代流水线而言，结构冒险通常出现在浮点单元周围，而浮点单元可能不是完全流水线化的。而控制冒险通常出现在定点程序中，因为其中条件分支指令出现的频率更高，也更难预测。数据冒险在定点和浮点程序中都可能成为性能瓶颈。浮点程序中的数据冒险通常更容易处理，因为其中条件分支指令的频率更低并且存储访问更规则，这使编译器能够调度指令以避免冒险。而由于定点程序的存储器访问更不规则且包含大量指针，实现这样的优化更困难。正如我们将在 4.10 节中看到的那样，有很多编译器和基于硬件的技术通过调度减少数据间的依赖。

重点 流水线增加了可同时执行的指令数目以及指令开始和结束的速率。流水线并不能减少执行单条指令所需时间，即延迟（latency）。例如，一个五级流水线仍然需要五个时钟周期才能完成一条指令。用第 1 章中使用的术语来描述就是流水线提高了指令吞吐率，而不是减少了单条指令的执行时间或延迟。

延迟（流水线）：流水线的阶段数，或执行过程中两条指令间的阶段数。

对于流水线设计者来说，指令系统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。分支预测和前递能够在保证得到正确结果的前提下提高计算机性能。

自我检测 对于下面的每个代码序列，说明它是否必须停顿，或者只使用前递就可以避免停顿，或者既不需要停顿也不需要前递就可以执行。

指令1	指令2	指令3
ld　x10, 0(x10)	add　x11, x10, x10	addi　x11, x10, 1
add　x11, x10, x10	addi　x12, x10, 5	addi　x12, x10, 2
	addi　x14, x11, 5	addi　x13, x10, 3
		addi　x14, x10, 4
		addi　x15, x10, 5

4.6　流水线数据通路和控制

图 4-31 显示了 4.4 节中提到的单周期数据通路，并且标识了流水线阶段。将指令划分成五个阶段意味着五级流水线，还意味着在任意单时钟周期里最多执行五条指令。相应的，我们必须将数据通路划分成五个部分，将每个部分用对应的指令执行阶段来命名：

眼中所看到的东西比实际上要复杂。
Tallulah Bankhead, remark to Alexander Woollcott, 1922

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

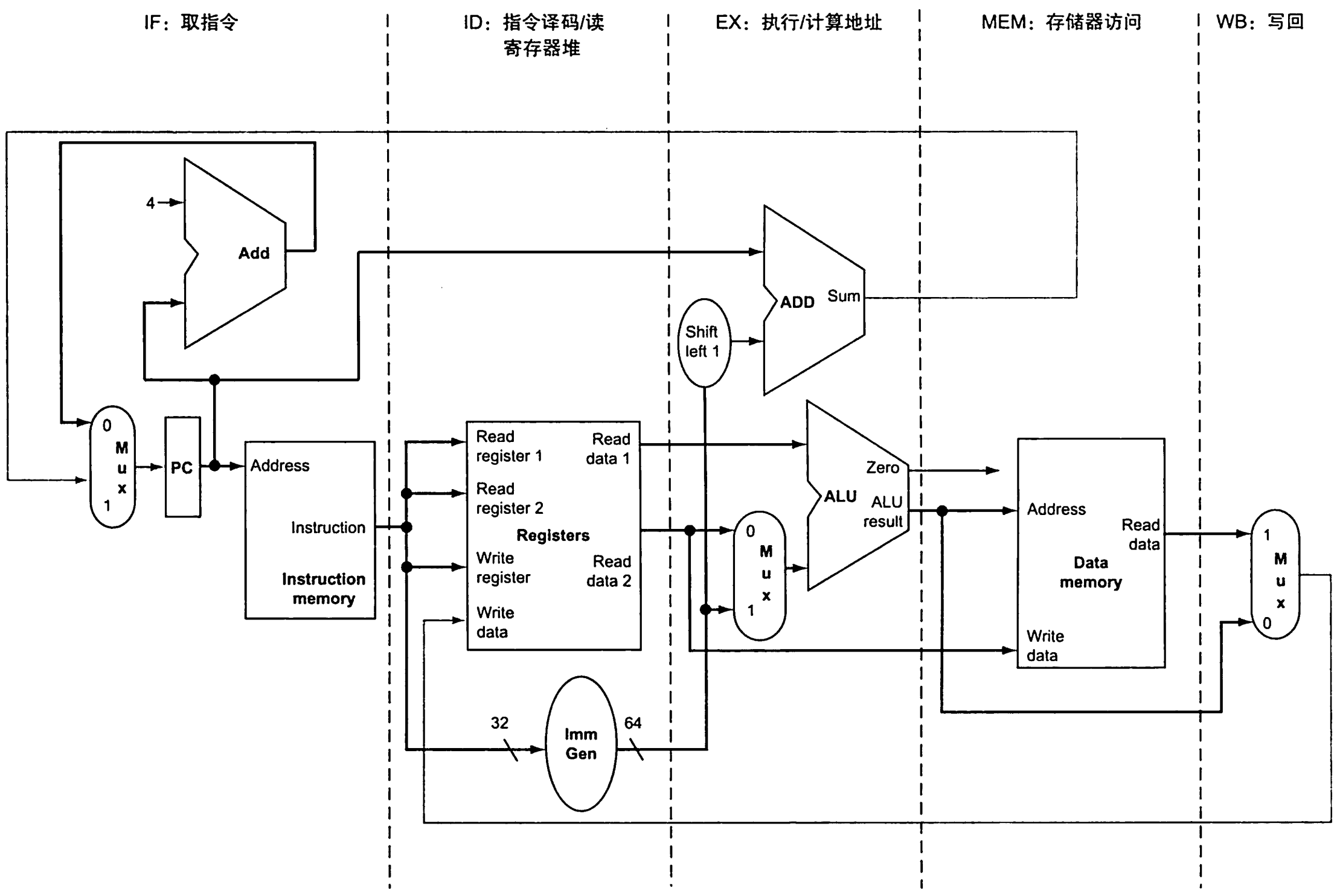


图 4-31　4.4 节中的单周期数据通路（与图 4-17 类似）。指令执行的每一步都从左至右地映射到数据通路中。唯一的例外是 PC 更新与写回的步骤（在图中用灰色表示），以上步骤发送 ALU 运算结果或存储中的数据到左侧，写入寄存器堆中（通常我们使用灰线表示控制，但在这里表示数据通路）

在图 4-31 中，这五个部分与图中数据通路的绘制方式是对应的，指令和数据通常随着执行过程从左到右依次通过这五个阶段。再回到我们的洗衣类比，在通过工作线路时衣服依次被清洁、烘干和整理，同时永远不会逆向移动。

然而，在从左到右的指令流动过程中存在两个特殊情况：

- 在写回阶段，它将结果写回位于数据通路中段的寄存器堆中。
- 在选择下一 PC 值时，在自增 PC 值与 MEM 阶段的分支地址之间进行选择。

从右到左的数据流向不会对当前的指令造成影响，这种反向的数据流动只会影响流水线中的后续指令。需要注意的是，第一种特殊情况会导致数据冒险，第二种会导致控制冒险。

一种表示流水线数据通路如何执行的方法是假定每一条指令都有独立的数据通路，然后将这些数据通路放在同一时间轴上来表示它们之间的关系。图 4-32 通过在公共时间轴上显示私有数据通路来表示图 4-27 中的指令的执行。我们使用图 4-31 中的数据通路的格式来表示图 4-32 中的关系。

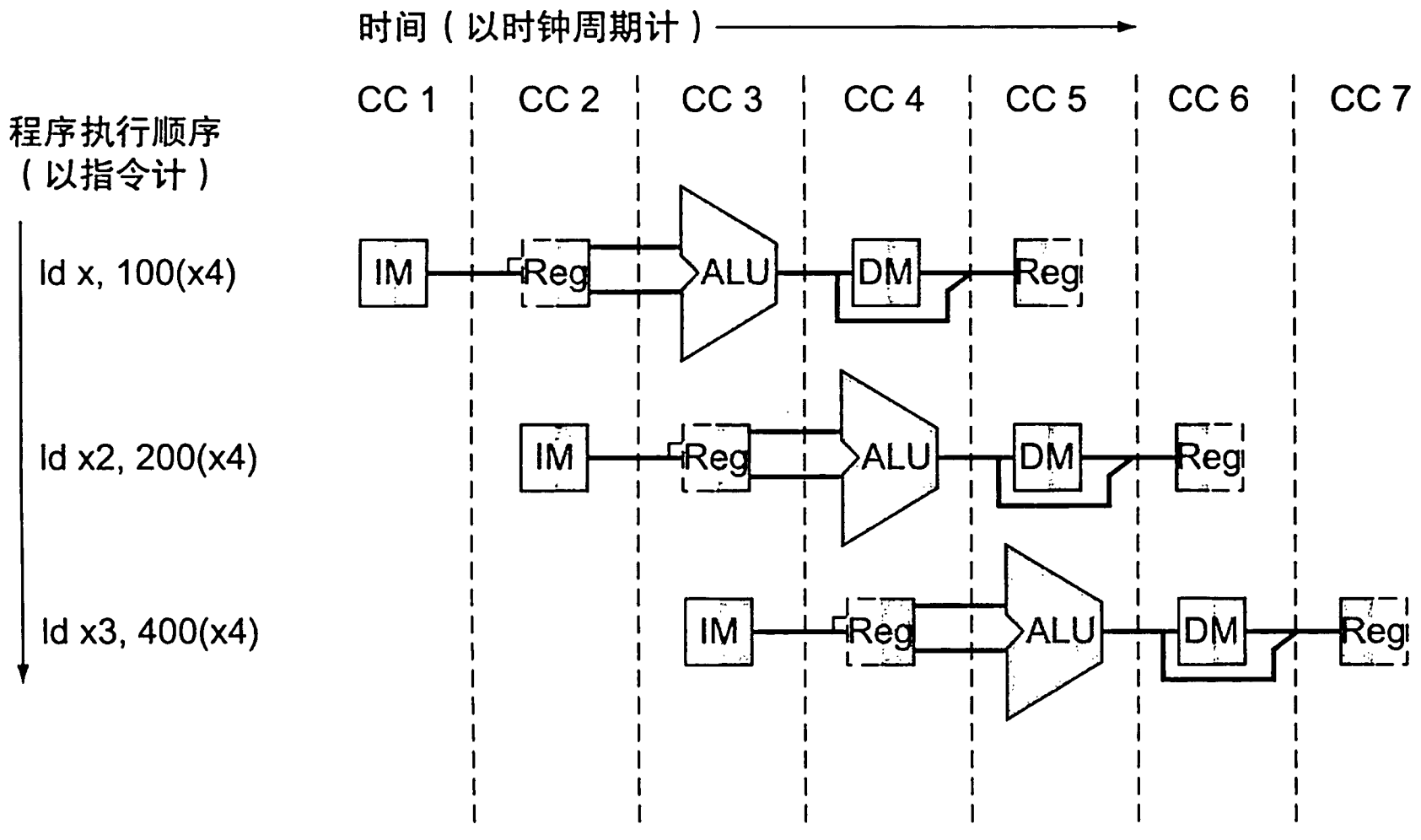


图 4-32 使用图 4-31 中的单周期数据通路执行的指令，假设指令以流水线方式执行。类似于图 4-26~ 图 4-28，该图假设每条指令都有自己独立的数据通路，并且根据使用情况将相应的部分涂上阴影。与那些图不同的是，每个阶段都被该阶段使用的物理资源标记，分别对应于图 4-31 中数据通路相应的部分。IM 表示指令寄存器和取值阶段的 PC，Reg 表示指令译码 / 寄存器读取阶段（ID）的寄存器堆和符号扩展单元，等等。为了保持正确的时序，这种形式化的数据通路将寄存器堆划分成两个逻辑部分：寄存器读取阶段（ID）的寄存器读和写回（WB）阶段的寄存器写。这种复用被表示为：在 ID 阶段，当寄存器堆没有被写入时，使用虚线绘制未被着色的寄存器堆的左半部分；在 WB 阶段，当寄存器堆没有被读取时，使用虚线绘制未被着色的寄存器堆的右半部分。与前文一致，我们假设寄存器堆是在时钟周期的前半部分写入的，在时钟周期的后半部分被读取

图 4-32 似乎表明三条指令需要三条数据通路，但事实上，我们可以通过引入寄存器保存数据的方式，使得部分数据通路可以在指令执行的过程中被共享。

举例来说，如图 4-32 所示，指令存储器只在指令的五个阶段中的一个阶段被使用，而在其他四个阶段中允许被其他指令共享。为了保留在其他四个阶段中的指令的值，必须把从指令存储器中读取的数据保存在寄存器中。类似的理由适用于每个流水线阶段，所以我们必

须将寄存器放置在图 4-31 中每个阶段之间的分隔线上。再回到洗衣例子中，我们会在每两个步骤之间放置一个篮子，用于存放为下一步所准备的衣服。

图 4-33 显示了流水线数据通路，其中的流水线寄存器被高亮表示。所有指令都会在每一个时钟周期里从一个流水线寄存器前进到下一个寄存器中。寄存器的名称由两个被该寄存器分开的阶段的名称来命名。例如，IF 和 ID 阶段之间的流水线寄存器被命名为 IF/ID。

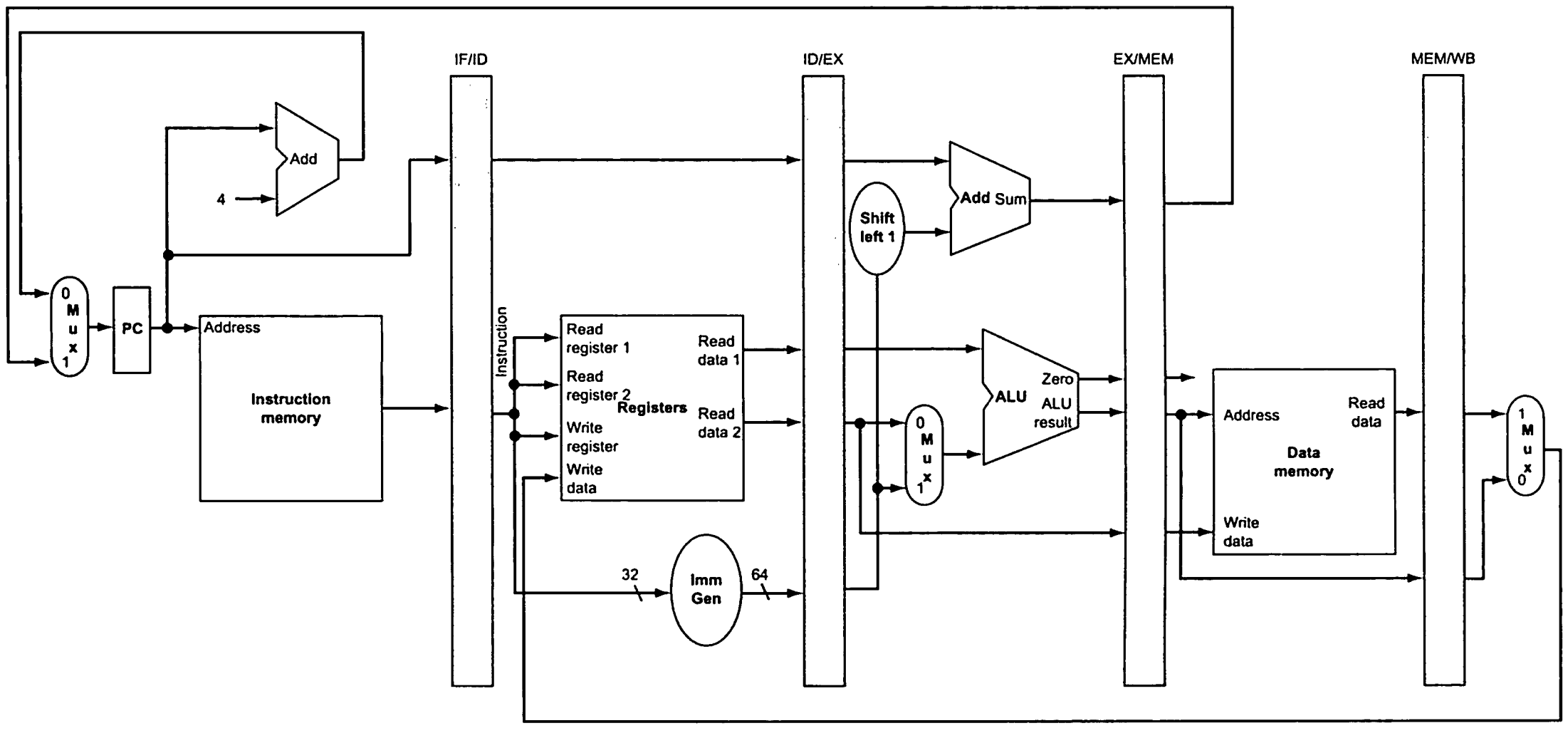


图 4-33　图 4-31 数据通路的流水线版本。在图中用灰色表示的流水线寄存器将流水线的各阶段分开。它们被标记为被它们所分开的阶段，例如，第一个流水线寄存器被标记为 IF/ID，因为它将取值和指令译码阶段分开。寄存器的位宽必须足够大以存储通过它们的所有数据。例如，IF/ID 寄存器的位宽必须为 96 位，因为它需要同时存储从存储器中提取出的 32 位指令以及自增的 64 位 PC 地址。我们将在本章中逐渐增加这些寄存器的位宽，不过目前，其他三个流水线寄存器的位宽分别为 256 位、193 位和 128 位

需要注意的是，在写回阶段的最后没有流水线寄存器。所有的指令都必须更新处理器中的某些状态，如寄存器堆、存储器或 PC 等，因此，单独的流水线寄存器对于已经被更新的状态来说是多余的。例如，加载指令将它的结果放入 32 个寄存器中的一个，此后任何需要该数据的指令只需要简单地读取相应的寄存器即可。

当然，每条指令都会更新 PC，无论是通过自增还是通过将其设置为分支目标地址。PC 可以被看作一个流水线寄存器：它给流水线的 IF 阶段提供数据。不同于图 4-33 中被标记阴影的流水线寄存器，PC 是可见体系结构状态的一部分，在发生例外时，PC 中的内容必须被保存，而流水线寄存器中的内容则可以被丢弃。在洗衣的例子中，你可以将 PC 看作在清洗步骤之前盛放脏衣服的篮子。

为了说明流水线的工作原理，在本章中，我们使用一系列图片来演示这一系列操作。这些额外的内容看似使你要花费更多时间去理解，但是不要害怕，这些图片比它们看上去容易理解，因为你可以通过对比来观察每个时钟周期中发生的变化。4.7 节描述了当指令流水线中存在数据冒险时的情况，现在请先忽略它们。

图 4-34 ～图 4-37 是我们的第一个序列，显示了加载指令在通过流水线的五个阶段时数据通路高亮的活动部分。我们首先展示加载指令是因为它在五个阶段中都是活跃的。如

图 4-26 ～图 4-28 所示，当寄存器或存储器被读取时，我们高亮显示它们的右半部分；当它们被写入时，我们高亮显示它们的左半部分。

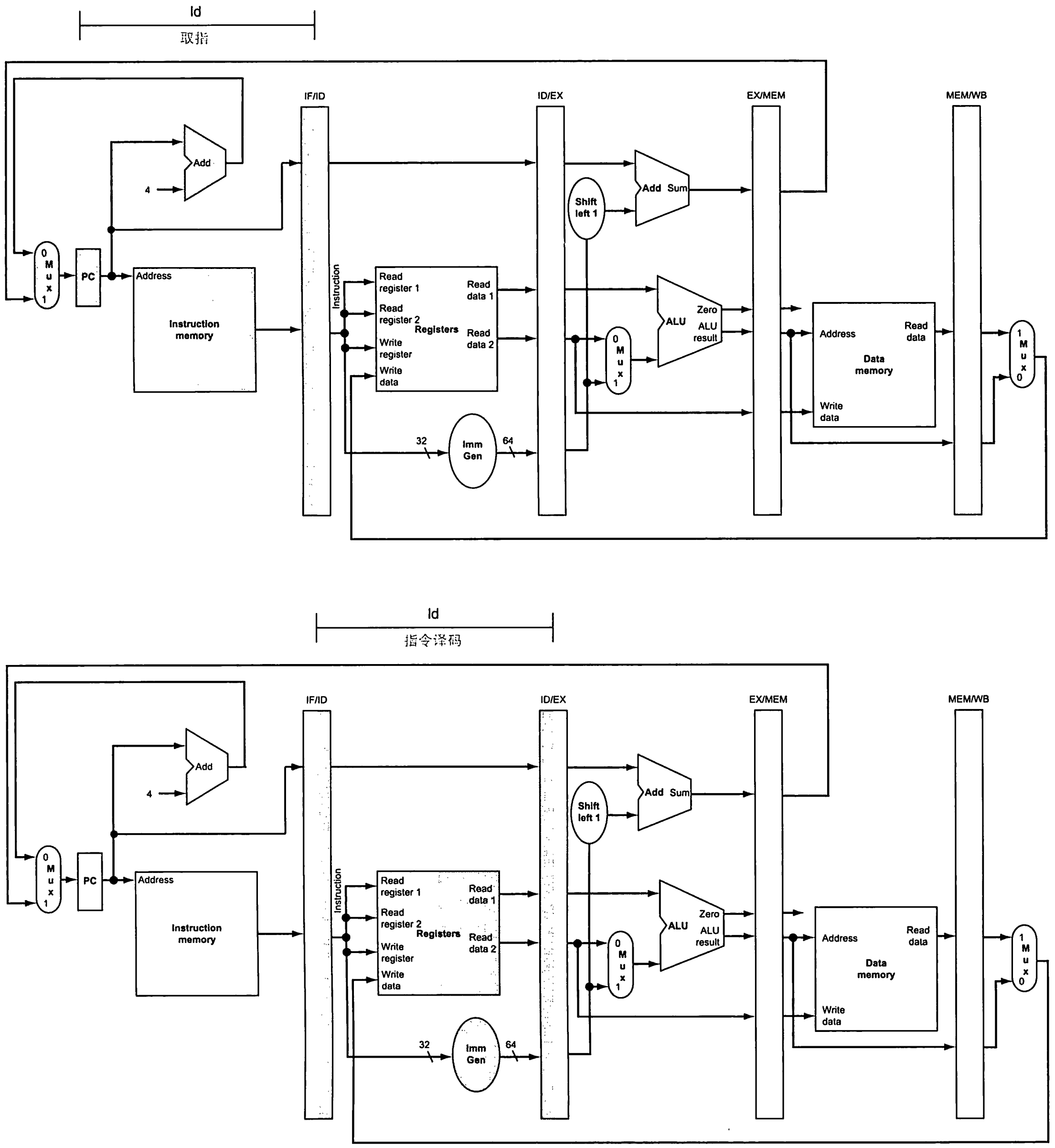


图 4-34 IF 和 ID：一条指令在指令流水线中的第一和第二步，图 4-33 中数据通路的活跃部分被高亮显示。这种高亮的表示方法与图 4-26 中的表示相同。如 4.2 节所述，读写寄存器时不会发生混乱，这是因为寄存器中的内容仅在时钟边沿上发生变化。尽管在阶段二中加载指令只需要寄存器 1 中的值，但是处理器此时并不知道当前是哪一条指令正在被译码，因此处理器将符号扩展后的 16 位常量以及两个寄存器中的值都存入 ID/EX 流水线寄存器中。我们并不一定需要全部的这三个操作数，但是保留全部三个操作数可以简化控制

我们在图中标识出指令 ld 在每一副图中的活跃流水阶段。这五个阶段如下：

1. 取指：图 4-34 的顶端描绘了使用 PC 中的地址从存储器中读取指令，然后将指令放入 IF/ID 流水线寄存器中。PC 中的地址自增 4，然后写回 PC，以为下一时钟周期做准备。这个 PC 值也保存在 IF/ID 流水线寄存器中，以备后续的指令使用（例如 beq）。计算机并不知道当前正在提取的是哪一种指令，因此它必须为任何一种指令做好准备，并且将所有可能有用的信息沿流水线传递出去。

2. 指令译码和读寄存器堆：图 4-34 的底部显示了 IF/ID 流水线寄存器的指令部分，该指令提供一个 64 位符号扩展的立即数字段，以及两个将要读取的寄存器编号。所有这三个值都与 PC 地址一起存储在 ID/EX 流水线寄存器中。在这里我们再次向右传递在之后的时钟周期里指令可能用到的所有信息。

3. 执行或地址计算：图 4-35 显示了加载指令从 ID/EX 流水线寄存器中读取一个寄存器的值和一个符号扩展的立即数，并且使用 ALU 部件将它们相加，它们的和被存储在 EX/MEM 流水线寄存器中。

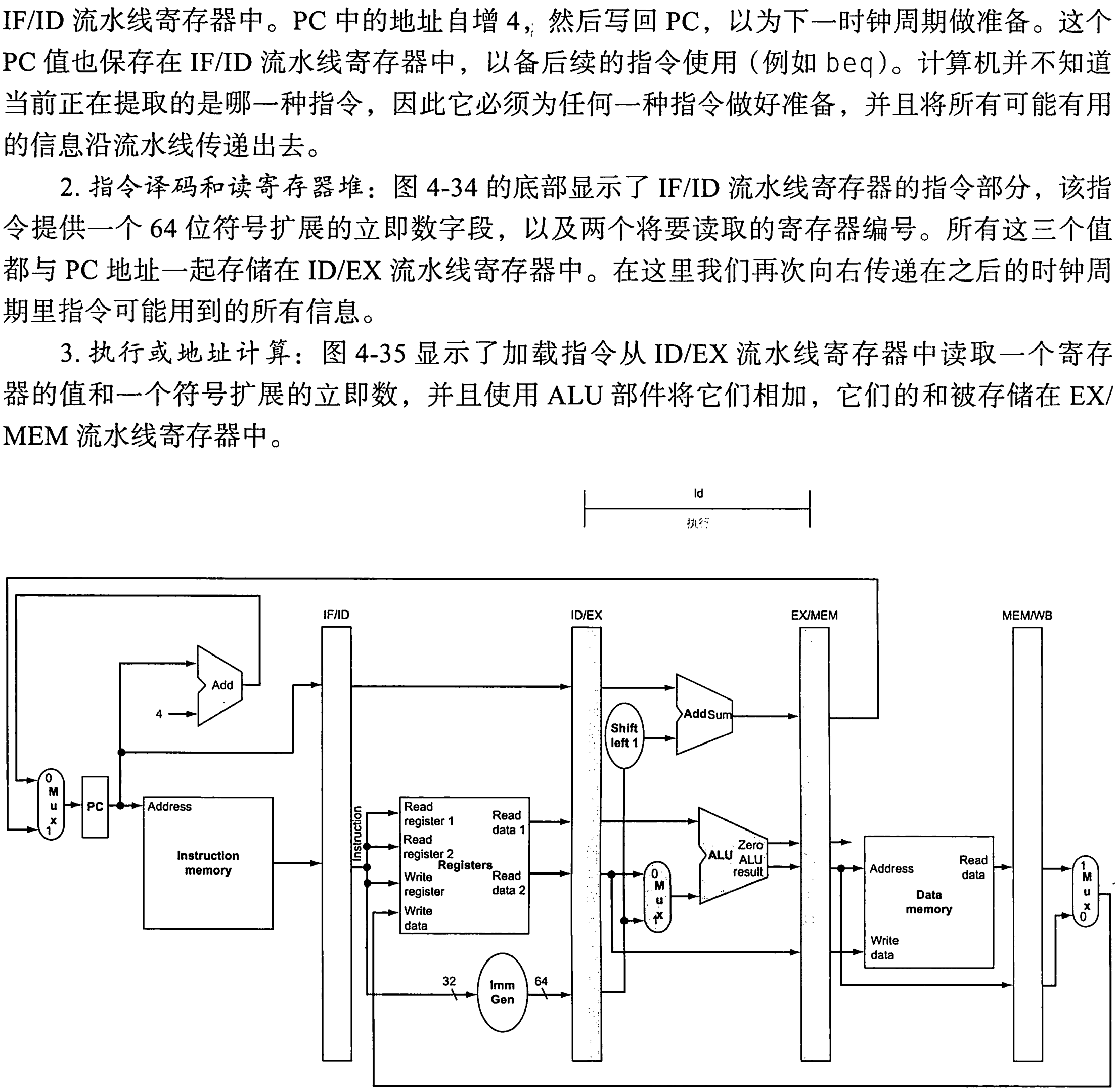


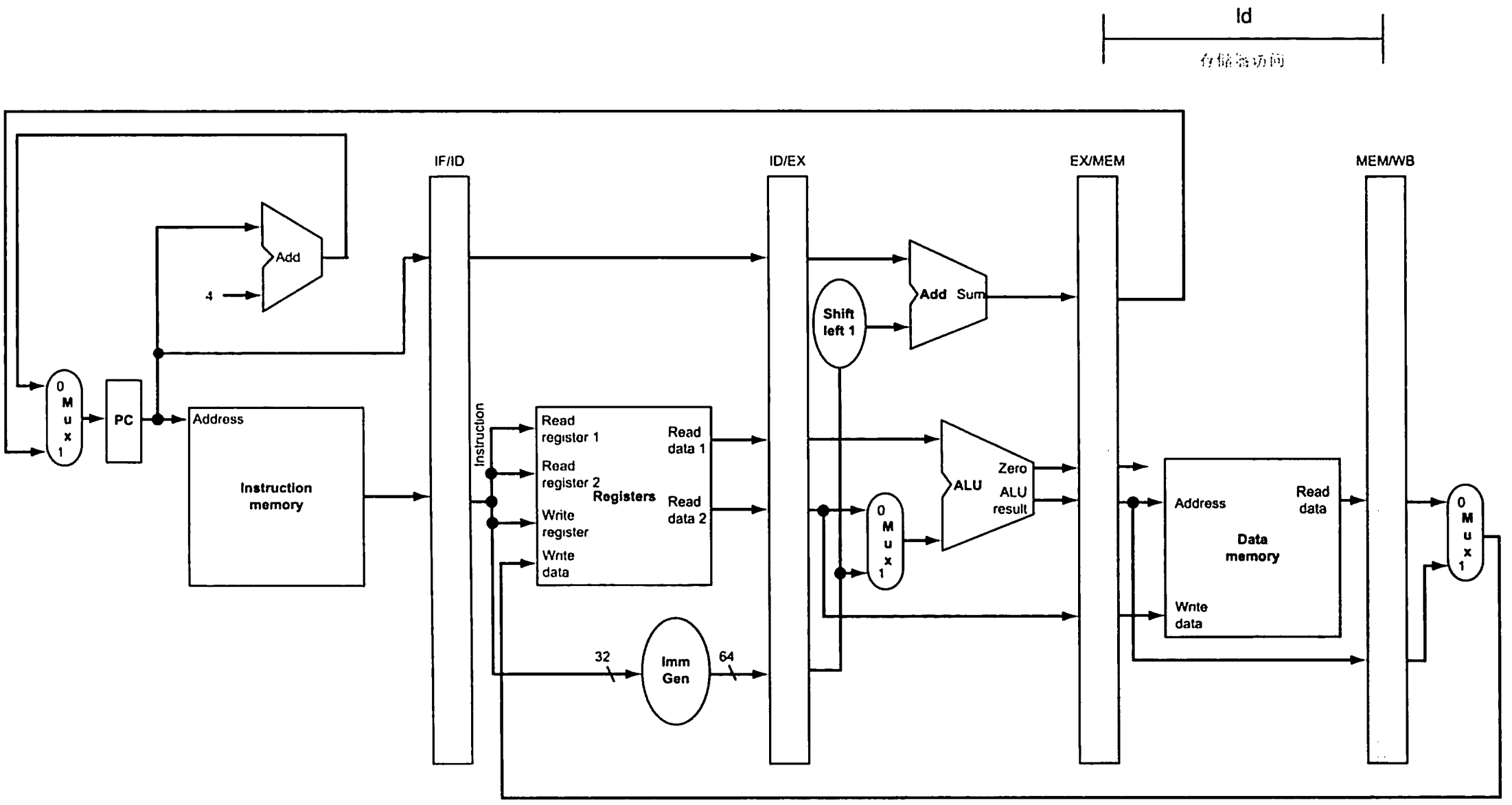
图 4-35　EX：加载指令在指令流水线中的第三个阶段，高亮显示该阶段中使用的图 4-33 中的数据通路部分。将寄存器中的值与符号扩展后的立即数相加，并将和放入 EX/MEM 流水线寄存器中

4. 存储器访问：图 4-36 的顶部显示了加载指令使用来自 EX/MEM 流水线寄存器中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 4-36 的底部显示了最后一步：从 MEM/WB 流水线寄存器中读取数据，并将它写入图中间的寄存器堆中。

对加载指令的演示表明，在后续流水线阶段所需的任何信息，都需要通过流水线寄存器传递。存储指令的执行过程也与此过程类似，需要将信息传递给后续阶段。下面是存储指令的五个执行步骤：

1. 取指：使用 PC 中的地址从存储器中读取指令，然后将其放入 IF/ID 流水线寄存器中。该阶段发生在指令被识别之前，因此图 4-34 的顶端同时适用于加载和存储指令。



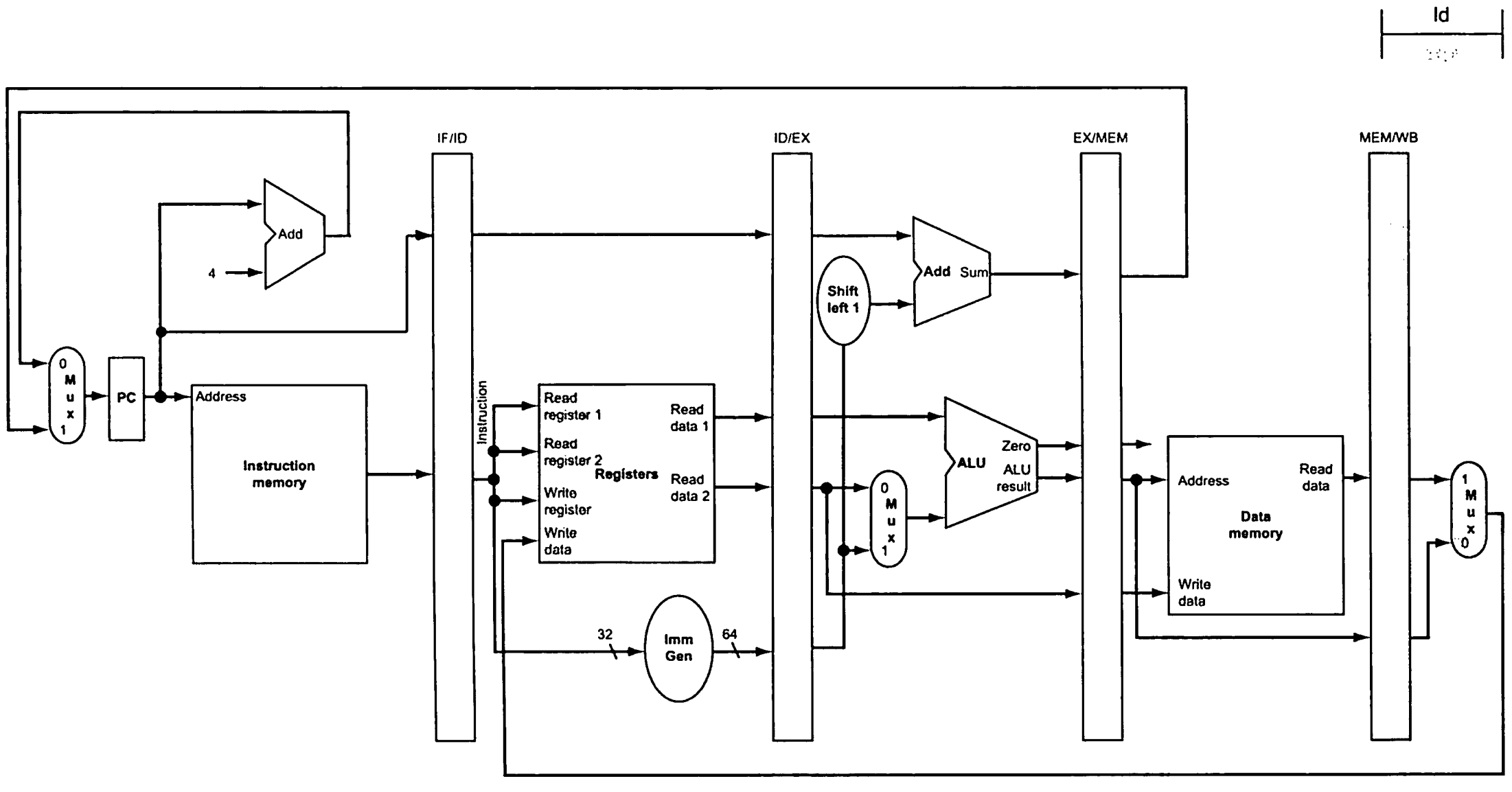


图 4-36 MEM 和 WB：加载指令在指令流水线中的第四、第五阶段，高亮显示该阶段中使用的图 4-33 中的数据通路部分。利用 EX/MEM 流水线寄存器中的地址读取数据存储器，之后将该数据存储到 MEM/WB 流水线寄存器中。接下来，数据从 MEM/WB 流水线寄存器中被读取，然后写入数据通路中间的寄存器堆中。注意：这个设计中存在一个错误，将在图 4-39 中修复

2. *指令译码和读寄存器堆*：IF/ID 流水线寄存器中的指令提供了用于读取寄存器的两个寄存器编号以及一个符号扩展的立即数。这三个 64 位的值都存储在 ID/EX 流水线寄存器中。图 4-34 的底端既可以表示加载指令，也可以表示存储指令的第二个流水阶段。因为此时还不知道指令的类型，所以所有的指令都会执行这两个阶段。（虽然存储指令使用 rs2 字段读取本流水线阶段中的第二个寄存器，但是该流水线图中并未显示这个细节，因此我们可以使用

相同的图表。）

3. 指令执行和地址计算：图 4-37 显示了指令流水线中的第三步，有效地址被存放在 EX/MEM 流水线寄存器中。

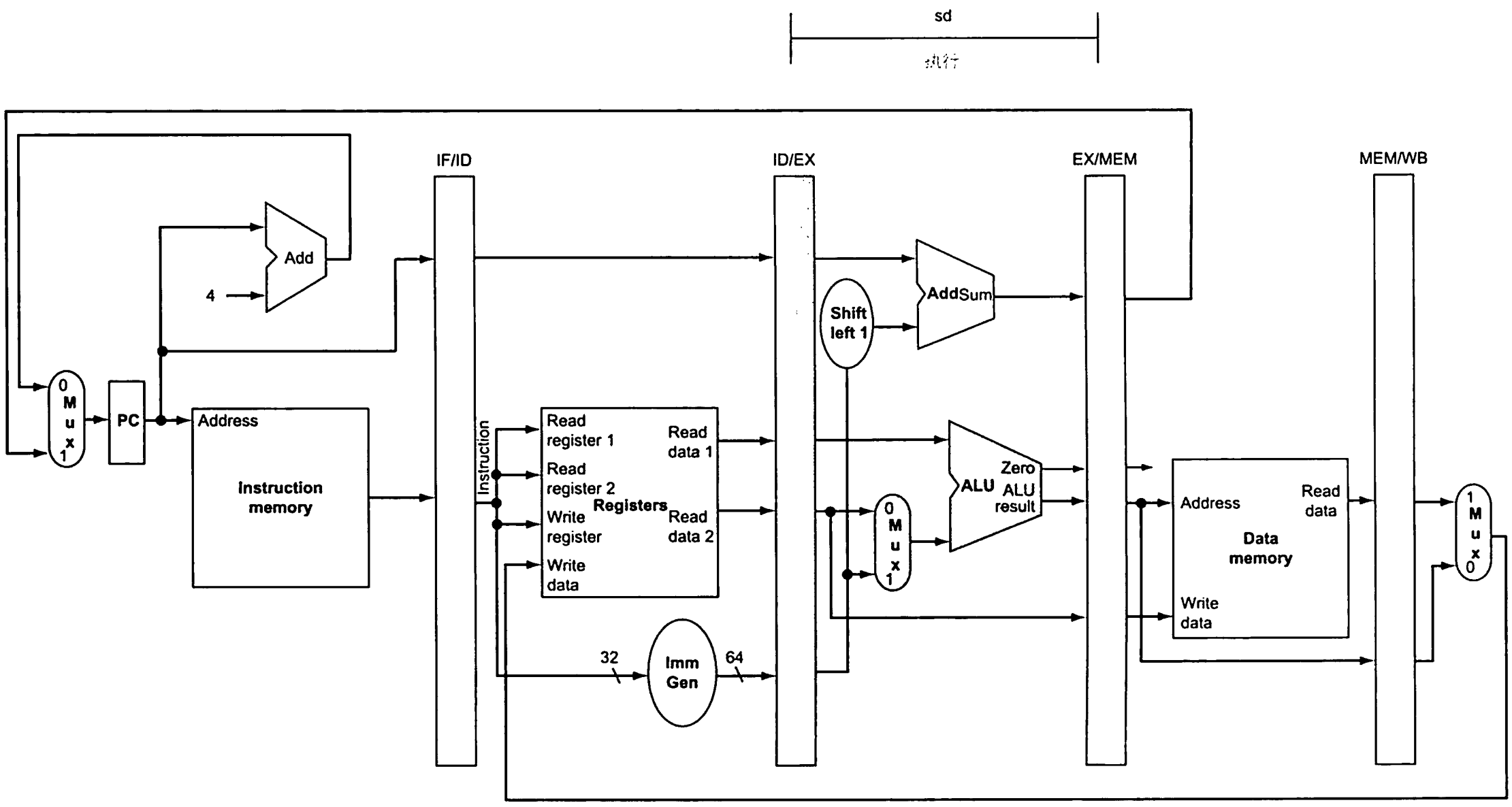


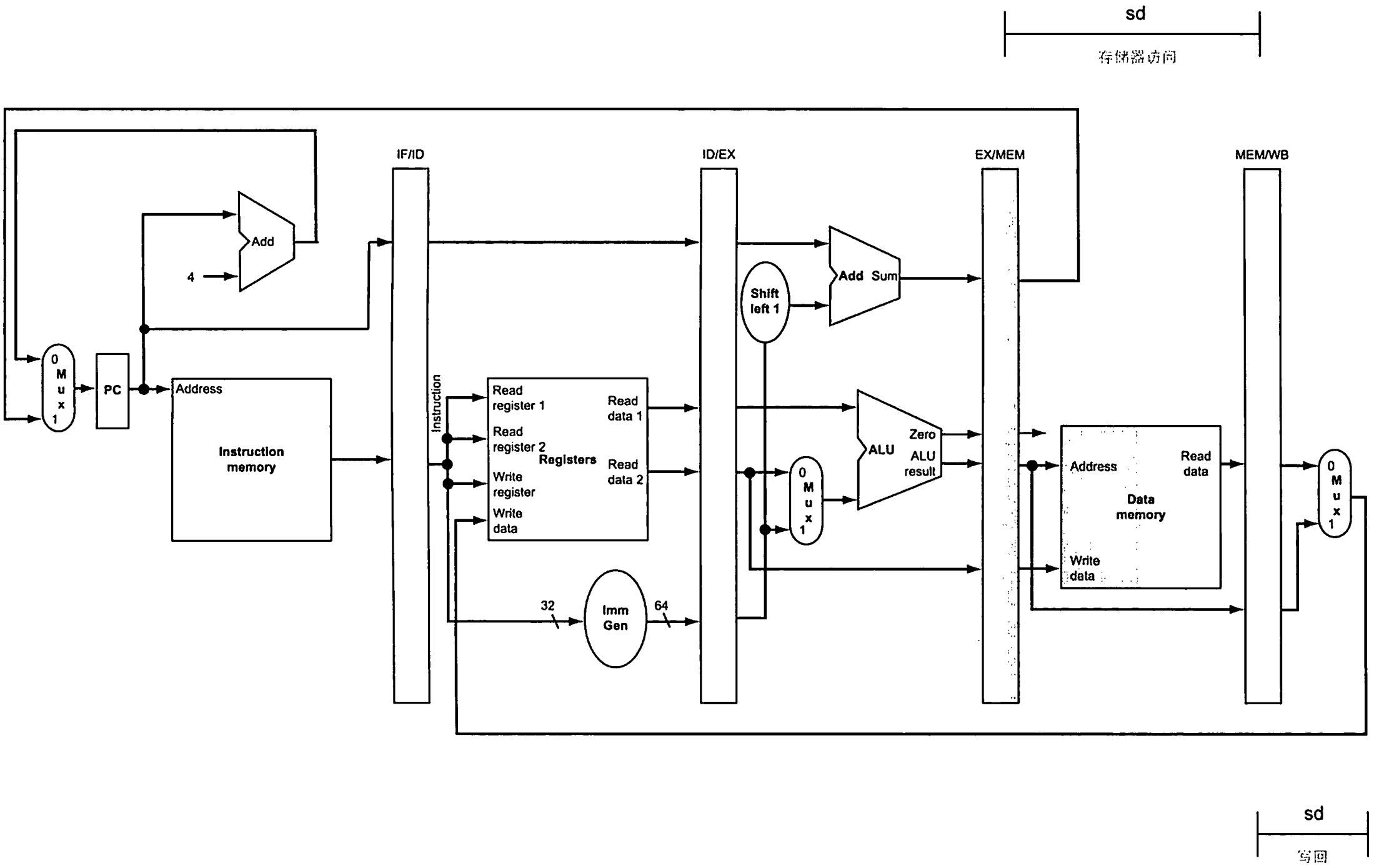
图 4-37　EX：存储指令在流水线中的第三个阶段。不同于图 4-35 中加载指令的第三个流水阶段，第二个寄存器中的值被加载到 EX/MEM 流水线寄存器中以用于下一个阶段。尽管总是将第二个寄存器中的值写入 EX/MEM 流水线寄存器中不会造成任何影响，但为了使流水线更容易被理解，我们仅在存储指令中写第二个寄存器中的值

4. 存储器访问：图 4-38 的顶端显示了正在被写入存储器的数据。需要注意，包含要被存储的数据的寄存器在较早的流水线阶段就已经被读取并存储在 ID/EX 流水线寄存器中。在 MEM 阶段获得这个数据的唯一方法就是在 EX 阶段中将该数据放入 EX/MEM 流水线寄存器中，就像我们将有效地址存储在 EX/MEM 中那样。

5. 写回。图 4-38 的底端显示了存储指令的最后一步。对存储指令来说，在写回阶段不会发生任何事情。由于存储指令之后的每一条指令都已经进入流水线中，所以我们无法加速这些指令。因此，任何指令都要经过流水线中的每一个阶段，即使它在这个阶段没有任何事情要做，因为后续指令已经按照最大速率在流水线中进行处理了。

存储指令再次说明了如果要将相关信息从之前的流水线阶段传递到后续的流水线阶段，就必须将它们放置在流水线寄存器中。否则，当下一条指令进入流水线时，该信息就会丢失。对于存储指令来说，我们需要将在 ID 阶段读取的寄存器信息传递到 MEM 阶段，然后写入存储器中。这些数据最初放置在 ID/EX 流水线寄存器中，之后被传送到 EX/MEM 流水线寄存器中。

其次，加载和存储指令还说明了第二个关键点：在流水线数据通路设计中的每一个逻辑部件（例如指令存储器、寄存器读端口、ALU、数据存储器、寄存器写端口等）只能在单个流水线阶段中被使用，否则就会发生结构冒险（参见 4.5.2 节）。因此，这些部件以及对它们的控制只能与一个流水线阶段相关联。



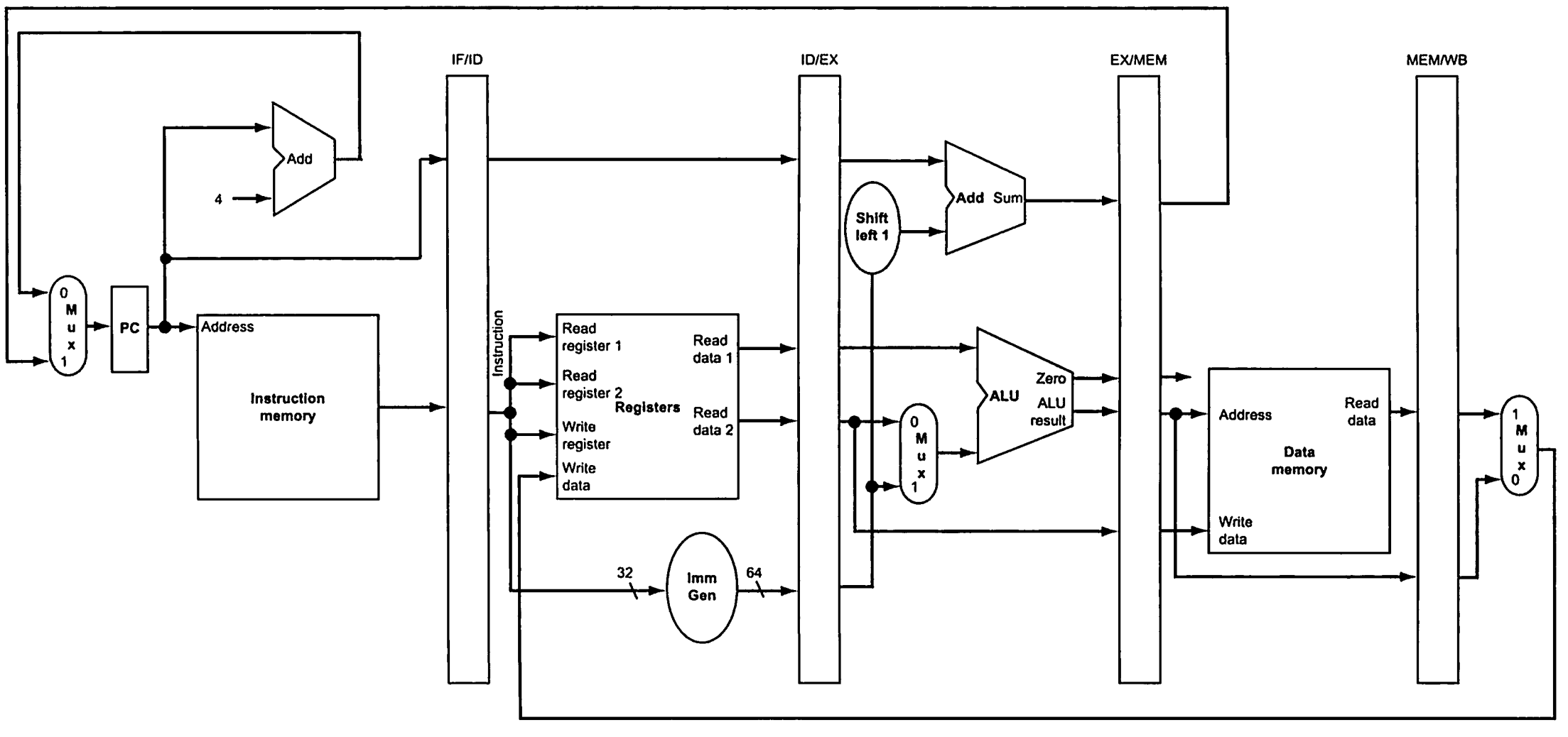


图 4-38 MEM 和 WB：存储指令在流水线中的第四和第五阶段。在第四阶段，利用 EX/MEM 流水线寄存器中的地址读取数据寄存器，并将读取的数据写入到 MEM/WB 流水线寄存器中。一旦数据被写入存储器，存储指令就没有任何事情可做，所以在第五阶段中没有任何事情发生

现在我们就可以发现加载指令设计中的一个错误。你发现了吗？在加载指令流水的 WB 阶段改写了哪个寄存器？更具体地说，此时的寄存器号是哪条指令提供的？ IF/ID 流水线寄存器中的指令提供了写入寄存器编号。但是，这条指令是加载指令之后的指令了（这就是错误所在）。

因此，我们需要在加载指令的流水线寄存器中保留目标寄存器编号。就像存储指令为了 MEM 阶段的使用而将寄存器值从 ID/EX 中传递到 EX/MEM 流水线寄存器中那样，加载指

令需要为了 WB 阶段的使用而将寄存器编号从 ID/EX 通过 EX/MEM 传递到 MEM/WB 流水线寄存器。换一个角度来看，为了共享流水线数据通路，我们需要在 IF 阶段保存读取的指令，因此每个流水线寄存器都要保存当前阶段和后续阶段所需的部分指令信息。

图 4-39 展示了修正后的数据通路的正确版本。将写入寄存器编号先传递到 ID/EX 寄存器中，然后传送到 EX/MEM 寄存器，最后传送到 MEM/WB 寄存器。寄存器编号在 WB 阶段被使用，指定了要写入的寄存器。图 4-40 是修正后数据通路图，它高亮显示了在图 4-34 ～图 4-36 中的加载指令在所有五个流水线阶段中用到的硬件。4.8 节解释了如何使分支指令按照预期的方式工作。

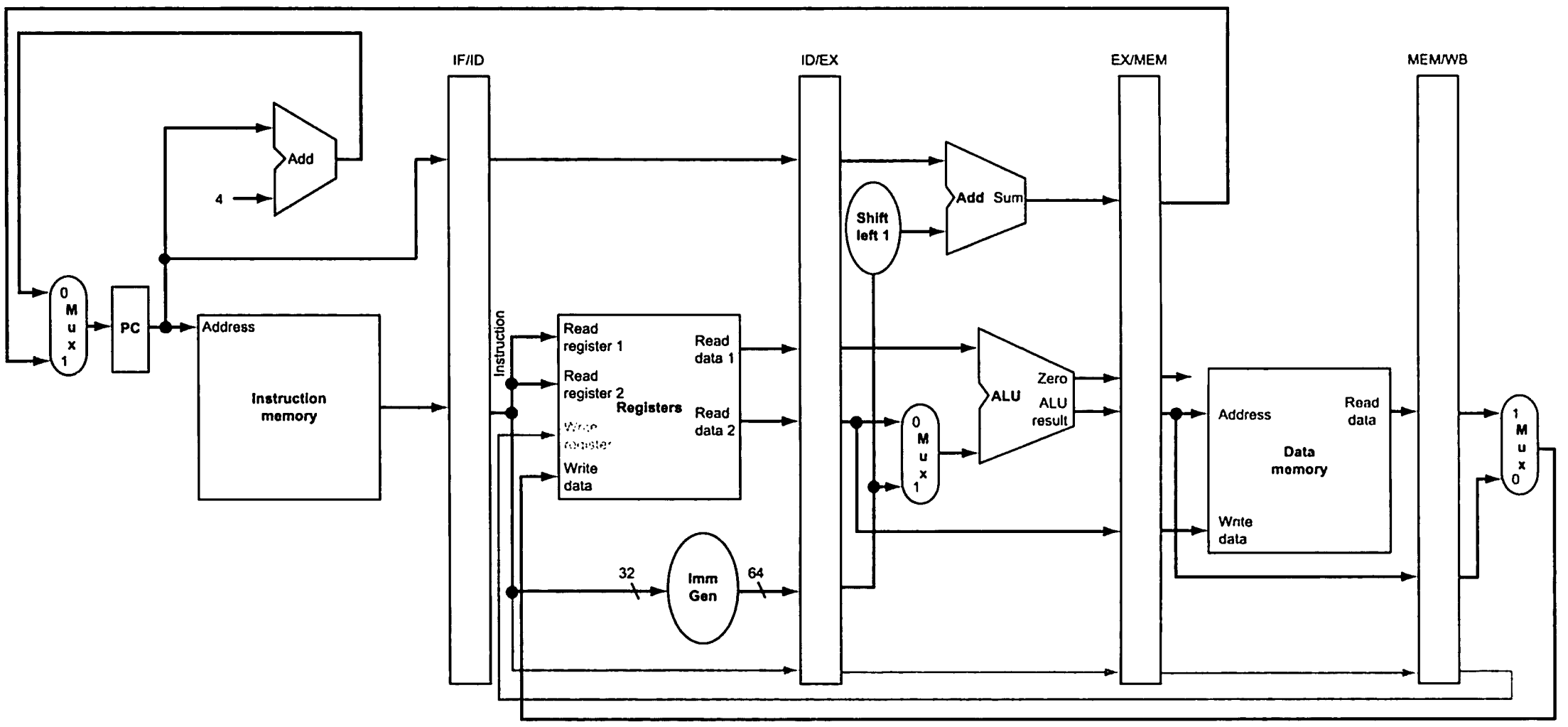


图 4-39　修正后的流水线数据通路，可以正确处理加载指令。写入寄存器编号和数据现在来自 MEM/WB 流水线寄存器。通过在最后三个流水线寄存器中额外添加 5 位，使得这个寄存器编号可以从 ID 流水线阶段开始传递，一直到达 MEM/WB 流水线寄存器。这条新的路径在图中用灰色表示

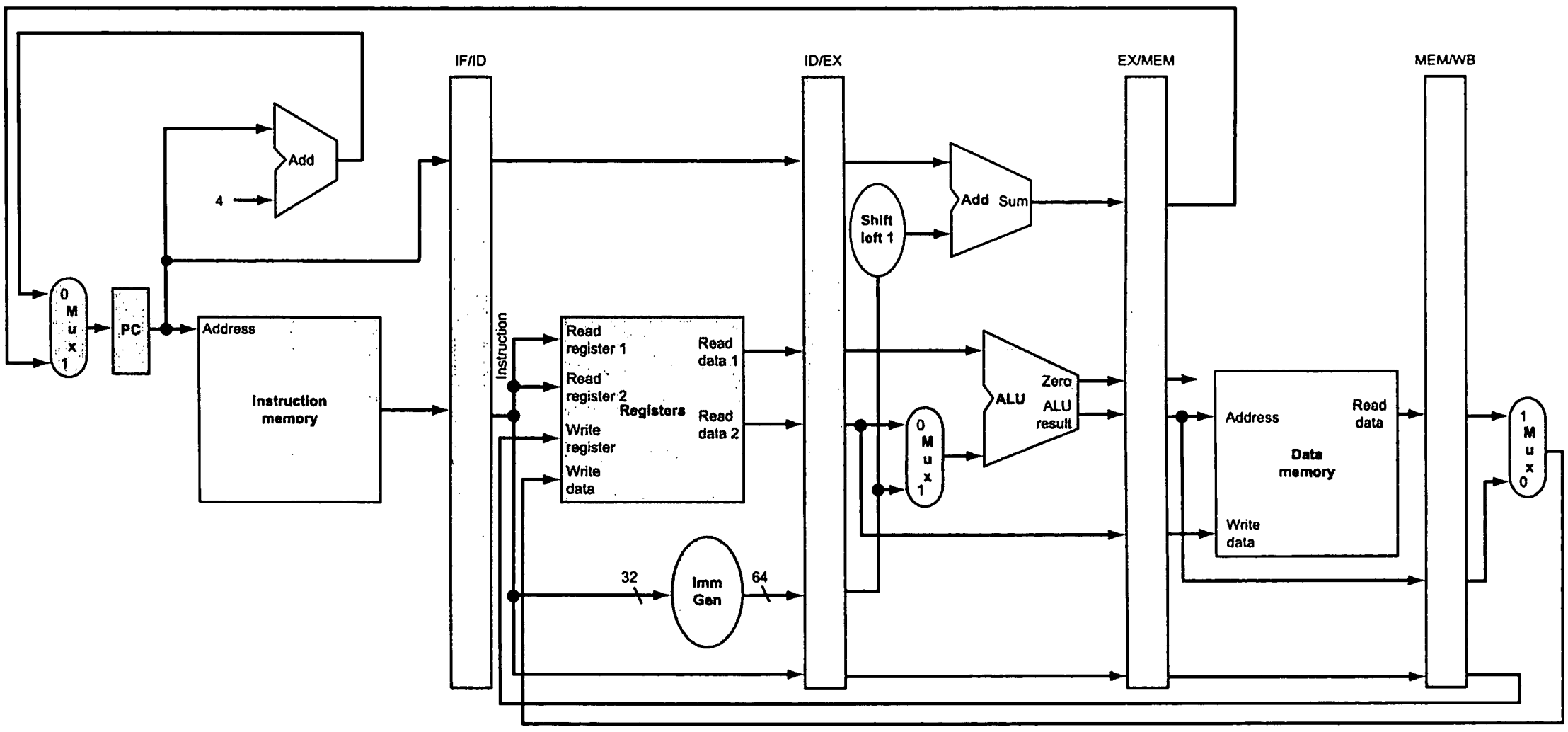


图 4-40　图 4-39 数据路径中用于加载指令的全部五个流水线阶段的部分