4.8　控制冒险

> 也许一千个人对罪恶进行了不痛不痒的批判，才会出现一个人真正地动摇罪恶的根基。
> *Henry David Thoreau, Walden, 1854*

迄今为止，我们只将对冒险的关注局限在算术操作和数据传输中。然而，正如4.5节所示，流水线冒险也包括条件分支。图4-59中画出了一个指令序列，并标明在这个流水线中分支是何时发生的。每个时钟周期都必须进行取值操作以维持流水线，不过在我们的设计中，要等到MEM流水线阶段才可以决定分支是否发生。正如4.5节中所述，这种为了决定正确执行指令所产生的延迟被称为控制冒险或分支冒险，这与我们之前讨论的数据冒险相对应。

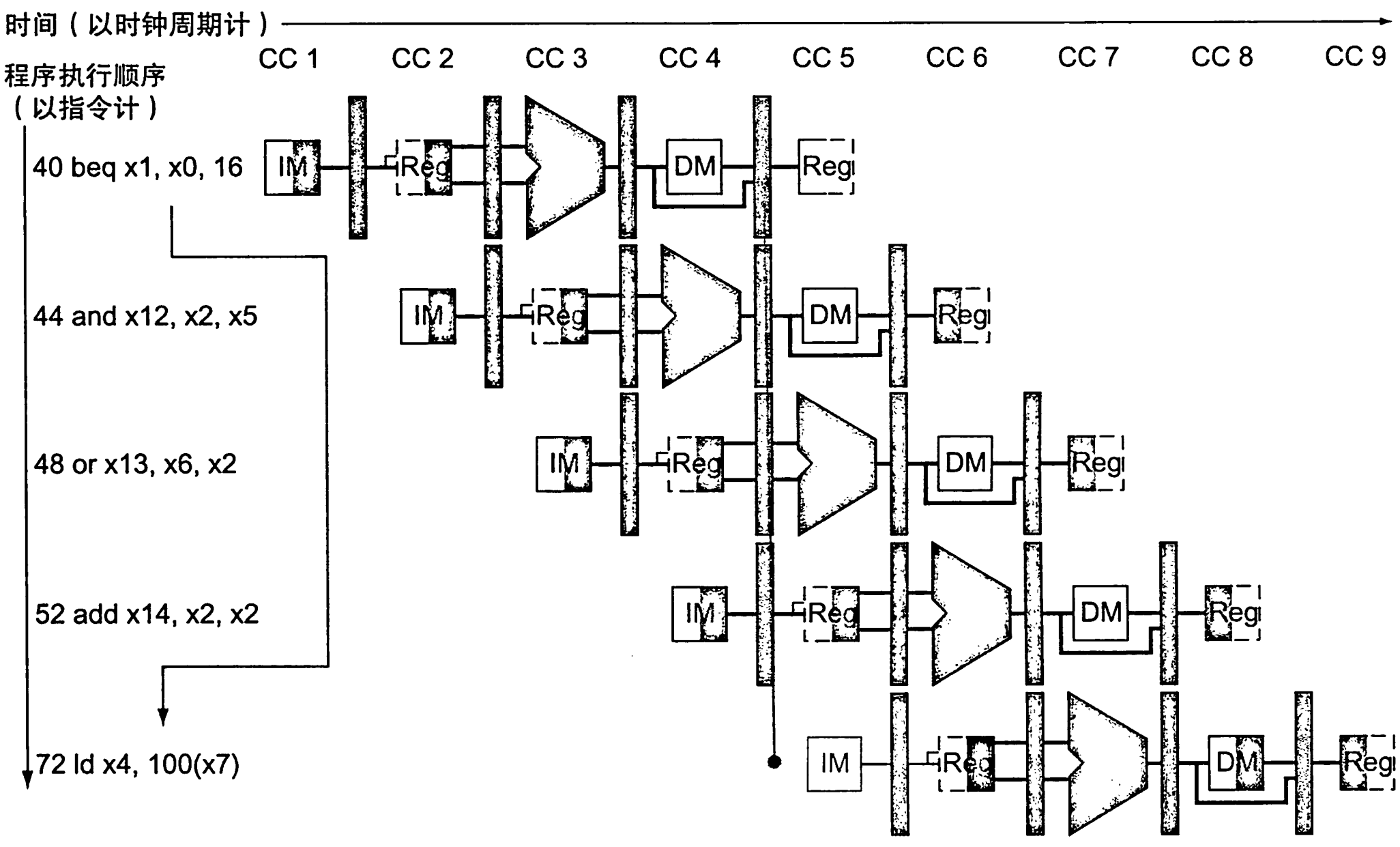


图4-59　分支指令对流水线的影响。指令左边的数字（40、44等）代表指令的地址。因为分支指令在MEM阶段决定是否跳转（也就是图中beq指令在第4个时钟周期内的操作），分支指令后续的三条指令都将被取值并且开始执行。如果不进行干预，这三条后续指令会在beq指令跳转到地址72上的ld指令之前就开始执行（图4-29使用了额外的硬件以减小控制冒险至一个时钟周期，而本图使用的是没有优化的数据通路。）

本节关于控制冒险的篇幅会短于之前叙述数据冒险的小节。这是因为控制冒险相对更好理解，发生的频率也比数据冒险更低，而且相对于数据冒险，现在还没有解决控制冒险的有效手段。因此，我们采用了更简单的描述方案。本节介绍了两种解决控制冒险的方案以及一种提升解决方案性能的优化方法。

4.8.1　假设分支不发生

正如4.5节中所述，阻塞流水线直到分支完成的策略非常耗时。一种提升分支阻塞效率的方法是预测条件分支不发生并持续执行顺序指令流。一旦条件分支发生，已经被读取和译码的指令就将被丢弃，流水线继续从分支目标处开始执行。如果条件分支不发生的概率是50%，同时丢弃指令的代价又很小，那么这种优化方式可以减少一半由控制冒险带来的代价。

想要丢弃指令，只需要将初始控制值变为0即可，这与指令停顿以解决加载–使用的数据冒险类似。不同的是，丢弃指令的同时也需要改变当分支指令到达MEM阶段时IF、ID和

EX 阶段的三条指令；而在加载 - 使用的数据停顿中，只需要将 ID 阶段的控制信号变为 0 并且将该阶段的指令从流水线中过滤出去即可。丢弃指令，意味着我们必须能够将流水线中 IF、ID 和 EX 阶段中的指令都**清除**。

清除：丢弃流水线中的指令，通常是因为发生了一个未预料到的事件。

4.8.2 缩短分支延迟

一种提升条件分支性能的方式是减少发生分支时所需的代价。到目前为止，我们假定分支所需的下一 PC 值在 MEM 阶段才能被获取，但如果我们将流水线中的条件分支指令提早移动执行，就可以刷新更少的指令。要将分支决定向前移动，需要两个操作提早发生：计算分支目标地址和判断分支条件。其中，将分支地址提前进行计算是相对简单的。在 IF/ID 流水线寄存器中已经得到了 PC 值和立即数字段，所以只需将分支地址从 EX 阶段移动到 ID 阶段即可。当然，分支地址的目标计算将会在所有指令中都执行，但只有在需要时才会被使用。

困难的部分是分支决定本身。对于相等时跳转指令，需要在 ID 阶段比较两个寄存器中的值是否相等。相等的判断方法可以是先将相应位进行异或操作，再对结果按位进行或操作。将分支检测移动到 ID 阶段还需要额外的前递和冒险检测硬件，因为分支可能依赖还在流水线中的结果，在优化后依然要保证运行正确。例如，为了实现相等时跳转指令（或者不等时跳转指令），需要在 ID 阶段将结果前递给相等测试逻辑。这里存在两个复杂的因素：

1. 在 ID 阶段需要将指令译码，决定是否需要将指令旁路至相等检测单元，并且完成相等测试以防指令是一条分支指令，此时可以将 PC 设置为分支目标地址。对分支指令的操作数进行前递的操作原先是由 ALU 前递逻辑处理的，但是在 ID 阶段引入相等检测单元后就需要添加新的前递逻辑。需要注意的是，旁路获得的分支指令的源操作数既可以从 EX/MEM 流水线寄存器中获得，也可以从 MEM/WB 流水线寄存器中获得。

2. 在 ID 阶段分支比较所需的值可能在之后才会产生，因此可能会产生数据冒险，所以指令停顿也是必需的。例如，如果一条 ALU 指令恰好在分支指令之前，并且这条 ALU 指令产生条件分支检测时所需的操作数，那么一次指令停顿就是必需的，因为 ALU 指令的 EX 阶段将发生在分支指令的 ID 阶段之后。又例如，如果一条加载指令恰好在条件分支指令之后，并且条件分支指令依赖加载指令的结果，那么两个时钟周期的停顿就是必需的，因为加载指令的结果要在 MEM 阶段的最后才能产生，但是在分支指令的 ID 阶段的开始就需要了。

尽管这很困难，但是将条件分支指令的执行移动到 ID 阶段的确是一个有效的优化，因为这将分支发生时的代价减轻至只有一条指令，也就是分支发生时正在取的那条指令，下面的例题展示了实现前递路径和检测冒险的更多实现细节。

为了清除 IF 阶段的指令，我们添加了一条称为 IF.Flush 的控制线，它将 IF/IF 流水线寄存器中的指令字段设置为 0。将寄存器清空的结果是将已经取到的指令转换成一条 nop 指令，该指令不进行任何操作，也不改变任何状态。

例题 流水线分支

请描述当这个指令序列中发生分支跳转时会发生什么，这里假定流水线对分支不发生进行了优化，并且将分支执行移动到了 ID 阶段：

```
36  sub  x10, x4, x8
40  beq  x1,  x3, 16  // PC-relative branch to 40+16*2=72
44  and  x12, x2, x5
48  or   x13, x2, x6
```

```
52  add  x14, x4, x2
56  sub  x15, x6, x7
. . .
72  ld   x4, 50(x7)
```

答案 图 4-60 展示了条件分支发生时的情况，不同于图 4-59，这里在分支发生时只有一个流水线气泡。

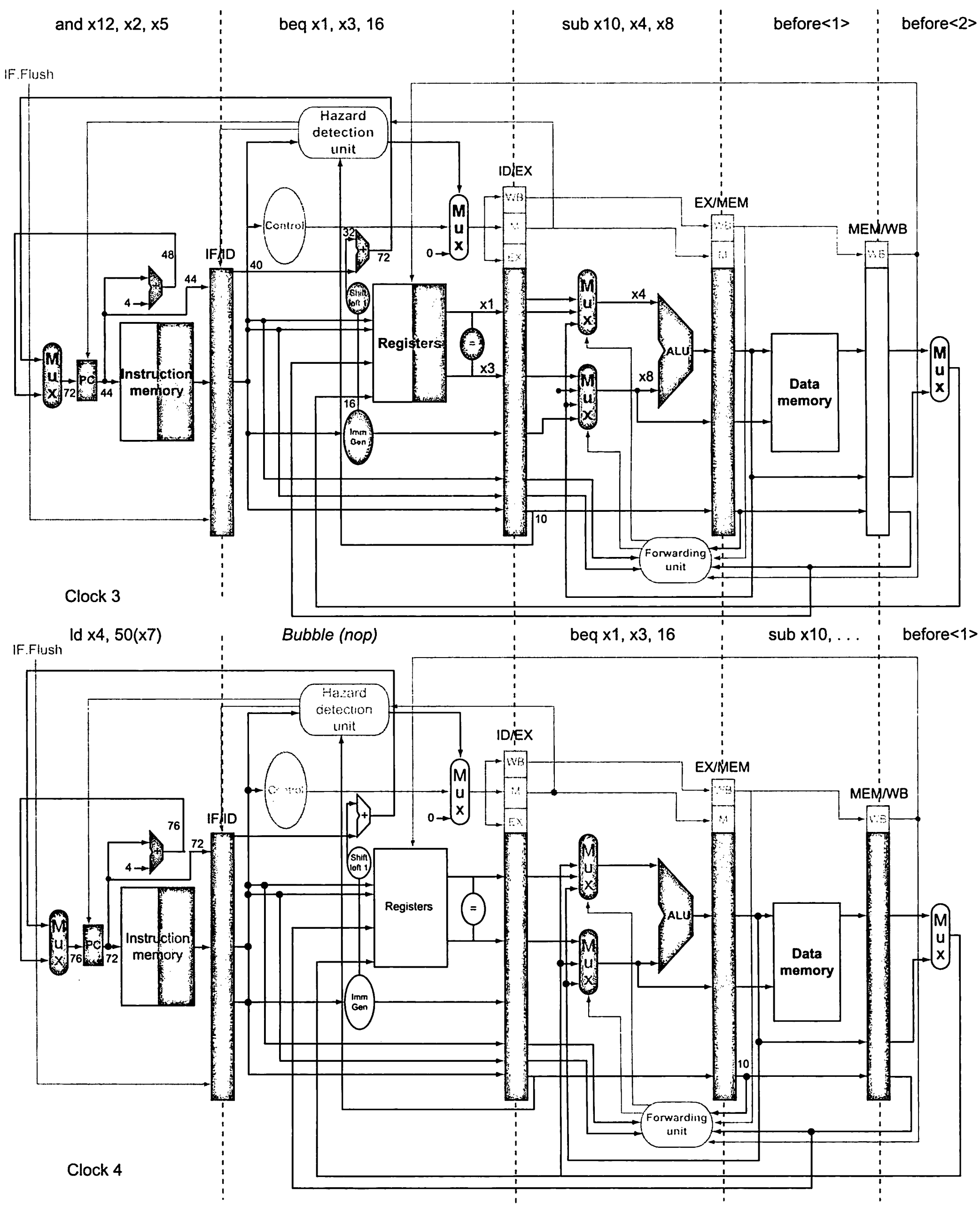


图 4-60　在第 3 个时钟周期的 ID 阶段决定分支执行必须被执行，因此选择 72 作为下一 PC 跳转地址，并且将下个时钟周期获取到的指令置 0。第 4 个时钟周期显示了地址 72 中的指令被获取，并且因为分支发生而在流水线中产生了一个气泡或者 nop 指令

4.8.3 动态分支预测

假定条件分支不发生是一种简单的分支预测形式。在这种形式下，我们预测分支不发生，并在预测错误时清空流水线。对于简单的五级流水线来说，这种方法再结合基于编译的预测，就基本足够了。而对于更深的流水线，从时钟周期的角度来说，分支预测错误的代价会增大。与之相似，对于多发射的情况，从指令丢失的角度来说，分支预测错误的代价也会增大。两者组合起来意味着在一个激进的流水线中，简单的静态预测机制会在性能上造成非常多的浪费。正如在 4.5 节中所述，使用更多的硬件可能可以在程序执行的过程中尝试进行分支预测。

一种方法是检查指令中的地址，查看上一次该指令执行时条件分支是否发生了跳转，如果答案是肯定的，则从上一次执行的地址中取出指令。这种技术称为**动态分支预测**。

动态分支预测：在程序运行时使用运行信息进行分支预测。

这种方法的一种实现方案是采用**分支预测缓存**或**分支历史表**。分支预测缓存是一块按照分支指令的低位地址定位的小容量存储器。这块存储器包含了一个比特，用于表明一个分支最近是否发生了跳转。

分支预测缓存：也称分支历史表，一块按照分支指令的低位地址定位的小容量存储器，包含一个或多个比特以表明一个分支最近是否发生了跳转。

该预测使用一种最简单的缓存，事实上，我们并不知道该预测是否是正确的——这个位置可能已经被另一条拥有相同低位地址的条件分支指令的跳转状态所替换。不过，这并不会影响这种预测方法的准确性。预测只是一种我们希望是正确的假设，所以我们会在预测发生的方向上进行取舍。如果这个假设最终被证明是错误的，这个不正确的预测指令就会被删除，它的预测位也会被置为相反值，之后正确的指令序列会被取指并执行。

这种 1 位的预测机制在性能上有一个缺点：即使一个条件分支总是发生跳转，但一旦其不发生跳转时，就会造成两次预测错误，而不是只造成一次错误。下面这个示例说明了这件事。

例题 **循环与预测**

现在考虑一个循环分支，它在一行代码中发生了 9 次跳转，之后产生 1 次未跳转。假定这个分支的预测位在预测缓存中，请计算这条分支指令的预测正确率。

答案 稳定状态的预测行为会在第一次以及最后一次预测循环中预测失效。其中，最后一次迭代的预测失效是不可避免的，因为此时该分支的预测位会设置为跳转，因为分支在这行代码上已经发生了 9 次跳转。在第一次迭代时分支预测错误是因为在循环的上一次迭代执行中该预测位被设置为不跳转。因此，这个实际发生跳转率为 90% 的分支的分支预测正确率只有 80%（2 次不正确的预测，8 次正确预测）。

理想状态下，对于规律性很强的分支来说，分支预测的准确率应该与分支发生的频率相匹配。2 位预测机制经常为了纠正上述缺点而被使用。在 2 位预测机制中，只有在发生两次错误时预测结果才会被改变。图 4-61 是这个 2 位预测机制的有限状态自动机。

分支预测缓存可以被实现为一个小而专用的、可以被 IF 阶段的指令地址所访问的缓存。如果指令被预测为跳转，一旦新的 PC 已知就开始从目标地址取指，正如之前所述，这个操作可以被前移至 ID 阶段。否则，就会顺序取值并继续执行。如果分支预测错误，预测位就会如图 4-61 中所示的那样改变。

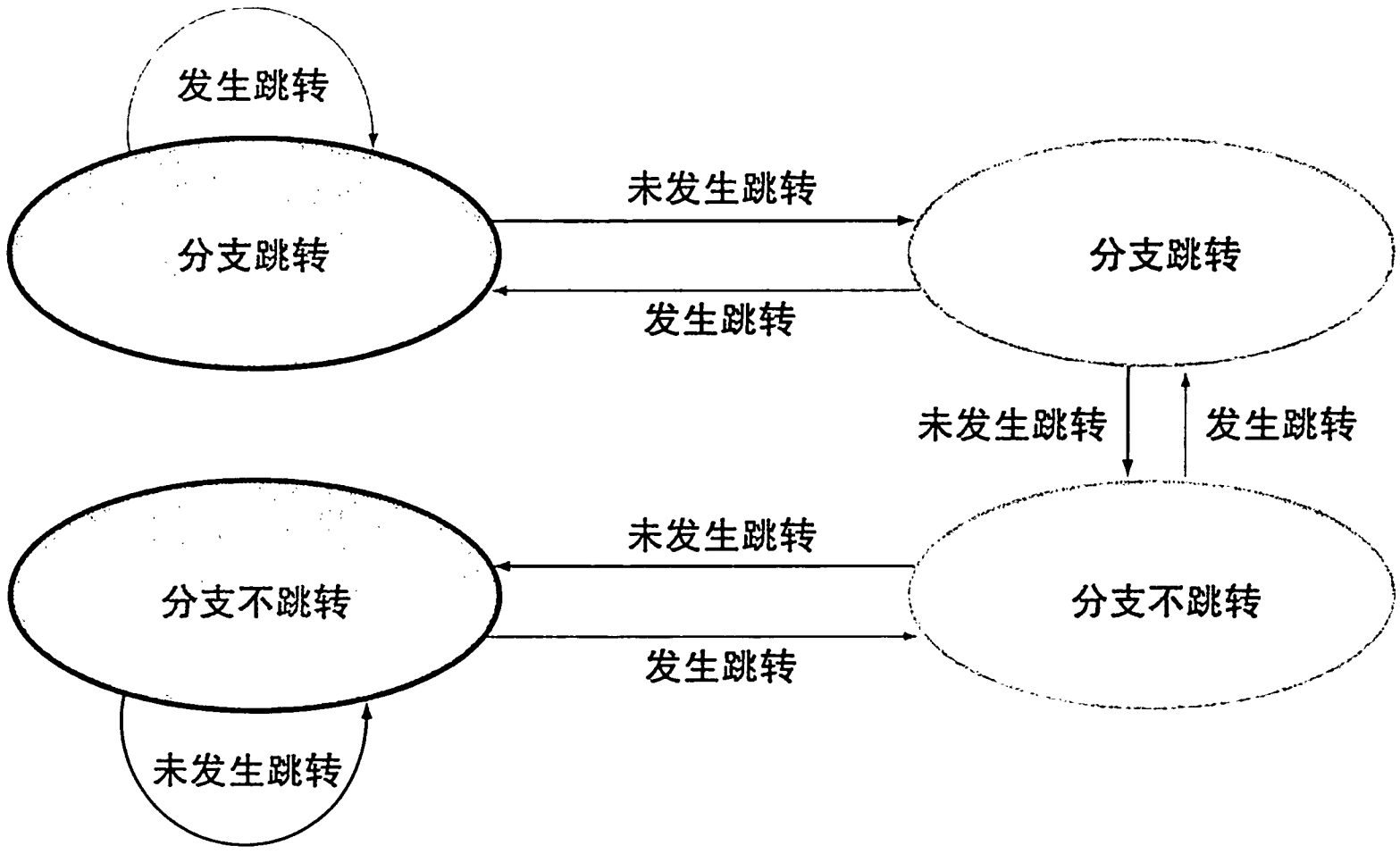


图 4-61　2 位预测机制的状态转换图。通过使用 2 位而不是 1 位的预测位，在一个分支经常发生跳转或经常不跳转的情况下（大多数分支都是这样的）只会发生一次预测失效。2 位预测位在系统中可以被编码为四个状态。2 位预测机制是基于计数器的预测器的一个实例，该预测器在分支跳转时加 1，在分支不跳转时减 1，并且使用表示范围的中位数作为预测分支跳转与不跳转之间的分界点

详细阐述　分支预测器可以告知我们条件分支是否会发生跳转，但依然需要对分支目标地址进行计算。在五级流水线中，这种计算需要一个时钟周期，这意味着发生跳转时需要一个时钟周期的代价来计算分支目标地址。一种解决方案是使用一个缓存来保存目标地址或目标指令以作为分支目标缓存。

分支目标缓存：一个缓存分支目标 PC 值或目标指令的结构。通常被组织为一个带有标记的缓存，相比简单的预测缓存需要更多的硬件消耗。

2 位动态预测机制仅仅使用特定分支的信息。研究表明，对于相同的预测位来说，同时使用局部分支和最近执行分支的全局行为的信息能够获得更好的预测准确率。这种预测器被称为**相关预测器**。一个典型的相关预测器对于每个分支都提供两个 2 位预测器，预测器之间的选择基于分支的上一次执行时跳转还是不跳转。因此，全局分支行为可以被看作在预测查找表中添加了一个额外的索引位。

相关预测器：一种组合了特殊分支指令的局部行为和最近执行的一些分支指令的全局行为信息的分支预测器。

另一种分支预测方法是使用锦标赛预测器。**锦标赛分支预测器**对于每个分支使用多种预测器，并最终给出一个最佳的预测结果。典型的锦标赛预测器对每个分支地址使用两个预测：一个基于局部信息，而另一个基于全局分支行为。一个选择器用于选择采取哪一个预测器的信息进行预测。这个选择器的操作与 1 位或 2 位预测器类似，选择两个预测器中更准确的那个。一些最新的微处理器使用了这种预测器。

锦标赛预测器：一个对于每个分支具有多种预测的分支预测器，其具有一种选择机制，该机制选择对于给定分支选择哪个预测器作为预测结果。

详细阐述　一种减少条件分支数量的方法是添加条件移动指令。与条件分支指令改变 PC 值不同，条件移动指令根据条件改变移动的目标寄存器。例如，ARMv8 指令系统中有一个条件选择指令称为 CSEL。它特别定义了一个目标寄存器、两个源寄存器和一个条件。如果条件为真，则目标寄存器获得第一个操作数的值，否则获得第二个操作数的值。对于指令 CSEL X8，X11，X4，NE 而言，如果条件代码得出操作结果不等于零，则将寄存器 11 中的

值拷贝至寄存器 8，如果结果等于零，则将寄存器 4 中的值拷贝至寄存器 8 中。因此，使用 ARMv8 指令架构的程序相比用 RISC-V 写出的程序拥有更少的条件分支指令。

4.8.4 流水线总结

我们从洗衣房示例开始，介绍了日常生活中的流水线规则。用这个示例类比，我们逐步解释了指令的流水化，从单时钟周期数据通路开始，之后加入了流水线寄存器、前递路径、数据冒险检测、分支预测以及在分支预测错误或加载 – 使用数据冒险时清除指令的机制。图 4-62 是最终的数据通路和控制。现在，我们已经准备好处理另外一种控制冒险——例外，这是一个棘手的问题。

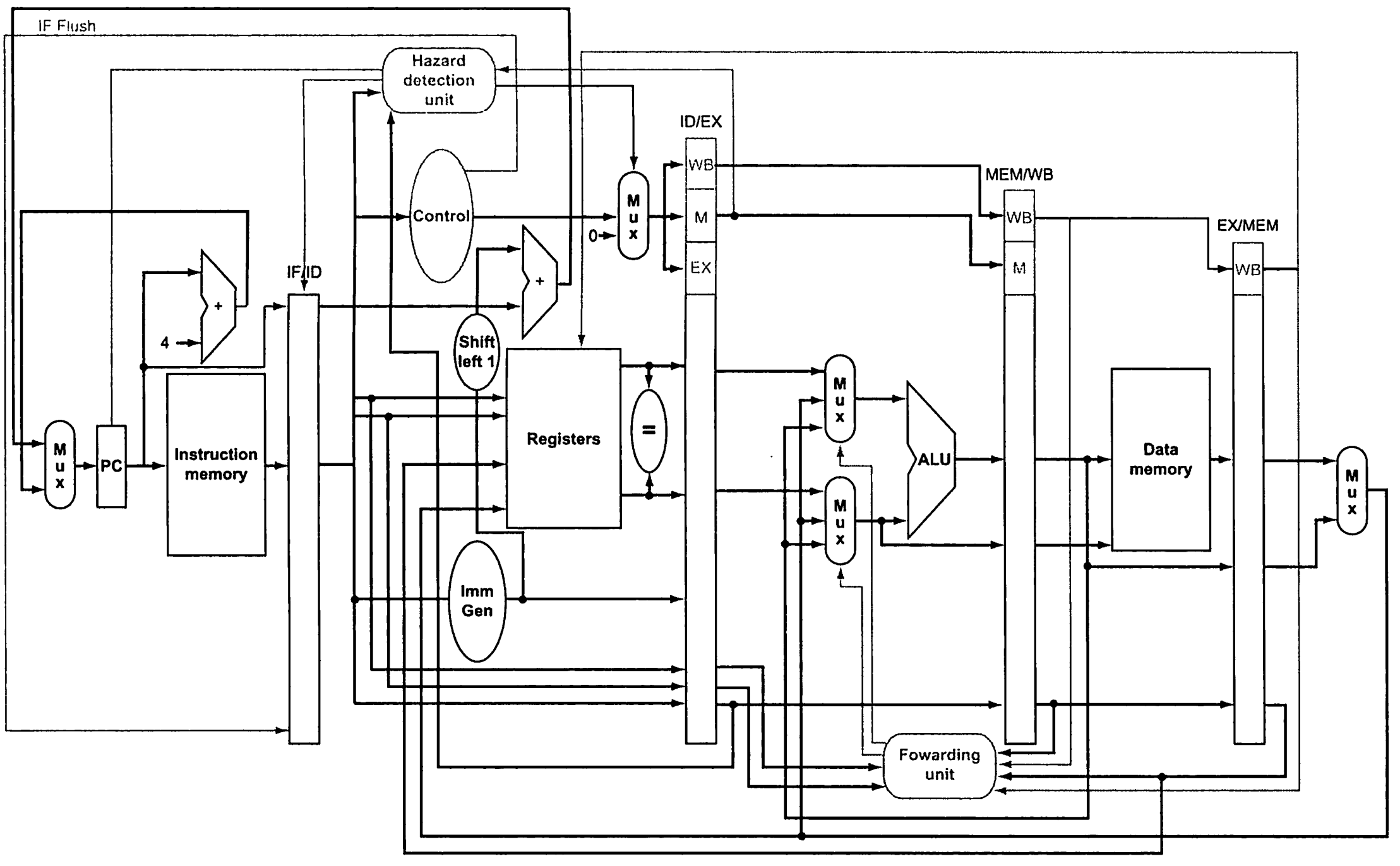


图 4-62 本章最终的数据通路和控制图。需要注意的是，本图只是一张示意图，没有标注诸如图 4-55 中的 ALUsrc Mux 和图 4-49 中的多选控制器这样的完整数据通路中的细节

自我检测 考虑三个分支预测机制：预测分支不发生、预测分支发生以及动态预测。假定它们在预测正确时代价为零，在预测错误时代价为两个时钟周期。假定动态预测的平均预测准确率为 90%。对于下面这些分支来说，哪一种预测机制是最佳选择？

1. 分支发生概率为 5% 的条件分支
2. 分支发生概率为 95% 的条件分支
3. 分支发生概率为 70% 的条件分支

4.9 例外

控制逻辑是处理器设计中最有挑战的部分：验证正确性最为困难，同时也最难进行时序

优化。**例外**（exception）和**中断**（interrupt）是控制逻辑需要实现的任务之一。除分支指令外，它是另一种改变指令执行控制流的方式。最初，人们使用它们是为了处理 CPU 内部的意外事件，例如未定义指令。后续经扩展也可处理与 CPU 进行通信的 I/O 设备，这部分将在第 5 章进行讨论。

> 为处理器设计自动中断处理并不是一件简单的事情，因为中断信号产生时处于处理器各个阶段的指令数目将会非常多。
> *Fred Brooks, Jr., Planning a Computer System: Project Stretch, 1962*

许多体系结构设计者和相关书籍作者并不区分中断和例外，经常使用其中一种同时指代两者。比如，Intel x86 中就是使用中断。在本书中，我们使用例外来指代意外的控制流变化，而这些变化无须区分产生原因是来自于处理器内部还是外部；使用中断仅仅指代由处理器外部事件引发的控制流变化。下表是一些示例，包括例外的类型、引发例外的事件来源以及在 RISC-V 体系结构中的表示。

> **例外**：也称中断，指那些打断程序正常执行的意外事件，比如未定义指令。

> **中断**：来自于处理器外部的例外，一些体系结构也会使用中断表示所有的例外。

事件类型	例外来源	RISC-V中的表示
系统重启	外部	例外
I/O设备请求	外部	中断
用户程序进行操作系统调用	内部	例外
未定义指令	内部	例外
硬件故障	皆可	皆可

例外处理的许多功能需求来自于引发例外的特定场合。因此，第 5 章时我们将重新讨论这些内容，届时将对满足这些功能需求的动机有更深刻的理解。在本章节中，我们只涉及检测例外类型的控制逻辑实现，这些例外和我们之前讨论过的指令系统及微结构实现是相关的。

通常，检测和处理例外的控制逻辑会处于处理器的时序关键路径上，这对处理器时钟频率和性能都会产生重要影响。如果对控制逻辑中的例外处理不给予充分重视，一旦尝试在复杂设计中添加例外处理，将会明显降低处理器的性能。这和处理器验证一样复杂。

4.9.1　RISC-V 体系结构中如何处理例外

在目前所讲过的实现中，只存在两种例外类型：未定义指令和硬件故障。例如，假设在指令 `add x1, x2, x1` 执行时出现硬件故障。当例外发生时，处理器必须执行的基本动作是：在*系统例外程序计数器*（Supervisor Exception Program Counter，SEPC）中保存发生例外的指令地址，同时将控制权转交给操作系统。

之后，操作系统将做出相应动作，包括为用户程序提供系统服务，硬件故障时执行预先定义好的操作，或者停止当前程序的执行并报告错误。完成例外处理的所有操作后，操作系统使用 SEPC 寄存器中的内容重启程序的正常执行。可能是继续执行原程序，也可能是终止程序。对于重启程序执行，在第 5 章将进行更为深入的讨论。

操作系统进行例外处理，除了引发例外的指令外，还必须获得例外发生的原因。目前使用两种方法来通知操作系统。RISC-V 中使用的方法是设置*系统例外原因寄存器*（Supervisor Exception Cause Register，SCAUSE），该寄存器中记录了例外原因。

另一种方法是使用**向量式中断**（vectored interrupt）。该方法用基址寄存器加上例外原因（作为偏移）作为目标地址来完成控制流转

> **向量式中断**：一种中断处理机制，根据例外原因来决定后续控制流的起始地址。

换。基址寄存器中保存了向量式中断内存区域的起始地址。比如，我们可以根据例外类型定义下述两类例外的例外向量起始地址。

例外类型	例外向量地址（即偏移，与中断向量表基地址相加）
未定义指令	00 0100 0000_2
系统错误（硬件故障）	01 1000 0000_2

操作系统可根据例外向量起始地址来确定例外原因。如果不使用此种方法，如 RISC-V，就需要为所有例外提供统一的入口地址，由操作系统解析状态寄存器来确定例外原因。对于使用向量式例外的设计者，每个例外入口需要提供比如 32 字节或 8 条指令大小的区域，供操作系统记录例外原因并进行简单处理。

通过添加一些额外寄存器和控制信号，并稍微扩展控制逻辑，就可以完成对各种例外的处理。假设，我们使用统一入口地址的方式实现例外处理，设置该地址为 0000 0000 1C09 0000_{16}。（实现向量式例外与此难度相当。）我们需要在当前 RISC-V 的实现中添加两个额外的寄存器。

- SEPC：64 位寄存器，用来保存引起例外的指令的地址。（该寄存器在向量式例外中也需要使用。）
- SCAUSE：用来记录例外原因的寄存器。在 RISC-V 体系结构中，该寄存器为 64 位，大多数位未被使用。假设对上述提及的两种例外类型进行编码并记录，其中未定义指令的编码为 2，硬件故障的编码为 12。

4.9.2 流水线实现中的例外

流水线实现中，将例外处理看成另一种控制冒险。例如，假设 add 指令执行时产生硬件故障。正如之前章节中处理发生跳转的分支一样，我们需要在流水线上清除掉 add 之后的指令，并从新地址开始取指。和处理分支指令不同的是，例外会引起系统状态的变化。

处理分支预测错误时，我们将取指阶段的指令变为空操作（nop），以此来消除影响。对于进入译码阶段的指令，增加新逻辑控制译码阶段的多选器使输出为 0，流水线停顿。添加一个新的控制信号 ID.Flush，它与来自于冒险检测单元的 stall 信号进行或（OR）操作。使用该信号对进入译码阶段的指令进行清除。对于进入执行阶段的指令，我们使用一个新的控制信号 EX.Flush，使得多选器输出为 0。RISC-V 体系结构中使用 0000 0000 1C09 0000_{16} 作为例外入口地址。为保证从正确地址开始取指，我们为 PC 多选器新增一个输入，保证能将上述例外入口地址送给 PC 寄存器。具体见图 4-63。

上述例子指出了例外处理需要注意的一个问题：如果我们在 add 指令执行完毕后检测例外，程序员将无法获得 x1 寄存器中的原值，因为它已更新为 add 指令的执行结果。如果我们在 add 指令的 EX 阶段检测例外，可以使用 EX.Flush 信号去避免该指令在 WB 阶段更新寄存器。有一些例外类型，需要最终完成引发例外的指令的执行。最简单的方法就是清除掉该指令，并在例外处理结束后从该指令重新开始执行。

最后一步是，在 SEPC 寄存器中保存引发例外的指令的地址。图 4-63 中显示了详细的数据通路，包括处理分支的硬件逻辑和处理例外新增修改。

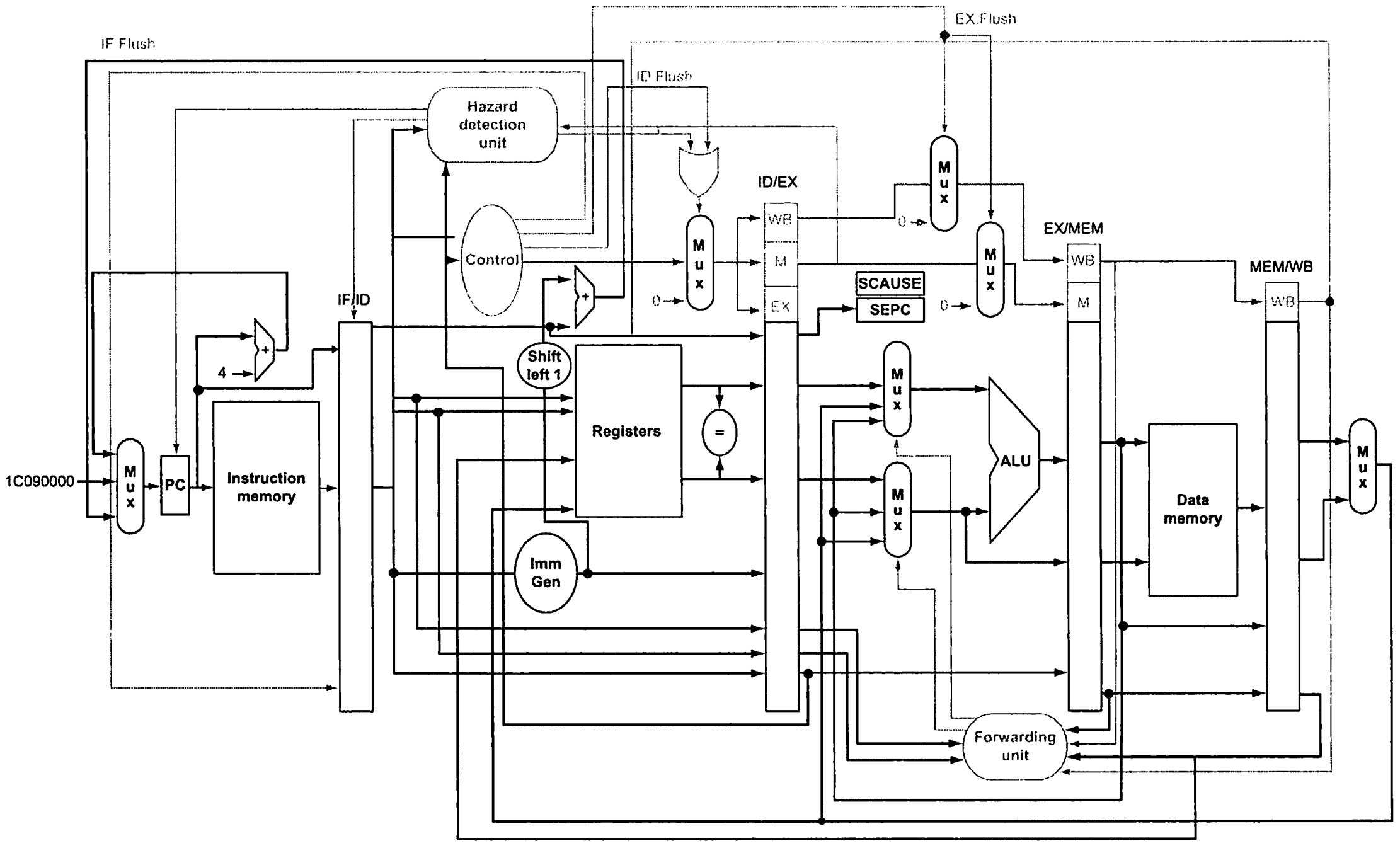


图 4-63 例外处理的数据通路和控制信号。新增的关键逻辑包括：PC 选择器新增输入 0000 0000 1C09 0000_{16}；新增 SCAUSE 寄存器来记录例外原因；新增 SEPC 寄存器保存引起例外的指令的地址。0000 0000 1C09 0000_{16} 是例外处理程序的统一入口地址

例题 | 流水线处理器中的例外

给定下面的指令序列：

```
40hex   sub    x11, x2, x4
44hex   and    x12, x2, x5
48hex   or     x13, x2, x6
4Chex   add    x1,  x2, x1
50hex   sub    x15, x6, x7
54hex   ld     x16, 100(x7)
. . .
```

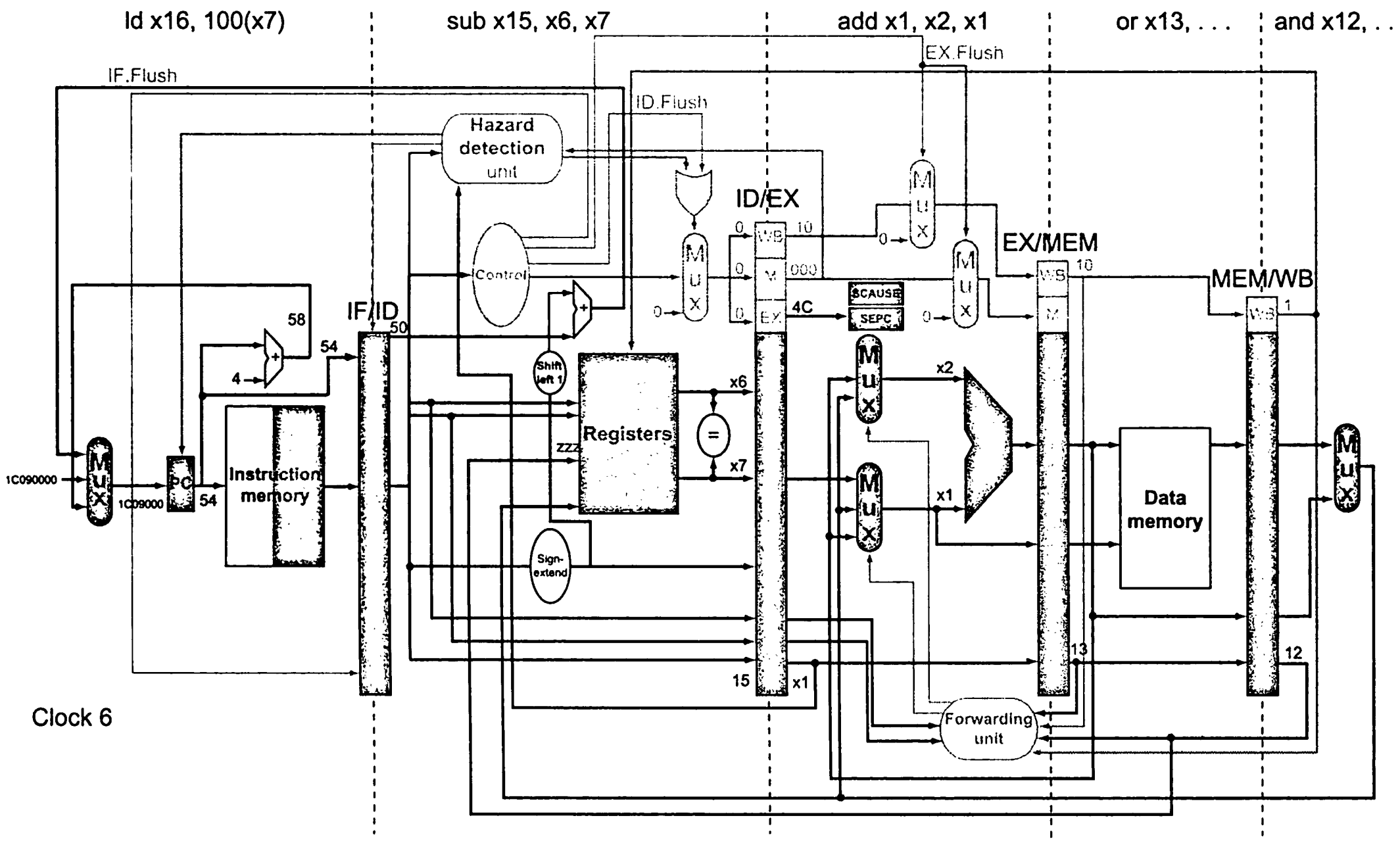
假设例外处理程序入口的指令序列如下：

```
1C090000hex   sd    x26, 1000(x10)
1C090004hex   sd    x27, 1008(x10)
. . .
```

如果 add 指令发生硬件故障例外，流水线会如何变化？

答案 | 图 4-64 中从 add 指令进入 EX 阶段开始，假设该阶段检测出硬件故障，0000 0000 1C09 0000_{16} 被送给 PC 寄存器。Clock 7 时，add 及其后续指令都被清除掉，从例外处理程序的第一条指令开始取值。注意，add 指令的地址 $4C_{16}$ 被保存。

前面提到了一些关于例外的例子，将来在第 5 章时还会讨论。对于流水线处理器，每个周期同时有 5 条指令在流水线中执行，例外处理的挑战在于如何将各种例外与指令进行对应。而且，同一个周期内可以同时发生多个例外。解决方案是，对例外进行优先级排列，便于判断服务顺序。在 RISC-V 实现中，硬件实现例外的优先级排序。



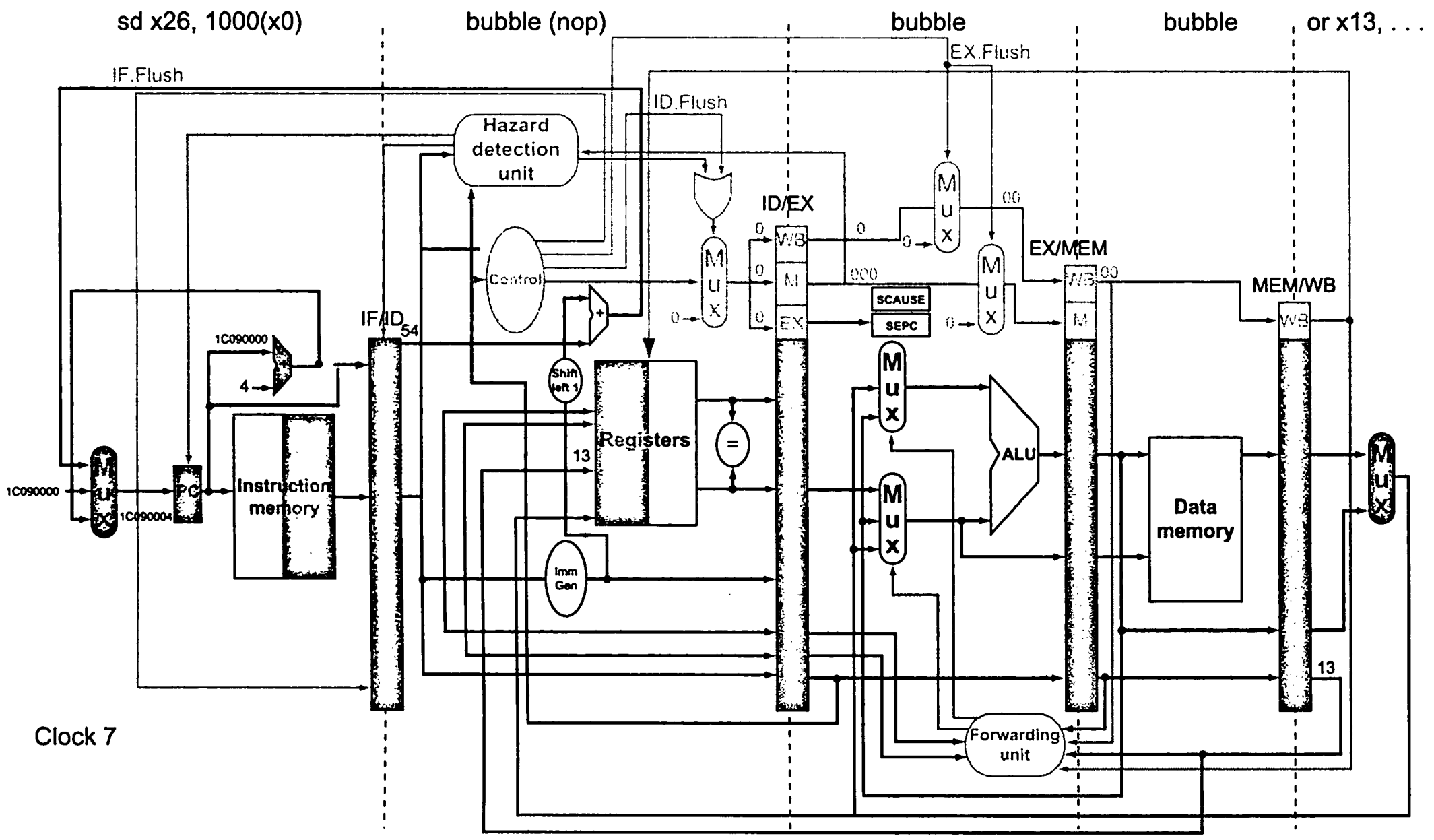


图 4-64 add 指令发生硬件故障的例外处理结果。第 6 个周期，处于 EX 阶段的 add 指令检测到例外，在 SEPC 寄存器中保存 add 指令的地址（$4C_{16}$），所有 Flush 信号进行重置，add 指令的所有控制信号置为无效（数值设为 0）。第 7 个周期，在流水线中插入气泡，从例外处理例程入口地址 0000 0000 1C09 0000_{16} 开始取第一条指令 sd x26, 1000(x0)。注意：and 和 or 指令在 add 指令之前进入流水线，因此这两条指令需要完成

I/O 设备请求和硬件故障与指令无关，因此例外响应和处理有较大的灵活性。因而，复用其他例外的处理机制也能正常工作。

SEPC 寄存器保存引发例外的指令的地址。如果有多个例外同时发生，SCAUSE 寄存器中记录当前最高优先级的例外信息。

硬件/软件接口 硬件和操作系统必须协同工作，例外行为才会如设计者所愿。按照约定，硬件暂停引发例外的指令的执行，完成流水线中所有之前指令的操作，清除掉之后的指令，设置寄存器记录例外原因，保存引发例外的指令的地址，并转向例外处理程序入口执行。操作系统查询例外原因寄存器并做出相应动作。对于未定义指令或者硬件故障，一般操作系统会停止程序的执行并返回例外类型。对于 I/O 设备请求或系统调用，操作系统保存程序状态，完成预定任务，并在不久之后恢复状态重新开始执行程序。特别是 I/O 设备请求，操作系统会调度其他程序执行，直到该 I/O 请求完成再继续执行发生例外的程序。若想加快例外处理的速度，保存和恢复程序状态是非常关键的。一种常见的重要例外是处理器的缺页异常（page fault）。第 5 章我们将更详细的介绍它们。

详细阐述 在流水线处理器中，将例外和指令正确关联起来存在一定难度，一些处理器设计者在非关键情况下考虑放松对此的要求。这样的处理机制被称为非精确中断（imprecise interrupts）或者非精确例外（imprecise exception）。在上文的例子中，当流水线检测到例外时程序计数器 PC 已经变为 58_{16}，而发生例外的是地址为 $4C_{16}$ 的 add 指令。具有非精确例外的处理器将在 SEPC 寄存器中保存 58_{16}，让操作系统去判断到底是哪条指令引发了例外。与大多数处理器相同，RISC-V 支持精确中断或精确例外。一个原因是，当流水线不断加深时，非精确例外会增加操作系统处理的难度和复杂度。为避免该问题，深度流水线应该和五级流水线一样，记录引发例外的指令的地址，这会简化软硬件的设计。另一个原因是考虑到支持虚拟内存，此点我们在第 5 章再进行讨论。

非精确中断：也称为非精确例外。流水线中的例外或中断并未和引发该例外或中断的指令精确对应。

精确中断：也称为精确例外。流水线中的例外或中断和引发该例外或中断的指令精确对应。

详细阐述 RISC-V 指令系统中使用 0000 0000 1C09 0000_{16} 作为例外入口地址，这个选择稍显随意。一些 RISC-V 处理器在实现时将例外入口地址保存在特殊的寄存器（Supervisor Trap Vector，STVEC）中，操作系统可以通过它来配置例外入口地址。

自我检测 在下述指令序列中，那种例外将先被检测出来？

```
1. xxx x11, x12, x11    // undefined instruction
2. sub x11, x12, x11    // hardware error
```

4.10 指令间的并行性

说明：本节是对一些高级复杂专题的概述。如果你想了解更多内容，可参考我们的另一本书：《计算机体系结构：量化研究方法》(第 5 版)。本节这几页内容将扩展为 200 页，包括附录。

流水线技术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**（ILP）。提高指令级并行度主要有两种方法。第一种是增加流水线的级数，让更多的指令重叠执行。仍然使用上文提到的洗衣店进行类比。假设洗衣阶段所需时间比其他阶段都长，我们可以将洗衣阶段再细分为洗涤、漂洗和甩干三个阶段。这样就将一个四级流水线变为六级流水线。不论是处理器还是洗

指令级并行：指令间的并行性。

衣店，如需获得最高加速比，还要重新调整其他阶段的时长至相等来平衡流水线。加深流水线后，由于有更多的操作可以重叠执行，指令间的并行度更高。同时，时钟周期变短，主频变高，处理器性能也就更高。

另一种提高指令并行度的方法是，增加流水线内部的功能部件数量，这样可以每周期发出多条指令。这种技术被称为**多发射**（multiple issue）。一个拥有三个洗衣机和三个烘干机的多发射洗衣店代替了之前的家庭式洗衣机和烘干机。也许你还需要招聘一些助手来折叠和收纳，这样就能在相同时间内完成之前的三倍工作量。唯一的缺点在于，需要在相邻流水阶段之间传递负载，并保证所有机器都满负荷工作，这增加了额外的工作量。

多发射：一个时钟周期内可以发射多条指令的策略。

每周期发射多条指令，使得指令执行频率可以超过时钟频率。换句话来说，就是 CPI 可以小于 1。在第 1 章中我们提过，有时候衡量指标的倒数也是有用的，例如 IPC，即每个周期的执行指令数。举例，一个主频为 3GHz、发射宽度为 4 的多发射处理器，峰值速度为每秒执行 120 亿条指令，理论上 CPI 为 0.25，或 IPC 为 4。如果这是一个五级流水的处理器，那么同一时间内流水线中最多会有 20 条指令在执行。目前高端处理器的发射宽度为每周期 3 ～ 6 条指令，普通处理器的发射宽度一般为 2。不过，多发射技术会有一些限制，例如哪些指令可以同时执行、如果发生冒险如何处理等。

实现多发射处理器主要有两种方法，区别在于编译器和硬件的不同分工。如果指令发射与否的判断是在编译时完成的，称为**静态多发射**（static multiple issue）。如果指令发射与否的判断是在动态执行过程中由硬件完成的，称为**动态多发射**（dynamic multiple issue）。这两个方法可能还有其他一些名称，但都不够准确或限制过严。

静态多发射：多发射的一种实现方法，由编译器完成发射相关判断。

动态多发射：多发射的一种实现方法，在动态执行过程中由硬件完成发射相关判断。

在多发射流水线中，需要处理如下两个主要任务：

1. 将指令打包并放入**发射槽**。处理器如何判断本周期发射多少条指令？发射哪些指令？在大多数静态发射处理器中，编译器会完成这部分工作。而在动态发射处理器中，这部分工作通常会在运行时由硬件自动完成，编译器可以通过指令调度来提高发射效率。

发射槽：指令发射时所处位置，可类比为起跑位置。

2. 处理数据和控制冒险。在静态发射处理器中，编译器静态处理了部分或所有指令序列中存在的数据和控制冒险。相应的，大多数动态发射处理器是在执行过程中使用硬件技术来解决部分或所有类型的冒险。

对于静态发射和动态发射，虽然我们把它们描述成两种截然不同的方法，但在实际中，不同的方法间经常互相借鉴，没有哪一种方法能说自己很纯粹。

4.10.1 推测的概念

推测是另一种非常重要的深度挖掘指令级并行的方法。以预测思想为基础，**推测**方法允许编译器或处理器来“猜测”指令的行为，并允许其他与被推测指令相关的指令提前开始执行。例如，我们可以对分支指令结果进行推测，这样分支指令之后的指令可以提早执行。再例如，对于先 store 再 load 的指令序列，可以推测两条指令的访存地址不同，这样允许 load 先于 store 执行。推测的难点在于预测结果可能出现错误。因此，所有推测机制都必须包括预测结果正确性的检查机制，以及预测出错后的恢复机制，以消除推测式执行带来的

推测：编译器或者处理器“猜测”指令的行为，以尽早消除掉该指令与其他指令之间的依赖关系。

影响。这种恢复机制的实现增加了结构设计的复杂度。

可以在编译时完成推测，也可以在执行时由硬件完成推测。例如，编译器可以根据推测结果进行指令顺序重排，将分支后指令移动到分支指令前，或者将 load 指令移动到 store 指令前。处理器硬件也可以在动态执行时完成相同的操作，所使用的技术将在本节稍后讨论。

实现推测错误时的恢复机制非常困难。在软件实现的推测中，编译器经常需要插入额外的指令来检查推测的正确性，并在检测到推测错误时提供例程进行恢复。在硬件推测式执行中，处理器通常会保存推测的结果直到推测被确定是正确的。如果推测是正确的，将使用保存的推测结果更新寄存器或存储器，完成推测路径上的指令。如果推测是错误的，硬件清除推测结果，并从正确的指令处重新开始执行。推测错误需要对流水线进行恢复或者停顿，这显然会极大地降低性能。

推测式执行还会引入另一个问题：对某条指令进行推测还可能引入不必要的例外。例如，假设某条 load 指令处于推测式执行，同时该 load 指令的访存地址发生了越界，则会引发例外。如果推测是错误的，这就意味着发生了本不该发生的例外。这个问题非常复杂，因为如果这条 load 指令不是推测执行，那么例外是一定会发生的。对于编译支持的推测式执行，可以通过添加特定支持来避免这样的问题，对此类例外一直延迟响应直到确认推测正确。对于硬件推测式执行，例外将被记录直到确认推测正确，这时被推测的指令将被提交，检测到例外，转入正常的例外处理程序进行处理。

如果推测正确，处理器的性能将被改善；一旦推测错误，处理器的性能会受到较大影响。人们投入大量的精力去研究何时进行推测。在本节后面的内容中，我们将详细介绍静态和动态的推测技术。

4.10.2　静态多发射

静态多发射处理器是由编译器来支持指令打包和处理指令间的冒险。对于静态多发射处理器，可以将同一周期发射出去的指令集合（一般被称为**发射指令包**）看成一条需要进行多种操作的“大指令”。这样说并不仅仅是为了类比。因为静态多发射处理器通常会对同一周期发射的指令类型进行限制，将发射指令包看成一条预先定义好、需要进行多种操作的指令，这正符合**超长指令字**（Very Long Instruction Word，VLIW）的设计思路。

发射指令包：同一周期发射的指令组合。可能是由编译器静态打包，也可能是由处理器在动态执行过程中进行调度。

超长指令字：一种类型的指令系统体系结构，支持在单条指令中使用不同的编码位来定义多个可同时被发射的独立操作。

同时，大多数静态发射处理器也依赖编译器来处理数据和控制冒险。编译器的任务包括静态分支预测和代码调度，以减少或消除所有的冒险。在描述更先进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 RISC-V 处理器的例子。

举例：静态多发射 RISC-V 处理器

为了解静态多发射技术，我们考察一个简单的双发射 RISC-V 处理器。其中，指令序列中的一条指令是定点 ALU 指令或者分支指令，另一条指令是 load 或者 store 指令。通常，一些嵌入式处理器正是如此来使用。单个周期内发射两条指令需要同时取指和译码 64 位指令。在许多静态单发射处理器，特别是超长指令字处理器中，为简化指令的译码和发射，对可同时发射的指令组合做出了限制。例如，需要指令成对，指令地址需要 64 位边界对齐，ALU

指令和分支指令放在前面。而且，如果指令对中的一条指令无法发射，需要将其替换成 nop 指令。这样一来，就保证了指令总是成对发射，当然其中一条可能是 nop。图 4-65 给出了指令成对进入流水线的过程。

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

图 4-65 静态双发射流水线操作。ALU 和数据传输类指令同时被发射。假设使用和单发射流水线相同的五级流水结构。虽然这样的流水级划分并不是严格必需的，但确实能带来一些好处。尤其是，所有指令统一在最后一个流水级进行寄存器更新，这样有助于实现精确例外模型，简化例外处理的实现。如上所述，例外处理在多发射处理器实现中是一个难点

静态多发射处理器对于潜在的数据和控制冒险有不同的解决方案。在一些设计实现中，由编译器来实现所有冒险的解决、代码的调度以及插入相应的 nop。因此在代码动态执行过程中，硬件可以完全不去关心冒险检测或者流水线停顿的产生。而在另一些设计实现中，使用硬件来检测两个指令包之间的数据冒险，并产生相应的流水线停顿。编译器只负责在单个指令包中检测所有类型的相关。即便如此，单个冒险也通常会导致整个指令包的发射停顿。不论是采用软件来解决所有的冒险，还是仅在两个指令包间降低冒险发生的比例，如果使用上文中提到的“单条大指令”的思想来进行分析，将更有助于加深理解。在下文的例子中，我们假设使用的是第二种方法，即用硬件来检测两个指令包之间的数据冒险，编译器只负责在单个指令包中检测所有类型的相关。

如果想同时发射 ALU 和数据传输类指令，除了上文所说的冒险检测和流水线停顿逻辑，首先需要添加的硬件资源是寄存器堆的读写口（具体见图 4-66）。在同一个时钟周期内，ALU 指令需要读取两个源寄存器，store 指令可能需要读取两个以上的源寄存器；ALU 指令需要更新一个目标寄存器，load 指令也需要更新一个目标寄存器。由于 ALU 部件只负责 ALU 指令的执行，因此还需要额外增加一个加法器来进行访存地址的计算。如果不增加这些额外的硬件资源，我们的双发射流水线将产生大量的结构冒险。

很明显，这种双发射处理器最多能提高两倍的性能，但这也需要程序中存在两倍的、可重叠执行的指令数目。而这种重叠执行又会因增加数据和控制冒险而导致性能损失。例如，在我们的简单五级流水线结构中，load 指令有一个周期的**使用延迟**（use latency）。如果下一条指令需要使用 load 指令的结果，那么它必须停顿一周期。同样，在双发射五级流水线结构中，load 指令也存在一个周期的使用延迟，而这时需要停顿后续两条指令（ALU 和 load/store 指令）的执行。而且，在单发射五级流水线中，ALU 指令本来是没有使用延迟的。但

使用延迟：为保证能够正确使用 load 指令的执行结果，在 load 指令和后续相关指令间插入的时钟周期数。

在双发射流水线中，需要同时发射 ALU 指令和 load 或 store 指令。如果这两条指令存在数据冒险，则 load 或 store 指令不能被发射，相当于 ALU 指令增加了一个周期的使用延迟。为有效挖掘多发射处理器中可用的并行性，需要使用更高级的编译器或硬件动态调度技术，静态多发射处理器对编译器提出了更高的要求。

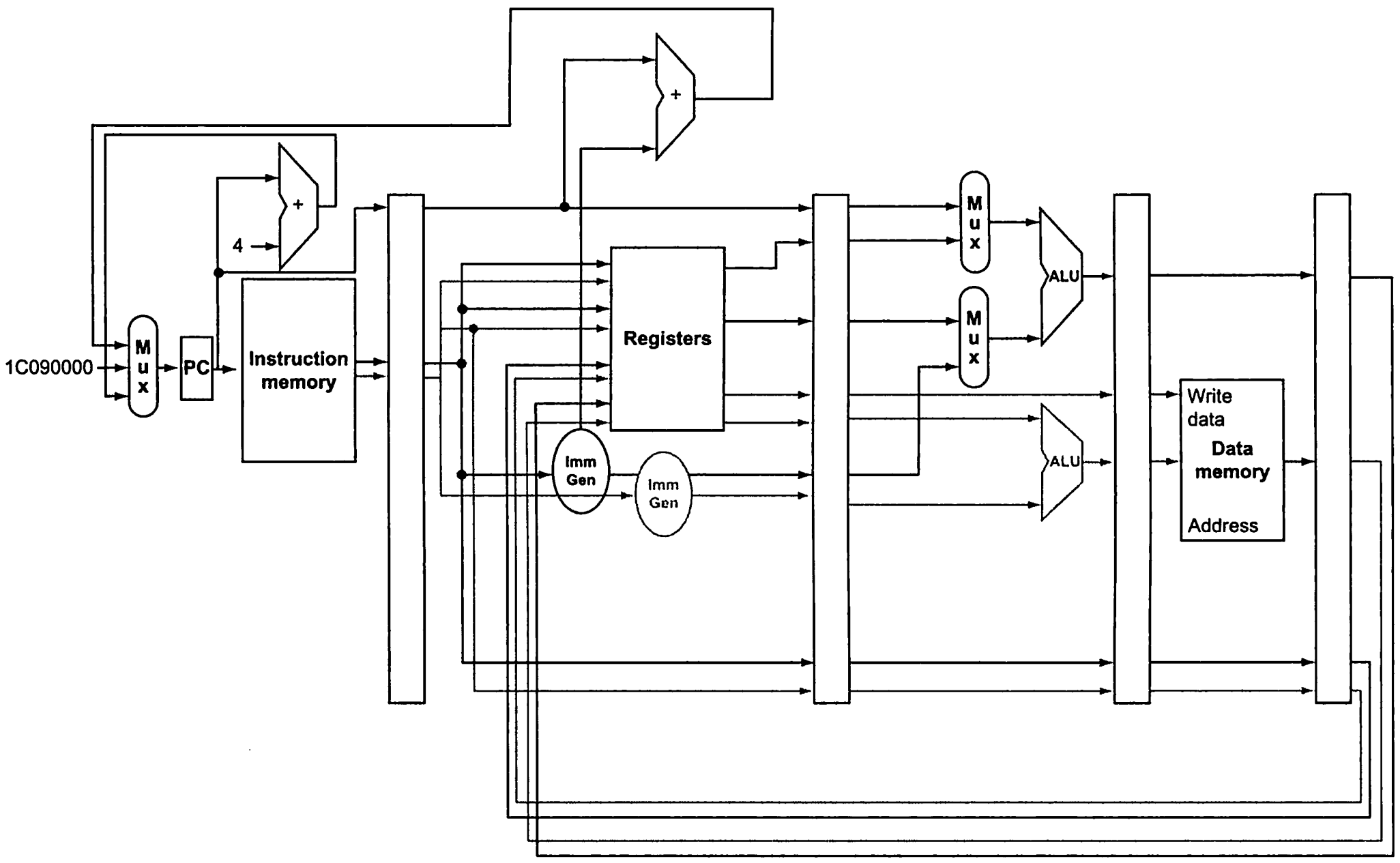


图 4-66 静态双发射数据通路。灰色部分是为静态双发射增加的数据通路，包括另一条从指令存储中取出来的 32 位指令，以及新添的寄存器堆的两个读端口和一个写端口，以及新增的一个 ALU。新增的 ALU 用来进行访存地址计算，另一个 ALU 用来处理其他指令

例题 | 简单的多发射代码调度

如果是 RISC-V 的静态双发射流水线实现，下面的这个循环体应该如何调度？

```
Loop: ld   x31, 0(x20)     // x31=array element
      add  x31, x31, x21   // add scalar in x21
      sd   x31, 0(x20)     // store result
      addi x20, x20, -8    // decrement pointer
      blt  x22, x20, Loop  // compare to loop limit,
                           // branch if x20 > x22
```

重新排列上述指令，尽可能避免流水线。假设分支是可以被预测的，也就是控制冒险由硬件解决。

答案 | 前三条指令具有数据相关，因此重点调度后两条。图 4-67 中给出了该指令序列的最佳调度方案。注意，只有一对指令占用了两个发射槽。每次循环需要花费 4 个时钟周期，也就是 4 个时钟周期完成 5 条指令的执行，CPI 为 0.8 或者 IPC 为 1.25。理论上，使用双发射技术，CPI 可以达到 0.5 或者 IPC 为 2。注意，在计算 CPI 或者 IPC 时，我们并不考虑 nop 的影响。这样做只是为了 CPI 的计算，对性能没有任何帮助。