4.6.1 流水线的图形化表示

掌握流水线技术可能会很困难，因为在每个时钟周期内同时有多条指令在一个单数据通路中执行。为了帮助理解，这里提供了两种基本的流水线图，分别是多时钟周期流水线图（如图 4-32）和单时钟周期流水线图（如图 4-34 ～图 4-38）。多时钟周期流水线图相对来说更简单，但并不包含所有细节。例如，考虑如下的 5 条指令所组成的序列：

```
ld      x10, 40(x1)
sub     x11, x2, x3
add     x12, x3, x4
ld      x13, 48(x1)
add     x14, x5, x6
```

图 4-41 是这组指令的多时钟周期流水线图。与图 4-23 中的洗衣流水线类似，图中时间从左边前进到右边，指令从顶端执行到底端。沿着指令轴分布的是流水线的各个阶段，它们占据相应的时钟周期。这种形式化的数据通路表示了图形化流水线的五个阶段，不过也可以用矩形块来命名每个流水线阶段。图 4-42 是多时钟周期流水线图的一种更加传统的版本。需要注意的是，图 4-41 显示了在每个流水线阶段中使用的物理资源，而图 4-42 显示了每个流水线阶段的名称。

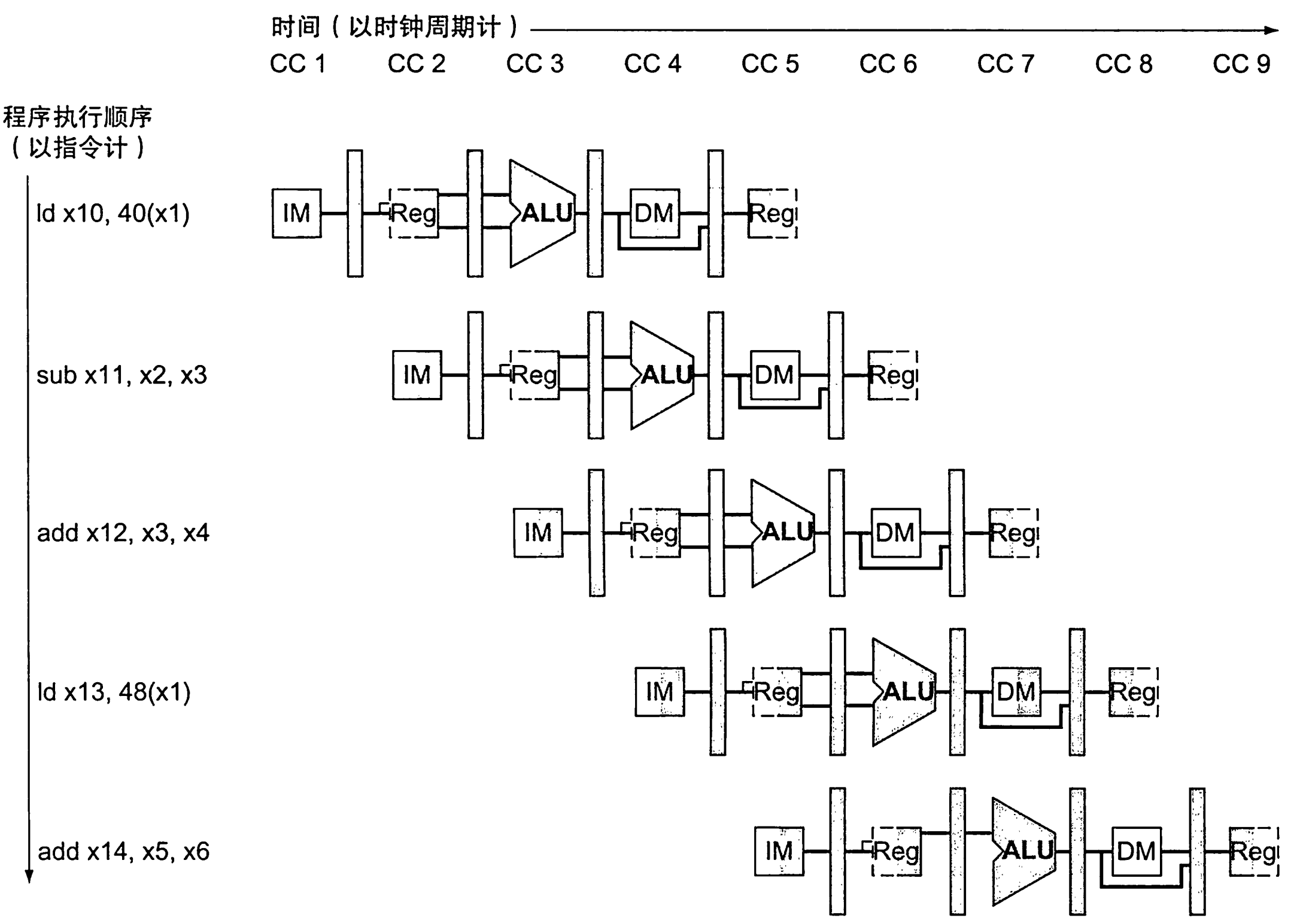


图 4-41 五条指令的多时钟周期流水线图。这种流水线表示在一幅图内展示了完整的指令执行过程。指令在“指令执行序列”中从上到下执行，时钟周期从左向右移动。不同于图 4-26，在本图中我们给出了每个阶段之间的流水线寄存器。图 4-42 给出了本图的一种传统画法

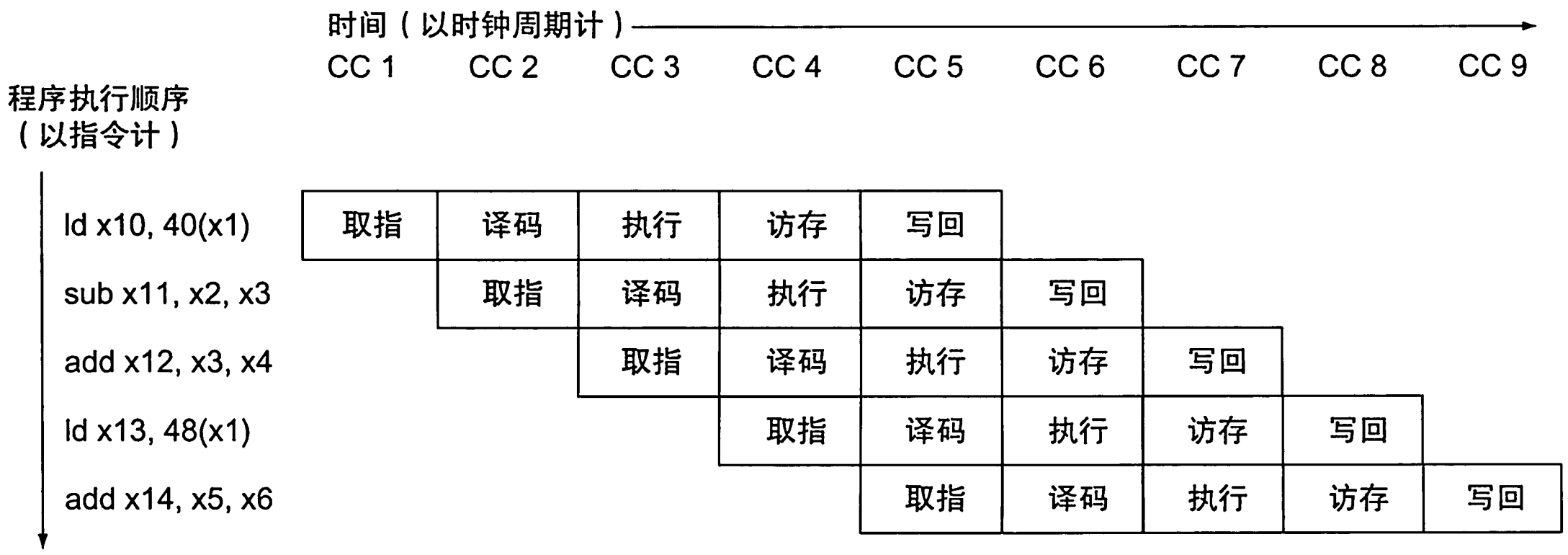


图 4-42　图 4-41 中五条指令的多时钟周期流水线图的传统画法

单时钟周期流水线图显示了在一个单时钟周期内整个数据通路的状态，通常所有五条指令都在流水线中，被各自流水线阶段的标签所标识。我们使用这种类型的图来表示每个时钟周期内流水线中所发生的事情的细节。通常，这种图以组的形式出现，以显示一系列时钟周期内的流水线操作。我们使用多时钟周期图来概括描述流水线情况（如果你想要了解图 4-41 的更多细节，4.13 节中给出了更多关于单时钟图的说明）。单时钟周期图代表在一组多时钟周期图中一个时钟周期的垂直切片，展示了流水线在指定时钟周期上每条指令对数据通路的使用情况。例如，图 4-43 是对应于图 4-41 和图 4-42 中第五个时钟周期的单时钟周期图。显然，这张单时钟周期图包含更多细节，并且在显示相同数量的时钟周期时需要占用更多空间。在练习题中你需要为其他的代码序列创建这类流水线图。

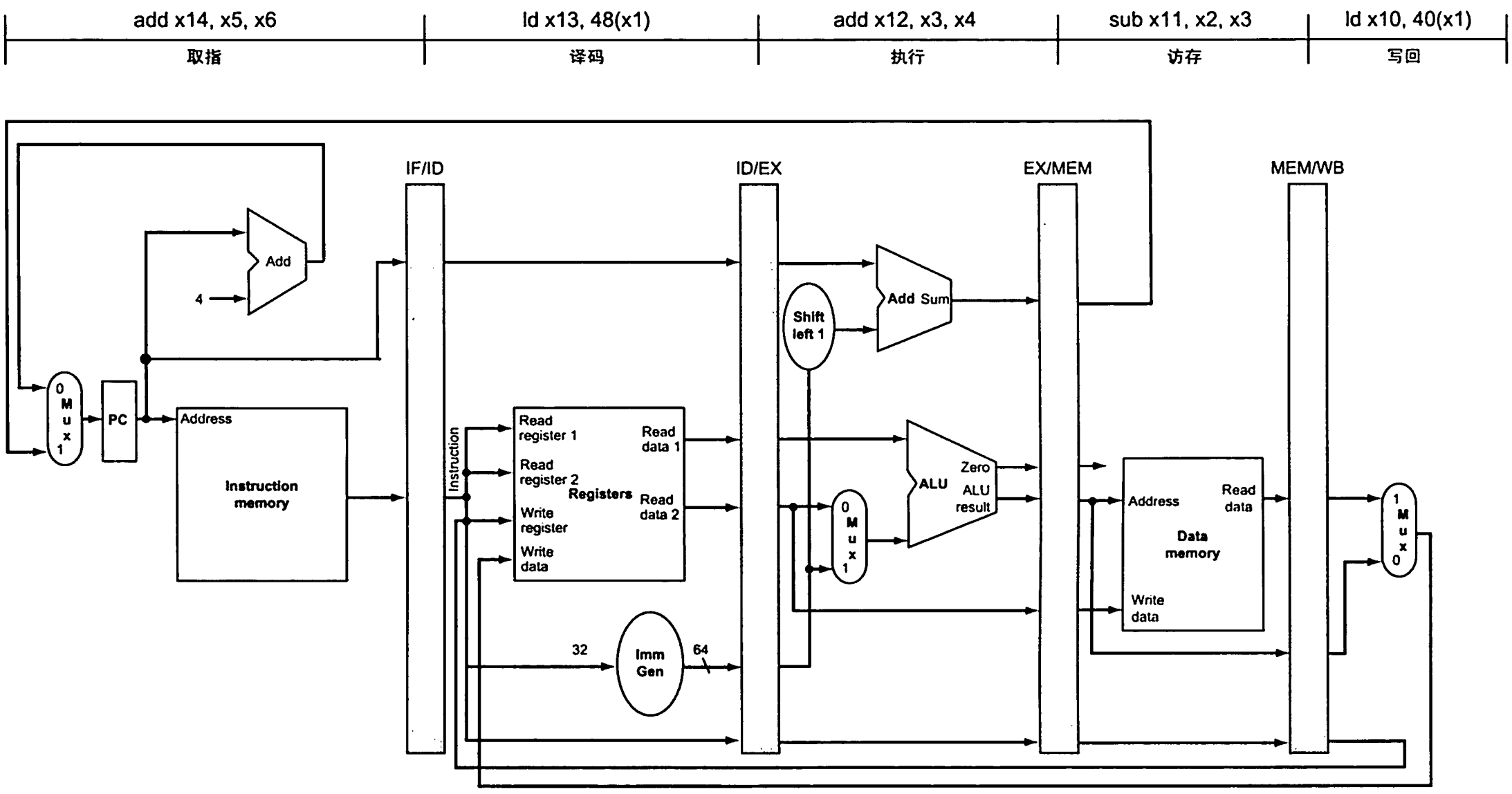


图 4-43　图 4-41 和图 4-42 中第五个时钟周期的单时钟周期图。从图中可以看出，单时钟周期图就是多时钟周期图中的一个垂直切片

自我检测　一群学生正在争论五阶段流水线的效率，其中一位学生指出并不是所有指令在每个阶段中都是活跃的。在决定忽略冒险的影响后，他们做出了以下四条陈述，请指出哪些是正确的。

1. 允许分支和 ALU 指令使用比加载指令所需的五级更少的流水线级数，这样做可以在所有情况下提升流水线性能。
2. 允许一些指令使用更少的时钟周期并不能提高性能，因为吞吐是由时钟周期决定的，每个指令所使用的流水线级数只影响延迟，并不影响吞吐。
3. 因为需要写回结果，因此 ALU 指令不能使用更少的时钟周期。但是分支指令则不需要写回，可以使用更少的时钟周期。因此提升性能的机会还是存在的。
4. 我们应该致力于使得流水线更长，而不是使指令使用更少的时钟周期。虽然这样做使得指令需要更多的时钟周期，但是每个周期变得更短了，这样就可以提升性能。

4.6.2 流水线控制

> 或许 CDC6600 计算机中的控制系统不同于之前所有的计算机。
> *James Tornton, Design of a Computer: The Control Data 6600, 1970*

正如我们在 4.4 节中将控制添加到单周期数据通路中那样，现在我们要将控制添加到流水线数据通路中。我们从一个简单的设计开始，从乐观的角度来看待这个问题。

第一步是在现有的数据通路上标记控制线。在图 4-44 中可以看到这些线。我们尽可能地借鉴图 4-17 中简单数据通路的控制逻辑，特别地，我们使用相同的 ALU 控制逻辑、分支逻辑和控制线，这些功能部件在图 4-12、图 4-16 和图 4-18 中定义。我们将图 4-45 ～图 4-47 放在一起并重现其中的关键信息，以便接下来讨论的内容更容易被理解。

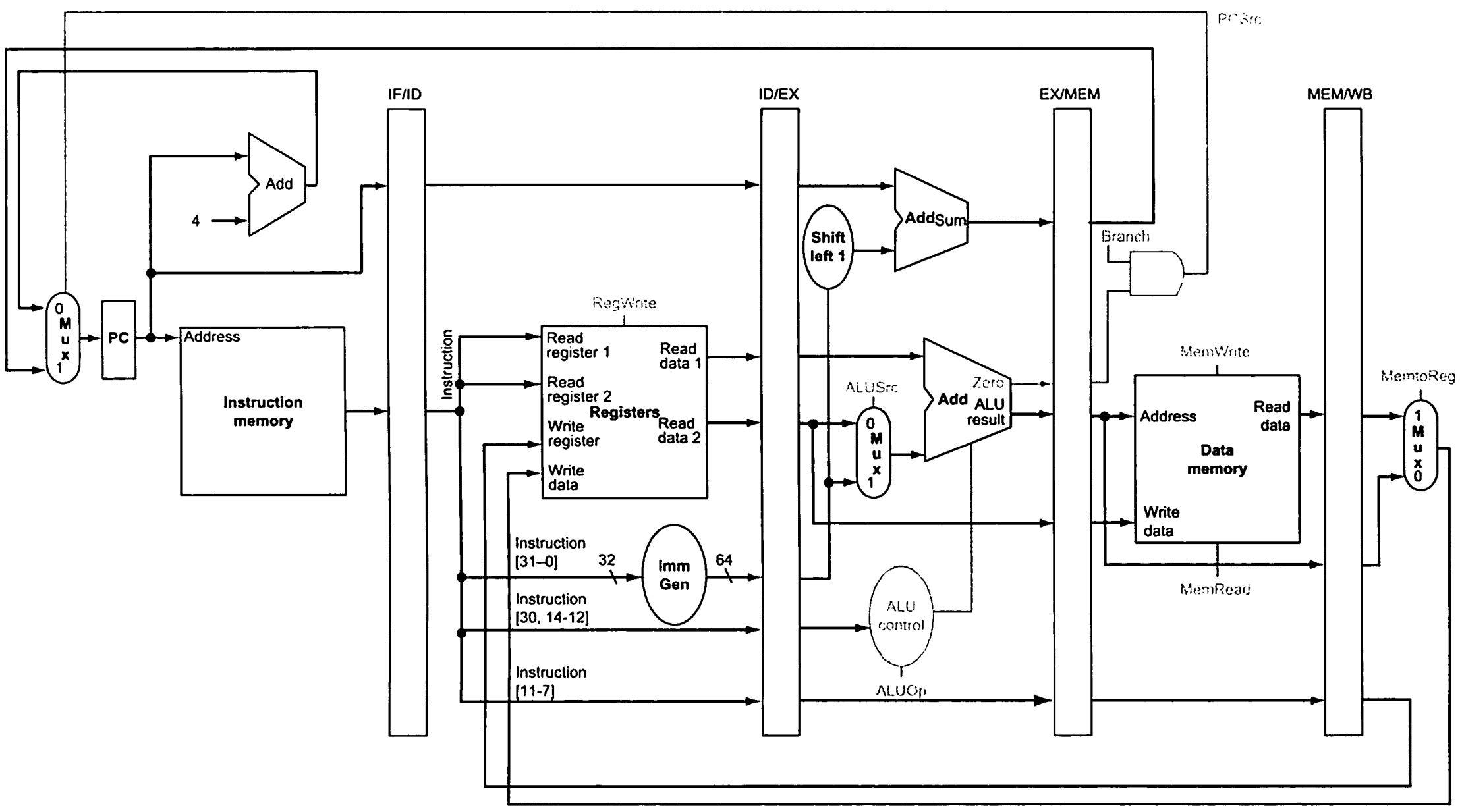


图 4-44 带有控制信号标记的图 4-39 中的流水线数据通路。这个数据通路借用了 4.4 节中的 PC 来源的控制逻辑、目标寄存器编号和 ALU 控制。请注意，我们现在需要 EX 阶段中指令的功能字段作为 ALU 控制的输入，因此这些位也必须包含在 ID/EX 流水线寄存器中

指令	ALUOp	操作	funct7字段	funct3字段	ALU操作行为	ALU控制输入
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

图 4-45　图 4-12 的副本。本图展示了如何根据 ALUop 控制位和不同的 R 型指令操作码来设置 ALU 控制位

信号名	无效时的效果	有效时的效果
RegWrite	无	被写的寄存器号来自Write register信号的输入，数据来自Write data信号的输入
ALUSrc	第二个ALU操作数来自第二个寄存器堆的输出（即Read data 2信号的输出）	第二个ALU操作数是指令的低12位符号扩展
PCSrc	PC值被adder的输出所替换，即PC+4的值	PC值被adder的输出所替换，即分支目标
MemRead	无	读地址由Address信号的输入指定，输出到Read data信号的输出中
MemWrite	无	写地址由Address信号的输入指定，写入内容是Write data信号的输入中的值
MemtoReg	寄存器写数据的输入值来自ALU	寄存器写数据的输入值来自数据存储器

图 4-46　图 4-16 的副本。定义了六个控制信号的功能。ALU 控制线（ALUop）定义在图 4-45 的第二列中。当一个二路选择器的控制位有效时，多选器选择与 1 相对应的输入；否则，如果控制位无效，多选器选择与 0 相对应的输入。需要注意的是，PCSrc 是由图 4-44 中的与门控制的，如果分支信号和 ALU 零信号都有效，则 PCSrc 为 1，否则为 0。控制仅在 beq 单元中才设置分支信号有效，其他时候 PCSrc 都为 0

指令	执行/地址计算阶段控制线		存储器访问阶段控制线			写回阶段控制线	
	ALUOp	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

图 4-47　根据流水线的最后三个阶段划分成三组的控制线，其值与图 4-18 中相同

与单周期实现的情况一样，我们假定 PC 在每个时钟周期被写入，因此 PC 没有单独的写入信号。同理，流水线寄存器（IF / ID、ID / EX、EX / MEM 和 MEM / WB）也没有单独的写入信号，因为流水线寄存器也在每个时钟周期内都被写入。

为了详细说明流水线的控制，我们需要在每个流水线阶段上设置控制值。由于每条控制线都只与一个流水线阶段中的功能部件相关，因此我们可以根据流水线阶段将控制线也划分

成五组。

1. 取指：读指令存储器和写 PC 的控制信号总是有效的，因此在这个阶段没有什么需要特别控制的内容。

2. 指令译码 / 读寄存器堆：在 RISC-V 指令格式中两个源寄存器总是位于相同的位置，因此在这个阶段也没有什么需要特别控制的内容。

3. 执行 / 地址计算：要设置的信号是 ALUOp 和 ALUSrc（见图 4-45 和图 4-46），这个信号选择 ALU 操作，并将读数据 2 或者符号扩展的立即数作为 ALU 的输入。

4. 存储器访问：本阶段要设置的控制线是 Branch、MemRead 和 MemWrite。这些信号分别由相等则分支、加载和存储指令设置。除非控制电路标示这是一条分支指令并且 ALU 的输出为 0，否则将选择线性地址的下一条指令作为 PCSrc 信号。

5. 写回：两条控制线是 MemtoReg 和 RegWrite，MemtoReg 决定是将 ALU 结果还是将存储器值发送到寄存器堆中，RegWrite 写入所选值。

由于流水线数据通路并没有改变控制线的意义，因此可以使用与单数据通路相同的控制值。图 4-47 中是和 4.4 节中相同的值，不过现在这七条控制线按照流水线阶段进行了分组。

实现控制意味着在每条指令的每个阶段中将这七条控制线设置为这些值。

由于控制线从 EX 阶段开始，我们可以在指令译码阶段为之后的阶段创建控制信号。传递这些控制信号最简单的方式就是扩展流水线寄存器以包含这些控制信息。图 4-48 显示，随着指令沿着流水线向下流动，这些控制信号被用于适当的流水线阶段，就像图 4-39 中目标寄存器编号随着加载指令在流水线中流动那样。图 4-49 显示了具有扩展流水线寄存器并且将控制线连接到相应流水线阶段的完整数据通路。（如果你想要了解更多细节，4.13 节以单时钟图表的形式给出了更多关于 RISC-V 代码在流水线硬件上的执行示例。）

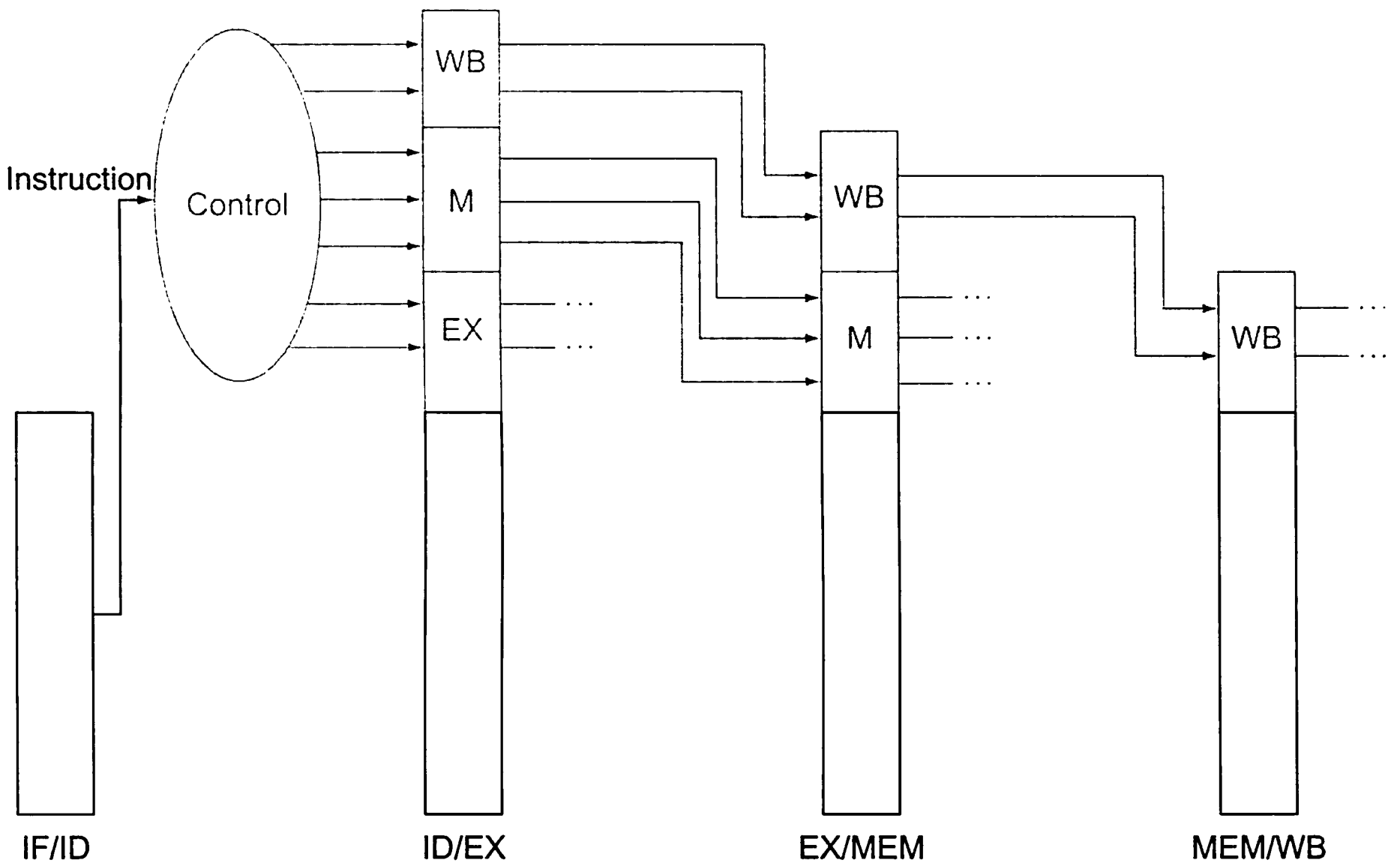


图 4-48 最后三个阶段的七条控制线。需要注意的是，在 EX 阶段使用了七条控制线中的两条，剩下的五条被传递到扩展的 EX/MEM 流水线寄存器中以保持控制线；在 MEM 阶段中使用了三条控制线，最后两条传递到 MEM/WB 寄存器用于 WB 阶段

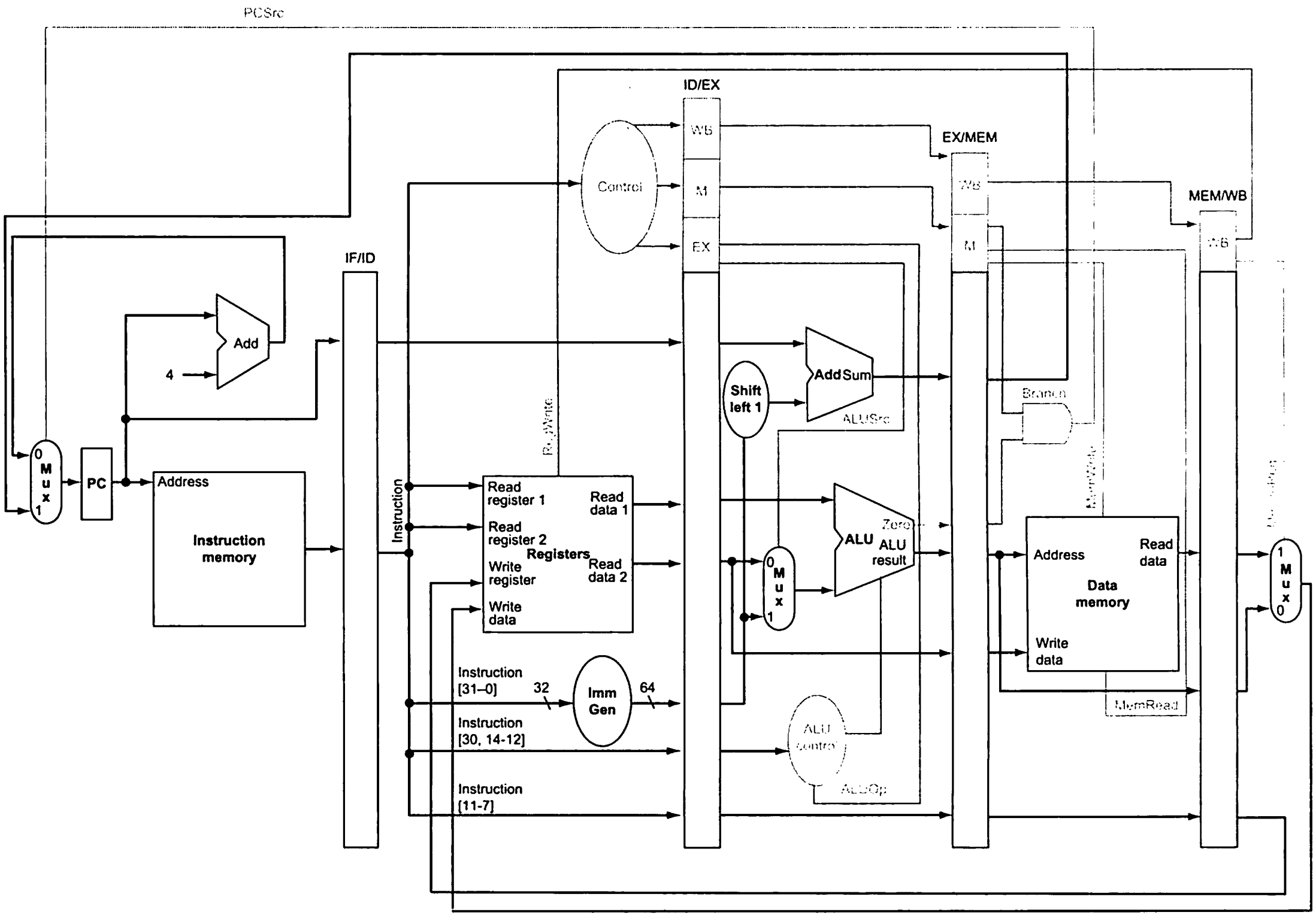


图 4-49 图 4-44 中的流水线数据通路，将控制信号连接到流水线寄存器中的控制部分。在指令译码阶段创建之后三个阶段的控制值，然后将其置于 ID/EX 流水线寄存器中。流水线每个阶段使用相应的控制线，其余的控制线被传递到下一个流水线阶段中

4.7 数据冒险：前递与停顿

> 你是什么意思，你问我为什么要建立旁路？因为这是旁路。你必须建立旁路。
> *Douglas Adams, The Hitchhiker's Guide to the Galaxy, 1979*

上一节的示例中展示了流水线的强大功能以及硬件如何通过流水线的方式执行任务。现在从一个更实际的例子出发，看看在程序真正执行的时候会发生什么。图 4-41 ～图 4-43 中的 RISC-V 指令是相互独立的，没有用到其他任何指令计算的结果。然而，在 4.5 节中，我们已经看到数据冒险是流水线执行的阻碍。

现在来看一个存在更多相关的指令序列，相关关系以灰色显示：

```
sub   x2, x1, x3    // Register z2 written by sub
and   x12, x2, x5   // 1st operand(x2) depends on sub
or    x13, x6, x2   // 2nd operand(x2) depends on sub
add   x14, x2, x2   // 1st(x2) & 2nd(x2) depend on sub
sd    x15, 100(x2)  // Base (x2) depends on sub
```

后四条指令（and、or、add、sd）都相关于第一条指令（sub）中得到的存放在寄存器 x2 中的结果。假设寄存器 x2 在 sub 指令执行之前的值为 10，在执行之后值为 −20，那么程序员希望在后续指令中引用寄存器 x2 时得到的值为 −20。

这个指令序列在流水线中是如何执行的？图 4-50 使用了多时钟周期流水线表示来说明这些指令的执行。为了在我们当前的流水线中演示这组指令序列的执行，图 4-50 顶部显示了寄存器 x2 的值，这个值在第五个时钟周期被改变，此时 sub 指令写回指令的执行结果。

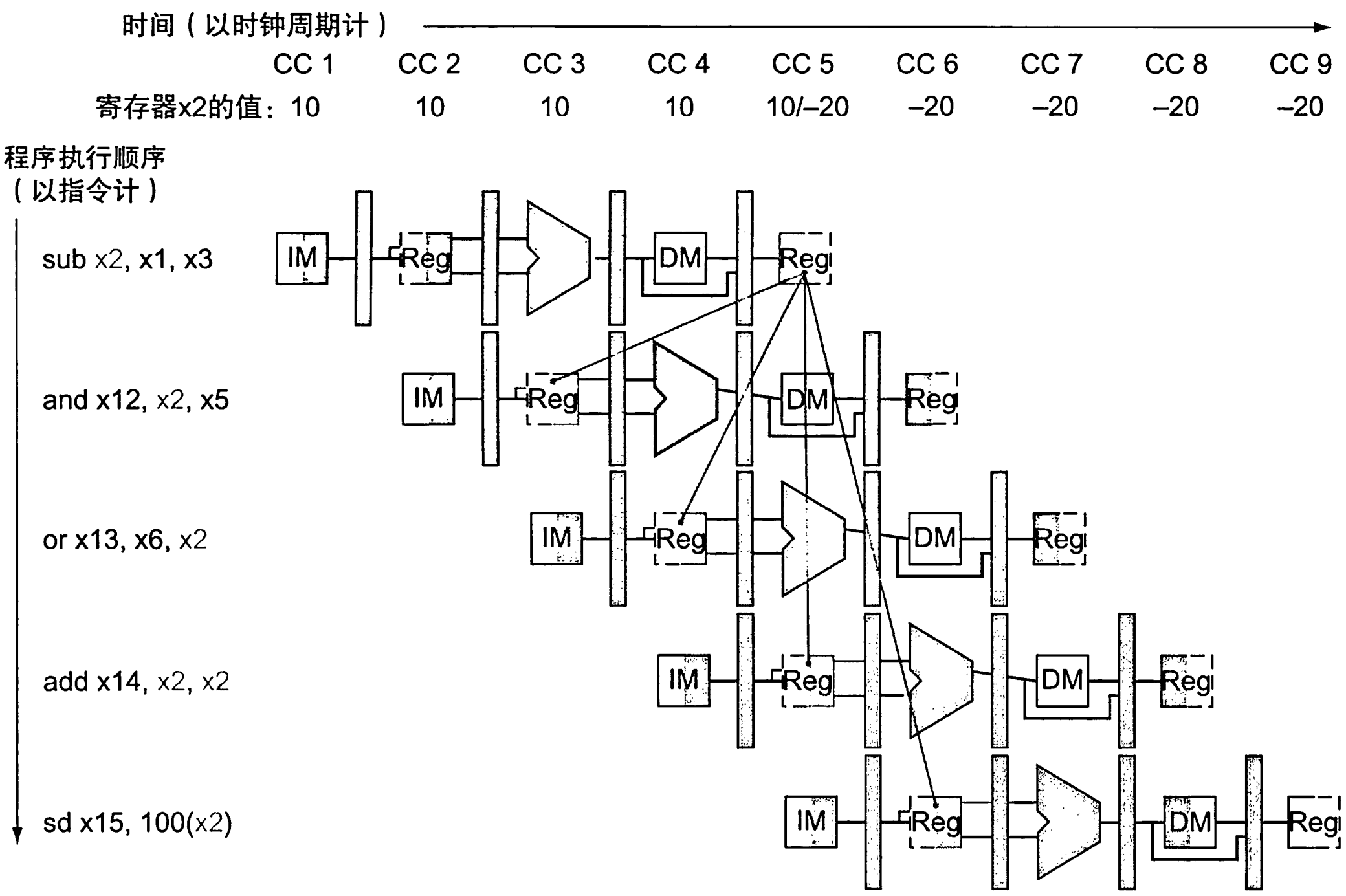


图 4-50 使用简化的数据通路表示具有相关性的五条指令序列中的流水线相关关系。所有的相关操作在图中都以灰线显示，图中顶部的“CC1”表示第一个时钟周期。第一条指令向 x2 中写入数据，后续的指令都是从 x2 中读取数据。x2 寄存器在第五个时钟周期时被写入，因此在第五个时钟周期前正确的寄存器值是不可用的。（当这样的写操作发生时，本时钟周期内读寄存器的值返回本周期内前半个周期末被写入的值。）从数据通路顶部到底部的灰色线表示相关关系。那些导致时间后退的相关就是流水线数据冒险

最后一个潜在的冒险可以通过寄存器堆的硬件设计来解决：当一个寄存器在同一个时钟周期内既被读取又被写入时会发生什么？我们假定写操作发生在一个时钟周期的前半部分，而读操作发生在后半部分。所以读操作会得到本周期内被写入的值。这种假定与很多寄存器堆的实现是一致的。在这种情况下不会发生数据冒险。

如图 4-50 所示，在第五个时钟周期之前，对寄存器 x2 的读操作并不能返回 sub 指令的结果。因此，图中的 add 和 sd 指令可以得到正确结果 –20，但是 and 和 or 指令却会得到错误的结果 10。在这种类型的图中，每当相关线在时间线上表示为后退时（箭头指向左上方），这个问题就会变得很明显。

正如 4.5 节中提到的那样，在第三个时钟周期也就是 sub 指令的 EX 指令阶段结束时就可以得到想要的结果。那么在 and 和 or 指令中是什么时候才真正需要这个数据呢？答案是在 and 和 or 指令的 EX 阶段开始的时候，分别对应第四和第五个时钟周期。因此，只要可以一得到相应的数据就将其前递给等待该数据的单元，而不是等待其可以从寄存器堆中读取出来，就可以不需要停顿地执行这段指令了。

前递是怎样工作的？在本节的余下部分，为了简化内容，我们只考虑如何解决将 EX 阶段产生的操作数前递出去的问题，该数据可能是 ALU 或是有效地址的计算结果。这意味着当一个指令试图在 EX 阶段使用的寄存器是一个较早的指令在 WB 阶段要写入的寄存器时，

我们需要将该数据作为 ALU 的输入。

命名流水线寄存器字段是一种更精确的表示相关关系的方法。例如，ID/EX. RegisterRs1 表示一个寄存器的编号，它的值在流水线寄存器 ID.EX 中，也就是这个寄存器堆中第一个读端口的值。该名称的第一部分，也就是点号的左边，是流水线寄存器的名称；第二部分是寄存器中字段的名称。使用这种表示方法，可以得到两对冒险的条件：

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1

1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd = ID/EX. RegisterRs2

在本节开头的代码中，指令序列中的第一个冒险发生在寄存器 x2 上，位于 sub 指令 sub x2, x1, x3 的结果和 and 指令 and x12, x2, x5 的第一个读操作数之间。这个冒险可以在 and 指令位于 EX 阶段、sub 指令位于 MEM 阶段时被检测到，因此这种冒险属于 1a 类型：

EX/MEM.RegisterRd = ID/EX.RegisterRs1 = x2

例题 | 相关性检测

将本节开头的指令序列的相关性进行分类：

```
sub x2, x1, x3       // Register x2 set by sub
and x12, x2, x5      // 1st operand(z2) set by sub
or  x13, x6, x2      // 2nd operand(x2) set by sub
add x14, x2, x2      // 1st(x2) & 2nd(x2) set by sub
sd  x15, 100(x2)     // Index(x2) set by sub
```

答案 | 正如上文中所提到的，在 sub 指令和 and 指令之间存在类型为 1a 的冒险。其余的冒险类型分别是：

- sub 指令和 or 指令之间存在类型为 2b 的冒险：

  MEM/WB.RegisterRd = ID/EX.RegisterRs2 = x2

- 在 sub 指令和 add 指令之间的两个相关性都不是冒险，因为在 add 指令的 ID 阶段寄存器堆已经可以提供 x2 的正确值了。
- 在 sub 指令和 sd 指令之间不存在数据冒险，因为 sd 指令在 sub 指令将结果写回至 x2 之后才读取 x2 的值。

因为并不是所有的指令都会写回寄存器，所以这个策略是不正确的，它有时会在不应该前递的时候也将数据前递出去。一种简单的解决方案是检查 RegWrite 信号是否是有效的：检查流水线寄存器在 EX 和 MEM 阶段的 WB 控制字段以确定 RegWrite 信号是否有效。回忆一下，RISC-V 要求每次使用 x0 寄存器作为操作数时必须认为该操作值为 0。如果流水线中的指令以 x0 作为目标寄存器（例如 addi x0, x1, 2 这条指令），我们希望避免前递非零的结果值。不前递以 x0 为目标寄存器的结果可以使得汇编程序员和编译器不需要考虑将 x0 作为目标寄存器的情况。只要我们将 EX/MEM.RegisterRd ≠ 0 添加到第一类冒险条件，并将 MEM/WB.RegisterRd ≠ 0 添加到第二类冒险条件中，就可以使得上述条件正常工作。

现在我们可以检测冒险了。一半的问题已经解决了，剩下一半的问题是前递正确的数据。

图 4-51 的代码序列与图 4-50 中的相同，显示了流水线寄存器和 ALU 输入之间的相关性。不同的是，图 4-51 中的相关性开始于流水线寄存器，而不是等待 WB 阶段写入寄存器堆。因此，只要流水线寄存器保存了将要被前递的数据，后续的指令就可以得到所需的数据。

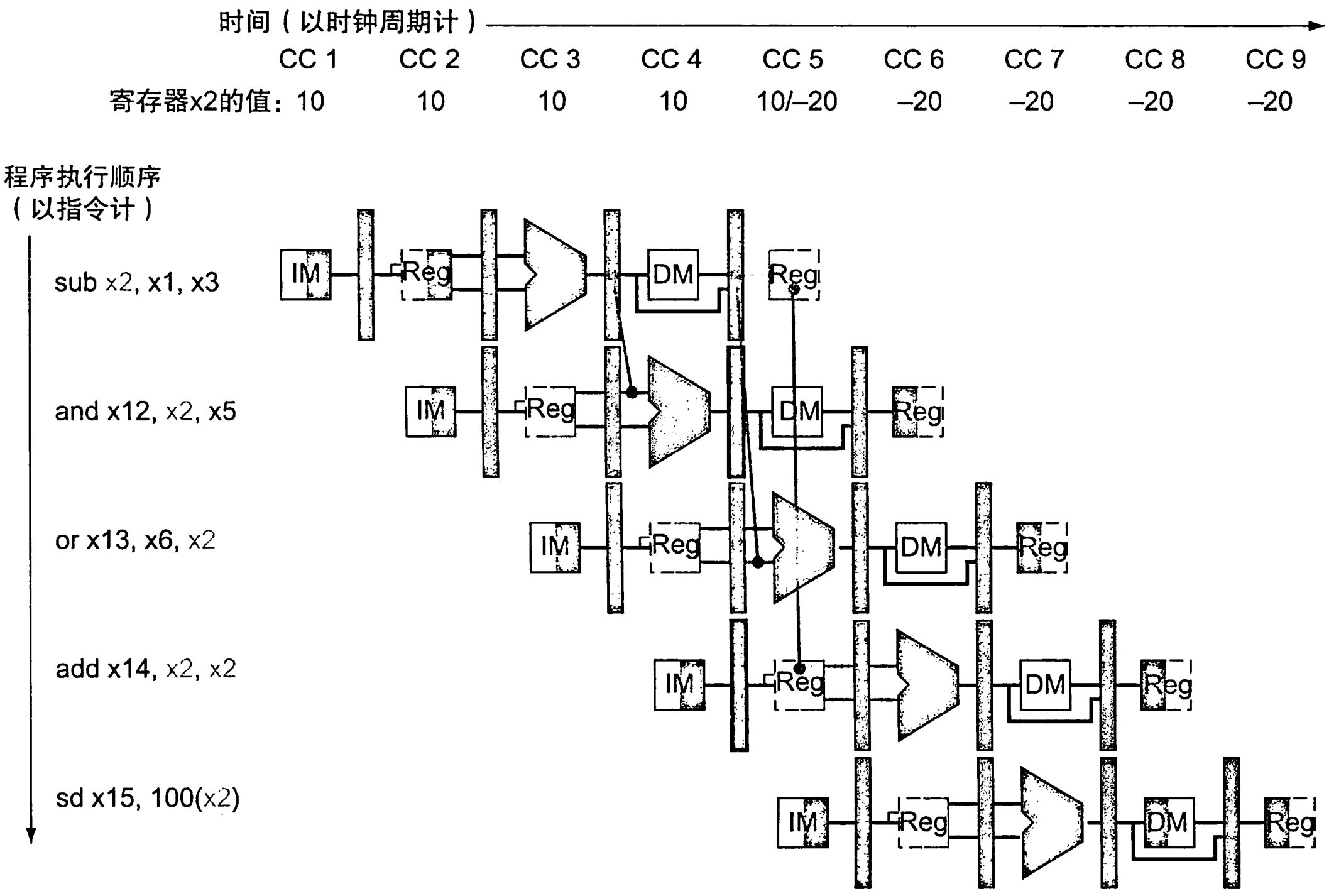
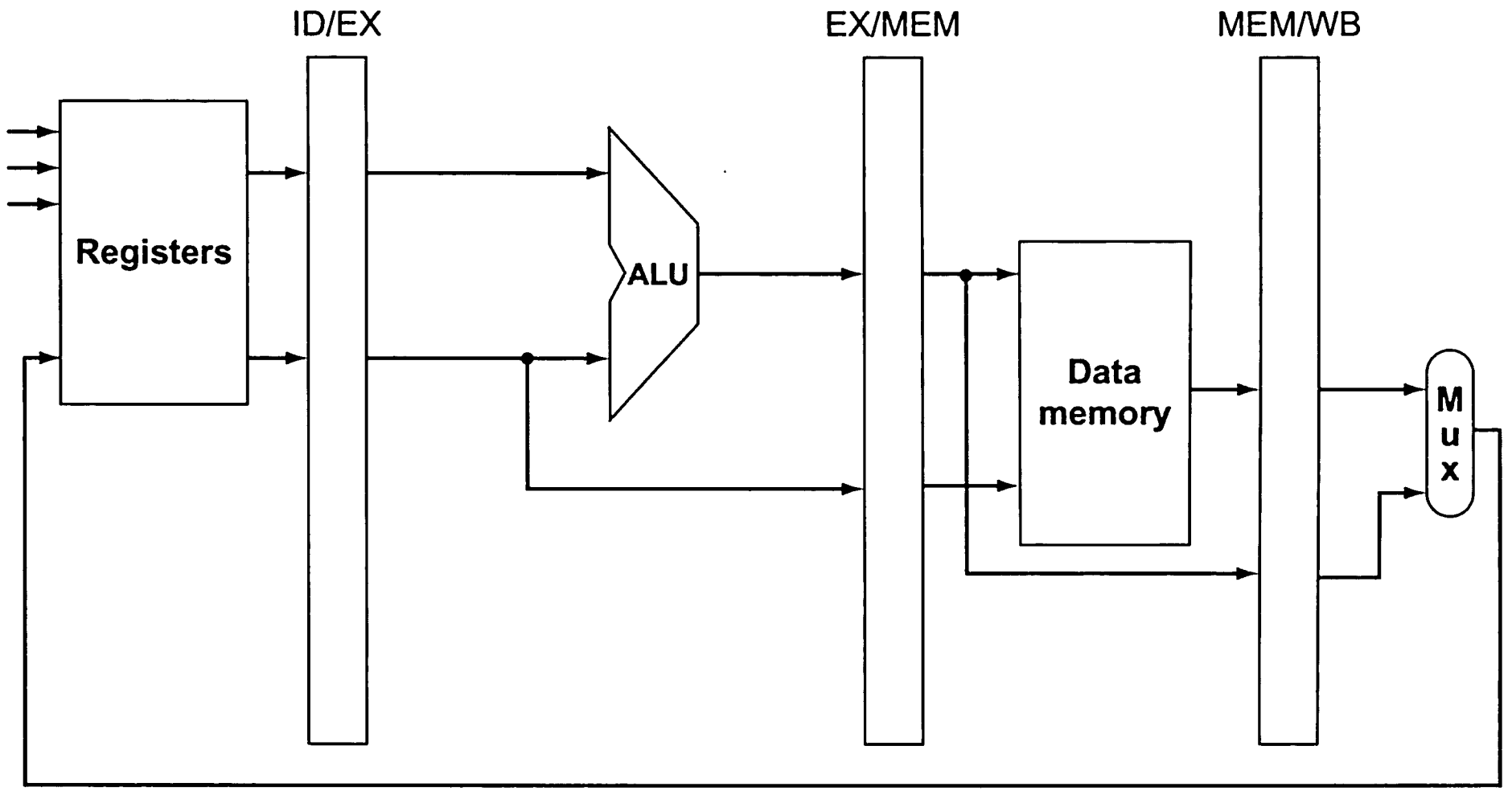


图 4-51 各流水线寄存器之间的相关关系会随着时间向前移动，因此可以通过前递在流水线寄存器中找到的结果，以提供 and 指令或 or 指令所需的 ALU 的输入。流水线寄存器中的值表示所需的值在被写入寄存器堆之前就是可用的。我们假设寄存器堆可以前递在同一时钟周期内要被读写的数据，这样 add 指令就不需要停顿了，不过这些值来自流水线寄存器而不是寄存器堆。寄存器堆前递，即读操作获得的值是本时钟周期内写操作的结果，这就是为什么第五个时钟周期中显示寄存器 x2 在前半个周期内的值为 10 而在周期结束时的值为 −20

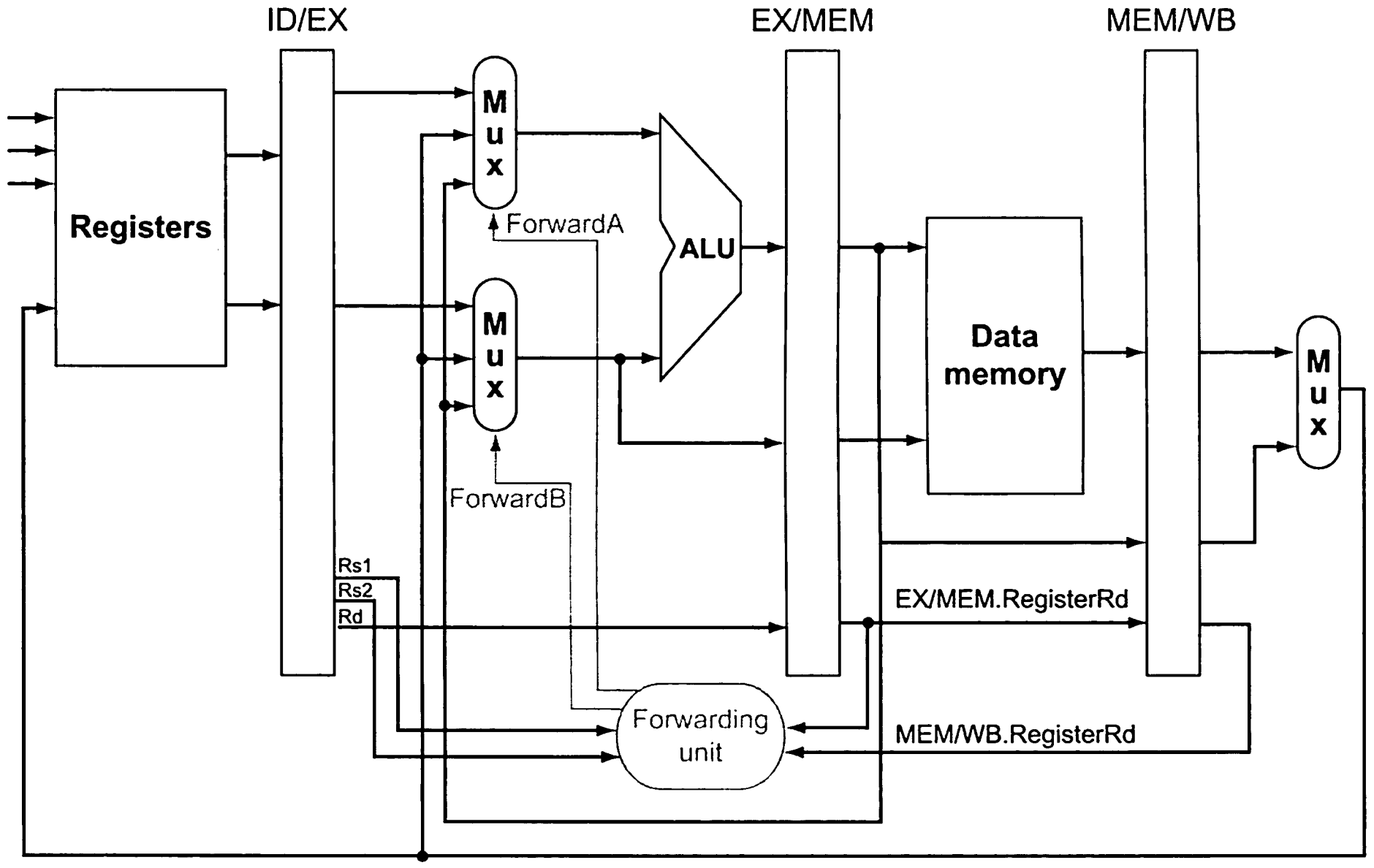
如果我们可以从任何流水线寄存器而不仅仅是 ID/EX 中得到 ALU 的输入，那就可以前递正确的数据。通过在 ALU 的输入上添加多选器再辅以适当的控制，就可以在存在数据冒险的情况下全速运行流水线。

现在，假设需要前递的指令只有这四种形式：add、sub、and 和 or 指令。图 4-52 是 ALU 和流水线寄存器在添加前递之前和之后的“特写”。图 4-53 是 ALU 多选器的控制线的值，它选择寄存器堆的值或是被前递的值中的一个。

这个前递控制将发生在 EX 阶段，因为 ALU 前递多选器在 EX 阶段。因此，我们必须在 ID 阶段通过 ID/EX 流水线寄存器将操作数寄存器编号传递出去，以决定是否需要前递值。在加入前递机制之前，ID/EX 流水线寄存器无须保存 rs1 字段和 rs2 字段，但是因为前递机制的需要，现在要将保存 rs1 和 rs2 所需的空间添加到 ID/EX 流水线寄存器中。



a）添加前递前



b）添加前递后

图 4-52　添加前递前后的 ALU 和流水线寄存器。上图未添加前递，下图将多选器扩展为前递路径，并在图中显示前递单元。新的硬件在图中以灰色显示。本图只是一张示意图，没有标注诸如符号扩展硬件这样的完整数据通路中的细节

多选器控制	源	解释
ForwardA = 00	ID/EX	ALU的第一个操作数来自寄存器堆
ForwardA = 10	EX/MEM	ALU的第一个操作数来自上一个ALU计算结果的前递
ForwardA = 01	MEM/WB	ALU的第一个操作数来自数据存储器或者更早的ALU计算结果的前递
ForwardB = 00	ID/EX	ALU的第二个操作数来自寄存器堆
ForwardB = 10	EX/MEM	ALU的第二个操作数来自上一个ALU计算结果的前递
ForwardB = 01	MEM/WB	ALU的第二个操作数来自数据存储器或者更早的ALU计算结果的前递

图 4-53　图 4-52 中多选器的控制值。ALU 的另一个输入符号扩展的立即数会在本节最后的拓展阅读中说明

现在给出检测冒险的条件以及解决相应冒险的控制信号：

1. EX 冒险

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

这种情况是将前一条指令的结果前递到任何一个 ALU 的输入中。如果前一条指令想要写寄存器堆，并且将要写的寄存器编号与 ALU 输入口 A 或 B 要读取的寄存器编号一致（前提是该寄存器编号不为 0），那么就控制多选器直接从 EX/MEM 流水线寄存器中取值。

2. MEM 冒险

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

正如上文所述，在 WB 阶段不存在冒险，因为我们假定在 ID 阶段的指令读取的寄存器与 WB 阶段要写入的寄存器相同时，寄存器堆能够提供正确的结果。也就是说，寄存器堆提供了另外一种形式的前递，只不过这种前递发生在寄存器堆内部。

一种复杂的潜在数据冒险是在 WB 阶段指令的结果、MEM 阶段指令的结果和 ALU 阶段指令的源操作数之间发生的。例如，在一个寄存器中对一组数据做求和操作时，一系列的指令将会读和写一个相同的寄存器：

```
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
. . .
```

在这种情况下，结果应该是来自 MEM 阶段前递的数据，因为 MEM 阶段中的结果就是最近的结果。因此，MEM 冒险的控制应该是（在下文中以灰色标注）：

```
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.RegisterRs1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) 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.RegisterRs2))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

图 4-54 显示了为了支持前递 EX 阶段的结果，这个操作所需要添加的硬件。注意 EX/MEM.RegisterRd 字段是 ALU 指令或者加载指令的目标寄存器。

如果你想查看更多使用单时钟周期流水线绘制方式的示例说明，4.13 节中给出了两份存在需要前递的冒险的 RISC-V 代码。

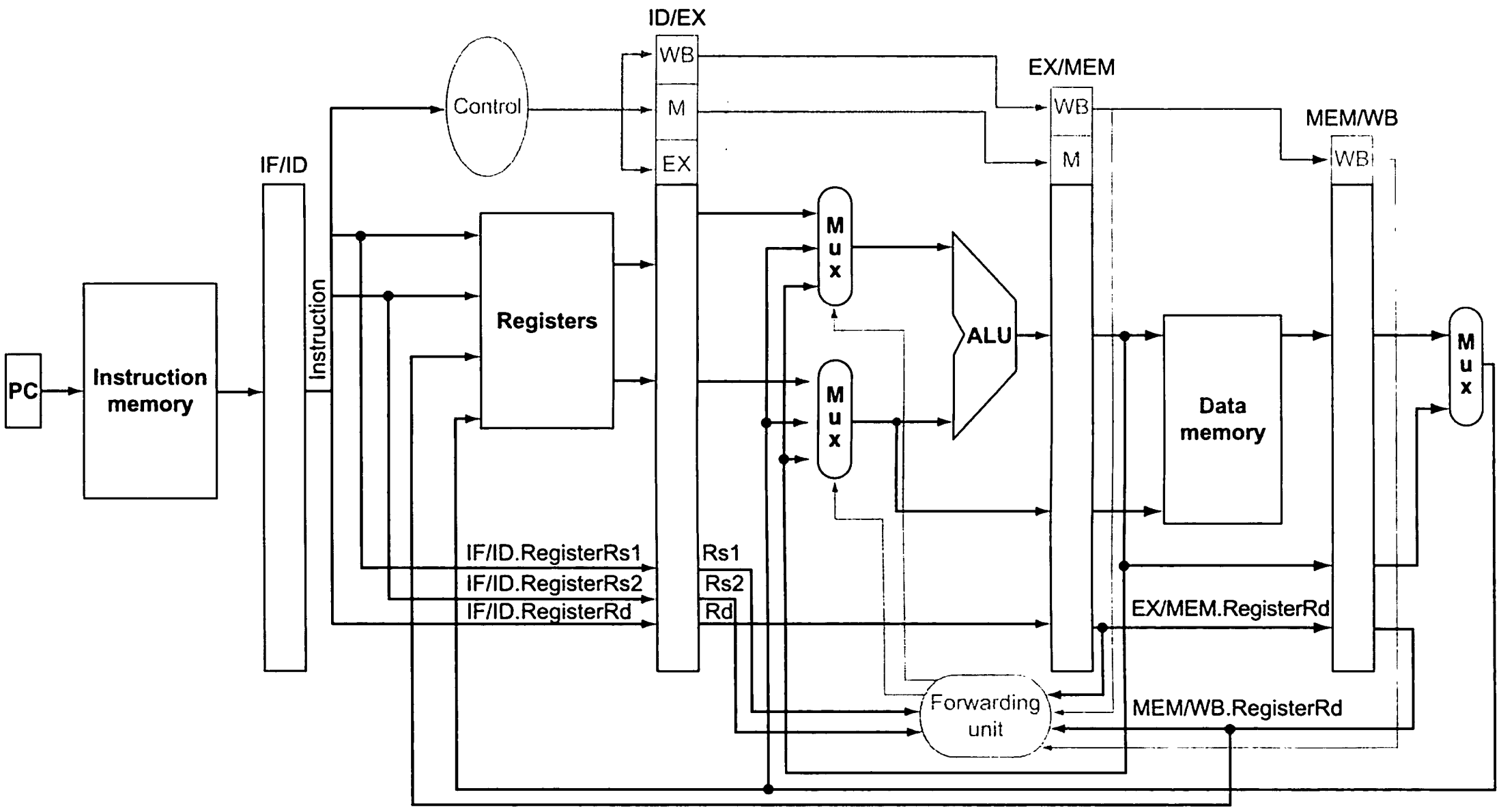


图 4-54 通过前递解决冒险的数据通路。与图 4-49 中的数据通路相比，图 4-54 在 ALU 的输入上添加了多选器。本图只是一张示意图，没有标注诸如分支硬件和符号扩展硬件这样的完整数据通路中的细节

详细阐述 前递还有助于解决当存储指令与其他指令相关时造成的冒险。因为 sd 指令在 MEM 阶段只使用一个数据值，因此设置前递是很简单的。然而，需要考虑加载指令紧跟在存储指令之后的情况，这在 RISC-V 架构中执行内存之间的拷贝操作时非常有用。因为复制是频繁的，所以我们需要添加更多的前递硬件使其运行得更快。如果重绘图 4-51，将 sub 指令和 and 指令替换为 ld 和 sd 指令，我们会发现避免一次停顿是可能的，因为 sd 指令在 MEM 阶段所使用的数据会及时保存在 ld 指令的 MEM/WB 寄存器中。为了实现这个操作，需要在存储器访问阶段加入前递。我们将这个修改作为练习留给读者。

此外，ld 和 sd 指令所需的一种作为 ALU 输入的符号扩展立即数，在图 4-54 中的数据通路里没有画出。因为寄存器和立即数之间的判定是由中央控制判断的，而前递单元选择流水线寄存器中的一个作为 ALU 的一个寄存器输入，最简单的解决方案就是加入一个二选一的多选器，在 ForwardB 多选器输出和符号扩展立即数之间做选择。图 4-55 中展示了这个添加的部件。

数据冒险与停顿

> 如果你在开始的时候没有成功，那你可以重新定义成功的意义。
> *佚名*

正如 4.5 节中提到的那样，当一条指令试图在加载指令写入一个寄存器之后读取这个寄存器时，前递不能解决此处的冒险。图 4-56 说明了这个问题。在第 4 个时钟周期时，数据还正在存储器中被读取，但此时 ALU 已经在为下一条指令执行操作了。当加载指令后跟着一条需要读取加载指令结果的指令时，流水线必须被阻塞以消除这种指令组合带来的冒险。

因此，除了一个前递单元外，还需要一个冒险检测单元。该单元在 ID 流水线阶段操作，从而可以在加载指令和相关加载指令结果的指令之间加入一个流水线阻塞。这个单元检测加载指令，冒险控制单元的控制逻辑满足如下条件：

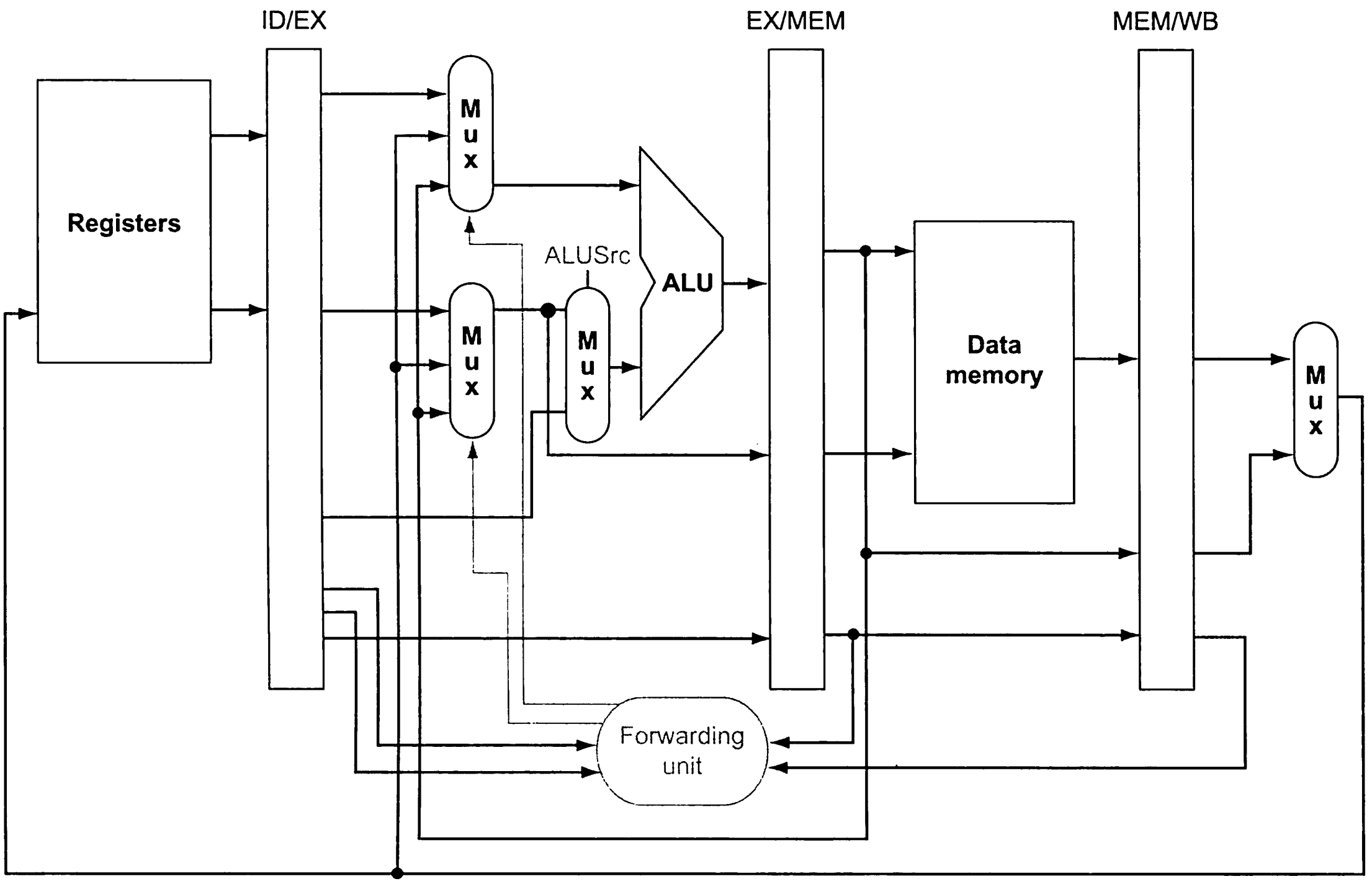


图 4-55 图 4-52 中数据通路的“特写”，展示了一个二选一多选器，它被加入图中以选择符号扩展立即数作为 ALU 的输入

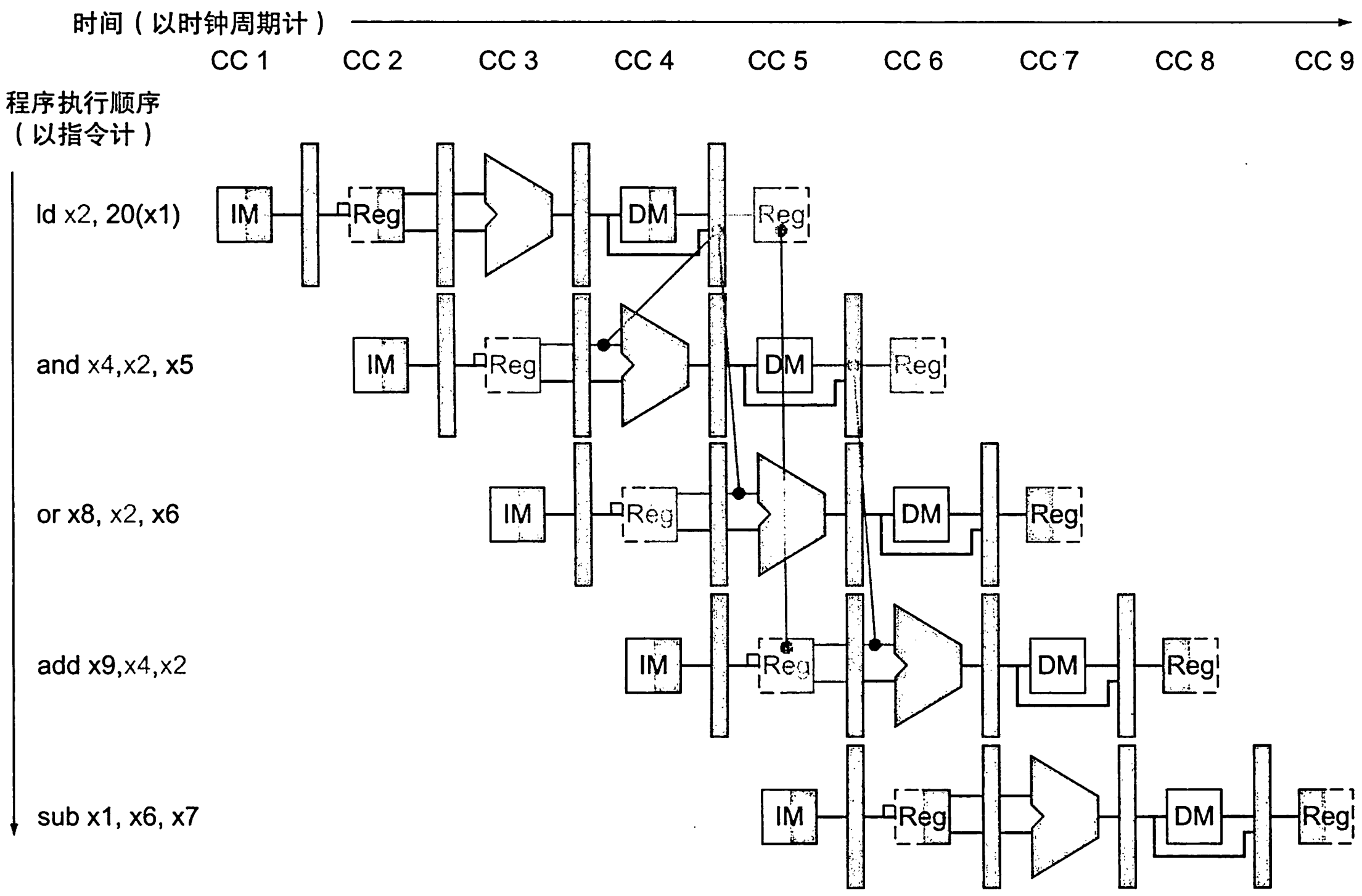


图 4-56 一个流水化的指令序列。因为 ld 指令和 and 指令之间的相关性在时间上是逆向的，所以这种冒险不能使用前递来解决。因此，这种指令组合会导致冒险检测单元产生一个停顿

```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
      stall the pipeline
```

回想一下，我们在加载指令和R型指令中使用RegisterRd也就是指令的7至11位（11:7）表示指定的寄存器。第一行测试是为了查看指令是否是加载指令：只有加载指令需要读取数据存储器。接下来的两行检测在EX阶段的加载指令的目标寄存器是否与ID阶段的指令中的某一个源寄存器相匹配。如果条件成立，指令会停顿一个时钟周期。在一个时钟周期后，前递逻辑就可以处理这个相关并继续执行程序了。（如果没有前递，那么图4-56中的指令还需要再停顿一个时钟周期。）

如果处于ID阶段的指令被停顿了，那么在IF阶段中的指令也一定要被停顿，否则已经取到的指令就会丢失。只需要简单地禁止PC寄存器和IF/ID流水线寄存器的改变就可以阻止这两条指令的执行。如果这些寄存器被保护，在IF阶段的指令就会继续使用相同的PC值取指令，同时在ID阶段的寄存器就会继续使用IF/ID流水线寄存器中相同的字段读寄存器。再回到我们的洗衣例子中，这就像是你重新开启洗衣机洗相同的衣服并且让烘干机继续空转一样。当然，就像烘干机那样，EX阶段开始的流水线后半部分必须执行没有任何效果的指令，也就是**空指令**。

空指令：一种不执行任何操作、不改变任何状态的指令。

如何在流水线中插入空指令（就像气泡那样）呢？从图4-47中可知，解除EX、MEM和WB阶段的七个控制信号（将它们设置为0）就可以产生一个“没有任何操作”的指令，也就是空指令。通过识别ID阶段的冒险，我们可以通过将ID/EX流水线寄存器中EX、MEM和WB的控制字段设置为0来向流水线中插入一个气泡。这些不会产生负面作用的控制值在每个时钟周期向前传递并产生适当的效果：在控制值均为0的情况下，不会有寄存器或者存储器被写入数据。

图4-57显示了硬件中的具体实现细节：and指令所在的流水线执行槽变成了nop指令，并且所有在and指令之后的指令都被延后了一个时钟周期。就像水管中出现了一个气泡那样，这个停顿气泡延后了它之后的所有指令的执行，并且随着每个时钟周期沿着流水线继续前进，直到其退出流水线。在本例中，这个冒险使得and指令和or指令在第4个时钟周期内重复了它们在第3个时钟周期内做过的事情：and指令读寄存器和解码，or指令从指令存储器中重新取了一遍指令。这种重复看起来就像是停顿一样，它的影响是拉伸了and指令和or指令，并且延后了取第2个and指令的时间。

图4-58中高亮显示了流水线中冒险检测单元和前递单元之间的连接。和原来一样，前递单元控制ALU多选器，用相应的流水线寄存器中的值替换通用寄存器中的值。冒险检测单元控制PC和IF/ID流水线寄存器的写入，以及在实际控制值和全0之间选择的多选器。如果加载–使用冒险被检测为真，则冒险检测单元会停顿并清除所有控制字段。如果想要了解更多细节，4.13节中给出了一个RISC-V代码示例，用于说明在单时钟周期流水线图中造成停顿的冒险。

重点 尽管编译器通常依赖于硬件来解决冒险并保证指令正确执行，但编译器仍然需要理解流水线以获得最优性能。否则，未预料到的停顿就会降低编译后代码的性能。

详细阐述 上文中提到将控制线置为0以避免写入寄存器或存储器：事实上，只要将RegWrite和MemWrite信号设置为0即可，不需要考虑其他的控制信号的值。

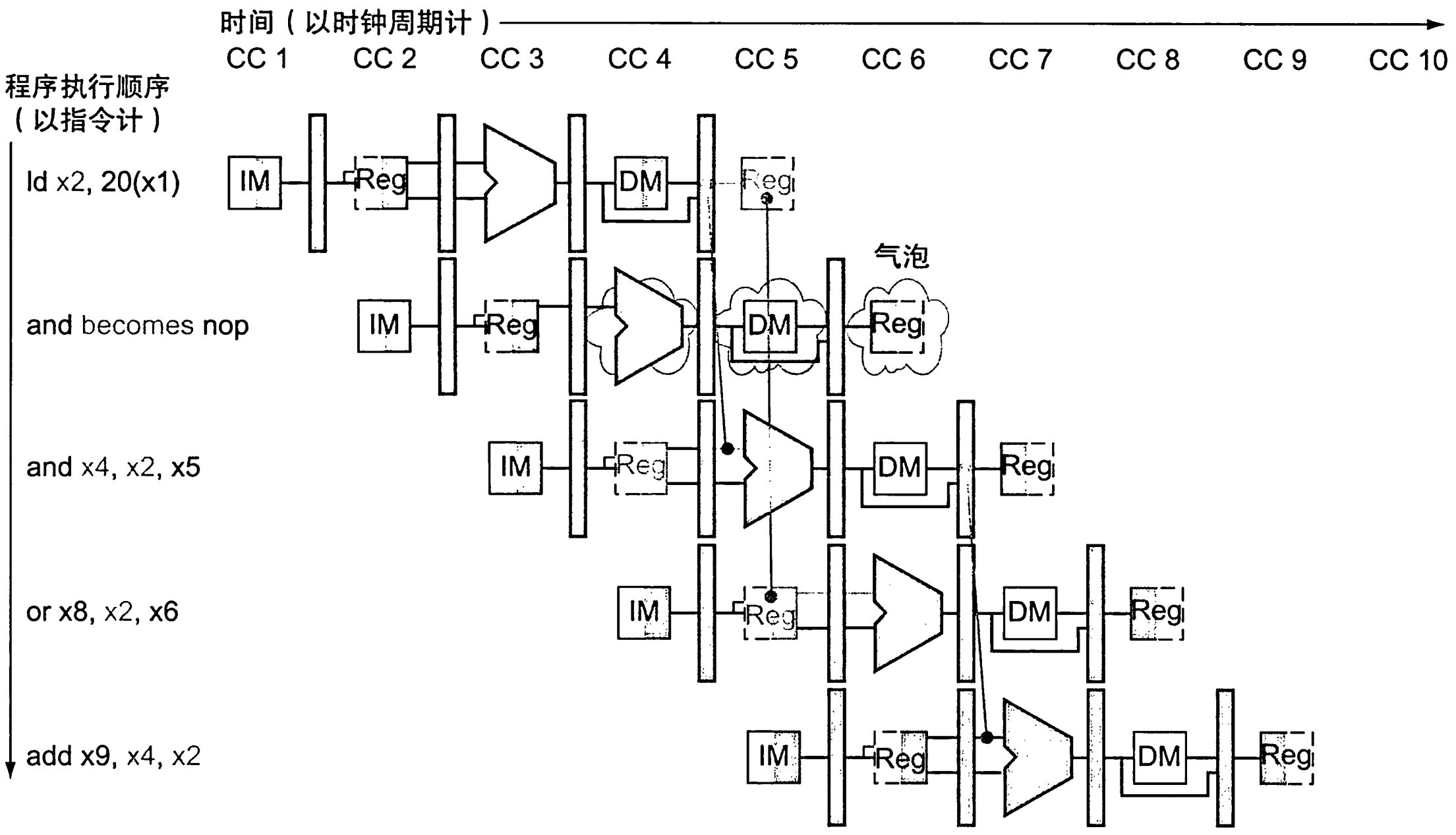


图 4-57 在流水线中插入停顿的方法。通过将 and 指令替换成 nop，在第 4 个时钟周期中插入了一个停顿。需要注意的是 and 指令在第 2 和第 3 个时钟周期内被译码和解码，但它的 EX 阶段被延后到第 5 个时钟周期中（如果没有停顿，EX 阶段应该发生在第 4 个时钟周期中。）相应的，or 指令在第 3 个时钟周期被译码，但它的 ID 阶段被延后到第 5 个时钟周期中（如果没有停顿，ID 阶段应该发生在第 4 个时钟周期中。）在插入气泡后，所有的相关性沿着时间轴继续向前，但是不会再发生冒险了

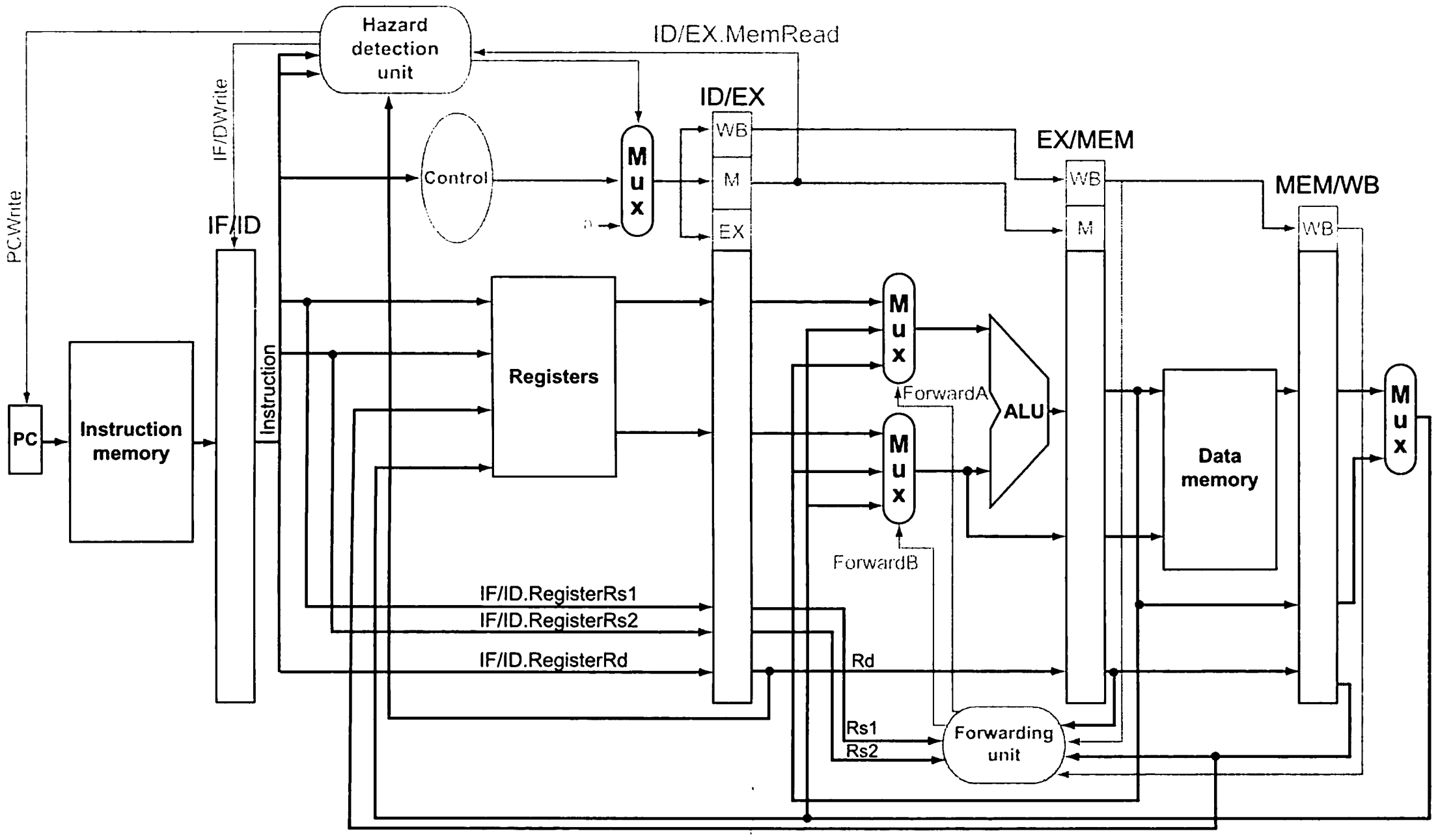


图 4-58 流水线控制图概览，图中包括两个前递多选器、一个冒险检测单元和一个前递单元。尽管简化了 ID 和 EX 阶段（图中省略了符号扩展立即数和分支逻辑），但是本图还是说明了前递所需的最基本的硬件