4.4. 流水线冒险

流水线会出现在下一个时钟周期中的下一条指令不能执行的情况，称为**冒险（hazard）**。

流水冒险发生在指令间的冲突，包括访问同一资源（结构冒险）和使用未准备好的数据/地址（数据冒险/控制冒险）。后者表现为数据流在时间轴上逆向访问，会产生数据冒险。如 Figure 4-32 所示，在单条指令从左到右的数据流中有两个例外：WB 级，将结果写回寄存器堆；MEM 级确定下一个 PC 时，需要在计算后返回的分支地址和 PC+4 中选择。前者会导致数据冒险，后者会导致控制冒险。

冗余是解决冒险的一种方法，可以在硬件空间上冗余，也可以在执行时间上冗余（阻塞）。最普通的解决方法就是**流水线阻塞（pipeline stall）**，或称为**气泡（bubble）**。没有任何干预情况下，等待前面的冲突执行完毕后再执行其他指令，会浪费一定的时钟周期数。所以阻塞不是好办法，它会拖慢整个指令序列的执行过程，使处理器性能降低。因此，流水控制逻辑必须检测冒险，然后采取一定的策略解决冒险。

4.4.1. 结构冒险

结构冒险（structural hazard）是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。本章的流水线处理器中存在的结构冒险有两种：内存访问冲突和寄存器冲突，如所示。

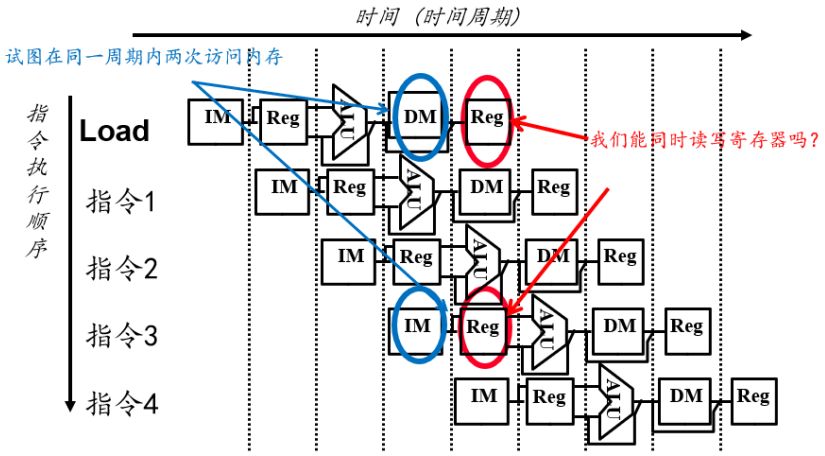

Figure 4-44 Structural Hazard in Registers (red) and Memory (blue)

结构冒险（基本上）总是可以通过增加硬件资源的方式加以解决。对于内存访问冲突，可以使用两块 Cache 解决，一块存储指令，一块存储数据；对于寄存器冲突，可以使用独立的读写端口构建寄存器堆。

此外，还可以把寄存器堆的访问拆成 2 部分，在时钟周期的前一半进行写操作而在时钟周期的后一半进行读操作，以此来解决寄存器冲突，可以在同一时钟周期内进行寄存器堆的读写。

Note

结构冒险大多必须使用硬件解决。

例如，如果不把指令和数据内存分成两个，每个时钟周期都会有 IF 和 MEM，尤其是遇到访存操作 lw 和 sw 时，一定会出现结构冒险，而且不能通过重排指令等解决（因为所有指令包括空指令都需要取指）。

4.4.2. 数据冒险

数据冒险（data hazard）是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。如 Figure 4-45 所示，红色数据流会产生数据冒险，绿色数据流则不会。

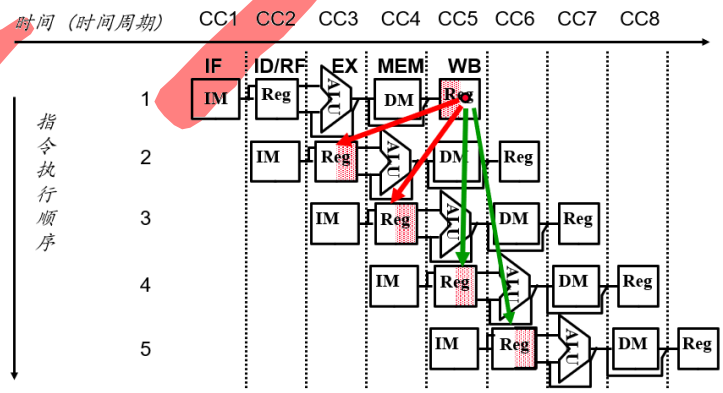

Figure 4-45 Data Flow of Data Hazard

根据 WB 级写回数据的来源：来自 EX 级的运算结果，来自 MEM 的访存结果，可将数据冒险分为两种情况处理。

由于每个时钟周期，流水线处理器都从流水线寄存器中读取操作数和控制信号，所以我们可以用“流水线寄存器名称.流水线寄存器字段”来表示从流水线寄存器中读取的数据。例如，ID/EX.RegisterRt 表示存储在 ID/EX 中的 rt 寄存器号，MEM/WB.RegWrite 表示存储在 MEM/WB 中的控制信号 RegWrite 值。

4.4.2.1. 冒险检测

观察 Figure 4-45，指令 2 在 CC4 时需要从 ID/EX 中读取数据，此时指令 1 的 EX 级运算结果存储在 EX/MEM，MEM 级访存结果还未存储；指令 3 在 CC5 时需要从 ID/EX 中读取数据，此时指令 1 的 EX 级运算结果存储在 MEM/WB，MEM 级的访存结果存储在 MEM/WB；指令 4 及之后的指令不再可能与指令 1 产生冲突（指令 4 可以和指令 1 在一个周期内分别读写寄存器堆）。

此外，由于$zero 始终为 0，无需考虑目标寄存器是$zero 的情况（例如 R 型指令 sll）。值得注意的是，只有需要写回的指令才有可能产生数据冒险，检测 RegWrite 控制信号来判断写回操作是否发生。

下面考虑连续的三条指令 1、2、3（以上分析表面只有连续的三条指令间才可能存在数据冒险，并且指令间须有相关性），来实现冒险检测，进而针对不同情况的冒险进行处理。

来自指令 1 的 EX 级数据冒险

(1) 指令 1 和指令 2 的冒险情况：

1a.

EX/MEM.RegWrite &&

EX/MEM.RegisterRd != 0 &&

ID/EX.RegisterRs = EX/MEM.RegisterRd

1b.

EX/MEM.RegWrite &&

EX/MEM.RegisterRd != 0 &&

ID/EX.RegisterRt = EX/MEM.RegisterRd

以 1a 类冒险为例，它代表“指令 1 需要写回寄存器堆（EX/MEM.RegWrite=1），指令 2 的 ALU 源操作数 1（ID/EX.RegisterRs）是指令 1 的 ALU 运算结果（EX/MEM.RegisterRd），且目标寄存器不为$zero”的情况。

(2) 指令 1 和指令 3 的冒险情况：

当指令 1 和指令 3 之间存在冒险时，指令 2 和指令 3 也可能存在冒险，比如指令序列：

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

其中指令 3 的 rt 就面临着选择指令 1 的 rd 还是指令 2 的 rd，前者来自 MEM/WB.RegisterRd，后者来自 EX/MEM.RegisterRd。

首先，指令 1 和 3、指令 2 和 3 同时存在冒险意味着三条指令两两存在冒险，是(1)中相邻指令存在冒险的特殊情况，指令 3 的 ALU 源操作数来自 EX/MEM.RegisterRd。

因此，在(2)中我们只需要考虑指令 1 和 3 存在冒险但指令 2 和 3 不存在冒险的情况，此时指令 3 的 ALU 源操作数来自 MEM/WB.RegisterRd。

2a.

MEM/WB.RegWrite &&

MEM/WB.RegisterRd != 0 &&

!(指令 2 和 3 的冒险条件) &&

ID/EX.RegisterRs = MEM/WB.RegisterRd

2b.

MEM/WB.RegWrite &&

MEM/WB.RegisterRd != 0 &&

!(指令 2 和 3 的冒险条件) &&

ID/EX.RegisterRt = MEM/WB.RegisterRd

这里“指令 2 和 3 的冒险条件”为：1a || 1b。

来自指令 1 的 MEM 级数据冒险

在本章 MIPS 指令子集中，MEM 产生数据的指令只有 lw，是 I 型指令，其将从内存取得的数据存放在 rt 寄存器。

指令 1 和指令 2 的冒险情况：

3.

ID/EX.MemRead &&

(IF/ID.RegisterRs = ID/EX.RegisterRt) || (IF/ID.RegisterRt = ID/EX.RegisterRt)

我们在解决这种冒险后，会发现指令 1 和指令 3 不再有冒险发生，即该冒险仅发生在 lw 和其紧随的一条相关指令之间，称之为**取数-使用型（load-use）数据冒险**。

4.4.2.2. 旁路

旁路（bypassing）或前推/重定向（forwarding）是一种解决数据冒险的方法，具体做法是从内部缓存提前取出数据，而不需要等待一条指令完全执行结束。

对于情况 1 和情况 2，我们可以在连续三条指令间添加旁路路径来解决冒险，如 Figure 4-46 所示：

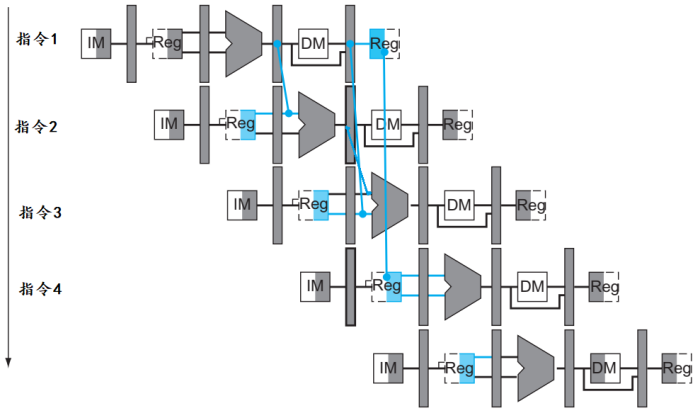

Figure 4-46 A Genral Pipelined Sequence of Instructions with Forwarding

图中寄存器堆内的旁路使得在该时钟周期读到的值是写入后的值。

对于情况 3，只添加旁路是无法解决的，这意味着在时间轴上逆流。因此必须阻塞 lw 紧随的使用指令，再添加旁路，如 Figure 4-47 所示：

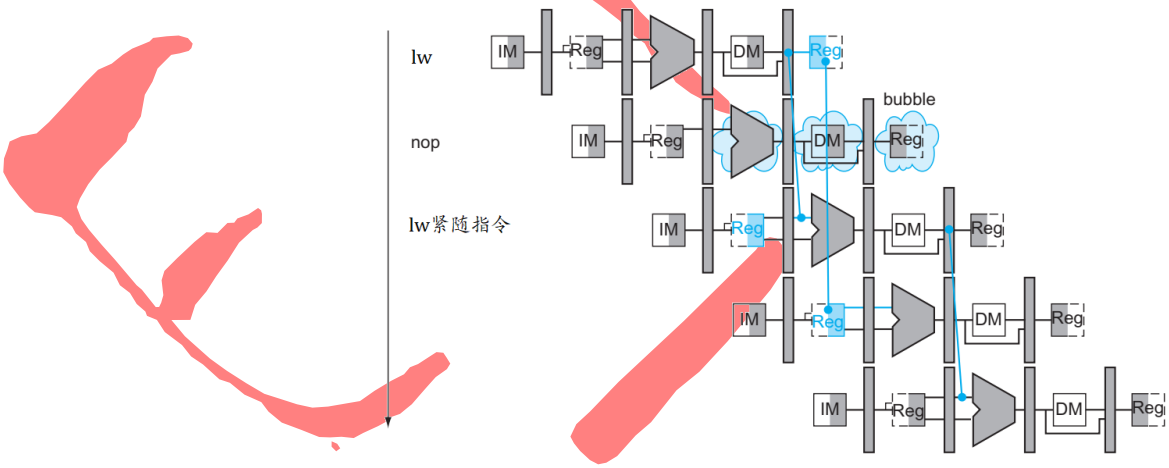

Figure 4-47 Load-Use Data Hazard with Stalls Inserted and Forwarding

具体执行时，我们称载入指令之后的流水槽称为载入延迟槽（load delay slot）。

在 lw 紧随指令执行到 IF 级时 lw 执行到 ID 级，并在本周期结束时 lw 紧随指令的数据存储在 IF/ID，lw 存储到 ID/EX。接着在 lw 紧随指令的译码阶段，可以识别取数-使用冒险，并进行流水线阻塞。

通过硬件阻塞载入延迟槽内的指令，等价于显式的放入一条空指令（nop）到槽中，并保持 PC 和 IF/ID 不变。空指令在 EX、MEM 和 WB 级什么都不做，所以要求将 9 个控制信号清除，而控制信号来自于 ID/EX，需要在译码阶段将控制信号置 0 并放在 ID/EX。

这样，在 lw 的 EX 级结束后，PC 和 IF/ID 都存储 lw 紧随指令的信息，ID/EX 存储 nop 指令执行 EX、MEM 和 WB 的信息（9 个为 0 的控制信号），EX/MEM 存储 lw 指令继续执行的信息——相当于在 lw 和 lw 紧随指令间插入了一条空指令，空指令的 IF 和 ID 级判断是否执行空指令，后三级“执行”空指令。

Note

空指令 nop 实际上是

```
sll $0, $0, 0
```

nop 后的指令整体被推迟了 1 个时钟周期，冒险不再发生（lw、nop 和 lw 紧随指令构成连续 3 个指令，之后的指令不再会和 lw 产生冒险）。

旁路解决数据冒险

对于一般指令序列的数据冒险，我们完全可以只使用旁路解决，只需要根据不同的冒险条件来控制进行相应的处理，这种控制可以增加一个**旁路单元（Forwarding Unit）**来实现。

旁路单元的输入是情况 1 和情况 2 条件判断涉及到的数据，即 2 个控制信号（EX/MEM.RegWrite, MEM/WB.RegWrite）和 4 个寄存器号（ID/EX.RegisterRs, ID/EX.RegisterRs, EX/MEM.RegisterRd, MEM/WB.RegisterRd）。输出是选择 ALU 来自寄存器的源操作数 1 和 2 的 2 位控制信号，传入旁路多选器后，可以在 ID/EX（无旁路发生）、EX/MEM（情况 1）和 MEM/WB（情况 2）的对应字段中选择，之后再和立即数进行选择。

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 4 44 The Control Values for the Forwarding Multiplexors

对于取数-使用的数据冒险，需要在译码阶段解决（一般旁路是在 ID 之后），可以再增添一个**冒险检测单元（Hazard Detection Unit）**。冒险检测单元的输入是检测条件涉及到的数据，即 1 个控制信号 ID/EX.MemRead 和 3 个寄存器号（IF/ID.RegisterRs, ID/EX.RegisterRt, IF/ID.RegisterRs）。输出是 3 个控制信号，分别用来保持 PC（PCWrite=0）、保持 IF/ID（IF/IDWrite=0）和清除 9 个控制信号。

为了突出主要矛盾，Figure 4-48 省略了经过符号扩展的立即数和分支逻辑涉及到的单元，展示了用旁路和阻塞解决数据冒险的流水线概览图。

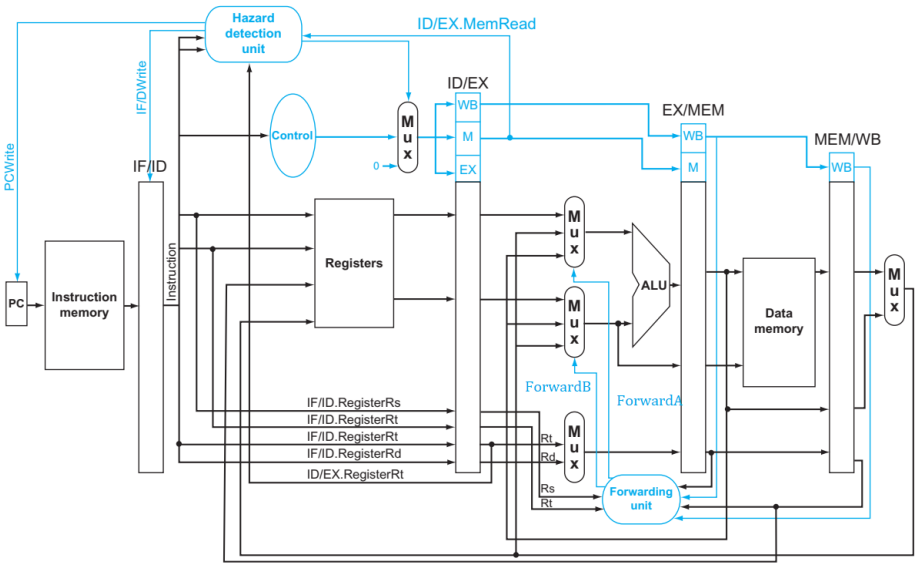

Figure 4-48 Pipelined Control Overview with Forwarding

4.4.2.3. 代码重排

由于指令间的数据冒险发生的前提是三条连续指令且指令间有相关性（依赖性），因此我们可以通过软件上重新安排代码顺序来避免这种连续三条指令间依赖关系的发生。

然而，指望编译器产生无数据冒险发生的汇编代码是几乎不可能的，因为这种依赖关系经常发生，牵一发而动全身。旁路的设计能让我们无需阻塞即可处理来自 EX 级数据造成的数据冒险，关键是使用代码重排来避免取数-使用冒险的发生，从而无需任何阻塞。

综上，编译器可以不用处理来自 EX 级数据造成的数据冒险，只需避免取数-使用冒险：在载入延迟槽内放置一条与 Load 无关的指令。

Example 4.4

判断以下指令序列是否发生数据冒险，发什么哪种数据冒险。如果发生取数-使用冒险，请重排指令顺序以避免流水线阻塞。

(1)

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

(2)

```
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)
```

(1)

sub-add 是 1a 类冒险

sub-or 是 2b 类冒险

均可以通过旁路消除

(2)

两条 sw 指令都各自依赖上一条 add 指令，存在 1b 类冒险，可以通过旁路解决

两条 add 指令都各自依赖于上一条 lw 指令，因此均存在取数-使用冒险

重排后的指令可以消除两个取数-使用冒险：

```
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)
```

Note

数据冒险只有在连续三条指令间才可能发生，取数-使用冒险仅发生在 lw 和 lw 紧随指令间且后者对前者有依赖关系。

依赖关系的判断是观察前面指令三个方面：是否执行写寄存器操作，目标寄存器是否为$zero，目标寄存器是否被后面指令当做操作数使用。

4.4.3. 控制冒险

控制冒险（control hazard）也称为分支冒险，是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，异常也是一种控制冒险)而导致指令不能在预定的时钟周期执行。

控制冒险出现的频率比数据冒险小得多，但还没有有效的方法能够解决控制冒险。因此在介绍以下解决控制冒险的方案后，需要对其进行优化。在本节，我们先解决非异常的控制冒险。

4.4.3.1. 解决方案

预测（Predict）

最简单的预测是假定分支不发生：预测分支不发生，并继续执行顺序的指令流；如果分支发生，则丢弃已经读取并译码的指令。

假定分支不发生的实现关键是如何清除（flush）指令。与取数-使用数据冒险类似，清除相当于将被丢弃的指令变为空指令，需要将每级开始的控制信号置 0。由于分支指令在 MEM 级才能决定是否发生分支，如果不加干涉，在分支指令 MEM 级结束时，已经多执行了 3 条指令，如 Figure 4-49 所示：

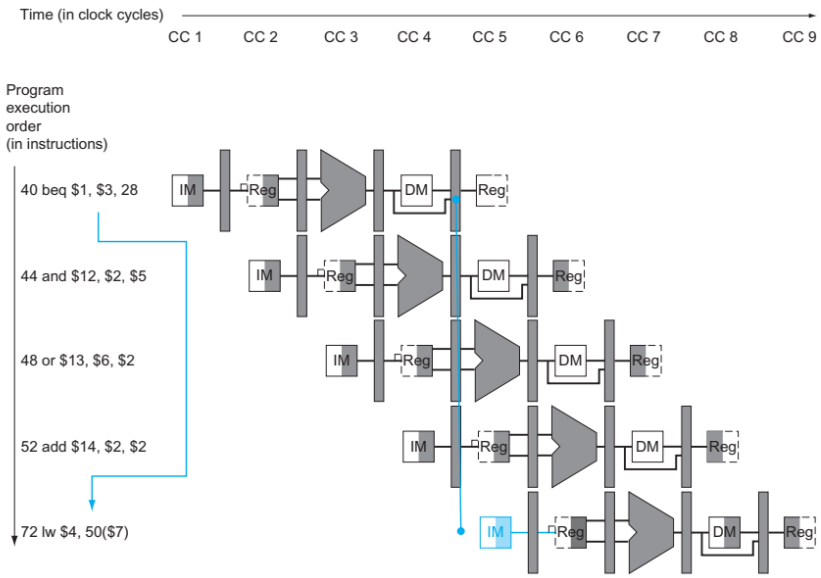

Figure 4-49 The Impact of the Pipeline on the Branch Instruction

因此，当分支指令到达 MEM 级时，必须将在 IF、ID、EX 级的三条指令（在 Figure 4-49 中依次是 add、or、and）的控制信号置 0。

预测正确时不会降低流水线速度，但是一旦预测错误，就需要清除所有错误指令。因此对于预测的优化涉及两方面：减少错误代价，提高预测准确率。假定分支不发生的预测准确率是 50%，错误代价是 3 条指令。

延迟分支（Delayed Branch）

延迟分支（分支延迟）重新定义了分支。分支原来的定义是：如果产生分支，紧接在分支指令后的指令会根据分支结果而不被意外执行；而延迟分支的定义是：无论分支结果如何，紧接着分支指令后的那条指令一定被执行。

与载入延迟槽类似，分支指令之后的流水槽称之为分支延迟槽（branch delay slot）。我们可以用一条与分支指令无关（无论是否分支都会执行）的指令放入分支延迟槽，抵消掉一个阻塞——延迟分支也是一种代码重排，可以利用编译器实现或者人工重排。

但如 Figure 4-49 所示，分支指令不加干涉的话，最坏的情况会产生 3 条空指令，意味着 3 个阻塞的时钟周期。所以对延迟分支的优化有两种方向：让最坏的情况下产生 1 条空指令（减少错误代价到 1 条指令），或者增加硬件来处理超过 1 个时钟周期的延迟分支。

4.4.3.2. 优化方案

缩短延迟

一般的分支指令都在 MEM 级才能确定下一条指令的 PC，但是分支目标地址确定得越早，多执行的指令就越少，产生的空指令（阻塞）也就越少。为了尽量提前分支决策，需要提前两个动作：计算分支目标地址和判断分支条件。

计算分支目标地址可以使用 IF/ID 中的 PC 和立即数字段，将 EX 级分支地址的计算提前到 ID 级。

判断分支条件需要比较源操作数。MIPS 中只有相等/不等则分支指令，其他指令都是基于以上指令的伪指令。因此，我们需要比较从 ID 级取得的两个寄存器的值是否相等，由**相等检测单元（equality unit）**处理。

> **Note**
>
> 相等检测可以不使用 ALU，使用更简单的部件以提高性能。可以先将两个源操作数对应位进行异或，然后把结果的各位间进行或操作：输出为 0，则相等；输出为 1，则不等。

复杂的问题在于源操作数的来源，和数据冒险类似，可以来自 IF/ID（顺流）、EX/MEM（逆流）、MEM/WB（逆流）。与数据冒险不同的是，缩短延迟将分支比较提前到了 ID 级，意味着必须阻塞一个周期，才能让 ID 级可以使用解决 EX 级数据冒险的旁路。注意，EX 级数据冒险中的取数-使用冒险必须阻塞一个周期，倘若分支指令依赖的前一条指令正好是 lw（取数-分支冒险），则总共需要阻塞两个周期。

尽管有这些阻塞的发生，看似又将 ID 级延后到 EX 级。但控制冒险很少发生，我们既可以在编译代码时避免产生对前面指令有依赖性的分支指令，也可以在硬件上我们需要为延迟分支增添额外的旁路单元和冒险检测单元来控制阻塞的发生。

综上，将分支执行提前到 ID 级依然是一种有效的改进，它将错误的代价减小到只有一条指令。此时，若要清除错误执行的指令，只需要一个 IF.Flush 的控制信号将 IF/ID 的 9 个控制信号置 0，即只需要清除 IF 级的指令即可。假设分支不发生，在执行相等则分支时，我们可以利用相等检测单元，若相等，则将 IF.Flush 置为有效；否则，置为无效。

分支目标缓存

即便我们可以提前成功预测是否发生分支，但计算分支目标需要一个时钟周期，延迟分支可以用无关指令填充这个周期。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指令，从而在发生分支时直接取用而不必计算。

分支目标缓存一般是带标志位的 cache，硬件开销较大。

动态预测

动态分支预测是根据运行信息在执行过程中进行分支预测，其实现需要额外的硬件支持——分支预测器。由于摩尔定律，动态分支预测虽然硬件开销大，但成本相对更低，处理器可以使用更多晶体管来实现更复杂但准确率更高的分支预测以提高性能。

动态预测的实现一般是使用分支历史记录表或者分支预测缓存记录分支最近发生的情况，然后不同的设计会用不同长度、不同含义的预测位来代表分支发生的各种情况，根据不同的情况进行分支预测（假设）。

对某一分支的预测位可用 1 位数据说明分支上一次是否发生，据此进行预测。若上一次发生，则从上次分支发生的地方取指；若上一次不发生，则假设分支不发生。如果假设错误，则将预测位取反。更精确的是使用两位预测位，只有连续两次假设错误时才会改变假设分支方向。常见的动态分支预测有以下几种：

(1) 特定分支预测器

这种策略只考察某条特定分支的最近情况，而与其他分支的执行情况无关。

(2) 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。

(3) 竞争预测器

对每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，从中选择一个预测器作为给定分支的预测结果。

目前，竞争预测器的预测最准确。

减少分支

如果解决不了问题，我们可以解决提出问题的人，即减少条件分支的数量。

一种方法是加入条件移动指令（conditional move instruction），它可以根据条件改变移动目的寄存器。如果条件不成立，条件移动指令就相当于一条 nop 指令，其实现方法是在指令中添加条件字段。例如，某一版本的 MIPS 中含有 movn（move if not zero）和 movz（move if zero）指令。对于指令

```
movn $8, $11, $4
```

若寄存器$4 的值不为 0，则将$11 的值复制到$8；否则，该指令什么也不执行。

4.4.3.3. 控制冒险的最终处理方案

将以上不同的控制冒险解决方案进行综合优化，使用预测和延迟分支解决。

对于预测，使用动态预测、缩短延迟优化并可以处理预测错误的情况。对此，我们可以在已经实现数据冒险处理的数据通路上添加分支预测器（动态预测）、IF.Flash 控制信号（清除预测错误的指令）、相等检测单元，并将分支的比较和计算提前到 ID 级（缩短延迟）。

此外，在软件上我们可以通过代码重排（分支延迟）来解决控制冒险。

4.4.4. 处理冒险的数据通路

忽略了 ALUSrc 多选器和 ALU 控制单元，Figure 4-50 概略地展示了可以处理非异常冒险的数据通路及控制：

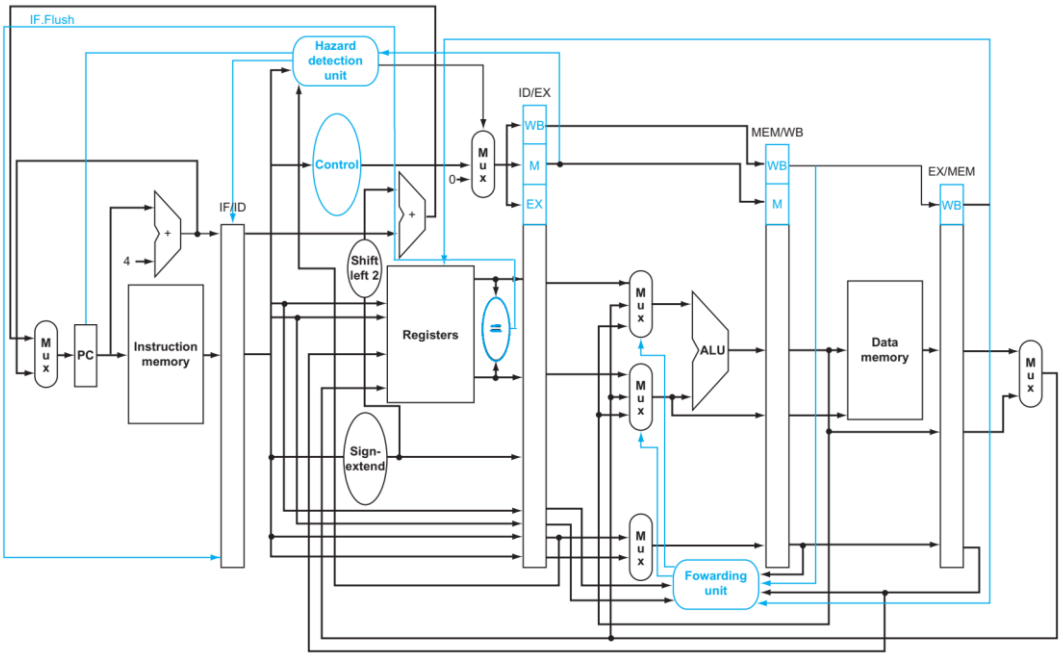

Figure 4-50 The Datapath and Control Dealing with Hazards

4.4.5. 性能对比

Example 4.5: Data Hazard

假设指令中有部分存在 RAW（read after write）数据相关，即生成结果的流水级和使用结果的下面指令（产生结果指令后紧跟的第 1、2、3 条指令等）存在后者对前者的依赖性。现考虑会产生数据冒险的 RAW，其在全部指令中的占比如下所示：

EX to 1st Only	MEM to 1st Only	EX to 2nd Only	MEM to 2nd Only	EX to 1st and MEM to 2nd	Other RAW Dependences
5%	20%	5%	10%	10%	10%

假设无数据冒险时处理器的 CPI 为 1；

考虑 EX 级根据旁路机制的不同有不同的延迟时间，各级流水线延迟如下：

IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/MEM only)	EX (FW from MEM/WB only)	MEM	WB
120 ps	100 ps	120 ps	130 ps	120 ps	120 ps	120 ps	100 ps

(1) 上述五种 RAW 冒险在没有任何机制处理的情况下，分别需要插入多少条空指令（阻塞）才能正确执行？

(2) 在只存在数据冒险的情况下，分别计算如下流水线的 CPI：

(i) 无任何旁路机制

(ii) EX/MEM 旁路机制（情况 1 的条件）

(iii) MEM/WB 旁路机制（情况 2 但没有!（指令 2 和指令 3 冒险）的判断）

(iv) 全旁路机制

然后据此比较后三者相对于无任何旁路机制的性能加速比。

(3) 为什么 EX to 1st and EX to 2nd 可能产生冒险，而 MEM to 1st and MEM to 2nd 数据冒险并未在上表中列出？

(1)

EX to 1st Only	MEM to 1st Only	EX to 2nd Only	MEM to 2nd Only	EX to 1st and EX to 2nd
指令 1	指令 1	指令 1	指令 1	指令 1
nop	nop	指令 2	指令 2	nop
nop	nop	nop	nop	nop
指令 2	指令 2	指令 3	指令 3	指令 2
指令 3	指令 3			指令 3

(2)

只需要计算额外执行多少的空指令，相当于对应的每条冒险指令的执行时钟周期（CPI）增加多少

(i)

$$CPI = 1 + 5\% \times 2 + 20\% \times 2 + 5\% \times 1 + 10\% \times 1 + 10\% \times 2 = 1.85$$

(ii)

这种情况解决了 EX to 1st Only 冒险，使得执行额外空指令

EX to 1st Only 为 0

EX to 2nd Only 为 1

EX to 1st and EX to 2nd 为 1（不需要第一次阻塞，已被 EX/MEM 旁路解决）

而对 MEM 级没有影响，故

$$CPI = 1 + 5\% \times 0 + 20\% \times 2 + 5\% \times 1 + 10\% \times 1 + 10\% \times 1 = 1.65$$

(iii)

这种情况解决了 EX to 2nd Only 冒险，使得执行额外空指令

EX to 1st Only 为 1

EX to 2nd Only 为 0

EX to 1st and EX to 2nd 为 1（不需要第二次阻塞，已被 EX/MEM 旁路解决）

MEM to 1st Only 为 1（取数-使用冒险必须阻塞）

MEM to 2nd Only 为 0（已被 EX/MEM 旁路解决）

而对 MEM 级没有影响，故

$$CPI = 1 + 5\% \times 1 + 20\% \times 1 + 5\% \times 0 + 10\% \times 0 + 10\% \times 1 = 1.35$$

(iv)

完全旁路机制可以解决除了取数-使用冒险的所有数据冒险，只剩下 MEM to 1st Only 需要额外执行 1 条空指令，故

$$CPI = 1 + 20\% \times 1 = 1.2$$

将性能（CPU 执行时间的倒数）的相关量（设指令总数为 n）总结如下，可知性能加速比：

指标	无旁路	EX/MEM 旁路	MEM/WB 旁路	全旁路
CPI	1.85	1.65	1.35	1.2
时钟周期/ps	120	120	120	130
CPU 执行时间	222n	198n	162n	156n
加速比	——	1.12	1.37	1.42

(3)

MEM to 1st 存在冒险时，对应的是取数-使用冒险，无论使用何种旁路机制，必须阻塞一个周期，这种阻塞使得第二条指令的 ID 级对应基础指令的 WB 级。即

便存在 MEM to 2nd 冒险（第二条指令的 EX 级对应基础指令的 MEM 级），也因阻塞而消除。

而对于 EX to 1st and EX to 2nd，使用全旁路机制可以避免这两种数据冒险，只使用 EX/MEM 或 MEM/WB 旁路机制，两种冒险产生的阻塞不同，所以需要区分。

Example 4.6: Control Hazard

假设分支指令的动态执行频度为25%，以及各种分支预测器的精度如下：

Always-Taken	Always-Not-Taken	2-Bit
45%	55%	85%

(1) 假设没有数据冒险且不使用分支延迟槽，也没有缩短延迟措施，求分别使用以上各种分支预测器的处理器 CPI。

(2) 假设有缩短延迟措施，并且配备有需要的硬件，其他条件不变。对于 2 位分支预测器而言，将一半分支指令用 ALU 指令替代（2 条 ALU 指令替代 1 条分支指令），且被正确预测和错误预测的分支指令被取代的概率相同。求其相对于(1)中 2 位分支预测器的加速比。

(1)

本问条件下，每一次预测失误都需要阻塞 3 个周期（产生 3 条空指令）

分支总发生：

$$\text{CPI} = 1 + 25\% \times (1 - 45\%) \times 3 = 1.4125$$

分支总不发生：

$$\text{CPI} = 1 + 25\% \times (1 - 55\%) \times 3 = 1.3375$$

2 位分支预测器：

$$\text{CPI} = 1 + 25\% \times (1 - 85\%) \times 3 = 1.1125$$

(2)

缩短延迟将预测错误的代价减小到一条空指令，设原来的指令总数为n

用 2 条 ALU 指令取代 1 条分支指令，指令总数变为$(1 + 25\% \times 0.5)n = 1.125n$

此时另一半分支指令占比 $12.5\%n/1.125n$=11%

$$\text{CPI} = 1 + 11\% \times (1 - 85\%) \times 3 = 1.05$$

加速比$= \frac{T \cdot n \cdot 1.1125}{T \cdot 1.125n \cdot 1.05} = 0.9418$

4.5. 指令级并行

4.5.1. 并行化的类型

指令级并行（Instruction-Level Parallelism, ILP）是指令间的并行。衡量 ILP 性能的标准是处理器在执行某程序时，平均能够同时执行指令的数量，称为机器并行性。

数据级并行（Data-Level Parallelism, DLP）指处理器能够同时处理多条数据，是对不同的数据进行相同的操作获得的并行，属于单指令多数据（Single Program Multiple Data, SPMD）模型。比如，有如下 C 语言循环代码：

```
for (i=0; i<100; ++i)
    a[i] = b[i] + c[i];
```

如果没有 DLP，则需要对其循环多遍进行。而循环展开后，支持 DLP 的 CPU 可以使用一条指令同时执行 100 次 b[i] + c[i]，从而只需要一条指令计算。DLP 需要有额外的特殊指令和特殊寄存器。

线程级并行（Thread-Lever Parallelism, TLP）是指一个处理器可以同时处理多个线程。

一个程序（procedure）包含着若干个进程（process），一个进程包含着若干个线程（thread），即是“程序>进程>线程”这样的包含关系。在操作系统处理一个程序时，进程是程序的抽象，线程是进程的抽象。

Note

每个线程对操作系统来说，是一个程序的细小部分，就是 CPU 当前要处理的一件事。一个进程不仅仅可能包含这个多个线程，进程还包含着：被操作系统分配的资源空间，比如操作系统在内存中分配给它的栈，数据区域等。这些资源空间被多个线程所共享。比如你和朋友们一次聚餐，这就很像一个进程。桌子上那些饭菜就是资源，你们每个人都是一个线程，每个人每次伸手去夹菜就是在使用这个共享的资源。

一般地，一个执行单元和一个控制单元被称为一个核（core），核上可以处理线程，而存储单元作为资源由多个线程共享。在单核处理上，超线程技术（Hyper-Threading, HT）可以在一个核上处理双线程；在多核处理上，可以在一块处理器上增加多个物理核心（多个控制单元和执行单元，并且共享存储单元）来实现线程级并行。

Note

注意“多核处理器≠多处理器”，前者是一块处理器上有多个物理核心，后者是有多个处理器。显然前者性能更高，因为多处理器间有多个存储单元需要共享资源，但前者的成本也比相同处理能力的后者高。例如，一个 3 处理器的四核（每个核带有 HT 技术）芯片可以同时处理 3×4×2 个线程。

4.5.2. 增加指令级并行程度

有两种方法增加潜在的指令级并行程度：增加流水线的深度（流水级数），以便使更多指令重叠进行，缩短时钟周期；增加流水线的宽度，使得每个流水级可以处理多条指令。

然而流水线级数越多，需要更多的旁路/冒险的硬件逻辑，同时也会带来更多的流水线迟滞开销（流水迟滞在时钟周期时间中占的比重越来越高）。对于更深的流水线，超级流水线（Super Pipeline），其设计与本章所讲的流水线思路一致，不过更加复杂，在此不再赘述。

本节重点入门如何增加流水线的宽度，将多条指令打包到发射槽（issue slot）中，在给定时钟周期内发射多条指令，这种技术被称为多发射（multiple issue）。实现一个多发射处理器的方式有两种，主要区别是编译器和硬件之间的工作分工。由于不同的实现方式将导致某些决策是静态进行（在编译时）还是动态进行（在执行时），因此这两种方式分别被称为静态多发射（static multiple issue）和动态多发射（dynamic multiple issue）。实际上这两种方法经常借用对方的技术，没有哪种方法称得上是完全独立的。

4.5.3. 静态多发射处理器

静态多发射也称为超长指令字（Very Long Instruction Word, VLIW），在一个时钟周期内发射多条指令，其集合称为发射包（issue packet）。

一种 MIPS 静态双发射处理器的一条超长指令字中的一条指令是整型 ALU 操作或分支指令，另一条指令可以是访存指令，如 Figure 4-51 所示。为了简化译码和发射的过程，必须严格限制可同时发射的指令。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，并且顺序不能颠倒；实在找不到另一条可同时发射的指令时，就用 nop 指令代替。

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Figure 4-51 Static Two-Issue Pipeline in Operation

为了实现 MIPS 静态双发射的数据通路，还需要额外的硬件支持。比如寄存器堆多出两个读端口和一个写端口，以及一个额外的 ALU。

Note

取数-使用冒险同样存在于 VLIM，需要编译器来调度指令避免使用延迟。Figure 4-51 中如果发生取数使用冒险，后续指令都需要阻塞一个周期。

循环展开

循环展开（loop unrolling）将循环体复制多份，从而可以将不同循环体内部的指令调度在一起，以获得更高的指令级并行。

在循环展开的过程中，有可能因为寄存器名的重用导致虚假的数据相关，称为反相关（autidependence）/名字相关（name dependence）。此时需要将寄存器重命名（register renaming），比如引入几个临时寄存器，以消除反相关。

4.5.4. 动态多发射处理器

动态多发射也称为超标量（superscalar）。

Note

超标量与超长指令字的根本区别就是前者由硬件保证正确性，后者由编译器保证正确性。后者代码从一个 VLIM 处理器移到另一个 VLIM 处理器时，需要重新编译以保证正确执行，而前者则始终正确执行（无论是否经过调度，调度只是提高性能，与正确性无关）。

超标量也可以有发射包，并不是与超长指令字的根本区别。

动态流水线调度

动态流水线调度（dynamic pipeline scheduling）的流水线被划分为三个主要单元，被许多超标量处理器使用，如 Figure 4-52 所示：

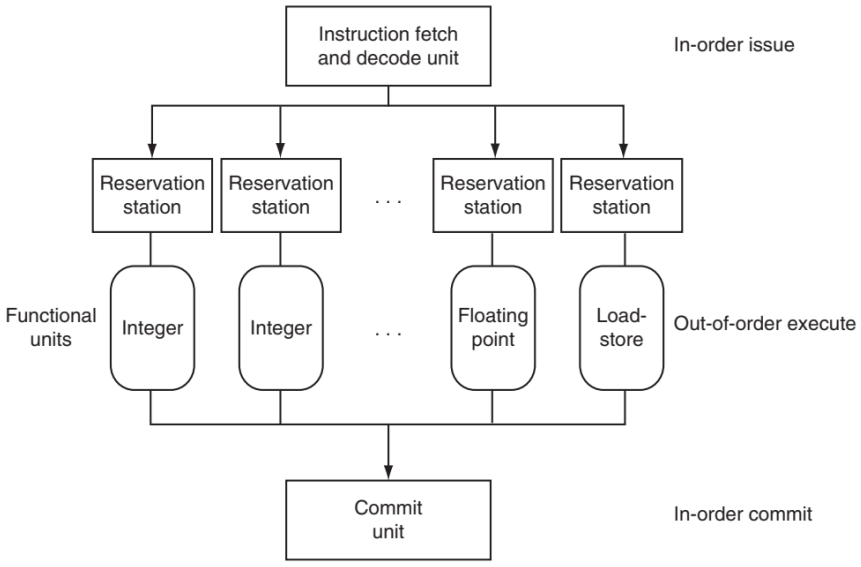

Figure 4-52 The Three Primary Units of a Dynamically Scheduled Pipeline

取指与发射单元（instruction fetch and decode unit）。每个周期处理器决定发射几条指令，可借用静态多发射的技术，使用编译器进行指令调度以消除指令间的相关关系，从而达到较高的发射速率。顺序发射保证了指令按逻辑顺序执行。

多个功能单元（functional unit）。每个功能单元都有自己的缓冲区，用来保存操作数和操作，称为保留站（reservation station）。如果执行的指令被阻塞，

由于各功能单元的保留站有操作副本，可以不受影响地乱序执行（物理上）。此外，保留站还可以暂存来自其他功能单元的计算结果供自己的功能单元使用。

提交单元（commit unit）。提交单元中有一个称为重排序缓冲区（reorder buffer）的缓冲区，用于暂时保存功能单元执行的结果，等到安全时按顺序“提交”给相应的存储单元。

Note

关于动态流水线调度，可以联系旁路进行理解，只需要记住顺序发射（逻辑）、乱序执行（物理）、顺序提交（逻辑）。

4.5.5. 数据相关

到目前为止，我们遇到了三种数据相关，当两条数据相关的指令距离足够近时，分别会产生对应的冒险：

反相关（名字相关），是寄存器名的重用而导致的相关，而没有发生通过该寄存器名的数据流动。反相关可能导致读后写（WAR）冒险，常发生在循环展开之中。

输出相关，两条指令都要对同一个存储单元或寄存器进行写操作。输出相关可能导致写后写（WAW）冒险，通常发生在动态流水线调度（乱序执行）中。

真数据相关，是本章前面所讨论的原始流水线冒险，可能导致写后读（RAW）冒险。

Note

读后写冒险发生时，前一条指令的写操作尚未完成，因此当前指令无法正确获取所需的数据；写后写冒险发生时，无法保证最后写入的值是指令序列顺序执行时应当写入的值；写后读冒险发生时，如果当前指令的写操作尚未完成，后续指令将读取到错误的数据。

判断名字相关最快的方法，是将同名寄存器换为其他无关寄存器，看指令执行是否符合原意，若仍符合，说明没有数据流动，即名字相关。

名字相关只有离得近才会发生读后写冒险，否则不会有数据的流动，所以叫“可能”发生冒险。读后写和写后读冒险的针对的都是当前指令，前者重点在于当前指令在当前周期无法“读”正确的值，后者重点在于后续指令在当前周期无法“读”正确的值。

流水线处理器除了开始的填充和最后的清空阶段，每个时钟周期可以看做执行一条指令，该时钟周期执行的指令以在此时钟周期取指的指令为准。注意，空指令也需要取指，除了多发射的发射包内部指令（发射包间也不允许，发射包相当于一条长指令），不允许同一时钟周期取多条指令。

原始流水线中只存在 RAW 冒险，因为没有乱序执行，所以不会存在 WAW 冒险；因为没有超长指令字，一个周期只能执行一条指令，所也不存在当前指令在当前周期无法“读”正确的值的 WAR 冒险。

Example 4.7

指出下列指令序列存在的数据相关，各种相关指出一种可能即可：

```
Loop:
    lwcl $f0, 0(x1) # lw
    add.s $f4, $f0, $f2
    swcl $f4, 0(x1) # sw
    addiu x1, x1, 4
    bne x1, x2, Loop
```

swcl 与 addiu 存在名字相关，把同名寄存器 x1 换为其他也可以正常执行。

lwcl 和 add.s 存在真数据相关，一个典型的取数-使用冒险。

两次相邻循环展开后的两条 addiu 指令间存在输出相关，一个循环体内没有。

综上所述，各种流水线主要的性能瓶颈在于那些不能立即解决的相关性。