第7章 指令流水线

第 6 章介绍的 MIPS 处理器实现方案都是采用串行执行指令的方法，同一时刻 CPU 中只有一条指令在执行，多周期实现方案相比单周期方案时钟周期更短，成本更低廉，但各功能部件使用率不高。现代计算机普遍采用指令流水线技术并行执行指令，同一时刻有多条指令在 CPU 的不同功能部件中并发处理，可大大提升功能部件的并行性和程序执行效率。本章主要介绍指令流水线的基本概念，流水线数据通路构成、流水线的冲突(冒险)现象及其解决方法。

7.1 流水线概述

7.1.1 流水线的概念

流水线处理技术并不是计算机领域特有的技术。在计算机出现之前，流水线技术已经在工业领域得到广泛应用，如汽车装配生产流水线等。计算机中的流水线技术是把一个复杂的任务分解为若干个阶段，每个阶段与其他阶段并行处理，其运行方式和工业流水线十分相似，因此被称为流水线技术。

可以从两方面来提高计算机内部的并行性：一个是采用时间上的并行性，通过将一个任务划分成几个不同的子过程，并且各子过程在不同的功能部件上并行执行，使得在同一个时钟周期内同时解释多条机器指令，提高程序的执行速度，这种方法即为流水线处理技术；另一个是采用空间上的并行性处理技术，即在一个处理机内设置多个执行相同功能的独立操作部件，并且让这些操作部件并行工作，这种处理机被称为多操作部件处理机或超标量处理机。

把流水线技术应用于数据运算的执行过程，就形成了运算操作流水线，也称为部件级流水线，如浮点运算流水线将浮点数加法运算过程分解为求阶差、对阶、尾数加和规格化 4 个子过程。将流水线技术应用于指令的解释执行过程，就形成了**指令流水线。**

7.1.2 MIPS 指令流水线

MIPS 本义是无内部互锁流水级微处理器，英文全称为 Microprocessor without Interlocked Pipeline Stages，是典型的 RISC 指令系统，通常内部采用指令流水线结构，设计初衷是通过编译器解决流水段中的冲突。以 MIPS32 指令系统为例，其指令的执行过程细分为五个阶段，每个阶段由对应的功能部件完成，这里每个阶段也被称为一个“功能段”。

- 取指令(IF)：负责从指令存储器取出指令；
- 指令译码(ID)：操作控制器对指令字进行译码，同时从寄存器堆取操作数；
- 指令执行(EX)：执行运算操作或计算地址；
- 访存(MEM)：对存储器进行读写操作；
- 写回(WB)：将指令执行结果写回寄存器堆。

在 MIPS 单周期实现中，这五个功能段是串行连接在一起的，为方便描述，后文将这五个功能段分别简称为 IF、ID、EX、MEM、WB 段。如图 7.1 所示，程序计数器 PC 的值

送 IF 段取指令，然后依次进入 ID、EX、MEM 段进行处理，注意虽然不是所有指令都必须经历完整的五个阶段，但只能以速度最慢的指令作为设计其时钟周期的依据，单周期 CPU 的时钟频率取决于数据通路中的关键路径（最长路径），也就是五个阶段的延迟总和 T_{total}，所以单周期 CPU 指令执行效率不佳。

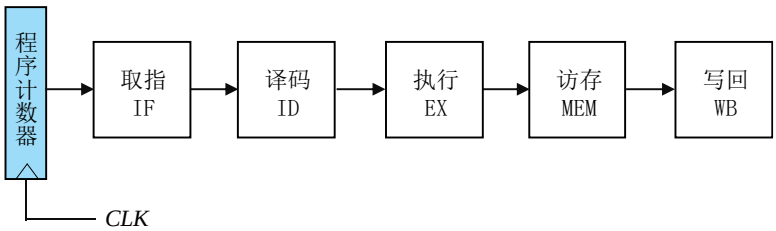

图 7.1 单周期 MIPS CPU 逻辑架构

MIPS 指令流水线在每个执行阶段的后面都需要增加一个**流水寄存器**，用于锁存本段处理完成的所有数据或结果，以保证本段的执行结果能在下一个时钟周期给下一个阶段使用，如图 7.2 所示。程序计数器、流水寄存器均采用统一公共时钟进行同步，每来一个时钟，各段组合逻辑功能部件处理完成的数据将会锁存到段尾的流水寄存器中，作为后段的输入，同时当前段也会接收到前段通过流水寄存器传递过来的新指令或数据，一条指令会依次进入 IF、ID、EX、MEM、WB 五个功能段进行处理，当第一条指令进入 WB 段后，流水线各段都包含一条不同的指令，流水线中将同时存在五条不同的指令并行执行，理想情况下此后每隔一个时钟周期，都会有一条新的指令进入流水线，同时也有一条指令执行完毕流出流水线。

假设五个功能段中的最大关键延迟为 T_{max}，则指令流水线最小时钟周期为 T，执行 n 条指令需要时间为$(n+4)\,T_{max}$，指令执行的 CPI 为 n/(n+4)，当 n 较大时，指令流水线 CPI 约等于 1，由于时钟周期相对单周期方案更短，指令流水线性能加速比=T_{total}/T_{max}。

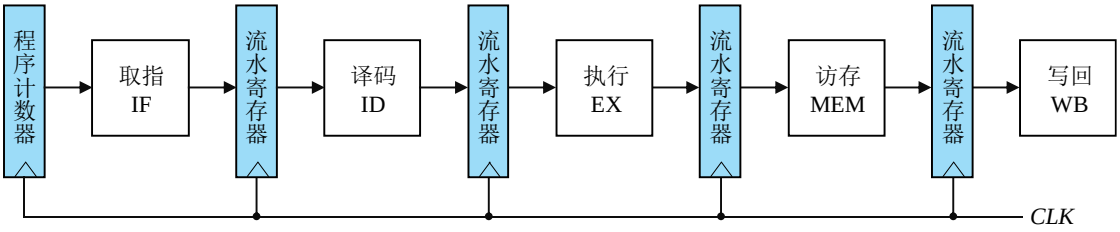

图 7.2 MIPS 指令流水线逻辑架构

流水寄存器的引入使得各段中的指令可以在时间上并行，流水寄存器在同一时钟的驱动下可以锁存流水线前段（左侧）加工完成的数据以及控制信号，锁存的数据和控制信号将用于后段的继续加工处理。流水寄存器在这里类似于传统工业流水线中的传送装置，流水线寄存器时钟频率决定了流水线的传送速度，如果频率过快，各功能段可能无法及时完成数据处理，为保证指令流水线正确运行，流水寄存器的时钟频率应取决于五个功能段中最慢的一段的关键延迟，所以流水分段时应该尽量让各段时间延迟相等。

7.1.3 流水线的时空图表示

可以用时空图的方式描述指令在不同功能段中的执行情况。图 7.3 为单周期 MIPS 处

理器的时空图，横坐标表示时间，纵坐标为空间，表示当前指令所处的功能部件。假设指令执行各功能段时间延迟均为 T，则横坐标被分割成相同长度的时间段 T；单周期 CPU 中由于各功能段完全串联，无法并行处理，一条指令会依次进入 IF、ID、EX、MEM、WB 段，指令周期为 5T。当一条指令执行完毕后，第二条指令才能进入 IF 段取指令，同一时刻只有一个功能部件工作，各功能部件无法并行，所以执行每条指令的执行时间都是 5T。

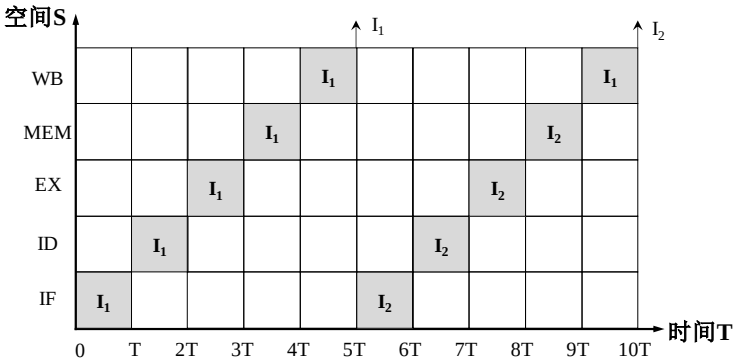

图 7.3 单周期 MIPS CPU 时空图

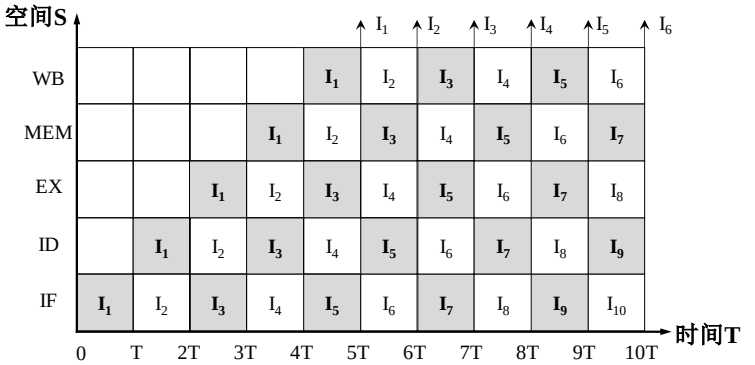

图 7.4 五段 MIPS 流水线时空图

图 7.4 为五段 MIPS 流水线的时空图，图中第一个时钟节拍 I_1 指令进入 IF 段取指令，第 2 拍 I_1 指令进入 ID 段译码取操作数，同时 I_2 指令进入 IF 段取指令，第 5 个时钟节拍，流水线充满，每个功能段均包含一条指令，至此每隔一个时钟周期 T 流水线将完成一条指令的执行。相比单周期 CPU 提升 5 倍。如需要执行 n 条指令，执行时间为(n+4)*T，当然这是理想情况，实际指令流水线存在很多的冲突冒险，会使得流水线暂停或阻塞，还需要通过其他机制进行处理，后文会进行详细介绍。

指令流水线将一条指令的执行划分成若干个阶段，每个阶段由一个独立的功能部件来完成，依靠各功能部件的并行工作来提高系统的吞吐率和处理速度。注意指令流水线并不能改变单条指令的执行时间，但当指令流水线充满时可大幅提高程序的执行效率。

只有大量连续任务不断输入到流水线中，保证流水线的输出端有任务不断地从流水线中输出，才能充分发挥流水线的性能，而指令的执行正好是连续不断的，非常适合采用流水线技术。对于计算机中的其他部件级流水线，如浮点运算流水线，乘法运算流水线，同样也仅仅适合于提升浮点运算密集型、乘法运算密集型应用的性能，对于单个的运算也是无法提升性能的。

7.2 流水线数据通路

简单的单周期 MIPS 处理器数据通路如图 7.5 所示，根据图中虚线所示，单周期 MIPS 处理器数据通路从左到右依次分成 5 个阶段：取指令 IF 段、译码取数 ID 段、指令执行 EX 段、访存 MEM 段、写回 WB 段。其中 IF 段包括程序计数器 PC、指令存储器以及计算下条指令地址逻辑；ID 段包括操作控制器、取操作数逻辑、立即数符号扩展模块；EX 段主要包括算术逻辑运算单元 ALU、分支地址计算模块；MEM 段主要包括数据存储器读写模块；WB 段主要包括寄存器写入控制模块。

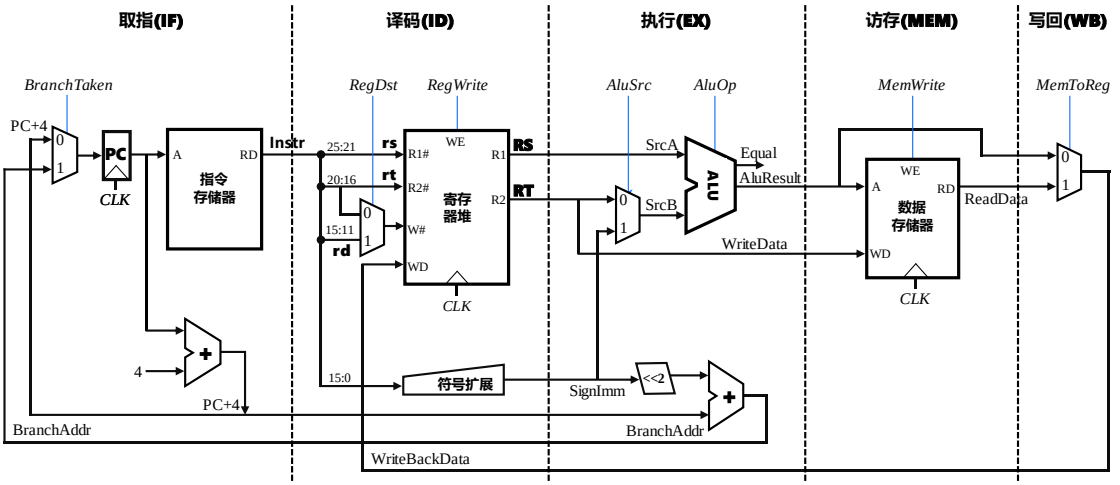

图 7.5 单周期 MIPS 数据通路细分

7.2.1 单周期数据通路流水改造

要将单周期数据通路改造成流水线架构，只需在图中虚线位置加入长条形的流水寄存器部件，如图 7.6 所示，流水寄存器用于锁存前段加工处理完毕的数据和控制信号，通常这些数据和控制信号都会横穿流水寄存器传递到下一段，注意后文中所有横穿流水寄存器的线缆默认都是具有相同功能的数据或信号。这里总共增加了四个流水寄存器，根据其所连接的功能段的名称分别命名为 IF/ID、ID/EX、EX/MEM、MEM/WB，数据通路被被 4 个流水寄存器细分为五段流水线。注意 WB 段的后面没有流水寄存器，但该段的数据最终写回到了寄存器堆，程序计数器 PC 也可以看做一个流水寄存器，为 IF 段取指令提供数据。

所有流水寄存器，程序计数器 PC、寄存器堆、数据存储器均采用统一时钟 CLK 进行同步，每来一个时钟，就会有一条新的指令进入流水线取指令 IF 段，同时流水寄存器就会锁存前段加工处理完成的数据和控制信号，为下一段的功能部件提供数据输入，指令流水线各功能段通过流水寄存器完成一次数据传送。

增加流水寄存器后，同一时刻各功能段可并行独立工作，指令流水线充满后将会有五条指令进入流水线并行执行。需要注意的是 ID 段的寄存器堆属于比较特殊的功能部件，其在 ID 段负责读取寄存器操作数的操作，读操作属于组合逻辑；同时还负责 WB 段的指令执行结果的写回操作，写入操作需要时钟配合，是时序逻辑。

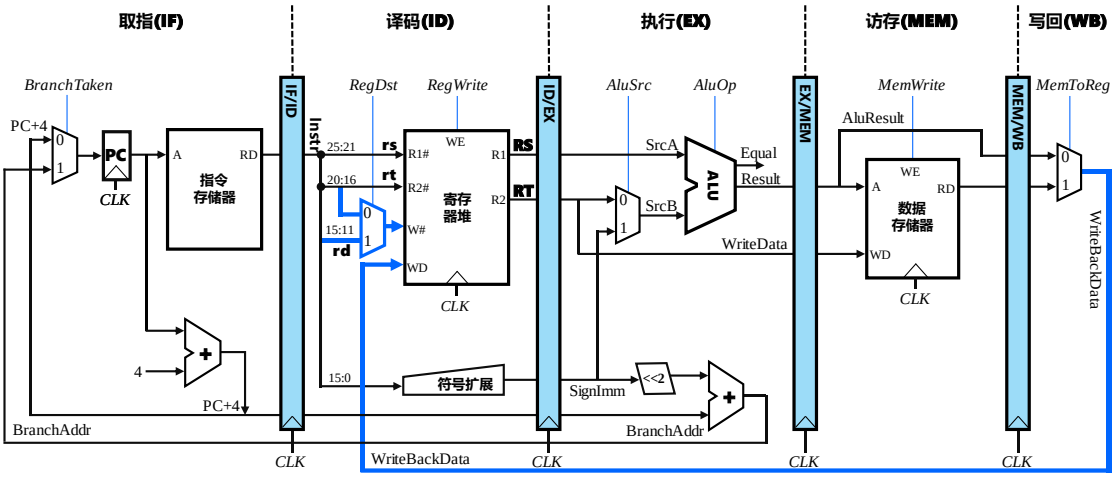

图 7.6 单周期数据通路流水改造

仔细观察图 7.6 中的数据通路，会发现寄存器堆的写回数据通路还存在一定的问题，寄存器堆写寄存器编号 W#的输入来源是根据 ID 段的指令字由 *RegDst* 信号控制多路选择器进行选择的，而写数据 WD 却来自于 WB 段，也就是写入地址和写入数据分属不同的指令，这会造成数据紊乱。所以这里还需要对数据通路进行适当的改造，首先调整 ID 段多路选择器输出的写寄存器编号的输出位置，其不再送寄存器堆的 W#端，而是直接送 ID/EX 流水寄存器锁存，并逐段依次向后传递到 WB 段，最后再由 WB 段的 MEM/WB 流水寄存器送回到寄存器堆的写寄存器编号 W#端口，具体见图 7.7 中粗线所示的数据通路，注意图中 ID 段多路选择器略微调整了位置，另外数据通路修改后 ID/EX、EX/MEM、MEM/WB 流水寄存器都需要增加位宽存放 *WriteReg#*数据信息。

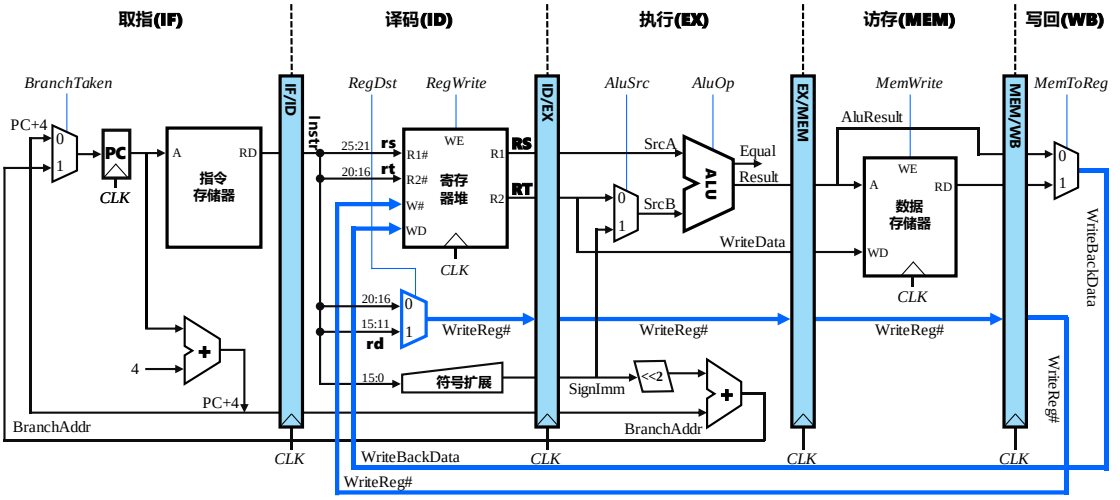

图 7.7 流水线中写回数据通路改造

不同的流水寄存器锁存传递的数据信息并不相同，图 7.7 中 IF/ID 流水寄存器需要锁存指令存储器取出的指令字以及 PC+4 的值；ID/EX 流水寄存器需要锁存从寄存器堆中取出的两个操作数 RS、RT（指令字中 *rs*，*rt* 字段对应寄存器的值），写寄存器编号 WriteReg# 以及立即数符号扩展的值、PC+4 等后段可能用到的操作数；EX/MEM 流水寄存器需要锁

存 ALU 运算结果、数据存储器待写入数据、写寄存器编号 *WriteReg#*等数据，MEM/WB 流水寄存器需要锁存 ALU 运算结果、数据存储器读出数据、写寄存器编号 *WriteReg#*等数据。

7.2.2 流水数据通路中的控制信号及传递

图 7.7 仅仅给出了数据通过流水寄存器进行传递的情况，但各段所需要的 7 个操作控制信号又是从哪里来，如何产生呢？

这里首先对流水线数据通路中所需要的主要控制信号先进行一个简单的分类，具体如表 7.1 所示，你会发现部分操作控制信号控制的功能段和信号来源并不一致，如 ID 段的分支跳转信号 *BranchTaken* 来源于 EX 段，ID 段的 *RegWrite* 信号来源于 WB 段。

表 7.1　控制信号分类

控制信号	位置	来源	功能说明
BranchTaken	**IF**	**EX**	分支跳转信号，为 1 表示跳转，由 EX 段的 Branch 信号与 equal 标志逻辑与生成
RegDst	ID	ID	写入目的寄存器选择，为 1 时目的寄存器为 rd 寄存器，为 0 时为 rt 寄存器
RegWrite	**ID**	**WB**	控制寄存器堆写操作，为 1 时数据需要写回寄存器堆指定寄存器
AluSrc	EX	EX	ALU 的第二输入选择控制，为 0 时输入寄存器 RT，为 1 时输入扩展后的立即数
AluOp	EX	EX	控制 ALU 进行不同运算，具体取值和位宽与 ALU 的设计有关
MemWrite	MEM	MEM	控制数据存储器写操作，为 0 时进行读操作，为 1 时进行写操作
MemToReg	WB	WB	为 1 时将数据存储器读出数据写回寄存器堆，否则写回 ALU 运算结果

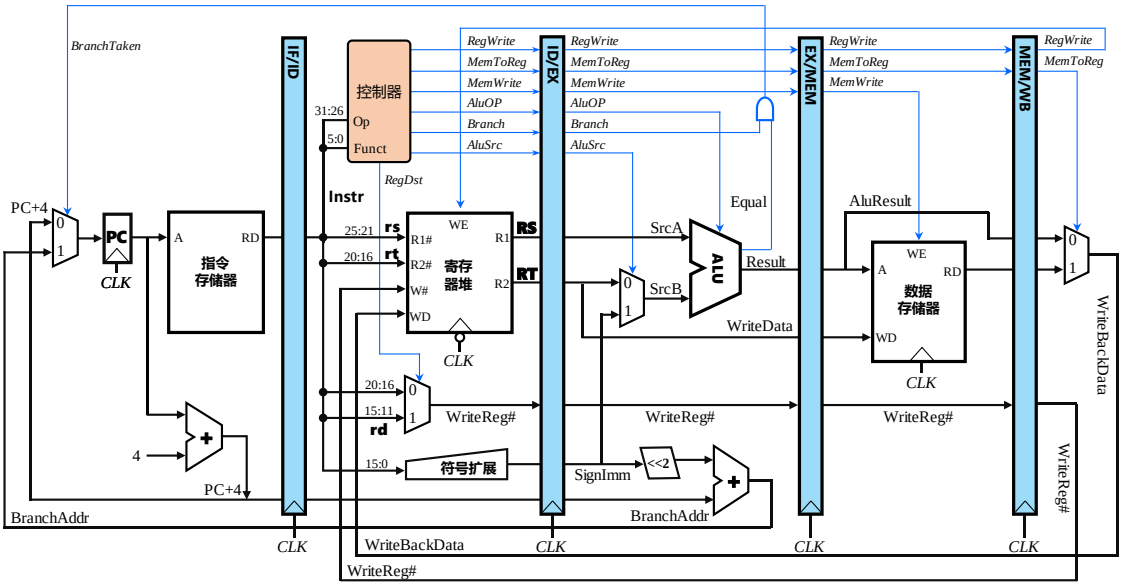

图 7.8 五段流水控制信号与传递

流水线数据通路由单周期数据通路改造而来，与单周期处理器使用了相同的操作控制信号，因此流水线也可以复用单周期处理器中的操作控制器。五段指令流水线中 ID 段负责指令译码生成操作控制信号，所以操作控制器应该设置在译码 ID 段，如图 7.8 所示，操作控制器输入为 IF/ID 流水寄存器锁存指令字中的 *opcode* 和 *funct* 字段，内部为组合逻辑电路，输出为 7 个控制信号，其中 *RegDst* 信号为 ID 段使用，其他 6 个后段使用的控制信号输出到 ID/EX 流水寄存器。*RegWrite* 信号必须传递至 WB 段后才能反馈到 ID 段的寄存器

堆写入控制端 WE，条件分支译码信号 *Branch* 也需要传递到 EX 段，与 ALU 运算的标志 *equal* 信号进行逻辑与操作后反馈到 IF 段控制多路选择器进行分支处理。

与传统工业生产流水线不同，在指令流水线中，通过流水线寄存器传递的不仅仅是当前指令当前阶段待加工的数据，还需要向后段传递数据如何进行加工的操作控制信号，这些控制信号要应与各功能段处理的指令同步，每一个操作控制信号均只在某一功能段使用一次，操作控制信号使用完毕后就不再向后段继续传递，因此图 7.8 中 ID 段产生的控制信号在从前到后(从左到右)传递的过程中数目逐渐减少。

7.2.3 指令在流水线中的执行过程

在单周期处理器中，不同类型的指令所使用的数据通路并不一样，经历的功能部件也不尽相同。同样在指令流水线中，不同指令所需要使用的数据通路也不一样，具体如表 7.2 所示，这里假设所有指令都需要进入 EX 段执行，从表中可以看出 *lw* 指令需要使用所有的 5 个功能段，而 *sw* 指令不需要 WB 段，*add*、*addi* 指令不需要 MEM 段、*beq* 指令不需要 MEM 段和 WB 段。

表 7.2　指令在流水线中执行情况

#	指令	IF	ID	EX	MEM	WB
1	*lw*	√	√	√	√	√
2	*sw*	√	√	√	√	
3	*beq*	√	√	√		
4	*add*	√	√	√		√
5	*addi*	√	√	√		√

需要注意的是，由于流水线的特殊结构，所有指令都需要完整经过流水线的各功能段，只不过某些指令在某些功能段没有任何实质性的操作，只是等待一个时钟周期，这也就意味着单条指令的执行时间还是五个功能段时间延迟的总和。假设表 7.2 中的 5 条指令按先后顺序执行，下面将通过图片集的方式展示该程序在流水线各功能段上执行的完整过程。

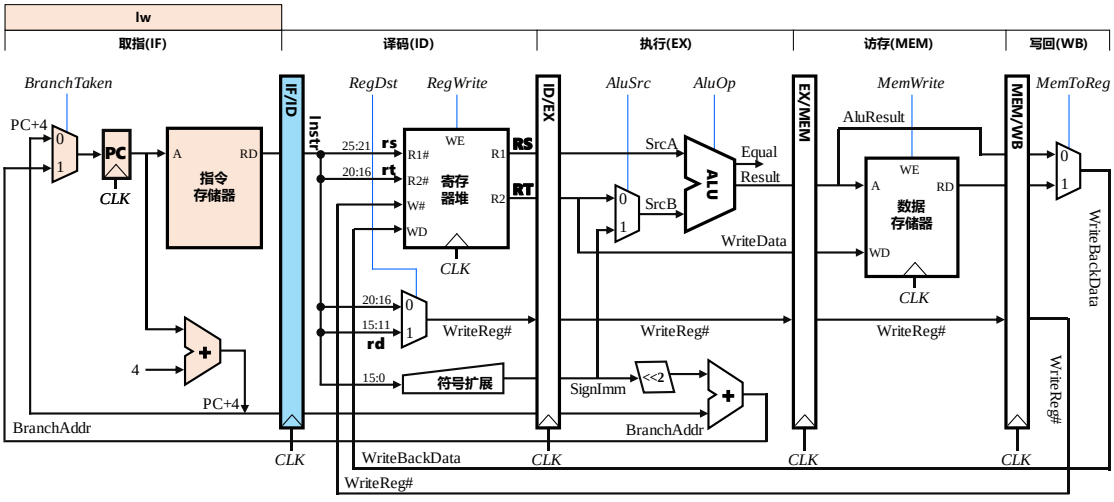

图 7.9　*lw* 指令进入 IF 段

1）取指令：图 7.9 为程序第一条指令 *lw* 指令进入 IF 段取指令的示意图，*lw* 指令在当

前时钟节拍所使用的数据通路用深色表示，*lw* 指令字由程序计数器 PC 提供的地址访问指令存储器得到，并将指令存储器 RD 输出端的 *lw* 指令字送 IF/ID 流水寄存器输入端；另外程序计数器 PC 的值与 4 相加形成顺序指令地址 PC+4，送 PC 输入端以便下一个时钟周期可以取出下一条指令。注意虽然 lw 指令在后续功能段并不会使用 PC+4，但 PC+4 还是会传送给 IF/ID 流水寄存器，以备其他指令（如 *beq*）使用。流水线各功能段并不区分指令的功能，所有数据信息和操作控制信号都来自段首的流水寄存器输出，所以只要是后续功能段有可能要用到的数据和控制信号都要向后传递。时钟到来时指令字将锁存在 IF/ID 流水寄存器中，同时 PC 更新为 PC+4 的值，*lw* 指令进入 ID 段，同时 IF 段取出下一条指令 *sw*。

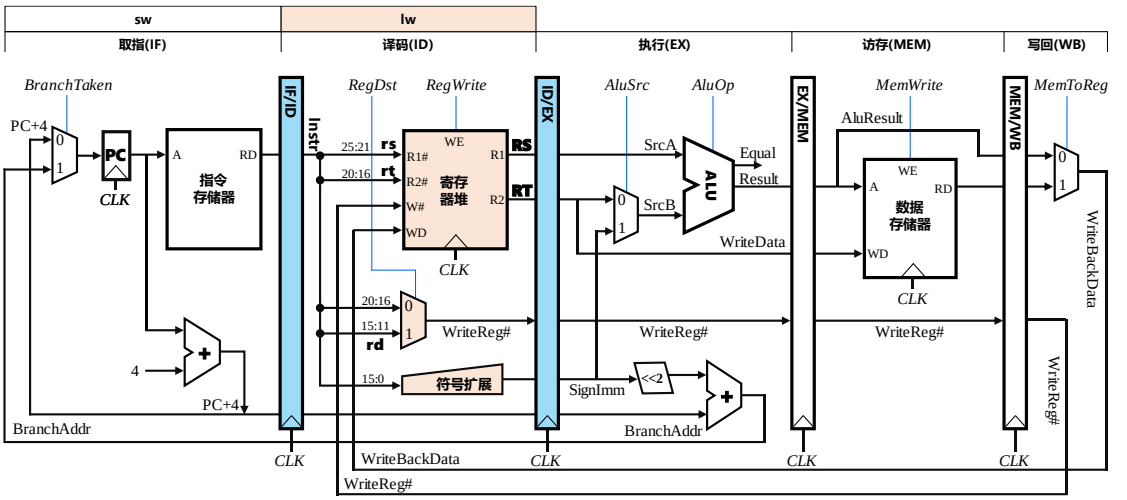

图 7.10　*lw* 指令进入 ID 段

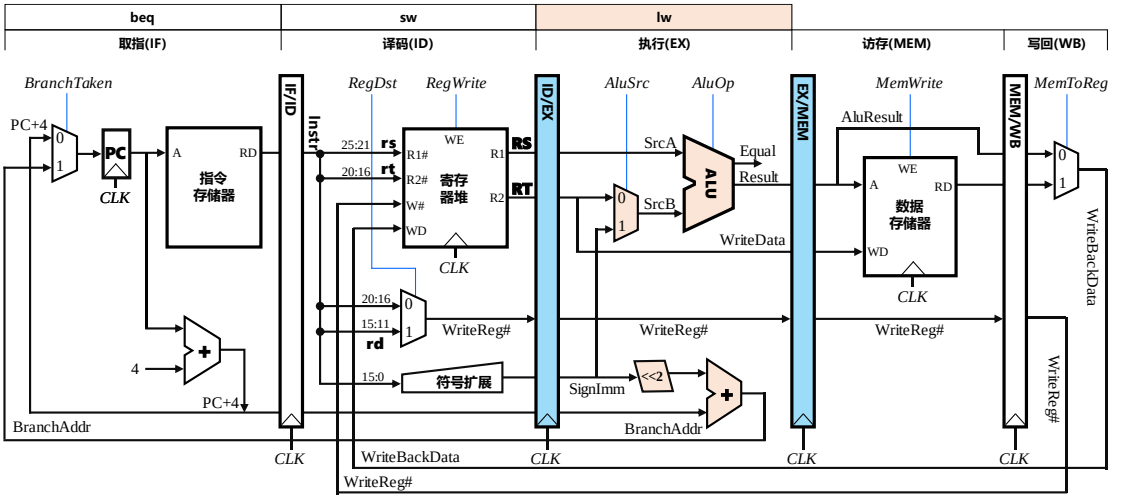

图 7.11　*lw* 指令进入 EX 段

2）指令译码、取操作数：图 7.10 为 *lw* 指令进入 ID 段译码的示意图，具体数据通路用深色表示，ID 段由操作控制器根据 IF/ID 流水寄存器中的指令字生成以后各段所需要的操作控制信号并向后传输，具体见图 7.8；另外 ID 段还会根据指令字中的 *rs*、*rt* 字段读取寄存器堆中的 *rs* 和 *rt* 寄存器的值 *RS*、*RT*；符号扩展单元会将指令字中的 16 位立即数符号扩展为 32 位；多路选择器根据指令字生成指令可能的写寄存器编号 *WriteReg#*（有些指令并

不需要写寄存器）。这 4 个数据连同顺序指令地址 PC+4 一起传输给 ID/EX 流水寄存器，时钟到来时这些数据信息连同操作控制器产生的操作控制信号都会锁存在 ID/EX 流水寄存器中，lw 指令进入 EX 段，同时 *sw* 指令进入 ID 段、*beq* 指令进入 IF 段。

3） 执行或访存地址运算：图 7.11 为 *lw* 指令进入 EX 段的示意图，具体数据通路用深色表示。对于 *lw* 指令来说，EX 段主要用于计算访存地址，将 ID/EX 流水寄存器中的 *RS* 的值与符号扩展后的立即数相加得到访存地址送 EX/MEM 流水寄存器；如果是 *beq* 指令 EX 段还需要计算分支目标地址，生成分支跳转信号 *BranchTaken*。ID/EX 流水寄存器中 *RT* 的值会在 MEM 段作为写入数据使用，所以 *RT* 的值会作为写入数据 *WriteData* 送 EX/MEM 流水寄存器；另外 ID/EX 流水寄存器中的写寄存器编号 *WriteReg#*也将直接传送给 EX/MEM 流水寄存器。同样时钟到来后这些数据信息连同后段所需要的操作控制信号都会锁存在 EX/MEM 流水接口中的寄存器中，*lw* 指令进入 MEM 段，*sw*、*beq*、*add* 指令分别进入 EX、ID、IF 段。

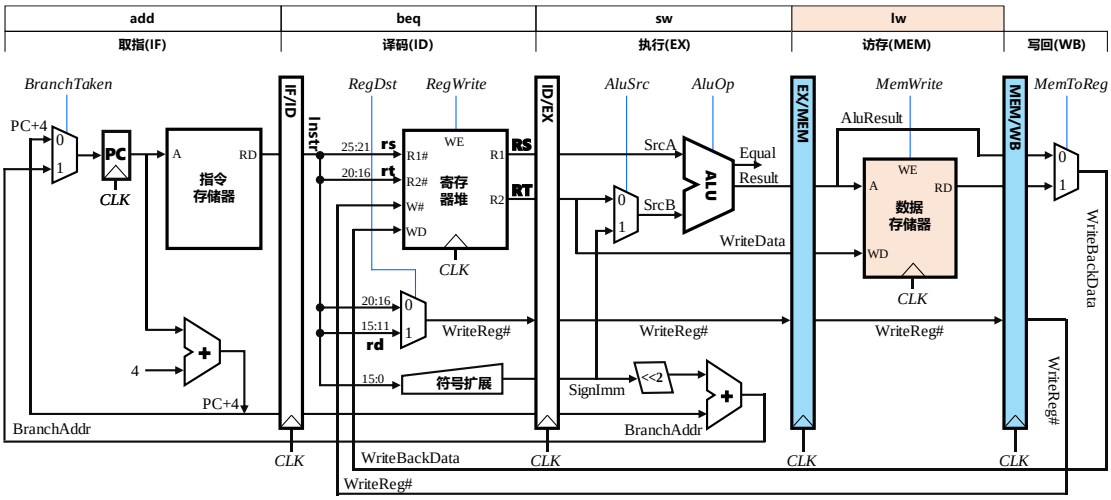

图 7.12　*lw* 指令进入 MEM 段

4) 存储器访问：图 7.12 为 *lw* 指令进入 MEM 段的示意图，具体数据通路用深色表示。该阶段功能比较单一，主要根据 EX/MEM 流水寄存器中锁存的 ALU 运算结果---访存地址和写入数据对于存储器进行读或写操作，EX/MEM 流水寄存器中的 ALU 运算结果、*WriteReg#*、数据存储器读出的数据都会送 MEM/WB 流水寄存器输入端，同样时钟到来后这些数据信息连同后段所需要的操作控制信号都会锁存在 MEM/WB 流水寄存器中，*lw* 指令进入 WB 段，*sw*、*beq*、*add*、*addi* 指令分别进入 MEM、EX、ID、IF 段，此时指令流水线充满。

5) **结果写回：** 图 7.13 为 *lw* 指令进入 WB 段的示意图，具体数据通路用深色表示。WB 段从 MEM/WB 流水寄存器中选择 ALU 运算结果或内存访问数据写回到寄存器堆指定寄存器 *WriteReg#*中，时钟到来时寄存器堆会完成数据写入，*lw* 指令离开流水线。注意此时 *sw* 指令也进入了最后阶段 MEM 段，同时 *beq* 指令也进入了最后阶段 EX 段，同一时刻实际上有 3 条指令执行完毕，当然这些指令即使执行完成也需要在流水线中继续向后传递直至 WB 段。

在指令流水线执行程序的过程中，会出现一些相互依赖的问题，比如中 ID 段的 *add* 指

令如果使用的源寄存器的值如果正好在 WB 段写回，就会争用寄存器堆，形成资源或结构冲突；还有 EX 段的条件分支指令 *beq* 如果成功分支，已经进入流水线 ID 段的 *add* 指令和 IF 段的 *addi* 指令就不应该继续执行，这就是分支冲突或结构冲突，下一节我们将重点讨论指令流水线的冲突处理。

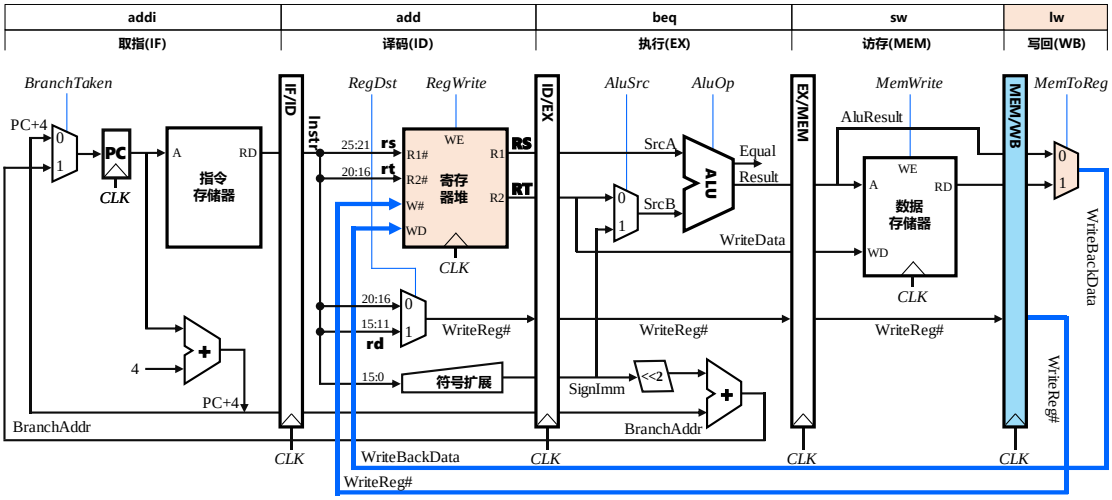

图 7.13　*lw* 指令进入 WB 段

7.3 流水线冲突与处理

7.3.1　流水线冲突

理想的流水线所有待加工对象均需要通过相同的阶段，不同阶段之间无共享资源，且各段传输延迟一致，进入流水线的对象也不应受其他功能段的影响，但这仅仅适合工业生产流水线，计算机指令流水线存在大量的的指令相关和数据依赖，通常会引起流水线的阻塞/暂停(*Stall*)。

所谓**指令相关**，是指在指令流水线中，如果某指令的某个阶段必须等到它前面的某条指令的某个阶段完成才能开始，也即是两条指令间存在着某种依赖关系，则两条指令存在指令相关。指令相关包括数据相关、结构相关、控制相关，指令相关会导致**流水线冲突/冒险**（Hazzard）。流水线冲突是指由于指令相关的存在，导致指令流水线出现“阻塞”或“暂停”，下一条指令不能在预期的时钟周期加载到流水线中。流水线冲突包括数据冲突、结构冲突、控制冲突三种。

1）　结构冲突

由于多条指令在同一时钟周期都需使用同一操作部件而引起的冲突称为结构冲突。假如流水线只有一个存储器，数据和指令都存放在同一个存储器中，当 *Load* 类指令进入 MEM 段时，IF 段也需要同时访问存储器取出新指令，这时就会产生访存结构冲突。这样的结构冲突实际上在单周期 CPU 的设计中就存在，解决方法是采用独立的指令存储器和数据存储器（哈佛结构），现代 CPU 中指令 *cache* 和数据 *cache* 分离也是这种结构，指令流水线中也可以采用同样的解决方案。另外还有一种方案是阻塞程序计数器 PC，使得 IF 段暂停一个时钟周期，下一个时钟来时同步清空 IF/ID 流水寄存器，进入 ID 段的是一个气泡操作

（全零的 MIPS 指令相当于空操作），等到 *Load* 指令访存操作结束以后，IF 段再次重新启动，相比哈佛结构这种方案会使得流水线暂停一个时钟周期，引起性能的损失。

2） 控制冲突

当流水线遇到分支指令或其他会改变 PC 值的指令时，在分支指令之后载入流水线的相邻指令可能因为分支跳转不能进入执行阶段，这种冲突称为控制冲突，也称为分支冲突。由于分支指令是否分支跳转、分支目标地址的计算要等到 EX 段才能确定（具体哪个功能段与设计有关），所以分支指令相邻的后续若干条指令已经预取进入流水线，当分支指令成功跳转时，流水线预取的指令不能进入执行阶段，此时需要清空这些预取指令，同时修改程序计数器 PC 的值，取出分支目标地址处的指令。发生控制冲突时，流水线会清空预取指令，浪费了若干时钟周期，这部分性能损失又称为**分支延迟**，会引起流水线性能降低。

通常 MIPS 中主要采用分支延迟槽(*Branch Delay Slot*)技术解决控制冲突，也就是分支指令后的一条或几条指令无论分支是否成功，都会进入执行阶段，这种方式减少分支延迟损失，现代 MIPS 处理器普遍在 ID 段执行分支指令，加上延迟槽技术，就可以将控制冲突引起的分支延迟损失降为零。但如何将程序中的有效指令放置在延迟槽中且不影响程序功能是编译器要解决的难题，当然最坏的情况是在延迟槽中放置一条空操作指令，延迟槽技术会给编译器带来麻烦，也会使得汇编程序代码可读性变差。

另外还可以采用基于软件或硬件的方法提前进行分支预测，在 IF 段根据程序计数器 PC 中指令地址查找历史分支信息统计表预测当前指令的下一条指令的正确地址，以尽量减少由于控制相关而导致流水线性能下降，现代高性能处理器中的分支预测算法精度已经非常高，新兴的 RISC-V 处理器就放弃了延迟槽技术，采用了动态分支预测技术，后面将进行详细介绍。

3） 数据冲突

当前指令要用到先前指令的操作结果，而这个结果尚未产生或尚未送达指定的位置，会导致当前指令无法继续执行，称为数据冲突。根据指令读访问和写访问的顺序，常见的数据冲突包括先写后读冲突（*Read after Write*，RAW）、先读后写冲突（*Write after Read*，WAR）、写后写冲突（*Write after Write*，WAW）。假定连续的两条指令 I1 和 I2，其中指令 I1 在指令 I2 之前进入流水线，两条指令之间可能引起的数据冲突如下：

(1)先写后读冲突 RAW

如果指令 I2 的源操作数是指令 I1 的目的操作数，这种数据冲突被称之为先写后读冲突 RAW。当指令按照流水的方式执行的时候，由于指令 I2 要用到指令 I1 的结果，如果指令 I2 在指令 I1 将结果写寄存器之前就在 ID 段读取了该寄存器的旧值，则会导致读取数据出错。下面是一个发生先写后读冲突的 MIPS 程序实例，第 2 条 *sub* 指令使用的源操作数$1 寄存器是第 1 条 *and* 指令的目的操作数，存在先写后读冲突。

```
1  and $1, $2, $3            # $1 寄存器为目的操作数
2  sub $2, $1, $3            # $1 寄存器为源操作数
```

(2)先读后写冲突 WAR

如果指令 I2 的目的操作数是指令 I1 的源操作数，这种数据冲突被称之为先读后写冲突（WAR）。前面例子程序中 *sub* 指令的 2 号寄存器就存在这种冲突，当指令 I2 去写该寄存器的时候，指令 I1 已经读取过该寄存器，所以这种数据相关对指令的执行不构成任何影响。

(3)写后写冲突 WAW

如果指令 I2 和指令 I1 的目的操作数是相同的，这种数据冲突被称之为写后写冲突（WAW），如前例中 *and* 指令和 *sub* 指令目的操作数都是 1 号寄存器。如果指令 I2 的写操作发生在指令 I1 的写操作之后，当指令按照流水的方式执行的时候，这种 WAW 冲突对指令的执行也没有影响，但在乱序调度的流水线中，有可能指令 I2 的写操作发生在 I1 指令的写操作之前，此时会发生写入顺序错误，目标单元中最终存储的是指令 I1 的执行结果，而不是指令 I2 的执行结果。

正常的程序都会存在着较多的 RAW 数据冲突，为了避免程序运行出错，最简单的处理方法就是推后执行与其相关的指令，直至目的操作数写入才开始取源操作数，以保证指令和程序执行的正确性。如果利用软件方法的解决就是在存在 RAW 冲突的指令间插入若干空指令直至这种冲突消失，这需要编译器的支持；如果采用硬件方法解决就是所谓插入“气泡”法，ID 段从寄存器堆取源操作数时如果检测到与 EX、MEM 或 WB 段指令存在数据冲突，则 IF、ID 段正在处理的指令暂停一个时钟周期(PC、IF/ID 流水寄存器值保持不变)，同时尝试在时钟到来时在 EX 段插入一个空操作气泡，先前进入 EX、MEM、WB 段的指令继续执行。下一个时钟到来 EX 段是一个空操作气泡，MEM、WB 段仍然存在指令，如果 ID 段指令仍然存在数据相关，继续重复暂停 IF、ID 段，在 EX 段插入气泡的逻辑，直至数据冲突完全消失。

7.3.2 结构冲突处理

当多条指令在同一时钟周期都需使用同一操作部件而引起的冲突称为结构冲突，也称**资源冲突**。在流水线设计中也会存在各种结构冲突，如计算 PC+4、计算分支目标地址、运算器运算都需要使用运算器，访问指令和访问数据都需要使用存储器。解决方案是增设加法部件避免运算冲突，增设指令存储器避免访存冲突，也可以通过插入气泡延缓取指令的方式解决访存冲突，但这种方式有流水线能损失。另外 ID 段读寄存器与 WB 段写寄存器的操作也存在结构冲突，但由于 MIPS 寄存器堆的读写逻辑是完全独立的逻辑，读写地址和数据均通过不同的端口进入，读写逻辑可以并发操作，所以这种结构冲突并不存在。

7.3.3 控制冲突处理

分支指令会引起控制冲突，要解决控制冲突，在执行程序分支跳转时必须清除流水线中的分支指令后续的若干条误取指令。以图 7.14 为例，EX 段正在执行一条 *beq* 指令，假设分支条件成立，也就是 ALU 的 *equal* 标志位为 1，分支指令译码信号 *Branch* 与 *equal* 信号逻辑与后得到分支跳转信号 *BranchTaken* 送 IF 段多路选择器选择控制端，选择分支目标地址送程序计数器 PC，具体见图中左上角的粗线所示的路径。EX 段负责将 ID/EX 流水寄存器中的符号扩展的立即数左移两位后和 PC+4 的值相加得到分支目标地址，并送 IF 段多路选择器送 PC 输入端，时钟到来时即可完成分支跳转，具体见图中左下角黑色粗线所示的数据通路。

在 EX 段执行分支指令，其后续相邻的两条指令 *add*、*addi* 分别进入了 ID、IF 段，由于程序不再顺序执行，所以这两条指令都属于误取进入流水线的指令，不应该继续在流水线中执行，需要从流水线中清空。

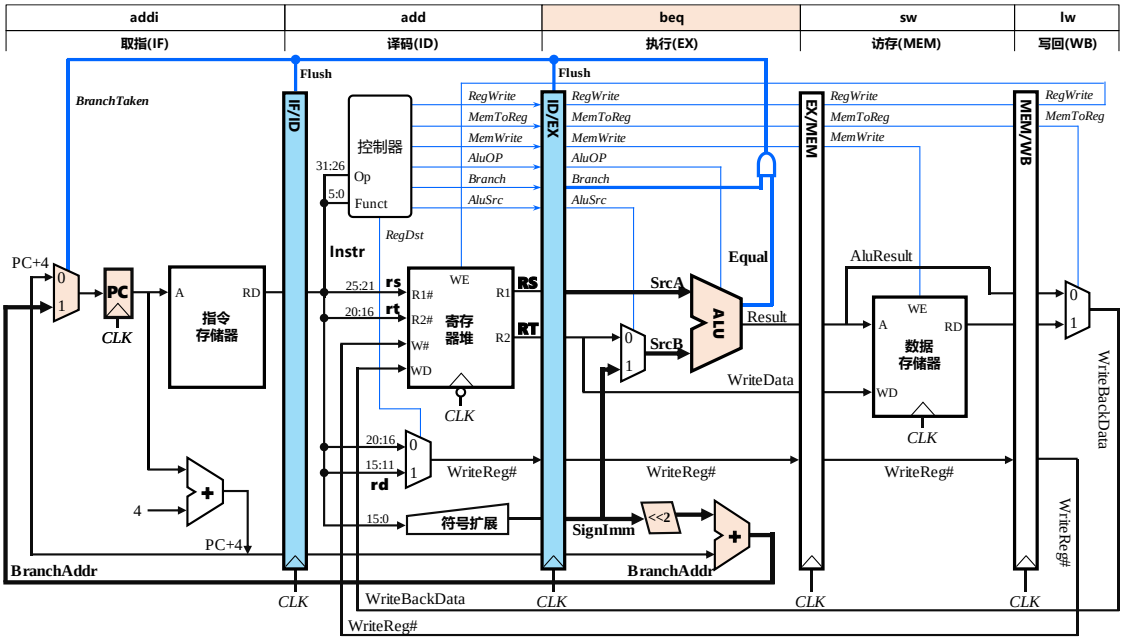

图 7.14 MIPS 流水线中的分支相关处理

为了实现指令清空操作，可以将分支跳转信号 *BranchTaken* 作为流水线清空信号 Flush 直接连接 IF/ID、ID/EX 流水寄存器的同步清零控制端，这样时钟到来后 *beq* 指令进入 MEM 段，而 IF 段取分支目标地址处的新指令，同时 IF/ID、ID/EX 流水寄存器中的数据和控制信号全部清零，由于全零的 MIPS 指令是 *sll $0,$0,0* 指令，等同于空操作 NOP 指令，不会改变 CPU 状态，所以 ID 段以及 EX 段的指令变成了空操作，误取的两条指令成功清除，如图 7.15 所示。

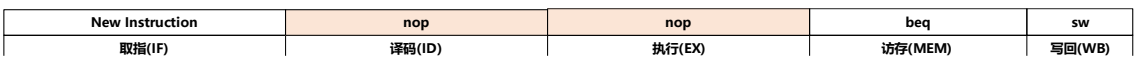

图 7.15 分支指令执行后流水线状态

需要注意的是这里流水寄存器清零信号必须是时钟敏感型的同步清零信号，而不是电平敏感的异步清零信号，如果采用异步清零信号，会导致 ID/EX 流水寄存器中 beq 指令所需的数据和控制信号被立即清除掉，程序无法实现正常分支。

```
1  beq  $0, $0, 8          # 跳转到第 4 条指令，分支目标地址为 PC+4+8
2  add  $1, $2, $3         # 不应执行
3  addi $4, $5, $6         # 不应执行
4  beq  $1, $2, 8          # 跳转到第 7 条指令，假设两寄存器不相等
5  lw  $5, 4($1)           # 不应执行
6  sw  $6, 8($1)           # 不应执行
```

图 7.16 为 MIPS 五段流水线执行上一段程序时处理分支相关的流水线时空图，注意这里横坐标是空间，代表各功能段部件，纵坐标是时间，每一格代表一个时钟节拍，单元格中为当前功能段当前节拍正在处理的指令，假设 *beq*、*bne* 指令一定会产生分支，从图中可以清晰的看出执行分支指令引起流水线清空误取指令的情况，图中深色格指令为误取指令，条纹格为清空指令形成的气泡。

CLKs	取指 IF	译码 ID	执行 EX	访存 MEM	写回 WB
1	**beq**				
2	*add*	**beq**			
3	*addi*	*add*	**beq**		
4	**beq**			**beq**	*sw*
5	*lw*	**beq**			**beq**
6	*sw*	*lw*	**beq**		
7	*Next Instr*			**beq**	

图 7.16 EX 段执行分支指令的流水空时图

上例中分支指令是在 EX 段执行，分支跳转会造成流水线被阻塞暂停两个时钟周期，分支指令的执行也可安排在其他功能段，在 ID、EX、MEM、WB 段执行分支处理时流水线中的误取指令数分别为 1、2、3、4，显然越早执行分支指令，分支延迟损失越小。图 7.17 为 ID 段执行分支指令的流水时空图，对比图 7.16 可以看出相比 EX 段执行分支节约了两个时钟周期。现代 MIPS 处理器多在 ID 段处理分支指令，配合延迟槽技术减少分支指令带来的流水损失。

CLKs	取指 IF	译码 ID	执行 EX	访存 MEM	写回 WB
2	*add*	**beq**			
3	**beq**		**beq**		
4	*lw*	**beq**		**beq**	
5	*Next Instr*		**beq**		**beq**

图 7.17 ID 段执行分支指令的流水空时图

综上所述，对于分支指令造成的控制相关，只需要在实际分支跳转时，将分支指令所在功能段左侧所有即将存放误取指令的流水寄存器同步清零即可，如果是 EX 段执行分支指令，需要清除 IF/ID、ID/EX 流水寄存器，如果是 ID 段执行，则只需要清除 IF/ID 流水寄存器。

7.3.4 插入气泡解决数据冲突

1. 程序中的数据相关

前面已经成功解决了流水线中的结构冲突和控制冲突，这里进一步解决数据相关引起的数据冲突，下面是一段存在数据相关的程序。

```
1  and $1, $1, $2          # $1 寄存器为目的操作数
2  sub $2, $1, $0          # $1 寄存器为源操作数,对应指令字段中 rs 字段
3  add $3, $1, $1          # $1 寄存器为源操作数,对应指令字段中 rs，rt 字段
4  or  $4, $5, $1          # $1 寄存器为源操作数,对应指令字段中 rt 字段
5  and $5, $6, $1          # $1 寄存器为源操作数,对应指令字段中 rt 字段
```

该程序包括五条指令，后四条指令的 *rs*、*rt* 字段均与第一条指令目的寄存器$1 存在数据相关。而在 MIPS 五段流水线中， ID 段从寄存器堆取操作数时才会发生数据相关，只需要考虑 ID 段指令 EX、MEM、WB 段的前三条指令之间的数据相关性。

ID 段和 WB 段的数据相关可以采用**先写后读**的方式解决，寄存器堆写入控制采用下跳

沿触发，而所有流水寄存器采用上跳沿触发（假设完整时钟周期从 1 开始，0 结束，中间是下跳沿），这样在一个时钟节拍的中间时刻(下跳沿)可以完成寄存器堆的数据写入，时钟节拍的后半段就可以利用组合逻辑读取寄存器正确的值。解决了 ID 段和 WB 段之间的数据相关，就只需要考虑连续三条指令的数据相关性，在流水线实现时只需要考虑 ID 段与 EX、MEM 两段之间的数据相关。

CLKs	取指 IF	译码 ID	执行 EX	访存 MEM	写回 WB
1	and $1, $1, $2				
2	sub $2, $1, $0	and $1, $1, $2			
3	add $3, $5, $1	sub $2, **$1**, $0	and **$1**, $1, $2		
4	add $3, $5, $1	sub $2, **$1**, $0		and **$1**, $1, $2	
5	add $3, $5, $1	sub $2, $1, $0			and $1, $1, $2
6	or $4, $5, $1	add $3, $5, $1	sub $2, $1, $0		

图 7.18 插入气泡解决数据相关的流水空时图

假设该程序已有 3 条指令进入流水线，如图 7.18 中第 3 个时钟周期。流水线应设置相应的硬件逻辑检测 ID 段指令与 EX、MEM 段指令的数据相关性，这里 ID 段 *sub* 指令与 EX 段 *and* 指令存在数据相关。由于 ID 段 *sub* 指令所需的$1 寄存器还没有写回，如果 *sub* 指令继续执行，将会取出错误的操作数值，因此只能阻塞 IF、ID 段指令的执行，并尝试在时钟到来时在 EX 段**插入气泡**以消除数据相关。下一个时钟，IF、ID 段仍是原有指令，EX 段变成空操作气泡，*and* 指令进入 MEM 段，此时 ID 段与 MEM 段仍存在数据相关，按原有处理逻辑继续阻塞 IF、ID 段指令执行，继续在 EX 段插入气泡。第 5 个时钟周期，*and* 指令进入 WB 段，EX 段、MEM 段全是气泡，ID 段 *sub* 指令数据相关消除，可正常取操作数，第 6 个时钟周期 *sub* 指令进入 EX 段。从图中可以看出，相邻的指令如果存在数据相关，需要先后插入两个气泡才能消除这种相关性。注意虽然插入气泡可以解决数据冲突，但会引起流水线阻塞暂停，影响指令流水线性能。

2. 数据相关检测与处理逻辑

流水线中必须增加硬件逻辑实现 ID 段与 EX、MEM 段指令的数据相关性检测，MIPS 指令包括 0~2 个源操作数，分别是 *rs*、*rt* 字段对应的寄存器，其中 0 号寄存器恒零，不需要考虑相关性。要想确认 ID 段指令使用的源寄存器是否在前两条指令中写入，只需要检查 EX、MEM 段的寄存器堆写入控制信号 *RegWrite* 是否为 1，且写寄存器编号 *WriteReg#*是否和源寄存器编号相同即可，因此流水线中的数据相关检测逻辑如下：

```
DataHazzard =  RsUsed & (rs≠0) & EX.RegWrite  & (rs==EX.WriteReg#)
             + RtUsed & (rt≠0) & EX.RegWrite  & (rt==EX.WriteReg#)
             + RsUsed & (rs≠0) & MEM.RegWrite & (rs==MEM.WriteReg#)
             + RtUsed & (rt≠0) & MEM.RegWrite & (rt==MEM.WriteReg#)
# rs、rt 分别表示指令字中的 rs、rt 字段，分别对应指令字中的 25~21、20~16 位
# RsUsed、RtUsed 分别表示 ID 段指令需要读 rs、rt 字段对应的寄存器
# EX.RegWrite 表示 EX 段的寄存器堆写使能控制信号 RegWrite，锁存在 ID/EX 流水寄存器中
# MEM.WriteReg#表示 MEM 段的写寄存器编号 WriteReg#，锁存在 EX/MEM 流水寄存器中
```

有了数据相关检测逻辑，只需考虑如何暂停 IF、ID 段指令的执行以及如何插入气泡的问题，插入气泡可以参考控制相关中的流水清空信号 *Flush*，当发生数据相关时给 ID/EX 流水寄存器一个同步清空信号 *Flush* 即可；而要暂停 IF、ID 段指令执行，只须保证程序计数器 PC 的和 IF/ID 流水寄存器的值不变即可，要做到这一点，只需要控制寄存器使能端即可，当使能端为 1 时，寄存器正常工作，为 0 时则忽略时钟输入，寄存器值保持不变。只需要将数据相关检测逻辑生成的数据相关信号 DataHazzard 作为暂停信号 *Stall* 取反后送对应的使能端即可。

进一步综合控制冲突处理逻辑可知流水线清空信号 IF/ID.*Flush*、ID/EX.*Flush*，以及流水线阻塞暂停信号 *Stall* 的逻辑表达式如下：

```
Stall = DataHazzard                          # 数据相关时要阻塞暂停 IF、ID 段指令的执行
PC.EN = ~Stall                               # 程序计数器 PC 使能端输入
IF/ID.EN = ~Stall                            # IF/ID 寄存器使能端输入
IF/ID.CLR = BranchTaken                        # 出现分支跳转时要清空 IF/ID
ID/EX.CLR = Flush = BranchTaken + DataHazzard    # 出现分支或数据相关时要清空 ID/EX
```

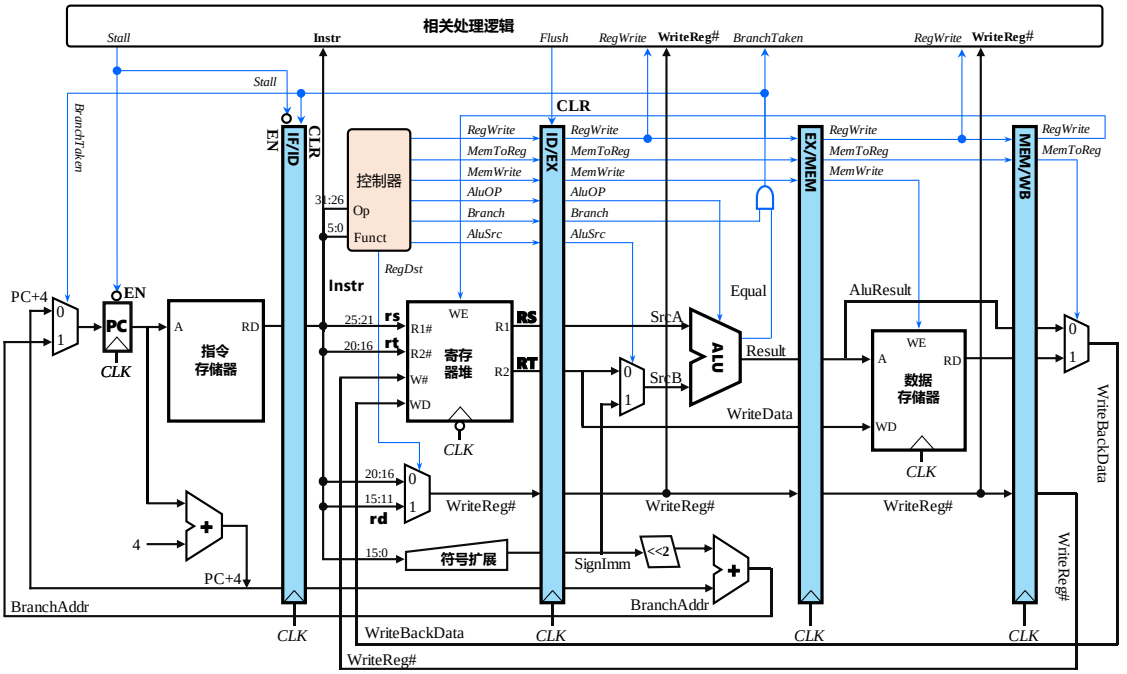

图 7.19 气泡流水线顶层视图

需要注意的是，当 EX 段如果是分支指令，且分支条件成立时，如果此时 ID 段同时检测到数据相关，那么 IF/ID 流水寄存器将同时接收到阻塞信号 *Stall* 和清空信号 *Flush*，IF/ID 流水寄存器应该如何动作呢？根据流水线控制冲突处理逻辑，此时 ID 段的指令不论是否相关都属于误取指令，所以不应该继续执行，所以应该优先进行同步清零的动作，在设计流水接口部件的时候要注意，同步清零信号不受使能端控制。

采用插入气泡的方式处理数据相关的处理后，流水线数据通路如图 7.19 所示，图中相关处理逻辑的输入信号除 EX、MEM 段的 *RegWrite*、*WriteReg*#信号外，还包括 ID 段的指令字 Instr、EX 段的分支跳转 *BranchTaken* 信号，输出则为阻塞暂停信号 *Stall*、流水清空信

号 ID/EX.*Flush*，具体可根据前面的逻辑表达式设计生成组合逻辑电路。

例 7.1 请给出如下程序在图 7.19 所示气泡流水线中运行的时空图。

```
1  lw  $5, 4($1)          # $5 为目的寄存器
2  add $6, $5, $7         # $5 依赖第 1 条指令的访存结果
3  sub $1, $2, $3         # 无数据相关
4  or  $7, $6, $7         # $6 依赖第 2 条指令的运算结果
5  and $9, $7, $6         # $7 依赖第 4 条指令的运算结果
```

解：该程序第 2 条和第 1 条指令在$5 寄存器存在数据相关，第 3 个节拍，当第 2 条 *add* 指令进入 ID 段时，流水线检测到 ID 段 *add* 指令与 EX 段 *lw* 指令的数据相关性，此时 IF，ID 段暂停，EX 段插入一个气泡，第 4 个节拍 ID 段 *add* 指令与 MEM 段 *lw* 指令仍然存在数据相关，继续暂停 IF、ID 段，再次插入一个气泡，数据相关解除；另外第 7 拍 ID 段会检测到第 4 条 *or* 指令与 MEM 段 *add* 指令的数据相关，会插入一个气泡。第 9 拍 ID 段 *and* 指令和 EX 段 *or* 指令存在数据相关，第 10、11 拍均会在 EX 段插入两个气泡，具体流水时空图如图 7.20 所示。

CLKs	取指 IF	译码 ID	执行 EX	访存 MEM	写回 WB
1	lw $5, 4($1)				
2	add $6, $5, $7	lw $5, 4($1)			
3	sub $1, $2, $3	add $6, **$5**, $7	lw $5, 4($1)		
4	sub $1, $2, $3	add $6, **$5**, $7		lw **$5**, 4($1)	
5	sub $1, $2, $3	add $6, $5, $7			lw $5, 4($1)
6	or $7, $6, $7	sub $1, $2, $3	add $6, $6, $7		
7	and $9, $7, $6	or $7, **$6**, $7	sub $1, $2, $3	add **$6**, $6, $7	
8	and $9, $7, $6	or $7, $6, $7		sub $1, $2, $3	add $6, $6, $7
9	xor $5, $9, $3	and $9, **$7**, $6	or **$7**, $6, $7		sub $1, $2, $3
10	xor $5, $9, $3	and $9, **$7**, $6		or **$7**, $6, $7	
11	Next Instr	and $9, $7, $6			or $7, $6, $7

图 7.20 气泡流水线时空图

第 7 个时钟节拍气泡流水线数据通路图 7.21 所示。图中相关处理逻辑将 ID 段 *or* 指令使用的两个源寄存器编号与 EX、MEM 段的写寄存器编号 *WriteReg#*进行比较，并结合 EX、MEM 段的寄存器写入控制信号 *RegWrite* 的值，判断存在数据相关，生成 ID/EX.CLR 需要的 *Flush* 清空信号，同时生成 *Stall* 暂停信号阻塞 PC 和 IF/ID 寄存器，锁存 IF、ID 两段指令不变，具体数据通路如图中加粗线缆所示。时钟上跳沿到来后，ID/EX 流水寄存器同步清零，EX 段插入了一个气泡，此时数据相关消失，相关处理逻辑自动撤除阻塞暂停信号 *Stall*、清空信号 *Flush*，流水线重新恢复正常。

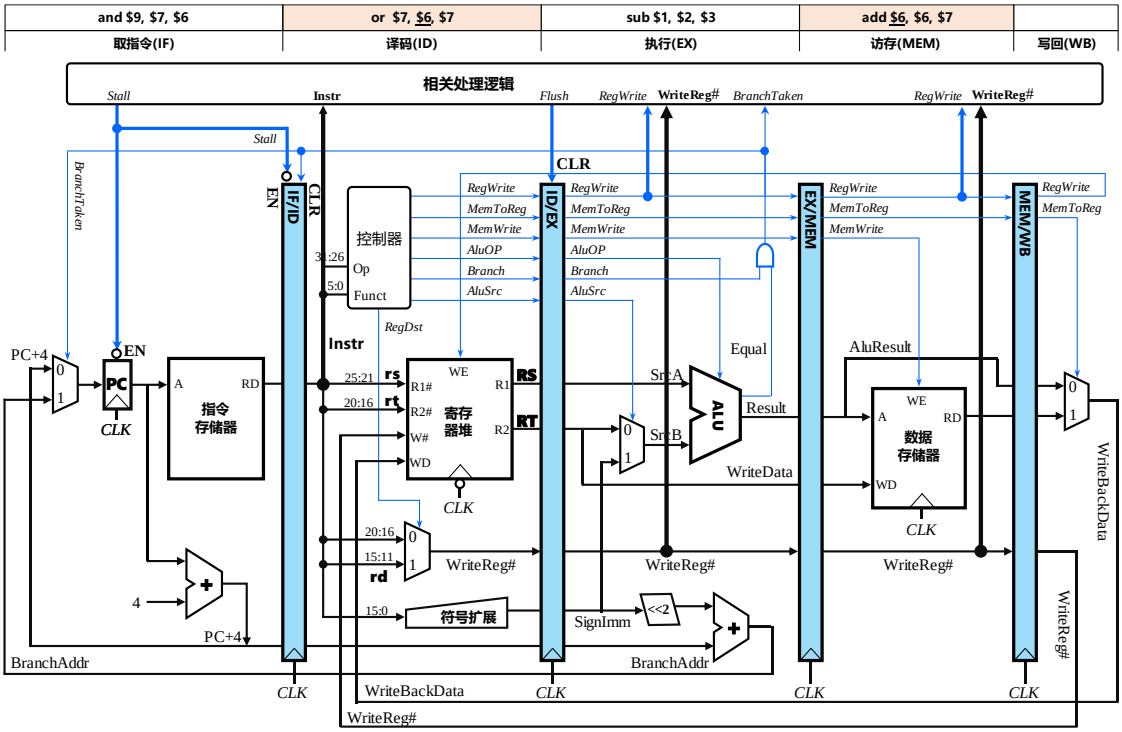

图 7.21 气泡流水线数据相关处理

7.3.5 使用重定向解决冲突

1. 重定向原理

气泡流水线通过延缓 ID 段取操作数动作的方式解决数据冲突问题，但大量气泡的插入会严重影响指令流水线性能，还有一种思路是先不考虑 ID 段所取的寄存器操作数是否正确，而是等到指令实际使用这些寄存器操作数时再考虑正确性问题。

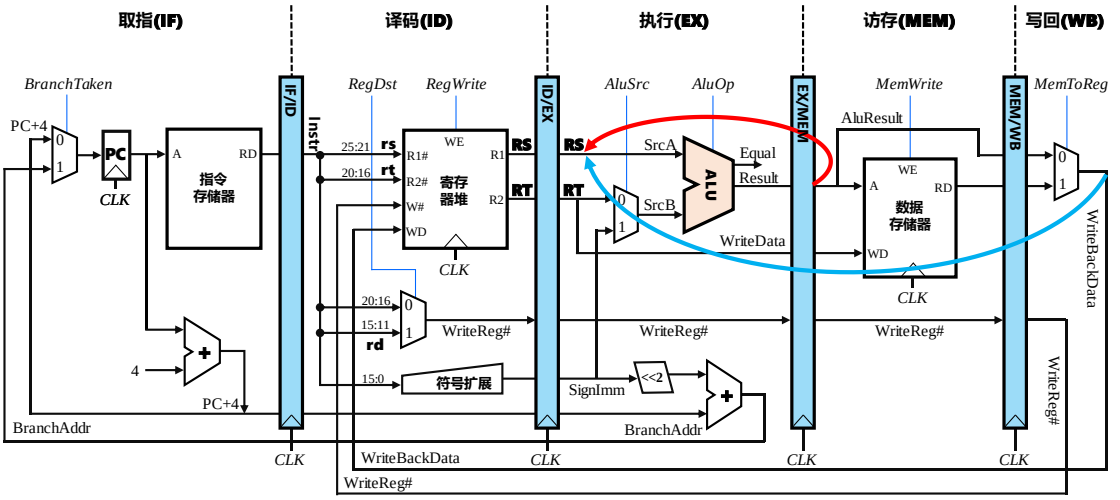

图 7.22 数据重定向示意图

如不考虑 ID 段取操作数的正确性，对应指令进入 EX 段可能和前两条指令也就是 MEM、WB 段均存在数据相关，如存在数据相关，EX 段的寄存器操作数 *RS*、*RT* 就是错误数据，

正确数据应来自于 MEM、WB 段指令的目的操作数，而这些指令已经通过了 EX 段完成了运算，除 *Load* 类访存指令外，目的操作数都已实际存放在 EX/MEM、MEM/WB 流水寄存器中，可以直接将正确的操作数从其所在位置重定向（Forwarding）到 EX 段合适的位置（也称为旁路 Bypass），如图 7.22 的弧线所示，可以将 EX/MEM 流水寄存器中的 *AluResult* 或 WB 段的 *WriteBackData* 直接送到 EX 段的 *RS* 处，作为 SrcA 送 ALU 参与运算，当然 *RT* 寄存器也可以采用这种方式处理，重定向方式无需插入气泡，可以解决大部分的数据相关问题，避免插入气泡引起的流水线性能下降，大大优化流水线性能。

以 ID/EX.RS 为例，此输出会送到 ALU 的第一个操作数端 SrcA，但可能不是最新的值，所以应在 ID/EX.*RS* 的输出端增加一个多路选择器 FwdA，此多路选择器的默认输入来源为 ID/EX.*RS*，另外也可能来自 EX/MEM.*AluResult* 的重定向，也可能是来自 MEM/WB.*AluResult* 或 MEM/WB.*ReadData*，如图 7.23 所示。为了简化实现，直接将 WB 段的多路选择器的输出 *WriteBackData* 重定向到 FwdA，多路选择器 FwdA 的选择控制信号为 *Rs_F*。

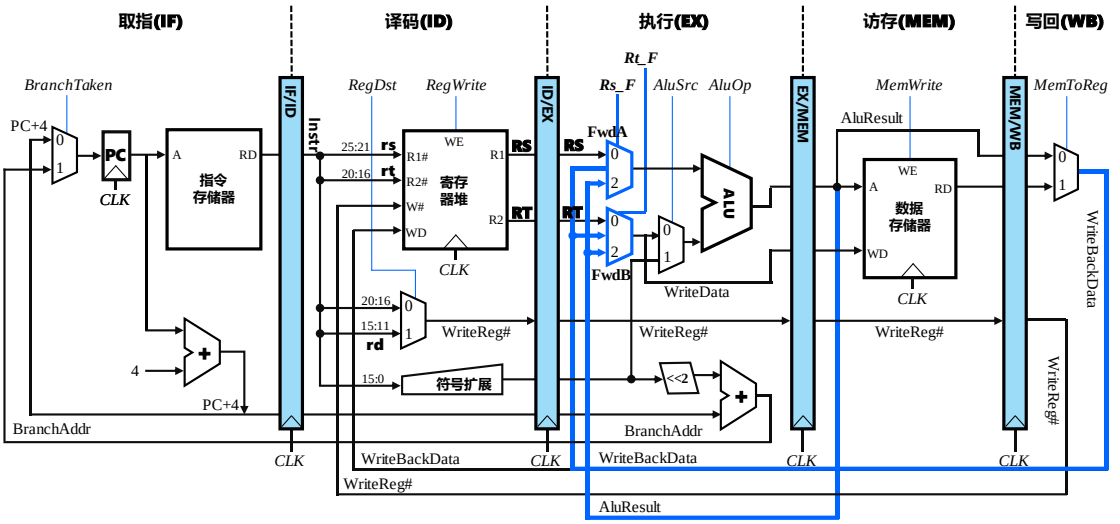

图 7.23 数据重定向数据通路

同样 ID/EX. *RT* 输出端也可以再增加一个多路选择器 FwdB 进行同样的重定向处理，FwdB 选择控制信号为 *Rt_F*，数据重定向的详细通路如图 7.23 中加粗线缆所示。这里两个多路选择器的选择控制可以根据数据相关检测情况自动生成，既可以直接在 EX 段生成，也可以在 ID 段进行数据相关检测时自动生成然后经过 ID/EX 流水寄存器传递而来，为了和气泡流水线数据相关检测机制一致，这里将采用第二种方法实现。

需要注意的是，如果相邻两条指令存在数据相关，且前一条指令是访存指令时（称为 *Load-Use* 相关），这种数据相关并不能采用重定向方式进行处理，如图 7.24 所示。图中 EX 段 and 指令和 MEM 段 *lw* 指令在$2 寄存器存在 *Load-Use* 相关，这时$2 号寄存器的值必须等待数据存储器读操作完成后才会出现在 RD 引脚上，如果直接将该引脚的输出值重定向到 EX 段，功能上可以实现，但这样做的后果是 EX 段的关键路径延迟变成了 MEM 段访存延迟加 EX 段运算器运算延迟，而流水线频率取决于流水线中最慢的功能段，这会使得流水线频率大大降低。

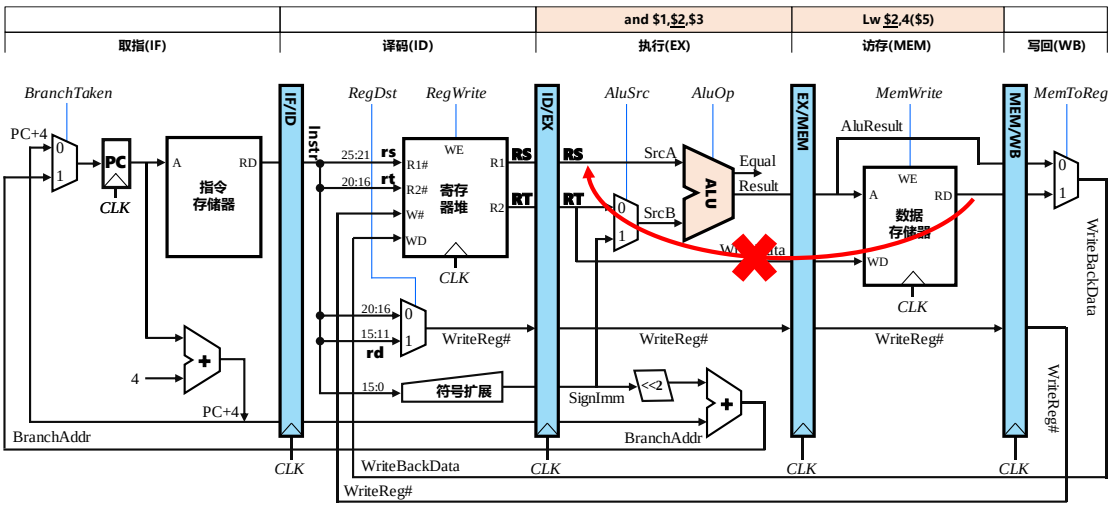

图 7.24 Load-Use 相关

所以对于 *Load-Use* 数据相关，不能采用重定向方式解决数据冲突，必须在发生 *Load-Use* 的两条相邻指令之间强制插入一个气泡以消除这种相关，插入气泡既可以在 EX 段完成也可以在 ID 段实现，同样为了和前面的实现方案统一，这里仍然在 ID 段实现。如图 7.25 所示，当 ID 段检测到 *Load-Use* 相关后在 EX 段插入一个气泡，这样当 *and* 指令进入 EX 段时，*lw* 指令已经抵达 WB 段，内存读出数据已经锁存在 MEM/WB 流水寄存器中，就可以采用重定向就解决这种数据相关。

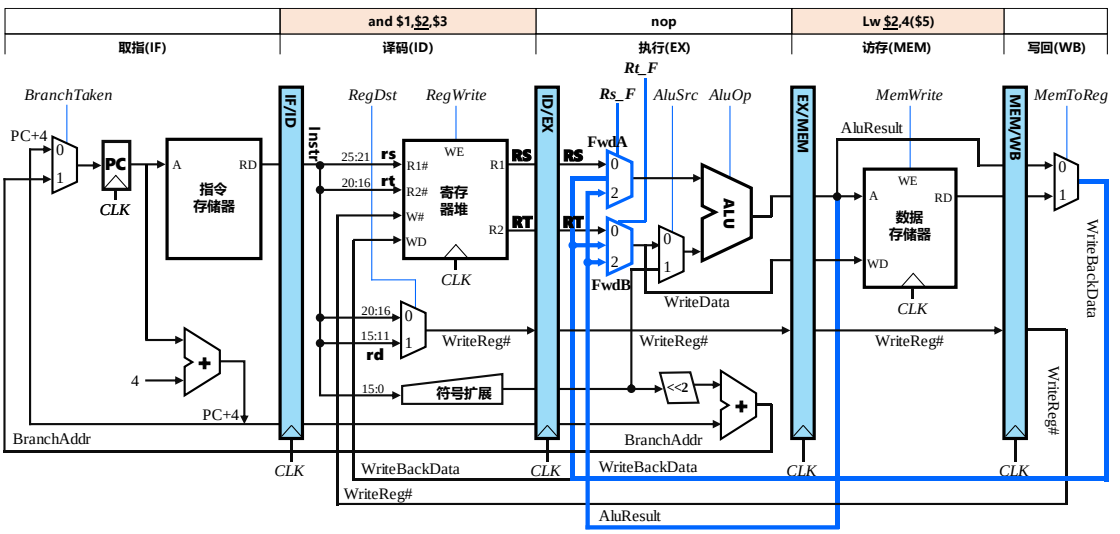

图 7.25 插入气泡消除 Load-Use 相关

需要注意的是本节介绍的数据重定向机制都是基于 EX 段执行指令的，现代 MIPS 处理器将分支指令提前到 ID 段执行，此时还应该增加到 ID 段的重定向通路，图 7.26 中 ID 段的 *beq* 指令与 EX、MEM 段指令均存在数据相关，似乎可以将后段对应的数据直接进行重定向解决这种数据相关。

但实际上这种重定向只是逻辑可行，EX 段 ALU 运算结果需要经过 ALU 的运算延迟才能得到，MEM 段数据存储器 RD 端口数据也需要经历完整的访存周期才能得到，都不能直

接重定向到 ID 段，这两类重定向都会大大增加 ID 段的关键路径延迟，影响流水线的时钟频率，此时唯一能进行重定向的数据只有 EX/MEM.*AluResult*。ID 段分支指令与 EX 段的数据相关，以及 ID 段分支指令与 MEM 段的 *Load-Use* 相关，还有 EX 段与 MEM 段的 *Load-Use* 相关都需要插入一个气泡。

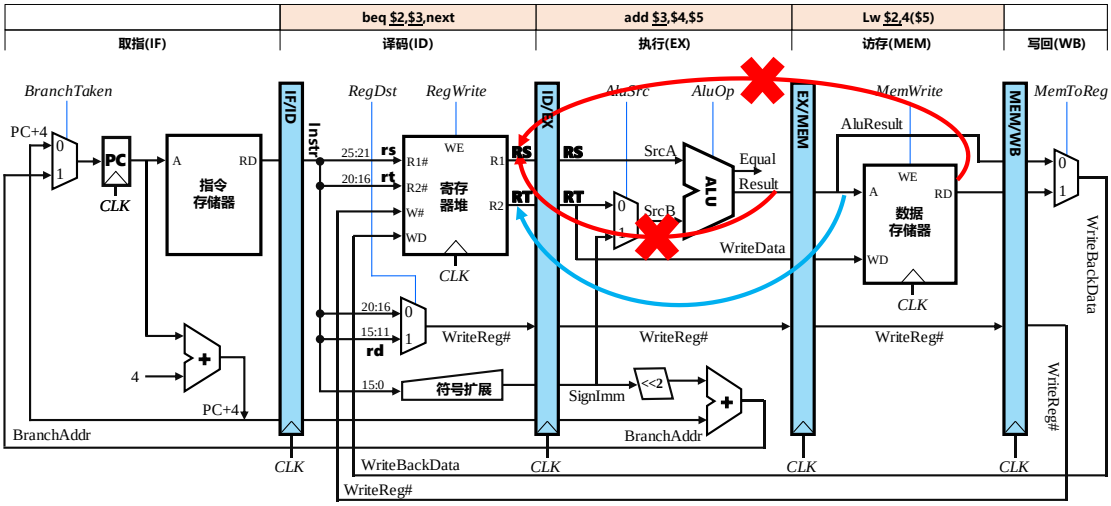

图 7.26 ID 段执行分支的重定向问题

2.采用重定向机制的数据相关检测与处理

采用重定向机制后，流水线中的相关处理逻辑必须进行适当的修订，由于重定向中 *Load-Use* 数据相关仍然需要通过插入气泡方式进行消除，所以相关处理逻辑应该能检测出 *Load-Use* 相关，其逻辑表达式如下：

```
LoadUse =  RsUsed & (rs≠0) & EX.MemRead & (rs==EX.WriteReg#)
+ RtUsed & (rt≠0) & EX.MemRead & (rt==EX.WriteReg#)
# 注意单周期 CPU 实现中为了简化电路，只实现了 MemWrite 写信号，没有实现 MemRead 信号，但由于该信号和 MemToReg 信号是同步的，所以可以用 MemToReg 信号代替 MemRead 信号
```

其他数据相关都可以采用重定向方式以无阻塞的方式解决，相关处理逻辑只需要在 ID 段生成两个重定向选择信号 *RsFoward*、*RtFoward* 传输给 ID/EX 流水寄存器即可，以 *RsFoward* 为例，其赋值逻辑如下：

```
IF (RsUsed & (rs≠0) & EX.RegWrite & (rs==EX.WriteReg#))
  RsFoward = 2            # ID 段与 EX 段数据相关
else IF (RsUsed & (rs≠0) & MEM.RegWrite & (rs==MEM.WriteReg#))
  RsFoward = 1            # ID 段与 MEM 段数据相关
else RsFoward = 0           # 无数据相关
```

当发生 Load-Used 相关时，需要暂停 IF、ID 段指令执行、并在 EX 段插入气泡，需要控制 PC 使能端 EN、IF/ID 使能端 EN、ID/EX 清零端 CLR，而 EX 段执行分支指令时会清空 ID 段、EX 段中的误取指令，会使用 IF/ID 清零端 CLR、ID/EX 清零端 CLR。综合两部分逻辑，可以得到相关处理逻辑阻塞信号 *Stall*、清空信号 *Flush*，各控制端口的逻辑：

```
Stall = LoadUse                                  # Load-Use 相关时要暂停 IF、ID 段指令执行
```

```
IF/ID.CLR = BranchTaken                          # 出现分支跳转时要清空 IF/ID
ID/EX.CLR = Flush = BranchTaken + LoadUse  # 分支跳转或 Load-Use 相关时要清空 ID/EX
PC.EN = ~Stall                                   # 程序计数器 PC 使能端输入
```

采用插入重定向方式处理数据相关的处理后，流水线完整数据通路如图 7.27 所示，图中相关处理逻辑输入除 EX、MEM 段的 *RegWrite*、*WriteReg#*信号外，还包括 ID 段的指令字 Instr、EX 段的分支跳转 *BranchTaken* 信号，输出为暂停信号 Stall、ID/EX.*Flush*、*RsForward*、*RtForward*，整体逻辑为组合逻辑电路。

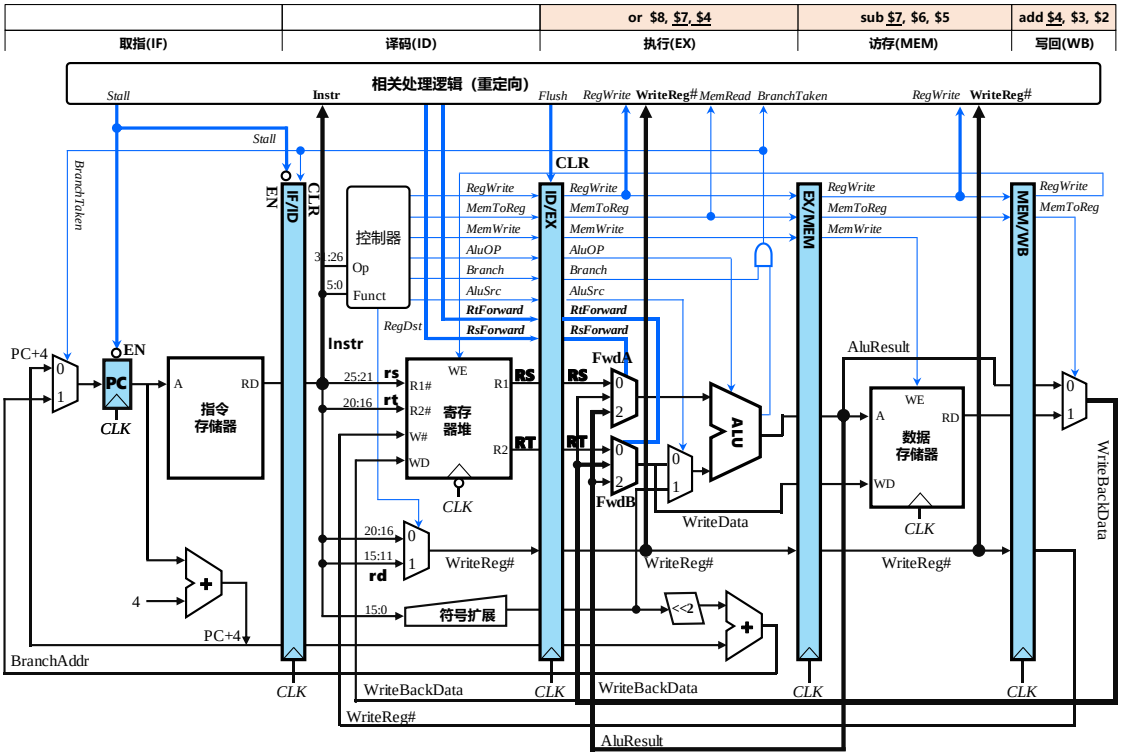

图 7.27 重定向流水线顶层视图

图 7.27 中 EX 段的 or 指令使用了 *rs*=7 和 *rt*=4 两个源寄存器，这两个寄存器分别和 MEM、WB 段两条指令数据相关，此时 EX.*RsFoward*=2、EX.*RtForward*=1，多路选择器 FwdA 选择 MEM 段 *AluResult* 输出到 ALU，多路选择器 FwdB 选择 WB 段的 *WriteBackData* 输出。注意 EX 段的 *WriteData* 也使用了重定向后的正确数据，所以 MEM 段不再需要考虑数据的相关问题。

例 7.2 下面的程序与例 2.1 中的程序完全相同，请给出该程序在图 7.27 中所示重定向流水线中运行的时空图。

```
1  lw  $5, 4($1)          # $5 为目的寄存器
2  add $6, $5, $7         # $5 依赖第 1 条指令的访存结果
3  sub $1, $2, $3         # 无数据相关
4  or  $7, $6, $7         # $6 依赖第 2 条指令的运算结果
5  and $9, $7, $6         # $7 依赖第 4 条指令的运算结果
```

解：该程序第 2 条和第 1 条指令在$5 寄存器存在 Load-Use 数据相关，第 3 个时钟节

拍，当第 2 条 *add* 指令进入 ID 段时，流水线检测到 ID 段与 EX 段 *lw* 指令的 *Load-Use* 数据相关，此时 IF，ID 段阻塞暂停，EX 段插入一个气泡即可消除该数据相关；

后续多条指令虽然存在数据相关性，但并不是 *Load-Use* 相关，所以不需要插入气泡，指令可以无阻塞的通过流水线，具体流水时空图如图 7.28 所示，相比气泡流水线，同样的程序，重定向流水线减少了 6 个气泡，大大提升了指令流水线的执行效率。

CLKs	IF	ID	EX	MEM	WB
1	lw $5, 4($1)				
2	add $6, $5, $7	lw $5, 4($1)			
3	sub $1, $2, $3	add $6, **$5**, $7	lw **$5**, 4($1)		
4	sub $1, $2, $3	add $6, $5, $7		lw $5, 4($1)	
5	or $7, $6, $7	sub $1, $2, $3	add $6, $6, $7		lw $5, 4($1)
6	and $9, $7, $6	or $7, $6, $7	sub $1, $2, $3	add $6, $6, $7	
7	xor $5, $9, $3	and $9, $7, $6	or $7, $6, $7	sub $1, $2, $3	add $6, $6, $7
8	Next Instr	xor $5, $9, $3	and $9, $7, $6	or $7, $6, $7	sub $1, $2, $3

图 7.28 重定向流水线时空图

7.3.6 动态分支预测技术

1.动态分支预测原理

采用重定向机制后，指令流水线中数据相关基本不需要插入气泡就可解决，只有少数 Load-Use 相关还需要插入一个气泡，流水线性能得到极大的提升。此时流水线中的控制冲突对流水线性能影响最大，基于加快经常性事件的原理，应优先考虑如何减少分支指令引起的分支延迟损失。为减少分支延迟损失，应尽可能提前执行分支指令，比如将分支指令放在 ID 段完成。

进一步降低分支延迟的主要方法有**静态分支预测**与**动态分支预测**两种。静态分支预测主要是基于编译器的编译信息对分支指令后续指令地址进行预测，预测信息是静态的不能改变的，与分支的实际执行情况无关，通常是采用一些简单的策略进行预测或处理，具体如下：

预测分支失败：这是缺省逻辑，与无分支预测的指令流水方案相同。

预测分支成功：其逻辑效果和第一种相同。

延迟分支：由编译器将一条或多条有用的指令或空指令放在分支指令后，作为分支指令的延迟槽，不论分支指令是否跳转，都要按顺序执行延迟槽中的指令。如果分支指令放在 ID 段执行，延迟槽技术可以完全消除分支延迟，但这需要编译器进行有效的指令调度，取决于编译器能否将程序中的有效指令放入延迟槽且不影响程序功能，对编译器有较高的要求。

动态分支预测依据分支指令的分支跳转历史，不断的对预测策略进行动态调整，具有较高的预测准确率，现代处理器中均支持动态分支预测技术。分支行为之所以可以预测是因为程序中分支指令的分支局部性，比如 *while* 循环生成的汇编代码第一条指令应该是判断表达式条件的分支指令，该指令只有在循环最后退出时才进行跳转，其他时间全部不跳

转；而 *do while* 循环正好相反，一次不跳转其他全跳转；*for* 循环生成的分支指令也有类似的局部性行为，动态分支预测正是利用了分支指令的分支局部性进行预测，相关程序详见本章习题。

最简单的动态分支预测策略是分支预测缓冲器（*Branch Prediction Buffer*），用于存放分支指令的分支跳转历史统计信息。BTB 表每个表项主要包括 *valid* 位、分支指令地址，分支目标地址，历史跳转信息描述位（预测状态位）、置换标记五项，如表 7.3 所示，其中 *valid* 位用于标记当前表项是否有效。BTB 表本质上是一个全相联的 *cache*，表项为 8 或 16 项，用于缓存经常访问的分支指令的分支跳转历史统计信息，BTB 表中的指令通常是程序中循环体的对应的分支指令。

每一条分支指令执行时，会将分支指令地址、分支目标地址、是否发生跳转等信息送 BTB 表，BTB 以分支指令地址为关键字，在 BTB 表内进行全相联并发比较，如果数据缺失，表示当前分支指令不在 BTB 表中，需要将该分支指令的相关信息载入，并设置合适的分支预测历史位初值，以方便后续预测，注意载入过程中可能涉及到淘汰。如果数据命中，表明当前分支指令历史分支信息已存放在 BTB 表中，此时需要根据本次分支是否发生跳转的信息调整对应表项中的分支预测历史位，以提升预测准确率，并且处理与淘汰相关的置换标记信息即可。

表 7.3　分支历史表 BTB 格式

#	Valid	分支指令地址	分支目标地址	分支预测历史位	置换标记
0	1（高电平有效）	xxxx	xxxx	11 预测跳转	xxxx
1	1	xxxx	xxxx	10 预测跳转	xxxx
2	1	xxxx	xxxx	01 预测不跳转	xxxx
3	1	xxxx	xxxx	00 预测不跳转	xxxx
4	0（无效）	xxxx	xxxx	xxxx	xxxx
5	0	xxxx	xxxx	xxxx	xxxx
6	0	xxxx	xxxx	xxxx	xxxx
7	0	xxxx	xxxx	xxxx	xxxx

分支预测历史位本质上是当前分支指令历史跳转情况的统计信息，是进行分支预测的依据，最早分支预测历史位仅采用一位数据表示，为 1 表示预测跳转，为 0 表示预测不跳转。科学研究表明，双预测位可在较低的成本下实现很高的预测准确率，所以现在普遍采用双位预测，一个典型的双位预测位状态转换图如图 7.29 所示。当状态位为 00、01 时预测不跳转，为 10、11 时预测跳转，当前分支指令是否发生跳转将会决定状态的变迁。注意，预测位初始值的设置也很重要，对于无条件分支指令，初始值如果是 00，则预测失败次数是 2 次，实际设计时应适当动态调整预测位初始值。

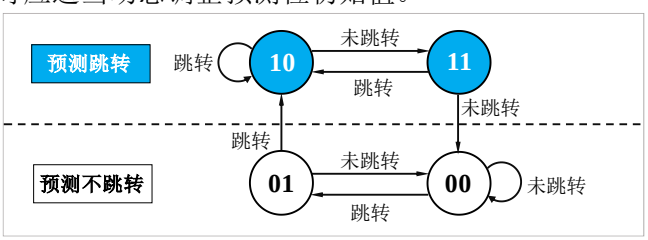

图 7.29　双预测位状态转换图

BTB 表会放在 IF 段，利用 PC 的值作为关键字进行全相联比较，此过程应与指令存储器取指令操作并发，并不需要取出指令即可进行分支预测。BTB 表命中表示当前指令是分支指令，可以根据 BTB 表中当前指令的历史预测位决定下条指令的地址是 PC+4 还是 BTB 表中的分支目标地址，注意这个分支目标地址不能在 IF 段取指令后计算，而是由 BTB 表中的 BTB 表项提供的。如果 BTB 表缺失，表明当前指令可能不是分支指令或者是不经常使用的分支指令，则按照 PC+4 取下条指令。

注意分支指令在 EX 段执行时，如果 IF 段预测正确，指令流水线不会停顿，如果预测失败，则分支指令在实际执行时还是应该清空误取的指令，并重新修正 PC 地址取出正确的指令。由于双位预测的高准确率，动态分支预测技术可以消除指令流水线中的大多数的分支延迟损失，新兴的 RISC-V 处理器中普遍采用动态分支预测技术。

2.动态分支预测硬件实现

动态分支预测逻辑必须用硬件实现，内部是一个全相联的 *cache* 结构，其主要输入输出引脚及功能说明如表 7.4 所示。

表 7.4　分支历史表 BTB 逻辑引脚说明

#	*valid*	I/O 类型	位宽	功能说明
1	CLK	输入	1	时钟控制信号，BTB 表载入新表项或更新预测位需要时钟配合
2	PC	输入	32	程序计数器地址，是在 BTB 表中全相联查找的关键字
3	EX.Branch	输入	1	EX 段分支指令译码信号，1 为分支指令，分支指令才会更新 BTB
4	EX.BranchTaken	输入	1	EX 段分支指令是否跳转，1 为跳转，0 为不跳转
5	EX.PC	输入	32	EX 段指令对应的 PC 地址
6	EX.BranchAddr	输入	32	EX 段分支指令的分支目标地址
7	PredictJump	**输出**	1	预测跳转信息位，向右依次传输给 EX 段，为 1 表示预测跳转
8	JumpAddr	**输出**	32	预测跳转位为 1 时输出 ID 段跳转指令的分支目标地址

PC 为 IF 段程序寄存器 PC 的输出值，IF 段利用 PC 值在 BTB 表中进行全相联比较，一旦数据命中，根据对应表项中的分支预测历史位的值输出预测跳转信息位 *PredictJump*，预测历史位为 10、11 时 *PredictJump*=1，表示程序要跳转执行，同时 BTB 表还要输出该指令的分支目标地址。如未命中，当前指令无法预测下条指令的地址，*PredictJump*=0，程序顺序执行。*PredictJump* 位最终用于选择顺序地址 PC+4 与分支目标地址中的一路送程序计数器 PC 输入端，这一部分逻辑由 ID 段执行，不会改写 BTB 表中的内容，是组合逻辑电路。

表中带 EX 前缀的四个输入都是流水线 EX 段反馈回来的数据和操作控制信号，是 BTB 表的写入逻辑，属于时序逻辑，由 EX 段负责。当 EX 段执行分支指令时，也就是 EX.*Branch*=1 时，BTB 表会根据 EX.PC 全相联查找，如数据缺失要将当前分支指令的信息载入 BTB 表，如果 BTB 表已满，还需要进行淘汰置换；如果数据命中，则只需要根据 EX 段指令实际分支跳转情况 EX.*BranchTaken* 的值依照预测位状态机更新分支预测历史位的值，以提升预测准确率。

图 7.30 为动态分支预测 BTB 逻辑的具体实现，注意图中指令存储器取指令和 BTB 分支预测逻辑是并发工作的，只需要知道指令地址即可进行分支预测，BTB 表的引入并不会

增加 IF 段的关键延迟。图中粗线为多位宽的数据，细线为操作控制信号。当 EX 段从段首的 ID/EX 流水寄存器接收到从 IF 段传送来的预测跳转信息 *PredictJump* 时，当实际跳转和预测跳转不符时就会将预测失败信号 EX.*PredictErr* 置 1，该信号反馈回 IF 段控制 PC 输入端的多路选择器重新取指令，具体取那一个地址的指令由实际跳转情况 EX.*BranchTaken* 选择，当 EX.*BranchTaken* 为 1 时应该选择从 EX.*BranchAddr* 处取指令，否则从 EX.PC+4 处取指令。

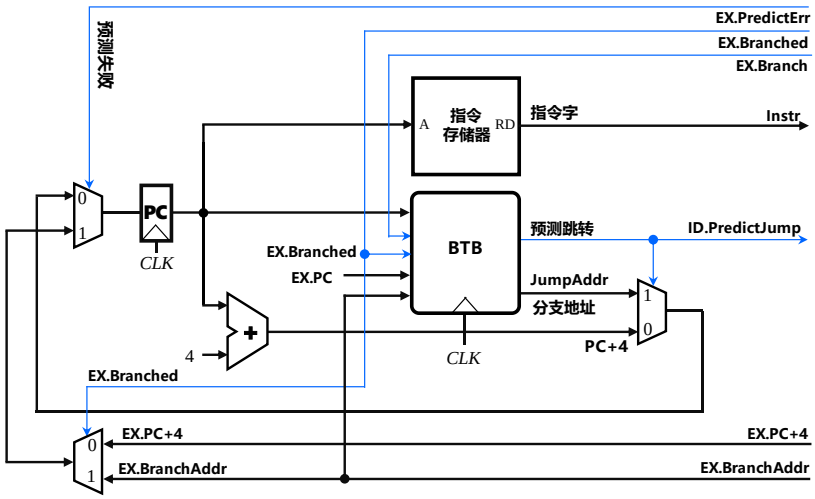

图 7.30　动态分支预测 BTB 逻辑实现

将图 7.30 的硬件逻辑加入 MIPS 指令流水线中就得到图 7.31 所示的支持动态分支预测的五段 MIPS 流水线，注意新的流水线数据通路中 IF 段增加了一个 *PredictJump* 信号需要通过流水线寄存器逐级传递到 EX 段，并最终与分支信号 *BranchTaken* 进行比较，不相等时表示预测失败。由于 EX 段只有检测到分支预测错误才会清空流水线中的误取指令，所以这里将预测失败信号 *PredictErr* 接入相关处理逻辑中原 *BranchTaken* 引脚即可复用原电路，其他逻辑并没有任何变化。

相关处理逻辑各输出信号、以及各流水线寄存器使能信号和清空信号逻辑如下：

```
Stall = LoadUse                              # 数据相关时要暂停 IF、ID 段指令执行
IF/ID.CLR = PredictErr                       # 预测失败时要清空 IF/ID
ID/EX.CLR = Flush = PredictErr + LoadUse     # 预测失败转或 Load-Use 相关时要清空 ID/EX
PC.EN = ~Stall                               # 程序计数器 PC 使能端输入
IF/ID.EN = ~Stall                            # IF/ID 寄存器使能端输入
```

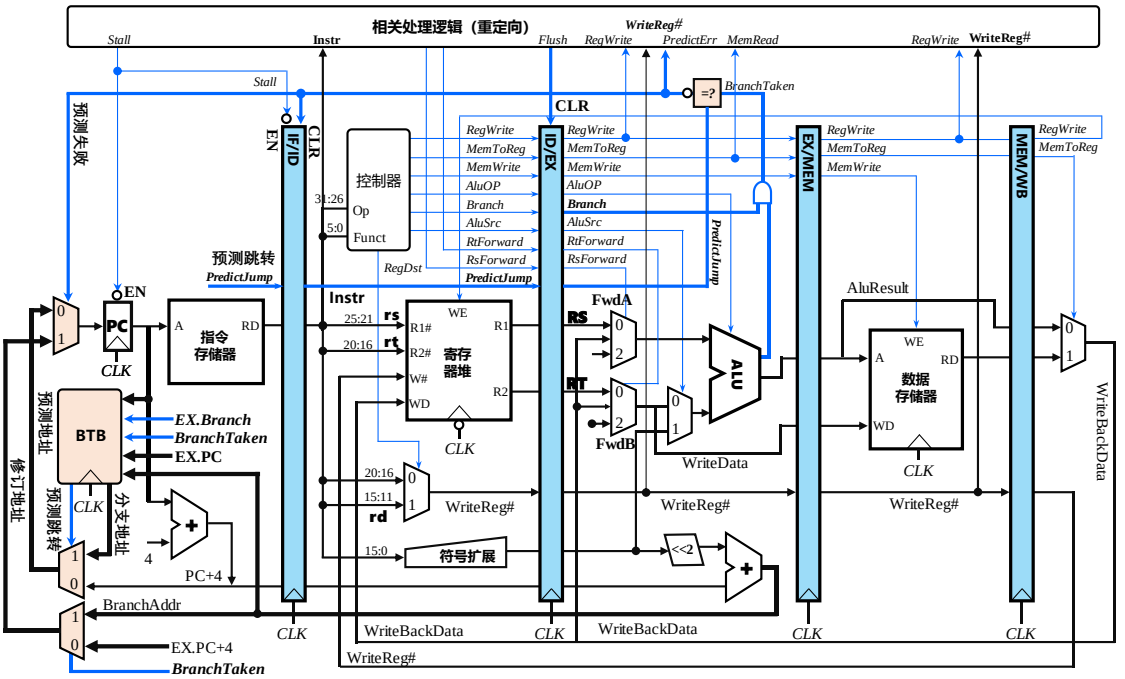

图 7.31 支持动态分支预测的 MIPS 五段流水线

采用动态分支预测后，分支指令在 IF 段进行预测，并根据预测情况取下条指令，同时将预测跳转信息 PredictJump 向后段传送。当分支指令进入 EX 段时，将预测跳转信息与实际跳转情况进行比较判断预测是否成功，BTB 表会根据预测是否成功更新表中对应指令的历史预测位。如果成功预测，流水线无需停顿，继续运行，如果预测失败，需要清空流水线中的误取指令，流水线停顿两个时钟周期，采用 BTB 后流水线运行的情况如表 7.5 所示。

表 7.5　采用 BTB 后流水线运行情况

指令在 BTB 中？	预测情况	下条指令地址	实际情况	预测情况	流水停顿周期
命中	预测跳转	分支目标地址	跳转	预测成功	0
命中	预测跳转	分支目标地址	未跳转	预测失败	2
命中	预测不跳转	顺序地址	跳转	预测失败	2
命中	预测不跳转	顺序地址	未跳转	预测成功	0
缺失		顺序地址	跳转		2
缺失		顺序地址	未跳转		0

7.3.7　流水线性能分析

指令流水线充满后每隔一个时钟周期即可完成一条指令，理想情况下指令流水线 CPI 应该是 1，但由于流水阻塞或清空会损失一些周期，所以指令流水线的实际 CPI 略高，且与具体执行程序的相关性存在密切的关系。

而流水线的最小时钟周期取决于各功能段中最慢功能段的关键延迟，如公式 7-1 所示。

$$T_{min\_clk} = \max(T_{if\_max}, T_{id\_max}, T_{ex\_max}, T_{ex\_max}, T_{mem\_max}, T_{wb\_max}) \quad (7\text{-}1)$$

以图 7.31 所示流水线数据通路为例，假设采用如表 6.8 所示的 65*nm*CMOS 工艺，各功能段时间延迟计算如表 7.6 所示，假设存储器读写延迟一致：

表 7.6 流水各功能段关键延迟

#	功能段	标识	功能段延迟	65nmCMOS 工艺实际值
1	IF	T_{if_max}	$T_{clk\_to\_q} + T_{mem} + T_{setup}$	300*ps* = 30+250+20
2	ID	T_{id_max}	$2^* (T_{clk\_to\_q} + T_{RF\_read} + T_{setup})$	400*ps* = 2*(30+150+20)
3	EX	T_{ex_max}	$T_{clk\_to\_q} + 2^*T_{mux} + T_{alu} + T_{setup}$	300*ps* = 30+2*25+200+20
4	MEM	T_{mem_max}	$T_{clk\_to\_q} + T_{mem} + T_{setup}$	300*ps* = 30+250+20
5	WB	T_{wb_max}	$T_{clk\_to\_q} + T_{mux} + T_{setup}$	*75ps* = 30+25+20

注意 ID 段采用先写后读方式解决 ID 段与 WB 段的数据相关，在时钟周期的后半段进行取操作数的过程,所以关键延迟应该是取操作数逻辑延迟的两倍。而读取寄存器操作数之前必须等待写回值稳定，所以还需要经历一个寄存器延迟 $T_{clk\_to\_q}$，再考虑寄存器读延迟 T_{RF_read} 和流水寄存器建立时间延迟 T_{setup}。

从表中可以看出，ID 段由于先写后读的操作称为五个功能段的瓶颈，流水线最小时钟周期应该是 400*ps*，最大时钟频率 2.5GHz，这个时钟周期并没有做到单周期处理器时钟周期 925*ps* 的 1/5，所以指令流水线实际性能提升并不是理想的 5 倍，这里大概也就 2.3 倍。

例 7.3 SPECINT2000 基准测试程序包含的取数据、存数据、条件分支指令、跳转指令、R 型算术逻辑运算指令比例分别为 25%、10%、11%、2%、52%。假设 40%的取数据指令存在 load-use 相关，而 1/4 的条件分支指令分支预测会失败，跳转指令不预测，不考虑其他冲突，请计算该程序在 MIPS 指令流水线中执行的 CPI。如果流水线时钟周期是 400ps，求测试程序执行时间。

解：程序的 CPI 等于各指令 CPI 的加权平均，取数据指令如果不存在 load-use，只需要一个时钟周期，如果存在 load-use 相关，需要插入一个气泡消除这种相关性，此时 CPI = 1，所以取数据指令的 CPI = 1×0.6 + 2×0.4 = 1.4。

假设条件分支指令在 EX 段执行，如果预测失败，则会引起流水线清空，造成两个时钟周期的损失，此时 CPI = 3，考虑预测失败因素，分支指令的 CPI = 1×0.75 + 3×0.25 = 1.5。

跳转指令不进行预测，一定会造成两个时钟周期的流水线能损失，所以 CPI = 3。其他指令 CPI = 1。

因此对于基准测试程序，在 MIPS 指令流水线上的 CPI 为：

$$CPI = 1.4\times25\% + 1\times10\% + 1.5\times11\% + 3\times2\% + 1\times52\% = 1.195$$

故测试程序执行时间为：

$$T_{total} = \text{指令条数}\times CPI\times T_{min\_clk} = 1000\times10^{-8}\times1.195\times400\times10^{-12} = 47.8\ \text{秒}$$

执行相同的测试程序，相比单周期处理器的 92.5 秒，多周期处理器的 133.9 秒指令流水线明显具有性能优势，相比单周期处理器约两倍，但远没有达到预期。这一方面是因为程序中的相关性引起了流水线能损失，另一方面是因为寄存器延迟 $T_{clk\_to\_q}$ 和流水寄存器建立时间 T_{setup} 叠加到了每一个功能段，且各功能段关键延迟不相等，导致流水线受限于最慢功能段的原因，在实际设计时应充分考虑这个问题，尽可能的将流水功能段划分的更细，

如 7.5 节介绍的超流水技术。

7.4 流水线异常与中断

中断和异常会改变程序的执行顺序，也会引发流水线的控制冲突，可以采用类似分支指令一样的方式进行处理，本节主要讨论指令流水线的中断异常处理机制。

1.中断类别

流水线中的中断异常处理与单周期 MIPS 有较大的不同，主要区别是同一时刻有多条指令进入流水线，每条指令都可能触发异常，表 7.7 给出了流水线各功能段可能出现的指令异常，从表中可以看出 IF 和 MEM 段主要是访存相关的异常，WB 段没有异常。注意这里的异常是因为异常指令执行而同步产生的，所以又称为**同步中断**，同步中断必须立即处理，处理完毕后异常指令可能需要重新执行；而外部 I/O 中断请求和硬件故障异常与指令无关，通常称为**异步中断**，不一定要立即响应，可以在处理器方便的时候进行处理。

表 7.7　五段流水各段指令异常分类

#	IF	ID	EX	MEM	WB
1	缺页或 TLB 异常	未定义指令	算术运算溢出	缺页或 TLB 异常	无异常
2	未对齐指令地址	除数为零	自陷异常	未对齐数据地址	
3	存储保护违例			存储保护违例	

2.异步中断处理

当流水线检测到异步中断请求时，由于中断请求与指令无关，那到底在流水线中的哪一段进行中断响应呢？在单周期处理器中 CPU 响应外部中断请求的时机是指令执行结束后的公操作阶段，该时刻之后的指令都不再执行。对于流水线来说，也需要给出一个功能段，该段指令之后的指令都不会执行，实际上只需要保证该段之后的指令均不改变 CPU 的状态（寄存器、存储器的值）即可。五段流水线中只有 MEM，WB 段有可能修改 CPU 状态，假设分支指令在 EX 段执行，则异步中断可以选择 IF、ID、EX、MEM 段，但考虑 IF 段、ID 段可能存在误取指令，并不是程序执行的正常路径，所以选择 EX 段、MEM 实现更加容易实现，当然实际中断处理时还需要考虑当前段的指令是否是气泡，否则无法保存正确的程序断点。

确定了中断响应的功能段后，当流水线检测到异常时，如果当前段不是数据相关引起的气泡，则进行中断响应，具体流程如下：

- **保存断点**：当前段的顺序地址 PC+4 作为断点送 CP0 协处理器中的 EPC 寄存器；
- **设置中断原因**：将 CP0 中的 CAUSE 寄存器设置为外部中断；
- **关中断**：设置 CP0 中 STATUS 寄存器中的中断使能位为 0；
- **中断识别**：将正确的中断程序入口地址送 IF 段程序计数器 PC；
- **指令清空**：将当前段指令之后的所有指令从流水线中清空。

中断服务程序的最后一条指令是 *eret* 指令，该指令会开中断，并将 EPC 寄存器的值送 IF 段程序计数器 PC，所以执行完中断服务程序后又可以回到原程序继续执行。

3.同步中断处理

对于同步中断，可以先考虑最简单的情况，假设同一时刻只有一条指令出现异常，此时中断异常处理流程和异步中断大同小异，具体流程如下：

- **保存断点**：由于大多数异常指令需要重新执行，所以应将异常指令地址送 EPC 寄存器，如果是自陷指令，断点还应该是顺序地址 PC+4。
- **设置中断原因**：将 CP0 中的 CAUSE 寄存器设置为具体异常类别；
- **关中断**：设置 CP0 中 STATUS 寄存器中的中断使能位为 1；
- **中断识别**：正确的异常处理程序入口地址送程序计数器 PC；
- **指令清空**：将异常指令及其后的所有指令从流水线中清空。

执行异常处理程序时操作系统会查看异常发生的原因并采取相应的操作，对于未定义指令、硬件故障异常或算术溢出异常，操作系统会终止用户程序并返回原因；对于其他可修复的异常，操作系统会尝试纠正异常，如缺页异常时操作系统会进行页面调度，完成异常处理后，会返回 EPC 寄存器存放的断点继续执行异常指令。注意在所有异常处理中，最重要也经常发生的是缺页和 TLB 异常。

同一时刻各功能段可能同时产生指令异常，这时就存在优先级的问题，指令异常处理应该严格遵循指令在程序中的先后顺序，按时间先后顺序各段异常指令处理的优先级应该是 MEM>EX>ID>IF，如果同时还有异步中断请求，其优先级最低。

在大多数 MIPS 处理器中，通常是最先发生异常的指令被中断，但这也会带来新的问题，下面这段程序，每条指令都对应一个异常，按照先后顺序，应该处理第 1 条 *lw* 指令引起的缺页异常，但是当 *lw* 指令进入 EX 段的时候，第 2 条未定义指令已经在 ID 段译码产生了异常，此时第 1 条指令还没有进入 MEM 段，未产生异常，这里流水线的多指令并发导致异常产生的顺序发生了紊乱。

```
1  lw  $5, 4($1)            # 缺页异常
2  xxx $5, $6, $7           # 未定义指令
3  add $1, $2, $3           # 运算溢出
```

如果此时立即处理 ID 段的异常，就会发生程序错误，为解决这种故障，可以不立即进入异常处理，各段检测到异常时只是记录异常的原因和并和断点一起存放在特殊寄存器中，通过流水寄存器逐级传递直至 WB 段，这样 WB 段的硬件逻辑检测检测到的异常一定是按实际顺序最先发生异常的指令，WB 段一旦检测到异常就可以将流水线中的所有指令全部清空，然后转异常服务处理程序执行。

7.5 指令级并行技术

指令流水线提升了指令执行的并行性，这种并行性又称为指令级并行(*instruction–level parallelism*,ILP)，要想进一步提升指令级并行性，主要可以采用超流水线技术和多发射技术两种方式。

超流水线(*superpipelined*)技术主要通过增加流水线功能段数目，尽可能细化减少各段关键延迟，从而提高流水线主频的方式提升流水线性能，例如 Pentium pro 的流水线就多达 14 段。

多发射 (*multiple issue*)技术类似工业流水线增加产能的方式，通过复制计算机内部功能部件的数量，如增加指令译码逻辑、寄存器堆端口数、运算器、重定向通路等，使得各流水功能段能同时处理多条指令，处理器一次可以发射多条指令进入流水线进行处理。一个四发射处理器，如果采用五段流水线，流水线充满后同一时刻流水线中将会有 20 条指令并行，当然这必然引起更多的相关性，冲突冒险问题更难处理。

如果这些冲突冒险全部交给编译器静态解决，就是**静态多发射**，如传统的超长指令字(*Very long Instruction Word*)技术，由编译器将多条无相关的常规指令储存在一个超长的指令字中，让它们同时发射到流水线中处理。这种方式硬件不处理冲突冒险，完全由编译器处理冲突冒险，对同时发射的指令是有严格要求的，有时候也只能采取插入空指令的方式解决冲突，编译器生成的代码可移植性较差，程序从一个处理器移植到另一个处理器上运行时，可能需要重新编译。

动态多发射技术由硬件动态处理多发射流水线运行过程中出现的各种冲突冒险，虽然多发射流水线也需要编译器对程序进行高效的调度以优化流水线性能，但即使编译器不处理，程序也可以在流水线上正确运行，这种技术也称为**超标量** (*superscalar*)技术。

采用超流水线技术的 CPU 在流水充满后每隔一个时钟周期可以执行一条机器指令，CPI=1，但其主频更高；而多发射流水线每个时钟周期可以处理多条指令，CPI<1，相对而言多发射流水线成本更高，控制更加复杂。结合二者的优势，也曾经出现过使用超流水线技术的多发射处理器。