第 5 章 静态流水线

前 4 章分别介绍了计算机系统结构的基本概念、二进制和逻辑电路以及指令系统结构。有了这些基础，这一章以一个简单的CPU为例介绍CPU的流水线设计，后面 2 章再介绍比较复杂的流水线和多发射结构。

我们从MIPS指令集拣选部分代表性的指令作为简单CPU需要实现的指令集，其中指令及其编码列举在表 5.1 中，指令的具体含义及指令集的其它定义请参看本书的第 4 章。

表 5.1 简单 CPU 指令和指令编码

	31-26	25-21	20-16	15-11	10-6	5-0
ADDU	000000	rs	rt	rd	00000	100001
SUBU	000000	rs	rt	rd	00000	100010
SLT	000000	rs	rt	rd	00000	101010
SLTU	000000	rs	rt	rd	00000	101011
AND	000000	rs	rt	rd	00000	100100
OR	000000	rs	rt	rd	00000	100101
XOR	000000	rs	rt	rd	00000	100110
NOR	000000	rs	rt	rd	00000	100111
SLLV	000000	rs	rt	rd	00000	000100
SRLV	000000	rs	rt	rd	00000	000110
SRAV	000000	rs	rt	rd	00000	000111
ADDIU	001001	rs	rt	immediate		
LW	100011	base	rt	offset		
SW	101011	base	rt	offset		
BEQ	000100	rs	rt	offset		
BNE	000101	rs	rt	offset		
BLEZ	000110	rs	00000	offset		
BGTZ	000111	rs	00000	offset		

5.1 数据通路设计

基于指令系统的定义，先设计这个简单CPU的数据通路，其主要模块包括一个指令存储器，一个数据存储器，一个通用寄存器堆和一个程序计数器（PC），如图 5.1 所示。

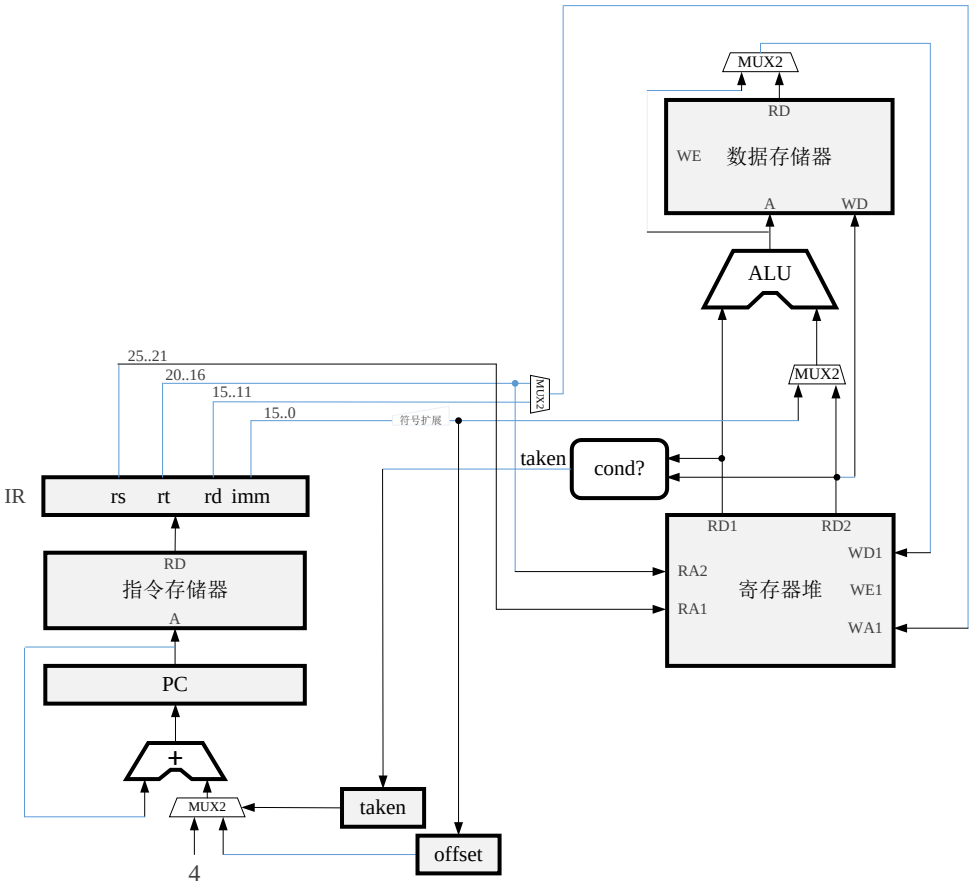


图 5.1 主要数据通路

CPU 工作时，首先用 PC 作为地址去指令存储器中取指令。PC 的值是怎么来的呢？有两种情况，第一种是执行完一条指令顺序执行时，下一条指令的 PC（Next PC，简称 NPC）的值是 PC+4，因为指令占四个字节；第二种是执行转移指令时 NPC 值是延迟槽 PC+offset。因为延迟槽指令总是需要执行的，所以当前指令是跳转的转移指令时并不能立即修改 PC 为跳转目标，只能是延迟槽指令在 CPU 里时才能修改。这样，生成 NPC 的部分有一个 2 选 1 逻辑根据转移指令跳转是否成功来选择 offset 值和 4，选择之后再由一个加法器跟 PC 的值相加，并送到 PC 中。然后，根据这个 PC 的值到指令存储器取指，指令取出来以后放到指令寄存器 IR 中。IR 中的指令包含操作码（op）和功能码（func），目标寄存器号（rd），两个源寄存器号（rs，rt），还有立即数/偏移量（imm），其中立即数/偏移量有 16 位，与 rd 和 func 域有部分重叠。

通用寄存器堆、运算部件和存储器的通路由 IR 中的域统一控制。通用寄存器的内部电路结构如图 5.2 所示，其读地址 RA1 通过控制一个 32 选 1 从 32 组寄存器中选出一组将其值输出至 RD1，同样的 RA2 控制另一个 32 选 1 从 32 组寄存

器中选出另一组将其值输出至 RD2；当发生写时，写地址 WA1 通过译码器得到各组的选择信号再与上全局写使能 WE1 形成每一组寄存器的写使能，用来控制将写入数据 WD1 写入到相应的寄存器组中。IR 的 rs 域连接到通用寄存器堆的读端口 1 的地址输入，从中选出一个将其值送到 ALU 的其中一端；IR 的 rt 域连接到通用寄存器堆的读端口 2 的地址输入，从中也选出一个值来，并和符号扩展后的立即数/偏移量 2 选 1 后送到 ALU 的另外一端。这是因为 ADDIU、LW 和 SW 指令不用寄存器读出的值作为第 2 个源操作数进行运算，而是用指令中的立即数/偏移量进行运算。转移指令也用到立即数/偏移量，但仅在计算 NPC 时使用，这里我们使用独立的加法器进行 NPC 的计算。ALU 完成计算操作之后要把算术运算或逻辑运算的结果写回到通用寄存器堆里去，写回到哪个寄存器由指令中的 rd 或 rt 域来控制，目标连接到通用寄存器堆的写端口 1 的地址输入，进而选中一个寄存器并打开其写使能。对于 LW 指令来说，其目标寄存器号来自于指令的 rt 域而非其它指令的 rd 域，所以需要通过一个 2 选 1 逻辑选择出目标寄存器号。访存指令 LW 和 SW 把 ALU 的运算结果作为访存地址。LW 从数据存储器中把值取出然后写回到目标寄存器去，所以写入通用寄存器堆的数据也需要通过一个 2 选 1 逻辑从 ALU 运算结果和数据存储器读出结果之间选择。SW 将寄存器堆中读出的值写入到数据存储器中。

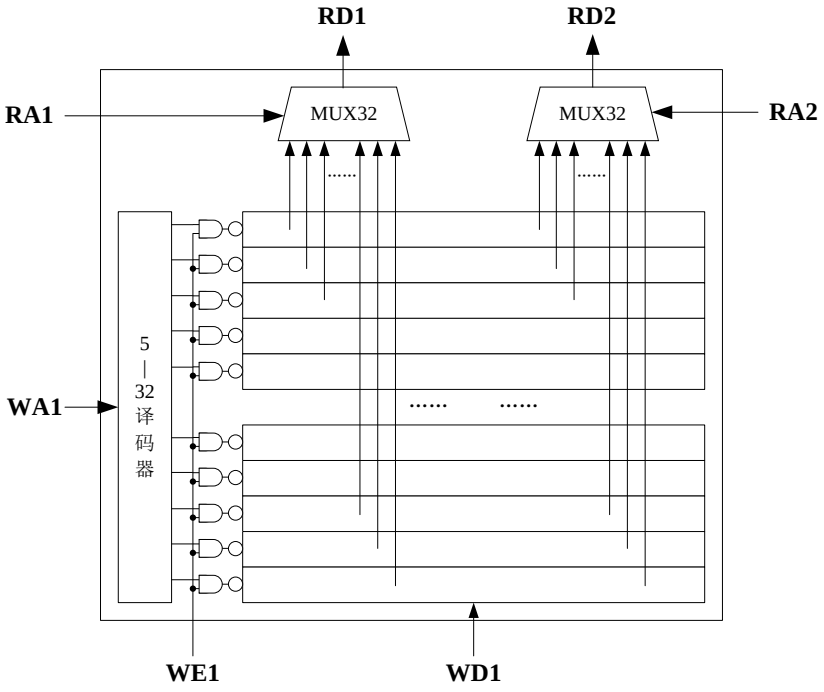


图 5.2 寄存器堆电路结构

上述描述实现了这个 CPU 中的主要数据通路，并涵盖了指令系统中定义的所有指令，但没有描述这个通路的控制逻辑部分。下面我们一步一步地往里加东西。

5.2 控制逻辑设计

实现了 CPU 的数据通路之后，下面先添加 CPU 的控制逻辑。控制逻辑根据指令的要求控制数据在数据通路中流动。

从上述数据通路可以看出，为了让数据根据指令的要求在数据通路中正确地流动，需要对以下通路进行控制：计算 PC 的加法器是否需要看转移跳转情况决定是加 4 还是加 offset（C1）；是选择寄存器的值还是选择立即数作为 ALU 的第 2 个源操作数（C2）；ALU 做什么运算(ALUOp)；运算结果是把 ALU 的运算结果写回，还是把从数据存储器读出来的结果写回（C3）；目的寄存器号是来自指令的 rd 域还是 rt 域（C4）；什么情况下使能通用寄存器堆的写使能（C5），因为有一些指令是不写寄存器的，如 SW 指令和转移指令；以及什么情况下使能数据存储的写使能（C6）。

根据指令的功能和数据通路的情况，表 5.2 给出了 CPU 中控制逻辑的真值表。其中 X 表示是 0 或 1 无所谓。

表 5.2 控制逻辑真值表

指令	op 域	func 域	C1	C2	C3	C4	C5	C6	ALUOp
ADDU	000000	100001	0	0	0	0	1	0	0000
SUBU	000000	100010	0	0	0	0	1	0	0001
SLT	000000	101010	0	0	0	0	1	0	0010
SLTU	000000	101011	0	0	0	0	1	0	0011
AND	000000	100100	0	0	0	0	1	0	0100
OR	000000	100101	0	0	0	0	1	0	0101
XOR	000000	100110	0	0	0	0	1	0	0110
NOR	000000	100111	0	0	0	0	1	0	0111
SLLV	000000	000100	0	0	0	0	1	0	1000
SRLV	000000	000110	0	0	0	0	1	0	1010
SRAV	000000	000111	0	0	0	0	1	0	1011
ADDIU	001001	xxxxxx	0	1	1	0	1	0	0000
LW	100011	xxxxxx	0	1	1	1	1	0	0000
SW	101011	xxxxxx	0	1	0[1]	X	0	1	0000

[1] 单就区分目的寄存器时来自 rd 域还是 rt 域，SW、BEQ、BNE、BLEZ 和 BGTZ 指令的 C4 控制信号可以是 0 或 1，但此处设为 0 是为了 5.5 小节中生成流水线阻塞控制逻辑的简洁。

BEQ	000100	xxxxxx	1	X	0^1	X	0	0	xxxx
BNE	000101	xxxxxx	1	X	0^1	X	0	0	xxxx
BLEZ	000110	xxxxxx	1	X	0^1	X	0	0	xxxx
BGTZ	000111	xxxxxx	1	X	0^1	X	0	0	xxxx

根据上述真值表，可以给出相应控制信号的逻辑表达式：

```
wire [5:0] op    = inst[31:26];
wire [5:0] func  = inst[5:0];
wire inst_addu   = op==6'b0 && func==6'b100001;
wire inst_subu   = op==6'b0 && func==6'b100010;
wire inst_slt    = op==6'b0 && func==6'b101010;
wire inst_sltu   = op==6'b0 && func==6'b101011;
wire inst_and    = op==6'b0 && func==6'b100100;
wire inst_or     = op==6'b0 && func==6'b100101;
wire inst_xor    = op==6'b0 && func==6'b100110;
wire inst_nor    = op==6'b0 && func==6'b100111;
wire inst_sllv   = op==6'b0 && func==6'b000100;
wire inst_srlv   = op==6'b0 && func==6'b000110;
wire inst_srav   = op==6'b0 && func==6'b000111;
wire inst_addiu  = op==6'b001001;
wire inst_lw     = op==6'b100011;
wire inst_sw     = op==6'b101011;
wire inst_beq    = op==6'b000100;
wire inst_bne    = op==6'b000101;
wire inst_blez   = op==6'b000110;
wire inst_bgtz   = op==6'b000111;
wire c1          = inst_beq  | inst_bne
                 | inst_blez | inst_bgtz;
wire c2          = inst_addiu | inst_lw | inst_sw;
wire c3          = inst_addiu | inst_lw;
wire c4          = inst_lw;
wire c5          = ~(inst_sw | c1);
wire c6          = inst_sw;
wire [3:0] aluop;
assign aluop[0]  = inst_subu | inst_sltu | inst_or
                 | inst_nor | inst_srav;
assign aluop[1]  = inst_slt | inst_sltu | inst_xor
                 | inst_nor | inst_srlv | inst_srav;
assign aluop[2]  = inst_and | inst_or | inst_xor | inst_nor;
assign aluop[3]  = inst_sllv | inst_srlv | inst_srav;
```

上述控制逻辑加在数据通路图的基础上，如图 5.3 所示。图中虚线部分是新加上去的控制逻辑。由 6 位操作码 op 和 6 位功能码 func 形成控制信号，C1 与转移条件判断的结果 cond 相“与”后控制 NPC 加法器源 2 的 2 选 1，C2 控制 ALU 第二个操作数的 2 选 1， ALUOp 控制运算部件 ALU，C3 控制结果写回的 2 选 1，C4 控制目的寄存器的 2 选 1，C5 控制通用寄存器的写使能，C6 控制数据存储的

写使能。这些控制都是组合逻辑，这个过程称为译码操作，即把操作码翻译成 CPU 主要数据通路所需要的控制信号。

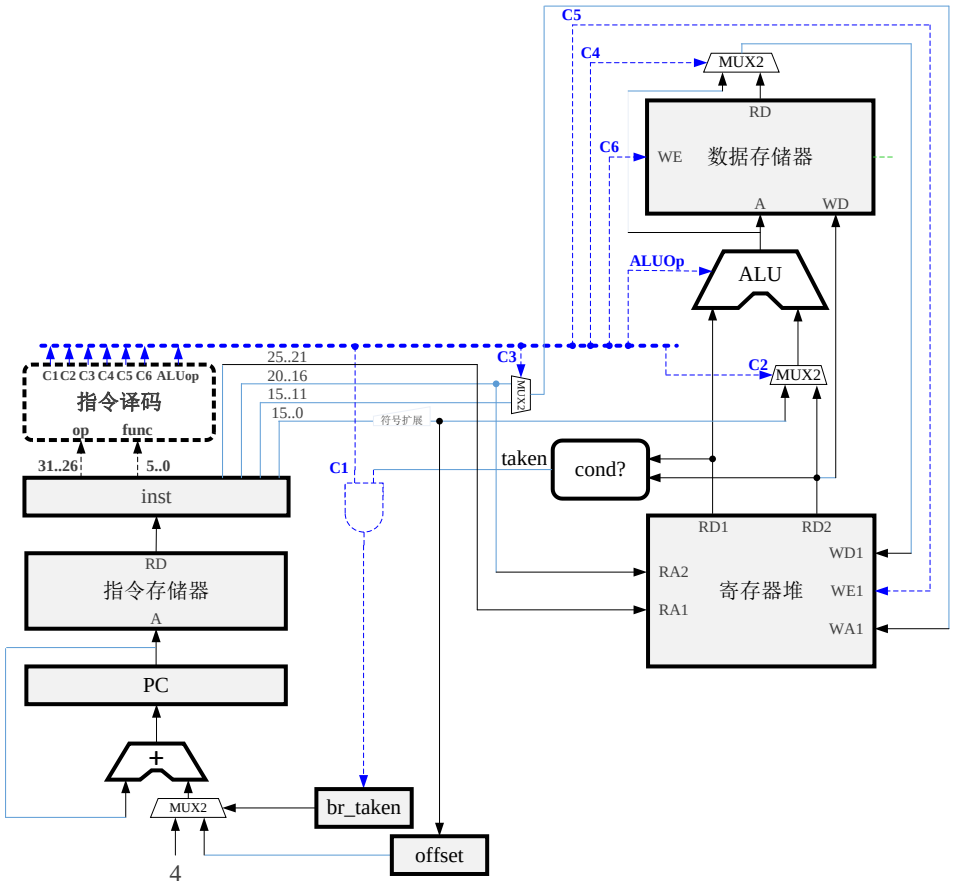


图 5.3 主要控制逻辑

5.3 时序

到目前为止，基本实现了 CPU 的主要数据通路和控制通路，但是这个 CPU 还是不能工作，还差什么呢？还差时钟部分。时钟就是 CPU 的驱动，就像人的心跳。第 3 章介绍过，时序电路是需要由时钟驱动的。

可以简单直观地把一条指令的执行分成三步：第一步计算 PC 值；第二步根据这个 PC 值把指令从指令存储器里面取出来放到指令寄存器里；第三步根据指令寄存器的内容来控制这个指令的执行并把结果写到寄存器或数据存储器。上述三步工作可以分别由三个时钟来驱动，这三个时钟可以通过对一个统一的时钟 C 进行分频得到。第一个时钟把下一条指令的 PC 值送到 PC 里面去；等到根据 PC 寄存器的值完成取指后，第二个时钟把取出的指令送到 IR 里面去；等到指令执行完后，第三个时钟把指令的运算结果送入到通用寄存器或存储器。一条指令执

行完以后，再执行下一条，CPU 就能循环往复地工作了，如图 5.4 所示。

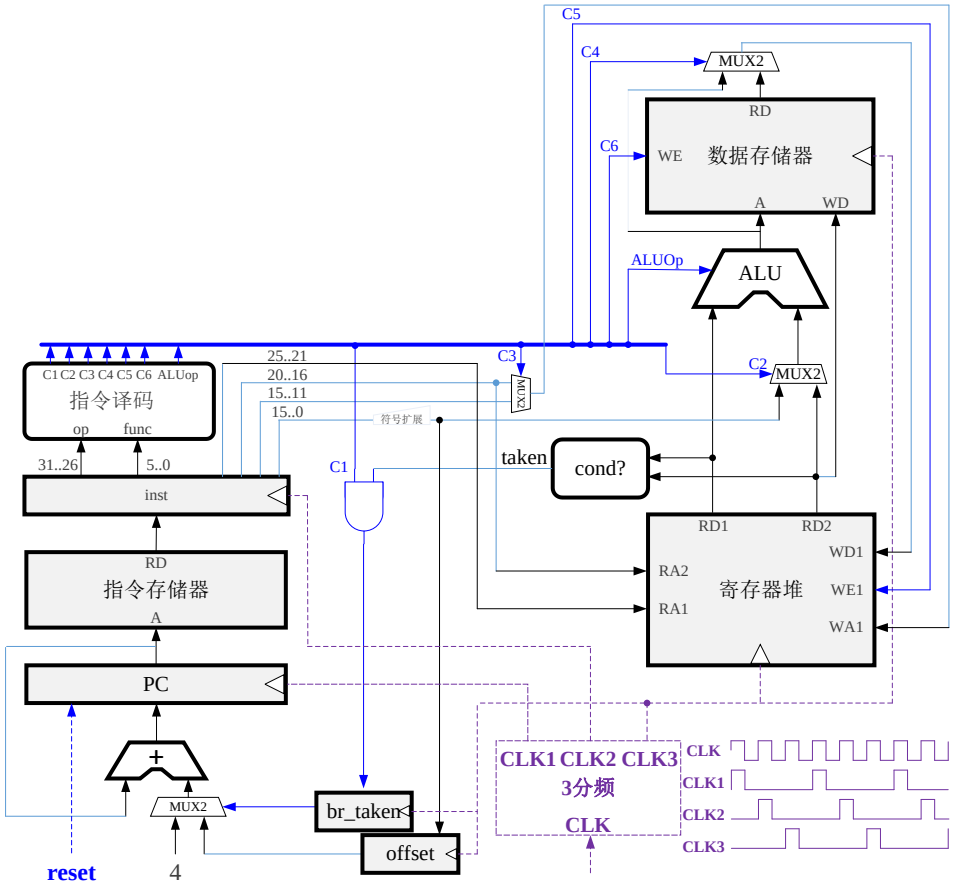


图 5.4 带时序的控制逻辑

图 5.5 描述了上述指令执行的三个阶段。从图中可以看出，上述指令执行的三个阶段是顺序进行的。

图 5.5 指令执行的三个阶段

5.4 流水线技术

现在这个 CPU 一拍一拍、一条一条地执行指令。就像在一个超市里面，第一个人进去了，第二个人就不许进，等他买完了东西，从出口出来了以后，才放第二个人进去。当然，从功能上考虑这是不会出错的，但是它不够高效，所以需要改进。

当前的设计中的 3 个步骤（计算 PC、取指、执行）可以合并成两个：计算下一条指令的 PC 和指令执行可以重叠。这样重叠不会出错，因为计算 PC 跟

当前指令执行没有直接关联；PC+4 或根据转移指令 PC+offset，也可以理解为是对当前指令的执行。因此，可以在指令执行的同时计算下一个 PC。这样就可以把 3 步变成 2 步来完成指令的执行，其中第 2 步把计算 PC 和指令执行重叠。相应地，把原来 3 个时钟变成 2 个时钟就可以了，如图 5.6 所示。

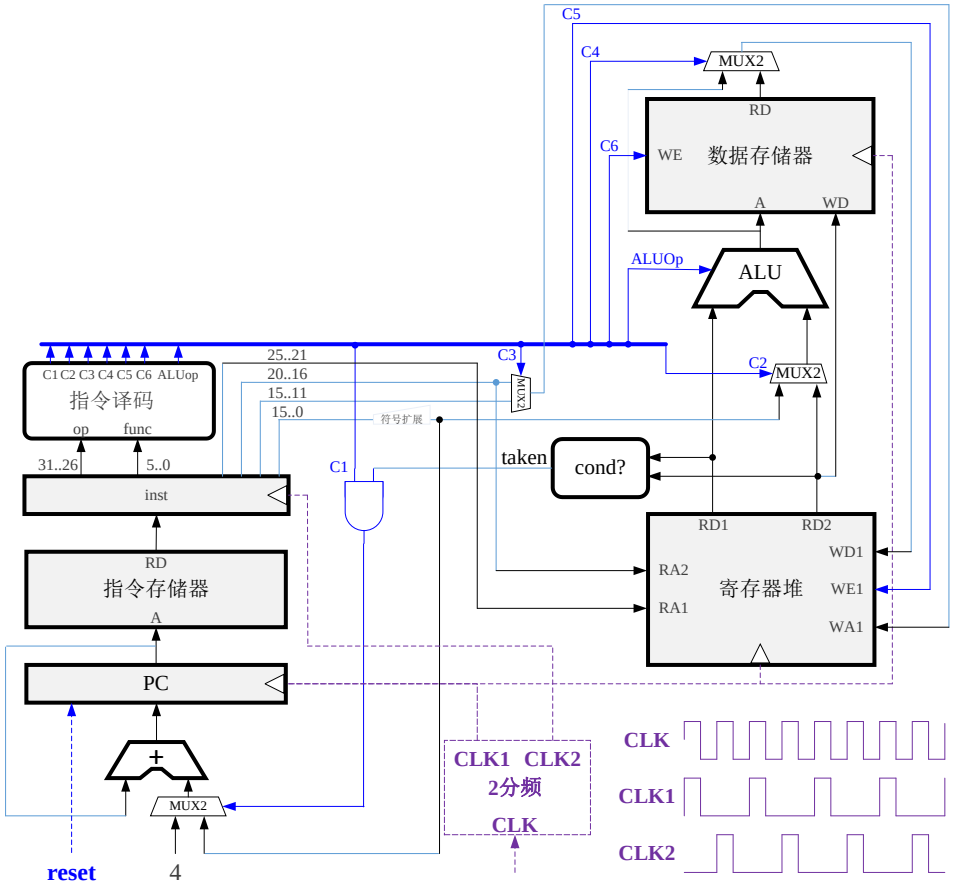


图 5.6 改进后的时序（计算 PC 与执行重叠）

图 5.7(a)描述了把上述流水线中第 N 条指令的执行和第 N+1 条指令计算 PC 重叠后的流水线阶段，图 5.7 (b)描述了把执行和计算 PC 合并的流水线阶段。

(a)

(b)

图 5.7 计算 PC 跟执行重叠的流水阶段

在此基础上，还可以进一步对该 CPU 结构进行改进，看能不能把取指和运算也重叠。在大多数的情况下，取指和运算是可以重叠的。例如当第 N+1 条指令执行的时候，第 N 条的指令已经执行完，它已经把所运算的结果写回到寄存器了，第 N+1 条指令可以使用第 N 条指令的结果。但是有一种特殊情况，如果第 N+1 条指令的取指（即 PC 的计算）也要用到第 N 条指令的结果，那么第 N+1 条指令的取指必须等到第 N 条指令执行结束了才能执行。什么情况下第 N+1 条指令的取指会用到第 N 条指令的结果呢？如果第 N 条指令是转移指令就会出现这种情况。在这种情况下，第 N+1 条在取指的时候，需要知道转移指令成功不成功，往哪里跳转，这个时候就不能把第 N+1 条指令的取指和第 N 条指令的运算叠在一起了。这时就能看出定义延迟槽指令的作用了。所谓延迟槽指令，是指紧挨着转移指令后面的那条指令，不论转移是否成功都执行，这样第 N+1 条指令的取指就不会用到第 N 条指令的结果了。经过上述改进，整个 CPU 的所有部分只要一个时钟就行了，如图 5.8 所示。

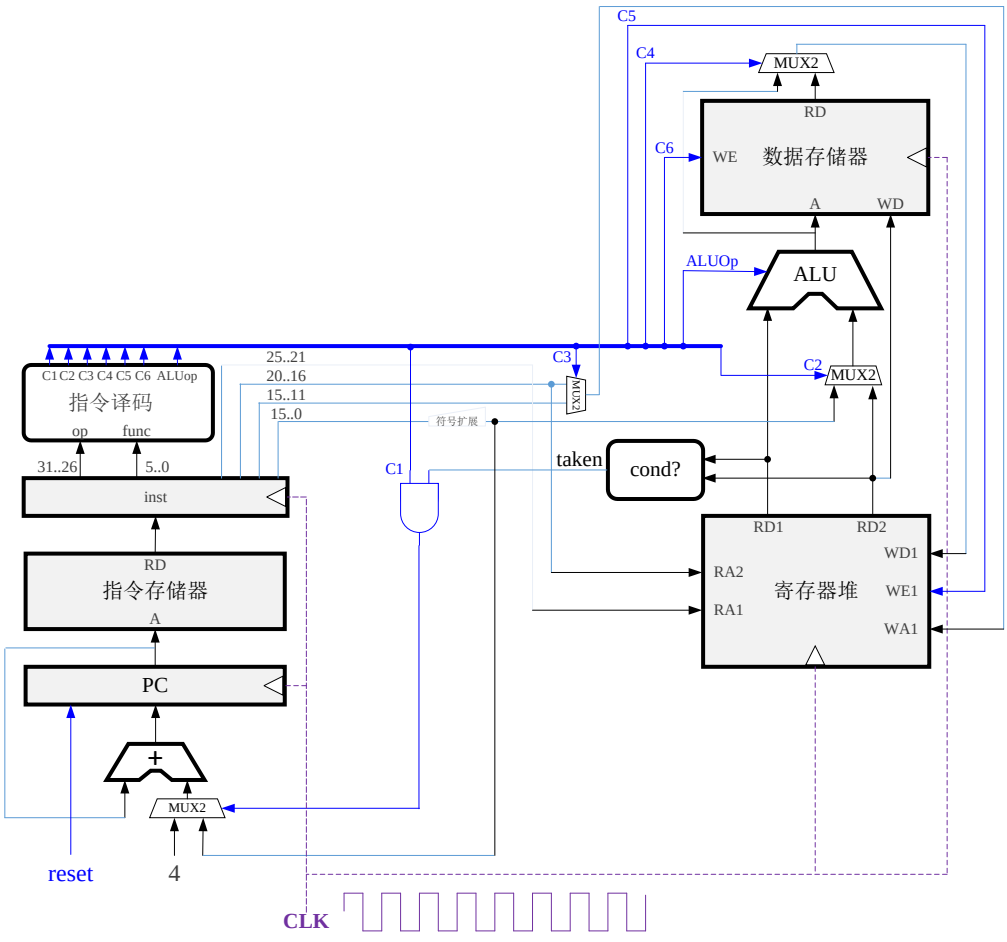
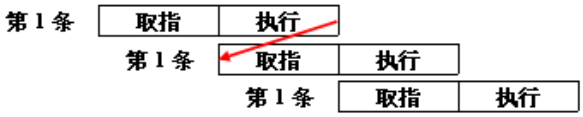


图 5.8 再次改进后的时序

图 5.9 描述了把上述流水线中第 N 条指令的执行和第 N+1 条指令的取指重叠后的流水线阶段。

(a)



(b)

图 5.9 取指与执行重叠的流水阶段

然而，上述流水线的设计有一个问题，就是在指令执行阶段需要干的事情太多了。根据上述流水级的划分，指令进入指令寄存器 IR 后，要进行译码、读寄存器的值、送到运算器进行运算、有时还要把运算结果当作地址进行访存、最后把结果写回到寄存器。我们在前面学习二进制和逻辑电路的时候，知道任何逻辑运算都是有延迟的，所以指令执行阶段的延迟就非常大。这样设计出来的 CPU 虽然能工作，但主频不高，上一个时钟脉冲把指令送到指令寄存器后要等比较长的时间，才能发出下一个时钟脉冲把结果写到寄存器中。

可以通过进一步细分执行阶段来减少每一拍的工作量，比如把原来执行阶段的工作切分为译码、运算、访存、写回四个阶段。这时指令的执行先做译码，然后读寄存器把值读出来并送到 ALU，运算完成后如果是一个访存操作就根据算出来的地址访问数据存储器，最后把运算或访存的结果写回到寄存器堆。流水线的阶段划分如图 5.10 所示。上述流水级的划分在 CPU 执行阶段增加了几组寄存器，用于存放每个阶段的中间结果，即每一阶段结束后通过时钟脉冲把其结果保存在中间寄存器中。上述设计就是传统 RISC CPU 设计中的标准五级流水线。在五级流水线 CPU 中，同时有 5 条指令在执行，每条指令所处的流水线阶段不一样，所以控制也不一样。不仅要把每个流水阶段的数据存在中间寄存器，还要把每个流水线阶段及其后续阶段要用到的控制信号（ALUOp，C3,C5,C6）和目标寄存器号（dest）存起来跟着指令走。这样就可以在流水线中同时执行 5 条不同阶段的指令。

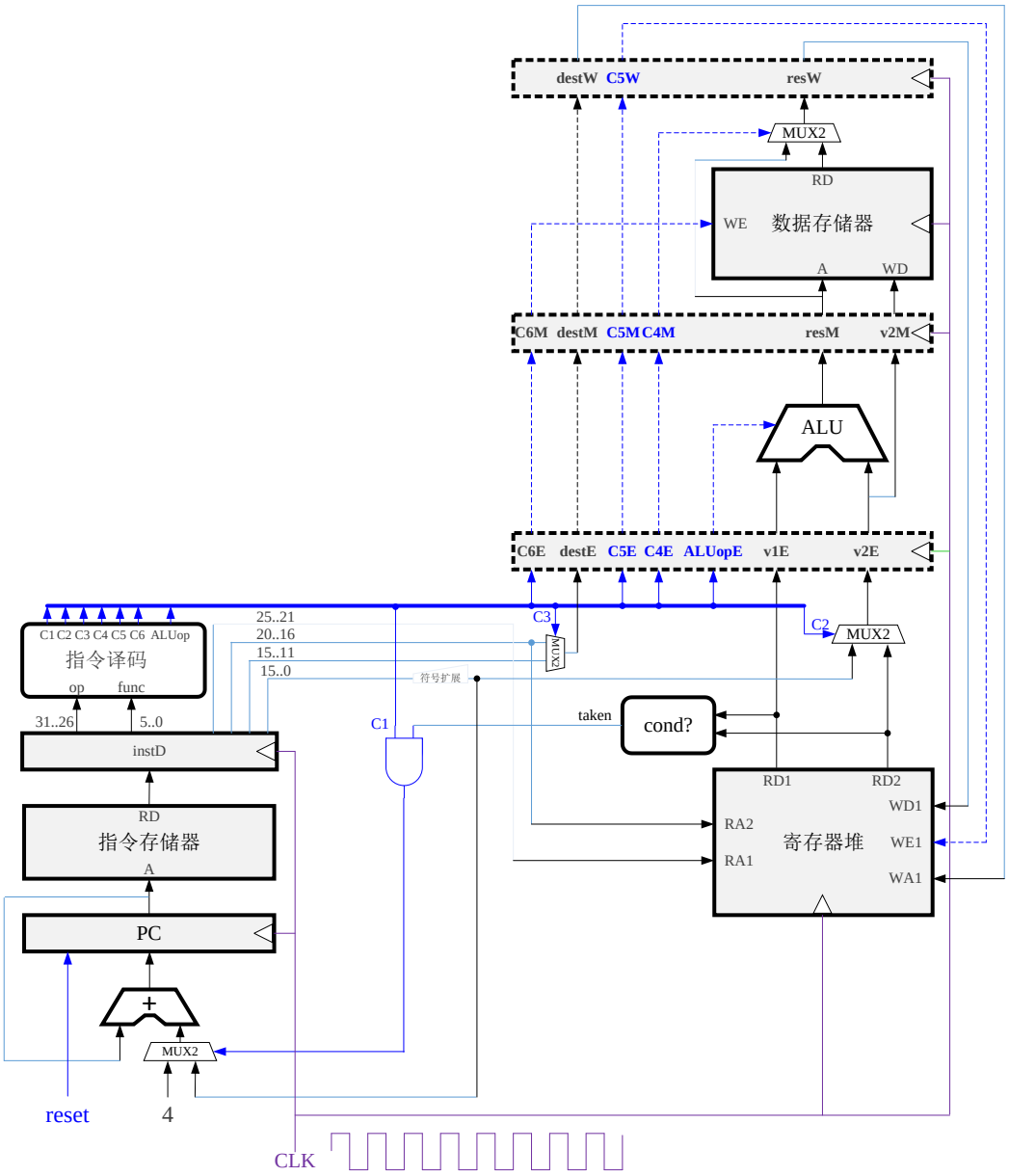


图 5.10 标准 5 级静态流水线

还把这个 CPU 比作一个超市的话，现在这个超市可以同时有五个人在不同的地方选择商品。以后会介绍动态流水线和多发射，这些现代的 CPU“超市”中，可以同时有几十甚至几百个人在选择不同的商品。当然，CPU 比超市复杂得多，因为超市里的人互相之间是没有太多关系的，只要他们不争夺同一件商品就相安无事。但 CPU 的指令之间存在各种关系，例如一条指令需要用到前一条指令的运算结果或者一条指令是否执行需要等前面的转移指令结束后才能确定。

5.5 指令相关和流水线冲突

前面设计的流水线结构把一条指令的执行分成 5 个阶段，或者称为 5 级流水，

分别是取指（IF）、译码（ID）、执行（EX）、访存（MEM）和写回（WB）。在同一个周期内，可以有五条指令分别处在不同的流水级。这个流水线的划分可以提高主频，但是由于指令间的相关可能会导致执行结果的错误。比如说，第 N 条指令把结果写到 R1 寄存器，第 N+1 条指令要用到 R1 的值进行运算。在上述五级流水线中，第 N 条指令在第 5 级才把结果写回到寄存器，而第 N+1 条指令在第 2 级译码阶段就要读寄存器的值，导致第 N 条指令还没有把结果写回到寄存器 R1 时，第 N+1 条指令就把老的值读出来用了，造成了运算结果的错误。这就是由于指令相关导致执行结果错误。

指令的相关（dependency）可以分成 3 类：数据相关、控制相关及结构相关。在程序中，如果两条指令访问同一个寄存器或内存单元，而且这两条指令中至少有一条是写该寄存器或内存单元的指令，则这两条指令之间存在数据相关；如果两条指令中一条是转移指令且另外一条指令是否被执行取决于该转移指令的执行结果，则这两条指令之间存在控制相关；如果两条指令使用同一个功能部件（如两条都是加法指令），则这两条指令之间存在结构相关。

在程序中，指令间的相关是普遍存在的。指令间的相关给指令增加了一个序关系，要求指令的执行必须满足这些序关系，否则执行的结果就会出错。为了保证程序的正确执行，计算机体系结构设计必须满足这些序关系。指令之间的序关系有些是很容易满足的，例如两条相关的指令之间隔得足够开，后面的指令开始执行时前面的指令早就执行完了。但如果两条相关的指令挨得很近，尤其是都在指令流水线的不同阶段时，就需要结构设计来保证这两条指令执行时它们的相关关系得到满足。

相关的指令在一个具体的结构中执行时可能导致冲突（hazard）。在 5 级静态流水线中，第 N+1 条指令使用第 N 条指令的结果就是冲突的例子。当两条相关的指令之间隔得足够开时不会引起冲突，因为后面的指令开始执行时前面的指令早就执行完了，程序规定的序关系自然得到满足；如果相关的指令隔得不够开，结构设计就必须保证冲突的指令要按照程序规定的序关系执行。下面看看在 5 级静态流水线中存在的冲突及解决办法。

先看数据相关。数据相关可以根据冲突访问读和写的次序分为 3 种。第 1 种是写后读相关（Read After Write，RAW），就是后面指令要用到前面指令所写的数据，这是最常见的类型也称为真相关。第 2 种是写后写相关（Write After

Write，WAW)，也称为输出相关，即两条指令写同一个单元，在乱序执行的结构中如果后面的指令先写，前面的指令后写，就会产生错误的结果。第 3 种是读后写相关（Write After Read，WAR)，在乱序执行的结构或者读写指令流水级不一样时，如果后面的写指令执行得快，在前面的读指令读数之前就把目标单元原来的值覆盖掉了，导致读数指令读到了该单元“未来”的值，从而引起错误。在原型 CPU 的 5 级简单流水线中，只有 RAW 相关会引起流水线冲突，WAR 和 WAW 相关不会引起冲突。但在以后介绍的乱序执行流水线中，WAR 和 WAW 相关也会引起冲突。

图 5.11 给出了一个流水线的数据和控制冲突的例子。在图 5.11 中，从第 1 条指令的 WB 流水级指向后续 3 条指令的 ID 流水级的箭头表示数据相关会引起的冲突，即如果第 2、3、4 条指令需要使用第 1 条指令写回到寄存器的结果，那么第 2、3、4 条指令读取寄存器时就必须保证第 1 条指令的结果已经写回到寄存器。而按照目前五级流水线的结构，如果不加控制，第 2、3、4 条指令就会先于第一条指令写回之前读取寄存器从而引起数据错误。为了保证正确，第 2 条指令要在译码阶段等待 3 拍，然后才能读取寄存器的值，后面的第 3、4、5 条指令也相应等待（由于流水线是有序的，第 5 条指令虽然与第一条指令没有数据相关也要等待)，引起流水线的阻塞（stall)。

图 5.11 流水线的数据和控制冲突

在 5 级静态流水线结构中如何实现上述阻塞过程呢？流水线的阻塞要求在指令译码阶段读取寄存器时，如果发现该寄存器是流水线先前指令的目标寄存器并且还没有写回，那么该指令就要在译码阶段等待。具体实现可以将 ID 流水级指令的两个源寄存器号 rs、rt(ADDIU 和 LW 指令中的 rt 不是源寄存器，不参与比较)分别跟 EX、MEM、WB 流水级指令的目标寄存器号 dest 进行相等比较，如果有一个相等且该寄存器不是 0 号寄存器（0 号寄存器的值恒为 0)，这条指令就不能前进。为了阻塞流水线，需要对程序计数器 PC 和指令寄存器 IR 的输入使能

（enable）进行控制，如果相关判断逻辑的结果为 1，就控制 PC 和 IR 的输入使能，使 PC 和 IR 保持当前的值不变。而 EX 阶段的流水线则输入指令无效信号，用流水线空泡（bubble）填充。图 5.12 示意了上述流水线数据相关引起阻塞的控制逻辑。

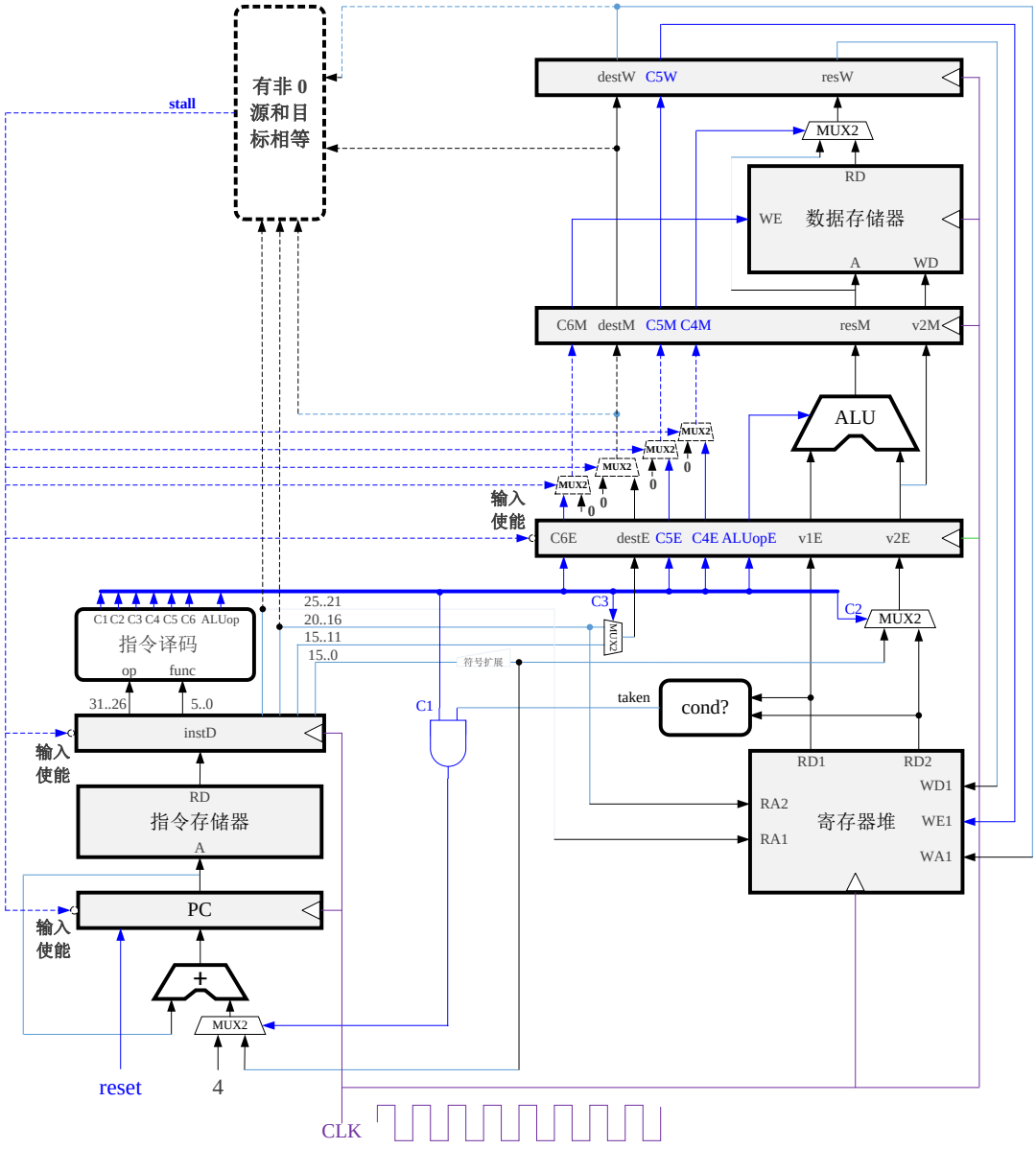


图 5.12 数据相关引起流水线阻塞的控制逻辑示意图

再看控制相关。控制相关引起的冲突本质上是对程序计数器 PC 的冲突访问引起的。在图 5.11 中，从第 1 条指令的 ID 流水级到第 2 条指令的 IF 流水级的箭头表示控制相关而引起的冲突，即如果第 1 条指令是转移指令，则第 2 条指令的取指需要等 1 拍，等待第一条指令的译码阶段结束才能开始，因为每条指令取指时需要用到 PC 的值而转移指令会修改 PC。这也是为什么要通过专用的运算部

件在译码阶段就计算下一条指令的 PC。否则的话，如果转移指令也在 EX 阶段用运算部件计算 PC 值然后在 WB 阶段写回，那后面的指令就不是等 1 拍而是等 4 拍。在译码阶段执行转移指令时，先进行条件判断，然后根据条件判断结果选择 offset 或 4 跟 PC 相加得到下一条指令的 PC 值。此时取指阶段的指令是延迟槽指令，它的执行不依赖于转移指令的结果，不需要被取消。总之，在 5 级静态流水线中，通过在译码阶段对转移指令进行判断以及设置转移指令的延迟槽指令可以避免控制相关引起的流水线阻塞。

当然，如果转移指令在译码阶段根据寄存器的值进行条件判断时，该寄存器是 EX、MEM 和 WB 阶段的目标寄存器时，即转移指令与先前的指令存在数据相关，转移指令也会由于数据相关堵在 ID 流水级，延迟槽指令则堵在 IF 流水级。

解决上述转移指令引起的控制相关的方法是非常简单的，只适用于五级简单静态流水线。而在现代处理器的超流水、多发射、乱序执行流水线中，则需要复杂的转移猜测技术，这些以后会专门介绍。

最后看看结构相关，结构相关引起冲突的原因是两条指令要同时访问流水线中的同一个功能部件。在 CPU 流水线中，指令的重叠执行需要功能部件能流水，并且要求存在多份资源以满足所有流水线中指令所处阶段的可能组合。例如，我们的原型 CPU 中，取指令和访存有单独的存储器，如果共用一个的话，取指和取数都访存时就会冲突，访存的时候下一条指令就不能取指了；在有除法指令的 CPU 中，由于除法功能部件不是全流水的实现，前面一条除法指令在运算时后面的除法指令需要等待；在多发射结构中，也可能存在两条运算指令需要使用同一个功能部件所引起的等待；等等。在原型 CPU 的 5 级流水中，尽管普通运算指令不用访存，但也需要经历 MEM 流水级，其本质原因是如果有些指令 5 拍写回，而有些指令 4 拍写回，就会引起通用寄存器堆写端口的结构冲突。

总之，程序中的指令存在相关，相关会在流水线中引起冲突，冲突会引起流水线阻塞，阻塞会降低流水线效率。

5.6 流水线的前递（Forwarding）技术

前面讲的 5 级静态流水线的控制通过指令的阻塞保证在指令流水线中的指令按照程序规定的次序执行，但是阻塞必然引起流水线效率的降低，可以通过软件和硬件的方法提高上述流水线的效率。

先看通过硬件优化提高流水线效率的方法。在上节的 5 级静态流水线中，如果流水线中的前后指令有数据相关，后面的指令要等到前面的指令把执行结果写回到寄存器后再从寄存器中读取。有没有可能让前面的指令直接把运算结果传给后面的指令从而减少后面指令的等待呢？答案是可以的。打个比方，有一个甲同学找我借了一本书，看完后要还给我，乙同学也要找我借，我在北京，甲乙都在上海。如果甲把书送回来，让乙再到北京来取，大家肯定觉得这是三个傻瓜，打个电话过去，甲直接给乙不就行了吗？这就是流水线中的前递（Forwarding）技术，也称为旁路（Bypass）技术。

前递技术的具体实现是在流水线的运算器前通过多路选择直接把前面指令的运算输出作为后面指令的输入。图 5.13 给出了在原来流水线的基础上添加了部分前递通路的情况，只考虑数据前递给 ALU，不考虑前递给存数指令和转移指令以及寄存器堆同时读写时的旁路,ALU 每一输入端都添加了一个 3 选 1 逻辑，3 个输入分别是原来的 ALU 输入，下一级流水线输出的结果（即 EX 流水级 ALU 的运算结果）以及再下一级流水线输出的结果（即 MEM 流水级的结果）。这样，如果后面指令要用到前面指令的运算或访存结果，就可以直接通过运算器前面的多路选择器选择前面指令的运算或访存结果，不用等到前面指令把结果写回到寄存器后再从寄存器读取。

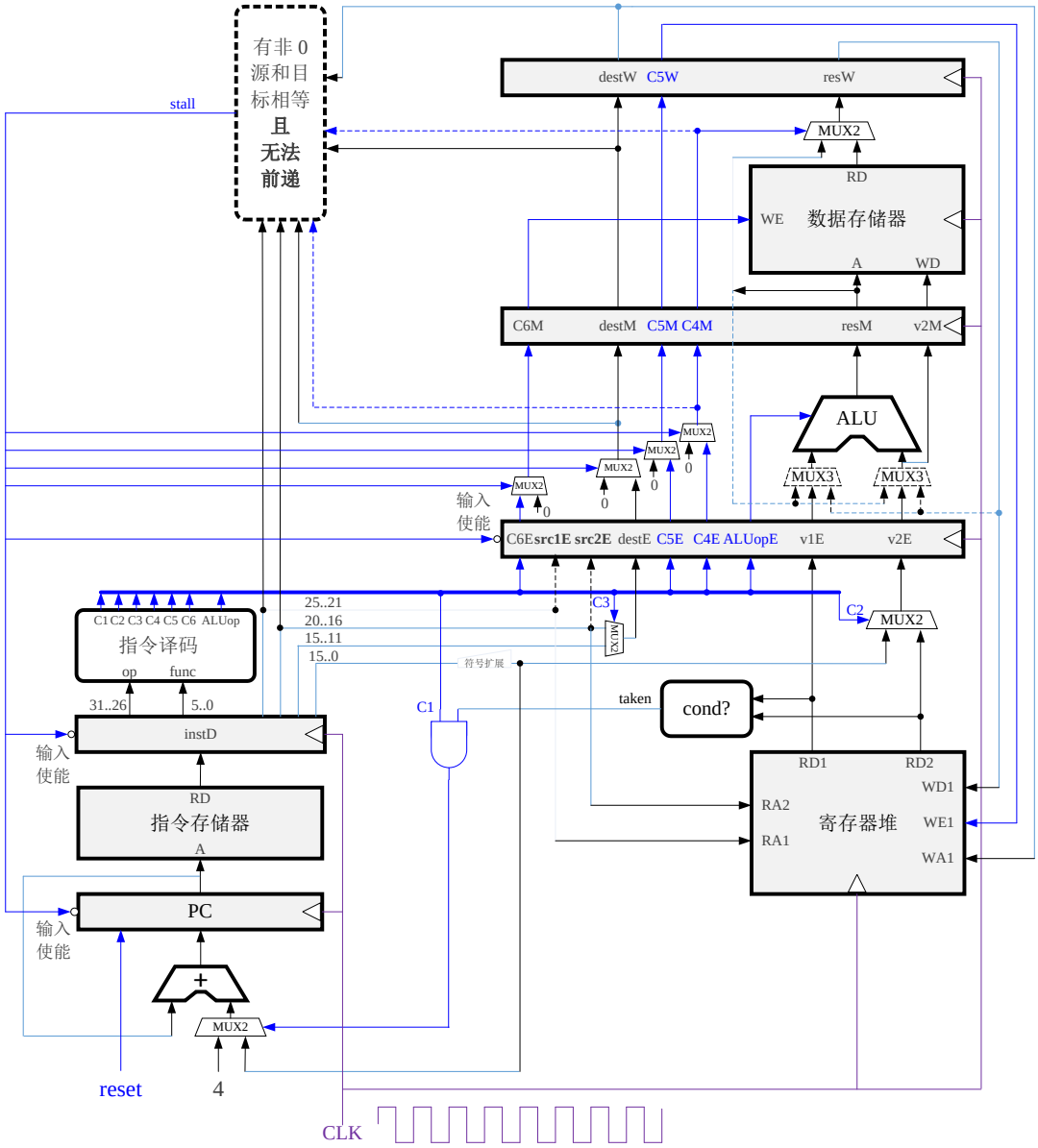


图 5.13 带有前递技术的流水线

下面看看前递通路的选择，即运算器前面的 3 选 1 该怎么控制。为了进行前递，需要比较处于 EX 流水级的指令的源寄存器号跟处于 MEM 或 WB 流水级的指令的目标寄存器号是否相等。如果相等且不是 0 号寄存器，则说明处于 EX 流水级的指令与前面的指令有数据相关，需要直接读取前面指令的结果用于运算器的输入。在原来的流水线中，因为不用前递，指令的源寄存器号是不用随流水线传递的，而采用了前递技术后就要把指令的两个源寄存器号传递到 EX 流水级，分别为 Esrc1 和 Esrc2。以 ALU 左端的输入为例，当 Esrc1 和前面两级目标寄存器 Mdst、Wdst 都不相等的时候，选择中间通路，即选择正常的 ALU 输入通路；当 Esrc1 等于 Wdst 时，选择右边通路；当 Esrc1 等于 Mdst 且在 MEM 流水级上不是

load 操作时，选择左边通路；如果 Esrc1 等于 Mdst 且在 MEM 流水级上是个 load 操作时，目标寄存器 Mdst 的值还没有形成，流水线等一拍，后面的流水线暂停，往前面的流水线送空操作。这样，通过前递技术，CPU 流水线的效率又提高了很多。当然，这是有硬件开销的，比如增加了 3 选 1 逻辑，而且这一级流水线的延迟也会增加一点。不过，实践表明这种硬件复杂度的增加相对于性能的提高还是值得的。

软件调度的方法也可以避免由于指令相关引起的冲突。下面通过一个例子来分析指令在 5 级静态流水线中的执行情况，并且介绍编译器的静态指令调度技术如何隔开相关的指令使之避免流水线冲突。例如，要做“A=B+C，D=E-F”的运算，MIPS 汇编指令的实现如图 5.14 所示。在具有前递功能的 5 级流水线中，由于存在 RAW 相关，LW 和 ADD 指令之间要空 1 拍，同样 LW 和 SUB 之间也要空 1 拍，所以执行这 8 条指令需要 10 拍。如果把这些指令适当调度一下，在不影响程序正确性的前提下进行重排序，指令执行的效率就会提高。如图所示，把一条 LW 跟一条 SW 对调一下，使 ADD 和 SUB 相关的 LW 都隔开了一拍，流水线减少了 2 拍堵塞。在相同流水线结构的情况下，软件调度技术对于提高指令序列执行的效率是非常重要的。

LW Rb, B	LW Rb, B
LW Rc, C	LW Rc, C
ADD Ra, Rb, Rc	**LW Re, E**
SW Ra, A	ADD Ra, Rb, Rc
LW Re, E	LW Rf, F
LW Rf, F	**SW Ra, A**
SUB Rd, Re, Rf	SUB Rd, Re, Rf
SW Rd, D	SW Rd, D
（a）调度前	（b）调度后

图 5.14 静态指令调度技术

到目前为止，我们把简单的 CPU 越做越复杂，但是效率越做越高。首先通过细分流水线，可以提高 CPU 的主频；其次通过采用解决各种相关的技术，流水级的效率也提高了，所花费的一点代价是值得的。

5.7 流水线和例外

到目前为止，我们完成了半个 CPU 的设计。为什么说只完成半个 CPU 的设计呢？因为我们完成了数据通路、控制通路、流水线等逻辑，并且假设所有指令都在规规矩矩地工作，没有任何意外，而实际情况不是这样的。在 CPU 的设计中，还存在各种例外情况，而对付这些情况才是最难的。

当我们写这样一个小程序：“B=1；C=2；A=B+C”，A 等于几？一年级的小学生都知道 A 等于 3。从应用的角度看，A 是等于 3，但从实现的角度就很复杂。从体系结构的角度来看，流水线在工作过程中可能发生各种例外情况。例如，我本来安排好今天上午要在办公室完成一件工作，但中间可能得接 10 个电话，有些电话里的事情还得放下手头的工作去处理一下，而且我要完成的工作也可能由于计划不周而出现计划外的情况，这些都是例外。我当年做龙芯 1 号的时候，体会最深刻的就是，好不容易把流水线做得挺好了，觉得对付例外就是增加一个例外处理模块连到流水线中就行了。但后来发现，例外很复杂，跟每个模块都有关系，每个模块都得改。如果没有考虑例外，只是做了半个 CPU。在我学的所有教科书中例外都是单独用 1 章介绍的，但具体实现时，例外跟每个部分都有关系。

回到 A=B+C 这个例子。假设 A、B、C 都在内存里面了，我们看看处理器完成这样一个计算可能发生什么例外。首先要根据 PC 来取指，而在现代处理器中取指的地址是虚地址，需要把它转换为物理地址，这就需要查 TLB。假设 TLB 里面刚好没有那个表项，这样就会发生一个 TLB 不命中例外。好不容易把指令取上来了，在指令译码时刚好有人敲了一下键盘，碰到了外部中断，这也是一个例外事件。在程序执行的过程中，要把 B 和 C 从内存取到寄存器中去，即使虚实地址转换的表项在 TLB 中，还有可能发现存放 B 和 C 的内存页不在内存里，而是在硬盘交换区（swap）中，这样就会发生 TLB 失效例外。数据取进来后，读寄存器，然后做加法操作，当然 1+2 是不会有例外的，但是如果是两个比较大的数相加，可能出现溢出的情况，这又是一个例外。当加法操作执行完之后要把 A 写到内存单元去，发现该程序是一个用户程序，而 A 在核心态地址空间里，是只有操作系统才能用的空间，结果又发生一个例外。还有可能有其它例外，例如某部分的硬件发生故障等。

现代计算机一碰到例外就会暂停当前程序的执行，转移到一个事先规定的例外入口地址让操作系统来处理例外情况，处理完了之后再回来重新执行当前程序。例外又分好多种，有同步例外和异步例外，有用户请求和系统强制例外，有可屏

蔽和不可屏蔽例外，有指令内和指令间例外，还有可恢复和不可恢复例外。发生不可恢复例外（如硬件的严重故障和掉电）时，将终止程序的执行。

在 CPU 的设计中，可恢复例外的处理比较难，要求做得非常精准，当处理完例外之后，回来接着做 A=B+C，还要算得对，就好象没有发生过例外一样。操作系统存储管理和浮点 IEEE 运算规范都要求精确例外，即在操作系统开始处理例外时，硬件要保证例外指令前面的指令都执行完了，后面的指令一条都没动；在流水线中的多条指令同时发生例外的情况下，要保证有序的处理。另外，条件转移的延迟槽又增加了例外处理的难度。延迟槽指令发生例外时，例外处理完了回哪儿执行呢？如果还回到延迟槽条指令，就会从延迟槽之后的指令取指顺序往下执行，而实际上是要转移的，导致程序执行的错误。

在 5 级静态流水线中，为了实现精确例外，可以把指令执行过程中发生的例外先记录下来，到流水线中写回阶段的时候进行处理，这样就保证前面的指令都执行完了，而后面的指令都没有修改机器的状态，而且有两条或多条指令发生例外时，可以处理最前面那条指令的例外。图 5.15 给出了一个流水线中两条指令在不同阶段发生例外的例子，后面的指令先发生例外，但把例外统一到写回阶段处理后，还是前面指令的例外先被处理。外部中断是异步的，什么时候处理都可以，但可以在译码阶段对外部中断进行统一采样，然后随译码阶段的指令前进到写回时统一处理。

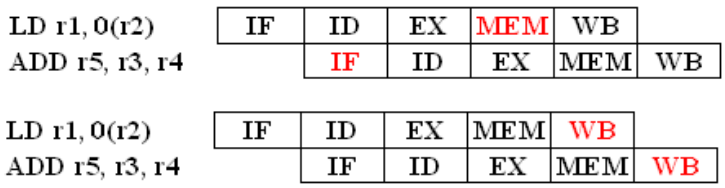


图 5.15 例外延迟到写回阶段统一处理

结合原型 CPU 的设计，例外信号（EX）及指令的 PC 要随着流水线前进到写回阶段。每个流水级中间寄存器都增加一个 EX 项和一个 PC 项，用来记录发生例外以及例外发生时指令的 PC。PC 给操作系统进行例外处理时使用，当发生例外的指令处在写回阶段时，CPU 要保存该指令的 PC 值到一个专用的寄存器 EPC 中，然后把 PC 置为例外处理程序的入口地址。需要在 PC 的输入端增加一个 2 选 1，一个是正常的 PC 值，一个是例外程序的入口，由例外选通信号来选择。

图 5.16 给出了原型 CPU 的例外处理通路。这是一个原理性的通路，是不完整的。例如它没有保存例外原因，也没有保存发生例外的指令是不是延迟槽指令的信息，也没有例外返回时把 EPC 寄存器值送给 PC 的通路等。当然，操作系统只要知道例外指令的 PC，就可以通过模拟指令的执行知道发生例外的原因，也可以通过分析指令的上下文知道该指令是不是延迟槽指令，还可以通过专用的指令修改 PC。

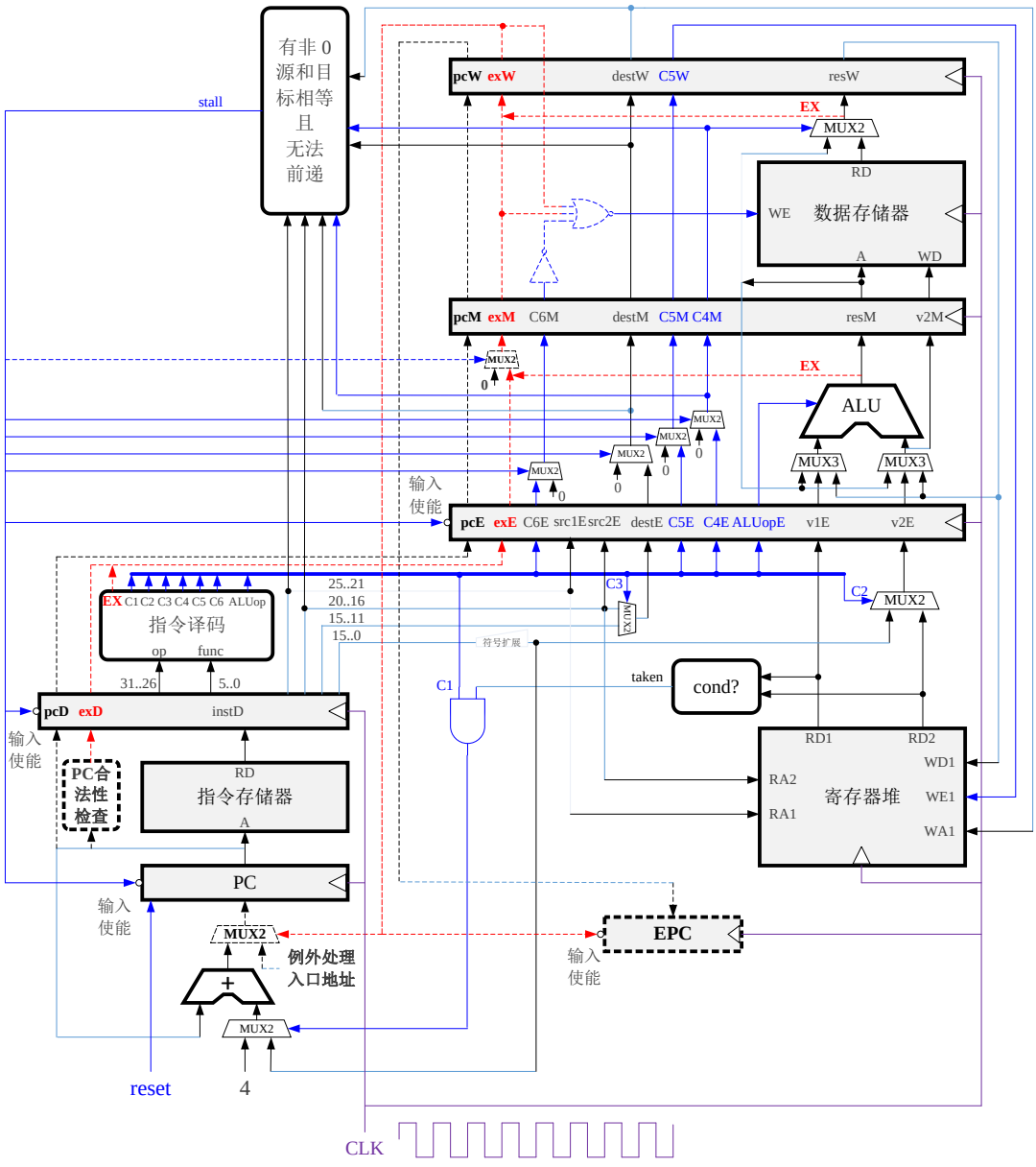


图 5.16 带有精确例外的流水线

5.8 多功能部件与多拍操作

前面我们介绍的简单 CPU 只有一个功能部件，下面我们分析在多个功能部

件和多拍操作情况下的静态流水线。

在 CPU 中通常存在着不同类型的指令，不同的指令常常由不同的功能部件执行。例如，加减、逻辑运算、转移一般都在定点 ALU 里执行，定点乘除法通常有专门的部件；浮点的加减、取绝对值、取非、定点与浮点的转换操作在浮点 ALU 中执行，浮点乘除法通常也有专门的部件；CPU 的访存指令还需要专门的访存部件。

不同的功能部件一般在指令流水线的执行阶段需要不同的执行拍数。例如定点 ALU 执行 1 拍就够了；定点乘法需要 2、3 拍；浮点 ALU 需要 4、5 拍；浮点乘法需要 5、6 拍；浮点除法和浮点开根号的拍数不确定，除 1 马上就算出来了，但如果两个数总是除不尽，算的拍数就很多；访存部件的延迟也是不确定的，Cache 命中和 Cache 不命中的时候不一样，Cache 有多个层次，Cache 不命中访问内存时还可能赶上内存刚好在刷新；等等。

在多功能部件和多拍操作的指令流水线中，结构相关经常发生。例如，如果访存部件不流水，则会引起多个访存操作的等待，一个 Load 操作访问 Cache 不命中时就要访问内存，这可能需要上百拍，后面的访存指令就得等。又如结果写回相关，不同的功能部件延迟不一致，在同一拍写回时会引起寄存器堆写端口冲突。图 5.17 是上述两种结构相关引起流水线阻塞的流水线时空图。

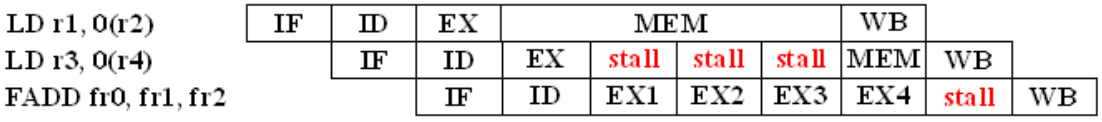


图 5.17 结构相关引起流水线阻塞

在多功能部件和多拍操作的情况下，会由于 WAW（Write After Write）相关引起冲突。例如前面一条取数指令和后面一条加法指令都要写 R1 寄存器。取数指令需要执行多拍才能写回，而且还可能由于 Cache 失效导致拍数不确定，加法指令执行一拍就可以写回。如果不加以控制，寄存器 R1 中最后的执行结果就会是取数指令而不是加法指令的结果。为了避免由于 WAW 相关引起错误，可以阻塞后面的加法指令，直到前面指令写回后再写回，如图 5.18 所示。

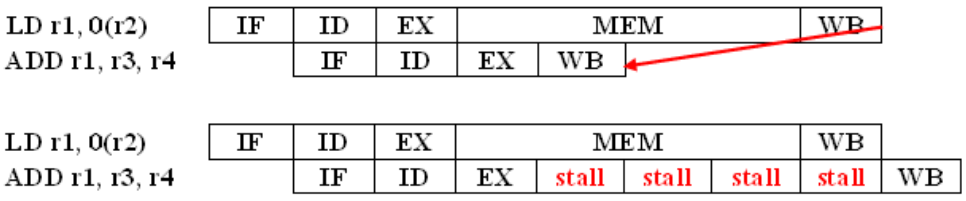


图 5.18 WAW 相关引起流水线阻塞

在多功能部件和多拍操作情况下，RAW（Read After Write）相关引起的冲突更加严重。在前面的简单流水线中用前递技术可以避免多数由 RAW 相关引起的冲突，但在多功能部件和多拍操作的情况下，前递技术的作用十分有限。例如，如果后面的加法指令需要使用前面的访存指令的结果，访存指令需要执行多拍而且不能确定拍数，加法指令就需要等多拍，前递技术只能少等 1、2 拍，如图 5.19 所示。

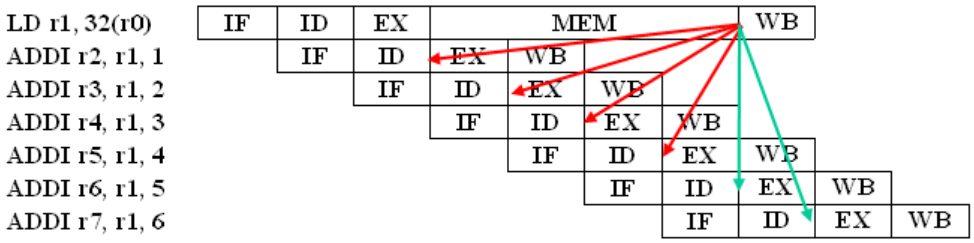


图 5.19 RAW 相关引起流水线阻塞更加严重

在静态调度的指令流水线中，即使在多功能部件和多拍操作情况下，WAR（Write After Read）相关也不会引起冲突。因为读操作数是在译码阶段读的，而在译码阶段指令是有序的，前面的指令没有完成译码，后面的指令就不能前进，因此，后面的指令写寄存器肯定在前面的指令读寄存器之后。但在动态调度的指令流水线中，后面的指令可以越过前面的指令读寄存器并执行，就会由指令的 WAR（Read After Write）相关引起冲突。

在多功能部件和多拍操作情况下，由于指令会乱序结束并写寄存器，例外处理更加复杂。例如，在除法指令后面跟着加法指令，除法指令在执行阶段需要几十拍，而加法指令执行一拍就够了。当后面的加法指令执行完并写回到寄存器以后，除法指令发生了例外。这就麻烦了，因为在加法指令写完寄存器后，除法指令的例外现场不对了。发生例外后，要由操作系统进行例外处理，操作系统处理完后再回来继续执行除法指令，但是这个时候加法指令已经被执行一遍了，再执行一遍就错了。

可见在多功能部件和多拍操作的情况下，流水线处理指令相关和例外都更加复杂。当然，所有问题都可以根据发生冲突和产生相关的具体情况，通过流水线堵塞来解决。例如，可以把在所有流水线执行阶段的指令的目标寄存器号和译码阶段的源寄存器号以及目标寄存器号进行比较，如果发现有寄存器号相等的情

况就阻塞译码阶段的指令，这样就可以避免由于数据相关引起冲突。注意，要把译码阶段指令的目标寄存器号也跟前面指令的目标寄存器号进行比较以避免WAW 相关导致冲突。同样，不管指令要执行多少拍，都可以要求所有指令顺序写回，而且所有例外在写回阶段统一处理。

当然，一味地通过阻塞流水线解决所有问题会引起流水线效率的下降。因此，实际处理器实现时都会采取很多其它方法来提高流水线效率。例如，对于例外的处理就有好多种办法。一是不要精确例外，典型的例子像做科学计算的机器，不做精确例外，一旦发生例外，就不恢复了，或者设置精确例外和非精确例外模式，在非精确例外情况下流水线的效率高，在精确例外情况下严格保证指令的执行次序，效率低。二是通过增加硬件把指令执行结果先缓存起来，直到前面的指令都执行完了而且没有例外之后，再把结果写回；如果前面的指令发生例外就把后面的指令取消掉。三是硬件不负责精确例外现场，但发生例外时保留足够的信息以便软件可以恢复现场。

5.9 本章小结

回顾一下这个简单 CPU 的设计。先从 MIPS 指令集中选取了十几条简单指令构成了一个简单的指令系统。根据该指令系统搭了一个数据通路，主要是寄存器、ALU、存储器、选择器等。在数据通路的基础上实现了控制逻辑，就是根据指令来控制数据通路，指令就是强制性的命令，指令往指令寄存器中一站，整个通路和功能部件就得听它的。然后给 CPU 加上时钟，每条指令分成计算 PC、取指、执行并写回三个步骤，再把一条一条指令串起来。可以把计算 PC 和执行合并成一个步骤，并和取指重叠执行，相当于是两级流水线，一条在取指，一条在执行。在此基础上，把执行阶段再细分，得到标准的 5 级静态流水线，提高了主频。在 5 级流水线中，指令相关容易引起冲突，冲突时通过堵塞流水线保证指令的正确执行。数据相关的指令通过寄存器传递数据，可以通过数据前递直接把运算结果连接到 ALU 的输入，以提高流水线效率。指令在流水线中可能出现各种各样的例外，例外通路和数据通路都是处理器的重要组成部分；为了实现精确例外，在 5 级静态流水线中把例外信息随流水线记录下来等到写回的时候统一处理；例外处理时，把发生例外指令的 PC 存起来以便在例外处理后恢复，并跳转到约定的例外入口地址交由操作系统处理。最后，介绍了多功能部件和多拍操作对流水线的

影响。

这个简单的 CPU，一步一步地，从无到有，从简单到复杂，把前几章讲的东西串在了一起。希望大家用 Verilog 把这个简单的 CPU 写出来，调试通了，并运行一个小程序。当你把它调通能跑程序以后，你会觉得非常有意思，自己做出了一个能跑的 CPU。