存储器做读操作,存储器就可将指定地址单元内的指令读至 MDR,再由 MDR 送至 IR。

在 CPU 内部必须给 ALU 提供数据,因此 ALU 必须可直接访问 MDR 和用户可见寄存器,ALU 的外围还可以有另一些寄存器,这些寄存器用于 ALU 的输入输出以及用于和 MDR 及用户可见寄存器交换数据(如图 9.4 中的 Y 和 Z 寄存器)。

在 CPU 的控制和状态寄存器中,还有用来存放程序状态字 PSW 的寄存器,该寄存器用来存放条件码和其他状态信息。在具有中断系统的机器中还有中断标记寄存器。

3. 举例

不同计算机的 CPU 中,寄存器组织是不一样的,图 8.3 画出了 Z8000、8086 和 MC68000 三种计算机的寄存器组织。

图 8.3　三种微处理器的寄存器组织

Zilog Z8000 有 16 个 16 位的通用寄存器,这些寄存器可存放地址、数据,也可作为变址寄存器,其中有两个寄存器被用作栈指针,寄存器可被用作 8 位和 32 位的运算。Z8000 中有 5 个与程序状态有关的寄存器,一个用于存放状态标记,两个用于程序计数器,两个用于存放偏移量。确定一个地址需要两个寄存器。

Intel 8086 采用不同的寄存器组织，尽管某些寄存器可以通用，但它的每个寄存器大多是专用的。它有 4 个 16 位的数据寄存器，即 AX（累加器）、BX（基址寄存器）、CX（计数寄存器）和 DX（数据寄存器），也可兼作 8 个 8 位的寄存器（AH、AL、BH、BL、CH、CL、DH、DL）。另外，还有两个 16 位的指针（栈指针 SP 和基址指针 BP）和两个变址寄存器（源变址寄存器 SI 和目的变址寄存器 DI）。在一些指令中，寄存器是隐式使用的，如乘法指令总是用累加器。8086 还有 4 个段地址寄存器（代码段 CS、数据段 DS、堆栈段 SS 和附加段 ES）以及指令指针 IP（相当于 PC）和状态标志寄存器 F。

Motorola MC68000 的寄存器组织介于 Zilog 和 Intel 微处理器之间，它将 32 位寄存器分为 8 个数据寄存器（$D_0 \sim D_7$）和 9 个地址寄存器（$A_0 \sim A_7'$）。数据寄存器主要用于数据运算，当需要变址时，也可作变址寄存器使用。寄存器允许 8 位、16 位和 32 位的数据运算，这由操作码确定。地址寄存器存放 32 位地址（没有段），其中两个（A_7 和 A_7'）也可用作堆栈指针，分别供用户和操作系统使用。针对当前执行的模式，这两个寄存器在某个时刻只能用一个。此外，MC68000 还有一个 32 位的程序计数器 PC 和一个 16 位的状态寄存器。

与 Zilog 的设计者类似，Motorola 设计的寄存器组织也不含专用寄存器。至于到底什么形式的寄存器组织最好，目前尚无一致的观点，主要由设计者根据需要自行决定。

计算机的设计者们为了给在早期计算机上编写的程序提供向上的兼容性，在新计算机的设计上经常保留原设计的寄存器组织形式。图 8.4 就是 Zilog 80000 和 Intel 80386 的用户可见寄存器组织，它们分别是 Z8000 和 8086 的扩展，它们都采用 32 位寄存器，但又分别保留了原先的一些特点。由于受这种限制，因此 32 位处理器在寄存器组织的设计上只有有限的灵活性。

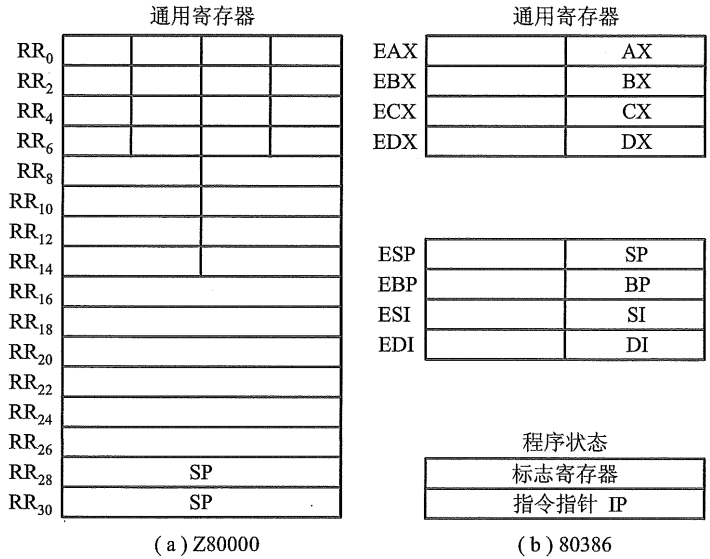

图 8.4　两种 32 位微处理器寄存器组织

8.1.4　控制单元和中断系统

控制单元(CU)是提供完成计算机全部指令操作的微操作命令序列部件。现代计算机中微操作命令序列的形成方法有两种:一种是组合逻辑设计方法,为硬连线逻辑;另一种是微程序设计方法,为存储逻辑。具体内容详见第 4 篇。

中断系统主要用于处理计算机的各种中断,详细内容在 8.4 节介绍。

8.2　指令周期

8.2.1　指令周期的基本概念

CPU 每取出并执行一条指令所需的全部时间称为指令周期,也即 CPU 完成一条指令的时间,如图 8.5 所示。图中的取指阶段完成取指令和分析指令的操作,又称取指周期;执行阶段完成执行指令的操作,又称执行周期。在大多数情况下,CPU 就是按“取指—执行—再取指—再执行…”的顺序自动工作的。

由于各种指令操作功能不同,因此各种指令的指令周期是不相同的。例如,无条件转移指令“JMP X”,在执行阶段不需要访问主存,而且操作简单,完全可以在取指阶段的后期将转移地址 X 送至 PC,以达到转移的目的。这样,“JMP X”指令的指令周期就是取指周期。又如一地址格式的加法指令“ADD X”,在执行阶段首先要从 X 所指示的存储单元中取出操作数,然后和 ACC 的内容相加,结果存于 ACC,故这种指令的指令周期在取指和执行阶段各访问一次存储器,其指令周期就包括两个存取周期。再如乘法指令,其执行阶段所要完成的操作比加法指令多得多,故它的执行周期超过了加法指令,如图 8.6 所示。

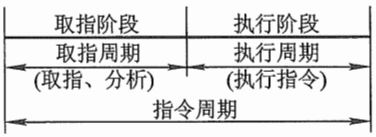

图 8.5　指令周期定义示意图

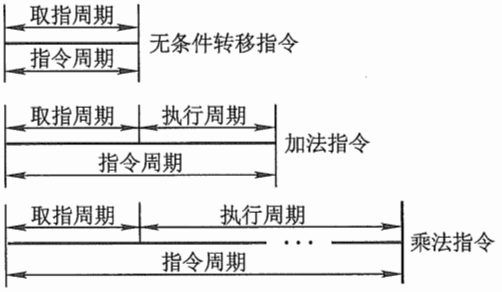

图 8.6　各种指令周期的比较

此外，当遇到间接寻址的指令时，由于指令字中只给出操作数有效地址的地址，因此，为了取出操作数，需先访问一次存储器，取出有效地址，然后再访问存储器，取出操作数，如图 7.11(a)所示。这样，间接寻址的指令周期就包括取指周期、间址周期和执行周期 3 个阶段，其中间址周期用于取操作数的有效地址，因此间址周期介于取指周期和执行周期之间，如图 8.7 所示。

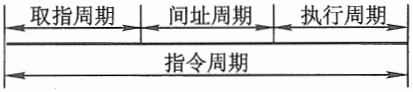

图 8.7　具有间址周期的指令周期

由第 5 章可知，当 CPU 采用中断方式实现主机与 I/O 设备交换信息时，CPU 在每条指令执行阶段结束前，都要发中断查询信号，以检测是否有某个 I/O 设备提出中断请求。如果有请求，CPU 则要进入中断响应阶段，又称中断周期。在此阶段，CPU 必须将程序断点保存到存储器中。这样，一个完整的指令周期应包括取指、间址、执行和中断 4 个子周期，如图 8.8 所示。由于间址周期和中断周期不一定包含在每个指令周期内，故图中用菱形框判断。

总之，上述 4 个周期都有 CPU 访存操作，只是访存的目的不同。取指周期是为了取指令，间址周期是为了取有效地址，执行周期是为了取操作数(当指令为访存指令时)，中断周期是为了保存程序断点。这 4 个周期又可称为 CPU 的工作周期，为了区别它们，在 CPU 内可设置 4 个标志触发器，如图 8.9 所示。

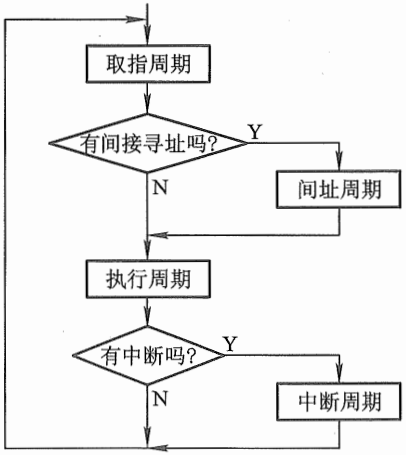

图 8.8　指令周期流程

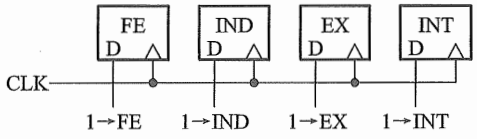

图 8.9　CPU 工作周期的标志

图 8.9 所示的 FE、IND、EX 和 INT 分别对应取指、间址、执行和中断 4 个周期，并以“1”状态表示有效，它们分别由 1→FE、1→IND、1→EX 和 1→INT 这 4 个信号控制。

设置 CPU 工作周期标志触发器对设计控制单元十分有利。例如，在取指阶段，只要设置取指周期标志触发器 FE 为 1，由它控制取指阶段的各个操作，便获得对任何一条指令的取指命令序列。又如，在间接寻址时，间址次数可由间址周期标志触发器 IND 确定，当它为“0”状态时，表

示间接寻址结束。再如,对于一些执行周期不访存的指令(如转移指令、寄存器类型指令),同样可以用它们的操作码与取指周期标志触发器的状态相“与”,作为相应微操作的控制条件。这些特点读者在控制单元的设计中可进一步体会。

8.2.2　指令周期的数据流

为了便于分析指令周期中的数据流,假设 CPU 中有存储器地址寄存器 MAR、存储器数据寄存器 MDR、程序计数器 PC 和指令寄存器 IR。

1. 取指周期的数据流

图 8.10 所示的是取指周期的数据流。PC 中存放现行指令的地址,该地址送到 MAR 并送至地址总线,然后由控制部件 CU 向存储器发读命令,使对应 MAR 所指单元的内容(指令)经数据总线送至 MDR,再送至 IR,并且 CU 控制 PC 内容加 1,形成下一条指令的地址。

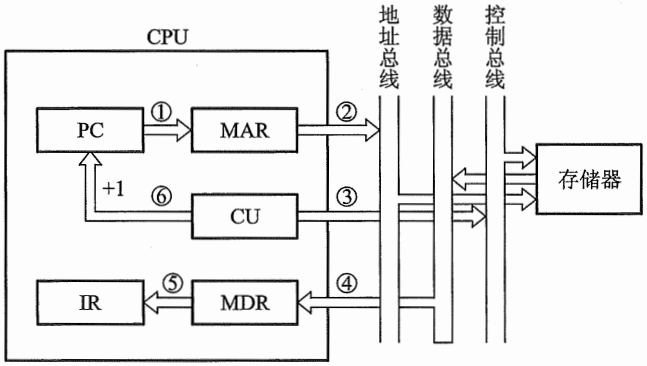

图 8.10　取指周期数据流

2. 间址周期的数据流

间址周期的数据流如图 8.11 所示。一旦取指周期结束,CU 便检查 IR 中的内容,以确定其

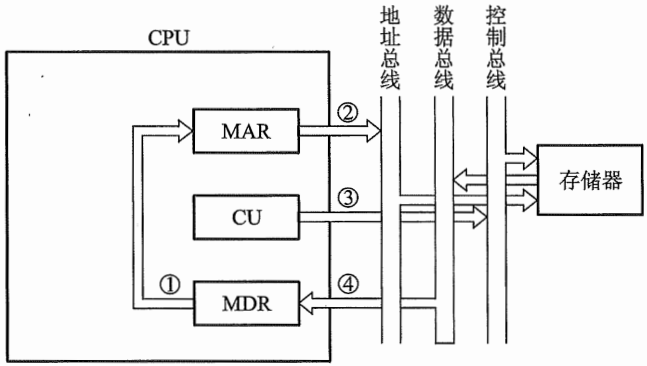

图 8.11　间址周期数据流

是否有间址操作，如果需要间址操作，则 MDR 中指示形式地址的右 N 位（记作 Ad(MDR)）将被送到 MAR，又送至地址总线，此后 CU 向存储器发读命令，以获取有效地址并存至 MDR。

3. 执行周期的数据流

由于不同的指令在执行周期的操作不同，因此执行周期的数据流是多种多样的，可能涉及 CPU 内部寄存器间的数据传送、对存储器（或 I/O）进行读写操作或对 ALU 的操作，因此，无法用统一的数据流图表示。

4. 中断周期的数据流

CPU 进入中断周期要完成一系列操作（详见 9.1 节），其中 PC 当前的内容必须保存起来，以待执行完中断服务程序后可以准确返回到该程序的间断处，这一操作的数据流如图 8.12 所示。

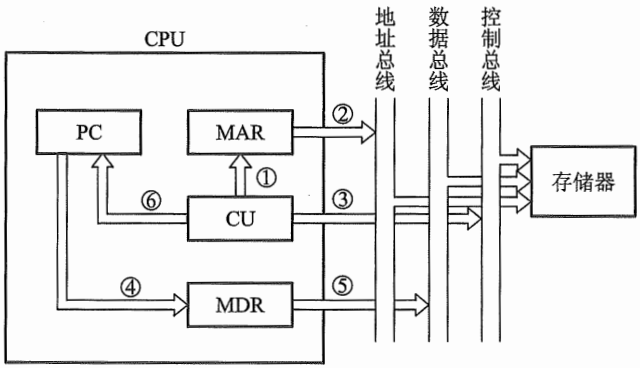

图 8.12　中断周期数据流

图中由 CU 把用于保存程序断点的存储器特殊地址（如栈指针的内容）送往 MAR，并送到地址总线上，然后由 CU 向存储器发写命令，并将 PC 的内容（程序断点）送到 MDR，最终使程序断点经数据总线存入存储器。此外，CU 还需将中断服务程序的入口地址送至 PC，为下一个指令周期的取指周期做好准备。

8.3　指令流水

由前面各章的介绍可知，为了提高访存速度，一方面要提高存储芯片的性能，另一方面可以从体系结构上，如采用多体、Cache 等分级存储措施来提高存储器的性能/价格比。为了提高主机与 I/O 交换信息的速度，可以采用 DMA 方式，也可以采用多总线结构，将速度不一的 I/O 分别挂到不同带宽的总线上，以解决总线的瓶颈问题。为了提高运算速度，可以采用高速芯片和快速进位链，以及改进算法等措施。为了进一步提高处理机速度，通常可从提高器件的性能和改进系统的结构，开发系统的并行性两方面入手。

(1) 提高器件的性能

提高器件的性能一直是提高整机性能的重要途径,计算机的发展史就是按器件把计算机分为电子管、晶体管、集成电路和大规模集成电路4代的。器件的每一次更新换代都使计算机的软硬件技术和计算机性能获得突破性进展。特别是大规模集成电路的发展,由于其集成度高、体积小、功耗低、可靠性高、价格便宜等特点,使人们可采用更复杂的系统结构造出性能更高、工作更可靠、价格更低的计算机。但是由于半导体器件的集成度越来越接近物理极限,使器件速度的提高越来越慢。

(2) 改进系统的结构,开发系统的并行性

所谓并行,包含同时性和并发性两个方面。前者是指两个或多个事件在同一时刻发生,后者是指两个或多个事件在同一时间段发生。也就是说,在同一时刻或同一时间段内完成两种或两种以上性质相同或不同的功能,只要在时间上互相重叠,就存在并行性。

并行性体现在不同等级上。通常分为4个级别:作业级或程序级、任务级或进程级、指令之间级和指令内部级。前两级为粗粒度,又称为过程级;后两级为细粒度,又称为指令级。粗粒度并行性(Coarse-grained Parallelism)一般用算法(软件)实现,细粒度并行性(Fine-grained Parallelism)一般用硬件实现。从计算机体系上看,粗粒度并行性是在多个处理机上分别运行多个进程,由多台处理机合作完成一个程序;细粒度并行性是指在处理机的操作级和指令级的并行性,其中指令的流水作业就是一项重要技术。这里只讨论有关指令流水的一些主要问题,其他有关粗粒度并行和粗粒度并行技术将在"计算机体系结构"课程中讲述。

8.3.1 指令流水原理

指令流水类似于工厂的装配线,装配线利用了产品在装配的不同阶段其装配过程不同这一特点,使不同产品处在不同的装配段上,即每个装配段同时对不同产品进行加工,这样可大大提高装配效率。将这种装配生产线的思想用到指令的执行上,就引出了指令流水的概念。

从上面的分析可知,完成一条指令实际上也可分为许多阶段。为简单起见,把指令的处理过程分为取指令和执行指令两个阶段,在不采用流水技术的计算机里,取指令和执行指令是周而复始地重复出现,各条指令按顺序串行执行的,如图8.13所示。

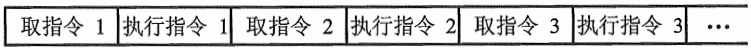

图 8.13 指令的串行执行

图中取指令的操作可由指令部件完成,执行指令的操作可由执行部件完成。进一步分析发现,这种顺序执行虽然控制简单,但执行中各部件的利用率不高,如指令部件工作时,执行部件基本空闲,而执行部件工作时,指令部件基本空闲。如果指令执行阶段不访问主存,则完全可以利用这段时间取下一条指令,这样就使取下一条指令的操作和执行当前指令的操作同时进行,如图

8.14 所示，这就是两条指令的重叠，即指令的二级流水。

图 8.14　指令的二级流水

由指令部件取出一条指令，并将它暂存起来，如果执行部件空闲，就将暂存的指令传给执行部件执行。与此同时，指令部件又可取出下一条指令并暂存起来，这称为指令预取。显然，这种工作方式能加速指令的执行。如果取指和执行阶段在时间上完全重叠，相当于将指令周期减半。然而进一步分析流水线，就会发现存在两个原因使得执行效率加倍是不可能的。

① 指令的执行时间一般大于取指时间，因此，取指阶段可能要等待一段时间，也即存放在指令部件缓冲区的指令还不能立即传给执行部件，缓冲区不能空出。

② 当遇到条件转移指令时，下一条指令是不可知的，因为必须等到执行阶段结束后，才能获知条件是否成立，从而决定下条指令的地址，造成时间损失。

通常为了减少时间损失，采用猜测法，即当条件转移指令从取指阶段进入执行阶段时，指令部件仍按顺序预取下一条指令。这样，如果条件不成立，转移没有发生，则没有时间损失；若条件成立，转移发生，则所取的指令必须丢掉，并再取新的指令。

尽管这些因素降低了两级流水线的潜在效率，但还是可以获得一定程度的加速。为了进一步提高处理速度，可将指令的处理过程分解为更细的几个阶段。

- 取指(FI)：从存储器取出一条指令并暂时存入指令部件的缓冲区。
- 指令译码(DI)：确定操作性质和操作数地址的形成方式。
- 计算操作数地址(CO)：计算操作数的有效地址，涉及寄存器间接寻址、间接寻址、变址寻址、基址寻址、相对寻址等各种地址计算方式。
- 取操作数(FO)：从存储器中取操作数(若操作数在寄存器中，则无须此阶段)。
- 执行指令(EI)：执行指令所需的操作，并将结果存于目的位置(寄存器中)。
- 写操作数(WO)：将结果存入存储器。

为了说明方便起见，假设上述各段的时间都是相等的(即每段都为一个时间单元)，于是可得图 8.15 所示的指令六级流水时序。在这个流水线中，处理器有 6 个操作部件，同时对 6 条指令进行加工，加快了程序的执行速度。

图中 9 条指令若不采用流水线技术，最终出结果需要 54 个时间单元，采用六级流水只需要 14 个时间单元就可出最后结果，大大提高了处理器速度。当然，图中假设每条指令都经过流水线的 6 个阶段，但事实并不总是这样。例如，取数指令并不需要 WO 阶段。此外，这里还假设不存在存储器访问冲突，所有阶段均并行执行。如 FI、FO 和 WO 阶段都涉及存储器访问，如果出现冲突就无法并行执行，图 8.15 示意了所有这些访问都可以同时进行，但多数存储系统做不到这点，从而影响了流水线的性能。

还有一些其他因素也会影响流水线性能，例如，6 个阶段时间不等或遇到转移指令，都会出现讨论二级流水时出现的问题。

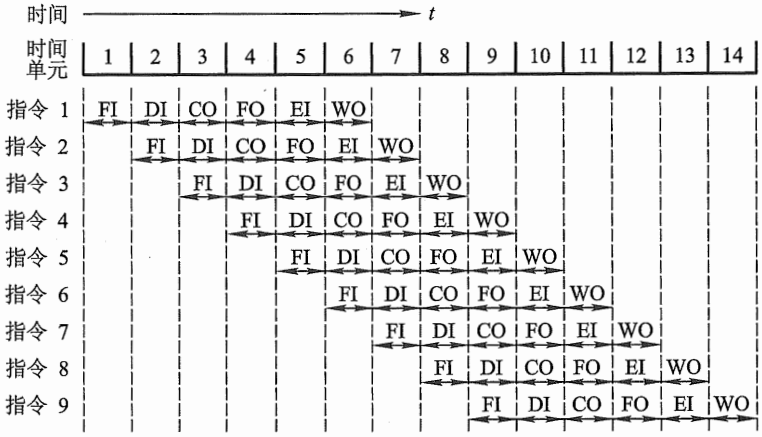

图 8.15　指令六级流水时序

8.3.2　影响流水线性能的因素

要使流水线具有良好的性能，必须设法使流水线能畅通流动，即必须做到充分流水，不发生断流。但通常由于在流水过程中会出现三种相关，使流水线不断流实现起来很困难，这三种相关是结构相关、数据相关和控制相关。

结构相关是当多条指令进入流水线后，硬件资源满足不了指令重叠执行的要求时产生的。数据相关是指令在流水线中重叠执行时，当后继指令需要用到前面指令的执行结果时发生的。控制相关是当流水线遇到分支指令和其他改变 PC 值的指令时引起的。

为了讨论方便起见，假设流水线由 5 段组成，它们分别是取指令(IF)、指令译码/读寄存器(ID)、执行/访存有效地址计算(EX)、存储器访问(MEM)、结果写回寄存器(WB)。

不同类型指令在各流水段的操作是不同的，表 8.1 列出了 ALU 类指令、访存类(取数、存数)指令和转移类指令在各流水段中所进行的操作。

表 8.1　不同类型指令在各流水段中所进行的操作

流水段	指　令		
	ALU	取/存	转移
IF	取指	取指	取指
ID	译码 读寄存器堆	译码 读寄存器堆	译码 读寄存器堆
EX	执行	计算访存有效地址	计算转移目标地址， 设置条件码

续表

流水段	指　令		
	ALU	取/存	转移
MEM	—	访存(读/写)	若条件成立,将转移目标地址送 PC
WB	结果写回寄存器堆	将读出的数据写入寄存器堆	—

下面分析上述三种相关对流水线工作的影响。

1. 结构相关

结构相关是当指令在重叠执行过程中,不同指令争用同一功能部件产生资源冲突时产生的,故又有资源相关之称。

通常,大多数机器都是将指令和数据保存在同一存储器中,且只有一个访问口,如果在某个时钟周期内,流水线既要完成某条指令对操作数的存储器访问操作,又要完成另一条指令的取指操作,这就会发生访存冲突。如表 8.2 中,在第 4 个时钟周期,第 i 条指令(LOAD)的 MEM 段和第 i+3 条指令的 IF 段发生了访存冲突。解决冲突的方法可以让流水线在完成前一条指令对数据的存储器访问时,暂停(一个时钟周期)取后一条指令的操作,如表 8.3 所示。当然,如果第 i 条指令不是 LOAD 指令,在 MEM 段不访存,也就不会发生访存冲突。

表 8.2　两条指令同时访存造成结构相关冲突

指令	时钟周期							
	1	2	3	4	5	6	7	8
LOAD 指令	IF	ID	EX	MEM	WB			
指令 i+1		IF	ID	EX	MEM	WB		
指令 i+2			IF	ID	EX	MEM	WB	
指令 i+3				IF	ID	EX	MEM	WB
指令 i+4					IF	ID	EX	MEM

表 8.3　解决访存冲突的一种方案

指令	时钟周期								
	1	2	3	4	5	6	7	8	9
LOAD 指令	IF	ID	EX	MEM	WB				
指令 i+1		IF	ID	EX	MEM	WB			
指令 i+2			IF	ID	EX	MEM	WB		
指令 i+3				停顿	IF	ID	EX	MEM	WB
指令 i+4						IF	ID	EX	MEM

解决访存冲突的另一种方法是设置两个独立的存储器分别存放操作数和指令，以免取指令和取操作数同时进行时互相冲突，使取某条指令和取另一条指令的操作数实现时间上的重叠。还可以采用指令预取技术，例如，在 CPU(8086)中设置指令队列，将指令预先取到指令队列中排队。指令预取技术的实现基于访存周期很短的情况，例如，在执行指令阶段，取数时间很短，因此在执行指令时，主存会有空闲，此时，只要指令队列空出，就可取下一条指令，并放至空出的指令队列中，从而保证在执行第 K 条指令的同时对第 $K+1$ 条指令进行译码，实现“执行 K”与“分析 $K+1$”的重叠。

2. 数据相关

数据相关是流水线中的各条指令因重叠操作，可能改变对操作数的读写访问顺序，从而导致了数据相关冲突。例如，流水线要执行以下两条指令：

```
ADD   R1,R2,R3        ;(R2)+(R3)→R1
SUB   R4,R1,R5        ;(R1)-(R5)→R4
```

这里第二条 SUB 指令中 R_1 的内容必须是第一条 ADD 指令的执行结果。可见正常的读写顺序是先由 ADD 指令写入 R_1，再由 SUB 指令来读 R_1。在非流水线时，这种先写后读的顺序是自然维持的。但在流水线时，由于重叠操作，使读写的先后顺序关系发生了变化，如表 8.4 所示。

表 8.4　ADD 和 SUB 指令发生先写后读(RAW)的数据相关冲突

指令	时钟周期					
	1	2	3	4	5	6
ADD	IF	ID	EX	MEM	WB	
SUB		IF	ID	EX	MEM	WB
			读 R_1		写 R_1	

由表 8.4 可见，在第 5 个时钟周期，ADD 指令方可将运算结果写入 R_1，但后继 SUB 指令在第 3 个时钟周期就要从 R_1 中读数，使先写后读的顺序改变为先读后写，发生了先写后读(RAW)的数据相关冲突。如果不采取相应的措施，按表 8.4 的读写顺序，就会使操作结果出错。解决这种数据相关的方法可以采用后推法，即遇到数据相关时，就停顿后继指令的运行，直至前面指令的结果已经生成。例如，流水线要执行下列指令序列：

```
ADD R1,R2,R3      ;(R2)+(R3)→R1
SUB R4,R1,R5      ;(R1)-(R5)→R4
AND R6,R1,R7      ;(R1) AND (R7)→R6
OR  R8,R1,R9      ;(R1) OR (R9)→R8
XOR R10,R1,R11    ;(R1) XOR (R11)→R10
```

其中，第一条 ADD 指令将向 R_1 寄存器写入操作结果，后继的 4 条指令都要使用 R_1 中的值作为一个源操作数，显然，这时就出现了前述的 RAW 数据相关。表 8.5 列出了未对数据相关进行特

殊处理的流水线，表中 ADD 指令在 WB 段才将计算结果写入寄存器 R_1 中，但 SUB 指令在其 ID 段就要从寄存器 R_1 中读取该计算结果。同样，AND 指令、OR 指令也要受到这种相关关系的影响。对于 XOR 指令，由于其 ID 段（第 6 个时钟周期）在 ADD 指令的 WB 段（第 5 个时钟周期）之后，因此可以正常操作。

表 8.5　未对数据相关进行特殊处理的流水线

指令	时钟周期								
	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
SUB		IF	ID	EX	MEM	WB			
AND			IF	ID	EX	MEM	WB		
OR				IF	ID	EX	MEM	WB	
XOR					IF	ID	EX	MEM	WB

如果采用后推法，即将相关指令延迟到所需操作数被写回到寄存器后再执行的方式，就可解决这种数据相关冲突，其流水线如表 8.6 所示。显然这将要使流水线停顿 3 个时钟周期。

表 8.6　对数据相关进行特殊处理的流水线

指令	时钟周期											
	1	2	3	4	5	6	7	8	9	10	11	12
ADD	IF	ID	EX	MEM	WB							
SUB		IF				ID	EX	MEM	WB			
AND			IF				ID	EX	MEM	WB		
OR				IF				ID	EX	MEM	WB	
XOR					IF				ID	EX	MEM	WB

另一种解决方法是采用定向技术，又称为旁路技术或相关专用通路技术。其主要思想是不必待某条指令的执行结果送回到寄存器后，再从寄存器中取出该结果，作为下一条指令的源操作数，而是直接将执行结果送到其他指令所需要的地方。上述 5 条指令序列中，实际上要写入 R_1 的 ADD 指令在 EX 段的末尾处已形成，如果设置专用通路技术，将此时产生的结果直接送往需要它的 SUB、AND 和 OR 指令的 EX 段，就可以使流水线不发生停顿。显然，此时要对 3 条指令进行定向传送操作。图 8.16 示出了带有旁路技术的 ALU 执行部件。图中有两个暂存器，当 AND 指令将进入 EX 段时，ADD 指令的执行结果已存入暂存器 2，SUB 指令的执行结果已存入暂存器 1，而暂存器 2 的内容（存放送往 R_1 的结果）可通过旁路通道，经多路开关送到 ALU 中。这里的定向传送仅发生在 ALU 内部。

根据指令间对同一寄存器读和写操作的先后次序关系，数据相关冲突可分为写后读相关（Read After Write，RAW）、读后写相关（Write After Read，WAR）和写后写相关（Write After Write，WAW）。例如，有 i 和 j 两条指令，i 指令在前，j 指令在后，则三种不同类型的数据相关含义如下。

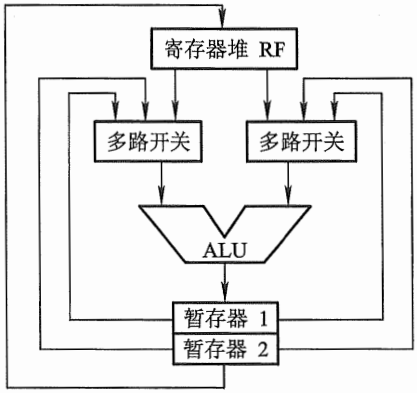

图 8.16　带有旁路技术的 ALU 部件

① 写后读相关：指令 j 试图在指令 i 写入寄存器前就读出该寄存器内容，这样，指令 j 就会错误地读出该寄存器旧的内容。

② 读后写相关：指令 j 试图在指令 i 读出寄存器之前就写入该寄存器，这样，指令 i 就错误地读出该寄存器新的内容。

③ 写后写相关：指令 j 试图在指令 i 写入寄存器之前就写入该寄存器，这样，两次写的先后次序被颠倒，就会错误地使由指令 i 写入的值成为该寄存器的内容。

上述三种数据相关在按序流动的流水线中，只可能出现 RAW 相关。在非按序流动的流水线中，由于允许后进入流水线的指令超过先进入流水线的指令而先流出流水线，则既可能发生 RAW 相关，还可能发生 WAR 和 WAW 相关。

3. 控制相关

控制相关主要是由转移指令引起的。统计表明，转移指令约占总指令的 1/4，比起数据相关来，它会使流水线丧失更多的性能。当转移发生时，将使流水线的连续流动受到破坏。当执行转移指令时，根据是否发生转移，它可能将程序计数器 PC 内容改变成转移目标地址，也可能只是使 PC 加上一个增量，指向下一条指令的地址。图 8.17 示意了条件转移的效果。这里使用了和

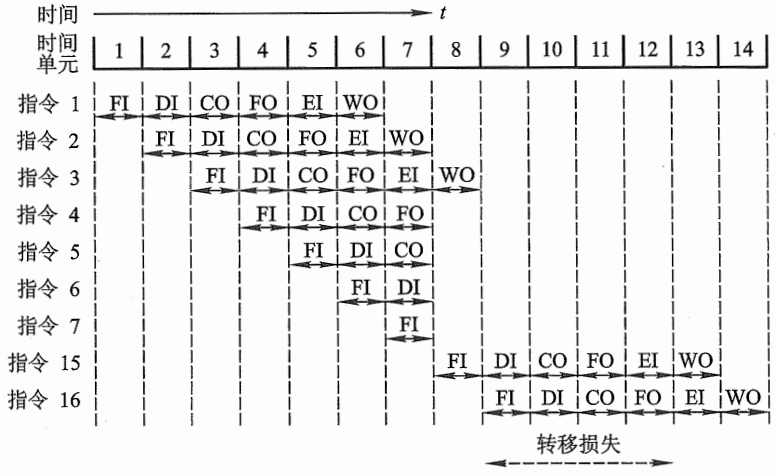

图 8.17　条件转移对指令流水操作的影响

图 8.15 相同的程序，并假设指令 3 是一条条件转移指令，即指令 3 必须待指令 2 的结果出现后（第 7 个时间单元）才能决定下一条指令是 4（条件不满足）还是 15（条件满足）。由于结果无法预测，此流水线继续预取指令 4，并向前推进。当最后结果满足条件时，发现对第 4、5、6、7 条指令所做的操作全部报废。在第 8 个时间单元，指令 15 进入流水线。在时间单元 9~12 之间没有指令完成，这就是由于不能预测转移条件而带来的性能损失。而图 8.15 中因转移条件不成立，未发生转移，得到了较好的流水线性能。

为了解决控制相关，可以采用尽早判别转移是否发生，尽早生成转移目标地址；预取转移成功或不成功两个控制流方向上的目标指令；加快和提前形成条件码；提高转移方向的猜准率等方法。有关的详细内容，读者可查阅相关资料进一步了解。

8.3.3　流水线性能

流水线性能通常用吞吐率、加速比和效率 3 项指标来衡量。

1. 吞吐率（Throughput Rate）

在指令级流水线中，吞吐率是指单位时间内流水线所完成指令或输出结果的数量。吞吐率又有最大吞吐率和实际吞吐率之分。

最大吞吐率是指流水线在连续流动达到稳定状态（参见图 8.15 第 6~9 个时间单元，流水线中各段都处于工作状态）后所获得的吞吐率。对于 m 段的指令流水线而言，若各段的时间均为 Δt，则最大吞吐率为

$$T_{p\max}=\frac{1}{\Delta t}$$

流水线仅在连续流动时才可达到最大吞吐率。实际上由于流水线在开始时有一段建立时间（第一条指令输入后到其完成的时间），结束时有一段排空时间（最后一条指令输入后到其完成的时间），以及由于各种相关因素使流水线无法连续流动，因此，实际吞吐率总是小于最大吞吐率。

实际吞吐率是指流水线完成 n 条指令的实际吞吐率。对于 m 段的指令流水线，若各段的时间均为 Δt，连续处理 n 条指令，除第一条指令需 $m\cdot\Delta t$ 外，其余 $(n-1)$ 条指令，每隔 Δt 就有一个结果输出，即总共需 $m\cdot\Delta t+(n-1)\Delta t$ 时间，故实际吞吐率为

$$T_p=\frac{n}{m\Delta t+(n-1)\Delta t}=\frac{1}{\Delta t[1+(m-1)/n]}=\frac{T_{p\max}}{1+(m-1)/n}$$

仅当 $n\gg m$ 时，才会有 $T_p\approx T_{p\max}$。

图 8.15 所示的六级流水线中，设每段时间为 Δt，其最大吞吐率为 $\frac{1}{\Delta t}$，完成 9 条指令的实际吞吐率为 $\frac{9}{6\Delta t+(9-1)\Delta t}$。

2. 加速比(Speedup Ratio)

流水线的加速比是指 m 段流水线的速度与等功能的非流水线的速度之比。如果流水线各段时间均为 Δt,则完成 n 条指令在 m 段流水线上共需 $T=m\cdot\Delta t+(n-1)\Delta t$ 时间。而在等效的非流水线上所需时间为 $T'=nm\Delta t$。故加速比 S_p 为

$$S_p=\frac{nm\Delta t}{m\Delta t+(n-1)\Delta t}=\frac{nm}{m+n-1}=\frac{m}{1+(m-1)/n}$$

可以看出,在 $n\gg m$ 时,S_p 接近于 m,即当流水线各段时间相等时,其最大加速比等于流水线的段数。

3. 效率(Efficiency)

效率是指流水线中各功能段的利用率。由于流水线有建立时间和排空时间,因此各功能段的设备不可能一直处于工作状态,总有一段空闲时间。图 8.18 是 4 段($m=4$)流水线的时空图,各段时间相等,均为 Δt。图中 $mn\Delta t$ 是流水线各段处于工作时间的时空区,而流水线中各段总的时空区是 $m(m+n-1)\Delta t$。通常用流水线各段处于工作时间的时空区与流水线中各段总的时空区之比来衡量流水线的效率。用公式表示为

$$E=\frac{mn\Delta t}{m(m+n-1)\Delta t}=\frac{n}{m+n-1}=\frac{S_p}{m}=T_p\Delta t$$

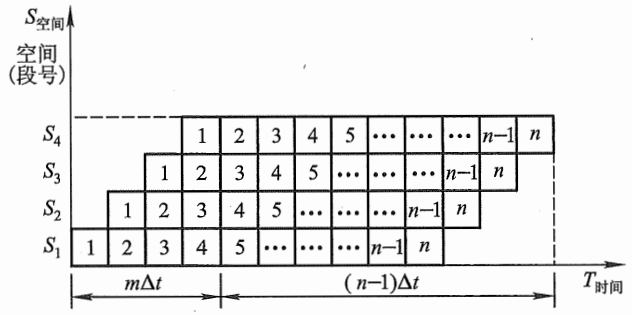

图 8.18　各段时间相等的流水线时空图

例 8.1　假设指令流水线分取指(IF)、译码(ID)、执行(EX)、回写(WR)4 个过程段,共有 10 条指令连续输入此流水线。

(1) 画出指令周期流程。

(2) 画出非流水线时空图。

(3) 画出流水线时空图。

(4) 假设时钟周期为 100 ns,求流水线的实际吞吐率。

(5) 求该流水处理器的加速比。

解:(1) 指令周期包括 IF、ID、EX、WR 这 4 个子过程,图 8.19(a)为指令周期流程图。

(2) 非流水线时空图如图 8.19(b)所示。假设一个时间单位为一个时钟周期,则每隔 4 个