第5章 中央处理机

本章从分析 CPU 的功能和内部结构入手，详细讨论机器完成一条指令的全过程，以及为了进一步提高数据的处理能力、开发系统的并行性所采取的流水技术。详细讲述 CPU 的功能和基本组成，指令周期的概念，时序产生器，微程序控制器，硬连线控制器，传统 CPU 的结构。在此基础上，介绍流水 CPU、RISC CPU 等先进的计算机科学技术成果。

5.1 CPU 的功能和组成

5.1.1 CPU 的功能

当用计算机解决某个问题时，我们首先必须为它编写程序。程序是一个指令序列，这个序列明确告诉计算机应该执行什么操作，在什么地方找到用来操作的数据。一旦把程序装入内存储器，就可以由计算部件来自动完成取指令和执行指令的任务。专门用来完成此项工作的计算机部件称为中央处理机，通常简称 CPU。

CPU 对整个计算机系统的运行是极其重要的，它具有如下四方面的基本功能：

(1) 指令控制。程序的顺序控制，称为指令控制。由于程序是一个指令序列，这些指令的顺序不能任意颠倒，必须严格按程序规定的顺序进行，因此，保证机器按顺序执行程序是 CPU 的首要任务。

(2) 操作控制。一条指令的功能往往是由若干操作信号的组合来实现的，因此，CPU 管理并产生由内存取出的每条指令的操作信号，把各种操作信号送往相应的部件，从而控制这些部件按指令的要求进行动作。

(3) 时间控制。对各种操作实施时间上的定时，称为时间控制。因为在计算机中，各种指令的操作信号均受到时间的严格定时。另一方面，一条指令的整个执行过程也受到时间的严格定时。只有这样，计算机才能有条不紊地自动工作。

(4) 数据加工。所谓数据加工，就是对数据进行算术运算和逻辑运算处理。完成数据的加工处理，是 CPU 的根本任务。因为，原始信息只有经过加工处理后才能对人们有用。

5.1.2 CPU 的基本组成

早期的 CPU 由运算器和控制器两大部分组成。但是随着 ULSI 技术的发展，早期放在 CPU 芯片外部的一些逻辑功能部件，如浮点运算器、Cache、总线仲裁器等纷纷移入 CPU 内

部,因而使 CPU 的内部组成越来越复杂。这样 CPU 的基本部分变成了运算器、Cache、控制器三大部分。

为便于读者建立计算机的整体概念,图 5.1 给出了 CPU 的整体模型。

控制器由程序计数器、指令寄存器、指令译码器、时序产生器和操作控制器组成。对于冯·诺依曼结构的计算机而言,一旦程序进入存储器后,就可由计算机自动完成取指令和执行指令的任务,控制器就是专用于完成此项工作的,它是发布命令的"决策机构",它负责协调并控制计算机各部件执行程序的指令序列,其基本功能是取指令、分析指令和执行指令。

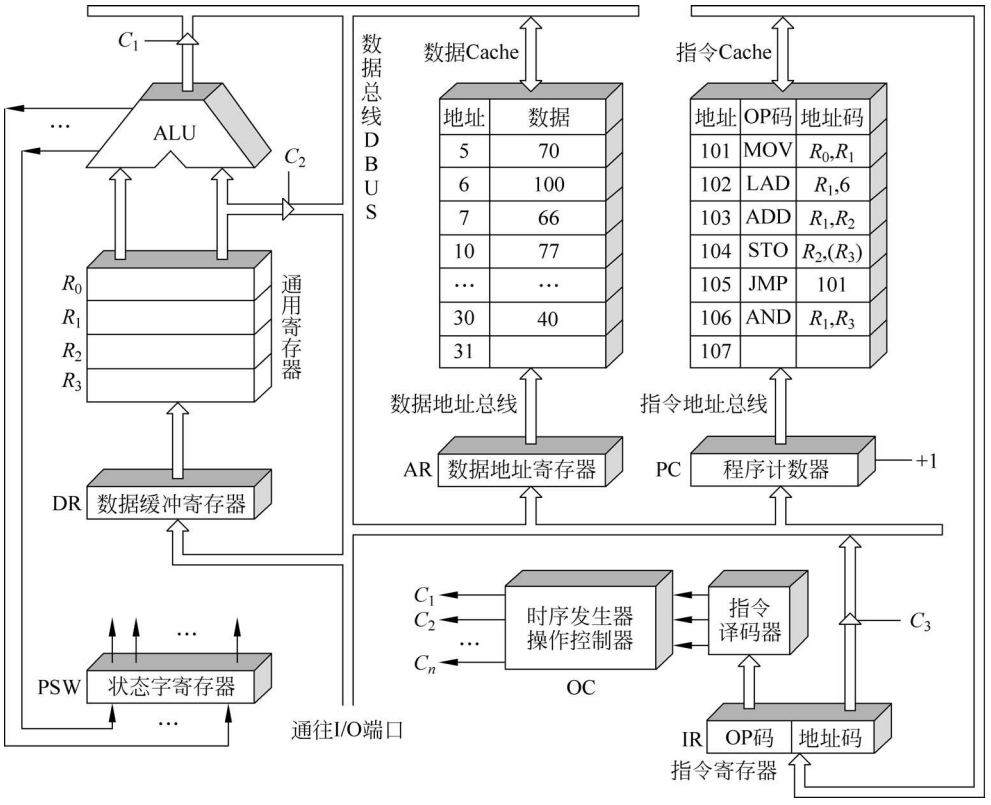


图 5.1 CPU 模型

控制器的主要功能有:

(1) 从指令 Cache 中取出一条指令,并指出下一条指令在指令 Cache 中的位置。

(2) 对指令进行译码或测试,并产生相应的操作控制信号,以便启动规定的动作。例如一次数据 Cache 的读/写操作,一个算术逻辑运算操作,或一个 I/O 操作。

(3) 指挥并控制 CPU、数据 Cache 和 I/O 设备之间数据流动的方向。

运算器由算术逻辑单元(Arithmetic and Logic Unit,ALU)、通用寄存器、数据缓冲寄存器 DR 和状态条件寄存器 PSW 组成,它是数据加工处理部件。相对控制器而言,运算器接受控制器的命令而进行动作,即运算器所进行的全部操作都是由控制器发出的控制信号来指挥的,所以它是执行部件。运算器有两个主要功能:

(1) 执行所有的算术运算。

(2) 执行所有的逻辑运算,并进行逻辑测试,如零值测试或两个值的比较。

通常，一个算术操作产生一个运算结果，而一个逻辑操作则产生一个判决。

鉴于第 2、3 章中已经详细讨论了运算器和存储器，所以本章重点放在控制器上。

5.1.3 CPU 中的主要寄存器

各种计算机的 CPU 可能有这样或那样的不同，但是在 CPU 中至少要有六类寄存器，如图 5.1 所示。这些寄存器是：①指令寄存器(IR)；②程序计数器(PC)；③数据地址寄存器(AR)；④数据缓冲寄存器(DR)；⑤通用寄存器(R_0～R_3)；⑥状态字寄存器(PSW)。

上述这些寄存器用来暂存一个计算机字。根据需要，可以扩充其数目。下面详细介绍这些寄存器的功能与结构。

(1) 数据缓冲寄冲器(DR)。数据缓冲寄存器用来暂时存放 ALU 的运算结果，或由数据存储器读出的一个数据字，或来自外部接口的一个数据字。缓冲寄存器的作用是：

① 作为 ALU 运算结果和通用寄存器之间信息传送中时间上的缓冲。

② 补偿 Cache 和内存、外围设备之间在操作速度上的差别。

(2) 指令寄存器(IR)。指令寄存器用来保存当前正在执行的一条指令。当执行一条指令时，先把它从指令存储器(简称指存)读出，然后再传送至指令寄存器。指令划分为操作码和地址码字段，由二进制数字组成。为了执行任何给定的指令，必须对操作码进行测试，以便识别所要求的操作。一个叫作指令译码器的部件就是做这项工作的。指令寄存器中操作码字段的输出就是指令译码器的输入。操作码一经译码后，即可向操作控制器发出具体操作的特定信号。

(3) 程序计数器(PC)。为了保证程序能够连续地执行下去，CPU 必须具有某些手段来确定下一条指令的地址。而程序计数器(PC)正是起到这种作用，所以它又称为指令计数器。在程序开始执行前，必须将它的起始地址，即程序的第一条指令所在的指存单元地址送入 PC，因此 CPU 的内容即是从指存提取的第一条指令的地址。当执行指令时，CPU 将自动修改 PC 的内容，以便使其保持的总是将要执行的下一条指令的地址。由于大多数指令都是按顺序来执行的，所以修改的过程通常只是简单的对 PC 加 1。

但是，当遇到转移指令如 JMP 指令时，那么后继指令的地址(即 PC 的内容)必须从指令寄存器中的地址字段取得。在这种情况下，下一条从指存取出的指令将由转移指令来规定，而不是像通常一样按顺序来取得。因此程序计数器的结构应当是具有寄存器和计数器两种功能的结构。

(4) 数据地址寄存器(AR)。数据地址寄存器用来保存当前 CPU 所访问的数据 Cache 存储器中(简称数存)单元的地址。由于要对存储器阵列进行地址译码，因此必须使用地址寄存器来保持地址信息，直到一次读/写操作完成为止。

地址寄存器的结构和数据缓冲寄存器、指令寄存器一样，通常使用单纯的寄存器结构。信息的存入一般采用电位—脉冲方式，即电位输入端对应数据信息位，脉冲输入端对应控制信号，在控制信号作用下，瞬时地将信息输入寄存器。

(5) 通用寄存器(R_0～R_3)。在我们的模型中，通用寄存器有 4 个(R_0～R_3)，其功能

是：当算术逻辑单元(ALU)执行算术或逻辑运算时，为 ALU 提供一个工作区。例如，在执行一次加法运算时，选择两个操作数(分别放在两个寄存器)相加，所得的结果送回其中一个寄存器(比如 R_2)中，而 R_2 中原有的内容随即被替换。

目前 CPU 中的通用寄存器，多达 64 个，甚至更多。其中任何一个可存放源操作数，也可存放结果操作数。在这种情况下，需要在指令格式中对寄存器号加以编址。从硬件结构来讲，需要使用通用寄存器堆结构，以便选择输入信息源。通用寄存器还用作地址指示器、变址寄存器、堆栈指示器等。

(6) 状态字寄存器(PSW)。状态字寄存器保存由算术指令和逻辑指令运算或测试结果建立的各种条件代码，如运算结果进位标志(C)，运算结果溢出标志(V)，运算结果为零标志(Z)，运算结果为负标志(N)等。这些标志位通常分别由 1 位触发器保存。

除此之外，状态条件寄存器还保存中断和系统工作状态等信息，以便使 CPU 和系统能及时了解机器运行状态和程序运行状态。因此，状态条件寄存器是一个由各种状态条件标志拼凑而成的寄存器。

从上面叙述可知，CPU 中的 6 类主要寄存器，每一类完成一种特定的功能。然而信息怎样才能在各寄存器之间传送呢？也就是说，数据的流动是由什么部件控制的呢？

通常把许多寄存器之间传送信息的通路称为数据通路。信息从什么地方开始，中间经过哪个寄存器或三态门，最后传送到哪个寄存器，都要加以控制。在各寄存器之间建立数据通路的任务，是由称为操作控制器的部件来完成的。操作控制器的功能，就是根据指令操作码和时序信号，产生各种操作控制信号，以便正确地选择数据通路，把有关数据输入一个寄存器中，从而完成取指令和执行指令的控制。

根据设计方法不同，操作控制器可分为时序逻辑型和存储逻辑型两种。第一种称为硬布线控制器，它是采用时序逻辑技术来实现的；第二种称为微程序控制器，它是采用存储逻辑来实现的。本书重点介绍微程序控制器。

操作控制器产生的控制信号必须定时，为此必须有时序产生器。因为计算机高速地进行工作，每一个动作的时间是非常严格的，不能太早也不能太迟。时序产生器的作用，就是对各种操作信号实施时间上的控制。

CPU 中除了上述组成部分外，还有中断系统、总线接口等其他功能部件，这些内容将在以后各章中陆续展开。

5.1.4 寄存器结构举例

不同计算机的 CPU 中，寄存器结构是不一样的，图 5.2 画出了 Zilog Z8000、Intel 8086 和 Motorola MC68000 三种微处理器的寄存器结构。

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做栈指针，寄存器可被用作 8 位和 32 位的运算。Zilog 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。

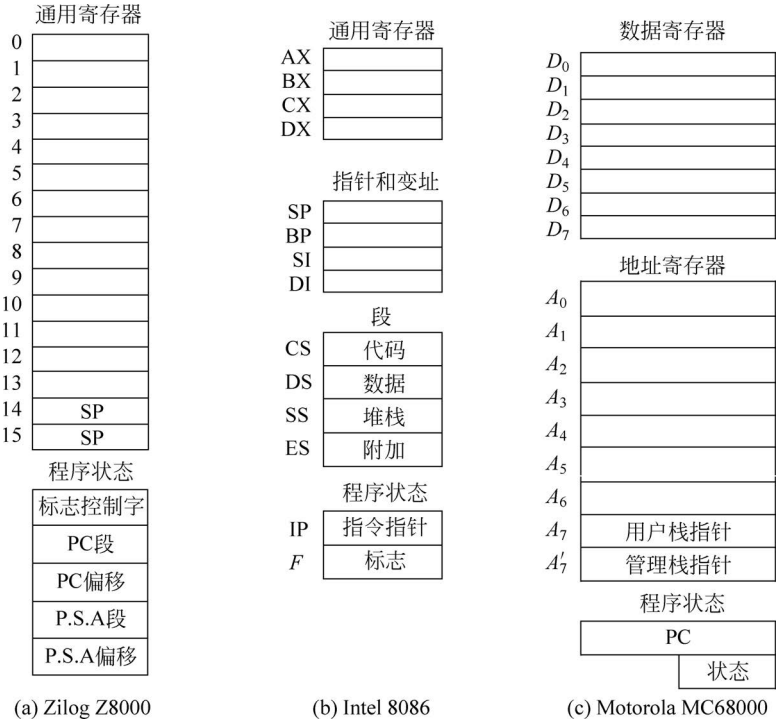


图 5.2 三种微处理器的寄存器结构

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(堆栈指针 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。Intel 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 (相当于程序计数器)和状态标志寄存器 F。

Motorola 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。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结构形式。图 5.3 就是 Zilog 8000 和 Intel 80386 的用户可见寄存器结构，它们分别是 Zilog Z8000 和 Intel 8086 的扩展，它们都采用 32 位寄存器，但又分别保留了原先一些特点。由于受这种限制，因此 32 位处理器在寄存器结构的设

计上只有有限的灵活性。

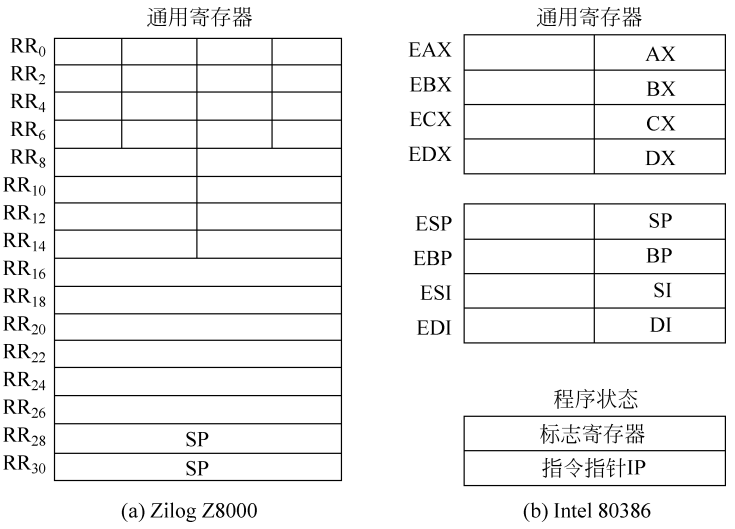


图 5.3 两种 32 位微处理器的寄存器结构

5.2 指令周期

5.2.1 指令周期的基本概念

我们知道,指令和数据从形式上看,它们都是二进制代码,所以对人来说,很难区分出这些代码是指令还是数据。然而 CPU 却能识别这些二进制代码:它能准确地判别出哪些是指令字,哪些是数据字,并将它们送往相应的地方。本节我们将讨论在一些典型的指令周期中,CPU 的各部分是怎样工作的,从而能加深对这一问题的理解和体验。

计算机之所以能自动地工作,是因为 CPU 能从存放程序的内存里取出一条指令并执行这条指令;紧接着又是取指令,执行指令……如此周而复始,构成了一个封闭的循环。除非遇到停机指令,否则这个循环将一直继续下去,其过程如图 5.4 所示。

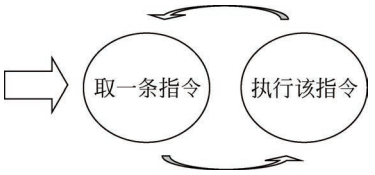


图 5.4 取指令-执行指令序列

CPU 每取出一条指令并执行这条指令,都要完成一系列的操作,这一系列操作所需的时间通常叫作一个指令周期。换言之指令周期是取出一条指令并执行这条指令的时间。由于各种指令的操作功能不同,因此各种指令的指令周期是不尽相同的。例如,一条加法指令的指令周期,同一条乘法指令的指令周期是不相同的。

指令周期常常用若干个 CPU 周期数来表示,CPU 周期也称为机器周期。由于 CPU 内部的操作速度较快,而 CPU 访问一次内存所花的时间较长,因此通常用内存中读取一个指令字的最短时间来规定 CPU 周期。这就是说,一条指令的取出阶段(通常称为取指)需要一个 CPU 周期时间。而一个周期时间又包含有若干个时钟周期(通常称为节拍脉冲或 T

周期，它是处理操作的最基本单位）。这些时钟周期的总和则规定了一个周期的时间宽度。图 5.5 示出了采用定长周期的指令周期示意图。从这个例子知道，取出和执行任何一条指令所需的最短时间为两个周期。就是说，任何一条指令，它的指令周期至少需要两个周期，而复杂一些的指令，可能需要更多的周期。

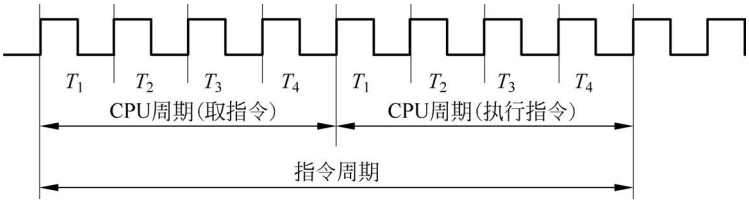


图 5.5　指令周期

表 5.1 列出了由 6 条指令组成的一个简单程序。这 6 条指令是有意安排的，因为它们是非常典型的，既有 RR 型指令，又有 RS 型指令；既有算术逻辑指令，又有访存指令，还有程序转移指令，信息存储采用哈佛结构。我们将在下面通过取出一条指令并执行这条指令的分解动作，来具体认识每条指令的指令周期。

表 5.1　6 条典型指令组成的一个简单程序

	八进制地址	指令助记符	说　明
指令存储器	100		1. 程序执行前(R_0)=00,(R_1)=10,(R_2)=20,(R_3)=30
	101	MOV R_0,R_1	2. 传送指令 MOV 执行(R_1)→R_0
	102	LAD R_1,6	3. 取数指令 LAD 从数存 6 号单元取数(100)
	103	ADD R_1,R_2	4. 加法指令 ADD 执行(R_1)+(R_2)→R_2,结果为(R_2)=120
	104	STO R_2,(R_3)	5. 取数指令 STO 用(R_3)间接寻址,(R_2)=120 写入数存 30 号单元
	105	JMP 101	6. 转移指令 JMP 改变程序执行顺序到 101 号单元
	106	AND R_1,R_3	7. 逻辑乘 AND 指令执行(R_1)·(R_3)→R_3
	八进制地址	八进制数据	说　明
数据存储器	5	70	
	6	100	执行 LAD 指令后，数存 6 号单元的数据 100 仍保存在其中
	7	66	
	10	77	
	…	…	
	30	40(120)	执行 STO 指令后，数存 30 号单元的数据由 40 变为 120

5.2.2　MOV 指令的指令周期

MOV 是一条 RR 型指令，其指令周期如图 5.6 所示。它需要两个 CPU 周期，其中取指周期需要一个 CPU 周期，执行周期需要一个 CPU 周期。

取指周期中 CPU 完成三件事：①从指存取出指令；②对程序计数器 PC 加 1，以便为取下一条指令做好准备；③对指令操作码进行译码或测试，以便确定进行什么操作。

执行周期中 CPU 根据对指令操作码的译码或测试，进行指令所要求的操作。对 MOV 指令来说，执行周期中完成两个通用寄存器 R_0、R_1 之间的数据传送操作。由于时间充足，执行周期一般只需要一个 CPU 周期。

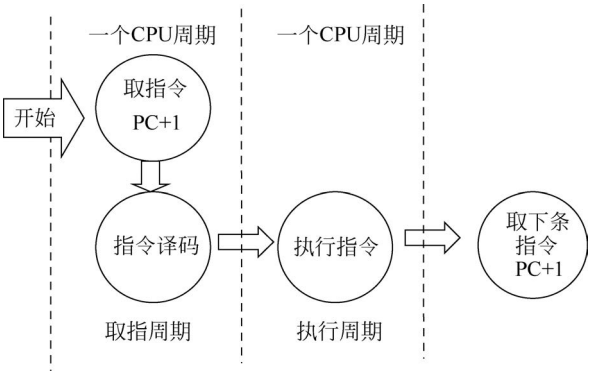


图 5.6 MOV 指令的指令周期

1. 取指周期

第一条指令的取指周期示于图 5.7。我们假定表 5.1 的程序已装入指存中,因而在此阶段内,CPU 的动作如下:

(1) 程序计数器中装入第一条指令地址 101(八进制)。

(2) 程序计数器的内容被放到指令地址总线 ABUS(I)上,对指存进行译码,并启动读命令。

(3) 从 101 号地址读出的 MOV 指令通过指令总线(IBUS)装入指令寄存器 IR。

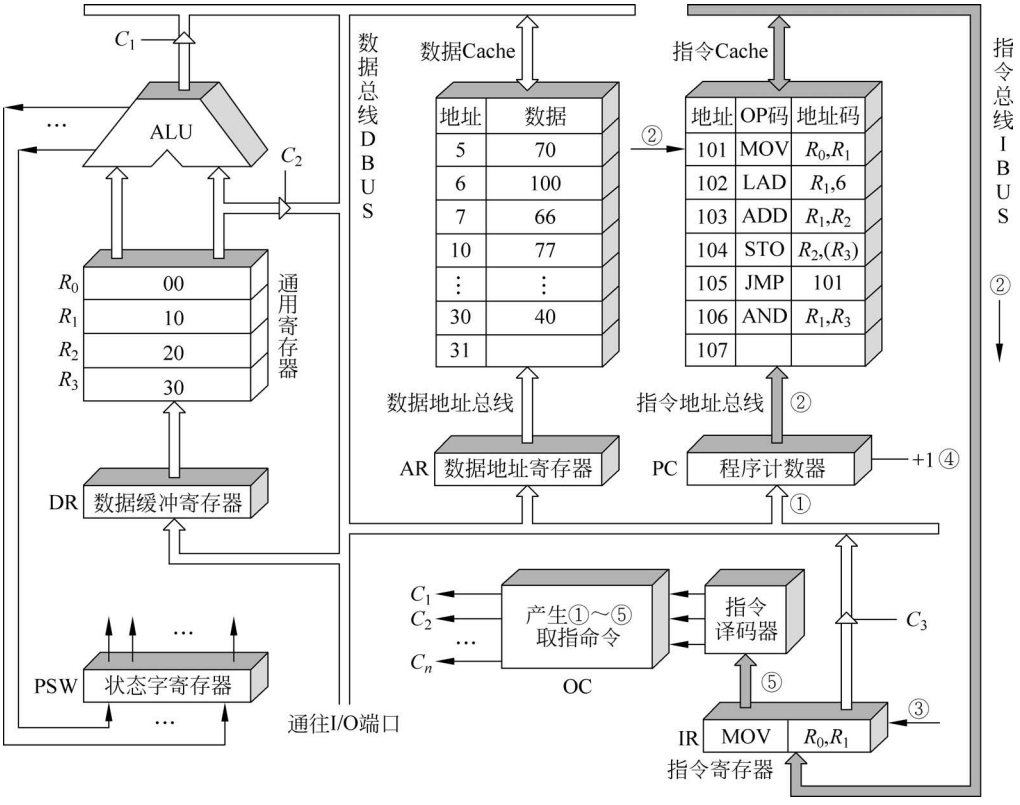


图 5.7 MOV 指令取指周期

(4) 程序计数器内容加 1,变成 102,为取下一条指令做好准备。

(5) 指令寄存器中的操作码(OP)被译码。

(6) CPU 识别出是 MOV 指令,至此,取指周期即告结束。

2. 执行指令阶段(执行周期)

MOV 指令的执行周期示于图 5.8 中,在此阶段,CPU 的动作如下:

(1) 操作控制器(OC)送出控制信号到通用寄存器,选择 R_1(10)作源寄存器,选择 R_0 作目标寄存器。

(2) OC 送出控制信号到 ALU,指定 ALU 做传送操作。

(3) OC 送出控制信号,打开 ALU 输出三态门,将 ALU 输出送到数据总线(DBUS)上。注意,任何时候 DBUS 上只能有一个数据。

(4) OC 送出控制信号,将 DBUS 上的数据输入数据缓冲寄存器 DR(10)中。

(5) OC 送出控制信号,将 DR 中的数据 10 输入目标寄存器 R_0 中,R_0 的内容由 00 变为 10。至此,MOV 指令执行结束。

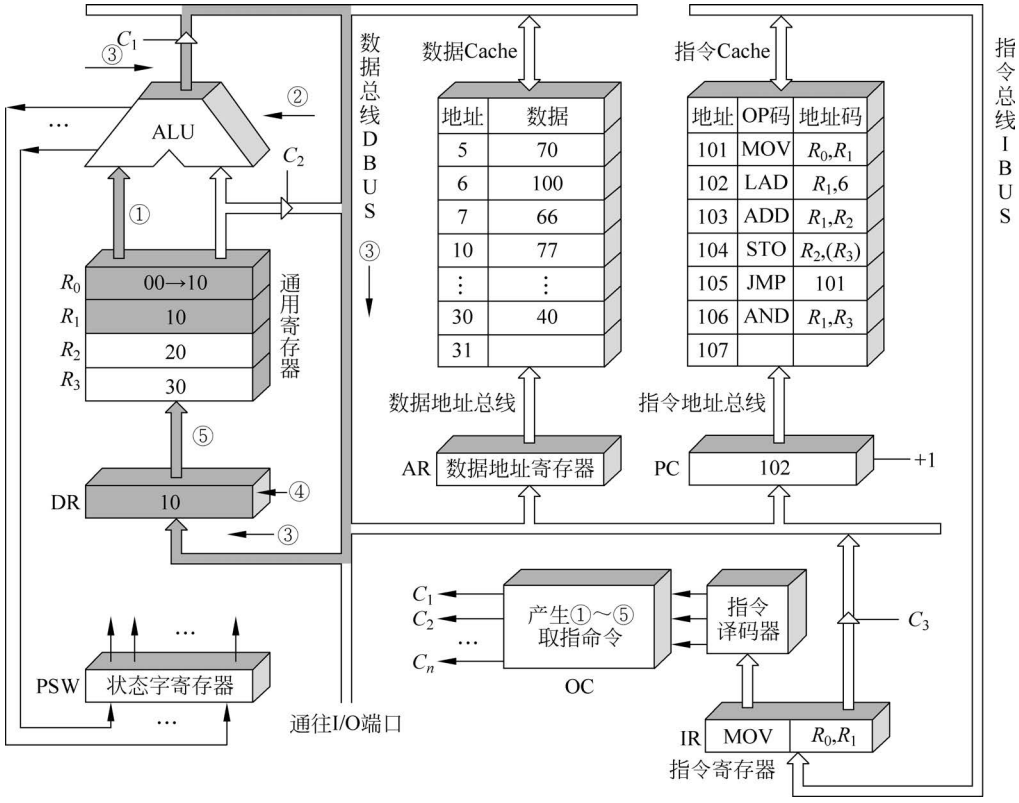


图 5.8　MOV 指令执行周期

5.2.3　LAD 指令的指令周期

LAD 指令是 RS 型指令,它先从指令存储器取出指令,然后从数据存储器 6 号单元取出数据 100 装入通用寄存器 R_1,原来 R_1 中存放的数据 10 被更换成 100。由于一次访问指

存,一次访问数存,LAD 指令的指令周期需要 3 个 CPU 周期,如图 5.9 所示。

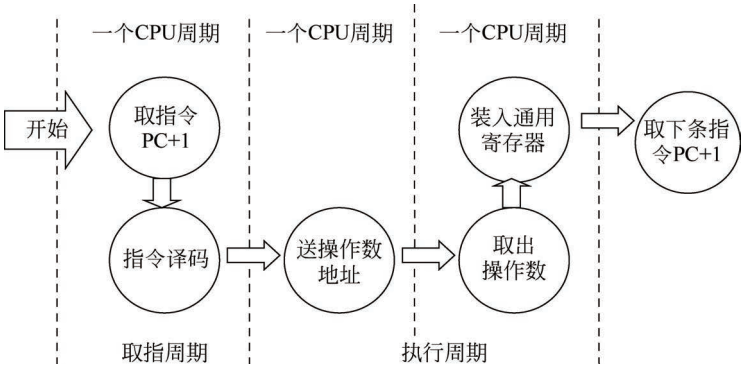


图 5.9 LAD 指令的指令周期

1. 取指周期

在 LAD 指令的取指周期中,CPU 的动作完全与 MOV 指令取指周期中一样(如图 5.7),只是 PC 提供的指令地址为 102,按此地址从指令存储器读出"LDA R_1,6"指令放入 IR 中,然后将 PC+1,使 PC 内容变成 103,为取下条 ADD 指令做好准备。

以下 ADD、STO、JMP 三条指令的取指周期中,CPU 的动作完全与 MOV 指令一样,不再赘述。

2. 执行周期

LAD 指令的执行周期如图 5.10 所示。CPU 执行的动作如下:

(1) 操作控制器(OC)发出控制命令打开 IR 输出三态门,将指令中的直接地址码 6 放到数据总线(DBUS)上。

(2) OC 发出操作命令,将地址码 6 装入数存地址寄存器(AR)。

(3) OC 发出读命令,将数据存储器 6 号单元中的数 100 读出到 DBUS 上。

(4) OC 发出命令,将 DBUS 上的数据 100 装入缓冲寄存器(DR)。

(5) OC 发出命令,将 DR 中的数 100 装入通用寄存器 R_1,原来 R_1 中的数 10 被冲掉。至此,LAD 指令执行周期结束。

注意,数据总线(DBUS)上分时进行了地址传送和数据传送,所以需要 2 个 CPU 周期。

5.2.4 ADD 指令的指令周期

ADD 指令是 RR 型指令,在运算器中用两个寄存器 R_1 和 R_2 的数据进行加法运算。指令周期只需两个 CPU 周期,其中一个是取指周期,与图 5.7 相同。下面只讲执行周期,CPU 完成的动作如图 5.11 所示。

(1) 操作控制器(OC)送出控制命令到通用寄存器,选择 R_1 做源寄存器,R_2 做目标寄存器。

(2) OC 送出控制命令到 ALU,指定 ALU 做 R_1(100)和 R_2(20)的加法操作。

(3) OC 送出控制命令,打开 ALU 输出三态门,运算结果 ALU 放到 DBUS 上。

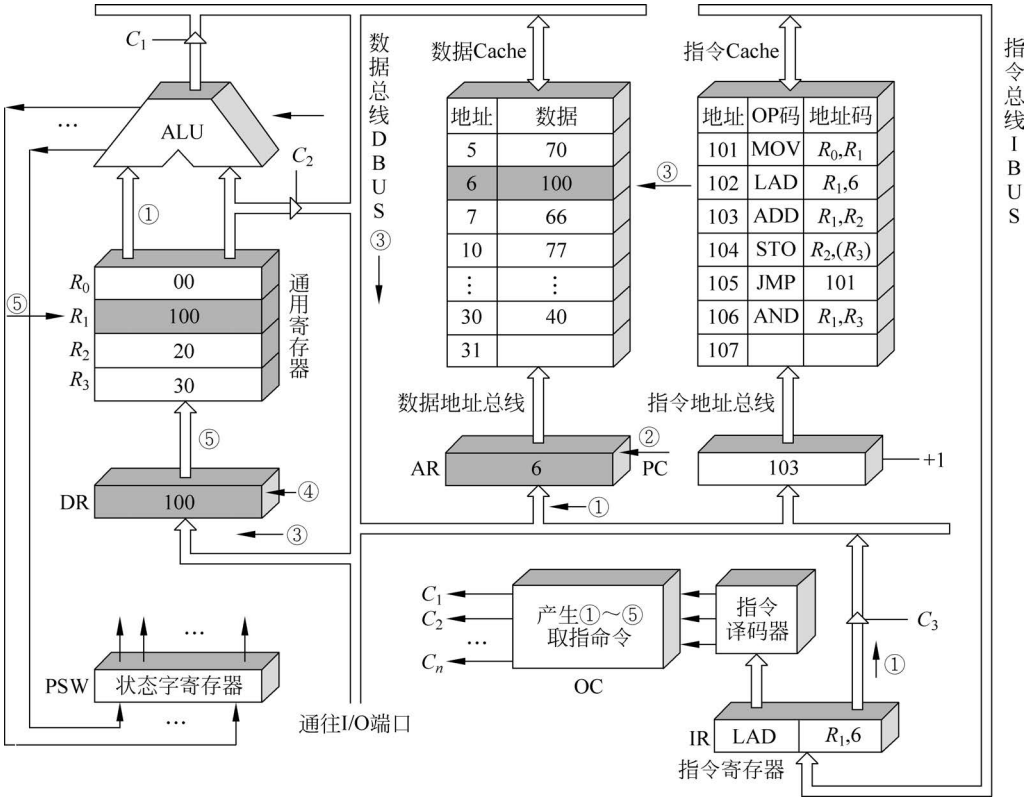


图 5.10　LAD 指令的执行周期

(4) OC 送出控制命令,将 DBUS 上数据输入缓冲寄存器 DR 中,ALU 产生的进位信号保存在状态字寄存器(PSW)中。

(5) OC 送出控制命令,将 DR(120)装入 R_2,R_2 原来的内容 20 被冲掉。

5.2.5　STO 指令的指令周期

STO 指令是 RS 型指令,它先访问指存取出 STO 指令,然后按(R_3)=30 地址访问数存,将(R_2)=120 写入 30 号单元。由于一次访问指存,一次访问数存,因此指令周期需 3 个 CPU 周期,其中执行周期为 2 个 CPU 周期,如图 5.12 所示。

完成的动作如图 5.13 所示。

(1) 操作控制器(OC)送出操作命令到通用寄存器,选择(R_3)=30 做数据存储器的地址单元。

(2) OC 发出操作命令,打开通用寄存器输出三态门(不经 ALU 以节省时间),将地址 30 放到 DBUS 上。

(3) OC 发出操作命令,将地址 30 输入 AR 中并进行数据存储器地址译码。

(4) OC 发出操作命令到通用寄存器,选择(R_2)=120,作为数据存储器的写入数据。

(5) OC 发出操作命令,打开通用寄存器输出三态门,将数据 120 放到 DBUS 上。

(6) OC 发出操作命令,数据写入数据存储器 30 号单元,它原先的数据 40 被冲掉。至此,STO 指令执行周期结束。

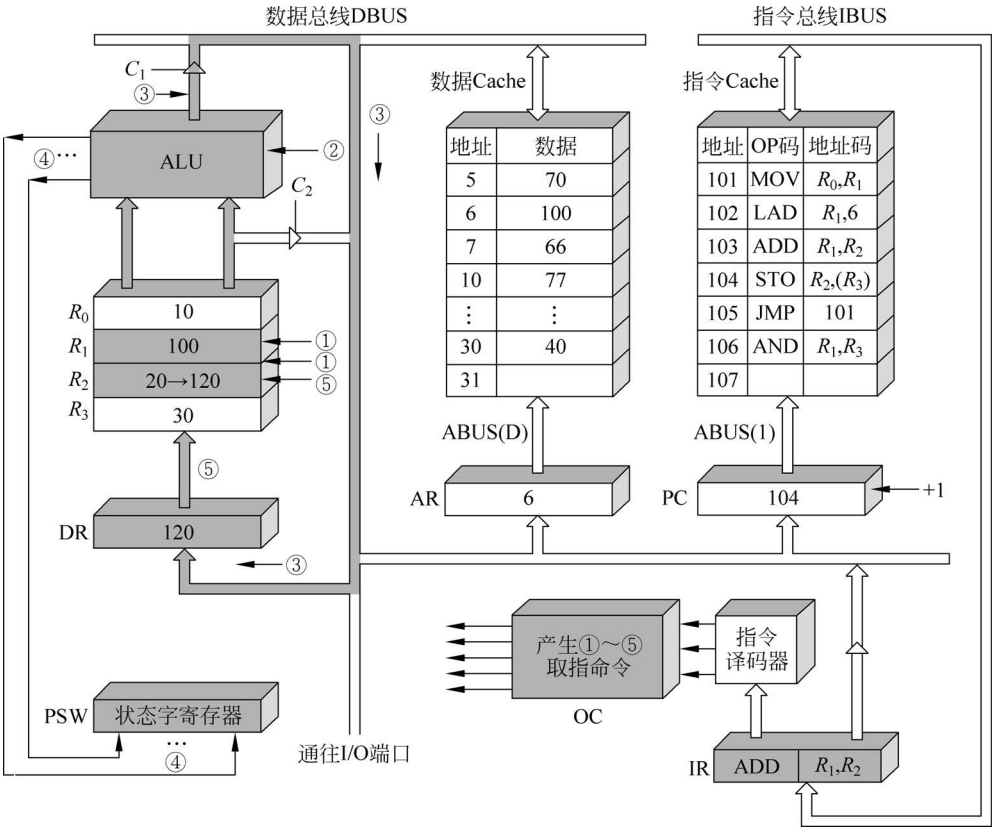


图 5.11 ADD 指令执行周期

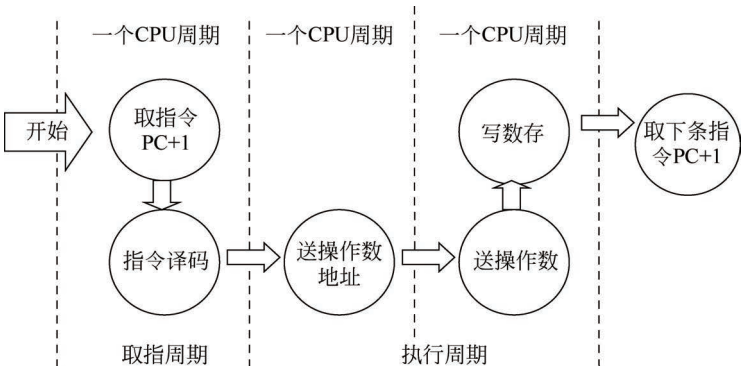


图 5.12 STO 指令的指令周期

注意,DBUS 是单总线结构,先送地址(30),后送数据(120),必须分时传送。

5.2.6 JMP 指令的指令周期

JMP 指令是一条无条件转移指令,用来改变程序的执行顺序。指令周期为两个 CPU 周期,其中取指周期为 1 个 CPU 周期,执行周期为 1 个 CPU 周期。CPU 完成的动作如图 5.14 所示。

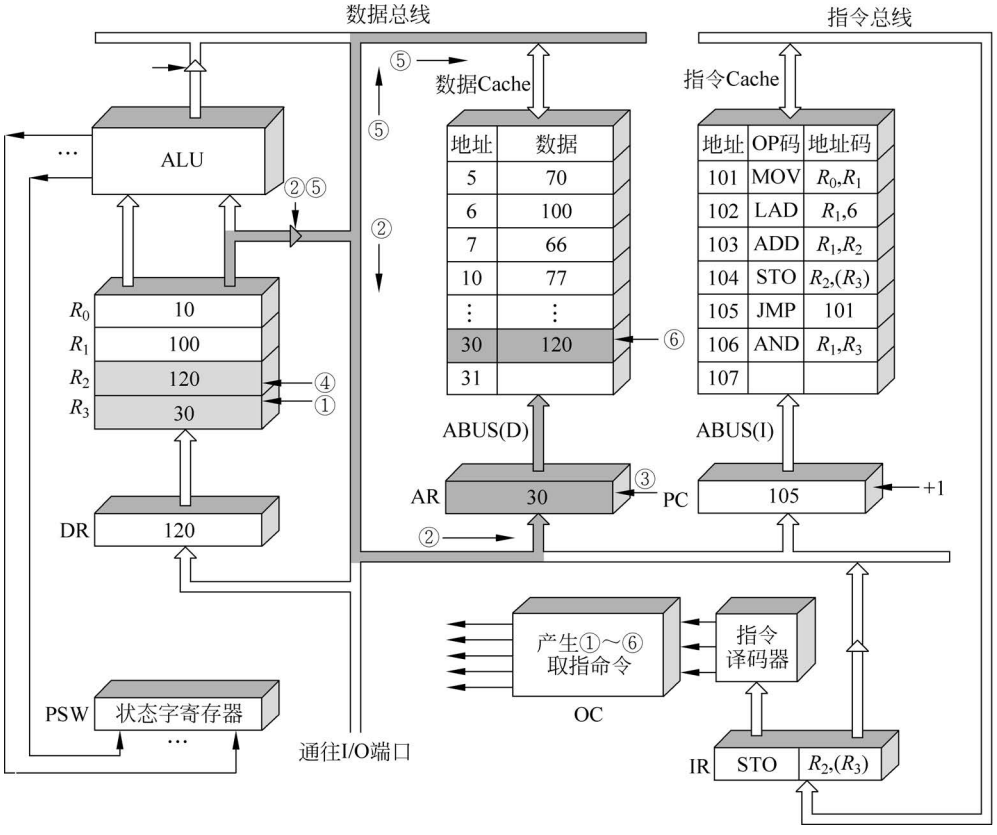


图 5.13　STO 指令执行周期

(1) OC 发生操作控制命令,打开指令寄存器(IR)的输出三态门,将 IR 中的地址码 101 发送到 DBUS 上。

(2) OC 发出操作控制命令,将 DBUS 上的地址码 101 输入程序计数器(PC)中,PC 中的原先内容 106 被更换。于是下一条指令不是从 106 号单元取出,而是转移到 101 号单元取出。至此 JMP 指令执行周期结束。

应当指出,执行"JMP　101"指令时,我们此处所给的五条指令组成的程序进入了死循环,除非人为停机,否则这个程序将无休止地运行下去。当然,我们此处所举的转移地址 101 是随意的,仅仅用来说明转移指令能够改变程序的执行顺序而已。

5.2.7　用方框图语言表示指令周期

我们在上面介绍了 5 条典型指令的指令周期,从而使我们对一条指令的取指过程和执行过程有了一个较深刻的印象。然而我们是通过画示意图或数据通路图来解释这些过程的。之所以这样做,主要是为了教学的目的。但是在进行计算机设计时,如果用这种办法来表示指令周期,那就显得过于繁琐,而且也没有这种必要。

在进行计算机设计时,可以采用方框图语言来表示指令的指令周期。一个方框代表一个 CPU 周期,方框中的内容表示数据通路的操作或某种控制操作。除了方框以外,还需要

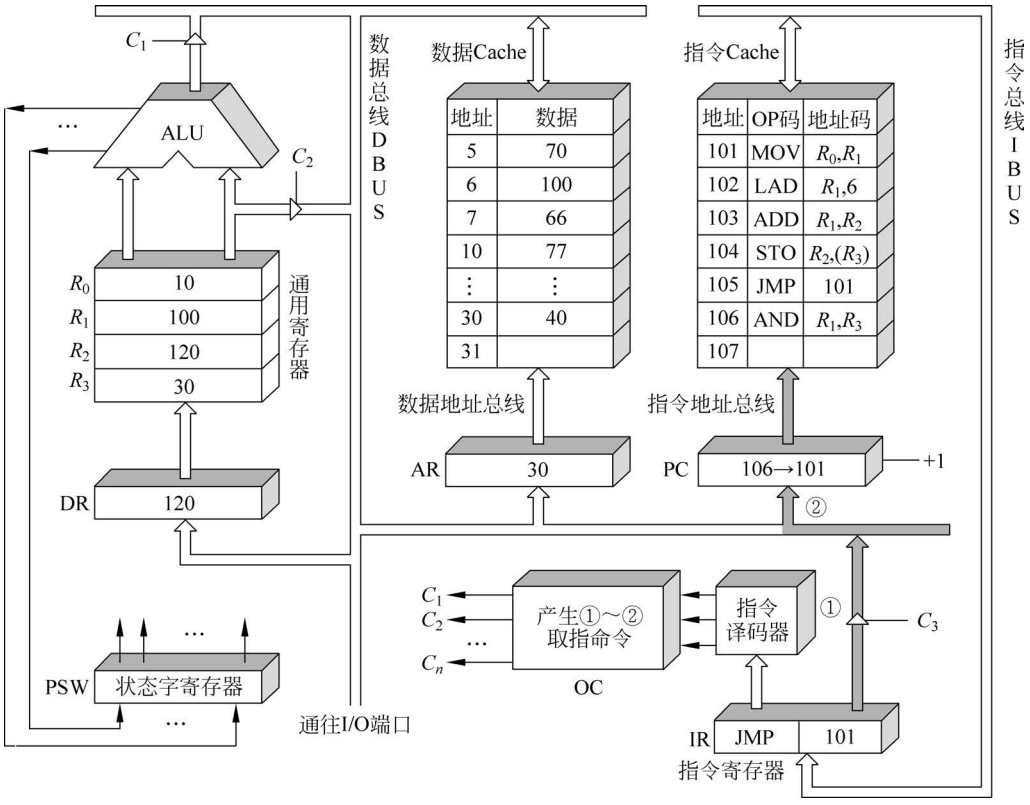


图 5.14 JMP 指令执行周期

一个菱形符号,它通常用来表示某种判别或测试,不过时间上它依附于紧接它的前面一个方框的 CPU 周期,而不单独占用一个 CPU 周期。

我们把前面的 5 条典型指令加以归纳,用方框图语言表示的指令周期示于图 5.15。

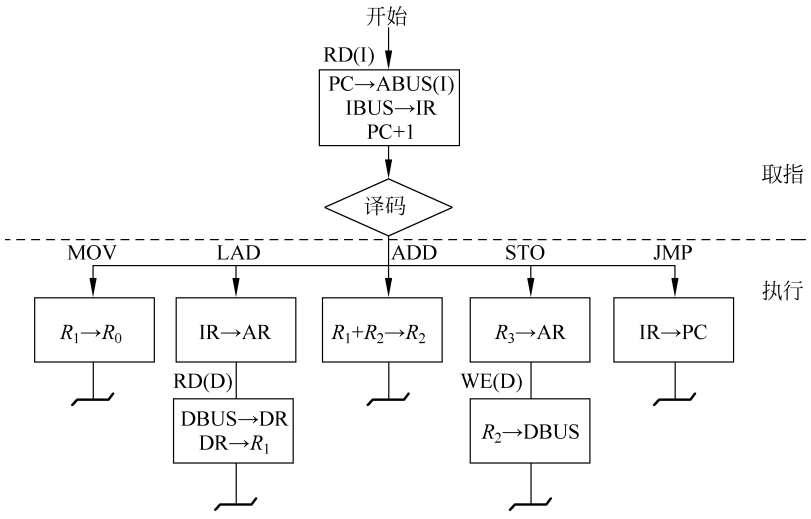


图 5.15 用方框图语言表示指令周期

我们明显地看到，所有指令的取指周期是完全相同的，而且是一个 CPU 周期。但是指令的执行周期，由于各条指令的功能不同，所用的执行周期是各不相同的，其中 MOV、ADD、JMP 指令是一个 CPU 周期；LAD 和 STO 指令是两个 CPU 周期。框图中 DBUS 代表数据总线，ABUS(D)代表数存地址总线，ABUS(I)代表指存地址总线，RD(D)代表数存读命令，WE(D)代表数存写命令，RD(I)代表指存读命令。

图 5.15 中，还有一个“～”符号，我们称它为公操作符号。这个符号表示一条指令已经执行完毕，转入公操作。所谓公操作，就是一条指令执行完毕后，CPU 所开始进行的一些操作，这些操作主要是 CPU 对外围设备请求的处理，如中断处理、通道处理等。如果外围设备没有向 CPU 请求交换数据，那么 CPU 又转向指存取下一条指令。由于所有指令的取指周期是完全一样的，因此，取指令也可认为是公操作。这是因为，一条指令执行结束后，如果没有外设请求，一定转入“取指令”操作。

【例 5.1】 图 5.16 所示为双总线结构机器的数据通路，IR 为指令寄存器，PC 为程序计数器(具有自增功能)，M 为主存(受 R/$\overline{\text{W}}$ 信号控制)，它既存放指令又存放数据，AR 为地址寄存器，DR 为数据缓冲寄存器，ALU 由加、减控制信号决定完成何种操作，控制信号 G 控制的是一个门电路，它相当于两条总线之间的桥。另外，线上标注有小圈表示有控制信号，例中 y_i 表示 Y 寄存器的输入控制信号，R_{1O} 为寄存器 R_1 的输出控制信号，未标字符的线为直通线，不受控制。

(1) “ADD　R_2,R_0”指令完成$(R_2)+(R_0)\rightarrow R_0$ 的功能操作，画出其指令周期流程图，假设该指令的地址已放入 PC 中。并列出相应的微操作控制信号序列。

(2) “SUB　R_1,R_3”指令完成$(R_3)-(R_1)\rightarrow R_3$ 的功能操作，画出其指令周期流程图，并列出相应的微操作控制信号序列。

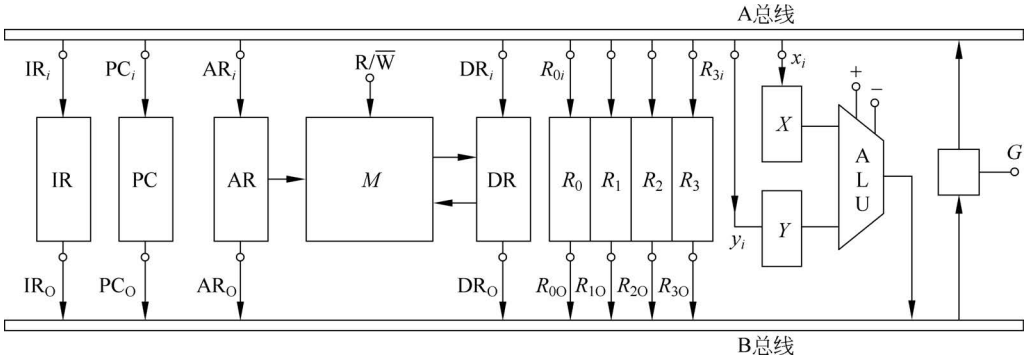


图 5.16　双总线结构机器的数据通路

解：①“ADD　R_2,R_0”指令是一条加法指令，参与运算的两个数放在寄存器 R_2 和 R_0 中，指令周期流程图包括取指令阶段和执行指令阶段两部分(为简单起见，省去了“→”号和左边各寄存器代码上应加的括号)。根据给定的数据通路图，“ADD　R_2,R_0”指令的详细指令周期流程图如图 5.17(a)所示，图的右边部分标注了每一个机器周期中用到的微操作控制信号序列。②SUB 减法指令周期流程图如图 5.17(b)所示。

思考题　为了缩短“ADD　R_2,R_0”指令的取值周期，修改图 5.17 数据通路，在此基础上画出该指令周期流程图。

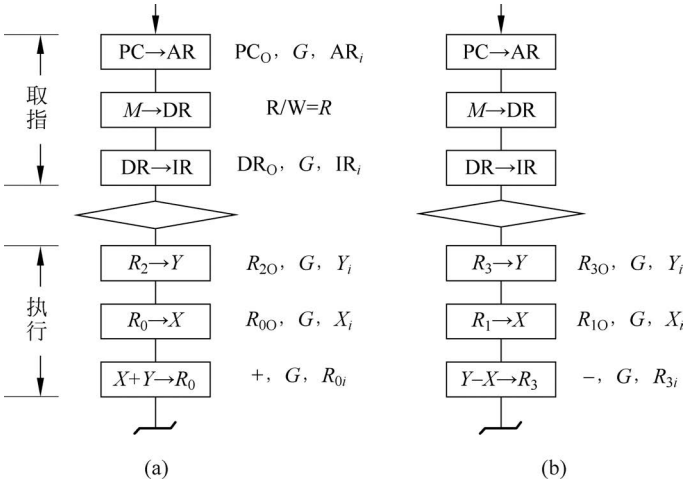


图 5.17 加法和减法指令周期流程

5.3 时序产生器和控制方式

5.3.1 时序信号的作用和体制

在日常生活中,我们学习、工作和休息都有一个严格的作息时间。比如早晨 6:00 起床;8:00～12:00 上课,12:00～14:00 午休……每个教师和学生都必须严格遵守这一规定,在规定的时间里上课,在规定的时间里休息,否则就难以保证正常的教学秩序。

CPU 中也有一个类似“作息时间”的东西,它称为时序信号。计算机之所以能够准确、迅速、有条不紊地工作,正是因为在 CPU 中有一个时序信号产生器。机器一旦被启动,即 CPU 开始取指令并执行指令时,操作控制器就利用定时脉冲的顺序和不同的脉冲间隔,有条理、有节奏地指挥机器的动作,规定在这个脉冲到来时做什么,在那个脉冲到来时又做什么,给计算机各部分提供工作所需的时间标志。为此,需要采用多级时序体制。

让我们再来考虑上一节中提出的一个问题:用二进制码表示的指令和数据都放在内存里,那么 CPU 是怎样识别出它们是数据还是指令呢?事实上,我们通过上一节讲述指令周期后,就自然会得出如下结论:从时间上来说,取指令事件发生在指令周期的第一个 CPU 周期中,即发生在“取指令”阶段,而取数据事件发生在“执行指令”阶段。从空间上来说,如果取出的代码是指令,那么一定送往指令寄存器,如果取出的代码是数据,那么一定送往运算器。由此可见,时间控制对计算机来说是太重要了。

不仅如此,在一个 CPU 周期中,又把时间分为若干个小段,以便规定在这一小段时间中 CPU 干什么,在那一小段时间中 CPU 又干什么,这种时间约束对 CPU 来说是非常必要的,否则就可能造成丢失信息或导致错误的结果。因为时间的约束是如此严格,以至于时间进度既不能来得太早,也不能来得太晚。

总之,计算机的协调动作需要时间标志,而时间标志则是用时序信号来体现的。一般来说,操作控制器发出的各种控制信号都是时间因素(时序信号)和空间因素(部件位置)的函

数。如果忽略了时间因素，那么我们学习计算机硬件时往往就会感到困难，这一点请读者加以注意。

组成计算机硬件的器件特性决定了时序信号最基本的体制是电位—脉冲制。这种体制最明显的一个例子，就是当实现寄存器之间的数据传送时，数据加在触发器的电位输入端，而输入数据的控制信号加在触发器的时钟输入端。电位的高低，表示数据是 1 还是 0，而且要求输入数据的控制信号到来之前，电位信号必须已稳定。这是因为，只有电位信号先建立，输入寄存器中的数据才是可靠的。当然，计算机中有些部件，例如算术逻辑运算单元只用电位信号工作就可以了。但尽管如此，运算结果还是要送入通用寄存器，所以最终还是需要脉冲信号来配合。

硬布线控制器中，时序信号往往采用主状态周期—节拍电位—节拍脉冲三级体制。一个节拍电位表示一个 CPU 周期的时间，它表示了一个较大的时间单位；在一个节拍电位中又包含若干个节拍脉冲，以表示较小的时间单位；而主状态周期可包含若干个节拍电位，所以它是最大的时间单位。主状态周期可以用一个触发器的状态持续时间来表示。

在微程序控制器中，时序信号比较简单，一般采用节拍电位—节拍脉冲二级体制。就是说，它只有一个节拍电位，在节拍电位中又包含若干个节拍脉冲(时钟周期)。节拍电位表示一个 CPU 周期的时间，而节拍脉冲把一个 CPU 周期划分成几个较小的时间间隔。根据需要，这些时间间隔可以相等，也可以不相等。

5.3.2　时序信号产生器

前面我们已分析了指令周期中需要的一些典型时序。时序信号产生器的功能是用逻辑电路来实现这些时序。

各种计算机的时序信号产生电路是不尽相同的。一般来说，大型计算机的时序电路比较复杂，而微型机的时序电路比较简单，这是因为前者涉及的操作动作较多，后者涉及的操作动作较少。另一方面，从设计操作控制器的方法来讲，硬连线控制器的时序电路比较复杂，而微程序控制器的时序电路比较简单。然而不管是哪一类，时序信号产生器最基本的构成是一样的。

图 5.18 示出了微程序控制器中使用的时序信号产生器的结构图，它由时钟源、环形脉冲发生器、节拍脉冲和读写时序译码逻辑、启停控制逻辑等部分组成。

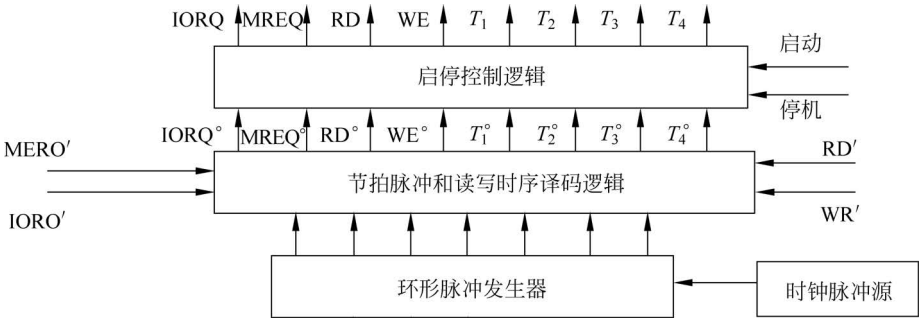


图 5.18　时序信号产生器框图

1. 时钟源

时钟源用来为环形脉冲发生器提供频率稳定且电平匹配的方波时钟脉冲信号。它通常由石英晶体振荡器和与非门组成的正反馈振荡电路组成,其输出送至环形脉冲发生器。

2. 环形脉冲发生器

环形脉冲发生器的作用是产生一组有序的间隔相等或不等的脉冲序列,以便通过译码电路来产生最后所需的节拍脉冲,其启停控制的电路如图 5.19 所示。

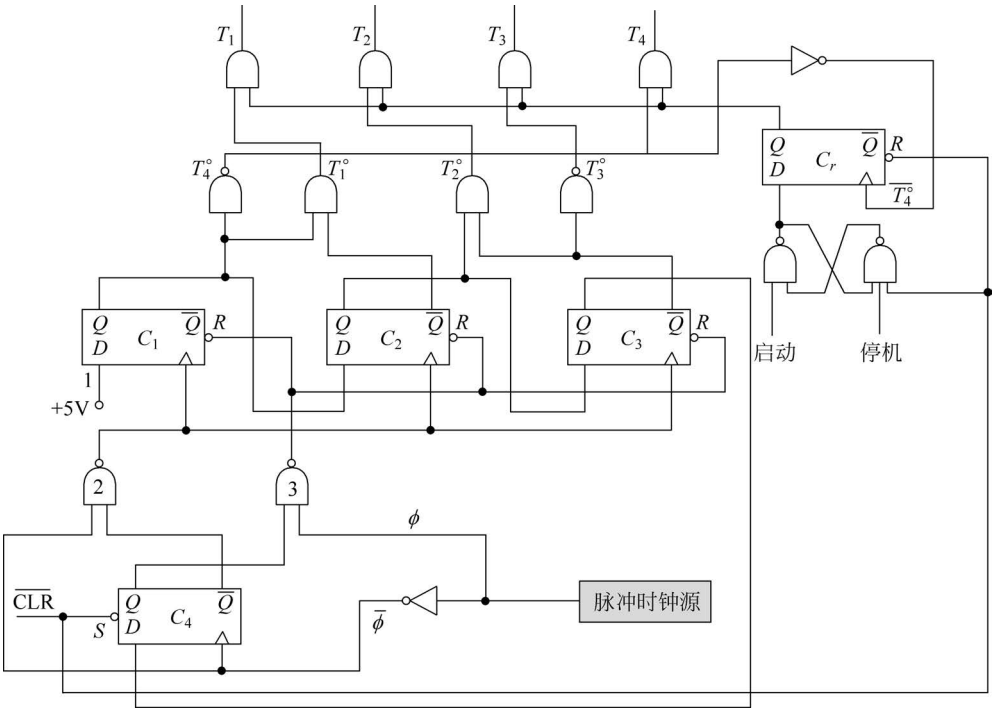


图 5.19 环形脉冲发生器

3. 节拍脉冲和存储器读/写时序

我们假定在一个 CPU 周期中产生四个等间隔的节拍脉冲 $T_1^\circ \sim T_4^\circ$,每个节拍脉冲的脉冲宽度均为 200ns,因此一个 CPU 周期便是 800ns,在下一个 CPU 周期中,它们又按固定的时间关系重复。不过注意,图 5.19 中画出的节拍脉冲信号是 $T_1 \sim T_4$,它们在逻辑关系上与 $T_1^\circ \sim T_4^\circ$是完全一致的,是后者经过启停控制逻辑中与门以后的输出,图中忽略了一级与门的时间延迟细节。

存储器读/写时序信号 RD°、WE°用来进行存储器的读/写操作。

在硬连线控制器中,节拍电位信号是由时序产生器本身通过逻辑电路产生的,一个节拍电位持续时间正好包含若干个节拍脉冲。然而在微程序设计的计算机中,节拍电位信号可由微程序控制器提供。一个节拍电位持续时间,通常也是一个 CPU 周期时间。例如图 5.20 中的 RD′、WE′信号持续时间均为 800ns,而一个 CPU 周期也正好是 800ns。关于

微程序控制器如何产生节拍电位信号，将留在下一节介绍。

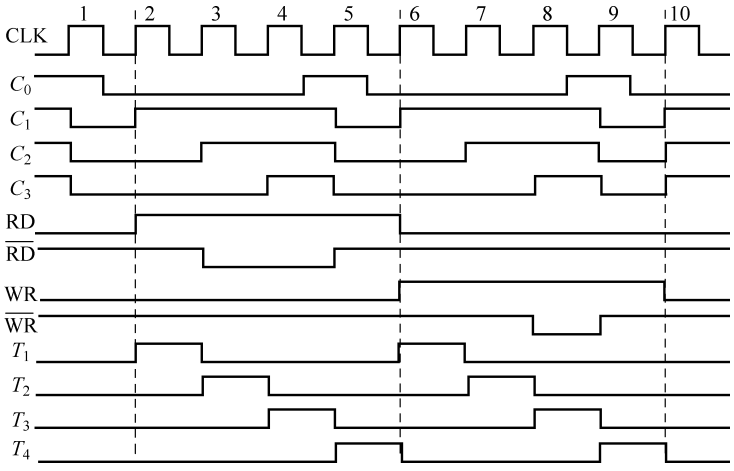


图 5.20　节拍电位与节拍脉冲时序关系图

4. 启停控制逻辑

机器一旦接通电源，就会自动产生原始的节拍脉冲信号 $T_1^\circ \sim T_4^\circ$，然而，只有在启动机器运行的情况下，才允许时序产生器发出 CPU 工作所需的节拍脉冲 $T_1 \sim T_4$。为此需要由启停控制逻辑来控制 $T_1^\circ \sim T_4^\circ$ 的发送。同样，对读/写时序信号也需要由启停逻辑加以控制。图 5.21 给出了启停控制逻辑电路。

启停控制逻辑的核心是一个运行标志触发器 C_r。当运行触发器为“1”时，原始节拍脉冲 $T_1^\circ \sim T_4^\circ$ 和读/写时序信号 RD°、WE°通过门电路发送出去，变成 CPU 真正需要的节拍脉冲信号 $T_1 \sim T_4$ 和读/写时序信号 $\overline{RD}$、$\overline{WE}$。反之，当运行触发器为“0”时，就关闭时序产生器。

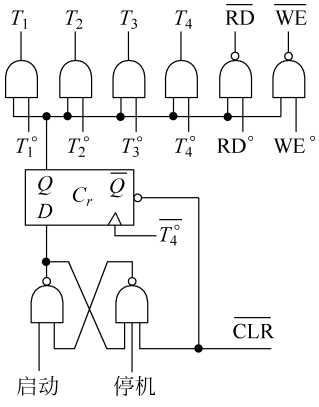


图 5.21　启停控制逻辑

由于启动计算机是随机的，停机也是随机的，为此必须要求：当计算机启动时，一定要从第 1 个节拍脉冲前沿开始工作，而在停机时一定要在第 4 个节拍脉冲结束后关闭时序产生器。只有这样，才能使发送出去的脉冲都是完整的脉冲。图 5.21 中，在 C_r（D 触发器）下面加上一个 RS 触发器，且用 $\overline{T_4^\circ}$ 信号作 C_r 触发器的时钟控制端，那么就可以保证在 T_1 的前沿开启时序产生器，而在 T_4 的后沿关闭时序产生器。

5.3.3　控制方式

从 5.2 节知道，机器指令的指令周期是由数目不等的 CPU 周期组成，CPU 周期的多少反映了指令动作的复杂程度，即操作控制信号的多少。对一个 CPU 周期而言，也有操作控制信号的多少与出现的先后问题。这两种情况综合在一起，说明每条指令和每个操作控制信号所需的时间各不相同。控制不同操作序列时序信号的方法，称为控制器的控制方式。常用的有同步控制、异步控制、联合控制三种方式，其实质反映了时序信号的定时方式。

1. 同步控制方式

在任何情况下,指令在执行时所需的机器周期数和时钟周期数都是固定不变的,称为同步控制方式。根据不同情况,同步控制方式可选取如下方案:

(1) 采用完全统一的机器周期执行各种不同的指令。这意味着所有指令周期具有相同的节拍电位数和相同的节拍脉冲数。显然,对简单指令和简单的操作来说,将造成时间浪费。

(2) 采用不定长机器周期。将大多数操作安排在一个较短的机器周期内完成,对某些时间紧张的操作,则采取延长机器周期的办法来解决。

(3) 中央控制与局部控制结合。将大部分指令安排在固定的机器周期完成,称为中央控制,对少数复杂指令(乘、除、浮点运算)采用另外的时序进行定时,称为局部控制。

2. 异步控制方式

异步控制方式的特点是:每条指令、每个操作控制信号需要多少时间就占用多少时间。这意味着每条指令的指令周期可由多少不等的机器周期组成;也可以是当控制器发出某一操作控制信号后,等待执行部件完成操作后发回"回答"信号,再开始新的操作。显然,用这种方式形成的操作控制序列没有固定的 CPU 周期数(节拍电位)或严格的时钟周期(节拍脉冲)与之同步。

3. 联合控制方式

此为同步控制和异步控制相结合的方式。一种情况是,大部分操作序列安排在固定的机器周期中,对某些时间难以确定的操作则以执行部件的"回答"信号作为本次操作的结束。例如 CPU 访问主存时,依靠其送来的"READY"信号作为读/写周期的结束。另一种情况是,机器周期的节拍脉冲数固定,但是各条指令周期的机器周期数不固定。

5.4 微程序控制器

5.4.1 微程序控制原理

微程序控制器同硬布线控制器相比较,具有规整性、灵活性、可维护性等一系列优点,因而在计算机设计中逐渐取代了早期采用的硬布线控制器,并已被广泛地应用。在计算机系统中,微程序设计技术是利用软件方法来设计硬件的一门技术。

微程序控制的基本思想,就是仿照通常的解题程序的方法,把操作控制信号编成所谓的"微指令",存放到一个只读存储器里。当机器运行时,一条又一条地读出这些微指令,从而产生全机所需要的各种操作控制信号,使相应部件执行所规定的操作。

1. 微命令和微操作

一台数字计算机基本上可以划分为两大部分——控制部件和执行部件。控制器就是控制部件,而运算器、存储器、外围设备相对控制器来讲,就是执行部件。那么两者之间是怎样

进行联系的呢？

控制部件与执行部件的一种联系，就是通过控制线。控制部件通过控制线向执行部件发出各种控制命令，通常把这种控制命令叫作微命令，而执行部件接受微命令后所进行的操作，叫作微操作。

控制部件与执行部件之间的另一种联系是反馈信息。执行部件通过反馈线向控制部件反映操作情况，以便使控制部件根据执行部件的“状态”来下达新的微命令，这也叫作“状态测试”。

微操作在执行部件中是最基本的操作。由于数据通路的结构关系，微操作可分为相容性和相斥性两种。所谓相容性的微操作，是指在同时或同一个 CPU 周期内可以并行执行的微操作。所谓相斥性的微操作，是指不能在同时或不能在同一个 CPU 周期内并行执行的微操作。

图 5.22 示出了一个简单运算器模型，其中 ALU 为算术逻辑单元，R_1、R_2、R_3 为三个寄存器。三个寄存器的内容都可以通过多路开关从 ALU 的 X 输入端或 Y 输入端送至 ALU，而 ALU 的输出可以送往任何一个寄存器或同时送往 R_1、R_2、R_3 三个寄存器。在我们给定的数据通路中，多路开关的每个控制门仅是一个常闭的开关，它的一个输入端代表来自寄存器的信息，而另一个输入端则作为操作控制端。一旦两个输入端都有输入信号时，它才产生一个输出信号，从而在控制线能起作用的一个时间宽度中来控制信息在部件中流动。图中每个开关门由控制器中相应的微命令来控制，例如，开关门 4 由控制器中编号为 4 的微命令控制，开关门 6 由编号为 6 的微命令控制等。三个寄存器 R_1、R_2、R_3 的时钟输入端 1、2、3 也需要加以控制，以便在 ALU 运算完毕而输出公共总线上电平稳定时，将结果输入某一寄存器中。另外，我们假定 ALU 只有＋、－、M(传送)三种操作。C_y 为最高进位触发器，有进位时该触发器状态为“1”。

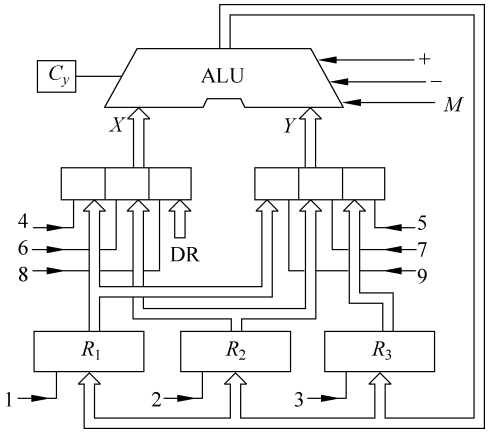


图 5.22　简单运算器数据通路图

ALU 的操作(加、减、传送)在同一个 CPU 周期中只能选择一种，不能并行，所以＋、－、M(传送)三个微操作是相斥性的微操作。类似地，4、6、8 三个微操作是相斥性的，5、7、9 三个微操作也是相斥性的。

微操作 1、2、3 是可以同时进行的，所以是相容性的微操作。另外，ALU 的 X 输入微操作 4、6、8 分别与 Y 输入微操作 5、7、9 任意两个微操作也都是相容性的。

2. 微指令和微程序

在机器的一个 CPU 周期中,一组实现一定操作功能的微命令的组合,构成一条微指令。

图 5.23 表示一个具体的微指令结构,微指令字长为 23 位,它由操作控制和顺序控制两大部分组成。

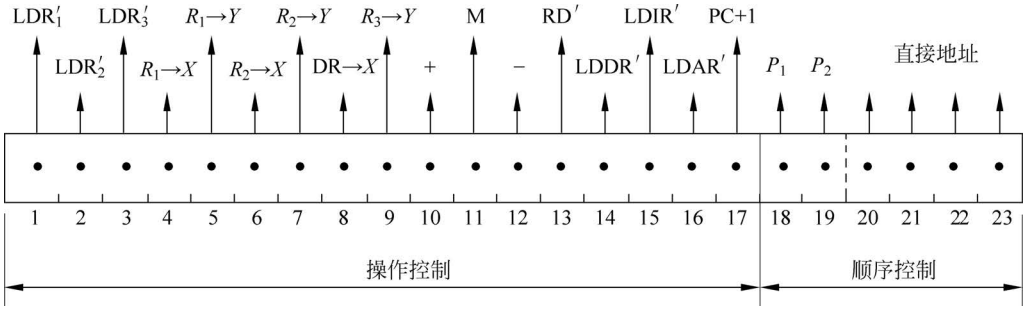


图 5.23 微指令基本格式

操作控制部分用来发出管理和指挥全机工作的控制信号。为了形象直观,在我们的例子中,该字段为 17 位,每一位表示一个微命令。每个微命令的编号同图 5.22 所示的数据通路相对应,具体功能示于微指令格式的左上部。当操作控制字段某一位信息为“1”时,表示发出微命令;而某一位信息为“0”时,表示不发出微命令。例如,当微指令字第 1 位信息为“1”时,表示发出 LDR_1' 的微命令,那么运算器将执行 ALU→R_1 的微操作,把公共总线上的信息输入寄存器 R_1 中。同样,当微指令第 10 位信息为“1”时,则表示向 ALU 发出进行“+”的微命令,因而 ALU 就执行“+”的微操作。

注意,图 5.23 中微指令给出的控制信号都是节拍电位信号,它们的持续时间都是一个 CPU 周期。如果要用来控制图 5.22 所示的运算器数据通路,势必会出现问题,因为前面的三个微命令信号(LDR_1'、LDR_2'、LDR_3')既不能来得太早,也不能来得太晚。为此,要求这些微命令信号还要加入时间控制,例如同节拍脉冲 T_4 相与而得到 LDR_1 ~ LDR_3 信号,如图 5.24 右图所示。在这种情况下,控制器最后发给运算器的 12 个控制信号中,3 个是节拍脉冲信号(LDR_1、LDR_2、LDR_3),其他 9 个都是节拍电位信号,从而保证运算器在前 600ns 时间内进行运算。600ns 后运算完毕,公共总线上输出稳定的运算结果,由 LDR_1(或 LDR_2、LDR_3)信号输入相应的寄存器中,其时间关系如图 5.24 所示。

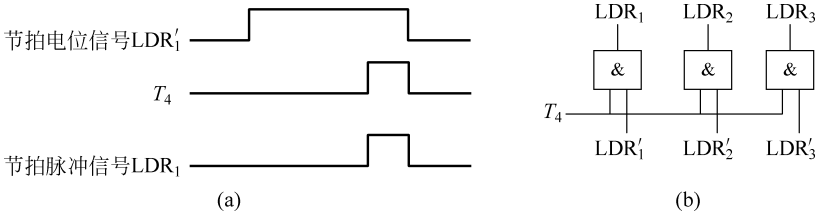


图 5.24 运算器操作时序与产生逻辑

微指令格式中的顺序控制部分用来决定产生下一条微指令的地址。下面我们将会知道,一条机器指令的功能是用许多条微指令组成的序列来实现的,这个微指令序列通常叫作

微程序。既然微程序是由微指令组成的，那么当执行当前一条微指令时，必须指出后继微指令的地址，以便当前一条微指令执行完毕后，取出下一条微指令。

决定后继微指令地址的方法不止一种。在我们所举的例子中，由微指令顺序控制字段的 6 位信息来决定。其中 4 位(20～23)用来直接给出下一条微指令的地址。第 18、19 两位作为判别测试标志。当此两位为“00”时，表示不进行测试，直接按顺序控制字段第 20～23 位给出的地址取下一条微指令；当第 18 位或第 19 位为“1”时，表示要进行 P_1 或 P_2 的判别测试，根据测试结果，需要对第 20～23 位的某一位或几位进行修改，然后按修改后的地址取下一条微指令。

3. 微程序控制器原理框图

微程序控制器原理框图如图 5.25 所示。它主要由控制存储器、微指令寄存器和地址转移逻辑三大部分组成，其中微指令寄存器分为微地址寄存器和微命令寄存器两部分。

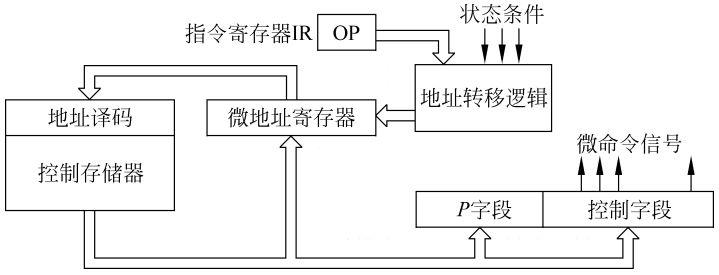


图 5.25　微程序控制器组成原理框图

(1) 控制存储器。控制存储器用来存放实现全部指令系统的微程序，它是一种只读型存储器。一旦微程序固化，机器运行时则只读不写。其工作过程是：每读出一条微指令，则执行这条微指令；接着又读出下一条微指令，又执行这一条微指令……读出一条微指令并执行微指令的时间总和称为一个微指令周期。通常，在串行方式的微程序控制器中，微指令周期就是只读存储器的工作周期。控制存储器的字长就是微指令字的长度，其存储容量视机器指令系统而定，即取决于微程序的数量。对控制存储器的要求是速度快、读出周期要短。

(2) 微指令寄存器。微指令寄存器用来存放由控制存储器读出的一条微指令信息。其中微地址寄存器决定将要访问的下一条微指令的地址，而微命令寄存器则保存一条微指令的操作控制字段和判别测试字段的信息。

(3) 地址转移逻辑。在一般情况下，微指令由控制存储器读出后直接给出下一条微指令的地址，通常我们简称微地址，这个微地址信息就存放在微地址寄存器中。如果微程序不出现分支，那么下一条微指令的地址就直接由微地址寄存器给出。当微程序出现分支时，意味着微程序出现条件转移。在这种情况下，通过判别测试字段 P 和执行部件的“状态条件”反馈信息，去修改微地址寄存器的内容，并按改好的内容去读下一条微指令。地址转移逻辑就承担自动完成修改微地址的任务。

4. 微程序举例

一条机器指令是由若干条微指令组成的序列来实现的。因此，一条机器指令对应着一

个微程序,而微程序的总和便可实现整个指令系统。

现在我们举“十进制加法”指令为例,具体看一看微程序控制的过程。

“十进制加法”指令的功能是用 BCD 码来完成十进制数的加法运算。在十进制运算时,当相加二数之和大于 9 时,便产生进位。可是用 BCD 码完成十进制数运算时,当和数大于 9 时,必须对和数进行加 6 修正。这是因为,采用 BCD 码后,在二数相加的和数小于或等于 9 时,十进制运算的结果是正确的;而当二数相加的和数大于 9 时,结果不正确,必须加 6 修正后才能得出正确结果。

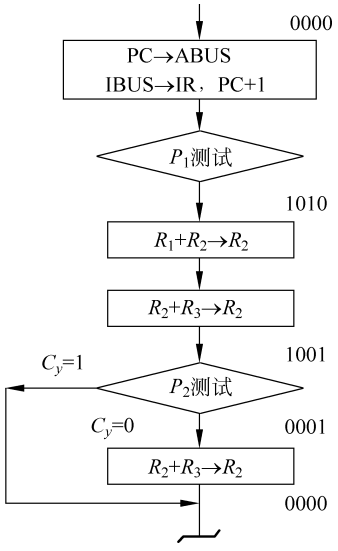


图 5.26 十进制加法微程序

假定指令存放在指存中,数据 a、b 及常数 6 已存放在图 5.22 中的 R_1、R_2、R_3 三个寄存器中,因此,完成十进制加法的微程序流程图示于图 5.26 中。执行周期要求先进行 a+b+6 运算,然后判断结果有无进位:当进位标志 $C_y=1$,不减 6;当 $C_y=0$,减去 6,从而获得正确结果。

我们看到,十进制加法微程序流程图由四条微指令组成,每一条微指令用一个长方框表示。第一条微指令为“取指”周期,它是一条专门用来取机器指令的微指令,任务有三个:一是从内存取出一条机器指令,并将指令放到指令寄存器(IR)中。在我们的例子中,取出的是“十进制加法”指令。二是对程序计数器加 1,做好取下一条机器指令的准备。第三个任务是对机器指令的操作码用 P_1 进行判别测试,然后修改微地址寄存器内容,给出下一条微指令的地址。在我们所示的微程序流程图中,每一条微指令的地址用数字示于长方框的右上角。注意,菱形符号代表判别测试,它的动作依附于第一条微指令。第二条微指令完成 a+b 运算。第三条微指令完成加 6 运算,同时又进行判别测试。不过这一次的判别标志不是 P_1 而是 P_2,P_2 用来测试进位标志 C_y。

根据测试结果,微程序或者转向公操作,或者转向第四条微指令。当微程序转向公操作(用符号~表示)时,如果没有外围设备请求服务,那么又转向取下一条机器指令。与此相对应,第三条微指令和第四条微指令的下一个微地址就又指向第一条微指令,即“取指”微指令。

假设我们已经按微程序流程图编好了微程序,并已事先存放到控制存储器中。同时假定用图 5.22 所示的运算器做执行部件。机器启动时,只要给出控制存储器的首地址,就可以调出所需要的微程序。为此,首先给出第一条微指令的地址 0000,经地址译码,控制存储器选中所对应的“取指”微指令,并将其读到微指令寄存器中。此例采用传统存储模式,并未采用哈佛结构。

第一条微指令的二进制编码是

000	000	000	000	11111	10	0000

在这条微指令中,操作控制字段有五个微命令:第 16 位发出 PC→ABUS(I),将 PC 内容送到指存地址总线 ABUS(I);第 13 位发出指存读命令 RD(I),于是指存执行读操作,从

指存单元取出“十进制加法”指令放到指令总线(IBUS)上,其数据通路可参阅图 5.5。第 15 位发出 LDIR′,将 IBUS 上的“十进制加法”指令打入到指令寄存器(IR)。假定“十进制加法”指令的操作码为 1010,那么指令寄存器的 OP 字段现在是 1010。第 17 位发出 PC+1 微命令,使程序计数器加 1,做好取下一条机器指令的准备。

另一方面,微指令的顺序控制字段指明下一条微指令的地址是 0000,但是由于判别字段中第 18 位为 1,表明是 P_1 测试,因此 0000 不是下一条微指令的真正的地址。P_1 测试的“状态条件”是指令寄存器的操作码字段,即用 OP 字段作为形成下一条微指令的地址,于是微地址寄存器的内容修改成 1010。

在第二个 CPU 周期开始时,按照 1010 这个微地址读出第二条微指令,它的二进制编码是

010	100	100	100	00000	00	1001

在这条微指令中,操作控制部分发出如下四个微命令:$R_1 \rightarrow X$、$R_2 \rightarrow Y$、+、LDR_2',于是运算器完成 $R_1+R_2 \rightarrow R_2$ 的操作,其数据通路如图 5.22 所示。

与此同时,这条微指令的顺序控制部分由于判别测试字段 P_1 和 P_2 均为零,表示不进行测试,于是直接给出下一条微指令的地址为 1001。

在第三个周期开始时,按照 1001 这个微地址读出第三条微指令,它的二进制编码是

010	001	001	100	00000	01	0000

这条微指令的操作控制部分发出 $R_2 \rightarrow X$、$R_3 \rightarrow Y$、+、LDR_2'四个微命令,运算器完成 $R_2+R_3 \rightarrow R_2$ 的操作。

顺序控制部分由于判别字段中 P_2 为 1,表明进行 P_2 测试,测试的“状态条件”为进位标志 C_y。换句话说,此时微地址 0000 需要进行修改,我们假定用 C_y 的状态来修改微地址寄存器的最后一位:当 $C_y=0$ 时,下一条微指令的地址为 0001;当 $C_y=1$ 时,下一条微指令的地址为 0000。

显然,在测试一个状态时,有两条微指令作为要执行的下一条微指令的“候选”微指令。现在我们假设 $C_y=0$,则要执行的下一条微指令地址为 0001。

在第四个 CPU 周期开始时,按微地址 0001 读出第四条微指令,其二进制编码是

010	001	001	001	0000	00	0000

微指令发出 $R_2 \rightarrow X$、$R_3 \rightarrow Y$、−、LDR_2'四个微命令,运算器完成了 $R_2-R_3 \rightarrow R_2$ 的操作功能。顺序控制部分直接给出下一条微指令的地址为 0000,按该地址取出的微指令是“取指”微指令。

如果第三条微指令进行测试 $C_y=1$ 时,那么微地址仍保持为 0000,将不执行第四条微指令而直接由第三条微指令转向公操作。

当下一个 CPU 周期开始时,“取指”微指令又从内存读出第二条机器指令。如果这条机器指令是 STO 指令,那么经过 P_1 测试,就转向执行 STO 指令的微程序。

以上是由四条微指令序列组成的简单微程序。从这个简单的控制模型中,我们就可以看到微程序控制的主要思想及大概过程。

5. CPU 周期与微指令周期的关系

在串行方式的微程序控制器中,微指令周期等于读出微指令的时间加上执行该条微指令的时间。为了保证整个机器控制信号的同步,可以将一个微指令周期时间设计得恰好和CPU 周期时间相等。图 5.27 示出了某计算机中 CPU 周期与微指令周期的时间关系。

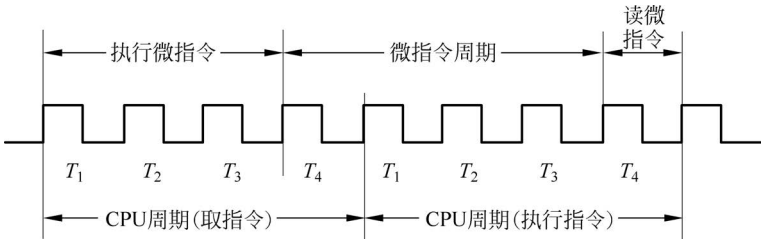


图 5.27 CPU 周期与微指令周期的时间关系

一个 CPU 周期为 0.8μs,它包含四个等间隔的节拍脉冲 $T_1 \sim T_4$,每个脉冲宽度为 200ns。用 T_4 作为读取微指令的时间,用 $T_1+T_2+T_3$ 时间作为执行微指令的时间。例如,在前 600ns 时间内运算器进行运算,在 200ns 时间的末尾运算器已经运算完毕,可用 T_4 上升沿将运算结果输入某个寄存器中。与此同时可用 T_4 间隔读取下条微指令,经时间延迟,下条微指令又从只读存储器读出,并用 T_1 上升沿输入微指令寄存器中。如忽略触发器的翻转延迟,那么下条微指令的微命令信号就从 T_1 上升沿起就开始有效,直到下一条微指令读出后输入微指令寄存器中为止。因此一条微指令的保持时间恰好是 0.8μs,也就是一个 CPU 周期的时间。

6. 机器指令与微指令的关系

经过上面的讲述,应该说,我们能够透彻地了解机器指令与微指令的关系。也许读者会问:一会儿取机器指令,一会儿取微指令,它们之间到底是什么关系?

现在让我们把前面内容归纳一下,作为对此问题的回答。

第一,一条机器指令对应一个微程序,这个微程序是由若干条微指令序列组成的。因此,一条机器指令的功能是由若干条微指令组成的序列来实现的。简言之,一条机器指令所完成的操作划分成若干条微指令来完成,由微指令进行解释和执行。

第二,从指令与微指令、程序与微程序、地址与微地址的一一对应关系来看,前者与内存储器有关,后者与控制存储器有关。与此相关,也有相对应的硬设备,如图 5.28 所示。

第三,我们在讲述本章 5.2 节时,曾讲述了指令与机器周期概念,并归纳了五条典型指令的指令周期(参见图 5.15)。现在我们看到,图 5.15 就是这五条指令的微程序流程图,每一个 CPU 周期就对应一条微指令。这就告诉我们如何设计微程序,也将使我们进一步体验到机器指令与微指令的关系。

5.4.2 微程序设计技术

我们已经了解了微程序控制器的基本原理。这使我们认识到,如何确定微指令的结构仍是微程序设计的关键。

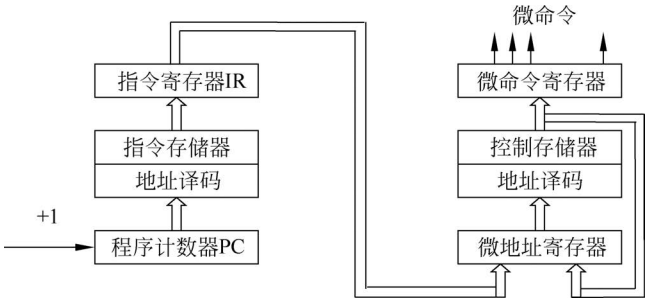


图 5.28　机器指令与微指令的关系

设计微指令结构应当追求的目标是：①有利于缩短微指令字长度；②有利于减小控制存储器的容量；③有利于提高微程序的执行速度；④有利于对微指令的修改；⑤有利于提高设计的灵活性。

1. 微命令编码

微命令编码，就是对微指令中的操作控制字段采用的表示方法。通常有以下三种方法。

(1) 直接表示法。采用直接表示法的微指令结构见前面图 5.23，其特点是操作控制字段中的每一位代表一个微命令。这种方法的优点是简单直观，其输出直接用于控制。缺点是微指令字较长，因而使控制存储器容量较大。

(2) 编码表示法。又称字段译码表示法。编码表示法是把一组相斥性的微命令信号组成一个小组(即一个字段)，然后通过小组(字段)译码器对每一个微命令信号进行译码，译码输出作为操作控制信号，其微指令结构如图 5.29 所示。

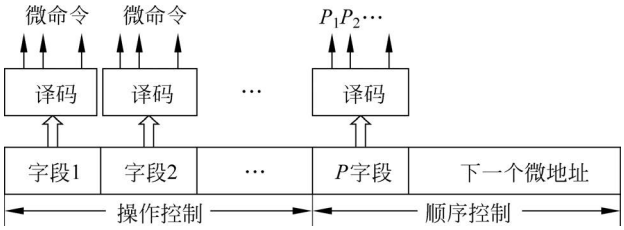


图 5.29　字段译码表示法

采用字段译码的编码方法，可以用较小的二进制信息位表示较多的微命令信号。例如 3 位二进制位译码后可表示 7 个微命令，4 位二进制位译码后可表示 15 个微命令，因为每个译码字段都要留一个“无操作”控制信号。与直接控制法相比，字段译码控制法可使微指令字大大缩短，但由于增加译码电路，使微程序的执行速度稍稍减慢。目前在微程序控制器设计中，字段直接译码法使用较普遍。

(3) 混合表示法。这种方法是把直接表示法与字段编码法混合使用，以便能综合考虑微指令字长、灵活性、执行微程序速度等方面的要求。

另外，在微指令中还可附设一个常数字段。该常数可作为操作数送入 ALU 运算，也可作为计数器初值用来控制微程序循环次数。

2. 微地址的形成方式

微指令执行的顺序控制问题，实际上是如何确定下一条微指令的地址问题。通常，产生

后继微地址有两种方法。

(1) 计数器方式。这种方法同用程序计数器来产生机器指令地址的方法相类似。在顺序执行微指令时,后继微地址由现行微地址加上一个增量来产生;在非顺序执行微指令时,必须通过转移方式,使现行微指令执行后,转去执行指定后继微地址的下一条微指令。在这种方法中,微地址寄存器通常改为计数器。为此,顺序执行的微指令序列就必须安排在控制存储器的连续单元中。

计数器方式的基本特点是:微指令的顺序控制字段较短,微地址产生机构简单。但是多路并行转移功能较弱,速度较慢,灵活性较差。

(2) 多路转移方式。一条微指令具有多个转移分支的能力称为多路转移。例如,"取指"微指令根据操作码 OP 产生多路微程序分支而形成多个微地址。在多路转移方式中,当微程序不产生分支时,后继微地址直接由微指令的顺序控制字段给出;当微程序出现分支时,有若干"后选"微地址可供选择:即按顺序控制字段的"判别测试"标志和"状态条件"信息来选择其中一个微地址,其原理如图 5.25 所示。"状态条件"有 1 位标志,可实现微程序两路转移,涉及微地址寄存器的一位;"状态条件"有 2 位标志,可实现微程序 4 路转移,涉及微地址寄存器的两位。依此类推,"状态条件"有 n 位标志,可实现微程序 2^n 路转移,涉及微地址寄存器的 n 位。因此执行转移微指令时,根据状态条件可转移到 2^n 个微地址中的一个。

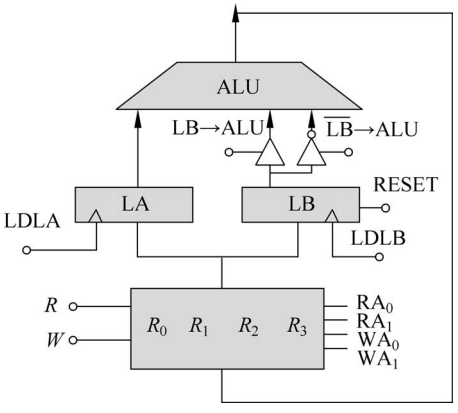


图 5.30 某运算器内部结构

多路转移方式的特点是,能以较短的顺序控制字段配合,实现多路并行转移,灵活性好,速度较快,但转移地址逻辑需要用组合逻辑方法设计。

【例 5.2】 某计算机的微指令格式如下,运算器示意图如图 5.30 所示,不考虑取指过程,写出如下三条指令执行部分所对应的微程序。

(1) ADD R_0,R_1; $(R_0)+(R_1)\rightarrow R_1$。

(2) SUB R_2,R_3; $(R_3)-(R_2)\rightarrow R_3$。

(3) MOV R_2,R_3; $(R_2)\rightarrow(R_3)$。

RA_1 RA_0	WA_1 WA_0	R	W	LDLA	LDLB	LB→ALU	$\overline{LB}$→ALU	RESET	～～～

解:

各微信号含义如下。

RA_1,RA_0:读 R_0～R_3 的选择控制。

WA_1,WA_0:写 R_0～R_3 的选择控制。

R:寄存器读命令。

W:寄存器写命令。

LDLA:输入 LA 的控制信号。

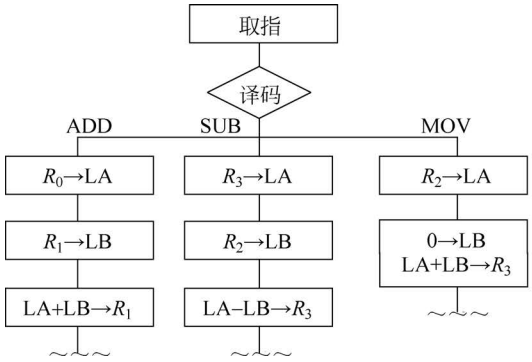
LDLB:输入 LB 的控制信号。

LB→ALU：传送 LB 的控制信号。

$\overline{\text{LB}}$→ALU：传送 $\overline{\text{LB}}$ 的控制信号，并使加法器最低位加 1。

Reset：清暂存器 LB 为 0 的信号。

～～～：一段微程序结束，转入取机器指令的控制信号。



指令	微程序代码
ADD	① 00 ×× 10100000 ② 01 ×× 10010000 ③ ×× 0 1 01001001
SUB	④ 11 ×× 10100000 ⑤ 10 ×× 10010000 ⑥ ×× 1 1 01000101
MOV	⑦ 10 ×× 10100000 ⑧ ×× 1 1 01001011

【例 5.3】 微地址寄存器有 6 位（μA_5～μA_0），当需要修改其内容时，可通过某一位触发器的强置端 S 将其置“1”。现有三种情况：①执行“取指”微指令后，微程序按 IR 的 OP 字段（IR_3～IR_0）进行 16 路分支；②执行条件转移指令微程序时，按进位标志 C 的状态进行 2 路分支；③执行控制指令微程序时，按 IR_4、IR_5 的状态进行 4 路分支。请按多路转移方法设计微地址转移逻辑。

解：按所给设计条件，微程序有三种判别测试，分别为 P_1、P_2、P_3。由于修改 μA_5～μA_0 内容具有很大灵活性，现分配如下：

（1）用 P_1 和 IR_3～IR_0 修改 μA_3～μA_0。

（2）用 P_2 和 C 修改 μA_0。

（3）用 P_3 和 IR_5、IR_4 修改 μA_5、μA_4。

考虑时间因素 T_4（假设 CPU 周期最后一个节拍脉冲），转移逻辑表达式为

$$\mu A_5 = P_3 \cdot IR_5 \cdot T_4$$

$$\mu A_4 = P_3 \cdot IR_4 \cdot T_4$$

$$\mu A_3 = P_1 \cdot IR_3 \cdot T_4$$

$$\mu A_2 = P_1 \cdot IR_2 \cdot T_4$$

$$\mu A_1 = P_1 \cdot IR_1 \cdot T_4$$

$$\mu A_0 = P_1 \cdot IR_0 \cdot T_4 + P_2 \cdot C \cdot T_4$$

由于从触发器强置端修改，故前 5 个表达式可用“与非”门实现，最后一个用“与或非”门实现。

3. 微指令格式

微指令的编译方法是决定微指令格式的主要因素。考虑到速度、成本等原因，在设计计算机时采用不同的编译法。因此微指令的格式大体分成两类：水平型微指令和垂直型微指令。

1) 水平型微指令

一次能定义并执行多个并行操作微命令的微指令,叫作水平型微指令。水平型微指令的一般格式如下

控制字段	判别测试字段	下地址字段

按照控制字段的编码方法不同,水平型微指令又分为三种:一种是全水平型(不译码法)微指令,第二种是字段译码法水平型微指令,第三种是直接和译码相混合的水平型微指令。

2) 垂直型微指令

微指令中设置微操作码字段,采用微操作码编译法,由微操作码规定微指令的功能,称为垂直型微指令。

垂直型微指令的结构类似于机器指令的结构。它有操作码,在一条微指令中只有1～2个微操作命令,每条微指令的功能简单,因此,实现一条机器指令的微程序要比水平型微指令编写的微程序长得多。它是采用较长的微程序结构去换取较短的微指令结构。

下面用4条垂直型微指令的微指令格式加以说明。设微指令字长为16位,微操作码3位。

(1) 寄存器—寄存器传送型微指令。

15～13	12～8	7～3	2～0
000	源寄存器编址	目标寄存器编址	其他

其功能是把源寄存器数据送到目标寄存器。13～15位为微操作码,源寄存器和目标寄存器编址各5位,可指定31个寄存器。

(2) 运算控制型微指令。

15～13	12～8	7～3	2～0
001	左输入源编址	右输入源编址	ALU

其功能是选择ALU的左、右两输入源信息,按ALU字段所指定的运算功能(8种操作)进行处理,并将结果送入暂存器中。左、右输入源编址可指定31种信息源之一。

(3) 访问主存微指令。

15～13	12～8	7～3	2～1	0
010	寄存器编址	存储器编址	读写	其他

其功能是将主存中一个单元的信息送入寄存器或者将寄存器的数据送往主存。存储器编址是指按规定的寻址方式进行编址。第1、2位指定读操作或写操作(取其之一)。

(4) 条件转移微指令。

15～13	12～4	3～0
011	D	测试条件

其功能是根据测试对象的状态决定是转移到D所指定的微地址单元,还是顺序执行下一条微指令。9位D字段不足以表示一个完整的微地址,但可以用来替代现行的低位地址。测试条件字段有4位,可规定16种测试条件。

3）水平型微指令与垂直型微指令的比较

（1）水平型微指令并行操作能力强，效率高，灵活性强，垂直型微指令则较差。

在一条水平型微指令中，设置有控制信息传送通路（控制门）以及进行所有操作的微命令，因此在进行微程序设计时，可以同时定义比较多的并行操作的微命令，来控制尽可能多的并行信息传送，从而使水平型微指令具有效率高及灵活性强的优点。

在一条垂直型微指令中，一般只能完成一个操作，控制一两个信息传送通路，因此微指令的并行操作能力低，效率低。

（2）水平型微指令执行一条指令的时间短，垂直型微指令执行时间长。

因为水平型微指令的并行操作能力强，因此与垂直型微指令相比，可以用较少的微指令数来实现一条指令的功能，从而缩短了指令的执行时间。而且当执行一条微指令时，水平型微指令的微命令一般直接控制对象，而垂直型微指令要经过译码，会影响速度。

（3）由水平型微指令解释指令的微程序，有微指令字较长而微程序短的特点。垂直型微指令则相反，微指令字较短而微程序长。

（4）水平型微指令用户难以掌握，而垂直型微指令与指令比较相似，相对来说，比较容易掌握。

水平型微指令与机器指令差别很大，一般需要对机器的结构、数据通路、时序系统以及微命令很精通才能设计。

垂直型微指令的设计思想在奔腾4、安腾系列机中得到了应用。

4. 动态微程序设计

微程序设计技术还有静态微程序设计和动态微程序设计之分。对应于一台计算机的机器指令只有一组微程序，而且这一组微程序设计好之后，一般无须改变而且也不好改变，这种微程序设计技术称为静态微程序设计。本节前面讲述的内容基本上属于静态微程序设计的概念。

当采用EEPROM作为控制存储器时，还可以通过改变微指令和微程序来改变机器的指令系统，这种微程序设计技术称为动态微程序设计。采用动态微程序设计时，微指令和微程序可以根据需要加以改变，因而可在一台机器上实现不同类型的指令系统。

5.5 硬布线控制器

1. 基本思想

硬连线控制器是早期设计计算机的一种方法。这种方法是把控制部件看作产生专门固定时序控制信号的逻辑电路，而此逻辑电路以使用最少元件和取得最高操作速度为设计目标。一旦控制部件构成后，除非重新设计和物理上对它重新布线，否则要想增加新的控制功能是不可能的。这种逻辑电路是一种由门电路和触发器构成的复杂树形逻辑网络，故称之为硬连线控制器。

硬连线控制器是计算机中最复杂的逻辑部件之一。当执行不同的机器指令时，通过激活一系列彼此很不相同的控制信号来实现对指令的解释，其结果使得控制器往往很少有明

确的结构而变得杂乱无章。结构上的这种缺陷使得硬连线控制器的设计和调试非常复杂且代价很大。正因为如此,硬连线控制器被微程序控制器所取代。但是随着新一代机器及VLSI技术的发展,硬连线逻辑设计思想又得到了重视。

图5.31示出了硬连线控制器的结构方框图。

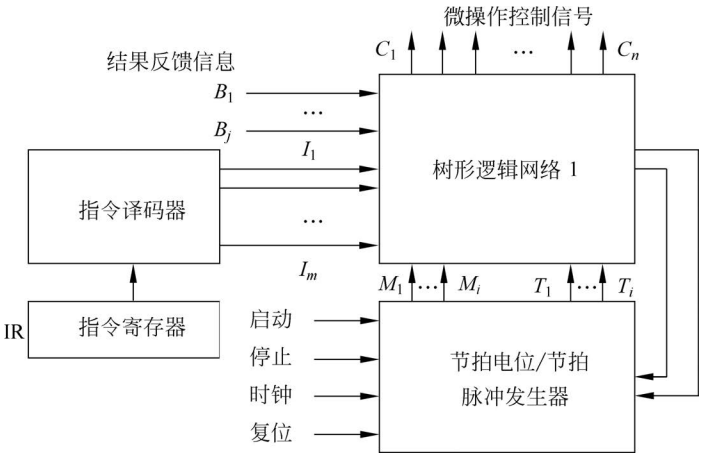


图5.31 硬连线控制器结构方框图

逻辑网络的输入信号来源有三个:①来自指令操作码译码器的输出 $I_1 \cdots I_m$;②来自执行部件的反馈信息 $B_1 \cdots B_j$;③来自时序产生微操作控制信号器的时序信号,包括节拍电位信号 M 和节拍脉冲信号 T。其中节拍电位信号就是5.3节规定的机器周期(CPU周期)信号,节拍脉冲信号是时钟周期信号。

逻辑网络 N 的输出信号就是微操作控制信号,它用来对执行部件进行控制。另有一些信号则根据条件变量来改变时序发生器的计数顺序,以便跳过某些状态,从而可以缩短指令周期。显然,硬连线控制器的基本原理,归纳起来可叙述为:某一微操作控制信号 C 是指令操作码译码器输出 I_m、时序信号(节拍电位 M_i,节拍脉冲 T_k)和状态条件信号 B_j 的逻辑函数,即

$$C = f(I_m, M_i, T_k, B_j)$$

这个控制信号是用门电路、触发器等许多器件采用布尔代数方法来设计实现的。当机器加电工作时,某一操作控制信号 C 在某条特定指令和状态条件下,在某一序号的特定节拍电位和节拍脉冲时间间隔中起作用,从而激活这条控制信号线,对执行部件实施控制。显然,从指令流程图出发,就可以一个不漏地确定在指令周期中各个时刻必须激活的所有操作控制信号。例如,对引起一次主存读操作的控制信号 C_3 来说,当节拍电位 $M_1=1$,取指令时被激活;而节拍电位 $M_4=1$,三条指令(LAD、ADD、AND)取操作数时也被激活,此时指令译码器的LAD、ADD、AND输出均为1,因此 C_3 的逻辑表达式可由下式确定

$$C_3 = M_1 + M_4(\mathrm{LAD} + \mathrm{ADD} + \mathrm{AND})$$

一般来说,还要考虑节拍脉冲和状态条件的约束,所以每一个控制信号 C_n 可以由以下形式的布尔代数表达式来确定

$$C_n = \sum (M_i, T_k, B_j, I_m)$$

与微程序控制相比，硬连线控制的速度较快。其原因是微程序控制中每条微指令都要从控存中读取一次，影响了速度，而硬连线控制主要取决于电路延迟。因此在某些超高速新型计算机结构中，又选用了硬连线控制器，或与微程序控制器混合使用。

2. 指令执行流程

我们在介绍微程序控制器时曾提到，一个机器指令对应一个微程序，而一个微指令周期则对应一个节拍电位时间。一条机器指令用多少条微指令来实现，则该条指令的指令周期就包含了多少个节拍电位时间，因而对时间的利用是十分经济的。由于节拍电位是用微指令周期来体现的，因而时序信号比较简单，时序计数器及其译码电路只需产生若干节拍脉冲信号即可。

在用硬连线实现的操作控制器中，通常，时序产生器除了产生节拍脉冲信号外，还应当产生节拍电位信号。这是因为，在一个指令周期中要顺序执行一系列微操作，需要设置若干节拍电位来定时。例如图 5.15 所示五条指令的指令周期，其指令流程可用图 5.32 来表示。

由图 5.32 可知，所有指令的取指周期放在 M_1 节拍。在此节拍中，操作控制器发出微操作控制信号，完成从指令存储器取出一条机器指令。

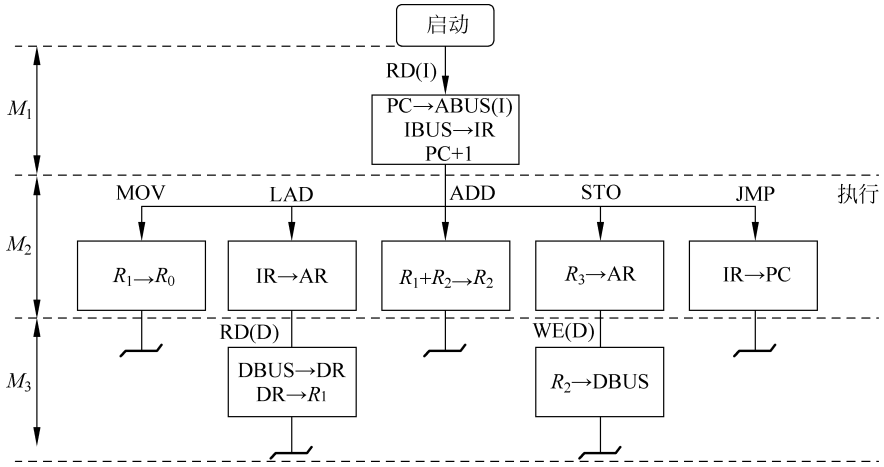


图 5.32　硬连线控制器的指令周期流程图

指令的执行周期由 M_2、M_3 两个节拍来完成。MOV、ADD 和 JMP 指令只需一个节拍（M_2）即可完成。LAD 和 STO 指令需要两个节拍（M_2、M_3）。为了简化节拍控制，指令的执行过程可采用同步工作方式，即各条指令的执行阶段均用最长节拍数 M_3 来考虑。这样，对 MOV、ADD、JMP 三条指令来讲，在 M_3 节拍中没有什么操作。

显然，由于采用同步工作方式，长指令和短指令对节拍时间的利用都是一样的。这对短指令来讲，在时间的利用上是浪费的，因而也降低了 CPU 的指令执行速度，影响到机器的速度指标。为了改变这种情况，在设计短指令流程时可以跳过某些节拍，例如 MOV 指令、ADD 指令和 JMP 指令执行 M_2 节拍后跳过 M_3 节拍而返回 M_1 节拍。当然在这种情况下，节拍信号发生器的电路相应就要复杂一些。

节拍电位信号的产生电路与节拍脉冲产生电路十分类似，它可以在节拍脉冲信号时序器的基础上产生，运行中以循环方式工作，并与节拍脉冲保持同步。

3. 微操作控制信号的产生

在微程序控制器中,微操作控制信号由微指令产生,并且可以重复使用。

在硬连线控制器中,某一微操作控制信号由布尔代数表达式描述的输出函数产生。

设计微操作控制信号的方法和过程是,根据所有的机器指令流程图,寻找出产生同一个微操作信号的所有条件,并与适当的节拍电位和节拍脉冲组合,从而写出其布尔代数表达式并进行简化,然后用门电路或可编程器件来实现。

为了防止遗漏,设计时可按信号出现在指令流程图中的先后次序来书写,然后进行归纳和简化。要特别注意控制信号是电位有效还是脉冲有效,如果是脉冲有效,必须加入节拍脉冲信号进行相"与"。

【例 5.4】 根据图 5.31,写出以下操作控制信号 RD(I)、RD(D)、WE(D)、LDPC、LDIR、LDAR、LDDR、PC+1、LDR_2 的逻辑表达式。其中每个操作控制信号的含义是:

RD(I)——指令存储读命令　　RD(D)——数据存储读命令

WE(D)——数据存储写命令　　LDPC——输入程序计数器

LDIR——输入指令寄存器　　LDAR——输入数据存储地址寄存器

LDDR——输入数据缓冲寄存器　　PC+1——程序计数器+1

LDR_2——输入 R_2 寄存器

解:设 M_1、M_2、M_3 为节拍电位信号,T_1、T_2、T_3、T_4 为一个 CPU 周期中的节拍脉冲信号,MOV、LAD、ADD、STO、JMP 分别表示对应机器指令的 OP 操作码译码输出信号,则有如下逻辑表达式

$RD(I)=M_1$　　(电位信号)

$RD(D)=M_3\mathrm{LAD}$　　(电位信号)

$WE(D)=M_3T_3\mathrm{LAD}$　　(脉冲信号)

$LDPC=M_1T_4+M_2T_4\mathrm{JMP}$

$LDIR=M_1T_4$

$LDAR=M_2T_4(\mathrm{LAD}+\mathrm{STO})$

$LDDR=M_2T_3(\mathrm{MOV}+\mathrm{ADD})+M_3T_3\mathrm{LDA}$

$PC+1=M_1$

$LDR_2=M_2T_4\mathrm{ADD}$

5.6 指令流水

计算机自诞生到现在,人们追求的目标之一是很高的运算速度,因此,并行处理技术便成为计算机发展的主流。

早期的计算机基于冯·诺伊曼的体系结构,采用的是串行处理。这种计算机的主要特征是:计算机的各个操作(如读/写存储器,算术或逻辑运算,I/O 操作)只能串行地完成,即任一时刻只能进行一个操作。而并行处理则使得以上各个操作能同时进行,从而大大提高了计算机的速度。

广义地讲，并行性有着两种含义：一是同时性，指两个以上事件在同一时刻发生；二是并发性，指两个以上事件在同一时间间隔内发生。计算机的并行处理技术可贯穿于信息加工的各个步骤和阶段，概括起来，主要有三种形式：①时间并行；②空间并行；③时间并行＋空间并行。

时间并行指时间重叠，在并行性概念中引入时间因素，让多个处理过程在时间上相互错开，轮流重叠地使用同一套硬件设备的各个部分，以加快硬件周转而赢得速度。

时间并行性概念的实现方式就是采用流水处理部件。这是一种非常经济而实用的并行技术，能保证计算机系统具有较高的性能价格比。目前的高性能微型机几乎无一例外地使用了流水技术。

空间并行指资源重复，在并行性概念中引入空间因素，以“数量取胜”为原则来大幅度提高计算机的处理速度。大规模和超大规模集成电路的迅速发展为空间并行技术带来了巨大生机，因而成为目前实现并行处理的一个主要途径。空间并行技术主要体现在多处理器系统和多计算机系统。但是在单处理器系统中也得到了广泛应用。

时间并行＋空间并行指时间重叠和资源重复的综合应用，既采用时间并行性又采用空间并行性。例如，奔腾 CPU 采用了超标量流水技术，在一个机器周期中同时执行两条指令，因而既具有时间并行性，又具有空间并行性。显然，第三种并行技术带来的高速效益是最好的。

5.6.1　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。

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 5.33 所示。

图 5.33　指令的串行执行

图 5.33 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 5.34 所示，这就是两条指令的重叠，即指令的二级流水。

图 5.34　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，就将暂存的指令传送给执行部件执行。与此同时，指令部件又

可取出下一条指令并暂存起来,这称为指令预存。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。

取指令(IF):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(ID):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、基址、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),于是可得表 5.2 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 IF、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,表 5.2 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。

表 5.2 指令六级流水时序

指令	时钟													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
指令 1	IF	ID	CO	FO	EI	WO								
指令 2		IF	ID	CO	FO	EI	WO							
指令 3			IF	ID	CO	FO	EI	WO						
指令 4				IF	ID	CO	FO	EI	WO					
指令 5					IF	ID	CO	FO	EI	WO				
指令 6						IF	ID	CO	FO	EI	WO			
指令 7							IF	ID	CO	FO	EI	WO		
指令 8								IF	ID	CO	FO	EI	WO	
指令 9									IF	ID	CO	FO	EI	WO

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。

5.6.2 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 5.3 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。

下面分析上述三种相关对流水线工作的影响。

表 5.3 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
MEM	——	访存(读/写)	若条件成立则转移目标地址送 PC
WB	结果写回寄存器堆	将读出的数据写入寄存器堆	——

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 5.4 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 5.5 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 5.4 两条指令同时访存造成结构相关冲突

指令	时钟							
	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		

续表

指　令	时　钟							
	1	2	3	4	5	6	7	8
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

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。

表 5.5　解决访存冲突的一种方案

指　令	时　钟								
	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	
指令 $i+4$						IF	ID	EX	MEM

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 R_1,R_2,R_3;　　$(R_2)+(R_3)\longrightarrow R_1$

SUB R_4,R_1,R_5;　　$(R_1)+(R_5)\longrightarrow R_4$

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 5.6 所示。

表 5.6　ADD 和 SUB 指令发生先写后读(RAW)的数据相关冲突

指　令	时　钟					
	1	2	3	4	5	6
ADD	IF	ID	EX	MEM	WB(写 R_1)	
SUB		IF	ID(读 R_1)	EX	MEM	WB

由表 5.6 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 5.6 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

$$\begin{aligned}
&\text{ADD} \quad R_1,R_2,R_3; \quad (R_2)+(R_3)\longrightarrow R_1\\
&\text{SUB} \quad R_4,R_1,R_5; \quad (R_1)+(R_5)\longrightarrow R_4\\
&\text{ADD} \quad R_6,R_1,R_7; \quad (R_1)\ \text{AND}\ (R_7)\longrightarrow R_6\\
&\text{OR} \quad R_8,R_1,R_9; \quad (R_1)\ \text{OR}(R_9)\longrightarrow R_8\\
&\text{XOR} \quad R_{10},R_1,R_{11}; \quad (R_1)+(R_{11})\longrightarrow R_{10}
\end{aligned}$$

其中，第一条指令将向 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 5.7 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 5.7　未对数据相关进行特殊处理的流水线

指　令	时　钟								
	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 5.8 所示。显然这将要使流水线停顿 3 个时钟周期。

表 5.8　对数据相关进行特殊处理的流水线

指令	时　钟											
	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 5.35 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。

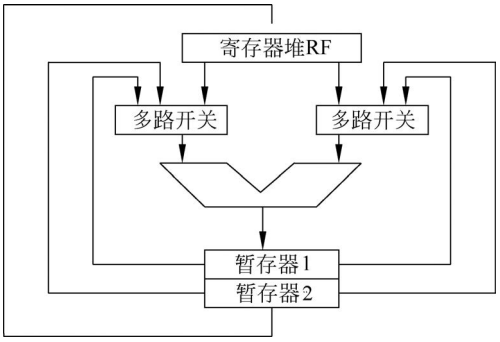


图 5.35 带有旁路技术的 ALU 执行部件

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。

(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 左右,比起数据相关,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 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。表 5.9 示意了条件转移的效果。这里使用了和表 5.2 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。而表 5.2 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。

表 5.9 条件转移对指令流水操作的影响

指令	时钟													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
指令 1	IF	ID	CO	FO	EI	WO								
指令 2		IF	ID	CO	FO	EI	WO							
指令 3			IF	ID	CO	FO	EI	WO						
指令 4				IF	ID	CO	FO							
指令 5					IF	ID	CO							
指令 6						IF	ID							
指令 7							IF							
指令 8								IF	ID	CO	FO	EI	WO	
指令 9									IF	ID	CO	FO	EI	WO

5.6.3 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

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（参见表 5.2 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 $T_p\approx T_{p\max}$。

表 5.2 所示的六级流水线中，设每段时间为 Δt，其最大吞吐率$\frac{1}{\Delta t}$，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

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 $T=m\Delta t+(n-1)\Delta t$ 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

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 5.36 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

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

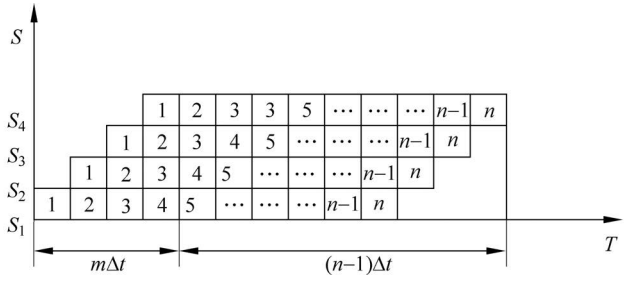


图 5.36 各段时间相等的流水线时空图

【例 5.5】 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 100ns,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 5.37(a)为指令周期流程图。

(2) 非流水线时空图如图 5.37(b)所示。假设一个时间单位为一个时钟周期,每隔 4 个时钟周期才有一个输出结果。

(3) 流水线时空图如图 5.37(c)所示。图中可见,第一条指令出结果需要 4 个时钟周期。当流水线满载时,以后每个时钟周期可以出一个结果,即执行完一条指令。

(4) 如图 5.37 所示,若有 10 条指令进入流水线,在 13 个时钟周期结束时,CPU 执行完 10 条指令,故实际吞吐率为

$$10/(100\text{ns}\times 13)\approx 0.77\times 10^7(\text{条指令/秒})$$

(5) 在流水处理器中,当任务饱满时,指令不断输入流水线,不论是几级流水线,每隔一个时钟周期都输出一个结果。对于本题四级流水线而言,处理 10 条指令所需的时钟周期数为 $T_4=4+(10-1)=13$,而非流水线处理 10 条指令需 $4\times 10=40$ 个时钟周期,故该流水处

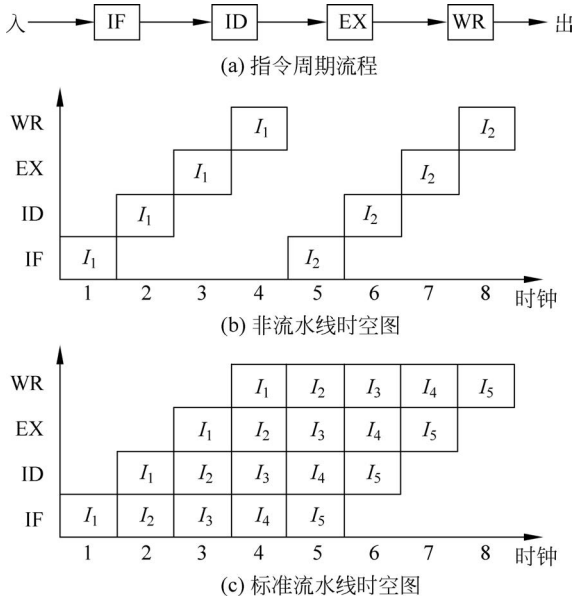


图 5.37　例 5.5 题答图

理器的加速比为 40/13≈3.08。

5.6.4　流水线中的多发技术

流水线技术使计算机系统结构产生重大革新，为了进一步发展，除了采用好的指令调度算法、重新组织指令执行顺序、降低相关带来的干扰以及优化编译外，还可开发流水线中的多发技术，设法在一个时钟周期(机器主频的倒数)内，产生更多条指令的结果。常见的多发技术有超标量技术、超流水线技术和超长指令字技术。假设处理一条指令分 4 个阶段：取指(IF)、译码(ID)、执行(EX)和回写(WR)。图 5.38 是三种多发技术与普通四级流水线的比较，其中图 5.38(a)为普通四级流水线，一个时钟周期出一个结果。

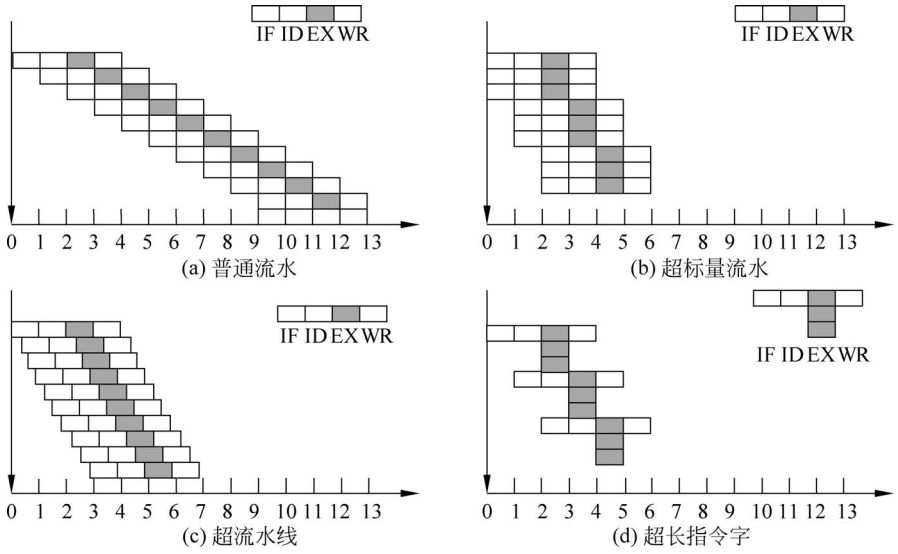


图 5.38　四种流水技术的比较

1. 超标量技术

超标量技术(Super Scalar)如图 5.38(b)所示。它是指在每个时钟周期内可同时并发多条独立指令,即以并行操作方式将两条或两条以上(图中所示为 3 条)指令编译并执行。

要实现超标量技术,要求处理机中配置多个功能部件和指令译码电路,以及多个寄存器端口和总线,以便能实现同时执行多个操作,此外还要编译程序决定哪几条相邻指令可并行执行。

例如,下面两个程序段

程序段 1	程序段 2
MOV BL,8	INC AX
ADD AX,1756H	ADD AX,BX
ADD CL,4EH	MOV DS,AX

左边程序段中的 3 条指令是互相独立的,不存在数据相关,可实现指令级并行。右边程序段中的 3 条指令存在数据相关,不能并行执行。超标量计算机不能重新安排指令的执行顺序,但可以通过编译优化技术,在高级语言翻译成机器语言时精心安排,把能并行执行的指令搭配起来,挖掘更多的指令并行性。

2. 超流水线技术

超流水线(Super Pipe Lining)技术是将一些流水线寄存器插入到流水线段中,好比将流水线再分段,如图 5.38(c)所示。图中将原来的一个时钟周期又分成 3 段,使超流水线的处理器周期比普通流水线的处理器周期短,如图 5.38(a)所示,这样,在原来的时钟周期内,功能部件被使用 3 次,使流水线以 3 倍于原来时钟频率的速度运行。与超标量计算机一样,硬件不能调整指令的执行顺序,靠编译程序解决优化问题。

5.6.5 流水线结构

1. 指令流水线结构

指令流水线是将指令的整个执行过程用流水线进行分段处理,典型的指令执行过程分为“取指令—指令译码—形成地址—取操作数—执行指令—回写结果—修改指令指针”这几个阶段,与此相对应的指令流水线结构由图 5.39 所示的几个部件组成。

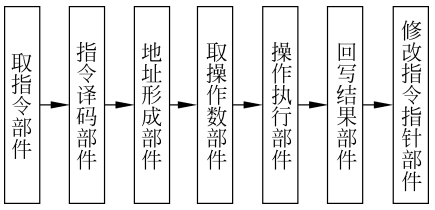


图 5.39 指令流水线结构框图

指令流水线对机器性能的改善程度取决于把处理过程分解成多少个相等的时间段数。如上述共分 7 段,若每一段需要一个时钟周期,则当不采用流水技术时,需 7 个时钟周期出

一个结果。采用流水线后，假设流水线不出现断流（如遇到转移指令），则除第一条指令需 7 个时钟周期出结果外，以后所有的指令都是一个时钟周期出一个结果。因此，在理想的情况下（流水线不断流），该流水线的速度约提高到 7 倍。

2. 运算流水线

上述讨论的指令流水线是指令级的流水技术，实际上流水技术还可用于部件级。例如，浮点加法运算，可以分成“对阶”“尾数加”“结果规格化”3 段，每一段都有一个专门的逻辑电路完成操作，并将其结果保存在锁存器中，作为下一段的输入。如图 5.40 所示，当对阶完成后，将结果存入锁存器，便又可进入下一条指令的对阶运算。

若执行浮点乘运算也按浮点加运算那样分段，即分成阶码运算、尾数乘和结果规格化三级流水线，就不够合理。因为尾数乘所需的时间比阶码运算和规格化操作长得多，而且尾数乘可以和阶码运算同时进行，因此，尾数乘本身就可以用流水线。

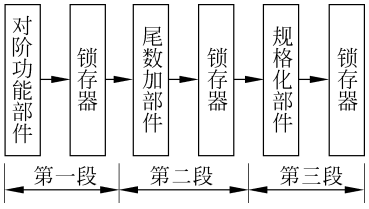


图 5.40　浮点加运算操作流水线

由图 5.40 可见，流水线相邻两段在执行不同的操作，因此在相邻两段之间必须设置锁存器或寄存器，以保证在一个时钟周期内流水线的输入信号不变。这一指导思想也适用于指令流水。此外，只有当流水线各段工作饱满时，才能发挥最大作用。上例中如果浮点运算没有足够的数据来源，那么流水线中的某些段甚至全部段都处于空闲状态，使流水线的作用没有充分发挥。因此具体是否采用流水线技术以及在计算机的哪一部分采用流水线技术需根据情况而定。

5.6.6　奔腾 CPU

1. 奔腾的技术性能

奔腾是 Intel 公司生产的超标量流水处理器，早期使用 5V 工作电压，后期使用 3.3V 工作电压。CPU 的主频是片外主总线时钟频率（60MHz 或 66MHz）的倍频，有 120MHz、166MHz、200MHz 等多种。

CPU 内部的主要寄存器宽度为 32 位，故认为它是一个 32 位微处理器。但它通向存储器的外部数据总线宽度为 64 位，每次总线操作可同时传输 8 字节。以主总线（存储器总线）时钟频率 66MHz 计算，64 位数据总线可使 CPU 与主存的数据交换速率达到 528MB/s；CPU 支持多种类型的总线周期，其中一种称猝发模式，在此模式下，可在一个总线周期内读出或写入 256 位（32 字节）的数据。

CPU 外部地址总线宽度是 36 位，但一般使用 32 位宽，故物理地址空间为 4096MB（4GB）。虚拟地址空间为（64TB），分页模式除支持 4KB 页面外（与 486 相同），还支持 2MB 和 4MB

页面。其中 2MB 页面的分页模式必须使用 36 位地址总线。

CPU 内部分别设置指令 Cache 和数据 Cache,外部还可接 L_2 Cache。CPU 采用 U、V 两条指令流水线,能在一个时钟周期内发射两条简单的整数指令,也可发射一条浮点指令。操作控制器采用硬布线控制和微程序控制相结合的方式。大多数简单指令用硬布线控制实现,在一个时钟周期内执行完毕。对微程序实现的指令,也在 2～3 个时钟周期内执行完毕。

奔腾具有非固定长度的指令格式,9 种寻址方式,191 条指令,但是在每个时钟周期又能执行两条指令。因此它具有 CISC 和 RISC 两者的特性,不过具有的 CISC 特性更多一些,因此被看成为一个 CISC 结构的处理器。以 CISC 结构实现超标量流水线,并有 BTB 方式的转移预测能力,堪称当代 CISC 机器的经典之作。

2. 奔腾 CPU 的结构框图

奔腾 CPU 的结构框图如图 5.41 所示。下面重点介绍超标量流水线、指令 Cache 和数据 Cache、浮点单元、转移预测四个方面的新型体系结构特点。

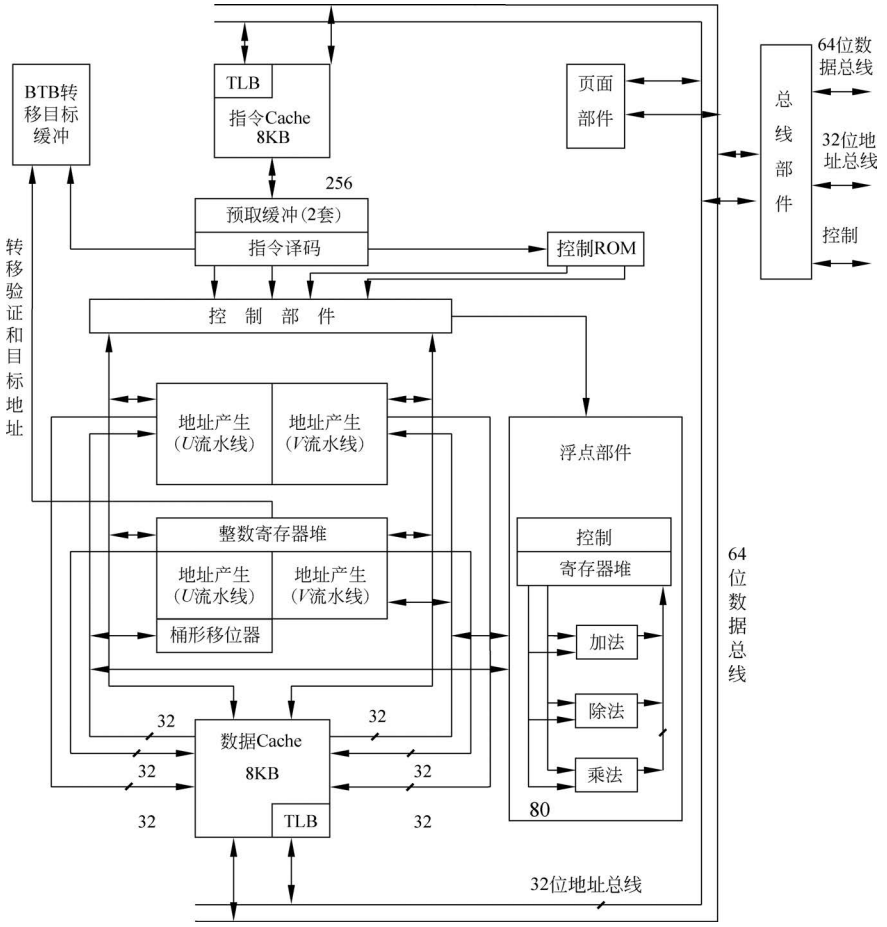


图 5.41 奔腾 CPU 结构框图

1) 超标量流水线

超标量流水线是奔腾系统结构的核心。它由 U 和 V 两条指令流水线构成,每条流水线

都有自己的 ALU、地址生成电路、与数据 Cache 的接口。两个指令预取缓冲器，每个都是 32 字节，负责由指令 Cache 或主存取指令并放入其中。

指令译码器除完成译码指令外，还要完成指令配对检查。两条连续的指令 I_1、I_2 前后被译码，然后判断是否将这一对指令并行发射出去。发射一对指令必须满足如下条件：①两条指令是简单指令；②两条指令不发生数据相关；③每条指令都不同时含有立即数和偏移量；④只有 I_1 允许带有指令前缀。CPU 对 U、V 两条流水线的调度采用按序发射按序完成策略。检查合格的一对指令同时被发射到 U、V 流水线的下一段。如果不满足配对条件，只允许 I_1 指令发射到 U 流水线的下一段。在指令配对条件下，流水线在每个时钟周期内执行两条简单的整数指令，但一般只能执行一条浮点数指令。原因是：浮点数指令流水线是 8 段，而前 4 段与 U、V 流水线共享，而且某些浮点操作数是 64 位，所以浮点数指令不能与整数指令同时执行。

控制 ROM 属于微程序控制器，其中存放一组解释指令操作顺序的微指令代码。

两个地址生成器用于计算存储器操作数地址。各种模式下的逻辑地址最终要转换成物理地址来访问数据 Cache，并用转换后援缓冲器(TLB)来加速这种地址转换过程。寄存器堆有 8 个 32 位整数寄存器，用于地址计算、保存 ALU 的源操作数和目的操作数。

2) 指令 Cache 和数据 Cache

80486 CPU 中有 8KB 的指令和数据共用的 Cache。而奔腾 CPU 则分设指令 Cache 和数据 Cache，各为 8KB。指令 Cache 是只读的，以单端口 256 位(32B)向指令预取缓冲器提供超长指令字代码。数据 Cache 是可读可写的，双端口，每个端口 32 位，与 U、V 两条流水线交换整数数据，或组合成一个 64 位端口与浮点预算部件交换浮点数据。两个 Cache 与 64 位数据、32 位地址的 CPU 内部总线相连接。

两个 Cache 都是 2 路组相联结构，每行 32 字节。数据 Cache 可设置成行写回或全写法方式，并遵守 MESI 协议来维护 L_1 Cache、L_2 Cache 的一致性。

两个 Cache 都是用物理地址。每个 Cache 都有一个后援缓冲器(TLB)，负责将 TLB 命中的线性地址转换成 32 位物理地址。

指令 Cache 与数据 Cache 独立设置是对标量流水线的有力支持，它不仅使指令预取和数据读写能无冲突地同时完成，而且可同时与 U、V 两条流水线分别交换数据。

3) 浮点运算部件

奔腾 CPU 内部包含了一个 8 段的流水浮点运算器。前 4 段为指令预取(IF)、指令译码(D1)、地址生成(D2)、取操作数(EX)，在 U、V 流水线中完成；后 4 段为执行 1(X1)、执行 2(X2)、结果写回寄存器堆(WR)、错误报告(ER)，在浮点运算部件中完成。一般情况下，只能由 U 流水线完成一条浮点数操作指令。

浮点部件支持 IEEE 754 标准的单、双精度格式的浮点数，另外还使用一种称为临时实数的 80 位浮点数。其中有浮点专用加法器、乘法器和除法器，有 8 个 80 位寄存器组成的寄存器堆，内部的数据总线为 80 位宽。对于浮点数的常用指令如 LOAD、ADD、MUL 等采用了新的算法，用硬件来实现，其执行速度是 80486 CPU 的 10 倍多。

4) 动态转移预测技术

执行转移指令时为了不使流水线断流，奔腾采用了动态转移预测技术。转移目标缓冲器(BTB)是一个小容量的 Cache。当一条指令导致程序转移时，BTB 便记录这条指令及其

转移目标地址。以后遇到这条转移指令时,BTB 会依据前后转移发生的历史来预测该指令这次是转移取还是顺序取。若预测为转移取,则将 BTB 记录的转移目标地址立即送出可用。

两个指令预取缓冲器,每个容量为 32 字节,当前总是使用其中一个(假设为缓冲器 1)。当在指令译码(DI)段译出一条转移指令时立即检索 BTB。若预测为"顺序取",则继续从缓冲器 1 取指令;若预测为"转移取",则立即冻结缓冲器 1,启动另一个缓冲器 2,由给出的转移目标地址处开始取分支程序的指令序列。这样,保证了流水线的指令预取步骤永远不会空置。并且预测转移取错误时,正确路径的指令已经在另一个缓冲器中,使流水线的性能损失减至最小。

3. 奔腾 4 CPU 的结构框图

奔腾 4 CPU 结构框图如图 5.42 所示,它是奔腾 CPU 流水线的进一步改进和发展。

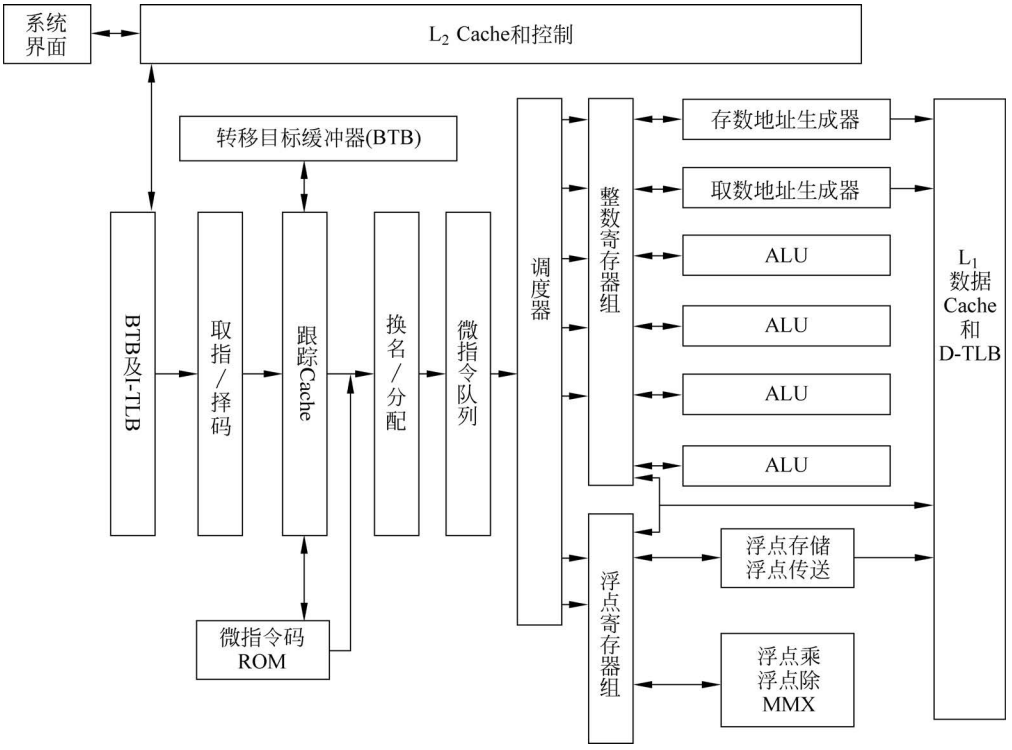


图 5.42 奔腾 4 CPU 结构框图

CPU 的基本操作有如下四点:

(1) 处理器以静态程序的顺序由存储器取指令。

(2) 每条指令译成一个或多个定长的 RISC 指令,它们实际上是微指令。

(3) 处理器在超标量流水线上执行微指令,通过调度器,微指令以乱序方式执行。

(4) 最后,处理器以原程序流的顺序,将每个微指令的执行结果转交到 CPU 的公用寄存器组。

从图 5.42 看到,奔腾 4 的体系结构由外层的 CISC 壳和内部的 RISC 核所组成。内部

的 RISC 微指令至少要通过包含 20 段的流水线，如图 5.43 所示。与奔腾上使用的 5 段流水线相比，增加了很多流水段，这是因为微指令要求多个执行段，导致流水线更长。

1　2	3　4	5	6	7　8	9	10	11	12	13	14	15	16	17	18	19	20
TCNI	TCF	驱动	分配	寄存器换名	排队	调度	调度	调度	派遣	派遣	RF	RF	执行	标志	转检	驱动

图 5.43　奔腾 4 CPU 流水线

下面对流水线的操作做简要说明：

(1) 微指令的生成。奔腾 4 CPU 中包含了一个有序进行的前端，它由转移目标缓冲器(BTB)和指令转移后援缓冲器(I-TLB)单元、取指/译码单元组成。这个前端向“跟踪 Cache”单元的 L_1 指令 Cache 提供指令。从“跟踪 Cache”起，才算是流水线的正式开始。通常，CPU 从“跟踪 Cache”取指令，如未命中，有序前端向它提供新指令。

得到“BTB 和 I-TLB”单元的支援，“取指/译码”单元由 L_2 Cache 取奔腾 4 的机器指令，每次 64 字节。默认的情况是顺序取指令。转移预测逻辑经由“BTB 和 I-TLB”单元可改变操作顺序。I-TLB 将指令指针的线性地址转换为物理地址，以便访问 L_2 Cache。BTB 使用静态转移预测技术来确定下次取哪条指令。

一旦取出指令，先被“取指/译码”单元扫描其字节，以确定指令边界(奔腾指令是变长的)。译码器再将每条机器指令译成 1～4 个微指令，每个微指令是长 118 位的 RISC 型指令。产生的微指令存储于“跟踪 Cache”中。

(2) TCNI。流水线前 2 段 TCNI 的意思是跟踪 Cache 下一指令指针，它负责在“跟踪 Cache”中选择指令。每当在指令流中遇到一条转移指令时，就去检查 BTB，它保存有转移指令的相关信息，以确定是否转移发生？如果是，将转移目标地址作为下一指针；如果否，则顺序取下一条指令。BTB 是一个 4 路组相联 Cache。

(3) TCF。流水线的 3、4 段称为“跟踪 Cache 取指”，它取得由指令译码器译出指令，并将它们装成按程序顺序的微指令序列。少数机器指令要求多于 4 个微指令。这些微指令被传送到“微指令码 ROM”单元，它为每条复杂的机器指令保持一个对应的微指令串组，因此“微指令码 ROM”实际上是一个微程序控制器。

(4) 驱动。流水线第 5 段称为驱动，它负责将译码后的指令由“跟踪 Cache”递交给“换名/分配器”模块。第 20 段也是驱动。设两个驱动段的目的是加快传输时间。

(5) 分配。该段为微指令的执行进行资源分配。每个时钟周期有 3 个微指令到达分配器，且在“重排序缓冲器(ROB)”中分配。ROB 能保存多达 126 个微指令，并含有 128 个硬件寄存器。微指令按序进入 ROB，然后由它出发去排队、被调度、被派遣以及去执行，都将是乱序的。

(6) 寄存器换名。该流水段将对 8 个浮点寄存器和 8 个 EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP 寄存器的引用重新映射到 128 个物理寄存器。其目的是避免体系结构寄存器数量有限引起的虚假数据相关性，保留真实数据相关性。

(7) 微指令排队。在资源分配和寄存器换名之后，微指令进入流水线第 9 段。微指令被放入两个队列之一，并保持在其中直到调度器去取它们。两个队列中一个用于存储器操

作(取数和存数),另一个用于不涉及存储器访问的其他微指令。每个队列遵循先入先出规则。一个微指令是否出队列,与另一个队列的微指令没有次序关系,这个调度器提供了较大灵活性。

(8) 微指令调度和派遣。调度器负责由队列取出微指令并派遣它们去执行。为此,调度器查找那些状态指示已具备的微指令:若它所需的执行单元可用,则调度器取出此微指令并将它派遣到相应的执行单元。如果多个微指令使用同一个执行单元,调度器按队列中顺序逐个派遣它们。此时指令流重新排列了,实际上是乱序执行。调度器有 4 个端口与执行单元连接。其中端口 0 用于整数和浮点运算,端口 1 用于简单整数运算和转移预测失败处理,余下 2 个端口分别用于存储器取数和存数。

(9) RF、执行、标志。RF 是整数和浮点寄存器组的缩写。执行单元从 RF 和 L_1 数据 Cache 中取出所需的数值用于运算,并获得运算的状态标志(如零、负等),它们为转移指令所需要。

(10) 转移检查。第 19 流水段开始转移检查。它将实际转移的结果与预测进行比较,在最后的驱动阶段完成转移检查结果。如果预测是错的,则在各个阶段正在进行的微指令必须由流水线清除出去,将正确的目标地址提供给转移预测器,从而由新的目标地址重新启动整个流水线。

5.7 RISC CPU

5.7.1 RISC 的特点

第一台 RISC(精简指令系统计算机)于 1981 年在美国加州大学伯克利分校问世。它是在继承了 CISC(复杂指令系统计算机)的成功技术,并在克服了 CISC 缺点的基础上发展起来的。

尽管众多厂家生产的 RISC 处理器实现手段有所不同,但是 RISC 概括的三个基本要素是普遍认同的。这三个要素是:①一个有限的简单的指令系统;②CPU 配备大量的通用寄存器;③强调对指令流水线的优化。

RISC 的目标绝不是简单的缩减指令系统,而是使处理器的结构更简单,更合理,具有更高的性能和执行效率,并降低处理器的开发成本。基于三要素的 RISC 的特征是:

(1) 使用等长指令,目前的典型长度是 4 字节。

(2) 寻址方式少且简单,一般为 2～3 种,最多不超过 4 种,绝不出现存储器间接寻址方式。

(3) 只有取数指令、存数指令访问存储器。指令中最多出现 RS 型指令,绝不出现 SS 型指令。

(4) 指令系统中的指令数目一般少于 100 种,指令格式一般少于 4 种。

(5) 指令功能简单,控制器多采用硬布线方式,以期更快的执行速度。

(6) 平均而言,所有指令的执行时间为一个处理时钟周期。

(7) 指令格式中,用于指派整数寄存器的个数不少于 32 个,用于指派浮点数寄存器的个数不少于 16 个。

(8) 强调通用寄存器资源的优化使用。

(9) 支持指令流水并强调指令流水的优化使用。

(10)RISC 技术的复杂性在它的编译程序,因此软件系统开发时间比 CISC 长。

表 5.10 中列出了 RISC 与 CISC 的主要特征对比。

表 5.10 RISC 与 CISC 的主要特征对比

比 较 内 容	CISC	RISC
指令系统	复杂、庞大	简单、精简
指令数目	一般大于 200	一般小于 100
指令格式	一般大于 4	一般小于 4
寻址方式	一般大于 4	一般小于 4
指令字长	不固定	等长
可访存指令	不加限制	只有取数/存数指令
各种指令使用频率	相差很大	相差不大
各种指令执行时间	相差很大	绝大多数在一个周期内完成
优化编译实现	很难	较容易
程序源代码长度	较短	较长
控制器实现方式	绝大多数为微程序控制	绝大多数为硬布线控制
软件系统开发时间	较短	较长

5.7.2 RISC CPU 实例

1. MC88110 CPU 结构框图

MC88110 CPU 是 Motorola 公司的产品,其目标是以较好的性能价格比作为和工作站的通用微处理器。它有 12 个执行功能部件,3 个 Cache 和 1 个控制部件。其结构框图如图 5.44 所示。

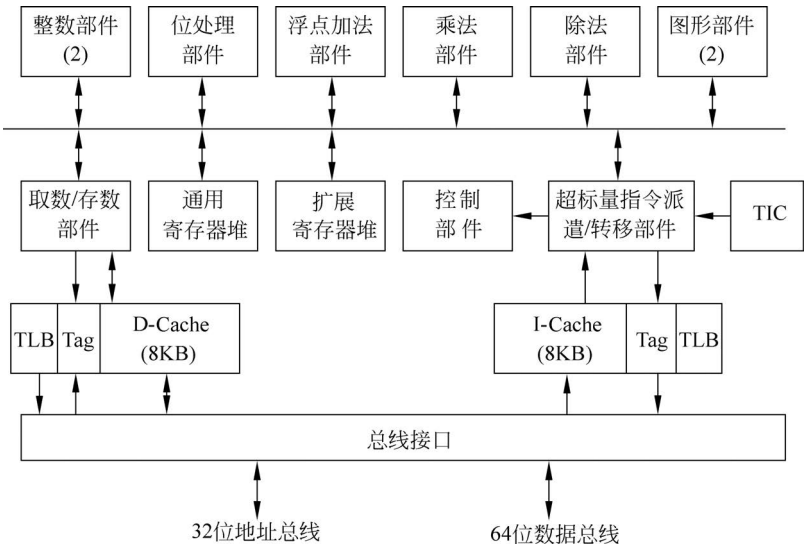


图 5.44 MC88110 CPU 结构框图

在三个 Cache 中,1 个是指令 Cache,1 个是数据 Cache,它们能同时完成取指令和取数据,另一个是目标指令 Cache(TIC),它用于保存转移目标指令。

两个寄存器堆:一个是通用寄存器堆,用于整数和地址指针,其中有 $R_0 \sim R_{31}$ 共 32 个寄存器(32 位长);另一个是扩展寄存器堆,用于浮点数,其中有 $X_0 \sim X_{31}$ 共 32 个寄存器(长度可以是 32 位、64 位或 80 位)。

12 个执行功能部件是:取数/存数(读写)部件、整数运算部件(2 个)、浮点加法部件、乘法部件、除法部件、图形处理部件(2 个)、位处理部件、用于管理流水线的超标量指令派遣/转移部件。

所有这些 Cache、寄存器堆、功能部件,在处理器中通过六条 80 位宽的内部总线相连接。其中 2 条源 1 总线,2 条源 2 总线,2 条目标总线。

2. MC88110 的指令流水线

由于 MC88110 是超标量流水 CPU,所以指令流水线在每个机器时钟周期完成两条指令。流水线分为三段:取指和译码(F&D)段、执行(EX)段、写回(WB)段,如图 5.45 所示。

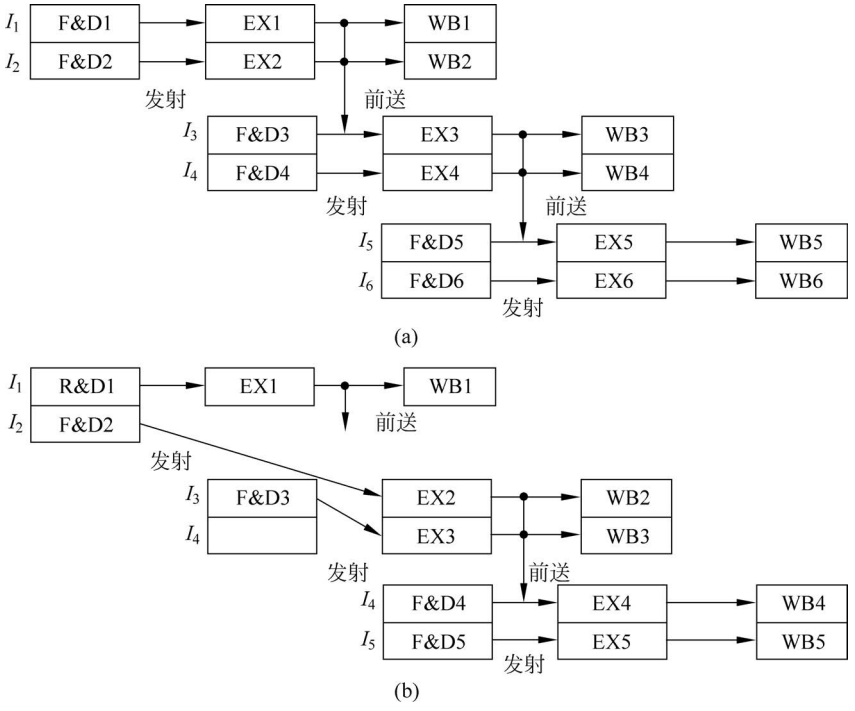


图 5.45 MC88110 CPU 的超标量指令流水线

F&D 段需要一个时钟周期,完成由指令 Cache 取一对指令并译码,并从寄存器堆取操作数,然后判断是否把指令发射到 EX 段。如果所要求的资源(操作数寄存器、目标寄存器、功能部件)发生资源使用冲突,或与先前指令发生数据相关冲突,或转移指令将转向新的目标指令地址时,则 F&D 段不再向 EX 段发射指令,或不发射紧接转移指令之后的指令。

EX 段对于大多数指令只需一个时钟周期,某些指令可能多于一个时钟周期。EX 段执行的结果在 WB 段写回寄存器堆,WB 段只需时钟周期的一半。为了解决数据相关冲突,EX 段执行的结果一方面在 WB 段写回寄存器堆,另一方面经定向传送电路提前传送到

ALU，可直接被当前进入 EX 的指令所使用。图 5.45(a)表示 MC88110 超标量流水线正常运行情况。

3. 指令动态调度策略

MC88110 采用按序发射、按序完成的指令动态调度策略。指令派遣单元总是发出单一地址，然后从指令 Cache 取出此地址及下一地址的两条指令。译码后总是力图同一时间发射这两条指令到 EX 段。若这对指令的第一条指令由于资源冲突或数据相关冲突，则这一对指令都不发射，两条指令在 F&D 段停顿，等待资源的可用或数据相关的消除。若是第一条指令能发射第二条指令不能发射，则只发射第一条指令，而第二条指令停顿并与新取的指令之一进行配对等待发射，此时原第二条指令作为配对的第一条指令对待。可见，这样实现的方式是按序发射，图 5.45(b)示出了指令配对情况。

为了判定能否发射指令，使用了记分牌方法。记分牌是一个位向量，寄存器堆中每个寄存器都有一个相应位。每当一条指令发射时，它预约的目的寄存器在位向量中的相应位上置“1”，表示该寄存器“忙”。当指令执行完毕并将结果写回此目的寄存器时，该位被清除。于是，每当判定是否发射一条指令(STO 存数指令和转移指令除外)时，一个必须满足的条件是：该指令的所有目的寄存器、源寄存器在位向量中的相应位都已被清除。否则，指令必须停顿等待这些位被清除。为了减少经常出现的数据相关，流水线采用了如前面所述的定向传送技术，将前面指令执行的结果直接送给后面指令所需此源操作数的功能部件，并同时将位向量中的相应位清除。因此，指令发射和定向传送是同时进行的。

如何实现按序完成呢？因为执行段有多个功能部件，很可能出现无序完成的情况。为此，MC88110 提供了一个指令执行队列，称之为历史缓冲器。每当一条指令发射出去，它的副本就被送到 FIFO 队尾。队列最多能保存 12 条指令。只有前面的所有指令执行完，这条指令才到达队首。当它到达队首并执行完毕后才离开队列。

对于转移处理，MC88110 使用了延迟转移法和目标指令 Cache(TIC)法。延迟转移是个选项(.n)。如果采用这个选项(指令如 bcnd.n)，则跟随在转移指令后的指令将被发射。如果不采用这个选项，则在转移指令发射之后的转移延迟时间片内没有任何指令被发射。延迟转移通过编译程序来调度。

TIC 是一个 32 项的全相联 Cache，每项能保存转移目标路径的前两条指令。当一条转移指令译码并命中 Cache 时，能同时由 TIC 取来它的目标路径的前面两条指令。

【例 5.6】 超标度为 2 的超标量流水线结构模型如图 5.46(a)所示。它分为 4 个段，即取指(F)段、译码(D)段、执行(E)段和写回(W)段。F、D、W 段只需 1 个时钟周期完成。E 段有多个功能部件，其中取数/存数部件完成数据 Cache 访问，只需 1 个时钟周期；加法器完成需 2 个时钟周期，乘法器需 3 个时钟周期，它们都已流水化。F 段和 D 段要求成对的输入。E 段由内部数据定向传送，结果生成即可使用。

现有如下 6 条指令序列，其中 I_1、I_2 有 RAW 相关，I_3、I_4 有 WAR 相关，I_5、I_6 有 WAW 相关和 RAW 相关。

I_1 LAD R_1,A；取数 $M(A)\rightarrow R_1$，$M(A)$是存储单元

I_2 ADD R_2,R_1；$(R_2)+(R_3)\rightarrow R_2$

I_3 ADD R_3,R_4；$(R_3)+(R_4)\rightarrow R_3$

I_4 MUL R_4,R_5；$(R_4)\times(R_5)\rightarrow R_4$

I_5 LAD R_6,B；取数 $M(B)\rightarrow R_6$，$M(B)$是存储器单元

I_6 MUL R_6,R_7；$(R_6)\times(R_7)\rightarrow R_6$

请画出：(1)按序发射按序完成各段推进情况图。

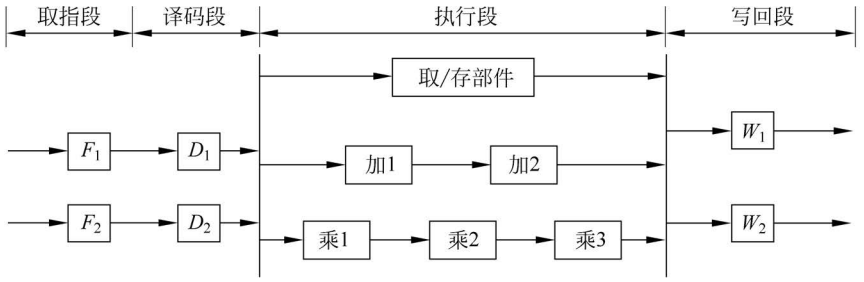
(2) 按序发射按序完成的流水线时空图。

解：(1)按序发射按序完成各段情况推进图如图 5.46(b)所示。由于 I_1、I_2 间有 RAW 相关，I_2 要推迟一个时钟才能发射。类似的情况也存在于 I_5、I_6 之间。

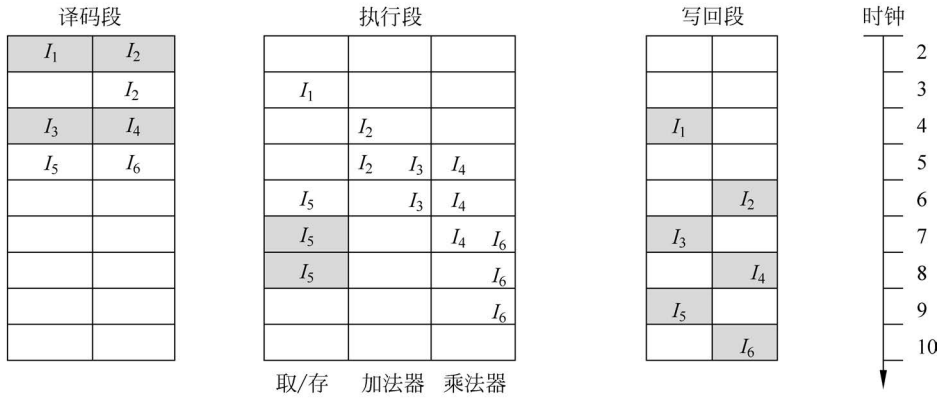
I_3、I_4 之间有 WAR 相关，但按序发射，即使 I_3、I_4 并行操作，也不会导致错误。

I_5、I_6 间还有 WAW 相关，只要 I_6 的完成放在 I_5 之后，就不会出错。注意，I_5 实际上已在时钟 6 执行完毕，但一直推迟到时钟 9 才写回，这是为了保持按序完成。超标量流水线完成 6 条指令的执行任务总共需要 10 个时钟周期。

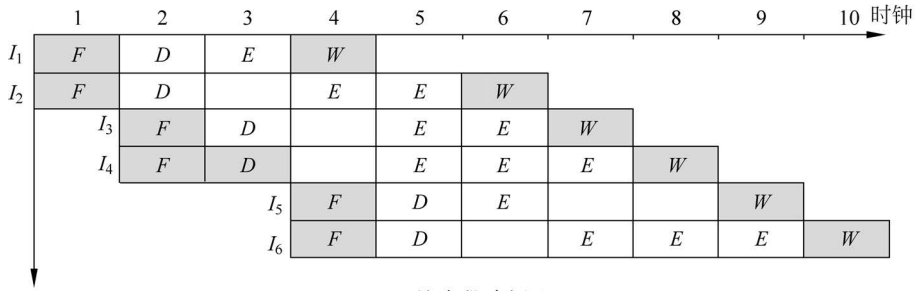
(2) 根据各段推进情况图可画出流水线时空图，如图 5.46(c)所示。



(a) 超标量流水模型结构



(b) 各阶段推进情况



(c) 流水段时空图

图 5.46 超标量流水线各段推进情况图和时空图

本章小结

CPU是计算机的中央处理部件，具有指令控制、操作控制、时间控制、数据加工等基本功能。

早期的CPU由运算器和控制器两大部分构成。随着集成电路技术的发展，CPU芯片变成由运算器、控制器和Cache三部分组成，其中还有浮点运算器和存储管理等部件。CPU中主要包含的寄存器有六类，具体是：指令寄存器、程序计数器、地址寄存器、数据缓冲寄存器、通用寄存器和状态条件寄存器。

指令周期包括CPU从存储器中取出一条指令并执行这条指令所用的时间总和。时序信号产生器提供CPU周期所需的时序信号，操作控制器利用这些时序信号进行定时，有条不紊的取出一条指令并执行这条指令。本章对典型指令进行了详细介绍。

微程序设计技术是利用软件思想设计操作控制器的一门技术，具有规整性、灵活性、可维护性等一系列优点。

不论微型机还是超级计算机，并行处理技术已成为计算机技术发展的主流。并行处理技术主要有三种形式：时间并行、空间并行、时间并行＋空间并行。流水CPU就是一种非常经济而实用的并行技术。流水技术中主要问题是资源相关、数据相关和控制相关，为此需要采取相应的技术对策，才能保证流水线畅通。

习题

题库

习题答案

1. 计算机操作的最小时间单位是_____。
   A. 时钟周期　　B. 指令周期　　C. CPU周期　　D. 微指令周期
2. 程序状态字寄存器用来存放_____。
   A. 算术运算结果　　B. 逻辑运算结果
   C. 运算类型　　D. 算术、逻辑运算及测试指令的结果状态
3. 指令译码器对_____进行译码。
   A. 整条指令　　B. 指令中的操作码字段
   C. 指令的地址　　D. 指令中的操作数字段
4. CPU组成中不包括_____。
   A. 指令寄存器　　B. 指令译码器　　C. 程序计数器　　D. 地址译码器
5. 在取指令之后，程序计数器中存放的是_____。
   A. 当前指令的地址　　B. 不转移时下一条指令的地址
   C. 程序中指令的数量　　D. 指令的长度
6. 微程序控制器中，微程序的入口地址是由_____形成的。
   A. 机器指令的地址码字段　　B. 微指令的微地址码字段
   C. 机器指令的操作码字段　　D. 微指令的操作码字段

7. 微程序存放在_____中。

A. 控制存储器　　B. RAM　　C. 指令寄存器　　D. 内存储器

8. 关于微指令的编码方式中,下面叙述正确的是_____。

A. 水平编码法和垂直编码法不影响指令的长度

B. 一般情况下,直接表示法的微指令位数多

C. 一般情况下,最短编码法的位数多

D. 以上说法都不对

9. 若机器 M 的主频为 1.5GHz,在 M 上执行程序 P 的指令条数 5×10^5,P 的平均 CPI 为 1.2,则 P 在 M 上的指令执行速度和用户 CPU 时间分别为_____。

A. 0.8GIPS、0.4ms　　B. 0.8GIPS、0.4μs

C. 1.25GIPS、0.4ms　　D. 1.25GIPS、0.4μs

10. 假设主机框图如图 5.47 所示,各部分间的连线表示数据通路,箭头表示信息传递方向。

求:(1) 标明图中 a、b、c、d 四个寄存器的名字。

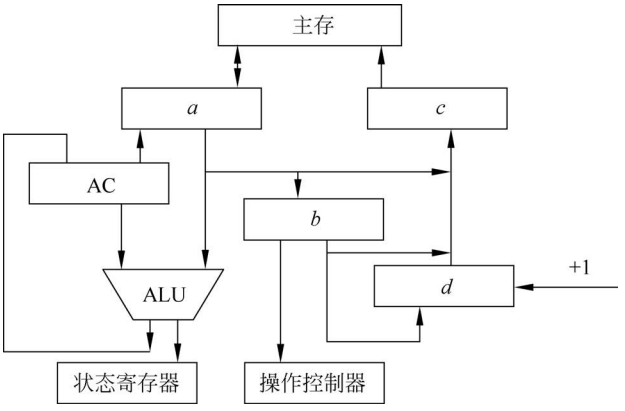


图 5.47　某主机框图

(2) 简述取指令的数据通路。

(3) 简述取数指令和存数指令执行阶段的数据通路。

11. 已知某机器采用微程序控制方式,其控制存储器容量为 512×48 位。微指令字长 48 位,微指令可在整个存储器中实现转移,可控制微程序转移的条件共 4 个(直接控制),微指令采用水平型格式,如图 5.48 所示。

(1) 求微指令中三个字段分别应为多少位?

(2) 画出围绕这种微指令格式的微程序控制器逻辑框图。

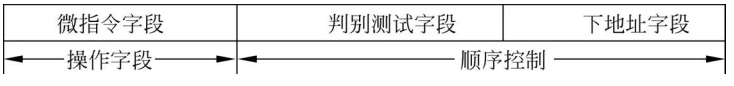


图 5.48　水平型微指令格式

12. 水平微指令和垂直微指令各有什么特征?

13. 微控制器方式下得到下一条微指令地址可能有哪些方式?

14. 设某机主频为 8MHz,每个机器周期平均含 2 个时钟周期,每条指令平均有 2.5 个

机器周期，试问：(1)该机的平均指令执行速度为多少 MIPS？(2)若机器主频不变，但每个机器周期平均含 4 个时钟周期，每条指令平均有 5 个机器周期，则该机的平均指令执行速度又是多少 MIPS？(3)由此可得出什么结论？

15. 某机采用微程序控制方式，微指令字长 24 位，采用水平型编码控制的微指令格式，采用断定方式，共有微命令 30 个，构成 4 个相斥类，各包括 5 个、8 个、14 个和 3 个微命令，可测试的外部条件有 3 个。要求：

(1) 设计出微指令的具体格式。

(2) 控制存储器的容量应为多少？