第 5 章　处 理 器

在第 2 章中看到，通过检查可知，由处理器（Processor）执行的功能可以划分成以下三个单元（参见图 5.1）。

（1）**控制器**（Control Unit），它负责：

① 从存储器中获取指令。

② 解码这些指令。

③ 对微命令进行排序，确保所识别的指令的操作。

（2）**处理单元**（Processing Unit），它产生基本算术和逻辑运算的结果。

（3）**寄存器**（Register），它用于把处理单元处理的信息存储进存储器中。

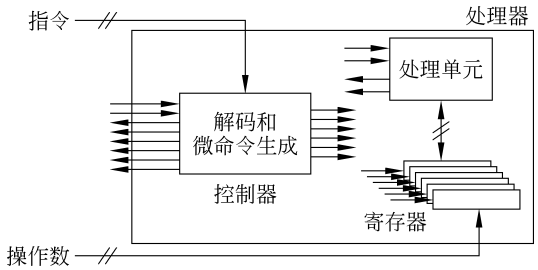

图 5.1　处理器的组件

在按顺序发送的微命令的环境中，将使用以下术语。

（1）**微指令**（Microinstruction）：一个布尔型（Boolean）矢量，表示这些来源于控制器的微命令的逻辑状态。

（2）**微操作**（Micro-Operation）：微指令的执行（参见图 5.2）。

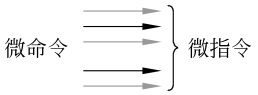

图 5.2　微命令和微指令

本章在介绍了一个指令执行的示例之后，将详细描述控制器的组件及其设计中所涉及的概念。

5.1　控 制 总 线

我们强调了处理器任务的必要性：读取指令、解码、执行、读取指令等。这种循环可以通过外部信号改变，这些信号是在**控制总线**（Control Bus）的线路上传送的。我们已经提过了等待线路的情况（参见图 5.3）。

还有其他一些来源于处理器的信号，用于指示处理器的内部状态或者给外部设备提供信息（图 5.4 中显示了确认的示例），它们是在这条相同的总线上传送的。

通常通过触发器来记住控制线路的状态，然后反遇出激活它们的事件。

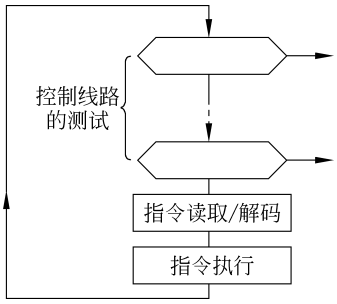

图 5.3 控制线路的测试

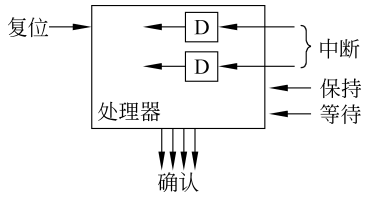

图 5.4 控制线路

5.1.1 复位线路

复位（Reset）线路将导致处理器“重新引导”。当这条线路改变状态时，将初始化多个触发器和内部寄存器，然后获得引导地址。

（1）直接：利用给定的值初始化程序计数器（Intel I8080 和 Intel I8086 微处理器就是这样，对于前者来说，将把程序计数器复位为 0；对于后者来说，其物理（Physical）引导地址是 $FFFF0_{16}$）。

（2）间接：程序计数器在加载时具有一个值，它是处理器从存储器中的某个固定地址（例如，8 位 Motorola MC6800 微处理器中的 $FFFE_{16}$ 和 $FFFF_{16}$，参见图 5.5）获取的。

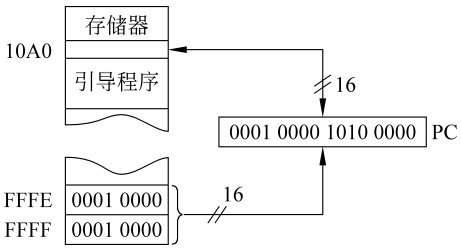

图 5.5 Motorola MC6800 中的引导方式

5.1.2 保持线路

当除了处理器以外的某个单元（另一个处理器或交换器）尝试控制总线时，就会使用**保持**（Hold）线路。如果这个尝试成功，处理器就必须放弃访问总线。协议使用控制总线的两条线路：HOLD（保持）线路和 HOLD Acknowledge（保持确认）线路。为了响应这个保持请求，处理器将发送一个确认，指示数据和地址总线切换为“高阻抗”状态。然后就称处理器处于**空闲模式**（Idle Mode）。

5.1.3 等待控制线路

如我们已经见过的，当处理器（或者另一个输入输出设备）尝试获得长时间访问存储器的权限时，**等待控制**（Wait Control）线路就被证明是必要的。这种 **WAIT**（**等待**）**控制**

线路或 **READY**（**就绪**）**线路**允许处理器在读或写操作期间插入**等待周期**（Wait Cycle）或**等待状态**（Wait State），同时等待必需的数据变得可用，或者输入输出单元做好准备。

5.1.4　中断线路

中断（Interrupt）线路用于把外部事件考虑在内。可以执行一个指令序列来响应这种请求。这些事件的处理被称为中断，将在第 6 章介绍输入输出时讨论它。

5.1.5　概念图

受 Motorola MC6800 微处理器的启发而得到的操作图可以表示为如图 5.6 所示。

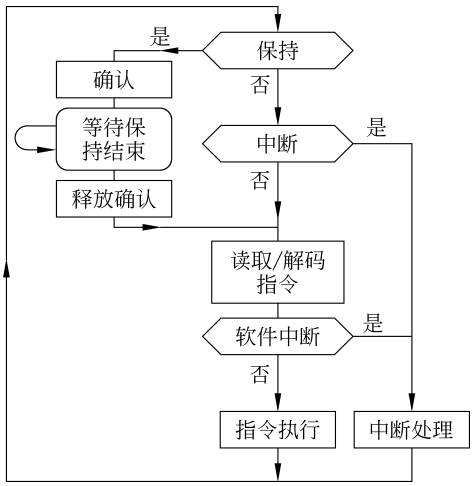

图 5.6　操作图的示例

执行指令的完整阶段被称为**指令周期**（Instruction Cycle），这个周期的长度依赖于机器的体系结构。

5.2　指令的执行：一个示例

本节将利用一个示例演示刚才所介绍的，这个示例基于 Intel I8080 微处理器的内部操作（参见图 5.7）[INT 79]。依据 Intel 定义的表示法，我们所选的指令是 add M。

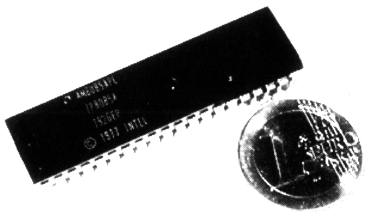

图 5.7　Intel I8085

这个指令对寄存器 A（累加器）与存储器字（其地址由 HL 寄存器（用作地址寄存器）的内容提供）的内容执行求和。结果存储在寄存器 A 中（参见图 5.8）。

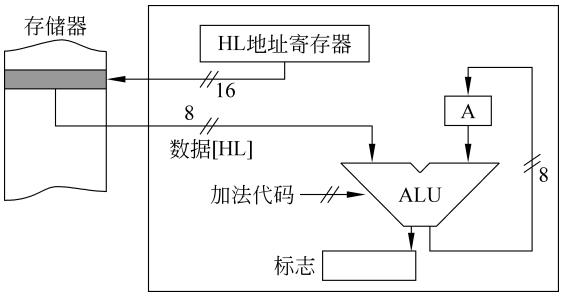

图 5.8　add M 指令的操作图

这个指令将引发两个存储器访问：一个是读取指令；另一个是把操作数读入存储器中。

之所以选择这个示例，是因为 Intel I8080 微处理器的操作非常简单，也是因为每条指令中的每个步骤的描述都可以在 Intel 公司提供的文档中找到。

add M 被划分在三个机器周期中：M1、M2 和 M3。依据以下模式，将每个周期划分成“状态”T1、T2 等。

		M1					
		T1	T2	T3	T4	T5	→
add M	代码 86_{16}	PC OUT 状态	PC++	INST →IR	(A) → ACT	☒	

→	M2			M3		
	T1	T2	T3	T1	T2	T3
	HL OUT 状态	数据	→TMP		(ACT)+(TMP) →A	☒

机器简化的内部结构如图 5.9 所示。

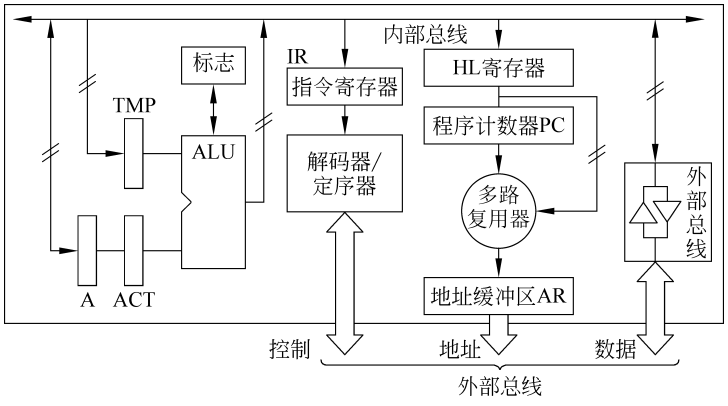

图 5.9　CPU 的简化结构

这幅图显示了主要的寄存器：指令寄存器 IR（8 位）、标志（8 位）、程序计数器（16

位）、寄存器 HL（16 位）、寄存器 A（8 位）和两个缓冲寄存器，分别称为 ACT（16 位）和 TMP（8 位）。地址缓冲区 AR（16 位）可以临时记住需要提供给地址总线的地址。使用双向**驱动器**（Driver）给数据总线供电，它们是**三态**（Three State）驱动器。

5.2.1　指令的执行

Intel 对**机器周期**（Machine Cycle）定义了一个顺序。这意味着每条指令都被划分在主要周期（1～5）中。在这里，求和指令被认为包含三个机器周期。其中每个周期都包含由基本时钟定义的 2～5 个**基本周期**（Elementary Cycle）（“状态”）。执行每个基本周期所涉及的微命令通过 μ_{jk}^{i} 表示，其中，j 是机器周期编号，k 是基本周期编号，i 则是线路编号。

（1）第一个机器周期：这对应于在存储器中获取指令，并且包含以下 4 个基本周期。

① 把程序计数器 PC 的内容传输给地址缓冲区 AR。与此同时，数据总线提供关于当前机器周期的指示，在这里是指令获取周期。这个阶段（通过 M1-T1 表示（参见图 5.10））涉及以下微代码。

- μ_{11}^{1} 命令多路复用器 Mux 把 PC 的内容转移给 AR。
- μ_{11}^{2} 命令加载缓冲区 AR。
- μ_{11}^{3} 指示位于周期 T1 中。这个信息显示在控制总线的线路上的名称 SYNC 下面，并且可以被外部逻辑使用。
- μ_{11}^{4} 命令“发送”方向上的数据总线驱动器等。

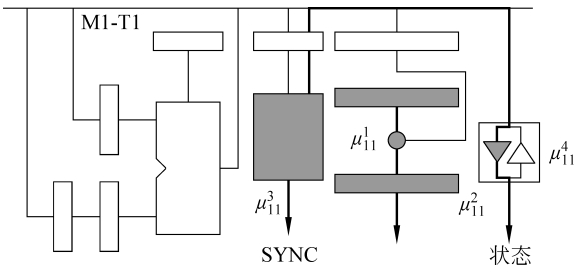

图 5.10　M1-T1 阶段

这些当然不是所有的微命令，但是这个列表很好地描述了信号在确保指令执行方面所起的作用。

② M1-T2 阶段包含递增程序计数器的内容。在处理占据单个字的指令时，程序计数器的这种递增的预期可以节省时间，并为存储器提供额外的时间来设置它自身（参见下一个阶段）。在微命令当中，将提及 μ_{12}^{1}、μ_{12}^{2} 以及 μ_{12}^{3} 或 $\overline{\text{RD}}$，其中，μ_{12}^{1} 指示程序计数器“递增模式”，μ_{12}^{2} 用于激活递增，μ_{12}^{3} 或 $\overline{\text{RD}}$ 则用于验证对存储器的访问（参见图 5.11）。

在这个阶段，将测试 READY 线路。如果它被置 0，就意味着存储器没有准备好递送指令代码，处理器将引入基本等待周期 TW。在这个阶段还将测试保持线路。

③ 在 M1-TW 等待阶段，在数据总线上期望的指令并不认为是稳定的。因此，它可能不会进入指令寄存器中（参见图 5.12）。

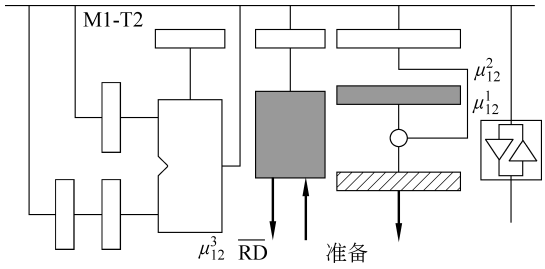

图 5.11 M1-T2 阶段

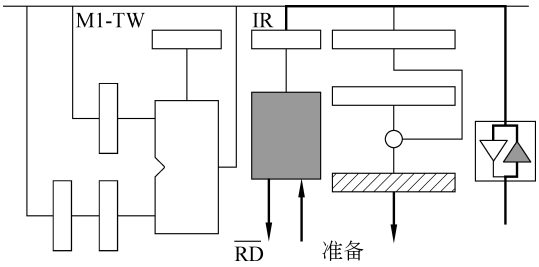

图 5.12 M1-TW 阶段

④ M1-T3 阶段（参见图 5.13）将实际地获取指令，它将被加载进指令寄存器 IR 中。在相关的微命令当中，可以提及以下微命令。

- μ_{13}^1 命令，用于数据总线驱动器。
- $\overline{RD}$ 存储器控制信号。
- μ_{13}^2 加载指令寄存器 IR。

从 M1-T1 阶段起，包含指令的存储器字的地址就已经位于总线上。

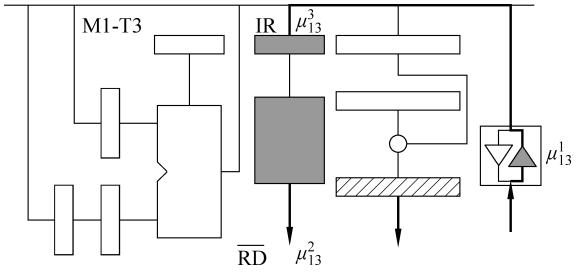

图 5.13 M1-T3 阶段

⑤ M1-T4 阶段标记指令的实际展开的开始。它对应于把寄存器 A 的内容传输到缓冲区 ACT，如以前所解释的，这个传输起着阻止发生操作意外事故的作用（参见图 2.10）。微命令 μ_{14}^1 触发 ACT 的加载（求和代码已经位于算术逻辑运算器上）（参见图 5.14）。

（2）第二个机器周期：这个周期由三个基本周期组成，对应于用于读取存储器中的操作数的机器周期。

① M2-T1 阶段（参见图 5.15）包括指示周期类型、**存储器读取**（Memory Read），并且与 M1-T1 阶段（**指令获取**（Instruction Fetch））的各个方面完全相同。

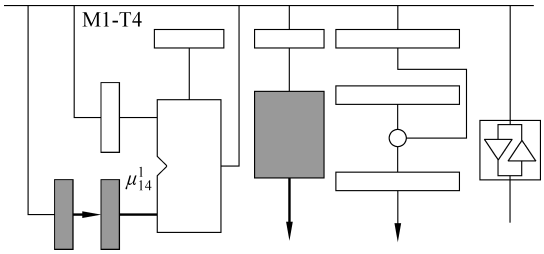

图 5.14　M1-T4 阶段

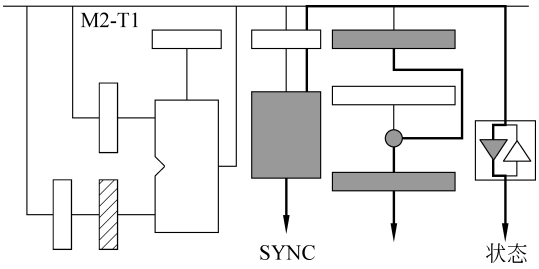

图 5.15　M2-T1 阶段

存储器中的操作数的地址（包含在 HL 中）存储在缓冲区 AR 中。

② 在 M2-T2 阶段（参见图 5.16），将测试 READY 线路，并根据需要插入等待周期。

③ M2-TW 等待阶段与 M1-TW 阶段的各个方面完全相同。

④ M2-T3 阶段（参见图 5.17）包括利用以下微命令获取第二个操作数：

- μ_{23}^{1}命令，用于数据总线驱动器。
- $\overline{\text{RD}}$，控制用于存储器的信号。
- μ_{23}^{2}，用于加载 TMP。

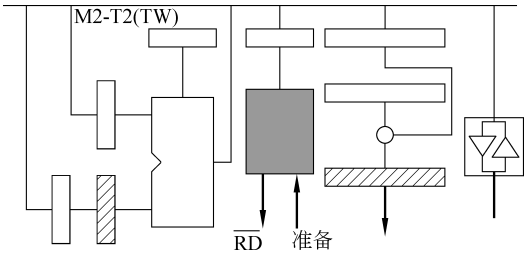

图 5.16　M2-T2 阶段

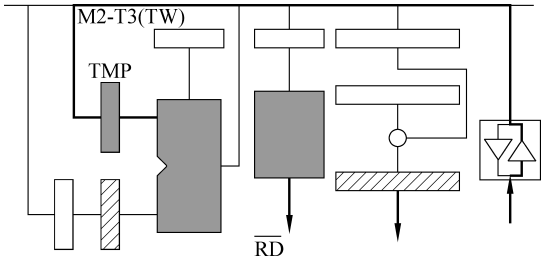

图 5.17　M2-T3 阶段

从 M2-T1 阶段起，包含操作数的存储器字的地址就已经位于地址总线上。就像在读操作中一样，将为存储器生成读命令，并且设置数据总线驱动器。唯一的区别是寄存器的加载：用 TMP 代替 IR。

（3）第三个机器周期：可以将其描述为指令执行周期。它包含以下两个基本周期。

① 在第一个到最后一个阶段即 M3-T1 阶段（参见图 5.18），当前操作不会发生任何事情。人为地引入它只是为了后面的指令开启它的周期。

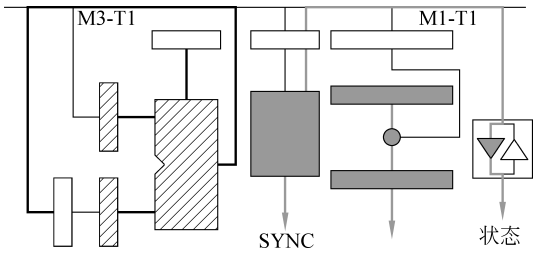

图 5.18　M3-T1 阶段（程序中的下一条指令的 M1-T1 阶段）

② 最后一个阶段即 M3-T2 阶段（参见图 5.19）包括从寄存器 A 获得求和运算的结果，并设置标志。在使用的微命令当中，可以提及以下微命令。

- μ_{32}^{1}，用于寄存器 A 的加载。
- μ_{32}^{2}，用于标志寄存器的加载。

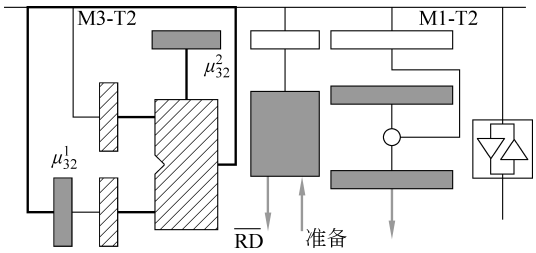

图 5.19　M3-T2 阶段（程序中的下一条指令的 M1-T2 阶段）

连续操作之间的重叠可以显著改进性能。对于求和指令，可以从所需的 9 个基本周期中获得两个基本周期的好处（一旦包括等待周期的话）。

5.2.2　时序图

我们将继续讨论上一节中研究的指令执行中涉及的命令，并且现在将在图 5.20 中描绘的**时序图**（Timing Diagram）中表示它们。这幅时序图基于 Intel 公司的 I8080 和 I8085 处理器。$\overline{RD}$ 线路实际上是随 Intel I8085 引入的。在 Intel I8080 中，使用 DBIN（Data Bus IN，数据总线 IN）线路来验证读取传输。

在 x 轴上表示时间，并在 y 轴上表示少数几个命令的逻辑状态。我们希望更详细地观察，尤其是在 M1-T1 到 M1-T3 阶段，它们对应于指令获取。

（1）在 M1-T1 阶段，涉及的微命令如下。

① μ_{11}^{1} 命令，用于多路复用器 Mux，在 M1-T1 阶段假定它等于 1。

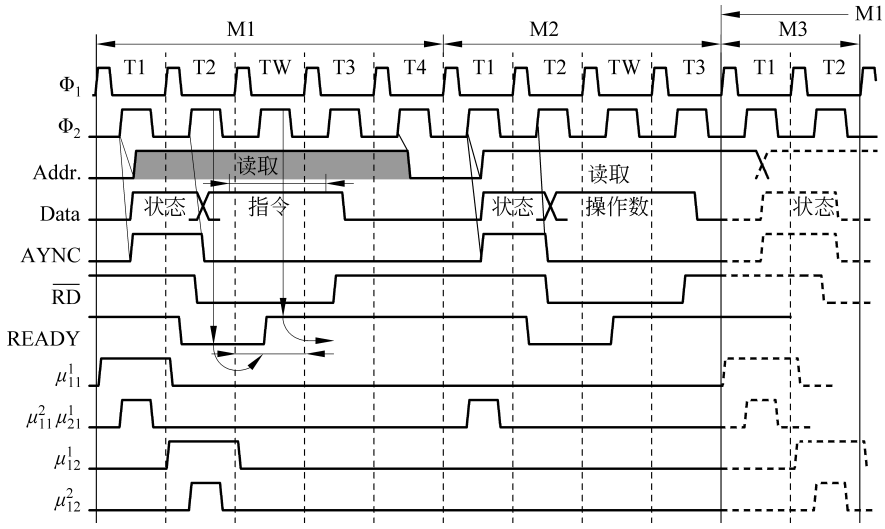

图 5.20　时序图

② μ_{11}^{2} 命令，用于加载缓冲区 AR，它等于 Φ_2。

在利用 HL 的内容加载 AR 的阶段，也使用这些命令获取操作数。我们假定 μ_{11}^{1}=0。

（2）在 M1-T2 阶段，指令的地址（阴影区域）在地址总线上还不是**有效**（Valid）的。为了递增程序计数器，将**递增**（Increment）模式 μ_{12}^{1} 设置为 1，然后使用 Φ_2 生成一个来自时钟 μ_{12}^{2} 的脉冲。

（3）以此类推。

5.3　定序器的构成

定序器（参见图 5.21）是一种生成微命令的自动机。这种自动机或**时序**（Sequential）机获取以下输入。

（1）指令代码，或者如果我们要区分**解码器**（Decoder）与**定序器**（Sequencer），那么它将获取解码器的输出。

（2）关于处理器的状态（特别是标志）的信息。

（3）控制总线的线路状态。

（4）来自时钟的信号。

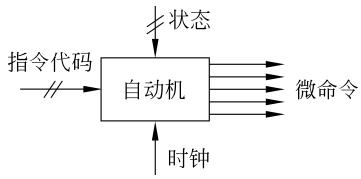

图 5.21　定序器

这种时序机的设计可以通过以下几个方面实现。

（1）基于硬连接的自动机的传统合成方法。

（2）基于微编程概念的方法。

（3）上述两种方法相结合。

5.3.1　传统的合成方法

传统的合成方法在很大程度上依赖于设计师的经验，包括对指令执行的不同阶段进行

编码。为了加以说明，我们将再次考虑用作一个示例的指令，它甚至简化了更多的处理器的操作。设 C_n 是唯一标识指令的变量，ϕ_{ij} 是规范地对 M_i-T_j 阶段进行编码的变量（参见图 5.22）。当然，它取自于彼此独立地合成所有指令的问题。任务将会非常困难。

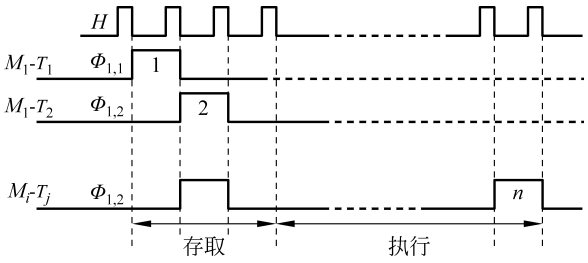

图 5.22 ϕ_{ij} 变量

用于多路复用“PC/HL”命令（单独用于该指令）的方程式可以写成：

$$\mu_{12}^2 = C_n \times \phi_{11} = \phi_{11}$$

因为 μ_{12}^2 独立于 C_n，这是由于无论指令是什么，那个阶段都会执行自身。注意：这可能会导致简化（最简单的表达式并不总是对应于最简单的设计）。用于寄存器 AR 的加载命令可能是：

$$\text{loadAR} = \phi_{11} \times \Phi_2 + C_n \times \phi_{21} \times \Phi_2$$

由于 incrPC 对于指令代码是独立的，用于程序计数器的递增命令将写成 incrPC = ϕ_{12}。

如果我们知道怎样把所有这些命令都写成方程式，那么我们的合成问题将简化成生成 ϕ_{ij}。然后可以绘制对应的时序机的**状态图**（State Diagram）（参见图 5.23）。

为这幅图中的跃迁设置条件的函数依赖于指令和条件代码。例如，条件 C_1 只对“空”指令（不做任何事情的指令或 nop）为真，而它的补集将对所有其他的指令为真。这样一条指令将生成序列 ϕ_{11}-ϕ_{12}-ϕ_{13}。序列“状态 1-状态 2-状态 3”将不可压缩，并且是所有指令的公共序列。

合成过程依赖于在进行状态编码时所达到的变量复杂性模型，其后接着用于状态变量和输出的逻辑方程式。图 5.24 显示了所得到的定序器的操作。

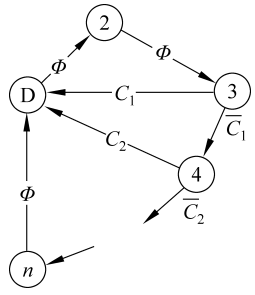

图 5.23 ϕ_{ij} 生成器的状态图

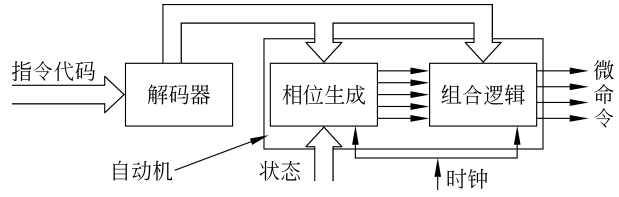

图 5.24 硬连接的定序器的操作图

这种方法的主要难点表现在设计（每次修改都需要把整个合成过程再执行一遍）、结果优化（最小化元素数量）以及结果测试这些方面。