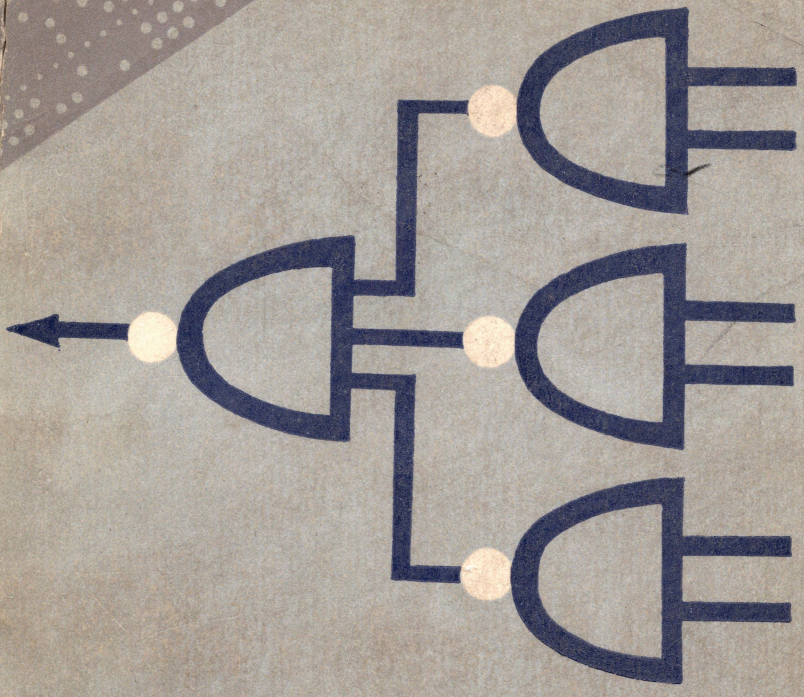


# 8080程序设计在逻辑设计中的应用

〔美〕亚当·奥斯本著·人民邮电出版社

封面设计：潘　攀



科技新书目：48-124
统一书号：15045
总2704—有5288
定　　价：1.70 元

# 8080程序设计在逻辑设计中的应用

〔美〕亚当·奥斯本 著
赵 辰 等译

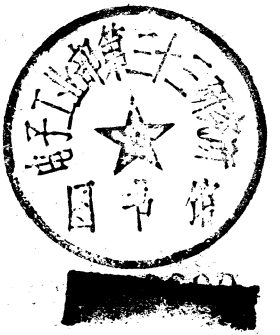


人民邮电出版社

# 8080程序设计在逻辑设计中的应用

〔美〕亚当·奥斯本 著

赵 辰 等译






人民邮电出版社

# 8080 PROGRAMMING FOR LOGIC DESIGN

by Adam Osborne

1976

## 内 容 提 要

本书是《微型计算机入门》丛书的续篇，主要内容是讲述利用汇编语言程序来代替组合逻辑的程序设计。内容叙述具体、实用。读者可通过举出的典型例子，领会编制汇编语言程序的要点，熟悉常用典型程序的用法。书中还结合实际讲解了子程序、宏程序和中断程序的编制，详细地介绍了 Intel 8080 每条指令的功能和用法。同时还介绍了一些常用的子程序。这些对于初次接触微型计算机应用和汇编语言程序设计的读者来说，都有实用参考价值。本书可供学习程序编制的科技人员、大专院校师生及初学微型计算机应用的工作人员阅读。

本书由赵辰同志担任全书的译校工作，参加翻译的有王钟馨、幸云辉、陈郁青、谢楚屏、李怀诚、孙德生、高锡武等同志。

## 8080程序设计在逻辑设计中的应用

〔美〕亚当·奥斯本 著

赵 辰 等译

*

人民邮电出版社出版
北京东长安街27号
北京印刷一厂印刷
新华书店北京发行所发行
各地新华书店经售

*

开本：787×1092 1/32　　1983年9月第一版
印张：13 24/32　页数：220　1983年9月北京第一次印刷
字数：310 千字　插页：1　印数：1—18,000 册
统一书号：15045·总2704-有5288
定价：1.70元

# 附录 A（续）

| 十六进制数表示 | ASCII (7 位) | EBCDIC (8 位) |
|---|---|---|
| CC | | |
| CD | | |
| CE | | |
| CF | | |
| D0 | | |
| D1 | | J |
| D2 | | K |
| D3 | | L |
| D4 | | M |
| D5 | | N |
| D6 | | O |
| D7 | | P |
| D8 | | Q |
| D9 | | R |
| DA | | |
| DB | | |
| DC | | |
| DD | | |
| DE | | |
| DF | | |
| E0 | | |
| E1 | | |
| E2 | | S |
| E3 | | T |
| E4 | | U |
| E5 | | V |
| E6 | | W |
| E7 | | X |
| E8 | | Y |
| E9 | | Z |
| EA | | |
| EB | | |
| EC | | |
| ED | | |
| EE | | |
| EF | | |
| F0 | | 0 |
| F1 | | 1 |
| F2 | | 2 |
| F3 | | 3 |
| F4 | | 4 |
| F5 | | 5 |
| F6 | | 6 |
| F7 | | 7 |
| F8 | | 8 |
| F9 | | 9 |
| FA | | |
| FB | | |
| FC | | |
| FD | | |
| FE | | |
| FF | | |

# 出 版 说 明

美国奥斯本(Osborne)公司为那些不了解计算机的读者出版了一套«微型计算机丛书»。为了在我国读者中普及微型计算机应用的基本知识，我社选译了其中五种，即：

一、微型计算机入门：初学者的书 (An Introduction to Microcomputers Volume 0：The Beginner's Book)，(已出版)。

二、微型计算机入门：基本概念(An Introduction to Microcomputers Volume I：Basic Concepts)，(已出版)。

三、6800 程序设计在逻辑设计中的应用(6800 Programming for Logic Design)，(已出版)。

四、8080 程序设计在逻辑设计中的应用(8080 Programming for Logic Design)。

本书是第四种，以后还将出版第五种，即«8080A/8085 汇编语言程序设计»。

这套丛书的主要目的是向读者介绍微型计算机原理和应用知识，从第三种书开始都是介绍微型计算机应用知识的。这本书是为读者学习用 Intel 8080 微型计算机来模拟典型逻辑器件提供的一本著作。以微型计算机模拟来代替常规数字逻辑电路，这是微型计算机应用的一个新的发展。这一新发展对设计者也提出了新要求，它不仅要求设计者对硬件进行逻辑设计，而且

也能对总线上的微型计算机器件进行软件的程序编制。本书的内容能适应这种要求，它通俗地叙述了用于逻辑设计的程序设计，并介绍了如何实际动手去编制程序，所以讲得具体实用，且直接用Intel 8080这种微型计算机作为具体对象来加以讨论。因此，对学习微型计算机应用和编写程序的初学者来说，本书是一本较有价值的读物。

人民邮电出版社

# 目　录

# 第一章 引 言

**组合逻辑**

本书解释在微型计算机系统中如何利用汇编语言程序来代替组合逻辑，也就是如何代替象 7400 系列标准数字逻辑集成电路那样现成的、非可编程序的逻辑器件所组成的电路。

假如你是一位逻辑设计者，本书将教给你如何使用新的方法，用微型计算机系统中产生的汇编语言程序来从事你的旧业。

假如你是一位程序设计者，本书将告诉你程序设计已经找到一种新的用途：用于逻辑设计。

这是一本教你如何实际动手的书。这样，它的内容也就必须十分具体，所以直接以 8080 A 这种类型的微型计算机作为具体对象来加以讨论。生产 8080 A 型微型计算机的厂商有好几家，它们的产品有：

AMD 9080 A
INTEL 8080 A
NEC 8080 A
TMS 8080 A
NS 8080 A

生产这些微型计算机的厂商是：

INTEL 有限公司
高级微型器件公司
德克萨斯仪器公司
NEC 微型计算机公司
国家半导体公司

## 1-1 这本书假定你已知道了···

本书是“微型计算机入门”的续篇。“入门”初版时为一册，再版时分为两册。

“微型计算机入门”一书介绍微处理机和微型计算机的概念；它并不涉及实现这些概念的实际手段。而本书则介绍了实现这些概念的实际手段。

由于本书是“入门”的续篇，所以就简单地假定你已阅读过或已了解“微型计算机入门”一书所包括的内容。不过，在你着手设计一个实际的方案之前，你还需要阅读具体介绍你所选用的各种器件的说明资料。

尤其应当注意的是，这本书并不介绍 8080 A/9080 A 中央处理机，或任何其它微型计算机器件的硬件和定时关系。由于本书的整个目的是介绍程序设计，所以第六章介绍 8080 A/9080A 指令系统。

## 1-2 了解汇编语言

汇编语言指令是微型计算机系统的传递函数，把它们组合起来，就构成了一套“指令系统”，用来描述微型计算机所能完成的各种操作。

你把应该在微型计算机里依次发生的事件定义为一个指令序列。那末，把这些指令序列组合起来就构成了一个汇编语言程序。

实际上，理解各条指令在微型计算机里做些什么是十分简单的；它是应用微型计算机的最简单的一个方面。然而，它会使初次编制程序的用户不知所措。如果其中也包括你自己，那末我们要奉劝你，如果忘记了助记符和指令系统也不要紧，当

你在本书中遇到它们时，一条一条地去查阅指令就行。如果你不了解这条指令是做什么的话，就请参阅一下第六章。

只有当你惧怕程序设计时，它的幽灵才会纠缠着你。

## 1-3 这本书是怎样排印的

请注意，本书正文是用粗体字和细体字印刷的。这是为了帮助你跳过本书中你所熟悉的那部分内容。细体字所阐述的，只是前面粗体字所介绍知识的延伸。因而，最初只需阅读粗体字，直到你需要知道得更多时，你再开始阅读细体字的内容。

本书中译本分别用老宋体(对应于原书的粗体字)和仿宋体(对应于原书的细体字)印刷，这样在阅读时，亦可根据需要加以区别。

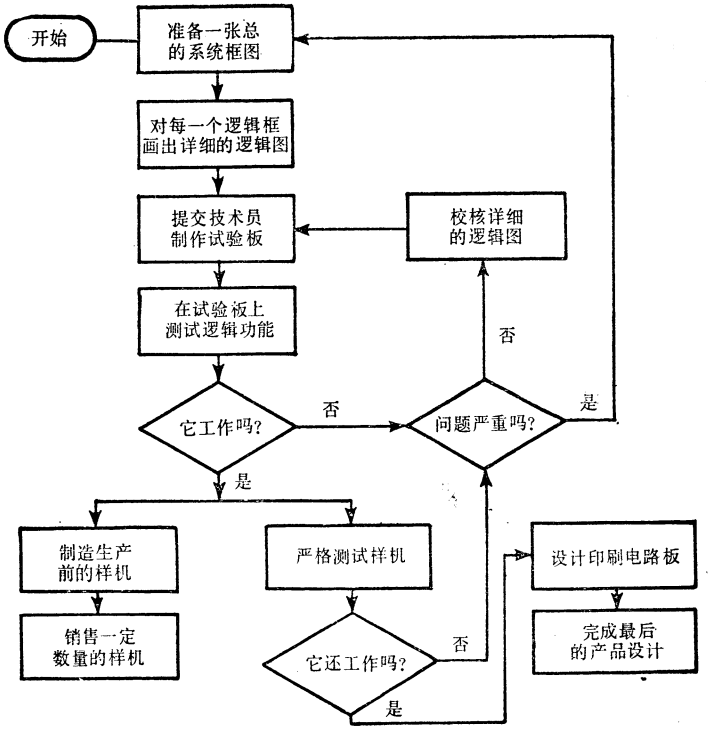
# 第二章　汇编语言和数字逻辑

## 2-1　设计周期

**数字逻辑设计周期**

任何由分立的数字逻辑元件组成的产品都要经过一个妥善规定好的设计周期。

让我们假定已经根据市场销售的观点制定



出产品规格。

已经取得一份列举必要的产品性能和特性的产品说明书。你的任务是提供一套能适合生产的设计。

设计周期如第4页的流程图所示。

在任何数字逻辑设计周期中都有一个代价昂贵而又缓慢的重复循环；如上图所示。它包括下列步骤：

重新绘制逻辑图

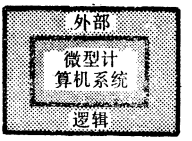
重新制做一块新的试验板

检测试验板是否有逻辑错误、技术性错误或组件缺陷。

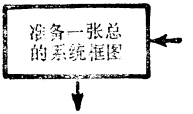
这些重复循环使得组合逻辑设计缓慢而昂贵。这不仅在设计的初始阶段如此，而且当你以后决定修改或提高产品质量时更会如此。

**微型计算机逻辑设计周期**

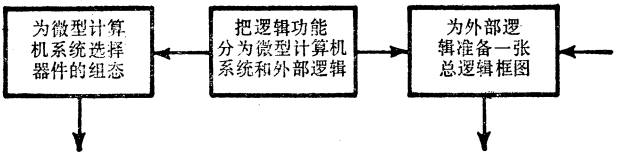
当你开始使用微型计算机时会发生什么情况呢？首先，你的一部分逻辑功能消失在一个“黑盒子”当中。它就是微型计算机系统：




你所要进行的第一步：




现在必须把它再做如下的划分：




如果你不了解微型计算机能做什么的话，那么要把你的应用课题划成微型计算机系统和外部数字逻辑两部分，看来也许是一个难题。

实际上，**一旦在你的产品里有一台微型计算机，假定能使这"黑盒子"担任尽量多的工作，那末在经济上是极为有利的；这样你就必须检查是否还需要每个外部逻辑门。**

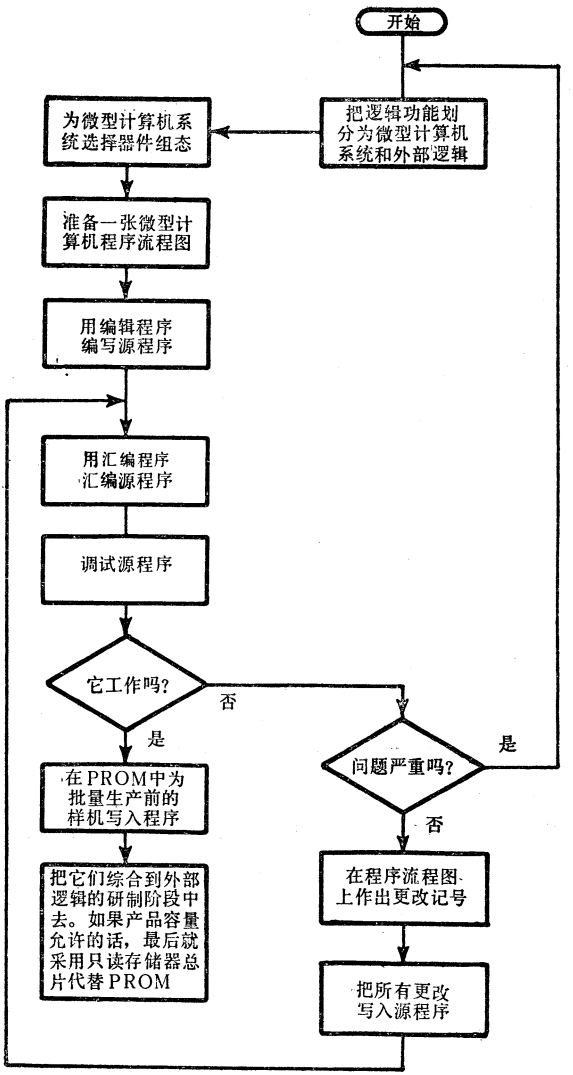
我们知道，存储器是由一定数目的单元组成的。为了在微型计算机系统内扩大所实现的逻辑功能，你可以简单地编写一些附加的指令序列，把这些指令序列驻留在否则就要被浪费的存储器中。或者，需要增加程序存储器，但增加的费用很小。

而且，与数字逻辑研制的费用相比，微型计算机逻辑的研制时间短而花费小。**一种典型的微型计算机系统的研制周期可以用第7页的图说明。**

在上图所示的微型计算机研制周期中仍然有重复循环，但同研制数字逻辑相比，微型计算机研制周期中的重复循环所需要的时间较短，费用也较少。

**每种微型计算机都是用一种开发系统研制出来的。**这些开发系统的特性和操作，因不同的生产厂家而有所不同；然而，**它们都具有以下功能：**

1）你可以模拟微型计算机系统，并使你确信没有必要制做一块试验板。


开始
把逻辑功能划分为微型计算机系统和外部逻辑
为微型计算机系统选择器件组态
准备一张微型计算机程序流程图
用编辑程序编写源程序
用汇编程序汇编源程序
调试源程序
它工作吗?
否
是
问题严重吗?
是
否
在PROM中为批量生产前的样机写入程序
把它们综合到外部逻辑的研制阶段中去。如果产品容量允许的话，最后就采用只读存储器总片代替PROM
在程序流程图上作出更改记号
把所有更改写入源程序

**源 程 序**

2）你可以执行一个驻留的编辑程序去建立你的源程序。应当记住，一个汇编语言指令序列就叫做一段“源程序”。

**目 标 程 序**

3）你可以在开发系统上正确地汇编源程序，从而建立目标程序。我们知道，在执行源程序之前，必须将它转换成二进制数字序列，称为目标程序。

4）你可以在一定的条件下执行目标程序，以便确信它是能够工作的。

利用典型的微型计算机开发系统可以在一天内完成几个主要的研制周期，而在整个数字逻辑的实现过程中也许每一个研制周期都要延续一到两个星期。在一个研制周期中你可以对程序做许多修正；一个简单的修正甚至不到一分钟时间就能完成，而这种修正却相当于从一块数字逻辑试验板上增添或拆去一个门或一种中规模集成电路的功能。

## 2-2 模拟数字逻辑

如上所述，逻辑系统最后必定划分成微型计算机系统和外部逻辑两部分。

我们来指出对这种逻辑系统划分的两个方面：

1）基于汇编语言的能力去模拟数字逻辑。为了估计一台微型计算机系统能做什么和不能做什么，我们必须拟订出一些简单的准则。

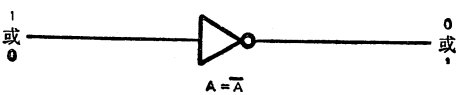
2）我们必须编制出一段程序来实现指定微型计算机系统完成的逻辑功能。遗憾的是，编写一段微型计算机程序有许多方法。一旦你掌握了用指令驾驭微型计算机系统的概念，下一步就应学习如何编写高效能的程序。

我们的探讨从模拟简单的数字逻辑开始。这是一个必要的

起点，因为在数字逻辑和微型计算机程序逻辑之间有某些基本概念是不同的。

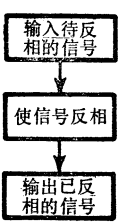
## 2-3 用微型计算机模拟信号反相器

假设你需要使单个信号反相：




**流程图**

为了一开始就养成一个良好的习惯，我们要用如下的逻辑流程图来说明信号反相器：

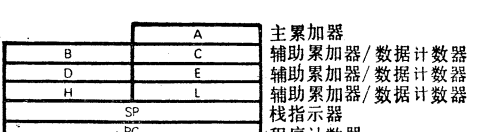



虽然你绝不会用一台微型计算机来简单地代替一个信号反相器，但仍然值得观察一下它是如何实现的。

### 2-3-1 微型计算机事件序列

**中央处理机寄存器**

大家还记得 8080 微型计算机有以下几个中央处理机寄存器：




下列简单指令：

CMA :COMPLEMENT ACCUMULATOR

CMA :累加器取反

**位数据**

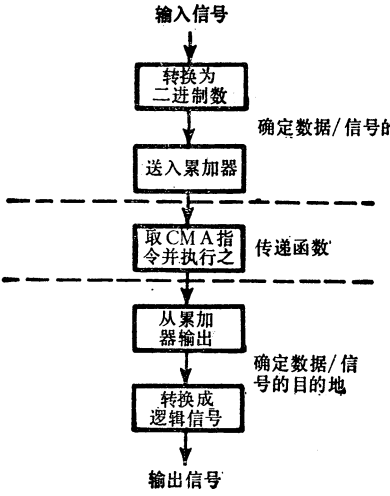
当把 CMA 指令转换为结果代码并执行之后，累加器的所有八位均被取反。但这并不是模拟反相器。首先必须选择累加器的一位二进制数字来代表被取反的信号。但是究竟选哪一位呢？

**数据的源和目的地程序定时**

如果已确定了所代表的那一位二进制数字，那末这一位又是如何首先到达累加器的呢？当一旦取反后，取反了的那一位又是如何再变成信号的呢？

如果为了实现实际的反相操作，必须执行 CMA 指令结果代码。这一结果代码又如何以及何时到达中央处理机呢？显然，这条指令执行的时刻必须在需要取反的二进制数进入累加器之后。

**把上述流程图扩充一下，就能说明采用一台微型计算机完**



成反相器功能所需要的步骤了。

如上所述，我们把这个问题划分为以下三个阶段：

1）数据/信号源的确定。我们确定出被操作的数据，把该数据传送到微型计算机的中央处理机能存取的存储单元内。

2）传递函数的执行。对于源数据必须完成的实际操作叫做传递函数。

3）数据信号目的地的确定。已经由传递函数处理的数据或信号现在必须传送到某一目的地。

现在我们编写一个指令序列来实现对上述反相器模拟的三个阶段。

### 2-3-2 传递函数的实现

**位数据**

CMA指令把累加器的每一位都取反。因此，CMA指令并不说明累加器的哪一位代表被反相的信号。这一说明中应包含数据是由哪里输入的，以及从微型计算机系统输出到哪里去。

### 2-3-3 确定数据的源和目的地

累加器的数据如何输入以及如何从微型计算机系统输出？为了回答这个问题，我们要涉及到微型计算机的一种基本能力（和复杂性），即它们的灵活性。

**作为源或目的地的外部逻辑**

输入信号和被反相的输出信号正如它们的名称所指出的，它们都是信号。但是，对于微型计算机系统来说，它们都属于“外部逻辑”。在外部逻辑和微型计算机系统之间传送信息总称为输入/输出（或I/O）。

**输入/输出**

在任何程序控制下的I/O操作中，必须记住：微型计算机是“主”，而外部逻辑则是“从”。

这就意味着微型计算机必须指出I/O(输入或输出)操作的方向，并能识别出被连接的外部逻辑。

**存储器地址空间中的I/O**

外部逻辑可以把某一存储器地址译成一个允许选通脉冲，以便把I/O当做存储器的读或写操作来处理。

假定现在汇编语言源程序中用标号INVD来标识被反相的信号，则用于对信号反相器进行模拟的指令序列为：

```
LDA     INVD    ;LOAD ACCUMULATOR FROM INVD
CMA             ;COMPLEMENT THE ACCUMULATOR
STA     INVD    ;STORE ACCUMULATOR CONTENTS AT INVD
```

```
LDA   INVD   :从INVD取数送入累加器
CMA          :累加器取反
STA   INVD   :将累加器的内容存入INVD
```

用微型计算机各器件表示的微型计算机结构，如第13页的图所示。

当执行LDA指令时，“地址译码逻辑”使“选择逻辑”把“数据输入”信号传送到数据总线。

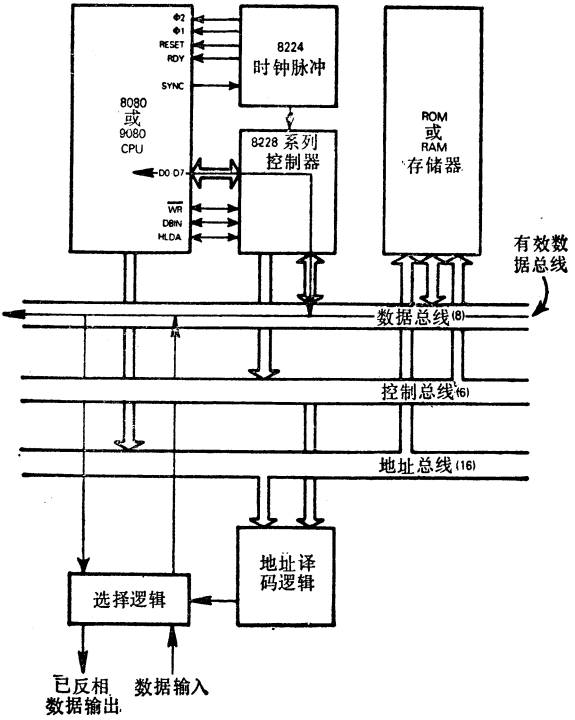
有八条数据总线；与“数据输入”信号相连的各条线的号码和在累加器内的有效位的号码是一一对应的。当LDA指令执行完毕后，数据总线的内容已送入累加器。

然后执行CMA指令。这条指令使累加器的每位取反。

当执行完STA指令时，累加器内容输出到数据总线。“地址译码逻辑”于是使“选择逻辑”将一条数据总线线上的内容加以输出，它就成为被反相的“数据输出”信号。

因为“选择逻辑”具有连接在同一条数据总线线上的“数据输入”信号和“数据输出”信号，而“数据输出”是“数据输入”的取反；所以信号反相器就被模拟出来了。

在微型计算机里必须有只读存储器(ROM)或随机存取存



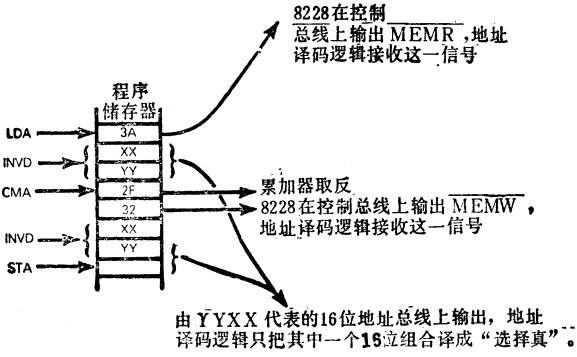

储器(RAM)，因为这三条指令的结果代码必须存入存储器和从存储器中取出来。

**结果代码的解释**

现在来详细地考察结果代码。三条源程序指令变成结果代码的情况如下：

存放结果代码字节的程序存储器地址是无关紧要的。即使没有存储器字节的话，只读存储器或随机存取存储器也能有以 YYXX 代表的地址，因为外部逻辑就是按此地址被选择的。

我们看到，16 位地址的两个字节 YYXX 当存入存储器时




被颠倒了，这种颠倒并不足为奇。它只是 8080 器件设计成的一种形式而已。

**经过I/O口的输入/输出**

现在假设与外部逻辑的连系是通过一个输入/输出外部接口器件进行的。

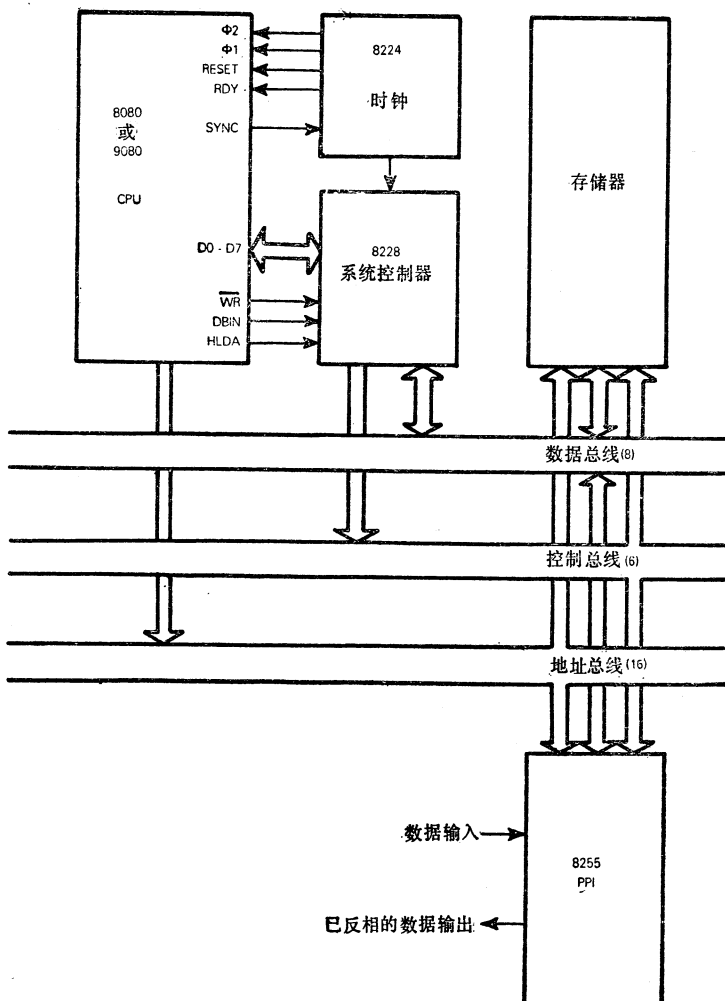
在汇编语言源程序指令内，标号 INVD 现在将代表一个输入/输出口。下面是用于模拟信号反相器的指令序列：

```
IN      INVD    ;INPUT TO ACCUMULATOR FROM PORT INVD
CMA             ;COMPLEMENT THE ACCUMULATOR
OUT     INVD    ;OUTPUT ACCUMULATOR TO PORT INVD
```

```
IN      INVD    ;从口 INVD 输入到累加器
CMA             ;累加器取反
OUT     INVD    ;从累加器输出到口 INVD
```

微型计算机的硬件结构如下页所示。

我们增设 8255 可编程序外部接口（PPI）的全部目的就在于为“数据输入”信号和反相的“数据输出”信号提供需要的“地址译码”和“选择逻辑”。现在，特定的位是否有效将取决于所连“数据输入”信号以及被反相的“数据输出”信号的 8255PPI的两



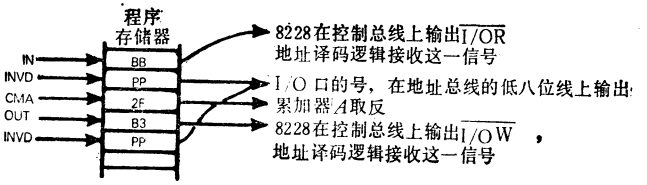

个引线端。这两个引线端取决于 8255 PPI 所采用的工作方式。

当采用 8255 PPI 时，有很多可选取的方式。但对你来说，

眼下并不急于考虑，因为这样反而会使你早先对汇编语言程序的理解发生混乱。因此，我们暂且不管 8255 PPI 工作方式的控制指令，而是简单地假定已经选好了适当的控制方式。

**结果代码的解释**

在此情况下，对于这三条指令的结果代码解释如下：

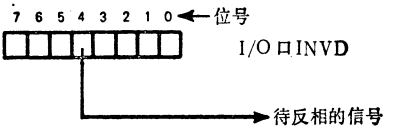



这里再次指出，存储以上结果代码的程序存储器字节的地址并不重要。

**输入/输出口引线端选择**

可以看到，尽管对应于被取反的信号只是一位，但我们仍然将输入/输出口 INVD 的每一位都取反。

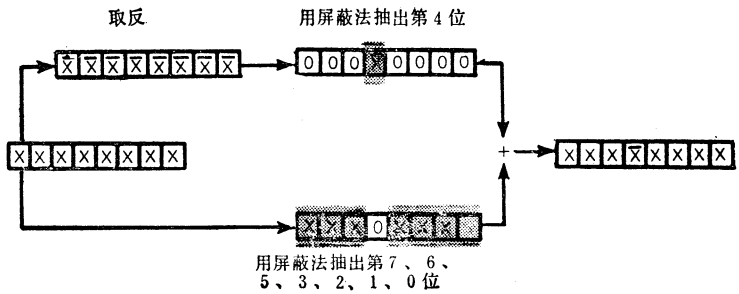
假定只有引线端 4 必须取反：




**位屏蔽**

为了使 I/O口的单个引线端上的信号反相，而其它所有各端保持不变，可以采用通常所说的“屏蔽”技术。在此例中，屏蔽可以说明如下(见第17页图)。

如上所示，X 代表一个二进制数字，$\overline{X}$代表它的反码。




以下的指令序列将使端 4 上的信号反相，而保留其它所有各端的信号不变：

```
IN      INVD    ;INPUT TO ACCUMULATOR FROM I/O PORT INVD
CMA             ;COMPLEMENT ACCUMULATOR
ANI     10H     ;ISOLATE BIT 4
MOV     B,A     ;SAVE IN REGISTER B
```

```
IN      INVD    ;INPUT TO ACCUMULATOR FROM I/O PORT INVD
ANI     EFH     ;CLEAR BIT 4
ORA     B       ;OR A AND B
OUT     INVD    ;OUTPUT ACCUMULATOR TO I/O PORT INVD
```

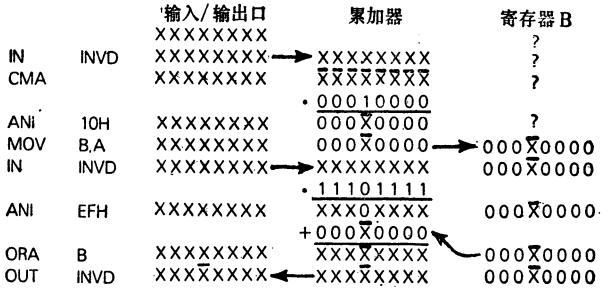
```
IN      INVD    ;从 I/O 口 INVD 输入到累加器(A)
CMA             ;累加器取反
ANI     10 H    ;抽出第 4 位
MOV     B,A     ;保留在寄存器 B 内
IN      INVD    ;从 I/O 口 INVD 输入到累加器(A)
ANI     EFH     ;第 4 位清零
ORA     B       ;A 和 B 相"或"
OUT     INVD    ;把累加器内容送到 I/O 口 INVD
```

**在操作数字段内的H**

作为操作数字段内最后的字符H表示在它前面的是十六进制数。例如 EFH 表示二进制数：

$$\underbrace{1110}_{E}\underbrace{1111}_{F}$$

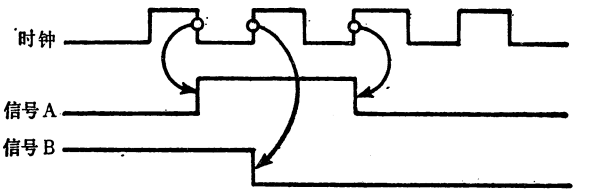
下图说明当执行以上指令序列时各寄存器内容所发生的变化(还是用X代表一个二进制数字)：




## 2-3-4 事件的定时

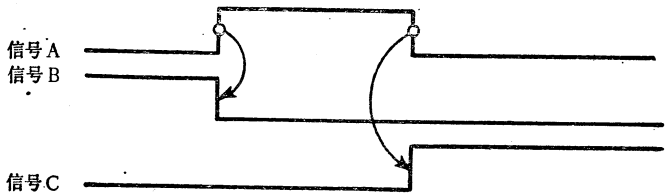
**同步逻辑**

在实现任何数字逻辑的过程中，事件可以根据时钟脉冲信号以同步方式定时：

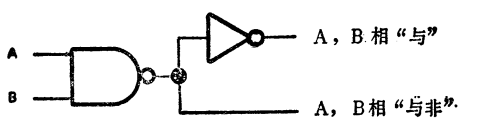



**异步逻辑**

或者根据一个器件改变状态时所输出的信号来触发另一个器件使之改变状态以实现异步定时：




然而，简单的门电路是连续动作的器件。考察以下的简单逻辑序列：




**门电路的建立时间**

信号反相器使其输入信号连续反相时，在输入信号和输出信号状态改变期间一个门电路的建立时间约为10毫微秒，这是唯一存在的延迟。

在微型计算机系统里，无论如何这三条指令必须在输出信号能反映输入信号的状态改变之前就执行完毕。

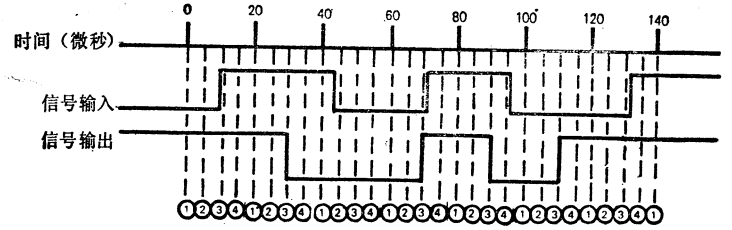
在微型计算机系统只是对一个反相器进行仿真而不做任何其它事情的情况下，反相器指令序列可以连续重复执行如下：

```
LOOP    LDA     INVD    ;LOAD ACCUMULATOR FROM INVD
        CMA             ;COMPLEMENT THE ACCUMULATOR
        STA     INVD    ;STORE ACCUMULATOR CONTENTS AT INVD
        JMP     LOOP    ;RE-EXECUTE THE SIGNAL INVERTER SEQUENCE
```

```
LOOP    LDA     INVD    ：从INVD取数送入累加器
        CMA             ：累加器取反
        STA     INVD    ：把累加器内容存入INVD
        JMP     LOOP    ：重复执行反相器指令序列
```

根据8080微型计算机的特性和时钟脉冲的频率，“信号反相器指令循环”执行一次约为20微秒；只要输入信号状态变化

的周期不小于20微秒，则微型计算机总是能实现信号反相器的功能。但是，在改变状态的输入信号和接着产生的输出信号之间，可能有最长达20微秒的时间延迟。这可用下图说明：




①＝执行LDA指令
②＝执行CMA指令
③＝执行STA指令
④＝执行JMP 指令

在上例中，由四条指令来均分20微秒。因此，执行每条指令要用5微秒。实际上，并非如此。第六章给出了每条指令的执行时间；例如，你将看到执行CMA指令比执行其它任何一条指令的时间要短得多。我们暂且忽略这一细节，以便集中考虑目前的概念，那就是**我们必须仔细地注意微型计算机系统内事件的序列。**

不论“信号输入”是在何时改变以及是怎样改变状态的，“信号输入”状态总是在时刻①（执行LDA指令时）以二进制数字的形式送入微型计算机系统内。

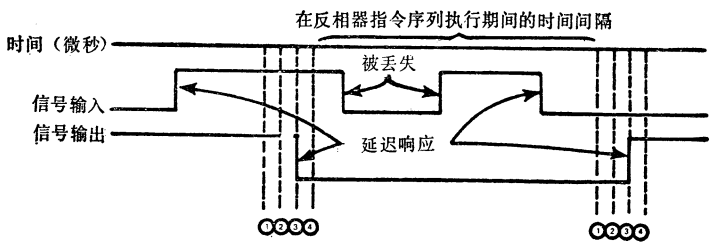
实际上，二进制数字的变反发生在时刻②。

当执行STA指令时，在时刻③上把已取反的二进制数字转换成“信号输出”。

因而，“信号输出”的时间关系可以与“信号输入”的大不相同。

当信号反相器指令序列只是一个微型计算机的比较长的程

序的一小部分时，会出现更严重的问题。在此情况下，在重复执行反相器指令序列的过程中，可能会浪费掉很多毫秒的时间。如果你不执行这样的重复循环而去碰运气的话，信号变反有可能完全被遗漏。最好的情况是在输入信号改变状态和接着产生的输出信号之间可能有适当的时间延迟。这种情况如下图所示：




①②③和④仍然各自表示执行LDA，CMA，STA和JMP指令。上面已经强调了微型计算机系统中定时的重要性，以及定时不准确所引起的后果，我们暂且先把这个问题搁置一下。这是因为当你模拟的是整个逻辑序列而不是个别的器件时，定时问题在很大程度上消失了。因此，定时问题应当在审查整个逻辑模拟的前后关系中解决，有关这一问题迄今我们还没有讨论。

## 2-4 缓冲器、放大器和信号负载

我们已经考察了定时问题，现在将转向另一些基本的数字逻辑概念。

**缓 冲 器**

信号缓冲器增强信号电流电平：

缓冲器

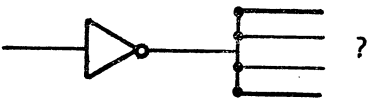
**放 大 器**

放大器、驱动器增强信号电压电平：

放大器，驱动器

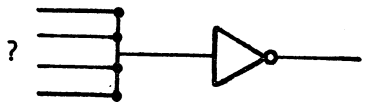
**扇 出**

每个器件有一个完全确定的扇出。扇出确定了能被连接到一个器件输出端的并联负载的数目：




**扇 入**

逻辑器件也有确定的扇入，扇入表示能被连接到一个器件输入端的并联负载的数目：




一旦你的逻辑溶化在微型计算机程序中，这些概念将发生什么变化呢？回答是简单的：这些概念随着数字逻辑一齐消失。

**扇 入**
**扇 出**
**TTL 负 载**
**信号缓冲**

对于实际微型计算机器件的引线端来说，扇入和扇出仍然是应保留的概念；因而在一台微型计算机的各种器件的引线端之间传送的信号可能需要加以放大和缓冲。例如，倘若一个8080中央处理机器件的扇出仅仅是一个或两个晶体管——晶体管逻辑(TTL)的负载；那就表

示，如果多于一个或两个同样的器件要与一个输出信号接通，此时该输出信号将没有足够的功率把有效信号传送给所有与之相连的器件。所以，除了最简单的微型计算机结构以外，总线上的各条线都必须是经过缓冲的。

**漏电流**

当决定总线是否需要缓冲时，不要忽视漏电流。例如，你有十六个只读存储器器件连接在系统总线上，而在任何时刻只能选中（因而，也就是只连接）一个器件，不要认为总的信号负载就只是那一个被选中的只读存储器器件。十五个未被选中的只读存储器器件将各自分取一些漏电流；仅仅是这些漏电流就可能需要对系统总线加以缓冲了。

然而，在微型计算机程序中，当逻辑线路完全用微型计算机的一段指令序列来表示时，除了涉及到二进制数字外，决不会涉及电压电平或电流电平。因为，一个二进制数字的状态只可能是许多逻辑计算的结果，所以扇入是没有限制的。又因为，你需要多少个二进制数字的状态，你就可以读出多少次，所以扇出也是没有限制的。由于一个二进制数字没有与电压或电流等效的参数，因而在微型计算机程序中缓冲器和放大器是没有意义的。一个二进制数字提供纯粹的、一定的解答。

从另一个角度来看一下作为微型计算机所模拟的信号反相器。

我们从概念上迈出一大步，假定信号反相器被隐藏在一段逻辑序列之中，以至于不会在微型计算机的任何器件的引线端上产生输入或输出信号。换句话说，信号反相器成为一个大的传递函数中的一小部分。

信号反相器的输入是由前面的逻辑线路所产生的二进制数字。

信号反相器的输出则成为输入到后面逻辑线路的二进制数字。

**一个存储器字节的取反**

微型计算机的外部逻辑不把到达微型计算机器件引线端的信号作为反相器的输入信号，已反相的信号也不经微型计算机的某一器件的引线端传送给外部逻辑线路。当然，外部逻辑和微型计算机之间的接口一定位于信号反相器之前或之后的某一点上。现在，信号反相器可以用同样的三条指令来表示：

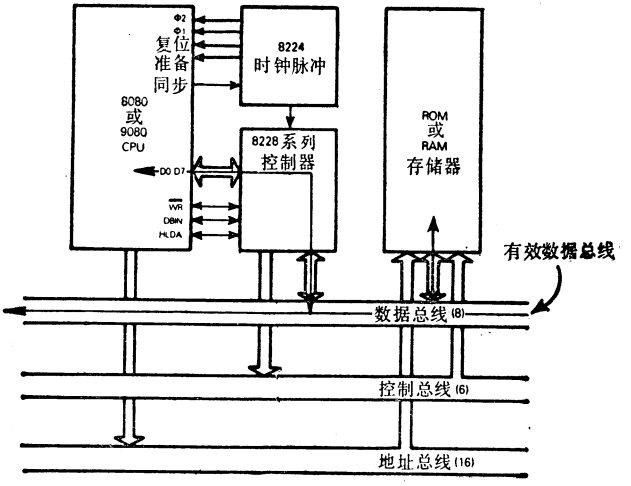
```
LDA     INVD    ;LOAD ACCUMULATOR FROM INVD
CMA             ;COMPLEMENT
STA     INVD    ;STORE ACCUMULATOR CONTENTS TO INVD
```

LDA　INVD：从 INVD 取数送入累加器

CMA　　：取反

STA　INVD：把累加器内容存入 INVD

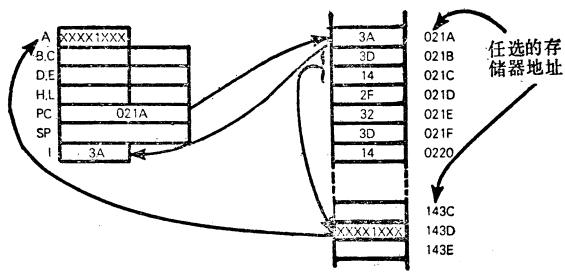
而信号的源和目的地则成为数据存储器的某些单元了；这



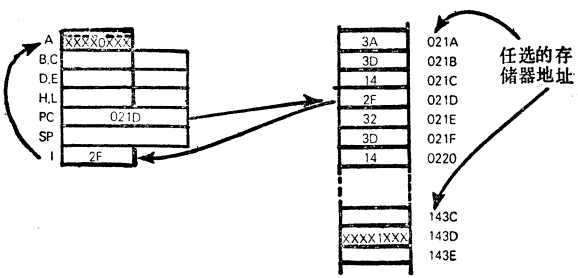
可用上图说明。

用存储器和CPU寄存器内容来表示，则信号反相器的工作进程如下所示：

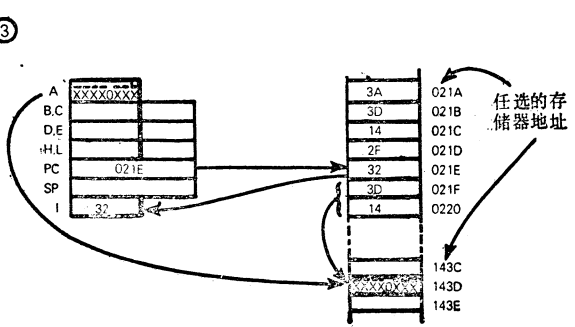
①




②




③



以上已经说明，字母A、B、C、D、E、H和L表示8080的七个CPU寄存器。PC表示程序计数器。SP表示栈指示器。I表示指令寄存器。

数据存储器字节143 $D_{16}$和寄存器A的内容用二进制格式表示。X表示任一个二进制数字。注意，我们任意选定第3位作为有效位。

在第① 步，执行LDA指令。这条指令使数据存储器字节143 $D_{16}$的内容送入累加器A。

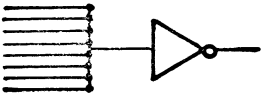
在第② 步期间，执行CMA指令。这条指令使累加器A的内容取反。

在第③ 步期间，累加器A的内容送回地址为143 $D_{16}$的存储器字节。

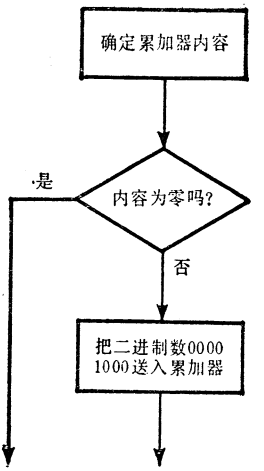
信号反相被数据存储器字节143 $D_{16}$的第3位（连同其它各位）的内容取反所模拟。

**在微型计算机程序中的扇入**

反相器的输入是从哪里来的呢？来自数据存储器中的一位。为了说明问题，我们假定反相器的输入是对于八个信号的“或”。我们不能用下列“线—或”八个信号的方法产生一个反相器输入：



但是，如果预先假定这八位信号是由累加器的八位二进制数字内容所表示，那末通过如下的逻辑序列产生反相器输入就没有困难了：




扇入逻辑是由以下指令序列来实现的：

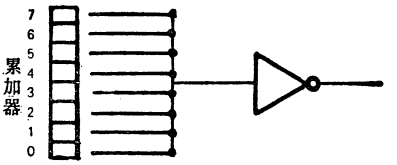
```
;ASSUME THE EIGHT SIGNALS ARE IN THE ACCUMULATOR.
;EACH REPRESENTED BY ONE ACCUMULATOR BIT
        ANA     A       ;AND ACCUMULATOR WITH ITSELF TO SET STATUS FLAGS
        JZ      NEXT    ;ACCUMULATOR HOLDS 0. SIGNAL IN MUST BE 0
        MVI     A,8     ;ACCUMULATOR HOLDS NONZERO. SIGNAL IN MUST BE 1
NEXT    STA     INVD    ;SAVE INVERTER INPUT
```

```
        ；假定这八个信号在此累加器内
        ；每个信号由累加器的一位所表示
        ANA   A       ：累加器和它自己相“与”，置状态位标志
        JZ    NEXT    ：累加器为零，则信号输入为“0”
        MVI   A, 8    ：累加器为非零，则信号输入为“1”
NEXT    STA   INVD    ：保存反相器输入
```

以上的指令序列是用微型计算机程序直接实现八个信号“线一或”的方法。让我们考察一下指令的逻辑功能。

我们假定这八个输入信号最初由累加器的八位二进制数的状态所表示：

而且，为了保持和以前的说明相一致，还假定数据字节的




第 3 位表示最终有效的反相器信号位。

因为信号反相器输入是八个信号的“线—或”，所以如果累加器的任一位是非零值，则程序逻辑必须将累加器的第 3 位置为“1”。如果累加器所有各位都是零，则必须将累加器的第 3 位置“0”。累加器的内容于是被存入由标号为 INVD 所表示的数据存储器字节中。前已指出 INVD 是表示存储器字节 143 $D_{16}$的一个标号。

下面来说明四条指令的序列是如何工作的。

**由累加器和它本身相“与”来决定状态位**

我们并不知道累加器内最初存放着什么，所以必须相应地置 CPU 状态标志来标示累加器的内容。为此，我们把累加器内容和它本身相“与”。这不会改变累加器的内容，但却建立了相应的状态标志。我们仅仅对零状态标志触发器感兴趣，假如累加器和它本身相“与”的结果为零，则零状态标志触发器将被置“1”；否则就被置“0”。

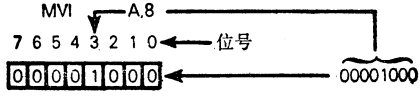
00000000
00000000
00000000

但是，如果累加器存放的是零，那么累加器和它本身相“与”将只可能是零（见29页上图）。

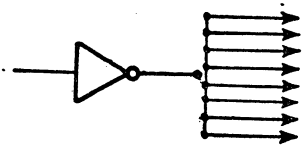
因而在 ANA 指令执行后，假如零状态标志触发器是“1”，那末累加器第 3 位必定已经是“0”，这就是我们所求出的结果。不需再行操作，就转移到 STA 指令。

假如零状态位是“0”，那末累加器中必定有一位或更多的

位是非零位。那末，MVI 指令把“1”送到累加器的第 3 位。




最后执行 STA 指令，把反相器输入信号送入相应的数据存储器字节。



现在假设反相器输出要分配给后继的许多器件。下列逻辑线路所表示的扇出是不能实现的：

**在微型计算机程序中的扇出**

在微型计算机程序中，扇出的整个概念已经消失。只需简单地重复执行一条 LDA 指令就能任意多次地取得反相器的输出。

```
LDA     INVD    ;LOAD INVERTER OUTPUT INTO AC-
                 CUMULATOR
.
.
.
LDA     INVD    ;LOAD INVERTER OUTPUT INTO AC-
                 CUMULATOR
.
.
.
LDA     INVD    ;LOAD INVERTER OUTPUT INTO AC-
                 CUMULATOR
.
.
.
LDA     INVD    ;LOAD INVERTER OUTPUT INTO AC-
                 CUMULATOR
.
.
.
LDA     INVD    ;LOAD INVERTER OUTPUT INTO AC-
                 CUMULATOR
```

```
LDA   INVD   :把反相器输出送入累加器
  ⋮
LDA   INVD   :把反相器输出送入累加器
  ⋮
LDA   INVD   :把反相器输出送入累加器
  ⋮
LDA   INVD   :把反相器输出送入累加器
  ⋮
LDA   INVD   :把反相器输出送入累加器
```

放大器和缓冲器是怎样呢？显然，就储存于存储器的二进制数据而言，它们是没有意义的。

如果为了存储器和处理机芯片的电特性的需要而设置放大器和缓冲器，那末需指出，这与用微型计算机程序实现逻辑功能完全是两回事。

## 2-5 用微型计算机模拟 7404/05/06/07 六反相器

这四种六反相器仅仅在电特性上有区别：

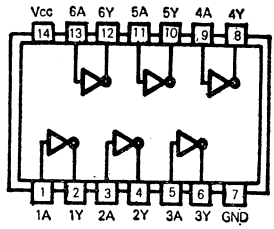
7404是一种简单的六反相器

7405是一种开集输出的六反相器

7406是一种开集、高压输出的六反相器缓冲器/驱动器。

这三种器件仅仅在它们的电特性上有区别，而在微型计算机汇编语言模拟方面则是相同的。所以我们仅考察 7404 即可。它由六个独立的信号反相器所组成，如第31页上图所示。

表示一个六反相器的指令序列和三条指令组成的一个信号反相器指令序列是完全一样的，因为 8080 型微型计算机是八位并行的器件。不管你是否愿意，这个信号反相器指令序列总是把八位独立的二进制数一起取反。因此，六反相器可以用微




$Y=\overline{A}$

型计算机中如下的指令序列来表示：

```
LDA     INVD    ;LOAD ACCUMULATOR FROM INVD
CMA             ;COMPLEMENT
STA     INVD    ;STORE ACCUMULATOR CONTENTS AT INVD
```

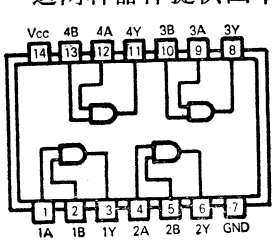
LDA INVD ：从 INVD 取数送入累加器
CMA ：取反
STA INVD ：将累加器内容存入 INVD

按照六反相器的含意，我们可以任意地标识其中六个有效位如下：

注意，以上有效位的选择是完全任意的。绝对没有任何实际的或理论上的原因一定要选用某一位，而不选用其它的位。


## 2-6 用微型计算机模拟 7408/09 二输入、四正“与”门

这两种器件提供四个独立的、二输入、一输出的“与”门，如下图所示：

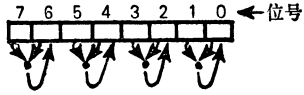



7409具有开集输出，这是它与7408不同的地方。但是，这种差别在微型计算机程序的模拟中是没有意义的，因此可以认为这两种器件是相同的。

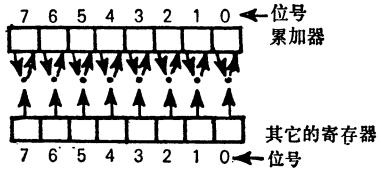
## 2-6-1 二输入功能

从微型计算机程序员的观点看，在7408“与”和7404反相器之间最值得注意的差别不在于它们的逻辑功能，而在于7408是个二输入器件。从概念上说，我们可以设想用以下两种方法中的一种来模拟7408“与”门。

(1) 八个输入信号送入CPU累加器寄存器。每个偶数位和它左边的一位相“与”，结果存入这两位中的偶数位：




(2) 两组四输入送到CPU累加器和另一个寄存器。两组输入相“与”的结果送回累加器：




考察8080微型计算机指令系统，你就会发现模拟7408的第二种方法是比较自然的。以下是它所需的指令序列：

```
LDA     SRCA    ;LOAD FIRST SET OF INPUTS, FROM SRCA
MOV     B,A     ;SAVE IN THE B REGISTER
LDA     SRCB    ;LOAD SECOND SET OF INPUTS, FROM SRCB
ANA     B       ;AND B WITH A
STA     DST     ;SAVE RESULT IN DST
```

```
LDA     SRCA    :将第一组输入从 SRCA 送入累加器
MOV     B, A    :保存在寄存器 B 内
LDA     SRCB    :将第二组输入从 SRCB 送入累加器
ANA     B       :B 和 A 相“与”
STA     DST     :结果保存在 DST 内
```

**源程序标号的分配**

假如你仍然弄不清标号SRCA、SRCB和DST的用法，那么让我们花一点时间来弄清楚它们。毕竟你具有存储器的容量是一定的，它们可以小自256个字节到多达65,536个字节。标号SRCA、SRCB和DST各自表示一个存储器字节。当你在编写源程序时，由每个标号所表示的实在的存储器字节是无关紧要的。在最后汇编源程序时，汇编程序的列表将打印出一张存储分配图。这张存储分配图将确定出与你所用的每个标号有关的实际存储器字节。查看这张存储分配图，你就能够判断出对于所有标号分配的地址是否都是有效的。假如任何一个标号的分配地址是无效的，你就应采取适当的行动。所谓适当的行动可能包含着在你的微型计算机结构中增加更多的存储器，或者可能必须改写源程序，从而使它更有效地利用你已有的存储器。

在目前讨论的水平，标号和存储器分配的问题是无关紧要的。简单地设想用每个标号都表示一个特定的存储器字节，那就不必担心最终找到的是哪一个存储器字节；这样，你的问题就不存在了。

上述模拟7408的指令序列绝不是模拟7408的唯一方法。

首先考虑几个次要的变动。CPU寄存器C、D、E、H或L能够用来代替寄存器B保存第二个数据输入。以下是一个例子：

```
LDA     SRCA    ;LOAD FIRST SET OF INPUTS, FROM SRCA
MOV     C,A     ;SAVE IN THE C REGISTER
LDA     SRCB    ;LOAD SECOND SET OF INPUTS, FROM SRCB
ANA     C       ;AND C WITH A
STA     DST     ;SAVE RESULT IN DST
```

```
LDA     SRCA    ;把第一组输入从SRCA送入累加器
MOV     C, A    ;把累加器内容保存在寄存器C中
LDA     SRCB    ;把第二组输入从SRCB送入累加器
```

```
ANA     C       ;C和A相"与"
STA     DST     ;结果保存在 DST 内
```

**用隐含存储器寻址**

我们并不鼓励用寄存器H或L来保存第二个输入。这两个寄存器主要用来保存数据存储器地址。例如，LDA 和 STA 指令能用以下指令序列来代替：

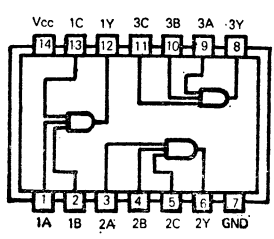
```
LXI     H,SCRA  ;LOAD ADDRESS FOR FIRST SET OF INPUTS INTO H,L
MOV     A,M     ;LOAD FIRST SET OF INPUTS INTO A
LXI     H,SCRB  ;LOAD ADDRESS OF SECOND SET OF INPUTS INTO H,L
ANA     M       ;AND SECOND SET OF INPUTS WITH A
LXI     H,DST   ;LOAD ADDRESS OF DESTINATION INTO H,L
MOV     M,A     ;STORE RESULT IN DST
```

```
LXI     H,SRCA  ;把第一组输入的地址送入H,L
MOV     A, M    ;把第一组输入送入A
LXI     H, SRCB ;把第二组输入的地址送入H, L
ANA     M       ;把第二组输入和A相"与"
LXI     H, DST  ;把目的地址送入H, L
MOV     M, A    ;结果保存在 DST 内
```

注：原程序中的 SCRA、SCRB 应为 SRCA、SRCB。

## 2-7 用微型计算机模拟 7411 三输入、三正"与"门

在 7411 "与"门和 7408 "与"门之间的主要区别是输入信号的数目。7411 产生三个输出信号，每个输出信号是三个输入信号相"与"的结果：




$Y = A \cdot B \cdot C$

## 2-7-1 三输入功能

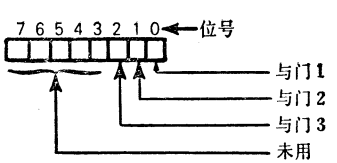
我们再次面临选择。我们可以把三组输入送入三个 CPU 寄存器(一个累加器和另外两个寄存器)，然后在保存结果前完成两次“与”操作：

```
ONE     LDA     SRCA    ;LOAD FIRST SET OF INPUTS, FROM SCRA
TWO     MOV     B,A     ;SAVE IN B REGISTER
THRE    LDA     SRCB    ;LOAD SECOND SET OF INPUTS, FROM SCRB
FOUR    MOV     C,A     ;SAVE IN C REGISTER
FIVE    LDA     SRCC    ;LOAD THIRD SET OF INPUTS, FROM SCRC
SIX     ANA     B       ;AND B WITH A
SEVN    ANA     C       ;AND C WITH A
EIGT    STA     DST     ;SAVE THE RESULT IN DST
```

```
ONE     LDA  SRCA   :将第一组输入从 SRCA 送入累加器
TWO     MOV  B, A   :保存在寄存器 B 内
THRE    LDA  SRCB   :将第二组输入从 SRCB 送入累加器
FOUR    MOV  C, A   :保存在寄存器 C 内
FIVE    LDA  SRCC   :将第三组输入从 SRCC 送入累加器
SIX     ANA  B      :B 和 A 相“与”
SEVN    ANA  C      :C 和 A 相“与”
EIGT    STA  DST    :结果保存在 DST 内
```

以上的指令序列已用序号表示，以便使叙述更易于理解。为了满足汇编语言源程序的需要，并不需要这些序号。

当执行指令 ONE 时，一个 8 位值从地址标号为 SRCA 的存储器字节中送入累加器。假定由下图表示“与”门的输入：




应当注意，上图中的各数据位的指定是完全任意的，只是

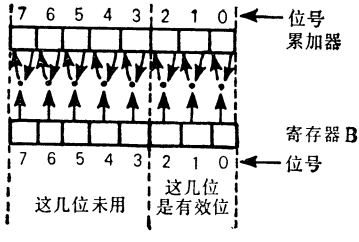
需要使所有后继的输入能保持连贯就可以了。

在执行完指令 ONE 后，第一组输入已在累加器内。假如用直接地址，累加器是数据可以送入的唯一的 CPU 寄存器。因此第一组输入必须保存在另外的寄存器内，以便空出累加器存放第二组输入。指令 TWO 把累加器的内容传送到寄存器 B。

指令THRE 和 FOUR 把第二组输入送入累加器，然后再把它传送到寄存器 C。这里假定对于第二组输入的位分配和上述第一组输入的位分配相同。

用指令 FIVE 把第三组输入，即最后一组输入送入累加器。

ANA 指令将 CPU 寄存器内容和累加器内容相“与”，结果保留在累加器内。指令 SIX 完成第一次相“与”，情况如下：




指令 SEVN 完成第二次“与”操作。这次是累加器和寄存器 C 相“与”。如上所述，累加器开始保存和 B 相“与”的结果，在执行完指令 SEVN 后，累加器内所存放的是三个输入相“与”的结果。

指令 EIGT 把最后的结果送回到由地址标号为 DST 的存储器字节。这样，就完成了对于 7411 “与”门的模拟。

现在来看对于 7411 “与”门的另一种模拟。我们可以把第一组输入送入累加器，而把第二组输入送到另一个寄存器。在

这两组输入相“与”后，可以把第三组输入送到同一个“另一个”寄存器，把它和第一次相“与”的结果再相“与”，然后把结果送回：

```
ONE     LDA     SRCA    ;LOAD FIRST SET OF INPUTS, FROM SCRA
TWO     MOV     B,A     ;SAVE IN B REGISTER
THRE    LDA     SRCB    ;LOAD SECOND SET OF INPUTS, FROM SCRB
FOUR    ANA     B       ;AND B WITH A. THE RESULT IS IN A
FIVE    MOV     B,A     ;SAVE THE RESULT IN B
SIX     LDA     SRCC    ;LOAD THIRD SET OF INPUTS, FROM SCRC

SEVN    ANA     B       ;AND B WITH A
EIGT    STA     DST     ;SAVE THE RESULT IN DST
```

```
ONE     LDA     SRCA    :将第一组输入从 SCRA 送入累加器 A
TWO     MOV     B,A     :保存在寄存器 B 内
THRE    LDA     SRCB    :将第二组输入从 SCRB 送入累加器 A
FOUR    ANA     B       :B 和 A 相“与”，其结果在 A 内
FIVE    MOV     B,A     :保存在寄存器 B 内
SIX     LDA     SRCC    :将第三组输入从 SCRC 送入累加器 A
SEVN    ANA     B       :B 和 A 相“与”
EIGT    STA     DST     :结果保存在 DST 内
```

我们来比较 7411 “与”门的第二种模拟和第一种模拟。指令ONE、TWO 和 THRE 是和第一种模拟相同的，在执行完这三条指令后，第一组输入在寄存器 B 内，而第二组输入在累加器内。

第一组输入 A 存放在寄存器 B 中

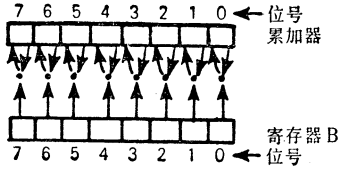
第二组输入 B 存放在累加器内

现在不是把第三组输入立即送入 CPU 寄存器，而是通过执行指令 FOUR 以完成第一组和第二组输入的相“与”。因为相“与”的结果存放在累加器中，所以我们执行指令 FIVE 把这一结果移入寄存器 B。其最终效果是：

“A · B” 存放在寄存器 B 中

现在指令 SIX 把第三组输入 C 送入累加器。指令 SEVN

把第三组输入和第一次相“与”的结果再相“与”，情况如下：




指令 EIGT 把结果从累加器送入地址标号为 DST 的存储器字节中。

## 2-7-2 尽可能减少对 CPU 寄存器的存取

对于 7411 “与”门的模拟，哪一种比较好呢？显然是第二种方案。这里有一个不明显的问题是和不加选择地使用 CPU 寄存器有关的。我们曾随意决定让寄存器 B 保存第一组输入。只要我们在模拟 7411 “与”门时不考虑它的前后的情况是什么，那末寄存器 B 的选择就是任意的；这样的选择既没有什么好处，也不会产生什么问题。

**在CPU寄存器利用中的冲突**

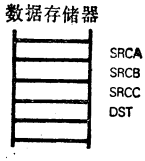
可以肯定，模拟 7411 “与”门这样的指令序列仅仅是整个程序中的一小部分。现在我们又该考虑用寄存器 B 容纳第二组输入是否会妨碍在这以前和以后使用寄存器 B。一个很普通的程序错误就是对 CPU 寄存器使用时发生冲突。例如，假定先前的几个逻辑步已经用寄存器 B 保存中间数据值，那将会怎样呢？现在 7411 模拟将抹去暂存在这个寄存器中的数据。

为了减少使用 CPU 寄存器时发生的冲突，在不需要付出显著代价的情况下，应尽可能少用一些 CPU 寄存器。这一直是选择指令序列应当优先考虑的问题。对于 7411 “与”门的模

拟，只用CPU寄存器B的方法和用CPU寄存器B及C的方法相比，两者所需的指令条数相同。因此，只用CPU寄存器B是个好方法。

**隐含寻址**

现在，我们用隐含寻址来模拟7411“与”门。假定送入“与”门的三个输入依次存放在数据存储器的相继字节中，而目的地紧接在最后的源字节如下图所示：




现在用隐含寻址，其指令序列如下：

```
ONE     LXI     H,SRCA    ;LOAD THE FIRST SOURCE ADDRESS INTO HL
TWO     MOV     A,M       ;LOAD THE FIRST SOURCE INTO THE ACCUMULATOR
THRE    INX     H         ;INCREMENT THE IMPLIED ADDRESS
FOUR    ANA     M         ;AND ACCUMULATOR WITH SECOND SOURCE
FIVE    INX     H         ;INCREMENT THE IMPLIED ADDRESS
SIX     ANA     M         ;AND ACCUMULATOR WITH THIRD SOURCE
SEVN    INX     H         ;INCREMENT THE IMPLIED ADDRESS
EIGT    MOV     M,A       ;SAVE THE RESULT
```

```
ONE     LXI     H,SRCA    ：将第一个源地址送入HL
TWO     MOV     A，M      ：将第一个输入源送入累加器
THRE    INX     H         ：隐含地址增1
FOUR    ANA     M         ：累加器和第二个输入源相“与”
FIVE    INX     H         ：隐含地址增1
SIX     ANA     M         ：累加器和第三个输入源相“与”
SEVN    INX     H         ：隐含地址增1
EIGT    MOV     M，A      ：保存结果
```

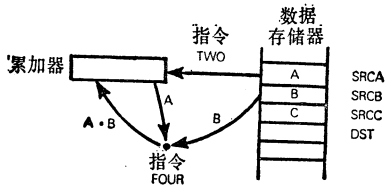
下面介绍指令序列是怎样执行的：

指令ONE把第一个源字节的地址送入寄存器H和L。

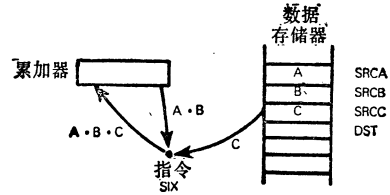
指令 TWO 把由 H 和 L 内容指向的存储器字节的内容送入累加器。

指令 THRE 把寄存器 H 和 L 中的 16 位地址增 1，它现在指向 SRCB。

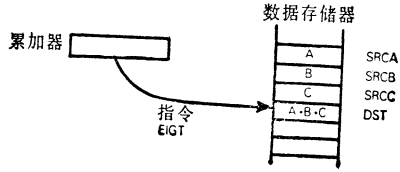
指令 FOUR 把累加器和由寄存器 H 和 L 的内容所指向的第二个输入源相“与”。结果存入累加器。这可用下图说明：




指令 FIVE 和指令 SIX 是：隐含地址增 1 和重复“与”操作，这次是把第一、第二两组输入相“与”的结果和第三组输入再相“与”。这可用下图说明：




在 H 和 L 中存放的地址再增 1，从而指向 DST。指令 EIGT 把结果存入目的地，如下图所示：



## 2-7-3 存储器利用率和执行速度的比较

现在，我们有三种用来模拟 7411 “与”门的程序：

程序 1 用直接寻址和三个 CPU 寄存器。

程序 2 用直接寻址和两个 CPU 寄存器。

程序 3 用隐含寻址。

让我们来比较存放各种程序所需的目标程序的字节数和执行每种程序所需的时钟脉冲的周期数。其结果归纳在表 2-1 中。表 2-1 包括有以上每种程序的指令助记符，它们可以帮助你了解目标程序的总字节数以及总的执行周期是如何计算的。如果要核实表 2-1，请查阅第六章的有关数据。

表 2-1 模拟 7411 “与”门时存储器利用率和执行速度的比较

| 程序 1 | | | 程序 2 | | | 程序 3 | | |
|---|---|---|---|---|---|---|---|---|
| 助记符 | 字节数 | 周期数 | 助记符 | 字节数 | 周期数 | 助记符 | 字节数 | 周期数 |
| LDA | 3 | 13 | LDA | 3 | 13 | LXI | 3 | 10 |
| MOV[1] | 1 | 5 | MOV[1] | 1 | 5 | MOV[2] | 1 | 7 |
| LDA | 3 | 13 | LDA | 3 | 13 | INX | 1 | 5 |
| MOV[1] | 1 | 5 | ANA[1] | 1 | 5 | ANA[2] | 1 | 7 |
| LDA | 3 | 13 | MOV[1] | 1 | 13 | INX | 1 | 5 |
| ANA[1] | 1 | 4 | LDA | 3 | 4 | ANA[2] | 1 | 7 |
| ANA | 1 | 4 | ANA[1] | 1 | 4 | INX | 1 | 5 |
| STA | 3 | 13 | STA | 3 | 13 | MOV[2] | 1 | 7 |
| TOTAL | 16 | 70 | TOTAL | 16 | 70 | TOTAL | 10 | 53 |

[1] 寄存器——寄存器指令格式
[2] 寄存器——存贮器指令格式

**直接寻址与隐含寻址**

程序 1 和程序 2 所用存储单元的数目和执行速度是相同的，这并不足为奇，因为它们所执行的指令是一样的，只是顺序有所变更而已。程序 3 采用了完全不同的原理，它用隐含寻址而不是用直接寻址来模拟 7411“与”门。其结果是引人注目的。它不仅节省了六个存储器字节，而且把程序执行时间也缩短了 24%。但是用程序 3 模拟有个附加的约束：即三个数据源和目的地必须占用四

个连续的存储器字节。

**对几种程序的评语**

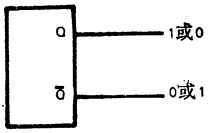
如何来评定这三种模拟方案呢？我们已经得出程序2优于程序1的结论，因为程序1毫无理由地多用了一个CPU寄存器。假如对数据源和目的地单元的约束条件是能够放宽的话，那末程序3也显然优于程序2。

程序3优于程序2这一事实值得再次提出，这一点在“微型计算机入门”一书中已经强调过，在微型计算机应用中不加区别地使用直接寻址会造成浪费。在使用小型计算机或大型计算机的背景下，在程序设计者看来，隐含寻址也许是原始的，然而它却是经济的。

## 2-8 用微型计算机模拟可预置和可清除的7474正跳沿触发双D触发器

在具体考察7474触发器之前，让我们先一般地探讨一下触发器。首先给出几个定义。

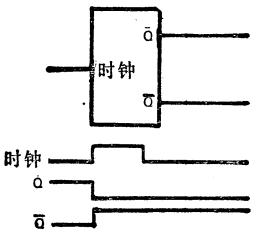
### 2-8-1 触发器的数字逻辑描述




触发器是一个双稳态逻辑器件，也就是说，它可以处于两个稳定状态中的一个状态。7474型触发器有Q和$\bar{Q}$两个输出端；于是这两个稳定状态可以如43页上图所示。

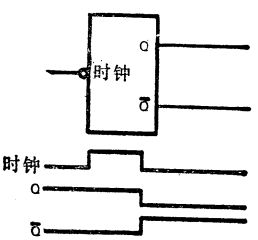
**正跳沿触发**

一个时钟信号使触发器从一个稳定状态变化到另一个稳定状态。一个正跳沿触发式触发器由“0”到“1”跳变的时钟信号来触发；




**负跳沿触发**

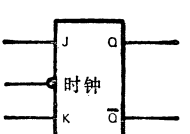
一个负跳沿触发式触发器由“1”到“0”跳变的时钟信号来触发：




**JK 触发器**

JK 触发器预先安排了将由下一个时钟脉冲沿触发而产生的 **Q** 和 $\bar{\mathbf{Q}}$ 的输出，其规律如下：

| 时钟信号出现时，J 和K 的状态 | | 时钟信号出现时所产生的输出 | |
|---|---|---|---|
| J | K | Q | $\bar{Q}$ |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 状态不变 | |
| 1 | 1 | 不管原来是什么状态，都要改变状态 | |




**时钟信号**

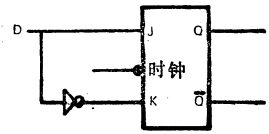
在上表中，对正跳沿触发的器件，时钟信号是一个零到一跳变的信号；对负跳沿触发的器件，时钟信号则是一个“1”到“0”跳变的信号。这一“时钟

信号”的定义也适用于下面将要介绍的D触发器。

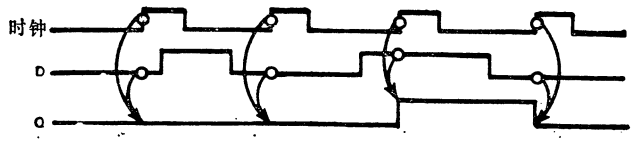
**D 触 发 器**

把J端的输入经过反相作为K端的输入，这就形成了D触发器。以下是这样得到的D触发器的性能：

| 时钟信号出现时J和K的状态 | | 时钟信号出现时所产生的输出 | |
|---|---|---|---|
| J=D | K=$\overline{J}$ | Q | $\overline{Q}$ |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |




下面是一个正跳沿触发的D触发器时间图：




所以D触发器的输出总是反映时钟脉冲到来前所呈现的输入状态。

**触发器预置**
**触发器清除**

“预置”(PR)输入表示触发器的输出可以被强制置为Q=1和$\overline{Q}$=0。当预置为“1”时，就强制将触发器置成这种状态。

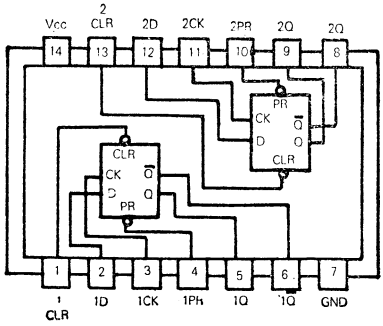
“清除”(CLR)输入恰好与“预置”输入相反，当它为“1”时，“清除”输入强制Q=0和$\overline{Q}$=1。

综合以上所给定义，就可得到7474触发器的功能表 见 第45页上图。

在功能表中，箭头↑表示由“0”到“1”跳变的时钟脉冲。$H^*$ 表示不稳定状态。$Q_0$是Q以前的状态。X表示任意状态。

功能表

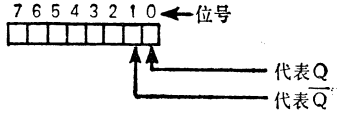
| 输入 | | | | 输出 | |
|---|---|---|---|---|---|
| 1PR或<br>2PR | 1CLR 或<br>2CLR | 1CK或<br>2CK | 1D 或<br>2D | 1Q或<br>2Q | 1$\overline{Q}$ 或<br>2$\overline{Q}$ |
| L | H | X | X | H | L |
| H | L | X | X | L | H |
| L | L | X | X | H* | H* |
| H | H | ↑ | H | H | L |
| H | H | ↑ | L | L | H |
| H | H | L | X | $Q_0$ | $\overline{Q}_0$ |




## 2-8-2 用汇编语言对触发器的一种模拟

现在，当试图模拟7474触发器时，第一个问题是由于在微型计算机指令系统中没有时钟信号，所以我们必须转而假定触发事件是用执行一条相应的指令，而不是用时钟信号跳变来实现的。

怎样表示输出Q和$\overline{Q}$呢？可以用存储器中的两位来表示这两个输出(见右图)。



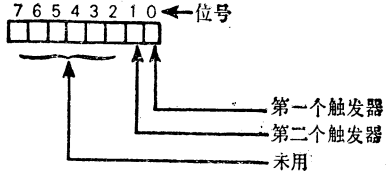

因为我们处理的是数据而不是信号，所以$\overline{Q}$是多余的。而且单个触发器可以由存储器的一位来模拟。由于一个7474器件包含有两个触发器，所以需要两个存储器位分别表示制做在芯

片上的每一个触发器。

对于这个结论是没有什么值得惊讶的：微型计算机读/写存储器的每一位就是一个简单的、双稳态元件；也可能确实就是一个触发器。

7474触发器的逻辑功能可以用一些指令来模拟，这些指令是：对存储器位清“0”，对存储器位置“1”，或者把某一个二进制数存入存储器位。

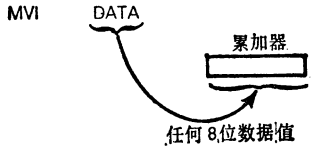
假定存储器位的分配如下：




7474的功能表现在变成以下几条指令：

| | 预置 | 清零 | D | 第一个触发器 | 第二个触发器 |
|---|---|---|---|---|---|
| | L | H | X | MVI 1 | MVI 2 |
| or | H | H | H | STA FLP | STA FLP |
| | H | L | X | MVI 0 | MVI 0 |
| or | H | H | L | STA FLP | STA FLP |
| | L | L | X | 不用 | |

根据上表，MVI对累加器内容的作用如下：




STA 指令把最后得出的累加器内容存入地址标号为FLP的存储器字内。而预先假设地址标号为FLP的存储器字的第0位和第1位相当于7474器件的两个触发器。

### 2-8-3 用微型计算机模拟一般触发器

总之，在微型计算机系统中，一个触发器成为读/写存储器的一位。

在微型计算机系统的模拟程序中，所有的触发器都是一样的。触发器逻辑功能可以归纳为以下四个问题：

1）什么时候执行一条指令把存储器位置“1”？

2）什么时候执行一条指令把存储器位置“0”？

3）什么时候执行一条指令把一个二进制数字存入存储器位？

4）什么时候执行一条指令读出某一存储器位的内容？

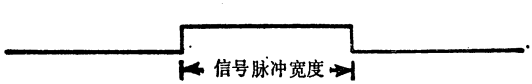
## 2-9 用微型计算机模拟实时器件

我们将考察两种类型的实时器件：单冲触发电路（包括单稳多谐振荡器）和主从触发器。我们具体介绍的器件是：

- signetics 555 单稳多谐振荡器
- 74121 单稳多谐振荡器
- 74107 带清零端的双 J-K 主从触发器

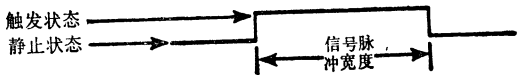
**单冲触发电路**

单冲触发电路是一种产生具有一定时间宽度信号脉冲的器件：




**单稳多谐振荡器**

单稳多谐振荡器是一种具有一个稳定状态或静止状态的器件。它产生如上所说的单冲触发输出信号。这里脉冲是处于不稳定状态或触发状态：




这种器件是一种“多谐振荡器”，因为它能输出一连串连续的信号，很象时钟信号。换句话说，一个单稳多谐振荡器的输出包含一连串连续的单冲触发信号。

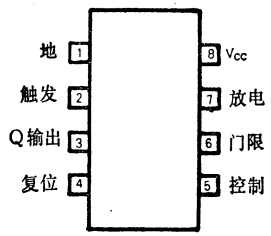
信号脉冲的时间宽度是一个实时值，可以是若干微秒、毫秒甚至秒。

**主从触发器**

主从触发器是一种根据早先输入信号状态产生输出信号的触发器。这里，我们再次遇到一个实时值，在输入信号和输出信号之间的时间延迟。

## 2-9-1 555 单稳多谐振荡器

signetics 555 单稳多谐振荡器可以用下图说明：



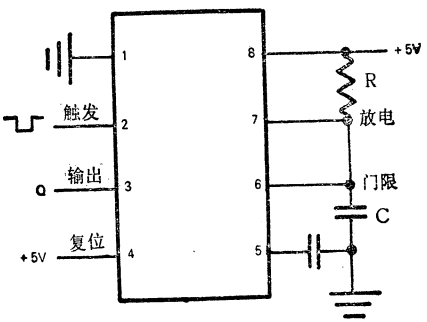

触发器输入端（第 2 端）上时钟信号的负跳沿使Q输出端产生一个正跳变。Q端高电平输出的持续时间由连接到放电引线端和门限引线端（分别为 7 和 6 脚）的电阻/电容电路所控制。

“复位”是一个标准的复位输入信号；一个低电平输入将使Q端输出保持低电平。

“控制”引线端用于控制多谐振荡器内的电压；它对全面了解 555 器件的工作情况并没有重要意义。

地线和电源端（分别为 1 和 8）由其本身的名称就能说明了。

下图是构成 555 单稳多谐振荡器的一种方式：




当一个由高变低的信号电平传送到触发输入端，在 6 端和地之间的电容就充电。Q 端输出高电平的持续时间由电阻 R 和电容 C 所控制的门限端和放电端的信号电平来控制。这个时间由下式给出：

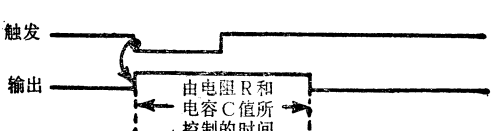
$$T = 1.1\,RC$$

式中 T 是以秒为单位的时间

R 是以兆欧为单位的电阻

C 是以微法为单位的电容

输出信号脉冲的产生如下图所示：




## 2-9-2 74121 单稳多谐振荡器

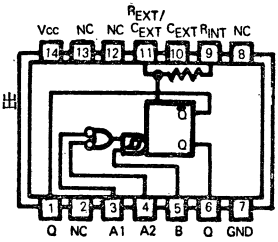
74121 单稳多谐振荡器可以说明如下：

功能表

| 输入 | | | 输出 | |
|---|---|---|---|---|
| A1 | A2 | B | Q | $\overline{Q}$ |
| L | X | H | L | H |
| X | L | H | L | H |
| X | X | L | L | H |
| H | H | X | L | H |
| H | ↓ | H | ⊓ | ⊔ |
| ↓ | H | H | ⊓ | ⊔ |
| ↓ | ↓ | H | ⊓ | ⊔ |
| L | X | ↑ | ⊓ | ⊔ |
| X | L | ↑ | ⊓ | ⊔ |

单稳态输出

单冲触发输出




如果在 $A_1$、$A_2$ 或 B 端上输入一个恒定的低电平信号，就能使 74121 单稳多谐振荡器保持在它的稳定状态，Q 端输出低电平，$\overline{Q}$ 端输出高电平。在 $A_1$ 和 $A_2$ 都输入高电平具有同样的效果。

有五种输入信号的组合能产生单冲触发输出。这些组合已表示在以上的功能表中。

功能表中所用符号的含义如下：

X 表示“任意值”

↓ 表示 1 ⟶ 0 的逻辑跳变

↑ 表示 0 ⟶ 1 的跳变

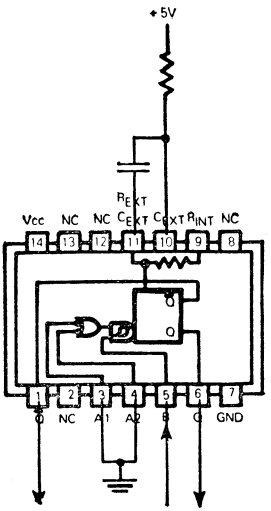
⊓ 表示具有零单稳逻辑电平和一脉冲电平的单冲触发脉冲

⊔ 是⊓的非

正如对 signetics 555 单稳多谐振荡器的描述一样，单冲触发输出的持续时间也是由一个电阻-电容网络决定的，但也有一些不同。74121 有一个内部电阻，把 $R_{INT}$（端 9）与 $V_{CC}$（端 14）相连就可利用这一电阻。在 $R_{INT}$（端 9）或 $R_{EXT}$（端 11）与 $V_{CC}$（端 14）之间可外接一可变电阻。

如果需要的话，可在 $C_{EXT}$（端 10）与 $R_{EXT}$（端 11）之间外接一个定时电容。

下图示出 74121 单稳多谐振荡器连接方法中的一种：




74121 单稳多谐振荡器的上述接法相当于功能表中的底下两行。

一个外部电阻/电容网络控制着单冲触发脉冲的持续时间。每个单冲触发脉冲要由端 5 （B）上的一个由低变高的跳变来触发。

**从编写程序的角度看，74121 单稳多谐振荡器的重要特点只有两个：**

1）单稳输出等效于一些数值固定的二进制数字。任何把“0”或“1”送入寄存器任一位的立即指令都能模拟单稳态输出。下面是一个例子：

```
MVI     B,4     ;SET BIT 3 OF REGISTER B TO 1. RESET ALL OTHER BITS
```

MVI B，4 ：寄存器 B 的第 3 位置“1”。其它所有各位置“0”。

现在寄存器 B 的第 3 位就等效于一个触发器；寄存器 B 的

其它各位以及其它每个寄存器的各位同样也等效于一个触发器。

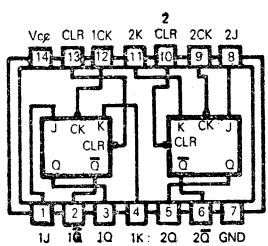
2）一个单冲触发输出变成为一个具有固定值的时间延迟。我们将说明时间延迟在微型计算机系统中是如何可以计算出来的，但首先我们来考察一下74107主从触发器。

## 2-9-3 74107 带清零端的双J—K主从触发器

考察74107主从触发器。这种触发器的功能可以用下图说明：

功能表

| 输入 | | | | 输出 | |
|---|---|---|---|---|---|
| 1CLR或2CLR | 1CK或2CK | 1J或2J | 1K或2K | 1Q或2Q | $1\overline{Q}$或$2\overline{Q}$ |
| L | X | X | X | L | H |
| H | ⎍ | L | L | 维持原状 | |
| H | ⎍ | H | L | H | L |
| H | ⎍ | L | H | L | H |
| H | ⎍ | H | H | 不管原先什么状态，都要改变状态 | |



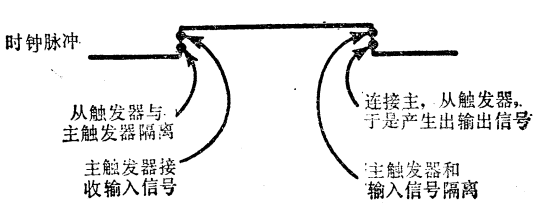

⎍ 表示一个时钟脉冲；其用法将在下面介绍。

X 表示“任意状态”

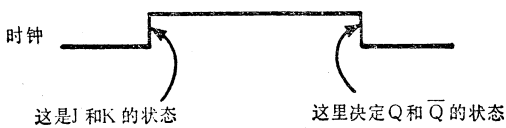
我们来看上述的功能表。除非你熟悉这种型号的逻辑器件，否则单从表上是看不出它们的特性的。

**主从触发器**

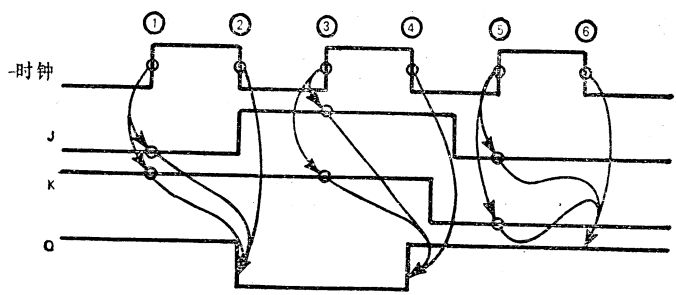
“主从”这一含义就表示出这种电路事实上由两个触发器组成。所以，在上述74107器件中有四个触发器。每对主从触发器对时钟信号的响应如下：



这个时钟信号响应的特点是触发器的输入必须出现在时钟信号的正跳沿处；当时钟信号是高电平时，这些输入必须是不变的。然而，直到时钟信号负跳沿时，触发器输出才改变状态。时钟信号可以用来产生时间延迟。74107 触发器的输出取决于早一些时间就存在的输入信号状态。这可用下图说明：




以下是一个具体的例子：




下面，依据时钟信号上用圆圈圈住的号码来逐个解释上述的时间关系图。

在②时，Q输出变成低电平，因为在①时 J 是低电平，K 是高电平。

在④时，Q端改变状态，因为在③时 J 和K 都是高电平。

在⑥时，Q端保持不变，因为在⑤时 J 和K 都是低电平。

## 2-9-4 用微型计算机模拟实时

555 单稳多谐振荡器和主从触发器的主要特性是什么呢？用微型计算机对这些器件进行模拟时，只有一个特性对我们现在的讨论是重要的，那就是实时概念。

555 单稳多谐振荡器在它的输出端产生高逻辑电平脉冲，而这一电平的持续时间是一个可控制的实时函数。

74107 主从触发器允许依据在先前某一实时上呈现的各输入状态而产生一个输出信号。

## 2-9-5 微型计算机定时指令循环

**短时间间隔的定时**

只要微型计算机不同时用来完成任何其它操作的话，用微型计算机系统产生一个时间延迟是足够简单的。现在来看以下的指令序列：

```
Cycles
                MVI   A,TIME  ;LOAD TIME CONSTANT INTO ACCUMULATOR
 5        LOOP  DCR   A       ;DECREMENT ACCUMULATOR
10              JNZ   LOOP    ;REDECREMENT IF NOT ZERO
```

周期数

```
                MVI   A, TIME;  把时间常数送入累加器
5        LOOP   DCR   A       ;  累加器减 1
10              JNZ   LOOP    ;  假如不为零，再减 1
```

以上指令序列把一个用标号 TIME 表示的数据值送入累加器。累加器一直递减到零为止。在此期间，程序连续执行。让我们假定微型计算机系统采用的是 500 毫微秒的时钟脉冲。执行 DCR 和 JNZ 指令总共用了 15 个周期，等效于 7.5 微秒。这意味着上述程序序列可以引起的时间延迟最小为 7.5 微秒（当 TIME 等于 1 时），以每步 7.5 微秒递增，一直增加到最大

时间延迟为 1920 微秒（它等效于 $7.5\times256$）。当 TIME 起始值为零时，就会形成这个最大的时间延迟值，因为在测试 TIME 是否为零之前，它就已经递减了；所以逾时是发生在当 1 递减到 0 时，而不是当 0 递减到 $FF_{16}$ 时

**长时间间隔的定时**

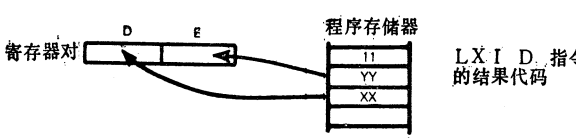
只要有一个 16 位的计数器就能产生长时间延迟。下面是相应的指令序列：

```
Cycles
                LXI   D,T16   ;LOAD TIME CONSTANT INTO D AND E
5         LOOP  DCX   D       ;DECREMENT DE
5               MOV   A,D     ;TEST FOR ZERO BY ORING
4               OR    E       ;D AND E CONTENTS VIA ACCUMULATOR
10              JNZ   LOOP
```

周期数

```
                LXI   D, T 16;  把时间常数送入D和E
5         LOOP  DCX   D       ;  D E减 1
5               MOV   A, D    ;  D和E的内容在累加器内相
4               OR    E       ;  "或"，然后测试它们的内容是否为零
10              JNE   LOOP
```

LXI 指令把标号 T 16 所表示的一个 16 位值送入 DE 寄存器对。LXI 指令是一条立即指令，它所产生的结果代码是三个字节。当执行 LXI 指令时，所发生的情况如下图所示：




**用 DCX 指令测试状态**

DCX 指令把寄存器 DE 中的 16 位值作为单个数据实体进行递减。然而，8080指令系统有一个别扭的地方就是忽略了根据 16 位递减的

结果来置状态位。这就意味着我们没有一种直接的方法能够即时知道寄存器对 DE 的内容是否为零。为了进行这一测试，我们必须把寄存器D的内容移入累加器，然后和寄存器E的内容相“或”。如果累加器中的结果为“0”，则寄存器D和E的内容必定都为“0”。如果结果不为“0”，则回过来再对 16 位值进行递减。

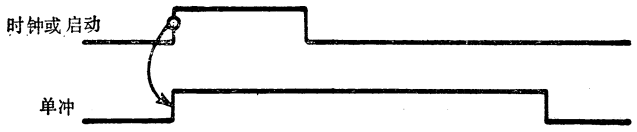
可以看到，长时间间隔指令循环执行一次需要 24 个周期。还是假定微型计算机是由 500 毫微秒的时钟来驱动的，则每执行一次指令循环需要 12 微秒。T 16 的最小值应为 1。最大值还是 0，因为递减发生在测试零状态之前；倘若最初置入寄存器D和E的是零，那么它在第一次零测试之前就递减到 $FFFF_{16}$。因而，长时间间隔指令循环产生的时间延迟是以 12 微秒逐步递增的，从最小值 12 微秒一直可以递增到最大值 0.786432 秒。

$$FFFF_{16}=65535_{10}$$

$$12\times 65536=786432\text{ 微秒}=0.786432\text{ 秒}$$

**时间延迟的启动**

现在，对单冲触发器模拟的真正复杂的地方是：我们虽然可以算出时间延迟，但却难以确定时间延迟从何时开始算起。对数字逻辑器件来说，这个回答是简单的：当输入信号改变状态时，时间延迟就开始了：




在微型计算机程序中与这一概念相对应的是，我们必须等

待另外某一程序序列执行完毕以后再开始时间延迟。这个概念可以说明如下：

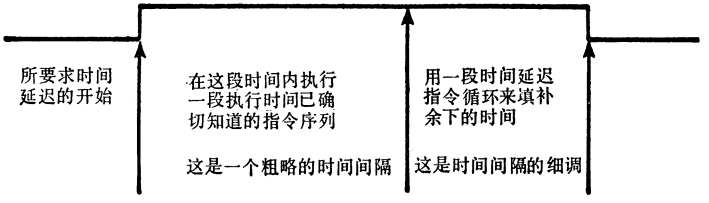
```
          .
          .
          .
          JMP     DELY    ;LAST INSTRUCTION OF SOME PRIOR SEQUENCE
          .

          .
DELY      MVI     A,TIME  ;SHORT TIME INTERVAL INSTRUCTION
LOOP      DCR     A       ;SEQUENCE
          JNZ     LOOP
```

```
          —
          —
          —
          JMP     DELY            ；某一先前序列的最后一条指令
          —
          —
          —
          DELY    MVI     A,TIME  ；短时间间隔指令序列
          LOOP    DCR     A       ；
                  JNZ     LOOP
```

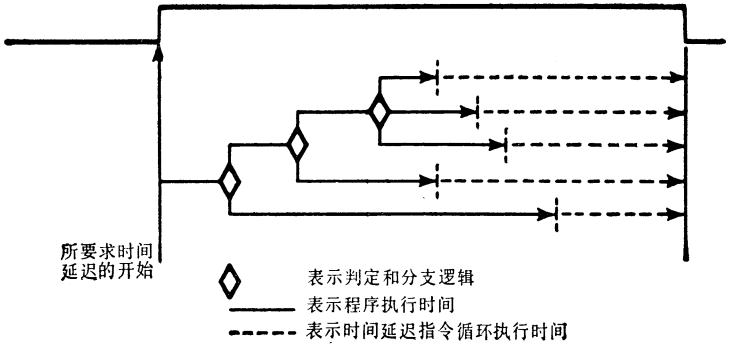
**在时间延迟期间执行程序**

正如我们已介绍过的，在微型计算机系统中，执行指令循环所产生的与时间延迟有关的另一类问题是：从本质上说，微型计算机在时间延迟期间所做的工作是毫无用处的。如果我们能够安排微型计算机在时间延迟期间执行一段程序，就能够简单地解决这个问题了。这可以举例说明(见第58页上图)。

必须假设我们能计算出在单冲触发时间延迟内所执行的程序的确切时间；而且，所计算出的时间必须小于或等于这一时间延迟。能符合上述要求的程序是不多的。例如，倘若根据目前的条件可以执行的指令序列不止一种，那末执行程序所需的




时间可以有很多不同的数值。然而，只要这些分支的数目是固定的、而且是可以识别的，则这一问题还是可以处理的，这可用下图说明：




现在，程序分支的每“支”将以下列方式结束：

```
MVI     A,DLY1    ;LOAD FIRST TIME DELAY
JMP     LOOP      ;START TIME DELAY LOOP
.
.
.
MVI     A,DLY2    ;LOAD SECOND TIME DELAY
JMP     LOOP      ;START TIME DELAY LOOP
.
.
.
```

```
        MVI     A,DLY3  ;LOAD THIRD TIME DELAY
        JMP     LOOP    ;START TIME DELAY LOOP
        .
        .
        .
        MVI     A,DLY4  ;LOAD FOURTH TIME DELAY
        JMP     LOOP    ;START TIME DELAY LOOP
        .
        .
        .
        MVI     A,DLY5  ;LOAD FIFTH TIME DELAY
        JMP     LOOP    ;START TIME DELAY LOOP
        .
        .
        .
LOOP    DCR     A       ;SHORT TIME INTERVAL INSTRUCTION
        JNZ     LOOP    ;SEQUENCE
```

```
MVI  A, DLY 1   ; 输入第一个时间延迟
JMP  LOOP       ; 启动时间延迟循环
—
—
—
MVI  A, DLY 2   ; 输入第二个时间延迟
JMP  LOOP       ; 启动时间延迟循环
—
—
—
MVI  A, DLY 3   ; 输入第三个时间延迟
JMP  LOOP       ; 启动时间延迟循环
—
—
—
MVI  A, DLY 4   ; 输入第四个时间延迟
JMP  LOOP       ; 启动时间延迟循环
—
—
—
MVI  A, DLY 5   ; 输入第五个时间延迟
```

```
        JMP   LOOP        ; 启动时间延迟循环
        —
        —
        —
LOOP    DCR   A           ; 短时间间隔指令
        JNZ   LOOP        ; 序列
```

微型计算机程序中包含许多条件转移是很普通的事情；根据目前状态的不同组合，或许有很多不同的、可能的执行时间。在要求时间延迟的时间间隔中，执行一段程序现在就变得不切实际了，因为计算大量程序分支的剩余时间所需要的逻辑实在是太复杂了。

## 2-9-6 对于数字逻辑模拟的局限性

只要在时间延迟期间不需要执行另一程序，或者在时间延迟期间只是执行一段具有很少分支的、十分简单的指令序列，那末，一台 8080 微型计算机就能够计算时间延迟。

**同时发生的时间延迟**

你不能模拟一些同时发生的时间延迟，也不能模拟必须在不能确定的并行程序的执行过程中同时发生的时间延迟。所有这样的时间延迟都必须由外部逻辑来处理。

## 2-9-7 与外部单冲触发电路的接口

应该注意，即使外部逻辑也许必须产生时间延迟，但由微型计算机系统来触发时间延迟的开始以及为外部逻辑通知时间延迟的结束是十分容易的。

**单冲触发器的启动**

我们能够通过简单地输出一个恰当的二进制数来标志时间延迟的开始。再来看用信号反相器模拟将“信号输出”接到外部逻辑的方法。

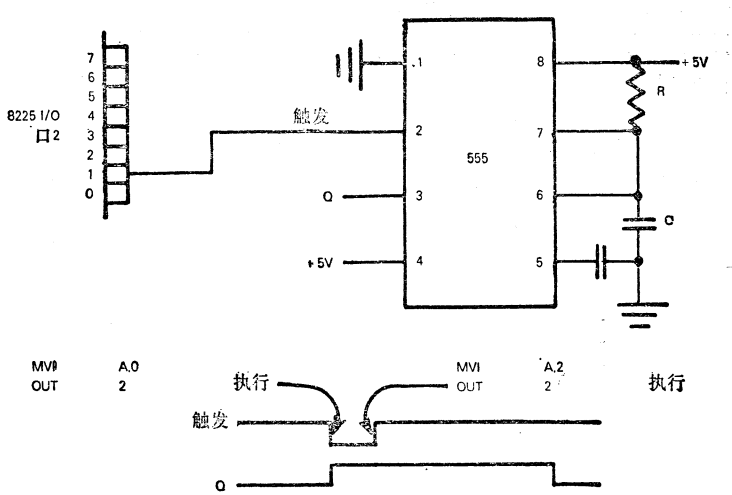
输出一个信号到外部逻辑确实是十分容易的。请看以下四条指令：

```
MVI     A,0     ;LOAD A 0 INTO THE ACCUMULATOR
OUT     2       ;OUTPUT VIA I/O PORT 2
MVI     A,2     ;LOAD A 1 INTO THE ACCUMULATOR BIT 1
OUT     2       ;OUTPUT VIA I/O PORT 2
```

```
MVI   A，0   ：把“0”送入累加器
OUT   2      ：经过I/O口2输出
MVI   A，2   ：把“1”送入累加器第1位
OUT   2      ：经过I/O口2输出
```

在I/O口2的端1上输出的是“1”。假定和I/O口有关的引线端被接到多谐振荡器的触发端，而且该点原先就是高电平，那么上述指令执行一次，就能触发一个单冲触发器。

这可用下图说明：

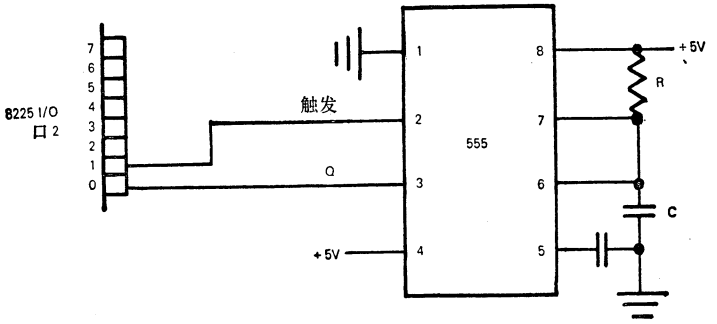



外部逻辑通知时间延迟结束也是容易的。

**利用状态位标识单冲触发电路逾时**

如果我们涉及“大于或等于”逻辑，那么把单冲触发器的输出连接到微型计算机I/O口的另一端子上是完全必要的。

到达I/O口各端子上的一些信号一经到达，即被缓冲。微型计算机所执行的程序在任何时间内都能够




输入I/O口的内容，并测试接至Q输出端的第0位的状态。当发现这一位等于“0”时，微型计算机程序逻辑就知道时间间隔已逾时。

以下指令序列将测试I/O口，恢复单冲触发器的输入和清除由I/O口2的第0端所置成的“时间间隔结束”状态。

```
        IN      2       ;INPUT CONTENTS OF I/O PORT 2 TO ACCUMULATOR
        ANI     1       ;MASK OUT ALL BITS BAR BIT 0
        JNZ     NEXT    ;CONTINUE IF BIT IS 1
;TIME OUT PROGRAM BEGINS HERE
        .
        .
        .
NEXT                    ;TIME NOT OUT PROGRAM BEGINS HERE
```

```
        IN      2       ;输入 I/O 口 2 的内容到累加器
        ANI     1       ;除第 0 位外屏蔽所有各位
        JNZ     NEXT    ;假如这一位是“1”则继续执行
                        ;逾时程序由此开始
```

```
        —
        —
        —
NEXT                    ;未逾时的程序由此开始
```

IN 指令把 I/O 口 2 的当前内容传送到累加器。

接着 ANI 指令把累加器的所有各位置“0”，但对应于I/O口 2 的第 0 端那一位除外：

```
76543210◄— 位号
XXXXXXXY   累加器内容
00000001   16进制01
0000000Y   相与的结果
```

假如从I/O口 2 的第 0 端输入的二进制数字是“1”，那么 Q 输出仍然是高电位。JNZ NEXT 指令简单地继续执行程序。

假如 I/O 口 2 的第 0 位是零，那么时间延迟已超过规定值；我们转移到紧接在发生逾时后的一段程序指令序列。

## 2-9-8 逾时和中断

逾时结束的确切时刻可以用中断通知微型计算机。现在，一旦单冲触发电路逾时，它将强制微型计算机系统停止执行现在正在执行的任何程序。一条转移指令将强制执行另外某一段程序，此段程序是专门为响应逾时而设计的。

与此有关的中断程序设计的考虑比我们在第二章涉及的范围更加复杂。所以我们把中断处理的详细情况放到本书的最后再予讨论。目前，只要了解使用中断逻辑能把逾时的确切时刻通知微型计算机系统就已足够。

# 第三章　数字逻辑的直接模拟

在第二章，我们所模拟的各种分立元件的逻辑器件并不是随意选择的；如果将它们正确地排列起来，就能用它们模拟图3-1所示的逻辑。这种逻辑是Qume Q系列和Sprint系列打印机的打印机接口的一部分。图3-2是与图3-1一致的时间图。下面对此二图做一简单介绍。

这一章的目的是给出微型计算机汇编语言程序和数字逻辑设计之间的一一对应关系。这样的对应关系是人为构成而不是自然形成的，这就是你必须理解的。编写微型计算机程序应该面向微型计算机本身的性质，而不是面向数字逻辑的特性。

编写微型计算机程序的正确方法将在第四章的前面介绍。

然而，在本章对于数字逻辑设计和编写微型计算机程序将给予同样的重视。这是把这两种概念联系在一起的一章；因而也是本书中最重要的一章。如果你是一名逻辑设计人员，这一章对你是重要的，因为它将打消你对于微型计算机不适用的数字逻辑概念。如果你是一名程序员，这一章对你也是重要的，因为它将告诉你一个新的编写程序的目标，高效能地实现逻辑。

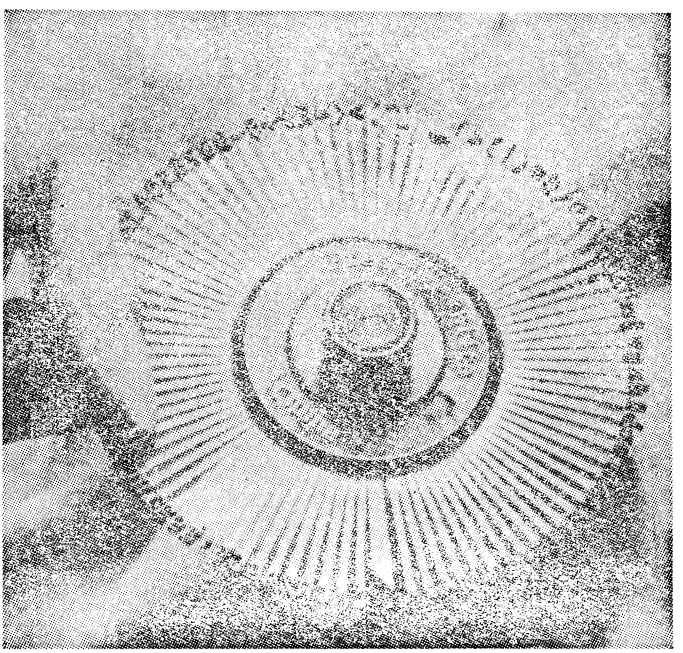
为了达到本章的目的，我们将介绍图3-1和图3-2所示的逻辑；叙述是很详细的，至于即便你不是逻辑设计人员，你也能读完这一章。我们在介绍逻辑设计的过程中，将在适当的时候把汇编语言加进去。

尤其重要的是，如果你已懂得数字逻辑，那么你可以确信只需阅读这一章中的粗体字部分就足够了。对于图3-1的逻辑已经做了充分详细的叙述，完全可以满足程序员或是没有逻辑

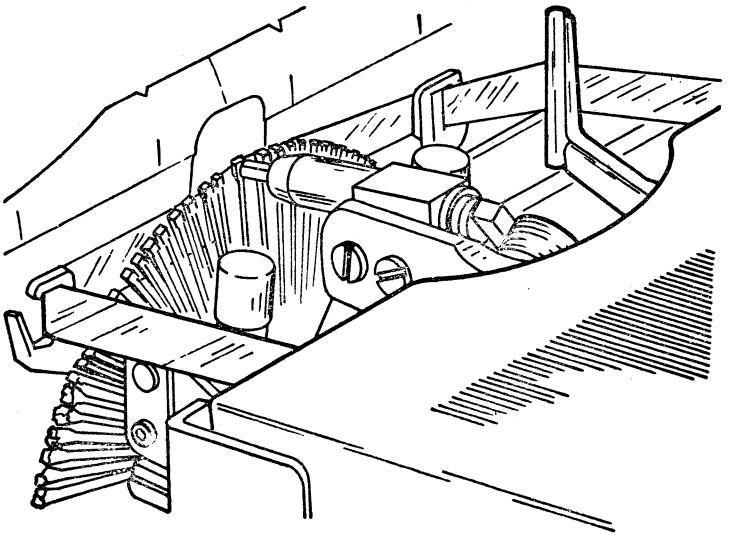
预备知识的读者的需要。

## 3-1 QUME 打印机是如何工作的

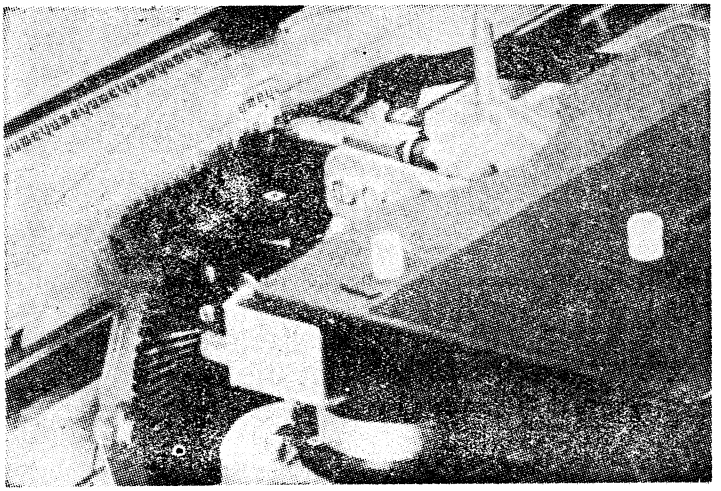
活动的 Qume 打印机部件是一个有 96 个字瓣的印字轮，每个瓣上部有一个字符。



字符是这样打印的，当印字轮转动到使相应的字瓣位于由圆筒形线圈驱动的打印锤的前面时，启动打印锤；击打字轮的该字瓣，于是字瓣就在纸上印出字符。



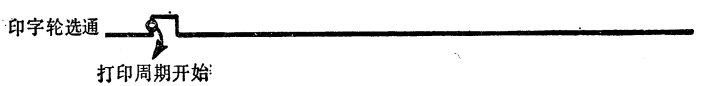
当一字符不处于打印过程中时，印字轮定位在与一个短瓣相垂直的位置上，从而能看得到刚刚打印的字符：



作为打印周期的一部分，必须移动打印机色带和输纸托架。按照一定的事件次序打印每个字符总称为“打印周期”。图3-1所示的逻辑电路就是控制这一字符打印周期的。以下所述是在打印周期里必然发生的事件：

**印字轮选通**

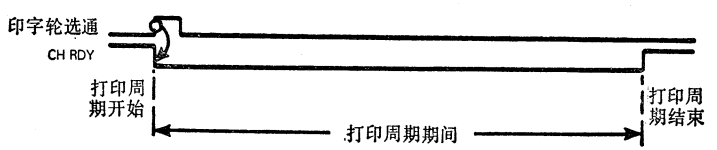
1）首先，必须启动打印周期。PW STROBE（印字轮选通）信号变成高电平，启动打印周期。




**印字轮准备**

**CH RDY**

2）打印周期要持续一定的时间间隔。显然，在这段时间里不应启动另一个打印周期。因此负责产生“印字轮选通”真值的那一外部逻辑应能给出一个标识打印周期持续时间的信号，这个信号就是“印字轮准备”，也简写为CH RDY：

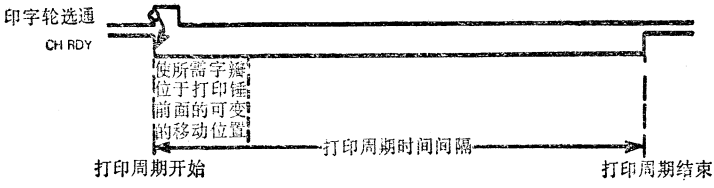



注：CH RDY—印字轮准备，以下均同。

在已确信当前打印周期尚未结束之前，外部逻辑不会开始打印下一字符的情况下，现在就能够使真正促使打印字符的那些事件序列继续进行下去了。

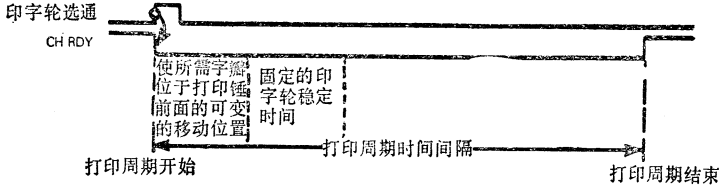
3）印字轮从可见位置开始移动，一直到相应的字瓣移到打印锤的前面为止：

显然，对于距离可见位置较远的字瓣进行定位要比对于和可见位置相邻的那一字瓣的定位需要更长的时间。因此印字轮



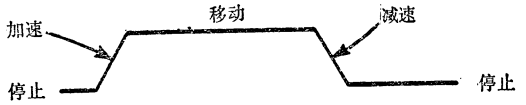

定位逻辑需要有可变的时间延迟。

4）在启动打印锤之前，必须给印字轮一定的稳定时间。一个固定的，2 微秒的时间延迟已经足够：

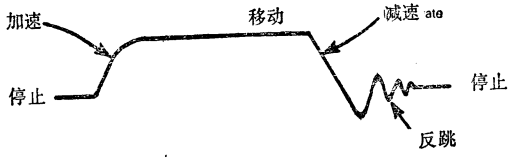



**稳定时间延迟**

稳定时间延迟是为任何形式的机械运动提供支持的逻辑的很重要的方面。很容易画出一条如下所示的平滑的线段来表示运动的速度：




但是实际上，运动的情形如下图所示：




跟在减速后面的反跳（颤动）必须经过一段稳定时间延迟才能消失。

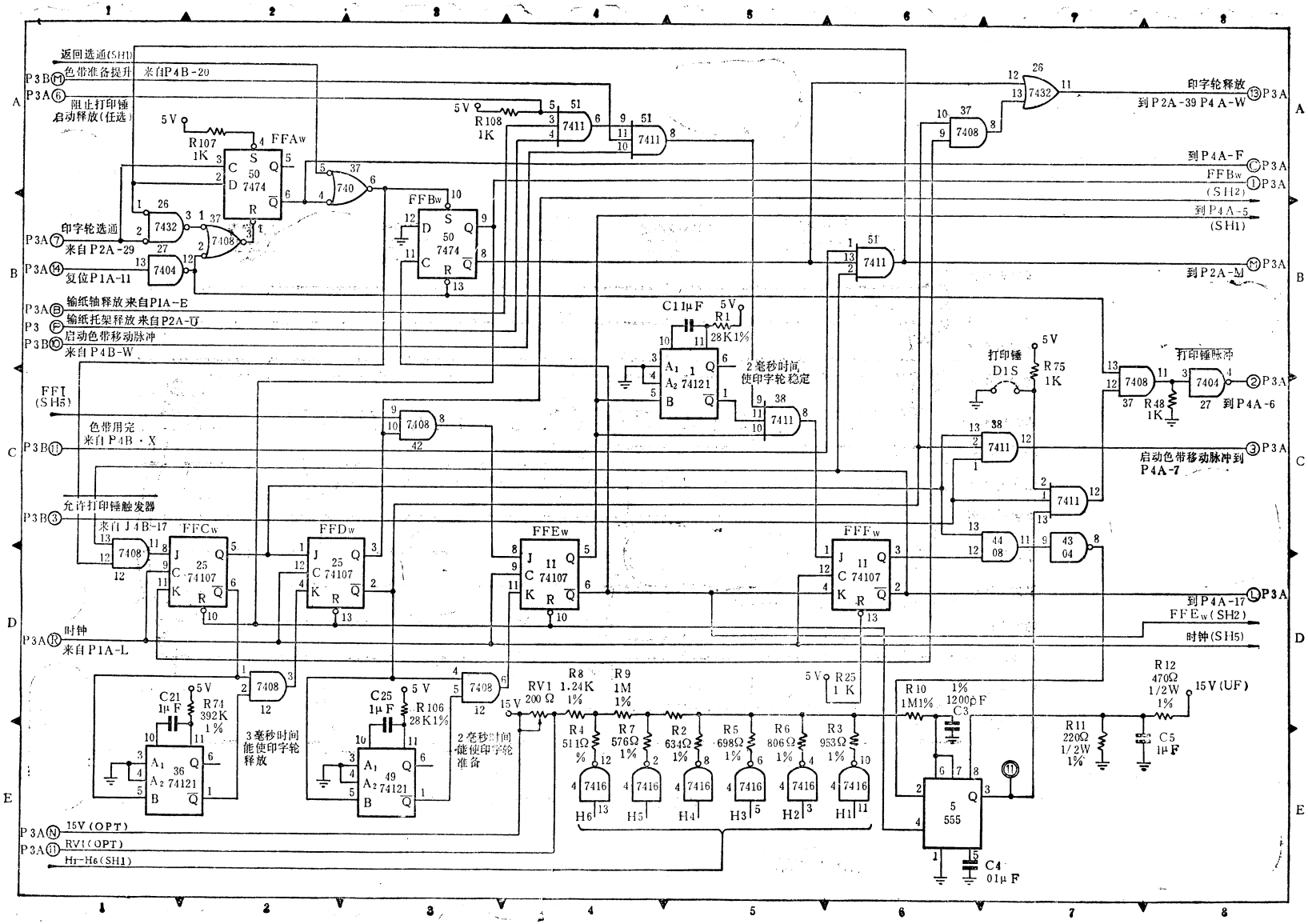



图 3-1 印字轮控制逻辑图

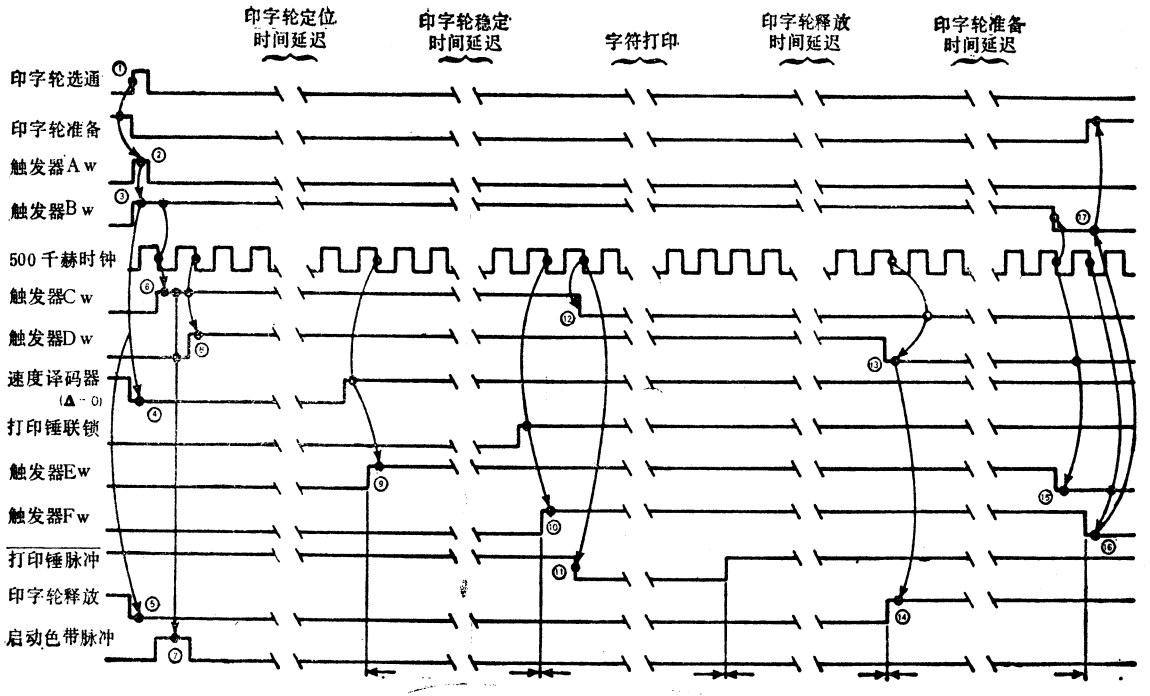
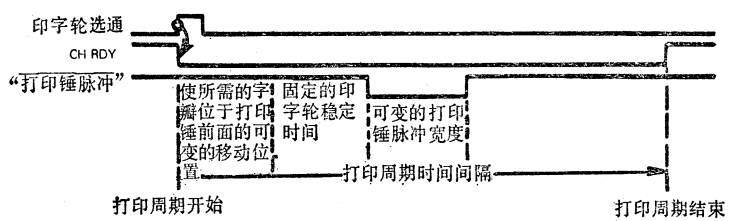



图 3-2　印字轮控制逻辑时间图

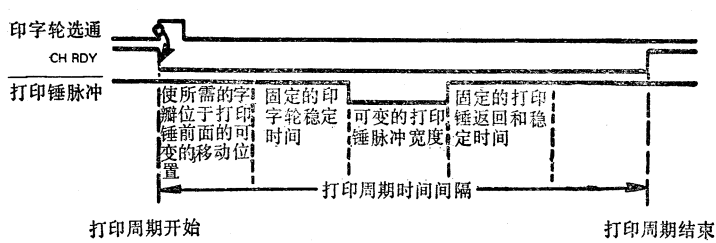
如果当打印锤击打一个面对纸页的字瓣时，印字轮仍在振动，那么将打印出模糊的字符来。

5）在印字轮稳定时间延迟结束时，打印锤方能被启动。输出一个脉冲送到线圈，从而启动打印锤。因为有些字符比其它字符有更大的表面面积，所以提供六种启动脉冲强度。击打一个象“W”那样比较大的表面面积和击打一个小的字符象“·”，如果使用同样的脉冲强度，将在打印文件上产生不均匀的密度。打印锤线圈脉冲的持续时间由下一个时间延迟控制：




打印锤脉冲上的一横，表示这个信号在工作时是低电平。

6）在打印锤脉冲时间延迟结束时，打印锤已经在纸上打印出一个字符。现在必须给以时间使打印锤恢复到准备启动的位置。为此产生3毫秒的时间延迟：

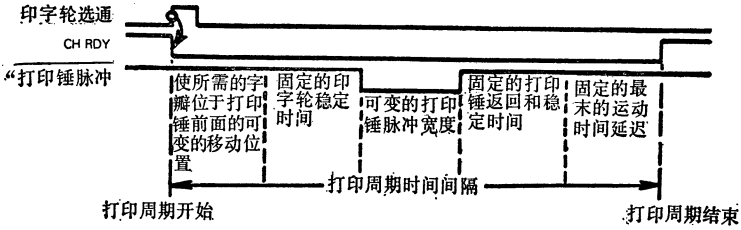



**印字轮可见位置**

7）现在印字轮能够移动到它的可见位置，而输纸托架能前进到下一个字符的位置。印字轮的“可见位置”是它的正常的静止位置；在这

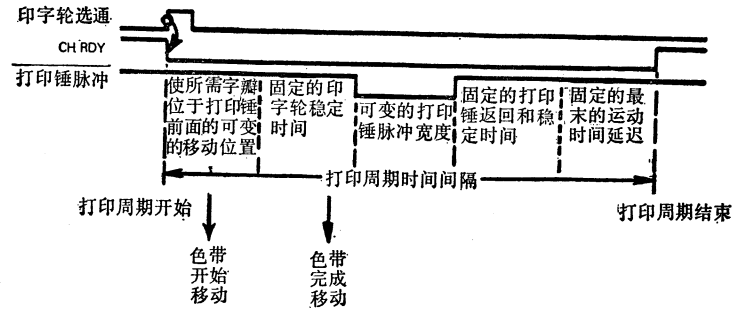
个位置上短瓣在打印字锤的前面，因此透过短瓣就能看见最新的打印字符了；所以叫做“可见位置”。在印字轮移到它的可见位置之前我们没有给打印锤返回时间，印字轮字瓣碰在仍是凸出的打印锤的顶端有可能被折弯。并在纸上留下墨迹。然而若给打印锤充分的退回时间，就不会发生上述这些问题了。

最终的2毫秒时间延迟允许印字轮和输纸托架本身回复到原来的位置：




**启动色带脉冲 FFA**

8）色带逻辑怎样？为了在纸上得到清楚的字迹，一段新的色带必处在字瓣和纸之间。因此在打印周期开始后不久，就有一个信号（启动色带移动脉冲）输出到真正控制色带运动的外部逻辑电路中去。这个外部逻辑电路（它不是图3-1的一部分）送回“色带移动完成”的信号（$\overline{\text{FFA}}$）。因为当色带仍在移动时我们不允许



打印锤启动。所以色带是在印字轮开始定位和稳定的过程中向前移动的(见70页下图)。

总之，一次打印周期由五段时间延迟组成：随着每段时间延迟的开始，都伴随出现一连串逻辑操作，然后是一段机械运动的时间。

## 3-2 输入/输出信号

现在，你已经对图 3-1 中的逻辑控制功能有了初步了解，下一步就要更仔细地观察输入和输出信号。

我们若要知道微型计算机该做什么和什么时候去做，就必须完全依据输入信号。同样，输出信号则表示我们能够把控制信息发送到外部逻辑设备的唯一手段。在这一点上，我们的目标仅限于了解每一输入和输出信号所实现的功能，以及我们如何具体去处理这些信号。我们先来讨论“如何处理”的问题。

### 3-2-1 输入/输出器件

**可编程序外部接口**

8080微型计算机系统和外部逻辑之间传送信号和数据的器件有两类。

第一类是可编程序外部接口器件 (PPI)，例如由几个厂家制造的 8255 系列的器件或是由德克萨斯仪器公司制造的 TMS 5501 器件。

**锁存缓冲器**

第二类是简单的锁存缓冲器，例如8212。

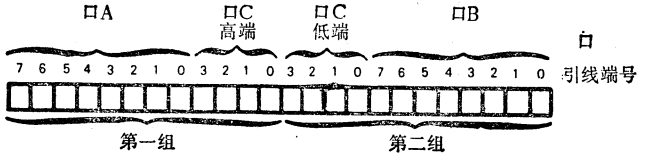
我们将要使用 8255 可编程序外部接口和 8212锁存缓冲器。

因为这些器件在“微型计算机入门”一书已经介绍过了，我们现在假定你对它们的性能和结构已有初步了解；如果不是这样，那么在继续阅读下去之前，你应先查阅微型计算机入门：

第二册——“某些具体产品”，否则你将不理解下面的内容。

### 3-2-2 8255可编程序外部接口

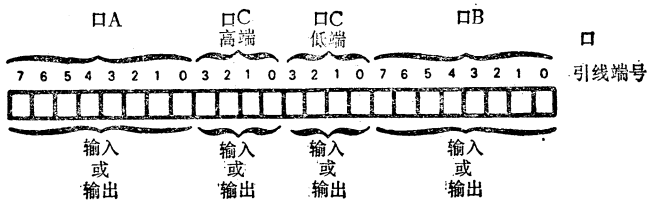
8255 可编程序外部接口提供 24 个 I/O 引线端，这些引线端可以组成如下几种口：



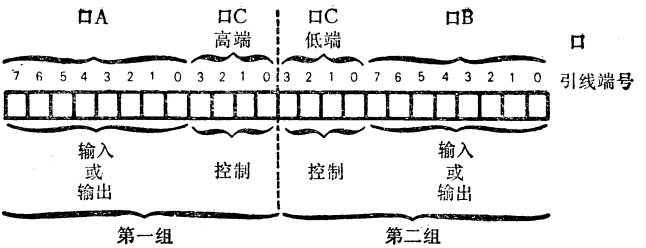

**I/O 口工作方式**

每组引线端都能够由程序控制以三种工作方式中的一种进行操作。第一组和第二组口不必以同一种工作方式操作。

当用方式 0 时，每个口或是作为简单的输入口，或是作为简单的输出口：

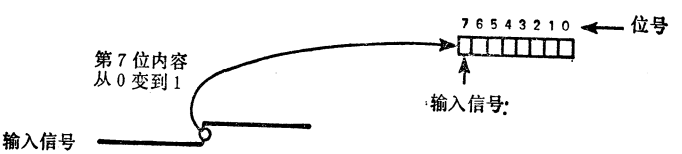



当用方式 1 时，口 A（或口 B）或是作为一个被选通的输入口，或是作为被选通的输出口。选通和控制信号由同一组里的口 C 的各引线端提供。



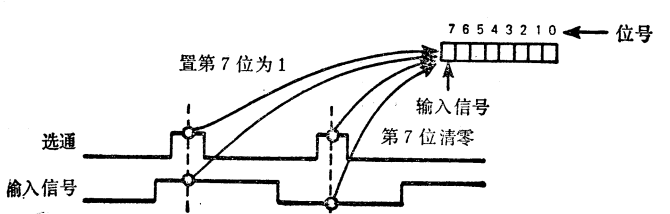

**简单I/O口**

基本(方式0)I/O口和选通(方式1)I/O口之间有什么区别呢？基本I/O口是立即接受和传送数据的。只要输入信号改变状态，就立即将I/O口缓冲器的适当位置位或复位。

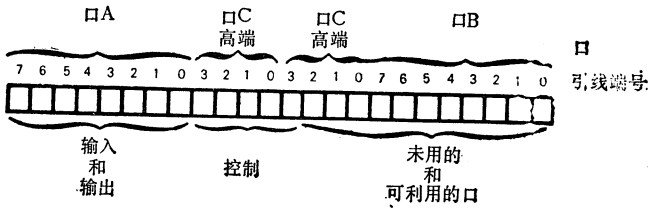



**选通I/O**

一个被选通的I/O口用控制信号来决定数据变化被记录的时刻。还是以输入信号为例，这可以用下图说明：



8255 I/O 引线端也可以被编程，作为一个双向的八位 I/O 口，并由五个控制信号所操作，如下图所示：




**I/O 口编址**

如上图所示，8255 PPI 工作于方式 2。不管是什么工作方式，每个 PPI 都被分配四个 I/O 口地址。三个8255引线端按照如下要求来选择器件和一个器件口：

$\overline{CS}$ ：输入“0”为选中器件。输入“1”为切断联系。

| $A_1$ | $A_0$ | |
|---|---|---|
| 0 | 0 | 访问口 A |
| 0 | 1 | 访问口 B |
| 1 | 0 | 访问口 C(高位部分和低位部分合起来组成一个八位单元) |
| 1 | 1 | 访问 PPI 器件的“控制口” |

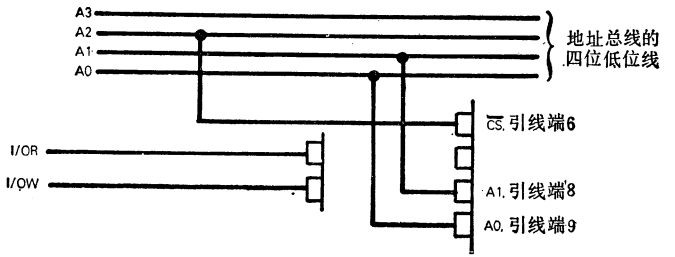
**I/O 口方式选择**

“控制口”这个器件是一个只写口。你只要将相应的代码写入控制口就能够选择口的工作方式。详细讨论控制口代码无助于你理解这一章的主题，所以我们把这个内容放在微型计算机入门：第二册——“某些具体产品”一书中讨论。

**I/O 口地址的确定**

现在，当 8080 CPU 执行 IN（输入）或 OUT(输出)指令时，口的号码从地址总线的低 8 位输出。所以我们把 8255 PPI 的接线作

如下安排：




这些引线端的接法使得 8255 I/O 口的地址编制如下：

| | $A_2$ | $A_1$ | $A_0$ |
|---|---|---|---|
| 口 0 —PPI | | | |
| 口 A | 0 | 0 | 0 |
| 口 1 —PPI | | | |
| 口 B | 0 | 0 | 1 |
| 口 2 —PPI | | | |
| 口 C | 0 | 1 | 0 |
| 口 3 —PPI | | | |
| 控制口 | 0 | 1 | 1 |

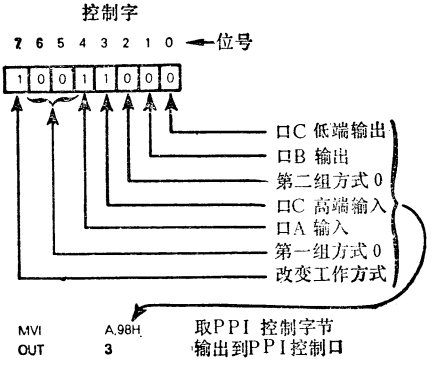
**地址总线译码**

当使用象 8212 I/O口这样的简单I/O器件时，你必须精心地对地址总线进行译码。当地址总线处于不工作的时候，8080型微处理机器件的地址总线输出诊断信息。用片选信号和适当的总线控制信号(在此情况下就是 I/OR 或 I/OW)相“与”，我们就能保证仅当 I/O 操作被启动时地址总线才被译码。

开始时，为了使事情尽可能简单些，我们将用程序使 8255 PPI 工作在方式 0，第一组口被指定为输入而第二组口被指定为输出。

**I/O 口方式选择指令序列**

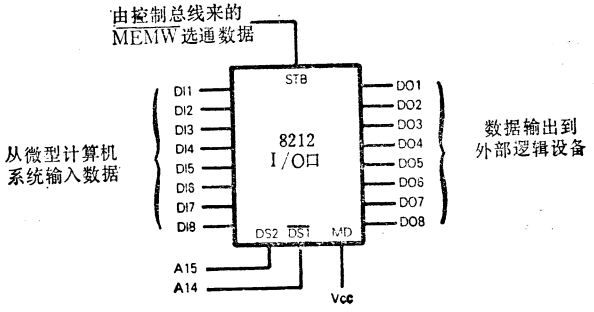
为了便于理解眼下的讨论，你并不需要知道 8255 PPI 的程序应如何编制才能符合我们的要求；然而，下面给出适当的指令序列和对于控制字的解释：




为了理解上述控制字的格式，可以查阅微型计算机入门：第二册——“某些具体产品”一书中关于 8255 PPI 的介绍。

## 3-2-3 8212 八位输入/输出口

这个器件是一种简单的，选通控制的八位输入/输出缓冲

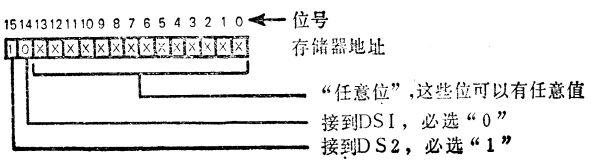


器。

8212 I/O 口有八个“数据输入”引线端和八个“数据输出”引线端。I/O 口的工作方式是由 MD 信号控制，而数据传输则是由 STB 信号选通的。我们将用 8212 I/O 口作为数据存储和输出器件，其接线如76页下图所示。

**I/O 口作为存储器编址**

只要存储器地址的两个高位是10，那么在响应存储器访问指令时我们的接线使 8212 I/O口选择它自己。换句话说，在 8000 H 到 BFFF 范围内的任何存储器地址都能选择 8212 I/O 口：




如果你注意到8212各个“任意位”所能取的数值范围，你就会得到按上述接线的8212所能响应的存储器地址的范围，它们是：

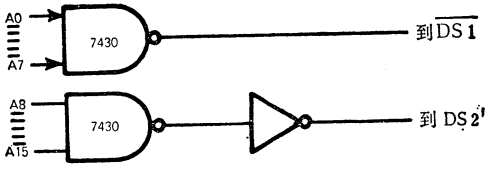
最小地址：$\underbrace{1\ 0\ 0\ 0}_{8}\ \underbrace{0\ 0\ 0\ 0}_{0}\ \underbrace{0\ 0\ 0\ 0}_{0}\ \underbrace{0\ 0\ 0\ 0}_{0}$

最大地址：$\underbrace{1\ 0\ 1\ 1}_{B}\ \underbrace{1\ 1\ 1\ 1}_{F}\ \underbrace{1\ 1\ 1\ 1}_{F}\ \underbrace{1\ 1\ 1\ 1}_{F}$

我们在访问一个 I/O 口时就已经用掉了 $3\ FFF_{16}$ 个存储器地址。有关系吗？实际上没有关系。我们还保留下 $COOO_{16}$ 个地址，这远远超过我们的需要。

我们可以为 I/O 口仅仅保留一个存储器地址，但这样意味着需要从16条地址线中综合出两个选择信号。下图说明如何用

单个地址 $FFFF_{16}$ 选择 8212 I/O 口：




7430是一个八输入“与非”门。

应当记住，7430输出必须和适当的控制信号相“与”，以保证只有当输出一个有效地址时，地址总线才被译码。

但是为什么要把财力耗费在增添不必要的逻辑电路上去呢？

### 3-2-4 输入信号

现在我们把注意力转到图 3-1 左半部的输入信号上来。我们将介绍每个信号，并分配给它一个相应的输入引线端，包括说明以最初级的水平存取信号的基本指令序列。

### 3-2-5 $\overline{\text{返回选通}}$(RETURN STROBE)

如果操作员想要看到刚刚打印完的字符，那么必定发生以下两件事情：

1）印字轮必须移动到它的可见位置。

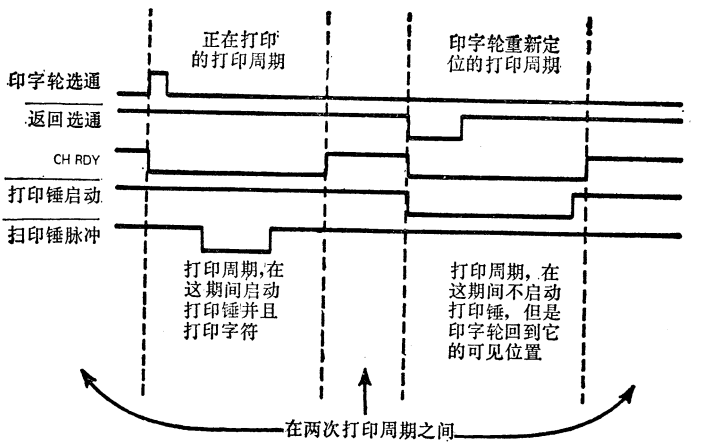
2）色带必须落下来。

外部逻辑电路能够使色带下落和提升，而图 3-1 中的逻辑电路产生允许印字轮移动的信号。

为使印字轮移到它的可见位置，当色带落下时，色带控制外部逻辑电路输入一个低电平的“$\overline{\text{返回选通}}$”信号。

**印字轮重新定位打印周期**

图 3-1 的逻辑电路用“返回选通”作为启动打印周期的另一种信号；然而，在“返回选通”为低电平的同时，“打印锤启动触发器”脉冲也是低电平，后者阻止打印锤的启动。因此，由“返回选通”低电平启动的打印周期是“哑”打印周期，这个周期仅仅使印字轮移回到它的可见位置，但是不启动打印锤，我们把这个周期叫做印字轮重新定位打印周期：



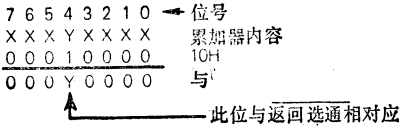

我们把 I/O 口 2 的引线端 4 分配给“返回选通”信号。

在两次打印周期之间，为了触发下一次新的打印周期，我们用下列指令序列测试这个引线端：

```
LOOP    IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     10H     ;MASK OUT ALL BAR BIT 4
        JNZ     LOOP    ;IF THIS BIT IS 1, RETURN AND RETEST
;NEW PRINT CYCLE INSTRUCTION SEQUENCE BEGINS HERE
```

```
LOOP    IN      2       把I/O口 2 的内容送入累加器
        ANI     10 H    除第 4 位外，屏蔽所有位
        JNZ     LOOP    如果这位是 1，返回并重新测试
;新的打印周期指令序列由此开始
```

下图可以说明上述指令序列中 ANI 指令是如何工作的：




### 3-2-6 阻止"打印锤启动"释放(PFL REL)

当输纸机构正在移动时，打印锤不能启动，因此这时外部逻辑设备输入的 PFL REL(阻止"打印锤启动"释放)信号是低电平。

只要输入的 PFL REL 信号为低电平，图3-1 的逻辑电路就延迟启动打印锤。

我们把 I/O口 0 的引线端 0 分配给 PFL REL。

执行启动打印锤指令之前，我们输入口 0 的内容并测试第零位。只要此位为"0"，我们就不执行打印锤启动指令序列。

下面的一些指令将实现所需要的测试：

```
LOOP    IN      0       ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     1       ;MASK OUT ALL BITS BAR BIT 0
        JZ      LOOP    ;THE BIT IS 0 DO NOT FIRE THE PRINTHAMMER
;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

```
LOOP    IN      0       把 I/O 口 0 的内容送入累加器
        ANI     1       抽出第 0 位
        JZ      LOOP    如此位为"0"，不启动打印锤
;打印锤启动指令序列由此开始
```

### 3-2-7 色带准备提升 (RIB LIFT RDY)

这个信号与 PFL REL 相似；当色带提升逻辑电路正在控制色带的移动时，此信号输入为低电平。正如当输纸机构在运

动时不启动打印锤一样，当色带正在移动时，它也不能启动打印锤。RIB LIFT RDY 被接到I/O口0的引线端1上。我们可以对“启动打印锤动作”指令序列做如下的调整：

```
LOOP    IN      0       ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     3       ;MASK OUT ALL BITS BAR 0 AND 1
        CMA             ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT PRE-
                        SENT
        JNZ     LOOP    ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS NOW 1, DO NOT
                        FIRE PRINTHAMMER
;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

```
LOOP    IN      0       把I/O口0的内容送入累加器
        ANI     3       抽出第0和第1位
        CMA             把结果取反，测试是否有“0”位
        JNZ     LOOP    现在任何“0”位都将变为“1”，如果现在
                        还有某一位是“1”，就不使打印锤动作
；打印锤动作指令序列由此开始
```

### 3-2-8 印字轮选通(PW STROBE)

我们已经见到过这个信号，它是外部逻辑电路产生的启动正常的打印周期的高电平脉冲。在此期间打印一个字符。

应当记住，返回选通是以输入低电平来启动打印周期的，在此期间，印字轮移动到它的可见位置，但不打印字符。

假定“印字轮选通”被接至I/O口2的引线端5，则两次打印周期之间执行的指令序列如下：

```
LOOP    IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     30H     ;ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
        CPI     10H     ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
        JZ      LOOP    ;IF TEST IS TRUE, STAY IN LOOP
;PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

```
LOOP    IN      2       把I/O口2的内容送入累加器
        ANI     30H     抽出第5位(印字选通)和第4位(返回选通)
```

```
CPI     10 H    测试是否"印字轮选通"=0，"返回选通"=1
JZ      LOOP    如果测试结果为"真"继续循环
```

；打印周期指令序列由此开始

可见，无论是"印字轮选通"=1，还是"$\overline{\text{返回选通}}$"=0 都能够触发打印周期开始，这就是为什么在只有"印字轮选通"=0 和只有"$\overline{\text{返回选通}}$"=1 时不能保持测试指令继续循环的原因。

**输入信号脉冲宽度**

上述四条指令执行起来总共需要 34 个时钟周期。若采用 500 毫微秒时钟的话，这四条指令能在 17 微秒内执行完毕，这是允许"印字轮选通"脉冲所具有的最小宽度。如果"印字轮选通"脉冲的高电平持续时间低于 17 微秒，我们的指令周期就可能错过这个脉冲。

## 3-2-9 "启动色带移动脉冲"($\overline{\text{FFA}}$)

这是另一个打印锤告警信号。当外部逻辑设备在推动色带时，该信号被置"0"。这个信号被接到 I/O 口 0 的引线端 2，我们可以将打印锤动作之前的指令序列修改如下：

```
LOOP    IN      0       ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     7       ;ISOLATE BITS 2, 1 AND 0
        CMA             ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        JNZ     LOOP    ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                        PRINTHAMMER
;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

```
LOOP    IN      0       把 I/O 口 0 的内容送入累加器
        ANI     7       抽出第 2、1 和 0 位
        CMA             把结果取反，测试有无含"0"的位
        JNZ     LOOP    现在任何"0"位都将变为"1"，如果有
                        一位是"1"，就不使打印锤动作
```

；打印锤动作指令序列由此开始

我们需要做的全部事情就是在执行打印锤动作指令序列之

前多加一个必须满足的测试条件。

### 3-2-10 复位(RESET)

这是在许多类型的逻辑中经常见到的一种预置信号。它的作用是保证所有的逻辑都能处于"开始"状态，在我们的情况下就是使其处于两次印字轮周期之间的状态。

在图 3-1 所示的逻辑电路中，"复位"信号接至各个逻辑器件上，以便使"复位"信号趋于高电平之后，将所有的逻辑器件强迫置入"开始"状态。

**使CPU 复位**

在微型计算机系统中，有许多方法能够处理"复位"信号。最简单的方案是把这个信号输入到 8080 CPU 的"复位"引线端。

处理"复位"的另一种方法是在两次打印周期之间测试这个信号，而且当"复位"信号为高电平时，阻止任何打印周期开始。为此，只要把"复位"信号连接到 I/O 口 2 的引线端 6 就能办到。于是可以将"在两次打印周期之间"指令序列修改如下：

```
LOOP    IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     40H     ;ISOLATE BIT 6 (RESET)
        JNZ     LOOP    ;IF RESET IS HIGH, STAY IN LOOP
;RESET IS LOW. TO TEST PW STROBE AND RETURN STROBE
        IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     30H     ;ISOLATE BIT 5 (PW STROBE) AND BIT 4 (RETURN STROBE)
        CPI     10H     ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
        JZ      LOOP    ;IF TEST IS TRUE STAY IN LOOP
;PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

```
LOOP    IN      2       把I/O 口 2 的内容送入累加器
        ANI     40H     抽出第 6 位("复位")
        JNZ     LOOP    如果"复位"是高电平，继续循环
；如果"复位"是低电平，测试"印字轮选通"和"返回选通"
        IN      2       把I/O 口 2 的内容送入累加器
```

```
ANI   30 H     抽出第 5 位("印字轮选通")和第 4 位("返回选通")
CPI   10 H     测试是否"印字轮选通"=0，"返回选通"=1
JZ    LOOP     如果测试结果为真，继续循环
```

；打印周期指令序列由此开始

**信号脉冲宽度**

现在执行这一较长的测试循环需要 61 个周期。那就意味着对于采用 500 毫微秒的时钟来说，"印字轮选通"的高电平脉冲至少要持续 30.5 微秒。

## 3-2-11 输纸轴释放(PFR REL)

这仍然是启动打印锤动作之前必须测试的另一个信号。它表示外部逻辑电路正在控制输纸机构的移动。在这样的情况下，我们不能启动打印锤。如果把这个信号接到输入口 0 的引线端 3，我们就只需对启动打印锤动作的指令序列做如下调整：

```
LOOP   IN     0      ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
       ANI    0FH    ;ISOLATE BITS 3, 2, 1 AND 0
       CMA           ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
       JNZ    LOOP   ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                     PRINTHAMMER
;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

```
LOOP   IN     0      把 I/O 口 0 的内容送入累加器
       ANI    0 FH   抽出第 3、2、1 和 0 位
       CMA           把结果取反，测试"0"位
       JNZ    LOOP   现在任何"0"位都是"1"只要有一位是
                     "1"，就不使打印锤动作
```

；打印锤动作指令序列由此开始

## 3-2-12 输纸托架释放(CA REL)

这个信号与 PFR REL 几乎相同，它来自控制托架运动的外部逻辑电路。我们将这个信号连接到输入口 0 的引线端 4，并且把"启动打印锤动作"的指令序列修改如下：

```
LOOP    IN      0       ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     1FH     ;ISOLATE BITS 4, 3, 2, 1 AND 0
        CMA             ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        JNZ     LOOP    ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                        PRINTHAMMER
PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

```
LOOP    IN      0       把 I/O 口 0 的内容送入累加器
        ANI     1 FH    抽出第 4、3、2、1 和 0 位
        CMA             把结果取反，测试"0"位
        JNZ     LOOP    现在任何"0"位将变为"1"，如果有一
                        位是"1",就不使打印锤动作
；打印锤动作指令序列由此开始
```

## 3-2-13 FFI

这个信号用来确定打印周期里的第一个延迟时间；在此期间，印字轮从它的可见位置发生移动，直到所需字瓣位于打印锤的前面为止。

FFI 由外部逻辑电路产生；当印字轮正在移动时，它是低电平；而当印字轮不再移动时，它是高电平。

**基于输入信号的时间延迟**

我们将 FFI 连接到 I/O 口 0 的引线端 7。下面的指令循环将产生一个一直继续到 FFI 变为高电平为止的时间延迟：

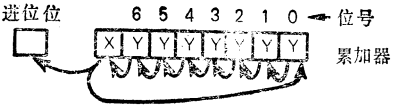
```
LOOP    IN      0       ;INPUT PORT 0 TO ACCUMULATOR
        RLC             ;SHIFT BIT 7 INTO THE CARRY
        JNC     LOOP    ;IF CARRY = 0 STAY IN THE LOOP
```

```
LOOP    IN      0       把 I/O 口 0 的内容送入累加器
        RLC             将第 7 位移入进位位
        JNC     LOOP    如果进位位= 0，继续循环
```

你能看出这个循环是如何实现的吗？在 I/O 口的内容已经进入累加器之后，我们感兴趣的只是第 7 位，因为这一位与

FFI 相当。

下图示出 RLC 指令执行的情况：




如果进位位状态等于“1”，印字轮移动时间延迟结束。如果进位位状态等于“0”，程序逻辑必须继续延迟下去。

### 3-2-14 色带用完(EOR DET)

这个信号表示已经到达色带的终端。在这种情况下不能再继续打印字符。

当产生这个信号时，仍旧有新的色带在印字轮的前面，所以这个信号不是用来禁止打印锤动作的，却是用来阻止打印周期结束被显示出来。这样就能有效地阻止一个新的打印周期的开始。

我们将 $\overline{\text{EOR DET}}$ 信号连接到 I/O 口 2 的第 7 位。因为 $\overline{\text{EOR DET}}$ 是负逻辑信号，我们在进入“在两次打印周期中间”循环之前对其测试如下：

```
;TEST FOR VALID END OF PRINT CYCLE
LOP1    IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        RLC             ;SHIFT BIT 7 INTO CARRY
        JNC     LOP1    ;IF ZERO IN CARRY, STAY IN PRINT CYCLE
;START OF IN BETWEEN PRINT CYCLES LOOP
LOOP    IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     40H     ;ISOLATE BIT 6 (RESET)
        JNZ     LOOP    ;IF RESET IS HIGH, STAY IN LOOP
;RESET IS LOW. TO TEST PW STROBE AND RETURN STROBE
        IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     30H     ;ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
        CPI     10H     ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
        JZ      LOOP    ;IF TEST IS TRUE STAY IN LOOP
;PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

```
; 测试打印周期是否有效结束
  LDP1  IN    2       把I/O口2的内容送入累加器
        RLC             将第7位移入进位位
        JNC   LOP1      如果进位位为"0"，停留在打印周期内
; 在两次打印周期之间循环的开始
  LOOP  IN    2       把I/O口2的内容送入累加器
        ANI   4OH       抽出第6位("复位")
        JNZ   LOOP      如果"复位"是高电平，继续循环
; 如果"复位"是低电平则测试"印字轮选通"和"返回选通"
        IN    2         把I/O口2的内容送入累加器
        ANI   3OH       抽出第5位("印字轮选通")和第4位("返回选通")
        CPI   1CH       测试是否"印字轮选通"=0，"返回选通"=1
        JZ    LOOP      如果测试结果为真，继续循环
; 打印周期指令序列由此开始
```

考查一下上述的指令序列，可以发现其中一些有趣的地方。

前三条指令是打印周期序列中最后的三条指令。而标号为LOOP的指令是下一打印周期开始之前一直在连续执行的那一指令序列中的第一条指令。这样。如果"$\overline{\text{EOR DET}}$"是低电平，程序逻辑将暂停在上述的前三条指令中，连续地在这三条指令中循环直到"$\overline{\text{EOR DET}}$"变成高电平为止。此时，打印周期结束而进入"在两次打印周期之间"的指令循环。而程序不确定地暂处于指令循环之中，直到与"复位"相当的第6位等于"0"，与"印字轮选通"相当的第5位等于"1"，与"返回选通"相当的第4位等于"0"时才退出循环。

上面的指令序列还有另外的有趣特点。如果我们希望的话，能够省去第二条IN指令如下：

```
;TEST FOR VALID END OF PRINT CYCLE
LOP1    IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        RLC             ;SHIFT BIT 7 INTO CARRY
        JNC     LOP1    ;IF ZERO IN CARRY, STAY IN PRINT CYCLE
;START OF IN BETWEEN PRINT CYCLES LOOP
        ANI     80H     ;ISOLATE BIT 6 (RESET)
        JNZ     LOP1    ;IF RESET IS HIGH, STAY IN LOOP
;RESET IS LOW. TO TEST PW STROBE AND RETURN-STROBE
        IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     30H     ;ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
        CPI     10H     ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
        JZ      LOP1    ;IF TEST IS TRUE STAY IN LOOP
;PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

```
; 测试打印周期是否有效结束
LOP 1   IN    2       把 I/O 口 2 的内容送入累加器
        RLC           把第 7 位移入进位位
        JNC   LOP 1   如果进位位为“0”，停留在打印周期内
; 两次打印周期之间循环开始
        ANI   80H     抽出第 6 位(“复位”)
        JNZ   LOP 1   如果“复位”是高电平，继续循环
; 如果“复位”是低电平，则测试“印字轮选通”和“返回选通”
        IN    2       把 I/O 口 2 的内容送入累加器
        ANI   30H     抽出第 5 位(“印字轮选通”)和第 4 位(“返回选通)
        CPI   10 H    测试是否“印字轮选通”= 0，“返回选通”= 1
        JZ    LOP 1   如果测试结果为真，继续循环
; 打印周期指令序列由此开始
```

去掉一条指令后，就节省了结果代码的两个字节。而付出的代价则是在整个指令循环中增加了14个时钟周期。这意味着在讨论“复位”信号时我们计算的“印字轮选通”的高电平脉冲从30.5微秒延长到37.5微秒。

**上述压缩的指令序列为什么也能正常工作呢？**理由是，假定在两次打印周期之间外部逻辑电路没有移动色带，因而在两次打印周期之间的指令执行循环期间“$\overline{\text{EOR DET}}$”总是高电平。如果是这样的话，RLC 指令总是把一位“1”移入进位位，

从而总是连续执行 ANI 指令。这样，前三条指令就没有妨碍了。应当注意，ANI 40 指令已经变为 ANI 80 指令，因为“复位”信号位由 RLC 指令左移一位。

### 3-2-15 允许打印锤触发器(HAMMER ENABLE FF)

这个信号在印字轮移动到它的可见位置之后阻止打印锤动作。请参阅前面有关返回选通信号的介绍。

我们把允许打印锤触发器连接到 I/O 口 0 的引线端 6，然后将“打印锤动作”之前的指令序列修改如下：

```
LOOP    IN      0       ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     5FH     ;ISOLATE BITS 6, 4, 3, 2, 1 AND 0
        CMA             ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        JNZ     LOOP    ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                        PRINTHAMMER
;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

```
LOOP    IN      0       把 I/O 口 0 的内容送入累加器
        ANI     5FH     抽出第 6、4、3、2、1 和 0 位
        CMA             把结果取反，测试是否有“0”位
        JNZ     LOOP    现在任何“0”位都将变为“1”，如果有一位是“1”，
                        就不使打印锤动作
;“打印锤动作”指令序列由此开始
```

### 3-2-16 时钟(CLK)

这是使图 3-1 中全部逻辑电路同步的时钟信号。虽然我们可以尝试一下，但在对图 3-1 的模拟中是不能把这个时钟信号包括进去的，因为微型计算机程序中的事件是由执行指令的顺序而不是由时钟来同步的。同样，另外两个信号，+5 V 和 RV 1，是供电用的。它们在微型计算机程序中是无意义的。

### 3-2-17 H1—H6

这六个信号从六种不同持续时间中选择出一种作为打印锤动作脉冲。我们把这些信号分配到 8212 I/O口。每当“打印锤动作”指令序列执行时，它就将这些信号送入累加器：

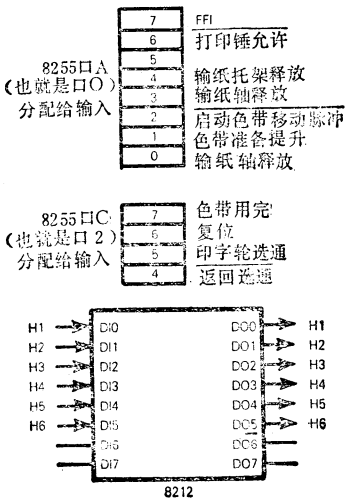
```
LDA     H1H6    ;INPUT FIRING PULSE TIME CODE
```

LDA H1H6 ；输入动作脉冲时间代码

H1H6是4字符的标号，代表I/O口所“选择”的存储器地址。

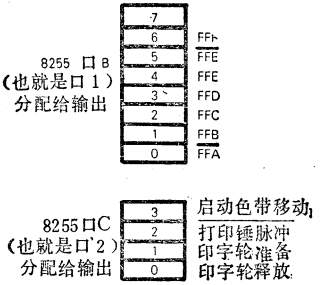
### 3-2-18 输入信号小结

总之，输入信号的分配可以用图表示如下：



### 3-2-19 输出信号

下面我们来讨论图 3-1 右侧示出的一些输出信号。介绍这些信号要比介绍输入信号容易的多。它们由六个触发器的输出所组成；它们只不过是由外部逻辑电路所使用的定时指示器而已，再加上四个控制信号。我们将把这些信号输出到 8255 可编程序外部接口的口 B 和口 C 的低位部分，如下所示：




尽管 FFC 并不是输出信号，但我们仍然为它分配一个引线端，因为 I/O 口 1 将有两种用途，作为数据存储单元和输出信号缓冲器。产生输出信号的简单例行程序是不容易混淆的；这就是图 3-1 中逻辑电路的全部目的，从而我们对于四个控制信号做如下简单定义：

1) PW REL(印字轮释放)。这个信号标志固定的打印锤返回和稳定时间延迟的结束以及固定的最后动作时间延迟的开始，在后一期间内，外部逻辑电路能操纵移动输纸机构和托架。

2) CH RDY。它也叫做“印字轮准备”信号。它规定了全部的打印周期时间间隔；在打印周期开始时，它变为低电平，一直到打印周期的结束为止。

3) HAMMER PULSE(打印锤脉冲)。这个信号在外部逻辑

电路将一个动作脉冲送入打印锤的线圈期间必须输出低电平。

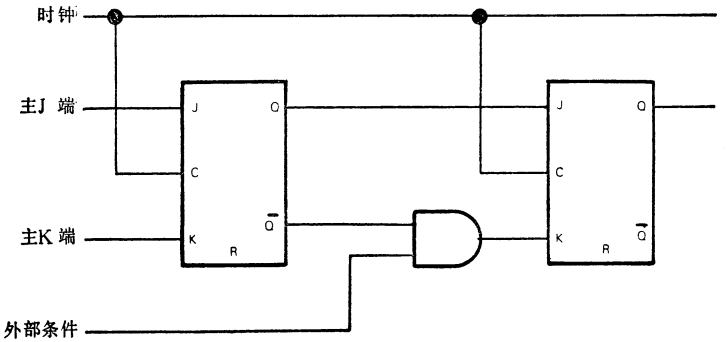
4) START RIBBON MOTION PULSE(启动色带移动脉冲)。这个信号在打印周期之前就是高电平脉冲，由它通知外部逻辑电路能可靠地开始使色带前进，以便当打印锤动作时有新的色带位于打印锤的前面。

## 3-3 面向数字逻辑电路的模拟

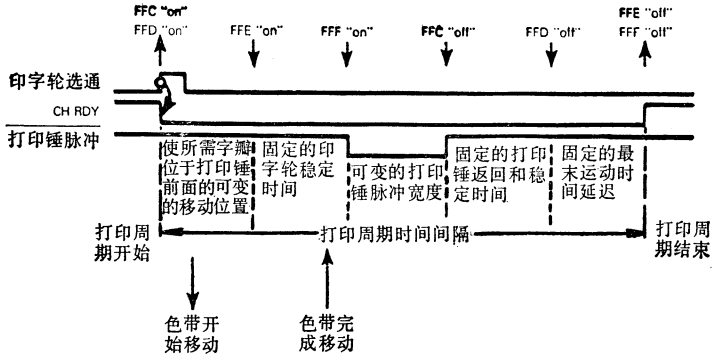
现在我们可以模拟图 3-1 所示的逻辑电路。但是，首先还要对它做一简要说明。

### 3-3-1 对于逻辑的简要说明

逻辑序列的中心是四个标号为 FFCw，FFDw，FFEw，FFFw 的 74107 触发器。你会在图 3-1 的中间和左侧看到它们。这四个触发器的形式是通常所说的“约翰逊计数器”。每个触发器受它前面触发器的输出所控制。触发器之间的耦合有一个外部条件测试端，见下图：




因此，可以设想这四个触发器是用如下的方式来启动打印周期内的各事件的：




注：图“on”置“1”，“off”置“0”

如上图所示，打印周期可以划分成五段时间间隔：

第一段时间间隔中，印字轮从它的可见位置一直移到使所需字瓣位于打印锤前面为止。这一段时间是由外部逻辑电路经由 FFI 的输入进行控制的。

其余的四段时间是由三个 74121 单冲触发器和一个 555 多谐振荡器控制的。

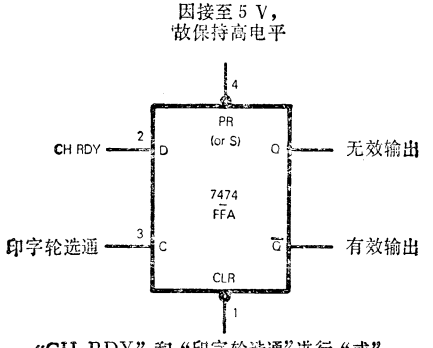
在图 3-1 左上角的两个 7474 触发器是做什么的呢？它们只不过是周期启动逻辑。触发器 FFA 是由使得打印周期能够开始所必须的信号组合来触发的。触发器 FFB作为四个 74107 触发器的开关，强制它们在两次打印周期之间置“0”。这是通过将触发器 FFB 的 Q 输出端接至 74107 触发器的复位输入端实现的。这样，只要 FFB 被置“0”，74007 触发器就必定处于复位状态。我们将在以后更详细地解释这是如何发生的。

现在我们将通过图 3-1 来考察一个打印周期，在讨论的过程中，将编写出逐个模拟各电路逻辑关系的微型计算机汇编语言程序。

### 3-3-2 触发器 FFAw

**7474 触发器**

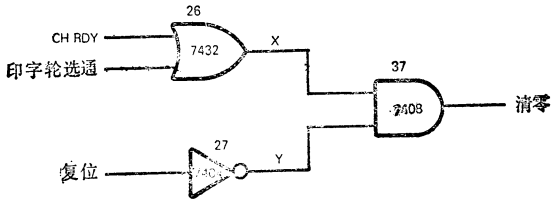
打印周期是从命名为FFAw的触发器7474开始算起的。你将在图3-1左上角上找到它。让我们单独地把FFAw画在下面：




再翻到前面参阅在第二章给出的触发器7474的一般功能表。

因为预置(PR)端被接至+5 V，总处于高电平，所以一个低电平的清零(CLR)输入将强使触发器复位为零状态。这时Q输出低电平而$\overline{Q}$输出高电平。

由图3-1你会发现"清零"(CLR)的产生如下图所示：



这是“清零”(CLR)的真值表：

| CH RDY | 印字轮选通 | X | 复位 | Y | 清零 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| | | | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| | | | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| | | | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |
| | | | 1 | 0 | 0 |

要使触发器 FFAw 置“1”，“CLR”必须是高电平；而要使“CLR”为高电平，“复位”必须是低电平，而且不是信号“CH RDY”，就是“印字轮选通”也必须是高电平。

“CH RDY”被接至触发器 FFAw 的数据（D）输入端，而“印字轮选通”接至时钟（C）输入端，所以可以得到的触发器 FFAw 的功能表如下：

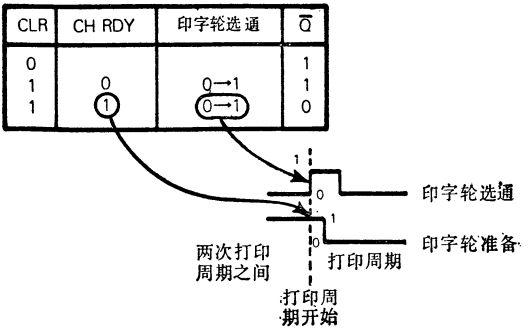
| 输入 | | | | 输出 | | |
|---|---|---|---|---|---|---|
| 预置 | CLR | 时钟<br>印字轮选通 | D<br>(CH RDY) | Q | $\bar{Q}$ | |
| 0 | 1 | 0 或 1 | 0 或 1 | 1 | 0 | 预置 = 1 |
| 1 | 0 | 0 或 1 | 0 或 1 | 0 | 1 | |
| 0 | 0 | 0 或 1 | 0 或 1 | 不稳定 | | 预置 = 1 |
| 1 | 1 | 0 → 1 | 1 | 1 | 0 | |
| 1 | 1 | 0 → 1 | 0 | 0 | 1 | |
| 1 | 1 | 0 | 0 或 1 | 原状态 | 原状态 | 不变化 |
| | | | | Q | $\bar{Q}$ | |

并可进一步简化为：

| CLR | CH RDY | 印字轮选通 | $\bar{Q}$ | |
|---|---|---|---|---|
| 0 | | | 1 | 置“0”条件 |
| 1 | 0 | 0 → 1 | 1 | 可能的置“1”条件 |
| 1 | 1 | 0 → 1 | 0 | |

要使触发器FFAw置“1”，“印字轮选通”就应从“0”变为“1”。然而，当FFAw置“1”时，如果“CH RDY”是“0”，那么$\overline{Q}$输出仍然是“1”，这表示置“0”条件。这样，要使触发器FFAw置“1”，“印字轮选通”必须从“0”变到“1”，而在此期间“CH RDY”一定是“1”。

回忆在两次打印周期之间“CH RDY”是一个输出高电平信号，而在打印周期期间是输出低电平的。这意味着只要“CH RDY”输出高电平，则在两次打印周期之间，如果“印字轮选通”脉冲为高电平，触发器 FFAw 就一定置“1”，见下图：




目前，不用担心当FFAw置“1”后不久“CH RDY”就变为“0”的情况。我们将在以后说明这一情况是怎样发生的。唯一需要着重指出的是，当“CH RDY”处于低电平时，即使“印字轮选通”是高电平脉冲也不起作用。

**复　　位**

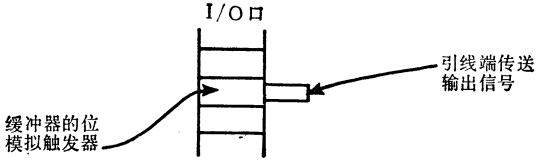
“复位”信号起什么作用呢？这个信号能使所有与触发器FFAw有关的其它逻辑电路全部无效；无论何时。只要“复位”是输入高电平，“CLR”就被强行置为低电平，从而不管正在进行着什么操作，它都使触发器FFAw置“0”。

### 3-3-3 触发器 FFAw 的模拟

在第二章中我们指出，在微型计算机系统中用一位读/写存储器来表示触发器。一个一位的读/写缓冲器也一样。

**用 I/O 口模拟触发器**

I/O 口 1 已被分配给输出信号，这个口的一些引线端被接至八位的缓冲器；这样，口缓冲器的每一位都将能模拟其输出经由该口引线端传送的那一触发器。




应该记住，FFA 已被分配给 I/O 口 1 的引线端 0。

至此，我们已为触发器 FFAw 的模拟作好了准备。

同时要问怎样模拟FFAw 左下方的三个门呢？这三个门的编号是 26，27 和 37。它们合在一起产生“CLR”输入信号。

应用以下指令序列可以分别模拟这三个门：

```
;SIMULATE GATE 27
        IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        CMP             ;COMPLEMENT ALL EIGHT BITS
        ANI     40H     ;ISOLATE BIT 6, IT REPRESENTS RESET COMPLEMENT
        MOV     B,A     ;SAVE COMPLEMENT IN REGISTER B
;SIMULATE GATE 26
        IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     22H     ;ISOLATE BITS 5 AND 1; THEY REPRESENT
        MOV     C,A     ;PW STROBE AND CH RDY. SAVE IN CPU REGISTER C
;SIMULATE GATE 37
        JZ      CLR0    ;IF BITS 1 OR 5 DO NOT EQUAL 1, CLR IS 0
        MOV     A,B     ;TEST COMPLEMENT OF RESET BY MOVING
        AND     A       ;TO ACCUMULATOR AND ANDING WITH ITSELF
        JZ      CLR0    ;IF RESULT IS 0, CLR IS 0
        MVI     D,1     ;CLR IS 1 SO STORE 1 IN D, BIT 0
        JMP     FFAW
```

```
CLR0    MVI     D,0     ;CLR IS 0 SO STORE 0 IN D, BIT 0
;SIMULATE FLIP-FLOP FFAW
FFAW    MOV     A,D     ;TEST STATUS OF CLR BY MOVING D TO
        RRC             ;ACCUMULATOR AND SHIFTING BIT 0 INTO CARRY
        JNC     FFA0    ;IF CLR IS 0, SET I/O PORT 1, BIT 0 TO 1
        MOV     A,C     ;CLR IS NOT 0. TEST PW STROBE
        ANI     20H     ;IF PW STROBE IS 0, CLOCK HAS NOT PULSED

        JZ      FFA0    ;SET BIT 0 OF I/O PORT 1 TO 1
        MOV     A,C     ;PW STROBE IS 1. TEST CH RDY
        ANI     02H     ;IF CH RDY IS 0, SET BIT 0 OF I/O PORT 1 TO 0
        JZ      FFA0
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     F7H     ;BIT 0 MUST BE RESET TO 0, SINCE FFA IS "ON"
        OUT     1
        JMP     FFB     ;JUMP TO FLIP-FLOP B SIMULATION
FFA0    IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     1       ;BIT 0 MUST BE SET TO 1 SINCE FFA IS "OFF"
        OUT     1
;FLIP-FLOP FFB SIMULATION FOLLOWS
```

门 27 的模拟

```
IN    2     把 I/O口 2 的内容送入累加器
CMP         所有八位取反
ANI   40H   抽出第 6 位；它表示"复位"的反码
MOV   B, A  把反码移入寄存器 B
```

门 26 的模拟

```
IN    2     把 I/O 口 2 的内容送入累加器
ANI   22 H  抽出第 5 位和第 1 位；它们表示
MOV   C, A  "印字轮选通"和"CH RDY"。存入 CPU 寄存器 C
```

门 37 的模拟

```
JZ    CLR0  如果第 5 位或第 1 位不等于"1"，"CLR"为"0"
MOV   A, B  移入累加器，并同它本身相"与"
AND   A     目的是测试"复位"的反码
JZ    CLRO  如果结果为"0"，"CLR"为"0"
MVI   D, 1  "CLR"若为"1"，则把"1"存入寄存器 D 的第 0 位
JMP   FFAW
CLRO  MVI D, O  "CLR"为"0"，则把"0"存入寄存器 D 的第 0 位
```

触发器 FFAw 的模拟

```
FFAW  MOV   A, D  把 D 的内容移入累加器，并将第 0 位移入
```

```
        RRC             进位位，测试"CLR"的状态
        JNC     FFAO    如果"CLR"为"0"，则置I/O口1的第0位为"1"
        MOV     A, C    "CLR"不为"0"，测试"印字轮选通"
        ANI     20H     若为"0"，则说明时钟未被启动
        JZ      FFAO    置I/O口1的第0位为"1"
        MOV     A, C    "印字轮选通"为"1"，测试"CH ROY"
        ANI     O2H     如果"CH RDY"为"0"，置I/O口1的第0位为"0"
        JZ      FFAO
        IN      1       把I/O口1送入累加器
        ANI     F7H     因为FFA置"1"，第0位必须"复位"为"0"
        OUT     1
        JMP     FFB     转到对触发器B的模拟
FFAO    IN      1       把I/O口1送入累加器
        CRI     1       因为FFA置"0"，第0位必须置为"1"
        OUT     1
```

接着对触发器FFB的模拟

了解如何适当地将各条指令组合在一起，编写为程序，这是非常重要的。你在继续阅读下去之前，必须彻底理解以上的指令序列是如何模拟FFAw以及与之有关的三个门的逻辑的。

现在我们来看上述的模拟。

**反相器模拟**

你还记得，"复位"信号已被接至8255 I/O口C的第6位；按照我们所选择的微型计算机系统内8255 PPI的接线方式，这个口被编址为I/O口2。为了使这个信号变反，我们把I/O口2的内容送入累加器，累加器的内容取反，然后把累加器的所有各位置"0"，第六位除外；从而抽出"复位"的反码：

来自I/O口2

```
IN      2       XXXXXXXX   送累加器
CMP             XXXXXXXX   取反
ANI     40H     01000000   抽出第六位
                0X000000
```

“复位”的反码暂存在CPU寄存器B中。这就完成了对于门27的模拟。

**或门模拟**

**状态标志用来表示逻辑**

对于门26的模拟就不是那么直截了当了。我们正在寻找“印字轮选通”和“CH RDY”的逻辑“或”。这两个信号分别由I/O口2的第5位和第1位来表示。现在我们要做的是把I/O口2的内容送入累加器，然后执行一条ANI指令。这条指令把除第5位和第1位外的所有各位置“0”。但实际上我们并不对其余的这两位进行逻辑“或”。为什么呢？这是因为当执行ANI指令时，它将零状态位置成“印字轮选通”以及“CH RDY”相“或”的反码了：

| A5或A1 | 累加器内容 A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | 十六进制数值 | 零状态位 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 02 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 20 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 22 | 0 |

印字轮选通

CH RDY

执行下面的ANI指令，零状态位是“印字轮选通”和“CH RDY”相“或”的反码

这样，我们再来看门37。

**零状态**

门37的用途是产生FFAw的“CLR”输入。我们打算用CPU寄存器D的第0位来模拟“CLR”。现在我们就从对门26的模拟转而模拟门37；此时如果“印字轮选通”和“CH RDY”相“或”为“1”，零状态位就是“0”；反之零状态位就是“1”。（应当记住，零状态位总是表示零状态的反面；换句话说，零状态引起零状态位被置“1”，非零

状态引起零状态位被置“0”。）

对于门37模拟的第一条指令就是利用了记录在零状态位的“印字轮选通”和“CH RDY”相“或”这一事实。如果零状态位是“1”，“CLR”必定为“0”。所以第一条JZ指令转移到使CPU寄存器D的第0位置“0”的逻辑。在对于门37进行模拟中的下一组指令测试储存在CPU寄存器B中“复位”的反码；方法是将寄存器B的内容移入累加器，然后再将累加器和它本身相“与”。“与”操作并不改变累加器的内容，但是根据“与”的结果将使各状态作相应的改变。如果“复位”的反码是“0”，那么下一条JZ指令将转移到使CPU寄存器D的第0位变为“0”的程序逻辑中去。如果“复位”的反码不是“0”，那么使得门37输出一个非零结果所应具备的条件已经具备，这个条件是由使CPU寄存器D的第0位置“1”的MVI D，1指令来模拟的。

然后模拟触发器FFA。这个触发器的状态可以定义如下：

如果“CLR”是“0”，则$\overline{Q}$是“1”。

如果“印字轮选通”是“0”，则$\overline{Q}$是“1”。

如果“CLR”是“1”，并且“印字轮选通”是“1”以及“CH RDY”是“0”，则$\overline{Q}$是“1”。

如果“CLR”是“1”，并且“印字轮选通”是“1”以及“CH RDY”是“1”，则$\overline{Q}$是“0”。

“CLR”由寄存器D的第0位来模拟。“印字轮选通”由寄存器C的第5位来模拟。“CH RDY”由寄存器C的第1位来模拟。

触发器FFA的模拟从标号为FFAw的指令开始。

**进位位状态**

我们首先将寄存器D的内容移入累加器，并且把第0位移入进位状态位来测试“CLR”的状态。因为8080的移位指令仅仅对累加器内容进行操作，所以这两步是必要的。如果进位位是“0”，那么寄存器D的第0位必

定为“0”，这就是说“CLR”是“0”；所以，转移到置I/O口1的第0位为“1”的那条指令。(应该看到我们模拟的是$\overline{Q}$，这就是为什么在置“0”条件下，我们置I/O口1的第0位为“1”，而不是把它复位为“0”)。

我们预先假定“CLR”的值为“1”，然后测试“印字轮选通”。为此我们把寄存器C的内容移入累加器，然后和一个适当的屏蔽相“与”而抽出“印字轮选通”位。如果“印字轮选通”是“0”，我们必须重新分支到把I/O口1的第0位置“1”的那条指令。

假定“印字轮选通”是“1”，那么余下的全部工作就是检查“CH RDY”的状态。为此我们重新把寄存器C的内容移入累加器，然后和一相应的屏蔽相“与”而抽出“CH RDY”，如果“CH RDY”是“0”，我们转移到把I/O口1的第0位置“1”的那条指令。

假定使触发器FFA置“1”的所有条件都已得到满足，我们就应置I/O口1的第0位为“0”。这一工作是这样来完成的，把I/O口1的内容送入累加器，和适当的屏蔽相“与”，然后把相“与”后的结果送回：

```
76543210    位号
XXXXXXXY    累加器内容
11111110    F7H
XXXXXXX0    与
```

**将某一位置“1”**

对于触发器FFA模拟的最后三条指令是把I/O口1的第0位置“1”(反映出触发器FFA被置“0”这一事实)，这三条指令将I/O口1的内容送入累加器，与适当的屏蔽相“或”，然后将结果送回原处：

```
76543210   位号
XXXXXXXY   累加器内容
00000001   1
XXXXXXX1   "或"
```

老实说，我们刚才介绍的程序序列对于模拟触发器FFA以及与它有关的三个门是一个笨拙的方法。

这是因为我们把每一个门当做一个独立的传递函数来模拟的。下面，我们把触发器以及它的三个门当做一个单独的传递函数来考虑，我们能用下面的状态定义来表示这一传递函数：

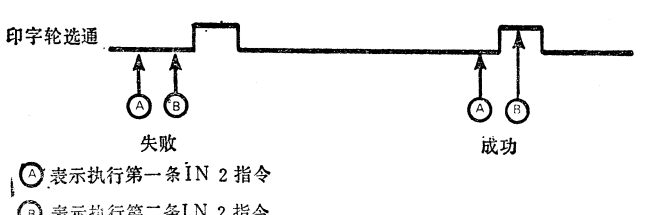
如果"复位"＝0，"CH RDY"＝1，并且"印字轮选通"从0变到1，则置$\overline{Q}$为"0"；否则，置$\overline{Q}$为"1"。

我们又如何来测试"印字轮选通"从0到1的跳变呢？

采用中断的方法进行测试是很简单的，但是我们在第五章以前都不使用中断。

**不用中断读出的信号电平变化**

如果不采用中断的话，就只有一种方法测试"印字轮选通"从0变到1的跳变。我们必须把I/O口2的内容送入累加器，抽出第5位，并保存这一结果。然后再次把I/O口2的内容送入累加器，再抽出第5位，然后比较这两次操作后的第5位，看是否旧值为"0"，新值为"1"。但是这个方案是冒险的；它只能捕捉非常幸运的发生在将I/O口2的内容送入累加器的两



条指令之间的信号的跳变：

**在微型计算机系统中事件的定时**

然而，在微型计算机程序的逻辑范围内，我们不需要依靠信号的跳变。由指令执行的顺序决定事件的顺序。在一个信号脉冲的前沿或后沿实现定时的概念是没有意义的。所以，我们用“印字轮选通”信号电平来代替这一信号的跳变。现在我们能用下面的状态定义来描述触发器 FFA 了：

如果“复位”＝0，“CH RDY”＝1，并且“印字轮选通”＝1，则置$\overline{Q}$为“0”。否则，置$\overline{Q}$为“1”。

**定时和逻辑序列**

如果你是一位逻辑设计人员，对于我们简单的用电平触发代替边沿触发的方法可能会感到十分困惑。我们在微型计算机系统中所以能够这样做，是因为与数字逻辑设计相比较编写微型计算机程序多了一个自由度：你把逻辑元件焊接到插件板时，对于所发生的逻辑事件的顺序是无能为力的，因为逻辑顺序是由边沿触发和电平触发控制的。但是你所编写的各条汇编语言指令的顺序却正是被执行指令的顺序。

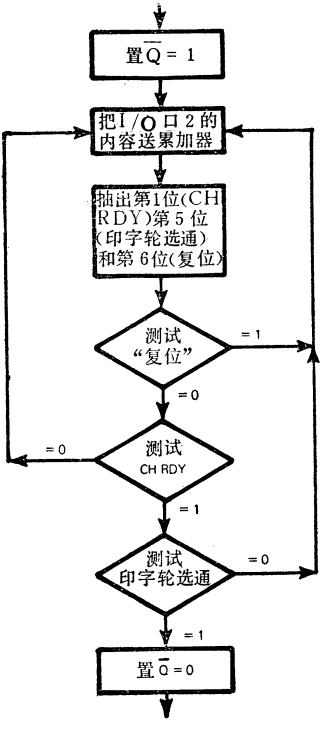
再来看下面的表示触发器 FFA 状态定义的流程图：

每个矩形框表示数据传送或数据的处理操作。

每个菱形框表示状态标志的测试条件。

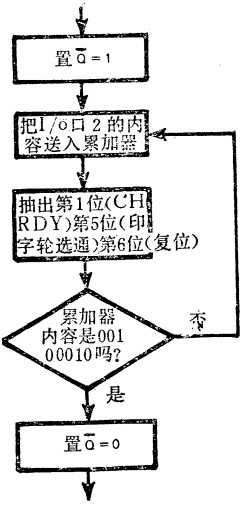
你编写的各条指令的顺序就是要执行的指令顺序。依据上面的流程图，执行顺序由带有指向下方箭头的连续线来表示。特殊的条件转移指令允许改变正常的顺序，这一改变是由菱形框的边上伸出的水平箭头表示的。你能顺着这一箭头找到条件转移指令所指引的位置。

现在我们把触发器和它的三个 CLR 逻辑门当做一个传递函数来重新模拟触发器 FFA。




因为"复位"，"CH RDY"和"印字轮选通"都是接到I/O口2的引线端，我们将I/O口2的内容送入累加器，并且抽出所有这三位。现在如果新的打印周期将要开始，那么这三位的数值组合只有一种。"复位"必定等于"0"，而"CH RDY"和"印字轮选通"都等于"1"。因此我们把程序流程图重新绘制于后：

**我们把指令序列缩减为以下的很少几条指令：**




```
;SIMULATION OF FFA AND ASSOCIATED LOGIC
        IN      1       ;INITIALLY SET BIT 0 OF I/O PORT 1 TO 1
        ORI     1
        OUT     1
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR
;AND ISOLATE BITS 1, 5 AND 6 FOR CH RDY,
;PW STROBE AND RESET, RESPECTIVELY
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     62H     ;ISOLATE BITS 6, 5 AND 1
        CPI     22H     ;IF RESET = 0, CH RDY = 1 AND
        JNZ     L10     ;PW STROBE = 1, NEW PRINT CYCLE STARTS
        IN      1       ;OTHERWISE RETURN TO L10. START NEW
        ANI     FEH     ;PRINT CYCLE BY SETTING I/O PORT 1, BIT 0 TO 0
        OUT     1
;NEW PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

对于 FFA 以及与之有关的逻辑的模拟

```
IN      1   最初把 I/O 口 1 的第 0 位置为“1”
ORI     1
OUT     1
```

把 I/O 口 2 的内容送入累加器

并且抽出对应于“CH RDY”，“印字轮选通”和“复位”的第 1 位，第 5 位和第 6

位。

```
L 10    IN    2     把 I/O 口 2 送入累加器
        ANI   62 H  抽出第 6 位，第 5 位和第 1 位
        CPI   22 H  如果“复位”= 0 、“CH RDY”= 1 并且
        JNZ   L 10  “印字轮选通”= 1 ，新的打印周期开始
        IN    1     否则返回到 L 10，开始新的循环
        ANI   FEH   将 I/O 口 1 的第 0 位置“ 0 ”,开始新的打印周期
        OUT   1
新打印周期指令序列由此开始
```

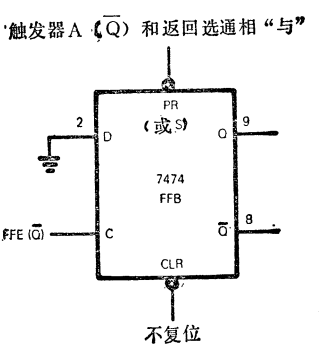
上述指令序列的前三条指令仅仅把I/O口1的第0位置“1”。这表示预期新的打印周期还未开始。以标号为L 10的那条指令开始的四条指令，都是为检验触发新的打印周期开始的一些条件所需要的指令。假定采用500毫微秒的时钟，则执行这四条指令需要34个时钟周期。这意味着“印字轮选通”的高电平脉冲至少必须持续17微秒。

若在“CH RDY”和“印字轮选通”都等于1的期间内使“复位”=0，则新的打印周期必定开始；所以最后三条指令把I/O口1的第0位置“0”。

这样，对触发器FFA的模拟就完成了。

### 3-3-4 触发器FFBw

在我们的逻辑序列中，下一个电路是在图3-1中标号为FFBw的另一个7474触发器。它刚好是在FFAw的右边。这个触发器如下图所示：



下面的功能表是按照上面的接线方式来描述FFB的，FFB的D输入端被接至“0”：

| FFA ($\overline{Q}$) | 返回选通 | 预置 | 不复位 (CLR) | FFE ($\overline{Q}$) = 时钟 | Q | $\overline{Q}$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | X | 1 | 0 |
| 0 | 1 | 0 | 0 | X | 不稳定 | |
| 1 | 0 | 0 | | | | |
| 1 | 1 | 1 | 0 | X | 0 | 1 |
| | | 1 | 1 | 0→1 | 0 | 1 |

第二章介绍了标准的7474触发器的功能表；我们所做的仅仅是除去D列和出现D＝1的行。我们也能除去CLR列，以及所有出现CLR＝0的行，因为CLR是接至“不复位”端的。在一次打印周期中“不复位”信号总是“1”，所以如果“不复位”是“0”，FFA就不能置成“1”。

现在假定CLR(“不复位”)总是“1”，并且假定D总是“0”，则下列简化的功能表就能用于FFB：

| FFA ($\overline{Q}$) ∧ $\overline{\text{返回选通}}$ = 预置 | FFE ($\overline{Q}$) =时钟 | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0或1 | 1 | 0 |
| 1 | 0→1 | 0 | 1 |

让我们考察一下FFB的预置输入，它是FFA($\overline{Q}$)和“$\overline{\text{返回选通}}$”相“与”的结果。

**印字轮重新定位打印周期**

我们知道“$\overline{\text{返回选通}}$”是一个由外部逻辑输入的信号，用于启动特殊的打印周期，这个打印周期把印字轮移回到它的可见位置，但不使打印锤动作或者打印出字符。我们称它为“印字轮重新定位”打印周期。因此，在两次打印周期之间，必须输入高电平的

“返回选通”。

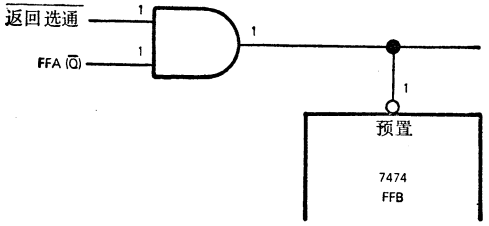
因为输入低电平的“返回选通”是启动打印周期的另一种方法，所以当我们模拟 FFB 时，就必须对“返回选通”做两种考虑：

1）作为提供“预置”输入的一个来源。

2）当做一个能够不通过触发器 FFA，而启动打印周期的信号。

首先我们规定在两次打印周期之间触发器 FFB 的状态。

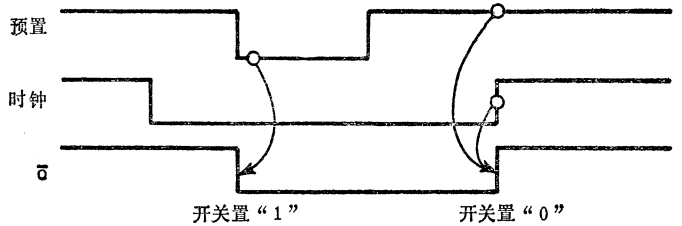
正如对触发器 FFA 的模拟中我们所看到的那样，一直到打印周期开始之前，FFA($\overline{Q}$) 输出都是高电平，这时 $\overline{Q}$ 趋于低电平。因此在两次打印周期之间 FFA($\overline{Q}$) 输出高电平。按照规定，由于以低电平的“返回选通”作为启动印字轮重新定位打印周期的信号，而在两次打印周期之间“返回选通”是高电平，所以，“FFB 预置输入”在两次打印周期之间是高电平：




因为在两次打印周期之间输入高电平的“预置”信号，我们将假定在打印周期开始时 FFB 是被置“0”的，即 Q 输出低电平而 $\overline{Q}$ 输出高电平。这也就是假定在很短的时间内，当触发器 FFE的 $\overline{Q}$ 输出从 0 变到 1 时，“预置输入”是高电平。以后你将会看到，这确实就是在每次打印周期结束时发生的情况。

因此，进入新的打印周期时，FFB 有一高电平的“预置输

入”和高电平的 $\overline{Q}$ 输出以及低电平的Q输出。现在，这个触发器相当于一个开关：由低电平的“预置”输入将触发器置“1”；接着在“预置”再次变为高电平之后，所发生的时钟从0到1的跳变把触发器置“0”：




以上叙述的将开关“置1”的操作是在以下两种情况下发生的：

1）紧接在新的打印周期开始之后，当FFA的输出 $\overline{Q}$ 为低电平时，将强制“预置”变为低电平。

2）当输入低电平的“返回选通”时，即标识出处于印字轮重新定位打印周期。

当FFE（$\overline{Q}$）输出从低电平变为高电平时，而输入高电平的“预置”时；开关被置“0”，这发生在每次打印周期的结束时。

### 3-3-5 触发器FFB的模拟

使开关置“1”

I/O口1的第1位已经分配给触发器FFB的Q输出。因此上述开关的置“1”由下面三条指令来模拟：

```
IN      .1      ;LOAD FLIP-FLOP DATA BYTE
ANI     FDH     ;RESET BIT 1 TO 0
OUT     .1      ;RESTORE FLIP-FLOP DATA BYTE
```

```
IN    1      取触发器的数据字节送入累加器
ANI   FDH    第一位清零
OUT   1      恢复触发器字节
```

ANI 指令实现的操作如下：

```
76543210   位号
XXXXXXYX   累加器内容
11111101   FDH
XXXXXX0X   "与"
```

使开关置"0"

接着模拟开关的置"0"如下：

```
IN     1     LOAD FLIP-FLOP DATA BYTE
ORI    2     SET BIT 1 TO 1
OUT    1     :RESTORE FLIP-FLOP DATA BYTE
```

```
IN    1 取触发器字节送累加器
ORI   2 第一位置"1"
OUT   1 恢复触发器字节
```

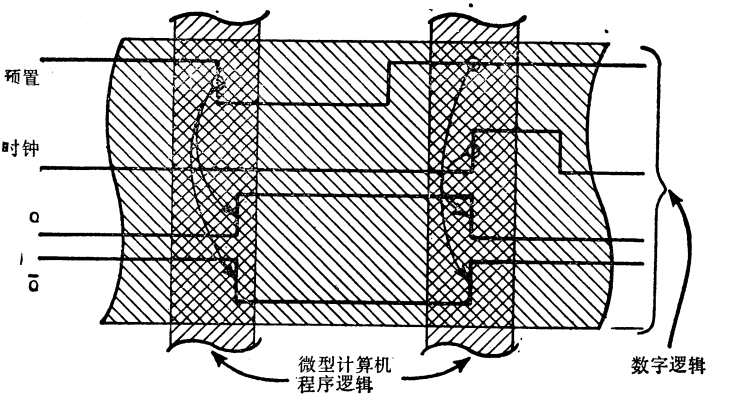
ORI 指令实现的操作如下：

```
76543210   位号
XXXXXXYX   累加器内容
00000010   2
XXXXXX1X   或
```

现在我们遇到这样一种情况，就是经各方面周密考虑之后，也不能直接模拟出我们的数字逻辑。

在逻辑图中绘出一个7474触发器并将其引线端接至适当的信号是很容易的。做到此点之后，你不必担心该信号何时改变或不改变状态。然而遗憾的是，汇编语言指令序列可没有引线

端，也没有信号；汇编语言只能模拟在一瞬间发生的一些事件。对于触发器FFB，这可以用图说明如下：




触发器FFA置“1”后会立即引进一个新的打印周期，FFA输出$\overline{Q}$为低电平时，使触发器FFB置“1”，而当FFE输出$\overline{Q}$为高电平时，一直到打印周期的较后部分FFB将不被置“0”。因此我们应将对于FFB的模拟划分为两部分：

1）在程序的开始部分，我们将模拟FFB开关被置“1”，虽然从时间顺序来说，它是在打印周期内发生的下一个事件。

2）此后，在程序内当我们模拟FFE置$\overline{Q}$端为高电平时，我们必须记住模拟FFB的开关被置“0”。

但这并不是对于FFB的全部的模拟。我们还必须修改在两次打印周期之间执行的指令序列，以便通过启动一个“印字轮重新定位”打印周期，能够模拟输入低电平的“返回选通”信号。

将经过修改的或新增加的指令用网线覆盖，我们得到的程序如下：

```
;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT 1 BIT 1 TO 0, BIT 0 TO 1.
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     1       ;SET BIT 0
        ANI     FDH     ;RESET BIT 1
        OUT     1       ;RETURN RESULT
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
;SIMULATION OF FFA AND ASSOCIATED LOGIC
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
;ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
;AND RESET, RESPECTIVELY
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     62H     ;ISOLATE BITS 6, 5 AND 1
        CPI     22H     ;IF RESET = 0, CH RDY = 1 AND PW STROBE = 1,
        JNZ     L10     ;START NEW PRINT CYCLE. OTHERWISE RETURN TO L10.
        IN      1       ;TO START NEW PRINT CYCLE, SET
        ANI     FEH     ;I/O PORT 1, BIT 0 TO 0
        OUT     1
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       RESTORE RESULT
```

```
程序在两次打印周期之间执行
最初置 I/O 口 1 的第 1 位为“0”，第 0 位为“1”
```

```
IN    1    把I/O 口 1 送入累加器
ORI   1    第 0 位置“1”
```

```
ANI   FDH  第 1 位清零
```

```
OUT   1    送回结果
```

```
测试“返回选通”是否为低电平
L10   IN    2     把 I/O 口 2 送入累加器
      ANI   10H   抽出“返回选通”
      JZ    FFB   如果它是“0”，转移到 FFB 模拟
```

对 FFA 和与之有关逻辑的模拟

I/O 口 2 的内容送入累加器，并且抽出分别对应于“CH RDY”，“印字轮选通”和“复位”的第 1 位，第 5 位和第 6 位

```
IN    2     把 I/O 口 2 送入累加器
ANI   62 H  抽出第 6 位，第 5 位和第 1 位
CPI   22 H  如果“复位”=0，“CH RDY”= 1 和“印字轮选通” =1
JNZ   L 10  新打印周期开始，否则返回到 L 10
IN    1     新的打印周期开始
ANI   FEH   置 I/O 口 1 的第 0 位为“ 0 ”
OUT   1
```

新的打印周期序列由此开始

```
模拟触发器 FFB 置“ 1 ”
FFB  IN    1     把 I/O 口 1 送入累加器
     ANI   FDH   将第 1 位清零
     OUT   1     恢复结果
```

我们还没有完全结束对于触发器 FFB 的模拟。注意到FFB的$\overline{Q}$输出接至：

1）大概约在 $B_6$ 坐标位置的 7411 “与”门。

2）在 $A_7$ 坐标位置的 7432 “或”门。

FFB(Q)输出同样也不是悬空的，我们将在稍后再去考虑它。

首先考虑位于 $B_6$ 坐标位置上的 7411 “与”门。

如果你查阅一下前面对输入信号的叙述，你会看到“CH RDY”在两次打印周期之间是高电平，而在打印周期期间是低电平。

事实上，“CH RDY”是由位于 $B_6$ 坐标 7411“与”门输出的，因此，在两次打印周期之间，加在这个“与”门的所有三个输入都必须是高电平。我们对触发器 FFB 的分析表明，它的 $\overline{Q}$ 输出在两次打印周期之间确实是高电平，但目前你必须证实

在两次打印周期之间输入到这个“与”门的其它两个信号也是高电平。

总之，触发器 FFB 被置“1”，它的$\overline{Q}$输出就变为低电平，这意味着无论加至 7411 “与” 门的其它两个输入是什么信号，“CH RDY”都将被驱入低电平。这个“CH RDY”的状态变化在我们的程序中加上下列两条指令即可加以模拟：

```
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
;SIMULATION OF FFA AND ASSOCIATED LOGIC
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
;ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
;AND RESET, RESPECTIVELY
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     62H     ;ISOLATE BITS 6, 5 AND 1
        CPI     22H     ;IF RESET = 0, CH RDY = 1 AND PW STROBE = 1,
        JNZ     L10     ;START NEW PRINT CYCLE. OTHERWISE RETURN TO L10
        IN      1       ;TO START NEW PRINT CYCLE, SET
        ANI     FEH     ;I/O PORT 1, BIT 0 TO 0
        OUT     1
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     2       ;RESTORE RESULT
```

测试“返回选通”是否为低电平

```
L10  IN    2     把 I/O 口 2 送入累加器
     ANI   10H   抽出“返回选通”
     JZ    FFB   如果它是“0”，转移到 FFB 模拟
```

对于 FFA 和与之有关的逻辑的模拟

把 I/O 口 2 的内容送入累加器，并且

抽出分别对应于“CH RDY”，“印字轮选通”和“复位”的第 1 位，第 5 位和第 6 位

```
IN    2     把I/O 口 2 送入累加器
ANI   62H   抽出第 6 位，第 5 位和第 1 位
```

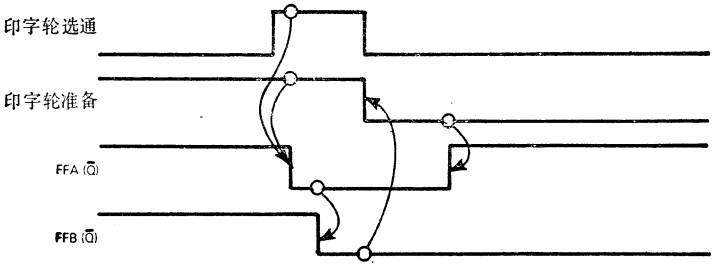
```
        CPI    22 H  如果"复位"=0，"CH RDY"=1，"印字轮选通"=1
        JNZ    L 10  新的打印周期开始，否则返回到 L 10
        IN     1     开始新的打印周期
        ANI    FEH   置 I/O 口 1 的第 0 位为"0"
        OUT    1
新的打印周期序列由此开始
模拟触发器 FFB 被置"1"
    FFB IN     1     把 I/O 口 1 送入累加器
        ANI    FDH   第一位清零
        OUT    1     恢复结果
```

```
模拟 7411 "与"门使"CH RDY"变为低电平
    IN     2     把 I/O 口 2 送入累加器
    ANI    FDH   将第 1 位清零
    OUT    2     恢复结果
```

现在我们面临一个有趣的问题。"CH RDY"变为触发器 FFA 的 D 输入，而且对于 FFA 的 CLR 输入也有贡献。那么当"CH RDY"响应 FFB 置"1"而变成低电平时将发生什么情况呢？

应当注意，现在只有"印字轮选通"是高电平的。因此为了对下面的"与"门提供高电平输入，位于 $B_2$ 坐标位置的"或"门就要依靠"CH RDY"的高电平。这个"与"门反过来又为触发器 FFA 提供高电平的"CLR"输入。换句话说，当触发器 FFB 置"1"以及"CH RDY"成为低电平时，"印字轮选通"就已经变成低电平了，因此输入的"印字轮选通"和"CH RDY"二者都是低电平。如果你翻回前面查看一下触发器 FFA 的"CLR"真值表，你就会发现当"CH RDY"和"印字轮选通"二者都是"0"时，"CLR"总是"0"。

所以触发器 FFA 将被置"0"，见下图：




这意味着什么呢？我们的结论是在打印周期开始时，触发器 FFA 转入置位状态，但是只保持到触发器 FFB 被置“1”为止。当 FFB 置“1”时，它将“CH RDY”置为低电平，而这又使触发器 FFA 被置“0”。

**定时和逻辑序列**

但是这里有一个难办的地方：如果你再看一下图 3-1 的话，你会发现触发器 FFA 除了使触发器 FFB 置“1”外，还协助产生触发器 FFC 的 J 端输入信号。

现在既然从时间上来说是串行的，我们就能够先模拟触发器 FFA 被置“0”，只要我们记住，当模拟触发器 FFC 时，它从触发器 FFA 接收到低电平的$\overline{Q}$。如果记住这一点。就能够将程序扩展如下页：

现在再来看 $A_7$ 坐标上的“或”门。为了产生“印字轮释放”脉冲，这个“或”门接收 FFB 的$\overline{Q}$输出作为它的一个输入信号。这个“或”门的另一输入是触发器 FFF 的 Q 输出和触发器 FFD 的$\overline{Q}$输出脉冲相“与”。很快你就会发现，这些触发器在两次打印周期之间也是被置“0”的，在打印周期进程期间它们依次被置“1”。在 FFB 被置“1”的瞬间，FFF 被置“0”，这意味着它的 Q 输出为低电平，从而位于 $A_6$ 坐标的“与”门将输出低电平，这就意味着“或”门 26 为了使输出“印字轮释放”脉冲为 （接119页）

```
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
;SIMULATION OF FFA AND ASSOCIATED LOGIC
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
;ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
;AND RESET, RESPECTIVELY
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     62H     ;ISOLATE BITS 6, 5 AND 1
        CPI     22H     ;IF RESET = 0, CH RDY = 1 AND PW STROBE = 1,
        JNZ     L10     ;START NEW PRINT CYCLE. OTHERWISE RETURN TO L10
        IN      1       ;TO START NEW PRINT CYCLE, SET
        ANI     FEH     ;I/O PORT 1, BIT 0 TO 0
        OUT     1
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF, SET BIT 0 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     1       ;SET BIT 0 TO 1
        OUT     1       ;RESTORE RESULT
```

测试“返回选通”是否为低电平

```
L10  IN   2     把 I/O 口 2 送入累加器
     ANI  10H   抽出“返回选通”
     JZ   FFB   如果它是“0”,转移到对 FFB 模拟
```

模拟 FFA 和与之有关逻辑

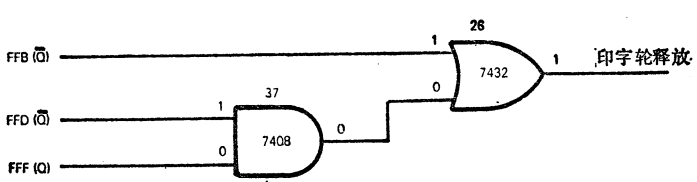
把 I/O 口 2 的内容送入累加器，并且抽出分别与“CH RDY”，“印字轮选通”和“复位”相对应的第 1 位，第 5 位和第 6 位

```
     IN   2     把 I/O 口 2 内容送入累加器
     ANI  62H   抽出第 6 位，第 5 位和第 1 位
     CPI  22H   如果“复位”=0，“CH RDY”=1 和“印字轮选通”=1
     JNZ  L10   新的打印周期开始，否则返回 L10
     IN   1     开始新的打印周期
     ANI  FEH   将 I/O 口 1 的第 0 位清零
```

```
        OUT   1
新的打印周期序列由此开始
模拟触发器 FFB 置“1”
    FFB  IN    1      把 I/O 口 1 送入累加器
         ANI   FDH    第 1 位清零
         OUT   1      恢复结果
模拟 7411“与”门使“CH RDY”变为低电平
         IN    2      把 I/O 口 2 送入累加器
         ANI   FDH    第 1 位清零
         OUT   2      恢复结果
```

```
“CH RDY”的低电平使触发器 FFA 置“0”，I/O 口 1 的第 0 位置“1”
    IN    1  把 I/O 口 1 送入累加器
    ORI   1  第 0 位置“1”
    OUT   1  恢复结果
```

（承117页）高电平，就必须依靠 FFB 的$\overline{Q}$输出高电平，见下图：




现在，当 FFB 被置“1”而输出端$\overline{Q}$为低电平时，“印字轮释放”也输出低电平。因此我们必须修改程序使 I/O 口 3 的第 0 位和第 1 位输出低电平，因为“印字轮释放”和“CH RDY”二者都将被驱入低电平。这个程序如下：

```
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
;SIMULATION OF FFA AND ASSOCIATED LOGIC
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
;ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
;AND RESET, RESPECTIVELY
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     62H     ;ISOLATE BITS 6, 5 AND 1
        CPI     22H     ;IF RESET = 0, CH RDY = 1 AND PW STROBE = 1
        JNZ     L10     ;START NEW PRINT CYCLE. OTHERWISE RETURN TO L10
        IN      1       ;TO START NEW PRINT CYCLE, SET
        ANI     FEH     ;I/O PORT 1, BIT 0 TO 0
        OUT     1
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FCH     ;RESET BITS 0 AND 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     1       ;SET BIT 0 TO 1
        OUT     1       ;RESTORE RESULT.
```

测试“返回选通”是否为低电平

```
L10   IN    2     把 I/O 口 2 送入累加器
      ANI   10H   抽出“返回选通”
      JZ    FFB   如果它是“0”，转移到 FFB 模拟
```

模拟 FFA 和与之有关的逻辑

把 I/O 口 2 的内容送入累加器，并且抽出分别对应于 “CH RDY”，“印字轮选通” 和“复位”的第 1 位，第 5 位和第 6 位。

```
      IN    2     把 I/O 口 2 送入累加器
      ANI   62H   抽出第 6 位，第 5 位和第 1 位
      CPI   22H   如果“复位”=0,“CH RDY”=1 和“印字轮选通”=1
      JNZ   L10   新的打印周期开始，否则返回 L10
      IN    1     启动新的打印周期
```

```
        ANI   FEH   I/O口1的第0位置"0"
        OUT   1
新的打印周期序列由此开始
模拟触发器FFB置"1"
    FFB IN    1     把I/O口1送入累加器
        ANI   FDH   第1位清零
        OUT   1     恢复结果
模拟7411"与"门使"CH RDY"成为低电平
```

| 模拟7432"或"门使"印字轮释放"成为低电平 |
|---|

```
        IN    2     把I/O口2送入累加器
```

| ANI FCH 将第0位和第1位清零 |
|---|

```
        OUT   2     恢复结果
"CH RDY"低电平使FFA置"0"，将I/O口1的第0位置"1"
        IN    1     把I/O口1送入累加器
        ORI   1     第0位置"1"
        OUT   1     恢复结果
```

关于触发器FFB的Q输出我们需要做什么呢？如果观察一下这个输出就会看到，它是直接连到触发器FFC，FFD，和FFE"复位"输入端的。它也是555多谐振荡器的一个输入信号。

事实上，FFB的Q输出是一个箝位信号；当它是低电平时，它将与其相接的四个电路置"0"（或封锁）；当它是高电平时，就使这四个电路置"1"（或开启）。

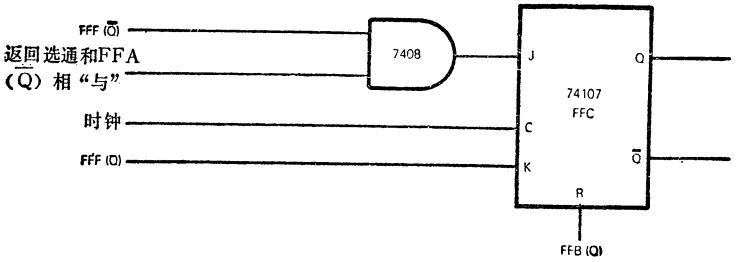
当我们模拟接到这四个信号的四个电路时，还需考虑FFB的Q的输出。这样，对于触发器FFB的模拟即告完成。

### 3-3-6 触发器FFC

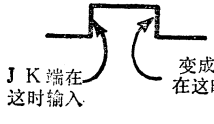
这是图3-1中坐标为$C_2$的74107触发器。因为我们要模

拟四个74107触发器，如果你不能立即回想起这种电路的特性，那么你应翻阅前面的第二章。

让我们单独取出触发器FFC，看看它是如何工作的：



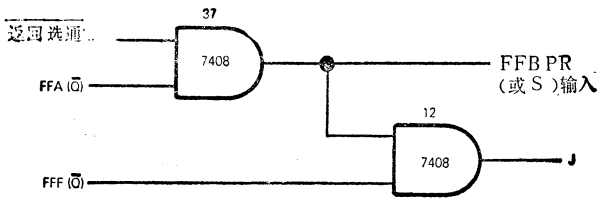

| 输入 | | | | 输出 | |
|---|---|---|---|---|---|
| R | C | J | K | Q | $\overline{Q}$ |
| L | X | X | X | L | H |
| H | ⊓ | L | L | 保持原状态 | |
| H | ⊓ | H | L | H | L |
| H | ⊓ | L | H | L | H |
| H | ⊓ | H | H | 状态翻转 | |




在两次打印周之期间，FFB的Q端输出低电平，将触发器FFC置“0”，因此，FFC的Q端输出低电平，而$\overline{Q}$端输出高电平。

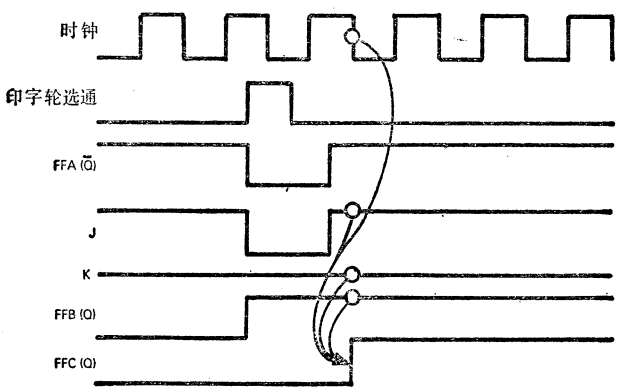
当FFB置“1”后会发生什么情况，这取决于到达FFC的J和K端的输入信号。

在两次打印周期之间，触发器FFF被置成“0”状态，因此，它的Q端输出低电平，FFC的K端输入来自FFF的Q端输出；所以，当FFC被置“1”时，它的K端输入为“0”，产生FFC J端输入的情况如下：




由于 FFF 被置“0”以后， FFF($\overline{Q}$) 端为高电平；因此，FFC 的 J 端输入将与我们已经讨论过的 FFB PR 输入相同。

将 FFC 置“1”的信号序列如下：




当 FFB Q 端输出由低电平跳向高电平时，并不将 FFC 强制置“1”， FFC 一直等待 FFA 的$\overline{Q}$端再次输出高电平。这时 FFC 将在它的 J 端接收到高电平输入，而在K端接收到低电平输入，在输入FFC 的下一个时钟脉冲的后沿，其Q端将输出高电平，而$\overline{Q}$端输出低电平。

因为当 FFA 被置“1”时，其$\overline{Q}$端输出低电平，所以，FFC 等待 FFA 的 $\overline{Q}$ 端再次输出高电平。而当 FFA ($\overline{Q}$)(或“返回选通”)跳向低电平时， FFC J 端接收低电平输入。只要

在FFC的J和K端输入均为低电平，它的输出就不改变状态，这是74107触发器的特性之一。

触发器FFC能够保持它的置"1"状态，直到这次打印周期将要结束前的某一时刻，当触发器FFF置"1"时为止。那时触发器FFC将在K端接收高电平输入，而在J端接收低电平输入，因而将引起FFC被置"0"。

### 3-3-7 触发器FFC的模拟

对触发器FFC的模拟确实很简单，它包括以下三步：

1）我们必须修改初始化指令，以保证触发器FFC能反映出它在两次打印周期之间是被置"0"的。

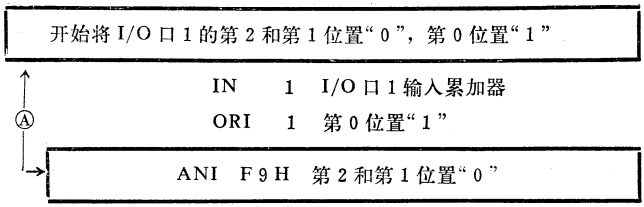
2）对于触发器FFB模拟之后，必须立即接模拟触发器FFC置"1"的指令序列。

3）我们必须记住要模拟FFC置"0"，但是这要到程序快要结束之前才进行。

现在就对程序的开头部份作如下的修改，以保证在两次打印周期之间，触发器FFC是被模拟为置"0"的。

```
;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT BITS 2 AND 1 TO 0, BIT 0 TO 1
A:      IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     1       ;SET BIT 0
        ANI     F9H     ;RESET BITS 2 AND 1
        OUT     1       ;RETURN RESULT
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
```

在两次打印周期之间程序的执行




测试返回选通脉冲是否为低电平

```
L10   IN    2     I/O口2输入累加器
      ANI   10H   抽出“返回选通”脉冲
      JZ    FFB   如果它是“0”，则转到对FFB的模拟
```

我们所做的一切只不过是修改了“与”屏敝，使I/O口的各位复位，并将第2位置“0”。

```
                    累加器内容
                    76543210 ← 位号
IN      1           XXXXXXXX
ORI     1           00000001
                    XXXXXXX1
ANI     F9H         11111001
                    XXXXX001
```

回忆一下I/O口的第2位已经分配给触发器FFC。

**定时和逻辑序列**

使触发器B和C分别置“1”的时间延迟如何呢?前已指出，在触发器FFB将FFA置“0”后，此前触发器FFC不会置“1”。如果这是一个印字轮重新定位打印周期，那么,FFC在返回选通再次输入高电平之前将不会被置“1”。

**定时问题的难易程度完全不取决于图3-1的逻辑。**在图3-1的逻辑内部，完全不涉及要求固定持续时间的时间延迟的问题，或者，为此要求有一段时间延迟将FFB和FFC的置“1”分隔开来。因此，我们将不注意考虑与FFC的置“1”有关

的定时问题，而我们将在程序的末尾，简单地加上几条模拟指令，如下所示：

```
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FCH     ;RESET BITS 0 AND 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     1       ;SET BIT 0 TO 1
        OUT     1       ;RESTORE RESULT
;SIMULATE 74107 FLIP-FLOP FFC SWITCHING ON
;SET BIT 2 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     4       ;SET BIT 2 TO 1
        OUT     1       ;RESTORE RESULT
```


新的打印周期序列由此开始

模拟触发器 FFB 置“1”

```
FFB  IN    1     I/O 口 1 送入累加器
     ANI   FDH   将第 1 位置“0”
     OUT   1     恢复结果
```

模拟 7411 “与”门，将“CH RDY”置成低电平

模拟 7432 “或”门，将“印字轮释放”置成低电平

```
     IN    2     I/O 口 2 送入累加器
     ANI   FCH   将第 0 和第 1 位置“0”
     OUT   2     恢复结果
```

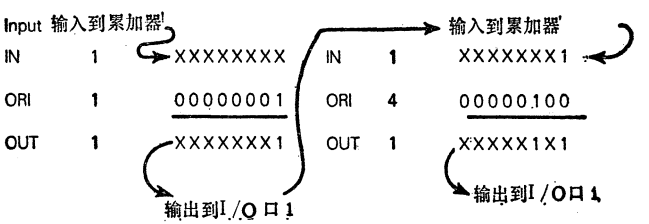
“CH RDY”低电平使 FFA 置“0”，将 I/O 口 1 的第 0 位置“1”

```
     IN    1     I/O 口 1 送入累加器
     ORI   1     将第 0 位置“1”
     OUT   1     恢复结果
```

```
模拟 74107 触发器 FFC 置"1"
将 I/O 口 1 的第 2 位置"1"
    IN    1   I/O 口 1 送入累加器
Ⓑ   ORI   4   将第 2 位置"1"
    OUT   1   恢复结果
```

**使程序缩短**

如果你开始让自己按照程序设计员的方式去思考，在模拟 FFC 置"1"的过程中，你将发现一种经济的方法。观察上面Ⓑ所指向的三条指令，仅仅是将 I/O 口 1 的一位置成"1"，这将发生如下事件的序列：




我们将以上的两步操作合并如下：

```
IN    1    XXXXXXXX
ORI   5    00000101
           XXXXX1X1
```

现在Ⓑ所指的几条指令被省略，而由Ⓒ所指的修改指令所代替：

```
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR.
        ANI     FCH     ;RESET BITS 0 AND 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
;ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     5       ;SET BITS 2 AND 0 TO 1
        OUT     1       ;RESTORE RESULT
```

模拟触发器 FFB 置“1”

```
FFB  IN   1     I/O 口 1 送入累加器
     ANI  FDH   将第 1 位置“0”
     OUT  1     恢复结果
```

模拟 7411“与”门，将“CH RDY”置成低电平

再模拟 7432 “或”门，将“印字轮释放”置成低电平

```
     IN   2     I/O 口 2 送入累加器
     ANI  FCH   将第 0 和第 1 位置“0”
     OUT  2     恢复结果
```

“CH RDY”低电平使 FFA 置“0”，将 I/O 口 1 的第 0 位置“1”

```
再模拟 FFC 置“1”，将 I/O 口 1 的第 2 位置“1”
     IN   1     I/O 口 1 送入累加器
     ORI  5     将第 2 和第 0 位置“1”
     OUI  1     恢复结果
```

现在我们完成了对触发器 C 置“1”的模拟

### 3-3-8 “启动色带移动”脉冲的模拟

回忆一下在打印周期开始时，“启动色带移动”输出信号跳向高电平，去触发推动色带的外部逻辑电路；于是，当打印锤

动作的时候，新带移到要打印的字符的前面，那个“启动色带移动”信号是由图 3-1 中位于座标 $C_7$ 上的 7411 “与”门（7 号）产生的，这个“与”门有三个输入端：

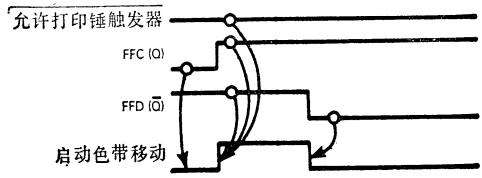
1） $\overline{\text{HAMMER ENABLE FF}}$（允许打印锤动作触发器），这是一个用来标识印字轮重新定位打印周期的输入信号。

2）触发器 FFC 的 Q 输出端。

3）触发器 FFD 的 $\overline{Q}$ 输出端

除非正在印字轮重新定位打印周期期间，否则$\overline{\text{允许打印锤动作触发器}}$将是高电平，在此期间，色带将不会移动。因此，这个信号是抑制“启动色带移动”脉冲的。

在两次打印周期之间，触发器 FFC 和 FFD 都被置“0”，因而，FFC（Q）端是低电平，而 FFD（$\overline{Q}$）端是高电平。FFC（Q）端输出保持“启动色带移动”信号为低电平。当 FFC 在一次正常的打印周期期间被置“1”时，“与”门 7 的所有输入端都是高电平，于是“启动色带移动”信号将跳向高电平，它将保持这个高电平，直到触发器 FFD 被置“1”为止。这时 FFD 的 $\overline{Q}$ 端输出低电平，而“启动色带移动”脉冲将下降为低电平。定时关系可用下图说明：




如果你观察一下图 3-2 的时间图，你将看到“启动色带移动”输出脉冲非常窄，因此，我们可以不用触发器FFD来确定，“启动色带移动”高电平脉冲的结束时间，而仅仅执行几条指令

将I/O口C的第3位置“1”，然后立即将它置“0”，如下所示：

```
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FCH     ;RESET BITS 0 AND 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
;ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     5       ;SET BITS 2 AND 0 TO 1
        OUT     1       ;RESTORE RESULT
;PULSE START RIB MOTION HIGH.
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     8       ;SET BIT 3 HIGH
        OUT     2       ;OUTPUT RESULT
        ANI     F7H     ;TURN BIT 3 OFF
        OUT     2       ;OUTPUT RESULT
```

新的打印周期序列在这里开始

模拟触发器FFB置“1”

```
FFB  IN   1     I/O口1送入累加器
          ANI   FDH   将第1位置“0”
          OUT   1     恢复结果
```

模拟7411“与”门，将“CH RDY”置成低电平

再模拟7432“或”门，将“印字轮释放”置成低电平

```
          IN    2     I/O口2送入累加器
          ANI   FCH   将第0和第1位置“0”
          OUT   2     恢复结果
```

“CH RDY”低电平使FFA置“0”，将I/O口1的第0位置“1”模拟FFC置“1”，将I/O口1的第2位置“1”

```
          IN    1     I/O口1送入累加器
          ORI   5     将第2和第0位置“1”
          OUT   1     恢复结果
```

```
"启动色带移动"脉冲置高电平
        IN    2       I/O口2送入累加器
        ORI   8       将第3位置"1"
        OUT   2       输出结果
        ANI   F7H     将第3位置"0"
        OUT   2       输出结果
```

**脉冲宽度的计算**

把I/O口2的引线端3置成高电平，然后再恢复为低电平期间，各条指令的执行时间相加，我们就能算出"启动色带移动"脉冲的宽度如下所示：

```
周期数      指令
  10        OUT    2      ;输出结果
   7        ANI    F7H    ;将第3位置"0"
  10        OUT    2      ;输出结果
```

脉冲宽度＝17个周期，若采用500毫微秒时钟则为8.5微秒

下面会发生什么情况呢？逻辑序列将把我们引向在FFC右边的触发器FFD，或者我们也可向下看，找到正好在FFC的右下方的36号单冲触发器74121。

36号单冲触发器有两个A输入端接地，这种接法使它有两个输入端都为低电平，如果你看一下第二章给出的74121的功能表，你将发现在这种结构中，单冲触发器的输出是用B点的由低向高的跳变来触发的。FFC（$\overline{Q}$）端提供这个触发脉冲，任何其它的B输入端都将使这个单冲电路处于截止状态。意味着Q端和$\overline{Q}$端分别输出低电平和高电平，直到打印周期将要结束，FFC被置"0"为止。这时，FFC的$\overline{Q}$端输出由低电平跳向高电平。

触发器 FFD 就成为需要模拟的下一种电路。

### 3-3-9 触发器 FFD

触发器 FFD 的 J 端输入直接接收 FFC(Q) 端的输出，它的K端输入接收 FFC($\overline{Q}$)端的输出。应当记住，因为 36 号单冲触发器仍然处于截止状态，它的$\overline{Q}$端将输出高电平。这就意味着“与”门 12 仅仅是允许 FFC($\overline{Q}$)端的输出直接通过，并成为 FFD(K)端的输入。

现在触发器 FFD 与 FFC 接收同样的清除和时钟信号，因此触发器 FFD 将仅仅比触发器 FFC 迟后一个时钟周期被置成“1”状态。

### 3-3-10 触发器 FFD 的模拟

触发器 FFD 的模拟与触发器 FFC 的模拟几乎相同，主要的区别是 I/O 口 1 的第 3 位被分配给触发器 FFD。这里我们仍然只是将触发器 FFD 置“1”作为前提，以保证在两次打印周期之间它的置位是正确的。

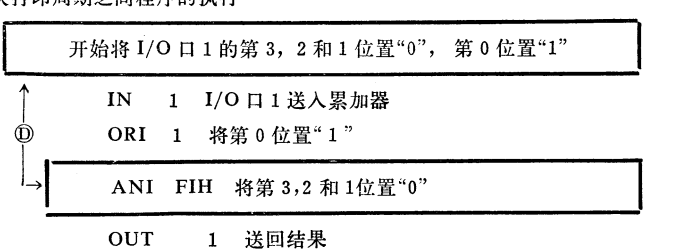
打印周期之后，触发器 FFD 被置“0”，因此我们必须记住在程序的最后将 FFD 置“0”。

下面是所需的对于程序的修改和补充：

```
;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT BITS 3, 2 AND 1 TO 0, BIT 0 TO 1
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     1       ;SET BIT 0
        ANI     F1H     ;RESET BITS 3, 2, AND 1
        OUT     1       ;RETURN RESULT
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
        .
        .
        .
```

```
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FCH     ;RESET BITS 0 AND 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1

;ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     5       ;SET BITS 2 AND 0 TO 1
        OUT     1       ;RESTORE RESULT
;PULSE START RIB MOTION HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     8       ;SET BIT 3 HIGH
        OUT     2       ;OUTPUT RESULT
        ANI     F7H     ;TURN BIT 3 OFF
        OUT     2       ;OUTPUT RESULT
;SIMULATE FFD TURNING ON. SET BIT 3 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     8       ;SET BIT 3 TO 1
        OUT     1       ;RESTORE RESULT
```

在两次打印周期之间程序的执行




测试“返回选通”脉冲是否为低电平

```
L10     IN      2       I/O口2输入累加器
        ANI     10H     分离出“返回选通”脉冲
        JZ      FFB     如果它是0，则转到对FFB的模拟
        -
```

模拟触发器 FFB 置“1”

```
FFB  IN   1    I/O 口 1 送入累加器
     ANI  FDH  将第 1 位置“0”
     OUT  1    恢复结果
```

模拟 7411“与”门，将“CH RDY”置成低电平

模拟 7432“或”门，将“印字轮释放”置成低电平

```
     IN   2    I/O 口 2 送入累加器
     ANI  FCH  将第 0 和第 1 位置“0”
     OUT  2    恢复结果
```

“CH RDY”低电平使 FFA 置“0”，将 I/O 口 1 的第 0 位置“1”；模拟FFC置“1”，将 I/O 口 1 的第 2 位置“1”

```
     IN   1    I/O 口 1 送入累加器
     ORI  5    将第 2 和第 0 位置“1”
     OUT  1    恢复结果
```

将“启动色带移动”脉冲置高电平

```
     IN   2    I/O 口 2 送入累加器
     ORI  8    将第 3 位置“1”
     OUT  2    输出结果
     ANI  F7H  将第 3 位置“0”
     OUT  2    恢复结果
```

```
模拟 FFD 置“1”，将 I/O 口 1 的第 3 位置 1
     IN   1    I/O 口 1 送入累加器
     ORI  8    将第 3 位置“1”          ←Ⓔ
     OUT  1    恢复结果
```

如果上述的对于程序的修改和补充不能一目了然的话，就把它们和模拟触发器C的程序比较一下，在你还没有理解对于触发器 FFD 的程序所做的改动之前，你就不要再继续进行下去。

**使程序缩短**

正如FFC置“1”的模拟程序Ⓑ被吸收到FFB的模拟(Ⓒ)中一样；如下图所示，FFD置“1”的模拟程序(Ⓔ)同样也能被吸收进去，如下：

```
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FCH     ;RESET BITS 0 AND 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
;ALSO SIMULATE FFC AND FFD TURNING ON. SET BITS 3 AND 2 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     0DH     ;SET BITS 3, 2 AND 0 TO 1
        OUT     1       ;RESTORE RESULT
;PULSE START RIB MOTION HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     8       ;SET BIT 3 HIGH
        OUT     2       ;OUTPUT RESULT
        ANI     F7H     ;TURN BIT 3 OFF
        OUT     2       ;OUTPUT RESULT
```

新的打印周期序列从这里开始

模拟触发器FFB置“1”

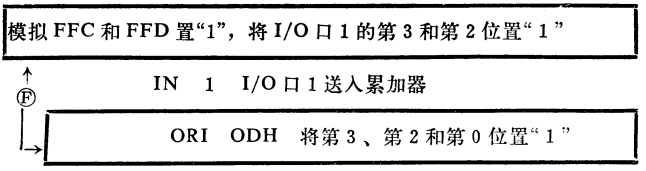
```
FFB  IN   1    I/O口1送入累加器
     ANI  FEH  将第1位置“0”
     OUT  1    恢复结果
```

模拟7411“与”门，将“CH RDY”置成低电平

模拟7432“或”门，将“印字轮释放”置成低电平

```
     IN   2    I/O口2送入累加器
     ANI  FCH  将第0和第1位置“0”
     OUT  2    恢复结果
```

“CH RDY”低电平使FFA置“0”，将I/O口1的第0位置“1”




“启动色带移动”脉冲置高电平

```
IN    2    I/O 口 2 送入累加器
ORI   8    将第 3 位置“1”
OUT   2    输出结果
ANI   F7H  将第 3 位置“0”
OUT   2    输出结果
```

如果将对触发器 FFC 和 FFD 的模拟合并起来(见Ⓕ)，则这两个触发器将正好在同一个瞬间被置成“1”。

在图 3-1 所示的逻辑中，FFD 的置“1”比 FFC 晚一个时钟脉冲，如果一个时钟周期是 2 微秒，那么触发器 FFD 的置“1”应比 FFC 的置 1 延迟 2 微秒，因而我们对二者的模拟是错误的。

**定时和模拟的限度**

这样的影响是否严重呢？坦白地说，用手头的资料还不能立即回答这个问题，因为我们不知道外部逻辑是如何使用 FFC 和 FFD 的输出。如果这两个触发器产生翻转之间的时间间隔应当很接近于 2 微秒，那么我们的模拟就行不通了。这时，或者把这两个触发器当做外部逻辑的一部份，或者必须找到另一种能模拟最后的总的功能的方法才能奏效。

如果外部逻辑要求有翻转时间的延迟，但对延迟的值无严格限制，那么我们对触发器 FFD 的模拟是合适的。

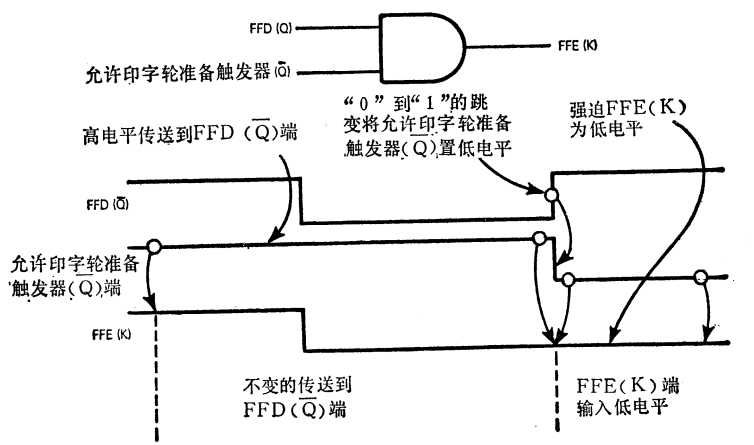
图 3-1 的逻辑所表示的触发器FFC和FFD之间的翻转时间延迟很可能仅仅是针对“启动色带移动”脉冲的前沿和后沿规定

的，但是我们已经采用相继执行的指令，使输出先为“1”，后为“0”，并送到I/O口2的第3位，而考虑到这个高电平脉冲了，既然已经考虑到图3-1的内部逻辑，也就不会对FFC和FFD发生翻转的时间延迟有所要求了。在这种情况下，我们将假定外部逻辑不要求触发器FFC和FFD的翻转有时间延迟，而且我们将采用更短的，以Ⓕ来标识的合并起来的模拟。

### 3-3-11 触发器FFE

逻辑序列中的下一个电路是FFE，这种触发器周围的电路几乎与FFD的完全相同。

FFE(K)端输入与FFD($\overline{Q}$)端输出相接，而FFD($\overline{Q}$)端是接在“与”门12的一个输入端上。这个“与”门的另一个输入端是单冲触发器49的$\overline{Q}$端输出。单冲触发器49的接线方式与我们

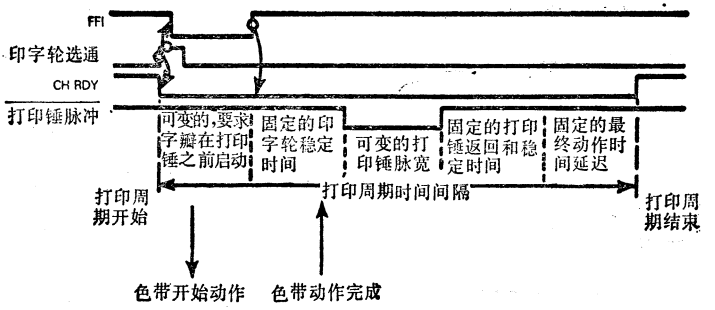



刚才已描述过的单冲触发器36相同。

当FFD被置“0”时，将会发生FFD的$\overline{Q}$端输出由0向1的跳变，而且这个跳变将触发单冲触发器49，因而，单冲触发器

49的$\overline{Q}$端将输出高电平，一直到FFD被置“0”为止。这意味着，当FFD被置“1”时，它的$\overline{Q}$端输出将直接通过这个“与”门传送过去与FFE(K)输入端相接。

触发器FFE的唯一特点是该触发器J输入产生的方式。这个输入是由FFD(Q)输出和输入信号FFI相“与“形成的，现在一旦FFD被置“1”，FFD的Q端输出就为高电平。但是从打印周期开始一直到印字轮正确定位为止，FFI是输入低电平的。(关于这个输入信号的功能，我们在第一章的前面已经介绍过)。与FFI有关的定时可以用下图说明：




只要FFI是低电平，触发器FFE就接收一个低电平的J输入。你会想到，当J和K端都为低电平时，触发器74107的Q端输出将保持原状态不变。这样，输入信号FFI就被用来产生打印周期的第一段时间延迟；这是要求印字轮的字瓣移动到打印锤的前面所需要的可变时间延迟。这种对于这个时间延迟的模拟足够简单，如下所示：

```
;PULSE START RIB MOTION HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     8       ;SET BIT 3 HIGH
        OUT     2       ;OUTPUT RESULT
        ANI     F7H     ;TURN BIT 3 OFF
        OUT     2       ;OUTPUT RESULT
;TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC    IN      0       ;INPUT I/O PORT 0 TO ACCUMULATOR
        RLC             ;SHIFT BIT 7 INTO CARRY
        JNC     VLDC    ;STAY IN LOOP IF CARRY IS 0
;AT END OF DELAY SIMULATE FFE SWITCHING ON
        IN      1       ;INPUT I/O PORT 1
        ANI     DFH     ;RESET BIT 5
        ORI     10H     ;SET BIT 4
        OUT     1       ;OUTPUT THE RESULT
```

将“启动色带移动”脉冲置高电平

```
IN    2     I/O 口 2 送入累加器
ORI   8     将第 3 位置“1”
OUT   2     输出结果
ANI   F7H   将第 3 位置“0”
OUT   2     输出结果
```

测试速度译码输入，从而确定是哪一种印字轮移动时间延迟，并产生相应的时间延迟

```
VLDC  IN    0     I/O 口 0 送入累加器
      RLC         将第 7 位移入进位位
      JNC   VLDC  如果进位位为 0，则继续循环
```

在延迟结束时，模拟 FFE 置“1”

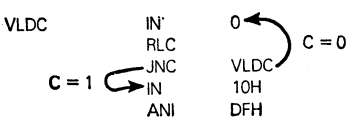
```
IN    1     输入 I/O 口 1
ANI   DFH   将第 5 位置“0”
ORI   10H   将第 4 位置“1”
OUT   1     输出结果
```

**可变长度的时间延迟**

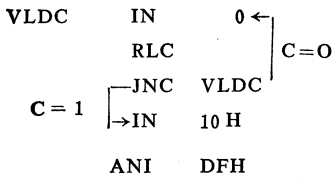
为了产生初始的时间延迟，我们可以简单地执行一段连续的循环程序。它将 I/O 口 0 的内容送入累加器中。I/O 口 0 的第 7 位已经分配给输入信号 FFI。我们可以用将它移入进位位的办法来测试

这一位的状态，如果进位位状态为“0”，FFI必然仍是低电平，我们就继续循环。一旦有一个“1”移入进位位，JNC指令就产生一个“假”的结果，顺序执行下条指令，从而跳出时间延迟程序循环，如下：

**若无进位则转移**




Jump on Not Carry means jump if Carry is 0 (Not 1). "Jump" means "do not go on to the next sequential instruction" instead go to VLDC




无进位转移指令表示若进位位是0(不是1)则转移，意味着“不顺序执行下条指令”，而转去执行标号为VLDC的指令

模拟FFE的最后四条指令表示这个触发器的两个输出都产生输出信号。这就满足了图3-1的要求。因此，我们将第5位置“0”(它表示$\overline{Q}$端输出)，而将第4位置“1”，(它表示Q端输出)。

对于在两次打印周期之间执行的指令序列需要进行修改，以保证第5位的初始状态被置为“1”，而第4位的初始状态被置为“0”，下面是需要修改的部分：

```
;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT BITS 4,3,2 AND 1 TO 0, BITS 5 AND 0 TO 1
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     21H     ;SET BITS 5 AND 0 TO 1
        ANI     0EH     ;RESET BITS 4,3,2 AND 1 TO 0
        OUT     1       ;RETURN RESULT
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
```

在两次打印周期之间程序的执行

```
开始将I/O口的第4、第3、第2和第1位置"0"，第
5和第0位置"1"
```

```
IN   1    I/O口1送入累加器
```

```
ORI  21H  将第5和第0位置"1"
ANI  E1H  将第4、3、2和1位置"0"
```

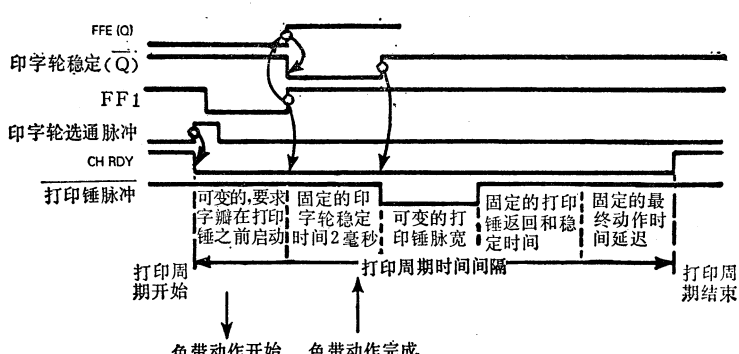
```
OUT  1    送回结果
```

测试"返回选通"脉冲是否为低电平

```
L10  IN   2     I/O口2送入累加器
     ANI  10H   抽出"返回选通"脉冲
     JZ   FFB   如果它是0，则转到对FFB的模拟
```

## 3-3-12 "稳定印字轮"单冲触发器

"稳定印字轮"单冲触发器是图3-1中座标为$B_5$的74121器件。在第二章中，我们已经介绍过这种器件。它有两个A输入端接地，而由B输入端上出现的由低到高的跳变来触发的。因为B输入端与FFE Q端相连，所以只要触发器FFE置"1"，就会产生这个跳变。



“稳定印字轮”单冲触发器具有2毫秒的时间延迟。这时间延迟由标有$C_1$和$R_1$的外部电容/电阻电路获得。因此一旦FFE被置“1”，“稳定印字轮”单冲触发器的$\overline{Q}$端就输出低电平，持续时间为2毫秒，见第141页图。

## 3-3-13 “稳定印字轮”单冲触发器的模拟

**单冲触发电路时间延迟的模拟**

模拟单冲触发器的时间延迟非常简单，可表示如下：

```
;PULSE START RIB MOTION HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULA-
                         TOR
        ORI     8       ;SET BIT 3 HIGH
        OUT     2       ;OUTPUT RESULT
        ANI     F7H     ;TURN BIT 3 OFF
        OUT     2       ;OUTPUT RESULT
;TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC    IN      0       ;INPUT I/O PORT 0 TO ACCUMULA-
                         TOR
        RLC             ;SHIFT BIT 7 INTO CARRY
        JNC     VLDC    ;STAY IN LOOP IF CARRY IS 0
;AT END OF DELAY SIMULATE FFE SWITCHING ON
        IN      1       ;INPUT I/O PORT 1
        ANI     DFH     ;RESET BIT 5
        ORI     10H     ;SET BIT 4
        OUT     1       ;OUTPUT THE RESULT
;SIMULATE 2 MS PW SETTING TIME DELAY
        MVI     A,0     ;LOAD ACCUMULATOR WITH 0
PWS     DCR     A       ;DECREMENT A
        JNZ     PWS     ;IF A DOES NOT DECREMENT TO 0,
                         RE-DECREMENT
```

将“启动色带移动”脉冲置为高电平

| | | |
|---|---|---|
| IN | 2 | I/O口2送入累加器 |
| ORI | 8 | 将第3位置“1” |
| OUT | 2 | 输出结果 |
| ANI | F7H | 将第3位置“0” |

```
        OUT   2      输出结果
```

测试速度译码输入，从而确定是哪一种印字轮移动时间延迟，并产生相应的时间延迟

```
VLDC    IN    0      I/O 口 0 送入累加器
        RLC          将第 7 位移入进位位
        JNC   VLDC   如果进位位为“0”，则继续循环
```

在时间延迟结束时，模拟 FFE 置“1”

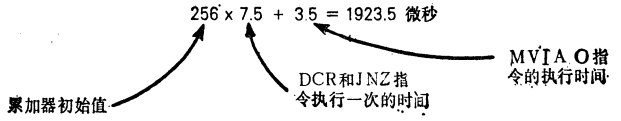
```
        IN    1      输入 I/O 口 1
        ANI   DFH    将第 5 位置“0”
        ORI   10 H   将第 4 位置“1”
        OUT   1      输出结果
```

```
模拟 2 毫秒稳定印字轮的时间延迟
        MVI   A, 0   将 0 送入累加器
PWS     DCR   A      A 内容减 1
        JNZ   PWS    若 A 内容尚未减到 0，则再减 1
```

2 毫秒的时间延迟循环说明了在“稳定印字轮”中存在一些有趣的变化。

时间延迟循环仅仅由两条指令所组成，DCR A 指令将累加器内容减 1，而 JNZ PWS 指令是如果在完成减 1 操作后，累加器内容尚未减到零，就返回去执行 DCR A 指令。执行这两条指令需要 15 个时钟周期，若采用 500 毫微秒的时钟，它共需 7.5 微秒的时间。

开始时，将 0 送入累加器，这两条指令将执行 256 次，因为累加器的第一次减“1”操作，将使其内容由 0 变为 $FF_{16}$。从而，整个时间延迟由下列等式给出：

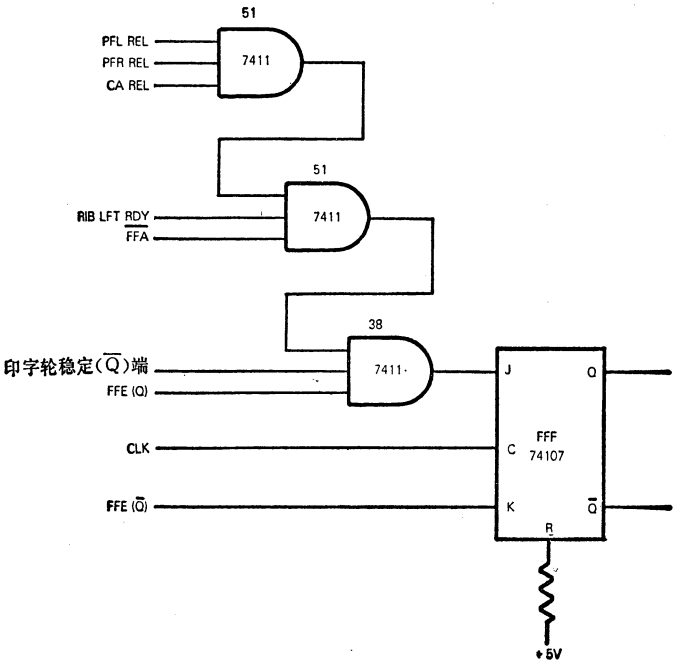



1923.5 微秒＝1.9235 毫秒

对于我们的需要而言，这已经相当接近于 2 毫秒了。

### 3-3-14 触发器FFF

一旦“稳定印字轮”单冲触发器暂停工作，我们就准备启动打印锤了。555 多谐振荡器实际上是产生“印字轮启动”脉冲的，



注：PFL　REL——阻止“打印锤启动”释放

PFR　REL——输纸轴释放

CA　REL——输纸托架释放

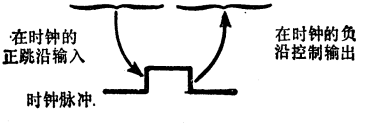
RIB　LFT　RDY——色带准备提升

CLK——时钟

但最重要的是保证当打印纸的任何部份或托架机构移动时，打印锤不会动作。因此，555 单冲触发器是由触发器 FFF 触发的，而触发器 FFF 又是由多个保护信号相“与”后的 J 端输入来置“1”的，让我们单独取出触发器 FFF，专门研究一下它的输入端，见第 144 页下图：

由于它的清除(R) 输入端与＋5V相接，触发器 FFF 具有如下的功能表。

| 输入 | | 输出 | |
|---|---|---|---|
| J | K | Q | $\overline{Q}$ |
| 0 | 0 | 不改变 | |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 取反 | |




在两次打印周期之间 FFE 被置“0”，因此 FFF 的K输入端处于高电平。因为 FFE(Q)输出低电平，而它又是FFF( J )端的一个输入端，所以触发器 FFF 的 J 输入端处于低电平。

因此，在两次打印周期之间，触发器 FFF 被置“0”，当它的 J 输入端是低电平而K输入端为高电平时，将产生稳定的输出；$Q=0$，$\overline{Q}=1$。这是触发器在置“0”状态时的特性。

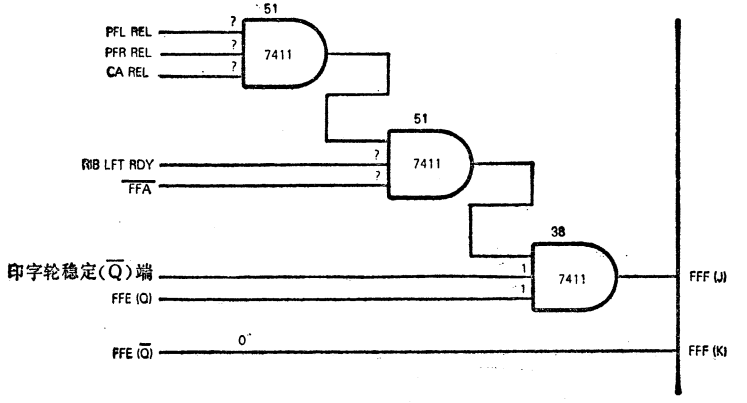
现在当 FFE 被置“1”时，它向 FFF 的K端输入低电平，只

要它的 J 输入端也是低电平，它就不会改变状态。一旦 FFF (J)端的 7 个有用信号都为高电平时，FFF 的 J 端就接收高电平，这将使触发器 FFF 被置“1”，其Q端输出高电平，而$\overline{Q}$端输出低电平。

### 3-3-15 触发器 FFF 的模拟

根据对 FFE 的模拟我们知道，为了使 FFF 置“1”，FFE (Q) 和 FFE($\overline{Q}$)具有正确的电平。

根据对于“稳定印字轮”单冲触发器的模拟我们知道，这个单冲触发器的$\overline{Q}$端必须输出高电平，见下图：

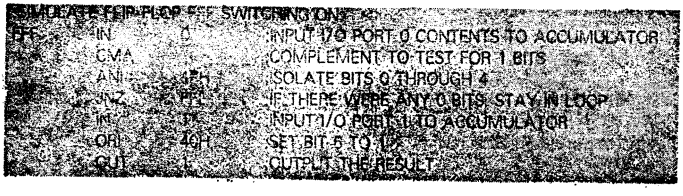



注：PFL REL——阻止“打印锤启动”释放
PFR REL——输纸轴释放
CA REL——输纸托架释放
RIB LFT RDY——色带准备提升

我们所要做的一切只是测试其余的五个联锁信号，只要它们都处于高电平，我们就模拟触发器 FFF 被置“1”。下面是所

需要的指令序列：

```
;PULSE START RIB MOTION HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     8       ;SET BIT 3 HIGH
        OUT     2       ;OUTPUT RESULT
        ANI     F7H     ;TURN BIT 3 OFF
        OUT     2       ;OUTPUT RESULT
;TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC    IN      0       ;INPUT I/O PORT 0 TO ACCUMULATOR
        RLC             ;SHIFT BIT 7 INTO CARRY
        JNC     VLDC    ;STAY IN LOOP IF CARRY IS 0
;AT END OF DELAY SIMULATE FFE SWITCHING ON
        IN      1       ;INPUT I/O PORT 1
        ANI     DFH     ;RESET BIT 5
        ORI     10H     ;SET BIT 4
        OUT     1       OUTPUT THE RESULT
;SIMULATE 2 MS PW SETTLING TIME DELAY
        MVI     A,0     ;LOAD ACCUMULATOR WITH 0
PWS     DCR     A       ;DECREMENT A
        JNZ     PWS     ;IF A DOES NOT DECREMENT TO 0,
                        ;RE-DECREMENT
```




将“启动色带移动”脉冲置高电平

```
IN    2     I/O 口 2 送入累加器
ORI   8     将第 3 位置“1”
OUT   2     输出结果
ANI   F7H   将第 3 位置“0”
OUT   2     输出结果
```

测试速度译码输入，从而确定是哪一种印字轮移动时间延迟，并产生相应的时间延迟。

```
VLDC  IN    0       I/O 口 0 送入累加器
      RLC           将第 7 位移入进位位
      INC   VLDC    若进位位是“0”,则继续循环
```

时间延迟结束时，模拟 FFE 置“1”

```
        IN      1       输入 I/O 口 1
        ANI     DFH     将第 5 位置“0”
        ORI     10 H    将第 4 位置“1”
        OUT     1       输出结果
```

模拟 2 毫秒稳定印字轮时间延迟

```
        MUI     A, O    将零送入累加器
PWS     DCR     A       A内容减 1
        JNZ     PWS     若A内容尚未减到零；则再减 1
```

```
模拟触发器 FFF 置“1”
FFF     IN      0       I/O 口 0 送入累加器
CMA                     为了测试各“1”位而取反
        ANI     1 FH    抽出第 0 至第 4 位
        JNZ     FFF     如果有任何位为“0”,则继续循环
        IN      1       I/O 口 1 送入累加器
        ORI     40H     将第 6 位置“1”
        OUT     1       输出结果
```

到现在为止，你应该能够理解程序中增加的一些指令了。

前面的四条指令仅仅是将 I/O 口 0 的内容寄存起来，并且测试其低端的五位中有哪几位是“1”。一直到所有这 5 位都是“1”为止，程序都将在这四条指令中循环。这个循环从 IN 0 指令开始，到 JNZ FFF 指令结束。

当第 0 至第 4 位都等于“1”时，CMA指令将所有这几位都变成“0”。

```
                        累加器内容
FFF     IN      0       × × × 1 1 1 1 1
        CMA             ‾ ‾ ‾ 0 0 0 0 0
        ANI     1 FH    0 0 0 1 1 1 1 1   零标志＝1
                        0 0 0 0 0 0 0 0
        JNZ     FFF     如果零标志＝0，则返回 FFF
```

IN　　1　　如果零标志=1，则从这里继续做下去

JNZ 指令使程序不再返回到 FFF 去执行，更确切地说，是允许顺序执行下条指令。

后面的三条指令模拟触发器 FFF 被置“1”。I/O 口 1 的第 6 位被分配给 FFF，因此这一位被置成“1”。

我们能够对指令序列作最后的修改，使之在两次打印周期之间，能正确地置触发器的状态。程序的结尾是：

```
;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT BITS 6, 4, 3, 2 AND 1 TO 0, 5 AND 0 TO 1
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     21H     ;SET BITS 5 AND 0 TO 1
        ANI     A1H     ;RESET BITS 6, 4, 3, 2 AND 1 TO 0
        OUT     1       ;RETURN RESULTS
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
```

在两个打印周期之间程序的执行

开始将I/O口的第 6、第 4、第 3、第 2 和第 1 位置“0”，将第 5 和第 1 位置“1”

```
        IN      1       I/O 口 1 送入累加器
```

```
        ORI     21 H    将第 5 和第 0 位置“1”
        ANI     A 1 H   将第 6、第 4、第 3、 第 2 和第 1 位置“0”，
```

```
        OUT     1       送回结果
```

测试“返回选通”脉冲是否低电平

```
L10     IN      2       I/O 口 2 送入累加器
        ANI     10 H    抽出“返回选通”脉冲
        JZ      FFB     如果它是 0，则转到对 FFB 的模拟.
```

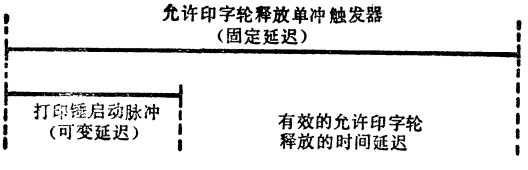
当触发器FFF被置“1”时，会发生什么情况呢？

FFF(Q)输出端向上接到坐标为A6 的“与”门 37 的引线端 9，这是提供“印字轮释放”信号的逻辑的一部份。然而 FFF

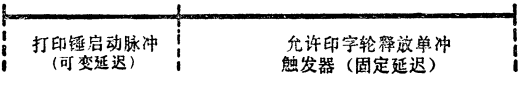
(Q)输出端由低向高的跳变是无效的，这是因为“与”门37的另一输入端是FFD($\overline{Q}$)输出，目前它是低电平。FFF(Q)输出端和“与”门37相连，以保证在打印周期开始，当FFD($\overline{Q}$)端是高电平时，保持“印字轮释放”为低电平。

EFF的Q端和$\overline{Q}$端输出是FFC的J端和K端提供输入的。FFF($\overline{Q}$)是“与”门12的一个输入端，它的输出成为FFC(J)端输入，这个“与”门的另一输入端是坐标为$A_3$上的“与”门37的输出。在这一打印周期中，它经常保持高电平。因此，当FFF($\overline{Q}$)端输出是低电平时，FFC（J）端输入也是低电平。FFC的K输入端是FFF(Q)端的输出；因此，当它的K端为高电平时，FFC将被置“0”，而在FFF置“1”之前，K端将不会是高电平。

但是，在我们的模拟中，把FFC的置“0”推迟到“打印锤脉冲”结束之前，这是因为FFC置“0”的目的是触发“允许印字轮释放”单冲触发器。这个单冲触发器能产生打印锤稳定返回所要求的时间延迟。从而，与其采用下述的并行的时间延迟：

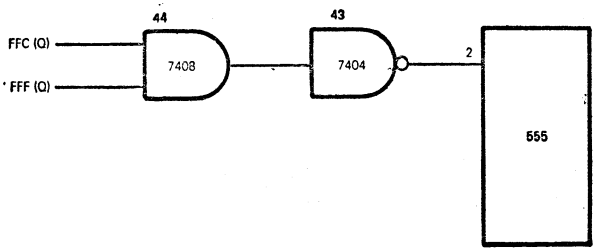



我们宁可采用串行的时间延迟，它能更直接地满足逻辑要求：

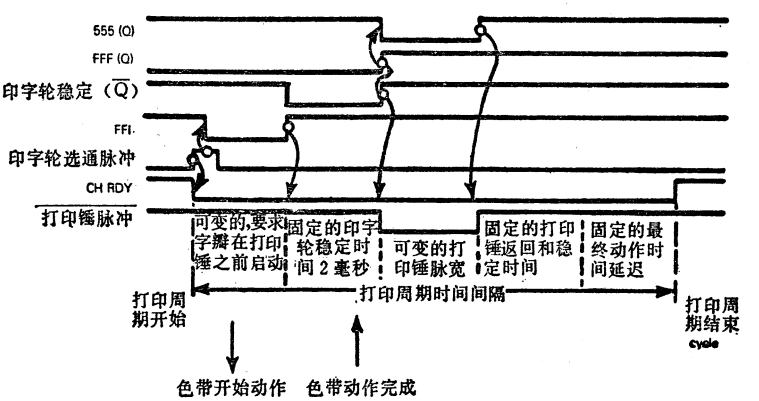



这个“打印锤启动”脉冲是由555单冲触发器产生的。因此，

555 单冲触发器在时间序列中提供下一事件：它是由引线端 2 上的由低到高的跳变来触发的。这个引线端输入的产生过程如下所示：




这是必须模拟的事件序列




## 3-3-16 555 多谐振荡器

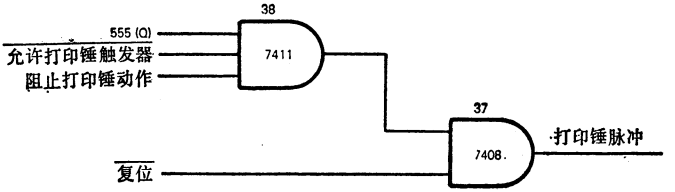
如果将图 3-1 中 555 多谐振荡器的连接方式与第二章介绍的多谐振荡器相比较，你会看到，在两次打印周期之间，触发器 FFB 由于在引线端 4 上输入低电平而使多谐振荡器“停振”。

正像我们刚才所讨论的，触发器FFF(Q)端的输出用于触发多谐振荡器。

**单冲触发电路可变脉冲**

单冲触发器输出脉冲的持续时间受输入H1至H6的控制。这六个输入之中只有一个是“真”值，而其他五个都是“假”值。这样，多谐振荡器每触发一次就输出单冲“高电平”脉冲，它的持续时间是六种可能值中的一种。

555多谐振荡器单冲输出最后转换成为“打印锤脉冲”输出。但是为了产生“打印锤脉冲”输出，分别位于坐标B 8和C7处的“与”门37和38的另一些输入必须都是高电平。我们可以用下图说明产生“打印锤脉冲”的逻辑：




在产生“打印锤脉冲”输出之前，我们将简单地测试一下“$\overline{\text{允许打印锤动作触发器}}$”输入。

我们必须模拟这个“阻止打印锤动作”开关。

我们可以不管“复位”脉冲，因为“复位”逻辑是在两次打印周期之间模拟的。

### 3-3-17 555多谐振荡器的模拟

555多谐振荡器的模拟由以下逻辑序列组成：

1）确定作为“打印锤脉冲”输出的555单冲触发器输出的传送条件是否已经满足。

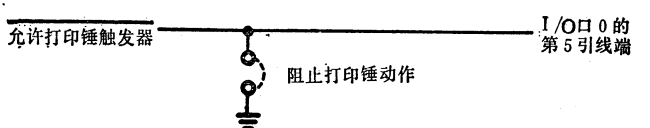
2）测试H1至H6的输入。由这些输入确定产生六种不同时间延迟中的一种。

3）如果已经满足“打印锤脉冲”输出的条件，那么555单冲触发器的输出就转换为“打印锤脉冲”输出。

首先，让我们看一下“打印锤脉冲”的输出允许逻辑，测试允许打印锤动作触发器的状态是足够简单的，它已分配在I/O口0的引线端6上。

**微型计算机以外的逻辑**

但是，在汇编语言程序中没有开关，我们如何来模拟阻止打印锤动作呢？我们可以将余下的一个引线端，I/O口0的引线端5，分配给输入信号，这个信号是由一个外部开关产生的。这好像简单地把这个开关接入允许打印锤动作触发器通路一样，如下图所示。




因此，我们将不管这个“阻止打印锤动作”开关，而使“打印锤脉冲”输出能提供高电平的允许打印锤动作触发器输入。

555多谐振荡器输出的六种可能的持续时间如何呢？我们在第二章曾经指出，将16位数存入两个寄存器中能产生多长的时间延迟。然后在程序循环中对这个寄存器的内容不断减“1”。这个程序循环一直进行到寄存器内容减到“0”为止。下面的指令循环被重复执行：

```
        LXI     D,T16   ;LOAD TIME CONSTANT INTO D AND E
LOOP    DCX     D       ;DECREMENT DE
        MOV     A,D     ;TEST FOR ZERO BY ORING
        OR      E       ;D AND E CONTENTS VIA ACCUMULATOR
        JNZ     LOOP
```

```
        LXI     D, T 16     将时间常数存入D和E寄存器
LOOP    DCX     D           DE 寄存器内容减 1
        MOV     A, D        DE寄存器内容通过累加器相"或"后，
        OR      E           测试是否为零。
        JNZ     LOOP
```

选择六种可能的时间延迟中的一种，如同选择六种可能的初始时间常数中的一个一样简单。现在我们能模拟 555 多谐振荡器，如下所示：

```
        IN      1           ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     40H         ;SET BIT 6 TO 1
        OUT     1           ;OUTPUT THE RESULT
;TEST HAMMER ENABLE FF
        IN      0           ;INPUT I/O PORT 0 TO ACCUMULATOR
        ANI     40H         ;ISOLATE BIT 6
        JZ      HPO         ;IF ZERO, BYPASS SETTING HAMMER PULSE LOW
;HAMMER ENABLE FF IS HIGH, SO HAMMER PULSE
;MUST BE OUTPUT LOW. THEREFORE SET BIT 2 OF
;I/O PORT 3 TO 0
        IN      2           ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     0BH         ;SET BIT 2 TO 0
        OUT     2           ;OUTPUT RESULT
;COMPUTE TIME DELAY
HPO     LXI     H,DELY      ;LOAD DATA ADDRESS BASE INTO HL
        LDA     H.IH6       ;LOAD SELECTOR INTO ACCUMULATOR
HP1     RRC                 ;ROTATE ACCUMULATOR, SET CARRY TO A0
        INX     H           ;INCREMENT HL CONTENTS BY 2
        INX     H
        JNC     HP1         ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
        MOV     D,M         ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
        INX     H
        MOV     E,M
TDLY    DCX     D           ;EXECUTE TIME DELAY LOOP
        MOV     A,D
        ORA     E
        JNZ     TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
        IN      2           ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     4           ;SET BIT 2 TO 1
        OUT     2           ;OUTPUT RESULT
```

```
        IN      1       I/O口1送入累加器
        ORI     40H     将第6位置"1"
        OUT     1       输出结果
```

```
测试"允许打印锤动作触发器"的状态
        IN      0       I/O口0送入累加器
        ANI     40H     抽出第6位
        JZ      HPO     如果是0，则绕过将"稳定打印锤脉冲"置低电平
                        的程序版
因为"允许打印锤动作触发器"是高电平，所以"打印锤脉冲"必须为低
电平输出，因此将I/O口3的第2位置"0"
        IN      2       I/O口2送入累加器
        ANI     FBH     将第2位置0
        OUT     2       输出结果
计算时间延迟
HPO     LXI     H, DELY 将数据基地址存入HL中
        LDA     H1, H6  将选择器存入累加器中
HP1     RRC             累加器内容循环右移一位，将A置入进位位。
        INX     H       将HL内容增2。
        INX     H
        JNC     HP1     如果RRC指令尚未将"1"移入进位位，就返回
        MOV     D,M     将16位时间延迟常数存入DE中
        INX     H
        MOV     E,M
TDLYDCX D               执行时间延迟循环
        MOV     A,D
        ORA     E
        JNA     TDLY
再次输出高电平的"打印锤脉冲"。
        IN      2       I/O口2送入累加器
        ORI     4       将第2位置"1"
        CUT     2       输出结果
```

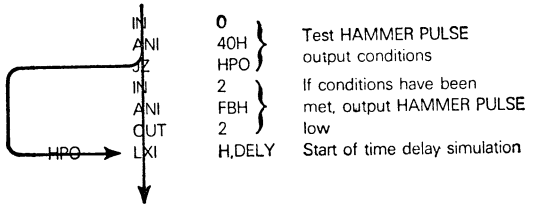
与迄今我们已模拟的其它设备比较起来，模拟555多谐振荡器需要更多的指令。虽然看起来这里好像有很深的道理，而实际上逻辑相当简单。我们来分段解释。

**信号允许** 首先，我们测试允许打印锤动作触发器的状态，只有当允许打印锤动作触发器是高电平时，“打印锤脉冲”才能输出低电平。用来测试允许打印锤动作触发器状态的三条指令是：

```
IN      0       ;INPUT I/O PORT 0 TO ACCUMULATOR
ANI     40H     ;ISOLATE BIT 6
JZ      HPO     ;IF ZERO, BYPASS SETTING HAMMER PULSE LOW
```

```
IN      0       I/O口0送入累加器
ANI     40 H    抽出第6位
JZ      HPO     若为“0”则跳过将“打印锤脉冲”输出低电平的指
                令序列
```

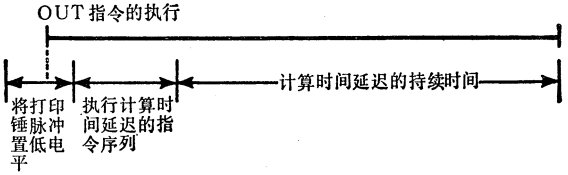
这三条指令有两方面需要说明：首先，逻辑是实现了，我们要确定“打印锤脉冲”输出低电平的条件是否已经满足。如果条件已经满足，则“打印锤脉冲”立即输出低电平。如果条件还不满足，则由 JZ HPO 指令分支跳过将“打印锤脉冲”输出低电平的指令序列如下：



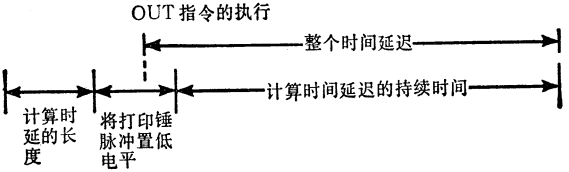

```
IN      0
ANI     40 H    } 测试“打印锤脉冲”输出的条件是否已经满足
JZ      HPO     }
IN      2
ANI     FBH     } 如果条件已经满足，则“打印锤脉冲”输出低电平
OUT     2       }
HPO  LXI  H, DELY   时间延迟的模拟开始
```

**事件序列**

在开始计算时间脉冲的持续时间之前，我们先输出低电平的“打印锤脉冲”。为什么呢？理由是为了节省时间。用来计算时间延迟长度的指令能在时间延迟开始时就执行。




我们也可以很容易地计算出时间延迟，然后将“打印锤脉冲”置为低电平，并执行时间延迟。这些事件按如下的时间顺序发生。

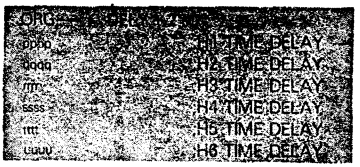



**许多在时间上重叠的事件就更有意义了。**

用来计算时间延迟的实际方法需要简单解释一下。在程序结束时，在存储器的 12 个字节中存有六个 16 位的常数，源程序如下所示：

```
HP0     LXI     H,DELY    ;LOAD ADDRESS OF FIRST DELAY INTO HL
        LDA     H1H6      ;LOAD SELECTOR INTO ACCUMULATOR
HP1     RRC               ;ROTATE ACCUMULATOR, SET CARRY TO A0
        INX     H         ;INCREMENT HL CONTENTS BY 2
        INX     H
        JNC     HP1       ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
        MOV     D,M       ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
        INX     H
        MOV     E,M
```

```
TDLY    DCX     D       ;EXECUTE TIME DELAY LOOP
        MOV     A,D
        ORA     E
        JNZ     TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     4       ;SET BIT 2 TO 1
        OUT     2       ;OUTPUT RESULT
```




```
HPO   LXI   H,DELY   第一个时间延迟地址存入 HL 中
      LDA   H1,H6    时间延迟选择指针存入累加器
HP1   RRC            累加器内容循环右移一位，将 A0
                     置入进位位。
      INX   H        HL 内容加 2
      INX   H
      JNC   HP1      如果 RRC 指令尚未将“1”移入
                     进位位，就返回
      MOV   D,M      16 位时间延迟常数存入 DE 中
      INX   H
      MOV   E,M
TDLY  DCX   D        执行时间延迟循环
      MOV   A,D
      ORA   E
      JNZ   TDLY
```

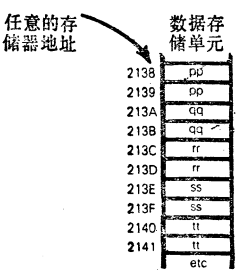
“打印锤脉冲”再次输出高电平

```
      IN    2    I/O口 2 送入累加器
      ORI   4    将第 2 位置“1”
      OUT   2    输出结果
      ■
```

```
        ORG     DELY + 2
        pppp            H 1 时间延迟
        qqqq            H 2 时间延迟
        rrrr            H 3 时间延迟
        ssss            H 4 时间延迟
        tttt            H 5 时间延迟
        uuuu            H 6 时间延迟
```

p、q、r、s、t 和 u 这些字母用来表示十六进制数，六个时间延迟可以表示从 OOOO 到FFFF 之间的任何值。

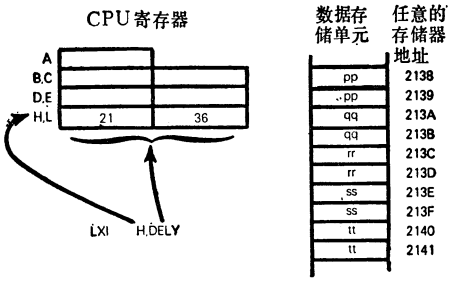
存储第一个时间延迟的第一个存储器字节的地址由表达式 DELY + 2 给出，假定这个存储器地址是2138：




DELY 是一个标号，必须给它赋予 2136 这个初值。这项任务是由一条 Equate 命令来完成的。如下所示，它出现在程序的开始

```
DELY    EQU   2136 H
```

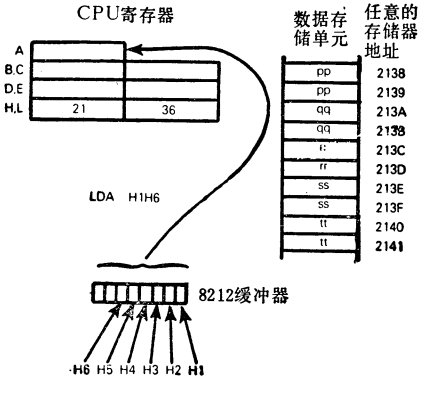
现在我们将地址 DELY 送入 H 和 L 寄存器中，并开始计算时间延迟。假定如上所示，标号DELY 赋值为 2136 H，那么在执行指令 LXI H,DELY 之后，情况如下图所示：




下条指令 LDA H1H6 将 8212 八位缓冲器的内容送入累加器中。使缓冲区选择它本身的那个存储器地址是用标号H1H6来代表的。假定这个存储器地址是 $FFFF_{16}$，那么 H1H6 就能用一条 Equate 命令在程序开始时给定如下：

```
DELY        EQU        2136H
H1H6        EQU        FFFFH
```

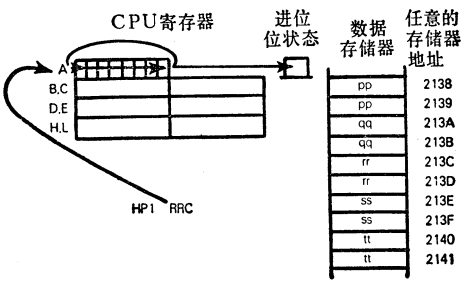
在我们对于输入信号的讨论中就曾指出，这六个输入信号 H1 至 H6 中，只有一个信号是高电平，而其它五个信号都是低电平。因此，执行完 LDA 指令后，累加器的后六位中将只有一位是“1”。



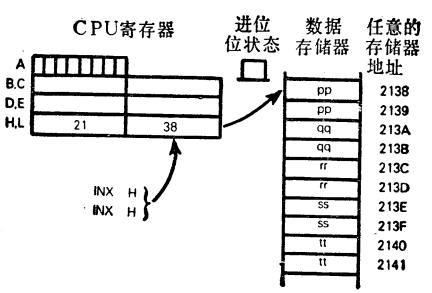
**数据存储地址计算**

根据累加器中“1”的位置所指定的次数，我们可以用H和L寄存器内容增2数次的方法，计算出所要求的时间延迟的地址，这可表示如下：

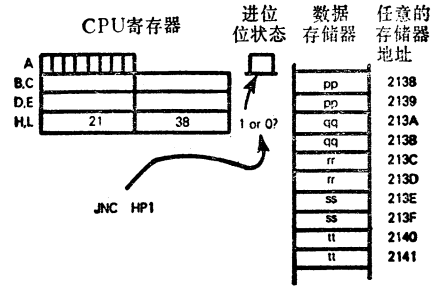
① 将累加器的内容向右移一位，移入进位位中；




② H，L寄存器内容增2；




③ 如果进位位不为1，则返回到①继续进行移位，直到进位位等于1为止；如果进位位为1，则在HL的内容即为正确的地址。




实现所要求的地址加法的逻辑由以下四条指令给出：

```
HP1     RRC             ;ROTATE ACCUMULATOR, SET CARRY TO A0
        INX     H       ;INCREMENT HL BY 2
        INX     H
        JNC     HP1     ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
```

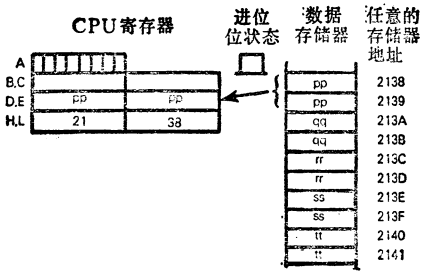
HP1　　RRC　　将累加器内容循环右移一位，将A0移入进位位

INX　H　　HL 寄存器内容增 2

INX　H

JNC　HP1　　如果 RRC 指令尚未将“1”移到进位位，则返回

现在，所要得到的正确时间延迟是由H和L寄存器来访问的，我们将所指向的时间延迟是常数送入寄存器D,E中。假定H 1是高电平输入信号，则结果如下：



所选择的地址 PPPP 由下列三条指令移入寄存器 D,E 中

```
MOV     D,M     ;MOVE CONTENTS OF BYTE 2138 TO D REGISTER
INX     H       ;ADDRESS BYTE 2139
MOV     E,M     ;MOVE CONTENTS OF BYTE 2139 TO E REGISTER
```

```
MOV   D,M   将 2138 的字节内容移入寄存器 D 中
INX   H     访问字节2139
MOV   E,M   将 2139 的字节内容移入寄存器 E 中
```

实际的时间延迟是由下列指令循环产生的这已在第二章讲过：

```
TDLY    DCX     D       ;DECREMENT DELAY COUNTER
        MOV     A,D     ;TEST FOR 0 IN DE BY ORING D
        ORA     E       ;WITH E IN ACCUMULATOR
        JNZ     TDLY    ;RETURN IF NOT ZERO
```

```
TDLY  DCX   D       时间延迟计数器内容减“1”
      MOV   A,D     测试 DE 内容是否为0，方法是
      ORA   E       将 E 和 D 通过累加器相“或”
      JNZ   TDLY    若不为零，则返回
```

上述三条指令使“打印锤脉冲”输出为高电平，而对于“打印锤脉冲”是否为低电平，不进行测试。因为如果输出的“打印锤脉冲”原来就是高电平，现在输出高电平不会产生明显的影响，所以这种逻辑是可行的。在这类情况下，执行最后三条指令所需的时间其实被浪费掉了。然而，为了测试“打印锤脉冲”曾被置入低电平也要用三条指令，所以这种浪费得到了弥补。

**时间延迟计算**

现在，让我们稍微思考一下计算时间延迟所需要的时间。有关指令的执行时间列述如下：

| Cycles | | Instruction | | |
|---|---|---|---|---|
| | | IN | 2 | |
| | | ANI | FBH | |
| | | OUT | 2 | |
| 10 | HP0 | LXI | H,DELY | ← Hammer pulse low starts here |
| 13 | | LDA | H1H6 | |
| 4 | HP1 | RRC | | These four instructions |
| 5 | | INX | H | will be executed between 1 |
| 5 | | INX | H | and 6 times. 26 cycles |
| 10 | | JNC | HP1 | are in this loop. |
| 7 | | MOV | D,M | |
| 5 | | INX | H | |
| 7 | | MOV | E,M | |
| 5 | TDLY | DCX | D | |
| 5 | | MOV | A,D | |
| 4 | | ORA | E | |
| 10 | | JNZ | TDLY | |
| 10 | | IN | 2 | |
| 7 | | ORI | 4 | |
| 10 | | OUT | 2 | ← Hammer pulse low ends here |
| 117 | | | | |

| 周期 | | 指令 | | |
|---|---|---|---|---|
| | | IN | 2 | |
| | | ANI | FBH | |
| | | OUT | 2 | |
| | | | | ←"打印锤脉冲"低电平从这里开始 |
| 10 | HPO | LXI | H,DELY | |
| 13 | | LDA | H1H6 | |
| 4 | HP1 | RRC | | |
| 5 | | INX | H | 这四条指令执行次数是 |
| 5 | | INX | H | 1到6之间，一次循环持续 |
| 10 | | JNC | HPI | 24个周期 |
| 7 | | MOV | D,M | |
| 5 | | INX | H | |
| 7 | | MOV | E,M | |
| 5 | TDLY | DCX | D | |
| 5 | | MOV | A,D | |
| 4 | | ORA | E | |

```
10          JNZ   TDLY
10          IN    2
 7          OR    4
            OUT   2
10                      ←—"打印锤脉冲"低电平在这里结束
117
```

假定一个时钟周期为500毫微秒，则执行时间由下式给出：

$$(46.5+12\times N)\text{微秒}$$

式中N是介于最短脉冲为1和最长脉冲为6之间的一个数，因此执行时间将在58.5微秒和118.5微秒的范围内。必须从相继产生的时间延迟中减去这些时间。例如，假定H1为高电平要求555输出一个单冲信号，它是持续1.65毫秒(近似地)的高电平信号。那么1.6毫秒的时间延迟加上58.5微秒的建立时间就已足够。

### 3-3-18 "允许印字轮释放"触发器

一旦555单冲触发器的输出重新变成低电平之后，就模拟触发器FFC被置"0"。当FFC被置"0"时。它的$\overline{Q}$端就输出由低到高的跳变，而且由它去触发"允许印字轮释放"单冲触发器，这是一个位于坐标$E_2$附近，标号为36的74121单冲触发器。这个单冲触发器的用途是在任何试图使印字轮重新定位之前就给予打印锤稳定返回的时间。这是用固定的打印锤返回和稳定的时间延迟来说明的。

### 3-3-19 "允许印字轮释放"触发器的模拟

**时间延迟**

这是一个真正地分为两步的模拟。首先，我们必须模拟触发器FFC被置"0"然后必须

执行适当的时间延迟，3 毫秒的时间延迟已经足够。将触发器 FFC 置“0”的指令将在 3 毫秒时间延迟之内执行完毕。下面是相应的指令序列：

```
        JNC     HP1         ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
        MOV     D,M         ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
        INX     H
        MOV     E,M

TDLY    DCX     D           ;EXECUTE TIME DELAY LOOP
        MOV     A,D
        ORA     E
        JNZ     TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
        IN      2           ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     4           ;SET BIT 2 TO 1
        OUT     2           ;OUTPUT RESULT
;SWITCH FLIP-FLOP FFC OFF
        IN      1           ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     FBH         ;SET BIT 2 TO 0
        OUT     1           ;OUTPUT RESULTS
;EXECUTE A 3-MILLISECOND TIME DELAY
        LXI     D,F7H       ;LOAD TIME CONSTANT INTO D,E
PWR1    DCX     D           ;DECREMENT D,E
        MOV     A,D         ;TEST FOR ZERO RESULT
        ORA     E
        JNZ     PWR1        ;REDECREMENT IF NOT ZERO
```

JNC　HP 1 如果 RRC 指令尚未将“1”移入进位位，则返回

MOV　D,M 将 16 位时间延迟常数送入 DE 寄存器中

INX　H

MOV　E,M

TDLY　DCX　D　执行时间延迟循环

MOV　A,D

ORA　E

JNZ　TDLY

“打印锤脉冲”再次输出为高电平

IN　2　I/O 口 2 送入累加器

ORI　4　将第 2 位置“1”

OUT　2　输出结果

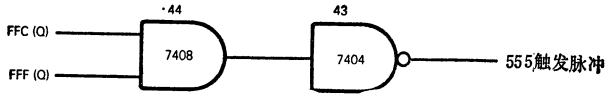
```
    将触发器FFC置"0"
          IN    1      I/O口1送入累加器
          ANI   FBH    将第2位置"0"
          OUT   1      输出结果
    执行3毫秒时间延迟
          LXI   D,F7H  将时间延迟常数送入D,E寄存器中
PWR1      DCX   D      D,E内容减1
          MOV   A,D    测试结果是否为零
          ORA   E
          JNZ   PWR1   如果尚未减到零，则再减1
```

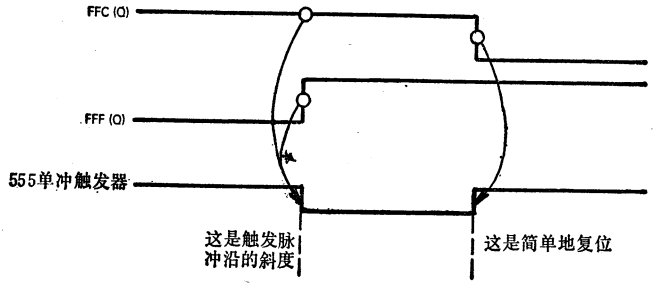
这个时间常数F 7 H等于$247_{10}$。前面的四条指令在34微秒内执行完毕，而在时间延迟循环中的四条指令在12微秒内执行完毕了，因此，整个时间延迟由下式给出：

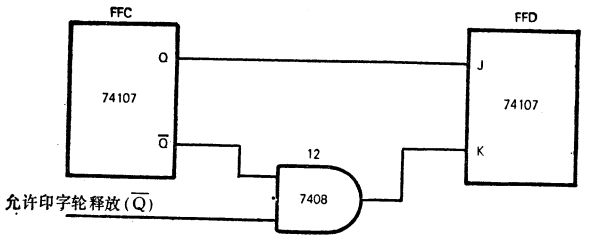
$$247\times12+34=2998\text{ 微秒}$$

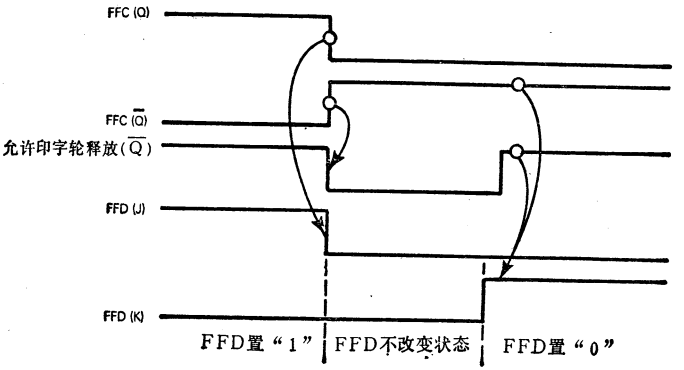
实际上3毫秒的时间延迟不是一个必须严格遵守的数据。2.5或3.5毫秒大致都可以。因此在这个例子中，上面的34微秒是没有什么意义的。尽管如此，在其它的应用中，对于时间延迟的持续时间的要求就可能很苛刻。那时，对上面讨论的时间上的考虑就很有意义了。

为了确定在"印字轮释放"时间延迟结束后会发生什么情况，我们必须观察FFC的Q端和$\overline{Q}$端输出。其Q输出端和"启动色带移动"脉冲"与"门以及555单冲触发器逻辑相接，无论在哪种情况下，Q端的由高向低的跳变都不起作用。因为"启动色带移动"脉冲早已是低电平，而555单冲触发器是由Q端的由高到低的跳变来触发的。由低到高的跳变仅仅使触发信号上升为高电平，这不需要模拟。


·44
43
FFC (Q)
FFF (Q)
7408
7404
555触发脉冲


FFC (Q)
FFF (Q)
555单冲触发器
这是触发脉冲沿的斜度
这是简单地复位


FFC
FFD
Q
74107
J
74107
K
12
7408
允许印字轮释放(Q)


FFC (Q)
FFC (Q)
允许印字轮释放(Q)
FFD (J)
FFD (K)
FFD置“1”
FFD不改变状态
FFD置“0”

FFC($\overline{Q}$)输出和“允许印字轮释放”单冲触发器的$\overline{Q}$端信号相“与”，以便产生FFD(K)输入。FFD(J)输入直接由FFC(Q)来提供。因此，一旦“允许印字轮释放”单冲输出再次被置成高电平，FFD的J输入端将接受一个低电平信号，而K输入端接受一个高电平信号。

当J端为低电平，K端为高电平时，触发器FFD被置“0”。并由它去触发“允许印字轮准备”单冲触发器。

### 3-3-20 “允许印字轮准备”单冲触发器的模拟

与这个单冲触发器有关的逻辑和“允许印字轮释放”单冲触发器几乎相同，FFD置“0”引起它的$\overline{Q}$端由低到高的跳变，并由它去触发“允许印字轮准备”单冲触发器。

现在我们来模拟2毫秒时间延迟，相继的指令序列与“允许印字轮释放”单冲触发器的模拟程序几乎相同。

```
        JNC     HP1     ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
        MOV     D,M     ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
        INX     H
        MOV     E,M
TDLY    DCX     D       ;EXECUTE TIME DELAY LOOP
        MOV     A,D
        ORA     E
        JNZ     TDLY

;OUTPUT HAMMER PULSE HIGH AGAIN
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     4       ;SET BIT 2 TO 1
        OUT     2       ;OUTPUT RESULT
;SWITCH FLIP-FLOP FFC OFF
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     FBH     ;SET BIT 2 TO 0
        OUT     1       ;OUTPUT RESULTS
;EXECUTE A 3 MILLISECOND TIME DELAY
        LXI     D,F7H   ;LOAD TIME CONSTANT INTO D,E
PWR1    DCX     D       ;DECREMENT D,E
        MOV     A,D     ;TEST FOR ZERO RESULT
        ORA     E
        JNZ     PWR1    ;REDECREMENT IF NOT ZERO
```

```
;SWITCH FLIP FLOP FF0 OFF
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     F7H     ;SET BIT 3 TO 0
        OUT     1       ;OUTPUT RESULTS
;EXECUTE A 2 MILLISECOND TIME DELAY
        MVI     A,83H   ;LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2    DCR     A       ;DECREMENT ACCUMULATOR
        JNZ     PWR2    ;REDECREMENT IF NOT ZERO
```

```
        JNC   HP1     如果 RRC 指令尚未将"1"移入进位位，则返回
        MOV   HP1     将 16 位时间延迟常数送入 D，E 寄存器中
        INX   H
        MOV   E, M
TDLY    DCX   D       执行时间延迟循环
        MOV   A, D
        ORA   E
        JNZ   TDLY
```

将"打印锤脉冲"重新输出高电平

```
        IN    2   I/O 口 2 送入累加器
        ORI   4   将第 2 位置 1
        OVT   2   输出结果
```

将触发器 FFC 置"0"

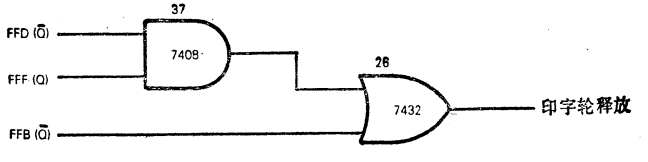
```
        IN    1       I/O 口 1 送入累加器
        ANI   FBH     将第 2 位置"0"
        OUT   1       输出结果
```

执行 3 毫秒时间延迟

```
        LXI   D, F7H  将时间常数送入 D，E 寄存器中
PWR1    DCX   D       D，E 寄存器内容减 1
        MOV   A,D     测试结果是否为 0
        ORA   E
        JNZ   PWR1    如果不为0，则再减 1
```

```
   将触发器 FFD 置"0"
   IN    1        I/O 口 1 送入累加器
   ANI   F7H   将第 3 位置"0"
   OUT   1        输出结果
   执行 2 毫秒时间延迟
   MVI   A, 83H   将时间常数送入累加器
PWR2  DCR   A        累加器内容减 1
   JNZ   PWR2   如果不为 0，则再减 1
```

当 FFD 置"0"时，"印字轮释放"再次输出高电平。下面是产生"印字轮释放"的逻辑：




此时，FFB（$\overline{Q}$）端仍为低电平，但是 FFD（$\overline{Q}$）端和 FFF（$\overline{Q}$）端二者都是高电平。因此，"与"门 37 输出高电平，它通过"或"门 26 将"印字轮释放"置成高电平

下列指令将"印字轮释放"置成高电平

```
          JNC     HP1     ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
          MOV     D,M     ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
          INX     H
          MOV     E,M
TDLY      DCX     D       ;EXECUTE TIME DELAY LOOP
          MOV     A,D
          ORA     E
          JNZ     TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
          IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
          ORI     4       ;SET BIT 2 TO 1
          OUT     2       ;OUTPUT RESULT
;SWITCH FLIP-FLOP FFC OFF
          IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
          ANI     FBH     ;SET BIT 2 TO 0
          OUT     1       ;OUTPUT RESULTS
```

```
;EXECUTE A 3 MILLISECOND TIME DELAY
        LXI     D,F7H     ;LOAD TIME CONSTANT INTO D,E
PWR1    DCX     D         ;DECREMENT D,E
        MOV     A,D       ;TEST FOR ZERO RESULT
        ORA     E
        JNZ     PWR1      ;REDECREMENT IF NOT ZERO
;SWITCH FLIP-FLOP FFD OFF
        IN      1         ;INPUT I/O PORT  TO ACCUMULATOR
        ANI     F7H       ;SET BIT 3 TO 0
        OUT     1         ;OUTPUT RESULT
;EXECUTE A 2 MILLISECOND TIME DELAY
        MVI     A,83H     ;LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2    DCR     A         ;DECREMENT ACCUMULATOR
        JNZ     PWR2      ;REDECREMENT IF NOT ZERO
;SET PW REL HIGH
        IN      2         ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     1         ;SET BIT 0 TO 1
        OUT     2         ;OUTPUT RESULT
```

```
        JNC   HPI     如果 RRC 指令尚未将"1"移入进位位，则返回
        MOV   D, M    将 16 位时间延迟常数送入 DE 中
        INX   H
        MOV   E, M
TDLY    DCX   D       执行时间延迟循环
        MOV   A, D
        ORA   E
        JNZ   TDLY
```

"打印锤脉冲"重新输出高电平

```
        JN    2   I/O 口 2 送入累加器
        ORI   4   将第 2 位置"1"
        OUT   2   输出结果
```

将触发器 FFC 置"0"

```
        IN    1     I/O 口 1 送入累加器
        ANI   FBH   将第 2 位置"0"
        OUT   1     输出结果
```

执行 3 毫秒时间延迟

```
        LXI   D, F7H   将时间常数送入 DE 寄存器中
PWR1    DCX   D        D, E 寄存器内容减 1
```

```
        MOV  A,D        测试结果是否为零
        ORA  E
        JNZ  PWR 1      如果不为零，则再减1
将触发器 FFD 置"0"
        IN   1          I/O 口 1 送入累加器
        ANI  F7H        将第 3 位置"0"
        OUT  1          输出结果
执行 2 毫秒时间延迟
        MVI  A，83H     将时间常数送入累加器
PWR 2   DCR  A          累加器内容减 1
        JNZ  PWR2       如果不为零，则再减 1
```

```
将"印字轮释放"置成高电平
        IN   2          I/O 口 2 送入累加器
        ORI  1          将第 0 位置"1"
        OUT  2          输出结果
```

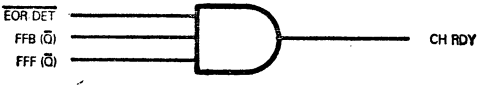
现在整个打印周期即将结束，触发器 FFD 的Q 端 和$\overline{Q}$端输出成为 FFE 的J 端和 K 端输入。Q 输出首先跟 FFI 相"与"，而这时 FFI 是恒定的高电平，因而，在此瞬间， FFD 是被置成"0"的；所以 FFE 的 J 输入端接收低电平。

FFE(K) 输入端在"允许印字轮准备"单冲信号结束之前是不会上升为高电平的，因为"允许印字轮准备"触发器的$\overline{Q}$输出端是和 FFD 的$\overline{Q}$端，相"与"，以便产生 FFE(K) 端的信号。

FFE 被置"0"是按时间顺序发生的下一个事件。

实际上，FFE 的置"0"会引起 FFB 和 FFF 被置"0"，FFB 被置"0"是由于 FFE（$\overline{Q}$）端的由低到高的正跳变引起的。这个正跳变构成 FFB 的时钟输入。FFF 被置"0"是由于它的 J 和 K 输入端直接与 FFE 的 Q 和$\overline{Q}$输出端相接的结果。

一旦 FFB 和 FFF 被置“0”，如果“$\overline{\text{EOR DET}}$”不标识色带结束，则将“CH RDY”重新置成高电平的所有条件都已得到满足：




于是我们可以用下列的指令序列来结束我们的模拟

```
          JNZ       PWR1      ;REDECREMENT IF NOT ZERO
;SWITCH FLIP-FLOP FFD OFF
          IN        1         ;INPUT I/O PORT 1 TO ACCUMULATOR
          ANI       F7H       ;SET BIT 3 TO 0
          OUT       1         ;OUTPUT RESULT
;EXECUTE A 2 MILLISECOND TIME DELAY
          MVI       A,83H     ;LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2      DCR       A         ;DECREMENT ACCUMULATOR
          JNZ       PWR2      ;REDECREMENT IF NOT ZERO
;SET PW REL HIGH
          IN        2         ;INPUT I/O PORT 2 TO ACCUMULATOR
          ORI       1         ;SET BIT 0 TO 1
          OUT       2         ;OUTPUT RESULT
;TURN OFF FLIP-FLOPS FFB, FFE AND FFF
          IN        1         ;INPUT I/O PORT 1 TO ACCUMULATOR
          ANI       ADH       ;RESET BITS 1, 4 AND 6 TO 0
          ORI       20H       ;SET BIT 5 TO 1
          OUT       1         ;OUTPUT RESULTS
;SET CH RDY HIGH
          IN        2         ;INPUT I/O PORT 2 TO ACCUMULATOR
          ORI       2         ;SET BIT 1 TO 1
          OUT       2         ;OUTPUT RESULT
;BRANCH TO TEST FOR VALID END OF PRINT CYCLE
          JMP       LOP1
```

JNZ PWR 1 如果不为零，则再减 1

将触发器 FFD 置“0”

| | | |
|---|---|---|
| IN | 1 | I/O 口 1 送入累加器 |
| ANI | F7H | 将第 3 位置 1 |
| OUT | 1 | 输出结果 |

执行 2 毫秒时间延迟

MUI A，83H 将时间常数送入累加器

```
PWR2  DCR  A          累加器内容减1
      JNZ  PWR2       如果不为0，则再减1
将"印字轮释放"置成高电平
      IN   2   I/O口2送入累加器
      ORI  1   将第0位置1
      OUT  2   输出结果
```

```
将触发器FFB，FFE和FFF置"0"
    IN   1      I/O口1送入累加器
    ANI  ADH    将第1、第4和第6位置"0"
    ORI  20H    将第5位置1
    OUT  1      输出结果
将"CH RDY"置成高电平
    IN   2   I/O口2送入累加器
    ORI  2   将第1位置1
    OUT  2   输出结果
转而测试打印周期是否有效结束
    JMP  LOP1
```

## 3-4 关于模拟的小结

本章所介绍的整个模拟程序示于图3-3。

我们可以断然做出结论，在微型计算机系统中，采用汇编语言指令来一对一地模拟数字逻辑是不合适的，而且也没有意义。

如果你不是一位数字逻辑的设计者，对于图3-1中所示出的各信号的组合，你也许会感到惶恐，这里所做的许多事与Qume打印机本身的要求无关；更确切地说，它反映一个逻辑设计者是怎样实现内部逻辑的。它的目的是在所有可能的情况下，保证有合适的外部信号序列。

如果你是一位逻辑设计者，你有机会用完全不同的方法来实现 Qume 打印机接口的特殊要求。对于上述的实现方法，你甚至可能相当不满意。

重要的是应当记住，数字逻辑中包含无数的技巧，它们适用于分立元件的逻辑电路。这些技巧与最终实现的结果没有关系。

汇编语言有它自己的技巧之处。它也与最终实现的结果没有关系。更确切地说，它的目的在于最有效地利用个别指令或指令序列。

因此，用汇编语言来正确地实现数字逻辑既不适宜，也是我们所不希望的。我们将撇开数字逻辑而从程序的观点来论述这个问题。

**汇编语言与数字逻辑**

数字逻辑和汇编语言之间的本质不同在于：汇编语言处理事件是按时间顺序进行的，而数字逻辑把逻辑划分成一些功能节点。于是，一个逻辑设备能产生在任何逻辑周期期间不同的时刻所发生的许多事件。而翻译成汇编语言程序的时候，每一个事件都变成一个孤立的指令序列。

例如，在图 3-1 中打印周期是从一连串触发器被置“1”开始，而又以一些触发器被置“0”来结束。在很多情况下，一个触发器被置“1”时能触发一个事件，而同一触发器置“0”时又能触发另一个完全不同的事件。在汇编语言程序内部，这两个事件没有任何共同之处。每一个事件都用一个位于程序中的不同的部位的完全独立的指令序列来表示。

数字逻辑和汇编语言的另一个主要不同在于定时概念方面。在同步的数字逻辑内部，如图 3-1 所示，定时必须具有时钟信号，在各种操作之间还需要一个清零信号。（接184页）

```
;ASSIGN LOCATIONS TO DELAY COUNT TABLE
;AND TIME DURATION SELECT LINES
DELY    EQU     xxxx
H1H6    EQU     yyyy
;TEST FOR VALID END OF PRINT CYCLE
LOP1    IN      2       ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        RLC             ;SHIFT BIT 7 INTO CARRY
        JNC     LOP1    ;IF ZERO IN CARRY, STAY IN PRINT CYCLE
;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT BITS 6, 4, 3, 2 AND 1 TO 0, 5 AND 0 TO 1
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     21H     ;SET BITS 5 AND 0 TO 1
        ANI     A1H     ;RESET BITS 6, 4, 3, 2 AND 1 TO 0
        OUT     1       ;RETURN RESULTS
;TEST FOR RETURN STROBE LOW
L10     IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE RETURN STROBE
        JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION
;SIMULATION OF FFA AND ASSOCIATED LOGIC
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
;ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
;AND RESET, RESPECTIVELY
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     62H     ;ISOLATE BITS 6, 5 AND 1
        CPI     22H     ;IF RESET=0, CH RDY=1 AND PW STROBE=1
        JNZ     L10     ;START NEW PRINT CYCLE. OTHERWISE RETURN TO L10
        IN      1       ;TO START NEW PRINT CYCLE, SET
        ANI     FEH     ;I/O PORT 1, BIT 0 TO 0
        OUT     1
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH     ;RESET BIT 1 TO 0
        OUT     1       ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FCH     ;RESET BITS 0 AND 1 TO 0
        OUT     2       ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
;ALSO SIMULATE FFC AND FFD TURNING ON. SET BITS 3 AND 2 OF I/O PORT 1 TO 1
        IN      1       ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     0DH     ;SET BITS 3, 2 AND 0 TO 1
        OUT     1       ;RESTORE RESULT
;PULSE START RIB MOTION HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     8       ;SET BIT 3 HIGH
        OUT     2       ;OUTPUT RESULT
        ANI     F7H     ;TURN BIT 3 OFF
        OUT     2       ;OUTPUT RESULT
;TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC    IN      0       ;INPUT I/O PORT 0 TO ACCUMULATOR
        RLC             ;SHIFT BIT 7 INTO CARRY
        JNC     VLDC    ;STAY IN LOOP IF CARRY IS 0
```

```
;AT END OF DELAY SIMULATE FFE SWITCHING ON
        IN      1       ;INPUT I/O PORT 1
        ANI     DFH     ;RESET BIT 5
        ORI     10H     ;SET BIT 4
        OUT     1       ;OUTPUT THE RESULT
;SIMULATE 2 MS PW SETTLING TIME DELAY
        MVI     A,0     ;LOAD ACCUMULATOR WITH 0
PWS     DCR     A       ;DECREMENT A
        JNZ     PWS     ;IF A DOES NOT DECREMENT TO 0,
                        ;RE-DECREMENT
;SIMULATE FLIP-FLOP FFF SWITCHING ON
FFF     IN      0       ;INPUT I/O PORT 0 CONTENTS TO ACCUMULATOR
        CMA             ;COMPLEMENT TO TEST FOR ALL 1 BITS
        ANI     1FH     ;ISOLATE BITS 0 THROUGH 4
        JNZ     FFF     ;IF THERE WERE ANY 0 BITS, STAY IN LOOP
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI     40H     ;SET BIT 6 TO 1
        OUT     1       ;OUTPUT THE RESULT
;TEST HAMMER ENABLE FF
        IN      0       ;INPUT I/O PORT 0 TO ACCUMULATOR
        ANI     40H     ;ISOLATE BIT 6
        JZ      HP0     ;IF ZERO, BYPASS SETTING HAMMER PULSE LOW
;HAMMER ENABLE FF IS HIGH, SO HAMMER PULSE
;MUST BE OUTPUT LOW. THEREFORE SET BIT 2 OF
;I/O PORT 3 TO 0
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     FBH     ;SET BIT 2 TO 0
        OUT     2       ;OUTPUT RESULT
;COMPUTE TIME DELAY
HP0     LXI     H,DELY  ;LOAD DATA ADDRESS BASE INTO HL
        LDA     H1H6    ;LOAD SELECTOR INTO ACCUMULATOR
HP1     RRC             ;ROTATE ACCUMULATOR, SET CARRY TO A0
        INX     H       ;INCREMENT HL CONTENTS BY 2
        INX     H
        JNC     HP1     ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
        MOV     D,M     ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
        INX     H
        MOV     E,M
TDLY    DCX     D       ;EXECUTE TIME DELAY LOOP
        MOV     A,D
        ORA     E
        JNZ     TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     4       ;SET BIT 2 TO 1
        OUT     2       ;OUTPUT RESULT
;SWITCH FLIP-FLOP FFC OFF
        IN      1       ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     FBH     ;SET BIT 2 TO 0
        OUT     1       ;OUTPUT RESULTS
```

```
;EXECUTE A 3 MILLISECOND TIME DELAY
        LXI     D,F7H     ;LOAD TIME CONSTANT INTO D,E
PWR1    DCX     D         ;DECREMENT D,E
        MOV     A,D       ;TEST FOR ZERO RESULT
        ORA     E
        JNZ     PWR1      ;REDECREMENT IF NOT ZERO
;SWITCH FLIP-FLOP FFD OFF
        IN      1         ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     F7H       ;SET BIT 3 TO 0
        OUT     1         ;OUTPUT RESULTS
;EXECUTE A 2 MILLISECOND TIME DELAY
        MVI     A,83H     ;LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2    DCR     A         ;DECREMENT ACCUMULATOR
        JNZ     PWR2      ;REDECREMENT IF NOT ZERO
;SET PW REL HIGH
        IN      2         ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     1         ;SET BIT 0 TO 1
        OUT     2         ;OUTPUT RESULT
;TURN OFF FLIP-FLOPS FFB, FFE AND FFF
        IN      1         ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     ADH       ;RESET BITS 1, 4 AND 6 TO 0
        ORI     20H       ;SET BIT 6 TO 1
        OUT     1         ;OUTPUT RESULTS
;SET CH RDY HIGH
        IN      2         ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     2         ;SET BIT 1 TO 1
        OUT     2         ;OUTPUT RESULT
;BRANCH TO TEST FOR VALID END OF PRINT CYCLE
        JMP     LOP1
;DELAY COUNT TABLE
        ORG     DELY + 2
        pppp              ;H1 TIME DELAY
        qqqq              ;H2 TIME DELAY
        rrrr              ;H3 TIME DELAY
        ssss              ;H4 TIME DELAY
        tttt              ;H5 TIME DELAY
        uuuu              ;H6 TIME DELAY
```

The letters x, y, p, q, r, s t and u represent hexadecimal values.

图 3-3 整个模拟程序

给时间延迟计数表和持续时间选择线路分配地址

```
DELY    EQU    XXXX
H1H6    EQU    YYYY
```

测试打印周期是否有效地结束

```
LOP1  IN     2        I/O口2内容送入累加器
```

```
        RLC             第7位移入进位位
        JNC   LOP1      若进位位为"0"，则打印周期结束
```

在两次打印周期之间执行的程序

开始时将I/O口1的第6、第4、第3、第2和第1位置"0"将第5位置"1"

```
        IN    1         I/O口1送入累加器
        ORI   21H       将第5和第0位置"1"
        ANI   A1H       将第6、第4、第3、第2和第1位置"0"
        OUT   1         送回结果
```

测试"返回选通"脉冲是否为低电平

```
L10     IN    2         I/O口2送入累加器
        ANI   10H       抽出"返回选通"脉冲
        JZ    FFB       如果它是0，则转至FFB的模拟
```

FFA和有关逻辑的模拟

将I/O口2的内容送入累加器，而且抽出分别对应于"CH RDY"，"印字轮选通"和"复位"的第1、第5和第6位

```
        IN    2         I/O口2送入累加器
        ANI   62H       抽出第6、第5和第1位
        CPI   22H       如果"复位"=0"CH RDY"=1和"印字轮选通"=1
        JNZ   L10       则启动新的打印周期，否则返回L10
        IN    1         启动新的打印周期、
        ANI   FEH       将I/O口1的第0位置"0"
        OUT   1
```

新的打印周期序列在这里开始

模拟触发器FFB置"1"

```
FFB     IN    1         I/O口1送入累加器
        ANI   FDH       将第1位置"0"
        OUT   1         恢复结果
```

模拟7411"与"门，将"CH RDY"置成低电平

再模拟7432"或"门，将"印字轮释放"置成低电平

```
        IN    2         I/O口2送入累加器
        ANI   FCH       将第0和第1位置"0"
        OUT   2         恢复结果
```

"CH RDY"低电平使 FFA 置"0"将 I/O 口 1 的第 0 位置"1"

再模拟 FFC 和 FFD 置"1"，将 I/O 口 1 的第 3 和第 2 位置"1"

```
        IN    1     I/O 口 1 送入累加器
        ORI   ODH   将第 3、第 2 和第 0 位置"1"
        OUT   1     恢复结果
```

将"启动色带移动"脉冲置高电平

```
        IN    2     I/O 口 2 送入累加器
        ORI   8     将第 3 位置高电平
        OUT   2     输出结果
        ANI   F7H   将第 3 位置"0"
        OUT   2     输出结果
```

测试速度译码输入，以确定是哪一种印字轮移动时间延迟，并产生相应的时间延迟

```
VLDC    IN    0           I/O 口 0 送入累加器
        RLC               将第 7 位移入进位位
        JNC   VLDC        如果进位位为 0，则停止循环
```

在时间延迟结束时模拟 FFE 置"1"

```
        IN    1     输入 I/O 口 1
        ANI   DFH   将第 5 位置"0"
        ORI   10H   将第 4 位置"1"
        OUT   1     输出结果
```

模拟 2 毫秒"印字轮稳定"时间延迟

```
        MUI   A,0   将"0"移入累加器 A
PWS     DCR   A     A 内容减 1
        JNZ   PWS   如果 A 内容尚未减至零、则再减 1
```

模拟触发器 FFF 置"1"

```
FFF     IN    0           I/O 口 0 的内容送入累加器
        CMA               为测试累加器中"1"的位数而取反
        ANI   1FH         抽出第 0 至第 4 位
        INZ   FFF         如果这几位都是"0"，则停止循环
        IN    1           I/O 口 1 送入累加器
        ORI   40H         将第 6 位置"1"
        OUT   1           输出结果
```

测试“允许打印锤动作”触发器的状态

```
        IN      0       I/O 口 0 送入累加器
        ANI     40H     抽出第 6 位
        JZ      HPO     如果这位为“0”，则跳过将“打印锤脉冲”置低电平的程序段
```

“允许打印锤动作”触发器为高电平，于是“打印锤脉冲”必须输出低电平，为此必须将I/O口 3 的第 2 位置“0”

```
        IN      2       I/O 口 2 送入累加器
        ANI     FBH     将第 2 位置“0”
        OUT             2 输出结果
```

计算时间延迟

```
HPO     LXI     H,DELY  将数据基地址送入 HL 中
        LDA     H1H6    将时间延迟选择指针送入累加器
HP1     RRC             累加器内容循环右移一位，将AO移入进位位
        INX     H       H,L 内容增 2
        INX     H
        JNC     HP1     如果 RRC 指令尚未将“1”移入进位位、则返回
        MOV     D,M     将 16 位时间延迟常数移入 DE 中
        INX     H
        MOV     E,M
TDLY    DCX     D       执行时间延迟循环
        MOV     A,D
        ORA     E
        JNA     TDLY
```

“打印锤脉冲”再次输出高电平

```
        IN      2   I/O 口 2 送入累加器
        ORI     4   将第 2 位置“1”
        OUT     2   输出结果
```

触发器 FFC 置“0”

```
        IN      1       I/O 口 1 送入累加器
        ANI     FBH     将第 2 位置“0”
        OUT     1       输出结果
```

执行 3 毫秒时间延迟

```
        LXI   D,F7H   时间常数送入D,E寄存器中
PWR1    DCX   D       D,E寄存器内容减1
        MOV   A,D     测试结果是否为零
        ORA   E
        JNZ   PWR1    若结果不为零、则再减1
```

触发器FFD置“0”

```
        IN    1       I/O口1送入累加器
        ANI   F7H     将第3位置“0”
        OUT   1       输出结果
```

执行2毫秒时间延迟

```
        MUI   A,83H   时间常数移入累加器
PWR2    DCR   A       累加器内容减1
        JNZ   PWR2    若尚未减到零，则再减1
```

将“印字轮释放”置高电平

```
        IN    2   I/O口2送入，累加器
        ORI   1   将第0位置1
        OUT   2   输出结果
```

将触发器FFB，FFE和FFF置“0”

```
        IN    1       I/O口1送入累加器
        ANI   ADH     将第1、第4和第6位置“0”
        ORI   20H     将第6位置“1”
        OUT   1       输出结果
```

将“CH RDY”置高电平

```
        IN    2 I/O口2送入累加器
        ORI   2 将第1位置“1”
        OUT   2 输出结果
```

转至测试打印周期是否有效结束

```
        JMP       LOP1
```

时间延迟计数表

```
        ORG   DELY+2
        pppp      H1时间延迟
        qqqq      H2时间延迟
```

```
rrrr        H 3 时间延迟
ssss        H 4 时间延迟
tttt        H 5 时间延迟
uuuu        H 6 时间延迟
```

字母 x,y,p,q,r,s,t 和 u 表示 16 进制数。

图 3-3　整个模拟程序

(承 176 页)在汇编语言程序内部，定时是严格地按指令的执行顺序来确定的。此外，数字逻辑电路中的元件能够并行地转接和操作。而在汇编语言程序内部，一切都必须串行地产生。

**在本章，要抓住的关键性概念是：把数字逻辑作为一种能够实现任何功能的手段，从根本上说并不正确。**事实上，我们不能采用汇编语言指令来精确地实现数字逻辑，这并不意味着汇编语言在任何方面都差一些。它仅仅意味着汇编语言能以不同的方式完成这个任务。

在本章我们用了一定篇幅比较了汇编语言和数字逻辑。现在我们将放弃利用数字逻辑的任何尝试。到第四章，图 3-1 所示的逻辑将再次地被模拟，但这次是从程序设计者的观点出发的。

# 第四章　一个简单程序

我们在第三章所介绍的有关数字逻辑电路的模拟问题，可以归纳为这样一个事实：我们尝试将逻辑电路分解成一些孤立的传递函数，其中每一传递函数相当于一个数字逻辑设备。现在我们将撇开数字和组合的逻辑电路，假设它们并不存在，而用另一种观点来看待图 3-1 和图 3-2。

## 4-1　汇编语言与数字逻辑在定时关系上的比较

**传递函数**

我们仍然考察图 3-1，并简单地忽略图中自左至右的内容。留下来的只有一组输入信号和一组输出信号。输出信号与输入信号由一组与数字逻辑设备毫无关系的传递函数来连系。

图 3-2 的时间图不严格地表示出图 3-1 的传递函数。所谓“不严格地表示”意味着什么呢？它意味着有关系统所要求的定时关系，和仅仅简单地反映数字逻辑电路需要的定时关系被不加区别地混同在一起了。我们可以不去考虑仅仅简单反映数字逻辑需要的定时概念。具体地说，打印锤仍然应由六个线圈中某一个的输出脉冲所启动，各种移动和建立稳定所需要的时间延迟也必须予以保留。但是我们可以不考虑仅仅为了使数字逻辑电路清零而使某一信号改变状态所需要的时间延迟。

因此，从程序员的角度来看，用图 4-1 所示的时间图来代替逻辑设计者在图 3-2 所示的时间图是完全可行的。

## 4-2 输入和输出信号

由图 4-1 可见，我们撇开了大量的而不是少数几个时间延迟，还撇开了多数的信号。但是有一个确定哪一信号是微型计算机系统中真正需要信号的简单准则，这就是如果该信号是唯一地参与模拟计算机系统以外的逻辑中的实时事件有关的，则此信号必须保留。如果信号的源和目的地都在微型计算机系统的“黑盒子”中，则这个信号可以不予考虑。基于这个准则，让我们从另一个角度来看输入和输出信号。

**输入信号**

首先讨论输入信号。

“返回选通”和“印字轮选通”是没有意义的信号。在数字逻辑电路中，这两个信号是启动打印周期序列的。而在汇编语言程序中，只需转移到该序列的第一条指令上，这就是你所需要的全部初始化操作。事实上，“返回选通”表示的是打印周期中打印锤尚未动作之前的无关紧要的那一部分，因为真正用来抑制打印锤脉冲的是允许打印锤动作脉冲。

我们将各种禁止打印锤动作的信号合并成一个打印锤状态输入信号。有五种这样的信号，它们是：“PFL REL”、“RIB LIFT RDY”、“RIBBON ADVANCE、PFR REL 和“CA REL”。这些信号中的每一个都来自图 3-1 以外的不同的外部逻辑电路。在数字逻辑电路的实现中，这些信号相“与”后产生一个总“打印锤连锁”信号。在汇编语言的实现中，将这些外部信号线“或”到一个引线端上，使之成为“打印锤连锁”状态。

“复位”将作为一个总清信号，接至 CPU 的复位端子上。因此“复位”可以被汇编语言程序所忽略；然而，应当记住“复位”信号一旦被触发后，程序就从保存在 0 号存储单元中的指令作为起点开始执行下去。

“色带用完”将被保留下来。就是这个信号检查色带是否用完，并且阻止打印周期结束，因此在色带用完之后，可以防止再打印字符。

允许打印锤动作触发器必须保留。在印字轮重新定位的各周期期间，它抑制“打印锤动作”脉冲。

由六个打印锤脉冲长度信号 $H_1$ 至 $H_6$ 所实现的功能必须保留。但这些信号本身将消灭掉。这里不用 I/O 口的六个引线端来标示打印锤脉冲的宽度，却直接用由 ASCII 字符码来产生的时间延迟。

现在我们来着重看输出信号。

开始时，可以忽略所有的触发器输出信号。在打印周期内的每个时间间隔的边界已由现有信号状态的改变来标示。如果不止一个外部逻辑事件必须由某一时间间隔到下一时间间隔的过渡所触发，则无需禁止由外部对相应信号的缓冲。这个信号用于触发大量的外部逻辑事件。但在微型计算机程序中，没有理由仅仅为了标示出从一个打印周期的时间间隔到下一个打印周期的过渡而输出重复的信号。

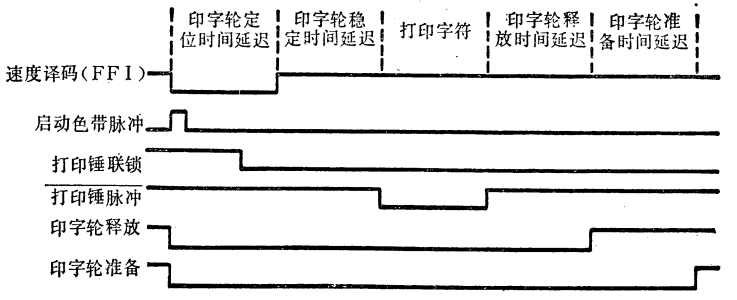
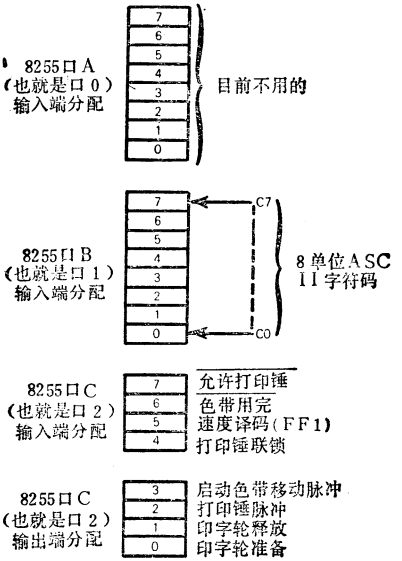



图 4-1　根据程序员的观点代替图 3-1 的时间图

其余的输出信号要保留。如果在微型计算机系统里用更多的汇编语言程序代替附加的外部逻辑电路的话，那么就有可能忽略某些输出信号。但是如果限定问题的范围，则当确定出打印周期的时间间隔时，余下的一些信号就需要加以保留。

**引线端的分配**

由于我们给出新的经过简化了的一组信号，从而可以省去8212 I/O缓冲器，而采用一个8255可编程序的外部接口(PPI)，以方式0操作，I/O口及引线端的分配如下图所示：




## 4-3 微型计算机器件的组态

我们现在选择程序执行过程中所需要的器件。这种选择的确是十分简捷的；除了CPU以外，还需要一个8255可编程序

外部接口，几个存放程序的只读存储器，以及几个存放一般数据的读/写存储器。实际上，CPU 本身是由三个器件组成的，即 CPU 本身、8224 时钟芯片和 8228 总线控制器。把这些器件组合起来，如图 4-5 所示即得到了微型计算机系统。如果你对图 4-5 不能马上了解的话，请先不要失望，因为在这张图中只有很少几个部分是同现在的讨论直接有关的。

### 4-3-1 一般的设计概念

由图 4-5 所得到的一个最重要的概念是：在微型计算机系统中用汇编语言程序进行逻辑设计时，你所编写的程序将是高度依赖于器件组态的。如图 4-5 所示的这种器件的组合方式并不是唯一的，其它的方式也同样可行。然而，对于不同的微型计算机器件组态所编写的汇编语言程序是有显著不同的，这就是当你编写微型计算机程序时不能忽视的一个因素。但是也不要害怕修改你所选择的硬件组态。关于这个问题我们将在第五章详细说明。微型计算机器件组态同汇编语言程序之间具有密切的相互影响。不允许将它们分开考虑。这两个步骤应包含在一个重复循环之中。在编写微型计算机程序的早期阶段，你应能在编写汇编语言程序的过程中发现硬件的某些特性是可以进一步被改善的，从而反过来意味着程序必须重写。

**高级语言**

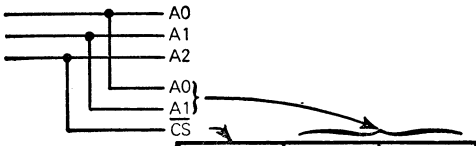
这就说明一个原因，为什么用微型计算机程序代替数字逻辑电路时，宁可不用高级语言。高级语言是面向问题的。譬如说，我们很难从 PL/M 程序语句上看出在语句执行的过程中，数据沿着微型计算机系统传送的具体途径；要把 PL/M 程序和具体的器件组态联系起来甚至就更困难了。另一方面，汇编语言却与硬件具有一一对应的关系。

## 4-3-2 8255 可编程序外部接口

**I/O 口选择**

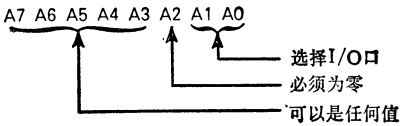
现在我们来看图 4-5 所示器件的具体接法。

8255 可编程序外部接口将响应与 I/O 口、A、B 和 C 相对应的 I/O 口地址 0、1 和 2。这是因为片选端是接到地址总线 A2 上的。在执行 IN(输入)或 OUT(输出)指令时，I/O 口地址在地址总线的 8 条低位线上输出，在图 4-2 中 8255 PPI 所响应的 I/O 口地址如下图所示：




| $\overline{CS}$ | A1 | A0 | 所选中的I/O口 |
|---|---|---|---|
| 0 | 0 | 0 | 口A |
| 0 | 0 | 1 | 口B |
| 0 | 1 | 0 | 口C |
| | | | (高位和低位) |
| 0 | 1 | 1 | 控制口 |

如果某个 PPI 被选中，其 $\overline{CS}$ 必须为“0”。这就是说在图 4-2 中，只要 A2 为“0”，就选择相应的 PPI，而不管 A3 到 A7 是什么值。于是，PPI 将响应除去 4，5，6 和 7 以外的任何一个 I/O 口地址。



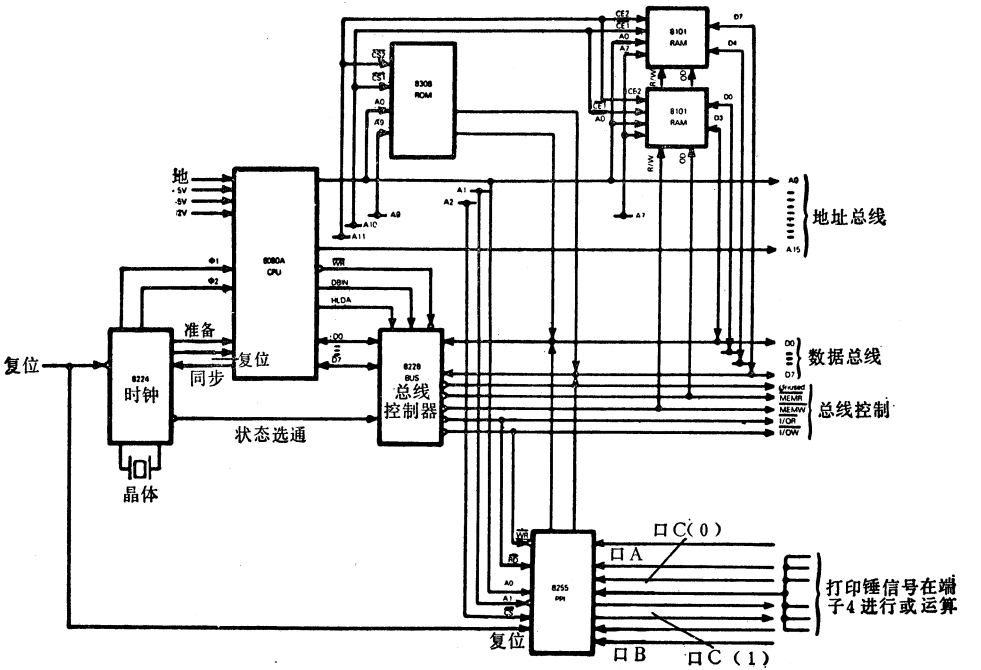

8101
RAM
8308
ROM
8080A
CPU
8224
时钟
8228
BUS
总线
控制器
8255
PPI
地
-5V
-12V
准备
复位
同步
状态选通
晶体
复位
地址总线
数据总线
总线控制
口C(0)
口A
口B
口C(1)
打印锤信号在端子4进行或运算

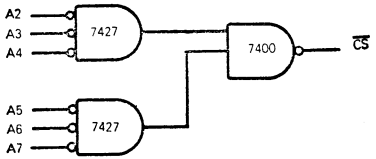
图 4-2　微型计算机组态

**简单系统中的芯片选择**

如果微型计算机组态中包含大量的可编程序外部接口，则芯片的选择逻辑电路就要稍微复杂些了。如果 PPI 只响应唯一的四个 I/O 口地址，除此以外就没有其它地址了，则芯片的选择输入必须由六条高位地址线按某种统一的方式来产生。当然，这意味着微型计算机系统将包含有 64 个 PPI，这是我们所不希望的。

**较大系统中芯片的选择**

假定图 4-2 的 8255 PPI 只响应 I/O 口地址 0、1、2 和 3。下面示出产生片选输入 $\overline{CS}$ 的一种方法：




仅当所有的六条地址线A2 到 A7 都处于低电平时，$\overline{CS}$ 才是低电平，这样就能选择这个 8255 PPI。

有关图 4-2 所示的 8255 PPI 的数据方向和口利用方式的说明并不是一个硬件特性。在任何时候，向 I/O 口 3 写入适当的控制字，就能修改口的利用方式。从 8225 PPI 的结构看，I/O 口 3 是控制口。

**复位逻辑**

对“复位”逻辑也需要做一些解释。我们将用一个硬件的“复位”信号来代替第三章曾介绍过的，在两次打印周期之间测试复位状态的方式。但这是在微型计算机的环境中实现的。连接到 8224 时钟和 8255 PPI 的这个“复位”信号，清除 8255 PPI 中所有的寄存器和 8080 CPU 的程序计数器。这意味着程序的执行将从保存在 0 号单元的存储器字节内的指令起始。所以必须有一个在这一存储单元上开始

算起的复位后的和系统初始化的程序步。

### 4-3-3 系统初始化

当系统被初始化后，必须立即重新建立“在两次打印周期之间”的状态。必要的步骤如下：

1）如果打印锤已经启动，则结束启动脉冲并留出打印锤缩回的时间。

2）把印字轮移回到可见位置。

3）保证各输出信号处于“两次打印周期之间”的状态。

**程序实现序列**

现在我们得出另一个基本的程序设计概念：用汇编语言编写源程序时，有一种“具有最高效率的”程序序列。我们可以先行写出一个初始化程序来执行“复位”，但这需要许多猜测。我们怎么能够知道打印锤已经启动了呢？我们如何能使印字轮移回到可见位置呢？如果“复位”信号将使打印周期提前结束，那么我们在知道它如何提前结束以前就应能编写出打印周期程序。

一般说来，你应选择逻辑中最重要的事件来用程序实现，然后由这开始部分再去实现其他有关的事件。

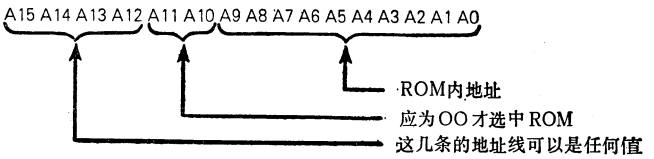
具体地说，实现“复位”逻辑的程序被推迟到打印周期程序编成之后再去编写。

### 4-3-4 只读存储器和随机存取存储器

**只读存储器**

使用复位信号的方法意味着必须有一个 0 地址的存储器字节。在图 4-2 中这个存储器字节位于 8308 ROM 中 ROM 是一个 1024 个字节的只读存储器。它所响应的存储地址从 0 到 03 FF16，而它的片选线连接到地址总线 A10 和 A11 。这可用图表示如下：

**存储器地址**




**在简单系统中ROM的选择**

为了使微型计算机系统简单起见，还是使用原来的ROM片选代码。对于1024个字节的只读存储器，规定地址范围为0到03 $FF_{16}$。但事实上，许多其它的地址也可以访问ROM存储器，地址线A12到A15可以为任何值。规定A10和A11二者为"0"，就能对ROM存储器进行存取。这里不阻拦你采用这种原始的方法来选中存储器芯片，因为你所使用的是一个很小的微型计算机系统。没有理由要增添费用来使用高位的六条地址总线产生复杂的片选代码。甚至从编制程序的观点看，在日后扩大系统和包含更多的存储器时，也不需要重新编写程序。这是考虑到现在你不会使用那些也能够选择ROM的其它地址线，而将来在某个时候你可以从这些地址组中取出一组地址来选择另一个ROM，并且不影响已经写好的程序。

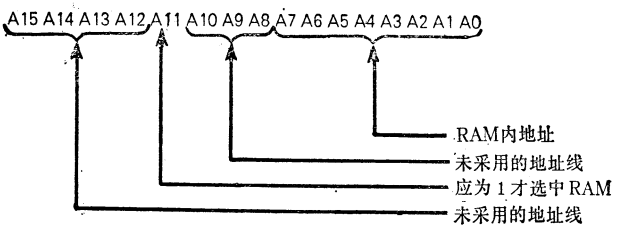
在规定ROM存储程序时，我们假定产品的批量足够大，从而有条件支付采用ROM掩模的费用。如果你的批量不允许支付ROM掩模的费用，那么可以采用可编程序只读存储器(PROM)。

**随机存取存储器**

**存储器地址**

两个8101 RAM器件，每个都提供1024位的读/写存储器。它们是以4位×256的形式组成的。因此每个RAM提供一半的读/写存储器字节。RAM中的256个单元的地址由

$0800_{16}$ 到 $08\,FF_{16}$，这可用图表示如下：




即使将存储器地址由 $0800_{16}$ 到 $08\,FF_{16}$ 规定为 RAM 的寻址空间，也仍然存在大量的其它地址能够选择 RAM。但应注意，决不允许 RAM 的地址同 ROM 的地址相重合：选择 ROM 时地址总线 A11 一定为“0”，而选择 RAM时 A11 一定为“1”，因此决不会发生地址的“冲突”。

对于目前我们讨论的水平来说，图 4-2 的另一些特点无关紧要。因此我们就无需更进一步地了解硬件结构的任何细节了。

### 4-3-5 程序流程图

现在来看微型计算机系统所必须实现的功能。这些功能由图 4-3 所示的流程图来标识。我们将一步一步地分析这个流程图。

我们用速度译码输入信号 (FFI) 标识新的打印周期的开始。因此，在两次打印周期之间，程序连续地使 I/O口 2 的内容送入累加器，抽出第 5 位进行测试，只要这一位等于“1”，新的打印周期就不能开始。一旦这一位等于“0”，就识别出一

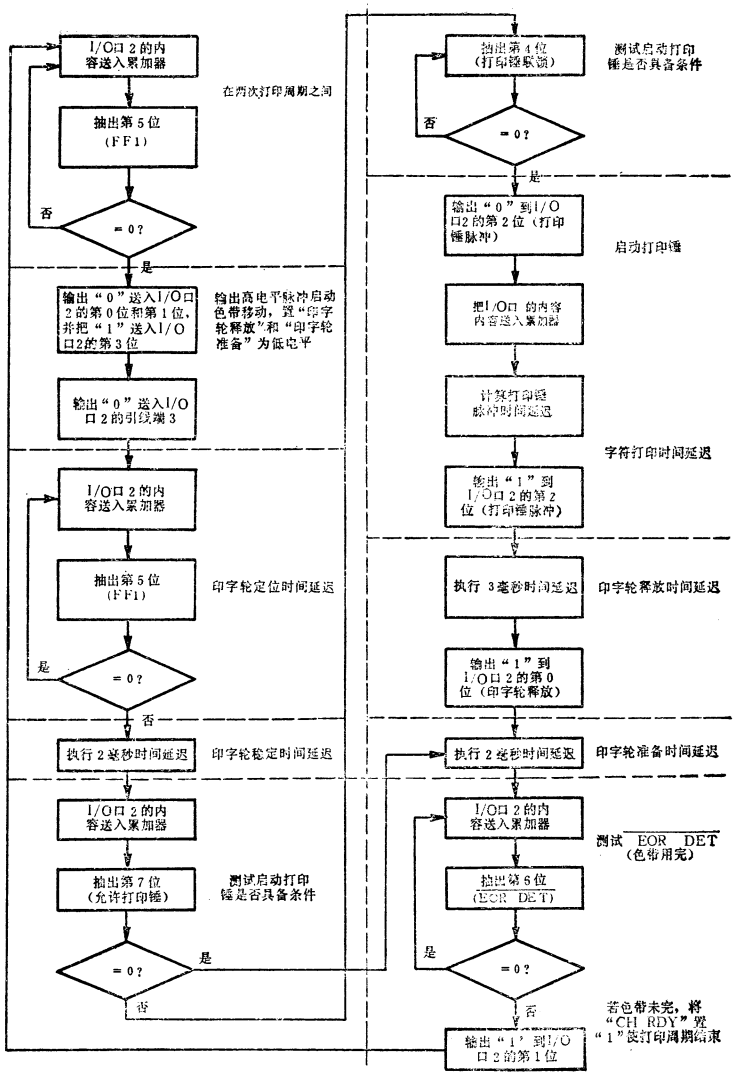
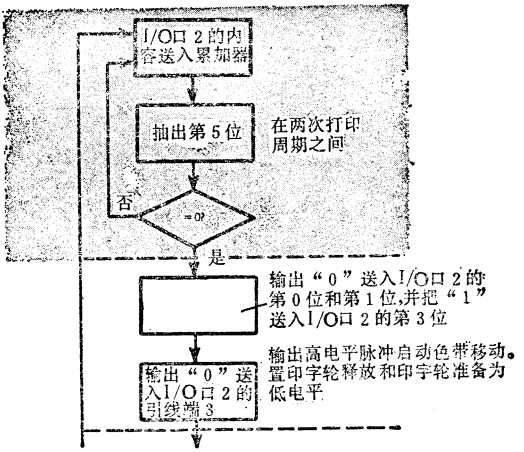


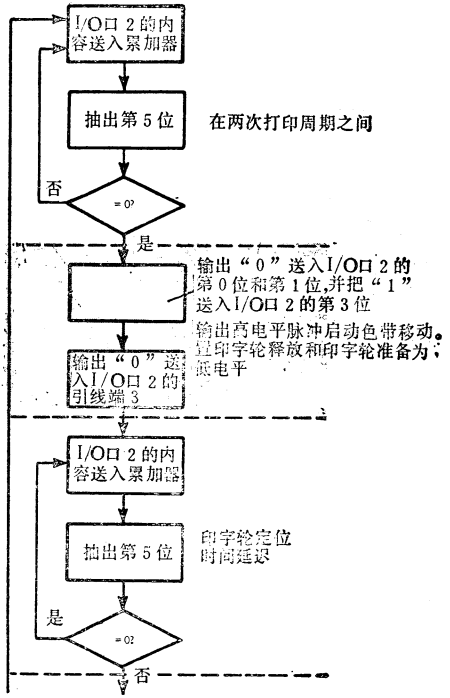

图 4-3 初次设计的程序流程

个新的打印周期：




在新的打印周期中所发生的第一个事件是输出一个高电平的“启动色带移动”脉冲，这是由 I/O口 2 的第 3 位写入“1”，然后写“0”实现的。因为在打印周期的开始时“印字轮释放”和“印字轮准备”两种信号都必须输出低电平，所以也把“0”输出送入 I/O口 2 的第 0 位和第 1 位。

印字轮定位时间延迟是由速度译码信号 FFI 来计算的。只要这个信号是低电平，就说明印字轮正在定位。所以进入了一个可变的时间延迟循环，从程序逻辑来看，这与“两次打印周期之间”的时间延迟循环刚好相反。I/O口 2 的内容再一次送入累加器，并测试第 5 位，而且在第 5 位为“1”之前一直逗留在时间延迟循环之内，一直到第 5 位为“1”，这时印字轮定

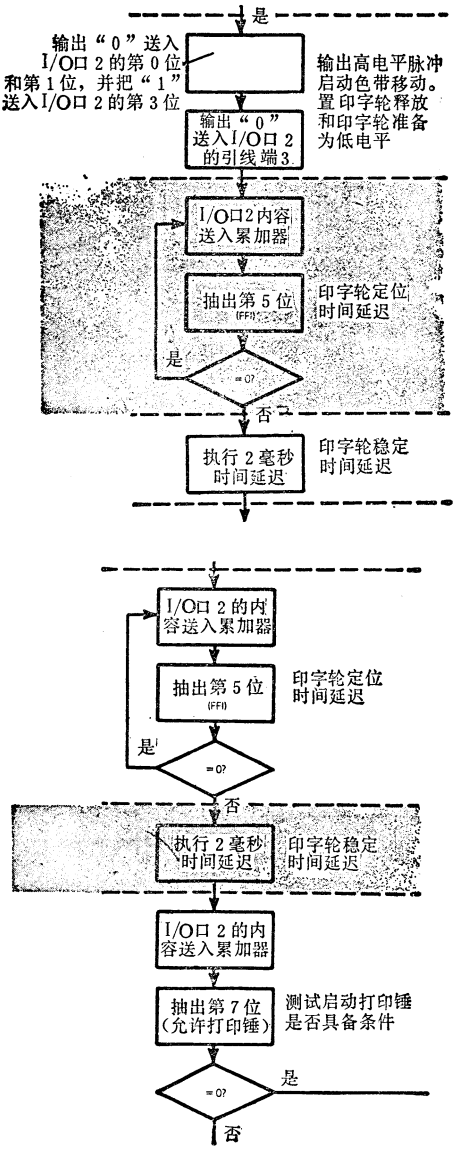



位时间延迟才告结束，见上图：

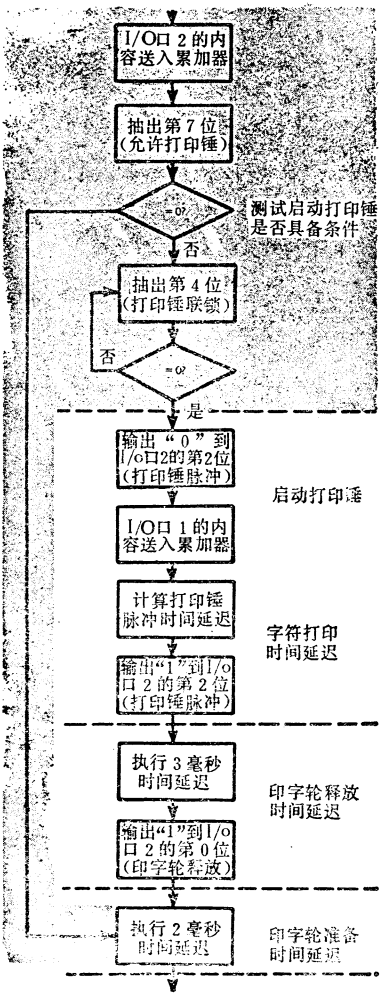
在印字轮定位时间延迟后，必须有 2 毫秒的印字轮稳定时间延迟。下面执行的是通常的时间延迟循环，见下图：

当印字轮稳定时间延迟结束时，如果“打印锤联锁”。信号为低电平，“允许打印锤动作”信号为高电平，打印锤就被启动。如前所述，“打印锤联锁”是一个单独的状态位，所有外部条件都能够用它来阻止打印锤的启动，任何信号对这一状态引线端输入一个高电平都将阻止打印锤的启动。

印字轮重新定位的打印周期是由“允许打印锤动作”输入

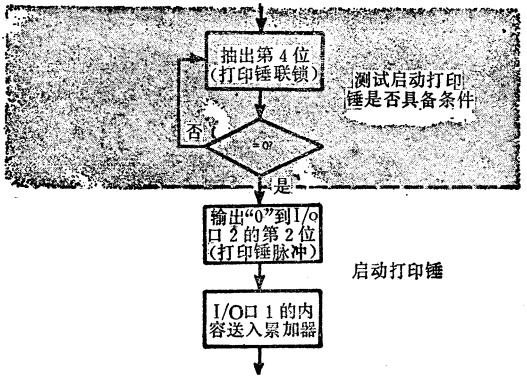

是
输出“0”送入I/O口2的第0位和第1位，并把“1”送入I/O口2的第3位
输出高电平脉冲启动色带移动。置印字轮释放和印字轮准备为低电平
输出“0”送入I/O口2的引线端3
I/O口2内容送入累加器
抽出第5位(FF1)
印字轮定位时间延迟
是
=0?
否
执行2毫秒时间延迟
印字轮稳定时间延迟
I/O口2的内容送入累加器
抽出第5位(FF1)
印字轮定位时间延迟
是
=0?
否
执行2毫秒时间延迟
印字轮稳定时间延迟
I/O口2的内容送入累加器
抽出第7位(允许打印锤)
测试启动打印锤是否具备条件
=0?
是
否

为低电平来识别的。这个状态是在测试“打印锤联锁”状态之前，输出I/O口2的第7位进行测试的，如果I/O口2的第7位等于“0”，则跳过整个打印锤启动程序，直接转移到印字轮准

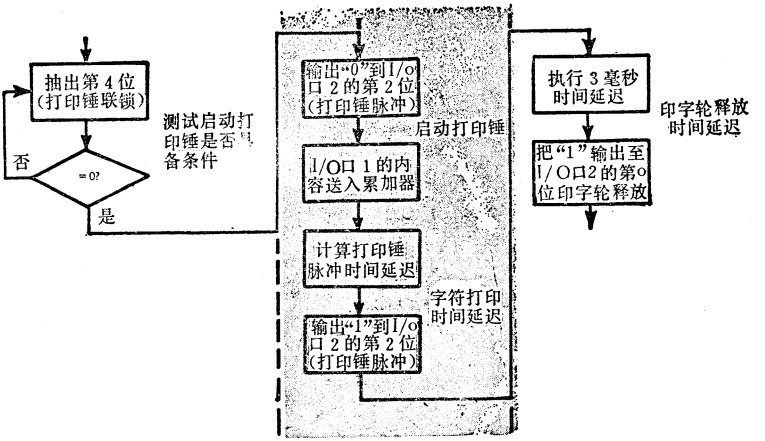


备时间延迟，它是打印周期中最后的时间延迟，如上页图所示。

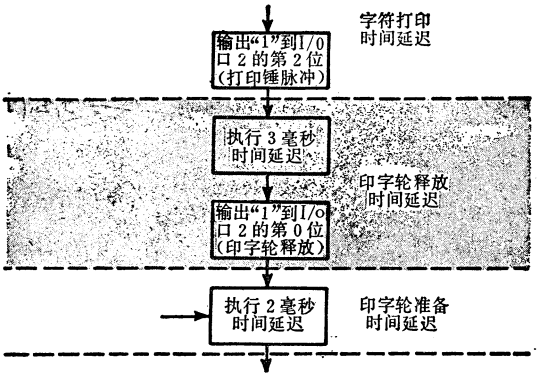
如果允许打印锤动作是高电平，则说明处于字符打印周期。但是只有当“打印锤联锁”为“0”时打印锤才被启动。只要与I/O口2的引线端4线“或”得到的任何信号是高电平，程序就逗留在一个重复的循环里，继续测试这个I/O口引线端的状态。只有当I/O口引线端处于“0”状态时，程序才进入启动打印锤指令序列，见下图：




为了启动打印锤，必须输出一个可变长度的启动脉冲。为此将“0”输出到I/O口2的引线端2，因为这是打印锤脉冲输出所要通过的引线端。然后计算出打印锤脉冲的时间延迟。我们在介绍完这一流程图之后，再介绍如何计算打印锤脉冲的宽度。在打印锤启动时间延迟的结尾，将一个“1”输出到I/O口2的第2位，从而结束打印锤脉冲。



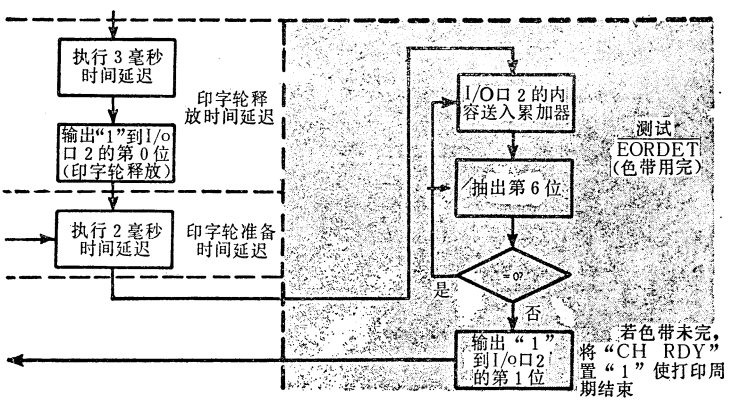

现在紧接着有两个稳定时间延迟。首先是3毫秒的"打印轮释放"时间延迟。它的结束是由将一个"1"输出到I/O口2的第0位来标示的，这就使"印字轮释放"输出为高电平，见下图：




然后执行2毫秒的"印字轮准备"时间延迟。这个时间延迟结束和打印周期的结束一样，都是由输出一个"1"到I/O口2

的第1位来识别的，这使“CH RDY”置为高电平。然而如果有色带用完状态存在，我们就不希望这样做了。这个状态由“色带用完”是低电平来标示。

因此程序把I/O口2的内容送入累加器，并且抽出第6位，从而把“色带用完”信号输入微型计算机系统。如果“色带用完”等于“0”，则程序驻留在一个重复循环中，连续地重新测试I/O口2的第6位，于是下一次打印周期就不能开始。仅当测试出“色带用完”等于“1”时，打印周期才结束，并将“CH RDY”置“1”，见下图：



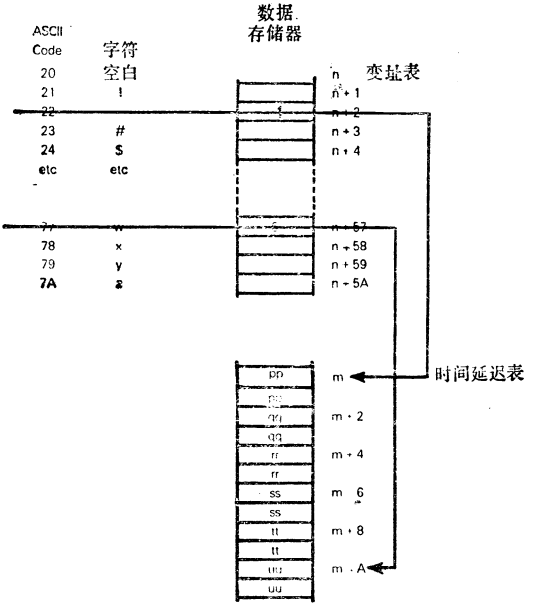

**打印锤启动时间延迟**

现在我们来讨论计算适当的打印锤启动时间延迟的方法。在图3-1中，由6条线（H1到H6）中的某一条线输出为“真值”来标识适当的打印锤启动时间延迟。某一外部逻辑需按照被打印字符的性质来激励这一“真值”线，在微型计算机程序中，这类操作是较易于做到的。

下面就是我们用来计算适当的打印锤启动时间延迟的方法；如附录A所示，每一个被打印的字符由一个ASCII码数

据字节来表示。

若不考虑高位的奇偶校验位，则可得到128种位组合。如果你看一下附录A中列出的ASCII码，则你将发现只有在$20_{16}$和$7A_{16}$之间的字符码才是有效的。因此只需计及$5A_{16}$（或$90_{16}$）个代码组合。这些代码组合中的每一个在90个字节的表中都被分配一个字节，而且在这个字节中存放着1到6之间的一个数。这个数能够标示该字符所需求的时间延迟。在一个12字节的表中，包括了与这六个数相联系的六个实际的时间延迟值。这一方案可用图表示如下：




在上图中，数据存储单元右边的字母“n”和“m”表示某一

有效的基地址。例如，“n”可以表示 $0380_{16}$，而“m”表示 $03FO_{16}$。

我们来讨论两个实例

ASCII 码中 $22_{16}$ 表示双引号符号（,,）。这个符号所要求的时间延迟最短，地址为“n＋2”的数据存储字节对应于这个 ASCII 码，这一数据字节中存放的是“1”。因此用 PPPP 表示的第一种时间延迟，必须在执行长时间延迟循环之前取到 D和 E 寄存器中，而这长时间延迟循环是为字符产生打印锤启动脉冲的。

ASCII 码的 $77_{16}$ 表示“W”，地址为 $n+57_{16}$ 的数据存储字

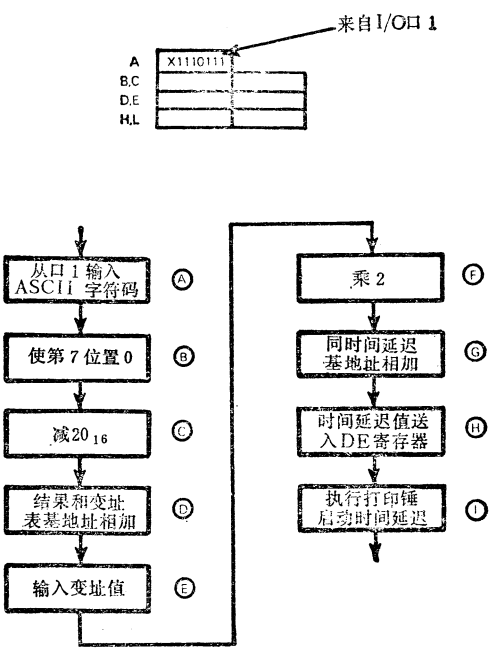



图 4-4　计算打印锤启动脉冲宽度的程序流程图

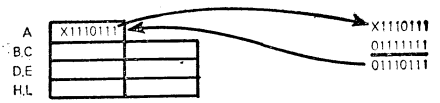
节对应于这个 ASCII 码。在这个数据存储字节里寄存的是“6”，这就是说 W 所要求打印锤启动时间延迟最长。因此，在执行产生“W”字符的打印锤启动脉冲的长时间延迟循环之前，应把用 UUUU 表示的这个时间延迟值取到 D 和 E 寄存器中。

图 4-4 示出了计算打印锤启动时间延迟的程序步。

为了能较好地理解图 4-4，对于W字符的例子，我们从第Ⓐ步做到第Ⓘ步。

Ⓐ 将字符W的小写体 ASCII 码送入累加器。

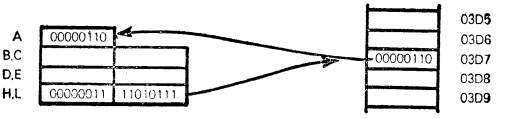
Ⓑ 我们必须将奇偶校验位置“0”。为此将累加器的内容同 7 FH 相“与”。



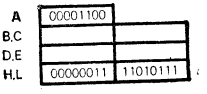

Ⓒ 相当于字符 W 小写体的变址表入口是由该字符的ASCII码减去 $20_{16}$ 后与变址表的基值地址相加而得到的。因为前 $1F_{16}$ 个代码没有等价的 ASCII 码，所以必须减去 $20_{16}$。


Ⓓ 将变址表基值地址取到H和L寄存器内。假定这个地址是 $0380_{16}$，然后将累加器的内容同这个 16 位的地址相加。


Ⓔ 从变址表取出相应的变址值送入累加器。

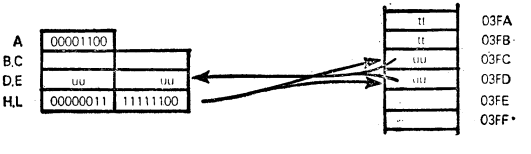



Ⓕ 由于实际的时间延迟是两个字节长，所以我们在计算适当的时间延迟的地址时，应该将变址值加倍后与时间延迟表的基地址相加。首先将变址值乘 2 。




Ⓖ 然后将乘 2 后的变址值与时间延迟表基地址相加，假定这个基地值是 $03FO_{16}$。这个基值地址再次送入H和L寄存器内，然后累加器的内容和H和L寄存器的内容相加。


Ⓗ 把H和L内两字节地址中的内容分别送入D和 E 寄存器内。




Ⓘ D和E寄存器现在寄存的是如第二章中所描述的将要执行的长时间延迟正确的初始值。

```
;HRINT CYCLE PROGRAM
;IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O
;PORT 2 FOR A 0 VALUE)
START   IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     20H     ;ISOLATE BIT 5
        JNZ     START   ;IF NOT 0, RETURN TO START
;INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0
;AND 1 OF I/O PORT 2. OUTPUT 1 TO BIT 3 OF
;I/O PORT 2
        MVI     A,0CH   ;LOAD MASK INTO ACCUMULATOR
        OUT             ;OUTPUT TO I/O PORT 2
;OUTPUT 0 TO BIT 3 OF I/O PORT 2. THIS COMPLETES
;START RIBBON MOTION PULSE
        MVI     A,4     ;LOAD MASK INTO ACCUMULATOR
        OUT     2       ;OUTPUT TO I/O PORT 2
;TEST FOR END OF PRINTWHEEL POSITIONING
LOP1    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     20H     ;ISOLATE BIT 5
        JZ      LOP1    ;IF 0 RETURN TO LOP1
;EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
        MVI     A,0     ;LOAD ACCUMULATOR WITH 0
LOP2    DCR     A       ;DECREMENT A
        JNZ     LOP2    ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
;TEST PRINTHAMMER FIRING CONDITIONS
LOP3    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        RLC             ;MOVE BIT 7 INTO CARRY
        JNC     PRD     ;IF CARRY IS 0, BYPASS PRINTHAMMER FIRING
        ANI     20H     ;ISOLATE BIT 4 WHICH IS NOW BIT 5
        JZ      LOP3    ;WAIT FOR NONZERO VALUE BEFORE FIRING
;FIRE PRINTHAMMER
        IN      2       ;SET HAMMER PULSE LOW. OUTPUT 0
        ANI     FBH     ;TO BIT 2 OF I/O PORT2
        OUT     2
        IN      1       ;INPUT ASCII CHARACTER TO ACCUMULATOR
        ANI     7FH     ;MASK OUT HIGH ORDER BIT
        SUI     20H     ;SUBTRACT 20H
        LXI     H,INDX  ;LOAD INDEX TABLE BASE ADDRESS TO HL
        ADD     L       ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
        MOV     A,M     ;LOAD INDEX INTO ACCUMULATOR
        ADD     A       ;MULTIPLY BY 2
        LXI     H,DELY  ;LOAD DELAY TABLE BASE ADDRESS INTO HL
        ADD     L       ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
        MOV     E,M     ;LOAD DELAY CONSTANT INTO D,E
        INX     H
        MOV     D,M
LOP4    DCX     D       ;EXECUTE LONG DELAY
        MOV     A,D
        ORA     E
        JNZ     LOP4
```

```
        IN      2       ;AT END OF DELAY OUTPUT 1 TO BIT 2
        ORI     4       ;OF I/O PORT 2 (HAMMER PULSE HIGH)
        OUT     2
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
        LXI     D,F7H   ;LOAD TIME CONSTANT INTO D,E
LOP5    DCR     D       ;DECREMENT D,E
        MOV     A,D     ;TEST FOR 0 IN D,E
        ORA     E
        JNZ     LOP5    ;REDECREMENT IF NOT 0
;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
;PW REL HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     1       ;SET BIT 0 TO 1
        OUT     2       ;OUTPUT RESULT
;EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD     MVI     A,0     ;LOAD ACCUMULATOR WITH 0
LOP6    DCR     A       ;DECREMENT A
        JNZ     LOP6    ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
;TEST FOR EOR DET (BIT 6 OF I/O PORT 2) EQUAL
;TO 0 AS A PREREQUISITE FOR ENDING THE PRINT CYCLE
LOP7    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     40H     ;ISOLATE BIT 6
        JZ      LOP7    ;RETURN AND RETEST IF 0
;AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT 2 TO 1
;THIS SETS CH RDY HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     2       ;SET BIT 1 TO 1
        OUT     2       ;OUTPUT RESULT
        JMP     START   ;JUMP TO NEW PRINT CYCLE TEST
```

图 4-5　无初始化，也不复位的一个简单的打印周期指令序列

**打印周期程序**

在两次打印周期之间测试 FFI(I/O 口 2 的第 5 位是否为“0”值)

```
START   IN    2       I/O口 2 的内容送入累加器
        ANI   20H     抽出第 5 位
        JNZ   START   如不为“0”，转到 START
```

打印周期初始化，将“0”输出至 I/O 口 2 的第 0 位和 1 位，将“1”输出至 I/O 口的第 3 位

```
        MVI   A. OCH  将屏蔽值送入累加器
        OUT   2       输出到 I/O 口 2
```

输出“0”到 I/O 口 2 的第 3 位，这就完成了“起动色带移动”脉冲

```
        MVI   A. 4    将屏蔽值送入累加器
        OUT   2       输出至 I/O口 2
```

```
测试印字轮定位是否结束
LOP1    IN    2       I/O口2的内容送入累加器
        ANI   20H     抽出第5位
        JZ    LOP1    如为"0"，返回LOP1
执行2毫秒的印字轮稳定时间延迟
        MVI   A. 0    累加器清"0"
LOP2    DCR   A       A减1
        JNZ   LOP2    如A还未减到零,则再减1
测试打印锤启动条件
LOP3    IN    2       I/O口2的内容送入累加器
        RLC           第7位右移到进位位
        JNC   PRD     如进位是"0"，跳过打印锤启动程序
        ANI   20H     抽出第4位，即当前的第5位
        JZ    LOP3    启动之前不是"0"，返回到LOP3
启动打印锤
        IN    2       置打印锤脉冲为低电平，将"0"
        ANI   FBH     输出至I/O口2的第2位
        OUT   2
        IN    1       将ASCII字符码送入累加器
        ANI   7FH     将高位屏蔽掉
        SUI   20H     减20H
        LXI   H. INDX 取变址表基地址送入HL寄存器
        ADD   L       累加器内容同HL寄存器内容相加
        MOV   L.A
        MOV   A. M    取变址值送入累加器
        ADD   A       乘2
        LXI   H.DELY  时间延迟表基地址送入HL寄存器
        ADD   L       累加器内容和HL寄存器相加
        MOV   L. A
        MOV   E. M    取时间延迟常数送入D. E寄存器
        INX   H
        MOV   D. M
```

```
LOP4        DCX  D        执行长时间延迟
            MOV  A.D
            ORA  E
            JN2  LOP4
            IN   2          在时间延迟结束时，将"1"输出至I/O
            ORI  4          口2的第2位(打印锤脉冲高电平)
            OUT  2
执行3毫秒印字轮释放时间延迟
            LXI  D. F7H  取时间常数送入D. E寄存器
LOP5        DCR  D       D. E减1
            MOV  A.D     测试D. E是否为零
            ORA  E
            JNZ  LOP5    如不为零，再次减量
将"1"输出至I/O口2的第0位，从而将"印字轮释放"置为高电平
            IN   2   I/O口2的内容送入累加器
            ORI  1   将0位置"1"
            OUT  2   输出结果
执行2毫秒的印字轮准备时间延迟
PRD         MVI  A. O    累加器清"0"
LOP 6       DCR  A       A减1
            JN 2  LOP 6  如A还未减到零，再减1
测试"色带用完"(I/O口2的第6位)等于"0"作为打印周期结束的一个先决条件
LOP 7       IN   2       I/O口2的内容送入累加器
            AN 1  40 H   抽出第6位
            JZ   LOP 7   如为"0"返回重新测试
在打印周期结束时，将I/O口2的第1位置"1"，从而将"CH RDY"置为高电平
            IN   2       I/O口2的内容送入累加器
            ORI  2       第1位置"1"
            OUT  2       输出结果
            JMP  START   转到新的打印周期测试
```

图 4-5　无初始化、也不复位的一个简单打印周期的指令序列

把图 4-3 和图 4-4 所示的程序图合并在一起，就得出如图 4-5 所示的整个所要求的程序。现在就来一段一段地说明这个程序。

在两次打印周期之间，下列三指令循环连续地测试I/O口 2 第 5 位的状态。FFI 信号是输入到这一引线端的，只要这信号输入高电平，新的打印周期便不能开始。一旦此信号输入低电平，就标示出印字轮已经运转，这就意味着新的打印周期开始。

```
;PRINT CYCLE PROGRAM
;IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O
;PORT 2 FOR A 0 VALUE)

                                    FFI=1
Enter                                
Program ---- START ---- IN ---- 2 ----  ;INPUT I/O PORT 2 TO ACCUMULATOR
                        ANI     20H     ;ISOLATE BIT 5
                        JNZ     START   ;IF NOT 0, RETURN TO START
                    FFI=0
;INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0
;AND 1 OF I/O PORT 2. OUTPUT 1 TO BIT 3 OF
;I/O PORT 2

                        MVI     A,0CH   ;LOAD MASK INTO ACCUMULATOR.
```

**打印周期程序**

在两次打印周期之间测试 FFI(I/O 口 2 的第 5 位是否为“0”)。

| | | | |
|---|---|---|---|
| 进入程序 START | IN | 2 | I/O 口 2 的内容送入累加器 |
| | ANI | 20 H | 抽出第 5 位 |
| | JNZ | START | 如不为“0”返回到 START |

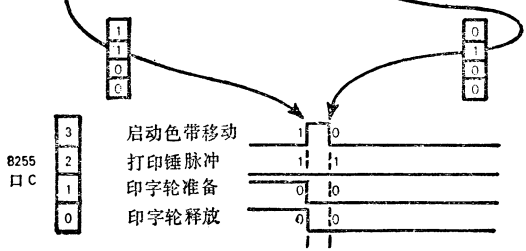
FFI=0

打印周期初始化，将“0”输出至 I/O 口 2 的第 0 位和第 1 位，将“1” 输出至 I/O 口 2 的第 3 位

MVI A.OCH 取屏蔽值送入累加器

一旦新的打印周期开始，就须将“印字轮释放”和“印字轮准备”信号输出为低电平，同时输出一个高电平的“启动色带移动”脉冲。于是，当打印锤启动时，新色带将呈现在该打印字符的前面，这些初始信号变化可以用图说明如下：

```
;INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0
;AND 1 OF I/O PORT 2. OUTPUT 1 TO BIT 3 OF
;I/O PORT 2
        MVI     A,0CH       ;LOAD MASK INTO ACCUMULATOR
        OUT     2           ;OUTPUT TO I/O PORT 2
;OUTPUT 0 TO BIT 3 OF I/O PORT 2. THIS COMPLETES
;START RIBBON MOTION PULSE
        MVI     A,4         ;LOAD MASK INTO ACCUMULATOR
        OUT     2           ;OUTPUT TO I/O PORT 2
```




打印周期初始化，将“0”输出至I/O口2的第0位和第1位，将“1”输出至I/O口2的第3位。

```
MVI   A,  0CH   取屏蔽值送入累加器
OUT   2         输出送到I/O口2
```

输出“0”到I/O口2的第3位，这就完成了“启动色带移动”脉冲

```
MVI   A,  4     取屏蔽值送入累加器
OUT   2         输出送到I/O口2
```

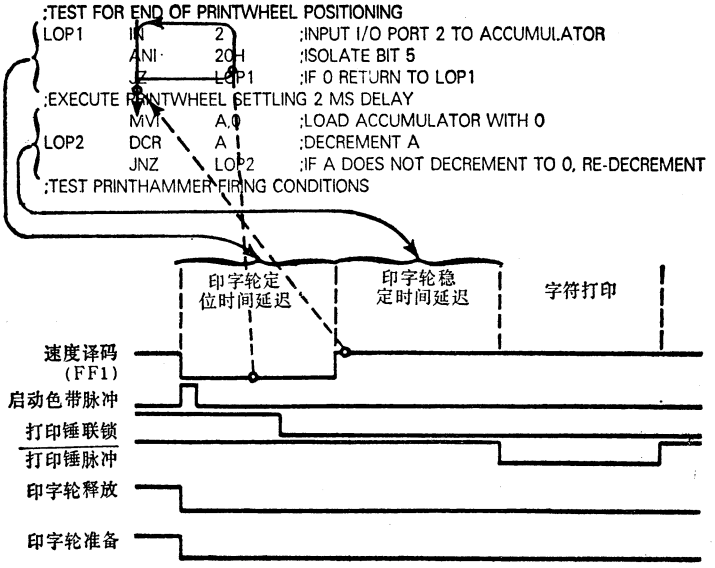
**编程信号脉冲**

在上图中应注意，I/O口C的引线端2被强制输出“1”，这是“$\overline{\text{打印锤脉冲}}$”引线端，这个引线端仅在启动打印锤脉冲的持续时间内输出低电平。而在打印周期的这一时刻，此信号是高电平，所以输出“1”并无妨碍。

**可变长度的时间延迟**

现在程序开始执行可变长度的时间延迟。在此期间，印字轮或是移动到把相应的字瓣呈现在印字轮的前面，或是印字轮返回移到可见位置。在这两种情况中的任何一种情况下，印字轮定位时间延迟

的持续时间内外部逻辑电路输出信号 FFI 是低电平。一旦印字轮已被定位，FFI 将被测出是高电平，程序逻辑进入 2 毫秒的“印字轮稳定”时间延迟。这种三指令时间延迟循环是我们前面经常遇到的：




; 测试印字轮定位是否结束

```
LOP 1   IN      2       I/O口 2 的内容送入累加器
        ANI     20 H    抽出第 5 位
        JZ      LOP 1   如为“0”，返回到 LOP1
```

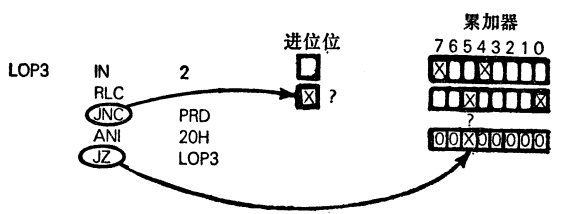
; 执行 2 毫秒的印字轮稳定时间延迟

```
        MVI     A. O    累加器清 0
LOP 2   DCR     A       A 减 1
        JNZ     LOP 2   如A 尚未减到零，则再减
```

; 测试打印锤启动条件

现在打印锤已做好启动的准备。首先测试“允许打印锤动作”的条件，这个信号被接至在I/O口2的引线端7上。如果此信号是低电平，则处在印字轮重新定位的打印周期内。跳过整个打印锤启动指令序列。应当注意，被测试的第7位的状态已经移入进位位。如果“允许打印锤动作”是高电平，则这次测试就通过了，但还要测试“打印锤联锁”信号。这个信号是输入到I/O口2的引线端4上的。

移位指令将第4位(代表“打印锤联锁”)移入第5位，把第7位(代表“允许打印锤动作”)移入进位位，见下图：




当I/O口2的内容已送入累加器一次之后，我们就串行地测试了这两位的状态，对于每一位的测试都分别通过下列六条指令，实现：

```
        IN      2       ;INPUT I/O PORT 2 TWO ACCUMULATOR
        ANI     80      ;ISOLATE BIT 7
        JZ      PRD     ;IF BIT 7 IS ZERO BYPASS PRINTHAMMER FIRING
LOP3    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H     ;ISOLATE BIT 4
        JZ      LOP3    ;WAIT FOR NONZERO VALUE BEFORE FIRING
```

| | | | |
|---|---|---|---|
| | IN | 2 | I/O口2的内容送入累加器 |
| | ANI | 80H | 抽出第7位 |
| | JZ | PRD | 如第7位为“0”，跳过打印锤启动程序 |
| LOP 3 | IN | 2 | I/O口2的内容送入累加器 |
| | ANI | 10 H | 抽出第4位 |
| | JZ | LOP 3 | 启动前，等待非零值 |

如测试到“允许打印锤动作”是低电平，就去分支执行标号为 PRD 的指令。你将发现这条指令处于执行 2 毫秒的“印字轮准备”时间延迟的指令序列的开始。

应当注意，图 4-5 所算的五条指令的序列。在测试“打印锤联锁”是否为高电平的循环内用来测试“允许打印锤动作”是否为低电平。现在，在打印周期的持续时间内“允许打印锤动作”将为高电平或低电平，而在打印周期内电平不变。因此这样不断地测试实在是多余的——它虽无效果，但也无害。

**然后打印锤被启动。**启动打印锤动作的指令序列从第Ⓐ步执行到第Ⓘ步，这在上面已经介绍过。为了便于理解这一指令序列，我们在下面重写的指令序列上添加上Ⓐ到Ⓘ的相应标号。

```
;FIRE PRINTHAMMER
          IN      2          ;SET HAMMER PULSE LOW. OUTPUT 0
          ANI     FBH        ;TO BIT 2 OF I/O PORT2
          OUT     2
     Ⓐ    IN      1          ;INPUT ASCII CHARACTER TO ACCUMULATOR
     Ⓑ    ANI     7FH        ;MASK OUT HIGH ORDER BIT
     Ⓒ    SUI     20H        ;SUBTRACT 20H
     Ⓓ    LXI     H,INDX     ;LOAD INDEX TABLE BASE ADDRESS TO HL
          ADD     L          ;ADD ACCUMULATOR CONTENTS TO HL
          MOV     L,A
     Ⓔ    MOV     A,M        ;LOAD INDEX INTO ACCUMULATOR
     Ⓕ    ADD     A          ;MULTIPLY BY 2
     Ⓖ    LXI     H,DELY     ;LOAD DELAY TABLE BASE ADDRESS INTO HL
          ADD     L          ;ADD ACCUMULATOR CONTENTS TO HL
          MOV     L,A
     Ⓗ    MOV     E,M        ;LOAD DELAY CONSTANT INTO D,E
          INX     H
          MOV     D,M
LOP4      DCX     D          ;EXECUTE LONG DELAY
     Ⓘ    MOV     A,D
          ORA     E
          JNZ     LOP4
          IN      2          ;AT END OF DELAY OUTPUT 1 TO BIT 2
          ORI     4          ;OF I/O PORT 2 (HAMMER PULSE HIGH)
          OUT     2
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
```

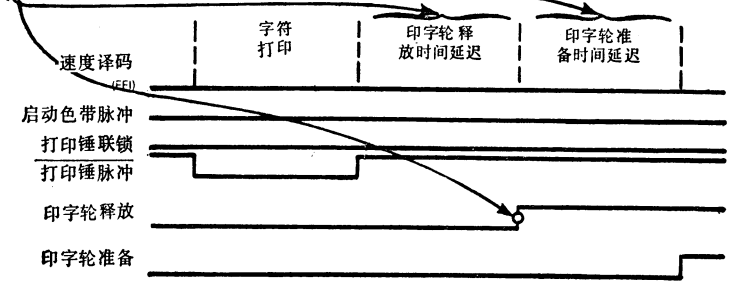
启动打印锤

```
        IN      2       置打印锤脉冲为低电平，将"0"输出
                        I/O 口的第 2 位
        ANI     FBH
        OUT     2
Ⓐ       IN      1       ASCII 字符送入累加器。
Ⓑ       ANI     7 FH    把高位屏蔽掉。
Ⓒ       SUI     20 H    减 20 H
Ⓓ       LXI     H.1 NDX 变址表基地址送入 H. L 寄存器
        ADD     L       累加器内容和 H. L 寄存器内容相加
        MOV     L, A
Ⓔ       MOV     A, M    变址值送入累加器
Ⓕ       ADD     A       乘 2
Ⓖ       LXI     H, DEY  时间延迟表基地址送入 H. L
        ADD     L       累加器内容和 H. L 寄存器内容相加
        MOV     L, A
Ⓗ       MOV     E, M    取时间延迟常数送入 D. E 寄存器
        INX     H
        MOV     D, M
LOP 4   DCX     D       执行长时间延迟
Ⓘ       MOV     A, D
        ORA     E
        JNZ     LOP 4
        IN      2       在时间延迟结束时，将"1"输出至 I/O 口 2 的第 2
        ORI     4       位(打印锤脉冲高电平)
        OUT     2
```

执行 3 毫秒的印字轮释放时间延迟

现在，执行了一个 3 毫秒的"印字轮释放"时间延迟，而这个时间延迟的结束是由"印字轮释放"信号输出高电平作为标记的。接着执行 2 毫秒的"印字轮准备"时间延迟：

```
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
        LXI     D,F7H     ;LOAD TIME CONSTANT INTO D,E
LOP5    DCR     D         ;DECREMENT D,E
        MOV     A,D       ;TEST FOR 0 IN D,E
        ORA     E
        JNZ     LOP5      ;REDECREMENT IF NOT 0
;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
;PW REL HIGH
        IN      2         ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     1         ;SET BIT 0 TO 1
        OUT     2         ;OUTPUT RESULT
;EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD     MVI     A,0       ;LOAD ACCUMULATOR WITH 0
LOP6    DCR     A         ;DECREMENT A
        JNZ     LOP6      ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
```




执行 3 毫秒印字轮释放时间延迟

```
        LXI   D. F 7 H   取时间常数送入D. E 寄存器
LOP 5   DCR   D          D. E 减 1
        MOV   A. D       测试 D. E是否为零
        ORA   E
        JNZ   LOP 5      如不为零再减 1
```

将“1”输出至 I/O 口 2 的第 0 位，从而置“印字轮释放”为高电平

```
IN      2   I/O 口 2 的内容送入累加器
ORI     1   使第 0 位置“1”
OUT     2   输出结果
```

执行 2 毫秒印字轮准备时间延迟

```
PRD     MVI   A. O      累加器清零
LOP 6   DCR   A          A减 1
        JNZ   LOP 6      如A还未减到零，再减 1
```

在输出高电平的“印字轮准备”(CHRDY)从而结束打印周期之前，程序必须保证色带没有用完。如果检测到“色带用完”是低电平，程序就停留在重复循环中，直到更换了色带为止。此时由外部逻辑电路输出高电平的“色带用完”信号。

当检测到“色带用完”是高电平时，程序中的最后几条指令将“印字轮准备”置为高电平，然后返回到程序的开始，等待下一个打印周期。

### 4-3-6 程序逻辑错误

在本章所讨论的程序中，包含了一个在数字逻辑实现的方案中不会产生的逻辑错误。这个错误发生在打印锤脉冲时间延迟的计算中。

在数字逻辑实现的方案中，任何字符的 ASCII 代码都被当成七个单独的信号来处理。这些信号用某种方式组合产生时间延迟信号 H1 到 H6 中的某一个。不管输入的是 ASCII 码的哪种形式的组合，时间延迟信号 H1 到 H6 中总有一个将输出高电平。如果信号生成逻辑有缺陷，则时间延迟信号也仍然能够形成，但它可能是错误的信号。

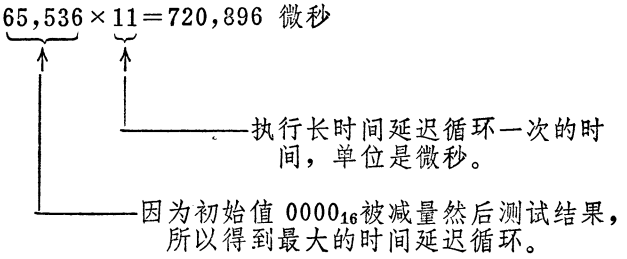
**范围检查**

现在来看汇编语言程序的实现。查看附录 A 的表是非常简单的，并且可以看到有效的 ASCII 码所占用的范围仅仅从 $20_{16}$ 到 $7A_{16}$。但是为了不妨碍逻辑设计者使用微型计算机系统，我们建立了一个专用系统，其中包括一些在正常的 ASCII 码范围以外的代码所代表的不常用的字符。在此情况下，程序可能输出某些很奇怪的结果。假定采用 ASCII 码 $10_{16}$ 表示一个专用字符。那么我们查找变址表就将把存储器字节 n $-10_{16}$ 中的内容装入累加器中。

这里并没告诉我们这个存储器字节中存入的是什么。这一

字节很可能是用来存储指令代码，也许是一个“二——十六”进制的数字值。假定它存放的是 $2A_{16}$，则下一程序步将做 $2A_{16}$ 乘2，再将结果与时间延迟表的基地址相加，并且从存储单元 $m+54_{16}$ 中取出起始时间延迟代码。

因为我们毫无保留地使用了简单的片选逻辑，所以根据图4-2所示的微型计算机组态，这个存储单元的内容很可能成为错误地访问某个存储器字节的一些双倍地址中的一个。倘若我们使用了较复杂的片选逻辑的话，那么现在就有机会使我们去访问一个根本不存在的存储器字节。在前一情况下没有告诉我们产生的是哪一种长度的打印锤脉冲；而在后一情况下则产生非常长的打印锤脉冲，因为我们从根本不存在的存储单元中得到的是“0”，而且把这一零值作为长时间延迟程序循环的初始时间延迟常数。这一打印锤脉冲的长度将是720毫秒。




**现在为了避免上面的难题，有两种方案可供选择：**

**1）程序逻辑可以简单地忽略任何无效的 ASCII 码。**

**2）对于无效的 ASCII 码，程序逻辑产生一个缺省打印锤脉冲宽度。**

如果忽略特殊字符，结论是明显的，微型计算机系统不能在一些要求打印特殊字符的场合中使用。因如果忽略特殊字符，则当输入时测试出这样的字符码时不会有打印锤脉冲，托

架不移动，也不会有定位脉冲。

对于特殊字符提供一个缺省打印锤脉冲，意味着这样的字符将被打印出来，但是它们在打印文本上的密度将是不均匀的。

作为一名逻辑设计人员应该事先做出决断。

下列两种指令序列中的任何一个序列都可以插入到现行程序中去：

```
                OUT     2
                IN      1       ;INPUT ASCII CHARACTER TO ACCUMULATOR
Check for  ---> ANI     7FH     ;MASK OUT HIGH ORDER BIT
valid ASCII     SUI     20H     ;SUBTRACT 20H
codes inserted  LXI     H,INDX  ;LOAD INDEX TABLE BASE ADDRESS TO HL
here            ADD     L       ;ADD ACCUMULATOR CONTENTS TO HL
                MOV     L,A
                MOV     A,M     ;LOAD INDEX INTO ACCUMULATOR
                ADD     A       ;MULTIPLY BY 2
```

```
                —
                —
                —
                OUT     2
                IN      1       ASCII 字符送入累加器，
                ANI     7FH     把高位屏蔽掉
在这里插入  ---->
测试是否为      SUI     20H     减去 20 H
                LXI     H,INDX  变址基地址送入 HL 寄存器
有效的ASCII     ADD     L       累加器的内容和 HL 寄存器相加
码的指令序列    MOV     L，A
                MOV     A，M    变址值送入累加器
                ADD     A       乘 2
```

下面是不考虑非标准的 ASCII 码的指令序列

```
        IN              ;INPUT ASCII CHARACTER TO ACCUMULATOR
        ANI     7FH     ;MASK OUT HIGH ORDER BIT
COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
        CPI     20H
        JM      PRD     ;IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING
;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CPI     7AH
        JP      PRD     ;IF CODE IS 7BH OR GREATER, BYPASS HAMMER FIRING
;ASCII CODE IS VALID
        SUI     20H     ;SUBTRACT 20H
        LXI     H,INDEX ;LOAD INDEX TABLE BASE ADDRESS TO HL
```

—

—

—

IN　1 ASCII 字符送入累加器

ANI　7FH　把高位屏蔽掉

ASCII 码同最低的规定值比较

CPI　20H

JM　PRD　如果代码为 1FH 或小于 1FH，跳过打印锤启动指令序列

ASCII 码同最高的规定值相比较

CPI　7 AH

JP　PRD　如果代码为是 7 BH 或大于 7 BH，跳过打印锤启动指令序列

ASCII 码是有效的

SUI　20 H　　减 20 H

LXI　H,INDEX　变址表基地址送入 HL 寄存器

—

—

—

第二种选择如下所示，用中等密度(即采用密度码 3)打印未知字符：

```
        IN      1       ;INPUT ASCII CHARACTER TO ACCUMULATOR
        ANI     7FH     ;MASK OUT HIGH ORDER BIT
;COMPARE ASCII CODE WITH SMALLEST LEGAL VALUE
        CPI     1FH
        JP      OK      ;IF CODE IS 20H OR MORE, TEST FOR HIGH LIMIT
;CODE IS ILLEGAL, ASSUME A DENSITY OF 3
NOK     MVI     A,6     ;LOAD TWICE DENSITY
        JMP     NEXT
;COMPARE ASCII CODE WITH LARGEST LEGAL VALUE
OK      CPI     7AH
        JP      NOK     ;IF CODE IS 7BH OR GREATER, ASSUME DENSITY OF 3
;ASCII CODE IS VALID
        SUI     20H     ;SUBTRACT 20H
        LXI     H,INDEX ;LOAD INDEX TABLE BASE ADDRESS INTO HL
        ADD     L       ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
        MOV     A,M     ;LOAD INDEX INTO ACCUMULATOR
        ADD     A       ;MULTIPLY BY 2
NEXT    LXI     H,DELY  ;LOAD DELAY TABLE BASE ADDRESS INTO HL
```

IN 1 ASCII 字符送入累加器

ANI 7FH 把高位屏蔽掉

ASCII 码同最低的规定值相比较

CPI IFH

JP OK 如果代码是 20 H或大于 20 H，测试上限

此代码是非标准的，假定密度为 3

NOK MVI A，6 装入双倍密度

JMP NEXT

ASCII 码同最高的规定值相比较

OK CPI 7 AH

JP NOK 如果代码是 7 BH 或大于 7 BH，假定密度为 3

ASCII 码是有效的

SUI 20 H 减去 20 H

LXI H,INDEX 把变址表基地址送入 HL 寄存器

ADD L 累加器内容与 HL 相加

MOV L，A

```
        MOV   A, M      把时间延迟值送入累加器
        ADD   A         乘 2
NEXT    LXI   H, DELY   把时间延迟表基地址送入 HL寄存器
        —
        —
        —
```

**比较立即数条件转移**

在解决这个问题的过程中，这两种无效ASCII码指令序列都很简单。

唯一的新特点是应用立即数比较(CPI)指令。这条指令是从累加器的内容减去操作数中的立即数据。相减的结果被丢掉了，这就是说累加器的内容不变。然而，状态标志位的状态却反映出相减的结果。我们利用JM[Jump Minus(若负则转移)]指令去识别负的结果，这个结果意味着操作数中的立即数比累加器的值大。同样，可以用JP[Jump on positive(若正则转移)]指令去识别一个小于或等于累加器中内容的立即操作数。

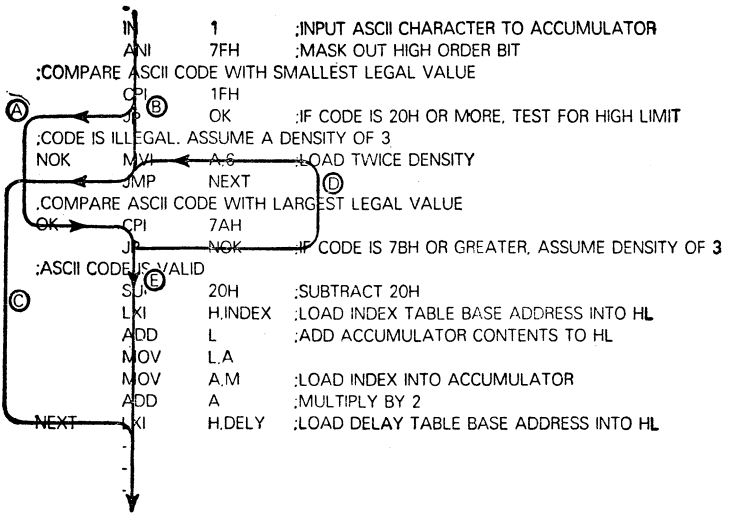
**条件指令执行路线**

在第二种指令序列中，如果在立即操作数内的值小于或等于累加器的内容，则由JP指令去执行下面的标号为OK的指令。作为第二种指令序列实际的程序执行路线，对于刚刚接触程序设计的人来说，可能会搞不清楚。因此将此程序的执行路线用图表示如下页图所示。

对于图中由带园圈的字母所表示的执行路线可以解释如下：

Ⓐ ASCII码已通过“最低规定值”测试，但现在还必须通过“最高规定值”的测试。

Ⓑ 如果该ASCII码没有通过”最低规定值“的测试。程序将缺省密度乘2送入累加器，并转移到访问相应于这个缺省




IN 1　　ASCII 字符送入累加器

ANI 7 FH　把高位屏蔽掉

ASCII 码同最小的规定值相比较

CPI IFH

Ⓑ

Ⓐ　JP CK　如果代码是 20 H或大于 20 H，测试上限

此代码为非标准码，假定密度为 3

NOK MVI A，6　装入双倍密度

JMP NEXT Ⓓ

ASCII 码同最大的规定值相比较

OK CPI 7 AH

JP NOK　如果代码是 7 BH 或大于 7 BH，假定密度为 3

ASCII 码是有效的

Ⓔ

SUI 20 H　减 20 H

Ⓒ　LXI H,INDEX　变址表基地址送入 HL 中

ADD L　累加器内容与 HL 相加

```
        MOV   L, A
        MOV   A, M      把变址值送入累加器
        ADD   A         乘 2
NEXT    LXI   H,DELY    把时间延迟表基地址送入 HL 中
```

密度的时间延迟常数的指令序列。这个转移如Ⓒ所示。

Ⓒ 一个通过了"最低规定 ASCII 码值"测试的字符，继续用"最高规定 ASCII 码值"来测试。如果测试结果是否定的，则程序执行转移，如Ⓓ所示，转移到假定缺省密度为 3 的指令上，实际上Ⓓ同Ⓑ汇合起来了。

Ⓓ ASCII 字符既通过"最低规定值"也通过"最高规定值"两个测试所通过的指令路线为Ⓔ，在这一路线上的指令将适当的密度变址值送入累加器。

### 4-3-7 复位和初始化

**为了完善我们的程序，必须有必要的复位和初始化指令。**

当输入到微型计算机系统内的"复位"信号为"真"时，开始执行复位指令。每当系统启动时执行初始化指令。

**没有什么理由一定要使复位指令和初始化指令相一致。**在许多应用场合下，可能需要两种分开的和不同的指令序列。**另一方面，利用复位指令来进行系统的初始化也是十分普遍的。**这就是说，当你首次对系统上电时，"复位"脉冲为"真"，而这个脉冲使整个微型计算机化逻辑系统启动起来。

在我们的情况下复位程序确实是简单的。我们所需要做的全部工作就是向 I/O 口 3 输出一个控制命令。该口是 8255 可编程序外部接口的控制口。然后将输出信号置成"两次打印周期之间"的状态。控制针对适当的 I/O 口选择方式 0。下面是需要的初始化指令序列：

```
        ORG     0
;SYSTEM RESET AND INITIALIZATION
;OUTPUT CONTROL CODE TO 8255 PROGRAMMABLE
;PERIPHERAL INTERFACE
        MVI     A17
        OUT     3
;SET HAMMER PULSE, PW READY AND
;PW REL HIGH. SET START RIBBON MOTION LOW
        MVI     9AH
        OUT     2
```

ORG 0

系统复位和初始化

输出控制码到 8255 可编程

外部接口

MVI A 17

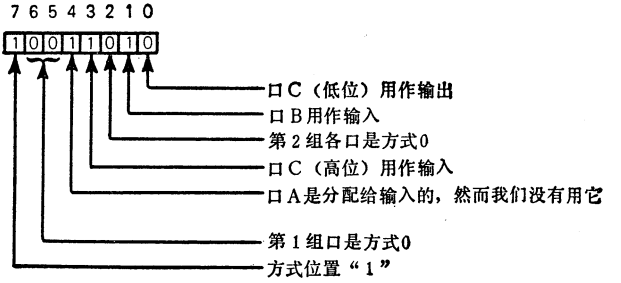
OUT 3

置“打印锤脉冲”、“印字轮准备”和“印字轮释放”为高电平，置“启动色带移动”为低电平。

MVI 9 AH

OUT 2

下面示出的是 8255 PPI 控制代码的格式：




## 4-4 程 序 小 结

首先，将本章所讨论的全部程序列写在一起是一个好方法。

```
INDEX   EQU     0380H   ;EQUATE INDEX TABLE BASE ADDRESS
DELY    EQU     03EEH   ;EQUATE DELAY TABLE BASE ADDRESS - 2
        ORG     0
;SYSTEM RESET AND INITIALIZATION
;INITIALLY SET HAMMER PULSE, PW READY AND
;PW REL HIGH, SET START RIBBON MOTION LOW
        MVI     A7
        OUT     2
;OUTPUT CONTROL CODE TO 8255 PROGRAMMABLE
;PERIPHERAL INTERFACE
        MVI     9AH
        OUT     3
;PRINT CYCLE PROGRAM
;IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O
;PORT 2 FOR A 0 VALUE)
START   IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     20H     ;ISOLATE BIT 5
        JNZ     START   ;IF NOT 0, RETURN TO START
;INITIALIZE PRINT CYCLE, OUTPUT 0 TO BITS 0
;AND 1 OF I/O PORT 2, OUTPUT 1 TO BIT 3 OF
;I/O PORT 2
        MVI     A,0CH   ;LOAD MASK INTO ACCUMULATOR
        OUT     2       ;OUTPUT TO I/O PORT 2
;OUTPUT 0 TO BIT 3 OF I/O PORT 2, THIS COMPLETES
;START RIBBON MOTION PULSE
        MVI     A,4     ;LOAD MASK INTO ACCUMULATOR
        OUT     2       ;OUTPUT TO I/O PORT 2
;TEST FOR END OF PRINTWHEEL POSITIONING
LOP1    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     20H     ;ISOLATE BIT 5
        JZ      LOP1    ;IF 0 RETURN TO LOP1
;EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
        MVI     A,0     ;LOAD ACCUMULATOR WITH 0
LOP2    DCR     A       ;DECREMENT A
        JNZ     LOP2    ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
;TEST PRINTHAMMER FIRING CONDITIONS
LOP3    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        RLC             ;MOVE BIT 7 INTO CARRY
        JNC     PRD     ;IF CARRY IS 0, BYPASS PRINTHAMMER FIRING
        ANI     20H     ;ISOLATE BIT 4 WHICH IS NOW BIT 5
        JZ      LOP3    ;WAIT FOR NONZERO VALUE BEFORE FIRING
;FIRE PRINTHAMMER
        IN      2       ;SET HAMMER PULSE LOW, OUTPUT 0
        ANI     FBH     ;TO BIT 2 OF I/O PORT 2
        OUT     2
        IN      1       ;INPUT ASCII CHARACTER TO ACCUMULATOR
        ANI     7FH     ;MASK OUT HIGH ORDER BIT
;COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
        CPI     20H
        JM      PRD     ;IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING
;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CPI     7AH
        JP      PRD     ;IF CODE IS 7BH OR GREATER, BYPASS HAMMER FIRING
```

```
ASCII CODE IS VALID
        SUI     20H     ;SUBTRACT 20H
        LXI     H,INDEX ;LOAD INDEX TABLE BASE ADDRESS TO HL
        ADD     L       ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
        MOV     A,M     ;LOAD INDEX INTO ACCUMULATOR
        ADD     A       ;MULTIPLY BY 2
        LXI     H,DELY  ;LOAD DELAY TABLE BASE ADDRESS INTO HL
        ADD     L       ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
        MOV     E,M     ;LOAD DELAY CONSTANT INTO D,E
        INX     H
        MOV     D,M
LOP4    DCX     D       ;EXECUTE LONG DELAY
        MOV     A,D
        ORA     E
        JNZ     LOP4
        IN      2       ;AT END OF DELAY OUTPUT 1 TO BIT 2
        ORI     4       ;OF I/O PORT 2 (HAMMER PULSE HIGH)
        OUT     2
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
        LXI     D,F7H   ;LOAD TIME CONSTANT INTO D,E
LOP5    DCR     D       ;DECREMENT D,E
        MOV     A,D     ;TEST FOR 0 IN D,E
        ORA     E
        JNZ     LOP5    ;REDECREMENT IF NOT 0
                        ;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
                        ;PW REL HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     1       ;SET BIT 0 TO 1
        OUT     2       ;OUTPUT RESULT
;EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD     MVI     A,0     ;LOAD ACCUMULATOR WITH 0
LOP6    DCR     A       ;DECREMENT A
        JNZ     LOP6    ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
                        ;TEST FOR EOR DET (BIT 6 OF I/O PORT 2) EQUAL
                        ;TO 0 AS A PREREQUISITE FOR ENDING THE PRINT CYCLE
LOP7    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     40H     ;ISOLATE BIT 6
        JZ      LOP7    ;RETURN AND RETEST IF NOT 0
                        ;AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT 2 TO 1
                        ;THIS SETS CH RDY HIGH
        IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     2       ;SET BIT 1 TO 1
        OUT     2       ;OUTPUT RESULT
        JMP     START   ;JUMP TO NEW PRINT CYCLE TEST
;INDEX TABLE FOLLOWS HERE
        ORG     0380H
        Data represented by 90 index entries follow here. Data appears in mnemonic field,
        one byte per line.
;DELAYS TABLE FOLLOWS HERE
        ORG     03F0H
        Data representing 6 delays follow here. Data appears in mnemonic field, two bytes
        per line.
```

图 4-6 简单打印周期程序

```
INDEX     EQU  038OH  等于变址表的基地址
DELY      EQU  03 EEH  等于时间延迟表基地址－2
          ORG  0
```

系统复位和初始化

先将“打印锤脉冲”，“印字轮准备”和“印字轮释放”置为高电平。将“启动色带移动”置为低电平

```
          MVI  A 7
          OUT  2
```

将控制码输到 8255 可编程外部接口

```
          MVI  9 AH
          OUT  3
```

打印周期程序

在两次打印周期之间测试 FFI(I/O 口 2 的第 5 位是否为“0”)

```
START     IN   2        I/O 口 2 的内容送入累加器
          ANI  20 H     抽出第 5 位
          JNZ  START    如不为“0”，返回到START
```

打印周期初始化，将“0”输出至 I/O 口 2 的第 0 位，将“1”输出至 I/O 口 2 的第 3 位

```
          MVI  A,OCH    屏蔽值送入累加器
          OUT  2        输出到 I/O 口 2
```

将“0”输出至 I/O 口 2 的第 3 位，这就完成了“启动色带移动”脉冲

```
          MVI  A，4 屏蔽值送入累加器
          OUT  2    输出到 I/O 口 2
```

测试印字轮定位是否结束

```
LOP1      IN   2        I/O 口 2 的内容送入累加器
          ANI  2 0H     抽出第 5 位
          JZ   LOP1     如不为“0”，返回到 LOP1
```

执行 2 毫秒的印字轮稳定时间延迟

```
          MVI  A，O     累加器清零
LOP2      DCR  A        A 减 1
          JNZ  LOP 2    如 A 还没减到零，再减量
```

测试打印锤启动条件

```
LOP3    IN    2        I/O 口 2 内容送入累加器
        RLC            第 7 位移入进位位
        JNC   PRD      如进位为"0",跳过打印锤启动程序
        ANI   20H      抽出第 4 位，即当前的第 5 位
        JZ    LOP3     启动前等待非"0"值
启动打印锤
        IN    2        置打印锤脉冲为低电平，将
        ANI   FBH      "0"输出至 I/O 口 2 的第 2 位
              OUT   2
              IN    1        ASCII 字符送入累加器
        ANI   7FH      屏蔽高位值
比较 ASCII 码和最低规定值
        CPI   20H
        JM    PRD      如该码是 1FH或小于 1FH,跳过打印锤启动
                       程序段
比较 ASCII 码和最高规定值
        CPI   7AH
        JP    PRD如该码是7BH 或大于7BH,跳过打印锤启动程序段
ASCII 码有效
        SUI   20H          减20H
        LXI   H, INDEX     变址表基地址送入 H.L 内
        ADD   L            累加器内容和 H. L 内容相加
        MOV   L. A
        MOV   A. M         变址值送入累加器
        ADD   A            乘 2
        LXI   H, DELY      时间延迟表基地址送入 H. L 内
        ADD   L            累加器内容和 H. L 内容相加
        MOV   L. A
        MOV   E. M         把时间延迟常数送入 DE 寄存器
        INX   H
        MOV   D, M
LOP4    DCX   D        执行长时间延迟
```

```
        MOV  A, D
        ORA  E
        JNZ  LOP4
        IN   2          时间延迟结束时,将"1"输出至I/O口
        ORI  4          2的第2位（打印锤脉冲为高电平）
        OUT  2
```

执行3毫秒的印字轮释放时间延迟

```
        LXI  D, F7H  时间常数送入D、E寄存器
LOP5    DCR  D       D、E内容减1
        MOV  A, D    测试D、E是否为零
        ORA  E
        JNZ  LOP5    如不为零，再减1
                     将"1"输出至I/O口2的第0位，
                     从而将"印字轮释放"置为高电平
        IN   2       I/O口2内容送入累加器
        ORI  1       使第0位置"1"
        OUT  2       输出结果
```

执行2毫秒"印字轮准备"时间延迟

```
PRD     MVI  A, O    累加器清"0"
LOP6    DCR  A       A减1
        JNZ  LOP6    如A未减到零，再减1
                     测试"色带用完"(I/O口2的第6位)是否
                     等于零，作为打印周期结束的先决条件
LOP7    IN   2       I/O口2内容送入累加器
        ANI  40H     抽出第6位
        JZ   LOP7    如不为零，返回重新测试
                     在打印周期结束时，使I/O口2的第1位
                     置"1",从而将"CH, RDY"置为高电平
        IN   2       I/O口2内容送入累加器
        ORI  2       使第1位置"1"
        OUT  2       输出结果
        JMP  START   转到测试新的打印周期
```

```
变址表接在这里
        ORG  038OH
        90个变址入口的数据接在这里。数据出现在助记符字段，每行一字节。
时间延迟表接在这里
        ORG  03FOH
        代表六种时间延迟的数据接在这里。数据出现在助记符字段，每行两字节。
```

图 4-6　简单打印周期程序

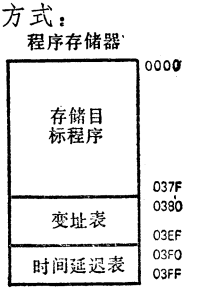
我们把必须的汇编命令包括进去了。最后得到的程序示于图4-6。

现在，这一程序全部讲完；应该看到，我们还没有使用随机存取存储器。CPU 寄存器已提供足够容量的读/写寄存器来处理所有的可变数据了。

1 K字节的只读程序存储器已经能容纳全部的程序再加上两个数据表。

如果你是在本章所囊括的逻辑的范围内来实现微型计算机系统的话，那么现在你还可以节省两个随机存取存储器器件。在所有可能的情况下，在微型计算机系统中可能包含有大量其它逻辑功能是更经济的，而且这些功能几乎无疑需要一些随机存取存储器。CPU 的七个寄存器和栈指示器提供了读/写存储器的九个字节。对于任何实际应用来说，这些通常是不够的。

下面示出了最后的程序存储分配。图中标示出图 4-6 中的程序使用 ROM 存储器的方式：



# 第五章　程序人员的看法

与第三章的数字模拟相比较，在第四章中介绍的程序相当简短。而且较易于学习。虽然我们在第四章中已经讨论了很长的篇幅，但仍然需要讨论下去。图 4-6 中的程序所处理的是执行简单传递函数的逻辑，但是它不是一个编得很好的程序。

对于数字逻辑设计者来说，关于程序设计的最令人困惑的事情是能够轻而易举地用十种不同的方法来做同一件事情。这是否意味着有一些执行过程要比另一些执行过程更有效一些呢？确实是这样，正像建立高效率的数字逻辑是一种技巧一样，对于编写出高效率的程序在很大程度上也是一种技巧。但是总还有一些规律。如果遵循这些规律，至少可以帮助您避免发生明显的错误。在这一章里我们将采用在第四章中编写的程序，并作更详细的探讨。

## 5-1　简单程序设计的效率

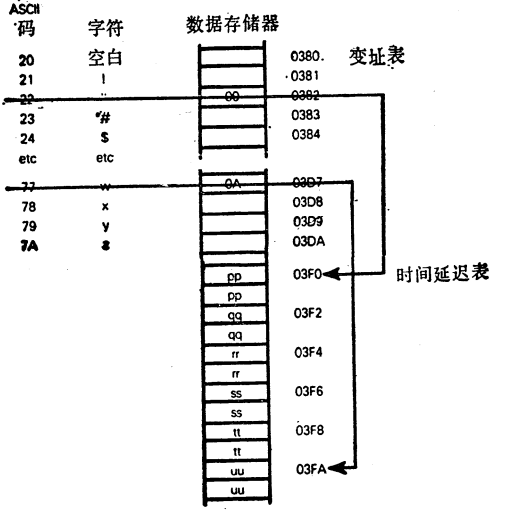
写完一个源程序之后你应该做的第一件事情是返回来审查一遍，看一看能否用一些基本方法去掉几条指令。

### 5-1-1　高效率的查表方法

通常情况下你将会发现，仅仅由于编写出比较高效率的指令序列，就可能将一段程序缩简为它原来长度的三分之二。在图 4-6 中有一个包括变址表在内的最明显的粗糙的程序设计的例子。

这个程序是从变址表字节中取出一个 1 至 6 之间的数值，

然后再将它与时间延迟表的基地址相加以前，先把这个值乘2。为什么不直接把变址值的2倍存储在变址表中呢？因为这样做能够省下一条指令，如下图所示：




```
;ASCII CODE IS VALID
        SUI     20H         ;SUBTRACT 20H
        LXI     H,INDEX     ;LOAD INDEX TABLE BASE ADDRESS
        ADD     L           ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
        MOV     A,M         ;LOAD INDEX X2 INTO ACCUMULATOR
        LXI     H,DELY      ;LOAD DELAY TABLE BASE ADDRESS INTO HL
        ADD     L           ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
```

ADD instruction dropped

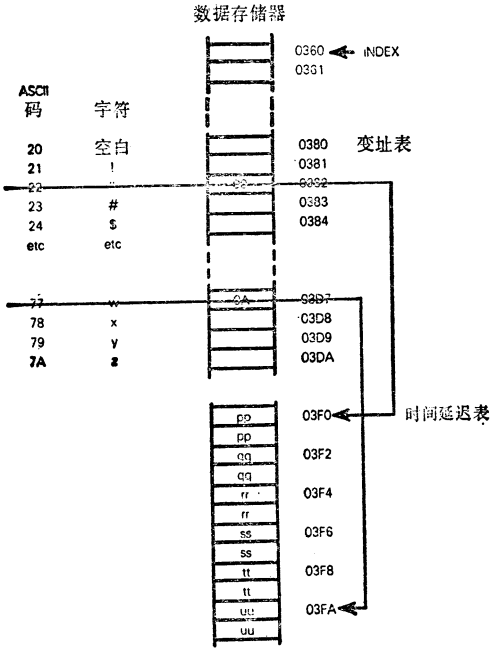
ASCII码有效

```
SUI     20 H        减 20 H
LXI     H, INDEX    装入变址表的基地址
ADD     L           累加器内容与 HL 相加
MOV     L, A
```

```
MOV  A, M   取变址值 X2 装入累加器
        LXI  H, DELY   时间延迟表基地址送入 HL
        ADD L          累加器内容和 HL 相加
        MOV L, A
ADD指令 —
被删去了。 —
        —
```

在上述的指令序列中，可以看到在加网线的 MOV 指令下边的一条指令已经被删去了。

还有许多方法能够使得查时间延迟表的效率更高。为什么要从 ASCII 码减去 $20_{16}$呢？譬如，倘若我们把 ASCII 码直接与基地址相加，这并不会阻止我们将一个实际的比变址表字节小



```
INDEX   EQU     0360H       EQUATE INDEX TO TABLE BASE ADDRESS -20H

        ;ASCII CODE IS VALID
        LXI     H,INDEX     ;LOAD INDEX TABLE BASE ADDRESS -20H
SUI     ADD     L           ;ADD ACCUMULATOR CONTENTS TO HL
instruction
dropped MOV     L,A
        MOV     A,M         ;LOAD INDEX X2 INTO ACCUMULATOR
        LXI     H,DELY      ;LOAD DELAY TABLE BASE ADDRESS INTO HL
        ADD     L           ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
```

```
INDEX  EQU  0360 H   使 INDEX 值等于表的基地址－20 H
        —
        —
        —
ASCII码有效
```

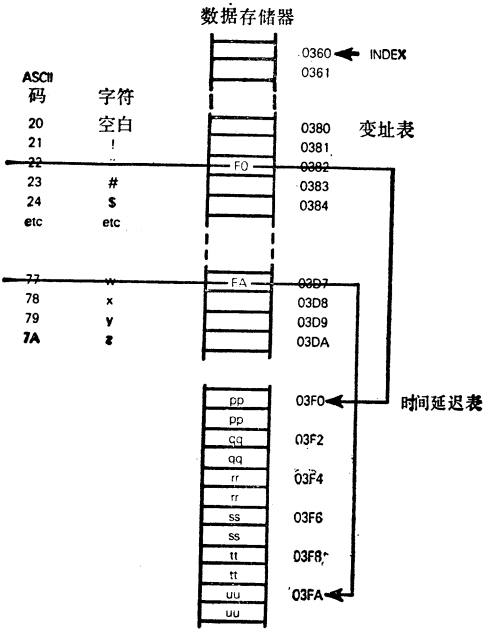
```
        LXI  H. INDEX   装入变址表基地址－20 H
```

```
SUI 指令被删去
        ADD  L          累加器的内容和 HL 相加
        MOV  L, A
        MOV  A,M        将INDEX×2 后送入累加器
        LXI  H,DELY     时间延迟表基地址送入 HL
        ADD  L          累加器内容和 HL 相加
        MOV  LA
```

$20_{16}$ 的值赋给由符号 INDEX 所表示的这一基地址。如上页图所示，现在在我们的指令序列内就可以进一步简化，如上页图所示：

好，现在正好使得 INDEX 值等于 0360—采用这样的方法后，我们不再需要从 ASCII 码中减去 $20_{16}$，我们已经删去用网线遮住的指令 LXI 的上一条指令 SUI。现在，与其存储变址表中的字符密度变址值的两倍，为什么不存储时间延迟表地址的

另一半呢？这样，我们的程序就进一步简略如下：




```
INDEX   EQU     0360H       ;EQUATE INDEX TO TABLE BASE ADDRESS -20H
        .
        .
        .
;ASCII CODE IS VALID
        LXI     H,INDEX     ;LOAD INDEX TABLE BASE ADDRESS -20H
        ADD     L           ;ADD ACCUMULATOR CONTENTS TO HL
        MOV     L,A
        MOV     L,M         ;LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS
        MVI     H,03H       ;LOAD HIGH ORDER BYTE OF DELAY TABLE ADDRESS
```

INDEX EQU 0360 H 使 INDEX 值等于表的基地址－20 H

—

ASCII码有效

```
LXI   H, INDEX   装入变址表基地址－20H
ADD   L          累加器内容和 HL 相加
MOV   L, A
```

```
MOV   L, M     取时间延迟表地址的低位字节
MVI   H, 03 H  取时间延迟表地址的高位字节
```

又有两条指令消失了。

现在我们已经取得了从存放启动打印锤起始时间延迟常数的指令序列中减掉四条指令的成果，然而，我们的工作仍然没有结束。

**把表安排得便于简化指令序列的存取**

为什么不把变址表整个移动一下，使之占用 $0320_{16}$ 到$037A_{16}$ 个而不是 $0380_{16}$到 03 $DA_{16}$ 个存储单元呢？被去掉奇偶校验位的 ASCII码现在变成变址表地址的低位字节，我们的指令序列进一步紧缩见下页图。

假定被打印的字符是“W”，在下页图的三条指令中的第一条被执行以前，由于上述两条指令

```
IN    1
ANI   7FH
```

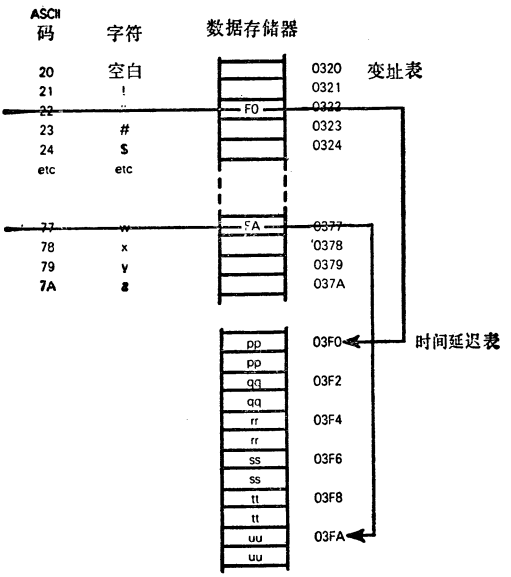
的执行，累加器的内容为 $77_{16}$，接着执行：

```
MVI   H, 03 H
```

此时寄存器H将存有 $03_{16}$，这是隐含的存储器地址的高位字节。下一条指令：

```
MOV   L, A
```

从累加器传送 $77_{16}$ 到寄存器L。现在寄存器H和L存有 $0377_{16}$，这就是有效地址。下一条指令：




```
:ASCII CODE IS VALID
        MVI     H,03H       ;LOAD INDEX TABLE ADDRESS, HIGH ORDER BYTE
        MOV     L,A         ;MOVE LOW ORDER BYTE OF ADDRESS TO L
        MOV     L,M         ;LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS
```

ASCII码有效

```
MVI   H, 03   装入变址表地址高位字节
MOV   L, A    地址的低位字节移入L
MOV   L, M    装入时间延迟表地址的低位字节
```

## MOV L, M

把由HL指向的存储器字节的内容传送到寄存器L。

HL存有$0377_{16}$，存储器字节$0377_{16}$中有$FA_{16}$，所以$FA_{16}$被传送到寄存器L。这意味着新的隐含地址是$03FA_{16}$，并且它就是所要求的时间延迟表的地址。

九条指令被减少成了三条，而所付出的唯一代价是我们把变址表搬到数据存储器的新的区域。

为了使你能理解现在的程序，我们把老的和新的指令序列一起并排表示在下面，而省略了注释部分：

| 老的程序 | | 新的程序 | |
|---|---|---|---|
| ASCII码有效 | | | |
| SUI | 20 H | MVI | H, 03 H |
| LXI | H, INDEX | MOV | L, A |
| ADD | L | MOV | L, M |
| MOV | L, A | | |
| MOV | A, M | | |
| ADD | A | | |
| LXI | H, DELY | | |
| ADD | L | | |
| MOV | L, A | | |

遗憾的是还没有可遵循的不变规律来保证我们永远编写出最短的程序。但当你已经编写出一些程序以后，你将懂得各条指令是如何工作的，这样也就会产生出效率来。前几页论述的目的是要说明一段紧凑的程序和一段直接编制的程序之间的差别是何等巨大。如果你所编制的程序是大量生产的话，就会使你不得不费一些时间和金钱来缩减程序的长度，从而有可能节省一些ROM芯片。

## 5-2 硬件的利用

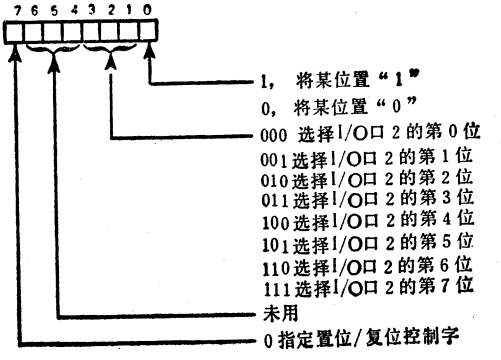
所有计算机的程序设计员都力求能编出高效率的汇编语言程序。但是也只有微型计算机的程序设计员才必须考虑硬件的利用情况，并使其能对提高程序设计效率有所贡献。

现在我们采用的是工作在方式 0 的 8255 可编程序外部接口。这是以最明显的方式访问接口中各位的。下面，我们来介绍另外一些方式。

## 5-2-1 硬件专用指令

**位置位/复位指令**

我们在过去的绝大部分时间里，一直是通过对于个别的置位和复位来形成输入和输出信号的。8255 PPI有一个控制 I/O 口，在我们所讨论的情况下，把它编为 I/O 口 3 。向这个口输出一个适当的控制字就能使 I/O 口 2 的个别位被置位或复位。以下是控制字的格式：




**与硬件相关的指令**

应当注意，这条置位/复位指令是与硬件有关的。它依赖于在 8255 PPI 内为实现指令所必须的逻辑。当读入一个输入信号的状态时，置位/复位控制指令并不能节省操作步。但是无论什么时候，当我们想要改变输出信号的状态时，这些控制指令都使三条指令变成两条指令。对于图 4-6 中所列出的整个程序，在这里作

了一些有效的修改：

| 老程序 | 新程序 |
| --- | --- |

启动打印锤，置打印锤脉冲为低电平，

将“0”输出至I/O口2的第二位

```
IN    2          MVI   A, 4
ANI   FBH        OUT   3
OUT   2          —
—                —
—                —
```

在延迟时间结束时将“1”输出至I/O口2的第2位(打印锤脉冲为高电平)

```
IN    2          MVI   A, 5
ORI   4          OUT   3
OUT   2          —
—                —
—                —
```

将“1”输出至I/O口2的第0位，从而使“印字轮释放”置高电平

```
IN    2          MVI   A, 1
ORI   1          OUT   3
OUT   2          —
—                —
```

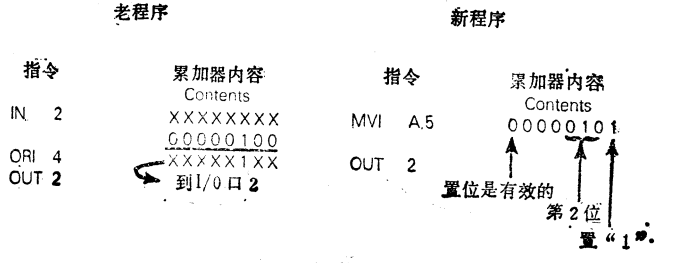
在打印周期结束时置I/O口2的第1位为“1”，从而置“CH RDY”为高电平。

```
IN    2          MVI   A, 3
ORI   2          OUT   3
OUT   2          JMP   START
```

JMP START

**置位/复位的说明**

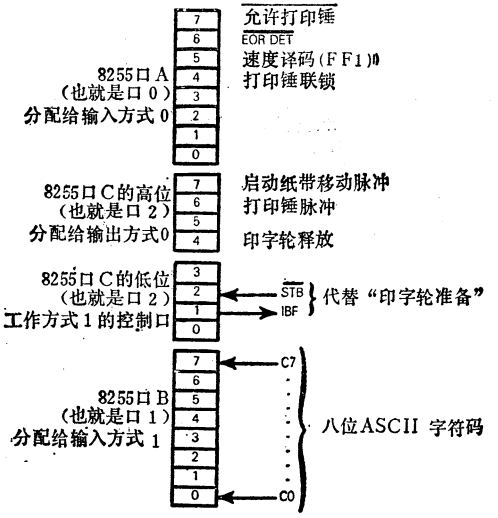
假如你对于置位和复位是怎样工作的仍有疑问的话，我们将用图示说明。这就要求将“1”输出至I/O口2的第2位：



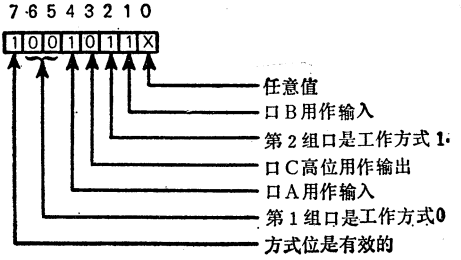

## 5-2-2 硬件特性的直接用途

我们现在假定外部逻辑用“印字轮准备”信号确保前面的代码被处理完以后，才会输入一个新的字符代码。在打印周期开始的时候，我们使用某些指令将“印字轮准备”信号置为低电平，然后在打印周期结束的时候，使它恢复高电平。

对于我们的微型计算机系统，如果没有一个关于外部逻辑的清楚的定义的话，就无法知道外部是否需要“印字轮释放”和“印字轮准备”输出信号，以便确定专门的时间延迟；或者，它们是否仅仅被用来保证我们在试图开始打印下一个字符以前完成前一个字符的打印。如果“印字轮准备”信号的作用仅仅是保证在老的字符被打印出来以前，新的字符不进入微型计算机系统，那么我们没有这一信号也一样能够工作，并且可以用处于方式1工作的I/O口1的选通输入来代替它。现在把我们的I/O口分配如下：

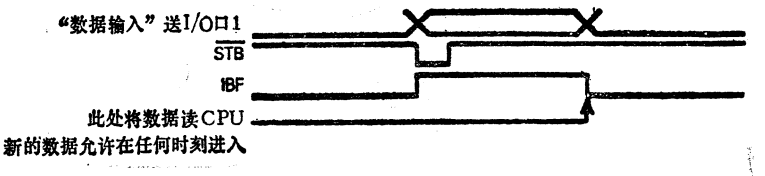



**PPI 控制代码** 8255 PPI控制的格式如下：




当外部逻辑在I/O口1输入一个新的字符时，它同时在I/O口2的第2个引线端上输入一个低电平的选通信号(STB)。在这一瞬间，8255 PPI将在I/O口2的第一个引线端上输出高电平的“IBF”脉冲。“IBF”将保持高电平直到读出I/O口1的内

容并进入CPU的那几条指令被执行完为止。在这一时刻，IBF将复位到低电平，这可以用图说明如下：




现在，在某一个打印周期里，打印周期逻辑仅仅把I/O口1的内容读出一次。处于方式1工作的I/O口是被缓冲的，因此所有外部逻辑仅仅是把“IBF”作为选通信号使用。只要“IBF”是低电平，外部逻辑就能自由地输入下一个ASCII字符。如果“IBF”是高电平，则外部逻辑就必须等待。这个方案消除了“印字轮准备”信号以及为处理这一信号所需要的任何指令。

应当注意，通过I/O口2的第0至第3位输出的数据信号已被移入到I/O口2的第7至第4位。被分配到I/O口2的第7至第4位的信号则被移入到I/O口0，这样的传送是很有理由的。大家总还记得，置位/复位仅仅控制口2。如果我们想要把改变一个信号的状态所要求的指令的数目从三条减少至两条的话，那么输出信号必须被配置到I/O口2的引线端上，因为我们不能减少抽样输入信号所要求的指令条数，所以我们也可以把这些信号传送到I/O口0。

## 5-3 子程序

如果你再看一下图4-6的程序，你将会发现，在这段程序中，有两处执行相同的指令序列来产生2毫秒的时间延迟。现在仅用三条指令来完成2毫秒的时间延迟。因此重复执行这

三条指令影响很大。然而如果你考虑到这一点的话，就会联想到在一般比较长的程序中，潜伏着某些对于存储器的很不经济的利用情况。

在第四章里我们使程序十分简单，因为它必须短到能够在一本书里加以讨论。然而，如果你愿意设想一个比较复杂的程序，例如其中有一段由30条指令组成的序列，而不是 3 条指令组成的序列需要重复执行的话，那么我们就必须寻找一种方法，使得包含这样的指令序列只有一次。每当需要的时候，可以从程序的一些不同单元地址上转移到这唯一的指令序列。利用子程序就能够达到这个目的。

让我们取出实现 2 毫秒时间延迟的三条指令，并把它改造成一般的子程序，下面就是这一程序有关部分所发生的变化：

```
        ORG     0
        LXI     H,08FFH     ;INITIALIZE STACK POINTER TO END OF
        SPHL                ;DATA AREA
        .
        .
        .
;EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
        CALL    D2MS
        .
        .
        .
;EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD     CALL    D2MS
        .
        .
        .
;AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT 2 TO 1
;THIS SETS CH RDY HIGH
        MVI     A,3         ;THESE ARE THE NEW INSTRUCTIONS
        OUT     3           ;TO SET A BIT OF I/O PORT 2
        JMP     START

;SUBROUTINE TO EXECUTE A 2 MILLISECOND DELAY
D2MS    MVI     A,0         ;LOAD ACCUMULATOR WITH 0
LOPD    DCR     A           ;DECREMENT A
        JNZ     LOPD        ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
        RET                 ;RETURN FROM SUBROUTINE
```

```
        ORG     0
        LXI     H,08 FFH    将栈指示器初始化
        SPHL                使之位于数据区的终点
        —
        —
        —
```

执行 2 毫秒的印字轮稳定时间延迟，

```
        CALL    D 2 MS
        —
        —
        —
```

执行 2 毫秒的印字轮准备时间延迟

```
PRD     CALL    D 2 MS
        —
        —
        —
```

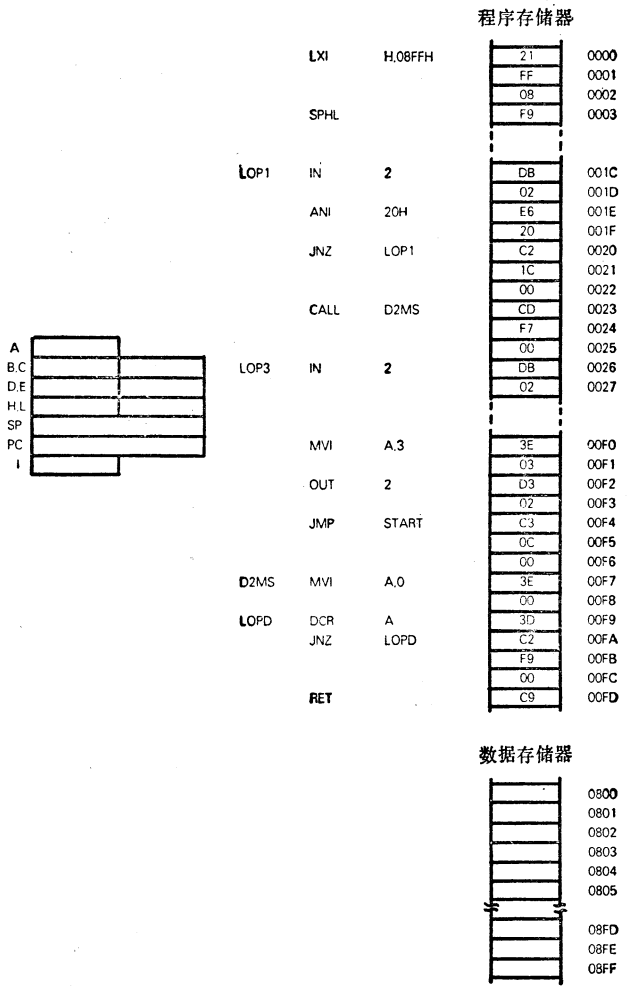
在打印周期的末尾，把 I/O 口 2 的第一位置成“1”；从而置“CH RDY”为高电平。

```
        MVI   A, 3      这是把 I/O 口 2 的一位置位的几条新指令
        OUT   3
        JMP   START
```

执行 2 毫秒时间延迟的子程序

```
D 2 MS  MVI   A, 0      把 0 送入累加器
LOPD    DCR   A         累加器减 1
        JNZ   LOPD      如果累加器没有减到零，继续减。
        RET             从子程序返回。
```

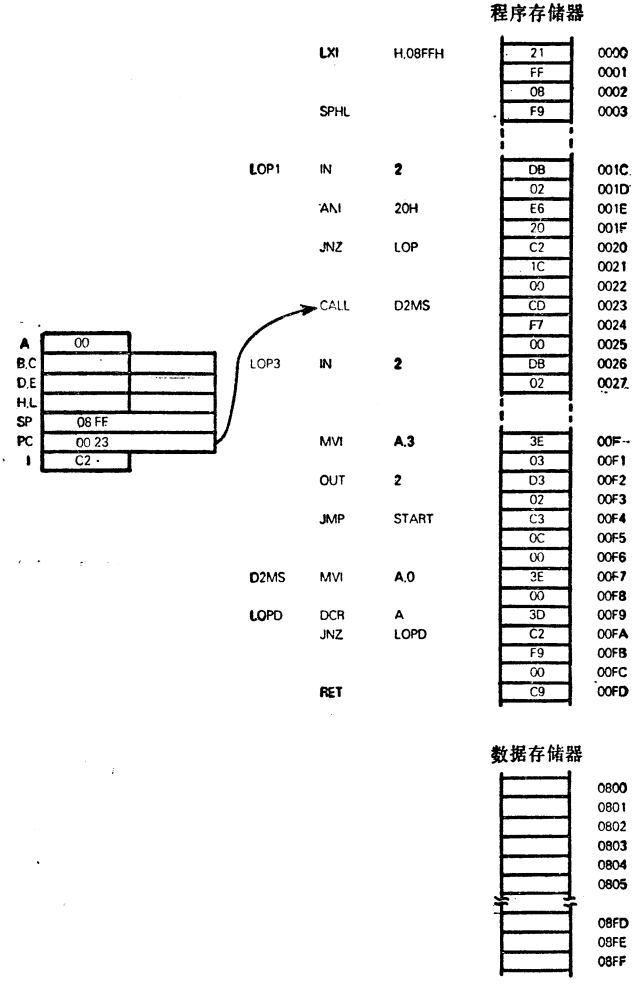
为了理解子程序是怎样工作的，我们将为我们的源程序的结果代码指定一些任意的存储器地址。我们将一步一步的说明，当一个子程序被调用时将发生什么情况以及从子程序返回期间又会发生什么情况？首先假定存储器分配如下：




## 5-3-1 子程序调用

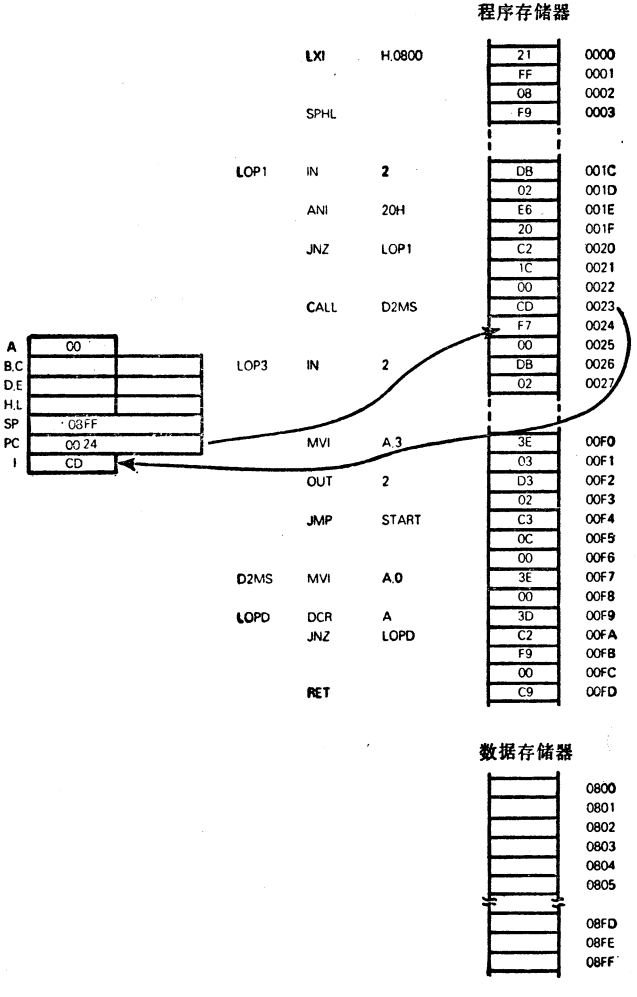
假定我们准备首先执行 CALL D2MS 指令，此时各寄存

器将保存下列数据：




程序计数器(PC)访问调用指令结果代码的第一个字节，

所用的地址是$0023_{16}$。指令寄存器保存着正在执行的指令的结果代码。它是存放在单元$0020_{16}$中的指令JNZ。你将会注意到，在程序开始时栈指示器已被初始化，其中存放着$08\,FF_{16}$。



由图 4-2 可见，这是读/写存储器的第一个字节的地址。因为栈还未使用，所以栈指示器将仍然存放 08 $FF_{16}$。

累加器内存放着 00，因为这是使得从 LOP 1 开始执行的持续循环中脱离出来的条件。

当执行调用指令时，所经历的步骤如下：

调用指令的结果代码存入指令寄存器，程序计数器增 1。

为了跳过 CALL 地址程序计数器增 2，增 2 后的值保存在头两个栈字节中。然后 CALL 地址才送入程序计数器。栈指示器减 2，使它访问浮在上面的第一个栈字节。

下一条被执行的指令的结果代码存放在存储器字节 00 $F7_{16}$ 中。它就是现在被程序计数器所访问的地址：

在 2 毫秒时间延迟循环中的各条指令现在被反复执行，直到累加器的内容从 01 减到 00 为止。要记住第一次累加器将从 00减到 $FF_{16}$，这就是为什么在以 LOPD 开始的两条指令的循环将被执行 256 次的缘故。

### 5-3-2 子程序的返回

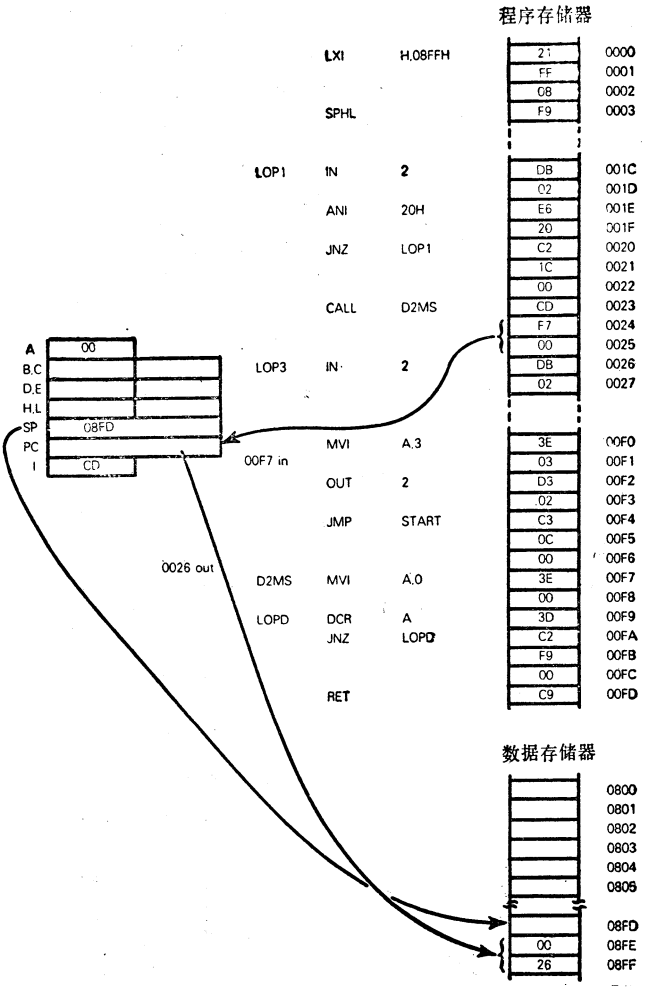
当累加器终于从 01 减到 00 的时候，就转而执行返回指令 (RET)。这条指令使栈指示器内容增 2，然后把栈顶的两个字节的内容移送到程序计数器。于是，程序返回到接在调用指令后边的一条指令：

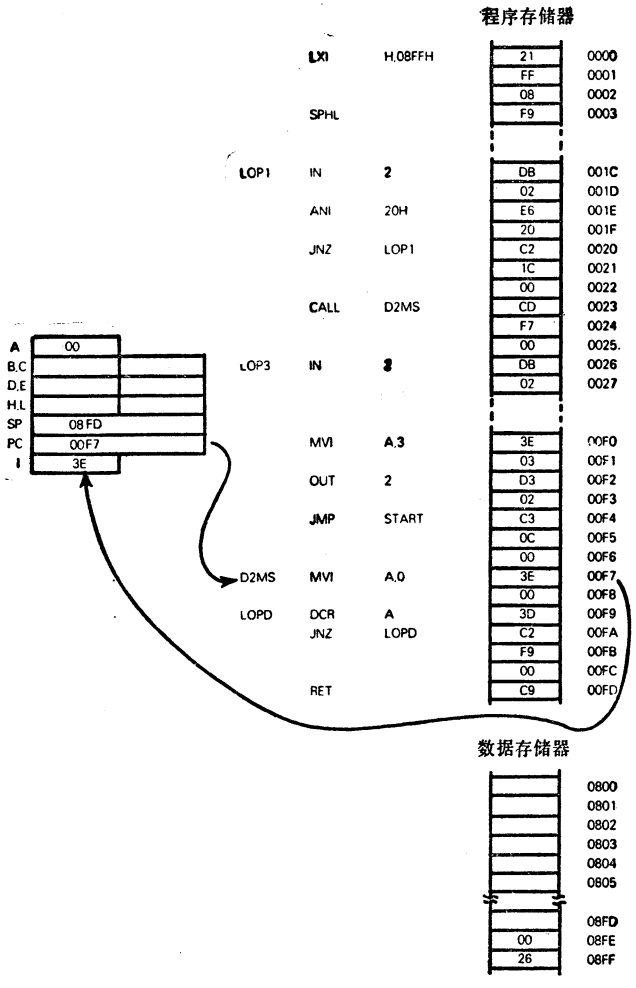
上面所发生的情况可以综述如下：

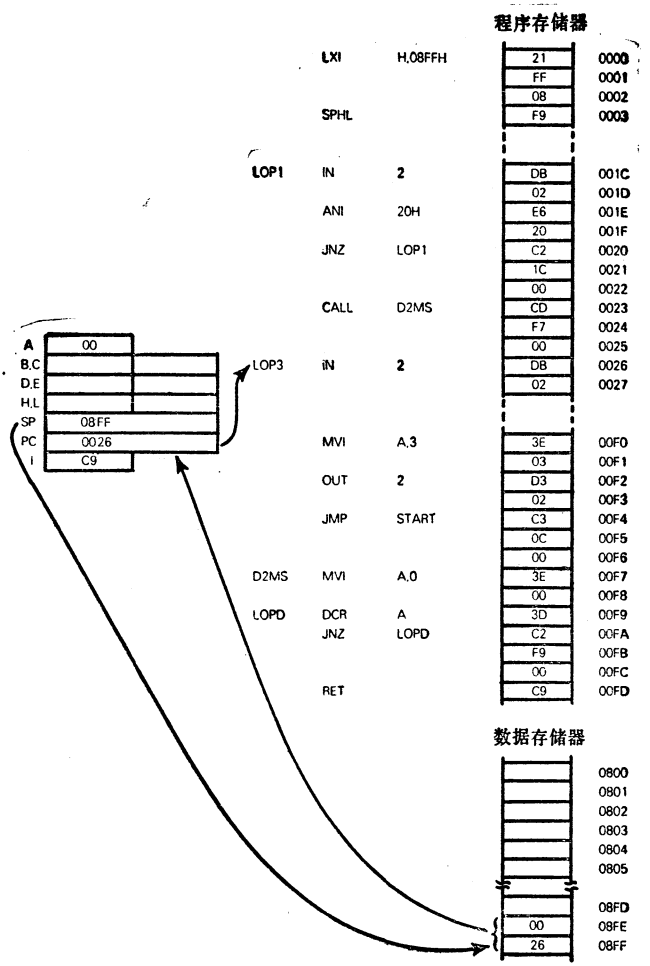
当调用指令被执行以后，从子程序返回的下一条指令的地址仍旧保存在栈中，调用指令提供了下一条要被执行的指令的地址。

下一条被执行的指令是子程序的第一条指令。

子程序的最后一条指令仅仅使保存在栈顶的地址送回程序


程序存储器
LXI H,08FFH
SPHL
LOP1 IN 2
ANI 20H
JNZ LOP1
CALL D2MS
LOP3 IN 2
MVI A,3
00F7 in
OUT 2
JMP START
0026 out
D2MS MVI A,0
LOPD DCR A
JNZ LOPD
RET
21 FF 08 F9
DB 02 E6 20 C2 1C 00 CD F7 00 DB 02
3E 03 D3 02 C3 0C 00 3E 00 3D C2 F9 00 C9
0000 0001 0002 0003
001C 001D 001E 001F 0020 0021 0022 0023 0024 0025 0026 0027
00F0 00F1 00F2 00F3 00F4 00F5 00F6 00F7 00F8 00F9 00FA 00FB 00FC 00FD
A 00
B,C
D,E
H,L
SP 08FD
PC
I CD
数据存储器
0800 0801 0802 0803 0804 0805
08FD 08FE 08FF
00 26


程序存储器
LXI H,08FFH
SPHL
LOP1 IN 2
ANI 20H
JNZ LOP1
CALL D2MS
LOP3 IN 3
MVI A,3
OUT 2
JMP START
D2MS MVI A,0
LOPD DCR A
JNZ LOPD
RET
21 0000
FF 0001
08 0002
F9 0003
DB 001C
02 001D
E6 001E
20 001F
C2 0020
1C 0021
00 0022
CD 0023
F7 0024
00 0025.
DB 0026
02 0027
3E 00F0
03 00F1
D3 00F2
02 00F3
C3 00F4
0C 00F5
00 00F6
3E 00F7
00 00F8
3D 00F9
C2 00FA
F9 00FB
00 00FC
C9 00FD
A 00
B,C
D,E
H,L
SP 08FD
PC 00F7
I 3E
数据存储器
0800
0801
0802
0803
0804
0805
08FD
00 08FE
26 08FF


程序存储器
LXI H,08FFH
SPHL
LOP1 IN 2
ANI 20H
JNZ LOP1
CALL D2MS
LOP3 IN 2
MVI A,3
OUT 2
JMP START
D2MS MVI A,0
LOPD DCR A
JNZ LOPD
RET
21 0000
FF 0001
08 0002
F9 0003
DB 001C
02 001D
E6 001E
20 001F
C2 0020
1C 0021
00 0022
CD 0023
F7 0024
00 0025
DB 0026
02 0027
3E 00F0
03 00F1
D3 00F2
02 00F3
C3 00F4
0C 00F5
00 00F6
3E 00F7
00 00F8
3D 00F9
C2 00FA
F9 00FB
00 00FC
C9 00FD
A 00
B,C
D,E
H,L
SP 08FF
PC 0026
I C9
数据存储器
0800
0801
0802
0803
0804
0805
08FD
00 08FE
26 08FF

计数器，从而执行接在调用指令后面的那条指令。

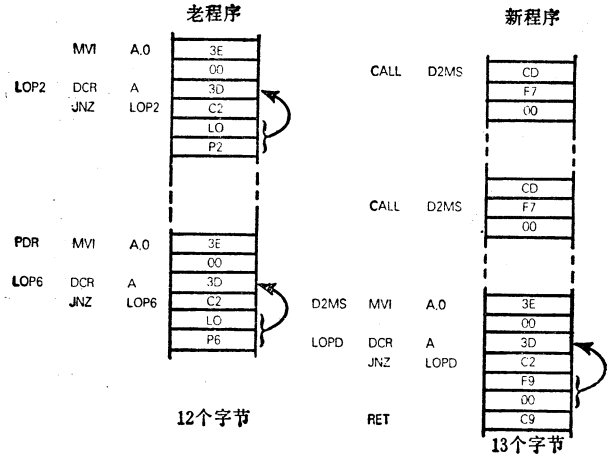
### 5-3-3 什么时候使用子程序

使用子程序是有代价的：

1）每一个调用指令需要增加三个结果代码字节来表示。

2）已经纳入子程序的指令序列必须再附加上一条返回指令，这要占用一个结果代码字节。

首先看一下我们的具体情况。构成2毫秒时间延迟的三条指令占有6个结果代码字节，这三条指令出现两次，所以它们总共占用12个结果代码字节。当把它们纳入到子程序的时候，增添一条返回指令使结果代码字节从6增加到7。此外还有两条调用指令，每条指令要求3个结果代码字节。这就是说，两条调用指令加上子程序一共需要13个结果代码字节。这可以用下图说明：



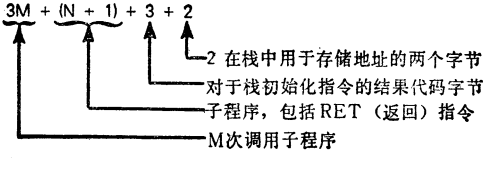
所以在我们的具体情况下，把2毫秒时间延迟指令序列纳入到子程序需要我们多付出一个结果代码字节的代价。栈指示器的初始化已经使我们又得增加4个结果代码字节；这样，我们的微型计算机系统现在就必须采用RAM存贮器了。

仅当有了读/写存储器后，才能把栈建立起来。

从以上分析不应得出结论，认为程序设计中使用子程序毫无把握，因而需谨慎从事；恰恰相反，当你编写程序的时候，很难想像在一段程序中会不包含几个子程序。但应记住，存在着一个最小的子程序长度，小于这个长度的子程序将是不经济的。

假定在一个我们希望把它改造成为子程序的指令序列中有N个结果代码字节，而且这N个结果代码字节出现M次。那么当N个结果代码字节被编成一个子程序时，它将被M个CALL指令所调用。

如果不采用子程序，就得占用M×N个字节，N个字节将被重复M次。而采用子程序以后，被占用的字节数目是：




要想使子程序成为值得采纳的方案，必须使3M＋N＋6小于M×N。表5-1示出经济的子程序的最小长度与子程序调用次数之间的关系。

### 5-3-4 子程序的条件返回

即使在图4-6所示的程序中的几个重复循环的指令序列

表 5-1 经济的子程序的最小长度与子程序调用次数之间的关系

| 子程序调用次数 | 经济的子程序的最小长度(N) |
|---|---|
| 2 | 12 字节 |
| 3 | 8 字节 |
| 4 | 6 字节 |
| 5 | 6 字节 |
| 10 | 4 字节 |
| 20 | 4 字节 |

中，也没有一个长到足以使其必须改编成子程序；然而，我们在下面仍然要探讨编写子程序的可能性。

正如我们在一个时间延迟循环里经常使用条件转移指令一样，这里也有子程序的条件调用指令和子程序的条件返回指令。

在比较长的，而其中又有几种可选择的执行路线的子程序中，子程序条件调用指令和条件返回指令特别有效。

现在来考察图 4-6 中打印锤启动指令序列。在所给出的程序中，这个指令序列只出现一次，这就说明把它转换成子程序是没有意义的，但可以设想一个能执行许多打印机接口操作的较为完善的程序，从而可以根据许多不同的理由来触发打印锤启动逻辑。因为打印锤启动逻辑是由相当长的指令序列组成的，所以将这些指令编成一个子程序将是绝对必要的。下面来看这一子程序的实现：

```
;PRINTHAMMER FIRING SUBROUTINE
PFIR    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        RLC             ;MOVE BIT 7 INTO CARRY
        RNC             ;IF CARRY IS NOT SET, RETURN
```

```
        ANI     20H         ;ISOLATE BIT 4 WHICH IS NOW BIT 5
        RZ                  ;IF ZERO, RETURN
;FIRE PRINTHAMMER
        MVI     A4          ;SET HAMMER PULSE LOW. OUTPUT 0
        OUT     3           ;TO BIT 2 OF I/O PORT 2
        IN      1           ;INPUT ASCII CHARACTER CODE TO ACCUMULATOR
        ANI     7FH         ;MASK OUT HIGH ORDER BIT
;COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
        CPI     20H
        RM                  ;IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING
;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CPI     7AH
        RP                  ;IF CODE IS 7BH OR GREATER, BYPASS FIRING
;ASCII CODE IS VALID
        MVI     H,03H       ;LOAD INDEX TABLE ADDRESS, HIGH ORDER BYTE
        MOV     L,A         ;MOVE LOW ORDER BYTE OF ADDRESS TO L
        MOV     L,M         ;LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS
        CALL    LDLY        ;CALL LONG DELAY SUBROUTINE
        MVI     A,5         ;AT END OF DELAY OUTPUT 1 TO BIT 2 OF
        OUT     3           ;I/O PORT 2 (HAMMER PULSE HIGH)
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
        LXI     H,MS3
        CALL    LDLY
;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
;PW REL HIGH
        MVI     A,1
        OUT     3
        RET                 ;RETURN FROM SUBROUTINE
;LONG DELAY SUBROUTINE. ASSUME H AND L
;ADDRESS THE FIRST OF TWO DATA BYTES WHICH HOLD
;THE INITIAL DELAY CONSTANT
LDLY    MOV     E,M         ;LOAD INITIAL DELAY CONSTANT
        INX     H
        MOV     D,M
LDLP    DCX     D           ;EXECUTE LONG DELAY
        MOV     A,D
        ORA     E
        JNZ     LDLP
        RET                 ;RETURN AT END OF LONG DELAY
MS3     00F7H               ;PRINTWHEEL RELEASE TIME DELAY CONSTANT
```

**打印锤启动子程序**

PFIR IN 2 I/O 口 2 送入到累加器

RLC 传送第 7 位到进位位

RNC 如果进位位未置位、返回

ANI 20H 抽出第 4 位，即现在的第 5 位

```
RZ        如果为零，返回
```

启动打印锤

```
MVI   A 4     将"打印锤脉冲"置为低电平
OUT   3       将"0"输出至I/O口2的第2位
IN    1       ASCII符号输入累加器
ANI   7 FH    屏蔽高位位
```

比较ASCII码和最低规定值

```
CPI   20 H
```

```
RM    如果代码是IFH或小于IFH跳过启动打印锤程序段
```

比较ASCII码和最高规定值

```
CPI    7 AH
```

```
RP    如果代码是7 BH或大于7 BH跳过启动打印锤程序段
```

ASCII码有效

```
MVI    H,03 H   取变址表地址高位字节
MOV    L,A      把地址的低位字节传送到L
MOV    L,M      取时间延迟表地址的低位字节
CALL   LDLY     调用长时间延迟子程序
MVI    A,5      在时间延迟结束时输出"1"送入
OUT    3        I/O口2的第2位("打印锤脉冲"高电平)
```

执行3毫秒的"印字轮释放"时间延迟

```
LXI    H,MS 3
CALL   LDLY
```

将"1"输出至I/O口2的第0位，从而将"印字轮释放"置成高电平

```
MVI   A,1
OUT   3
RET          从子程序返回
```

长时间延迟子程序，假定H和L内存放的是保存初始时间延迟常数的二个数据字节中的第一个字节的地址

```
LDLY   MOV    E,M   取初始时间延迟常数
```

```
        INX     H
        MOV     D,M
LDLP    DCX     D       执行长时间延迟
        MOV     A,D
        ORA     E
        JNZ     LDLP
        RET             在长时间延迟结束时返回
MS3     00F7H           "印字轮释放"的时间延迟常数
```

**条件返回**

上述子程序仅仅当所有必要的条件都满足时才启动打印锤。如果某个启动条件没有得到满足，就立即停止执行。在上述程序中的条件返回指令都加上了网线。

还应指出我们采用了较为紧凑的指令序列，既能将单个位的数据输出到I/O口2，又能识别出正确的打印锤启动时间延迟。

**嵌套子程序**

我们已经在子程序里边又使用了子程序，就是把长时间延迟指令序列改变成了子程序。这个指令序列的第一条指令用LDLY标识，这叫做"嵌套子程序"。

**子程序参数**

子程序LDLY的一个新的特征就是它要求把初始时间延迟常数存放在存储器的两个字节里。当LDLY被调用时，这第一个字节是由H和L寄存器来寻址的。在子程序LDLY中的指令实际上将初始时间延迟常数送入寄存器D和E中。这个初始时间延迟常数叫做一个参数。这种参数能让这子程序实现一个系列的时间延迟。子程序参数是使用子程序的很重要的特征。

第二次调用子程序LDLY时，我们把用符号MS3标识的地址送入寄存器H和L，而不是把所要求的初始常数（$F7_{16}$）

送入寄存器D和E。符号MS 3是在存储器内某处的两个数据字节的地址，在这两个数据字节中，必定存放着数值 00 F $7_{16}$。

### 5-3-5 多重子程序返回

子程序 PFIR 不能充分发挥它的作用。从这个子程序有四种条件返回，每一返回都是由一种不同的无效条件触发的。还有一个接在有效的打印锤启动后面的子程序返回。

在 PFIR 被调用以后，调用程序如何知道打印锤是否已经启动了呢？测试状态位是不可靠的，因为我们不能确切知道在打印锤启动指令执行期间状态条件又发生了什么变化，你也不能通过测试进位位状态来确定 RNC 指令是否就是引起从子程序 PFIR 退出的条件返回指令。这是因为当执行子程序的其余部分时，你不能说明进位位的状态发生什么变化。例如，倘若 RM 条件返回指令使得从子程序 PFIR 退出的话，那么进位位的状态应是“0”。

有些子程序除了一个标准的返回出口以外，还包括许多条件错误出口。这些子程序常常包括一些逻辑使之返回到在主程序中的一些不同的指令。以子程序 PFIR 的情况为例，调用这个子程序的指令序列如下：

```
RT0     CALL    PFIR    ;CALL PRINTHAMMER FIRING SUBROUTINE
        JMP     RT1     ;RETURN HERE FOR PRINTWHEEL REPOSITIONING
        JMP     RT0     ;RETURN HERE FOR HAMMER INTERLOCK LOW
        JMP     RT2     ;RETURN HERE FOR ASCII CODE LESS THAN 20H
        JMP     RT3     ;RETURN HERE FOR ASCII CODE GREATER THAN 7AH
;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED AFTER
;VALID PRINTHAMMER FIRING
        .
        .
        .
;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED FOR
;PRINTWHEEL REPOSITIONING
RT1     .
        .
        .
```

```
                ;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED FOR
                ;ASCII CODE LESS THAN 20H
                RT2     -
                        -
                        -
                ;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED
                ;FOR ASCII CODE GREATER THAN 7AH
                RT3
```

```
        —
        —
        —
RTO  CALL  PFIR   调用打印锤启动子程序
     JMP   RT 1   为使打印锤重新定位返回此处
     JMP   RT 0   为使"打印锤联锁"置为低电平，返回此处
     JMP   RT 2   当 ASCII 码小于 20 H，返回此处
     JMP   RT 3   若 ASCII 码大于 7 AH，返回此处
```

在有效的打印锤启动之后，接下去执行的指令

```
        —
        —
        —
```

为使打印锤重新定位接下去执行的指令：

```
RT 1    —
        —
        —
```

对于 ASCII 码小于 20 H。接着执行以下指令：

```
RT 2    —
        —
        —
```

对于 ASCII 码大于 7 AH，接着执行以下指令：

```
RT 3    —
        —
        —
```

对于现在这一个方案来说，每执行一次条件返回指令，子程序 PFIR 都必须使存放在栈顶的两个字节里的返回地址增 2 。

所以将子程序 PFIR 修改如下：

```
;PRINTHAMMER FIRING SUBROUTINE
PFIR    IN      2       ;INPUT I/O PORT 2 TO ACCUMULATOR
        RLC             ;MOVE BIT 7 INTO CARRY
        RNC             ;IF CARRY IS NOT SET, RETURN
        CALL    INCR    ;INCREMENT RETURN ADDRESS
        ANI     20H     ;ISOLATE BIT 4 WHICH IS NOW BIT 5
        RZ              ;IF ZERO, RETURN
        CALL    INCR    ;INCREMENT RETURN ADDRESS
;FIRE PRINTHAMMER
        MVI     A,4     ;SET HAMMER PULSE LOW, OUTPUT 0
        OUT     3       ;TO BIT 2 OF I/O PORT 2
        IN      1       ;INPUT ASCII CHARACTER CODE TO ACCUMULATOR
        ANI     7FH     ;MASK OUT HIGH ORDER BIT
;COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
        CPI     20H
        RM              ;IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING
        CALL    INCR    ;INCREMENT RETURN ADDRESS
;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CPI     7AH
        RP              ;IF CODE IS 7BH OR GREATER, BYPASS FIRING
        CALL    INCR    ;INCREMENT RETURN ADDRESS
;ASCII CODE IS VALID
        MVI     H,03H   ;LOAD INDEX TABLE ADDRESS, HIGH ORDER BYTE
        MOV     L,A     ;MOVE LOW ORDER BYTE OF ADDRESS TO L
        MOV     L,M     ;LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS
        CALL    LDLY    ;CALL LONG DELAY SUBROUTINE
        MVI     A,5     ;AT END OF DELAY OUTPUT 1 TO BIT 2 OF
        OUT     3       ;I/O PORT 2 (HAMMER PULSE HIGH)
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
        LXI     H,MS3
        CALL    LDLY
;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
;PW REL HIGH
        MVI     A,1
        OUT     3
        RET             ;RETURN FROM SUBROUTINE
;LONG DELAY SUBROUTINE. ASSUME H AND L
;ADDRESS THE FIRST OF TWO DATA BYTES WHICH HOLD
;THE INITIAL DELAY CONSTANT
LDLY    MOV     E,M     ;LOAD INITIAL DELAY CONSTANT
        INX     H
        MOV     D,M
LDLP    DCX     D       ;EXECUTE LONG DELAY
        MOV     A,D
        ORA     E
        JNZ     LDLP
        RET             ;RETURN AT END OF LONG DELAY
;SUBROUTINE TO INCREMENT THE RETURN ADDRESS
;OF THE CALLING SUBROUTINE
INCR    INX     SP      ;INCREMENT STACK POINTER TWICE
        INX     SP      ;TO ACCESS PFIR RETURN ADDRESS
        XTHL            ;EXCHANGE HL WITH PFIR RETURN ADDRESS
```

```
INX     H       ;ADD 3 TO RETURN ADDRESS
INX     H
INX     H
XTHL            ;RESTORE RETURN ADDRESS
DCX     SP      ;DECREMENT STACK POINTER TWICE
DCX     SP
RET             ;RETURN
```

打印锤启动子程序：

```
PFIR    IN      2       I/O 口 2 送入到累加器
        RLC             第 7 位移入进位位
        RNC             如果进位位没有置位，返回
        CALL    INCR    返回地址增 2
        ANI     20 H    抽出第 4 位，即现在的第 5 位
        RZ              如果是 0，返回
        CALL    INCR    返回地址增 2
```

启动打印锤：

```
        MVI  A, 4  将“打印锤脉冲”置为低电平
        OUT  3     将“0”输出至I/O口的第 2 位
        IN   1     ASCII 符号码输入到累加器
        ANI  7 FH  屏蔽高位
```

比较 ASCII 码和最低的规定值：

```
        CPI     20 H
        RM              如果代码等于或小于 1 FH 跳过打印锤启动序列
        CALL    I NCR   返回地址增 2
```

比较 ASCII 码和最高的规定值

```
        CPI     7 AH
        RP              如果代码等于或大于 7 BH 跳过打印锤启动序列
        CALL    INCR    返回地址增 2
```

ASCII 码有效

```
        MVI     H, 03 H 取变址表地址的高位字节
        MOV     L, A    传送地址的低位字节到 L
        MOV     L, M    取时间延迟表地址的低位字节
        CALL    LDLY    调用长时间延迟子程序
```

```
        MVI   A, 5      在时间延迟结束时将"1"输出至
        OUT   3         I/O 口 2 的第 1 位（"打印锤脉冲"为
```
高电平）执行 3 毫秒的"印字轮释放"时间延迟

```
        LXI   H, MS 3
        CALL  LDLY
```

将"1"输出至 I/O 口 2 的第 0 位，置"印字轮释放"为高电平

```
        MVI   A, 1
        OUT   3
        RET       从子程序返回
```

长时间延迟子程序，假定H和L保存着初始时间延迟常数的两个数据字节中的第一个字节的地址

```
LDLY    MOV   E, M    取初始时间延迟常数
        INX   H
        MOV   D, M
LDLP    DCX   D       执行长时间延迟
        MOV   A, D
        ORA   E
        JNZ   LDLY
        RET   从长时间延迟的终点返回
```

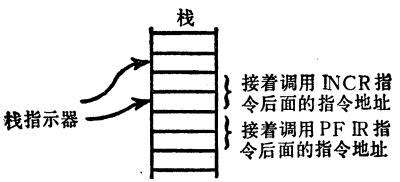
子程序使主程序返回地址增 2

```
INCR    INX   SP  为了存放 PFIR 返回地址
        INX   SP  栈指示器增 2
        XTHL      交换 HL 和 PFIR 的返回地址
        INX   H   返回地址增 3
        INX   H
        INX   H
        XTXL      恢复返回地址
        DCX   SP  栈指示器减 2
        DCX   SP
        RET       返回
```

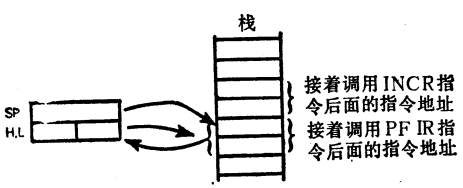
**栈 操 作**

子程序INCR是很有趣的，它说明栈是如何操作的，我们来观察这一操作过程。

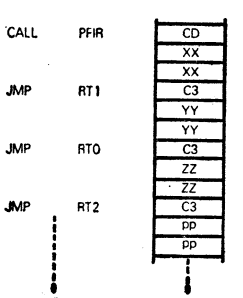
一旦进入子程序INCR，栈指示器的内容就增2，它的作用是存取PFIR返回地址，而不是INCR返回地址：




XTHL指令仅仅将寄存器H和L的内容保存在现在的栈顶，而把原先栈顶的内容移入寄存器H和L。




下面的三条指令将寄存器H和L现在存放的PFIR返回地址增3。我们将返回地址增3是因为如果你看到在调用序列上有一系列的转移指令接在后面的话，则每一条转移指令都占用3个字节。这就是说，每次跳过一条条件返回指令，我们都必须使返回地址增3。



下一条 XTHL 指令仅仅是在栈顶恢复已增 2 的 PFIR 返回地址。

最后，我们必须将栈指示器恢复到原来的内容，从而使 INCR 返回指令能取出正确的返回地址。

### 5-3-6 子程序的条件调用

现在我们建立另一种启动打印锤的子程序，但是不用测试来保证打印锤是否启动。这个子程序仅仅假设只要在累加器中有一个有效的ASCII码，那么打印锤就必须动作。判断打印锤是否是有效动作的所有逻辑都是属于打印锤启动子程序之外的。所以这个子程序被称为条件调用子程序，仅当所有的打印锤启动条件都已满足时才能调用它，这就是现在我们要考察的程序：

```
;TEST PRINTHAMMER FIRING CONDITIONS
LOP3    IN      2       INPUT I/O PORT 2 TO ACCUMULATOR
        RLC             ;MOVE BIT 7 INTO CARRY
        JNC     PRD     ;IF CARRY IS 0, BYPASS PRINTHAMMER FIRING
        ANI     20H     ;ISOLATE BIT 4 WHICH IS NOW BIT 5
        JZ      LOP3    ;WAIT FOR NONZERO VALUE BEFORE FIRING
;INPUT CHARACTER TO BE PRINTED
        IN      1       ;INPUT ASCII CHARACTER TO ACCUMULATOR
        ANI     7FH     ;MASK OUT HIGH ORDER BIT
;COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
        CPI     20H
        JM      PRD     ;IF CODE IS 1F OR LESS, BYPASS HAMMER FIRING
;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CPI     7BH     ;IF CODE IS LESS THAN 7BH, CALL
        CM      FIRE    ;PRINTHAMMER FIRING SUBROUTINE
;EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD     MVI     A,0     ;LOAD ACCUMULATOR WITH 0
```

—
—
—

测试打印锤启动条件

```
LOP 3       IN    2      将 I/O口 2 送入累加器
            RLC          第 7 位传送到进位位
            JNC   PRD    如果进位位是"0"不启动打印锤
```

```
        ANI   20 H      抽出第 4 位，即现在的第 5 位
        JZ    LOP 3     在启动以前如为非零值，则继续等待
输入被打印的字符
        IN    1         ASCII字符送入累加器
        ANI   7 FH      屏蔽高位位
比较ASCII码和最低的规定值
        CPI   20 H
        JM    PRD       如果该码是 1F 或比1F 更小，跳过打印锤启动序列
比较ASCII码和最高的规定值
        CPI   7 BH      如果代码等于小于 7 BH则调
```

| CM | FIRE | 用打印锤启动子程序 |
|---|---|---|

```
执行 2 毫秒的“打印锤准备”时间延迟
PRD     MVI   A,O       将“0”送入累加器
        —
        —
        —
```

应当注意，条件返回指令反映“或”的程序设计逻辑；反之，条件调用指令反映“与”的程序设计逻辑。子程序 PFIR 包括若干个条件返回指令，每一个条件返回指令都将在遇到某一种无效条件时被执行。另一方面，仅当最后一个必要的有效条件被测试通过后，才条件调用子程序 FIRE，在此没有详细说明子程序 FIRE，因为还要对条件调用指令稍加理解之后才能把它编写出来。参照图 4-6 可知子程序 FIRE 中的指令将实现下列功能：

将“打印锤脉冲”信号置为低电平，
执行打印锤启动脉冲时间延迟，
将“打印锤启动脉冲”置为高电平，
执行 3 毫秒的“打印锤释放”时间延迟，
“印字轮释放”脉冲输出为高电平。

## 5-4 宏 指 令

当谈到子程序的时候，我们回避了这样的事实，即你是一位程序设计人员。子程序还有另外一个意义，就是如果使用子程序能够减少源程序的指令数目，那么它将也能减少你化费在编写源程序上的时间。因为编写程序的时间直接和程序的长度成正比。

我们从另一方面来考虑2毫秒的时间延迟子程序，虽然在子程序的形式中，程序要求较多的结果代码字节，但它不要求更多的指令。

| 旧程序 | | | 新程序 | | |
|---|---|---|---|---|---|
| | MVI | A, O | CALL | D2MS | |
| LOP2 | DCR | A | — | | |
| | JNZ | LOP2 | CALL | D2MS | |
| | — | | — | | |
| | — | | — | | |
| | — | | — | | |
| | — | | — | | |
| PDR | MVI | A, D | D2MS | MVI | A, O |
| LOP6 | DCR | A | LOPD | DCR | A |
| | JNZ | LOP6 | | JNZ | LOPD |
| | | | | RET | |
| 6条指令（12个字节） | | | 6条指令(13个字节,不包括栈和初始化指令) | | |

子程序能够减少源程序的长度，然而却增加了你的目标程序的长度和程序的执行时间。

宏指令减少了你的源程序的长度，但对你的目标程序完全

无效。

### 5-4-1 什么是宏指令？

宏指令是程序设计中的一种“简写”形式。它允许用一个助记符来定义一个指令序列。

**宏 定 义**

下面来观察2毫秒时间延迟的指令序列，我们可以将它定义为一个宏指令，标号为D2MS，如下所示：

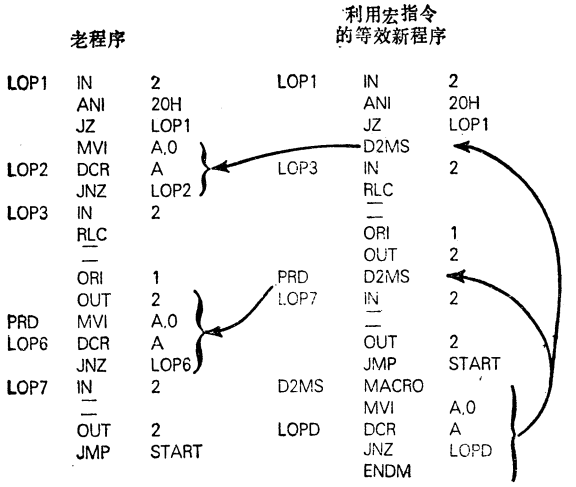
```
D2MS  MACRO
      MVI   A, 0
LOPD  DCR   A
      JNZ   LOPD
ENDM
```

**宏汇编命令**

上图中的两条被网线遮住的指令是汇编命令。它们代表了一段指令序列，从而可以把这段指令序列视为一组，用一个宏汇编命令作为这一组指令的标号。下面就是在打印周期程序内我们怎样使用2毫秒时间延迟程序的例子：

当汇编程序在记忆字段内遇到符号D2MS时，汇编程序就用汇编命令MACRO和ENDM括起来的那几条指令来代替这个符号。如果在你的程序中所用的宏指令不止一个，汇编程序也知道在当前事件中用的是哪一条宏指令，因为在记忆字符字段内的符号必须与宏指令的标号相同。

应该注意，汇编程序也可以完成相当数量的与使用宏指令有关的操作。上图所示的“老程序”中对于两条DCR指令赋与的标号分别为LOP 2和LOP 6；而新程序的宏指令只有一个标




号 LOPD。当宏指令按顺序地在源程序中被引用若干次时，汇编程序聪明得足以知道出现在宏定义中的标号必须变成一系列分别不同的标号。

**宏定义在源程序中的位置**

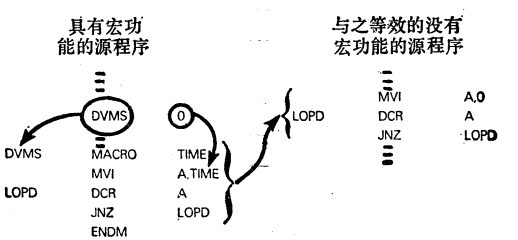
总之，你可以简单地取一个被重复执行的指令序列，将它们用 MACRO 和ENDM 汇编命令括起来，并赋予这个宏汇编命令一个唯一的标号。现在就可以把 MACRO 的标号当做指令的助记符那样使用，宏定义在源程序中的某处只许出现一次而且仅仅出现一次。有一个好主意就是将你所有的宏指令收集在一起，并且将它们编写在这个完整的源程序的开始或是结尾的地方。

## 5-4-2 具有参数的宏指令

在一个宏指令中的指令可以带有可变的操作数。例如，我们可以建立一个可变的时间延迟宏指令如下：

```
DVMS   MACRO   TIME
       MVI     A，TIME
LOPD   DCR     A
       JNZ     LOPD
       ENDM
```

在宏汇编命令的操作数字段中出现的符号被汇编程序认做“哑”符号。在源程序体中的宏基准必须包括一个等价的操作数字段。汇编程序将使宏基准的操作数字段与宏汇编命令的操作数字段等同起来，并作相应的置换。下图示出置换是如何进行的：




这是另一个例子，宏基准

```
DVMS   80 H
```

和以下程序是等价的：

```
       MVI   A，80H
LOPD   DCR   A
       JNZ   LOPD
```

根据你所使用的是哪一种汇编程序，在理论上（但在实际上不一定总是这样）你可以用宏参数表做许多有趣的游戏。这里没有对宏参数表的长度或性质做出任何限制。假定你想改变时间延迟指令序列中所用的寄存器，某些汇编程序将允许将程

序编写如下：

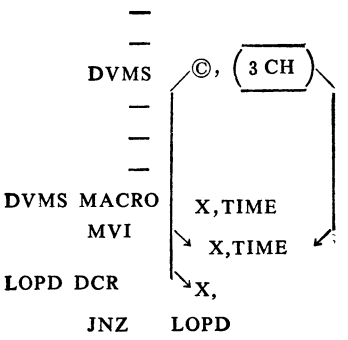
```
        DVMS    C,3CH
DVMS    MACRO   X,TIME
        MVI     X,TIME
LOPD    DCR     X
        JNZ     LOPD
        ENDM
```

The Assembler will substitute:

```
        DVMS    C,3CH
```

with:

```
        MVI     C,3CH
LOPD    DCR     C
        JNZ     LOPD
```




汇编程序将用

```
        MVI     C, 3 CH
LOPD    DCR     C
        JNZ     LOPD
```

来代替

```
        DVMS C, 3 CH
```

如果要知道你所采用的宏指令的确切特性，则需要查阅你的开发系统的汇编程序手册。

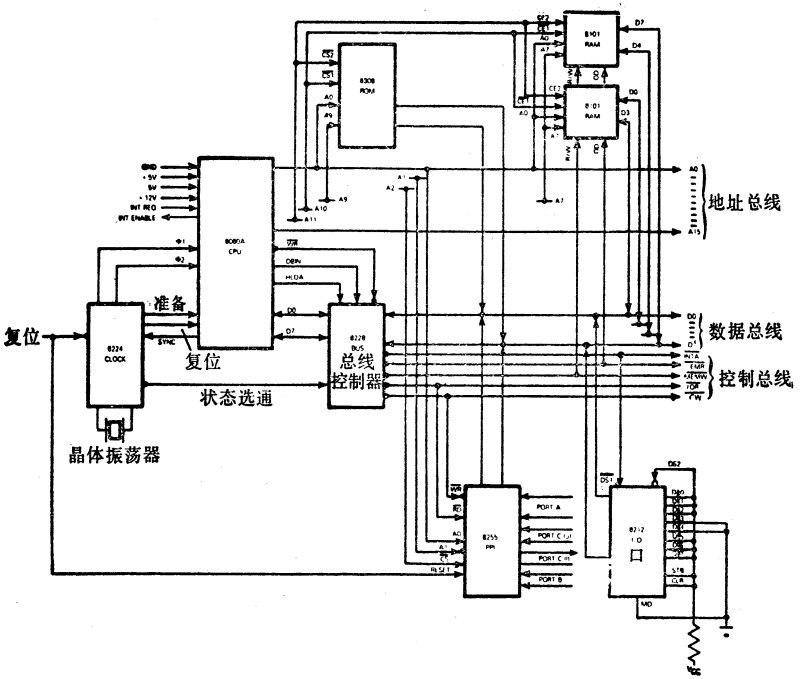



图 5-1 具有单级中断的微型计算机组态

## 5-5 中 断

**很难判断第四章所介绍的微型计算机系统中包含有中断的功能是否得当？事实上在微型计算机的应用中很少采用中断。**

**什么时候使用中断**

关于微型计算机使用中断的优缺点我们将不做大量的探讨。这一课题已在“微型计算机入门”一书第一册：“基本概念”一书中做了适当的讨论。总之，仅当需要处理快速异步事件时，在微型计算机系统里中断才是一种有效的工具。

现在虽然对于不分青红皂白地使用中断提出了衷告，但为了说明中断是怎样进行的，我们将在微型计算机程序中着手编制一段简单的中断处理程序。

### 5-5-1 中断硬件的考虑

对于在8080微型计算机系统中所处理的中断，当允许中断时，必须向CPU输入高电平的中断请求信号。

**中断允许（开中断）**

由分别执行E1和D1指令来实现开中断和关中断。开中断的条件是由8080 CPU的INT ENABLE (INTE—中断允许)端发出的高电平输出来标识的。在试图申请中断以前，外部逻辑是不查询这个信号的，当中断被禁止的时候，CPU很简单地不响应任何中断请求。

**中断响应**

**重新启动指令**

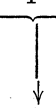
当开中断的时候，如果接收到一个中断请求，则执行完一条现行指令后，CPU通过8228系统控制器输出一个中断响应信号。请求中断的外部逻辑必须输入一个八位的指令代码来响应这一中断响应信号。所输入的八位指令码被译成下一条要执行的指令码。通常，它是8种可能的重新启动指令代码中的一种。这些指令和单字节的子程序调用指令是等效的，它们把程序计数器的内容压入栈中，然后程序从一个低位存储器地址上继续执行下去。这一低位存储器地址可以这样来计算：

RST N 指令码：1 1 1 × × × 1 1

0 0 0 N=0
0 0 1 N=1
0 1 0 N=2

0 1 1 N=3
1 0 0 N=4
1 0 1 N=5
1 1 0 N=6
1 1 1 N=7

程序计数器的新内容为：



0 0 0 0 0 0 0 0 0 0 × × × 0 0 0

可见，RST N 指令是和子程序调用指令 CALL 等效的，因为前者执行如下的转移：

| 子程序 | | |
|---|---|---|
| RST 0 | 转到 | $0000_{16}$ |
| RST 1 | 转到 | $0008_{16}$ |
| RST 2 | 转到 | $0010_{16}$ |
| RST 3 | 转到 | $0018_{16}$ |
| RST 4 | 转到 | $0020_{16}$ |
| RST 5 | 转到 | $0028_{16}$ |
| RST 6 | 转到 | $0030_{16}$ |
| RST 7 | 转到 | $0038_{16}$ |

**RST指令代码的生成**

生成一条适当的 RST 指令最简单的方法是通过外部的 8212 I/O 口。所需要的逻辑示于图 5-1 中。

现在，在图 5-1 中的 8212 I/O 口采用了可能的最简单的方式。我们来考察这个简单 I/O 口的用法。

**单级中断组态**

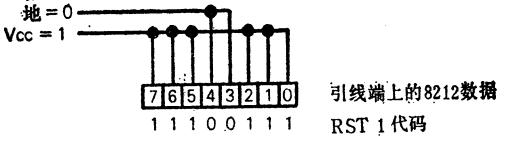
因为这里只有一个中断源，它直接联接到8080 CPU 的 INT REQ 输入端，所以 CPU 的 INT

ENABLE 输出也就不必管它了。

当能够请求中断的唯一的外部设备中断源向 CPU 输入一高电平的“INT REQ”时，在随后的某一时刻将通过总线控制器在控制总线上输出低电平的 $\overline{\text{INTA}}$ 来响应中断。现在，在8212 I/O 口上把 $\overline{\text{INTA}}$ 信号用来作为两个设备选择信号中的一个。另一个设备选择信号 DS 2 必须是高电平，所以把它接到 $V_{cc}$。

**用于中断系统中的8212 I/O 口**

在图 5-1 中，8212 I/O 口的唯一用途是在一旦由 $\overline{\text{INTA}}$ 输出高电平来响应中断之后，由8212 CPU 输入 RSTI 指令。我们所选择的 RSTI 指令是随意的，不过我们不能任意选择的唯一的 RST 指令是 RSTO，因为 0 号存储单元是接在复位之后采用的。RSTI 指令代码的生成是通过将一些适当的数据输入引线端接地而另一些则接至高电平，如图所示：

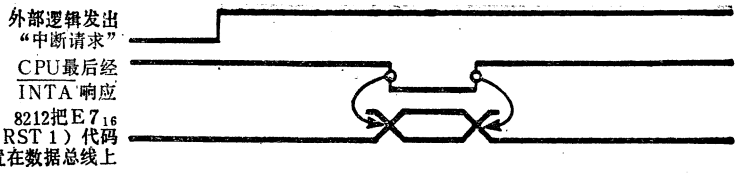



注：8080 微型计算机系统中，紧接在中断之后的重新起动指令代码的生成

工作方式引线端 MD 接地，因为 8212 I/O 口现在用于输入工作方式。

因为对于 8212 I/O 口是否输出数据的唯一的判别方法就是看 $\overline{\text{INTA}}$ 是否为低电平，所以将 STROBE(选通)和CLEAR(清除）输入端接到 $V_{cc}$，使之不起作用。

总之，当外部逻辑请求中断时所发生的情况如下图所示：




无论你是一位逻辑设计者还是一位程序设计员，你都不需要关心数据总线的定时关系。$\overline{\text{INTA}}$信号将正确地把RSTI指令码选通送入CPU，使得8212 I/O口的输出被译码为指令码，而不是把它译成数据。甚至只要简单地观察一下8218 I/O口联接到我们的微型计算机系统的方法，就能明显地看出，8212 I/O口有很多精巧的使用方法，例如可以用它处理多重中断，

把外部中断源直接连接到8080 CPU是处理中断的一种原始的方法，但是在微型计算机系统中仅有一个外部设备请求中断的场合采用这种中断是很适当的。

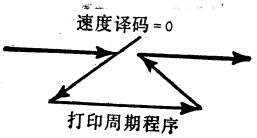
在我们讨论某些比较精巧的中断请求方案以前，先来考虑对于图5-1所示的简单方案的程序设计。

### 5-5-2 中断服务程序

首先我们怎样使用中断呢？

我们可以假定微型计算机系统的用途要比执行打印周期逻辑多得多。设想由打印机接口要求很多例行的辅助逻辑，从而整个打印周期可以被看作间断的异步事件。现在，我们不是执行一段“在两次打印之间”的指令循环；而是假定一些其他的程序正在两次打印周期之间被连续地执行。打印周期程序的执行由“速度译码”信号启动，把“速度译码”信号的取反信号联接到图5-1中的INT REQ端，指令执行的模式如下图所示：

在两次打印周期之间




接收到“速度译码”的信号而发生中断后，执行打印周期程序所需要的指令序列如下：

```
        ORG     8
;ORIGIN PRINT CYCLE PROGRAM INTERRUPT SERVICE ROUTINE
;AT 0008H, SINCE THIS IS THE EXECUTION ADDRESS
;WHICH FOLLOWS EXECUTION OF AN RST 1 INSTRUCTION
        CALL    START   ;CALL PRINT CYCLE PROGRAM AS A SUBROUTINE
        RET             ;RETURN FROM INTERRUPT
        ORG     NNNN
;SELECT ANY VIABLE ORIGIN FOR THE PRINT CYCLE PROGRAM
;INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0 AND 1 OF
;I/O PORT 2. OUTPUT 1 TO BIT 3 OF I/O PORT 2
START   MVI     A,0CH   ;LOAD MASK INTO ACCUMULATOR
        OUT     2       ;OUTPUT TO I/O PORT 2
        .
        .
        .
;AT END OF PRINT CYCLE, SET BIT 1 OF I/O PORT 2 TO 1
;THIS SETS CH RDY HIGH
        MVI     A,3
        OUT     3
        RET
```

ORG 8

打印周期程序的中断服务程序的起始地址为 0008 H，因为这是执行地址，在它之后执行 RSTI 指令

CALL START 调用打印周期程序作为子程序

RET 从中断返回

ORG NNNN

为打印周期程序选择某一个可用的起始地址。为打印周期置初值将“0”输出至 I/O 口2 的第 0，第 1 两位。将“1”输出至 I/O 口 2 的第 3 位

START MVI A, OCH 屏蔽值取入累加器

OUT 2 输出到 I/O 口 2

—
—
—

在打印周期的结尾，将 I/O 口 2 的第 1 位置“1”，把 CH RDY 置成高电平。

```
MVI  A, 3
OUT  3
RET
```

注意到“在两次打印周期之间”的各条指令已经被移去了，START 现在等同于打印周期本身的第一条指令。

**中断程序的起始地址**

为打印周期程序指定起始地址是不重要的，我们不知道在微型计算机系统中正在执行的还有哪些程序；或者，另一些程序可能存储在程序存储器的哪一部位。所以我们不能在此刻为打印周期程序指定存储区域。只有当你具体涉及整个微型计算机系统时，你才必须仔细地精确安排好每个程序在存储器中的存放部位，但是对于我们现在要阐明的目的，这些是完全不重要的。

应该注意，打印周期的最后三条指令是用置位控制指令来修改 CH RDY；但是这里并没有考虑采用 8255 I/O 口作为选通 I/O 的工作方式 1。

最后的 JMP START 指令被一条简单的返回 指令所替代，因为整个打印周期程序实际上是作为一个子程序调用的。

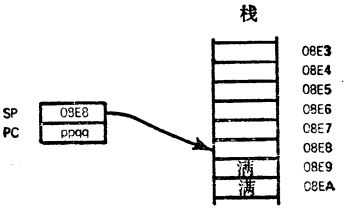
**栈　顶**

**微型计算机系统将尾随着利用栈的中断，围绕着存储器对它自己进行跟踪，下面就来解释这一情况。**

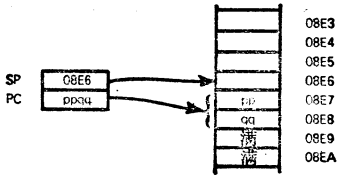
当我们最后一次观察栈的时候，看到栈是从 $08\,FF_{16}$ 起始的，这是读/写存储器的顶部，而且这时栈还没有被访问。我们现在假定：在两次打印周期之间所执行的某一程序对栈进行存取，所以当打印周期执行完毕的时候，栈指示器中含有地址

$08E8_{16}$。我们简单地假定：这里有某种程度的栈活动，但是我们既不需要知道它，也无需关心它。

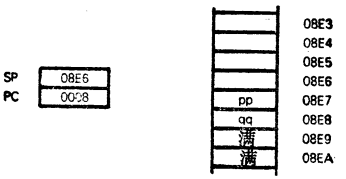
下面说明在中断响应之后栈的使用情况。1，当“速度译码”信号请求中断时，其情况如下：



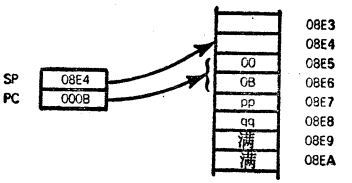

2，当中断响应之后；首先，程序计数器的内容被保护进栈：




3，下一条 RSTI 指令把 0008 送入程序计数器：

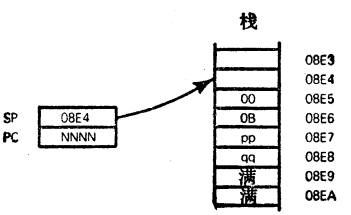



4，存储器 0008 单元存放的是 CALL START（调用 START）指令，使下一条指令的地址保护进栈：

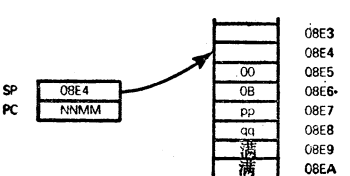



应当记住下一条（RET）指令是被存放在$000B_{16}$单元内的，因为CALL START指令占3个字节，$0008_{16}$，$0009_{16}$和$000A_{16}$。

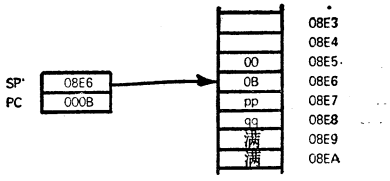
5，现在标号为START的指令的地址被送入程序计数器，这是打印周期服务程序中的第一条指令：



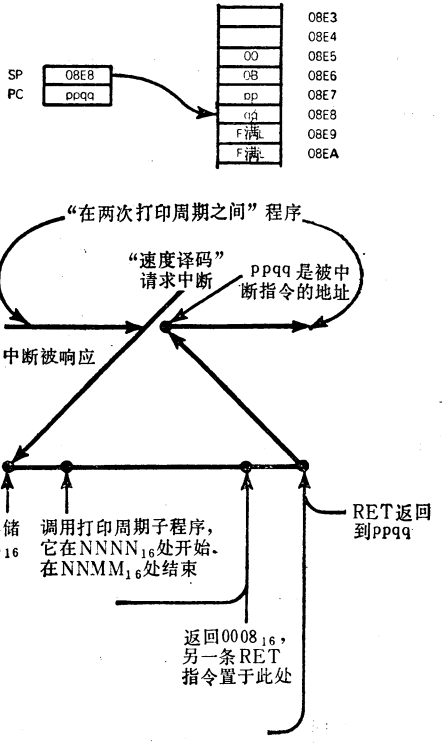

6，假定打印周期服务程序中最后一条RET指令存放在存储单元NNMM中，则当RET指令将要执行时情况如下：




7，当执行RET指令时，栈指示器增2，并且栈顶的两个栈字节被送入程序计数器。




8，另一条 RET 指令被存入存储单元 000B 中，同样的过程再重复一次。



被中断指令的地址已经恢复。这个操作的全过程可以用上页图来说明。

**保护寄存器和状态**

我们在上面介绍的处理中断的方法是相当的简单，这样就能使你顺着程序的执行路线走下去而不感到困难。但是这个程序并不能工作。我们曾经给出为了执行打印周期服务程序而被中断的背景程序。但是什么时候背景程序才被中断呢？大家还记得被中断的程序是和打印周期服务程序公用同一个 CPU 和几个寄存器的。我们假定被中断的程序把有用的信息存放在各个寄存器中，也许状态标志也是有意义的，它们也都必须被保存起来。迄今为止给出了一个中断服务程序，当我们从打印周期服务程

```
            ORG     8
;ORIGIN PRINT CYCLE PROGRAM INTERRUPT SERVICE ROUTINE
;AT 0008H, SINCE THIS IS THE EXECUTION ADDRESS
;WHICH FOLLOWS EXECUTION OF AN RST I INSTRUCTION
            CALL    START   ;CALL PRINT CYCLE PROGRAM AS A SUBROUTINE
            RET             ;RETURN FROM INTERRUPT
            ORG     NNNN
;PUSH CONTENTS OF ALL REGISTERS AND STATUS ONTO STACK
START       PUSH    PSW     ;SAVE ACCUMULATOR AND STATUS
            PUSH    B       ;SAVE B AND C
            PUSH    D       ;SAVE D AND E
            PUSH    H       ;SAVE H AND L
;SELECT ANY VIABLE ORIGIN FOR THE PRINT CYCLE PROGRAM
;INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0 AND 1 OF
;I/O PORT 2. OUTPUT 1 TO BIT 3 OF I/O PORT 2
            MVI     A,0CH   ;LOAD MASK INTO ACCUMULATOR
            OUT     2       ;OUTPUT TO I/O PORT 2

;AT END OF PRINT CYCLE, SET BIT 1 OF I/O PORT 2 TO 1
;THIS SETS CH RDY HIGH
            MVI     A,3
            OUT     3
;RESTORE INTERRUPTED PROGRAM'S REGISTERS CONTENTS AND STATUS
            POP     H       ;RESTORE H AND L
            POP     D       ;RESTORE D AND E
            POP     B       ;RESTORE B AND C
            POP     PSW     ;RESTORE ACCUMULATOR AND STATUS
            RET
```

```
    ORG  8
```

打印周期程序的中断服务程序起始地址为0008 H

因为这是执行地址，接着执行一条 RST 1 指令

```
    CALL  START     把打印周期程序作为子程序调用
    RET             从中断返回
    ORG   NNNN
```

```
;把所有寄存器和状态位的内容压入栈内。
START  PUSH  PSW   保护累加器和状态位
       PUSH  B     保护B和C
       PUSH  D     保护D和E
       PUSH  H     保护H和L
```

为打印周期程序选择一个适当的起始地址，打印周期初始化，将“0”输出至I/O口2的第0位和第1位，将“1”输出至I/O口2的第3位

```
    MVI  A, OCH   屏蔽值移入累加器
    OUT  2        输出到I/O口2
    —
    —
```

在打印周期结束时，把I/O口2的第1位置“1”，从而将 CH RDY 置为高电平

```
    MVI  A,3
    OUT  3
```

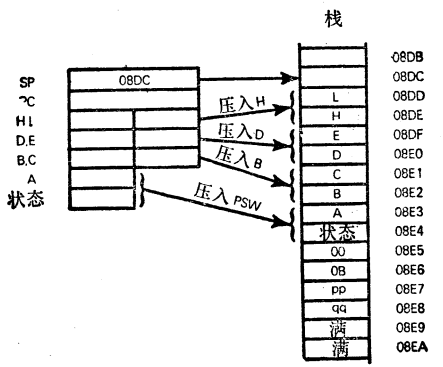
```
;恢复被中断程序的各寄存器的内容和状态:
  POP  H     恢复H和L
  POP  D     恢复D和E
  POP  B     恢复B和C
  POP  PSW   恢复累加器和状态标志
```

```
    RET
```

序返回到被中断的程序时，如果我们把打印周期服务程序结束时的无论哪一个寄存器的内容都送给被中断的程序那是不行的。所以我们必须在打印周期执行程序的前后增设一些指令，

在修改某一寄存器和状态位的内容以前把各个寄存器和状态位的内容保护进栈；在程序结束时必须恢复原来的寄存器和状态位的内容。这就是现在我们在下面看到的打印周期程序：

这个在栈中存放数据的方法是很简单明了的：




只要你记住从栈中弹出各寄存器和状态位的内容与把它们压进栈的次序刚好相反，那末就不会有什么问题了。

### 5-5-3 对于中断的评价

小型计算机程序设计员和大型计算机程序设计员不加选择的使用中断只是为了在各种不同的应用场合分担CPU的费用。

**中断的经济性**

你作为微型计算机的用户将不得不判断一下，在5至20美元之间的费用是否还值得分担？如果采用中断，则除了这一费用外你还必须付出产生中断请求信号的外部逻辑的费用，还需要重新启动指令，这就要增加额外的程序设计费用。经济上的综合权衡说明了在微型计算机系统里采用中断的方法未必合算。当你毫不犹疑地认为中断

是应当采用的方式之前，你应当仔细检验具体应用的条件，往往增加一个 CPU 或一部完整的微型计算机系统要比在几个不同的中断源之间采用中断来分享微型计算机系统来得便宜些。

**中断定时考虑**

即便中断在你的应用中还是合算的，那么定时上的考虑也是很重要的。

当你的应用正是处理一些异步事件时，中断肯定是很有吸引力的。在我们的情况下，假定平均打印周期的持续时间大体为 10 毫秒；而且不能肯定在两次打印周期之间的时间间隔是 1 毫秒还是 100 毫秒。在这种情况下，如果在两次打印周期之间的时间间隔里想要执行另一个程序，我们必须用中断来启动打印周期，因为我们不能预先知道下一次打印周期在什么时候开始。

实际上，两次打印周期之间的间隔时间是可以很精确确定的。一部打印机具有出厂规定的打印速度。假如这个速度是 45 个字符/秒，那么每打印一个字符就需要 22.2 毫秒。如果执行实际的打印周期程序需要 22 毫秒中的 10 毫秒，那么停留在打印周期之间的时间间隔为 12 毫秒。我们就不再需要中断。只要在打印周期之间执行的这一程序是分成程序段的，每一段都能在 12 毫秒或更短的时间内执行完毕，那么在每一段的结束处都加上一段指令循环，用它来测试速度译码输入的状态，以便确定何时启动下一个打印周期。

```
LOOP    IN      2       ;INPUT I/O PORT 2 CONTENTS
        ANI     20H     ;ISOLATE VELOCITY DECODE SIGNAL
        JNZ     LOOP    ;IF STILL 1, NEW PRINT CYCLE HAS NOT BEGUN
```

```
LOOP    IN      2       输入 I/O 口 2 的内容
        ANI     20 H    抽出速度译码信号
        JNZ     LOOP    如果还是“1”，新的打印周期不开始
```

**中断服务时间的浪费**

处理每一个中断都要付出时间的代价。试看必须在打印周期程序本身执行的前后执行的那些指令；这是四条压入指令，四条弹出指令，一条调用指令和一条返回指令。把这些指令所占用的周期的数目相加起来，你将发现执行所有这些指令需要111个周期，即每次中断需时55.5微秒。

这段时间约为执行全部打印周期程序所要求时间的5%。

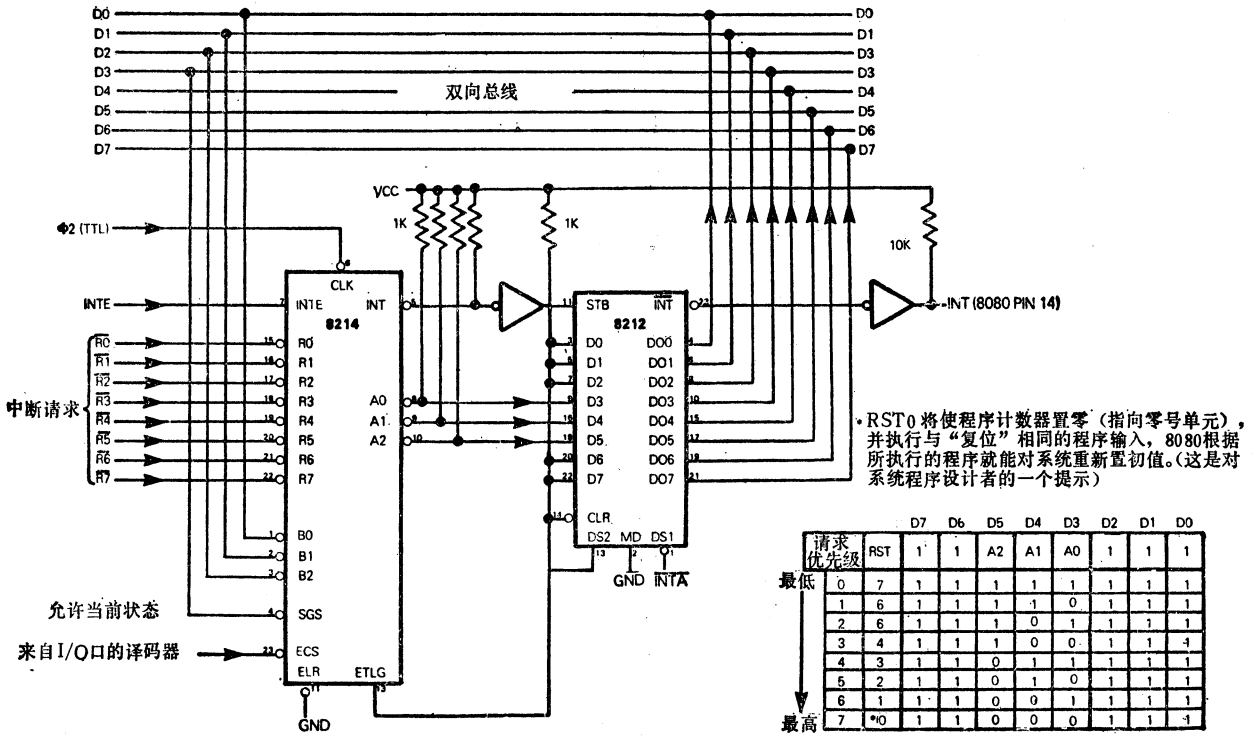
把这类"辅助操作"时间放在一个比较复杂的系统中考虑。在那里也许有10个不同的外部逻辑源都可以产生中断请求，这样一种复杂的中断方案对于小型计算机程序设计者也许是完全合理的。但是在微型计算机系统中，由于有可能为不创造价值的"辅助操作"浪费555微秒的时间；也就是说，你的微型计算机系统在50%的时间内什么也没有做，只不过是存放和恢复寄存器和状态位的内容。很明显，对于中断你必须谨慎从事。

### 5-5-4 多级中断

假如在你的应用中，所有关于不要采用多级中断的衷告都还适用。例如，在这一应用中有许多的异步事件，而且又不是经常发生的。它们对于现有的执行时间没有严重的影响。在8080型微型计算机系统中有许多实现多级中断的方法，把它们全部介绍出来无疑将超过本书的范围。然而所有的多级中断方案都要用到这样两个基本的概念：

1，外部中断请求不再直接接至CPU的中断请求引线端上，而用8255 PPI或8212 I/O口作为缓冲器并发送中断请求。

2，如果请求处理的外部中断最多为八个，那么八条"重新启动"指令，可以用一个8212 I/O口再加一个八中取一的译码器来实现：




| 请求优先级 | RST | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 最低 0 | 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 6 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 6 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 3 | 4 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 4 | 3 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 5 | 2 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 7 最高 | *0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

图 5-2　8080 微型计算机系统紧接中断之后重新启动指令代码的生成

如果你有8个外部中断，那么必然想到其中的某一个要用RST 0“重新启动”指令，这将和复位的条件相重合，因为两者都使程序的执行转移到0号存储单元。

如果你有七个或少于七个不同的外部中断，处理它们就相当简单了。

如果你有九个或更多的中断，并且试图用同一个CPU去处理它们，那么可以肯定，你在设计微型计算机系统时采用了错误的方法。也许有人会设想，利用具有一台CPU的微型计算机系统来实现上述复杂的中断方案也许是经济的，而你的系统恰恰就是这样一种系统，那么我们将乐于恭候你的佳音。

# 第六章　8080/9080指令系统

对于初次接触到程序设计的微型计算机用户来说，指令似乎是令人望而生畏的。然而如果把执行每条指令有关的操作都看成是一些孤立的事件，那就十分容易理解了。这也就是本章所要达到的目的。

为什么微型计算机的各条指令叫做指令系统呢？因为由某个微型计算机的设计者经过认真挑选的这些指令，能够很容易地把一些复杂的操作分解成一个简单的操作序列。而每一种简单的操作都是由精心设计的指令系统中的每一条指令来表示的。

本章的叙述是与“微型计算机入门”第二册，表 6-1，8080/9080 微型计算机指令系统摘要相一致的。而且，也采用了类似的指令分类方法。

除了简单地说明各条指令的功能之外，还指出在正常的程序设计逻辑中每条指令的用途。

## 6-1　缩写符号

下面是本章所采用的一些缩写符号：

| | | |
|---|---|---|
| A | 累加器 | |
| B | 寄存器 B | 有时作为寄存器对 |
| C | 寄存器 C | |
| D | 寄存器 D | 有时作为寄存器对 |
| E | 寄存器 E | |
| H | 寄存器 H | 这一寄存器对提供隐含存储器地址。 |
| L | 寄存器 L | |

## 表 6-1 8080/9080微型计算机指令系统摘要

| 类型 | 助记符 | 操作数 | 字节 | 状态 C Ac Z S P | 完成操作 |
|---|---|---|---|---|---|
| 输入/输出 | IN | DEV | 2 | | [A]←[DEV]<br>设备 DEV 输入A |
| | OUT | DEV | 2 | | [DEV]←[A]<br>A输出至DEV |
| 主要存储器访问指令 | LDAX | RP | 1 | | [A]←[[RP]]<br>将由BC(RP=B)或 DE(RP=D)隐含寻址的存储单元内容送入累加器 |
| | STAX | RP | 1 | | [[RP]]←A<br>将累加器内容存入由 RP 隐含寻址的存储单元内 |
| | MOV | R,M | 1 | | [R]←[[HL]]<br>由HL 隐含寻址的存储单元的内容送入某一寄存器 |
| | MOV | M,R | 1 | | [[HL]]←R<br>将某一寄存器的内容存入由 HL 隐含寻址的存储单元内 |
| | LDA | ADDR | 3 | | [A]←[ADDR]也就是[A]←[[13,12]]<br>直接寻址的存储单元的内容送入累加器 |
| | STA | ADDR | 3 | | [ADDR]←[A]也就是[[13,12]]←[A]<br>将累加器内容存入采用直接寻址的存储单元内 |
| | LHLD | ADDR | 3 | | [L]←[ADDR],[H]←[ADDR+1]也即<br>[L]←[[13,12]],[H]←[[13,12]+1]<br>用直接寻址的存储单元内容送入寄存器H和L |
| | SHLD | ADDR | 3 | | [ADDR]←[L],[ADDR+1]←[H]也即[13,12]←[L] [[13,12]+1]←[H]<br>将寄存器H和L的内容存入直接寻址的存贮单元内 |

续表

| 类　型 | 助记符 | 操作数 | 字　节 | 状　态<br>C Ac Z S P | 完　成　操　作 |
|---|---|---|---|---|---|
| 辅助存储器访问指令（存储器操作） | ADD | M | 1 | × × × × × | [A]←[A]+[[H,L]]<br>累加器内容和用H，L隐含寻址的存贮单元内容相加 |
| | ADC | M | 1 | × × × × × | [A]←[A]+[[H,L]]+[CS]<br>累加器内容和H，L隐含寻址的存储单元内容进行带进位的加法 |
| | SUB | M | 1 | × × × × × | [A]←[A]−[[H,L]]<br>累加器内容和用 H,L 隐含寻址的存储单元内容相减 |
| | SBB | M | 1 | × × × × × | [A]←[A]−[[H,L]]−[CS]<br>累加器内容和用 H,L 隐含寻址的存储单元内容进行带借位的减法 |
| | ANA | M | 1 | 0 × × × × | [A]←[A]∧[[H,L]]<br>用 H,L 隐含寻址的存储单元内容和累加器内容进行逻辑“与”运算 |
| | XRA | M | 1 | 0 × × × × | [A]←[A]∀[[H,L]]<br>用 H,L 隐含寻址的存储单元内容和累加器内容进行“异”运算 |
| | ORA | M | 1 | 0 × × × × | [A]←[A]∨[[H,L]]<br>用 H,L 隐含寻址的存储单元内容和累加器内容进行逻辑或运算 |

续表

| 类　型 | 助记符 | 操作数 | 字　节 | 状　态<br>C Ac Z S P | 完　成　操　作 |
|---|---|---|---|---|---|
| 辅助存储器访问指令（存储器操作） | CMP | M | 1 | × × × × × | 存储单元内容和累加器内容进行比较 |
| | INR | M | 1 | × × × × | [[H,L]]←[[H,L]]+1<br>存储单元内容增 1 |
| | DCR | M | 1 | × × × × | [[H,L]]←[[H,L]]−1<br>存储单元内容减 1 |
| 立即数操作 | LXI | RP,DATA 16 | 3 | | [RP]←DATA16<br>将16位立即数据送入BC(RP=B)，DE(RP=D)，HL(H=D)或SP(RP=SP) |
| | MVI | M,DATA | 2 | | [[H,L]]←DATA<br>将 8 位立即数据存入由H,L 隐含寻址的存储单元 |
| | MVI | R,DATA | 2 | | [R]←DATA<br>将 8 位立即数据送入某一寄存器 |
| 转移指令 | JMP | ADDR | 3 | | [PC]←ADDR<br>转移执行标号为 ADDR 的那条指令 |
| | PCHL | | 1 | | [PC]←[[H,L]]<br>转移执行存储在由H,L隐含寻址的该存储单元的指令 |

续表

| 类型 | 助记符 | 操作数 | 字节 | 状态 C Ac Z S P | 完成操作 |
|---|---|---|---|---|---|
| 子程序的调用和返回（立即数和栈） | CALL | ADDR | 3 | | [[SP]]←[PC], [PC]←ADDR,[SP]←[SP]－2<br>转移到起始地址为 ADDR 的子程序 |
| | CC | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR,[SP]←[SP]－2<br>若进位状态 C＝1,转移到子程序 |
| | CNC | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR [SP]←[SP]－2<br>若进位状态 C＝0,转移到子程序 |
| | CZ | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR,[SP]←[SP]－2<br>若零状态 Z＝1,转移到子程序 |
| | CNZ | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR [SP]←[SP]－2<br>若零状态 Z＝0,转移到子程序 |
| | CP | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR, [SP]←[SP]－2<br>若符号状态 S＝0,转移到子程序 |
| | CM | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR,[SP]←[SP]－2<br>若符号状态 S＝1,转移到子程序 |
| | CPE | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR, [SP]←[SP]－2<br>若奇偶性为偶，转移到子程序 |
| | CPO | ADDR | 3 | | [[SP]]←[PC],[PC]←ADDR,[SP]←[SP]－2<br>若奇偶性为奇，转移到子程序 |
| | RET | | 1 | | [PC]←[[SP]],[SP]←[SP]＋2<br>从子程序返回 |
| | RC | | 1 | | [PC]←[[SP]],[SP]←[SP]＋2<br>若进位状态 C＝1,从子程序返回主程序 |

续表

| 类　型 | 助记符 | 操作数 | 字　节 | 状　态<br>C Ac Z S P | 完　成　操　作 |
|---|---|---|---|---|---|
| 子程序的调用和返回（立即数和栈） | RNC | | 1 | | [PC]←[[SP]],[SP]←[SP]+2<br>若进位状态 C=0,从子程序返回主程序 |
| | RZ | | 1 | | [PC]←[[SP]],[SP]←[SP]+2<br>若零状态 Z=1,从子程序返回主程序 |
| | RNZ | | 1 | | [PC]←[[SP]],[SP]←[SP]+2<br>若零状态 Z=0,从子程序返回主程序 |
| | RM | | 1 | | [PC]←[[SP]],[SP]←[SP]+2<br>若符号状态 S=1,从子程序返回主程序 |
| | RP | | 1 | | [PC]←[[SP]],[SP]←[SP]+2<br>若符号状态 S=0,从子程序返回主程序 |
| | RPE | | 1 | | [PC]←[[SP]],[SP]←[SP]+2<br>若奇偶性为偶，从子程序返回主程序 |
| | RPO | | 1 | | [PC]←[[SP]],[SP]←[SP]+2<br>若奇偶性为奇，从子程序返回主程序 |
| 立即数操作 | ADI | DATA | 2 | × × × × × | [A]←[A]+DATA<br>累加器和立即数相加 |
| | ACI | DATA | 2 | × × × × × | [A]←[A]+DATA+[CS]<br>累加器和立即数进行带进位的加法 |
| | SUI | DATA | 2 | × × × × × | [A]←[A]−DATA<br>累加器和立即数相减 |
| | SBI | DATA | 2 | × × × × × | [A]←[A]−DATA−[CS]<br>累加器和立即数进行带借位减法 |

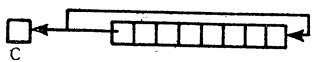
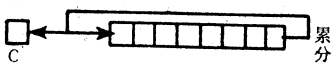
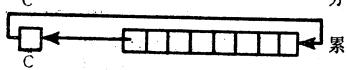
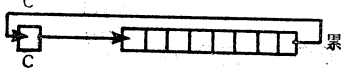
续表

| 类　型 | 助记符 | 操作数 | 字　节 | 状　态<br>C Ac Z S P | 完　成　操　作 |
|---|---|---|---|---|---|
| 立即数操作 | ANI | DATA | 2 | 0 × × × × | [A]←[A]∧DATA<br>立即数和累加器内容相“与” |
| | XRI | DATA | 2 | 0 0 × × × | [A]←[A]∀DATA<br>立即数和累加器相“异” |
| | ORI | DATA | 2 | 0 0 × × × | [A]←[A]∨DATA<br>立即数和累加器相“或” |
| | CPI | DATA | 2 | × × × × × | 立即数和累加器的内容相比较 |
| 按条件转移 | JC | ADDR | 3 | | [PC]←ADDR<br>若进位状态 C=1,转移 |
| | JNC | ADDR | 3 | | [PC]←ADDR<br>若进位状态C=0,转移 |
| | JZ | ADDR | 3 | | [PC]←ADDR<br>若零状态 Z=1,转移 |
| | JNZ | ADDR | 3 | | [PC]←ADDR<br>若零状态 Z=1,转移 |
| | JP | ADDR | 3 | | [PC]←ADDR<br>若符号状态 S=0,转移 |
| | JM | ADDR | 3 | | [PC]←ADDR<br>若符号状态 S=1,转移 |
| | JPE | ADDR | 3 | | [PC]←ADDR<br>若奇偶性为偶，转移 |
| | JPO | ADDR | 3 | | [PC]←ADDR<br>若奇偶性为奇，转移 |

续表

| 类　型 | 助记符 | 操作数 | 字　节 | 状　态<br>C Ac Z S P | 完　成　操　作 |
|---|---|---|---|---|---|
| 寄存器-寄存器传送 | MOV | D.S | 1 | | [R]←R<br>传送某一寄存器的内容(S)到另一寄存器(D) |
| | XCHG | | 1 | | [D]←→[H]，[E]←→[L]<br>DE和HL交换内容 |
| | SPHL | | 1 | | [H,L]←→[SP]<br>SP和HL 交换内容 |
| 寄存器-寄存器操作 | ADD | R | 1 | × × × × × | [A]←[A]+[R]<br>累加器和某寄存器的内容相加 |
| | ADC | R | 1 | × × × × × | [A]←[A]+[R]+[CS]<br>累加器和某寄存器进行带进位加法 |
| | SUB | R | 1 | × × × × × | [A]←[A]−[R]<br>从累加器减去某寄存器内容，结果送累加器 |
| | SBB | R | 1 | × × × × × | [A]←[A]−[R]−[CS]<br>累加器与某寄存器进行带借位减法，结果送累加器 |
| | ANA | R | 1 | 0 × × × × | [A]←[A]∧[R]<br>某寄存器和累加器相“与” |
| | XRA | R | 1 | 0 × × × × | [A]←[A]∀[R]<br>某寄存器和累加器相“异” |
| | ORA | R | 1 | × × × × × | [A]←[A]∨[R]<br>某寄存器和累加器相“或” |
| | CMP | R | 1 | × × × × × | 累加器和某寄存器相比较 |

续表

| 类　型 | 助记符 | 操作数 | 字　节 | 状　态<br>C Ac Z S P | 完　成　操　作 |
|---|---|---|---|---|---|
| 寄存器操作 | INR | R | 1 | × × × × | [R]←[R]+1<br>某寄存器内容增1 |
| | DCR | R | 1 | × × × × | [R]←[R]−1<br>某寄存器内容减1 |
| | CMA | | 1 | | [A]←[$\overline{A}$]<br>累加器的内容取反 |
| | DAA | | 1 | × × × × × | 十进制调整累加器A |
| 寄存器操作 | RLC | | 1 | × |  累加器A循环左移，分支到进位位 |
| | RRC | | 1 | × |  累加器A循环右移，分支到进位位 |
| | RAL | | 1 | × |  累加器A带进位位循环左移 |
| | RAR | | 1 | × |  累加器A带进位位循环右移 |
| 寄存器操作 | DAD | RP | 1 | | [H,L]←[H,L]+[RP]<br>某一寄存器对和H,L相加 |
| | INX | RP | 1 | | [RP]←[RP]+1<br>寄存器RP的内容增1，RP=BC，DE，HL或SP |
| | DCX | RP | 1 | | [RP]←[RP]−1<br>寄存器RP的内容减1 |

续表

| 类　型 | 助记符 | 操作数 | 字　节 | 状　态<br>C Ac Z S P | 完　成　操　作 |
|---|---|---|---|---|---|
| 栈 | PUSH | RP | 1 | | [[SP]]←[RP],[SP]←[SP]−2<br>寄存器 RP 内容压入栈内 |
| | POP | RP | 1 | | [RP]←[[SP]],[SP]←[SP]+2<br>栈内容弹出送寄存器 |
| | XTHL | | 1 | | [H, L]←→[[SP]]<br>H, L和栈顶交换内容 |
| 中　断 | EI | | 1 | | 中断允许(开中断) |
| | DI | | 1 | | 中断禁止(关中断) |
| | RST | | 1 | | 重新启动 |
| 状　态 | STC | | 1 | 1 | [CS]←1<br>进位位置 1 |
| | CMC | | 1 | × | [CS]←$\overline{[CS]}$<br>进位位取反 |
| | NOP | | 1 | | 不操作 |
| | HLT | | 1 | | 暂停 |

RP＝BC, DE或PSW

状态：C＝进位位
Ac＝从第 3 位进位
Z＝零状态
S＝符号位状态
P＝奇偶位状态
×＝状态位置位或复位
0＝状态复位
空白＝状态不变

CS　　进位位状态

Ac　　辅助进位位状态

ZS　　零状态

SS　　符号位状态

PS　　奇偶位状态

I　　指令寄存器

I 2　　第二结果代码字节

I 3　　第三结果代码字节

PC　　程序计数器

SP　　栈指示器

PSW　　程序状态字；如下页所示，其中的一些位被分配给各状态标志。

H　　出现在数字组的尾部（例如 213 AH）表示是十六进制数。

DATA　8 位立即数

DEV　　一个 I/O 设备

DATA16　十六位立即数据

REG　　寄存器 A、B、C、D、E、H 或 L

M　　存储器，由 HL 所隐含的储存地址。

LABEL　一个十六位地址，表示一个指令符号。

RP　　寄存器对：B 表示 BC 寄存器对，D 表示 DE 寄存器对，H 表示 HL 寄存器对，SP 表示栈指示器。

PORT　输入/输出（I/O）口，编号从 0 到 $EF_{16}$。

ADDR　指定一个数据存储器字节的十六位地址。

[ ]　　括号内是该单元的内容

[[]]　　由括号内所标地址来寻址的存储器字节。

←　　按箭头方向传送数据

← → 交换箭头两边单元中的内容。

\+ 加法

— 减法

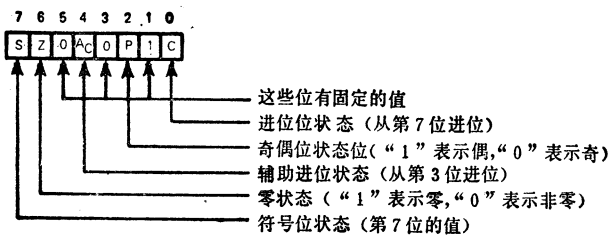
∧ 逻辑乘，“与”

∨ 逻辑加，“或”

∀ 异

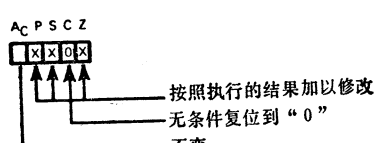
## 6-2 状　　态

五个状态标志存储在程序状态字（PSW）中的情况如下图所示：




程序状态字和累加器有时作为寄存器对处理。

**执行指令对状态的影响说明如下：**




**执行指令所引起的状态的变化**

在各条指令执行的图例中，×表示可以是置位或复位的任意状态，“0”表示总是清零的状态，空白表示该状态不变化。

## 表 6-2 指令结果代码和执行周期摘要

| 指令 | | 结果代码 | 字节数 | 周期数 | 指令 | | 结果代码 | 字节数 | 周期数 |
|---|---|---|---|---|---|---|---|---|---|
| ACI | DATA | CE YY | 2 | 7 | LXI | RP,DATA16 | 00XX0001 YYYY | 3 | 10 |
| ADC | REG | 10001XXX | 1 | 4 | | | | | |
| ADC | M | 8E | 1 | 7 | MOV | REG,REG | 01dddsss | 1 | 5 |
| ADD | REG | 10000XXX | 1 | 4 | MOV | M,REG | 01110sss | 1 | 7 |
| ADD | M | 86 | 1 | 7 | MOV | REG,M | 01ddd110 | 1 | 7 |
| ADI | DATA | C6 YY | 2 | 7 | MVI | REG,DATA | 00ddd110 YY | 2 | 7 |
| ANA | REG | 10100XXX | 1 | 4 | | | | | |
| ANA | M | A6 | 1 | 7 | MVI | M,DATA | 36 YY | 2 | 10 |
| ANI | DATA | E6 YY | 2 | 7 | NOP | | 00 | 1 | 4 |
| CALL | LABEL | CD ppqq | 3 | 17 | ORA | REG | 10110XXX | 1 | 5 |
| CC | LABEL | DC ppqq | 3 | 11/17 | ORA | M | B6 | 1 | 7 |
| CM | LABEL | FC ppqq | 3 | 11/17 | ORI | DATA | F6 YY | 2 | 7 |
| CMA | | 2F | 1 | 4 | OUT | PORT | D3 YY | 2 | 10 |
| CMC | | 3F | 1 | 4 | PCHL | | E9 | 1 | 5 |
| CMP | REG | 10111XXX | 1 | 4 | POP | RP | 11XX0001 | 1 | 10 |
| CMP | M | BE | 1 | 7 | PUSH | RP | 11XX0101 | 1 | 11 |
| CNC | LABEL | D4 ppqq | 3 | 11/17 | RAL | | 17 | 1 | 4 |
| CNZ | LABEL | C4 ppqq | 3 | 11/17 | RAR | | 1F | 1 | 4 |
| CP | LABEL | F4 ppqq | 3 | 11/17 | RC | | D8 | 1 | 5/11 |
| CPE | LABEL | EC ppqq | 3 | 11/17 | RET | | C9 | 1 | 10 |
| CPI | DATA | FE YY | 2 | 7 | RLC | | 07 | 1 | 4 |
| CPO | LABEL | E4 ppqq | 3 | 11/17 | RM | | F8 | 1 | 5/11 |
| CZ | LABEL | CC ppqq | 3 | 11/17 | RNC | | D0 | 1 | 5/11 |
| DAA | | 27 | 1 | 4 | RNZ | | C0 | 1 | 5/11 |
| DAD | RP | 00XX1001 | 1 | 10 | RP | | F0 | 1 | 5/11 |
| DCR | REG | 00XXX101 | 1 | 5 | RPE | | E8 | 1 | 5/11 |
| DCR | M | 35 | 1 | 10 | RPO | | E0 | 1 | 5/11 |
| DCX | RP | 00XX1011 | 1 | 5 | RRC | | 0F | 1 | 4 |
| DI | | F3 | 1 | 4 | RST | N | 11XXX111 | 1 | 11 |
| EI | | FB | 1 | 4 | RZ | | C8 | 1 | 5/11 |
| HLT | | 76 | 1 | 4 | SBB | REG | 10011XXX | 1 | 4 |
| IN | PORT | DB YY | 2 | 10 | SBB | M | 9E | 1 | 7 |
| INR | REG | 00XXX100 | 1 | 5 | SBI | DATA | DE YY | 2 | 7 |
| INR | M | 34 | 1 | 10 | SHLD | ADDR | 22 ppqq | 3 | 16 |
| INX | RP | 00XX0011 | 1 | 5 | SPHL | | F9 | 1 | 5 |
| JC | LABEL | DA ppqq | 3 | 10 | STA | ADDR | 32 ppqq | 3 | 13 |
| JM | LABEL | FA ppqq | 3 | 10 | STAX | RP | 000X0010 | 1 | 7 |
| JMP | LABEL | C3 ppqq | 3 | 10 | STC | | 37 | 1 | 4 |
| JNC | LABEL | D2 ppqq | 3 | 10 | SUB | REG | 10010XXX | 1 | 4 |
| JNZ | LABEL | C2 ppqq | 3 | 10 | SUB | M | 96 | 1 | 7 |
| JP | LABEL | F2 ppqq | 3 | 10 | SUI | DATA | D6 YY | 2 | 7 |
| JPE | LABEL | EA ppqq | 3 | 10 | XCHG | | EB | 1 | 4 |
| JPO | LABEL | E2 ppqq | 3 | 10 | XRA | REG | 10101XXX | 1 | 4 |
| JZ | LABEL | CA ppqq | 3 | 10 | XRA | M | AE | 1 | 7 |
| LDA | ADDR | 3A ppqq | 3 | 13 | XRI | DATA | EE YY | 2 | 7 |
| LDAX | RP | 000X1010 | 1 | 7 | XTHL | | E3 | 1 | 18 |
| LHLD | ADDR | 2A ppqq | 3 | 16 | | | | | |

ppqq 表示四位十六进制数字的存储器地址
YY 表示两位十六进制数字
YYYY 表示四位十六进制数字的数据
X 表示一位任意的二进制数字
ddd 表示用于标示目的地寄存器的任选二进制数字
sss 表示用于标示源寄存器的任选二进制数字

## 6-3 指令的结果代码

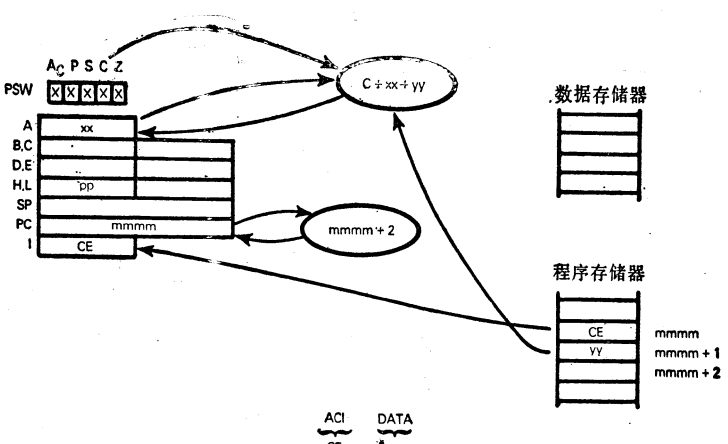
指令结果代码中，对于无变化的指令用二位十六进制数字表示，对于有变化的指令用八位二进制数字表示，从而用二进制数字表示可以标示出变化来。

## 6-4 指令的执行时间和代码

表 6-2 内按字母顺序列出了各条指令的结果代码和用机器周期表示的执行时间。

表内示出了两种指令周期，第一种是对于“条件不成立”的指令周期，而第二种是“条件成立”的指令周期。

## 6-5 ACI——立即数和累加器进行带进位的加法

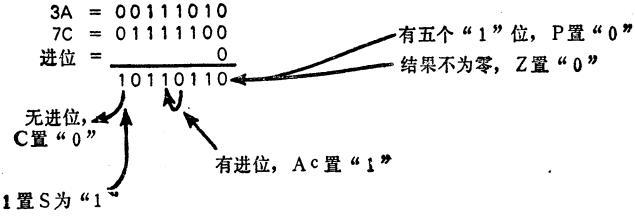



把一个程序存储器字节的内容以及进位状态位和累加器相加。

假定 XX＝3 $A_{16}$，YY＝7 $C_{16}$，C＝0 则执行指令：

ACI　7CH

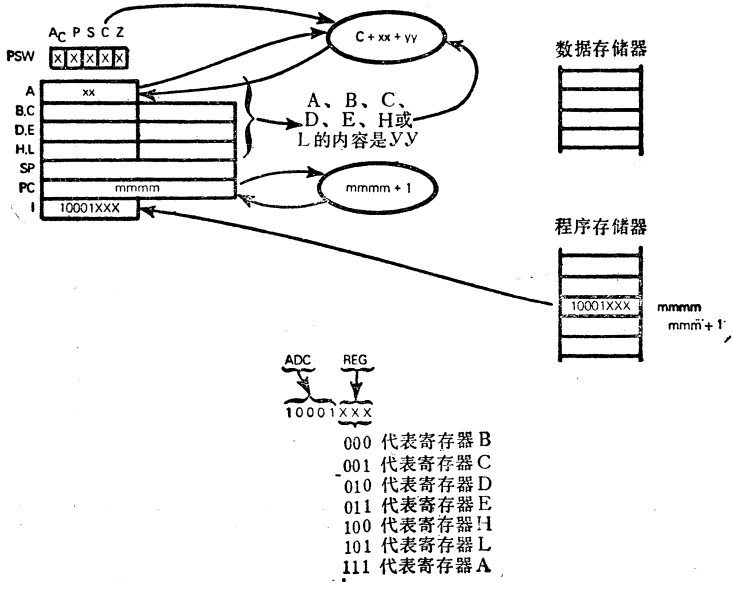
之后，累加器的内容是 $B_6$：




这是一条惯用的数据操作指令

## 6-6　ADC——寄存器或存储器和累加器进行带进位的加法

这条指令有两种形式。首先考虑寄存器和累加器相加。



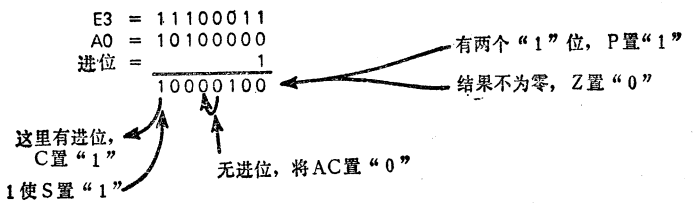

000 代表寄存器 B
001 代表寄存器 C
010 代表寄存器 D
011 代表寄存器 E
100 代表寄存器 H
101 代表寄存器 L
111 代表寄存器 A

累加器和寄存器A，B，C，D，E，H或L的内容以及进位状态位相加。

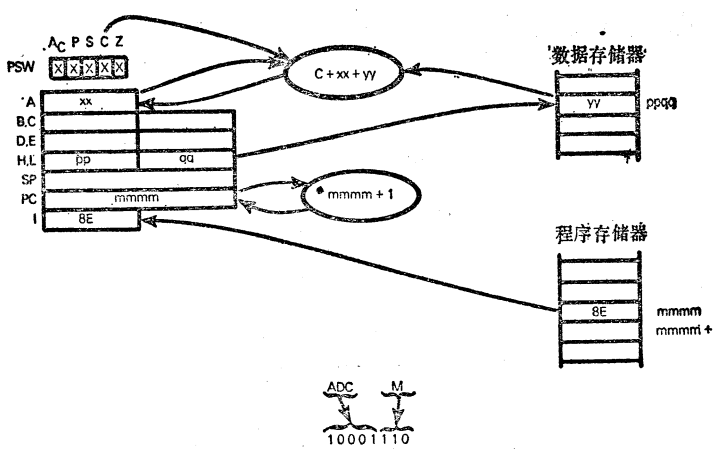
假定XX＝E $3_{16}$，寄存器E的内容为A $0_{16}$，进位位等于1，则执行指令

ADC E

之后，累加器的内容是$84_{16}$




存储器字节的内容也可以和累加器进行带进位的加法。




假如xx＝E $3_{16}$，yy＝A$0_{16}$和C＝1那么执行指令

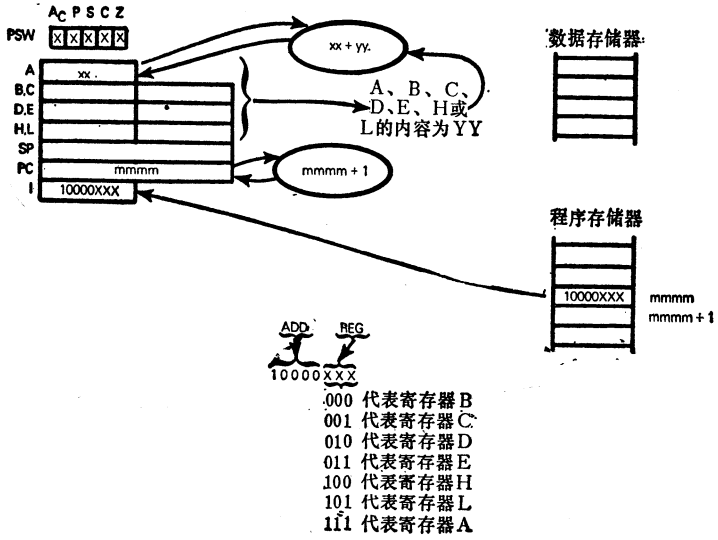
ADC M

和上面所介绍的执行ADC E指令产生的结果相同。

在多字节加法中经常采用ADC指令进行第二字节以及以下各字节的相加。

## 6-7 ADD——寄存器或存储器和累加器相加

这条指令有两种形式。首先考虑把寄存器的内容和累加器内容相加。

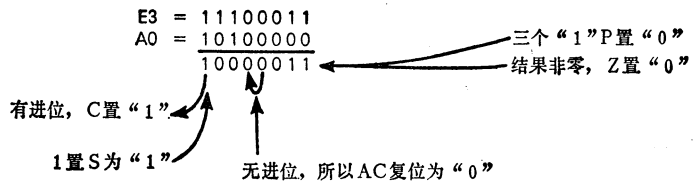



寄存器A,B,C,D,E,H或L的内容和累加器相加。

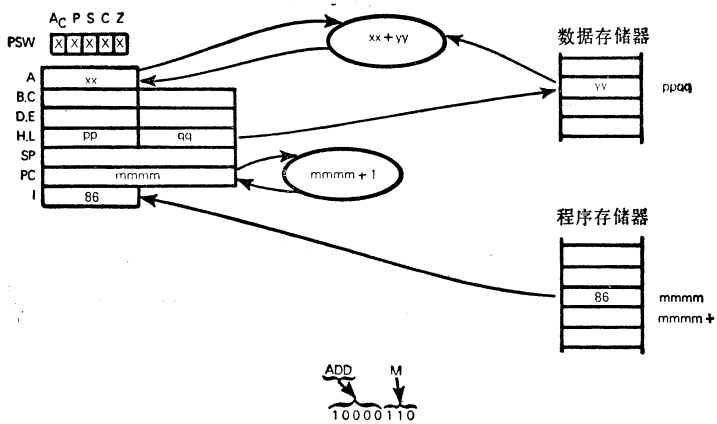
假定xx=$E3_{16}$，寄存器E的内容为$A0_{16}$，进位C=1，则执行指令

ADD E

之后，累加器的内容是$83_{16}$：




存储器字节的内容也可以和累加器相加。




假如 $xx=E3_{16}$，$YY=A0_{16}$，和 $C=1$，那么，执行指令

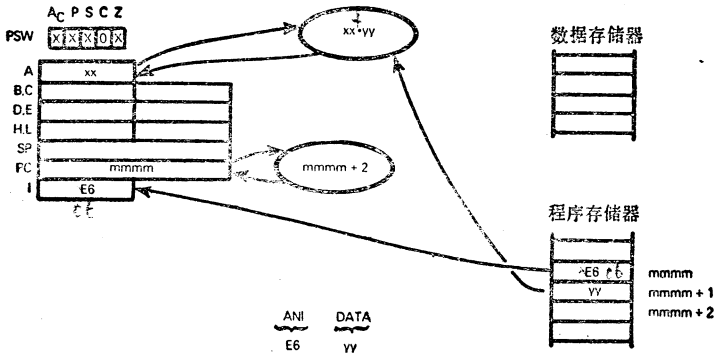
ADD M

与上面介绍的执行指令 ADD E 所产生的结果相同。

ADD 是正常使用的单字节操作的二进制加法指令。这条指令也用于两个多字节数的低位字节的相加。

## 6-8 ADI——立即数和累加器相加

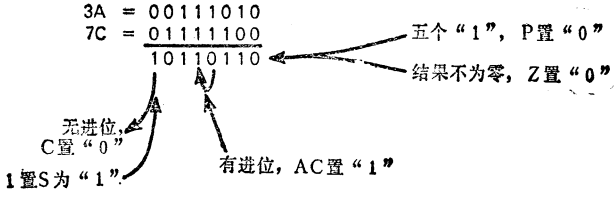
程序存储器下一个字节的内容和累加器的内容相加。




假定 XX＝3 $A_{16}$，YY＝7 $C_{16}$，C＝0，则执行指令

AD1　7CH

之后，累加器的内容是B6：

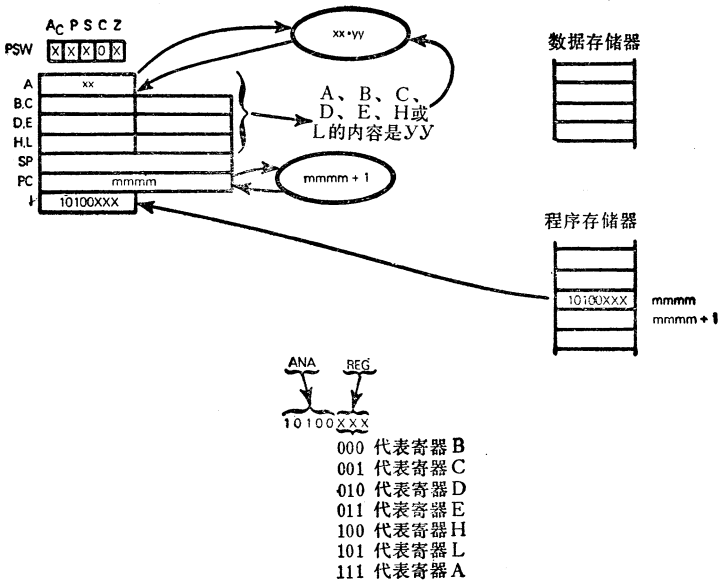



这是一个例行的数据操作指令。

## 6-9　ANA——寄存器或存储器和累加器相“与”

这条指令有两种形式，首先考虑寄存器的内容和累加器的内容相“与”

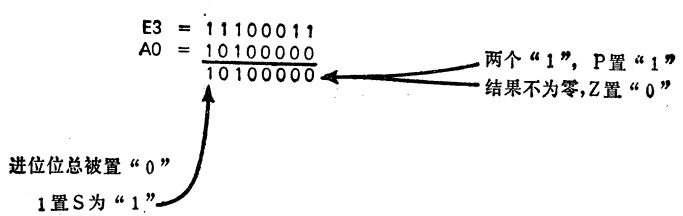
寄存器A，B，C，D，E，H或L的内容和累加器的内容相“与”，结果保存在累加器中。



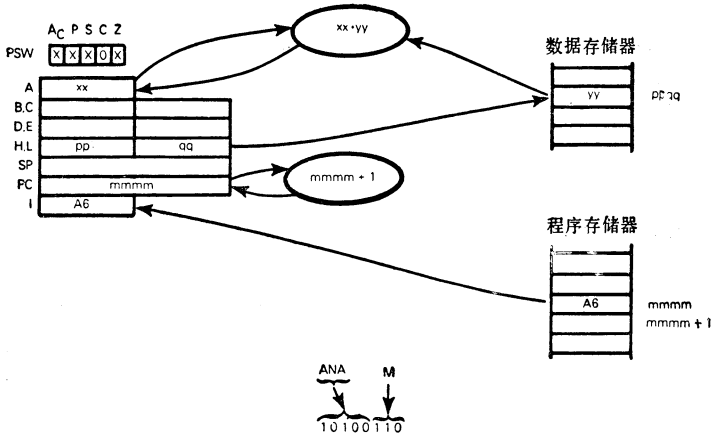

假定寄存器 E 的内容是 $A0_{16}$，则执行指令

ANA E

之后，累加器的内容是 $A0_{16}$：




存储器字节的内容也可以和累加器的内容相“与”。

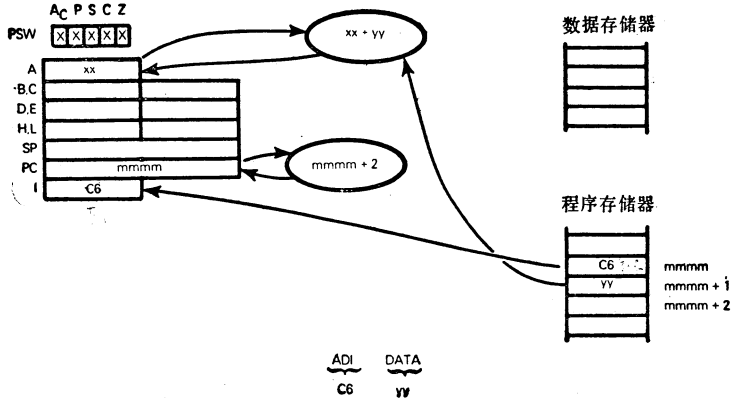



若 XX＝E3$_{16}$，YY＝A0$_{16}$ 和 C＝1，那么执行指令

ANA　M

同刚才介绍的执行指令 ANA　E 所产生的结果相同。ANA 常常用作逻辑操作指令。

## 6-10　ANI——立即数和累加器相“与”

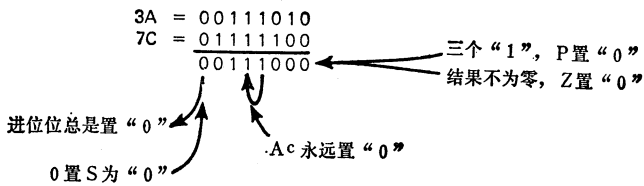


累加器的内容和下一个程序存储器的内容相“与”。

假定 XX＝ 3 $A_{16}$，YY＝ 7 $C_{16}$，则执行指令

ANI　7 CH

之后，累加器的内容是 $38_{16}$。

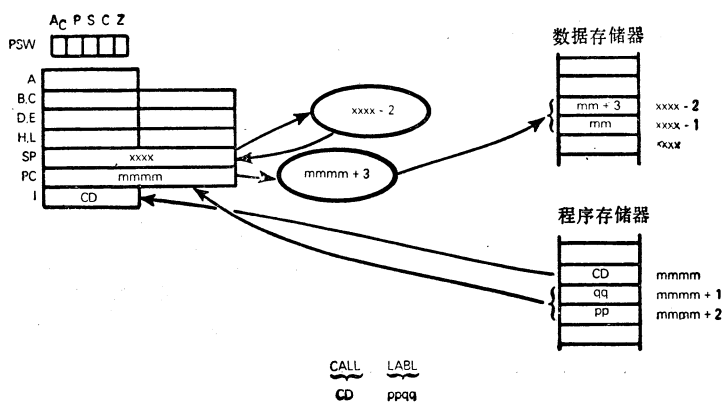



这是一条例行的逻辑指令，这条指令经常用于使某几位置“0”，例如指令

ANI　7 FH

将无条件地使累加器的最高位置“0”。

## 6-11　CALL——调用由操作数标示的子程序



把紧接在CALL后面的指令地址存放到栈顶；栈顶是由栈指示器所存取的数据存储器字节。然后使栈指示器减2以形成新的栈顶地址。把在CALL指令中的第二和第三结果代码字节内包含的16位地址内容送到程序计数器。

我们来探讨一下如下的指令序列：

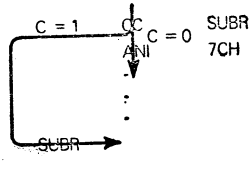
```
      CALL    SUBR
      ANI     7CH
      •

SUBR
```

在CALL指令执行之后，ANI指令地址保留在栈顶，栈指示器减2，接着执行下一条标号为SUBR的指令。

## 6-12 CC——调用由操作数标示的子程序，但仅当进位位状态等于1时才调用

CC
DC

我们来看如下的指令序列



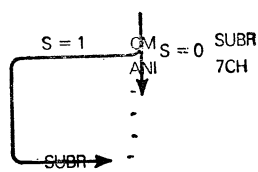

在执行了CC指令以后，假如进位位状态不等于1就执行ANI指令；如进位位状态等于1，ANI指令的地址就被保存在栈顶。栈指示器内容减2。下面执行标号为SUBR的那条指令。

## 6-13 CM——调用由操作数标示的子程序，但仅当符号状态位等于1时才调用

CM
FC

除了仅当符号状态位等于1时才调用被指定的子程序，否则将顺序执行CM指令以下的指令之外，这条指令与CALL指令相同。

我们来看如下的指令序列：




在执行CM指令以后，若符号位状态等于1就把ANI指令的地址保存在栈顶。栈指示器内容减2，下面再执行标号为SUBR的那条指令。

## 6-14 CMA——累加器内容取反

对累加器的内容取反，不影响其它寄存器的内容和状态位

假定累加器的内容为$3A_{16}$，则执行指令：

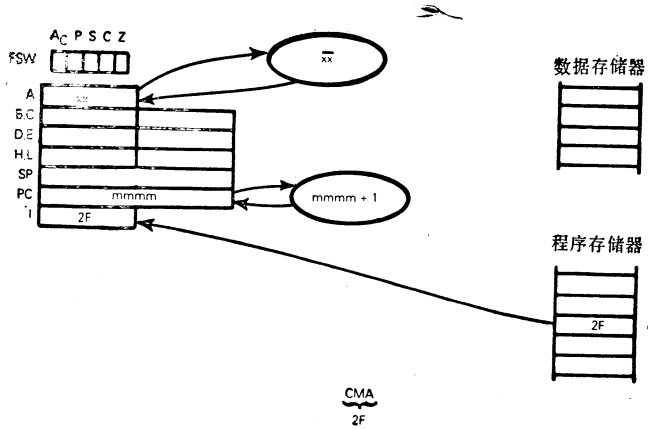
CMA

以后，累加器的内容是$C5_{16}$：

$3A_{16}$＝0 0 1 1 1 0 1 0

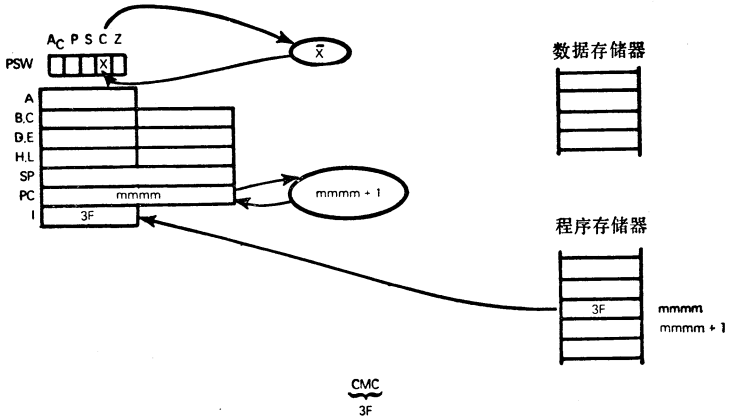
反码＝1 1 0 0 0 1 0 1

这是一条常用的逻辑指令。不要把它用于二进制的减法。




因为已有专门的减法指令(SUB 和 SBB)

## 6-15 CMC——进位位状态取反




进位位状态取反，不影响其它各状态位或寄存器的内容。

假定进位位状态的内容是 1，则执行指令

CMC

以后，进位位状态的内容将为“0”。

经常利用这条指令通过下面的指令序列强制进位位状态置“0”。

```
STC  :SET CARRY STATUS TO 1
CMC  :COMPLEMENT CARRY STATUS
```

STC　置进位位状态为“1”

CMC　进位位状态取反

应该注意，用下列指令

ANA A

能使进位位状态自动置“0”，又因为累加器是和它本身内容相“与”，所以不会改变任何寄存器的内容。指令

ORA A

也适合于同一用途。

## 6-16 CMP——寄存器或存储器和累加器相比较

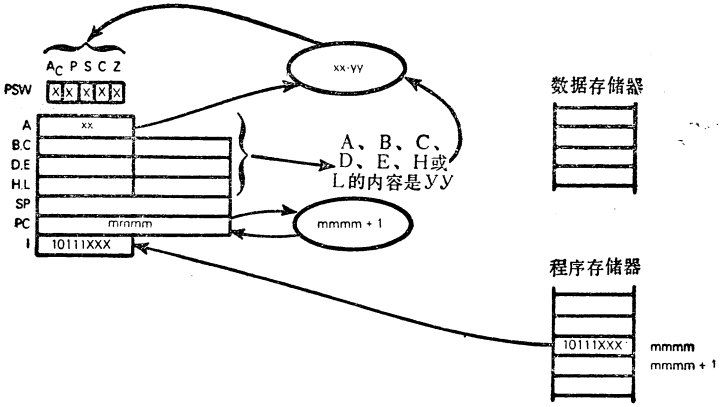
这条指令有两种形式。首先考察寄存器的内容和累加器相比较。

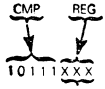
从累加器的内容减去寄存器A，B，C，D，H或L的内容。这两个数都是当作简单的二进制数据处理的。丢掉结果，只保留累加器的内容，但是修改相应的状态位使它们反映出相减的结果。

假定 $XX=E3_{16}$，寄存器E的内容是$A0_{16}$，则执行指令：

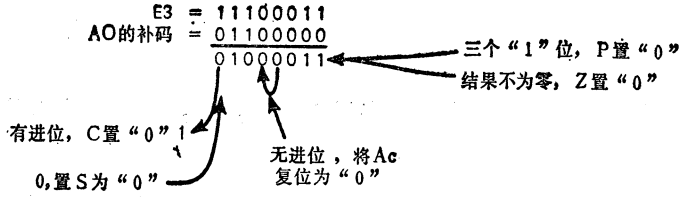
CMP E

以后，累加器的内容仍是$E3_{16}$，但是状态位将作如下变更：应当注意，所得进位被取反。




000 代表寄存器B
001 代表寄存器C
010 代表寄存器D
011 代表寄存器E
100 代表寄存器H
101 代表寄存器L
111 代表寄存器A

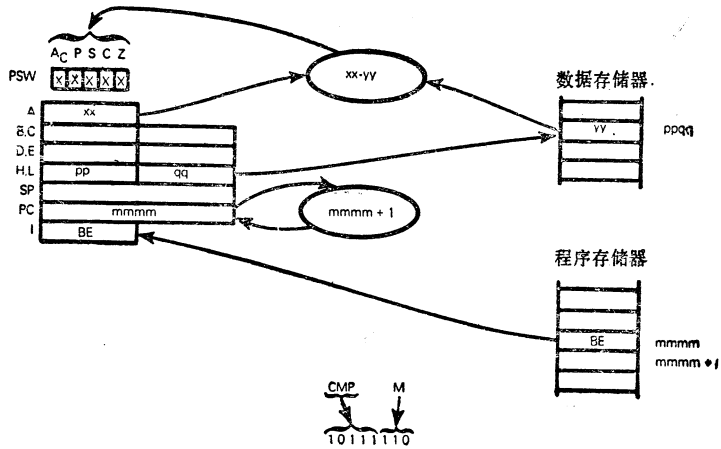



存储器字节的内容也可以和累加器的内容相比较：

假定 $xx=E3_{16}$，和 $yy=A0_{16}$，则

CMP M

指令的执行和刚才所描述的CMP E指令产生的结果相同。



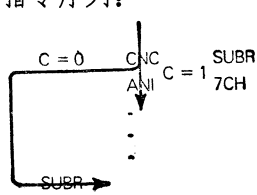

比较指令常常出现在条件调用，返回和转移指令的前面，立即数比较(CPI)指令比 CMP 指令更为有用。

## 6-17 CNC-调用由操作数标示的子程序，但仅当进位位状态等于 0 时才调用

CNC
D4

这条指令仅当进位位状态等于 0 时才调用所标示的子程序；否则，顺序执行 CNC 的下一条指令，其它与 CALL 指令相同。

我们来看下面的指令序列：



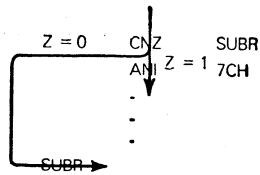
在执行了CNC指令以后，如进位位状态不等于0，就执行ANI指令，如进位位状态等于0，则ANI指令的地址保存在栈顶。下面将执行标号为SUBR的指令。

## 6-18 CNZ——调用由操作数标示的子程序，但仅当零状态等于0时才调用

CNZ
C4

这条指令仅当零状态等于0时才调用所标示的子程序，否则顺序执行CNZ的下一条指令，其它与CALL指令等同。

我们来看如下的指令序列：




在执行了CNZ指令以后，若零状态位不等于0，将执行ANI指令。若零状态位等于0，则ANI指令地址保存在栈顶。栈指示器减2，接着执行由SUBR所标示的指令。

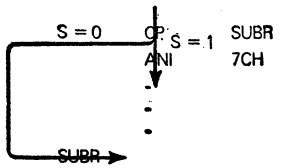
## 6-19 CP-调用由操作数标示的子程序，但仅当符号位状态等于0时才调用。

CP
F4

这条指令仅当符号位状态等于0时才调用所指示的子程序，否则将顺序执行CP指令下面的那条指令，其它与CALL

指令等同。

我们来看如下的指令序列：



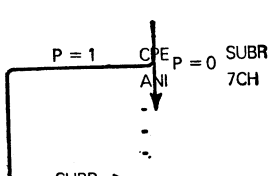

在执行了CP指令以后，若符号位状态不等于0，则执行ANI指令。若符号位状态等于0，则ANI指令地址保存在栈顶，栈指示器减2，将执行下一条由SUBR标示的指令。

## 6-20 CPE——调用由操作数标示的子程序，但仅当奇偶位状态等于1时才调用

CPE
EC

这条指令仅当奇偶位状态等于1时才调用所标示的子程序，否则将顺序执行CPE下面的那条指令，其余与CALL指令相同。

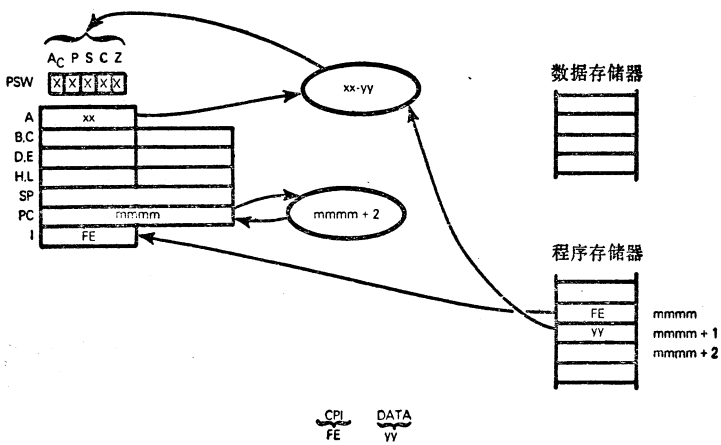
考虑如下的指令序列：




在执行CPE指令之后，若奇偶位状态不等于1，就执行ANI指令。若奇偶位状态等于1，ANI指令的地址保存在栈

顶。栈指示器减 2，下面将执行标号是 SUBR 的指令。

## 6-21 CPI——立即数与累加器内容相比较



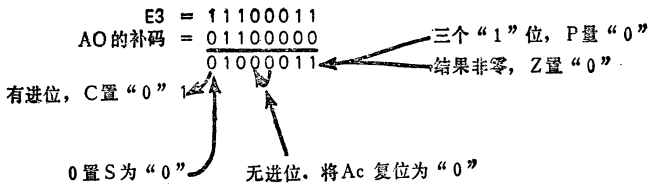

从累加器的内容减去第二个结果代码字节的内容，这两个数都简单地当做二进制数据处理的，丢掉其结果，只保留累加器内容。但是需要修改相应的状态标志以便反映相减的结果。

假定 xx＝E $3_{16}$，CPI 指令结果代码第二个字节为A $0_{16}$，则执行指令

CP 1 A0 H

之后，累加器内容是 E $3_{16}$，但是状态位将作如下的变更。注意所得进位值被取反。

这条指令最经常用于执行条件调用返回或转移指令之前置状态标志。

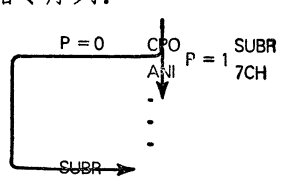



## 6-22 CPO——调用由操作数标示的子程序，但仅当奇偶位状态等于0时才调用

CPO
E4

这条指令仅当奇偶位状态等于0时才调用所标示的子程序，否则将顺序执行CPO下面的那条指令，除此以外，与CALL指令等同。

我们来看下面的指令序列：



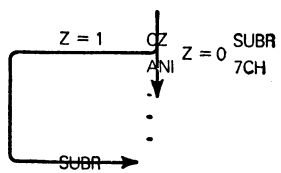

在执行CPO指令之后，若奇偶位状态不等于0，则执行ANI指令；若奇偶位状态等于0，则ANI指令的地址保存在栈顶。同时栈指示器减2，接着执行由SUBR所标示的指令。

## 6-23 CZ——调用由操作数标示的子程序，但仅当零状态位等于1时才调用

CZ
CC

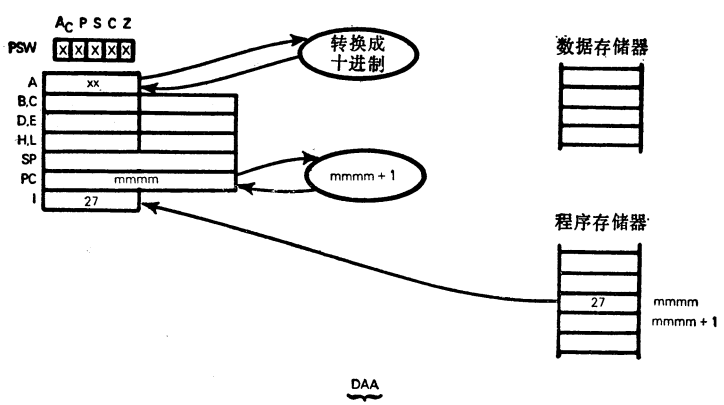
这条指令仅当零状态位等于1时才调用所标示的子程序，否则将顺序执行CZ指令下面的指令，除此以外与CALL指令相同。

我们来看下面的指令序列：




在执行了CZ指令之后，若零状态位不等于1，即执行ANI指令。若零状态位等于1，ANI指令的地址保存在栈顶。栈指示器减2，接着执行标号为SUBR的指令。

## 6-24 DAA——十进制调整累加器




把累加器内容从二进制码转换为二——十进制码的形式。仅当两个BCD数字相加后才能用这条指令。也就是把ADD

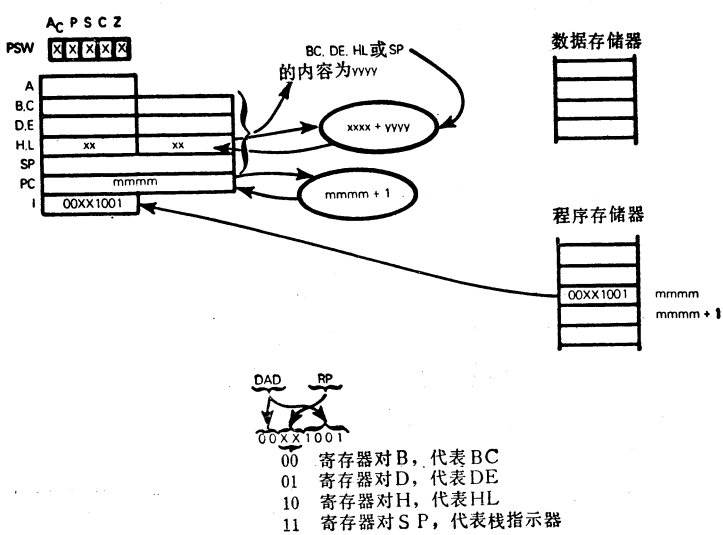
DAA 或 ADC DAA 或 SUB DAA 或 SBB DAA 当作是十进制数运算的复合指令。它们对 BCD 数的源进行运算后产生 BCD 的答数。

假定累加器的内容是$39_{16}$，寄存器 B 的内容是 $47_{16}$。则执行指令

```
ADD  B
DAA
```

以后，累加器的内容将是 $86_{16}$ 而不是 $80_{16}$。DAA 指令修改了全部状态标志位，但仅仅进位状态是有意义的，可以认为其它各状态位都被破坏了。

## 6-25 DAD——寄存器对和 H，L 相加

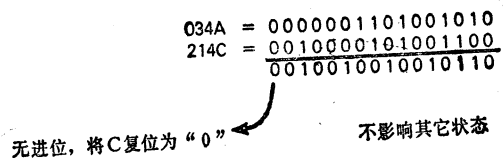


BC，DE或HL寄存器对或者栈指示器的16位值和HL寄存器对相加。

假定H.L的内容是034 $A_{16}$ B.C的内容是214 $C_{16}$，则执行指令

DAD B

以后，寄存器对HL的内容将是$2496_{16}$。




DAD指令是在8080指令系统中对于传统的程序编制最有用的指令之一，这条指令是与变址寻址等效的，而DADH指令等效于16位的向左移位。

## 6-26 DCR——寄存器或存储器的内容减1

从所指定的寄存器的内容中减1

假定寄存器$C$的内容是$3A_{16}$，则执行指令

DCR C

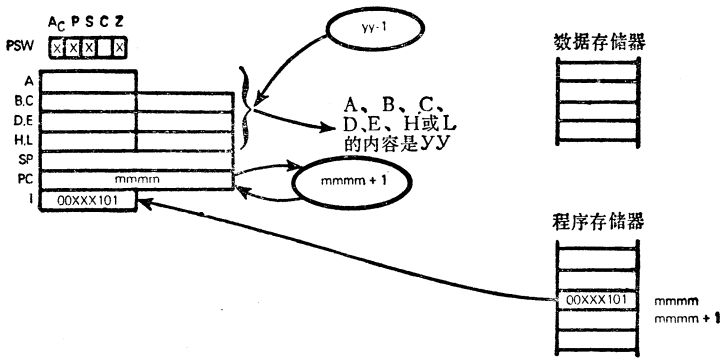
之后，寄存器C的内容将是$39_{16}$。

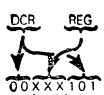
读/写存储器字节的内容也可以减1。

假定HL的内容是$3714_{16}$，则执行指令

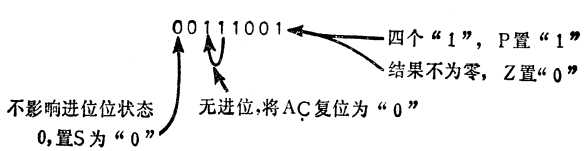
DCB M

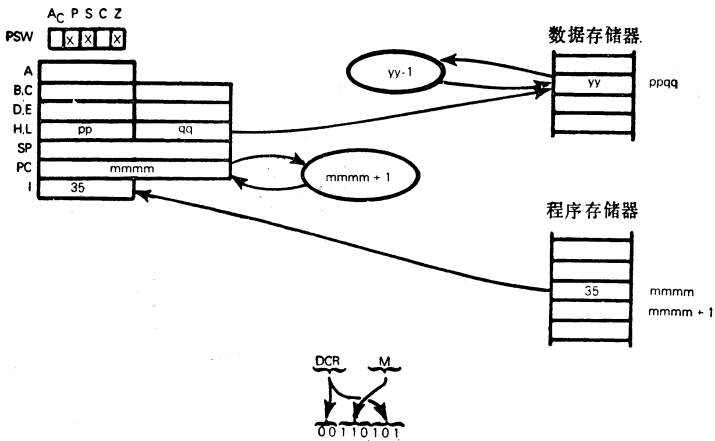
从地址为$3714_{16}$的存储器字节的内容中减1。状态标志的修改和DCR C指令相同。




000 代表寄存器 B
001 代表寄存器 C
010 代表寄存器 D
011 代表寄存器 E
100 代表寄存器 H
101 代表寄存器 L
111 代表寄存器 A






DCR 指令用于计数器值等于或小于 256 的重复指令循环中。典型的循环方式如下：

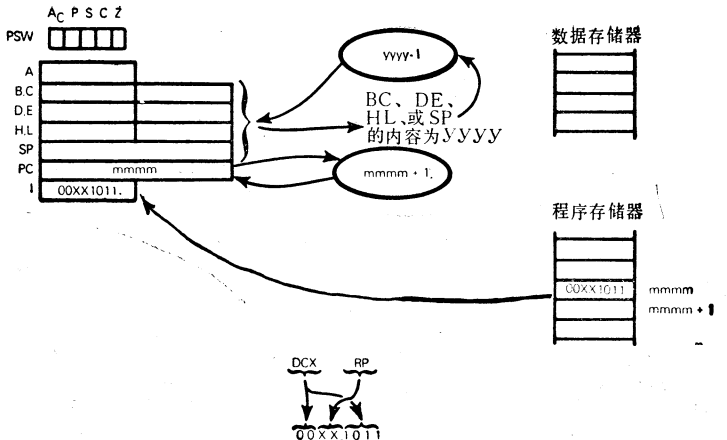
```
        MVI     REG,DATA    ;LOAD INITIAL COUNTER VALUE
LOOP    -                   ;FIRST INSTRUCTION OF LOOP
        -
        -
        -
        DCR     REG         ;DECREMENT COUNTER
        JNZ     LOOP        ;RETURN IF NOT ZERO
```

```
        MVI   REG   DATA   计数器置初值
LOOP    -                  循环的第一条指令
        -
        -
        -
        DCR   REG          计数器减 1
        JNZ   LOOP         若不为零，返回
```

## 6-27 DCX——寄存器对减 1

指定的寄存器对中的16位值减 1 。

假定栈指示器的内容为 $2F7A_{16}$，则执行指令：




00为寄存器对B，表示B，C
01为寄存器对D，表示D，E
10为寄存器对H，表示H，L
11为寄存器对SP，表示栈指示器

**DCX SP**

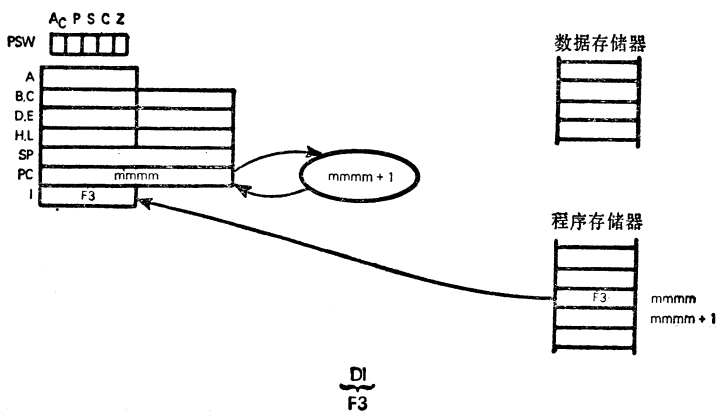
之后，栈指示器的内容是 $2F79_{16}$。

DCX 指令不能修改任何状态标志，这是 8080 指令系统的一个缺点。DCR 指令用于计数器值等于或小于 256 的重复的指令循环。而 DCX 指令必须在计数值大于 256 的情况下才使用。由于 DCX 指令不置任何状态标志，因此仅仅为了测试一下结果是否为零就要增加几条指令。下面是一个典型的循环方式：

```
        LXI     D,DATA  ;LOAD INITIAL 16-BIT COUNTER VALUE
LOOP    -       -       ;FIRST INSTRUCTION OF LOOP
        -       -
        -       -
        DCX     D       ;DECREMENT COUNTER
        MOV     A,D     ;TO TEST FOR ZERO, MOVE D TO A
        ORA     E       ;THEN OR A WITH E
        JNZ     LOOP    ;RETURN IF NOT ZERO
```

```
        L×1   D.DATA   计数器置初值
LOOP    -     -        循环的第一条指令。
        -     -
        -     -
        DCX   D        计数器减1
        MOV   A.D      为了测试结果是否为0把D移入A
        ORA   E        然后A和E相"或"
        JNZ   LOOP     如不为0，返回
```

## 6-28 DI——禁止中断(关中断)

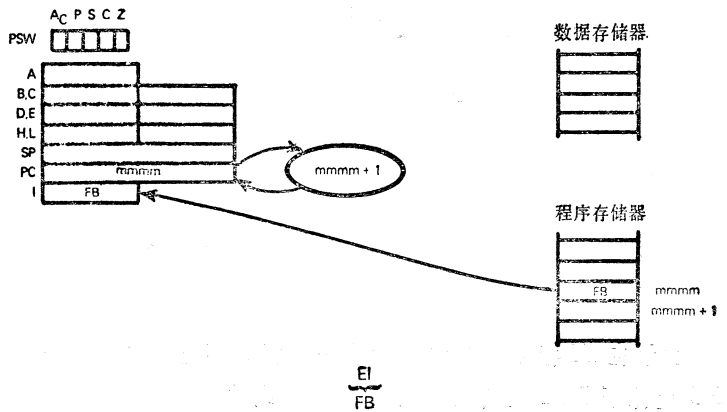



在执行这条指令以后，由8080 CPU输出低电平的 INTE 信号，不允许中断请求。各寄存器或状态标志不受影响。

应该记住，当已接受某一中断时，就自动地禁止中断了。

## 6-29 EI——允许中断(开中断)

当这条指令被执行以后，才允许中断。但必须等到上一条指令执行完毕才能执行中断。




大多数中断服务程序是用以下两条指令来结束的。

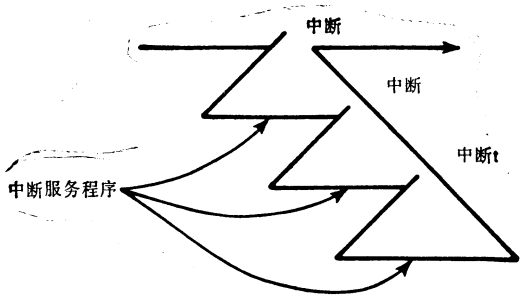
```
EI      ;ENABLE INTERRUPTS.
RET     ;RETURN TO INTERRUPTED PROGRAM.
```

EI　开中断

RET　返回被中断的程序

如果中断是串行处理的话，在中断服务程序的整个持续时间内，不允许所有其它的中断。这意味着在多重中断的应用中，当任何一种中断服务程序执行完毕时，很有可能有一个或几个中断源正在等待服务。

如果 EI 指令一经执行，马上就允许中断的话，那么就不可能执行返回指令。在这种情况下返回地址就一个接一个地叠放在栈中，因而不必要地浪费了栈的存储空间。这一情况可以用图说明如下：

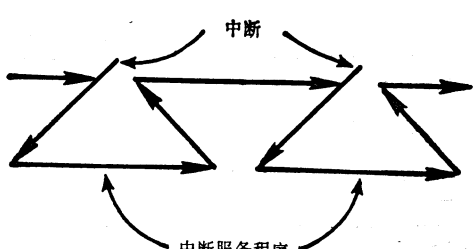


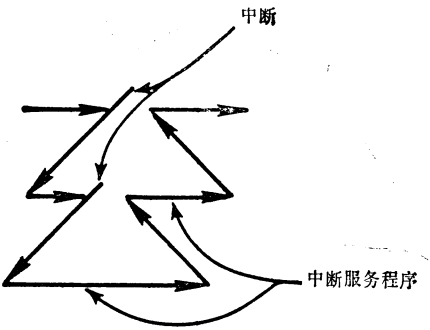

由于在EI指令执行之后为下一条指令禁止了中断，8080 CPU能保证RET指令在指令序列中得到执行：

```
EI        ;ENABLE INTERRUPTS
RET       ;RETURN FROM INTERRUPT
```

```
-
-
-
EI    开中断
RET   从中断返回
```

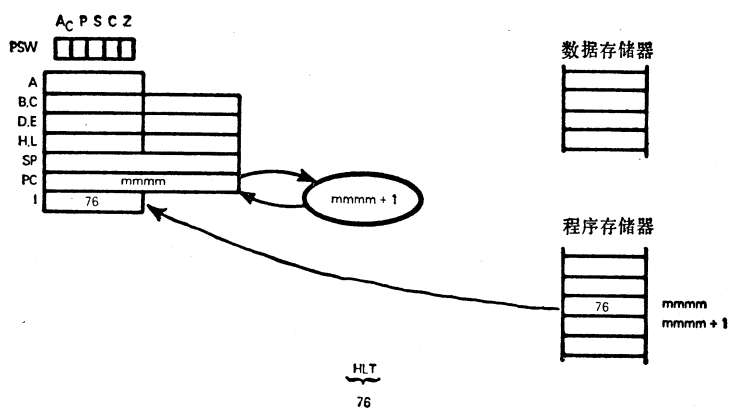
当执行中断服务程序时，保持关中断的方式是普遍的。中断是串行处理的，见下图：






假如中断是按照串行方式处理的话，那么只有在响应中断的过程中才能判断优先级。

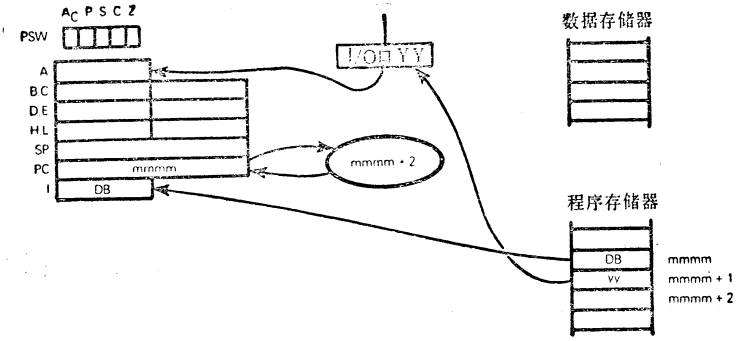
## 6-30 HLT——暂停




当执行 HLT 指令时，停止执行程序，需要一个中断或复位才能重新执行程序。各寄存器或状态位不受影响。

注意：如果在 HALT 指令以前没有用 EI 指令开中断，则除非通过硬件复位，否则8080 将不能退出暂停状态。

## 6-31 IN——输入累加器




从IN指令结果代码第二个字节所标示的I/O口送一个数据字节到累加器。

假如$36_{16}$保存在I/O口$1A_{16}$的缓冲器中，则执行指令IN IAH之后，累加器内容成为$36_{16}$，输入指令并不影响任何状态位。

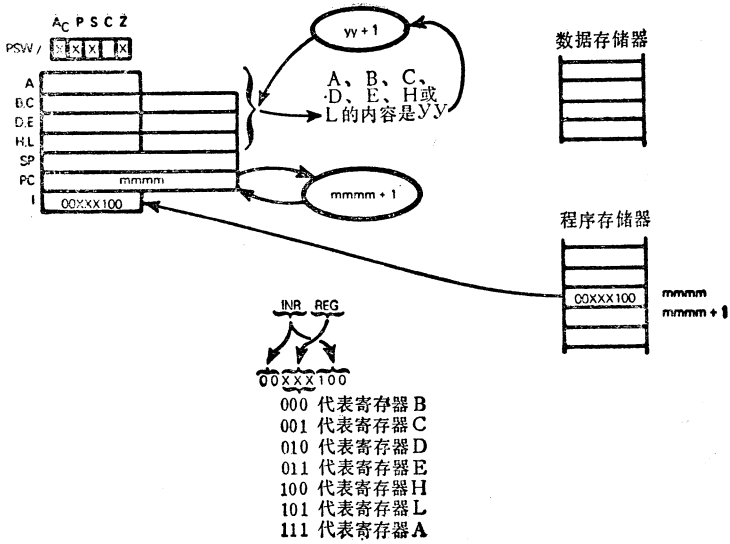
使用IN指令和硬件的关系是很密切的，有效的I/O口地址取决于I/O逻辑的实现方法，也有可能设计这样一种微型计算机系统，它是采用具有专用存储器地址的存储器访问指令来存取外部逻辑设备的。

## 6-32 INR——寄存器或存储器内容增1

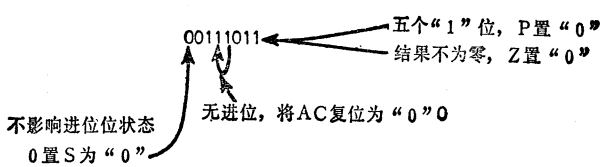
指定寄存器的内容增1，

假定寄存器C的内容是$3A_{16}$，则执行指令

INR C




之后，寄存器 C 的内容为 $3B_{16}$：

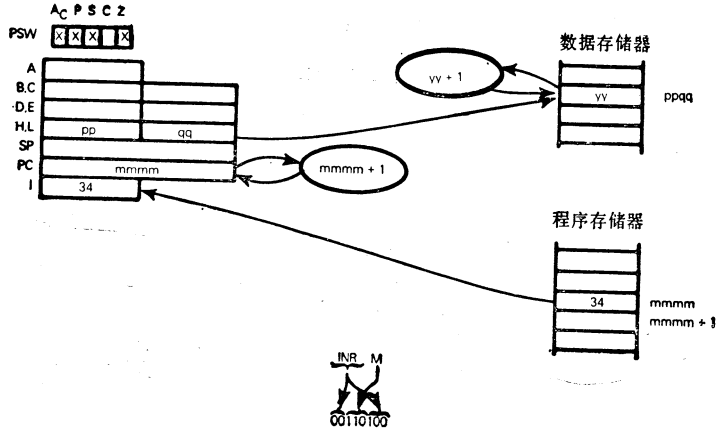



读/写存储器字节的内容也可以增 1：

假定寄存器 HL 的内容是 $3714_{16}$，则执行指令

INR M

后使地址为 $3714_{16}$ 的存储器字节内容增 1，状态标志的改变与对于 INR C 指令介绍的相同。




INR 指令用于计数器值等于或小于 256 的重复指令循环中。下面示出了典型的循环方式：

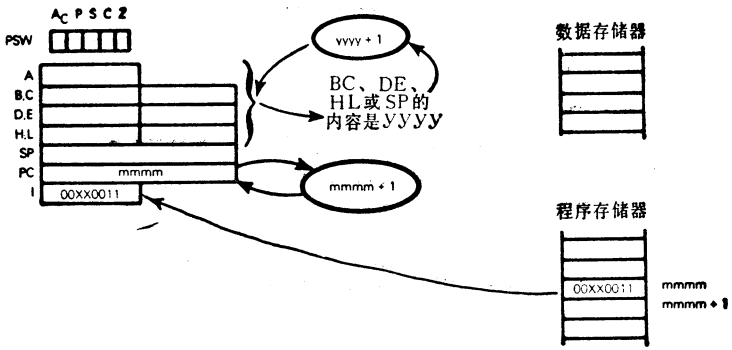
```
        MVI     REG,DATA    ;LOAD COMPLEMENT OF INITIAL COUNTER VALUE
LOOP    -       -           ;FIRST INSTRUCTION OF LOOP
        —       —
        —       —
        INR     REG         ;INCREMENT COUNTER
        JNZ     LOOP        ;RETURN IF NOT ZERO
```

```
        MVI   REG, DATA   将初始计数值的补码送累加器
LOOP    —     —           循环的第一条指令
        —     —
        —     —
        INR   REG         计数器值增 1
        JNZ   LOOP        如不为 0，返回
```

## 6-33 INX——寄存器对增 1

将指定寄存器对的 16 位值增 1。

假定寄存器 D 和 E 的内容是 $2F7A_{16}$，则执行指令





00 为表示B，C
01 为表示D，E
10 为表示H，L
11 为表示栈指示器

INX D

以后，寄存器D和E的内容为2F7B$_{16}$。

INX指令不改变任何状态标志。这是8080指令系统的缺点。DCR指令用于计数值等于或小于256的重复指令循环中，而INX指令却用于计数值大于256的情况。因为INX指令不对状态标志发生影响，所以仅仅为了测试结果是否为零也必须增加几条指令。下面是一个典型循环的形式：

```
        LXI     D,DATA      ;LOAD COMPLEMENT OF INITIAL 16-BIT COUNTER
                            VALUE
LOOP    -       -           ;FIRST INSTRUCTION OF LOOP
        -       -
        -       -
        INX     D           ;INCREMENT COUNTER
        MOV     A,D         ;TO TEST FOR ZERO, MOVE D TO A
        ORA     E           ;THEN OR A WITH E
        JNZ     LOOP        ;RETURN IF NOT ZERO
```

LXI D, DATA 向计数器置入16位初始值的补码

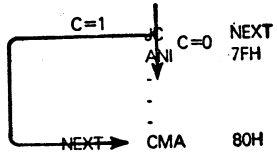
```
LOOP  —    —      循环的第一条指令
      —    —
      —    —
      INX  D      计数器增1
      MOV  A, D   为了测试结果是否为零，将D移入A
      ORA  E      然后A和E相或
      JNZ  LOOP   如不为0，返回
```

## 6-34 JC-若有进位，转移

JC
DA

这条指令和JMP指令等同。但仅当进位状态等于1时才执行转移，否则执行下一条指令

例如下列指令的序列：



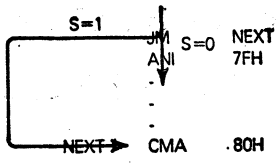

执行了JC指令以后，如果进位位状态等于1，则执行CMA指令；如果进位位状态等于0，则执行ANI指令。

## 6-35 JM——若为负，转移

JM
FA

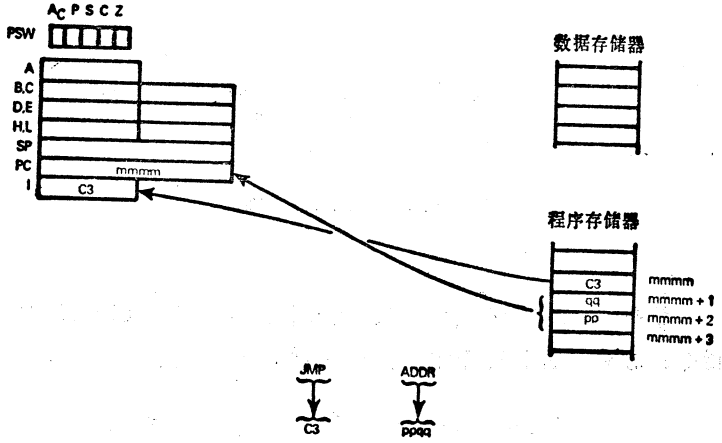
这条指令和JMP指令等同，但仅当符号位状态等于1时才执行转移，否则执行下一条指令。

例如下列指令的序列：



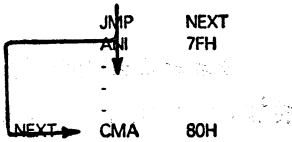

执行 JM 指令以后，若符号位状态等于 1 则执行 CMA 指令；若符号位状态等于 0 则执行 ANI 指令。

## 6-36 JMP——转移到由操作数标示的指令




将转移指令结果代码的第二和第三个字节的内容送入程序计数器；这就成为要执行的下一条指令的存储器地址。程序计数器原先的内容被丢失。

例如下列指令序列：




在执行 JMP 指令后，将执行 CMA 指令，而不再执行ANI 指令。除非在指令序列中有一条转移指令

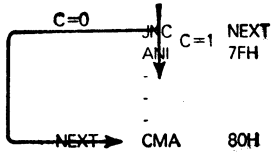
转移到 ANI 指令，否则这条指令是不会被执行的。

## 6-37 JNC——若无进位，转移

JNC
D2

这条指令和 JMP 指令等同，但仅当进位位状态等于 0 时才转移，否则执行下条指令。

例如下列指令序列：



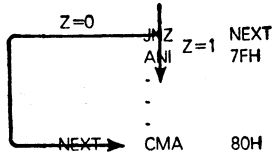

执行 JNC 指令以后，若进位位状态等于 0，则执行 CMA 指令；若进位位状态等于 1，则执行 ANI 指令。

## 6-38 JNZ——若不为零，转移

JNZ
C2

这条指令和 JMP 指令等同，但仅当零状态位等于 0 时才转移，否则执行下条指令。

例如下列指令序列：

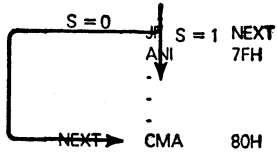



执行JNZ指令以后，若零状态位等于0，则执行CMA指令；若零状态位等于1，则执行ANI指令。

## 6-39 JP——若为正，转移


这条指令和JMP指令等同，但仅当符号位状态等于0时才转移，否则执行下一条指令。

例如下列指令序列：

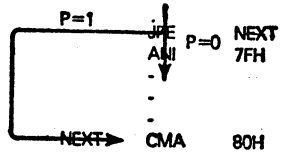



在执行JP指令以后，若符号位状态等于0，则执行CMA指令；若符号位状态等于1，则执行ANI指令。

## 6-40 JPE——若奇偶位状态为偶，转移


这一指令和JMP指令等同，但仅当奇偶位状态等于1时才执行转移，否则执行下一条指令。

例如下列指令的序列：

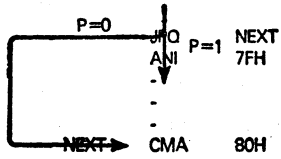



在执行 JPE 指令以后，若奇偶位状态等于 1，则执行CMA 指令；若奇偶位状态等于 0，则执行 ANI 指令。

## 6-41 JPO——若奇偶位状态为奇，转移


此指令和 JMP 指令等同，但仅当奇偶位状态等于 0 时才执行转移。

例如下列指令序列：

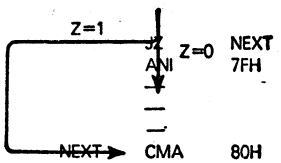



在执行 JPO 指令以后，如奇偶位状态等于 0，则执行 CMA 指令，若奇偶位状态等于 1，则执行 ANI 指令。

## 6-42 JZ——若结果为零，转移

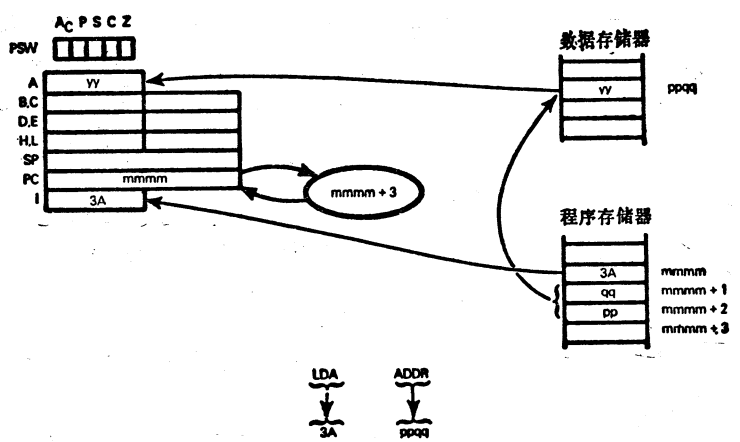
本指令和JMP指令等同，但仅当零状态位等于1时才执行转移，否则执行下条指令。

例如下列指令序列：




在执行JZ指令以后，若零状态位等于1，则执行CMA指令，若零状态位等于0，则执行ANI指令。

## 6-43 LDA——将直接寻址的存储单元内容送累加器




LDA指令结果代码的第二和第三个字节直接寻址的存储器字节的内容送入累加器。

假定存储器字节084 $A_{16}$的内容为3 $A_{16}$，则执行指令

```
LABEL   EQU     084AH
        .
        .
        .
        LDA     LABEL
```

之后，累加器的内容将是 3 $A_{16}$。

应该记住，EQU 是汇编命令，它不是一条指令，无论何处出现 LABEL 时，这个命令都告诉汇编程序采用 16 位数值 084 $A_{16}$。

指令　　LDA　LABEL

是与下面两条指令等效的

```
LXI     H,LABEL
MOV     A,M
```

当你要从存储器中取出一个简单的数据值时，用 LDA 指令是较为合适的。使用它只需要一条指令和三个目标程序字节。而用 LXI、MOV 指令组合起来需要两条指令和四个结果代码字节。另外，LXI、MOV 组合指令要用到寄存器 H 和 L，而 LDA 指令则不是这样。

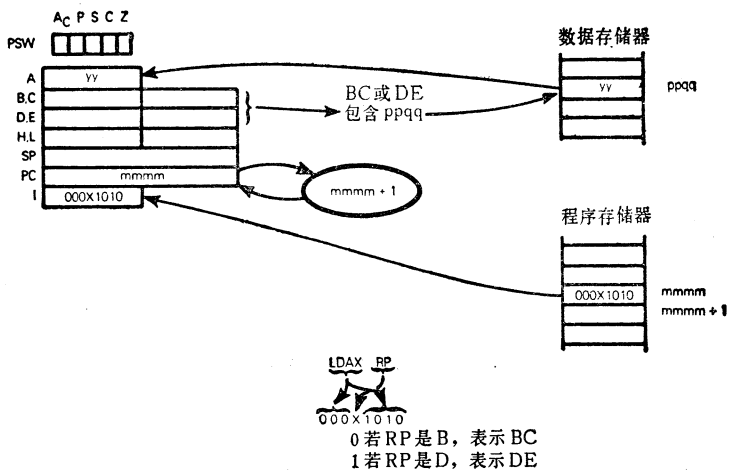
## 6-44　LDAX——由寄存器对寻址的存储单元内容送累加器

取出由 BC 或 DE 寄存器对寻址的存储器字节的内容送入累加器。

假定寄存器 B 的内容为 $08_{16}$，寄存器 C 的内容为 4 $A_{16}$，而存储器字节 084 $A_{16}$ 的内容是 3 $A_{16}$，则执行指令

LDAX　B

以后，累加器的内容是 3 $A_{16}$。




注意，这 LDAX H 指令是不存在的，因为它与 MOV、A．M 指令相似。一般，LDAX 和 LXI 指令是联合使用的，因为 LXI 指令取出一个 16 位地址送到 BC 或 DE 寄存器，即

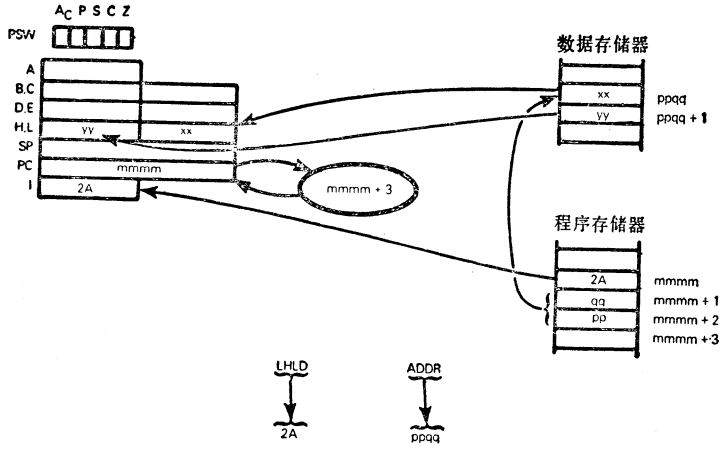
```
LXI     B,084AH
LDAX    B
```

应该注意，LDAX 指令仅能将存储单元的内容送入累加器，而 MOV 指令则能把存储单元的内容送入任一寄存器中。

## 6-45 LHLD——将直接寻址的存储单元内容送入寄存器 H 和 L

第二和第三结果代码字节提供数据字节的存储器地址，取出其内容送入寄存器 L，下一个相继数据字节的内容取出后送入寄存器 H。

假定存储器字节 $084\ A_{16}$ 内容为 $3\ A_{16}$，存储器字节




084 $B_{16}$ 的内容为 2 $C_{16}$，则执行指令

```
LABEL   EQU     084AH
        -
        -
        -
        LHLD    LABEL
```

以后，寄存器H的内容是 2 $C_{16}$，寄存器L的内容是 3 $A_{16}$。

应当记住，EQU 是汇编命令，它不是一条指令。无论何处出现LABEL时，这个命令都告诉汇编程序采用16位值 084 $A_{16}$。

LHLD 是LXIH，DATA 指令采用直接寻址方式的变型。例如指令：

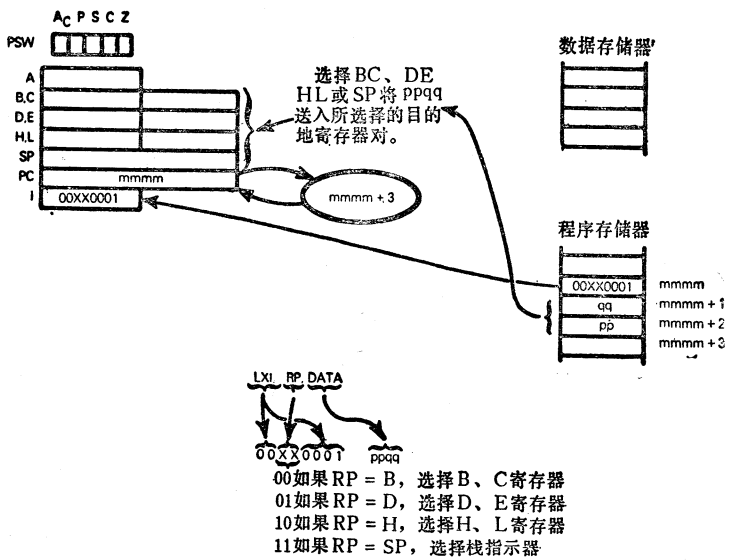
LXI H，2C3AH

也能将 2 $C_{16}$ 送入寄存器H并把 3 $A_{16}$ 送入寄存器L。对于不变化的16位数据，采用LXIH，DATA 而不用 LHLD ADDR。

应当记住，如果用 ADDR 对读/写存储器的一个字节进行

直接寻址，那末能够用LHLD指令改变送入寄存器H和L的数值。为此，只需将新的数值写入ADDR和ADDR+1中就可以了。

## 6-46 LXI——将16位立即数送入寄存器对




取出指令的第二和第三个结果代码字节送入指定的寄存器对，在执行指令

LXI SP，217 AH

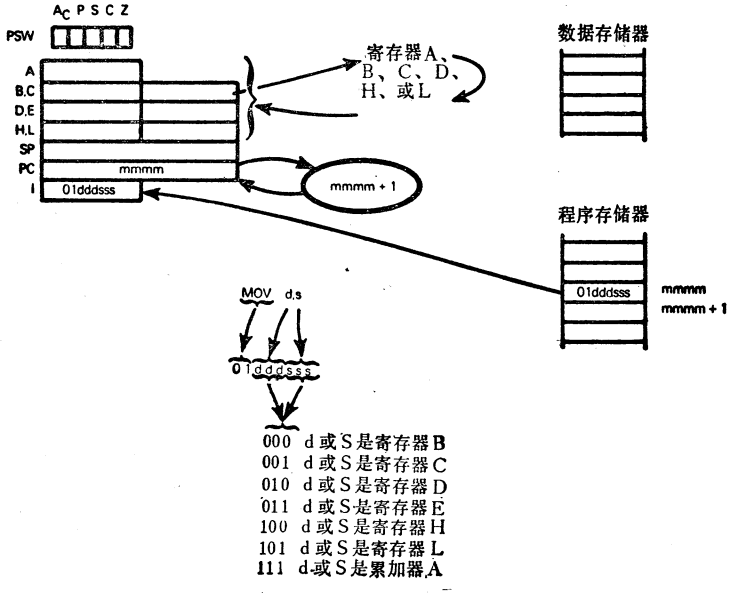
之后，栈指示器的内容是217 $A_{16}$。

LXI指令经常用于将地址送入寄存器对。

## 6-47 MOV——传送数据

这条指令有两种形式。先看将一个寄存器的内容传送到另

一个寄存器的指令。




将一个寄存器的内容传送到另一个寄存器。例如：

**MOV A, B**

将寄存器B的内容传送到累加器。

又如：

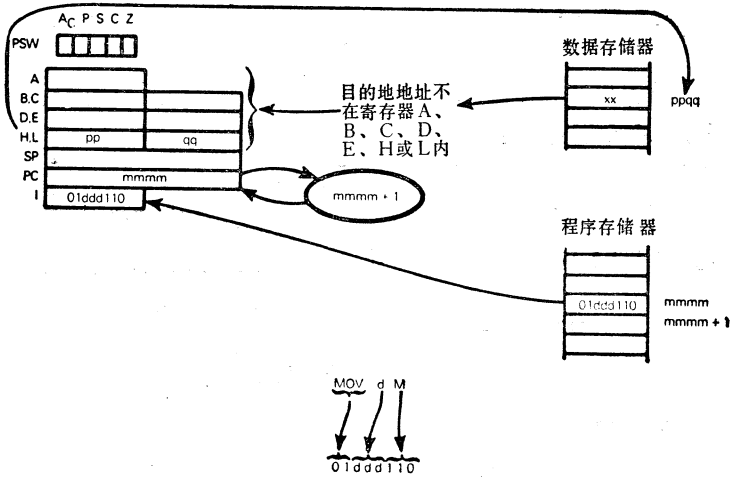
**MOV L, D**

将寄存器D的内容传送到寄存器L。

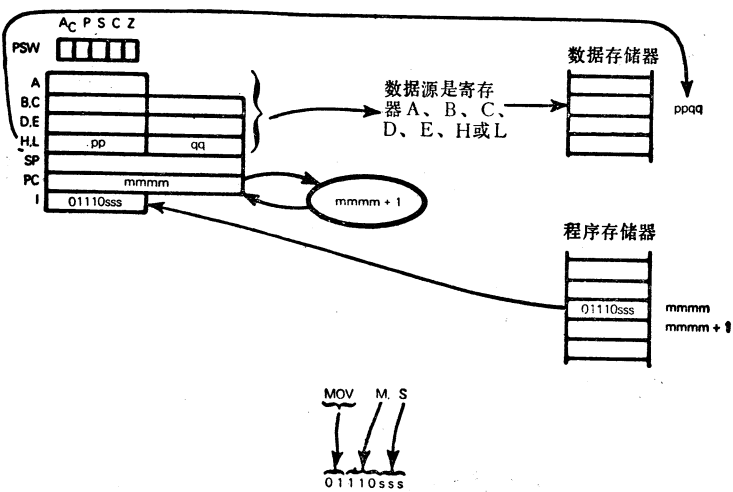
**MOV C, C**

因为在这条指令中寄存器C既是源又是目的地，所以什么操作也不进行。

存储器字节也可以作为数据的源：




或者一个存储器字节也可以是存放数据的目的地。




在两种情况下，ddd 或 sss 都是表示寄存器——寄存器的传

送。因而指令：

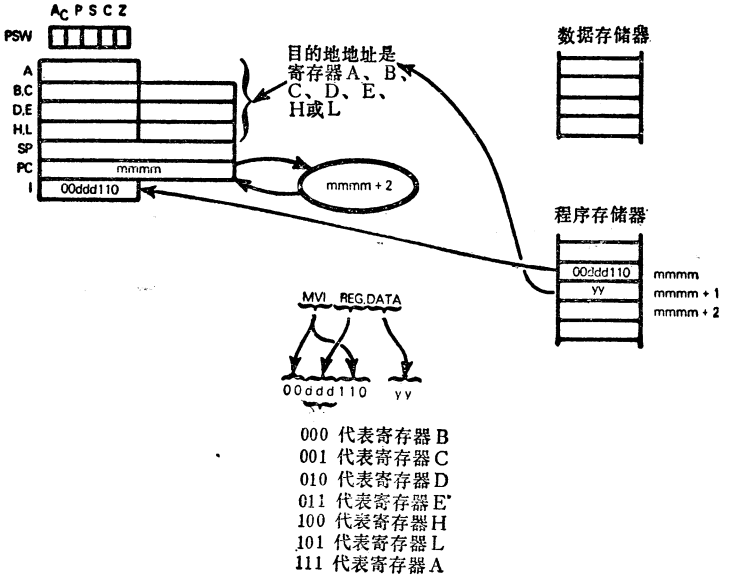
MOV M, A

将累加器内容送入由寄存器H或L的内容所指向的读/写存储器字节。指令：

MOV L, M

将由H和L寄存器所指向的存储器字节内容送入寄存器L。在8080指令中最常使用的就是各种各样的传送指令。

## 6-48 MVI——将立即数据送入寄存器或存储器




将第二个结果代码字节的内容送入某寄存器。

当执行指令：

MVI A，2AH

以后，$2A_{16}$ 被送入累加器。执行指令

MVI H，03H

将 $03_{16}$ 送入寄存器H。

将立即数据送入寄存器的指令在8080程序中用得十分经常。

应当注意，LXI 指令与两条 MVI 指令是等效的。

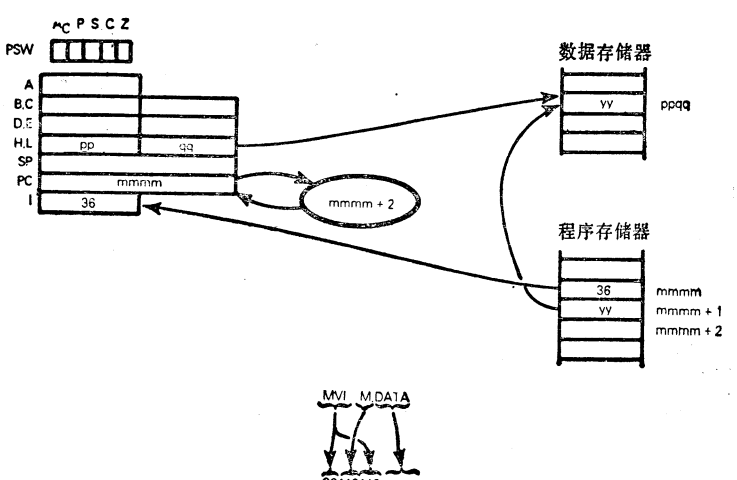
例如：

LXI H，032AH

等效于：

```
MVI H,03H
MVI L,2AH
```

也可以将立即数据送入读/写存储器的一个字节内。



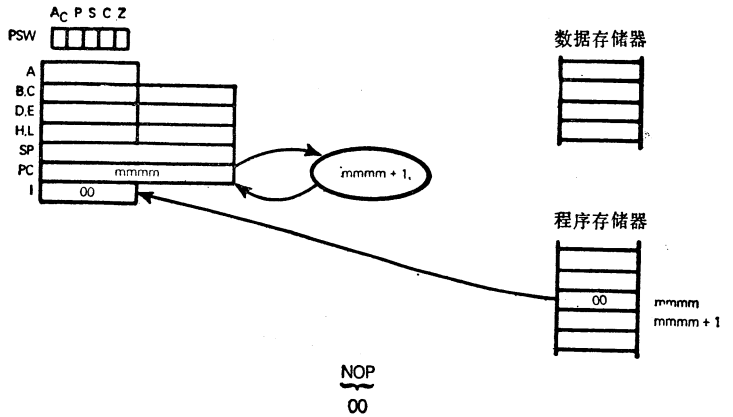
假定寄存器H的内容为$03_{16}$，和寄存器L的内容为$2A_{16}$，则执行指令

MVI M, 2CH

以后，$2C_{16}$被送入存储器字节$032A_{16}$中。

将立即数据传送到存储器的这条指令(即MVI,M,DATA)要比将立即数传送到寄存器的指令(即MVI REG, DATA)用得少得多。

## 6-49 NOP——不操作




当执行这条指令时，什么工作也没有做；不操作指令的存在有下述三个原因：

1) 从不存在的存储器中取结果代码的程序错误,取出的是00。这是一个保证不犯最普通的程序错误的很好的方法。

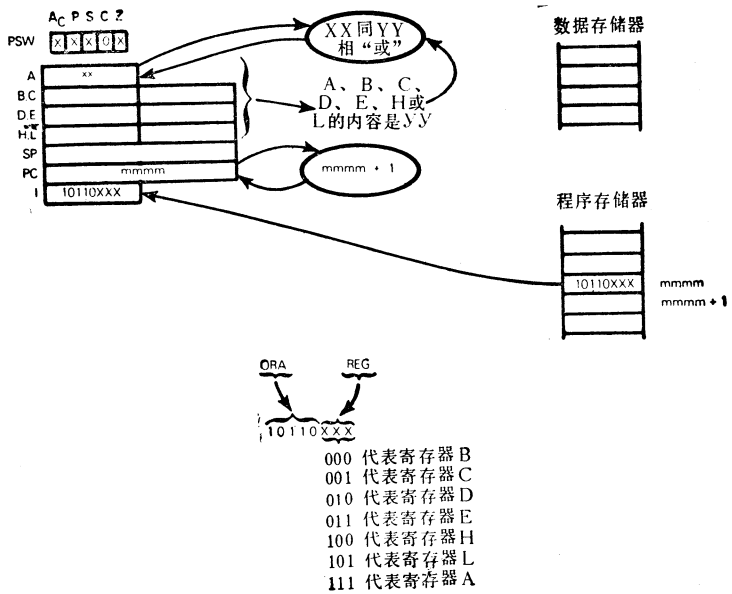
2) NOP 指令使你能够对目标程序字节赋予一个标号

HERE NOP

3) 细调延迟时间，每条 NOP 指令增加 4 个时钟脉冲周期

的时间延迟。NOP 并不是一条很有用的或常常使用的指令。

## 6-50 ORA——寄存器或存储器和累加器相“或”

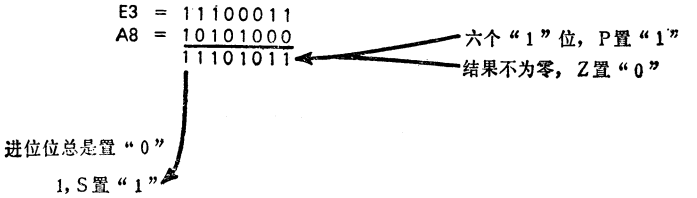



累加器的内容和任一寄存器的内容相“或”，把结果存放到累加器中。

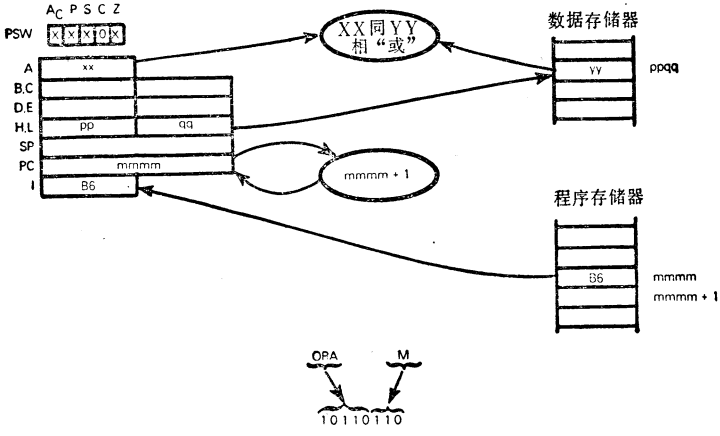
假定 XX=E $3_{16}$，寄存器 E 的内容是 A $8_{16}$，则执行指令

ORA E

以后，累加器的内容是 $EB_{16}$。




存储器字节的内容也可以和累加器相“或”。




假如 XX＝E $3_{16}$，YY＝A $8_{16}$，则执行指令

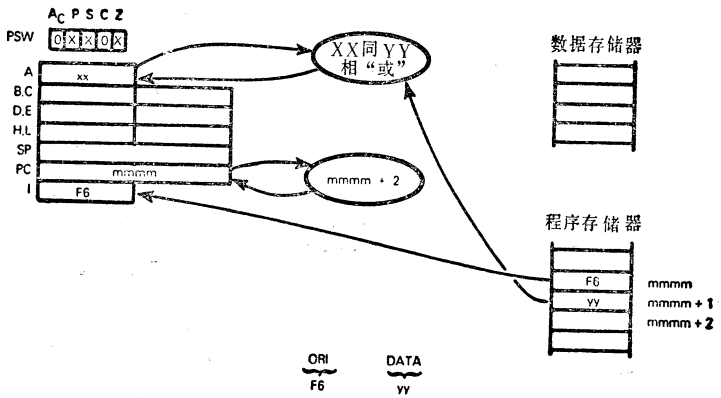
ORA　M

和执行刚才所描述的 ORA　E 指令所产生的结果相同。

ORA 指令没有立即“或”(ORI)指令用得经常。

应当注意，累加器和它自己相“或”(ORA　A)可以用来清除进位状态位；这条指令也用于 INX 和 DCX 指令之后置状态位。

## 6-51 ORI——立即数和累加器内容相“或”

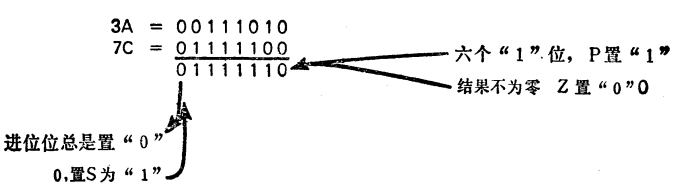



累加器内容和指令的第二个结果代码字节相“或”

假定 XX＝3 $A_{16}$，则执行指令：

ORI 7 CH

以后，累加器的内容是 7 $E_{16}$；

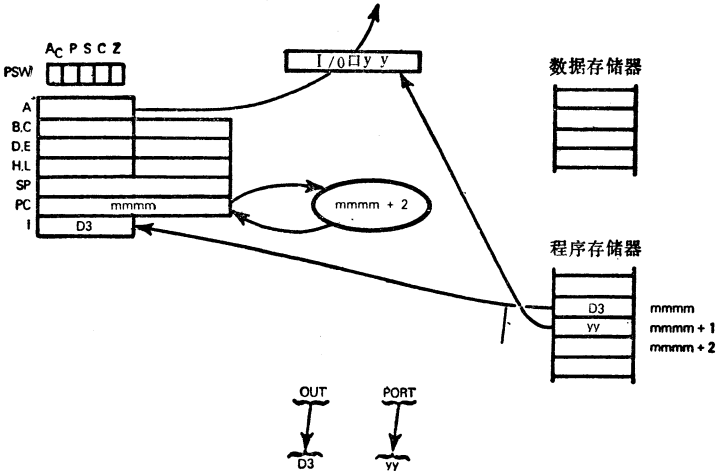



这是一条例行逻辑指令，它经常用来使某几位置“1”，例如执行指令

ORI 80 H

将无条件地将累加器的最高位置“1”。

## 6-52 OUT——从累加器输出




累加器内容送入由 OUT 指令第二个结果代码字节所标出的 I/O 口。

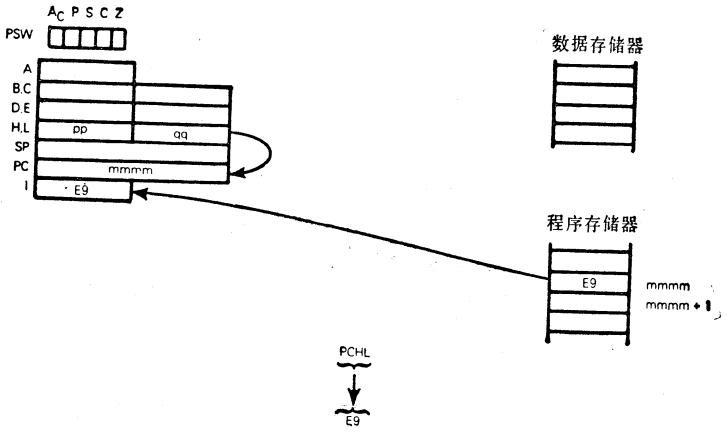
假定 $36_{16}$保存在累加器内则执行指令

OUT 1AH

之后，$36_{16}$送入 I/O 口 $1A_{16}$ 的缓冲器。OUT 指令不影响任何状态位。

OUT 指令的使用同硬件的关系非常大，有效的 I/O 口的地址取决于所实现的输入/输出逻辑功能。也可以设计一种微型计算机系统，它能访问具有专用的存储器地址并利用存储器访问指定的外部逻辑设备。OUT 指令常常用于以特殊方式控制 CPU 以外的微型计算机逻辑。

## 6-53 PCHL——转移到由寄存器 HL 所指定的地址




寄存器H和L的内容送入程序计数器，从而执行隐含地址转移。

指令序列

```
LX I  H,ADDR
PCHL
```

的实际效果与一条

```
JMP  ADDR
```

的指令完全一样。都使得下一次要执行的是标号为ADDR的指令。

当你想要对多重返回子程序的返回地址增1时，PCHL指令是有用的。

下面来看调用子程序SUB的实例：

```
CALL    SUB     ;CALL SUBROUTINE
JMP     ERR     ;ERROR RETURN
                ;GOOD RETURN
```

```
CALL    SUB   调用子程序
JMP     ERR   有错误返回
              无错误返回
```

使用RET指令从SUB返回将执行JMP ERR指令；因此如果执行SUB之后没有发现错误就返回如下

```
POP     H       ;POP RETURN ADDRESS TO HL
INX     H       ;ADD 3 TO RETURN ADDRESS
INX     H
INX     H
PCHL            ;RETURN
```

```
POP     H   返回地址弹出，送入寄存器 HL
INX     H   返回地址增 3
INX     H
INX     H
PCHL        返回
```

## 6-54 POP——从栈顶读出

将栈顶的两个字节弹出，送入指定的寄存器对。

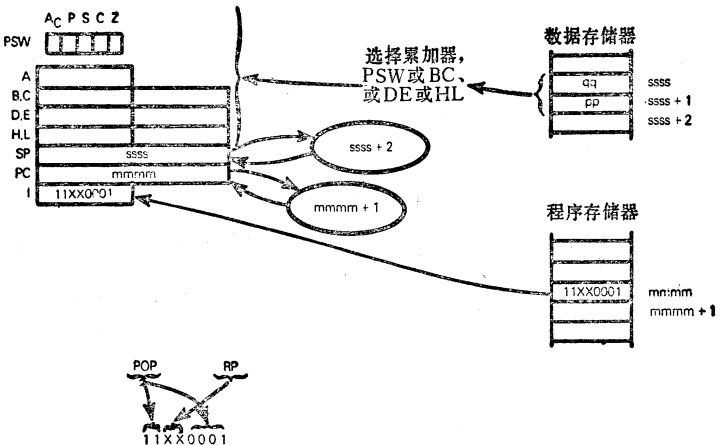
假定 qq＝$03_{16}$，pp＝ $2A_{16}$，则执行指令：

POP H

后将 $O3_{16}$ 送入寄存器 L，$2A_{16}$ 送入寄存器 H。执行指令

POP PSW

后将 $O3_{16}$ 送入状态标志，$2A_{16}$ 送入累加器。于是 C 状态位将被置“1”，其余的状态位将被清零。POP 指令通常用来恢复被保存在栈中的各寄存器和状态的内容，如在中断服务时那样。




00如果RP是B，选择B、C寄存器
01如果RP是D，选择D、E寄存器
10如果RP是H，选择H、L寄存器
11如果RP是PSW，选择累加器和状态标志，把它们作为一个16位的单元

## 6-55 PUSH——写入栈顶

把指定的寄存器对的内容压入栈顶。

假定寄存器 H 的内容是$O3_{16}$，寄存器 L 的内容是 2 $A_{16}$，则执行指令

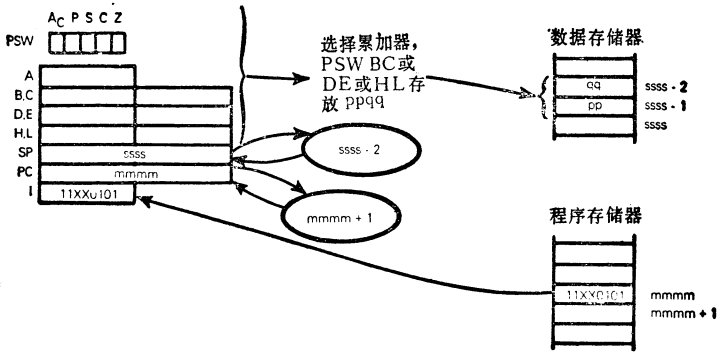
PUSH H

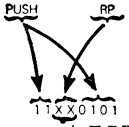
后，先把$O3_{16}$，然后把 2 $A_{16}$ 压入栈顶。而执行指令

PUSH PSW

则先压入累加器内容，然后把状态标志也压入栈顶。

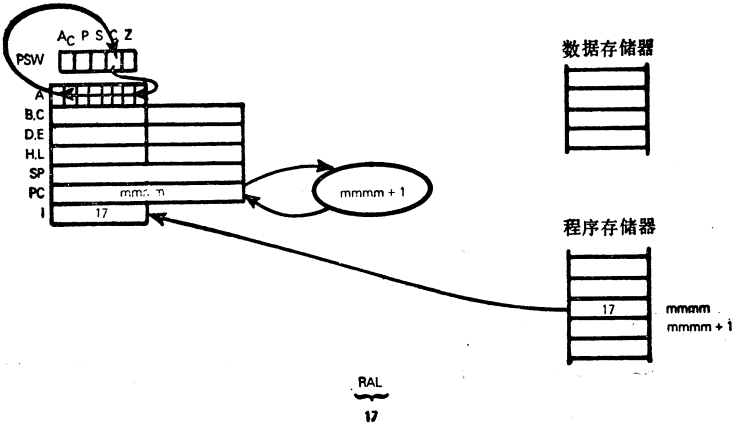
PUSH 指令最常用于保护寄存器和各状态位的内容。例如在输入中断服务之前就要用到这条指令。




00如果RP是B，选择寄存器B和C
01如果RP是D，选择寄存器D和E
10如果RP是H，选择寄存器H和L
如果RP是PSW，选择累加器和状态标志作为一个16位单元

## 6-56 RAL——累加器内容连同进位位循环左移



累加器内容经过进位位状态循环左移一位。

假定累加器内容为 $7A_{16}$，进位状态位被置“1”，则执行指令

**RAL**

以后，累加器内容是 $F5_{16}$，进位状态位复位为“0”。


| 累加器 | 进位位 | 累加器进位位 |
|---|---|---|
| 01111010 | 1⟶ | 11010l 0 |

正如在“微型计算机入门”第一册中所介绍的那样，RAL 指令常常用来执行多字节的向左移位。在第一次左移前，进位位状态清零，接着进位位状态把一个字节的高位移入下一个字节的低位。下面是一个把四个存储器字节的内容都左移一位的指令序列：

```
        LXI   H,DATA   ;LOAD ADDRESS OF LOW ORDER DATA BYTE
        ANA   A        ;INITIALLY CLEAR CARRY
        MVI   B,3      ;USE REGISTER B AS A COUNTER
LOOP    MOV   A,M      ;LOAD DATA BYTE INTO ACCUMULATOR
        RAL            ;ROTATE LEFT
        MOV   M,A      ;RESTORE RESULT
        INX   H        ;INCREMENT ADDRESS IN HL
        DCR   B        ;DECREMENT COUNTER
        JNZ   LOOP     ;RETURN FOR NEXT BYTE IF THERE IS ONE
```

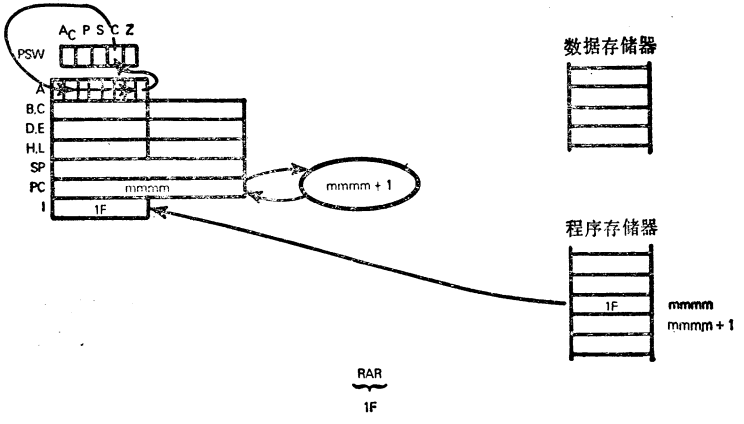
```
        LXI   H,DATA   将低位数据字节的地址送入H，L
        ANA   A        最初清除进位位
        MVI   B，3     以寄存器B作为计数器
LOOP    MOV   A，M     取出数据字节送入累加器
        RAL            循环左移
        MOV   M，A     恢复结果
        INX   H        寄存器H L中地址增1，
        DCR   B        计数器减1
```

JNZ　LOOP　　如果 z=0，则返回下一字节。

**状态位的状态**

注意应该仔细地思考一下给定的这几位状态位是置“1”还是置“0”。RAL 只影响进位位状态。INX 和 DCR 则对零状态位，符号状态位和奇偶状态位都有影响，但对进位位则无影响，因此在执行 RAL 指令后可以把进位位状态保留到下一次循环。

## 6-57　RAR——累加器内容连同进位位循环右移




累加器内容通过进位位状态循环右移一位。

假定累加器内容是 7 $A_{16}$，进位状态置“1”，则执行指令

RAR

以后，累加器的内容是 $BD_{16}$，进位状态位复位。

| 累加器 | 进位 | ⟶累加器 | 进位 |
|---|---|---|---|
| 01111010 | 1 | 10111101 | 0 |

正如在“微型计算机入门”第一册中所介绍的那样，RAR指令常常用来执行多字节的向右移位。进位状态在第一次右移以前清零。接着，进位状态把一个字节的低位送到下一个字节的高位。

下面给出将四个存储器字节的内容右移一位的指令序列：

```
        LXI     H,DATA  ;LOAD ADDRESS OF LOW ORDER DATA BYTE
        ANA     A       ;INITIALLY CLEAR CARRY
        MVI     B,3     ;USE REGISTER B AS A COUNTER
LOOP    MOV     A,M     ;LOAD DATA BYTE INTO ACCUMULATOR
        RAR             ;ROTATE RIGHT
        MOV     M,A     ;RESTORE RESULT
        INX     H       ;INCREMENT ADDRESS IN HL
        DCR     B       ;DECREMENT COUNTER
        JNZ     LOOP    ;RETURN FOR NEXT BYTE IF THERE IS ONE
```

```
        LXI   H，DATA   低位数据字节的地址，送寄存器对H，L
        ANA   A         将进位位预置零
        MVI   B，3      用寄存器B作为计数器
LOOP    MOV   A，M      取出数据字节送入累加器
        RAR             循环右移
        MOV   M，A      恢复结果
        INX   H         寄存器HL中后地址增“1”
        DCR   B         计数器减1
        JNZ   LOOP      若z=0，则返回
```

请参阅RAL指令对于各种若为1状态的说明。

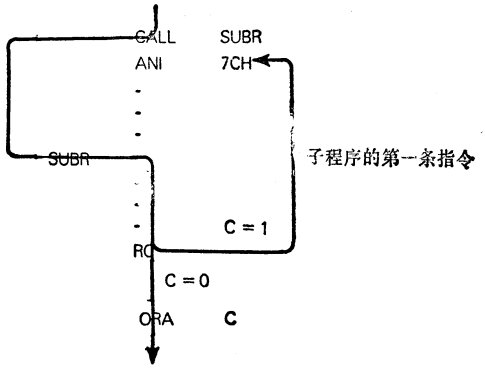
## 6-58 RC——若进位位状态等于1，返回

RC
D8

这条指令和RET指令相同，只是当执行RC指令时，除

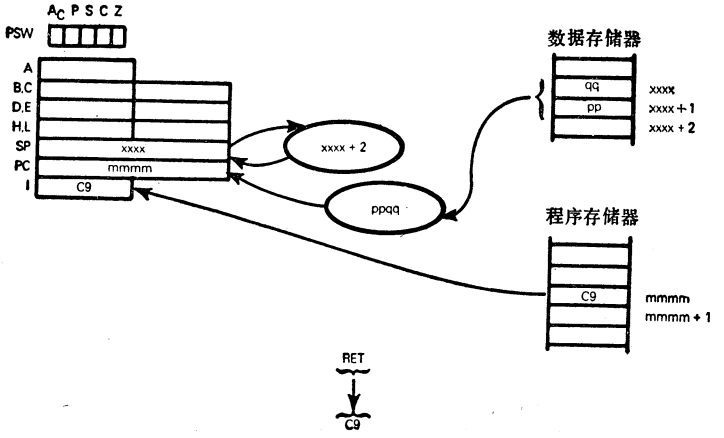
非进位状态等于1，否则就不返回。

例如下列指令序列：




在RC指令执行之后，若进位位状态等于“1”，返回到CALL下面的ANI指令，若进位位状态等于“0”就顺序执行下一条ORA指令。

## 6-59 RET——从子程序返回

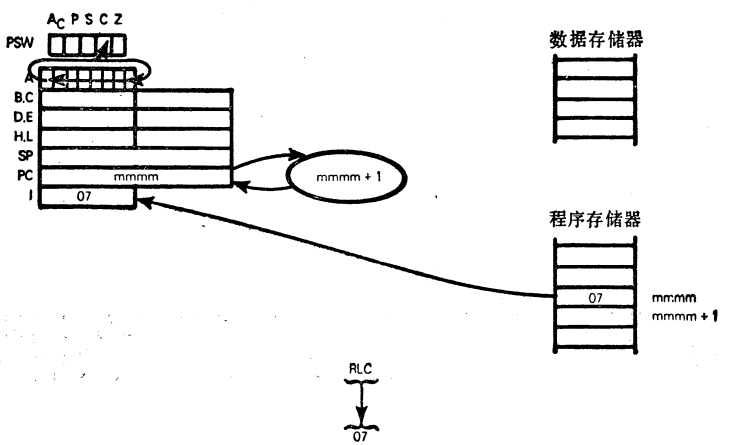


把栈顶的两字节内容送入程序计数器，这两个字节提供将要执行的下一条指令的地址，程序计数器原先的内容则被抹去。栈指示器增2形成新的地址送到栈顶。

每一个子程序中都必须至少有一条返回（或条件返回）指令。这就是在子程序内要执行的最后一条指令，使程序的执行返回调用程序，即返回主程序。

对于RET指令执行情况的举例说明可参看第五章。

## 6-60 RLC——累加器循环左移




累加器内容循环左移一位。

假定累加器内容为7 $A_{16}$，进位位状态为“1”，则执行指令

RLC

以后，累加器内容为F $4_{16}$，进位位状态被复位。

| 累加器 | 进位 | | 累加器 | 进位 |
|---|---|---|---|---|
| 01111010 | 1 | ⟶ | 11110100 | 0 |

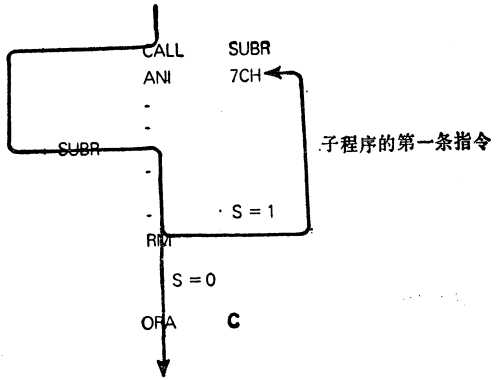
RLC应作为逻辑指令使用。

## 6-61 RM——若符号位状态等于1，返回

RM
F8

这条指令和RET指令相同，只是在执行RM指令时，除非符号位状态等于1，否则就不返回。

考虑指令序列：



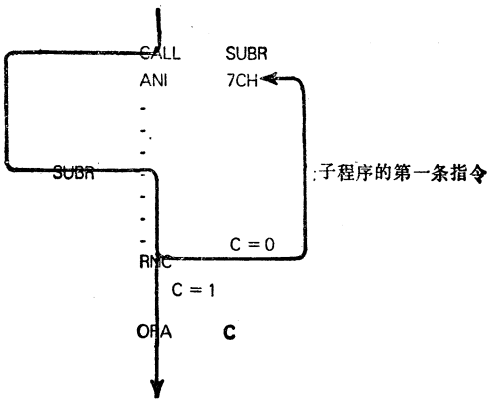

当执行完RM指令后，如果符号位状态等于“1”，则返回执行CALL后面的ANI指令。如果符号位状态等于“0”，则顺序执行下一条ORA指令。

## 6-62 RNC——若进位位状态等于0，返回

RNC
D0

这条指令和RET指令等同，只是在执行RNC指令时，除非进位位状态等于“0”，否则就不返回。

考虑指令序列：




当执行完RNC指令后，若进位位状态等于“0”，则返回执行CALL后面的ANI指令；若进位位状态等于“1”则顺序执行下一条ORA指令。

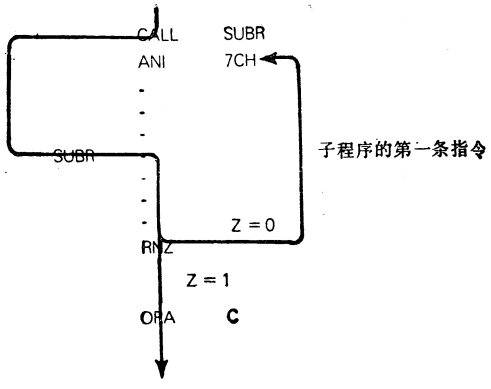
## 6-63 RNZ——若零状态位等于0，返回

RNZ
C0

这条指令和RET指令相同，只是在执行RNZ指令时，除非零状态位等于0，否则就不返回。

考虑指令序列：

当执行完RNZ指令后，若零状态位等于0，则返回执行在CALL后面的ANI指令；若零状态位等于“1”，则顺序执行下一条ORA指令。

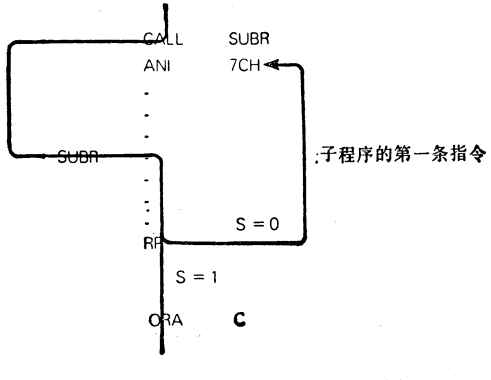



## 6-64 RP——若符号状态位等于0，返回

RP
F0

这条指令和RET指令相同，只是在执行RP指令时，除非符号状态位等于“0”，否则就不返回。

考虑指令序列：



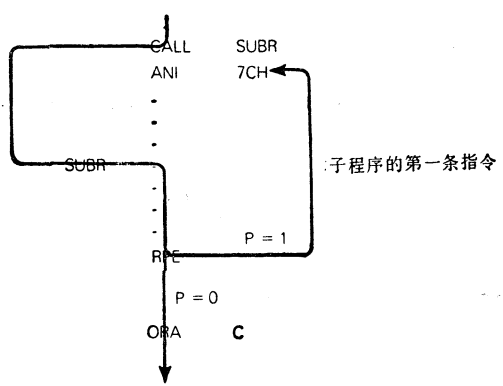
当执行完 RP 指令后，若符号状态位等于“0”，则返回执行 CALL 后面的 ANI 指令；若符号状态位等于“1”，则顺序执行下一条 ORA 指令。

## 6-65 RPE——若奇偶状态位等于 1，返回

RPE
E8

这条指令和 RET 指令相同，只是在执行 RPE 指令时，除非奇偶位状态位等于 1，否则就不返回。

考虑指令序列：

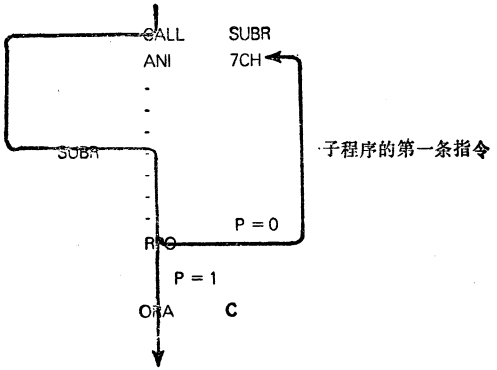



当执行完 RPE 指令后，若奇偶状态位等于“1”则返回执行 CALL 后面的 ANI 指令，若奇偶状态位等于“0”则按顺序执行下一条 ORA 指令。

## 6-66 RPO——若奇偶状态位等于 0，返回

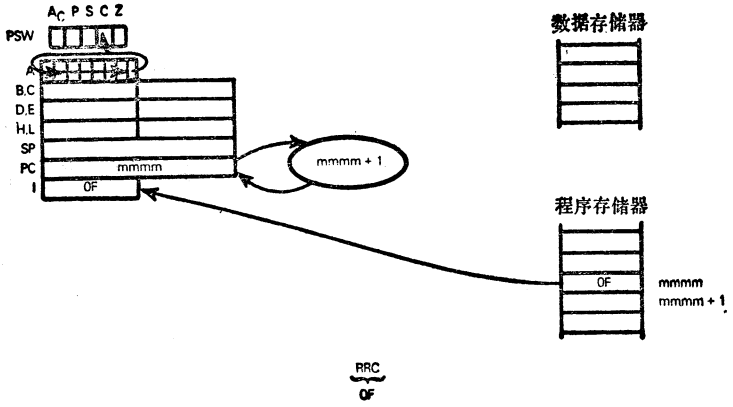
RPO
E0

考虑指令序列：




当执行完 RPO 指令后，若奇偶状态位等于“0”则返回执行 CALL 后面的 ANI 指令；若奇偶状态位等于“1”则顺序执行下一条 ORA 指令。

## 6-67 RRC——累加器循环右移



累加器的内容循环右移一位。

假定累加器的内容是 $7A_{16}$，进位位状态为“1”则执行指令

RRC

以后，累加器的内容是 $3D_{16}$，进位位状态复位。

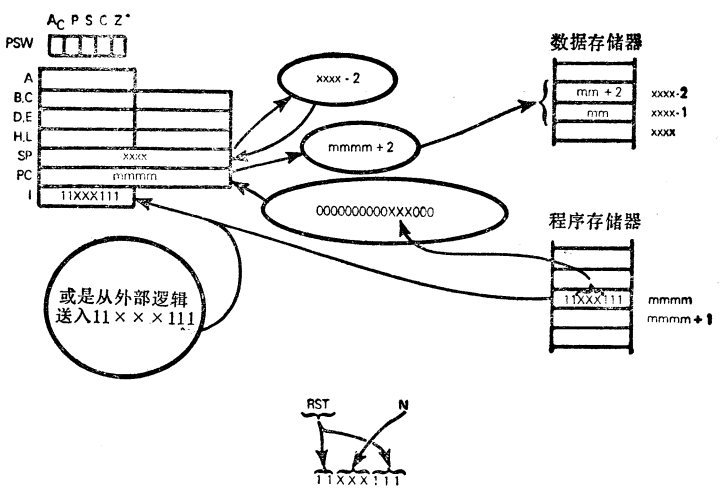
Accumulator C　　Accumulator C

01111010 1 ⟶ 00111101 0

| 累加器 | 进位 | 累加器 | 进位 |
|---|---|---|---|
| 01111010 | 1 ⟶ | 00111101 | 0 |

RRC 通常作为逻辑指令使用。

## 6-68 RST——重新启动




调用由N所指定的存储器低位地址为起始地址的子程序。当执行指令

RST 3

以后，就调用入口在存储单元 $0018_{16}$ 的子程序。程序计数

器原先的内容被压入栈顶。

正如第五章所述，通常 RST 指令用于中断处理。

**用RST指令调用子程序**

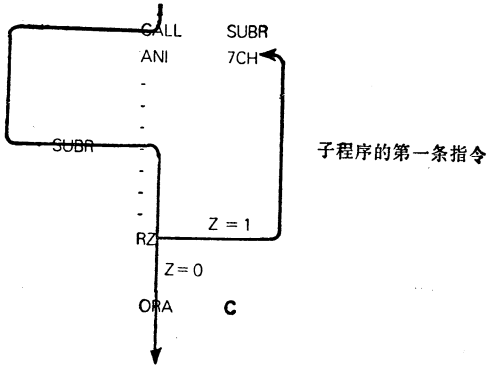
即使在你的用途中根本不用 RST 指令代码来执行中断服务，请你也不要忽略使用 RST 指令调用子程序的可能性。如果把经常使用的一些子程序的起始地址置于相应的 RST 地址上，那末调用这些子程序就只需用一字节的 RST 指令而不需要一个三字节的 CALL指令了。

## 6-69 RZ——若零状态位等于 1，返回

RZ
C8

这条指令和 RET 指令相同，只是在执行 RZ 指令时，除非零状态位等于 1，否则就不返回。

指令序列：

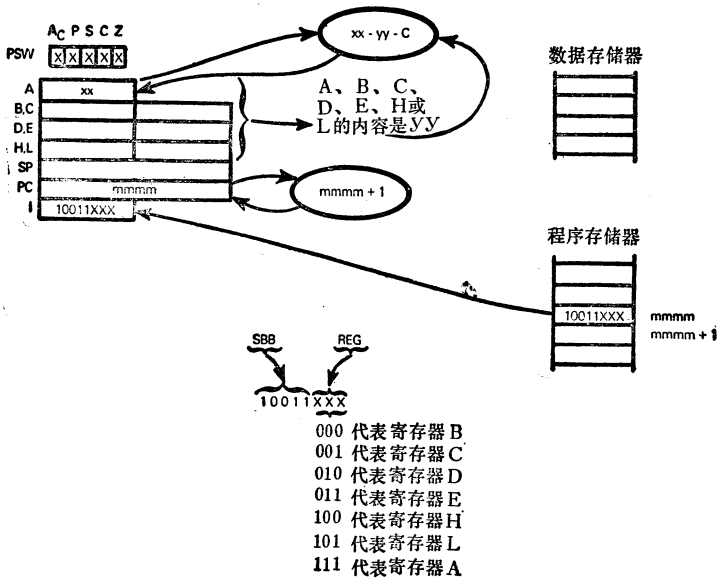



当执行完 RZ 指令后，若零状态位等于“1”，则返回执行在

CALL 后面的 ANI 指令；若零状态等于“0”，则顺序执行下一条 ORA 指令。

## 6-70 SBB——累加器内容与寄存器或存储器内容进行带借位减法

这条指令有两种形式。首先考虑从累加器减去寄存器的内容。

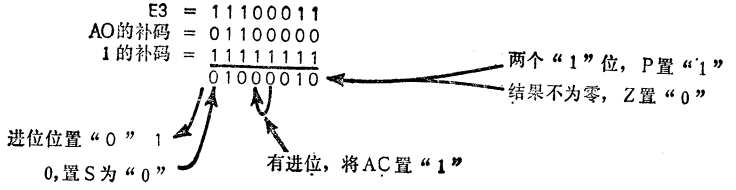



从累加器中减去指定的寄存器内容以及进位位状态，把寄存器的内容作为简单的二进制数据处理。

假定 XX＝$E3_{16}$，寄存器 E 的内容为 $AO_{16}$，进位位状态 C＝1，则执行指令

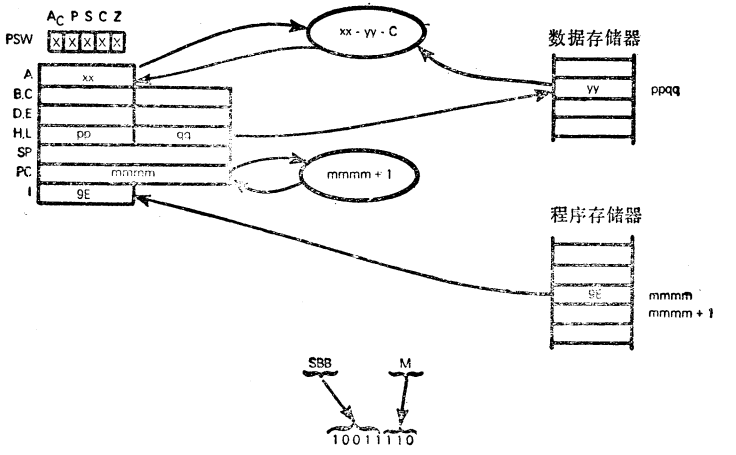
SBB E

之后，累加器的内容为 $42_{16}$：




应当注意，所得进位被取反。

也可以从累加器中的内容减去带借位的存储器字节的内容：



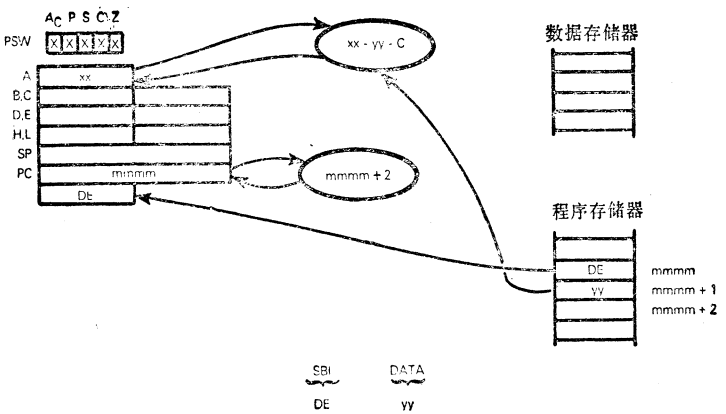

假定 $XX=E3_{16}$，$YY=AO_{16}$，进位位状态 $C=1$，则执行指令：

SBB M

后和执行刚才所介绍的 SBB E 指令产生的结果相同。

SBB 指令用于低位字节已经由 SUB 指令处理的多字节减法中。

# 6-71 SBI——累加器与立即数据进行带借位减法

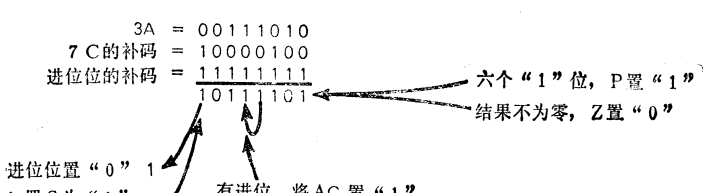



从累加器减去指令的第二个结果代码字节和进位位。

假如 XX＝$3A_{16}$ 进位位状态 C＝1 则执行指令

SBI　7 CH

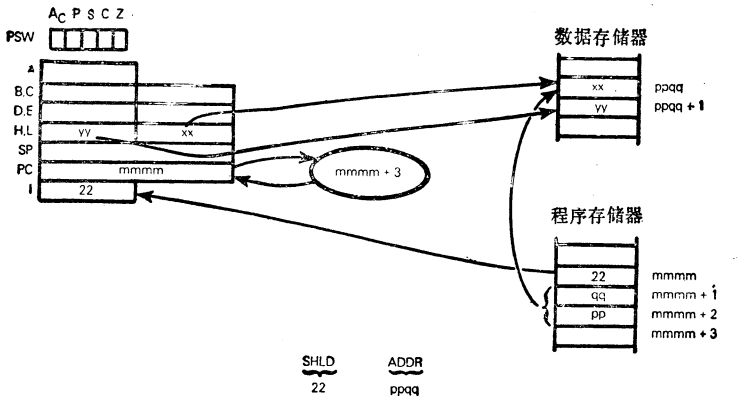
之后，累加器的内容是 $BD_{16}$：




应当注意，所得进位被取反。

这条指令没有 SUI 指令用得经常。

## 6-72 SHLD ——直接存入H和L寄存器




第二和第三结果代码字节提供一个数据字节的存储器地址，寄存器L的内容就写入这个字节，寄存器H的内容则写入相继的下一个字节中。

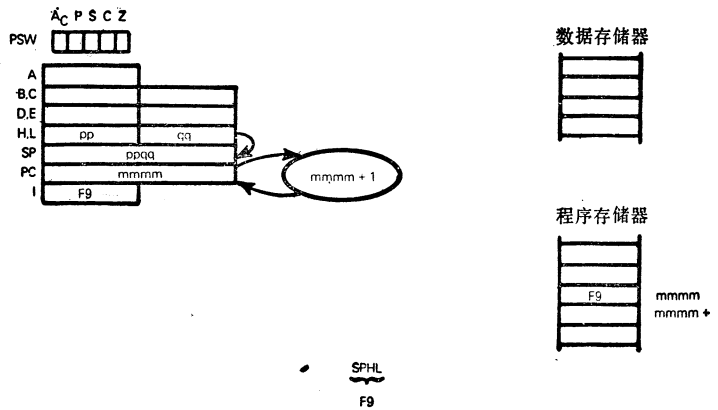
假定 XX＝2 $C_{16}$，YY＝3 $A_{16}$，则执行指令

```
LABEL   EQU     084AH

        SHLD    LABEL
```

之后，存储器字节 084 $A_{16}$ 的内容是2 $C_{16}$，存储器字节 084 $B_{16}$ 的内容是 3 $A_{16}$。

应当注意，EQU 是一个汇编命令，而不是一条指令，它告诉汇编程序只要一出现 LABEL，就使用 16 位值 084 $A_{16}$。

## 6-73 SPHL——将寄存器H和L的内容送入栈指示器




寄存器H和L的内容送入栈指示器。

假定 pp＝$08_{16}$ 和 qq＝3 $F_{16}$，则执行指令

SPHL

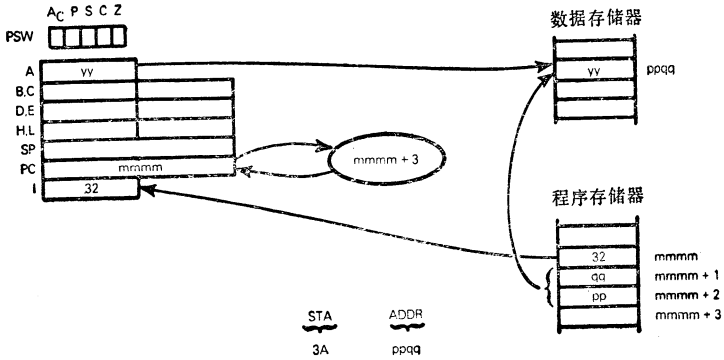
之后，栈指示器的内容是 083 $F_{16}$。

SPHL 指令能够用来访问两个栈单元。有一个空闲的地址保存在寄存器 H 和 L 内，经常使用栈存取文本字符串，或是必须以字节串行方式存取的数据。

必须记住的很重要的一点是栈逻辑能够用来代替具有自动增量的存储器隐含寻址。

## 6-74 STA——累加器内容存入采用直接寻址的存储单元

把累加器内容存放在被 STA 指令结果代码的第二和第三




字节所指定的直接寻址的存储器字节中。

假定累加器的内容是 3 $A_{16}$，则执行指令

```
LABEL   EQU     084AH

        STA     LABEL
```

以后，存储器字节 084 $A_{16}$ 的内容是 3 $A_{16}$。

应当注意，EQU 是一个汇编命令，它不是一条指令，它告诉汇编程序只要出现 LABEL 时，就采用 16 位的值 084 $A_{16}$。

指令：

STA LABLE

与下面两条指令是等效的：

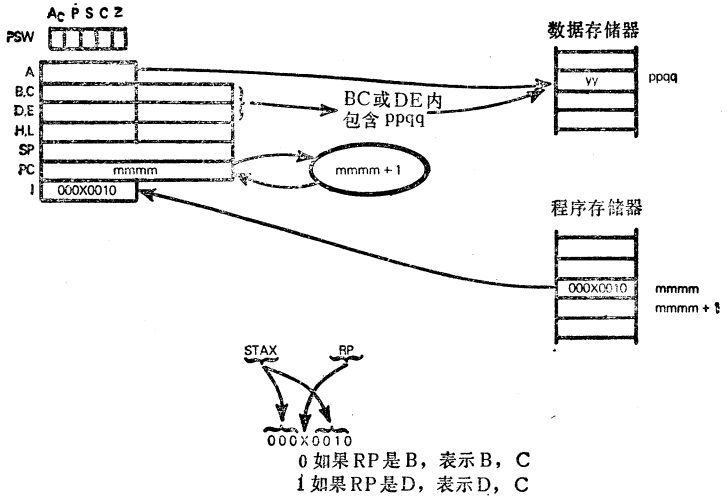
LXI H, LABEL

MOV M, A

当你在存储器中存储单个的数据值时，使用 STA 指令较为适合，因为它只用一条指令和三个目标程序字节来完成这个操作。而 LXI MOV 则要用两条指令的组合和四个目标程序字节。还有 LXI MOV 两条指令的组合要用到寄存器 H 和 L，

而 STA 指令则不需要。

## 6-75 STAX——累加器内容存入由寄存器对指定的存储单元




把累加器的内容存入由 BC 或 DE 寄存器对所指定的存储器字节内。

假定寄存器 B 的内容是 $08_{16}$，寄存器 C 的内容是 $4A_{16}$，累加器内容是 $3A_{16}$，则执行指令

STAX B

以后，存储器字节 $084A_{16}$ 的内容为 $3A_{06}$。

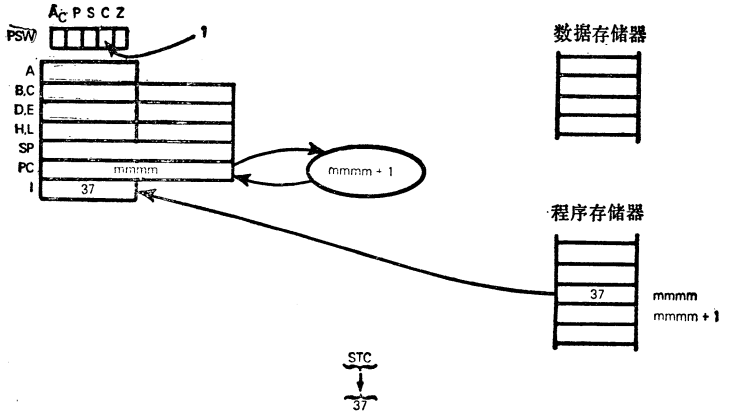
应当注意，没有设置 STAX H 指令，因为它等效于 MOV M，A 指令。一般情况下，把 STAX 和 LXI 指令放在一起，因为 LXI 指令能将 16 位地址送入 BC 或 DE 寄存器中。

LXI B，084 AH

STAX

一般是把STAX指令只能存储取自累加器的数据，而MOV指令能存储取自任何寄存器的数据。

## 6-76 STC——置进位位状态



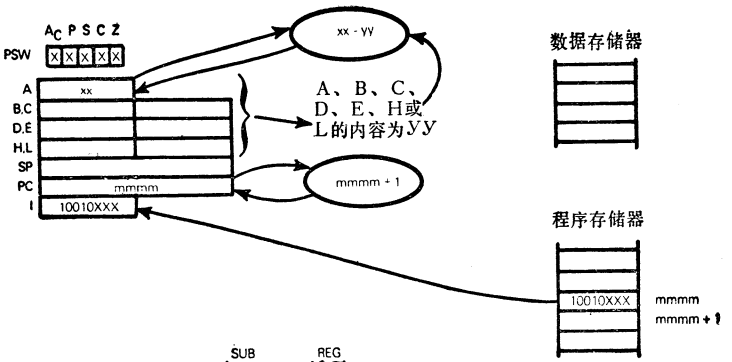

当执行STC指令以后，进位状态位置“1”，不管它原先的值是什么。这条指令不影响其它的状态位及寄存器的内容。

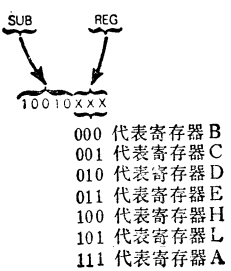
## 6-77 SUB——从累加器中减去寄存器或存储器的内容。

这条指令有两种形式。首先考虑从累加器减去寄存器的内容。

从累加器中减去指定寄存器的内容时，寄存器内容可当做简单的二进制数据处理。

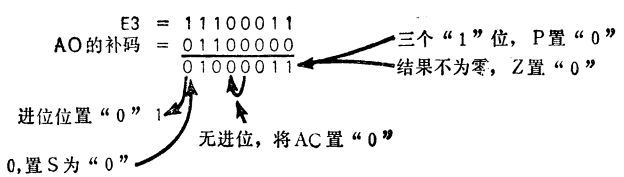
假定XX＝E $3_{16}$，寄存器E的内容为 $AO_{16}$，则执行指令



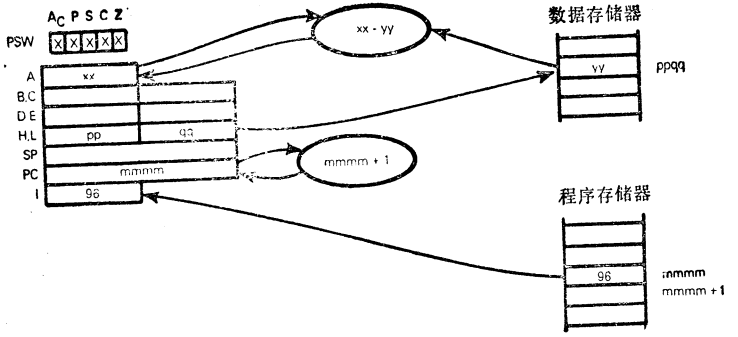

SUB E

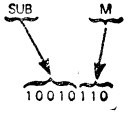
之后，累加器的内容是 $43_{16}$，




应当注意，所得进位被取反。

也可以从累加器中减去存储器字节的内容。




假如 XX＝E $3_{16}$ 和 YY＝A$O_{16}$，则执行指令

SUB　M

之后和执行刚才所介绍的 SUB 执行的结果相同。

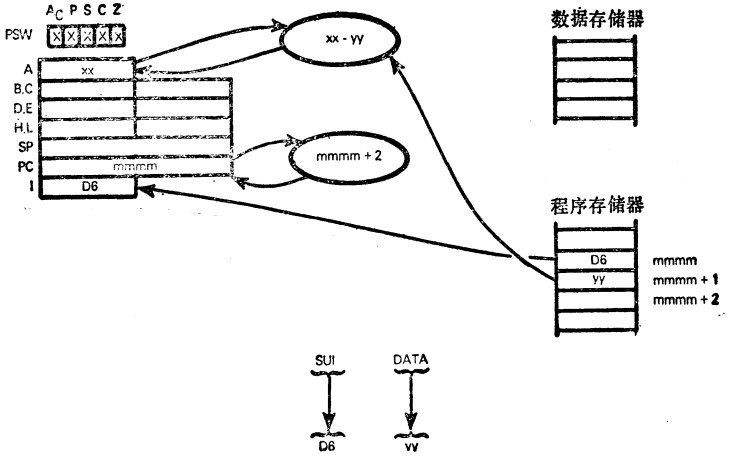
SUB 指令用来进行单一字节的减法或是在多字节减法中进行低位字节的减法。

## 6-78　SUI——从累加器减去立即数据

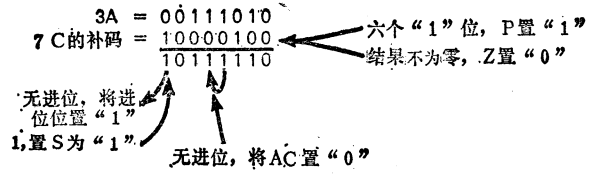
从累加器减去指令的第二结果代码字节。

假定 XX＝3 $A_{16}$，则执行指令

SBI　7 CH




之后，累加器的内容是$BE_{16}$、



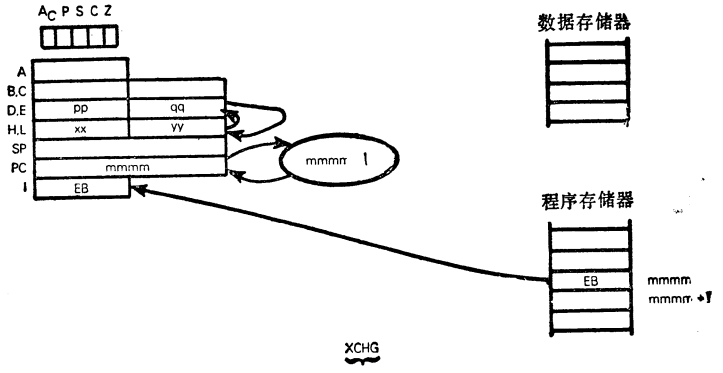

应当注意，算得结果进位位被取反。

这是一条推荐采用的立即数减法指令。

## 6-79 XCHG-——交换寄存器DE和HL的内容

寄存器D和E的内容同寄存器H和L的内容互相交换。

假定pp=$03_{16}$，qq=$2A_{16}$，XX=$41_{16}$和YY=$FC_{16}$，则




执行指令

XCHG

之后，H的内容是$03_{16}$，L的内容是$2A_{16}$，D的内容是$41_{16}$，E的内容是$FC_{16}$。下列两条指令

XCHG

MOV A，M

与LDAX D是等效的。

但是假如你想要从寄存器D和E存放的地址中取出数据送入寄存器B，则指令

XCHG

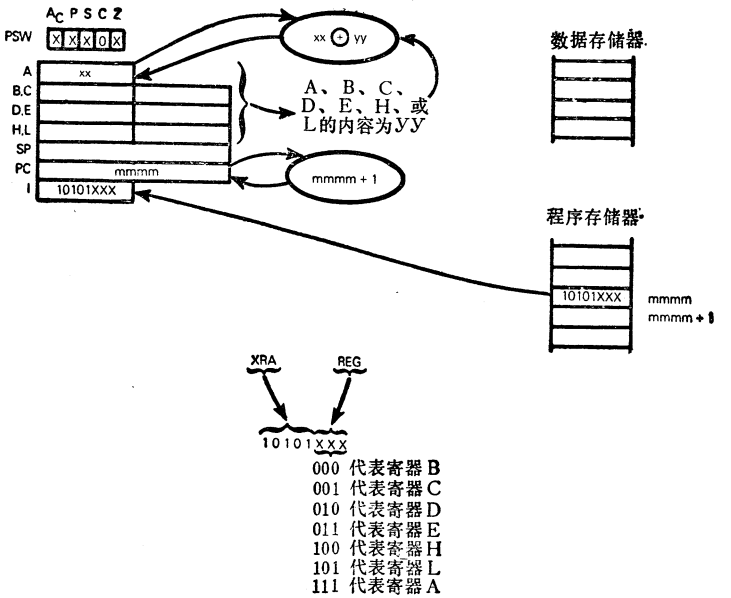
MOV B，M

就没有与之等效的单条指令了。

## 6-80 XRA——寄存器或存储器和累加器相“异”

这条指令有两种形式，首先考虑寄存器的内容与累加器相

异”。

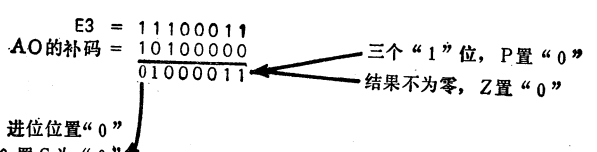



指定寄存器的内容和累加器相“异”。寄存器内容可作为简单的二进制数据处理。

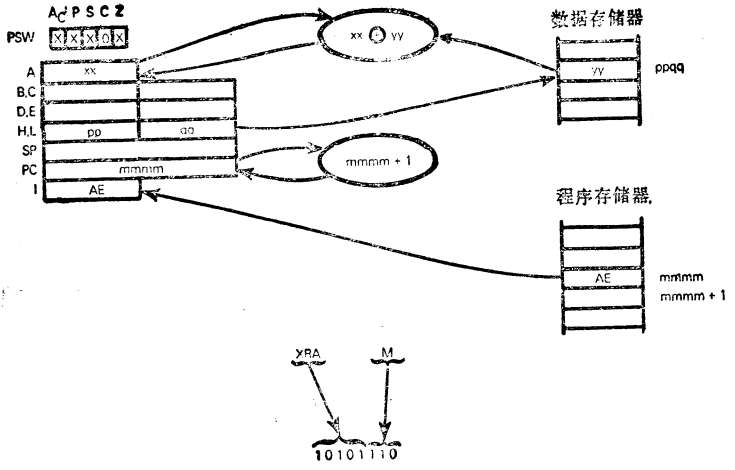
假定 XX＝E $3_{16}$，寄存器 E 的内容为 A $0_{16}$，则执行指令

XRA　E

之后，累加器的内容是 $43_{16}$




存储器字节的内容也可以和累加器相“异”




如果 XX＝E $3_{16}$ 和 YY＝A $0_{16}$，则执行指令

XRA　M

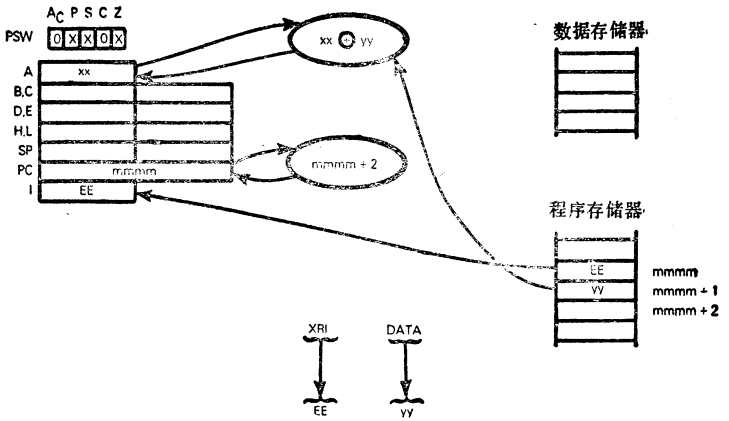
和执行刚才所描述的 XRA　E 指令的结果相同，“异”指令用于测试各状态是否发生变化。

## 6-81　XRI——立即数据和累加器相“异”

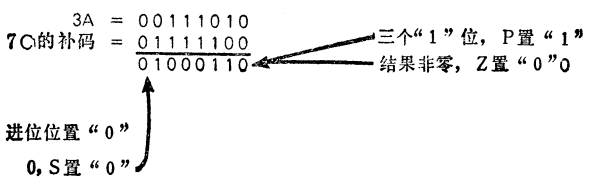
指令的第二结果代码字节和累加器相“异”。

假定 XX＝3 $A_{16}$ 则执行指令

XRI　7 CH




之后，累加器的内容是 $46_{16}$：



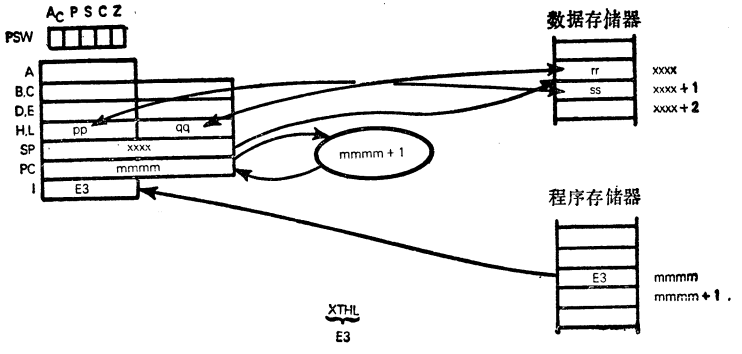

这是一条推荐采用的立即数减法指令

## 6-82 XTHL——栈顶和H，L交换

栈顶字节和寄存器L交换内容。寄存器H的内容和栈顶下一字节交换内容。假定 pp=$21_{16}$，qq=$FA_{16}$，rr=$3A_{16}$，ss=$E2_{16}$，则执行指令

XTHL

之后，分别有：H的内容为 $E2_{16}$，L的内容为 $3A_{16}$ 以及栈顶的两个字节分别为 $FA_{16}$ 和 $21_{16}$。




顺序执行下面的两条指令：

XTHL

XTHL

相当于不操作。

XTHL 指令用于存取和处理位于栈顶的数据。这已在第五章中列举的多重子程序中介绍过了。

# 第七章　一些常用的子程序

在许多微型计算机程序中出现大量的适用于各种用途的操作。在这一章就介绍这样一些常用的指令序列。

为了能更好地领会本章内容，在学习本章中的子程序时，应当学习直到能够很好地修改它们为止。作为一个简单的练习，读者可以尝试重新编写这些程序，使得它们虽然完成的任务相同，但是所用的指令执行周期、指令条数或两者都能节省一些。接下去可重新编写实现另一些不同操作的程序。例如，已经介绍了16位数的二进制乘法，那么32位数的乘法又如何实现呢？把每一个例子都看作是一典型的示范性的指令序列，读者可以把它们修改成满足各自迫切需要的程序。

这一章涉及的简单程序可分为下列四类：

1）存储器寻址

2）数据传送

3）运算

4）程序执行顺序逻辑

我们将按照上述程序种类的顺序来一一介绍。

## 7-1　存储器寻址

虽然8080/9080的存储器寻址方式仅限于直接寻址和隐含寻址，但是可以使用简单的指令序列来模拟任何其它方式的寻址。

我们将说明自动增1、自动减1、变址寻址、间接寻址和变址后间接寻址等寻址方式。所有这些寻址方式都已经在“微

型计算机入门”，第一册一书叙述并作了举例说明。

### 7-1-1 自动增1和自动减1

与同类型其它微型计算机比较，8080/9080指令系统的缺点之一是没有自动增1和自动减1的隐含寻址；本章稍后介绍的数据传送程序将表明，当处理数据缓冲或处理任何连续的数据存储器字节块时，地址增1或减1都是经常需要的。

**栈指示器存储器寻址**

在一些情况下，可以用栈指示器实现自动增1或自动减1，实现隐含存储器寻址，但是毕竟存在一些程序上的限制：

1）必须用压入指令代替向存储器写入指令，用弹出指令代替从存储器读出指令。这就限制只能在写入时自动减1，而读出时自动增1。

2）我们记得，存储器是按字节对来存取的。压入和弹出指令所涉及的都是寄存器对时，而不是单个的数据字节单元。如果能够将所执行的指令减半，那么这种寻址方式可能是有优点的，但是如果要处理的缓冲器字节个数是奇数，或者由于某种原因不能把数据按16位作为一个单元进行处理，那么这又成了这种寻址方式的缺点。

3）栈指示器的原先的内容指向当前栈顶。因此当栈指示器作为存储器地址寄存器使用时，这个原先的内容必须保护起来。当然，这就意味着在恢复栈指示器之前，你不能使用子程序，或者对栈进行访问。

**保护栈指示器**

如何保护栈指示器原先的内容呢？8080指令系统提供了将寄存器H、L内容送入栈指示器中的SPHL指令。但是它没有将栈指示器内容送入寄存器HL或其它寄存器的指令。因而必须先将H和L寄存器清零，然后再

执行一条DAD指令，具体作法如下：

```
MVI     H,0     ;CLEAR H
MOV     L,H     ;CLEAR L
DAD     SP      ;MOVE SP TO HL
```

```
MVI  H, O  清H
MOV  L,H   清L
DAD  SP    SP内容传送到HL
```

应当注意，DAD指令将修改进位位状态。现在就可以将栈指示器内容保存到任何一个寄存器对中去了：

```
XCHG                ;SAVE HL IN DE
```

```
XCHG        将HL存到DE
```

或：

```
MOV     B,H     ;SAVE HL IN BC
MOV     C,L
```

```
MOV  B, H  将HL存到BC
MOV  C, L
```

或者也可以保留两个读写存储器字节暂存栈指示器。

```
SHLD    STAK    ;Save HL at STAK and STAK + 1
```

```
SHLD  STAK  把HL内容保存在STAK和STAKM单元内
```

**恢复栈指示器**

恢复被保护的栈指示器内容是很容易的。如从BC中恢复，可使用下列指令：

```
MOV     H,B         ;MOVE BC TO HL
MOV     L,C
SPHL                ;MOVE HL TO SP
```

```
MOV   H, B   BC 内容移入 HL
MOV   L, C
SPHL         HL 内容移入 SP
```

如从 DE 中恢复，可使用下列指令：

```
XCHG            ;MOVE DE TO HL
SPHL            ;MOVE HL TO SP
```

```
XCHG   DE 内容移入 HL
SPHL   HL 内容移入 SP
```

如从存储器中恢复，可使用下列指令：

```
LHLD    STAK    ;LOAD HL FROM STAK AND STAK+1
SPHL            ;MOVE HL TO SP
```

```
LHLD  STAK   将 STAK 和 STAK+1 地址内容送入 HL
SPHL         HL 内容移入 SP
```

**将地址送入栈指示器**

一旦栈指示器内容被保护后，就可以立即取一个新地址送入栈指示器。

```
LXI     SP,ADDR
```

或者你可以把保存在读/写存储器的两个字节中的地址送入栈指示器：

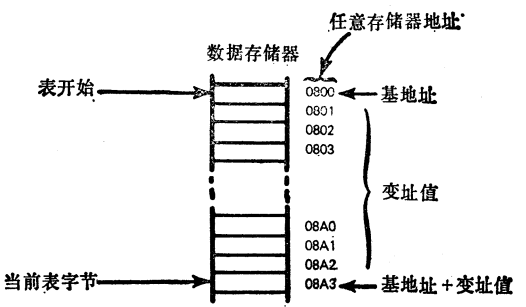
```
LHLD    ADDR    ;LOAD ADDRESS INTO HL
SPHL            ;MOVE HL TO SP
```

```
LHLD   ADDR   取地址到 HL
SPHL          HL 内容移入 SP
```

## 7-1-2 变址寻址

变址寻址方式所要求的地址是由基地址和由变址寄存器提供的位移量相加而得出的。

下面示出一例




一种可能性是基地址是一个不变的地址，而变址值经常变化，因此我们把基地址作为 16 位立即数进行存取，而变址值则由地址为 INDX 和 INDX+1 的两个读/写存储器字节给定。

现在可以产生一个变址地址如下：

```
LXI     B,BASE    ;LOAD BASE ADDRESS INTO BC.
LHLD    INDX      ;LOAD INDEX INTO HL
DAD     B         ;ADD BC TO HL
```

```
LXI     B, BASE   基地址送 BC
LHLD    INDX      将 INDX 送 HL
DAD     B         将 BC 与 HL 相加
```

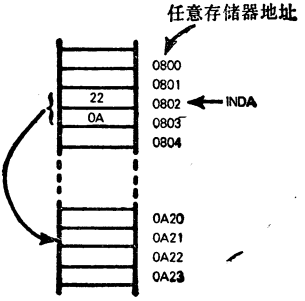
当前的变址值变化了怎么办？可以从 HL 的内容中减去基地址来恢复新的变址值，具体步骤如下：

```
LXI     B,BASE    ;LOAD BA SE ADDRESS INTO BC
MOV     A,L       ;SUBTRACT C FROM L
SUB     C
STA     INDX      ;SAVE IN INDX
MOV     A,H       ;SUBTRACT B FROM H WITH BORROW
SBB     B
STA     INDX+1    ;SAVE IN INDX+1
```

```
LXI     B, BASE     基地址送 BC
MOV     A, L        从L中减去C
SUB     C
STA     INDX        保存在 INDX 内
MOV     A, H        从H中对B作带借位减
SBB     B
STA     INDX+1      保存在 INDX+1 内
```

## 7-1-3 间接寻址

间接寻址把你所要求的存储器地址指定存放在两个存储器字节中。如图所示，存储器字节 $0802_{16}$ 和 $0803_{16}$ 存放着需要的存储器地址 OA $22_{16}$，低位地址字节放在高位地址字节的前头，这是为了与 8080 本身处理 16 位地址的方法相一致。




下列指令模拟间接寻址

```
        LHLD    INDA        ;LOAD ADDRESS INTO HL
;NOW ACCESS MEMORY USING NORMAL, IMPLIED
ADDRESSING, WITH HL PROVIDING THE ADDRESS
```

LHLD INDA 地址送 HL

现在访问存储器，用正规的，隐含寻址。HL 提供地址

### 7-1-4 变址后间接寻址

我们再来看一下使用变址寻址访问的表。假定表的基地址(BASE)可以从由 INDA 和 INDA+1 定址的两个存储器字节中找到，则当前表地址可以确定如下：

```
LHLD    INDA        ;LOAD BASE ADDRESS INTO HL
MOV     B,H         ;SAVE IN B,C
MOV     C,L
LHLD    INDX        ;LOAD INDEX INTO HL
DAD     B           ;ADD BC TO HL
```

```
LHLD    INDA        基地址送入 HL
MOV     B, H        存入 B，C
MOV     C, L
LHLD    INDX        INDX 送 HL
DAD     B           将 BC 和 HL 相加
```

这与变址后间接寻址是等效的。

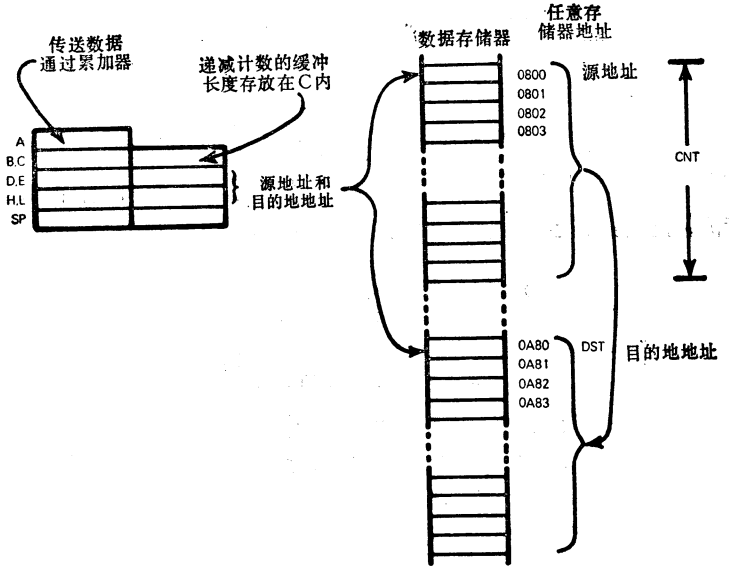
## 7-2 数据传送

我们现在来研究一些存放和传送相互连接的任意长度的数据字块即数据缓冲区的指令序列。

### 7-2-1 传送简单的数据块

首先用一个很简单的程序来研究连续的数据存储器字节块从存储器的一个区传送到另一个区的操作。

完成这一操作最简单的方法是用DE和HL寄存器分别作为数据源和目的地缓冲器来编址。以下的存储分配图描述了数据传送的操作。




这是数据传送程序：

```
        LXI     H,SRCE      ;LOAD SOURCE ADDRESS INTO HL
        LXI     D,DST       ;LOAD DESTINATION ADDRESS INTO DE
        MVI     C,CNT       ;LOAD BYTE COUNT INTO C
LOOP    MOV     A,M         ;LOAD SOURCE BYTE
        INX     H           ;INCREMENT SOURCE ADDRESS
        STAX    D           ;STORE IN DESTINATION
        INX     D           ;INCREMENT DESTINATION ADDRESS
        DCR     C           ;DECREMENT BUFFER LENGTH
        JNZ     LOOP        ;RETURN IF BUFFER NOT EMPTY
```

LXI H, SRCE 源地址送 HL

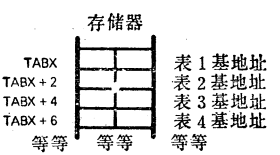
LXI D, DST 目的地地址送 DE

MVI C, CNT 字节计数值送 C

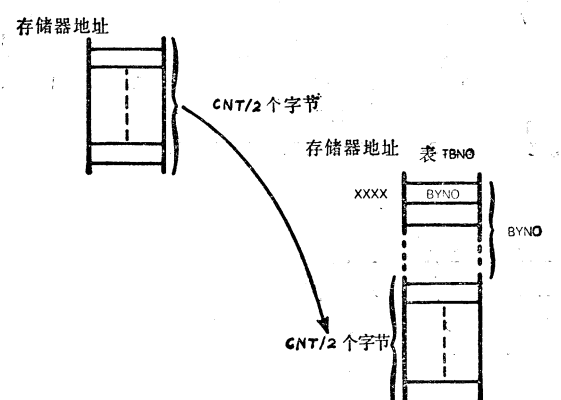
```
LOOP:  MOV   A, M    取源字节
       INX   H       源地址增1
       STAX  D       存入目的地地址
       INX   D       目的地地址增1
       DCR   C       缓冲长度减1
       JNZ   LOOP    缓冲长度不为零，返回
```

## 7-2-2 多重查表

下面考虑多重查表。它比我们刚才描述的数据传送更复杂些。数目不定的数据表，它们的起始地址存放在变址表中，变址表的起始地址由标号 TABX 给定：




有许多数据字节存在暂存器中，其起始地址由标号 CBASE 给定。数据字节的实际数目可以从标号为 CNT 的存储单元中



得到。这个源缓冲区是同我们刚描述过的数据传送程序中的源缓冲区相等效的。

该数据块的目的地地址是数据表中的某个。表号由符号TBNO标示，而符号TBNO是作为一个立即数加载的。每个表的前两个字节都标示出该表第一个自由字节的位移量。换句话说，我们假定每个表只被占用了一部分，数据块被移入所选择表的未占用的一端。所要求的数据传送可以描述如下：

这是相应的指令序列：

```
        LXI     D,TABX      ;LOAD BASE ADDRESS OF TABLE INDEX
        LXI     H,TBNO      ;LOAD TABLE NUMBER INTO HL
        DAD     D           ;COMPUTE ADDRESS OF TABLE BASE ADDRESS
        MOV     E,M         ;LOAD TABLE BASE ADDRESS INTO DE
        INX     H
        MOV     D,M
        XCHG                ;MOVE ADDRESS TO HL
        MOV     E,M         ;LOAD DISPLACEMENT TO FIRST FREE BYTE INTO DE
        INX     H
        MOV     D,M
        DAD     D           ;ADD TO HL, GIVING ADDRESS OF FIRST FREE BYTE
        LXI     D,CBASE     ;LOAD INPUT BUFFER BASE ADDRESS INTO DE

        LDA     CNT         ;LOAD BYTE COUNTER AND SAVE IN B
        MOV     B,A
LOOP    LDAX    D           ;MOVE NEXT BYTE FROM MEMORY LOCATION
        MOV     M,A         ;ADDRESSED BY DE TO LOCATION ADDRESSED BY HL
        INX     D           ;INCREMENT SOURCE AND DESTINATION
        INX     H           ;ADDRESSES
        DCR     B           ;DECREMENT BYTE COUNTER
        JNZ     LOOP        ;RETURN FOR MORE BYTES
```

```
LXI     D,TABX      取变址表基地址
LXI     H,TBNO      表号送 HL
DAD     D           求表基地址的地址
MOV     E,M         表基地址送 DE
INX     H
MOV     D,M
```

```
        XCHG            地址传送到 HL
        MOV     E,M     取到第一个自由字节地址的位移量
        INX     H
        MOV     D,M
        DAD     D       与 HL 相加，给出第一个自由字节地址
        LXI     D,CBASE 输入缓冲区基地址送 DE
        LDA     CNT     取字节计数值，并保存在 B 内
        MOV     B,A
LOOP:   LDAX    D       从存储单元传送下一个字节
        MOV     M,A     从 DE 指定的单元传送到 HL 指定的单元
        INX     D       源地址和目的地地址增 1
        INX     H
        DCR     B       字节计数值减 1
        JNZ     LOOP    如还有字节，返回
```

### 7-2-3 数据分类

到目前为止，我们说明了将数据从一个存储单元传送到另一个单元的两个简单程序设计的例子。因为整理数据也是很重要的，所以我们将在下面说明一个排序程序。

我们所描述的排序是将一串存储在相邻的存储单元中的带正负号的二进制数，按递增顺序把它们重新排列，最小的数排在前头，而最大的数排在最后。

我们要编写的分类程序采用“冒泡”算法。假定有一串数，由标号LIST标志出存储器中存放的第一个数的地址。下面是所需要的排序程序的操作：

**数据排序**

1) 在 LIST 开始启动一次扫描，将标志初始化以标示出“不交换”条件。

2) 比较一系列相邻的数，如果第一个数比第二个数小，无操作；否则，交换这两个数，并将标志置为“已交换”状态。

3）第二个数的地址与由标号 ENDL 所标示的表尾地址相比较。假如不是末尾，地址增 1 。因此当前这对数的第二个数变成了下一对数的第一个数，然后返回第二步。

4）如已达到表的末尾，检查“交换”标志。如果在扫描期间已经交换，返回第一步再做一次扫描。

5）如果这次扫描没有进行过交换，说明全部数已按次序排列好，转程序出口。

作为一个例子，假定数 1 到 10 的顺序是颠倒的，在第一遍扫描中将做九次交换，在第一遍扫描结束时，最大数将沉底。

| | 开始 | 第一遍扫描后 |
|---|---|---|
| LIST | 10 | 9 |
| | 9 | 8 |
| | 8 | 7 |
| | 7 | 6 |
| | 6 | 5 |
| | 5 | 4 |
| | 4 | 3 |
| | 3 | 2 |
| | 2 | 1 |
| ENDL | 1 | 10 |

另外八次扫描使全部数据按次序排好。所需要的第十次扫描是为了给出“不交换”出口条件而安排的。

SORT 是作为一个子程序来实现的。对于它所传送的参数存放在紧接子程序调用指令之后的单元中。规定了下列两个参数：

LIST　存放被排序各数据的数据缓冲区的起始地址

ENDL　存放被排序各数据的数据缓冲区的尾地址

## 下面是排序程序：

```
        CALL    SORT    ;CALLING SEQUENCE
        DW      LIST    ;ADDRESS OF START OF LIST
        DW      ENDL    ;ADDRESS OF END OF LIST (ON SAME PAGE AS LIST)
        .
        .
        .
SORT    POP     H       ;UNSTACK RETURN ADDRESS INTO HL
        MOV     E,M     ;LOAD LIST ADDRESS TO DE
        INX     H
        MOV     D,M
        INX     H
        MOV     C,M     ;LOAD LO BYTE OF ENDL ADDRESS TO C
        INX     H
        INX     H       ;INCREMENT PAST HO BYTE OF ENDL ADDRESS
        PUSH    H       ;STACK RETURN
        MOV     H,D     ;MOVE HO BYTE OF LIST ADDRESS FROM D TO H
        MVI     D,0     ;ZERO D AS A "NO SWAP" INDICATOR
LOOP1   MOV     L,E     ;MOVE LO BYTE OF LIST ADDRESS FROM E TO L
LOOP2   MOV     A,M     ;LOAD 1ST NUMBER OF PAIR TO A
        INR     L       ;INCREMENT LIST POINTER
        CMP     M       ;COMPARE (SUBTRACT MEMORY FROM A)
        JM      SORT1   ;JUMP IF MINUS (2ND NUMBER ALREADY > 1ST
                        ;NUMBER
        MOV     B,A     ;MOVE 1ST NUMBER TO B FROM A
        MOV     A,M     ;LOAD 2ND NUMBER TO A
        DCR     L       ;DECREMENT TO 1ST NUMBER LOCATION
        MOV     M,A     ;STORE 2ND NUMBER FROM A
        INR     L       ;INCREMENT LIST POINTER
        MOV     M,B     ;STORE 1ST NUMBER FROM B
        MVI     D,1     ;LOAD D WITH CONSTANT 1 AS A "SWAP MADE" FLAG
SORT1   MOV     A,C     ;MOVE LO BYTE OF ENDL ADDRESS TO A
        CMP     L       ;COMPARE WITH LO BYTE OF LIST ADDRESS IN L
        JNZ     LOOP2   ;LOOP BACK IF NOT AT END BYTE
        DCR     D       ;DECREMENT FLAG IN D
        JZ      LOOP1   ;LOOP BACK TO START LIST AGAIN IF SWAP WAS MADE
        RET             ;RETURN
```

```
        CALL    SORT    调用子程序
        DW      LIST    表起始地址
        DW      ENDL    表结束地址(和 LIST 在同一页内)
        .
        .
        .
SORT:   POP     H       将栈中的返回地址取出送 HL
        MOV     E,M     表地址送 DE
```

```
        INX   H
        MOV   D,M
        INX   H
        MOV   C,M       ENDL 地址的低位字节送 C
        INX   H
        INX   H         加 1，越过 ENDL 地址的高位字节
        PUSH  H         返回栈
        MOV   H,D       表地址高位字节从 D 传送到 H
        MVI   D,O       置 D为零，作为"不交换"指示符
LOOP1:  MOV   L,E       表地址低位字节从 E 传送 L
LOOP2:  MOV   A,M       一对数中第一个数送 A
        INR   L         表指示器增 1
        CMP   M         比较(从 A 中减去存储器单元内容)
        JM    SORT 1    如为负数转移(第二个数已经大于第一个数)
        MOV   B,A       第一个数从 A 送到 B
        MOV   A,M       第二个数送 A
        DCR   L         减 1，退回到第一个数地址
        MOV   M,A       存 A 中的第二个数
        INR   L         表指示器增 1
        MOV   M,B       存 B 中的第一个数
        MVI   D, 1      1 送 D，作为"已交换"标志
SORT 1: MOV   A,C       ENDL 地址的低位字节送 A
        CMP   L         与 L 中表地址的低位字节比较
        JNZ   LOOP 2    如不是结束字节，返回循环
        DCR   D         将 D 中的标志减 1
        JZ    LOOP 1    如已作过交换，再返回循环，开始查表
        RET             返回
```

## 7-3 运　　算

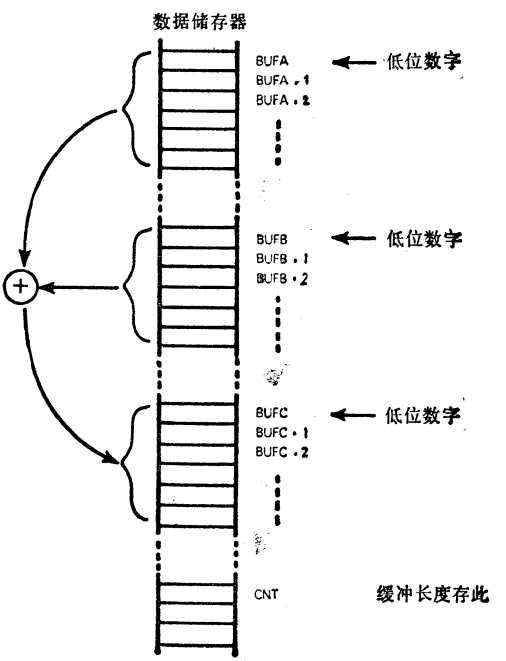
在以下内容中将叙述加法、减法、乘法和除法。因为超越函数比较复杂，需要专门的教科书来研究，因此我们将不予讨论。

就是在简单的加、减、乘、除范围内也存在着超过我们所涉及的内容以外的较困难的内容。由于参加运算的数的大小不同，所选用的算法也全然不同。因此，对于加法和减法我们将讨论大的和小的二进制、十进制数，而对于乘、除法则只涉及小的二进制数。

### 7-3-1 二进制加法

首先来看多字节的二进制加法。

设有两个正整数，都具有CNT个字节，彼此相加。它们的缓冲区的起始地址分别由B UFA 和 BUFB 给定。结果存放在起始地址为BUFC的缓冲区内。



多字节加法可以用图描述(见上页)。

这个指令序列实现了图中描述的加法：

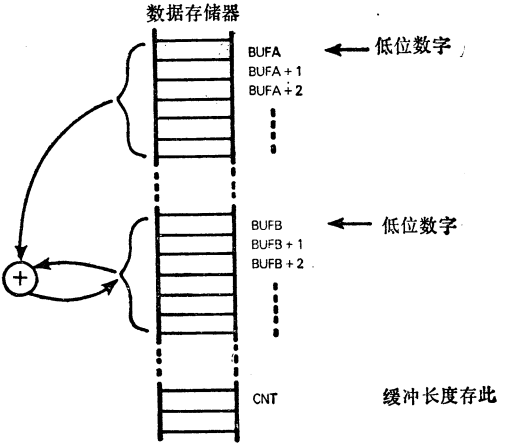
```
        LDA     CNT     ;LOAD BUFFER LENGTH AND SAVE IN C
        MOV     C,A
        LXI     H,BUFC  ;LOAD ANSWER BUFFER ADDRESS INTO H AND L
        PUSH    H       ;SAVE ON THE STACK
        LXI     D,BUFA  ;LOAD FIRST BUFFER ADDRESS INTO D AND E
        LXI     H,BUFB  ;LOAD SECOND BUFFER ADDRESS INTO H AND L
        XRA     A       ;CLEAR CARRY
LOOP    LDAX    D       ;LOAD NEXT BUF1 BYTE
        ADC     M       ;ADD NEXT BUF2 BYTE
        XTHL            ;SAVE IN NEXT ANSWER BUFFER BYTE
        MOV     M,A
        DCX     H       ;INCREMENT BUFC ADDRESS
        XTHL
        DCX     D       ;INCREMENT BUF1 ADDRESS

        DCX     H       ;INCREMENT BUFB ADDRESS
        DCR     C       ;DECREMENT COUNTER
        JNZ     LOOP    ;RETURN FOR MORE BYTES
```

```
        LDA     CNT     取出缓冲字节个数送C
        MOV     C,A
        LXI     H, BUFC 和数缓冲区地址送H和L
        PUSH    H       压入栈内
        LXI     D, BUFA 第一个缓冲区地址送D和E
        LXI     H, BUFB 第二个缓冲区地址送H和L
        XRA     A       清除进位位
LOOP:   LDAX    D       取下一个 BUFA* 字节
        ADC     M       与下一个BUFB* 字节相加
        XTHL            保存在下一个和数缓冲区字节内
        MOV     M, A
        INX     H       BUFC 地址增 1
        XTHL
        INX     D       BUFA 地址增 1
        INX     H       BUFB 地址增 1
```

注：原程序中 BUFI、BUFZ 应为BUFA、BUFB；倒数第三条指令 DCX 应为 INX。

```
DCR     C       计数值减1
JNZ     LOOP    如还有字节，返回
```

如果能把和数存到两个源缓冲器中的一个内，那么多字节加法就可以比较简单些了。




以下是简化后的指令序列：

```
        LDA     CNT     ;LOAD BUFFER LENGTH AND SAVE IN C
        MOV     C,A
        LXI     D,BUFA  ;LOAD FIRST BUFFER ADDRESS INTO D,E
        LXI     H,BUFB  ;LOAD SECOND AND ANSWER BUFFER ADDRESS INTO H,L
        XRA     A       ;CLEAR CARRY
LOOP    LDAX    D       ;LOAD NEXT BUFA BYTE
        ADC     M       ;ADD NEXT BUFB BYTE
        MOV     M,A     ;STORE ANSWER
        INX     D       ;INCREMENT BUFA ADDRESS
        INX     H       ;INCREMENT BUFB ADDRESS
        DCR     C       ;DECREMENT BUFFER LENGTH
        JNZ     LOOP    ;RETURN IF NOT END
```

```
LDA     CNT     取缓冲字节数送C
MOV     C,A
LXI     D,BUFA  取第一个缓冲区地址送D,E
LXI     H,BUFB  取第二个数与和数缓冲区地址送HL
XRA     A       清进位位
```

```
LOOP:  LDAX  D      取下一个 BUFA 字节
       ADC   M      与下一个 BUFB 字节相加
       MOV   M,A    存放和数
       INX   D      BUFA 地址增 1
       INX   H      BUFA 地址增 1
       DCR   C      缓冲字节数减 1
       JNZ   LOOP   如不是末尾，返回
```

### 7-3-2 二进制减法

因为 8080 设有专门的减法指令，所以二进制减法几乎与二进制加法相同。对于任意一个子程序只要简单地用 SBB 指令替换 ADC 指令，就能进行精确的二进制减法。

### 7-3-3 十进制加法

使用 8080 微型计算机进行十进制加法也是很容易的。在任何二进制加法程序中，简单地在 ADC 指令后插入一条 DAA 指令，就能进行十进制加法，例如：

```
LOOP   LDAX   D      ;LOAD NEXT BUF1 BYTE
       ADC    M      ;ADD NEXT BUF2 BYTE
       DAA           ;DECIMAL ADJUST RESULT
       XTHL          ;SAVE IN NEXT ANSWER BUFFER BYTE
        —
        —
        —
LOOP   LDAX   D   ; 取下一个 BUFA 字节
       ADC    M   ; 与下一个 BUFB 字节相加
       DAA       ; 按十进制修正结果
       XTHL      ; 保存下一个和数字节
        —
        —
        —
```

但是，有一点应当提起注意：十进制加法程序是假定有效的二—十进制数已存在源缓冲区中。如果在出错时，例如源缓冲区中的某一个数据出错，这将得出一个毫无意义的结果，并且不能发现它。

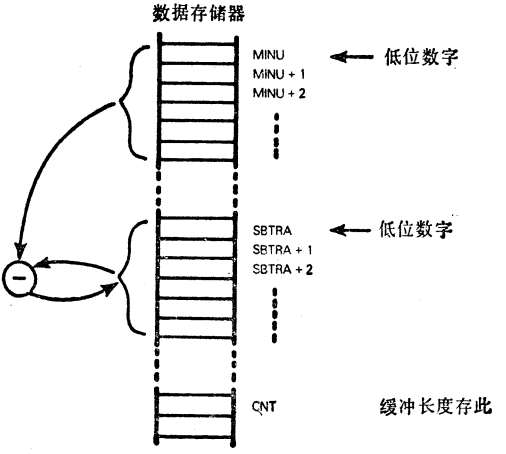
如果程序不能保证源缓区中是有效的二—十进制数，那么必须设计一个检查缓冲区内容的程序，保证在任何字节所包含的高 4 位和低 4 位单元中没有A到F之间的二进制码。

### 7-3-4 十进制减法

由于不能使用 8080 减法指令，十进制减法变得稍微复杂些。8080 减法指令只适用于二进制数，因为它们自动产生的是减数的二的补码。正如在“微型计算机入门”一书第一册中描述的那样，二—十进制减法要求给出减数的十的补码。

让我们回到简化后的二进制加法程序，并且按照这一加法程序生成一个等效的十进制减法程序。

这是相应的存储分配图：



下面是所要求的指令序列：

```
DSUB:   LXI     D,MINU      D AND E ADDRESS MINUEND
        LXI     H,SBTRA     ;H AND L ADDRESS SUBTRAHEND
        LDA     CNT         ;LOAD BUFFER LENGTH AND SAVE
        MOV     C,A         ;IN C
        STC                 ;SET CARRY INDICATING NO BORROW
LOOP:   MVI     A,99H       ;LOAD ACCUMULATOR WITH 99H
        ACI     0           ;ADD ZERO WITH CARRY
        SUB     M           ;PRODUCE NINES COMPLEMENT OF SUBTRAHEND
        XCHG                ;SWITCH D AND E WITH H AND L
        ADD     M           ;ADD MINUEND
        DAA                 ;DECIMAL ADJUST ACCUMULATOR
        XCHG                ;RESWITCH D AND E WITH H AND L
        MOV     M,A         ;STORE RESULT
        INX     D           ;ADDRESS NEXT BYTE OF MINUEND
        INX     H           ;ADDRESS NEXT BYTE OF SUBTRAHEND
        DCR     C           ;DECREMENT BYTE COUNT
        JNZ     LOOP        ;GET NEXT 2 DECIMAL DIGITS
DONE:   NOP
```

```
DSUB:   LXI     D, MINU     D和E存放被减数地址
        LXI     H, SBTRA    H和L存放减数地址
        LDA     CNT         取缓冲长度存入C
        MOV     C, A
        STC                 置进位位，表示无借位
LOOP:   MVI     A, 99H      99H送累加器
        ACI     0           连同进位位加零
        SUB     M           产生减数的9的补码
        XCHG                DE和HL交换内容
        ADD     M           加被减数
        DAA                 十进制调整累加器
        XCHG                DE和HL交换内容
        MOV     M, A        存放结果
        INX     D           产生下一个被减数字节地址
        INX     H           产生下一个减数字节地址
        DCR     C           字节计数器内容减1
```

```
        JNZ     LOOP        转下两个十进制数字
DONE:   NOP
```

## 7-4 乘法和除法

在微型计算机系统中，乘法和除法必须作为一个值得注意的问题来研究。乘法和除法的操作对于微型计算机的组织来说是不适宜的，因为任何复杂的乘法和除法只要一执行，就会严重地降低整机的性能。假如要用微型计算机来进行大量的乘法、除法或超越函数运算的话，那么应当从现有的各种计算器/算术运算芯片中选择一种商品，将复杂的算术运算移到这个芯片中进行。这样就能使微型计算机系统从不能适应转而为能够适应你的需要。

你可以用微型计算机进行一些不常用的或者不太费时间的简单的乘法和除法。

因此，我们将介绍几个简单的程序序列。

### 7-4-1 八位二进制乘法

先看两个不带正负号的8位数据值的乘法，乘积是16位。实现这个乘法的最简单的方法是用零加乘数，相加的次数由被乘数给定。例如，4乘以3可以用0加三次4得到：

```
        MVI     A,0     ;CLEAR AB TO INITIALIZE ANSWER
        MOV     B,A     ;BUFFER
        CMP     D       ;TEST FOR 0 IN D
        RZ              ;IF 0, ANSWER IS 0 SO END
LOOP    ADD     E       ;ADD MULTIPLIER TO LOW ORDER ANSWER BYTE
        JNC     NEXT    ;IF CARRY IS SET, INCREMENT B
        INR     B
NEXT    DCR     D       ;DECREMENT MULTIPLICAND
        JNZ     LOOP    ;IF NOT ZERO, RETURN TO ADD AGAIN
        RET
```

```
MVI   A，0   清AB，初始化
MOV   B，A
```

```
        CMP   D       测试D是否为零
        RZ            如D为零，乘积则为零，结束
LOOP:   ADD   E       乘数加到乘积的低位字节
        JNC   NEXT    如进位位置位，B增1
        INR   B
NEXT    DCR   D       被乘数减1
        JNZ   LOOP    如不为零，返回再加
        RET
```

有一个乘法的简便算法。由于我们能够使用的二进制数字仅限于0和1，因此这个方法对于每一位数字来说，就是把乘法简化成加或者不加。

让我们解释这个概念。试用普通的十进制表示法来观察如下的乘法：

```
   142      被乘数
 x 307      乘数
 42600
  0000      部分积
   994
 43594      乘积
```

142 = 乘数
307 = 被乘数

把 7 × 乘数加到乘积上
乘数左移二位，然后乘以 3 并加到乘积上

每个部分积都等于被乘数乘以乘数的一位，部分积左移，右边填以零，填到右边的零的个数等于当前该位乘数右边的位数。

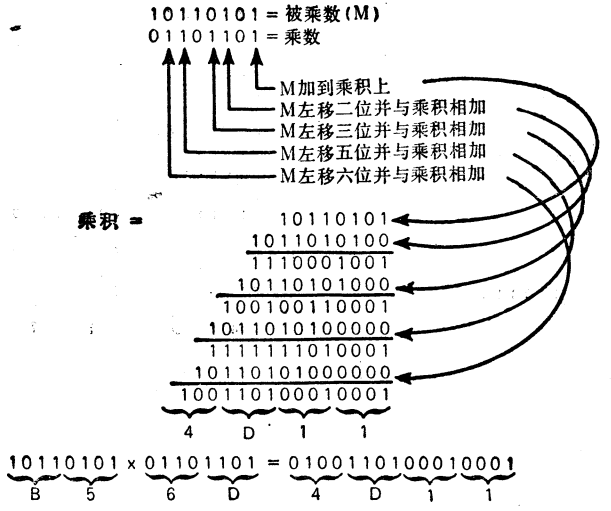
```
 142
 3xx
42600
```

因为 3 的右边有两个零，故填两个零
142 x 3

我们可以把同样的概念延伸到二进制运算，这时问题变得

很简单，因为二进制数字除 0 或 1 以外不可能有其它的值。在这种情况下，只能有两种选择：如果乘数的某一位是 0，则这个被移位的被乘数不加到乘积上；但是如果乘数的这一位是 1，就将移位后的被乘数加到乘积上。

下面是一个例子：




按如下步骤使用“移位—相加”的方法，一个单字节的被乘数乘以一个单字节的乘数，就能产生一个正确的二字节的结果：

a）检查乘数最低有效位，如为 0 转做 b 项；如为 1，把被乘数加到结果的最高有效字节。

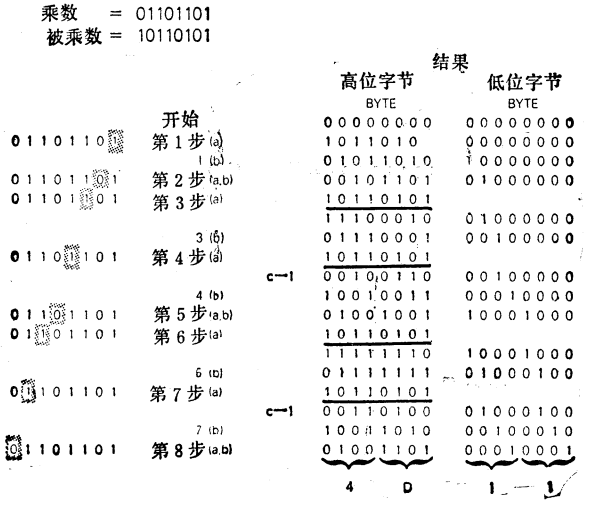
b）结果的两个字节一起右移一位。

c）重复 a 和 b 项，直到全部 8 位乘数都做完为止。

试看 B5＊6D，这是我们刚才描述过的二进制乘法：

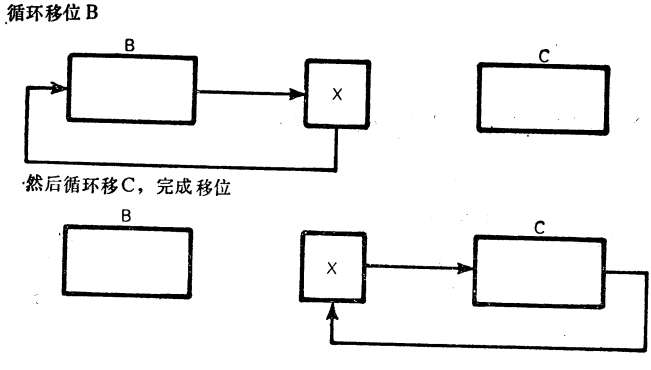
乘数　＝01101101

被乘数＝10110101




我们现在写一个程序来实现这个乘法算法。

寄存器 B 将存放结果的最高有效位字节，寄存器 C 将存放结果的最低有效位字节。结果的 16 位右移是由两条通过进位位循环右移指令来完成的，具体步骤如下图所示：



寄存器D存放被乘数，寄存器C最初存放乘数。

程序如下：

```
MULT    MVI     B,0     ;INITIALIZE MOST SIGNIFICANT BYTE
                        ;OF RESULT
        MVI     E,9     ;BIT COUNTER
MULT0   MOV     A,C     ;ROTATE LEAST SIGNIFICANT BIT OF
        RAR             ;MULTIPLIER TO CARRY AND SHIFT
        MOV     C,A     ;LOW ORDER BYTE OF RESULT
        DCR     E
        JZ      DONE    ;EXIT IF COMPLETE
        MOV     A,B
        JNC     MULT1
        ADD     D       ;ADD MULTIPLICAND TO HIGH ORDER BYTE
                        ;OF RESULT IF BIT WAS A ONE
MULT1   RAR             ;CARRY=0 HERE; SHIFT HIGH ORDER
                        ;BYTE OF RESULT
        MOV     B,A
        JMP     MULT0
DONE
```

```
MULT:   MVI   B，0      结果的最高有效位字节置初值
        MVI   E，9      置入位计数器
MULT 0: MOV   A，C      将乘数的最低有效位向进位位
        RAR             循环右移，结果的低位字节向右移位
        MOV   C，A
        DCR   E
        JZ    DONE      如完成，转出口
        MOV   A，B
        JNC   MULT 1
        ADD   D         如该位是1，被乘数和结果的高位字节相加
MULT 1: RAR             进位位为零转此，右移结果的高位字节
        MOV   B，A
        JMP   MULT 0
DONE:
```

## 7-4-2 八位二进制除法

用一个类似的方法，将一个不带正负号的16位数除以不带正负号的8位数。这里，在运算过程中用的是减法而不是加法，

并且用循环左移指令代替循环右移指令。

程序用寄存器B和C分别存放被除数的高位和低位字节；寄存器D存放除数，得到的8位商存放在寄存器C中，余数存入寄存器B中。

```
DIV     MVI     E,9         ;BIT COUNTER
        MOV     A,B
DIV0    MOV     B,A
        MOV     A,C         ;ROTATE CARRY INTO C REGISTER; ROTATE
        RAL                 ;NEXT MOST SIGNIFICANT BIT TO CARRY.
        MOV     C,A
        DCR     E
        JZ      DIV1
        MOV     A,B         ;ROTATE MOST SIGNIFICANT BIT TO
        RAL                 ;HIGH ORDER QUOTIENT
        SUB     D           ;SUBTRACT DIVISOR. IF LESS THAN
        JNC     DIV0        ;HIGH ORDER QUOTIENT, GO TO DIV0
        ADD     D           ;OTHERWISE ADD IT BACK
        JMP     DIV0
DIV1    RAL
        MOV     E,A
        MVI     A,FFH       ;COMPLEMENT THE QUOTIENT
        XRA     C
        MOV     C,A
        MOV     A,E
        RAR
DONE
```

```
DIV:    MVI   E, 9      置入位计数器
        MOV   A,B
DIVO:   MOV   B,A
        MOV   A,C       进位位循环移入寄存器C
        RAL             下一最高有效位循环移入进位位
        DCR   E
        JZ    DIV 1
        MOV   A,B       将最高有效位向高位商循环左移
        RAL
        SUB   D         如小于高位商，减除数
        JNC   DIVO      转向 DIVO
        ADD   D         否则，把除数加回去
```

```
        JMP   DIVO
DIV1:   RAL
        MOV   E,A
        MVI   A,FFH   商数取反
        XRA   C
        MOV   C, A
        MOV   A, E
        RAR
DONE
```

## 7-4-3 十六位二进制乘法

现在讨论两个 16 位数的乘法，得出一个 32 位的结果。

使用的算法如下：

1) 通过进位位左移乘数(高位 16 位)和部分积(低位16位)

2) 假如有一个 1 从乘数移入进位位，就将 16 位的被乘数加到部分积的三个字节中去

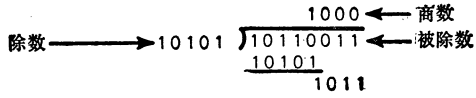
下面是所需要的指令序列：

```
MPY     LXI     H,0     ;INITIALIZE PARTIAL PRODUCT IN HL TO ZERO
        MVI     A,16    ;INITIALIZE COUNT
LOOP    DAD     H       ;ADD HL TO HL  - LEFT SHIFT LOGICAL INTO CARRY
        XCHG            ;EXCHANGE HL AND DE
        JC      MPY1    ;JUMP IF CARRY OUT FROM HL
        DAD     H       ;NO CARRY — SHIFT MULTIPLIER LEFT LOGICAL INTO CAR
                         RY
        JMP     MPY2    ;JUMP
MPY1    DAD     H       ;CARRY-SHIFT MULTIPLIER LEFT LOGICAL INTO CARRY
        INX     H       ;AND INCREMENT
MPY2    XCHG            ;REPOINT TO PARTIAL PRODUCT
        JNC     MPY3    ;JUMP IF NO ADD (MULTIPLIER BIT IN CARRY=0)
        DAD     B       ;ADD MULTIPLICAND IN BC TO PARTIAL PRODUCT IN HL
        JNC     MPY3    ;JUMP IF NO CARRY OUT
        INX     D       ;INCREMENT DE TO ADD CARRY
MPY3    DCR     A       ;DECREMENT COUNT
        JNZ     LOOP    ;LOOP BACK IF NOT ZERO
        RET             ;RETURN
```

```
MPY:  LXI   H,0     HL 中的部分积初始化置零
      MVI   A,16    计数器置初值
LOOP: DAD   H       HL 加 HL，逻辑左移，移入进位位
      XCHG          HL 与 DE 交换
      JC    MPY 1   如果 HL 有进位输出，转移
      DAD   H       无进位，乘数逻辑左移，进入进位位
      JMP   MPY 2   转移
MPY 1: DAD  H       若有进位，乘数逻辑左移进入进位位并增 1
      INX   H
MPY 2: XCHG         再指向部分积
      JNC   MPY 3   如不加，转移(在进位中的乘数
                    位等于零)
      DAD   B       BC 中的被乘数与 HL 中的部分积相加
      JNC   MPY 3   如无进位，转移
      INX   D
MPY 3: DCR  A       计数器值减 1
      JNZ   LOOP    如不为零，返回循环
      RET           返回
```

### 7-4-4 二进制除法

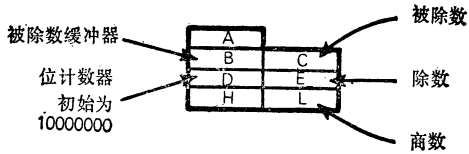
试看简单的 8 位数除法，例如 $B3_{16}$ 除以 $15_{16}$ 可以描述如下：




结果是 $8_{16}$，余数是 $B_{16}$。

除法算法是把被除数移入一个初始清零的寄存器。每当被除数移位缓冲器的内容超过了除数时，总是从移位缓冲器内容中减去除数，并且将二进制数字 1 添到对应的商的位置上去。

假设寄存器分配如下：




开始时，假定除数在寄存器E，被除数在寄存器C，产生的商在寄存器L。

下面是一个根据上述算法得到的除法程序：

```
INITIALLY CLEAR REGISTERS A, B, L AND CARRY
        XRA     A           ;EXCLUSIVE OR A WITH ITSELF. THIS CLEARS A
        MOV     B,A         ;CLEAR B
        MOV     L,A         ;CLEAR L
;INITIALIZE BIT COUNTER IN REGISTER D
        MVI     D,80H
SHIFT B AND C, AS A 16-BIT UNIT, ONE BIT LEFT
LOOP    MOV     A,C
        RAL
        MOV     C,A
        MOV     A,B
        RAL
        MOV     B,A
;COMPARE DIVISOR (IN E) WITH DIVIDEND. SHIFT BUFFER,
;CURRENTLY STILL IN A
        CMP     E
        JC      NEXT        ;IF DIVISOR IS LARGER, BYPASS SUBTRACT
;DIVISOR IS SMALLER. SUBTRACT FROM
;DIVIDEND SHIFT BUFFER
        SUB     E
        MOV     B,A
;SET TO 1 CURRENT BIT OF L. THE CURRENT BIT IS
;THE 1 BIT POSITION IN D
        MOV     A,D
        ORA     L
        MOV     L,A
;SHIFT D RIGHT ONE BIT POSITION AND CLEAR CARRY
NEXT    MOV     A,D
        RRC
        JNC     LOOP        ;IF CARRY IS NOT SET, RETURN FOR NEXT BIT
```

初始化清寄存器A、B、C和进位位

```
XRA   A     A与本身相异，清除A
MOV   B,A   清除B
MOV   L,A   清除L
```

```
寄存器D中的位计数器置初值
        MVI   D, 80H
B,C 作为一个 16 位数，左移一位
LOOP:   MOV   A,C
        RAL
        MOV   C,A
        MOV   A,B
        RAL
        MOV   B,A
除数(在E中)与被除数比较，移位缓冲器的当前内容在A中。
        CMP   E
        JC    NEXT   如除数大，不减
如果除数小于被除数，则从被除数移位缓冲器的内容中减去除数。
        SUB   E
        MOV   B,A
L的当前位置1，此当前位就是D的第1位
        MOV   A,D
        ORA   L
        MOV   L,A
D右移一位，并清除进位位
NEXT:   MOV   A,D
        RRC
        JNC   LOOP   如进位未置位，返回到下一位
```

结束时，商在L中，如有余数则在B中。

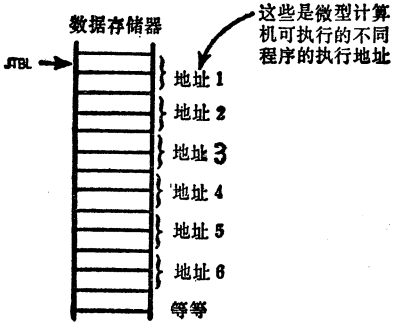
## 7-5 程序执行顺序逻辑转移表

在这一标题之下，需要解释的其实只有一个程序序列，它就是转移表。

我们知道 8080 指令系统中具有丰富的条件转移指令；其中包括按条件的转移、调用和返回指令。它们都具有 8 种不同

的条件作为是否执行的根据。这就意味着，当一个逻辑流程只是从两个分支中选择一个的时候，就不需要采用专门的程序。然而当你有三个或更多的选择的时候，转移表便成为有效的程序设计工具了。

转移表的核心是一些存储在相继存储器字节对中的16位地址序列：




我们假定这些相继的存储器地址表示了许多不同程序的起始地址。假定所要求的程序是用累加器中的程序号来识别的。下列指令序列能够引起向程序号在累加器中的程序进行转移。

```
;JUMP TABLE PROGRAM
        LXI     H,JTBL      ;LOAD JUMP TABLE BASE ADDRESS IN HL
        RLC                 ;MULTIPLY ACCUMULATOR BY 2
        ADD     L           ;ADD LOW ORDER ADDRESS BYTE TO A
        MOV     L,A         ;RESTORE SUM TO L
        MVI     A,0         ;ADD CARRY (IF ANY) TO H
        ADC     H
        MOV     H,A
        MOV     E,M         ;HL ADDRESSES REQUIRED ADDRESS
        INX     H           ;LOAD REQUIRED ADDRESS INTO D,E
        MOV     D,M
        XCHG                MOVE ADDRESS FROM DE TO HL
        PCHL                ;MOVE ADDRESS FROM HL TO PC
```

转移表程序：

```
        LXI     H,JTBL      转移表基地址送 HL
        RLC                 累加器内容乘 2
```

```
ADD     L       低位地址字节与A相加
MOV     L,A     和数再存入L
MVI     A,0     (若有)进位位则与H相加
ADC     H
MOV     H,A
MOV     E,M     HL 指向所要求的地址
INX     H       将所要求的地址送入 DE
MOV     D,M
XCHG            地址从 DE 传送到 HL
PCHL            地址从 HL 传送到 PC
```

# 附录 A　标准字符代码

| 十六进制数表示 | ASCII (7 位) | EBCDIC (8 位) |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| A | | |
| B | | |
| C | | |
| D | | |
| E | | |
| F | | |
| 10 | | |
| 11 | | |
| 12 | | |
| 13 | | |
| 14 | | |
| 15 | | |
| 16 | | |
| 17 | | |
| 18 | | |
| 19 | | |
| 1A | | |
| 1B | | |
| 1C | | |
| 1D | | |
| 1E | | |
| 1F | | |
| 20 | blank | |
| 21 | ! | |
| 22 | " | |
| 23 | # | |
| 24 | $ | |
| 25 | % | |
| 26 | & | |
| 27 | ' | |
| 28 | ( | |
| 29 | ) | |
| 2A | * | |
| 2B | + | |
| 2C | , | |
| 2D | − | |
| 2E | . | |
| 2F | / | |
| 30 | 0 | |

| 十六进制数表示 | ASCII (7位) | EBCDIC (8 位) |
|---|---|---|
| 31 | 1 | |
| 32 | 2 | |
| 33 | 3 | |
| 34 | 4 | |
| 35 | 5 | |
| 36 | 6 | |
| 37 | 7 | |
| 38 | 8 | |
| 39 | 9 | |
| 3A | : | |
| 3B | ; | |
| 3C | < | |
| 3D | = | |
| 3E | > | |
| 3F | ? | |
| 40 | @ | blank |
| 41 | A | |
| 42 | B | |
| 43 | C | |
| 44 | D | |
| 45 | E | |
| 46 | F | |
| 47 | G | |
| 48 | H | |
| 49 | I | |
| 4A | J | ¢ |
| 4B | K | . |
| 4C | L | < |
| 4D | M | ( |
| 4E | N | + |
| 4F | O | \| |
| 50 | P | & |
| 51 | Q | |
| 52 | R | |
| 53 | S | |
| 54 | T | |
| 55 | U | |
| 56 | V | |
| 57 | W | |
| 58 | X | |
| 59 | Y | ` |
| 5A | Z | ! |
| 5B | [ | $ |
| 5C | \ | * |
| 5D | ] | ) |
| 5E | | ; |
| 5F | | ¬ |
| 60 | | - |
| 61 | a | |

# 附录 A（续）

| 十六进制数表示 | ASCII (7 位) | EBCDIC (8 位) |
|---|---|---|
| 62 | b | |
| 63 | c | |
| 64 | d | |
| 65 | e | |
| 66 | f | |
| 67 | g | |
| 68 | h | |
| 69 | i | |
| 6A | j | |
| 6B | k | . |
| 6C | l | % |
| 6D | m | - |
| 6E | n | ) |
| 6F | o | ? |
| 70 | p | |
| 71 | q | |
| 72 | r | |
| 73 | s | |
| 74 | t | |
| 75 | u | |
| 76 | v | |
| 77 | w | |
| 78 | x | |
| 79 | y | |
| 7A | z | : |
| 7B | | # |
| 7C | | @ |
| 7D | | ' |
| 7E | | = |
| 7F | | " |
| 80 | | |
| 81 | | a |
| 82 | | b |
| 83 | | c |
| 84 | | d |
| 85 | | e |
| 86 | | f |
| 87 | | g |
| 88 | | h |
| 89 | | i |
| 8A | | |
| 8B | | |
| 8C | | |
| 8D | | |
| 8E | | |
| 8F | | |
| 90 | | |
| 91 | | j |
| 92 | | k |
| 93 | | l |
| 94 | | m |
| 95 | | n |
| 96 | | o |

| 十六进制数表示 | ASCII (7 位) | EBCDIC (8 位) |
|---|---|---|
| 97 | | p |
| 98 | | q |
| 99 | | r |
| 9A | | |
| 9B | | |
| 9C | | |
| 9D | | |
| 9E | | |
| 9F | | |
| A0 | | |
| A1 | | |
| A2 | | s |
| A3 | | t |
| A4 | | u |
| A5 | | v |
| A6 | | w |
| A7 | | x |
| A8 | | y |
| A9 | | z |
| AA | | |
| AB | | |
| AC | | |
| AD | | |
| AE | | |
| AF | | |
| B0 | | |
| B1 | | |
| B2 | | |
| B3 | | |
| B4 | | |
| B5 | | |
| B6 | | |
| B7 | | |
| B8 | | |
| B9 | | |
| BA | | |
| BB | | |
| BC | | |
| BD | | |
| BE | | |
| BF | | |
| C0 | | |
| C1 | | A |
| C2 | | B |
| C3 | | C |
| C4 | | D |
| C5 | | E |
| C6 | | F |
| C7 | | G |
| C8 | | H |
| C9 | | I |
| CA | | |
| CB | | |