# 第3章 系统软硬件构造

## 引言

艾伦·图灵奠定了计算机的理论基础，冯·诺依曼则创建了现代计算机的体系结构和基本原理。历经半个多世纪的发展，计算机的功能虽早已今非昔比，但其工作原理和体系结构在总体上依然是冯·诺依曼计算机结构。

也许很多读者没有考虑过今天已认为是理所当然的问题：为什么可以在屏幕上同时打开多个“窗口”？为什么总希望内存越大越好？文件存到了硬盘的什么地方？是怎么放进去的？等等。这一系列问题的答案就隐藏在计算机硬件系统和操作系统工作原理中。

本章从基本逻辑运算及其门电路入手，试图借助推理和“搭积木”的思维模式，去解析系统的“构造”过程，帮助读者理解什么是“抽象”和“封装”，并揭晓上述问题的答案。

## 教学目的

- 理解逻辑运算及基本逻辑门。
- 理解常用逻辑电路及其构造过程。
- 深入理解冯·诺依曼计算机结构和工作原理。
- 了解图灵机和计算机的关系。
- 了解冯·诺依曼结构的不足。
- 了解操作系统的基本功能。
- 深入理解进程的基本状态及进程的生命周期。
- 理解操作系统中存储器管理的功能。
- 了解文件、文件的组织结构及文件管理的功能。

## 3.1 逻辑代数基础

在2.1节中，详细讨论了计算机采用二进制的理由。其实，简单的根源就是二进制能够与开关元件的“开”和“关”两种状态一一对应起来。像开关元件这样只有两种可能状态

的器件称为逻辑器件。现代计算机正是由各种各样的逻辑器件组成的，而它的数学基础就是逻辑代数。

逻辑代数的发明人是英国数学家乔治·布尔(George Boole，1815—1864)，所以也称为布尔代数，主要研究和判断相关的逻辑运算，被广泛应用于开关电路和数字逻辑电路的分析中。

逻辑代数用字母表示变量(称逻辑变量)，只有 0 和 1 两个取值。逻辑代数表示的是逻辑关系，不是数学关系。所以，逻辑运算与算术运算不同，算术运算是将一个二进制数的所有位视为一个数值整体来考虑，低位的运算结果会影响到高位(如进位等)；而**逻辑运算是按位进行的运算**，其低位运算结果不会对高位产生影响。即逻辑运算没有进位或借位。

## 3.1.1 关于逻辑

从哲学的角度讲，逻辑(logic)是反映思维的规律。或者说，是推论和证明的思想过程。逻辑的基本表现形式是命题和推理。

**推理**是从前提推出结论的思维过程，前提是已知的命题，结论是通过推理规则得出的命题。所以，归根结底，要说清楚逻辑运算，需要先说清楚什么是命题。

### 1. 命题

所谓命题，简单地讲，就是能判断真假的陈述句。这种陈述句的判断只有两种可能，即正确和错误。命题的判断(可以理解为就是这句话表示的意思)叫做**真值**。真值为正确的命题称为“真”，为错误的命题称为“假”。因此，又可以说命题是具有确定的“真”或“假”的陈述句。比如，“水是无色的”，这是真命题；“冰是有色的”，就是假命题。

**【例 3-1】** 判断下列语句是否为命题。

(1) 雪是白色的。

(2) 2 是素数。

(3) 5 能被 3 整除。

(4) 明天上午有课吗？

(5) $x+y>8$。

(6) $2+5=7$。

在这 6 条语句中，(4)是问句，不是陈述句，所以不是命题；(1)～(3)和(6)是陈述句，并且具有明确的判断(即正确或错误)，所以它们是命题。其中，(1)、(2)和(6)所表达的含义是正确的(我们也说它们的“真值”是“真”的)，所以它们是真命题，(3)所表达的含义是错误的(即它们的“真值”是“假”的)，所以(3)是假命题；(5)虽然是陈述句，但它不是命题，因为它没有确定的真值，只有当 $x$ 和 $y$ 为给定值后，该语句才能成为命题。

从以上分析可以得出，判断一个句子是否为命题，首先看它是否为陈述句，其次看它的真值是否是唯一确定的。

语句(1)～(3)和(6)都是简单陈述句，不可再分解，因此也称它们为简单命题。

【例 3-2】 观察以下语句，判断是否为命题。

(1) 每位德国学生既要学习英语，也要学习法语。

(2) 从西安到北京可以乘火车去，或者乘飞机去。

(3) 3 不是偶数。

例 3-2 中第(1)句话可以说成“每位德国学生既要学习英语，并且也要学习法语”。这里，“每位德国学生要学习英语”是一个简单命题，“每位德国学生要学习法语”也是一个简单命题，且它们的真值都为真(亦即都是正确的)。这两个简单命题用“并且”联结在一起，表示了一种“同时存在”或“同时为真”的意思。

第(2)句话，用“或者”这个词将“从西安到北京可以乘火车去”和“从西安到北京可以乘飞机去”这两个简单命题联结在一起，表示两者任意一种都可以，或说只要有一个存在就可以(事实上，也不可能一个人同时又坐火车又乘飞机)。

第(3)句话的“不是偶数”也可以说成是“并非偶数”，表示了“是”的反意(将“是”的意思翻转)。

这里的“并且”、“或者”、“并非”在命题逻辑中称为联结词。这种用联结词联结起来的命题称为复合命题(例 3-2 中的 3 条语句都是复合命题)。而这些联结词则表示着一种关系或者运算。

由此可以得出，**任何复合命题都可以由简单命题通过联结词所表示的某种运算得到**。

### 2. 命题的符号化

将一个简单命题用英文字母来表示，称为命题的符号化。例如：

$A$：雪是黑色的。

$B$：2 是素数。

$P$：每位德国学生要学习英语。

$Q$：每位德国学生要学习法语。

这里，$A$ 是假命题，其余是真命题。

简单命题的真值是确定的，所以真值也可以符号化。通常用 1 表示真，用 0 表示假。另外，联结词也可以符号化。如果将上文提到的联结词“并且”、“或者”、“并非”，分别用 and、or、not 来表示，则例 3-2 中的 3 条复合命题就可以符号化为如下形式：

(1) $P$ and $Q$ ($P$：每位德国学生要学习英语，$Q$：每位德国学生要学习法语)

(2) $R$ or $S$ ($R$：从西安到北京可以乘火车去，$S$：从西安到北京可以乘飞机去)

(3) not $T$ ($T$：3 是偶数)

可以看出，当命题符号化之后，一个复杂的命题是可以由简单命题通过逻辑运算得到的。在以上符号化后的 3 条语句中：

- and 所表示的“并且”的含义是：同时存在，或同时为“真”。这种关系称为逻辑“与”。
- or 表示的“或者”，是意味着只要有一个存在或者说一个为“真”就可以，这种关系称为逻辑“或”。
- not 表示了“并非”，意为取反(将“3 是偶数”取反为“3 不是偶数”)，称为逻辑“非”。

命题是逻辑的基本表现形式，所以，联结词所表示的运算就是逻辑运算。由于逻辑的“真”和“假”，可以符号化为二进制的 1 和 0。因此，逻辑运算也就是 1 和 0 之间的运算。既然如此，逻辑运算和逻辑推理也就可以被计算机处理了。

## 3.1.2 基本逻辑运算

基本逻辑运算包括“与”、“或”、“非”3 种，在此基础上还可以演变出其他各种复杂的逻辑关系。

### 1. 逻辑“与”

逻辑“与”的含义是：当全部条件都满足时，结果才会发生。或者可以描述为：当输入条件全部为“真”时，输出的结果为“真”；若输入有一个条件不满足（为“假”），则结果就为“假”。这一点可以用从图 2-2 所示的二极管电路得出。

“与”运算用符号 $\wedge$ 或者 $\cdot$ 表示，遵循如下运算规则：

$$1 \wedge 1 = 1 \quad 1 \wedge 0 = 0 \quad 0 \wedge 1 = 0 \quad 0 \wedge 0 = 0 \tag{3.1}$$

式(3.1)所示规则的含义是：参加“与”操作的两位中只要有一位为 0，则相“与”的结果就为 0；仅当两位均为 1 时，其结果才为 1。

“与”运算相当于按位相乘（但不进位），所以又叫做“逻辑乘”。可以表示为

$$Y = A \wedge B \quad \text{或者} \quad Y = A \cdot B \tag{3.2}$$

式(3.2)的含义是：仅当 $A$ 和 $B$ 全部为“真”时，结果 $Y$ 才为“真”。

对两个多位二进制数来讲，逻辑运算执行按位运算（算术运算是按数据运算），对“与”运算来讲，就是两个数按位相“与”。

**【例 3-3】** 计算 11011010B∧10010110B。

**解：**

$$\begin{array}{r} 11011010 \\ \wedge\ 10010110 \\ \hline 10010010 \end{array}$$

即 11011010B∧10010110B=10010010B。

“与”运算通过称为“与门”的逻辑器件实现（详见 3.1.3 节），其逻辑关系反映在电路中，相当于开关的串联（如表 3-2 所示）。

### 2. 逻辑“或”

逻辑“或”的含义是：在全部输入中，只要有一项条件满足，结果就会发生。“或”运算用符号 $\vee$ 表示，遵循如下运算规则：

$$0 \vee 0 = 0 \quad 0 \vee 1 = 1 \quad 1 \vee 0 = 1 \quad 1 \vee 1 = 1 \tag{3.3}$$

运算规则表示：参加“或”运算的两位二进制数中，仅当两位均为 0 时，其结果才为 0；只要有一位为 1，则“或”的结果就为 1。该规则还可以表述为：当且仅当输入全部为假时，输出结果才为假。

“或”运算执行两个数按位相“或”的运算，又叫做“逻辑加”，可以表示为

$$Y = A \vee B \quad \text{或者} \quad Y = A + B \tag{3.4}$$

式(3.4)的含义是：当且仅当逻辑变量 $A$ 和 $B$ 均为0(假)时，$Y$ 为假；$A$ 和 $B$ 任意一个为1(真)，则 $Y$ 为真。其逻辑关系可以用表3-1的形式表示。由于表中反映了输出 $Y$(复合命题的真值)和输入 $A$、$B$(简单命题的真值)的逻辑关系，所以，该表也称为**真值表**。

**表 3-1 “或”逻辑关系真值表**

| *A* | *B* | *Y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**【例 3-4】** 计算 11011001B∨10010110B。

**解：**

$$\begin{array}{r} 11011001 \\ \vee\ 11111111 \\ \hline 11111111 \end{array}$$

即 11011001B∨11111111B = 11111111B。

“或”逻辑关系反映在电路中，相当于开关的并联(参见表3-2)。

同“与”运算类似，“或”运算通过称为“或门”的逻辑器件实现。

**思考** 试比较二进制数的“或”运算和加法运算的异同。

### 3. 逻辑“非”

逻辑“非”的含义是：当决定事件结果的条件满足时，事件不发生。“非”运算是按位取反的运算，即，1的“非”为0，而0的“非”为1。或者表述为：“真”的“非”为假，反之亦然。

“非”属于单边运算，即只有一个运算对象，其运算符为一条上横线。规则如下：

$$\overline{1} = 0 \quad \overline{0} = 1 \tag{3.5}$$

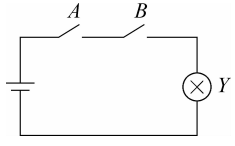
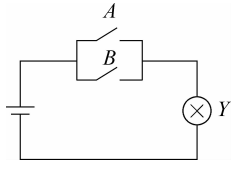
**【例 3-5】** 求数 10011011 的“非”。

**解：**只要对 10011011 按位取反即可。得：

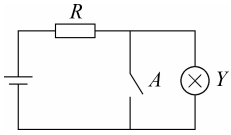
$$\overline{10011011\text{B}} = 01100100\text{B}$$

表3-2给出了这3种基本逻辑在开关电路中的表示。

**表 3-2 逻辑关系及其电路表述**

| 逻辑关系 | 运算符 | 逻辑关系描述 | 电路表示 |
|---|---|---|---|
| 逻辑与 | ∧ | 当且仅当输入全为1(真)时，输出才为1(真) |  |
| 逻辑或 | ∨ | 当且仅当输入全为0(假)时，输出才为0(假) |  |

续表

| 逻辑关系 | 运算符 | 逻辑关系描述 | 电路表示 |
| --- | --- | --- | --- |
| 逻辑非 | 上横线 | 输入为1(真),输出则为0(假),反之亦然 |  |



## 3.1.3 其他逻辑运算

通过对基本逻辑关系的变换,可以生成其他一些逻辑关系。常见的有“与非”运算、“或非”运算、“异或”运算和“同或”运算等。

### 1. “与非”运算

“与非”运算是“与”运算和“非”运算的组合,是对“与”运算的结果再求“非”。可以用以下逻辑函数表示:

$$Y = \overline{A \wedge B} \tag{3.6}$$

**【例 3-6】** 设 $A$=11011010B,$B$=10010110B,计算 $Y=\overline{A \wedge B}$。

**解**:先计算 $A \wedge B$,有

$$\begin{array}{r} 11011010 \\ \wedge\ 10010110 \\ \hline 10010010 \end{array}$$

再对相“与”的结果按位取反,就得到“与非”运算结果:$\overline{10010010}$=01101101。

### 2. “或非”运算

和“与非”运算类似,“或非”运算是“或”运算和“非” 运算的组合。即在或运算基础上,再对结果求“非”。“或非”运算的逻辑函数表达式为

$$Y = \overline{A \vee B} \tag{3.7}$$

**【例 3-7】** 设 $A$=11011001B,$B$=10010110B,计算 $Y=\overline{A \vee B}$。

**解**:先计算 $A \vee B$,有

$$\begin{array}{r} 11011001 \\ \vee\ 11111111 \\ \hline 11111111 \end{array}$$

再对结果求“非”,得:$Y=\overline{A \vee B}=\overline{11011001 \vee 11111111}=\overline{11111111}$=00000000。

### 3. “异或”运算

“异或”逻辑关系是在“与”、“或”、“非”3种基本逻辑运算基础上的变换。其逻辑代数表达式为

$$Y = \overline{A} \cdot B + A \cdot \overline{B} \tag{3.8}$$

“异或”运算是对两个变量的逻辑运算，用符号⊕表示：

$$Y = A \oplus B$$

**【例 3-8】** 计算 11010011B⊕10100110B。

**解：**

$$\begin{array}{r} 11010011 \\ \oplus 10100110 \\ \hline 01110101 \end{array}$$

即 11010011B⊕10100110B=01110101B。

二进制数的“异或”运算可以看做不进位的“按位加”，或者不借位的“按位减”。

#### 4. “同或”运算

“同或”运算是在“异或”运算的基础上再进行“非”运算的结果，所以，其运算规则可以直接表示为式(3.9)的函数表达式：

$$Y = \overline{A \oplus B} \tag{3.9}$$

**【例 3-9】** 设 $A$=11010011，$B$=10100110，计算 $Y=\overline{A \oplus B}$。

**解：** 按照“同或”逻辑关系，先对两数求“异或”，再做“非”运算，即对“异或”运算后的结果再按位取反，就得到两数相“同或”的结果：

$$\overline{11010011B \oplus 10100110B} = 100010101B$$

## 3.2 逻 辑 电 路

实现各种逻辑运算的电路称为逻辑门。本节介绍几种常用的逻辑门，以及由它们组合构成的计算机中的一些基本逻辑电路。作为入门级教材，本书将不涉及逻辑电路的内部结构，仅从应用的角度出发介绍它们的逻辑功能和符号表示。

### 3.2.1 基本逻辑门

实现上述各种基本逻辑运算的电路称为基本逻辑门电路。逻辑门是由若干半导体元件经过一定的组合，能够实现某一种逻辑关系的集成电路。物理上的一片集成电路芯片上，通常会集成若干个具有同样逻辑关系的逻辑门，例如将 4 个“与”门集成在一片芯片上构成的 4“与”门电路。

#### 1. “与”门(AND gate)

“与”门是对多个逻辑变量(取值范围只有 0 和 1)进行“与”运算的门电路，可以有多位输入，但只有 1 位输出。图 3-1 所示是两输入“与”门的两种逻辑符号。图中，$A$ 和 $B$ 为“与”门的输入，$Y$ 是“与”门的输出。**当且仅当输入全为 1 时，输出为 1；否则输出为 0。**

“与”门输入和输出之间的关系可以表示为式(3.2)，也可用表 3-3 来表示(该表也称

为“与”逻辑的真值表)。若从电平变化的角度描述,则仅当“与”门的输入 $A$ 和 $B$ 都是高电平时,输出 $Y$ 才是高电平,否则 $Y$ 就输出低电平。

**表 3-3 “与”门真值表**

| *A* | *B* | *Y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

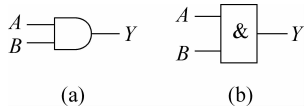



图 3-1 “与”门的逻辑符号

对图 3-1,有两点需要注意:

(1) 图中仅画出了 2 位输入($A$ 和 $B$),实际的“与”门电路可以有多位输入。

(2) 图中给出了“与”门的两种表示方法,其中,(a)为 IEEE 推荐符号,(b)为中国国家标准规定使用的符号,这两种图符目前均可以使用(以下类同)。为描述上的方便,本书后面以图中的(b)为主。

### 2. “或”门(OR gate)

“或”门是对多个逻辑变量进行“或”运算的门电路。和“与”门一样,也是多输入、单输出的门电路。其逻辑符号如图 3-2 所示。图中,$A$、$B$ 为“或”门的输入,$Y$ 是“或”门的输出。**当且仅当输入全为 0 时,输出为 0;输入有一位是 1,输出则为 1**。或者说:输入 $A$ 和 $B$ 只要有一个是高电平,输出 $Y$ 就为高电平;否则 $Y$ 输出低电平。

两输入“或”门可用图 3-2 所示的两种图符之一来表示,其真值表见表 3-4。和“与”门类似,本书中也将更多使用(b)所示的图符。

**表 3-4 “或”门真值表**

| *A* | *B* | *Y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

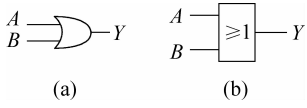



图 3-2 “或”门的逻辑符号

### 3. “非”门(NOT gate)

“非”门又称为反相器,是对单一逻辑变量进行“非”运算的门电路(如图 3-3 所示),其输入变量 $A$ 与输出变量 $Y$ 之间的关系可用以下函数式表示:

$$Y = \overline{A} \tag{3.10}$$

“非”运算也称求反运算,变量 $A$ 上的上划线 $\overline{\ }$ 在数字电路中表示反相之意。表 3-5 为“非”门的逻辑关系真值表。

计算机是由成千上万各种各样的逻辑门电路经过组合构成的,可以说,逻辑门是构成计算机的最小“细胞”单位,但任何复杂的逻辑门都是在基本逻辑门基础上的组合、变换。

因此，基本逻辑门及其逻辑关系是学习计算机科学非常重要的基础。

表 3-5 “非”门真值表

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

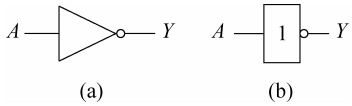



图 3-3 “非”门的逻辑符号

## 3.2.2 其他常用逻辑门

由基本逻辑门可以组合变换出其他各种逻辑电路。这些逻辑电路中，最常见也是最基本的有“与非”门、“或非”门、“异或”门和“同或”门等。

### 1. “与非”门(NAND gate)

将“与”门的输出连接到“非”门的输入，就构成了“与非”门。式(3.6)表示的逻辑函数可以表示为图 3-4 所示的两输入“与非”门，图中的小圆圈表示“非”(本书将始终采用(b)图表示方法)。表 3-6 是“与非”门的真值表。

表 3-6 “与非”门真值表

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

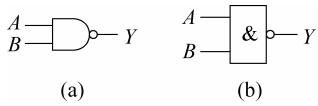



图 3-4 “与非”门的逻辑符号

### 2. “或非”门(NOR gate)

“或非”门是“或”门和“非”门的组合，即将“或”门的输出连接到“非”门的输入。式 3.7 表示的两变量逻辑函数对应的逻辑门如图 3-5 所示，真值表见表 3-7。

表 3-7 “或非”门真值表

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

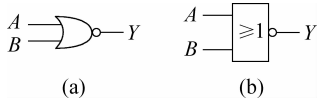



图 3-5 “或非”门的逻辑符号

### 3. “异或”门(XOR gate)

“异或”门是对两个逻辑变量(请注意这里的表述)进行“异或”运算的门电路，它有 2 位输入，1 位输出，其逻辑符号如图 3-6 所示。“异或”门输出对输入的关系可以简单地表

述为：输入相同则为 0，输入相异则为 1。即，当两输入状态相同时，输出结果为 0；两输入状态不同时，输出结果为 1。

表 3-8 “异或”门真值表

| *A* | *B* | *Y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

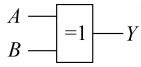



图 3-6 “异或”门的逻辑符号

### 4. “同或”门（XNOR gate）

“同或”门也称“异或非”门，即相当于将“异或”门的输出再连接到“非”门的输入的逻辑门电路，同样有 2 位输入，1 位输出。

“同或”门的逻辑图符如图 3-7 所示，其逻辑关系见表 3-9。“同或”门输出与输入之间的逻辑关系也可以简单表述为：输入相同则为 1，输入相异则为 0。即，当两输入状态相同时，输出结果为 1；两输入状态不同时，输出结果为 0。

表 3-9 “同或”门真值表

| *A* | *B* | *Y* |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

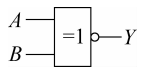



图 3-7 “同或”门的逻辑符号

上述 4 种门电路都是由基本逻辑门构造而成的。在实际数字电路设计中，“与非”门、“或非”门和“异或”门的使用非常广泛，“同或”门则相对使用得较少。

## 3.2.3 触发器

触发器（trigger，flip-flop），也称为双稳态多谐振荡器（bistable multivibrator），是一种可以使其输出端在高电平和低电平两种状态间相互翻转的逻辑电路。即，可以在外部触发信号的作用下，使输出由高电平变为低电平，或由低电平变换为高电平。而当外部触发信号不出现时，只要保持供电，其输出端的状态就可以稳定地保持不变。

触发器的这一特性，使其可以用于存储二进制数 0 和 1，也就是说，它是具有记忆功能的逻辑组件。

### 1. 触发器电路构造

触发器由各种逻辑门构成，同时，它也是计算机中更复杂电路的基本组成部件。在图 3-8 所示的电路中，两个“与非”门 $G_1$、$G_2$ 的输入端和输出端分别交叉连接在一起，这就

构成了一种新的功能器件，称为 RS 触发器。

RS 触发器是最基本的一种触发器，有两个输入端 $R$、$S$ 和两个输出端 $Q$、$\overline{Q}$。由图 3-8 可以看出，当 $S=0$，$R=1$ 时，$Q$ 端输出高电平，$\overline{Q}$ 端输出低电平；而当 $S=1$，$R=0$ 时，$\overline{Q}$ 端输出高电平，$Q$ 端输出低电平。即，**当在触发器的两个输入端加上不同逻辑电平时，其输出端 $Q$ 和 $\overline{Q}$ 存在两种互补的稳定状态**。所以，RS 触发器属于双稳态电路。

一般情况下，$Q$ 端的状态被视为触发器的输出状态。因为 $S=0$ 会直接使 $Q=1$，所以称 $S$ 端为置 1 端（或置位端）；当 $R=0$，$S=1$ 时，$Q=0$。所以，$R$ 端也称为触发器的复位端。

当 $R=S=1$ 时，“与非”门的输出不受影响，触发器状态保持不变；当需要触发器翻转时，要求在某一个输入端加上负脉冲。$R$ 和 $S$ 端不允许同时为低电平。

RS 触发器的输入输出逻辑关系如表 3-10 所示。

**表 3-10　RS 触发器逻辑真值表**

| $R$ | $S$ | $Q$ | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | — | — |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | × | × |

虽然 RS 触发器是由两个“与非”门组成的，但它具有独立的逻辑功能，是一种可以独立存在的逻辑器件。因此，RS 触发器可以抽象为图 3-9 所示的逻辑符号（图中，$S$ 和 $R$ 端的小圆圈表示是低电平有效）。

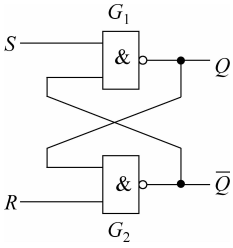



图 3-8　RS 触发器逻辑电路

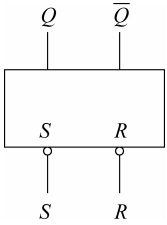



图 3-9　RS 触发器逻辑符号

图 3-10 所示的是另一种常见的触发器的基本原理图（没有考虑置位端和复位端），称为 D 触发器。它是在 RS 触发器的基础上又加入了两个“与非”门。其中，$D$ 为输入端，CP 为控制端。由图可知，当 CP=0 时，输入端 $D$ 被封锁（**当与门或者与非门的输入端有一位为低电平时，其输出状态就被确定，此时其他输入端的状态将对输出不再产生影响，故称为封锁**），输出保持不变；当 CP=1 时，输出 $Q$ 将会随着 $D$ 端状态的变换而变换。

作为一种有独立功能的逻辑器件，D 触发器也同样可以抽象为图 3-11 所示的逻辑符号。

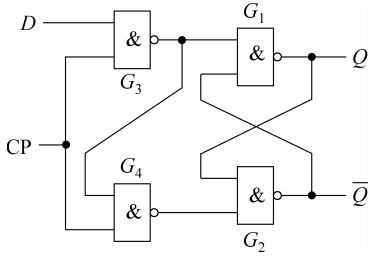



图 3-10　D 触发器基本原理图

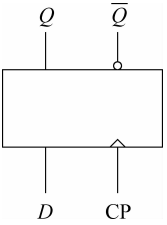



图 3-11　D 触发器逻辑符号

**思考** 读者是否可尝试自己总结一下 D 触发器的逻辑真值表？

## 2. 由触发器引起的思考

触发器是具有记忆功能的器件(也说它具有对信息的保持能力、保存能力或锁存能力)。一个触发器能够存储 1 位二进制码，$N$ 个触发器就可以存储 $N$ 位二进制数(还记得一个内存单元是多少位吗？可以想一下至少需要多少个触发器)。触发器的这种特性，使它成为构成计算机存储装置(寄存器、存储器等)的基本单元。

从基本逻辑门到触发器，从简单的 RS 触发器到稍微复杂一点的 D 触发器，从基本的 D 触发器再到图 3-12 所示的考虑了置位和复位信号的 D 触发器。可以看出，虽然电路的复杂度在逐步加深，但从本质上，它们都是基本逻辑门的组合。每用若干个基本逻辑门组成一个具有独立功能的部件后，这个部件就可以**封装**为一个整体，用一个逻辑符号来表示(如上述的 RS 触发器和 D 触发器)。此时，就可以不需要再了解它们内部的构造细节，而只要了解它们的整体功能和输入输出间的逻辑关系就可以了。

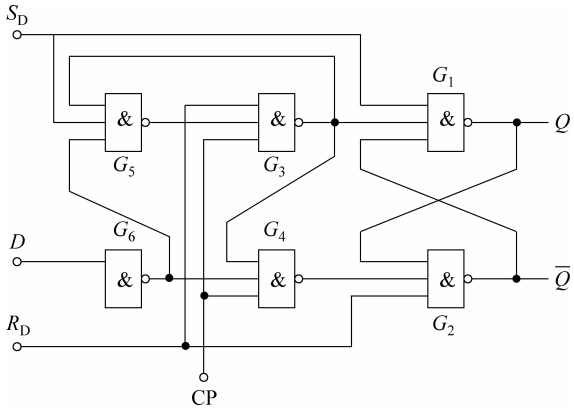



图 3-12 带置位和复位信号的 D 触发器

进一步，用触发器可以构造出计算机中的寄存器或存储器。图 3-13 是由 4 个 D 触发器构成的 4 位移位寄存器。这里，D 触发器成为寄存器的基础构件。如果将寄存器作为一个整体考虑(事实上，寄存器本身在逻辑上就是一个独立部件)，那就可以不再关心它内部有几个触发器以及它们之间的连接关系，而只需要了解寄存器本身的功能和工作原理。

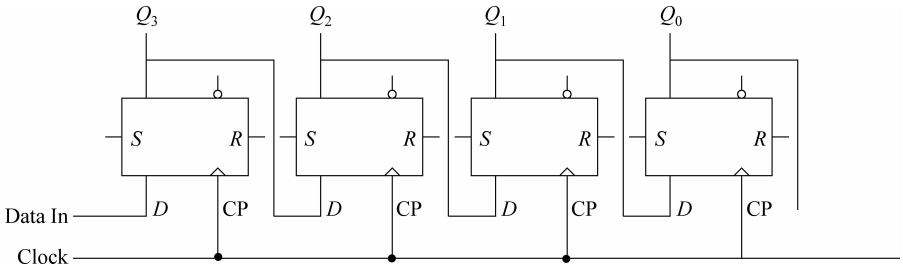



图 3-13 4 位移位寄存器

更进一步，寄存器、存储器这样的存储装置都是计算机硬件系统中的部件之一，由第1章的描述已知，硬件系统中除了存储装置，还有控制逻辑部件、运算器、I/O接口等。如果将计算机作为一个整体（计算机也的确就是一个独立、完整的机器），那么它内部各种部件之间的连接关系也就可以被封装了。

由此可以看出，计算机硬件系统实际上就是以基本逻辑门（当然还要加上一些其他辅助电路）为基础，经过一层一层地组合、封装而构成。整个系统可以分为若干层次，每一层都包含若干低一层的器件（或说由若干低层器件构成），并且用一个“符号”[①]实现对低一层的封装。这个过程称为硬件结构上的**抽象**。可以用图3-14来示意[②]。事实上，计算机硬件系统就是由若干基本逻辑部件经过一层一层的抽象构成的。

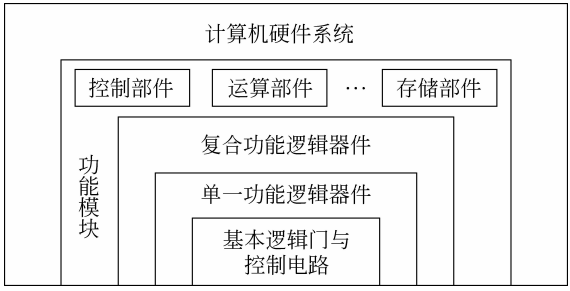



图3-14　硬件系统的分层抽象示意图

## 3.2.4　加法器

根据逻辑功能的不同特点，数字电路通常可以分为组合逻辑电路和时序逻辑电路两大类。**组合逻辑电路**在逻辑功能上的特点是任意时刻的输出仅仅取决于该时刻的输入，与电路原来的状态无关。如各种门电路等。而**时序逻辑电路**是指电路任何时刻的稳态输出不仅取决于当前的输入，还与前一时刻输入形成的状态有关，即具有记忆功能。触发器就属于时序逻辑电路，而加法器则属于组合逻辑电路。

计算机能够实现二进制的算术运算，算术运算在计算机中是由CPU的运算器完成的，而运算器的核心部件是算术逻辑单元（ALU）。由第2章的描述已知，减法运算可以通过引入补码而转换为加法运算；乘法运算相当于加法和移位操作；除法是乘法的逆运算，相当于减法和移位操作，而减法又可以转换为加法运算。因此，计算机中的算术运算主要是利用加法来实现的。

加法器（adder）是一种用于执行加法运算的数字电路，是构成CPU中算术逻辑单元的基础。加法器又分为半加器和全加器。

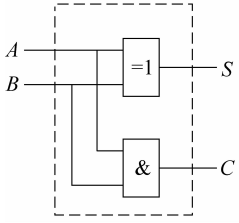
---

① 这里的“符号”是一个抽象含义，它既可以是一个具体的逻辑符号（如D触发器），也可以表示一种概念，如文中提到“运算部件”，只是一类功能部件的总称，而没有一个具体的符号与它对应。

② 图3-14仅为说明硬件系统的分层“抽象”而设计的示意图，并未包含各种辅助控制电路，所以并不是严谨的硬件系统结构图。

半加器的功能是将两个 1 位二进制数相加。它具有两个输入(加数、被加数)和两个输出(和、进位)。输出的进位信号代表了输入两个数相加后向高位的进位值。半加器是不考虑来自低位进位的加法器,图 3-15 给出了半加器的逻辑电路和真值表。它由一个“异或”门和一个“与”门组成,$A$、$B$ 为输入,$S$ 为输出的两数之和(sum),$C$ 是进位(carry)。由图知:

$$S=A\oplus B,\quad C=A\wedge B$$




| 输入 | | 输出 | |
|---|---|---|---|
| $A$ | $B$ | $S$ | $C$ |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

图 3-15　半加器逻辑电路及真值表

如果在半加器中添加一个“或”门来接收低位的进位输出信号,则两个半加器就构成一个全加器,如图 3-16 所示。

全加器是要考虑进位的加法器。它将两个 1 位二进制数相加,并根据接收到的低位进位信号,输出相加的和以及向高位的进位。按照分层抽象的方法,图 3-16 所示的逻辑电路可以用图 3-17 所示的逻辑符号表示。其中:$A$ 和 $B$ 是分别加数、被加数,$C_{in}$ 是来自低位的进位信号,$C_{out}$ 是向高位输出的进位信号,$S$ 是相加的和。

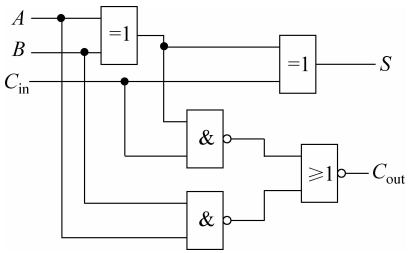
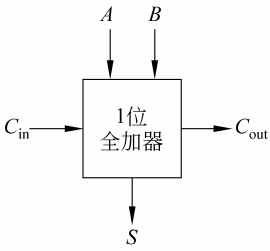



图 3-16　1 位全加器逻辑电路图

全加器只能实现 1 位二进制的加法,因为 CPU 的字长从早期的 8 位、16 位到现在的 32 位或 64 位,都不是只完成 1 位二进制运算,而需要同时进行多位二进制数的运算。因此,需要在全加器的基础上构造多位加法器。




| 输入 | | | 输出 | |
|---|---|---|---|---|
| $A$ | $B$ | $C_{in}$ | $S$ | $C_{out}$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

图 3-17　1 位全加器逻辑符号及真值表

多位加法器主要有涟波进位加法器(ripple-carry adder)和超前进位加法器(carry-lookahead adder)两种。

涟波进位加法器的实现方法比较简单，它用 $N$ 个全加器构成，其中对应低位的全加器将其进位输出信号 $C_{out}$ 连接到高一位全加器的进位输入端 $C_{in}$，并依次像“涟波”一样将进位信号向前传递。图 3-18 所示为 4 个 1 位全加器构成的 4 位涟波进位加法器。

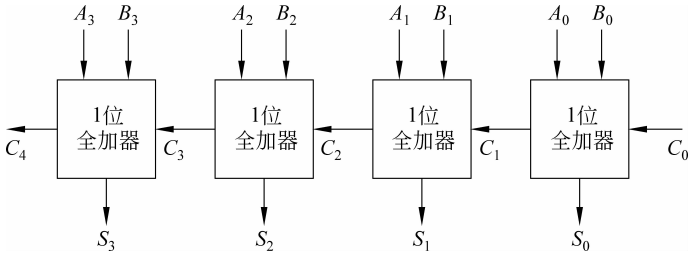



图 3-18　4 位涟波进位加法器

进位信号向前传递会产生传递延迟，最低位向前的进位需要顺序通过所有全加器才能产生最终的结果，这是这种加法器的主要缺点。

目前普遍使用的并行加法器是超前进位加法器。因篇幅原因，不再做进一步描述。

在这里介绍加法器，并非为了使读者学习加法器的设计（这样粗浅的介绍远达不到设计的可能），主要的目的有两个：一是帮助读者了解计算机进行加法运算的基本原理，更重要的是希望读者能通过本节由基本逻辑门到一些常见逻辑电路再到加法器（也可以说是计算机核心部件之一）的逐层抽象，初步建立硬件系统的构造思维模式。

# 3.3　冯·诺依曼结构

计算机历经半个多世纪的发展，在运算速度、存储能力及整体功能上都有了巨大的进步，根据摩尔定律[①]，计算机的整体性能每两年就会翻一番，事实也的确如此。今天，一台普通个人计算机的性能已远远超过 50 年前的巨型计算机。虽然发生了如此大的进步，但如今微型计算机的体系结构和工作原理依然沿用着冯·诺依曼 50 多年前的设计思想。

## 3.3.1　程序和指令

现代计算机不仅能够进行各种复杂的数值计算，还能够模拟人类的思维分析和处理各种事物。那么，计算机为什么能够做这么多的事情？它是如何完成每一项任务的？

计算机之所以能够按照要求完成一项一项的工作，是因为人向它发出了一系列的命令，这些命令通过输入设备以一定的方式送入计算机，并且能够为计算机所识别。这种能够被计算机识别的命令称为指令，一台计算机能够识别的所有指令的集合称为该计算机的指令系统，而保证对指令的这种执行能力的是计算机的硬件系统。

当人们需要计算机完成某项任务的时候，首先要将任务分解为若干个基本操作的集

① 当价格不变时，集成电路上可容纳的晶体管数目约每隔 18 个月便会增加一倍，性能也将提升一倍。

合，并将每一种操作转换为相应的指令，按一定的顺序组织起来，这就是程序。计算机完成的任何任务都是通过执行程序完成的。例如，在需要解一道数学题时，要先把题目的解算步骤按照一定的顺序用计算机能够识别的指令书写出来，命令计算机执行规定的操作。这些指令的序列就组成了程序，如图 3-19 所示。

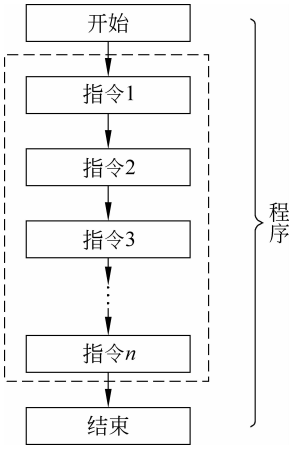


图 3-19　程序中的指令

计算机硬件能够直接识别并执行的指令称为机器指令。它们全部由 0 和 1 这样的二进制编码组成，其操作通过硬件逻辑电路实现。

不同的计算机系统通常都具有自己特有的指令系统，其指令在格式上也会有一些区别，但一般都包含这样 3 种信息，即完成何种操作（操作性质，如加、减、乘、除等）、对谁操作（操作的对象）以及操作结果的存放处。表征指令操作性质或者功能的称为操作码，表征操作对象的称为操作数（或地址码[①]）。指令的一般格式如图 3-20 所示。

| 操作码(OPC) | 操作数的目标地址，操作数的源地址 |
|---|---|

图 3-20　指令的一般格式

每台计算机都拥有由各种类型的机器指令组成的指令系统。指令系统的功能是否强大，指令类型是否丰富，决定了计算机的能力，也影响着计算机的结构。指令的不同组合方式可以构成完成不同任务的程序。计算机严格按照程序安排的指令顺序，有条不紊地执行规定的操作，完成预定任务。因此，程序是实现既定任务的指令序列，其中的每条指令都表示计算机执行的一项基本操作。一台计算机的指令种类是有限的，但通过人们的精心设计，可编写出无限多个实现各种任务处理的程序。

## 3.3.2　冯·诺依曼计算机基本结构

冯·诺依曼计算机的典型结构模型如图 3-21 所示，具有一个存储器、一个控制器、一个运算器以及输入和输出设备。所有的输入和输出都需要通过运算器。人们将这种结构称为冯·诺依曼结构，也称普林斯顿结构（Princeton architecture）。

冯·诺依曼结构的核心设计思想主要体现在：存储程序控制原理，以运算器为核心，采用二进制。可进一步描述为：

（1）将计算过程描述为由许多条指令按一定顺序组成的程序，并放入存储器保存。

（2）程序中的指令和数据都采用二进制编码（抛弃了十进制记数的设计思路），且能够被执行该程序的计算机所识别。

（3）指令和数据可一起存放在存储器中，并作同样处理。

（4）指令按其在存储器中存放的顺序执行，存储器的字长固定并按顺序线性编址。

① 表示运算数据及运算结果的存放处。

(5) 由控制器控制整个程序和数据的存取以及程序的执行。

(6) 计算机由运算器、逻辑控制装置、存储器、输入设备和输出设备 5 个部分组成，以运算器为核心，所有的执行都经过运算器。

冯·诺依曼计算机的设计思想简化了计算机的结构，大大提高了计算机的工作速度。图 3-21 是冯·诺依曼计算机的结构示意图。

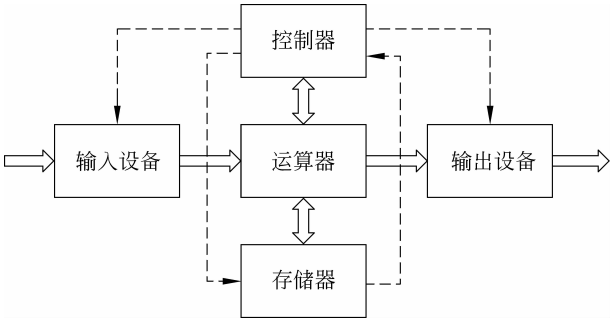



图 3-21　冯·诺依曼计算机结构示意图

半个多世纪过去了，虽然计算机软硬件技术都有了飞速的发展，但直至今天，微型计算机的基本结构形式并没有明显的突破，仍属于冯·诺依曼结构。计算机的基本工作原理仍然是存储程序控制原理。当然，二进制也依然是计算机硬件唯一能够直接识别的数制。

# 3.4　冯·诺依曼计算机基本原理

计算机的工作过程就是执行程序的过程，而程序是指令的序列。所以，计算机的工作过程就是一条条执行指令的过程。

## 3.4.1　指令的执行过程

由 3.3.1 节可知，指令是控制计算机完成某种操作并能够被计算机硬件所识别的命令。因此，指令才有如图 3-20 所示的格式(即包含指令码和操作数)。根据冯·诺依曼结构原理，程序在被执行前先要存放在(内)存储器中(为什么？学完 3.6 节就应该清楚了)，而程序的执行需要由 CPU 完成。因此，计算机在执行程序时，首先需要按某种顺序将指令从内存储器中取出(一次读取一条指令)并送入处理器，处理器分析指令要完成的动作，明确其操作性质和操作的对象，再去存储器中读取相应的操作数(如果需要)，然后执行相应的操作，最后将运算结果存放到内存储器中(如果这个结果不需要送到内存，当然就可以不送了)。这一过程直到遇到结束程序运行的指令才停止。

因此，指令的执行过程可简单地描述为 5 个基本步骤：取指令、分析指令、读取操作数、执行指令和存放结果。图 3-22 给出了一条指令的执行流程。

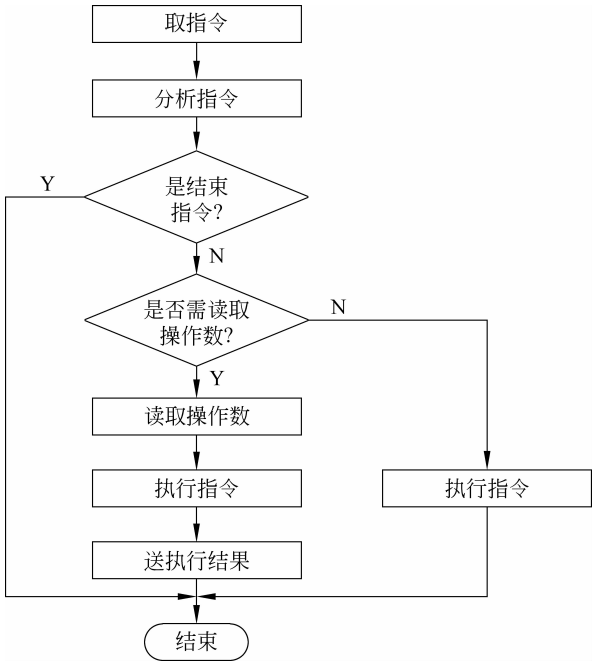



图 3-22　指令的执行过程

图 3-22 中的“是否需读取操作数”的分支，表示不是每一条指令都需要到内存中去读取操作数。当然，这不表示指令没有操作的对象，而是操作的对象可能是处理器本身。

以下暂且只讨论仅包括取指令、分析指令（也称指令译码）和执行指令这 3 个基本步骤时指令的执行方式。

在现代微处理器中，取指令、分析指令和执行指令的工作是由 3 个部件分别完成的。这 3 个部件可以同时工作（并行工作），也可以顺序方式工作（串行工作）。

### 1. 顺序工作方式

所谓顺序工作方式是指取指令、分析指令和执行 3 个部件依次工作，前一个部件工作结束后，下一个部件才开始工作。

指令顺序工作方式的工作过程如图 3-23 所示。在早期计算机系统中均采用这样的执行方式。

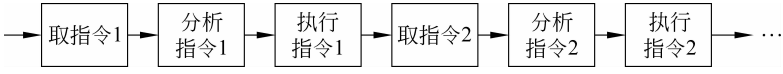



图 3-23　指令顺序执行方式示意图

顺序工作方式的优点是控制系统简单，实现比较容易；另外也节省硬件设备，使成本较低。缺点主要有两个，一是微处理器执行指令的速度比较慢，因为只有在上一条指令执行结束后，才能够执行下一条指令；二是处理器内部各个功能部件的利用率较低。如果以图 3-23 所示的流程工作，则在取指令部件从内存中读取指令时，分析指令和执行指令部

件都处于空闲状态；同样，在指令执行时也不能同时去取指令或分析指令。因此，顺序执行方式时系统总的效率是比较低的，各功能部件不能充分发挥作用。采用顺序方式执行 $n$ 条指令所用时间可用式(3.11)表示：

$$T_0 = \sum_{i=1}^{n}(t_{取指令i} + t_{分析指令i} + t_{执行指令i}) \tag{3.11}$$

假设计算机取指令、分析指令和执行指令所用的时间相等，均为 $\Delta t$，则完成一条指令的时间就是 $3\Delta t$，而执行完 $n$ 条指令需要的时间为

$$T_0 = 3n\Delta t \tag{3.12}$$

## 2. 并行工作方式

并行工作方式是使上述 3 个功能部件同时工作，即在指令被取入到处理器，开始进行分析的时候，取指令部件就可以去取下一条指令；而当指令分析结束开始被执行时，指令分析部件就可以进行下一条指令的译码工作，同时取指令部件又可以再去取新的指令……这样依次进行，在进入稳定状态后，就可以实现多条指令的并行处理。

图 3-24 给出了并行工作方式下的指令执行过程示意图。图中，当第 1 条指令进入指令分析部件时，取指令部件就开始从内存中取第 2 条指令，假如这 3 个功能部件的执行时间完全相等，均为 $\Delta t$，执行第 1 条指令需要的时间为 $3\Delta t$，之后每过一个 $\Delta t$ 时间，就有一条指令执行完成，则执行 $n$ 条指令所需要的时间为

$$T = 3\Delta t + (n-1)\Delta t = (2+n)\Delta t \tag{3.13}$$

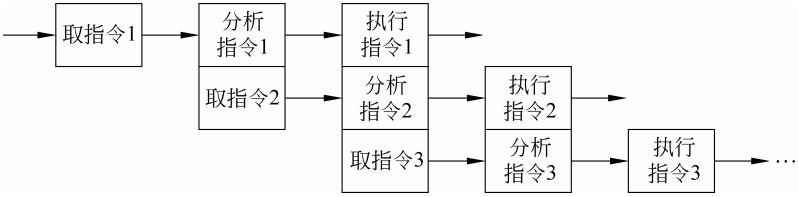



图 3-24　指令并行执行方式示意

由式(3.13)可以看出，与采用顺序执行方式所用的时间 $T_0$ 相比，并行执行方式缩短了系统执行程序的时间，且这种时间上的收益率会随着指令数量 $n$ 的增加而更加显著。

相对于顺序执行方式，并行方式减少的时间量可用系统加速比 $S$ 来描述：

$$S = T_0/T = 3n\Delta t/[(2+n)\Delta t] = 3n/(2+n) \tag{3.14}$$

**【例 3-10】** 某程序段经编译后生成 10 000 条机器指令，假设取指令、分析指令和执行指令所用的时间均为 $t$。分别求出使用顺序执行方式和并行流水线方式完成该程序段所需的时间，并说明使用顺序执行方式比并行方式慢多少(即系统的加速比)。

由题目知，$n$=10 000，$\Delta t = t$，则由式(3.2)，顺序执行完该程序所需时间为

$$T_0 = 3nt = 30\ 000t$$

采用并行流水方式执行该程序需要的时间为

$$T = (2+n)t = 10\ 002t$$

顺序执行与并行方式所耗费时间的比为

$$S = T_0/T = 30\ 000t/10\ 002t = 30\ 000/10\ 002 \approx 3$$

可见，顺序执行方式所花费的时间约为并行方式的 3 倍。

图 3-24 所示的模型是现代计算机流水线控制技术的基本模型。该模型所给出的是理想的情况，即每个部件的工作时间完全相同，也仅在这样的假设下，所示模型的流水线才不会“断流”。这在实际的系统中是不可能的。

为了解决流水线的断流问题，在现代计算机系统中，在取指令和指令译码部分，都设置有指令和数据缓冲栈，可以实现指令和数据的预取和缓存。指令执行部分设置有独立的定点算术逻辑运算部件、浮点运算部件等。另外，加入了预测、分析、多级指令流水线等多项技术，实现对指令和数据的预取和分析，以尽可能地保证流水线的连续。

## 3.4.2 微型计算机的一般工作过程

计算机的工作过程就是执行程序的过程，也就是逐条执行指令序列的过程。由于每一条指令的执行都包括取指令（含指令译码）和执行指令两个基本阶段，所以，微机的工作过程也就是不断地取指令和执行指令的过程。

当需要计算机完成某项任务时，最基本的工作是首先要使用某一种程序设计语言[①]编写出相应的程序。编写完成后，需要以文件形式（要起个名字）存放在外存储器中，运行时在操作系统控制下通过接口输入到内存。

CPU 是整个计算机的核心，所有程序的执行和控制都是由 CPU 完成的。为了说明微型机的工作过程，在图 1-5 所示的 CPU 基本结构的基础上，给出稍微详尽一点的 CPU 结构模型（如图 3-25 所示）[②]。在该图中，运算器部分中的核心部件就是 ALU。另一个需要关注的部件是程序计数器（Program Counter，PC），它是 CPU 控制程序走向（就是执行完一条指令后下边该执行哪条指令）的“指挥棒”。

进入内存后的程序，会按照逻辑上的顺序依次放入内存各单元。假设程序已存放到内存，当计算机要从停机状态进入运行状态时，处理器内部的程序计数器（PC）会指向程序的第一条指令。当 PC 所指向的指令被取出后，处理器将自行修改 PC 的值，使其指向下一条指令。指令的执行结果会暂存在内存中，最后在操作系统控制下存入外存或由输出设备送出。图 3-26 给出了程序在进入内存后、计算机按顺序执行方式执行一条指令的工作过程：

（1）控制器将要读取的指令在内存中的地址赋给 PC（图中假设为 04H），并送到地址寄存器 AR。

（2）PC 自动加 1，AR 的内容不变。

（3）将地址寄存器 AR 的内容发送到地址总线上，并送到内存储器，经地址译码器译码，选中相应的内存单元。

（4）CPU 的控制器发出“读”控制信号。

---

① 关于程序设计语言的相关介绍请参阅本书第 5 章。

② 本书引入该图的目的仅为下面描述的方便。有关 CPU 具体的工作原理请参阅其他硬件系统类书籍。