

# 计算机组成原理 期末复习

## 第 1 章 计算机系统概论

### 1.1 计算机的诞生和发展

#### 冯诺依曼体系

- (1) 计算机由运算器、控制器、存储器、输入设备和输出设备 5 部分组成。
- (2) 采用存储程序的方式, 程序和数据放在同一个存储器中, 并以二进制码表示。
- (3) 指令由操作码和地址码组成。
- (4) 指令在存储器中按执行顺序存放, 由指令计数器(即程序计数器 PC)指明要执行的指令所在的存储单元地址, 一般按顺序递增, 但可按运算结果或外界条件而改变。
- (5) 机器以运算器为中心, 输入输出设备与存储器间的数据传送都通过运算器。

### 1.2 计算机的硬件

#### 指令

一条指令通常分成两部分。

- (1) 操作码。规定该指令执行什么样的运算(或操作), 因此被命名为操作码。
- (2) 地址码。规定对哪些数据进行运算, 通常表示的是数据地址, 因此被称为地址码。

#### 硬件组成

组成计算机的基本部件有中央处理器(CPU, 运算器和控制器)、存储器和输入输出设备。

中央处理器又叫 CPU, 在早期的计算机中分成运算器和控制器两部分, 由于电路集成度的提高, 早已把它们集成在一个芯片中。

运算器是对信息或数据进行处理和运算的部件, 经常进行的是算术运算和逻辑运算, 所以在其内部有一个算术及逻辑运算部件(ALU)。算术运算是按照算术规则进行的运算, 例如加、减、乘、除、求绝对值、求负值等。逻辑运算一般是指非算术性质的运算, 例如数据、移位、逻辑乘、逻辑加和按位加等。在计算机中, 一些复杂的运算往往被分解成一系列算术运算和逻辑运算。

控制器主要用来实现计算机本身运行过程的自动化, 即实现程序的自动执行。在控制器控制之下, 从输入设备输入程序和数据, 并自动存放在存储器中, 然后由控制器指挥各部件(运算器、存储器等)协同工作以执行程序, 最后将结果打印(或以其他方式)输出。作为控制用的计算机则直接控制对象。

在计算机中, 各部件间来往的信号可分成 3 种类型, 即地址、数据和控制信号。通常这些信号是通过总线传送的, 如图 1.1 所示。CPU 发出的控制信号经控制总线送到存储器和输入输出设备, 控制这些部件完成指定的操作。与此同时, CPU(或其他设备)经地址总线向存储器或输入输出设备发送地址, 使得计算机各个部件中的数据能根据需要互相传送。输入输出设备和存储器有时也向 CPU 送回一些信号, CPU 可根据这些信号来调整本身发出的控制信号。现代计算机还允许输入输出设备直接向存储器提出读写要求, 控制数据传送。

计算机系统分类

1) 单指令流单数据流(SISD)计算机系统

通常由一个处理器和一个存储器组成。典型的 SISD 计算机每次执行一条指令,每次从存储器取(或存)一个数据,为了提高运算速度,有些 SISD 计算机设置了指令流水线和运算操作流水线,有些还设置了多个功能部件和多体交叉存储器。

2) 单指令流多数据流 (SIMD)计算机系统

通常由一个指令控制部件、多个处理器和多个存储器组成。各处理器和各存储器之间通过系统内部的互连逻辑电路进行通信。在程序运行时由指令控制部件向各个处理器传送同一条指令,处理器执行指令时所需的数据是从存储器中取的,各处理器所处理的数据是各不相同的,这就是多数据流。

3) 多指令流单数据流 (MISD) 计算机系统

MISD 计算机系统在同一时刻执行多条指令,但处理同一个数据。大多数人认为能列在这一系统中的计算机很少或根本不存在。

4) 多指令流多数据流 (MIMD) 计算机系统

典型的 MIMD 计算机系统由多台处理器(包括指令控制部件和处理器)和多个存储器组成,并有一个系统内部的互连逻辑电路实现各处理器和各存储器之间的通信。每台处理器执行各自的指令,存取各自的数据(各不相同)。

1.2 计算机的软件

机器语言 汇编语言 高级语言

早期,计算机的使用者必须用二进制码表示的指令编写程序(一般用八进制或十六进制书写),称为机器语言程序。在 20 世纪 50 年代,出现了符号式程序设计语言,称为汇编语言,程序员可用 ADD、SUB、MUL 和 DIV 等符号分别表示加法、减法、乘法、除法的操作码,并用符号来表示指令和数据的地址。汇编语言程序的大部分语句是和机器指令一一对应的。用户用汇编语言编写程序后,依靠计算机将它翻译成机器语言(二进制代码),然后再在计算机上运行。这个翻译过程是由系统程序员提供的汇编程序实现的。

翻译程序有编译程序和解释程序两种。

当前用户的应用程序一般是用高级语言(如 C++、FORTRAN 和 Java 等)编写的,是由英文字母、数字和运算符等按照一定的语法规则组成的。先将其翻译成操作系统支持的表达形式,再翻译成机器语言程序,才能在计算机硬件上运行,计算机系统的多级层次结构如图 1.2 所示。

图中的“中间件/平台级”在当前的计算机系统中还不是必须设置的,由于应用程序与操作系统之间的通信接口尚未标准化,造成了在不同计算机上编写的应用程序不能相互交换使用(即缺乏互操作性),期望经过专家讨论,能在操作系统之上形成较为标准的平台,降低上述弱点造成的影响。

当前计算机中使用的系统软件(高级语言的编译程序、中间件、操作系统等)需要翻译成机器语言后在用户的计算机中存放,以便于用户的使用。

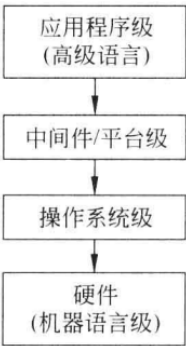


图 1.2 计算机系统的多级层次结构

操作系统合理地组织计算机的工作流程,管理和分配存储空间,控制和管理外部设备,并提供给用户使用计算机的良好界面,使用户不必了解硬件和软件的细节就可较方便地使用计算机。

第 2 章 计算机的逻辑部件

2.1 组合逻辑电路

三态电路

三态反相门的功能表及逻辑图示于图 2.1。它有一个三态控制端  $\bar{G}$ 。当  $\bar{G}=0$ ,反相门输出  $Y=\bar{A}$ ;当  $\bar{G}=1$ ,电路输出呈高阻,此时  $Y$  记作:  $Y=Z$ 。电路的输出表达式为

$$Y = \bar{G}Z + \bar{G}\bar{A}$$

还有一种三态电路,其三态控制端  $G=1$  时,电路处于正常态;当  $G=0$  时,  $Y=Z$ ,图 2.2 是它的功能表及逻辑图。

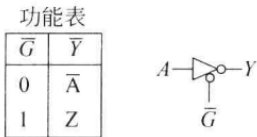


图 2.1 三态反相门(1)的功能表及逻辑图

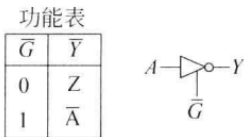


图 2.2 三态反相门(2)的功能表及逻辑图

若干个三态门共同驱动总线是最常见的应用。

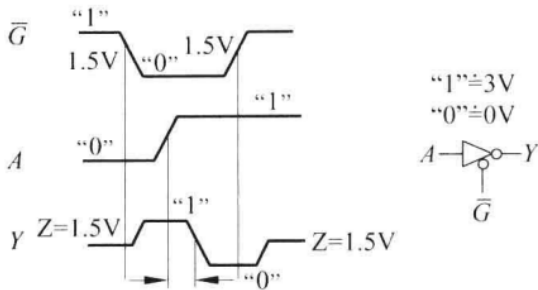


图 2.3 Z 态在时序图中的表示

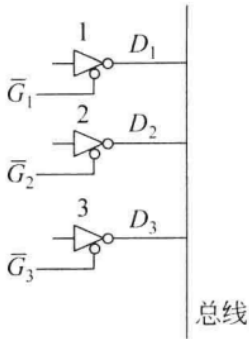


图 2.4 三态门应用实例

为可靠起见,三态电路由正常态转变为高阻态的过程总是快于高阻态向正常态的转变的(查阅电路参数手册可得),这样,即使  $\bar{G}_2$  的负跳变和  $\bar{G}_1$  的正跳变同时到来,也不能出现门 1 和门 2 同时处于正常态的情况。

异或门的应用

- (1) 原码/反码输出电路
- (2) 半加器
- (3) 数码比较器
- (4) 奇偶检测电路

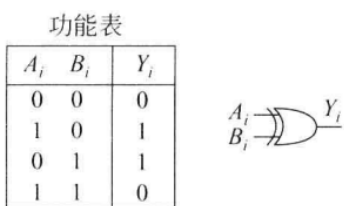


图 2.5 异或门的功能表和逻辑图

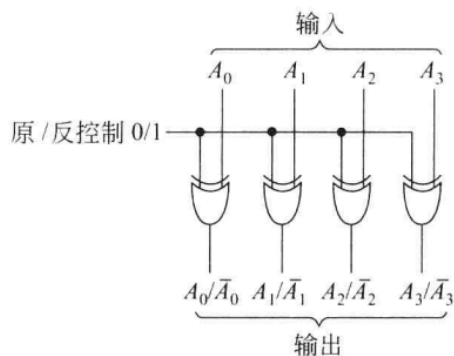


图 2.6 四位原/反码输出电路

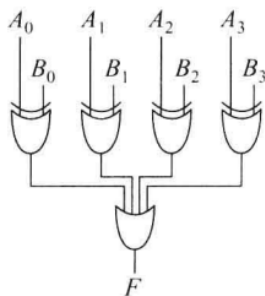


图 2.7 四位比较器的逻辑图

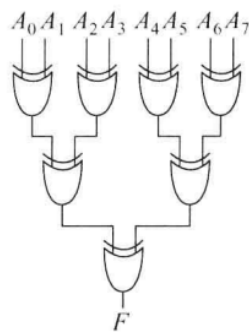


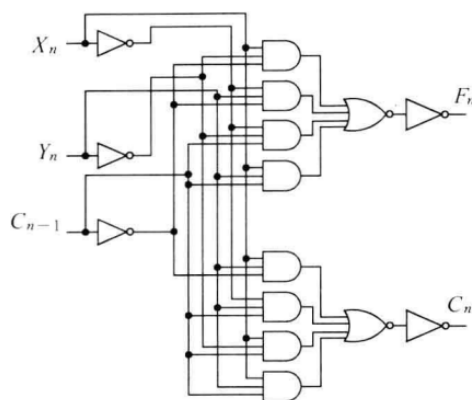
图 2.8 八位奇偶检测电路

## 加法器

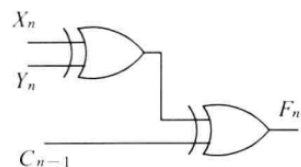
功能表

$X_n$	$Y_n$	$C_{n-1}$	$F_n$	$C_n$
0	0	0	0	0
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) 功能表



(b) 逻辑图



(c) 逻辑图

图 2.10 全加器的功能表及逻辑图

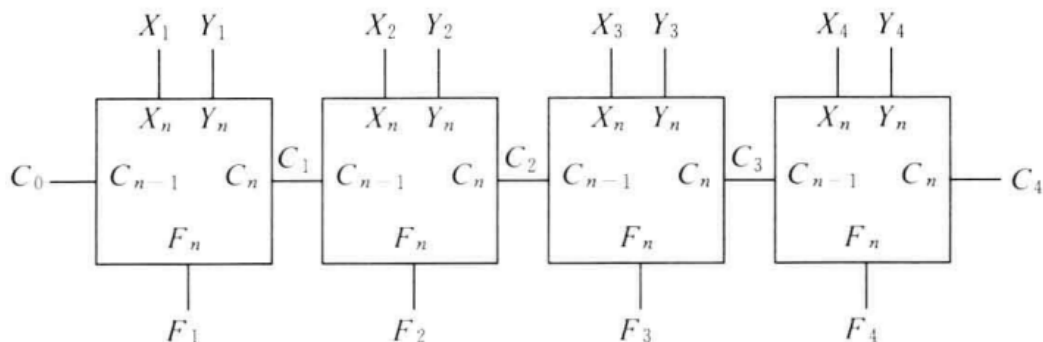


图 2.11 串行进位加法器



根据上述公式可画出“超前进位产生电路”及“四位超前进位加法器”的逻辑图,如图 2.12 所示。由图可以看到,只要  $X_1 \sim X_4, Y_1 \sim Y_4$  和  $C_0$  同时到来,就可几乎同时形成  $C_1 \sim C_4$ ,并且几乎同时形成  $F_1 \sim F_4$ 。

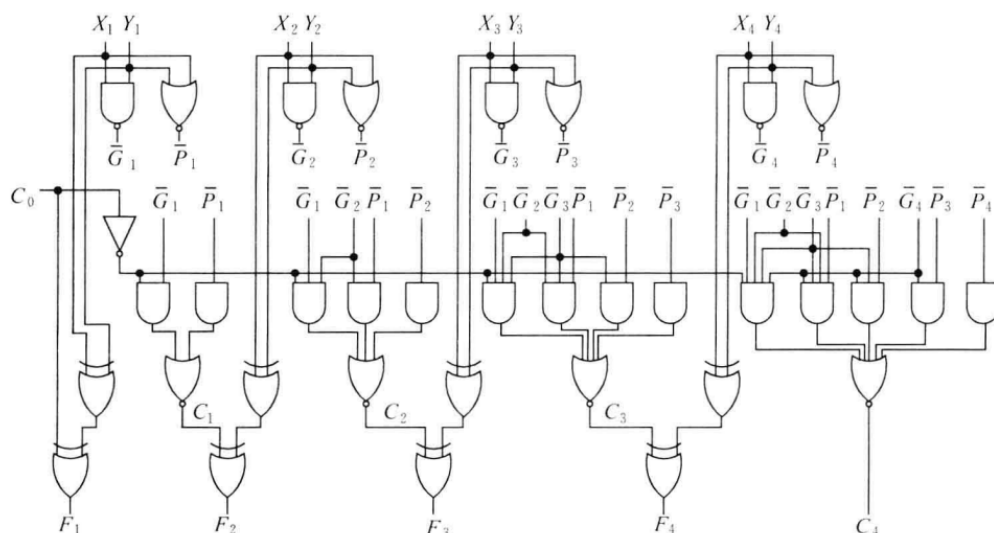


图 2.12 四位超前进位加法器

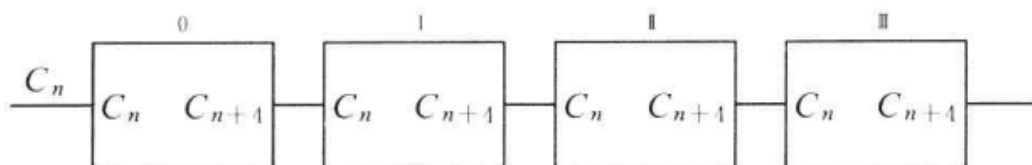


图 2.13 用 4 片 ALU 构成的 16 位 ALU

图 2.14 给出了用 4 片 ALU 和组间超前进位电路组成的 16 位快速 ALU。

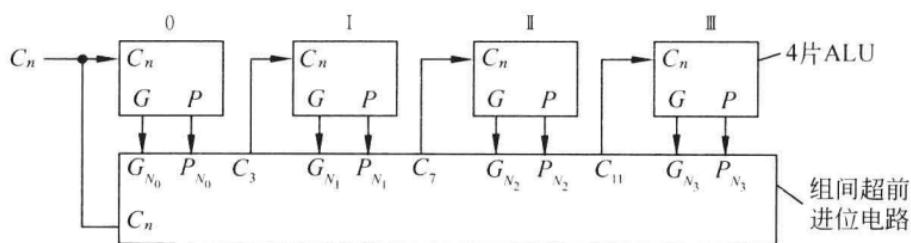


图 2.14 16 位快速 ALU

## 题：加法器

1. (6分) 串行加法器和并行加法器有何不同？影响加法运算速度的关键因素是什么？

答：1) 加法器通常指的是只用 1 位加法器，对数据的各位依次（从低位到高位）相加。(2 分)

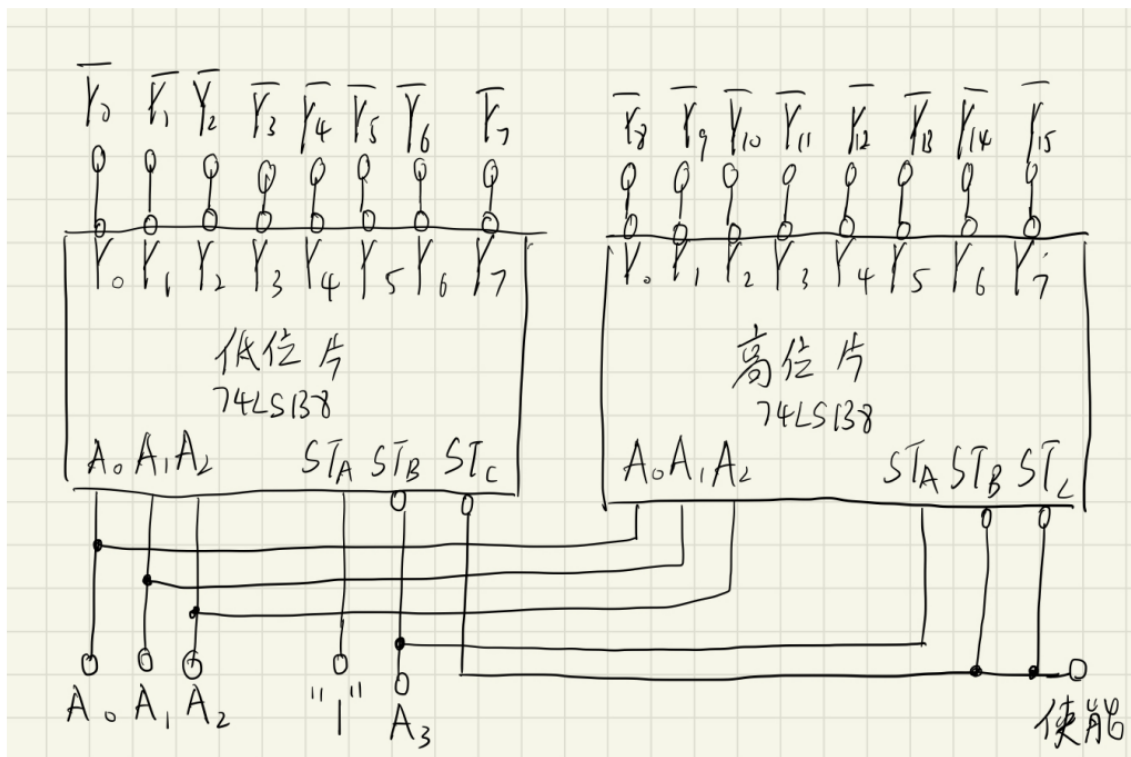
2) 并行加法器一般设置有  $n$  个全加器（ $n$  为数据字长），同时对  $n$  位数据进行相加。(2 分)

3) 低位产生的进位信号会影响高位的结果，在最不利的情况下（例如  $1111 \cdots 11 + 1$ ），从低位产生的进位信号逐位传递到最高位，这种串行进位的加法器主要因进位的延迟而影响加法运算速度。为了提高运算速度，一般采用并行进位（即超前进位）的方法。(2 分)

## 译码器

译码器有  $n$  个输入变量， $2^n$  个（或少于  $2^n$  个）输出。当输入为某一组合时，对应的仅有一个输出为 0（或为 1），其余输出均为 1（或为 0）。译码器的用途是把输入代码译成相应的控制电位，以实现代码所要求的操作。

2.5 画出逻辑图：用 2 个有 3 输入 8 输出译码器功能的芯片组成具有 16 输出的译码器。



## 数据选择器

数据选择器又称多路选择器或多路开关，从多个输入通道中选择某一个通道的数据作为输出。

功能表( $Y_1$ )

$S_1$	$S_0$	$D_3$	$D_2$	$D_1$	$D_0$	$\overline{G}$	$Y_1$
×	×	×	×	×	×	1	$Z$
1	1	$D_3$	×	×	×	0	$D_3$
1	0	×	$D_2$	×	×	0	$D_2$
0	1	×	×	$D_1$	×	0	$D_1$
0	0	×	×	×	$D_0$	0	$D_0$

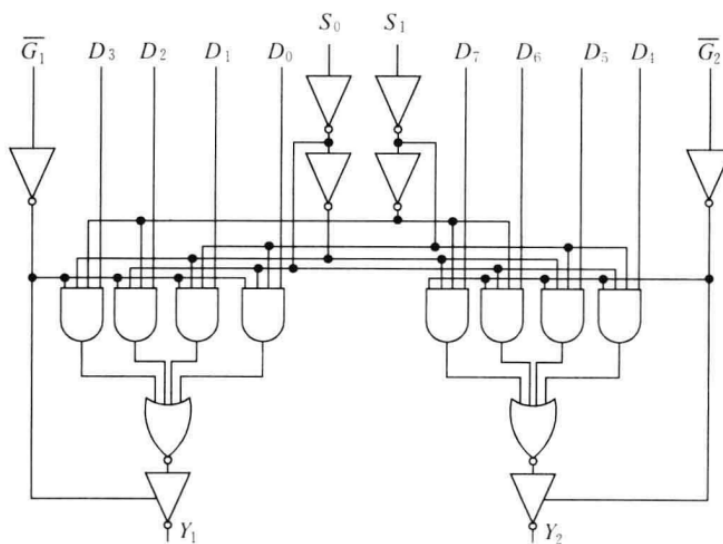


图 2.15 双 4 通道选 1 数据选择器

## 2.2 时序逻辑电路

### 触发器

电位触发器 锁存器 可以组成暂存器

边沿触发器 D 触发器 可以组成寄存器、计数器、移位寄存器

主从触发器 JK 触发器 可以组成计数器

寄存器 移位寄存器

寄存器是计算机的一个重要部件,用于暂存数据和指令等。它由触发器和一些控制门组成。在寄存器中,常用的是  $D$  触发器和锁存器。

图 2.19 所示是由正沿触发的  $D$  触发器组成的 4 位寄存器。在 CP 正沿作用下,外部数据才能进入寄存器。

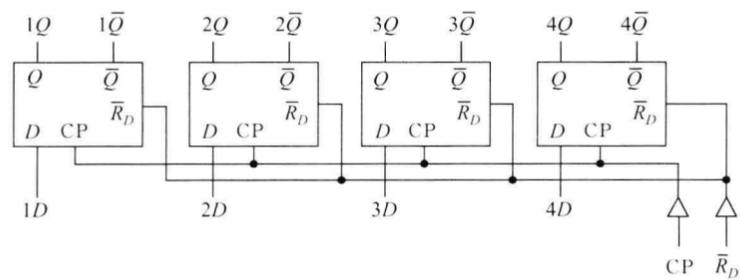


图 2.19 4D 寄存器

在计算机中常要求寄存器有移位功能。例如在进行乘法时,要求将部分积右移;在将并行数转换成串行数时 also 需移位。有移位功能的寄存器称为移位寄存器。此时应增加逻辑电路来控制触发器的输入数据(1D~4D)。

计数器

功能表

$P$	$T$	$L$	$\overline{R}_D$	CK	功 能
1	1	1	1	$\square$	计 数
$\times$	$\times$	0	1	$\square$	并行输入数据
0	1	1	1	$\times$	保 持
$\times$	0	1	1	$\times$	触发器保持, $RC=0$
$\times$	$\times$	$\times$	0	$\times$	异步清 0

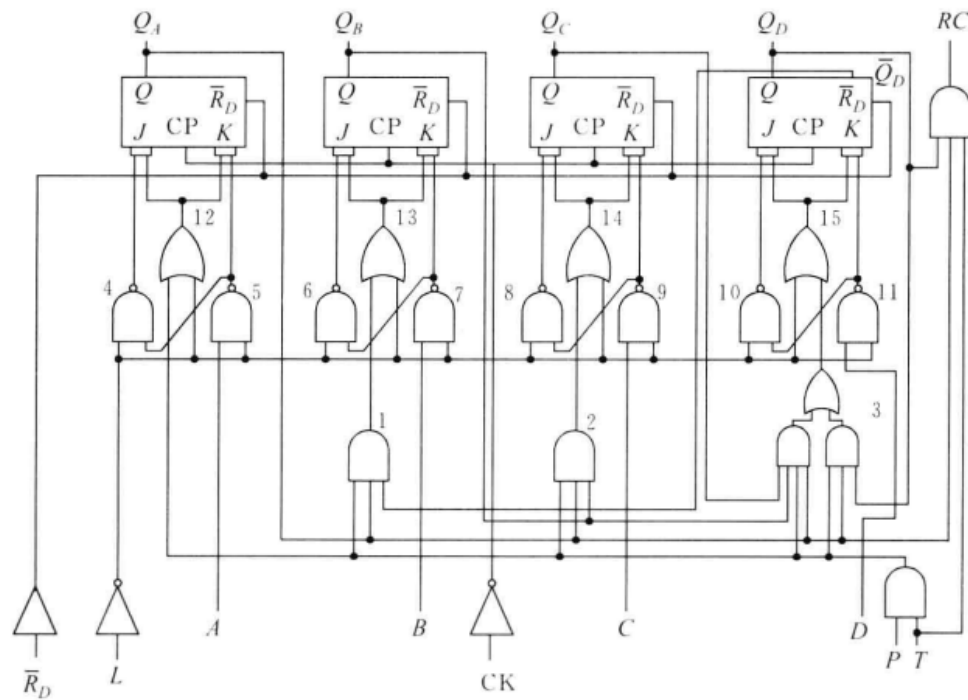


图 2.20 十进制同步计数器

## 2.3 阵列逻辑电路

### 只读存储器 ROM

用于存储固定信息，如监控程序、函数和常数

由地址译码器（与阵列）和存储单元（或阵列）构成

ROM 的工作原理如下：地址译码器根据输入地址选择某条输出（称字线），由它再去驱动该字线的各位线，以便读出字线上各存储单元所存储的代码。

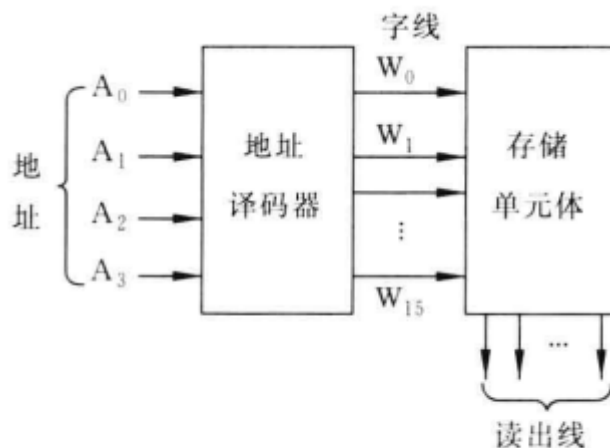


图 2.21 ROM 的结构

### 可编程序逻辑阵列 PLA

由与阵列、或阵列组成，都可编程

采用熔丝工艺

### 可编程序阵列逻辑 PAL

与阵列可编程，或阵列不可编程

采用熔丝工艺

### 通用阵列逻辑 GAL

通用阵列逻辑 (generic array logic, GAL) 器件是一种可用电擦除的，可重复编程的高速 PLD。可擦除重写 100 次以上，数据可保存 20 年以上，在数秒钟内即可完成擦除和编程过程。

GAL 的输出结构有一个输出逻辑宏单元 (OLMC)，通过对它的编程，使 GAL 有多种输出方式：寄存器型输出、组合逻辑输出，并可控制三态输出门，因此显得很灵活。

### 门阵列 GA 宏单元阵列 MCA

门阵列 (gate array, GA) 是一种逻辑功能很强的阵列逻辑电路。在芯片上制作了排成阵列形式的门电路，根据用户需要对门阵列中的门电路进行互连设计，再通过集成电路制作工艺来实现互连，以实现所需的逻辑功能。

宏单元阵列 (macro cell array, MCA) 是一种比 GA 功能更强、集成度更高的阵列电路，在芯片上排列成阵列的除门电路外，还有触发器、加法器、寄存器以及 ALU 等。

现场可编程门阵列 FPGA

现场可编程门阵列 (field programmable gate array, FPGA) 是一种集编程设计灵活性和宏单元阵列于一体的高密度电路。它与 GA 和 MCA 有区别, FPGA 内部按阵列分布的宏单元块都是用户可编程的。用户所需逻辑可在软件支持下, 由用户自己装入来实现, 而无须集成电路制造工厂介入, 并且这种装入是可以修改的, 因而其连接十分灵活。

第 3 章 运算方法和运算部件

3.1 数据的表示方法和转换

进制转换

N 转十: 位值原理

十转 N: 整数部分, 短除法 (除 N 取余); 小数部分, 乘 N 取整

十进制可以精确表示二进制, 二进制不能精确表示十进制

题: 进制转换

7. 将  $(0.73)_{10}$  转换成二进制码 (小数点后取 4 位有效值)。

答: (1) 用十进制转换。

结果	$0.73 \times 2$
高位 1	$.46 \times 2$
↓ 0	$.92 \times 2$
↓ 1	$.84 \times 2$
低位 1	$.68$

$(0.73)_{10} = (0.1011)_2$

编码

(1) BCD 编码

①概念

用 4 位二进制数来表示 1 位十进制中的数码(0~9)

②8421 码

a. 8421 码的映射关系

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

b. 8421 码的加法

例如：8 + 4 = (12)<sub>10</sub>，用 8421 码表示：

$1000 + 0100 = 1100$ （不在映射表中，则需“+6”进行修正） $\xrightarrow{+0110} 10010$

$$\begin{array}{r} 0001 \\ 0010 \\ \hline 1 \quad 2 \end{array}$$

注：若相加结果在合法范围内，则无需修正

2) 无权码

表示一个十进制数位的二进制码的每一位没有确定的权。用得较多的是余 3 码 (Excess-3 Code) 和格雷码 (Gray Code)，格雷码又称“循环码”。

余 3 码是在 8421 码的基础上，把每个编码都加上 0011 而形成的(如表 3.2 所示)。

当两个余 3 码相加不产生进位时，应从结果中减去 0011；产生进位时，应将进位信号送入高位，本位加 0011。

例 3.11  $(28)_{10} + (55)_{10} = (83)_{10}$

$$\begin{array}{r} 0101 \quad 1011 \quad (28)_{10} \\ +) 1000 \quad 11000 \quad (55)_{10} \\ \hline 1110 \quad 0011 \\ -) 0011 \quad +) 0011 \\ \hline 1011 \quad 0110 \end{array}$$

低位向高位产生进位，高位不产生进位。  
低位+3，高位-3。(0011)<sub>2</sub> = (3)<sub>10</sub>

格雷码的编码规则是：任何两个相邻编码只有 1 个二进制位不同，而其余 3 个二进制位相同。其优点是 从一个编码变到下一个相邻编码时，只有 1 位发生变化，用它构成计数器时可得到更好的译码波形(见第 6 章)。格雷码的编码方案有多种，表 3.2 给出两组常用的编码值。

表 3.2 4 位无权码

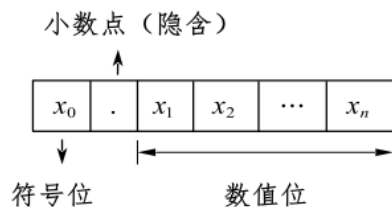
十进制数	余 3 码	格雷码(1)	格雷码(2)
0	0011	0000	0000
1	0100	0001	0100
2	0101	0011	0110
3	0110	0010	0010
4	0111	0110	1010
5	1000	1110	1011
6	1001	1010	0011
7	1010	1000	0001
8	1011	1100	1001
9	1100	0100	1000

## 3.2 数据表示和加减法

### 数据表示

#### 2) 定点小数

定点小数是纯小数（即小于1），约定小数点位置在符号位之后，有效数值部分最高位之前。



#### 3) 原、反、补、移码之间的转换

##### (1) 原码表示法

用机器数的最高位表示该数的符号（0：正；1：负），其余各位表示数的绝对值。

注：真值零的原码表示有2种： $[+0]_{\text{原}} = 0,000$  和  $[-0]_{\text{原}} = 1,000$

##### (2) 反码表示法

正数：反码与原码相同；

负数：原码符号位不变，数值部分全部取反（ $0 \rightarrow 1; 1 \rightarrow 0$ ）

注：真值零的反码表示有2种， $[+0]_{\text{反}} = 0,000$  和  $[-0]_{\text{反}} = 1,111$

##### (3) 补码表示法

正数：补码与原码相同；

负数：原码符号位不变，数值部分全部取反（ $0 \rightarrow 1; 1 \rightarrow 0$ ），末位加1（即“取反加1”）。

注：①此方法也可逆着用，即由  $[X]_{\text{补}}$  求  $[X]_{\text{原}}$

②真值零的补码表示只有1种， $[0]_{\text{补}} = 0,000$

##### (4) 移码表示法

将  $[X]_{\text{补}}$  的符号位取反即得到  $[X]_{\text{移}}$

注：①移码只能用来表示定点整数，（移码多用于表示浮点数的阶码）

②真值零的移码表示只有1种， $[0]_{\text{移}} = 1,000$

数值零的补码表示形式是唯一的，即：

$$[+0]_{\text{补}} = [-0]_{\text{补}} = 0.0000$$

这可根据补码定义计算如下：

当  $X = +0.0000$  时， $[X]_{\text{补}} = 0.0000$ 。

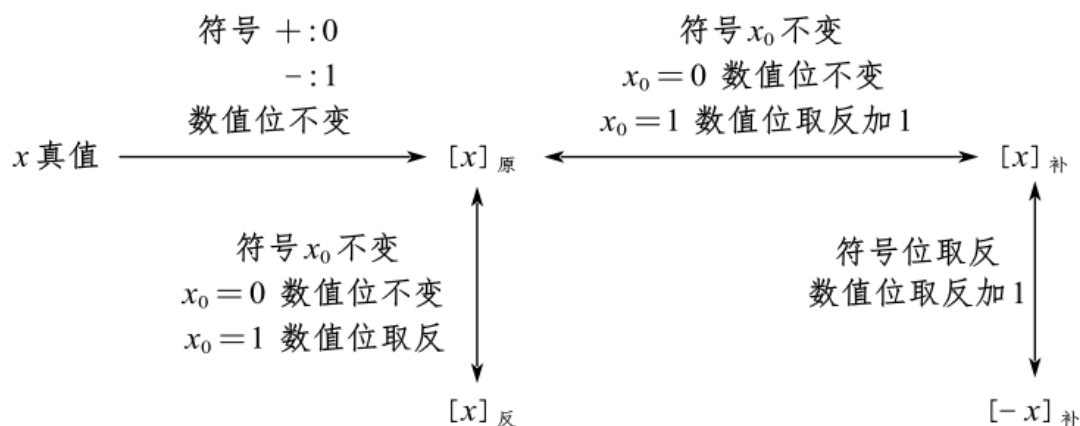
当  $X = -0.0000$  时， $[X]_{\text{补}} = 2 + X = 10.0000 + 0.0000 = 10.0000 = 0.0000 \pmod{2}$

当补码加法运算的结果不超出机器范围时，可得出以下重要结论：

(1) 用补码表示的两数进行加法运算，其结果仍为补码。

(2)  $[X+Y]_{\text{补}} = [X]_{\text{补}} + [Y]_{\text{补}}$ 。

(3) 符号位与数值位一样参与运算。



## 题：数据表示

26. 写出下列各二进制数的原码、补码和反码：

0.1010, 0, -0, -0.1010, 0.1111, -0.0100

答：

$X$	$[X]_{\text{原}}$	$[X]_{\text{补}}$	$[X]_{\text{反}}$
0.1010	0.1010	0.1010	0.1010
0	0.0000	0.0000	0.0000
-0	1.0000	0.0000	1.1111
-0.1010	1.1010	1.0110	1.0101
0.1111	0.1111	0.1111	0.1111
-0.0100	1.0100	1.1100	1.1011

27. 已知  $[X]_{\text{补}}$  为下列各值：(1)0.1110, (2)1.1100, (3)0.0001, (4)1.1111, (5)1.0000。求  $X$  (真值)。

答： $X$  的真值如下：

(1)0.1110 (2)-0.0100 (3)0.0001 (4)-0.0001 (5)-1.0000

28. 已知  $X = 0.1011$ ,  $Y = -0.0101$ 。求  $[X]_{\text{补}}$ 、 $[X/2]_{\text{补}}$ 、 $[X/4]_{\text{补}}$ 、 $[2X]_{\text{补}}$ 、 $[Y]_{\text{补}}$ 、 $[Y/2]_{\text{补}}$ 、 $[Y/4]_{\text{补}}$  和  $[2Y]_{\text{补}}$ 。

答：  $[X]_{\text{补}} = 0.1011$

$[X/2]_{\text{补}} = 0.0101$  舍去最低位

$[X/4]_{\text{补}} = 0.0010$  再舍去最低位

$[2X]_{\text{补}}$  为溢出

$[Y]_{\text{补}} = 1.1011$

$[Y/2]_{\text{补}} = 1.1101$  舍去最低位

$[Y/4]_{\text{补}} = 1.1110$  再舍去最低位

$[2Y]_{\text{补}} = 1.0110$

二进制数除以 2, 可将数右移 1 位实现之, 但要保留符号位不变; 乘以 2, 将数左移 1 位, 最低位补充 0, 要判别数据是否溢出。

补码右移时要计入符号位, 即移动符号位的值, 但是符号位本身保持不变。左移时, 如果符号位改变, 则发生溢出。



溢出检查

(1) 概念

溢出：指运算结果超出了数的表示范围。通常，称大于机器所能表示的最大正数为上溢，称小于机器所能表示的最小负数为下溢。

(2) 采用双符号位判断溢出

双符号位法也称模4补码。运算结果的两个符号位 $S_{S1}S_{S2}$ 相同，表示未溢出；运算结果的两个符号位 $S_{S1}S_{S2}$ 不同，表示溢出，此时最高位符号位代表真正的符号。

符号位 $S_{S1}S_{S2}$ 的各种情况如下：

- ① $S_{S1}S_{S2} = 00$ ：表示结果为正数，无溢出。
- ② $S_{S1}S_{S2} = 01$ ：表示结果正溢出。
- ③ $S_{S1}S_{S2} = 10$ ：表示结果负溢出。
- ④ $S_{S1}S_{S2} = 11$ ：表示结果为负数，无溢出。

浮点数

规格化就是尾数除了符号位之外的最高位一定为1

浮点数是指小数点位置可浮动的数据，通常以下式表示：

$$N = M \cdot R^E$$

其中， $N$  为浮点数， $M$  (mantissa) 为尾数， $E$  (exponent) 为阶码， $R$  (radix) 称为“阶的基数 (底)”，而且  $R$  为一常数，一般为 2、8 或 16。在一台计算机中，所有数据的  $R$  都是相同的，于是不需要在每个数据中表示出来。因此，浮点数的机内表示一般采用以下形式。

$M_s$	$E$	$M$
1 位	$n+1$ 位	$m$ 位

$M_s$  是尾数的符号位，设置在最高位上。

$E$  为阶码，有  $n+1$  位，一般为整数，其中有一位符号位，设置在  $E$  的最高位上，用来表示正阶或负阶。

$M$  为尾数，有  $m$  位，由  $M_s$  和  $M$  组成一个定点小数。 $M_s=0$ ，表示正号； $M_s=1$ ，表示负号。为了保证数据精度，尾数通常用规格化形式表示：当  $R=2$ ，且尾数值不为 0 时，其绝对值应大于或等于  $(0.5)_{10}$ 。对非规格化浮点数，通过将尾数左移或右移，并修改阶码值使之满足规格化要求。假设浮点数的尾数为 0.0011，阶码为 0100 (设定  $R=2$ )，规格化时，将尾数左移 2 位，而成为 0.1100，阶码减去  $(10)_2$ ，修改成 0010，浮点数的值保持不变。

当一个浮点数的尾数为 0 (不论阶码是何值)，或阶码的值比能在机器中表示的最小值还小时，计算机都把该浮点数看成零值，称为机器零。

根据 IEEE 754 国际标准，常用的浮点数有两种格式。

(1) 单精度浮点数 (32 位)，阶码 8 位，尾数 24 位 (内含 1 位符号位)。

(2) 双精度浮点数 (64 位)，阶码 11 位，尾数 53 位 (内含 1 位符号位)。

该标准还规定：基数为 2，阶码采用增码 (即移码)，尾数采用原码。因为规格化原码尾数的最高位恒为 1，所以不在尾数中表示出来，计算时在尾数的前面自动添上 1。

在多数通用机中，浮点数的尾数用原码或补码表示，阶码用补码或移码表示。移码的定

### 3.3 二进制乘法

#### 原码一位乘

例 3.31 设  $X=0.1101, Y=0.1011$ , 求  $X \cdot Y$ 。

解：计算过程如下：取双符号位

	部分积	乘数	
	00.0000	1011	
+X	00.1101		
	00.1101		
右移1位→	00.0110	1101	1(丢失)
+X	00.1101		
	01.0011		
右移1位→	00.1001	1110	1(丢失)
+0	00.0000		
	00.1001		
右移1位→	00.0100	1111	0(丢失)
+X	00.1101		
	01.0001		
右移1位→	00.1000	1111	1(丢失)
	乘积高位	乘积低位	

$$X \cdot Y = 0.1001111$$

乘积的符号位  $= X_0 \oplus Y_0 = 0 \oplus 0 = 0$ , 乘积为正数。

#### 补码一位乘

补码的X符号位加入运算

若  $Y < 0$ , 最后加  $[-X]_{补}$

补码一位乘法的规则如下(证明略)：

设被乘数  $[X]_{补} = X_0.X_1X_2 \cdots X_n$ , 乘数  $[Y]_{补} = Y_0.Y_1Y_2 \cdots Y_n$ , 则有：

$$[X \cdot Y]_{补} = [X]_{补} (0.Y_1Y_2 \cdots Y_n) - [X]_{补} \cdot Y_0$$

从公式中可见, 如果  $Y$  为负数(即  $Y_0 = 1$ ), 需要补充进行  $-[X]_{补}$  操作;  $Y$  为正数( $Y_0 = 0$ ), 则不需要。

例 3.32 设  $X = -0.1101, Y = 0.1011$ , 即:  $[X]_{补} = 11.0011, [Y]_{补} = Y = 0.1011$ , 求  $[X \cdot Y]_{补}$ 。

解：计算过程如下：

部分积	乘数	说明
00.0000	1 0 1 1	初始值
$+[X]_{\text{补}}$ 11.0011		$+[X]_{\text{补}}$
11.0011		
右移1位→ 11.1001	1 1 0 1	右移1位
$+[X]_{\text{补}}$ 11.0011		$+[X]_{\text{补}}$
10.1100		
右移1位→ 11.0110	0 1 1 0	右移1位
$+0$ 00.0000		$+0$
11.0110		
右移1位→ 11.1011	0 0 1 1	右移1位
$+[X]_{\text{补}}$ 11.0011		$+[X]_{\text{补}}$
10.1110		
右移1位→ 11.0111	0 0 0 1	右移1位
乘积高位	乘积低位	

$$[X \cdot Y]_{\text{补}} = 1.01110001$$

$$X \cdot Y = -0.10001111$$

例 3.33 设  $X = -0.1101$ ,  $Y = -0.1011$ , 即:  $[X]_{\text{补}} = 11.0011$ ,  $[Y]_{\text{补}} = 11.0101$ , 求  $[X \cdot Y]_{\text{补}}$ 。

解: 计算过程如下:

部分积	乘数	说明
00.0000	0 1 0 1	初始值
$+[X]_{\text{补}}$ 11.0011		$+[X]_{\text{补}}$
11.0011		
右移1位→ 11.1001	1 0 1 0	右移1位
$+0$ 00.0000		$+0$
11.1001		
右移1位→ 11.1100	1 1 0 1	右移1位
$+[X]_{\text{补}}$ 11.0011		$+[X]_{\text{补}}$
10.1111		
右移1位→ 11.0111	1 1 1 0	右移1位
$+0$ 00.0000		$+0$
11.0111		
右移1位→ 11.1011	1 1 1 1	右移1位
$+[ -X ]_{\text{补}}$ 00.1101		$+[ -X ]_{\text{补}}$
00.1000	1 1 1 1	
乘积高位	乘积低位	

$$[X \cdot Y]_{\text{补}} = 0.10001111$$

补码一位乘：布斯公式

布斯法，Y的符号位也参加运算，单符号位，Y最后补0（Y左右两边都要补数字）

部分积 + [(Y<sub>(i+1)</sub>-Y<sub>i</sub>\*X)补

注意，不需要全部右移完毕，在下题中右移4次就结束。

将前述补码乘法公式进行变换，可得出另一公式，是由布斯(Booth)提出的，又称为“布斯公式”。其运算规则如下(Y<sub>i+1</sub>与Y<sub>i</sub>为相邻两位)。

- (1)  $Y_{i+1}-Y_i=0$  ( $Y_{i+1}Y_i=00$  或  $11$ )，部分积加 0，右移 1 位。
- (2)  $Y_{i+1}-Y_i=1$  ( $Y_{i+1}Y_i=10$ )，部分积加  $[X]_{补}$ ，右移 1 位。
- (3)  $Y_{i+1}-Y_i=-1$  ( $Y_{i+1}Y_i=01$ )，部分积加  $[-X]_{补}$ ，右移 1 位。

最后一步( $i=n+1$ )不移位。

例 3.34 设  $X=-0.1101, Y=0.1011$ ，即： $[X]_{补}=11.0011, [Y]_{补}=0.1011$ ，求  $[X \cdot Y]_{补}$ 。

解：计算过程如下：

部分积	乘数	说明
00. 0000	0. 1 0 1 1 0	初始值,乘数最低位之后补0
+ 00. 1101		$Y_5Y_4=01$ $+[-X]_{补}$
00. 1101		
→ 00. 0110	1 0 1 0 1 1	右移1位
+ 00. 0000		$Y_4Y_3=11$ $+0$
00. 0110		
→ 00. 0011	0 1 0 1 0 1	右移1位
+ 11. 0011		$Y_3Y_2=10$ $+ [X]_{补}$
11. 0110		
→ 11. 1011	0 0 1 0 1 0	右移1位
+ 00. 1101		$Y_2Y_1=01$ $+ [-X]_{补}$
00. 1000		
→ 00. 0100	0 0 0 1 0 1	右移1位
+ 11. 0011		$Y_1Y_0=10$ $+ [X]_{补}$
11. 0111	0 0 0 1	
乘积高位	乘积低位	
$[X \cdot Y]_{补}=1.01110001$		
$X \cdot Y=-0.10001111$		

3.4 二进制除法

原码一位除：加减交替法

商带符号位，多一位

加减交替法是对恢复余数除法的一种修正。当某一次求得的差值(余数  $R_i$ )为负时,不是恢复它,而是继续求下一位商,但用加上除数( $+Y$ )的办法来取代 $-Y$ 操作,其他操作依然不变。其原理证明如下。

在恢复余数除法中,若第  $i-1$  次求商的余数为  $R_{i-1}$ ,下一次求商的余数为  $R_i$ ,则:

$$R_i = 2R_{i-1} - Y$$

如果  $R_i < 0$ , 商的第  $i$  位上 0, 并执行操作: 恢复余数( $+Y$ ), 将余数左移一位, 再减  $Y$ , 得  $R_{i+1}$ 。其过程可用公式表示如下:

$$R_{i+1} = 2(R_i + Y) - Y = 2R_i + 2Y - Y = 2R_i + Y, \text{由此得到证明。}$$

所以可得出加减交替法的规则如下:

当余数为正时, 商上 1, 求下一位商的办法是, 余数左移一位, 再减去除数; 当余数为负时, 商上 0, 求下一位商的办法是, 余数左移一位, 再加上除数。此方法不用恢复余数, 所以又叫不恢复余数法。但若最后一次上商为 0, 而又需得到正确余数, 则在这最后一次仍需恢复余数。

**例 3.36** 设  $X=0.1011, Y=0.1101$ , 用加减交替法求  $X/Y$ 。

解:  $[-Y]_{补} = 11.0011$

被除数(余数)		操作说明
0 0. 1 0 1 1	0 0 0 0 0	开始情形
+ ) 1 1. 0 0 1 1		$+[-Y]_{补}$
1 1. 1 1 1 0	0 0 0 0 0	不够减, 商上 0
1 1. 1 1 0 0	0 0 0 0 0	左移
+ ) 0 0. 1 1 0 1		$+Y$
0 0. 1 0 0 1	0 0 0 0 1	够减, 商上 1
0 1. 0 0 1 0	0 0 0 1 0	左移
+ ) 1 1. 0 0 1 1		$+[-Y]_{补}$
0 0. 0 1 0 1	0 0 0 1 1	够减, 商上 1
0 0. 1 0 1 0	0 0 1 1 0	左移
+ ) 1 1. 0 0 1 1		$+[-Y]_{补}$
1 1. 1 1 0 1	0 0 1 1 0	不够减, 商上 0
1 1. 1 0 1 0	0 1 1 0 0	左移
+ ) 0 0. 1 1 0 1		$+Y$
0 0. 0 1 1 1	0 1 1 0 1	够减, 商上 1
余数	商	

$$X/Y = 0.1101 \quad \text{余数} = 0.0111 \times 2^{-4}$$

最后说明如下:

(1) 对定点小数除法, 首先要比较除数和被除数的绝对值的大小, 以检查是否出现商溢出的情况。

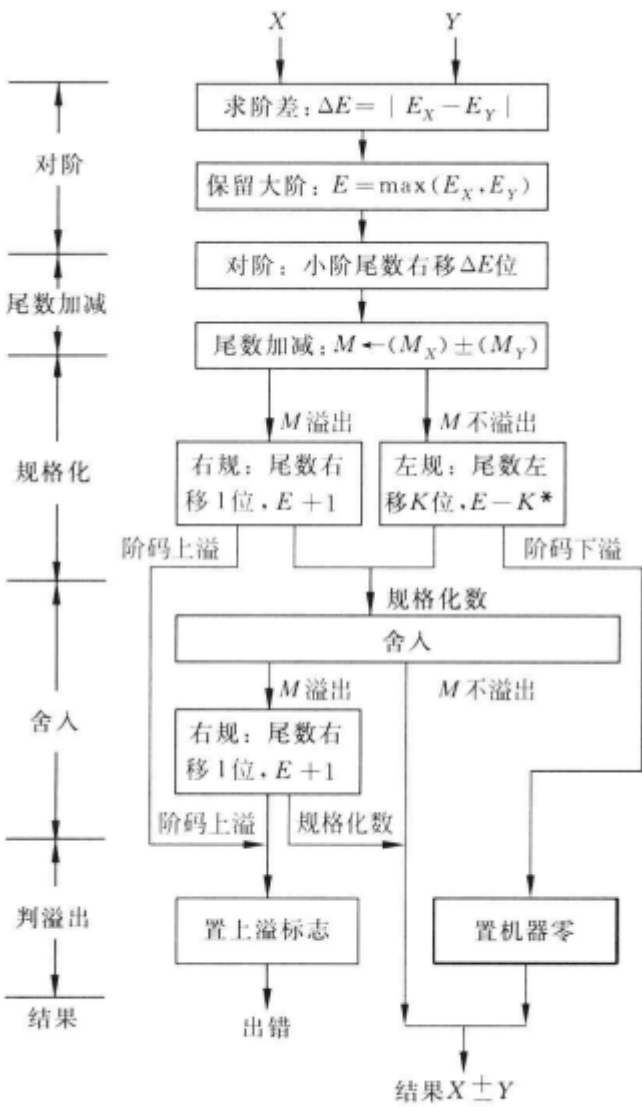
(2) 商的符号为相除两数的符号的半加和(异或值)。

(3) 被除数的位数可以是除数的两倍, 其低位的数值部分开始时放在商寄存器中。运算过程中, 放被除数和商的寄存器同时移位, 并将商寄存器中的最高位移到被除数寄存器的最低位中。

(4) 实现除法的逻辑电路与乘法的逻辑电路(如图 3.5 所示)极相似, 但在 A 寄存器中放被除数/余数, B 寄存器中放除数, C 寄存器中放商(如被除数为双倍长, 在开始时 C 中放被除数的低位)。此外, 移位电路应有左移 1 位的功能, 以及将  $Y$  或  $[-Y]_{补}$  送 ALU 的电路。

### 3.5 浮点数运算

#### 浮点数加减法



\*如果已为规格化数, 则  $K=0$ , 不移位。

图 3.8 规格化浮点数加减运算流程

#### 浮点数乘法

##### 2. 浮点数的舍入处理

在计算机中, 浮点数的尾数是用确定的位数来表示的, 但浮点数的运算结果却常常超过给定的位数。如加减运算过程中的对阶和右规处理, 会使尾数低位部分的一位或多位值因移位而丢失; 乘除运算(无论是定点数或浮点数)可能产生超过给定位数的结果, 在这里讨论如何处理这些多出来的位上的值。处理的原则是尽量减少本次运算所造成的误差以及按此



原则产生的累计误差。

(1) 无条件地丢掉正常尾数最低位之后的全部数值。这种办法被称为截断处理,其好处是处理简单,缺点是影响结果的精度。

(2) 运算过程中保留右移中移出的若干高位的值,然后再按某种规则用这些位上的值修正尾数。这种处理方法被称为舍入处理。较简单的舍入方法是:只要尾数最低位为1,或移出去的几位中有1,就把尾数的最低位置1,否则仍保持原有的0值。或者采用更简便的方法,即最低位恒置1的方法。另外一种办法是0舍1入法(相当于十进制中的四舍五入法),即当丢失的最高位的值为1时,把这个1加到最低数值位上进行修正,否则舍去丢失的各位的值,其缺点是要多进行一次加法运算。下面举例说明0舍1入的情况。

**例 3.40** 设有5位数(其中有一附加位),用原码或补码表示,舍入后保留4位结果。

设:  $[X]_{\text{原}} = 0.11011$       舍入后  $[X]_{\text{原}} = 0.1110$

$[X]_{\text{原}} = 0.11100$       舍入后  $[X]_{\text{原}} = 0.1110$

$[X]_{\text{补}} = 1.00101$       舍入后  $[X]_{\text{补}} = 1.0011$

$[X]_{\text{补}} = 1.00100$       舍入后  $[X]_{\text{补}} = 1.0010$

舍入后产生了误差,但误差值小于末位的权值。

### 3. 浮点乘法运算步骤

下面举例说明浮点乘法的运算步骤。

**例 3.41** 阶码4位(移码),尾数8位(补码,含1符号位),阶码以2为底。运算结果仍取8位尾数。

设:  $X = 2^{-5} \cdot 0.1110011$ ,  $Y = 2^3 \cdot (-0.1110010)$ ,  $X, Y$  为真值,此处阶码用十进制表示,尾数用二进制表示。运算过程中阶码取双符号位。

(1) 求乘积的阶码。乘积的阶码为两数阶码之和。

$$[E_X + E_Y]_{\text{移}} = [E_X]_{\text{移}} + [E_Y]_{\text{移}} = 00011 + 00011 = 00110$$

(2) 尾数相乘。用定点数相乘的办法,此处仅列出结果,不进行详细计算。

$$[X \cdot Y]_{\text{补}} = \begin{array}{cc} \underline{1.0011001} & \underline{1001010} \\ \text{高位部分} & \text{低位部分} \end{array} \quad (\text{尾数部分})$$

(3) 规格化处理。本例尾数已规格化,不需要再处理。

(4) 舍入。尾数(乘积)低位部分的最高位为1,需要舍入,在乘积高位部分的最低位加1,因此得到:  $[X \cdot Y]_{\text{补}} = 1.0011010$  (尾数部分)

(5) 判溢出。阶码未溢出,故结果正确。

$$X \cdot Y: \quad \begin{array}{cc} 0110 & 1.0011010 \\ \text{阶码(移码)} & \text{尾数(补码)} \end{array}$$

$$X \cdot Y = 2^{-2} \cdot (-0.1100110)$$

乘法运算可能产生溢出的情况如下。

在求乘积的阶码(即两阶码相加)时,有可能产生上溢或下溢的情况;在进行尾数向左规格化时,也有可能产生下溢。

## 3.7 数据校验码

### 奇偶校验码

奇偶校验码是一种开销小,能发现数据代码中一位或奇数个位出错情况的编码,常用于存储器读写检查,或数据传送过程中的检查。它的实现原理是使码距由 1 增加到 2。若编码中有奇数个二进制位的值出错了,由 1 变成 0,或由 0 变成 1,这个码都将成为非法编码。实现的具体方法通常是为一个字节补充一个二进制位,称为校验位,使字节的 8 位和该校验位含有 1 值的个数为奇数或偶数。在使用奇数个 1 的方案进行校验时,称为奇校验;反之,则称为偶校验。校验位的值是由专设的线路实现的(如图 3.10 所示)。例如,当要把一个字节( $D_1 \sim D_8$ )的值写进主存时,用此电路形成校验位的值,然后将这 9 位的代码作为合法数据编码写进主存。当下一次读出这一代码时,再用相应线路检测读出的 9 位码的合法性。这种方案只能发现一位错或奇数个位错,但不能确定是哪一位错,也不能发现偶数个位错。考虑到一位出错的几率比多位同时出错的几率高得多,该方案还是有很好的实用价值。

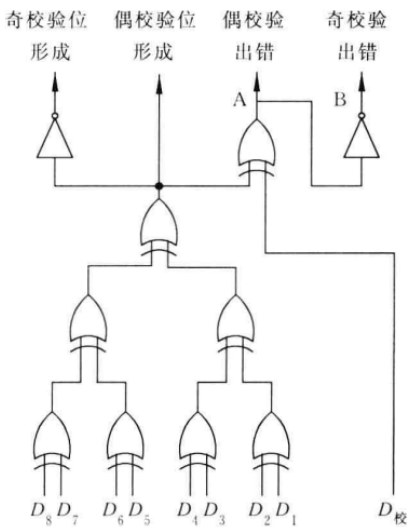


图 3.10 奇偶校验位的形成及校验

### 海明码

若海明码的最高位号为  $m$ ,最低位号为 1,即  $H_m H_{m-1} \cdots H_2 H_1$ ,则此海明码的编码规律通常是:

- (1) 校验位与数据位之和为  $m$ ,每个校验位  $P_i$  在海明码中被分在位号  $2^{i-1}$  的位置,其余各位为数据位,并按从低向高逐位依次排列的关系分配各数据位。
- (2) 海明码的每一位码  $H_i$  (包括数据位和校验位本身)由多个校验位校验,其关系是被校验的每一位位号要等于校验它的各校验位的位号之和。这样安排的目的是希望校验的结果能正确反映出出错位的位号。

表 3.6 出错的海明码位号和校验位位号的关系

海明码位号	数据位/校验位	参与校验的校验位位号	被校验位的海明码位号 = 校验位位号之和
$H_1$	$P_1$	1	$1=1$
$H_2$	$P_2$	2	$2=2$
$H_3$	$D_1$	1, 2	$3=1+2$
$H_4$	$P_3$	4	$4=4$
$H_5$	$D_2$	1, 4	$5=1+4$
$H_6$	$D_3$	2, 4	$6=2+4$
$H_7$	$D_4$	1, 2, 4	$7=1+2+4$
$H_8$	$P_4$	8	$8=8$
$H_9$	$D_5$	1, 8	$9=1+8$
$H_{10}$	$D_6$	2, 8	$10=2+8$
$H_{11}$	$D_7$	1, 2, 8	$11=1+2+8$
$H_{12}$	$D_8$	4, 8	$12=4+8$
$H_{13}$	$P_5$	13	$13=13$





2. 解:

(一)海明码的编码规律:

(1) 校验位与数据位之和为  $m$ , 每个校验位  $P_i$  在海明码中被分在位号  $2^{i-1}$  的位置, 其余各位为数据位, 并按从低向高逐位依次排列的关系分配各数据位。

(2) 海明码的每一位码  $H_i$ (包括数据位和校验位本身)由多个校验位校验, 其关系是被校验的每一位位号要等于校验它的各校验位的位号之和。这样安排的目的, 是希望校验的结果能正确反映出出错位的位号。 $m=k+r=4+3=7$

能发现并自动纠正移位错。

(二)  $H7H6H5H4H3H2H1, D4D3D2P3D1P2P1$

海明码位号	数据位/校验位	参与校验的校验位号	被校验位的海明码位号=校验位位号之和
H1	P1	1	1=1
H2	P2	2	2=2
H3	D1	1, 2	3=1+2
H4	P3	4	4=4
H5	D2	1, 4	5=1+4
H6	D3	2, 4	6=2+4
H7	D4	1, 2, 4	7=1+2+4

$$P1=D1 \oplus D2 \oplus D4$$

$$P2=D1 \oplus D3 \oplus D4$$

$$P3=D2 \oplus D3 \oplus D4$$

$$S1=P1 \oplus D1 \oplus D2 \oplus D4$$

$$S2=P2 \oplus D1 \oplus D3 \oplus D4$$

$$S3=P3 \oplus D2 \oplus D3 \oplus D4$$

(三) 海明校验线路略  $D4D3D2D1=1011$

(四) 海明码:  $P1=D1 \oplus D2 \oplus D4=1$ ,  $P2=D1 \oplus D3 \oplus D4=0$ ,  $P3=D2 \oplus D3 \oplus D4=0$ ,  $D4D3D2P3D1P2P1=1010101$ 。

## 第 4 章 主存储器

### 4.1 主存储器分类、技术指标和基本操作

#### 存储器的分类

## 1) 存储器的分类

### (1) 按在计算机中的层次分类

①主存储器，又称主存、内存。*CPU*可直接随机对其访问，其也可与高速缓冲存储器(*Cache*)和辅助存储器交换数据。

②辅助存储器，又称辅存、外存，辅存的内容需调入主存才能被*CPU*访问。

③高速缓冲存储器，简称*Cache*，位于主存与*CPU*之间，用来存放当前*CPU*经常使用的指令和数据，以便*CPU*能高速访问它们。

### (2) 按存储介质分类

分为磁表面存储器(磁盘、磁带)、半导体存储器和光存储器(光盘)等。

### (3) 按存取方式分类

①随机存储器(*RAM*)。存储器的任何一个存储单元都可以随机存取，且存取时间与存储单元的物理位置无关，*RAM*又分为静态*RAM*(*SRAM*)和动态*RAM*(*DRAM*)。

②只读存储器(*ROM*)。存储器的内容只能随机读出而不能写入，*ROM*与*RAM*的存取方式均为随机存取，信息写入之后就固定不变，即使断电，也不会丢失。

③串行访问存储器。包括顺序存取存储器(如磁带)和直接存取存储器(如磁盘)。

### (4) 按信息的可保存性分类

①易失性存储器(断电后，存储信息消失): *RAM*。

②非易失性存储器(断电后信息仍保持): *ROM*、磁盘、光盘。

③破坏性读出(某个存储单元的信息被读出时，原存储信息被破坏): 半导体存储器。

④非破坏性读出(某个存储单元的信息被读出时，原存储信息不被破坏): 磁盘。

## 主存储器的技术指标

### 2) 存储器的性能指标

(1) 存储容量 = 存储字数 × 字长 (如  $1K \times 8$  位)

存储字数表示存储器的地址空间大小, 字长表示一次存取操作的数据量。

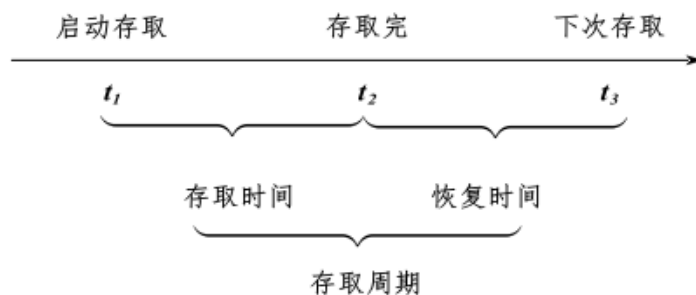
(2) 存储速度: 数据传输率 = 数据的宽度 / 存储周期

①存取时间 ( $T_a$ ): 存取时间是指从启动一次存储器操作到完成该操作所经历的时间, 分为读出时间和写入时间。

②存取周期 ( $T_m$ ): 存取周期又称读写周期或访问周期。它是指存储器进行一次完整的读写操作所需的全部时间, 即连续两次独立访问存储操作 (读或写操作) 之间所需的最小时间间隔。

③主存带宽 ( $B_m$ ): 主存带宽又称数据传输率, 表示每秒从主存进出信息的最大数量, 单位字/秒、字节/秒 (B/s) 或位/秒 (b/s)。

注: 存取时间不等于存储周期, 通常存储周期大于存取时间。



存取时间与存储周期的关系

## 4.2 读/写存储器

都是易失性存储器, 断电丢数据

SRAM 速度快, 体积小, 功率大

DRAM 便宜, 容量大

3. 静态随机存储器 (SRAM) 和动态随机存储器 (DRAM) 有何差别?

答: SRAM 用六管电路存储 1 位数据, 只要不停止供电, 数据不会丢失; DRAM 用一个管子和一个电容器存储 1 位数据, 电容上存储的电荷 (数据) 由于漏电而会消失, 因此每隔一定时间 (例如 2ms) 需要再充一次电, 这一过程称为再生或刷新。

SRAM 的集成度比 DRAM 低, 这是因为存储 1 位信息所需的芯片面积大, 所以价格也比较贵, 但读写时间短, 速度快。

从芯片的引出端来看, 地址线、数据线和读写命令两者都是需要的, 片选信号不同, SRAM 由片选允许 ( $\overline{CE}$ ) 或片选选择 ( $\overline{CS}$ ) 信号来控制是否能进行读写, DRAM 则由行选信号 ( $\overline{RAS}$ ) 来控制是否能进行读写。

计算机的主存储器一般使用 DRAM, 而 cache 存储器一般使用 SRAM。

静态随机存储器 SRAM

双稳态触发器

(1) SRAM 的工作原理

- ①存储元：存放一个二进制位的物理器件。
- ②存储单元：由地址码相同的多个存储元构成。
- ③存储体：若干个存储单元的集合称为存储体。

静态随机存储器（SRAM）的储元是用双稳态触发器（元晶体管 MOS）来记忆信息的，因此即使信息被读出后，它仍保持其原状态而不需要再生(非破坏性读出)。

SRAM 的存取速度快，但集成度低，功耗大，价格高，一般用于高速缓冲存储器。

动态随机存储器 DRAM

(2) DRAM 的工作原理

动态随机存储器（DRAM）是利用存储元电路中栅极电容上的电荷来存储信息的，但其电容上的电荷一般只能维持 $1\sim 2ms$ ，因此即使电源不断电，信息也会自动消失，为此，每隔一定时间必须刷新（即将存储单元的信息读出在立马写入），通常取 $2ms$ ，称为刷新周期。

为什么 DRAM 芯片的地址一般要分两次接收？

答：当芯片容量增大时，其地址线数量相应增加，分两次接收地址可将地址线的数量减少一半。引出端多，芯片面积大。

(3) SRAM 和 DRAM 的比较

特点 \ 类型	SRAM	DRAM
存储信息	触发器	电容
破坏性读出	非	是
需要刷新	不要	需要
送行列地址	同时送	分两次送
运行速度	快	慢
集成度	低	高
存储成本	高	低
主要用途	高速缓存	主机内存

4.3 非易失性半导体存储器

只读存储器 ROM

可编程 只读存储器 PROM

熔丝，一次性编程

**可擦可编程 只读存储器 EPROM**

编程次数不受限制，电写，紫外线擦除

**可电擦可编程序 只读存储器 E<sup>2</sup>EPROM**

电擦，电写，次数10万次

**快擦除读写存储器 Flash Memory**

即闪存

**4.4 容量扩展**

**位扩展**

1) 位扩展

位扩展指的是用多个存储器芯片对字长进行扩充。其连接方式是将多片存储器的地址、片选CS、读写控制端  $R/\overline{W}$  相应并联，数据端分别引出。图 4.15 是采用 2 个  $16K \times 4$  位芯片组成  $16K \times 8$  位的存储器，存储器字长 8 位。每个芯片字长 4 位，每片有 14 条地址线引出端和 4 条数据线引出端。

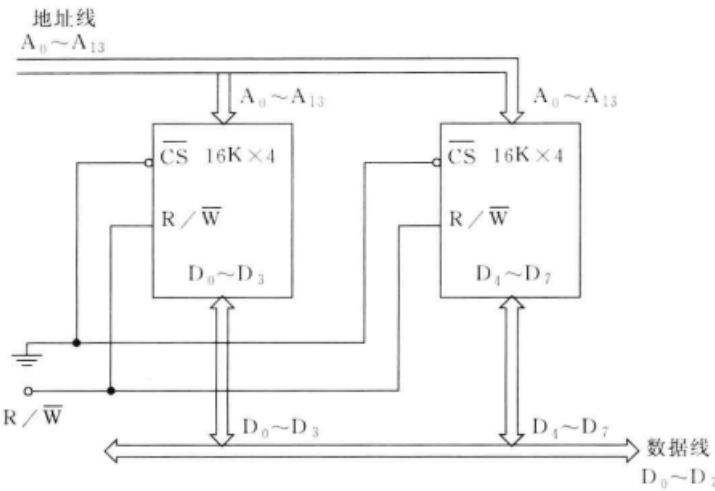


图 4.15 位扩展连接方式

**字扩展**

2) 字扩展

字扩展指的是增加存储器中字的数量。静态存储器进行字扩展时，将各芯片的地址线、数据线和读写控制线相应并联，而由片选信号来区分各芯片的地址范围。图 4.16 所示的字扩展存储器是用 4 个  $16K \times 8$  位芯片组成  $64K \times 8$  位存储器。数据线  $D_0 \sim D_7$  与各片的数据端相连，地址总线低位地址  $A_0 \sim A_{13}$  与各芯片的 14 位地址端相连，而两位高位地址  $A_{14}$ 、 $A_{15}$  经过译码器和 4 个片选端相连。图 4.15 中的  $R/\overline{W}$  与图 4.16 中的  $\overline{WE}$  是同一信号的不同表示。

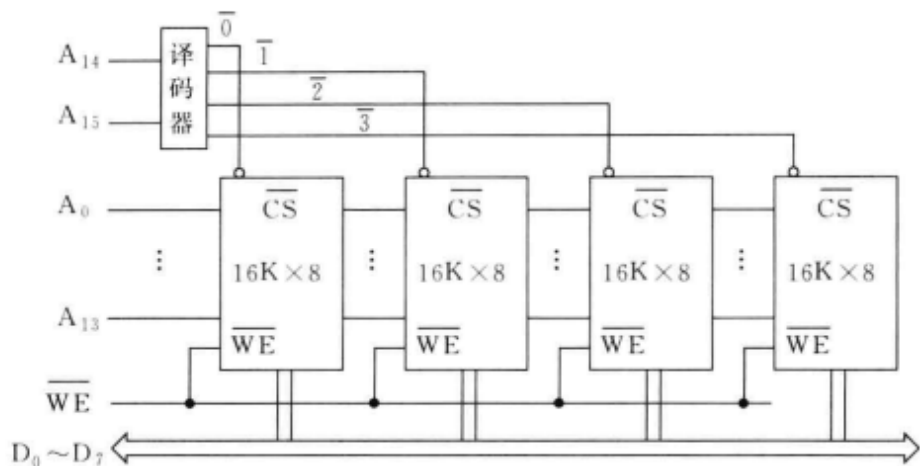


图 4.16 字扩展连接方式

## 字位同时扩展

11. 某机有一个地址空间为 0000H~1FFFH 的 ROM 区域,现在再用 SRAM 芯片 (8K×4) 组成一个 16K×8 位的 SRAM 区域,起始地址为 2000H,假设 SRAM 芯片有  $\overline{CS}$  和  $R/\overline{W}$  信号控制端。CPU 地址总线为  $A_{15} \sim A_0$ ,数据总线为  $D_7 \sim D_0$ ,控制信号为  $R/\overline{W}$  (读/写), $\overline{MREQ}$  (当存储器进行读或写操作时,该信号指示地址总线上的地址是有效的)。要求画出逻辑图。

答: 地址空间为 0000H~1FFFH 的 ROM 容量为 8KB (字长为 1B), ROM 和 SRAM 的地址为 13 位 ( $A_0 \sim A_{12}$ ),  $A_{13} \sim A_{15}$  经译码后控制 ROM 和 SRAM 的  $\overline{CS}$  端。  $\overline{CS}_0 = \overline{A_{15}} \overline{A_{14}} \overline{A_{13}}$ ,  $\overline{CS}_1 = \overline{A_{15}} \overline{A_{14}} A_{13}$ ,  $\overline{CS}_2 = \overline{A_{15}} A_{14} \overline{A_{13}}$ 。图 4.5 为逻辑图。

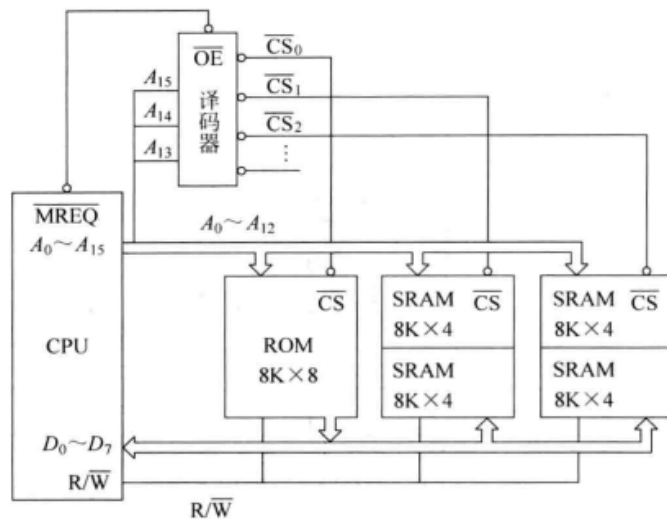


图 4.5 ROM 和 SRAM 存储器逻辑图

## 第 5 章 指令系统

### 5.2.2 指令操作码的扩展技术

通常是在指令字中用一个固定长度的字段来表示基本操作码,而对于一部分不需要某个地址码的指令,把它们的操作码扩充到该地址字段,这样既能充分地利用指令字的各个字段,又能在不增加指令长度的情况下扩展操作码的长度,使它能表示更多的指令。例如,设某机器的指令长度为 16 位,包括 4 位基本操作码字段和 3 个 4 位地址字段,其格式如图 5.1 所示。

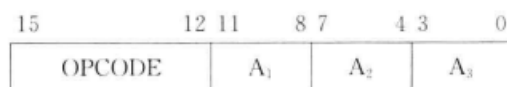


图 5.1 指令格式

若 4 位基本操作码全部用于表示三地址指令,则只有 16 条指令。但是,若三地址指令仅需 15 条,两地址指令需 15 条,一地址指令需 15 条,零地址指令需 16 条,共 61 条指令,应如何安排操作码?显然,只有 4 位基本操作码是不够的,必须将操作码的长度向地址码字段扩展才行。一种可供扩展的方法和步骤如下。

(1) 15 条三地址指令的操作码由 4 位基本操作码从 0000~1110 给出,剩下一个码点 1111 用于把操作码扩展到 A<sub>1</sub>,即 4 位扩展到 8 位。

(2) 15 条二地址指令的操作码由 8 位操作码从 11110000~11111110 给出,剩下一个码点 11111111 用于把操作码扩展到 A<sub>2</sub>,即从 8 位扩展到 12 位。

(3) 15 条一地址指令的操作码由 12 位操作码从 111111110000~111111111110 给出,剩下一个码点 111111111111 用于把操作码扩展到 A<sub>3</sub>,即从 12 位扩展到 16 位。

(4) 16 条零地址指令的操作码由 16 位操作码从 1111111111110000~1111111111111111 给出。

7. 某指令系统指令长 16 位,每个操作数的地址码长 6 位,指令分无操作数、单操作数和双操作数 3 类。若双操作数指令有  $K$  条,无操作数指令有  $L$  条,问单操作数指令最多可能有多少条?

答:本指令系统的操作码位数=指令长度-地址码长度=16-2×6=4 位。设单操作数指令最多有  $N$  条,则

$$L = ((2^4 - K) \times 2^6 - N) \times 2^6$$

$$N = (2^4 - K) \times 2^6 - L/2^6$$

如果  $L/2^6$  不是整数,则取大于  $L/2^6$  的最小整数。

## 第 6 章 中央处理器

### 6.1 控制器

#### 控制器的组成

##### 1. 程序计数器(PC)

即指令地址寄存器。用来存放当前正在执行的指令地址或即将要执行的下一条指令地址;而在有指令预取功能的计算机中,一般还需要增加一些程序计数器用来存放要预取的指令地址。

有两种途径来形成指令地址,其一是顺序执行的情况,通过程序计数器加 1 形成下一条指令地址(如存储器按字节编址,而指令长度为 4 个字节,则加 4)。其二是改变顺序执行程序的情况,一般由转移类指令形成转移地址送往程序计数器,作为下一条指令的地址。



## 2. 指令寄存器(IR)

用以存放当前正在执行的指令,以便在指令执行过程中控制完成一条指令的全部功能。

## 3. 指令译码器或操作码译码器

对指令寄存器中的操作码进行分析解释,产生相应的控制信号。

在执行指令过程中,需要形成有一定时序关系的操作控制信号序列,为此还需要下述组成部分。

## 4. 脉冲源及启停线路

脉冲源产生一定频率的脉冲作为整个机器的时钟脉冲,是机器周期和工作脉冲的基准信号,在机器刚加电时,还应产生一个总清信号(reset)。启停线路保证可靠地送出或封锁完整的时钟脉冲,控制时序信号的发生或停止,从而启动机器工作或使之停机。

## 5. 时序控制信号形成部件

当机器启动后,在 CLK 时钟作用下,根据当前正在执行的指令的需要,产生相应的时序控制信号,并根据被控功能部件的反馈信号调整时序控制信号。例如,当执行加法指令时,

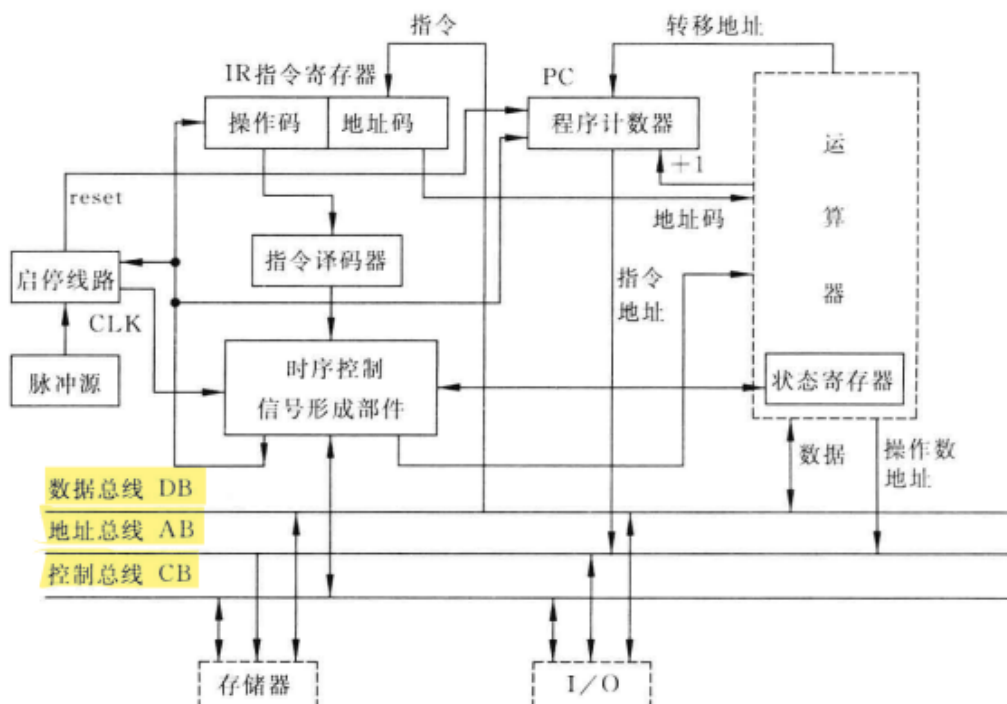


图 6.1 控制器基本组成框图

数据通路

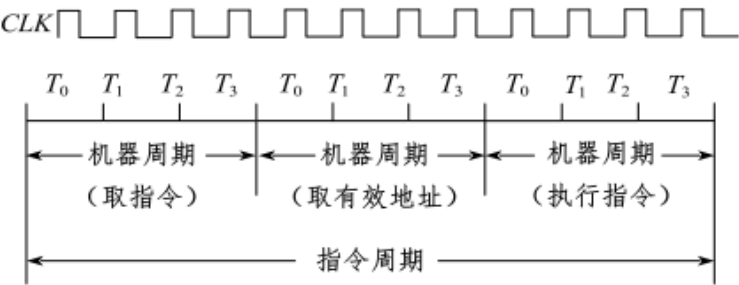
指令执行过程

1) 指令周期

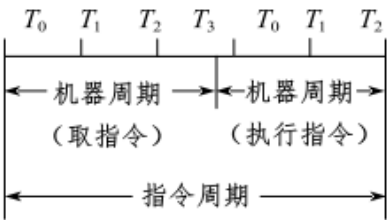
(1) 定义：CPU从主存中取出并执行一条指令的时间。

(2) 特点：

- ①指令周期通常用若干机器周期表示，一个机器周期又包含若干时钟周期(节拍或 $T$ 周期，为CPU操作的最基本单位)
- ②每个指令周期内的机器周期数可以不等，每个机器周期内的节拍也可以不等。



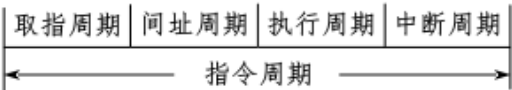
(a) 定长的机器周期



(b) 不定长的机器周期

指令周期、机器周期、节拍和时钟周期的关系

③完整的指令周期包括：取指、间址、执行、中断4个周期。



带有间址周期、中断周期的指令周期

取指周期：取指令

间址周期：取有效地址

执行周期：取操作数

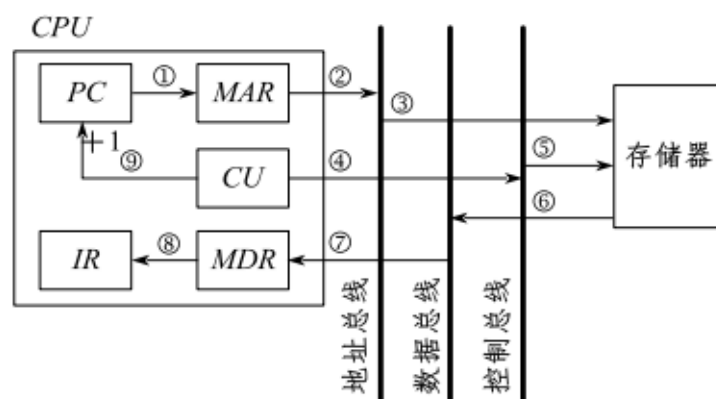
中断周期：保存程序断点

④CPU通过4个标志触发器FE、IND、EX和INT来区分取指、间址、执行、中断周期。

## 2) 指令周期的数据流

(1) 定义：即在指令执行的不同阶段，根据指令要求依次访问的数据序列。

(2) 取指周期：根据PC中的地址从主存中取出指令代码并存放于IR中。



取指周期的数据流

数据流：

①PC ②MAR ③地址总线 ④主存。

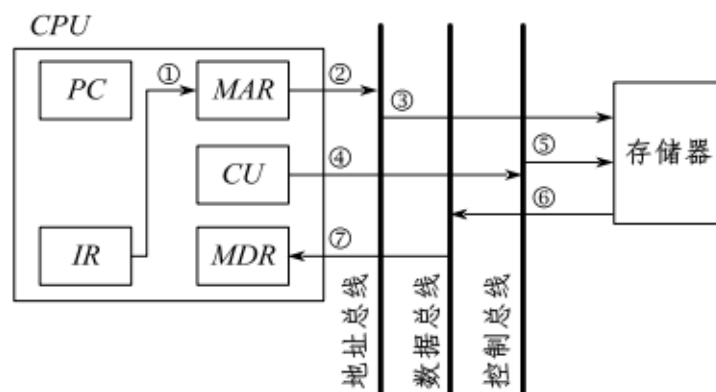
②CU 发出读命令 ④控制总线 ⑤主存。

③主存 ⑥数据总线 ⑦MDR ⑧IR (存放指令)。

④CU 发出控制信号 ⑨PC 内容加1。

## (3) 间址周期

取操作数的有效地址。



## 一次间址周期的数据流

数据流:

①  $Ad(IR)$  (或  $MDR$ ) ①  $MAR$  ② 地址总线 ③ 主存。

②  $CU$  发出读命令 ④ 控制总线 ⑤ 主存。

③ 主存 ⑥ 数据总线 ⑦  $MDR$  (存放有效地址)。

其中,  $Ad(IR)$  表示取出  $IR$  中存放的指令字的地址字段

### (4) 执行周期

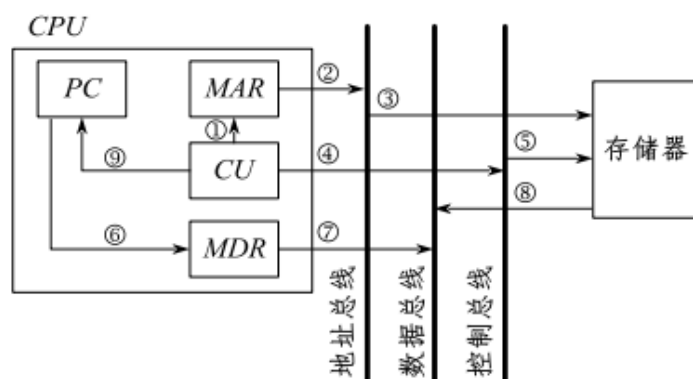
取操作数, 并根据  $IR$  中指令字的操作码通过  $ALU$  操作产生执行结果。

数据流: 不同指令执行周期操作不同, 无统一的数据流。

### (5) 中周期

处理中断请求。

假设程序断点存入堆栈中, 并用  $SP$  指向栈顶地址, 且进栈操作作为先修改栈顶指针, 后存入数据, 且默认为向下生长型。



## 中断周期的数据流

数据流:

①  $CU$  控制将  $SP$  减1,  $SP$  ①  $MAR$  ② 地址总线 ③ 主存

②  $CU$  发出写命令 ④ 控制总线 ⑤ 主存。

③  $PC$  ⑥  $MDR$  ⑦ 数据总线 ⑧ 主存 (程序断点存入主存)。

④  $CU$  (中断服务程序的入口地址) ⑨  $PC$ 。

### 3) 指令执行方案

(1) 单指令周期: 对于所有指令都选用相同的执行时间来完成。

特点:

① 指令之间串行执行, 即下一条指令只能在前一条指令执行结束后才能启动。

② 指令周期取决于执行时间最长的指令的执行时间, 降低了整个系统的运行速度。

(2) 多指令周期: 对不同类型的指令选用不同的执行步骤。

特点:

① 指令之间串行操作。

② 不同指令的周期数可不同, 不同指令的时钟周期数也可不同。

(3) 流水线方案: 指令之间并行执行, 力争在每个时钟周期完成一条指令的执行过程。

题：数据通路

都符合要求)。

(4) 时序控制信号的形成部件。在时钟、机器周期和指令译码信号等的控制下,产生全机工作所需的控制信号,有硬连线(组合逻辑)和微程序两种控制方式。

3. CPU 结构如图 6.1 所示,其中有一个累加寄存器 AC、1 个状态寄存器和其他 4 个寄存器,各部分之间的连线表示数据通路,箭头表示信息传送方向。要求:

(1) 标明图中 a、b、c、d 这 4 个寄存器的名称。

(2) 简述指令从主存取到控制器的数据

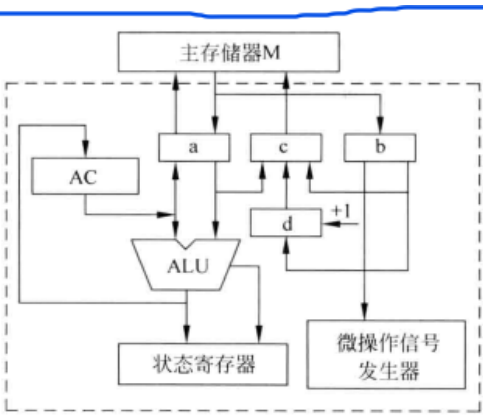


图 6.1 CPU 结构

通路。

(3) 简述数据在运算器和主存之间进行存取/访问的数据通路。

答: (1) 已知 AC 为累加器, ALU 为算逻运算部件。其输入的 2 个数来源为 (AC) 和 (a)。运算结果送入 AC。主存 M 与 CPU 之间的连线有地址线与数据线, 其中地址线的信息仅从 CPU 传送到主存, 数据则允许双方向传送, 由此可见 a 与 c 分别是主存的数据寄存器 MDR 和地址寄存器 MAR。MDR 和 MAR 一般设置在主存中。在 CPU 中必须有程序计数器 PC 和指令寄存器 IR。PC 送地址到地址寄存器, 并有自动加 1 的功能, 因此 d 寄存器为 PC。IR 接收从主存来的指令, 并控制微操作信号发生器, 因此 b 寄存器是 IR。IR 向 PC 传送的是转移地址 (当转移时, IR 的地址码字段存放的是转移地址)。

(2) 指令地址从 d(PC) 送到 c(地址寄存器 MAR), 数据从主存送到 b(指令寄存器 IR)。

(3) 读写地址由 b(IR 的地址段)→c(MAR), 读时数据从主存→a(MDR)→ALU→AC, 写时数据从 AC→a(MDR)→主存。

6.2 微程序

控存, 控制存储器, ROM

微指令: 在微程序控制的计算机中, 将由同时发出的控制信号所执行的一组微操作称为微指令, 所以微指令就是把同时发出的控制信号的有关信息汇集起来而形成的。将一条指令分成若干条微指令, 按次序执行这些微指令, 就可以实现指令的功能。组成微指令的微操作又称微命令。

微程序: 计算机的程序由指令序列构成, 而计算机每条指令的功能均由微指令序列解释完成, 这些微指令序列的集合就叫做微程序。

控制存储器: 微程序一般是存放在专用的存储器中的, 由于该存储器主要存放控制命令 (信号) 与下一条执行的微指令地址 (简称为下址), 所以被叫做控制存储器。一般计算机指令系统是固定的, 所以实现指令系统的微程序也是固定的, 于是控制存储器可以用只读存储器实现。又由于机器内控制信号数量比较多, 再加上决定下址的地址码有一定宽度, 所以控制存储器的字长比机器字长要长得多。

执行一条指令实际上就是执行一段存放在控制存储器中的微程序。

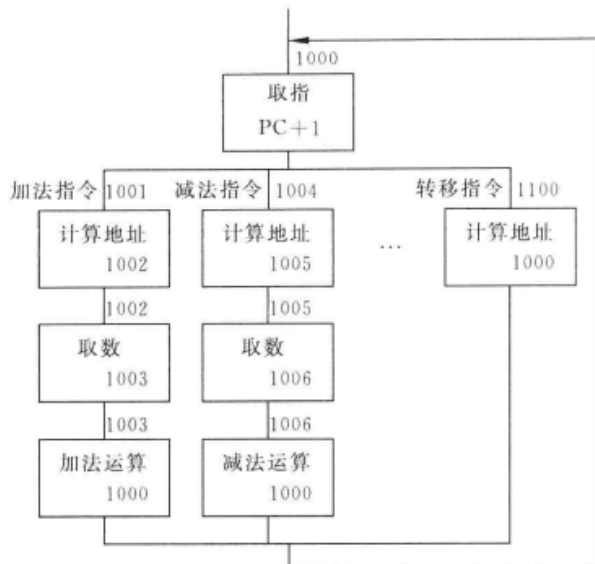


图 6.8 微程序流程图举例

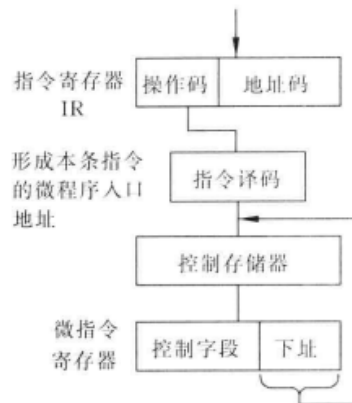


图 6.9 微程序控制器简化框图

## 6.3 微程序设计

### 微程序控制字段的编译法

#### 1. 直接控制法

在微指令的控制字段中，**每一位代表一个微命令**，在设计微指令时，是否发出某个微命令，只要将控制字段中相应位置成 1 或 0，这样就可打开或关闭某个控制门，这就是直接控制法，在 6.2 节中所讲的就是这种方法。但在某些复杂的计算机中，微命令甚至可多达三四百个，这使微指令字长达到难以接受的地步，并要求机器有大容量控制存储器，为了改进设计，出现了以下各种编译法。

#### 2. 字段直接编译法

在计算机中的各个控制门，在任一微周期内，**不可能同时被打开**，而且大部分是关闭的（即相应的控制位为 0）。所谓微周期，指的是一条微指令所需的执行时间。如果有若干个（一组）微命令，**在每次选择使用它们的微周期内，只有一个微命令起作用**，那么这若干个微命令是**互斥的**。例如，向主存储器发出的读命令和写命令是互斥的；又如在 ALU 部件中，送往 ALU 两个输入端的数据来源往往不是唯一的，而每个输入端在任一微周期中只能输入一个数据，因此控制该输入门的微命令是互斥的。**选出互斥的微命令，并将这些微命令编成一组，成为微指令字的一个字段，用二进制编码来表示**。例如，将 7 个互斥的微命令编成一组，用三位二进制码分别表示每个微命令，那么在微指令中，该字段就从 7 位减成 3 位，缩短了微指令长度。而在微指令寄存器的输出端为该字段增加一个译码器，该译码器的输出即为原来的微命令（如图 6.16 所示）。

字段长度与所能表示的微命令数的关系如下。

字段长度	微命令数
2 位	2~3
3 位	4~7
4 位	8~15



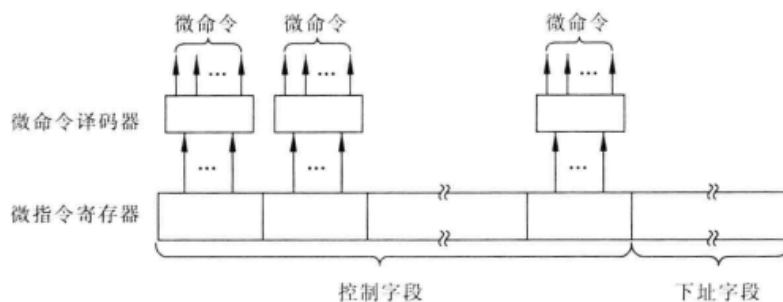


图 6.16 字段直接编译法

一般每个字段要留出一个代码,表示本段不发出任何微命令,因此当字段长度为 3 位时,最多只能表示 7 个互斥的微命令,通常代码 000 表示不发微命令。

## 题：微指令控制字段格式设计

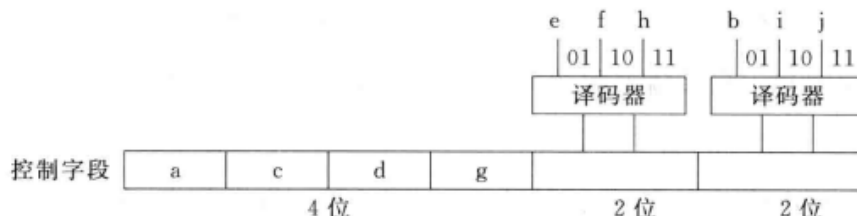
15. 某机有 8 条微指令 I1~I8,每条微指令所包含的微命令控制信号如下表所示。

微指令	微命令控制信号									
	a	b	c	d	e	f	g	h	i	j
I1	✓	✓	✓	✓	✓					
I2	✓			✓		✓	✓			
I3		✓						✓		
I4			✓							
I5			✓		✓		✓		✓	
I6	✓							✓		✓
I7			✓	✓				✓		
I8	✓	✓						✓		

a~j 分别对应 10 种不同性质的微命令信号,假设一条微指令的控制字段为 8 位,请安排微指令的控制字段格式。

答：由于微命令有 10 个,而控制字段只有 8 位,可以将互斥的命令编成一组,成为微指令字的一个字段,用二进制编码来表示。实现的方案可以有多种。下面举出两种方案。

方案 1:



a、c、d、g 为直接控制字段。

e、f、h 编成一组, b、i、j 编成一组。当编码为 00 时,为不操作。

方案 2:

a、c、d、i 为直接控制字段。

b、g、j 编成一组, e、f、h 编成一组。

设计思路:

① 直接控制法  $\boxed{8}$   $8 < 10$  不满足

② 字段直接 + 直接  $\boxed{2 \quad 6}$   
 $\Downarrow$   
 $3 + 6$   $9 < 10$  不满足

③ 字段直接 + 直接  $\boxed{3 \quad 5}$   
 $\Downarrow$   
最多可用 7, 实际需要 5  
 $\Downarrow$   
 $7 + 5$   
(15)  $10 = 10$  但找不到 5 条互斥的, 不满足

④ 字段直接 + 字段直接 + 直接  $\boxed{2 \quad 2 \quad 4}$  满足.  
互斥 互斥  
 $3 + 3 + 4$

$\begin{bmatrix} 01 \\ 10 \\ 11 \end{bmatrix} \begin{bmatrix} 01 \\ 10 \\ 11 \end{bmatrix} [a \ c \ d \ g]$   
 $\Downarrow$   
 $\begin{matrix} e \\ f \\ h \end{matrix} \quad \begin{matrix} b \\ i \\ j \end{matrix}$

## 微指令格式

### 1. 水平型微指令

在 6.2 节中所介绍的例子即是采用直接控制法进行编码的, 属于水平型微指令的典型例子, 其基本特点是在一条微指令中定义并执行多个并行操作微命令。在实际应用中, 直接



控制法、字段编译法(直接、间接编译法)经常应用在同一条水平型微指令中。

## 2. 垂直型微指令

在微指令中设置有微操作码字段,由微操作码规定微指令的功能,称为垂直型微指令。其特点是不强调实现微指令的并行控制功能,通常一条微指令只要求能控制实现一二种操作。这种微指令格式与指令相似,每条指令有一个操作码;每条微指令有一个微操作码。

## 3. 水平型微指令与垂直型微指令的比较

(1) 水平型微指令并行操作能力强,效率高,灵活性强,垂直型微指令则差。

(2) 水平型微指令执行一条指令的时间短,垂直型微指令执行时间长。

因为水平型微指令的并行操作能力强,因此与垂直型微指令相比,可以用较少的微指令数来实现一条指令的功能,从而缩短了指令的执行时间。

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

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

水平型微指令与机器指令差别很大,需要对机器的结构、数据通路、时序系统以及微命令很精通才能进行设计。对机器已有的指令系统进行微程序设计是设计人员而不是用户的事情,因此这一特点对用户来讲并不重要。然而某些计算机允许用户扩充指令系统,此时就需要注意是否容易编写微程序,有关问题将在 6.3.4 节的“动态微程序设计”中讨论。

# 6.5 流水线

## CPU 性能公式

### 3. CPU 性能公式

执行一个程序所需的 CPU 时间可以这样来计算:

$$\text{CPU 时间} = \text{执行程序所需的时钟周期数} \times \text{时钟周期时间}$$

其中时钟周期时间是时钟频率的倒数。

引入新的参数 CPI(Cycles Per Instruction),即每条指令的平均时钟周期数,有时简称为指令的平均时钟周期数。

$$\text{CPI} = \text{执行程序所需的时钟周期数} / \text{所执行的指令条数}$$

则有以下的 CPU 性能公式。

$$\text{CPU 时间} = \text{IC} \times \text{CPI} \times \text{时钟周期时间}$$

其中 IC 为所执行的指令条数。

根据这个公式可知,CPU 的性能取决于以下三个参数。

(1) 时钟周期时间。取决于硬件实现技术和计算机组成。

(2) CPI。取决于计算机组成和指令系统的结构。

(3) IC。取决于指令系统的结构和编译技术。

所执行的指令条数

改进任何一个参数,都能提高 CPU 的性能。不过,这些参数是相互关联的,很难做到能单独地改进某一个参数指标而不影响其他两个指标(变坏)。幸运的是,有可能设法使这

指令数 × 指令的周期数  
× 周期时间

## 题：CPU 性能

微机 A 和 B 是采用不同主频的 CPU 芯片,片内逻辑电路完全相同。若 A 机的 CPU 主频为 8MHz,B 机为 12MHz。则 A 机的 CPU 主振周期为  $\boxed{A}$   $\mu\text{s}$ 。如 A 机的平均指令执行速度为 0.4MIPS,那么 A 机的平均指令周期为  $\boxed{B}$   $\mu\text{s}$ ,B 机的平均指令执行速度为  $\boxed{C}$  MIPS。

供选择的答案

A~C: ①0.125; ②0.25; ③0.5; ④0.6; ⑤1.25; ⑥1.6; ⑦2.5。

答: A——①,B——⑦,C——④。

A 机主振周期 =  $1/8\text{MHz} = 0.125\mu\text{s}$

A 机的平均指令周期 =  $1/0.4\text{MIPS} = 2.5\mu\text{s}$

B 机的平均指令执行速度 =  $0.4\text{MIPS} \times \frac{12}{8} = 0.6\text{MIPS}$

## 流水线阻塞

流水线阻塞是指在流水线处理器中,由于某些原因导致流水线暂停或减速,从而影响处理器性能。流水线阻塞的主要原因有**数据相关**、**程序转移**和**中断请求**。下面分别解释这三种情况及其解决方法:

### 1. 数据相关 (Data Dependency) :

数据相关是指一个指令需要等待前一个或多个指令完成数据操作才能执行。这种情况下,流水线需要暂停,等待所需数据准备好。数据相关主要包括RAW (Read After Write, 先写后读)、WAR (Write After Read, 先读后写) 和WAW (Write After Write, 先写后写)。

解决方法:

- 指令调度: 对指令进行重新排序,使得数据相关的指令之间有足够的间隔,以便在等待数据时处理其他指令。
- 数据前推 (Data Forwarding): 将上一级流水线段的计算结果直接传递给下一级流水线段,而不是等待寄存器更新。
- 寄存器重命名: 通过使用更多的寄存器来消除WAW和WAR相关。

### 2. 程序转移 (Control Dependency) :

程序转移是指流水线处理器遇到分支指令 (如条件跳转、循环等) 时,需要等待分支指令执行完毕才能确定下一条指令的地址。这会导致流水线暂停,等待新指令地址。

解决方法:

- 预测分支 (Branch Prediction): 通过硬件或软件预测分支的结果,提前获取下一条指令的地址,以减少等待时间。
- 延迟分支 (Delayed Branching): 将分支指令后面的无关指令 (与分支无关) 提前执行,以减少流水线暂停时间。
- 分支目标缓冲 (Branch Target Buffer, BTB): 缓存分支指令的目标地址,以加速分支指令的执行。

### 3. 中断请求 (Interrupt Request) :

中断请求是指处理器在执行指令过程中收到外部或内部的中断信号,需要暂停当前任务,切换到中断服务程序。这会导致流水线暂停,等待中断处理完成。

解决方法:

- 中断屏蔽: 在处理关键任务时,暂时屏蔽低优先级的中断请求,以减少流水线阻塞。

- 优先级设置：为不同类型的中断设置优先级，确保高优先级中断能够及时处理，降低流水线阻塞的影响。
- 硬件支持：使用硬件支持的中断处理机制，如嵌套中断向量表（Nested Interrupt Vector Table），加速中断处理过程，减少流水线阻塞时间。

## 第 7 章 存储系统

### 7.1 存储系统的层次结构

#### 3) 存储系统的层次

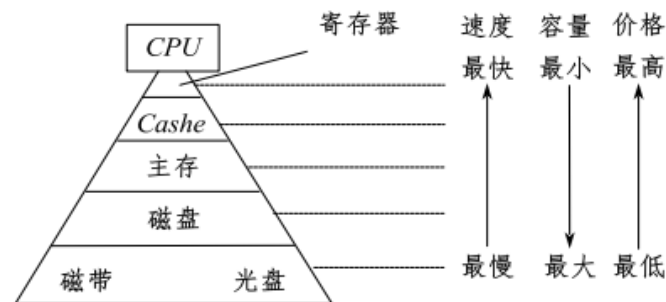


图1 多级存储器结构

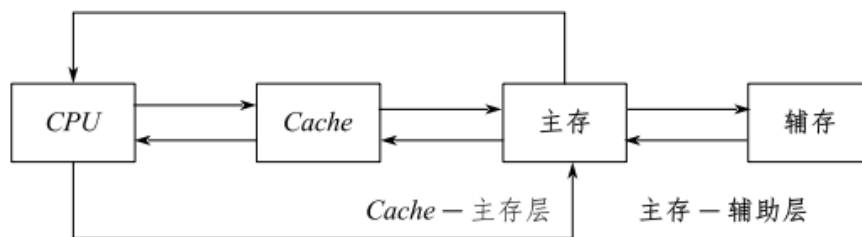


图2 三级存储系统的层次结构及其构成

注：①Cache - 主存层速度接近于Cache，容量和价格接近于主存；主存 - 辅存层速度接近于主存，容量和价格接近于辅存。

②主存与Cache之间的数据调动由硬件自动完成，对所有程序员是透明的（透明的意思是感受不到其存在）；主存与辅存之间的数据调动是由硬件和操作系统共同完成的，对应用程序员是透明的。

③Cache - 主存层与主存 - 辅存层中，上一层的内容只是下一层内容的副本。

### 7.2 高速缓冲存储器 cache

#### cache 工作原理

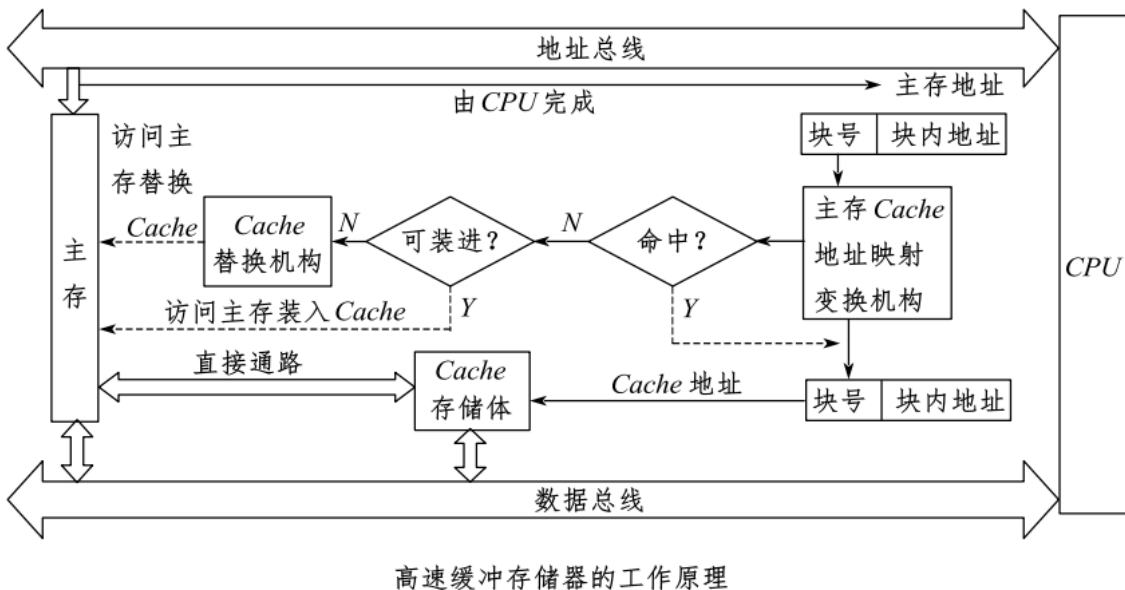
Cache — 主存层次结构通常来解决CPU和主存速度不匹配的问题,而支撑该方法的根本原理是:程序访问的局部性原理。

程序访问的局部原理:

(1) 时间局部性:最近的未来要用的信息,很可能是现在正在使用的信息。如:程序中的循环。

(2) 空间局部性:最近的未来要用的信息,很可能与现在正在使用的信息在存储空间上是邻近的。如:指令的顺序存放,数据的数组形式等。

### 1) Cache 的基本工作原理



相关概念:

(1) 块: 为了便于Cache和主存间交换信息,Cache和主存都被划分为相等的块,Cache块又称为Cache行,每个块由若干字节组成,块的长度称为块长(Cache行长)。

(2) 读操作: 当CPU发出读请求时,若访问地址在Cache中命中,就将此地址换成Cache地址,直接对Cache进行读操作,与主存无关;若Cache不命中,则仍需访问主存,并把此字所在的块一次性地从主存中调入Cache。若此时Cache已满,则需根据某种替换算法,用这个块替换Cache中原来的某块信息。整个过程全部由硬件实现。值得注意的是,CPU与Cache之间的数据交换以字为单位,而Cache与主存之间的数据交换则以Cache块为单位。

(3) 写操作: 当CPU发出写请求时,若Cache命中,有可能遇到Cache与主存中内容不一致的问题。例如,由于CPU写Cache,把Cache某单元中的内容从X修改成了X',而主存对应单元中的内容仍然是X,没有改变。所以若Cache命中,需要按照一定的写策略处理,常见的处理方法有全写法和回写法,详见本节的Cache写策略部分。

(4) 命中率: CPU欲访问的信息已在Cache中的比率称为Cache的命中率。设一个程序执行期间,Cache的总命中次数为 $N_c$ ,访问主存的总次数为 $N_m$ ,则命中率H为:

$$H = N_c / (N_c + N_m)$$

(5) Cache主存系统的平均访问时间:

设  $t_c$  为命中时的 Cache 访问时间,  $t_m$  为未命中时的主存访问时间,  $1-H$  表示未命中率, 则 Cache 主存系统的平均访问时间  $T_a$  为:

$$T_a = Ht_c + (1-H)t_m$$

根据 Cache 的读、写流程, 实现 Cache 时需解决以下关键问题:

- (1) 数据查找。如何快速判断数据是否在 Cache 中。
- (2) 地址映射。主存块如何存放在 Cache 中, 如何将主存地址转换为 Cache 地址。
- (3) 替换策略。Cache 满后, 使用何种策略对 Cache 块进行替换或淘汰。
- (4) 写入策略。如何既保证主存块和 Cache 块的数据一致性, 又尽量提升效率。

### 题: cache 命中率

10. 设某计算机有一个指令和数据合一的 cache, 已知 cache 的读写时间为 10ns, 主存的读写时间为 100ns, 取指的命中率为 98%, 数据的命中率为 95%, 在执行程序时, 约有 1/5 指令需要取一个数据, 并假设流水线从不阻塞, 问设置 cache 后, 与无 cache 比较, 计算机的运算速度可提高到多少倍?

注: 不命中时也要计入 cache 的时间

Cache: 10ns

RAM: 100ns

取指: 98% 命中, 100% 指令

取数: 95% 命中, 20% 指令

无 cache:  $100ns \times (100\% + 20\%)$   
 $= 120ns$

有 cache:

取指:  $100\% \times [10ns \times 100\% + 100ns \times 2\%]$   
 $= 1 \times [10 + 2]$   
 $= 12ns$

取数:  $20\% \times [10ns \times 100\% + 100ns \times 5\%]$   
 $= 0.2 \times [10 + 5]$   
 $= 3ns$

Sum =  $12 + 3 = 15ns$

倍率:  $120 \div 15 = 8$

3. (6分) cache的命中率与哪些因素有关? 分别阐述之。

答: cache的命中率与cache的容量、块的大小、地址映像方式和替换算法有关。(2分)

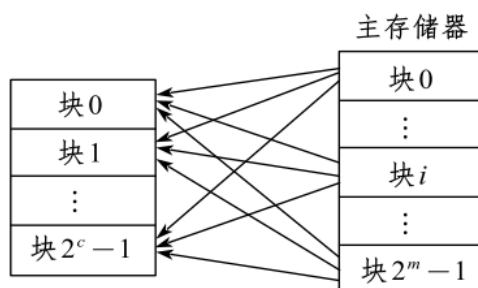
在其他因素不变的情况下, cache的容量大, 命中率高; 块的大小要适中, 其对命中率的影响与执行的程序有关; 在地址映像方面, 全相联的命中率最高, 但因所需硬件太多, 一般不采用, 直接映像命中率最低, 但是简单, 比较理想的是组相联; 在替换算法方面, LRU替换算法高于FIFO或随机替换算法, 一般采用修正后的LRU算法。(4分)

## cache 地址映像

### 2) Cache 和主存的映射方式

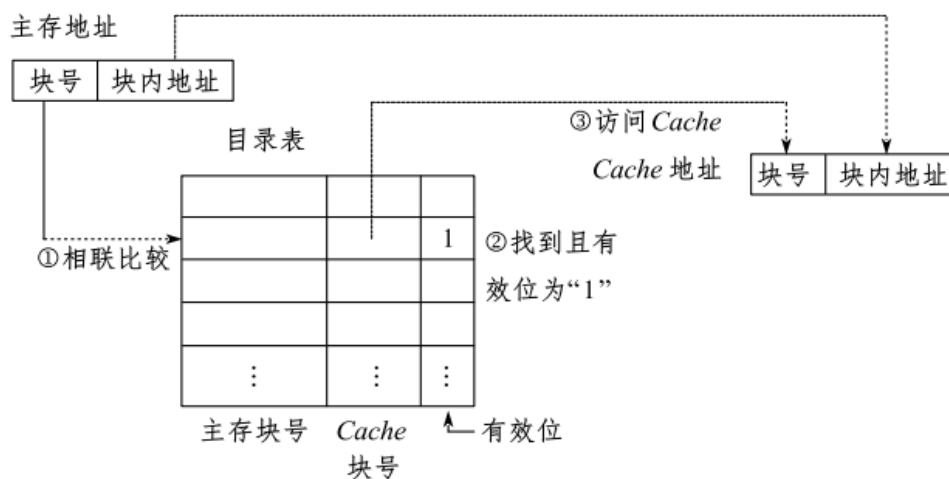
#### (1) 全相联映射及其地址变换

全相联地址映射是指主存中的每一块都可以映射到Cache中的任意块, 如下图所示。这种映射方法是最灵活的, 也是Cache利用率最高的一种方式, 但同时也是成本最高的一种方式。



全相联映射方式

在全相联映射方式下, 主存地址被分为两个部分: 高 $m$ 位表示主存块地址, 低 $b$ 位表示块内地址。同样, Cache的地址也分为两个部分: 高 $c$ 位表示Cache块地址, 低 $b$ 位表示块内地址。通常采用目录表记录主存块之间的映射关系, 并将目录表存放在一个相联存储器中。目录表中的每个存储字主要包括三个部分: 主存块号、Cache块号和有效位。有效位表示目录表中主存块号和Cache块号建立的映射关系是否有效。目录表共有 $2^c$ 个存储字, 即Cache中每个块对应一个存储字。



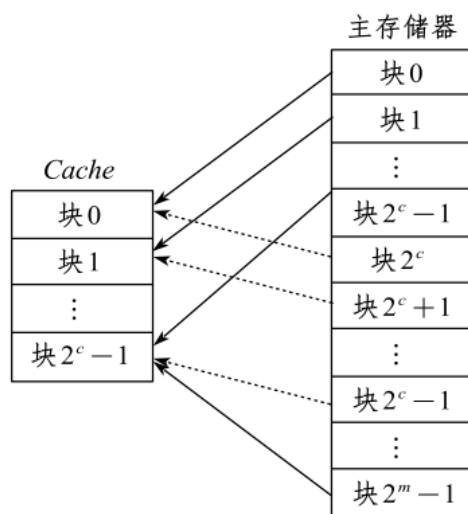
全相联映射的地址变换

## (2) 直接映及其地址变换

直接地址映射是指主存中的块只能映射到Cache中某个固定的块中，主存和Cache块的对应关系可用如下公式表示：

$$j = i \bmod 2^c$$

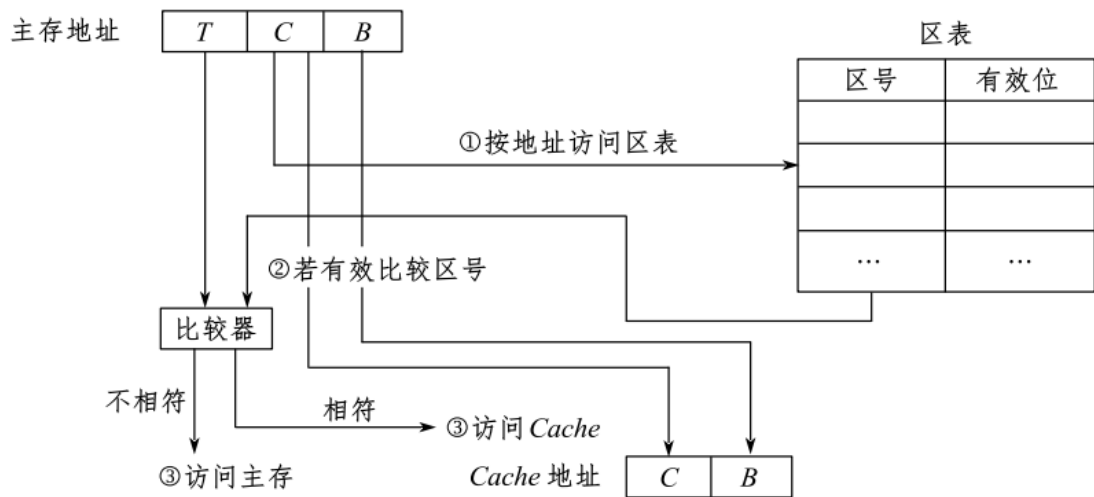
其中， $j$ 为数据在Cache中的块号， $i$ 为数据在主存中的块号。在这种映射方式中，主存的第0块，第 $2^c$ 块，…只能映射到Cache的第0块，而主存的第1块，第 $2^c + 1$ 块，…只能映射到Cache的第1块，以此类推。如下图所示。



直接地址映射方式

在直接地址映射方式下，主存地址由三部分组成：区号（ $t$ 位）、区内块号（ $c$ 位）和块内地址（ $b$ 位），其中， $m = t + c$ 。通常用区表来保存主存块与Cache块的映射关系。区表中的每个存储字主要包括两个部分：主存区号和有效位。有效位表示区表中的主存块是否已经装入Cache中。区表中共有 $2^c$ 个存储字。区表通常存放在一个小容量高速存储器中，按地址进行访问。

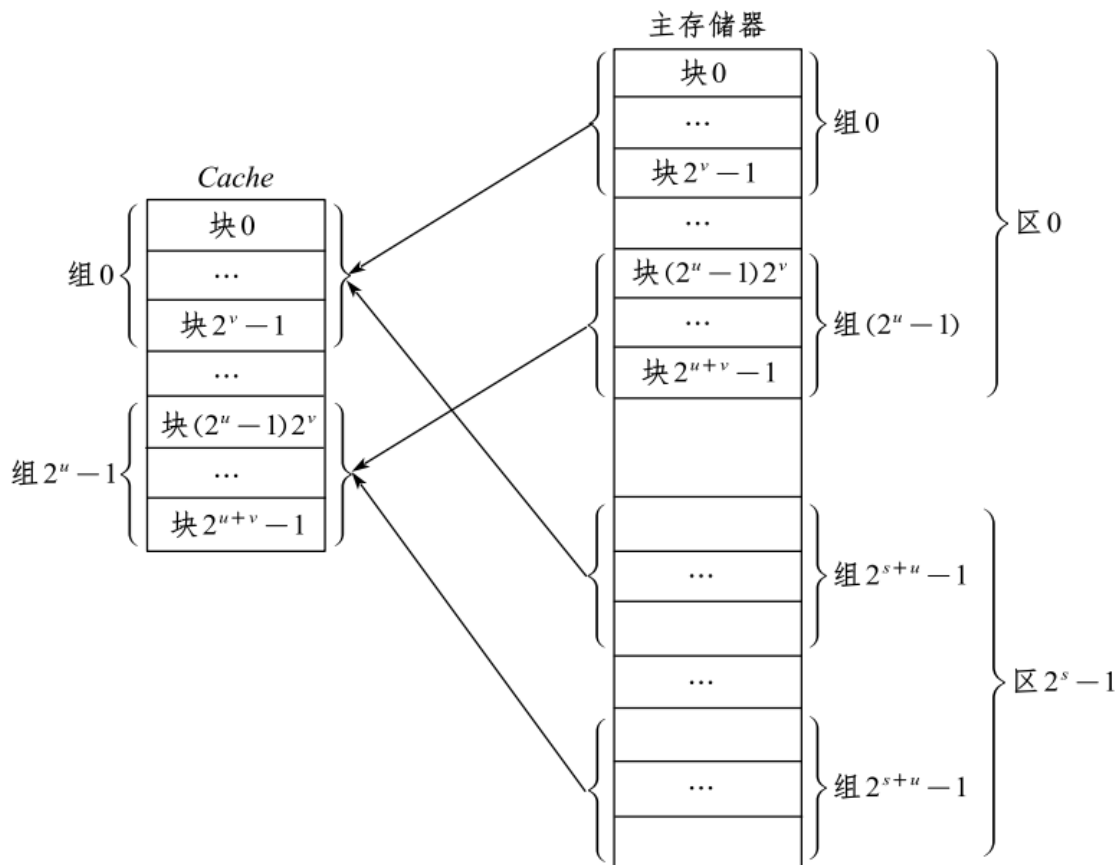




直接地址变换方法

### (3) 组相联映射及其地址变换

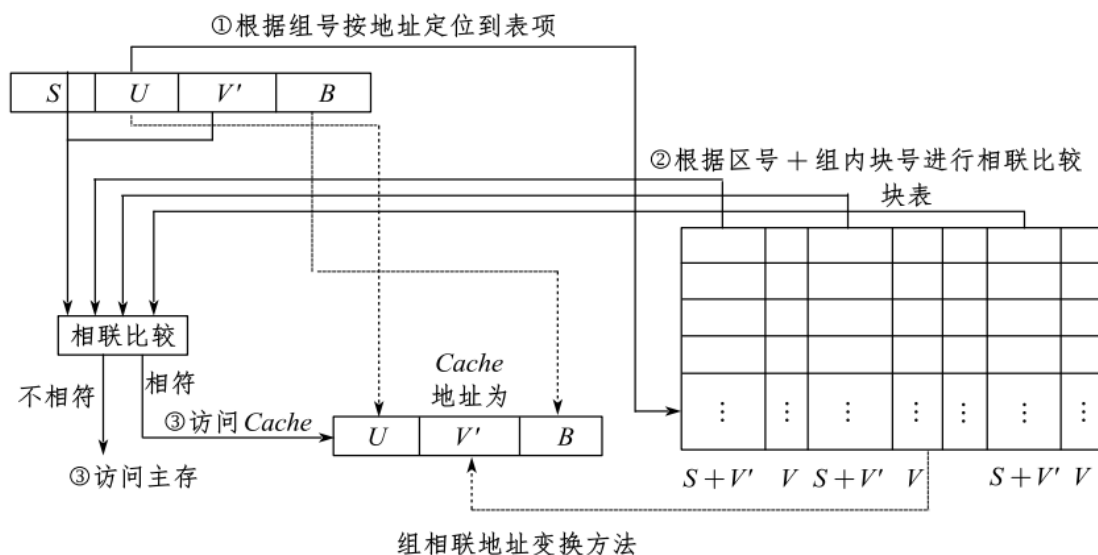
组相联映射 = 全相联映射 + 直接相联映射。如下图所示，主存和 Cache 的块先进行分组，（主存和 Cache 每组块数相同），在地址映射时，组间为直接相联映射，组内为全相联映射。



组相联映射方式

在上图中，Cache 被分为  $2^u$  组，每组  $2^v$  块；主存中共有  $2^s$  个区，每个区有  $2^u$  个组，即共有  $2^{s+u}$  个组。主存中的每个区的第  $i$  组都只能映射到 Cache 中的第  $i$  组，在组内块采用全相联方式，即每个块都可以映射到 Cache 的第  $i$  组的任一块。

CPU 发出的访存地址被分解为：区号  $S$ ，组号  $U$ ，组内块号  $V'$  和块内地址  $B$  四个部分。而 Cache 的地址可分解为：组号  $U$ ，组内块号  $V$  和块内地址  $B$  三个部分。下图给出了组相联地址变换方式。



注： $n$  路组相联是指：每组有  $n$  块。

#### (4) 三种映射方式的对比

三种映射方式中，直接映射的每个主存块只能映射到 Cache 中的某一固定行；全相联映射可以映射到所有 Cache 行； $N$  路组相联映射可以映射到  $N$  行。当 Cache 大小、主存块大小一定时，

①直接映射的命中率最低，全相联映射的命中率最高。

②直接映射的判断开销最小、所需时间最短，全相联映射的判断开销最大、所需时间最长。

③直接映射标记所占的额外空间开销最少，全相联映射标记所占的额外空间开销最大。

### 题：地址映像

注：存储块数 != 块数

区号：RAM地址位数 - cache地址位数

组号：cache总存储块数 / 块数

(块数：几路组相联就是几块数，一般为4；cache总存储块数 = cache总容量 / 一存储块的容量)

块号： $\log(\text{块数})$ ，一般为2

块内地址： $\log(\text{块的字数})$

六、(8分) 有一个cache的容量为2K字，每块为16字，问：

- (1) 该cache可容纳多少个块？(1分)
- (2) 如果主存的容量是256K字，则有多少个块？(1分)
- (3) 主存的地址有多少位？cache的地址有多少位？(2分)
- (4) 在直接映射方式下，主存中的第*i*块映射到cache中哪一个块？(1分)
- (5) 进行地址映射时，存储器地址分成哪几段？各段分别多少位？(3分)

解：(1) cache中有 $2048/16=128$ 个块。

(2) 主存有 $256K/16=214=16384$ 个块。

(3) 主存容量为 $256K=2^{18}$ 字，所以主存的地址有18位。

cache容量为 $2K=2^{11}$ 字，所以cache字地址为11位。

(4) 主存中的第*i*块映像到cache中第  $i \bmod 128$  个块中。

(5) 存储器的字地址分成三段：区地址、组地址、块内字地址。

区地址的长度为 $18-11=7$ 位，组地址为7位，块内字地址为4位。

### cache 替换算法

(2) 先进先出算法：选择最早调入的行进行替换。它比较容易实现，但也未依据程序访问的局部性原理，因为最早进入的主存块也可能是目前经常要用的。

(3) 近期最少使用算法 (LRU)：依据程序访问的局部性原理，选择近期内长久未访问过的Cache行作为替换的行，平均命中率要比FIFO的高，是堆栈类算法。

### 题：替换算法

m表示未命中，h表示命中

请求的页面从下往上顶掉最上方的页面，当请求的页面原本就在缓存中时视为命中。

LRU：命中时仍然执行不命中时的操作（请求的页面从下往上顶掉最上方的页面）

LRU：命中时把命中的块放到最下方，其他块顺次上移

FIFO：命中时不执行操作

28. 某程序对逻辑页面访问的序列为  $P_3P_4P_2P_6P_4P_3P_7P_4P_3P_6P_3P_4P_8P_4P_6$ 。(1)设主存容量为 3 个页面,求执行 FIFO 和 LRU 页面替换算法时各自的命中率(假设开始时主存为空)。(2)当主存容量增加到 4 个页面时,两种替换算法各自的命中率又是多少?

答:表 7.10 和表 7.11 分别表示主存为 3 个和 4 个页面时的调页情况,表中每一列表示页面淘汰次序,编号最大者(表 7.10 中的③和表 7.11 中的④)表示将被调出的页面。

(1) 主存页面为 3 时的调页情况(表 7.10)。

表 7.10 主存页面为 3 时的调页情况

页面请求		3	4	2	6	4	3	7	4	3	6	3	4	8	4	6
LRU	③	3	3	3	4	2	6	4	3	7	4	4	6	3	3	8
	②	/	4	4	2	6	4	3	7	4	3	6	3	4	8	4
	①	/	/	2	6	4	3	7	4	3	6	3	4	8	4	6
	命中	m	m	m	m	h	m	m	h	h	m	h	h	m	h	m
FIFO	③	3	3	3	4	4	2	6	3	3	7	4	4	6	3	8
	②	/	4	4	2	2	6	3	7	7	4	6	6	3	8	4
	①	/	/	2	6	6	3	7	4	4	6	3	3	8	4	6
	命中	m	m	m	m	h	m	m	m	h	m	m	h	m	m	m

注:采用 LRU 算法的命中率为  $6 \div 15 = 40\%$ ,采用 FIFO 算法的命中率为  $3 \div 15 = 20\%$ 。

(2) 主存页面为 4 时的调页情况(表 7.11)。

表 7.11 主存页面为 4 时的调页情况

页面请求		3	4	2	6	4	3	7	4	3	6	3	4	8	4	6
LRU	④	3	3	3	3	3	2	6	6	6	7	7	7	6	6	3
	③	/	4	4	4	2	6	4	3	7	4	4	6	3	3	8
	②	/	/	2	2	6	4	3	7	4	3	6	3	4	8	4
	①	/	/	/	6	4	3	7	4	3	6	3	4	8	4	6
	命中	m	m	m	m	h	h	m	h	h	h	h	h	m	h	h
FIFO	④	3	3	3	3	3	3	4	4	2	2	2	6	7	7	3
	③	/	4	4	4	4	4	2	2	6	6	6	7	3	3	4
	②	/	/	2	2	2	2	6	6	7	7	7	3	4	4	8
	①	/	/	/	6	6	6	7	7	3	3	3	4	8	8	6
	命中	m	m	m	m	h	h	m	h	m	h	h	m	m	h	m

注:采用 LRU 算法的命中率为  $9 \div 15 = 60\%$ ,采用 FIFO 算法的命中率为  $6 \div 15 = 40\%$ 。