1.数制与运算

1.1 数制

有符号数

- (单)符号位：“0”表示“正”，“1”表示“负”，固定为编码的最高位
- 双符号位： 00 正号，11 负号，01 上溢，10 下溢

10进制转换

【例1】求$(62.625)_{10}=(\ ?\ )_2$

先转换整数部分

2 | 62 ··· 余数 =0 = k_0 (LSB)
2 | 31 ··· 余数 =1 = k_1
2 | 15 ··· 余数 =1 = k_2
2 | 7 ··· 余数 =1 = k_3
2 | 3 ··· 余数 =1 = k_4
2 | 1 ··· 余数 =1 = k_5 (MSB)
0

再转换小数部分

.625
× 2
1 .250 进位“1” = k_{-1} （MSB）
× 2
0 .50 进位“0” = k_{-2}
× 2
1 .0 进位“1” = k_{-3} （LSB）

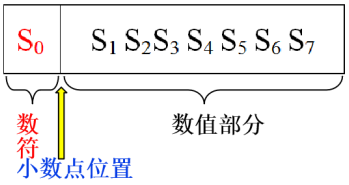
小数部分为0

$(62.625)_{10}=(111110.101)_2$

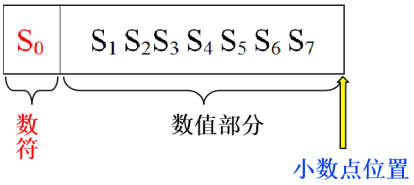
$(k_5k_4k_3k_2k_1k_0.\ k_{-1}k_{-2}k_{-3})_2$

定点数-与浮点数相对应

➢定点小数
如：01100000
是十进制的0.75



➢定点整数
如：01100000
是十进制的96



补码及补码运算

$$[x]_{原}=\begin{cases}x & 0\le x\le 2^{n-1}-1\\ 2^{n-1}-x & -(2^{n-1}-1)\le x\le 0\end{cases}$$

$$[x]_{反}=\begin{cases}x & 0\le x\le 2^{n-1}-1\\ (2^{n}-1)+x & -(2^{n-1}-1)\le x\le 0\end{cases}$$

$$[x]_{补}=\begin{cases}x & 0\le x\le 2^{n-1}-1\\ 2^{n}+x & -2^{n-1}\le x\le 0\end{cases}$$

$$[x]_{移}=2^{n-1}+x \qquad -2^{n-1}\le x\le 2^{n-1}-1$$

十进制数值	原码	反码	补码
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111
-0	1000	1111	0000
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8			1000

运算规则：补码的**符号位**和数值一起**参加运算**。若符号位产生了进位，则将进位舍弃

[X+Y]补= [X]补+ [Y]补，[X-Y]补=[X]补+[-Y]补

【例3】 X=+1101， Y=+0101，求$[X+Y]_{补}$

解：由于$X=(+1101)_2=(+13)_{10}$， $Y=(+0101)_2==(+5)_{10}$，13+5=18，所以必须用有效数字为5位的二进制数才能表示和，再加上1位符号位，则采用6位的二进制补码进行运算。

$[X]_{补}$=0 01101， $[Y]_{补}$=0 00101， $[-Y]_{补}$=1 11011，

$[X]_{补}+[Y]_{补}$=0 01101 + 0 00101 =0 10010

得 $[X+Y]_{补}=[X]_{补}+[Y]_{补}$= 0 10010

符号位为“0”，说明和为正数，

则$X+Y=(+10010)_2=(+18)_{10}$

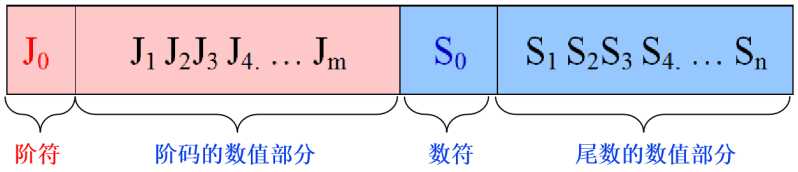
浮点数

两种表示方法

阶码+尾数

- 阶码
  - 移码表示：只有数值部分，实际阶数为**阶码数值部分-偏移量**

- 原码/补码表示，有1位**阶符**做符号位

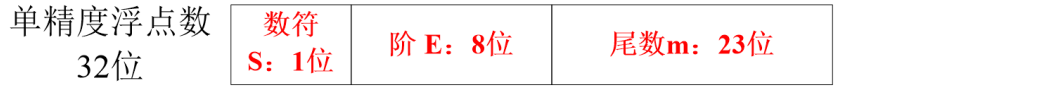


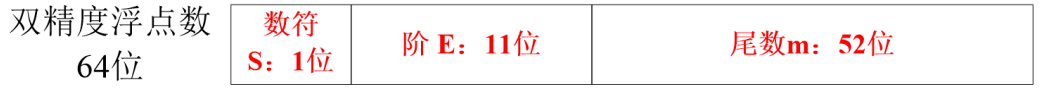
- 尾数：原码/补码表示，有1位**数符**做符号位

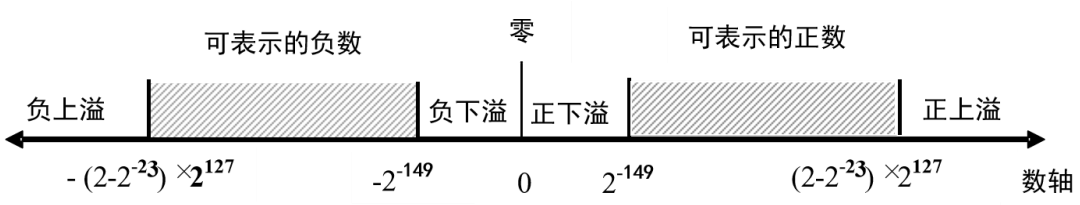
IEEE 754的表示方法

- 数符(**S**ign)：1位，0表示正数，1表示负数
- 阶码(**E**xponent)：**移码**表示，n 位阶码偏移量为 $2^{n-1}-1$。如8位阶码偏移量为 7FH（即127），11位阶码偏移量3FFH（即1023）
- 尾数(**M**antissa)：**原码**表示。**规格化**成小数点左侧一定为1，并且小数点前面这个**1作为隐含位被省略。**这样单精度浮点数尾数实际上为24位，即采用IEEE 754规格化标准增加了表示的精度。

  M=1.m





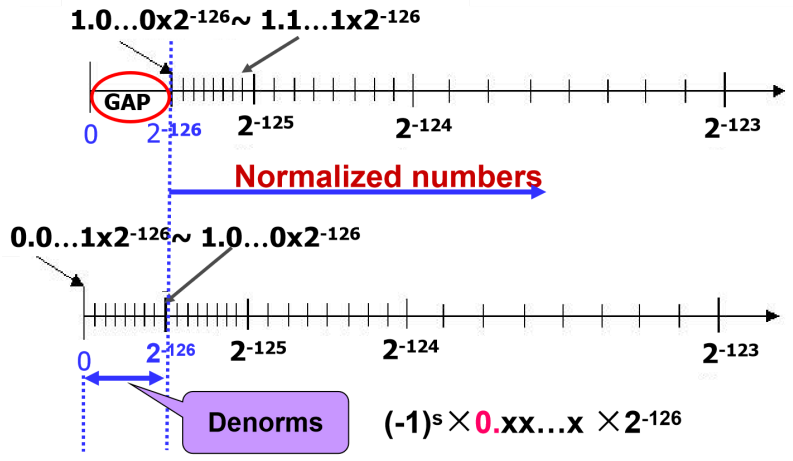


规格化与非规格化

规格化——增加浮点数精度

非规格化——提高了在0附近的精度

E=0 M≠0时，表示非规格化浮点数，非规格化浮点数尾数部分**小数点左侧为0**



特殊值

E	M	浮点数 N
$1 \le E \le 254$	$M \neq 0$	表示规范浮点数 $N=(-1)^S \times 1.m \times 2^{(E-127)}$
$E=0$	$M=0$	表示 $N=0$
$E=0$	$M \neq 0$	表示非规范浮点数 $N=(-1)^S \times 0.m \times 2^{-126}$
$E=255$	$M=0$	表示无穷大，由符号位 S 确定是正无穷大还是负无穷大
$E=255$	$M \neq 0$	NaN（Not a Number） 不是一个数

例

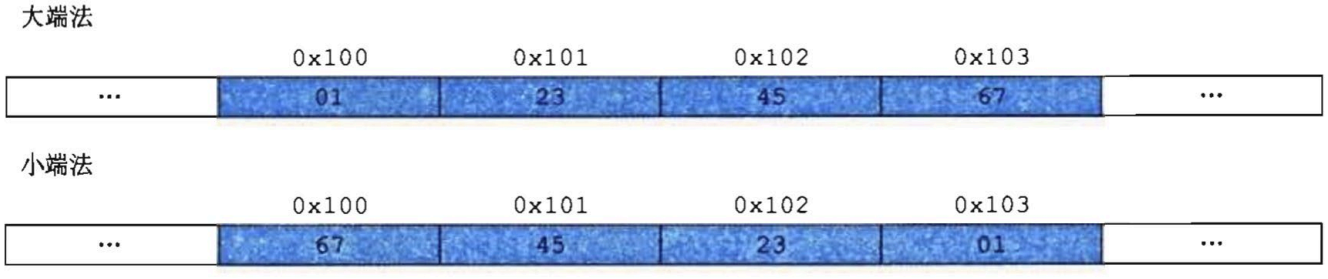
$$(178.125)_{10}=(10110010.001)_2$$
$$=1.0110010001\times 2^{111}$$
$$S=0$$
$$E=00000111+01111111$$
$$=10000110$$
$$m=01100100010000000000000$$

$$(-0.0449219)_{10}=(-0.0000101110)_2$$
$$=-1.01110\times 2^{-101}$$
$$S=1$$
$$E=-00000101+01111111$$
$$=01111010$$
$$m=01110000000000000000000$$

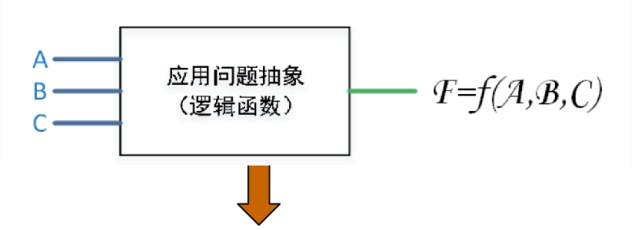
大小端

- 大端机器：MIPS, IBM 360/370, Motorola 68k, Sparc, HP PA
  多字节数据**高位字节存储在低地址处**，**低位字节存储在高地址处**。也就是说，数据的“**大头**”放在前面。
- 小端机器：Intel 80x86, DEC VAX
  多字节数据的**低位字节存储在低地址处**，**高位字节存储在高地址处**。也就是说，数据的“**小头**”放在前面。

以32位的无符号整数0x01234567为例



1.2 布尔代数



逻辑函数的标准表达式（由真值表到表达式）：

- 最小项表达式：最小项构成的与或式 $F(A,B,C)=\bar{A}\bar{B}C+\bar{A}B\bar{C}+A\bar{B}\bar{C}+ABC$
- 最大项表达式：最大项构成的或与式 $F(A,B,C)=(A+B+C)\cdot(A+\bar{B}+C)\cdot(\bar{A}+\bar{B}+\bar{C})$

逻辑函数表达式的化简：

- 代数化简：利用公理、定理和规则进行化简
- 卡诺图化简：利用卡诺图进行化简（本课程未涉及）

预备知识

基本公理和基本定理

基本公理	乘法运算公式	加法运算公式
0-1 律	$1\bullet A=A$　　$0\bullet A=0$	$0+A=A$　　$1+A=1$
交换律	$A\bullet B=B\bullet A$	$A+B=B+A$
结合律	$A\cdot(B\cdot C)=(A\cdot B)\cdot C$	$A+(B+C)=(A+B)+C$
分配律	$A\cdot(B+C)=A\cdot B+A\cdot C$	$A+(B\cdot C)=(A+B)\cdot(A+C)$
互补律	$A\cdot\bar{A}=0$	$A+\bar{A}=1$

基本定理	乘法运算公式	加法运算公式
重叠律	$A\cdot A=A$	$A+A=A$
还原律	$\bar{\bar{A}}=A$	
反演律 摩根定理	$\overline{A\cdot B}=\bar{A}+\bar{B}$	$\overline{A+B}=\bar{A}\cdot\bar{B}$
吸收律 1	$A(A+B)=A$	$A+AB=A$
吸收律 2	$A(\bar{A}+B)=AB$	$A+\bar{A}B=A+B$
吸收律 3	$(A+B)(A+\bar{B})=A$	$AB+A\bar{B}=A$
包含律	$(A+B)(\bar{A}+C)(B+C)=(A+B)(\bar{A}+C)$ $AB+\bar{A}C+BC=AB+\bar{A}C$	

三个基本规则

代入规则

- 将逻辑等式中的某一变量代以另一函数其等式仍然成立
- 用途：扩大基本公式和常用公式的使用范围

反演规则

- 对原函数求反函数的过程叫做反演。将原函数F中的全部“•”换成“+”，“+”换成“•”，“0”换成“1”，“1”换成“0”，原变量换成反变量，反变量换成原变量，所得到的新函数就是原函数的反函数
- 用途：直接求已知逻辑函数的反函数，可用于**公式的化简**

【例2】 试化简函数：$F=(A+B)(\overline{A}+C)(B+C+D)$

$$
\begin{aligned}
\text{解：}\quad \overline{F} &= \overline{A}\,\overline{B}+A\overline{C}+\overline{B}\,\overline{C}\,\overline{D}\\
&= \overline{A}\,\overline{B}+A\overline{C}+(A+\overline{A})\overline{B}\,\overline{C}\,\overline{D}\\
&= \overline{A}\,\overline{B}+\overline{A}\overline{B}\overline{C}\,\overline{D}+A\overline{C}+A\overline{B}\overline{C}\,\overline{D}\\
&= \overline{A}\,\overline{B}+A\overline{C}\\
\text{则：}\quad F &= \overline{\overline{F}}=(A+B)(\overline{A}+C)
\end{aligned}
$$

已知 $F_1=AB+\overline{(C+\overline{D})}B+\overline{BC}+0$ 则反函数 $\overline{F_1}=(\overline{A}+\overline{B})\cdot\overline{\overline{C}D}+\overline{B}\cdot\overline{\overline{B}+\overline{C}}\cdot 1$

对偶规则

- 将原函数F中的全部“•”换成“+”，“+”换成“•”，“0”换成“1”，“1”换成“0”，所得的新函数就是原函数的对偶式

例如：函数 $F_1=AB+\overline{(C+\overline{D})B}+\overline{BC}+0$

则对偶式是：$F_1'=(A+B)\cdot\overline{C\,\overline{D}+B}\cdot\overline{B+C}\cdot 1$

又如：$F_2=A+\overline{B+\overline{C}\cdot\overline{D+\overline{E}}}$

则对偶式是：$F_2'=A\cdot\overline{B\cdot(\overline{C}+\overline{D\cdot\overline{E}})}$

规则

1. 遵循“（）”→“•”→“+”的运算优先顺序；
2. **不属于单个变量上的“非号”在变换中不变。**

逻辑函数符号

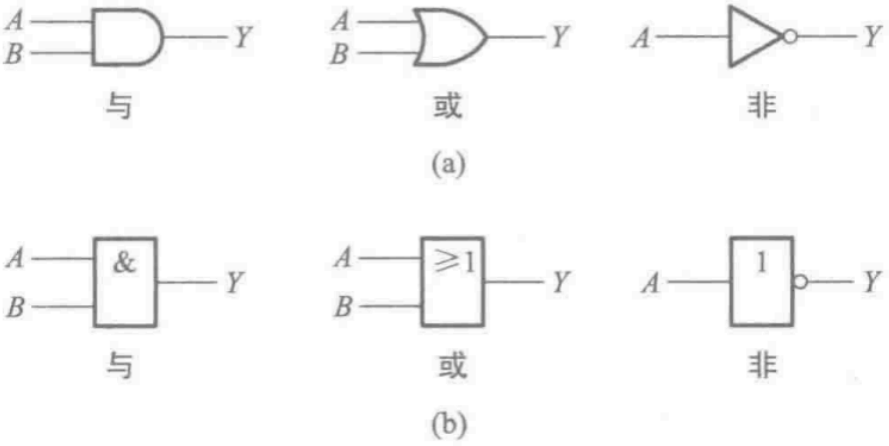
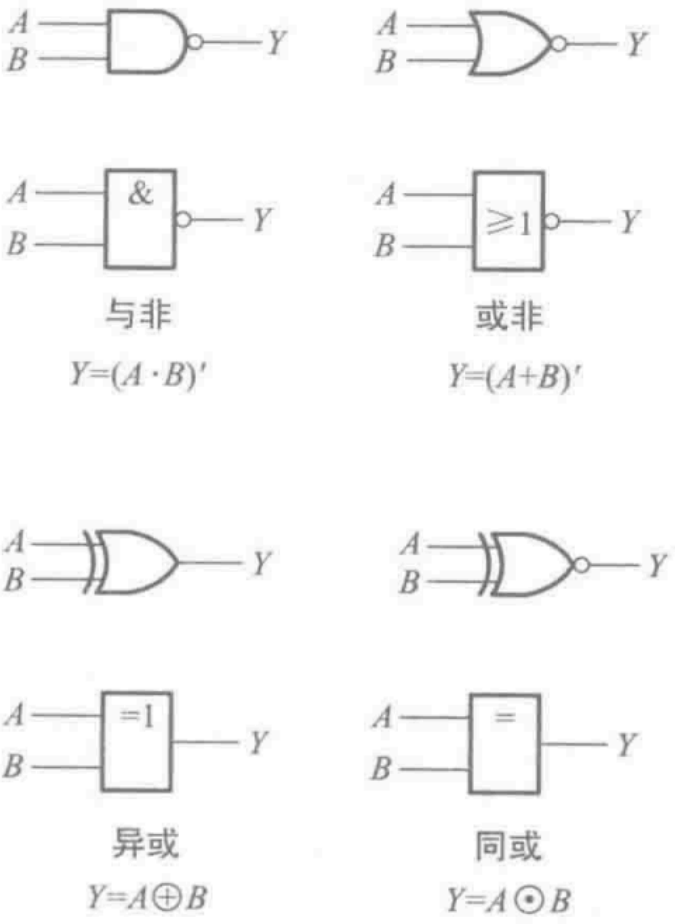


图 2.2.2　与、或、非的图形符号

（a）特定外形符号　（b）矩形轮廓符号



最小项与或式

最小项表达式：全部由最小项构成的**与或式**（积之和式），最小项记做 m_i

- 全体最小项之和为1；

- 任意两个最小项的乘积为0；

最小项编号

最小项	取值	编号表示
$\overline{A}\,\overline{B}\,\overline{C}$	000	m_0
$\overline{A}\,\overline{B}C$	001	m_1
$\overline{A}B\,\overline{C}$	010	m_2
$\overline{A}BC$	011	m_3
$A\,\overline{B}\,\overline{C}$	100	m_4
$A\,\overline{B}C$	101	m_5
$AB\,\overline{C}$	110	m_6
ABC	111	m_7

$$F(A,\ B,\ C) = \overline{A}\,\overline{B}C + \overline{A}B\,\overline{C} + A\,\overline{B}\,\overline{C} + ABC$$ 最小项表达式

$$F(A,\ B,\ C) = (A + B + C)\cdot(A + \overline{B} + C)\cdot(\overline{A} + \overline{B} + \overline{C})$$ 最大项表达式

最小项推导法

使**输出为1**的输入组合写成**乘积**项的形式，其中取值为**1**的输入用**原变量**表示，取值为**0**的输入用**反变量**表示，然后把这些**乘积项加起来。**

❖ 例：三人表决器设计（表决原则：少数服从多数）

- A，B，C表示输入，1表示赞成，0表示反对
- F表示输出，1表示通过，0表示不通过

$$F = \overline{A}BC + A\,\overline{B}C + AB\,\overline{C} + ABC$$
$$F(A,B,C) = m_3 + m_5 + m_6 + m_7$$
$$F(A,B,C) = \sum m(3,5,6,7)$$

最小项表达式

真值表

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

补充：最大项表达式

最大项表达式：全部由最大项构成的**或与式**（和之积式），最小项记做 M_i

- 全体最大项之积为0；

- 任意两个最大项之和为1。

最大项编号

最大项	取值	编号表示
$\overline{A}+\overline{B}+\overline{C}$	111	M_7
$\overline{A}+\overline{B}+C$	110	M_6
$\overline{A}+B+\overline{C}$	101	M_5
$\overline{A}+B+C$	100	M_4
$A+\overline{B}+\overline{C}$	011	M_3
$A+\overline{B}+C$	010	M_2
$A+B+\overline{C}$	001	M_1
$A+B+C$	000	M_0

最大项推导法

把使**输出为0**的输入组合写成**和**项的形式，其中取值为**0**的输入用**原变量**表示，取值为**1**的输入用**反变量**表示，然后把这些和项**乘**起来。

【例】：三人表决器设计的输出表达式

$$F=(A+B+C)(A+B+\overline{C})(A+\overline{B}+C)(\overline{A}+B+C)$$

$$F(A,B,C)=M_0\cdot M_1\cdot M_2\cdot M_4$$

$$F(A,B,C)=\Pi M(0,1,2,4)$$

真值表

A B C	F
0 0 0	**0**
0 0 1	**0**
0 1 0	**0**
0 1 1	1
1 0 0	**0**
1 0 1	1
1 1 0	1
1 1 1	1

逻辑代数的化简——化简题（5分简答）

最简或与表达式：或项个数最少（或门用的最少）；在满足以上条件下，或项中变量数最少（或门的输入端最少）。

化简方法

1. 利用对偶规则，将“或与”表达式**转换为 “与或”表达式。**
2. 实际化简“**与或**”表达式。
3. 利用对偶规则将最简“与或” 表达式**转为最简“或与” 表达式。**

与或表达式的化简

- **合并乘积项法**（相邻最小项，利用互补律消去1个变量）

  化简 $F = A(BC + \overline{B}\,\overline{C}) + AB\overline{C} + A\overline{B}C$

  解：$F = ABC + A\overline{B}\,\overline{C} + AB\overline{C} + A\overline{B}C$ 利用分配律展开

  $= (ABC + A\overline{B}C) + (A\overline{B}\,\overline{C} + AB\overline{C})$ 合并最小项

  $= AC(B + \overline{B}) + A\overline{C}(\overline{B} + B)$

  $= AC + A\overline{C}$ 互补律

  $= A(C + \overline{C}) = A$ 互补律

- **吸收项法**（3个吸收律，利用吸收律减少“与”项）

  化简 $F = A\overline{B} + \overline{A}B + ABCD + \overline{A}\,\overline{B}CD$

  解：$F = (A\overline{B} + \overline{A}B) + (AB + \overline{A}\,\overline{B})CD$

  $= (A\overline{B} + \overline{A}B) + \overline{A\overline{B} + \overline{A}\,\overline{B}} \cdot CD$

  $= A\overline{B} + \overline{A}B + CD$

  由吸收律2 $A+\overline{A}B=A+B$

- **配项法**（互补律、重叠率）

  化简 $F = AB + \overline{A}\,\overline{B}C + BC$

  解：$F = AB + \overline{A}\,\overline{B}C + BC(A + \overline{A})$

  $= AB + \overline{A}\,\overline{B}C + ABC + \overline{A}BC$

  $= (AB + ABC) + (\overline{A}\,\overline{B}C + \overline{A}BC)$

  $= AB(1 + C) + \overline{A}C(B + \overline{B})$

  $= AB + \overline{A}C$

- **消除冗余项法**（包含律，利用包含律减少“与”项）

1.3 码制

- **8421码**
- **余3码**：8421码+3，如0是0011，1是0100；n和9-n互为反码
- **2421码**
  1. 理解1：2+4+2+1
  2. 理解2：依旧8+4+2+1，但0-4不变，5-9变成原数加6

十进制数	8421码	余3码	2421码
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0010
3	0011	0110	0011
4	0100	0111	0100
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101

十进制数	8421码	余3码	2421码
8	1000	1011	1110
9	1001	1100	1111

2.数字逻辑

2.1 组合逻辑

半加器 全加器 ALU

加法器

半加器：对两个1位二进制数进行相加求和，并向高位进位的逻辑电路，**不考虑来自低位的进位**

全加器：对两个1位二进制数进行相加求和，**考虑来自低位的进位**，并向高位进位的逻辑电路

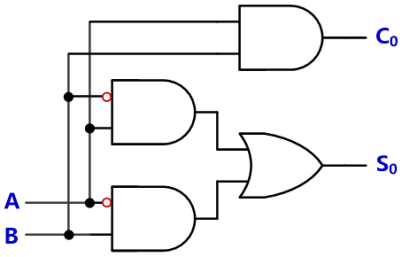
半加器

- 输入：两个1位二进制数， A、B
- 输出：两个，和S_0，进位C_0

真值表

A	B	S_O	C_O
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

逻辑电路图

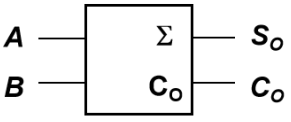


逻辑函数

$$C_o = A \cdot B$$

$$S_o = \overline{A}B + A\overline{B} = A \oplus B$$

符号表示



全加器

- 输入：3个，参加运算的二进制数 A、B，及低位进位C_i
- 输出：两个，和S_0，进位C_o

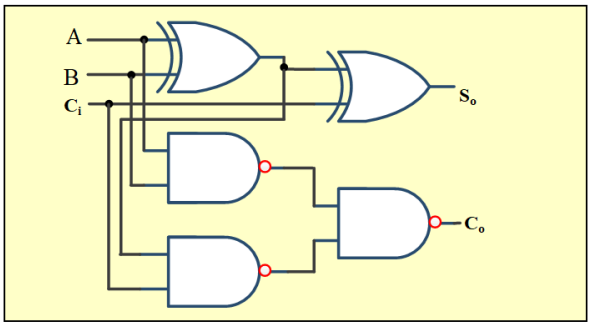
真值表

$A\ B\ C_I$	S_O	C_O
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

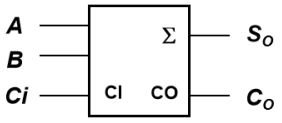
逻辑函数

$$S_o = \overline{A}\,\overline{B}C_i + \overline{A}B\overline{C}_i + A\overline{B}\,\overline{C}_i + ABC_i$$
$$= A \oplus B \oplus C_i$$
$$C_o = \overline{A}BC_i + A\overline{B}C_i + AB\overline{C}_i + ABC_i$$
$$= AB + C_i(A \oplus B)$$

逻辑电路图



符号表示

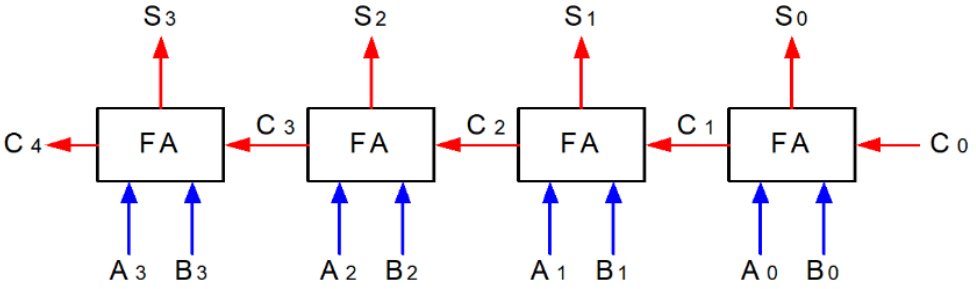


串行进位的多位加法器

1. 特点
   1. 进位串行传递
   2. 进位延时较长
2. 逻辑函数

$$C_1 = A_0B_0 + C_0(A_0 \oplus B_0)$$
$$C_2 = A_1B_1 + C_1(A_1 \oplus B_1)$$
$$C_3 = A_2B_2 + C_2(A_2 \oplus B_2)$$
$$C_4 = A_3B_3 + C_3(A_3 \oplus B_3)$$

3. 逻辑电路



并行进位的多位加法器

1. 特点
   1. 同时产生进位
   2. 加法延时缩短
   3. 实现相对复杂
2. 逻辑函数（代入C~0~到C~1~，代入C~1~、C~0~到C~2~……）

$$G_i = A_iB_i,\ P_i = A_i \otimes B_i$$

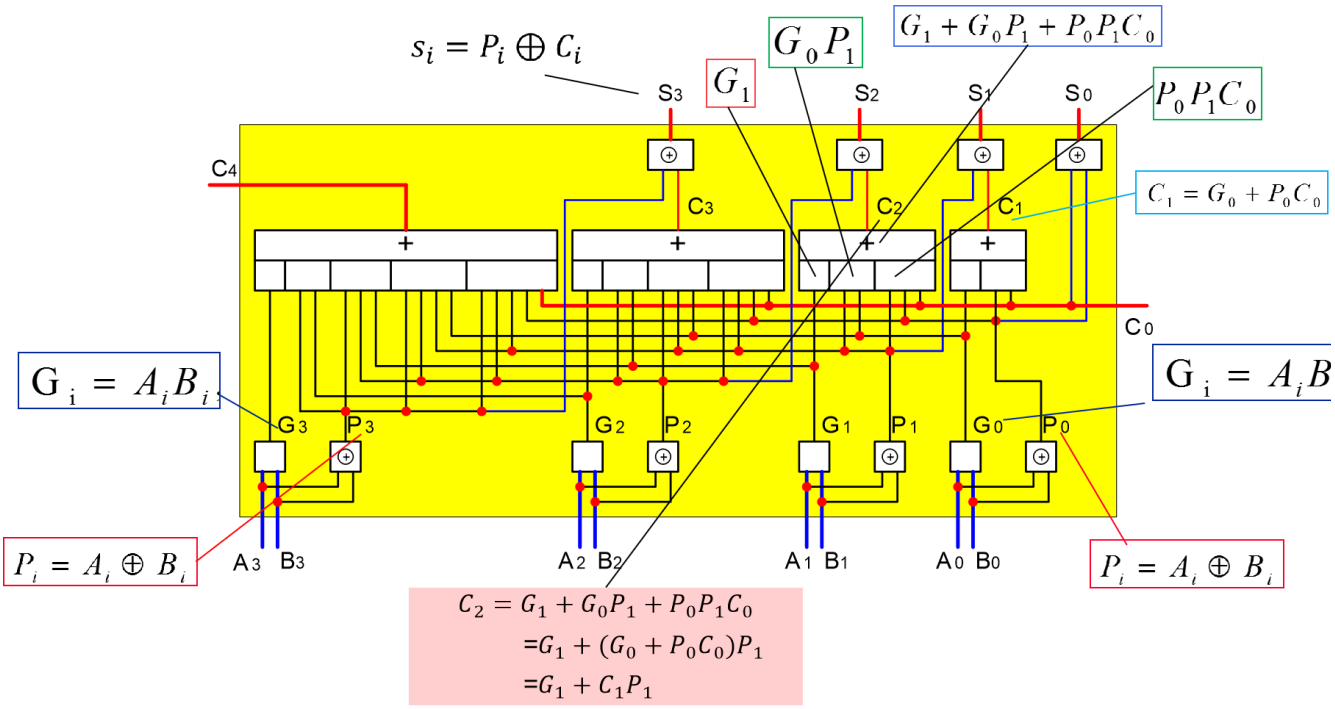
令 $G_i = A_i B_i$，$P_i = A_i \oplus B_i$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + C_1 P_1 = G_1 + G_0 P_1 + P_0 P_1 C_0$$

$$C_3 = G_2 + C_2 P_2 = G_2 + G_1 P_2 + G_0 P_1 P_2 + P_0 P_1 P_2 C_0$$

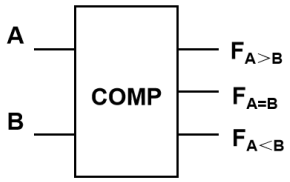
$$C_4 = G_3 + C_3 P_3 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + P_0 P_1 P_2 P_3 C_0$$



比较器

一种关系运算电路，它可以对两个二进制数进行比较，得出大于、小于和相等的结果。

❖ **1位比较器**



1位比较器真值表

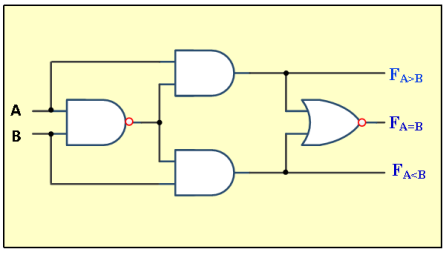
A B	$F_{A>B}$	$F_{A=B}$	$F_{A<B}$
0 0	0	1	0
0 1	0	0	1
1 0	1	0	0
1 1	0	1	0

$$F_{A>B} = A\overline{B} = A(\overline{A} + \overline{B}) = A\overline{AB}$$

$$F_{A<B} = \overline{A}B = B(\overline{A} + \overline{B}) = B\overline{AB}$$

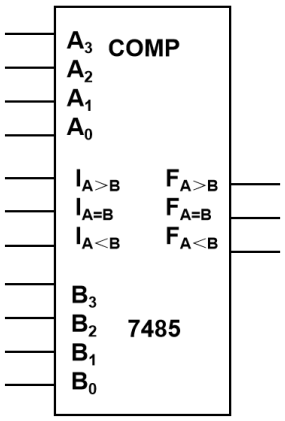
$$F_{A=B} = AB + \overline{A}\,\overline{B} = \overline{A\overline{B}} + \overline{B\overline{A}}$$

$$= \overline{A\overline{AB}} + \overline{B\overline{AB}}$$



1位比较器->4位比较器

❖ 4位比较器(7485芯片)

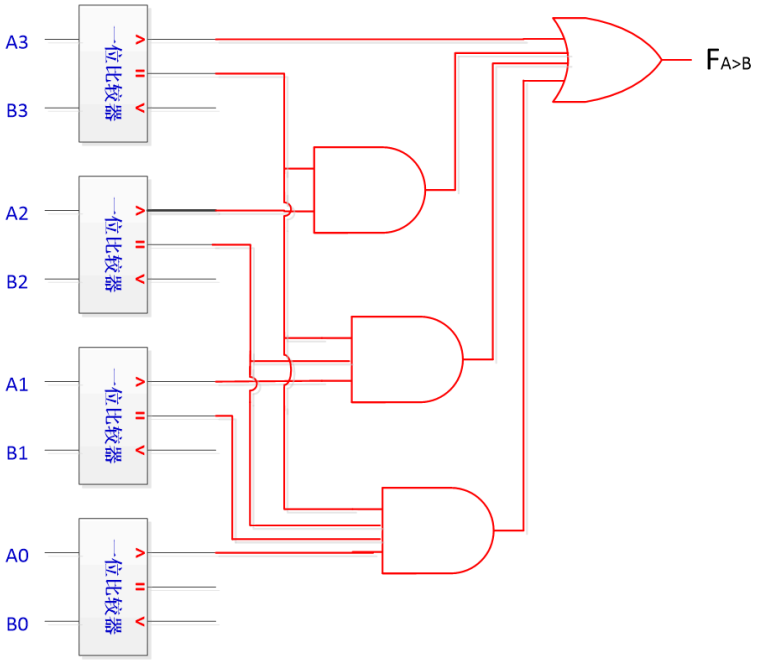


（2）功能表

A3 B3	A2 B2	A1 B1	A0 B0	$I_{A>B}$ $I_{A=B}$ $I_{A<B}$	$F_{A>B}$ $F_{A=B}$ $F_{A<B}$
A3>B3	X	X	X	X X X	1 0 0
A3<B3	X	X	X	X X X	0 0 1
A3=B3	A2>B2	X	X	X X X	1 0 0
A3=B3	A2<B2	X	X	X X X	0 0 1
A3=B3	A2=B2	A1>B1	X	X X X	1 0 0
A3=B3	A2=B2	A1<B1	X	X X X	0 0 1
A3=B3	A2=B2	A1=B1	A0>B0	X X X	1 0 0
A3=B3	A2=B2	A1=B1	A0<B0	X X X	0 0 1
A3=B3	A2=B2	A1=B1	A0=B0	a b c	a b c

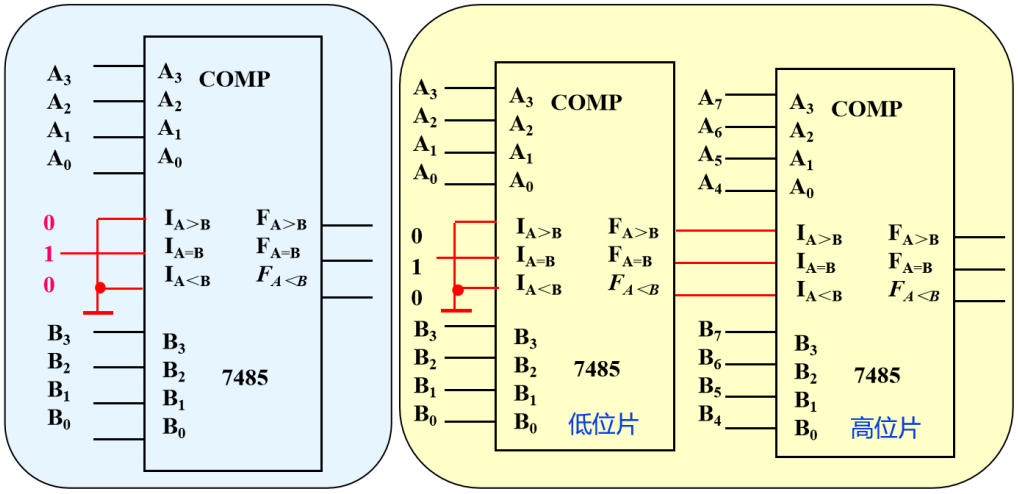
级联输入端，用于芯片的扩展

规则：从高位开始比较，高位不等时，数值的大小由高位决定；若高位相等，则再比较低位，数值的大小由低位比较结果决定。
比如：若$A_3>B_3$ 则A>B；若$A_3<B_3$ 则A<B；若$A_3=B_3$ 则再比较低位



4位比较器->8位比较器

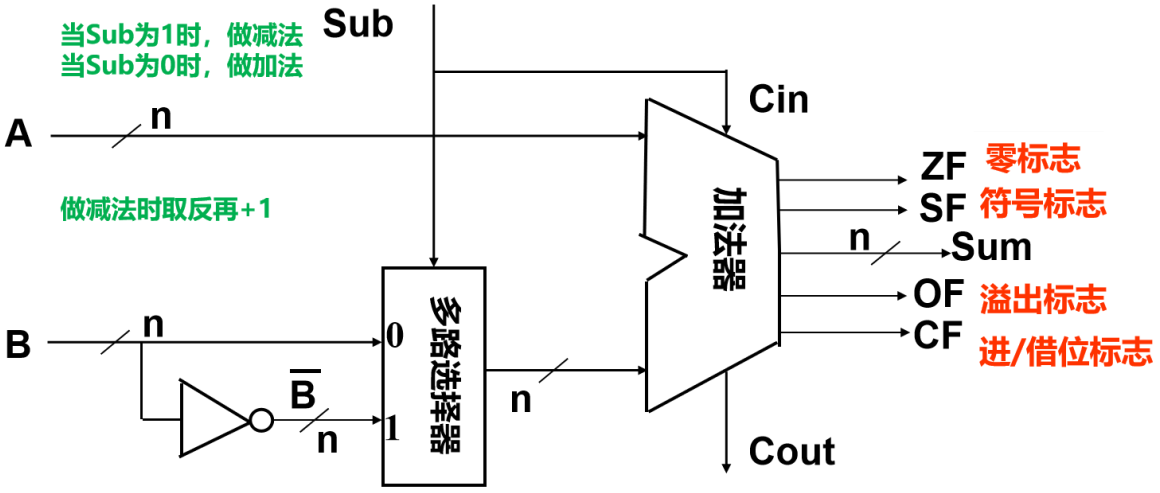
❖ 4位比较器(7485芯片)的使用与扩展



单片：4位数值比较器　　2片扩展：8位数值比较器

ALU

1. 计算机中所有算术运算都基于**加法器**实现。在此基础上，加上寄存器、移位器以及控制逻辑，就可实现乘/除运算以及浮点运算电路
2. 加法器**不知道所运算的是带符号数还是无符号数。**
3. 加法器**不判定对错**，总是取低n位作为结果，并生成标志信息。

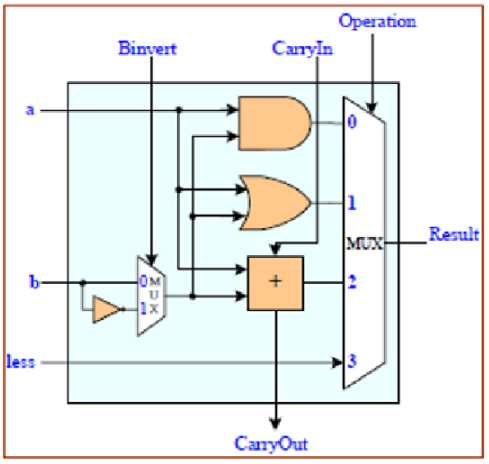


1位ALU

1位ALU(带比较功能)

- 逻辑：与、或
- 算术：
  - 加：Binvert=0, CarryIn = 0
  - 减：Binvert=1, CarryIn = 1
- 比较：Less(小于)，负（a-b）

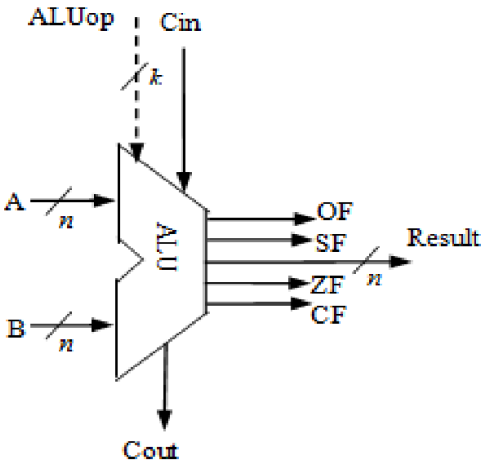
可以引入Bnegate代替Binvert和CarryIn来简化电路



1位ALU（带比较功能）

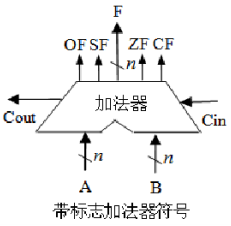
PROMAX ALU

- 进行基本算术运算与逻辑运算
  - 无符号整数加、减
  - 带符号整数加、减
  - 与、或、非、异或等逻辑运算
- 核心电路是整数加/减运算部件
- 输出和/差等，以及标志信息
- 比较大小，通过做减法得到的标志信息来判断
- 有一个操作控制端（ALUop），用来决定ALU所执行的处理功能。ALUop的位数k决定了操作的种类，例如，当位数k为3时，ALU最多只有2^3=8种操作。

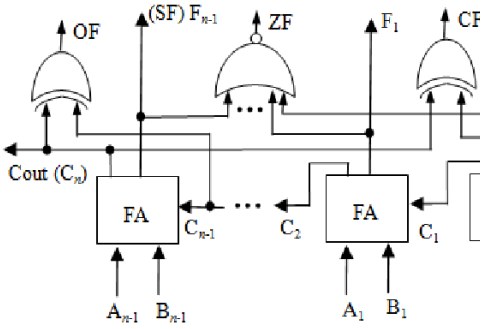


ALU 符号

- 条件标志位（条件码CC）
  - ① 溢出标志OF：$OF=C_n \oplus C_{n-1}$
    当有符号数运算结果超出了寄存器所能表示的范围时，V标志位被设置为1。
  - ② 符号标志SF：$SF=F_{n-1}$
    当运算结果为负数时，N标志位被设置为1。
  - ③ 零标志: ZF=1当且仅当F=0；其中F: SUM
    当运算结果为0时，Z标志位被设置为1。
  - ④ 进位/借位标志CF：$CF=Cout \oplus Cin$
    在无符号数运算中，如果最高位发生进位，C标志位被设置为1。



带标志加法器符号



带标志加法器的逻辑电路

1. 零标志ZF、溢出标志OF、进/借位标志CF、符号标志SF称为条件标志。
2. 条件标志（Flag）在运算电路中产生，被记录到专门的寄存器中。
3. 存放标志的寄存器通常称为程序/状态字寄存器或标志寄存器。每个标志对应标志寄存器中的一个标志位。 如，IA-32中的EFLAGS寄存器。

编码器、译码器

编码器

- 输入：**n个**输入信号，每个代表一个不同事件，一般应保证**任意时刻只有一个有效。**
- 输出：一组二值代码（**m位，**$n \leq 2^m$），对应输入信号中有效事件的编码。

二进制编码器

- 输入： 2^n 个信号
- 输出： n 位二进制代码
- 高电平输入有效：在输入等于1时对输入信号进行编码

- 低电平输入有效：在输入等于0时对输入信号进行编码

❖**8线-3线编码器**（高电平输入有效）已知真值表求电路连接

1. 利用最小项推导法写出各输出的逻辑函数表达式；

$$C = \overline{Y}_7\overline{Y}_6\overline{Y}_5Y_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0 + \overline{Y}_7\overline{Y}_6Y_5\overline{Y}_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0 + \overline{Y}_7Y_6\overline{Y}_5\overline{Y}_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0 + Y_7\overline{Y}_6\overline{Y}_5\overline{Y}_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0$$

2. 利用约束项化简：任何时刻**Y7~Y0**中仅有一个为**1**，输入变量取值组合仅有**8**种状态，其余均为约束项。利用这些约束项化简上式，得到：

若 $Y_4 = 1$，$\overline{Y}_7\overline{Y}_6\overline{Y}_5\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0 = 1$；

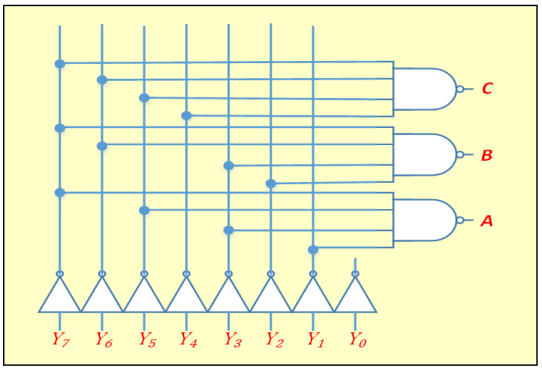
所以有：$C = Y_4 + Y_5 + Y_6 + Y_7$

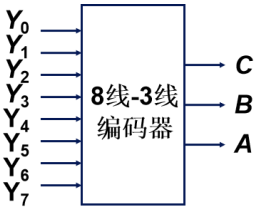
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	C	B	A
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	x	x	x
0	0	0	0	0	1	1	1	x	x	x
约束项 ……								……		
1	1	1	1	1	1	1	1	x	x	x

$$A = Y_1 + Y_3 + Y_5 + Y_7 = \overline{\overline{Y}_1 \cdot \overline{Y}_3 \cdot \overline{Y}_5 \cdot \overline{Y}_7}$$

$$B = Y_2 + Y_3 + Y_6 + Y_7 = \overline{\overline{Y}_2 \cdot \overline{Y}_3 \cdot \overline{Y}_6 \cdot \overline{Y}_7}$$

$$C = Y_4 + Y_5 + Y_6 + Y_7 = \overline{\overline{Y}_4 \cdot \overline{Y}_5 \cdot \overline{Y}_6 \cdot \overline{Y}_7}$$





编码表

输入	C B A
Y_0	0 0 0
Y_1	0 0 1
Y_2	0 1 0
Y_3	0 1 1
Y_4	1 0 0
Y_5	1 0 1
Y_6	1 1 0
Y_7	1 1 1

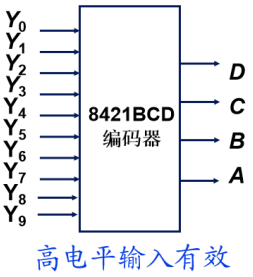
*8421BCD编码器

❖**8421BCD编码器：用4位二进制对1十进制数字进行编码。**

- 输入：Y0，Y1，…，Y9，分别代表十进制数字0~9；
- 输出：4位编码，DCBA=0000，代表“0”;DCBA=1001，代表“9”

编码表

输入	D C B A
Y_0	0 0 0 0
Y_1	0 0 0 1
Y_2	0 0 1 0
Y_3	0 0 1 1
Y_4	0 1 0 0
Y_5	0 1 0 1
Y_6	0 1 1 0
Y_7	0 1 1 1
Y_8	1 0 0 0
Y_9	1 0 0 1



高电平输入有效

$$D = Y_8 + Y_9 = \overline{\overline{Y_8} \cdot \overline{Y_9}}$$

$$C = Y_4 + Y_5 + Y_6 + Y_7 = \overline{\overline{Y_4} \cdot \overline{Y_5} \cdot \overline{Y_6} \cdot \overline{Y_7}}$$

$$B = Y_2 + Y_3 + Y_6 + Y_7 = \overline{\overline{Y_2} \cdot \overline{Y_3} \cdot \overline{Y_6} \cdot \overline{Y_7}}$$

$$A = Y_1 + Y_3 + Y_5 + Y_7 + Y_9 = \overline{\overline{Y_1} \cdot \overline{Y_3} \cdot \overline{Y_5} \cdot \overline{Y_7} \cdot \overline{Y_9}}$$

优先编码器

- 克服二进制编码器仅允许1个输入信号有限的局限，**允许两个以上输入信号同时有效**；
- 对所有输入信号**进行优先级别排序**，任何时刻**只对优先级最高的输入信号编码**，对优先级别低的输入信号则不响应，以保证编码器可靠工作。
- 74LS148：8线-3线优先编码器，8个输入信号，低电平有效；3个输出端，反码输出
- 74LS147：10线-4线优先编码器，10个输入信号，低电平有效；4个输出端，反码输出

❖**74147优先编码器**（低电平输入有效）

输入：$\overline{I_0}$ ~ $\overline{I_9}$，$\overline{I_9}$优先级最高，$\overline{I_0}$优先级最低

输出：$\overline{Y_3}$ ~ $\overline{Y_0}$，当$\overline{I_9}$ = 0（有效）时，$\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$=0110（“9”的BCD码之反码）

I_9	I_8	$\overline{I_7}$	$\overline{I_6}$	$\overline{I_5}$	$\overline{I_4}$	$\overline{I_3}$	$\overline{I_2}$	$\overline{I_1}$	$\overline{I_0}$	$\overline{Y_3}$	Y_2	$\overline{Y_1}$	$\overline{Y_0}$
0	x	x	x	x	x	x	x	x	x	0	1	1	0
1	0	x	x	x	x	x	x	x	x	0	1	1	1
1	1	0	x	x	x	x	x	x	x	1	0	0	0
1	1	1	0	x	x	x	x	x	x	1	0	0	1
1	1	1	1	0	x	x	x	x	x	1	0	1	0
1	1	1	1	1	0	x	x	x	x	1	0	1	1
1	1	1	1	1	1	0	x	x	x	1	1	0	0
1	1	1	1	1	1	1	0	x	x	1	1	0	1
1	1	1	1	1	1	1	1	0	x	1	1	1	0
1	1	1	1	1	1	1	1	1	0	1	1	1	1

低电平输入有效

输出为优先级最高的输入信号对应编号的反码

译码器

- **变量译码器（二进制译码器，地址译码器）**：实现输入变量状态全部组合的译码器，一般称为n线-2n线译码器。如2线-4线译码器，3线-8线译码器
- **码制变换译码器**：将输入的某个进制代码转换成对应的其他码制输出的译码器。如8421码至十进制码译码器（简称**BCD译码器**）、余3码至十进制码译码器等。
- **显示译码器**：将输入代码转换成驱动7段数码显示器各段电平信号的译码器。常用的有74xx47（低电平输出有效）、74xx49（高电平输出有效）、74xx48（高电平输出有效）

等。

用译码器设计组合逻辑电路

1. 首先将被实现的函数变成**最小项表示的与或表达式**。并将被实现函数的变量接到译码器的**代码输入端**；
2. 当译码器的输出为**高电平有效**时，选用**或门**；当输出为**低电平有效**时，选用**与非门**；
3. 将译码器输出与逻辑函数F所具有的**最小项相对应的所有输出端连接到一个或门（或者与非门）的输入端**，则或门（或者与非门）的输出就是被实现的逻辑函数。

例. 利用74LS138及一些门电路，设计一个多路输出的组合逻辑电路，输出表达式：

$$F_1 = A\overline{C} \qquad F_2 = BC + \overline{A}\overline{B}C$$

$$F_3 = AB + A\overline{B}C \qquad F_4 = ABC$$

由于74LS138的输出为低电平有效，故应选择与非门作输出门。将逻辑函数的变量A、B、C分别加到74LS138译码器的输入端A2A1A0，并将译码器输出与逻辑函数F1、F2、F3、F4中分别具有的最小项相对应的所有输出端，连接到一个与非门的输入端，则各个与非门的输出就可实现逻辑函数。如图所示

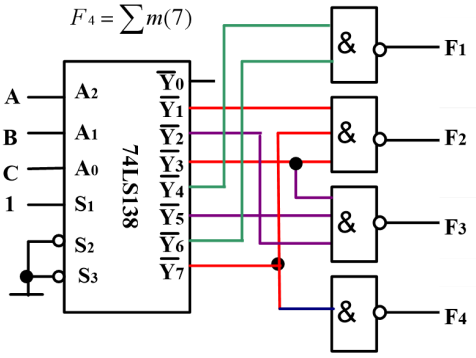
解：函数化为最小项标准表达式：

$F_1 = \sum m(4,6)$

$F_2 = \sum m(1,3,7)$

$F_3 = \sum m(5,6,7)$ 下面的F3线连错了

$F_4 = \sum m(7)$



❖用3线/8线译码器实现逻辑函数（高电平有效）

已知：$F = A \oplus B \oplus C = \sum m(1,2,4,7)$

$$= \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$

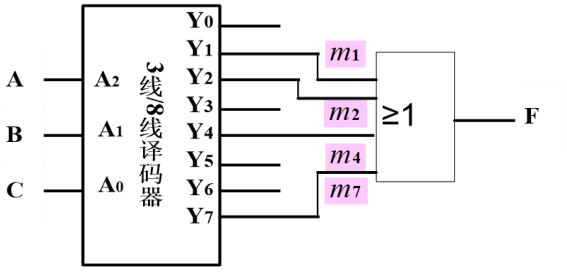
$Y_0 = \overline{A}_2\overline{A}_1\overline{A}_0 = m_0$

$Y_1 = \overline{A}_2\overline{A}_1A_0 = m_1$

$\vdots$

$Y_7 = A_2A_1A_0 = m_7$

设计组合电路：



❖用74LS138译码器实现(低电平输出）

$$F = A \oplus B \oplus C = \sum m(1,2,4,7)$$

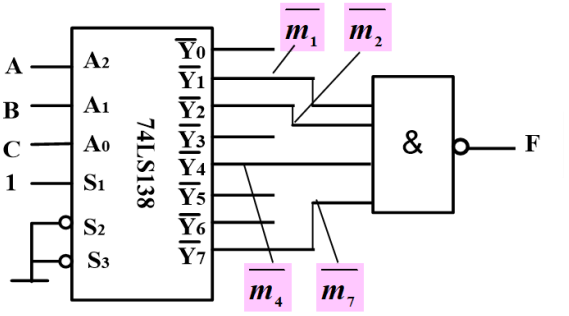
$$\overline{Y_0} = \overline{\overline{A_2}\,\overline{A_1}\,\overline{A_0}} = \overline{m_0} \qquad F = \overline{\overline{Y_1}\,\overline{Y_2}\,\overline{Y_4}\,\overline{Y_7}}$$

$$\overline{Y_1} = \overline{\overline{A_2}\,\overline{A_1}A_0} = \overline{m_1} \qquad = Y_1 + Y_2 + Y_4 + Y_7$$

$$\overline{Y_2} = \overline{\overline{A_2}A_1\overline{A_0}} = \overline{m_2} \qquad = m_1 + m_2 + m_4 + m_7$$

$\cdots$

$$\overline{Y_7} = \overline{A_2A_1A_0} = \overline{m_7}$$



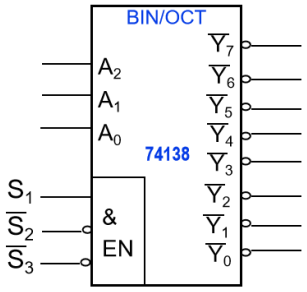
二进制译码器

- 任何时刻最多只允许**1个输出有效**
- 高电平输出有效时，每个输出都是对应的输入变量最小项；

- 低电平输出有效时，每个输出都是对应的输入变量最小项的反，所以二进制译码器也称为最小项译码器。

❖**3线-8线译码器（74138）**

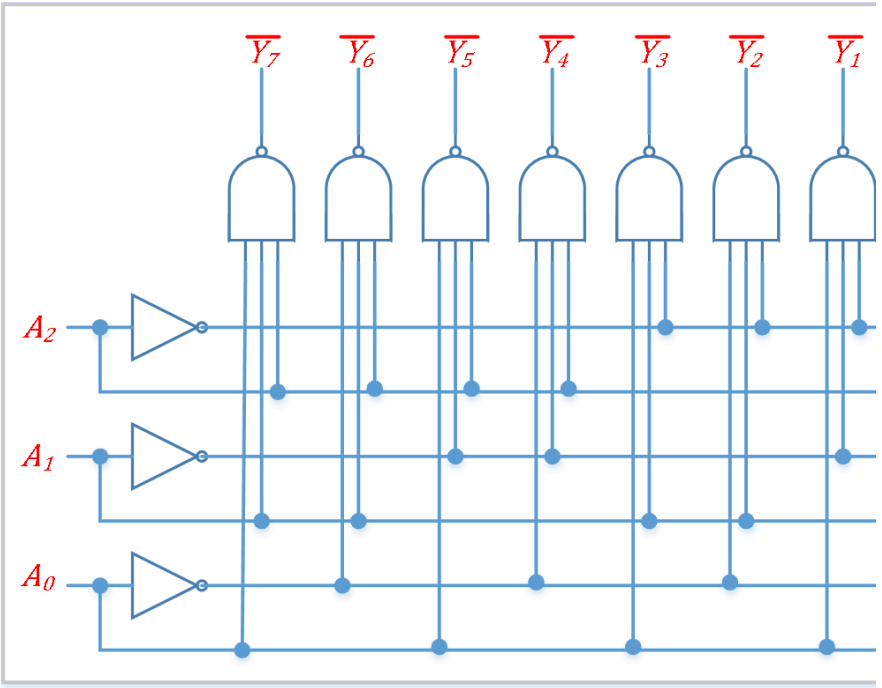
- 3个输入：A2，A1，A0。
- 8个输出：Y7~Y0，低电平输出有效。
- 3个使能控制：S0，S1，S2 为使能输入，仅当它们分别为1、0、0时，译码器才正常译码；否则禁止工作。



功能表

$S_1\overline{S_2}\overline{S_3}$	A_2 A_1 A_0	$\overline{Y_7}\overline{Y_6}\overline{Y_5}\overline{Y_4}\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$
≠100	X X X	1 1 1 1 1 1 1 1
=100	0 0 0	1 1 1 1 1 1 1 0
=100	0 0 1	1 1 1 1 1 1 0 1
=100	0 1 0	1 1 1 1 1 0 1 1
=100	0 1 1	1 1 1 1 0 1 1 1
=100	1 0 0	1 1 1 0 1 1 1 1
=100	1 0 1	1 1 0 1 1 1 1 1
=100	1 1 0	1 0 1 1 1 1 1 1
=100	1 1 1	0 1 1 1 1 1 1 1

$$\overline{Y}_0 = A_2 + A_1 + A_0 = \overline{A_2 A_1 A_0}$$
$$\overline{Y}_2 = A_2 + \overline{A}_1 + A_0 = \overline{A_2 A_1 A_0}$$
$$\overline{Y}_4 = \overline{A}_2 + A_1 + A_0 = \overline{A_2 A_1 A_0}$$
$$\overline{Y}_2 = \overline{A}_2 + \overline{A}_1 + A_0 = \overline{A_2 A_1 A_0}$$
$$\overline{Y}_1 = A_2 + A_1 + \overline{A}_0 = \overline{A_2 A_1 A_0}$$
$$\overline{Y}_3 = A_2 + \overline{A}_1 + \overline{A}_0 = \overline{A_2 A_1 A_0}$$
$$\overline{Y}_5 = \overline{A}_2 + A_1 + \overline{A}_0 = \overline{A_2 A_1 A_0}$$
$$\overline{Y}_7 = \overline{A}_2 + \overline{A}_1 + \overline{A}_0 = \overline{A_2 A_1 A_0}$$



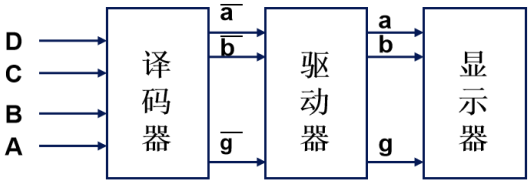
码制变换译码器

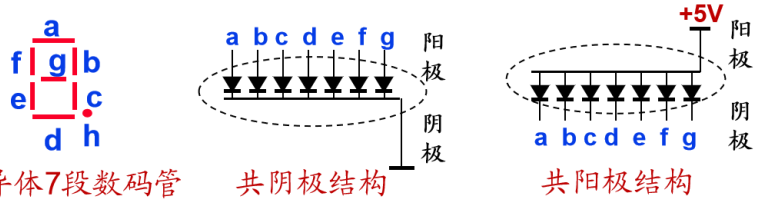
- **BCD译码器**（4-10线译码器）：**4位8421码转换成十进制数**的电路。用于驱动十进制数码显示管、指示灯等
  - **完全译码**BCD译码器：当输入出现伪码0101~1111时，译码器输出Y0~Y9**均为“1”**，如74xx42（输出为低电平有效）
  - **不完全译码**BCD译码器：当输入出现伪码0101~1111时，Y0~Y9**均为任意值**
- 余3码至十进制码译码器
- 余3循环码至十进制码译码器

显示译码器

❖显示译码器：用于驱动数码显示器，将二进制代码表示的数字、文字、符号用人们习惯的形式直观显示出来的电路。

电路结构（8421BCD译码显示电路）





半导体7段数码管　共阴极结构　共阳极结构

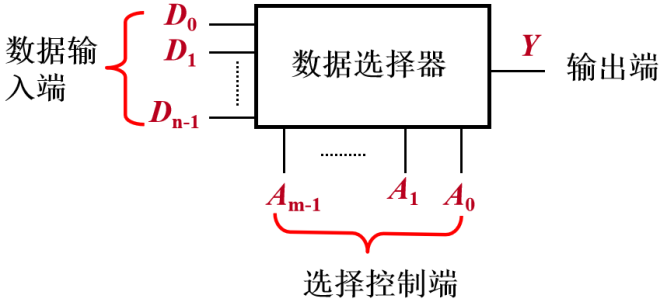
➢ 若使用共阴极LED数码管，则显示译码器的输出应为高电平输出有效；若使用共阳极LED数码管，则译码器应为低电平输出有效。

译码器（共阳器件）

DCBA	$\overline{a}$	$\overline{b}$	$\overline{c}$	$\overline{d}$	$\overline{e}$	$\overline{f}$	$\overline{g}$	显示数字
0000	0	0	0	0	0	0	1	0
0001	1	0	0	1	1	1	1	1
0010	0	0	1	0	0	1	0	2
0011	0	0	0	0	1	1	0	3
0100	1	0	0	1	1	0	0	4
0101	0	1	0	0	1	0	0	5
0110	1	1	0	0	0	0	0	6
0111	0	0	0	1	1	1	1	7
1000	0	0	0	0	0	0	0	8
1001	0	0	0	1	1	0	0	9
1010	X	X	X	X	X	X	X	
1011	X	X	X	X	X	X	X	
1100	X	X	X	X	X	X	X	
1101	X	X	X	X	X	X	X	
1110	X	X	X	X	X	X	X	
1111	X	X	X	X	X	X	X	

多路选择器

- 从一组输入数据选出其中一个作为数据输出的电路
- 以“与或非”门或以“与或”门为主体，在选择控制信号的作用下，能从多路平行输入数据中任选一路数据作为输出。
- 常用的集成数据选择器有2选1（74xx157）、4选1（74xx153）、8选1（74xx151）及16选1（74xx150）等。



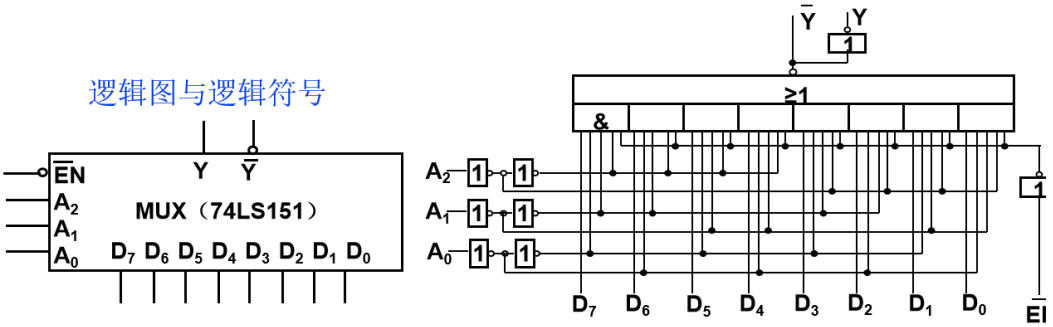
数据选择

功能一：8选1数据选择器。$D_7 \sim D_0$为数据输入端，$A_2A_1A_0$为选择控制端，使能控制输入EN为低电平时有效。

功能表（$\overline{EN}$=0）

A_2	A_1	A_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

逻辑图与逻辑符号



$$Y = \overline{A_2}\,\overline{A_1}\,\overline{A_0}D_0 + \overline{A_2}\,\overline{A_1}A_0D_1 + \overline{A_2}A_1\overline{A_0}D_2 + \overline{A_2}A_1A_0D_3 + A_2\overline{A_1}\,\overline{A_0}D_4 + A_2\overline{A_1}A_0D_5 + A_2A_1\overline{A_0}D_6 + A_2A_1A_0D_7$$

多功能运算电路，可实现任意组合逻辑电路的设计

功能二：多功能运算电路。 $D_7 \sim D_0$为控制输入端。

- 通过D_7~D_0取不同的值，从输入变量A_2、A_1、A_0的各个最小项中选取某几个最小项的或输出，实现不同的运算电路。
- 有2^8=256种功能，包含3个逻辑变量的各种最小项表达式，可实现任意组合逻辑电路的设计。

$$Y = D_0m_0 + D_1m_1 + D_2m_2 + D_3m_3 + D_4m_4 + D_5m_5 + D_6m_6 + D_7m_7$$

当$D_7 \sim D_0$为0000_0000时，Y=0
当$D_7 \sim D_0$为1111_1111时， Y=1
当$D_7 \sim D_0$为0000_0001时，$Y=m_0$
当$D_7 \sim D_0$为1010_0101时，$Y=m_7+m_5+m_2+m_0$

功能表（$\overline{EN}$=0）

A_2	A_1	A_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

给一个逻辑功能

1. 根据逻辑功能列出**真值表**
2. 用**最小项之和**写出**逻辑式**
3. 用数据选择器进行实现
4. 标注清楚哪个**线**是ABC，那些线是01
5. 标注**变量**：注意顺序（A-A2 B-A1 C-A0）

6. 芯片**高电平有效/低电平有效**

用数据选择器实现逻辑函数：　$F(A,B,C)=\overline{A}\overline{B}C+\overline{A}B\overline{C}+AB$

解： 使用**8选1数据选择器74151**

已知**74151**的函数表达式：

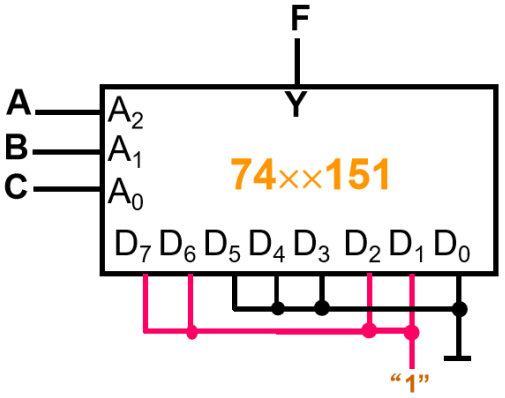
$$Y=D_0m_0+D_1m_1+D_2m_2+D_3m_3+D_4m_4+D_5m_5+D_6m_6+D_7m_7$$

而：

$$F=\overline{A}\overline{B}C+\overline{A}B\overline{C}+AB(\overline{C}+C)$$
$$=m_1+m_2+m_6+m_7$$

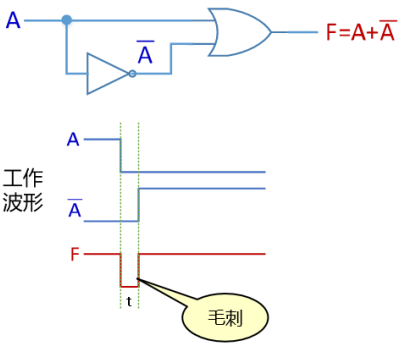
则得：

$D_0=0$，$D_1=1$，$D_2=1$，$D_3=0$，
$D_4=0$，$D_5=0$，$D_6=1$，$D_7=1$

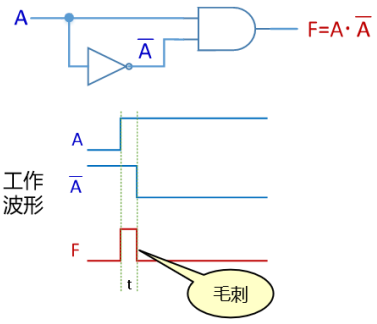


组合逻辑电路的竞争冒险

现象及原因



0冒险



1冒险

- **竞争**：在组合逻辑电路中，某个输入变量通过两条或两条以上的途径传到输出端，由于每条途径延迟时间不同，到达输出门的时间就有先有后，这种（两个或多个信号不同步）现象称为竞争。
- **冒险：**门电路因输入端的竞争而导致输出端产生不正常的尖峰干扰脉冲信号（毛刺）的现象，成为冒险。
- **竞争冒险的原因：门电路和导线传输的延时。**信号在器件内部通过连线和逻辑单元时，都有一定的延时。延时的大小与连线的长短和逻辑单元的数目有关，同时还受器件的制造工艺、工作电压、温度等条件的影响。信号的高低电平转换也需要一定的过渡时间。

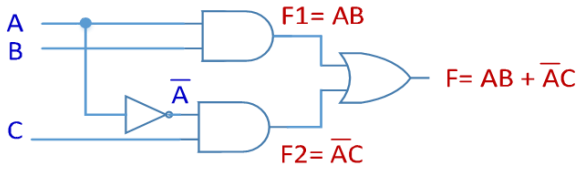
竞争冒险的判断

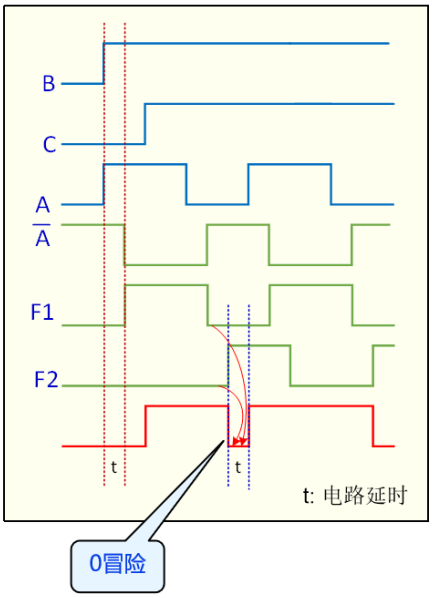
1. 代数法

逻辑函数 F 中，若某个变量（假定为 A）同时以原变量和反变量形式存在，逻辑函一定条件下（其他变量取特定的值1或0）可以简化为 $F = A + \overline{A}$ 或 $F = A \cdot \overline{A}$ 的形式，逻辑电路存在冒险。$F = A + \overline{A}$ 存在“0”冒险，$F = A \cdot \overline{A}$ 存在“1”冒险。

例： $F = AB + \overline{A}C$

当 B=1，C=1时，逻辑电路可简化为 $F = A + \overline{A}$





例：判断F是否存在冒险？ $F = AC + \overline{A}B + \overline{A}\overline{C}$

其中变量B不具备竞争条件，但A和C具有竞争条件的变量。先考虑A能否产生冒险。为了消去B和C，留下变量A，把B和C的各种可以能取值组合代入函数式中，如下表（a)所示。当B=C=1时，电路存在“0”型冒险。由表（b）可见，C的变化不会产生冒险。

表（a): A冒险判断

B	C	$F = AC + \overline{A}B + \overline{A}\overline{C}$
0	0	$F = A0 + \overline{A}0 + \overline{A}\overline{0} = \overline{A}$
0	1	$F = A1 + \overline{A}0 + \overline{A}1 = A$
1	0	$F = A0 + \overline{A}1 + \overline{A}0 = \overline{A}$
1	1	$F = A1 + \overline{A}1 + \overline{A}1 = A + \overline{A}$

产生“0”冒险

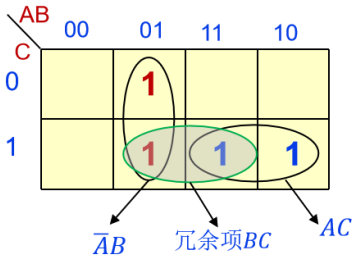
表（b): C冒险判断

A	B	F
0	0	$\overline{C}$
0	1	1
1	0	C
1	1	C

不产生冒险

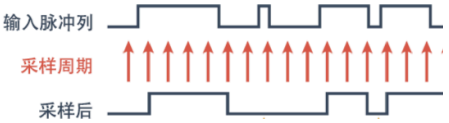
竞争冒险的消除

1. **修改逻辑设计**



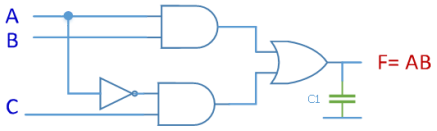
- 消除互补变量。如$L=(A+B)(\overline{A}+C)$，若直接据此设计逻辑电路，当B=C=0时，$L=A\cdot\overline{A}$，存在冒险。修改设计，$L=AC+\overline{A}B+BC$，逻辑函数功能不变，冒险消除。
- 增加冗余项。$L=AC+\overline{A}B$，当B=C=1时，$L=A+\overline{A}$，存在冒险。增加冗余项$L=AC+\overline{A}B+BC$，逻辑函数功能不变，冒险消除。

2. **引入采样脉冲**



- 在电路的输入端引入一个采样脉冲，采样脉冲的作用时间取在电路达到新的稳定状态之后，这样，逻辑电路的输出端不会出现毛刺。

3. **输出端并联电容**



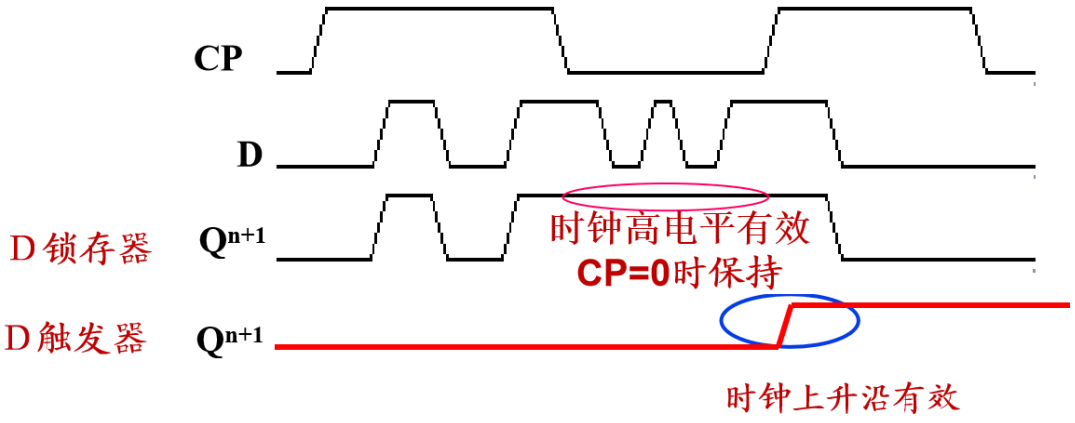
- 对于速度较慢的组合逻辑电路，由于冒险产生的尖峰脉冲一般情况下很窄，所以可以采用在电路输出端并联电容的方法消除尖峰脉冲。

2.2 时序逻辑

常见时序电路

锁存器与触发器的区别

- 锁存器是电位（电平）触发的，只有在时钟CP**有效电平**（高电平CP=1或者低电平CP = 0）期间，锁存器的状态才有可能发生变化。
- 触发器的状态变化只发生在时钟CP的**有效沿**（上升沿或者下降沿）期间，CP = 1、CP=0时触发器的状态不会发生变化。



D触发器

❖ 一个**D**触发器可以由两个反相的**D**锁存器构成，如下图**(a)**所示。图**(b)**为**D**触发器符号。

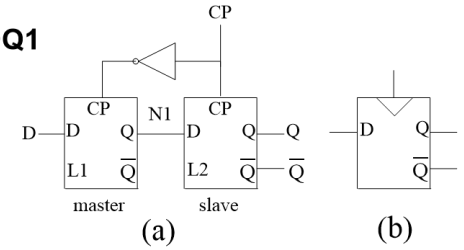
❖ 锁存器**L1**为主锁存器，**L2**为从锁存器

❖ 工作原理

- **CP＝0**；**L1**是通路，**L2**是断路，**Q1←D**，**Q2**值不变
- **CP**从**0**上升到**1**； **Q2←Q1**，触发时刻
- **CP＝1**； **L1**是断路，**L2**是通路，**Q1**值不变， **Q2←Q1**

❖ **D**触发器分类

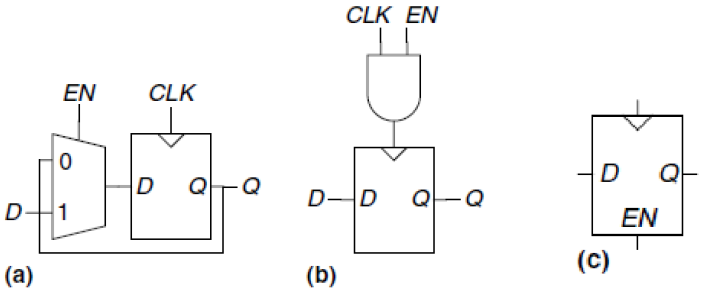
- 主从触发器
- 维持－阻塞触发器



带使能端的D触发器

❖ 增加输入使能信号**EN**（**ENable**），用于确定在时钟沿是否能够载入数据，如下图所示。

- **EN＝1**时，**D**触发器正常工作
- **EN＝0**时， **D**触发器状态不变
- 在时钟信号上一般不要设置逻辑，否则可能因延迟导致时序错误



(a, b)原理图，**(c)**电路符号

带复位功能的D触发器

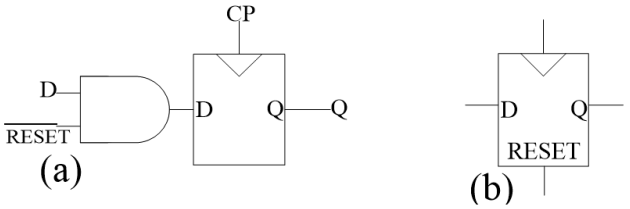
❖增加输入复位信号**RESET**，如下图所示。

- 当系统加电时，触发器设置为已知状态（**Q=0**）
- $\overline{\text{RESET}}$ 有效时（＝**0**），**D**触发器复位（**Q=0**）
- $\overline{\text{RESET}}$ 无效时（＝**1**）， **D**触发器正常工作

❖复位方式

- 同步复位：复位信号有效和时钟有效沿同时有效才能复位（置0）
- 异步复位：只要复位信号有效就能复位

❖有的触发器还带有置位（SET）功能（**Q=1**）



(a)原理图，**(b)**电路符号

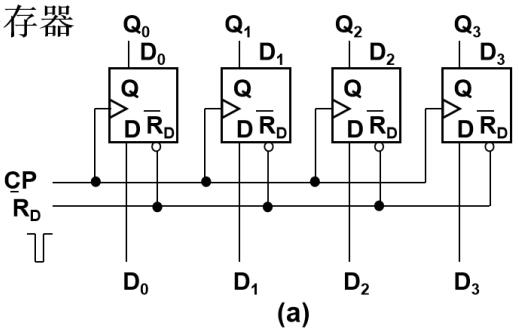
由D触发器构成寄存器

❖由同一时钟控制的*N*个**D**触发器可以构成*N*位寄存器

- 图**(a)**为**4**位寄存器，图**(b)**为电路符号

  D3D2D1D0：并行数据输入

  Q3Q2Q1Q0：并行数据输出



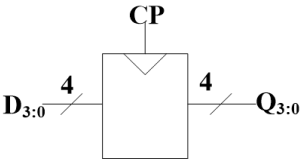
(a)

❖工作原理

（1）清除（复位）

当 $\overline{R_D} = 0$， $Q_0Q_1Q_2Q_3$=0000

（2）置数（复位端无效时） 当**CP**上升沿到来是，$Q_0Q_1Q_2Q_3 = D_0D_1D_2D_3$



(b)

❖工作方式（数据输入输出方式）

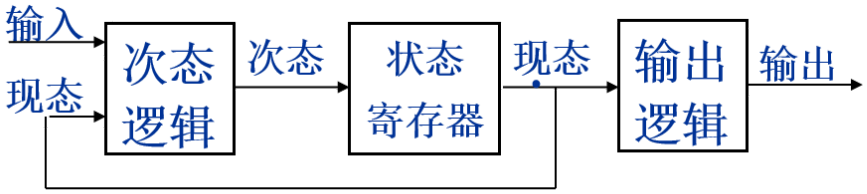
- 并入并出

有限状态机

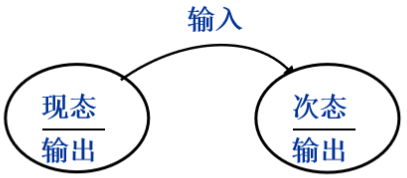
有限状态机是**组合逻辑**和**寄存器逻辑**的特殊组合。组合逻辑部分包括**次态逻辑**和**输出逻辑**，分别用于**状态译码**和**产生输出信号**；寄存器逻辑部分用于**存储状态**。

Moore Mealy

- 摩尔(Moore)型状态机--输出信号**仅与当前状态**有关

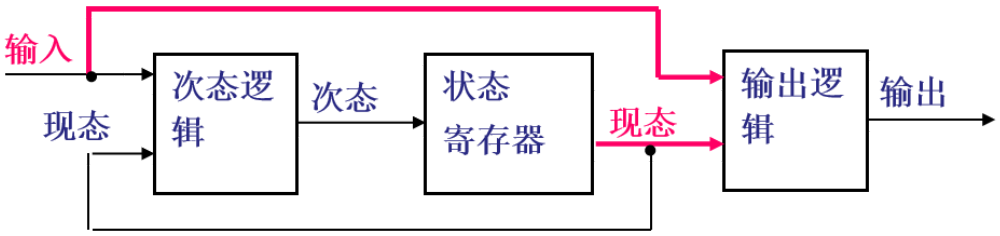


Moore型状态机典型结构



Moore型状态图的表示

- 米里(Mealy)型状态机--输出信号**与当前状态及输入信号**有关



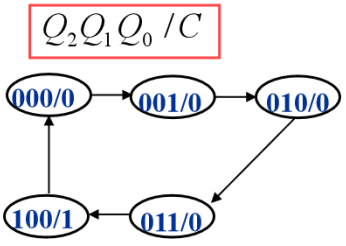
Mealy型状态机的典型结构

Mealy型状态图的

状态图、状态表、状态转移

状态图（State Diagram）、状态表（State Table）

图例，画状态转换图的时候要写



状态（转换）图

状态（转换）表

$Q_2^nQ_1^nQ_0^n$	$Q_2^{n+1}Q_1^{n+1}Q_0^{n+1}$	C
0 0 0	0 0 1	0
0 0 1	0 1 0	0
0 1 0	0 1 1	0
0 1 1	1 0 0	0
1 0 0	0 0 0	1

有限状态机根据描述区分类型，画状态图

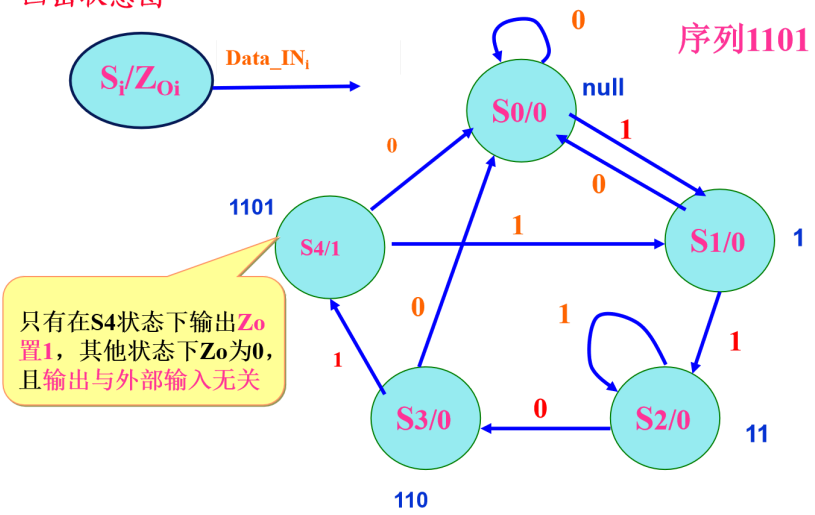
【例2】设计一个序列检测器。要求检测器连续收到串行码{1101}后，输出检测标志为1，否则输出检测标志为0。

第1步：分析设计要求，列出全部可能状态：

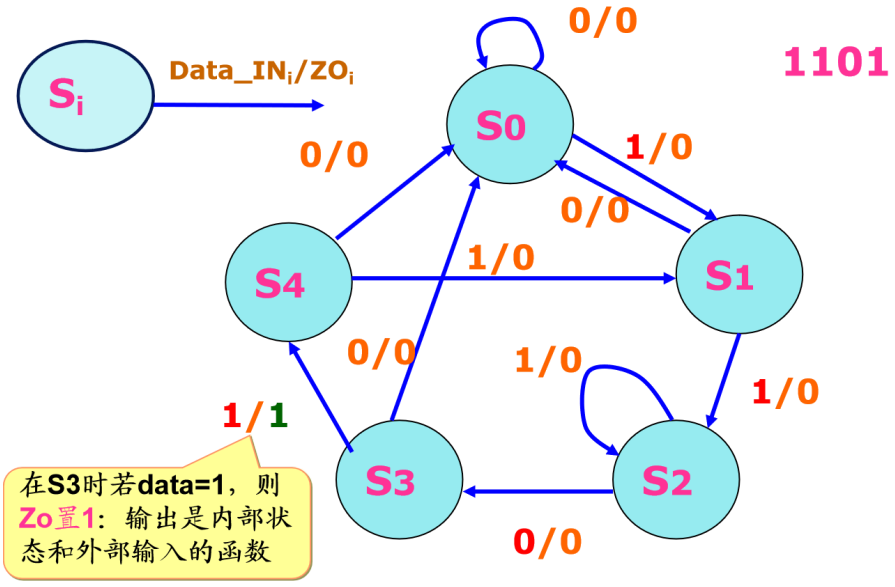
未收到一个有效位（0）　　：S0，输出0
收到一个有效位（1）　　：S1，输出0
连续收到两个有效位（11）　：S2，输出0
连续收到三个有效位（110）　：S3，输出0
连续收到四个有效位（1101）　：S4，输出1

❖由于序列检测器的输出只为状态机当前状态的函数，而与外部输入无关，所以为Moore型状态机

第2步：画出状态图

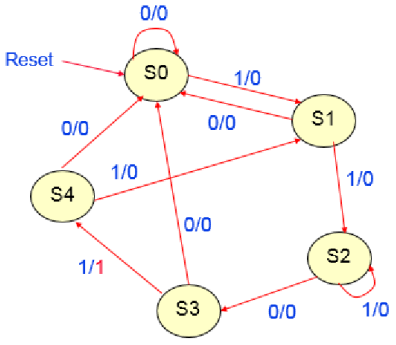


【例3】将【例2】采用Mealy型状态机实现。



根据状态转换图得到状态转换表

当前状态（$S_2S_1S_0$）	输入（A）	下一状态（$S'_2S'_1S'_0$）	输出（Y）
S0（000）	0	S0（000）	0
S0（000）	1	S1（001）	0
S1（001）	0	S0（000）	0
S1（001）	1	S2（010）	0
S2（010）	0	S3（011）	0
S2（010）	1	S2（010）	0
S3（011）	0	S0（000）	0
S3（011）	1	S4（100）	1
S4（100）	0	S0（000）	0
S4（100）	1	S1（001）	0

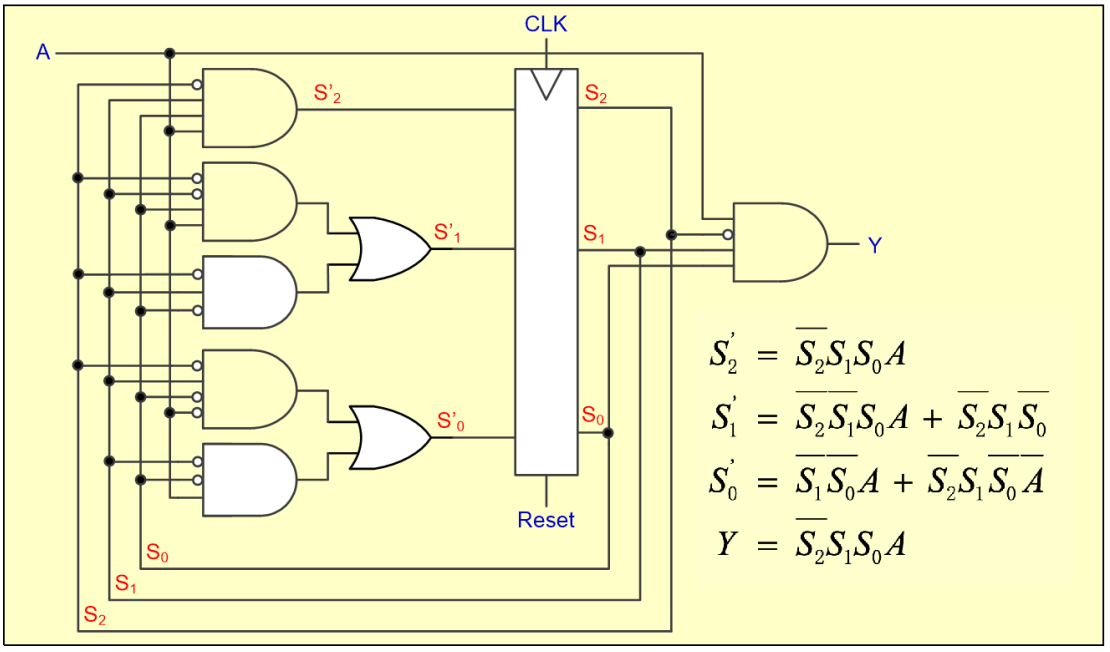


*根据状态转换表写出 **次态逻辑表达式** 和 **输出逻辑表达式**

$$
\begin{aligned}
S_2' &= \overline{S_2}S_1S_0A \\
S_1' &= \overline{S_2}\,\overline{S_1}S_0A + \overline{S_2}S_1\,\overline{S_0}\,\overline{A} + \overline{S_2}S_1\overline{S_0}A \\
&= \overline{S_2}\,\overline{S_1}S_0A + \overline{S_2}S_1\overline{S_0} \\
S_0' &= \overline{S_2}\,\overline{S_1}\,\overline{S_0}A + \overline{S_2}S_1\,\overline{S_0}\,\overline{A} + S_2\overline{S_1}\,\overline{S_0}A \\
&= \overline{S_1}\,\overline{S_0}A + \overline{S_2}S_1\,\overline{S_0}\,\overline{A} \\
Y &= \overline{S_2}S_1S_0A
\end{aligned}
$$

当前状态 ($S_2S_1S_0$)	输入 (A)	下一状态 ($S'_2S'_1S'_0$)	输出 (Y)
S0（000）	0	S0（000）	0
S0（000）	1	S1（001）	0
S1（001）	0	S0（000）	0
S1（001）	1	S2（010）	0
S2（010）	0	S3（011）	0
S2（010）	1	S2（010）	0
S3（011）	0	S0（000）	0
S3（011）	1	S4（100）	1
S4（100）	0	S0（000）	0
S4（100）	1	S1（001）	0

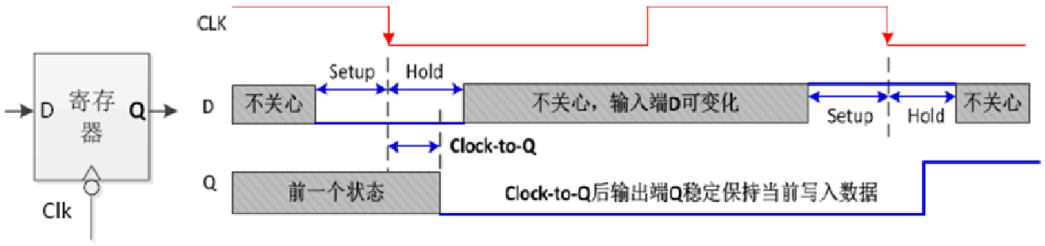
*根据逻辑表达式画出 **逻辑图**



$$
\begin{aligned}
S_2' &= \overline{S_2}S_1S_0A \\
S_1' &= \overline{S_2}\,\overline{S_1}S_0A + \overline{S_2}S_1\overline{S_0} \\
S_0' &= \overline{S_1}S_0A + \overline{S_2}S_1\,\overline{S_0}\,\overline{A} \\
Y &= \overline{S_2}S_1S_0A
\end{aligned}
$$

时序

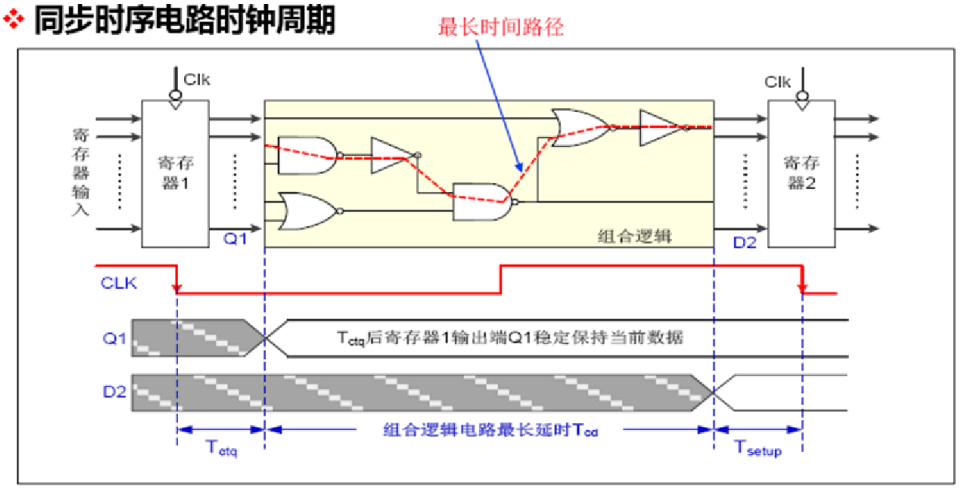
❖寄存器的时序



- 建立时间T_{setup}（Setup Time）：触发时钟边沿之前输入必须稳定的时间；
- 保持时间T_{hold}（Hold Time）：触发时钟边沿之后输入仍需稳定的时间；
- Clock-to-Q时间 T_{ctq}：从触发时钟边沿到输出稳定的时间。
- 孔径时间= T_{setup}+ T_{hold}，输入信号在孔径时间内必须稳定不变。

  T_{ccq}：触发时钟边沿到Q的最小延迟

❖同步时序电路时钟周期



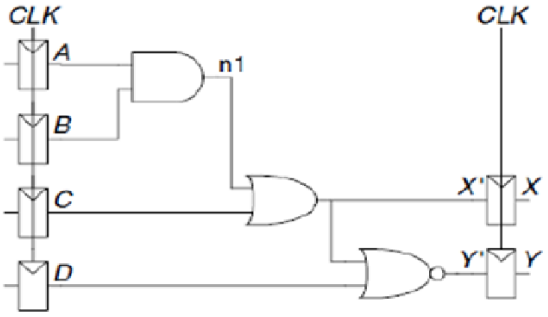
- 时钟周期要足够长：寄存器稳定输出时间 + 次态逻辑最长计算时间 + 寄存器输入建立时间

  T_C >= T_{ctq} + $T_{cd(max)}$ +T_{setup}
- 次态逻辑计算不能过快：寄存器最短输出时间 + 次态逻辑最短计算时间 >= 输入保持时间

  T_{ccq} + $T_{cd(min)}$ ≥ T_{hold}

【例】假定下面的电路触发器时钟到Q的最小延迟和稳定时间分别为30ps和80ps，建立时间和保持时间分别为50ps和60ps，每个门电路的最小延迟和最大延迟分别是25ps和40ps。该时序电路的最小时钟周期是多少？并对时序电路进行时序分析。

解：



（1）T_{ctq}=80，T_{setup}=50

组合逻辑最大延迟T_{cd}=3*40=120。

所以最小时钟周期为：

TC=T_{ctq}+T_{cd}+T_{setup}

= 80 + 120 + 50 = 250ps

（2）T_{ccq}=30，T_{hold}=60

组合逻辑最小延迟为25（D的改变最快经过25ps可能引起Y'改变）。

T_{ccq}+25=55<T_{hold} 违背了保持约束，Y'值（X'值也同样有可能）不能保持足够长的稳定时间，所以Y值实际上不可预测。因此，该电路在任何时钟周期下其功能都可能不正确。

3 汇编语言

3.1 指令格式、寻址方式

指令格式

- 指令的表示：机器表示、符号化表示（汇编语言）

操作数地址的数目

- 三地址：`Des ← (Sur1) OP (Sur2)`
- 双地址：`Des ← (Sur) OP (Des)`
- 单地址：累加器作为默认操作数的双操作数型，或单操作数型（如取反指令），跳转指令
- 无地址：隐含操作数型，或无操作数型，syscall

OP	Des Add	Sur1 Add	Sur2 Add

OP	Des Add	Sur Add

OP	Add

OP

寄存器

Name	Reg.Num	Usage	
zero	0	constant value = 0	恒为0，和0比大小
at	1	reserved for assembler	为汇编程序保留

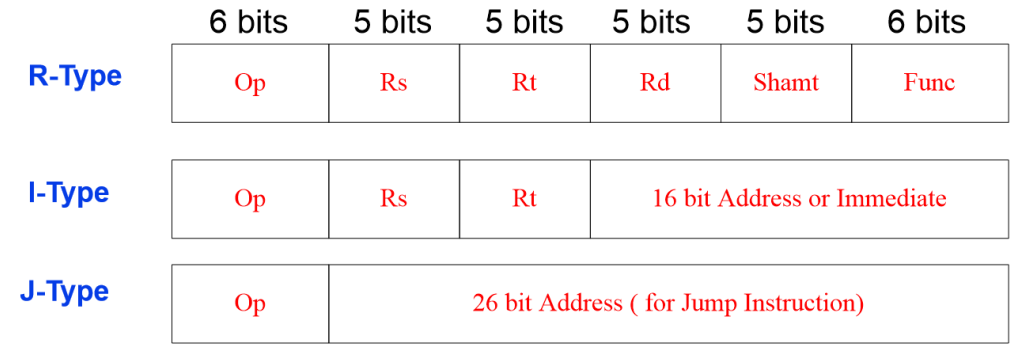
Name	Reg.Num	Usage	
v0-v1	2-3	values for results	过程调用返回值
a0-a3	4-7	Arguments	**过程调用参数**
t0-t7	8-15	Temporaries Caller	保存调用者的**临时变量**
s0-s7	16-23	Saved Callee	**保存被调用者**
t8-t9	24-25	more temporaries	**其他临时变量**
k0-k1	26-27	reserved for kernel	为OS保留
gp	28	global pointer	全局指针
sp	29	stack pointer	**栈指针**
fp	30	frame pointer	帧指针
ra	31	return address	过程调用返回地址，记住最近一次函数调用的caller的返回地址

R/I/J指令

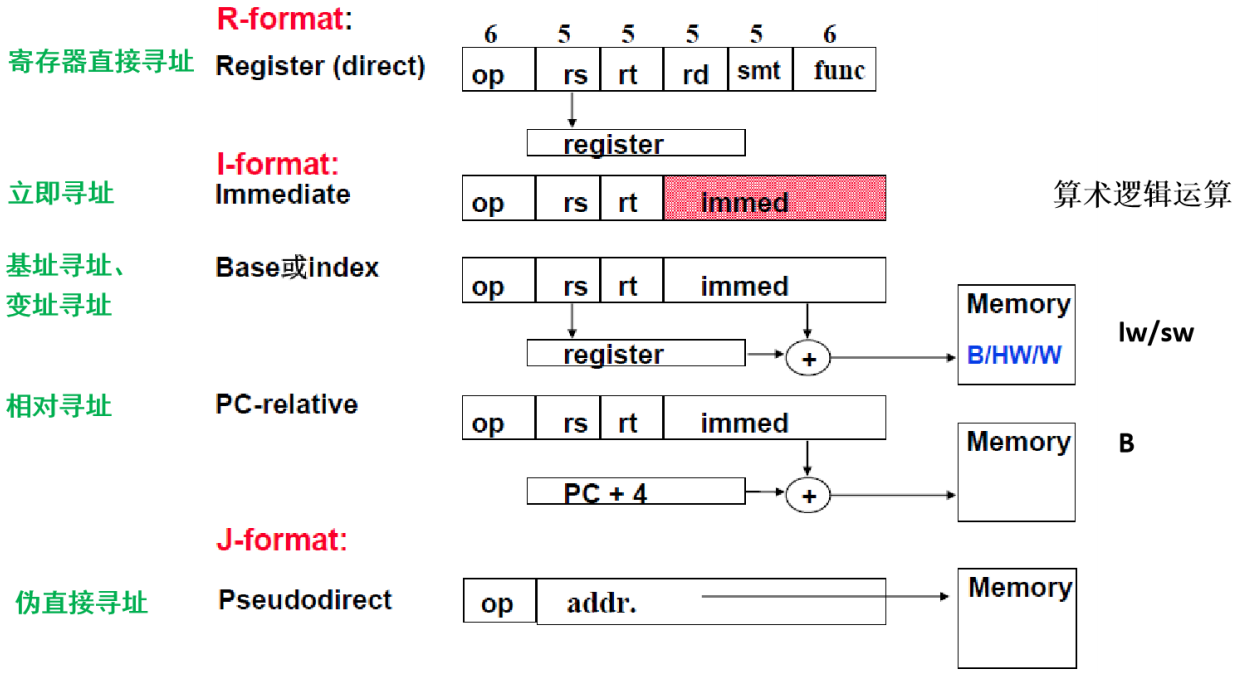
- MIPS只有3种指令格式，32位固定长度指令格式
  - **R-Type**（Register类型）指令：两个寄存器操作数计算，结果送第三个寄存器
  - **I-Type**（Immediate类型）指令：使用1个带符号的16位立即数作
  - **J-Type**（Jump类型）指令：跳转指令，26位跳转地址

❖ MIPS 指令格式

- ➢Op： 6 bits, Opcode 操作码
- ➢Rs: 5 bits, 第一个源寄存器
- ➢Rt: 5 bits, 第二个源寄存器
- ➢Rd: 5 bits, 目标寄存器
- ➢Shamt: 5 bits, Shift amount 逻辑/算数位移移几位，立即数
- ➢Func: 6 bits, function code 例如Opcode告诉算术运算，func告诉加减乘除
  - ▪ R-Type指令OP字段为“000000”，具体操作由func字段给定



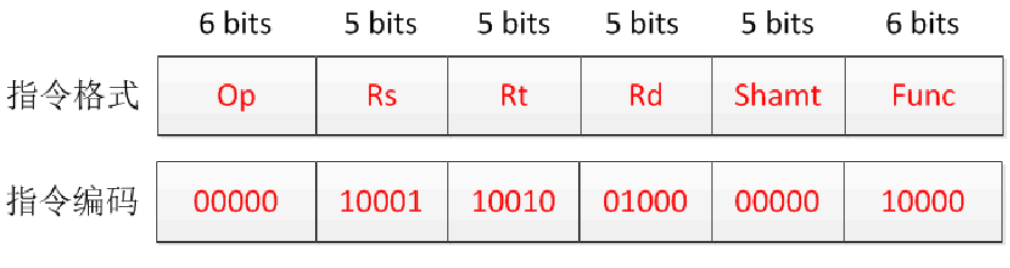
有符号16位二进制数



R-Type指令编码示例

加法 add： `add $t0, $s1, $s2`

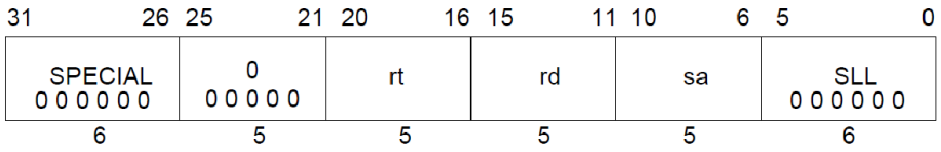
- 解释：$t0 \leftarrow (s1) + (s2)$
- 格式： `ADD rd, rs, rt`
-



Rd ← (Rs) + (Rt)

逻辑左移 Shift Word Left Logical：`sll $t1, $t2, 10`

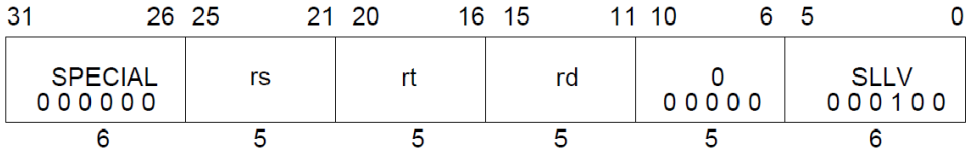
- 解释：$\$t1 = \$t2 << 10$
- 格式：`sll rd, rt, sa`
- **SLL** — Shift Word Left Logical



- 类似的，`srl`，逻辑右移

寄存器移位运算 Shift Word Left Logical Variable：`sllv $t1, $t2, $t3`

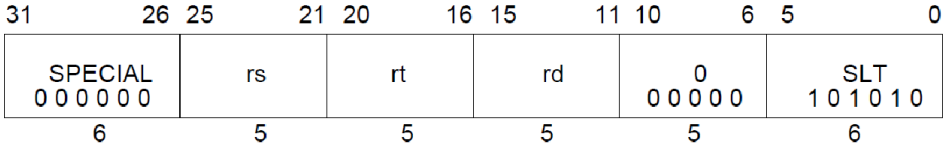
- 解释：$\$t1 = \$t2 << \$t3$
- 格式：`sllv rd, rt, rs`
- **Shift Word Left Logical Variable** — **SLLV**



- 类似的，`srlv`，逻辑变量右移

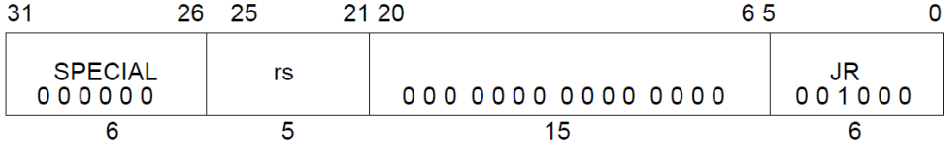
寄存器条件置1或0 Set On Less Than：`slt $t1, $t2, $t3`

- 解释：$\$t1 = (\$t2 < \$t3)$
- 格式：`slt rd, rs, rt`
- **SLT** — Set On Less Than



寄存器无条件跳转 Jump Register：`jr $t1`

- 解释：$PC = \$t1$
- 格式：`jr rs`
- **Jump Register** — **JR**

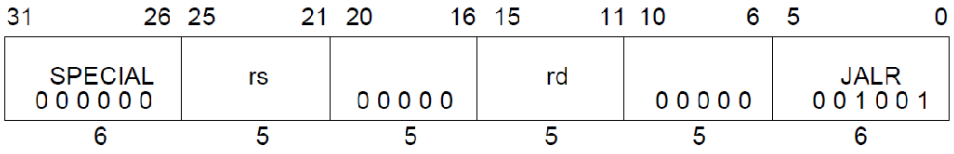


寄存器无条件跳转 Jump And Link Register：`jalr rd, rs`

- 解释：$re \leftarrow return\_addr(PC + 8), PC \leftarrow rs$

d:destination s:source

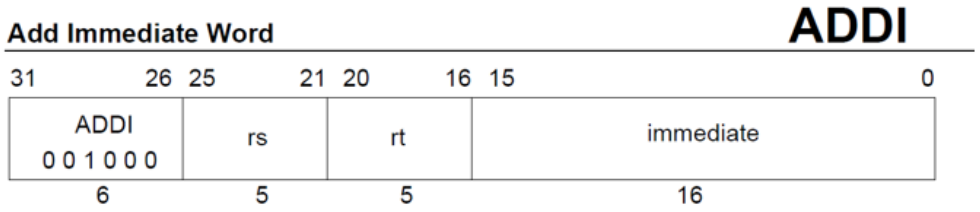
- **JALR** Jump And Link Register



I-Type指令编码示例

带立即数的算数/逻辑运算 Add Immediate Word：`addi $t1, $t2, 10`

- 解释：$rt \leftarrow rs + sign\_extend(Immediate)$
- 格式：`addi rt, rs, Immediate

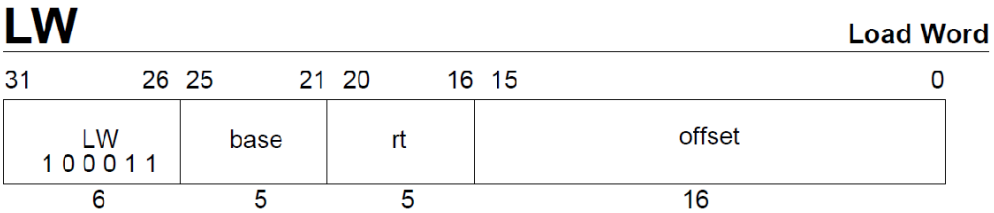


取数指令、存储指令

- 取数指令：
  - `LB` (取带符号扩展字节)
    - 符号扩展：如果最高位是1，则前面都补1，如 `LB $t2, 3($t0)`，而`$t0+3`的位置存了88(10001000)，则要扩展为0x ff ff ff 88
  - `LBU` (取不带符号字节)
  - `LH` (取带符号扩展半子(half word))
  - `LHU` (取不带符号的半字)
  - `LW` (取字 4个字节)
- 存储指令
  - `SB` (存字节)
  - `SH` (存半字)
  - `SW` (存字)

Load指令：`lw $t1, 4($t2)`

- 解释：$rt \leftarrow memory[base + offset]$
- 格式：`lw rt, offset(base)`
- 把`$t2+4`的东西存到`$t1`里

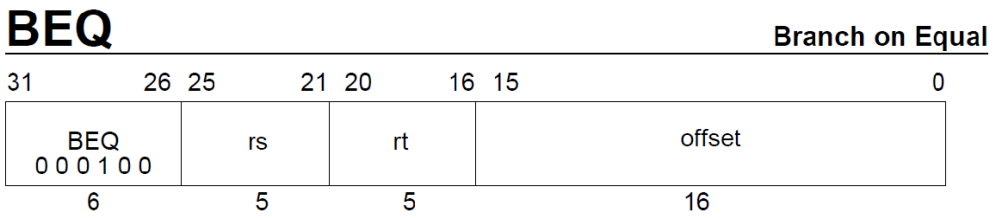


Store 指令：`sw $t1, 4($t2)`

- 解释：$memory[base + offset] \leftarrow rt$
- 格式：`sw rt, offset(base)`
- 把`$t1`的东西存到`$t2+4`里

分支指令（条件转移指令）

- Branch on Equal：`beq $t1, $t2, 8`
  - 解释：
    $if(rs = rt)\ then\ PC \leftarrow PC + sign\_extend(offset||00)$(相当于$offset *$
  - 格式：`beq rs, rt, offset`
  - 如果 $t1 = t2$，PC 跳转到 $PC + 8$

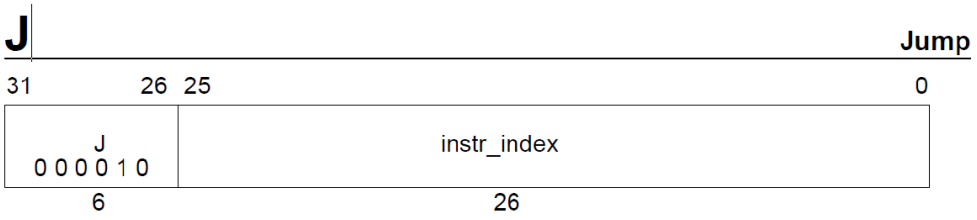


- Branch if Not Equal：`bne $t1, $t2, 8`，不等于就转移
- `bgez`：大于等于0转移
- `blez`：小于等于0转移
- `bltz`：小于0转移
- `bltzal`：小于等于0转移，带链接
- `bgezal`：大于等于0转移，带链接

J-Type指令编码示例

无条件跳转指令 Jump：`j target`

- 解释：$target$ 是地址标签，$PC \leftarrow PC_{GPRLEN...28}||instr\_index||00$
  + `GPRLEN` 指通用寄存器的长度，通常为32，因此，$PC_{GPRLEN...28}$实际上是指将PC的高4位取出。
  + `||`：这是位拼接操作符，表示将左侧的位串与右侧的位串拼接起来。
  + `instr_index`：这是`J`指令中的立即数部分，通常是26位长。这个立即数用于指定跳转的目标地址。
  + 即最终PC跳转的地址是 PC高4位 || 立即数 || 00



无条件跳转指令 Jump And Link：`jal target`

- 解释：$target$ 地址标签，$GPR[31] \leftarrow PC + 8$，
  $PC \leftarrow PC_{GPRLEN...28}||instr\_index||00$

- `GPR[31] ← PC + 8`：把PC+8存到 `$31` （`$ra`）。加8是因为MIPS指令是4字节对齐的，所以跳转指令后面的延迟槽指令占4字节，而 `jal` 指令本身也占4字节，总共8字节。
- `PC ← PCGPRLEN..28 || instr_index || 00`：最终PC跳转的地址是 PC高4位 || 立即数 || 00

SYSCALL

服务	服务号$v0	参数	返回结果
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory
exit (terminate execution)	10		
print character	11	$a0 = character to print	*See note below table*
read character	12		$v0 contains character read

如：

- `li $v0,5` `syscall` 读入一个int整数，放在 `$v0`
- `li $v0,10` `syscall` 程序正常退出

总结

Instruction	Example	Meaning	Com
add	add $1,$2,$3	$1← $2 + $3	3 operation
subtract	sub $1,$2,$3	$1← $2 – $3	3 operation
add immediate	addi $1,$2,100	$1← $2 + 100	+ constant
multiply	mult $2,$3	Hi,Lo← $2× $3	64-bit signed produ
divide	div $2,$3	Lo ← $2 ÷ $3 Hi ← $2 mod $3	Lo = quotient Hi = remainder
beq	beq $8, $9, 100	If $8==$9 Go to PC+100	Branch taken if equ
bne	bne $8, $9, 100	If $8<>$9 Go to PC+100	Branch taken if not
and	and $1,$2,$3	$1← $2 & $3	Logical AND
or	or $1,$2,$3	$1← $2 | $3	Logical OR
store	sw $3,500($4)	Mem($4+500)←$3	Store Word
load	lw $1,-30($2)	$1←Mem($2-30)	Load word
jump and link	j/jal 1000	$31=PC+4 Go to 1000	Procedure call
jump register	jr $31	Go to $31	procedure return
set on less than	slt $1,$2,$3	if ($2 < $3) than $1=1 else $1=0	

三种指令：R/I/J
常用指令- addi add slt
指令字段

寻址方式

单独访存

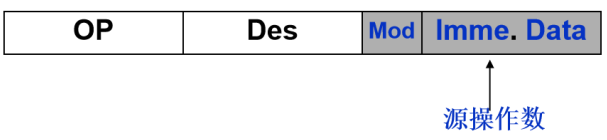
单独访存：mips的访存指令只能实现读写，不能同时进行运算

7种寻址方式

立即寻址：直接给数

❖立即寻址

➢操作数直接在指令代码中给出。



说明

➢立即寻址只能作为源操作数。

➢**Operand = Imme. Data**

➢例：**MOV AX,1000H （80X86指令， AX←1000H）**//没写H是10进制，写了H是16进制

addi $s1, $s2, 100 （MIPS指令, $s1 ← $s2+100）

寄存器直接寻址：数在寄存器里面

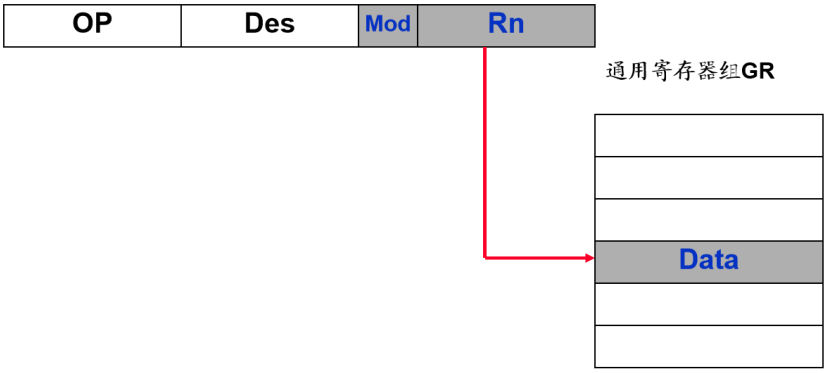
❖寄存器直接寻址

操作数在寄存器中，指令地址字段给出寄存器的地址（编码）

EA = Rn, Operand = (Rn)

例：**MOV [BX], AX** （80X86指令，将寄存器AX中的值存储到由寄存器BX所指向的内存地址中）

add $s1,$s2,$s3 （MIPS指令， **$s1←$s2+$s3**）



寄存器间接寻址：数在存储器里面，存储器的地址在寄存器里面

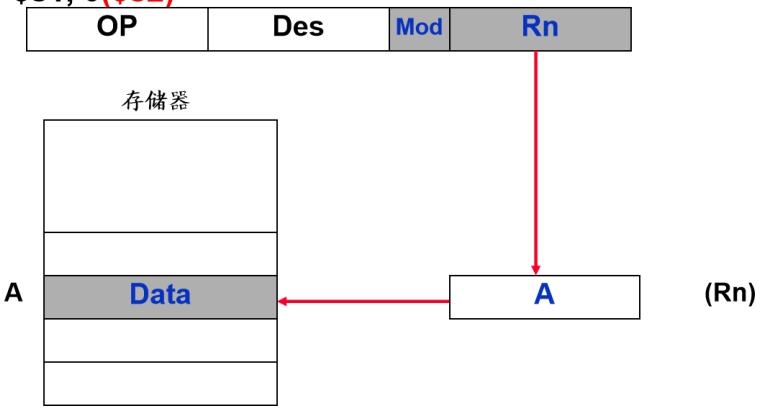
❖寄存器间接寻址

➢操作数在存储器中，指令地址字段中给出的寄存器的内容是操作数在存储器中的地址。

➢**EA = (Rn), Operand = Mem[(Rn)]**

➢例：**MOV AX, [BX]** （80X86指令, AX ←Mem[(BX)]）

lw $s1, 0($s2)



基址寻址：数的地址是寄存器的值+偏移量

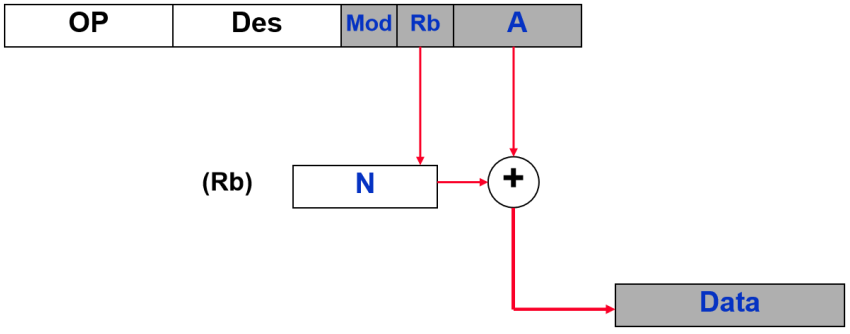
❖基址寻址

➢操作数在存储器中，指令地址字段给出一基址寄存器和一形式地址，基址寄存器的内容与形式地址之和是操作数的内存地址。

➢**EA = (Rb)+A, Operand = Mem[(Rb)+A]**

➢例：**MOV AX, 1000H[BX]** （80X86指令，AX←Mem[(Bx)+1000]）

lw $s1,100($s2) （MIPS指令，$s1 ←Mem[$s2+100]）

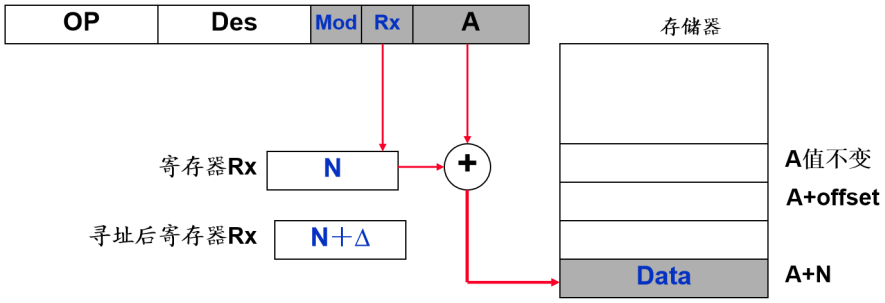


基址寻址：基址寄存器值（不变）+偏移量，较短的形式地址长度可以实现较大的存储空间的寻址

变址寻址：数的地址是寄存器的值+偏移量

❖变址寻址

- 操作数在存储器中，指令地址字段给出一变址寄存器和一形式地址，变址寄存器的内容与形式地址之和是操作数的内存地址。
- **EA = (Rx)+A, Operand = Mem[(Rx)+A]**
- 有的系统中，变址寻址完成后，变址寄存器的内容将自动进行调整。**Rx ← (Rx) + Δ**（操作数Data的字节数）
- 例：**MOV AX, 1000H[DI]** （80X86指令）　　**lb $t1, string($t0)** （MIPS指令）



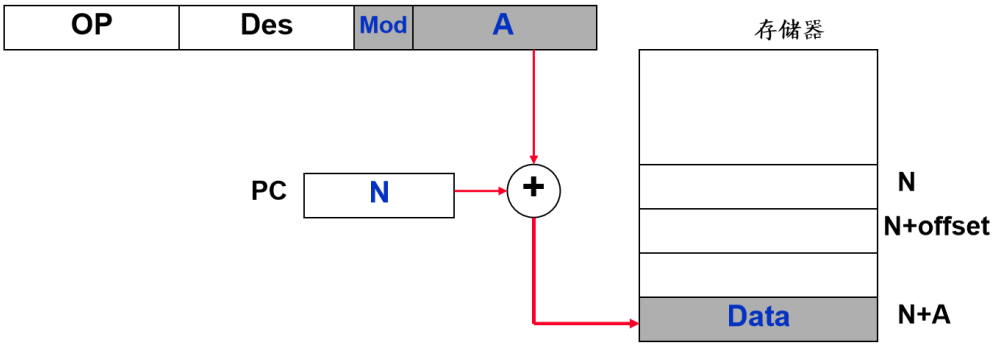
变址寻址：数组操作，字符串操作，用于循环操作，每次循环中改变寄存器的值

相对寻址：数的地址是PC寄存器的值+偏移量

❖相对寻址

- 基址寻址的特例，由程序计数器**PC**作为基址寄存器，指令中给出的形式地址作为位移量，二者之和是操作数的内存地址。
- **EA = (PC)+A, Operand = Mem[(PC)+A]**
- 例：**JNE A** （80X86指令）

  beq $s1, $s2, 100 (MIPS指令）

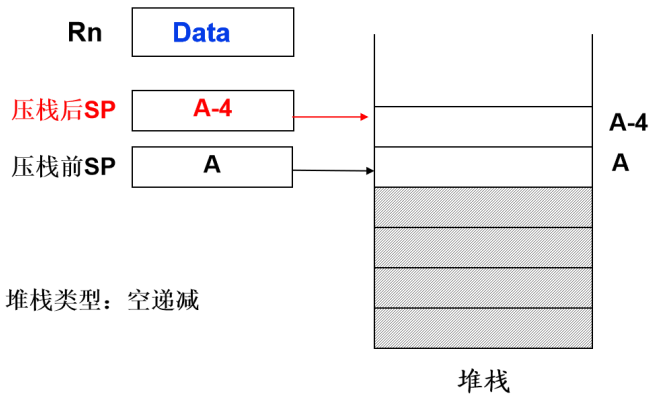


堆栈寻址：数在栈上

❖堆栈寻址

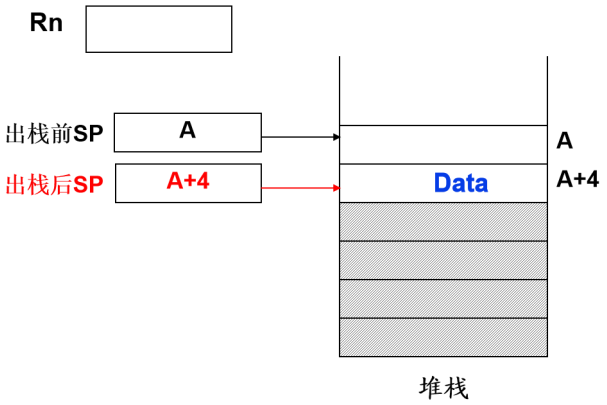
- 堆栈的结构：一段内存区域。
- 堆栈指针**(SP)**：是一个特殊寄存器部件，指向栈顶
- 压栈操作：**PUSH Rn**，假定寄存器**Rn**为**32**位寄存器

[(SP)] ← (Rn), SP ← (SP)-4



➢出栈操作：POP　Rn，假定寄存器Rn为32位寄存器

SP ← (SP) + 4，　Rn ← [(SP)]



总结

指令：OP A，R，……　　假设：A=地址字段值，R=寄存器编号，EA=有效地址，　(X)=地址X中的内容

方式	算法	主要优点	主要缺点
立即	操作数=A	指令执行快	操作数幅值有限
寄存器直接	操作数=（R）	指令执行快，指令短	地址范围有限
寄存器间接	EA=（R）	地址范围大	额外存储器访问
基址/变址/相对	EA=A+（R）	寻址灵活	寻址方式复杂
堆栈	EA=栈顶	指令短	寻址不灵活，应用有限

3.2 汇编程序

给汇编代码，知道程序功能，能转换成c代码，知道程序运行结果

MIPS汇编程序示例1：SWAP

```
swap:                                   准备 压栈 留出空间
  addi   $sp,$sp, –12     ; Make room on stack for 3 registers
  sw     $31, 8($sp)      ; Save return address
  sw     $s2, 4($sp)      ; Save registers on stack
  sw     $s3, 0($sp)
```

$a1数组下标k
$a0数组首地址v
代码实现int v[k] 与v[k+1]

```
  ....
```

```
  sll    $s2, $a1, 2      ; mulitply k by 4
  addu   $s2, $s2, $a0    ; address of v[k]
  lw     $t0, 0($s2)      ; load v[k]
  lw     $s3, 4($s2)      ; load v[k+1]
  sw     $s3, 0($s2)      ; store v[k+1] into v[k]
  sw     $t0, 4($s2)      ; store old v[k] into v[k+1]
```

```
  lw     $s3, 0($sp)      ; Restored registers from stack
  lw     $s2, 4($sp)
  lw     $31, 8($sp)      ; Restore return address
  addi   $sp,$sp, 12      ; restore top of stack
  jr     $31              ; return to place that called swap
```

恢复现场

例1：SWAP

```
swap:
# 函数作用：交换int类型v[k]和v[k+1]
# 调用函数时传入两个参数：参数1 数组首地址v，存到$a0     参数2 数组下标k，存到$a1

# 压栈
addi $sp, $sp, -12 # 留出空间主内压栈
sw $ra, 8($sp) # 函数调用返回地址
sw $s2, 4($sp)
sw $s3, 0($sp)

sll $s2, $a1, 2 # ($s2) = ($a1)*4 即 ($s2) = k*4
addu $s2, $s2, $a0 # ($s2) = ($s2)+($a0) 即 ($s2) = v[k]
lw $t0, 0($s2) # ($t0) = ($s2) 即 ($t0) = v[k]
lw $s3, 4($s2) # ($s3) = ($s2+4) 即 ($s3) = v[k+1]
sw $s3, 0($s2) # ($s2) = ($s3) 即 ($s2) = v[k] = v[k+1]
sw $t0, 4($s2) # ($s2+4) = ($t0) 即 ($s2+4) = v[k+1] = v[k]

# 退栈，恢复现场
lw $s3, 0($sp)
lw $s2, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra# 回到调用函数的位置
```

例2：符号扩展

```
.data   # 数据段
.align 2        # data内所有数据以2个字节对齐
array:.word 0×1234 0×2345 0×3456        # array是个lable

.text           # 代码段
la $t0, array   # 寄存器$t0存着array的地址

# 目前的array（设首地址为0x10010000）
# 0x10010000  0×34
# 0×10010001  0×12
# 0×10010002  0×00
# 0×10010003  0×00
# 0×10010004  0×45
# 0×10010005  0×23
# 0×10010006  0×00
# 0×10010007  0×00
# 0×10010008  0×56
# 0×10010009  0×34
# 0×10010006  0×00
# 0×10010007  0×00
```

```
li $t1, 0×88442211        # ($t1) = 0×88442211
sw $t1, 0($t0)            # ($t0) = 0×88442211
# 目前的array（设首地址为0x10010000）
# 0x10010000  0×11
# 0×10010001  0×22
# 0×10010002  0×44
# 0×10010003  0×88
# 0×10010004  0×45
# 0×10010005  0×23
# 0×10010006  0×00
# 0×10010007  0×00
# 0×10010008  0×56
# 0×10010009  0×34
# 0×10010006  0×00
# 0×10010007  0×00

lb $t2, 3($t0)            # 3($t0) 是 0x10010003
# 即$t2 = 0×ffffff88 (因为要带符号扩展，而0x88的二进制表示为1000 1000，首位为1，前面全补1（如果首位为0则前面全补0）)

sh $t2, 0($t0)            # sh取半个字符（低16位），即0xff88，存到$t0的低16位，array有如下变化
# 目前的array（设首地址为0x10010000）
# 0x10010000  0×88
# 0×10010001  0×ff
# 0×10010002  0×44
# 0×10010003  0×88
# 0×10010004  0×45
# 0×10010005  0×23
# 0×10010006  0×00
# 0×10010007  0×00
# 0×10010008  0×56
# 0×10010009  0×34
# 0×10010006  0×00
# 0×10010007  0×00

lw $t3, 0($t0)            # lw取整个word（32位），即0x8844ff88 存到$t3
```

如果$t1=0x12345678，$t3=0x12340012

例3：字符串部分逆置

```
.data
        string:.space 1000              # 存放输入字符串
        turn_over:.space 1000    # 存放翻转后的字符串
        newline:.asciiz "\n"
.text
```

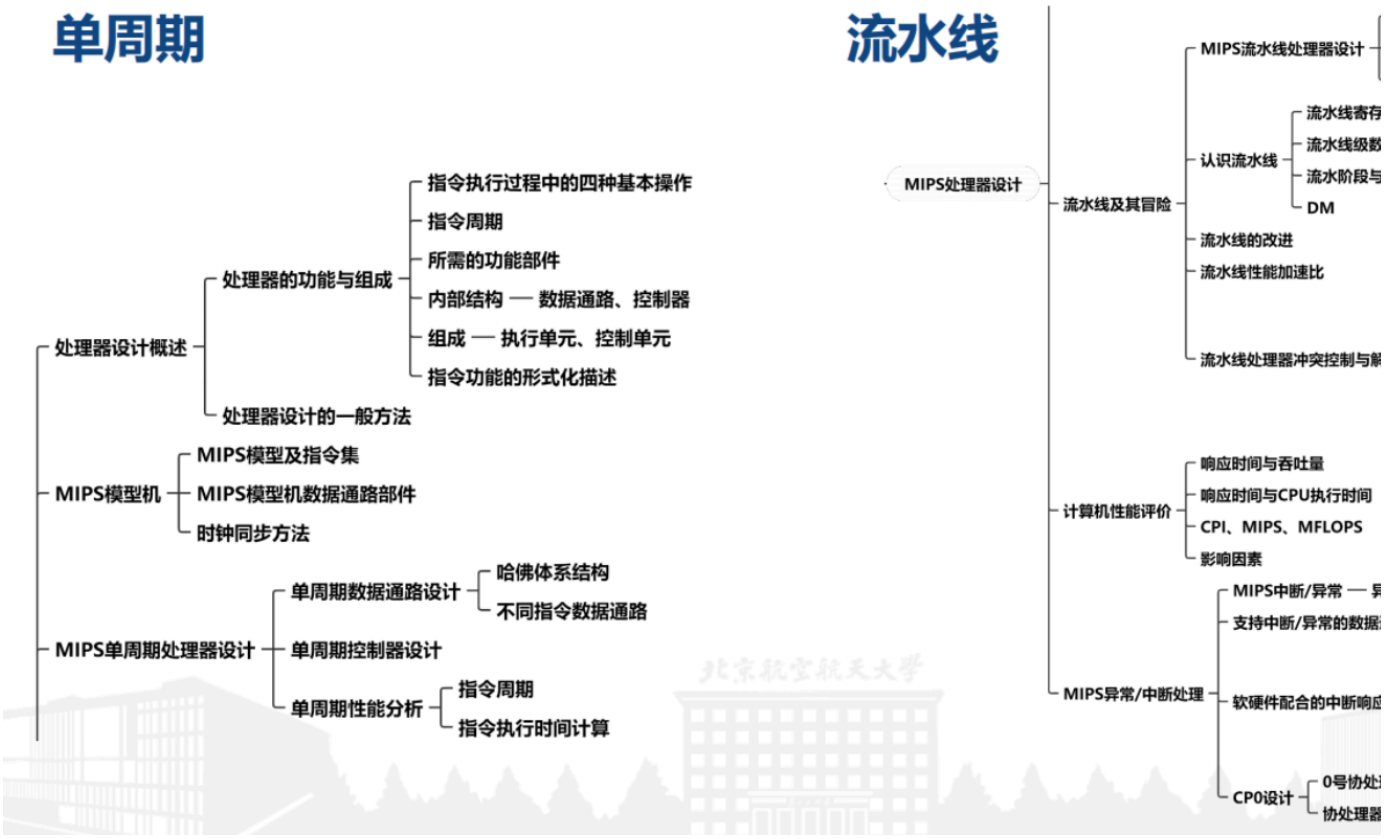
```
        li $v0, 5  # 5为读入一个int整数
        syscall
        move $s0, $v0 # $s0为总数
        li $v0, 5
        syscall
        move $s1, $v0 # $s1为起始位置
        li $v0,5
        syscall
        move $s2, $v0 # $s2为终止位置
        la $a0, string          # 把标签string代表的内存地址加载到寄存器$a0中
        addi $t0, $s0, 1 # 增加一个空字符（ASCII值为0）的位置 t0 = s0 + 1
        move $a1, $t0
        li $v0, 8               # syscall读入字符串时，参数$a0 = 字符串存放
的地址 $a1 = 读入的最大字符数量
        syscall         # 读入字符串
        sub $s3, $s2, $s1
        addi $s3, $s3, 1        # s3 = s2 - s1 + 1，为翻转数，用于loop2
        sub $s4, $s0, $s2       # s4为终止位置后的字符数，用于loop3
        move $s5, $s2   # s5为终止翻转的位置，用于loop2
        li $t0, 0       #$t0为在当前操作的字符位置
        li $t1, 0
        li $t2, 0
        li $t3, 0
        loop1: # 将起始翻转位置前的字符复制到目标数组turn_over
                beq $t1, $s1, loop2             #s1为起始翻转位置，$t1为
loop1的循环变量 if t1 == s1 jump to loop2
                lb $t4, string($t0)
                sb $t4, turn_over($t0)
                addi $t1, $t1, 1
                addi $t0, $t0, 1
                j loop1
        loop2:  # 翻转字符串
                beq $t2, $s3, loop3             # s3为翻转数，t2为loop2的
循环变量
                lb $t4, string($s5)             # s5为终止翻转的位置
                sb $t4, turn_over($t0)  # t0指向当前位置
                addi $s5, $s5, -1               # s5每次减1
                addi $t2, $t2, 1
                addi $t0, $t0, 1
                j loop2
        loop3:  # 同loop1，将终止翻转位置之后的字符复制到目标数组
                beq $t3, $s4, end                       # t3为loop3的循
环变量
                lb $t4, string($t0)
                sb $t4, turn_over($t0)
                addi $t3, $t3, 1
                addi $t0, $t0, 1
                j loop3
```

```
end:    la $a0, newline # 输出翻转后的字符串 把标签newline存到$a0中
        li $v0, 4       # 打印换行
        syscall
        la $a0, turn_over
        li $v0, 4       # 打印翻转后的字符串
        syscall
        finish:
                li $v0, 10
                syscall
```

例4：冒泡排序

```
li $t0, 0 #$t0=i, i=0
li $s0, (len-1) # s0=数组的元素个数-1
for_i:
        beq $t0, $s0,end_for_i #bge  is ok as well.
        li $t1, 0    #j=0
        sub $s1, $s0, $t0 #$s1=len – i - 1
        for_j:
                beq $t1, $s1, end_for_j  #bge  is ok as well.
                addi $t2, $t1, 1 #$t2=j+1，比较的第二个元素
                sll $t3, $t1, 2 #$t3为arr[j]的地址
                sll $t4, $t2, 2 #$t4为arr[j+1]的地址
                lw $t5, arr($t3) #$t5=arr[j]
                lw $t6, arr($t4) #$t6=arr[j+1]
                ble $t5, $t6, end_j  #if arr[j] ≤ arr[j+1], 跳过swap
                sw $t5, arr($t4) #swap $t5 and $t6
                sw $t6, arr($t3)
                end_j:
                        addi $t1, $t1, 1 #j++
                        j for_j
        end_for_j:
                addi $t0, $t0, 1 #i++
                j for_i
end_for_i:   ……
```

4.MIPS处理器设计

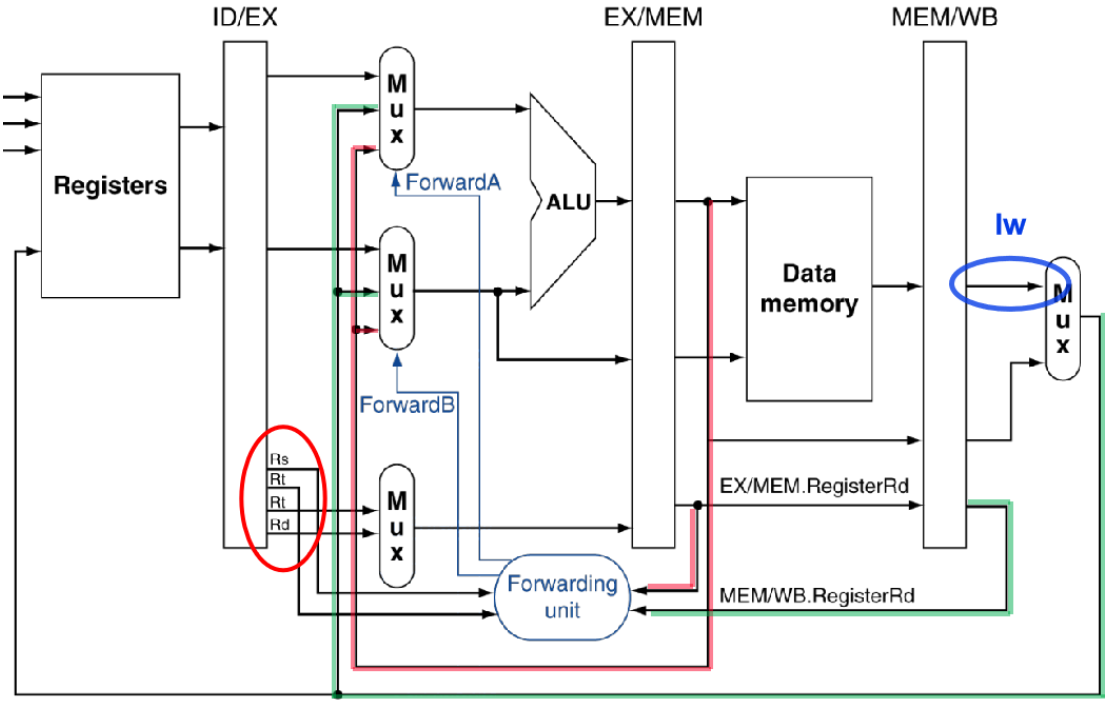


4.1 单周期CPU

4.2 流水线CPU(15分)

r-r指令

- 相邻指令(12)数据冲突-红线



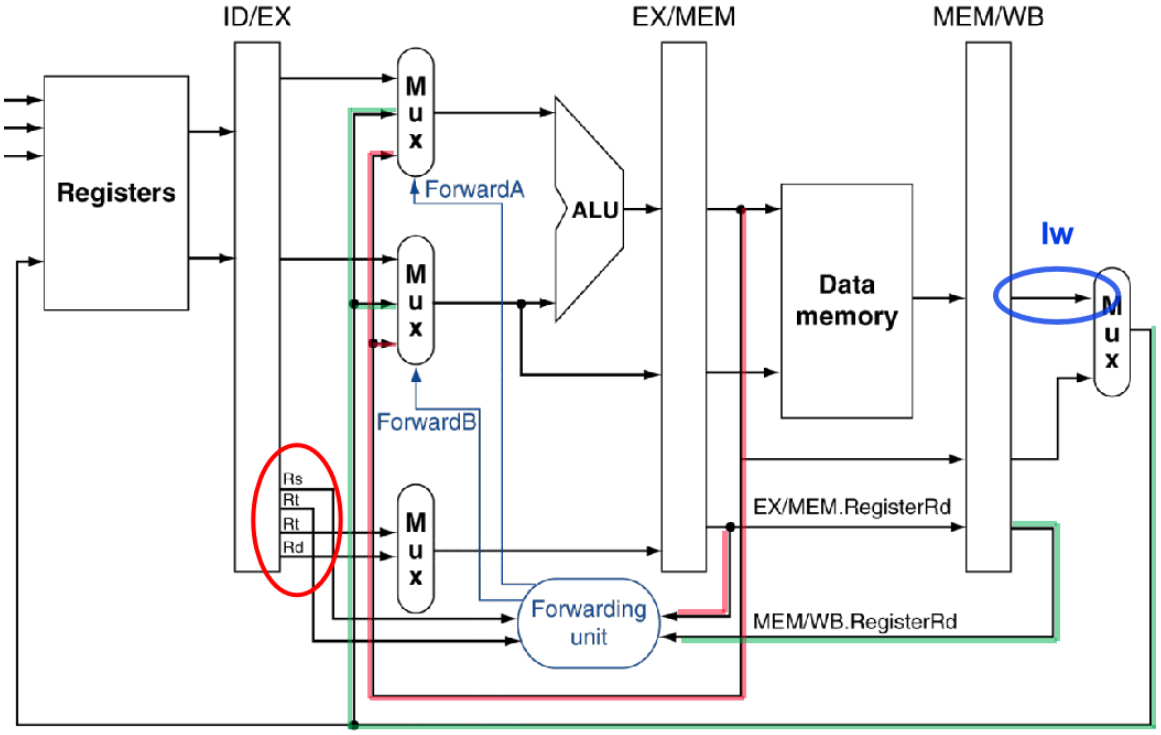
```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
```

前一条指令要写寄存器且
寄存器不为零且
前一条指令的目标寄存器=后一条指令的rs

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

前一条指令要写寄存器且
寄存器不为零且
前一条指令的目标寄存器=后一条指令的rt

- 间隔一个指令(13)数据冲突-绿线
  13 23都有冲突时，从2转发



= 应该是=而非\neq

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
```

若要从MEM/WB转发，需要满足没有从EX/MEM的转发

=

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

lw-r / lw-sw且sw用lw的结果作地址计算

从MEM/WB后面转发到ALU的MUX前面，并在lw后面加上一个nop

ID/EX存储器读控制信号为真，即上一条是lw指令

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
```

上一条指令的目的寄存器和下一条指令的源寄存器相同

lw-sw且sw用lw的结果作为存储的地址

加一个MEM/WB到DM的输入端的转发即可

其他

- 加入流水线寄存器
- 流水线的性能 —— 提高吞吐率、指令级并行
- 流水线时钟周期的长度——决定于**最长的流水段的时间长度**
- 流水线**不能缩短单条指令的流水时间**
- 冒险——结构 数据 控制
- 无论什么情况，只要间隔**两个指令**就一定不会数据冲突
- 计算机性能评价 MIPS CPI
- **根据不同转发方式判断数据冒险，需要加几个nop（阻塞）**
- **给出指令序列——指出所有数据冲突（根据分数找冲突）**
- **给出指令序列——指出所有暂停**
- **给出指令序列——时钟周期数 n+4+nop个数**
- **调整指令顺序——达到完备流水线而无需加nop**

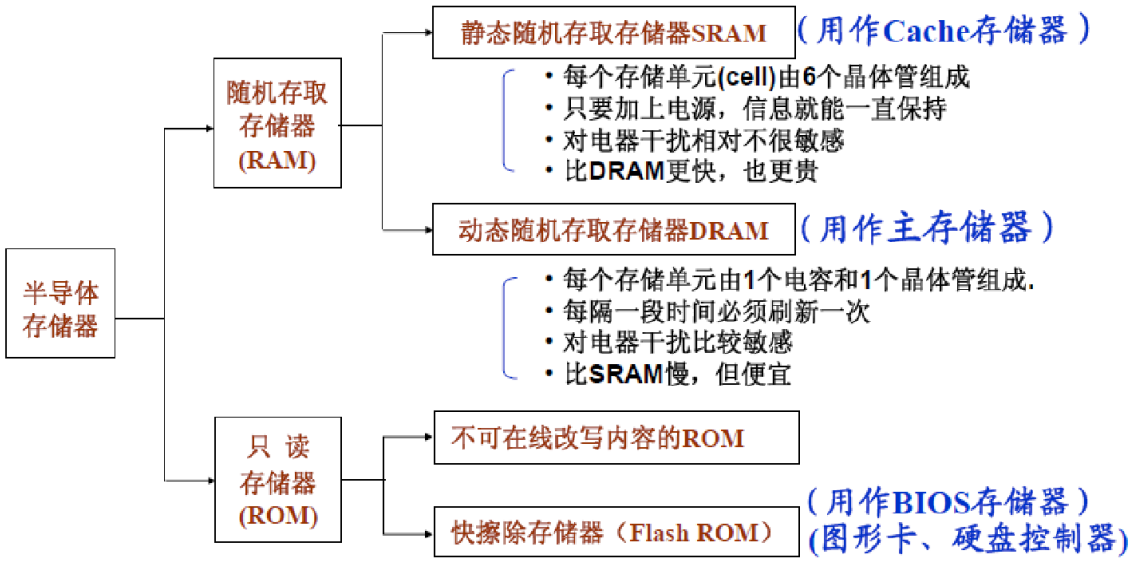
5.主存储器

5.1 存储系统概述

- RAM：随机访问存储器

- SRAM：静态xx，相对动态而言，集成度低，但不必刷新。
- DRAM：动态xx，需要刷新，相对而言，集成度高。

- ROM：只读存储器



- 性能指标
  - 访问时间（Access Time）：T_A
    - 随机访问存储器：访问时间指读或写操作所用时间，即**从给定地址到存储器完成读或写操作**所需时间。
    - 其他类型：指将读写机构定位到目标位置所需的时间。
  - 存储周期（Cycle Time）：T_C
    仅对RAM而言，指**两次访问存储单元间**的最小时间间隔。
    $T_C > T_A$
  - 带宽（Bandwidth） / 数据传输率（data Rate）：**单位时间内可以传输的数据量**，单位为bps(位每秒)或BPS(字节每秒)
    - 一般的随机访问存储器读写频率 $f = \frac{1}{T_C}$ 带宽 $= f * 字宽$
    - 其他类型：$T_N = T_A + \frac{N}{R}$
      T_N：读写 N Bits所需的平均时间
      T_A：访问时间
      N：N Bits
      R：存储部件的数据传输率（ bits /s)

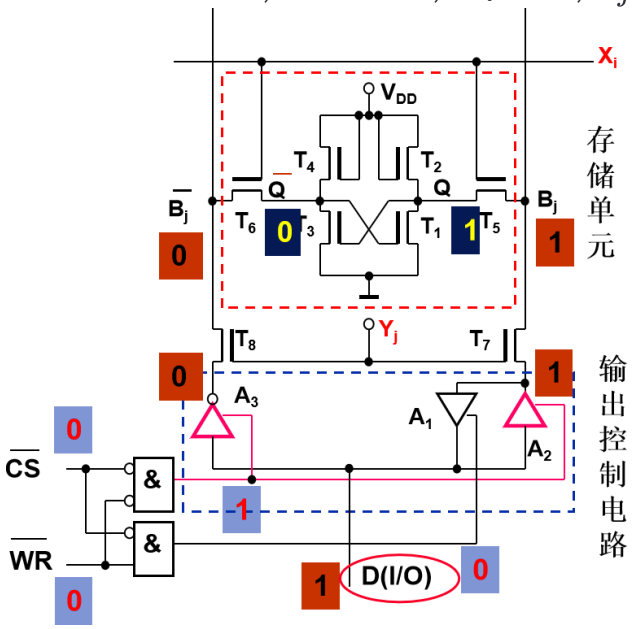
5.2 存储单元电路

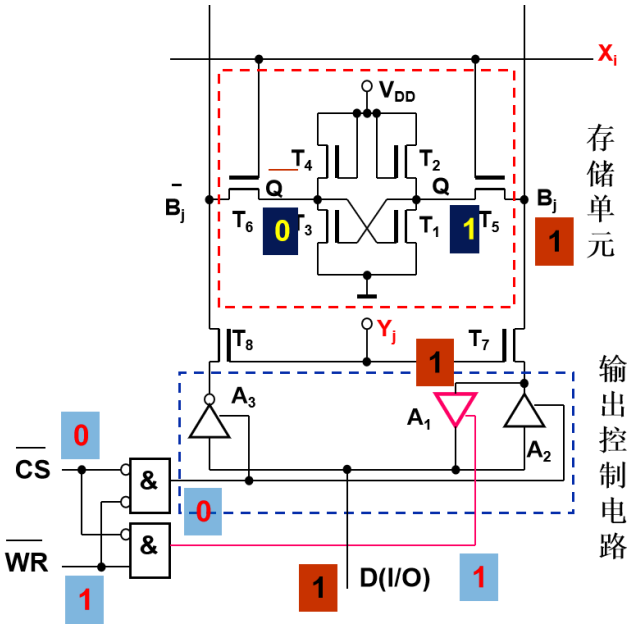
SRAM

D 为输入输出的地方， Q 为存的东西

- 写操作：$\overline{CS} = 0, \overline{WR} = 0, X_i = 1, Y_j = 1$

- 读操作：$\overline{CS}=0,\overline{WR}=1,X_i=1,Y_j=1$



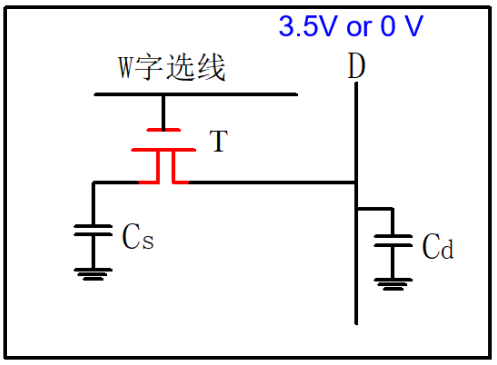


DRAM

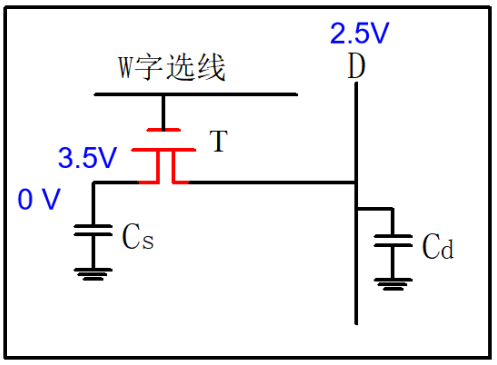
Cs 为存的东西，有电荷表示‘1’，没电荷表示‘0’。保持状态：字选线低电平，T 截止，理论上内部保持稳定状态。

- 写操作：D 线加高电平（1，即3.5V）或低电平（0），字选择线置高电平，T导通；
  - 写1时，D线高电平，对Cs充电；

- 写0时，D线低电平，Cs放电；



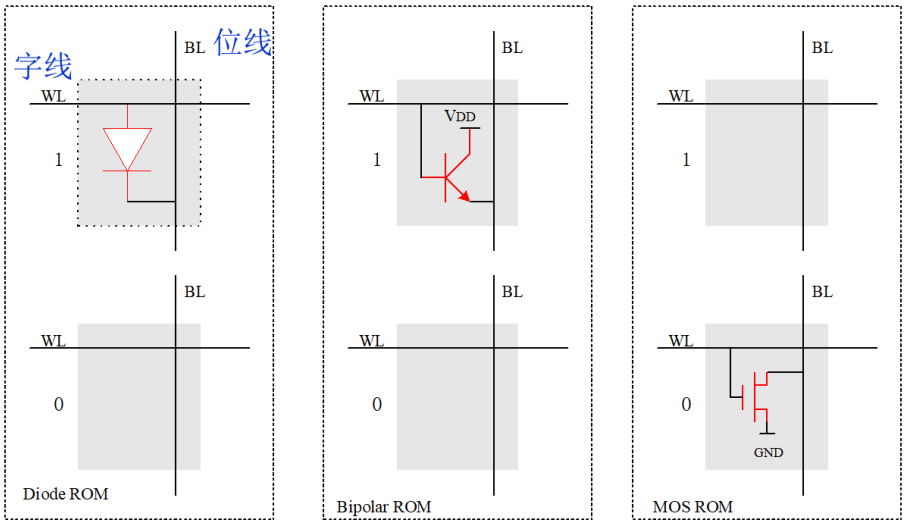
- 读操作：D 线先预充电到 Vpre=2.5V，然后字选线高电平，T导通
  - 若电路保存信息1，Vcs=3.5V，电流方向从**单元电路内部向外**；
  - 若电路保存信息 0，Vcs=0.0V，电流方向从**外向单元电路内部**；
  - 因此根据数据线上**电流的方向**可判断单元电路保存的是1还是0。读出过程实际上是Cs与Cd上的电荷重新分配的过程，也是Cs与Cd上的电压重新调整的过程。Cd上的电压，即是D线上的电压。



> 在保存二进制信息“ 1 ”的状态下，Cs有电荷，但Cs存在**漏电流**，Cs上的电荷会逐渐消失，状态不能长久保持 ，在电荷泄漏到威胁所保存的数据性质之前，需要补充所泄漏的电荷，以保持数据性质不变。这种电荷的补充称之为**刷新**（或再生）。

ROM

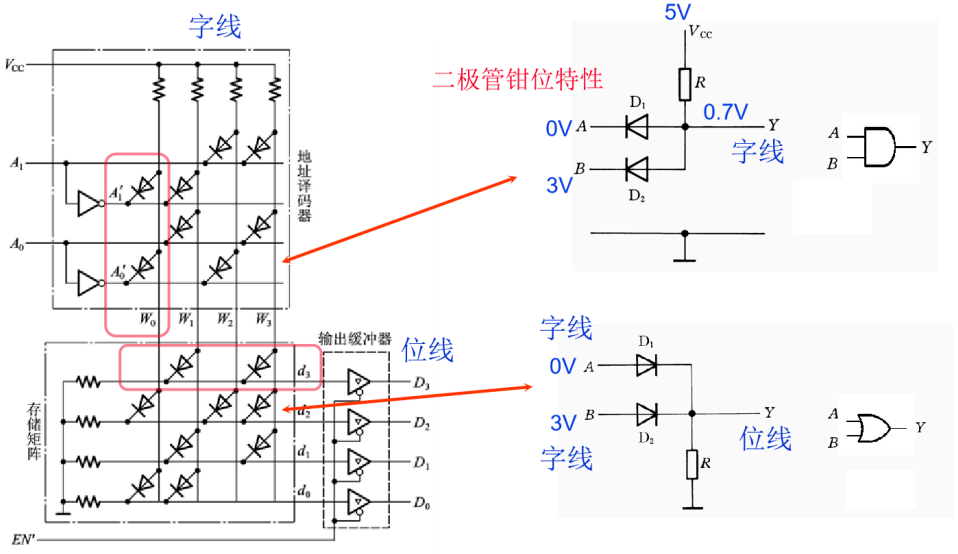
固定掩膜ROM单元电路

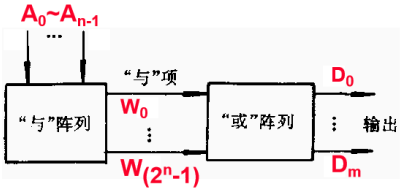


含二级管的电路表示1，不含电路表示0

含三级管的电路表示1，不含电路表示0

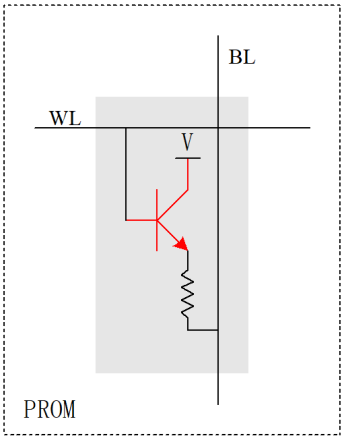
含MOS管的电路表示0，不含电路表示1





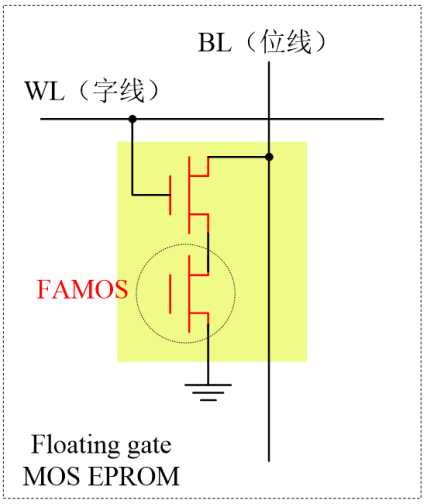
地　址		数　　据			
A_1	A_0	d_3	d_2	d_1	d_0
0	0	0	1	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	1	1	1	1	0

可编程的PROM单元电路 (programme)



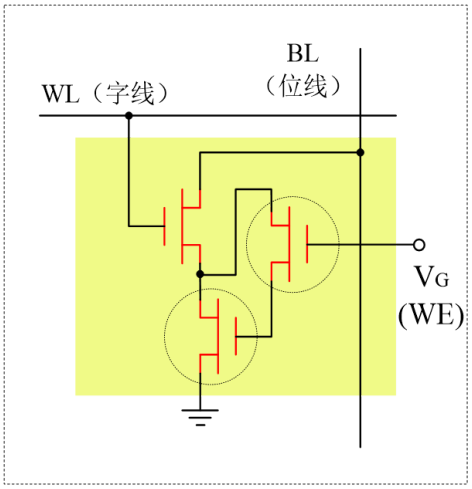
- 出厂时所有位均为**1**。
- 编程时（写入数据，需要特定编程器），对写**0**的单元加入特定的大电流，熔丝被烧断，变为另一种表示**0**的状态，且不可恢复。
- 工作时，加入正常电路。
- 一旦写入则不能更改。

紫外线擦除可编程的EPROM单元电路



- 出厂时所有位均为 1，FAMOS（浮空栅极MOS）G极无电荷，处于截止状态。
- 编程时（写入数据），对写0的单元加入特定的电压，FAMOS上的G极与D极被瞬时击穿，大量电子聚集到G极上，撤销编程电压后，G极上的聚集的电子不能越过隔离层，FAMOS导通，表示0。
- 工作时，加入正常电压，FAMOS 的状态维持不变。
- 擦除时，用紫外线照射，FAMOS聚集在G极上的电子获得能量，越过隔离层泄漏，FAMOS恢复截止状态。

电擦除可编程的EEPROM单元电路

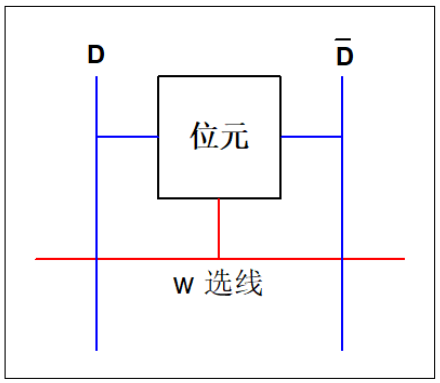


- 与EPROM相似，它是在EPROM基本单元电路的浮空栅的上面再生成一个浮空栅，前者称为第一级浮空栅，后者称为第二级浮空栅。第二级浮空栅引出一个电极，接某一电压V_G。
- 若V_G为正电压，第一浮空栅极与漏极之间产生隧道效应，使电子注入第一浮空栅极，即编程写入（写“0”）。
- 若使V_G为负电压，强使第一级浮空栅极的电子散失，即擦除（擦除为“1”）。擦除后可重新写入。

存储单元的符号表示

- 可存储1位二进制代码
  8个存储单元构成1个字节

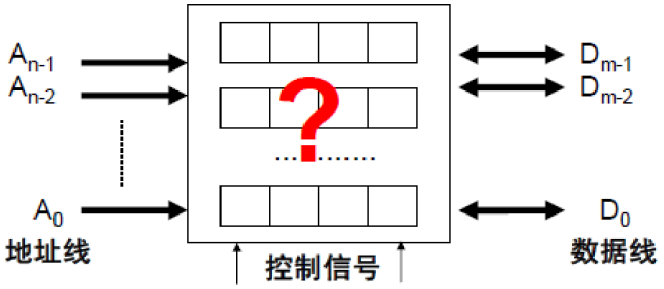
32个存储单元构成1个32位字



5.3 存储器芯片结构

存储芯片容量的描述: 2^n x m （字单元数 x 每个字单元的位数）

- 存储位元： 2^n x m 个
- 地址线： n位 → 2^n个字单元， $A_{n-1}..A_0$ 单向
- 数据线： m位 → m位/字单元， $D_{m-1}..D_0$ 双向

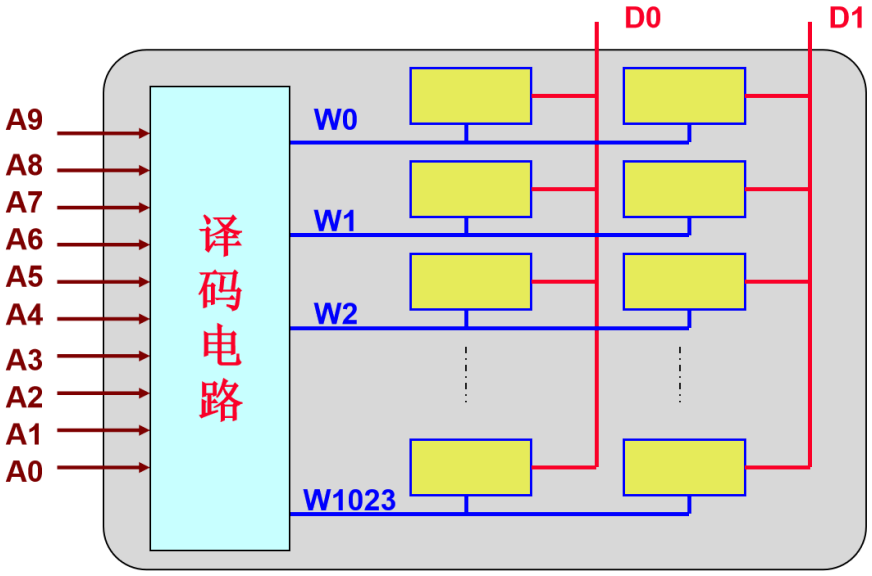


一维地址结构(线选法) / 二维地址结构（重合法）

如 1k*2：1024个字单元，每个字单元2位。任意时刻只能访问一个字单元，即每次读写的数据位数是2位
有 2k 个存储位元
需要 10 条地址线（2^10 = 1024个字单元）
需要 2 条数据线（每个字单元2位）

一维地址结构

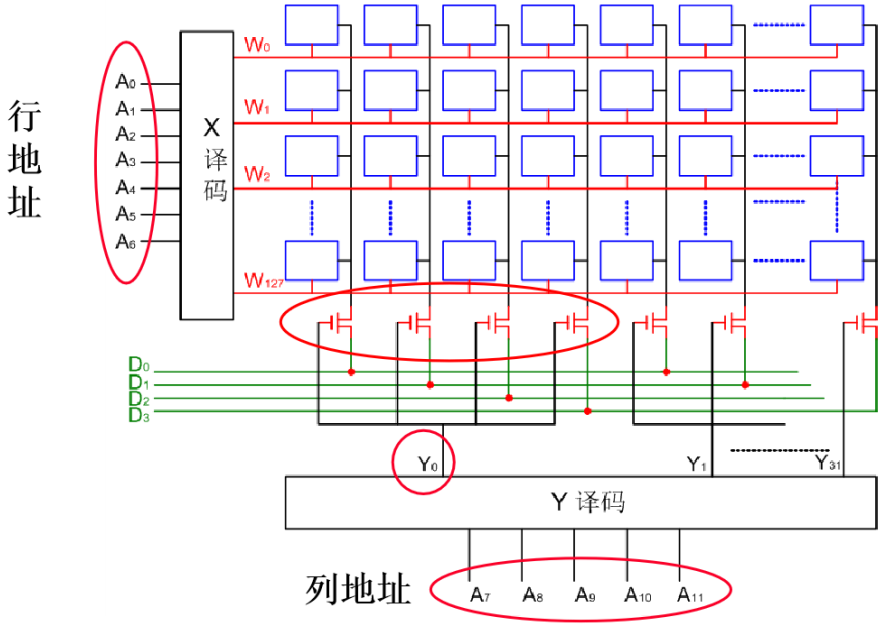
1024×2 ： 1024 个字单元，每个字单元 2 个二进制位。



二维地址结构

芯片示例：4096 × 4（4096 = 2^{12}个字，每个字 4 位）

- 4096×4 = 2^{14} 个位单元
- 存储矩阵： 2^7×2^7 (128行×128列)
- 行译码：行地址7位，一行含32个字共128位
- 列译码：列地址5位

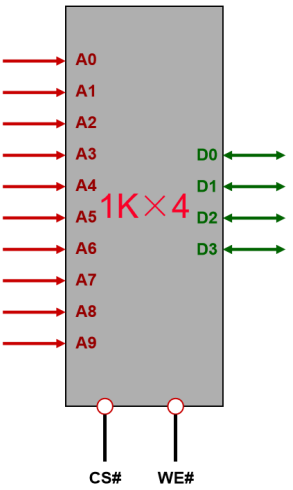


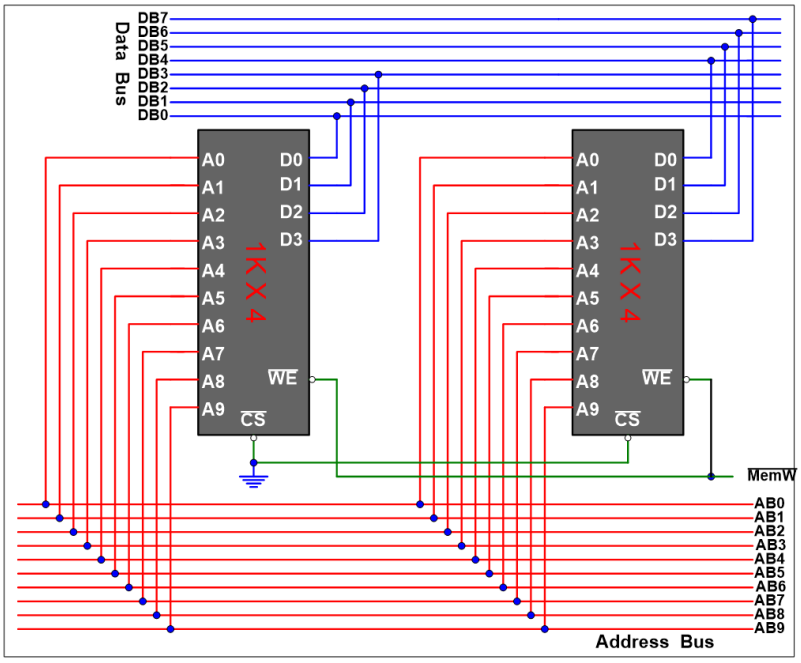
5.4 存储器扩展

5.4.1 位扩展

例：1Kx4 SRAM芯片构成1Kx8的存储器

- 1K×4 芯片管脚：
  - 10个地址管脚 A9～A0
  - 4个数据管脚 D3～D0
  - 1个片选输入管脚 CS＃
  - 1个读写控制管脚 WE#
  - 芯片地址空间：000H～3FF H
- CPU访问存储器需提供：
  - 地址总线10根：AB9～AB0
  - 数据总线8根：DB7～DB0
  - 读写控制信号：MemW
  - 存储器地址空间：000H～3FF H
- 需要芯片：（1K×8) / (1K×4）= 2片
  - 地址管脚：都连接到AB9～AB0
  - 数据管脚：分别连接到 DB7～DB4和 DB3～DB0
  - 芯片读写控制管脚：连接MemW

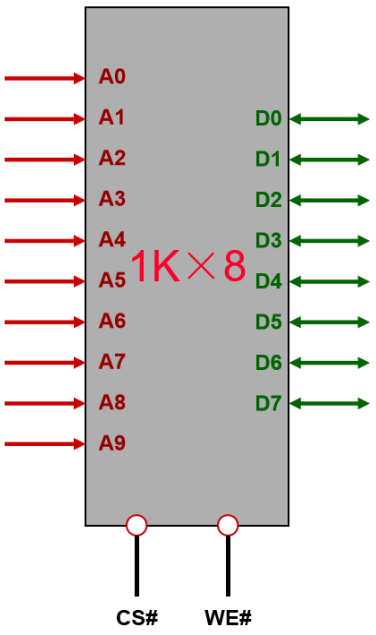




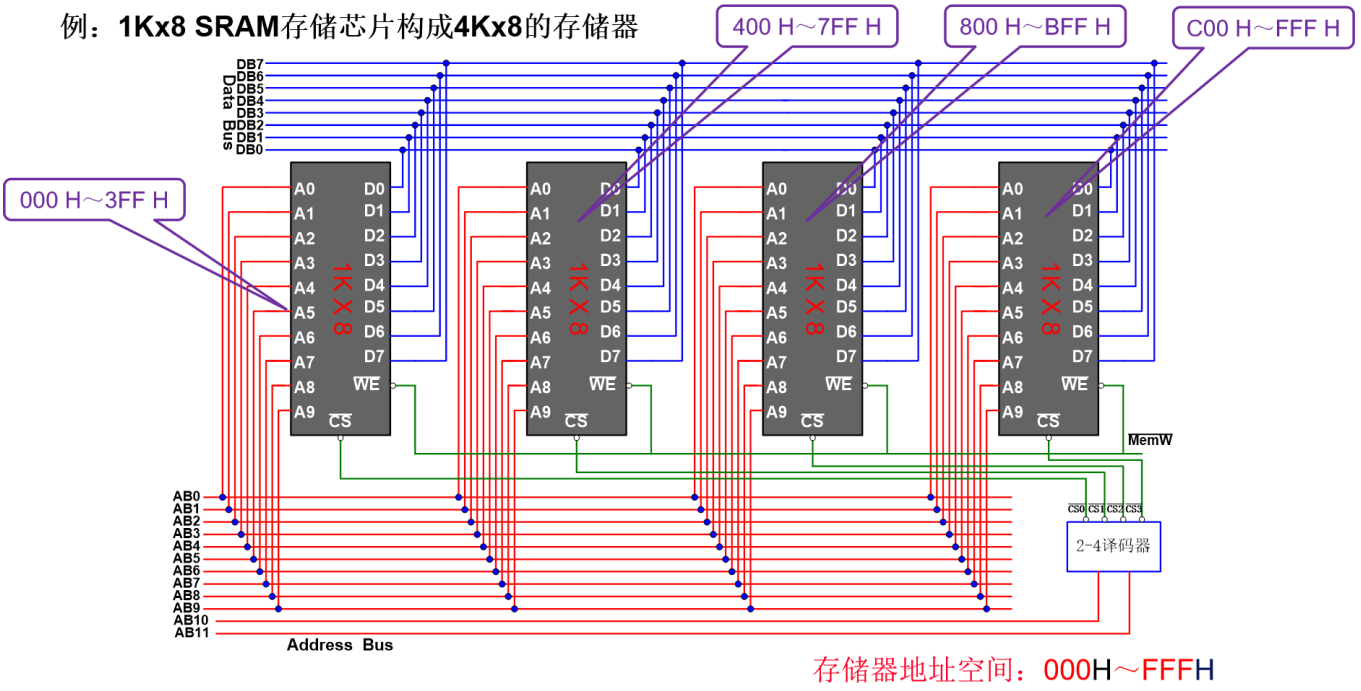
5.4.2 字扩展

例：1Kx8 SRAM芯片构成4Kx8的存储器

- 1K×8 芯片管脚：
  - 10个地址管脚 A9～A0
  - 8个数据管脚 D7～D0
  - 1个片选输入管脚 CS＃
  - 1个读写控制管脚WE#
  - 芯片地址空间：000H～3FF H
- CPU访问存储器需提供：
  - 地址总线12根：AB11～AB0
  - 数据总线8根：DB7～DB0
  - 读写控制信号：MemW
  - 存储器地址空间：000H～FFF H
- 需要芯片数：（4K×8) / (1K×8）= 4片
  - 地址管脚：都连接到AB9～AB0
  - 数据管脚：都连接到 DB7～DB0
  - 芯片读写控制管脚：连接MemW
- 一个2-4译码器产生4个片选信号
  - 译码器输入：AB11~AB10
  - 译码器输出：分别接4个芯片片选管脚



例：1Kx8 SRAM存储芯片构成4Kx8的存储器

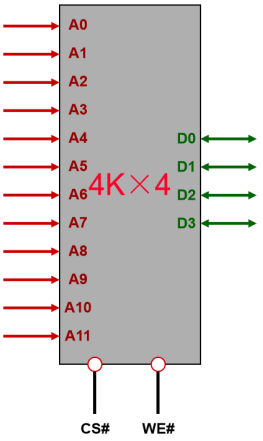


存储器地址空间：000H～FFFH

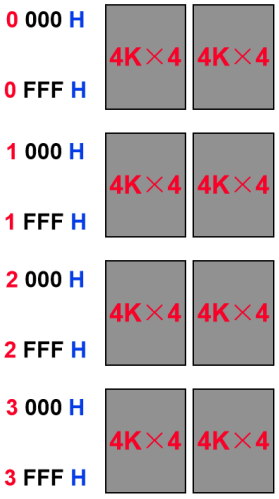
5.4.3 混合扩展——字位扩展

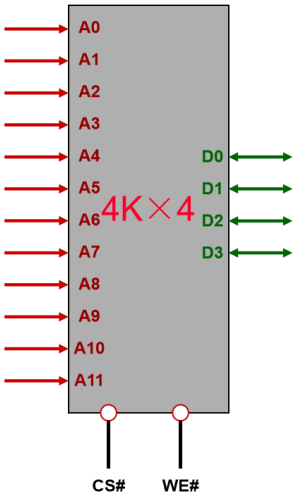
例：4Kx4 SRAM存储芯片构成16Kx8的存储器

- 4K×4芯片：
  - 12个地址管脚 A11～A0
  - 4个数据管脚 D3～D0
  - 1个片选输入管脚 CS＃
  - 1个读写控制管脚 WE#
  - 芯片地址空间：000H～FFF H
- CPU向存储器提供：
  - 地址总线14根：AB13～AB0
  - 数据总线8根：DB7～DB0
  - 读写控制信号：MemW
  - 存储器地址空间：0000H～3FFF H
- 需要芯片数：（16K×8) / (4K×4）= 4×2 = 8片
  - 分4组（字扩展），每组2个芯片（位扩展）
- 一个2-4译码器产生4个片选信号
  - 译码器输入：AB13~AB12
  - 译码器输出：分别接4组芯片片选管脚

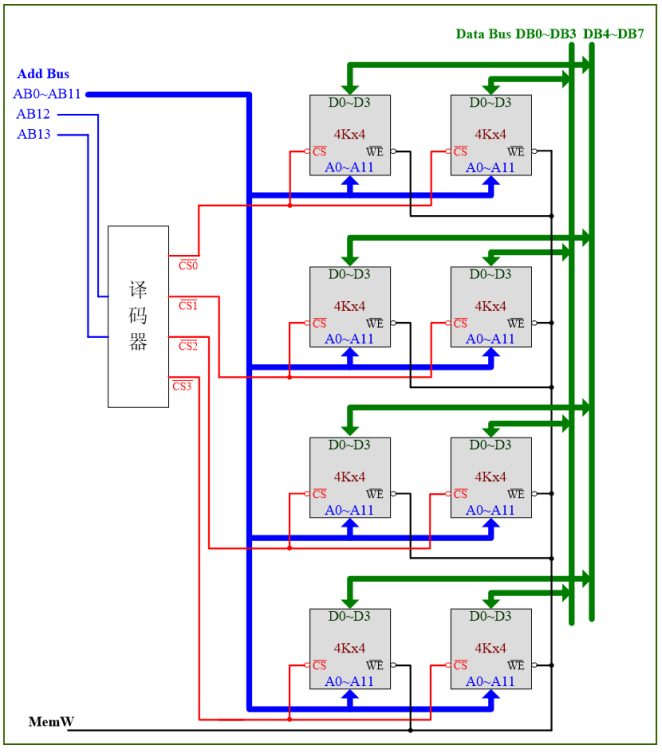


4Kx4 SRAM存储芯片构成16Kx8的存储器地址空间划分





4Kx4 SRAM存储芯片构成16Kx8的存储器连接图



5.5 存储器扩展的例题

CPU与主存的连接（示例）

CPU地址线A15～A0，数据线D7～D0，$\overline{WR}$为读/写信号，$\overline{MREQ}$为访存请求信号。0000H～3FFFH为系统程序区，4000H～FFFFH为用户程序区。用8K×4位ROM芯片和16K×8位RAM芯片构成该存储器，要求：地址译码方案，并将ROM芯片、RAM芯片与CPU连接。

解：因为0000H～3FFFH为系统程序区，ROM区高两位总是00，低14位为全译码。

ROM区大小为：2^{14}×8位=16K×8位=16KB

ROM芯片数为：16K×8位 / 8K×4位 = 2×2 = 4 字方向扩展2倍，位方向扩展2倍

ROM芯片内地址位数为13位，连到CPU低13位地址线A12～A0

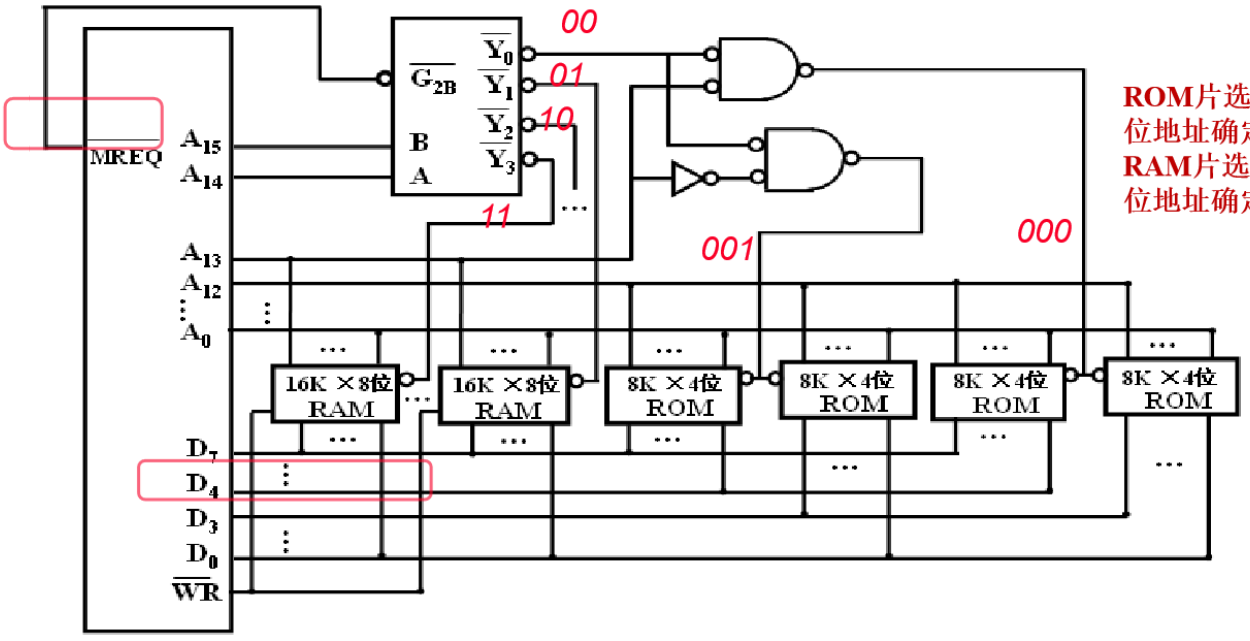
因为4000H～FFFFH为用户程序区，RAM区高两位是01、10、11，低14位为全译码。

RAM区大小为：3×2^{14}×8位=3×16K×8位= 48KB

RAM芯片数为：48K×8位 / 16K×8位 = 3×1 = 3，字方向上扩展3倍，位方向上不扩展。

RAM芯片内地址位数为14位，连到CPU低14位地址线A13～A0。

16位地址：2^{16}=64K

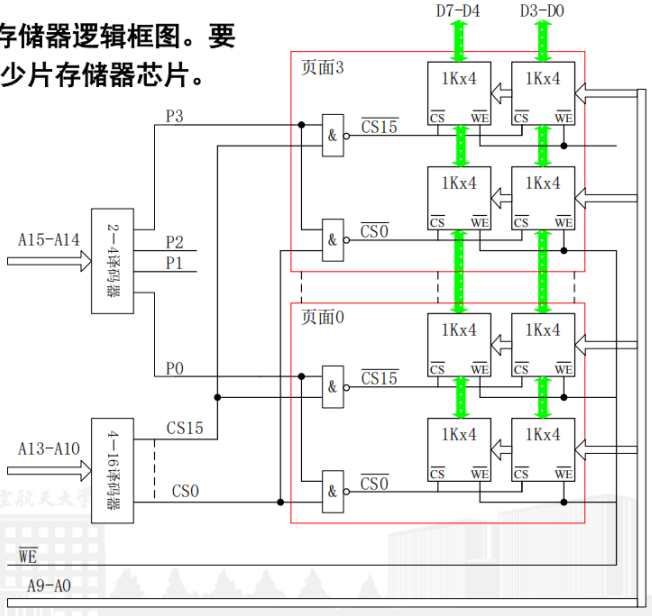


ROM片选由高3位地址确定。

RAM片选由高2位地址确定。

❖画出1K×4位的存储器芯片组成一个64K×8位的存储器逻辑框图。要求64K分成4个页面，每个页面分16组，指出共需多少片存储器芯片。

- 1K×4芯片组成64K×8存储器
- 需要进行字扩展和位扩展
- 字扩展：64K/1K = 64
- 位扩展：8/4 =2
- 芯片数：(64K×8)/(1K×4)=128
- 将64K字空间分为4个页面
- 整个存储器分成4个16K×8的小存储器



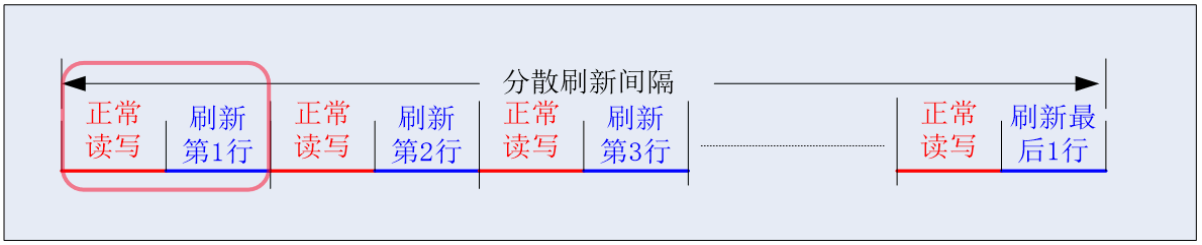
5.6 DRAM的刷新

- 刷新操作：读操作（**DRAM的读过程就是刷新过程**）
- 按行刷新、所有芯片同时进行

- 刷新操作与CPU访问内存分开进行
- 刷新周期：2ms, 4ms, 8ms, …, 64ms
- 刷新地址及刷新地址计数器

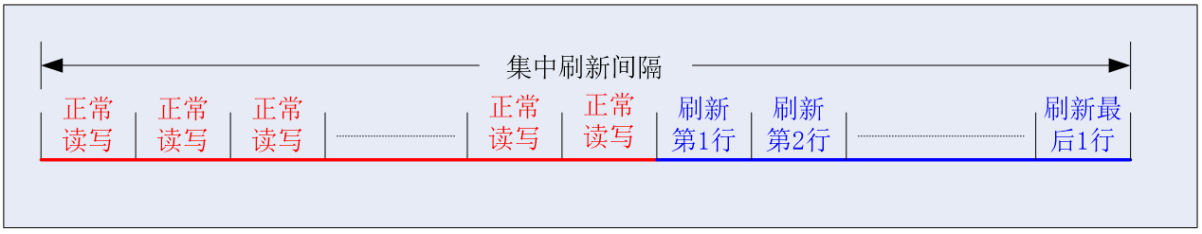
分散刷新方式

一个存储周期分为两段: 前一段用于正常读写，后一段用于刷新操作。一个存储周期刷新1行，下一存储周期刷新另一行，直至最后1行后，又开始刷新第1行。
存储周期加长，**效率降低，很少使用。**
分散刷新间隔 = 刷新行数×存储周期 = 刷新周期



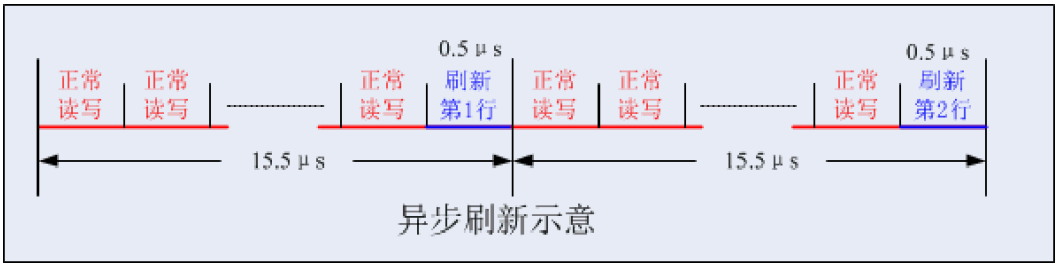
集中刷新方式

将存储器访问周期分成两部分，在一个时间段内刷新存储器所有行，另一个时间段CPU访问内存，刷新电路不工作。
集中**刷新时间长**，此时存储器**不能正常读写**（访存死区）。**很少使用该方法。**
集中刷新间隔 = 刷新周期



异步刷新方式

结合前两种方式，保证**在一个刷新周期内将存储芯片内的所有行刷新一遍，且只刷新一遍。**
异步刷新间隔 = 刷新周期
以128行为例，在2ms时间内必须轮流对每一行刷新一次，即每隔2ms/128=15.5μs刷新一行。
这时假定读/写与刷新操作时间都为0.5μs，则可用**前15μs进行正常读/写操作，最后0.5μs完成刷新操作。**
存储周期不变；没有访问死区



6.高速缓存

局部性原理（空间+时间）

大量典型程序的运行情况分析结果表明，无论是存取指令或存取数据所**访问的存储单元都趋于聚集在一个较小的连续存储区域中。**

- 空间局部性(spatial locality)：刚被访问过的存储单元的**临近单位**可能不久被访问。
- 时间局部性(temporal locality)：刚被访问过的存储单元可能**不久**又将被访问。

为什么要高速缓存(SRAM)

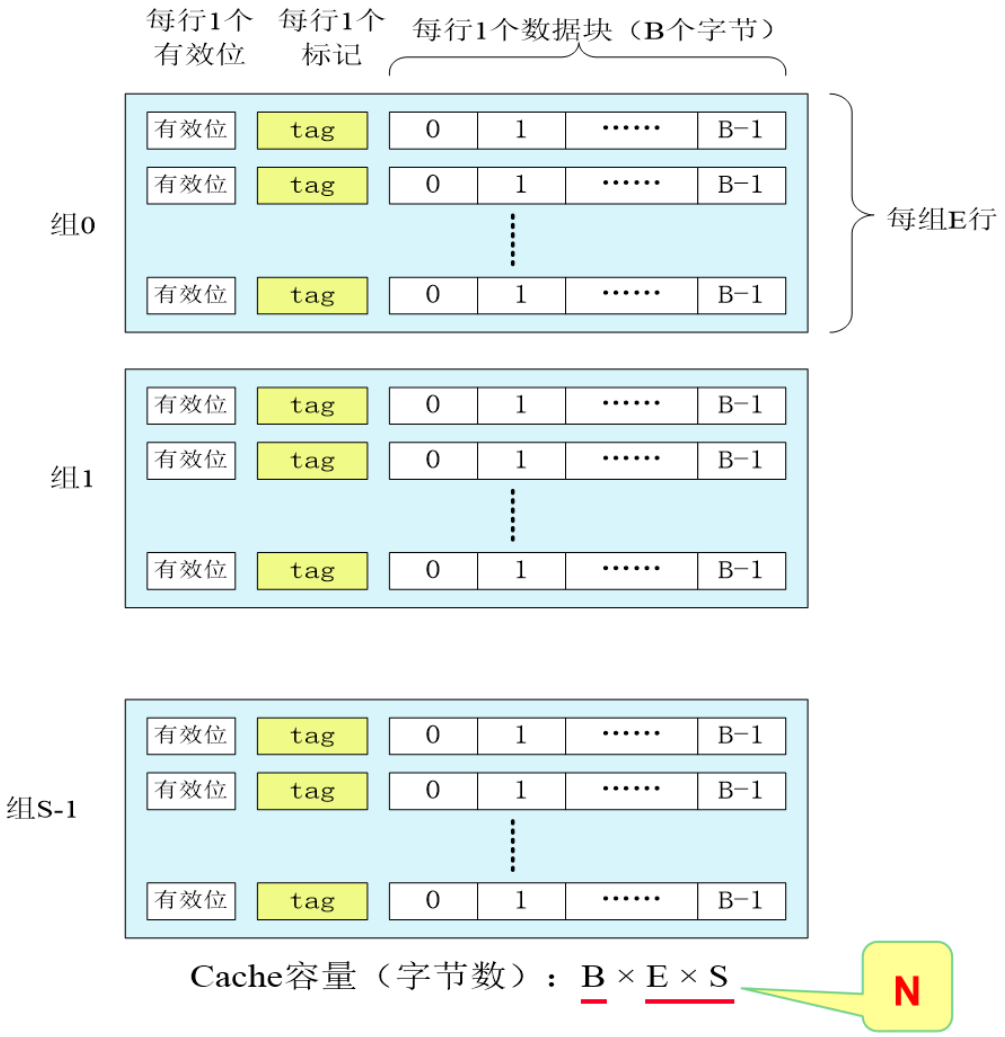
CPU太快，DRAM太慢，会造成**CPU资源的浪费**；需要比DRAM快的存储单元电路（但是SRAM更贵）

重要概念！！！

- **行**（line ）：Cache中 一个block及其 tag、valid bit构成1行。
- **组**（set）：若干行构成一个组，地址比较一般能在**组内各行间同时进行。**
- **路**（way）：**每组的行数**，Cache相关联的等级，每一路具有独立的地址比较机构，各路地址比较能同时进行（一般与组结合），路数即指一组内的块数。

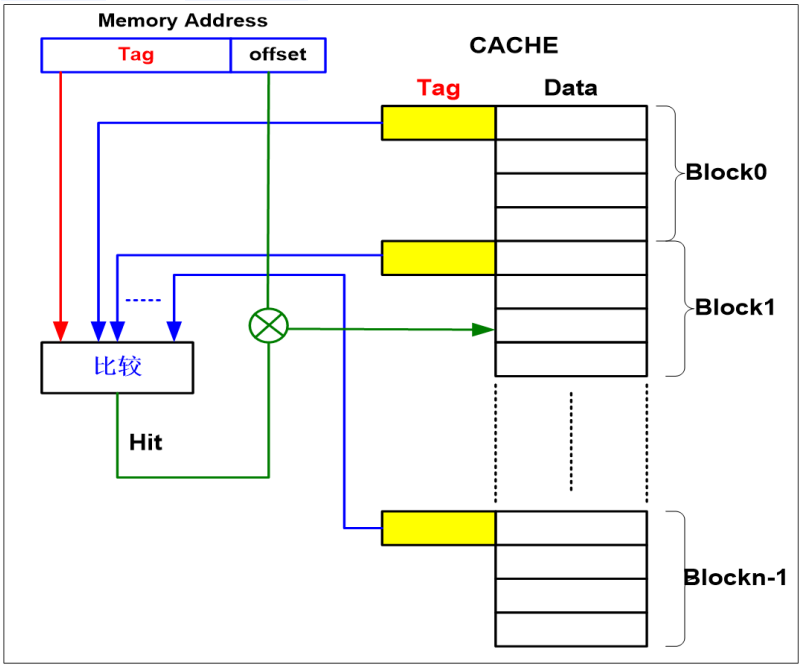
Cache结构示意！！！

- 总共N行
- 分S组
- 每组E行（Block, 数据块）N = S * E
- 每数据块（每行）包含B个字节
  Cache的容量：$N*(B*8+tag位+有效位)$bits



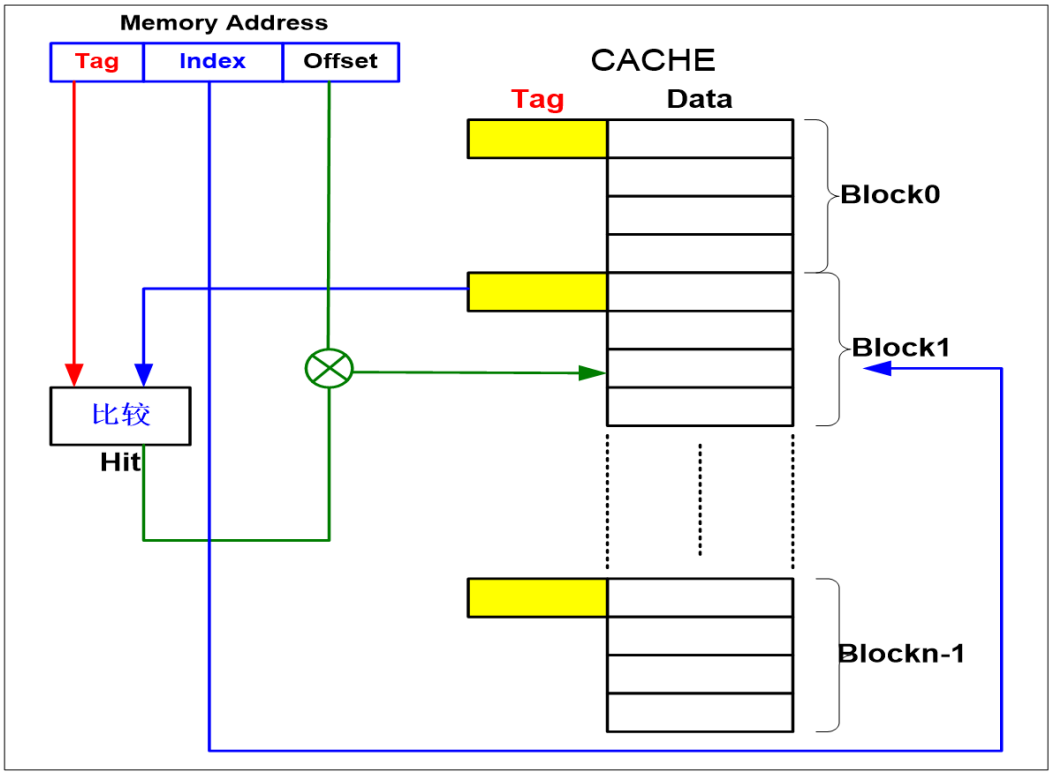
映射机制

- 全相联（Full Associate）：(**1对n，S=1**)**每个主存块映射到Cache的任意块中**
  只有1组，所以需要跟这一组的所有 `Tag` 进行比较，如果有，就去对应 `Tag` 所在的 `Block` 根据 `offset` 取东西

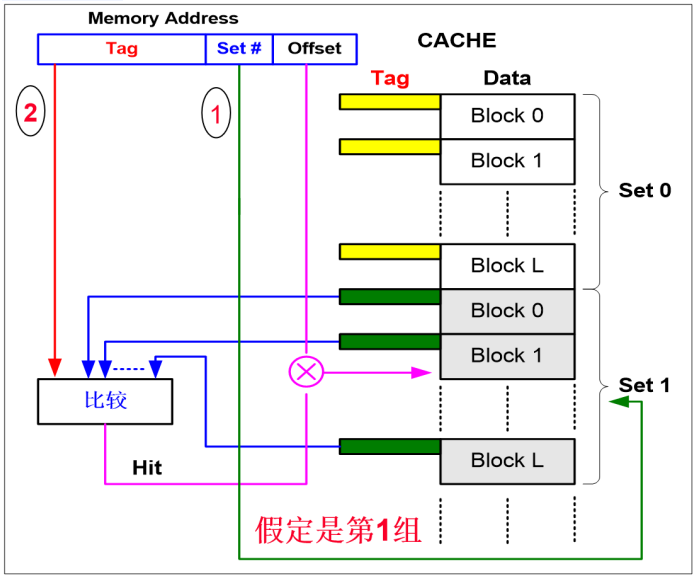


- 直接映射（Direct）：(**n对1，E=1**)每个主存块映射到Cache的**固定块中**：cache中的块和主存中的块分成n组，**一对多映射**

`Index`：第几组 `Tag`：每行同一时刻只有一个`Tag` `Offset`：偏移量
先通过 `Index` 找到是第几组，再比较 `Tag` 是否相同，最后去偏移量 `Offset` 取东西



- 组相联（Set Associate）：(**m对n**)每个主存块映射到Cache的**固定组中的任意块中**: 主存中的块和cache中的块分成k组，**对应组中的块多对多映射**
  先根据 `Set` 找是第几组，然后去对应组里找有没有相同的 `Tag`，如果有就去对应的 `Offset` 取东西

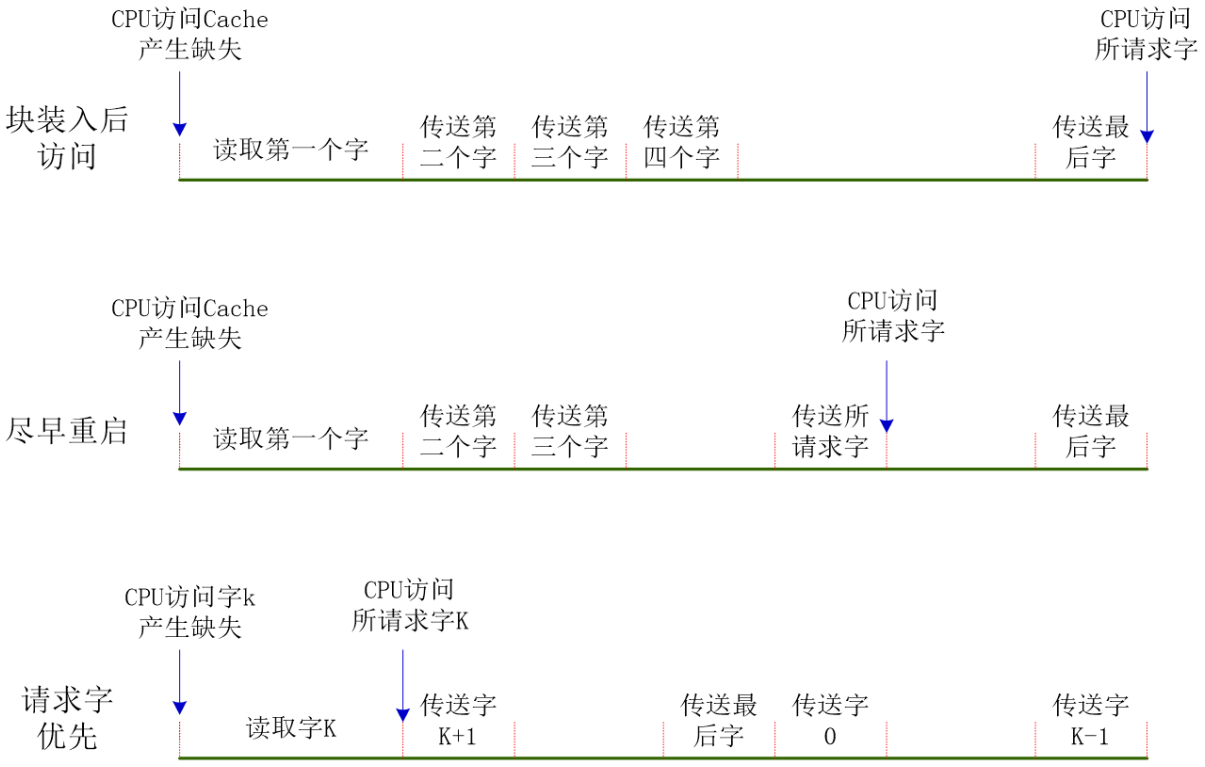


缺失处理

读操作

- **块装入后访问**：缺失数据块中各字按顺序**全部装入Cache后**，再从Cache中**访问**所请求的字（也是引起缺失的字）；
- **尽早重启**（early restart）：缺失数据块中各字**按顺序装入**Cache，**一旦所请求的字装入**Cache，CPU**立即访问**该字，控制机构**再继续传送剩余数据**到cache；

- **请求字优先**（requested word first）：**所请求的字先装入**Cache，CPU**立即访问**该字，控制机构再按照先从所请求字的下一个地址、再到块的起始地址的顺序**继续传送剩余数据**到cache 。



写操作

- “**穿通过**”(Write-Through)模式：在这种模式中高速缓存对于写操作就好像不存在一样，每次写时都**直接写到内存中**，所以实际上只是对读操作使用高速缓存，因而效率相对较低。
- “**写回**”(Write-Back)模式：写的时候**先写入高速缓存**，只是在该block要被新进入的数据**替换**时，才**更新内存**，或者由软件主动地“冲刷”（Flush）Cache中的该block。
  增加脏位，记录该块是否被修改，决定数据替换时是否要更新内存

替换策略！！！

先进先出法（FIFO，First-In-First-Out）

顾名思义，即选择**最早装入**的块进行替换。它的实现比较简单，不需要记录各个块的访问情况，比较容易实现，开销小。但是这种算法没有依据访存的局部性原理（最早调入的块有可能是经常访问的块），因此**不能提高Cache的命中率**。

- 缓存的每一块都设定一个计数器，**初始时均为0**。
- 当某块**被装入或被替换时该块的计数器清为0**，而同组的**其它各块的计数器均加1**，
- 当需要替换时就选择**计数值最大**的块被替换掉。

最低使用频率法（LFU，Least-Frequently Used）

即替换出使用频率最低的块，又称最不经常使用算法。

- 缓存的每一块都设定一个计数器，**初始时均为0**。

- 当某块**被装入或被访问时**该块的**计数器加1。**
- 当需要替换时就选择**计数值最小**的块被替换掉，如果计数值相同则替换出**装入时间更早的块。**

最近最少使用法（LRU，Least-Recently Used）

含义为替换出近期用的最少的块。它与LFU的区别在于它更加关注各个块在**近期**的访问情况，即近期的权重会更高。LRU需要随时记录Cache中各块的访问情况，以便确定近期最少使用的块。LRU比较复杂，一般来说我们采用简化的方法，只记录每个块最近一次使用的时间，替换时选择**距上一次使用经过时间最长的块。**

- 缓存的每一块都设置一个计数器，**初始时均为0。**
- 访问命中时，所有块的计数值与命中块的计数值进行比较
  - 如果某块计数值**小于命中块的计数值**， 则该块的计数值**加 1**
  - 如果该块的计数值**大于命中块的计数值**，则数值**不变**
  - 最后将**命中块的计数器清为0。**
- 访问未命中，需要替换/装入时，则选择**计数值最大的块**被替换/装入，其**计数器清为0**，而**其它的计数器则加1**（除了初始装入之外，计数值是不会出现相等情况的）。

  例：

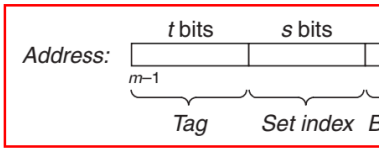
空：	0, 0, 0, 0
Block 1：	0, 1, 1, 1
Block 2：	1, 0, 2, 2
Block 3：	2, 1, 0, 3
Block 4：	3, 2, 1, 0
Block 5：	0, 3, 2, 1

*Cache的容量

- 不作特殊申明时，Cache的容量指Cache**数据块的容量**；
- Cache实际总的存储容量实际上还包含tag和valid bit的位数。

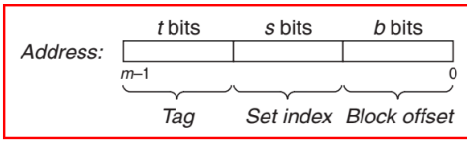
Cache的存储布局

- 例：假设一直接映射Cache，有16KB数据，块大小为4个字（32位字），主存地址32位，每个数据块包括1位有效位，计算实现该Cache所需总存储容量？



  - Cache每数据块大小：4×32 = 128 bits = 2^4 Bytes;
  - Cache块数：16K ÷ 2^4 = 2^{10} 块；
  - tag位数：32 – 10 – 4 = 18 bits
  - 有效位：1位
  - Cache实际总容量：2^{10} × (128+18+1) = 147K位 ≈ 18.4KB

- 例：假设一4路组相联Cache，数据存储空间大小64KB，块大小为16字节，主存地址32位，主存一个字包含4个字节，Cache采用写回策略，每个数据块包括1位有效位，Cache每个字用1位脏位来表示是否被修改。



  1. CPU 如何解释主存地址（主存地址格式）？
  2. 计算实现该Cache所需总存储容量？

- 解答
  - Cache每数据块大小：16×8 = 128 bits = 2^4 Bytes;
  - Cache块数：64K ÷ 2^4 = 2^{12} 块；
  - Cache组数： 2^{12} ÷ 4 = 2^{10} 组
  - tag位数：32 – 10 – 4 = 18 bits
  - 每个Block有效位：1位
  - 每个Block脏位：4位（1个Block包含4个字）
  - Cache实际总容量：2^{12} × (128+18+1+4) = 618496位 ≈ 75.5KB

主存地址格式

Tag位	组位	o
18	10	

*性能计算

■存储访问时间

若：T_m为主存储器的访问周期；

T_c为Cache的访问周期；

H为Cache命中率

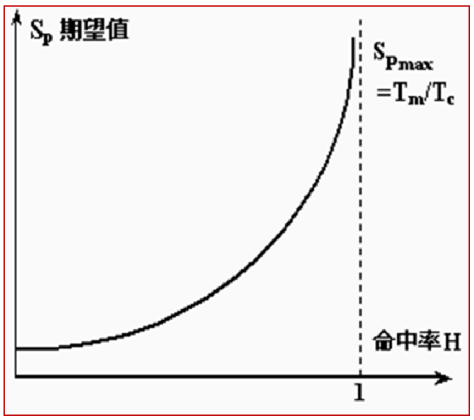
则存储系统的等效访问周期T为：

$$T = T_c \times H + T_m \times (1 - H)$$

■加速比SP（Speedup）

存储系统的加速比S_p为：

$$S_p = \frac{T_m}{T} = \frac{T_m}{H \times T_c + (1 - H) \times T_m} = \frac{1}{(1 - H) + H \times \frac{T_c}{T_m}}$$



加速比与命中率的关系

- 某计算机的存储系统有Cache和主存
- 若所访问的字在Cache中，则存取它要10ns
- 将所访问的字从主存装入Cache需要60ns
- 假定Cache的命中率为 0.9

计算该存储系统访问一个字的平均存取时间。

解： **0.9*(10)+0.1*(10+60)=16ns**

*写策略-Cache一致性问题

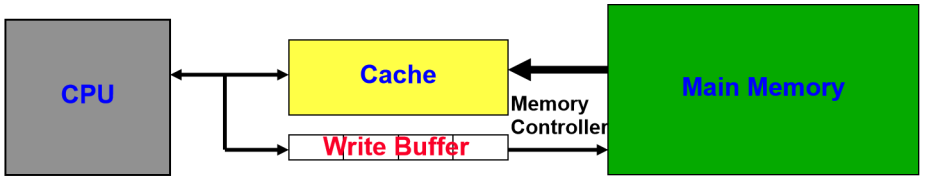
Cache一致性问题

- Cache中的内容是主存块副本，当对Cache中的内容进行更新时，就存在Cache和主存如何保持一致的问题。
- 当多个设备都允许访问主存时
  例如：DMA控制器读写主存时，如果对应Cache行中被修改，则DMA读出的内容无效；若DMA修改了主存单元的内容，则对应Cache行中内容无效。
- 当多个CPU（核）都有各自的Cache而共享主存（L3 cache）时
  某个CPU（核）修改了自身Cache中的内容，则对应的主存单元和其他CPU（核）中对应Cache内容都变为无效。
- 写操作有两种情况
  - 写命中（Write Hit）：要写的单元已经在Cache中
  - 写不命中（Write Miss）：要写的单元不在Cache中

写策略

- **写命中**

- Write Through (通过式写、写直达、**直写**、**写通过**)
  同时写Cache和主存单元，但**效率低**
  使用写缓冲（Write Buffer）



Write Through 模式的Cache结构

  - 一般**Write Buffer** 是**FIFO**，一般有**4**项（块）
  - **CPU**对**Cache**实行写的频率 **<< 1/DRAM Cycle Time**，否则就会发生阻塞
  - 如果解决**FIFO**阻塞
    - 加一个二级**Cache**，使用**Write Back**方式的**Cache**

- Write Back (一次性写、写回、**回写**)
  只写cache不写主存，缺失时一次写回，每行有个修改位（“dirty bit-脏位”），大大降低主存带宽需求，但控制较复杂

- **写不命中**
  - Write Allocate (**写分配**)
    将主存块装入Cache，然后更新相应单元
    试图利用空间局部性，但每次都要从主存读一个块
  - Not Write Allocate (**非写分配**)
    直接写主存单元，不把主存块装入到Cache

- 直写Cache可用写分配或非写分配

- 回写Cache通常用写分配

命中率和缺失率！！！

- 一般情况下，**增加路数**会**提高命中率**，**降低平均时间**，**但不绝对**
- 随着块大小的增加，缺失率**先降低后增加 (由于Cache空间受限，因为块数量会随之下降，带来块替换的增加)**

7.虚拟存储

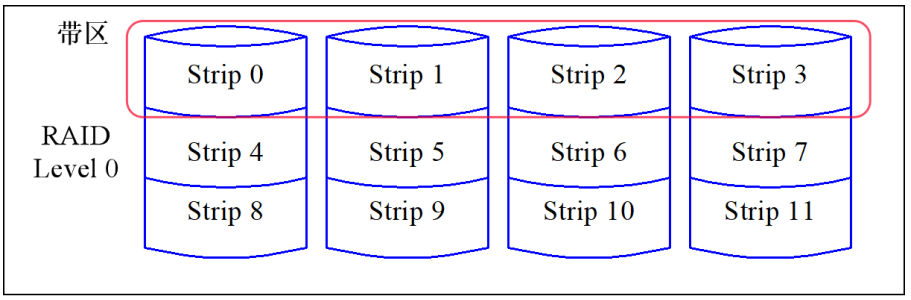
RAID：Reduntant Array of Independent Disks（独立冗余磁盘阵列）

- RAID由**多个物理硬盘**构成，但被操作系统**当成一个逻辑磁盘**，数据分布在不同的物理磁盘上，**冗余磁盘用于保存数据校验信息**，校验信息保证在出现磁盘损坏时能够有效地恢复数据
- RAID特点
  - 通过把多个磁盘组织在一起作为一个逻辑卷提供磁盘**跨越**功能
  - 通过把数据分成多个数据块（strip）**并行**写入/读出多个磁盘以提高访问磁盘的**速度**

- 通过镜像或冗余校验操作提供**容错**能力

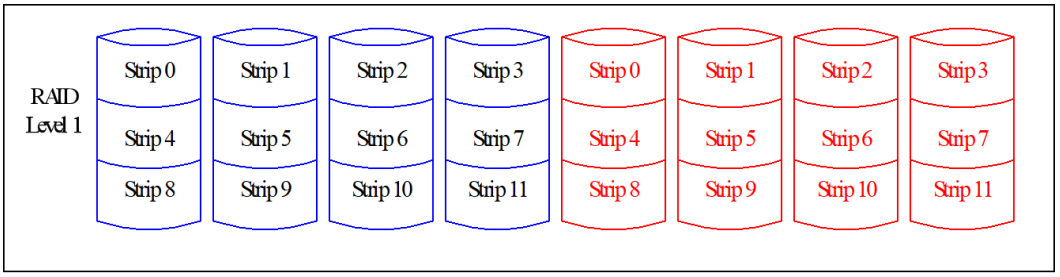
RAID 0

- **无差错控制**的带区组，可以**提高读写性能**



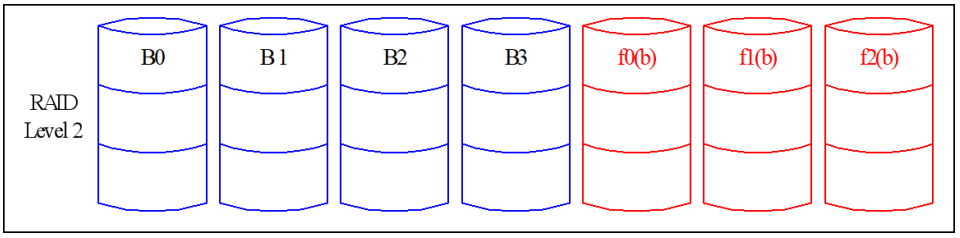
RAID 1

- **无校验的镜像结构**



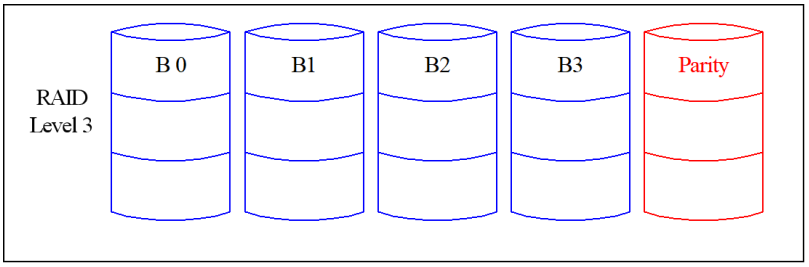
RAID 2

- 带**海明校验**



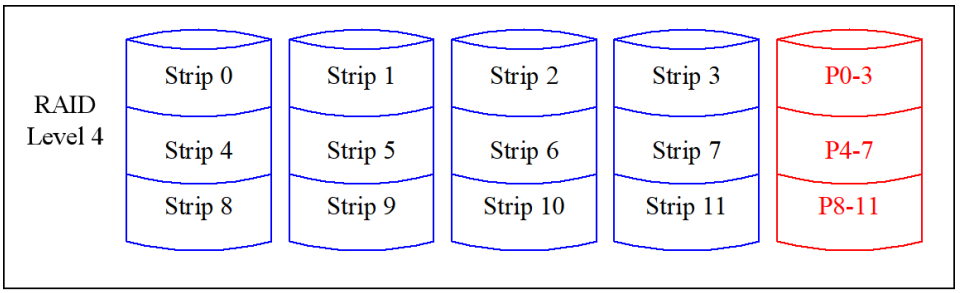
RAID 3

- 带**奇偶校验码**的并行传送



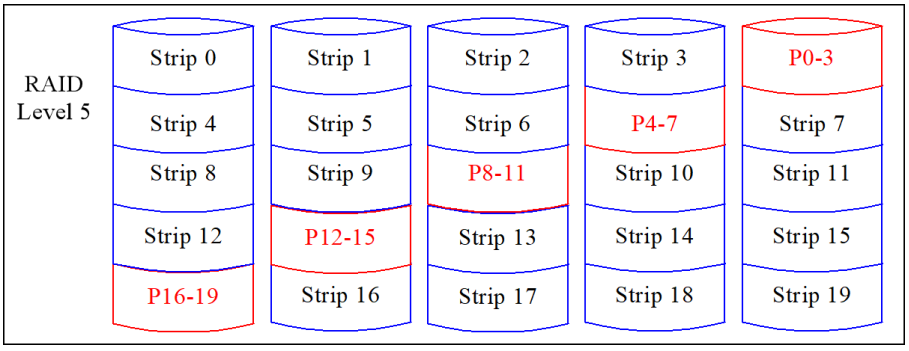
RAID 4

- 带**奇偶校验码**的**独立磁盘**结构



RAID 5

- **分布式奇偶校验**的独立磁盘结构



虚拟存储器

- **虚拟存储技术**：把**主存**当做**辅助存储器**的**高速缓存**的技术
- **虚拟存储器（virtual memory）**：程序中可以使用较大（不一定）的存储空间。
- 功能：
  - 虚拟存储器能从**逻辑**上为用户提供一个比物理存贮**容量大得多、可寻址**的“主存储器”。
  - 虚拟存储器的**容量与物理主存大小无关**，而受限于计算机的**地址结构**。
- 使用过程：
  - 程序执行时，把当前需要的程序段和相应的数据块调入主存，其他暂不用的部分存放在磁盘上
  - 指令执行时，通过硬件（MMU）将逻辑地址（也称虚拟地址或虚地址）转化为物理地址（也称主存地址或实地址）
  - 在发生程序或数据访问失效时，由操作系统进行主存和磁盘之间的信息交换。
- 解决问题：
  - 如何消除**小的主存容量**对编程的限制？
    - 编写编译程序时，不知道程序运行时将和哪些其他程序共享内存；编程者总是希望把每个程序**编译在它自己的地址空间**中
    - **单用户程序大小超过主存容量**
  - 多道程序（进程）如何有效安全的**共享内存**？
    - 同时运行的程序对内存的需求之和可能超过计算机实际内存容量

- 原理：局部性原理
- **虚拟地址与实际地址不同**
- **一般程序加载到主存时只加载到一部分**
- **主存不够的时候会像cache一样进行替换**

页表虚拟寄存器

- 进程划分成相同长度的程序块，称为**虚页**；虚地址 = 虚页号+页内地址
- 主存也分成大小相同的存储块，称为**实页**；实地址 = 实页号+页内地址
- **页表**：记录虚页与实页的映射关系，实现**虚实地址的转换**，页表建立在内存中，操作系统为每个进程建立一个页表。页表用**虚页号作为索引**，页**表项包括虚页对应的实页号和有效位。**
- 页表在主存中的地址由页表基址寄存器指出

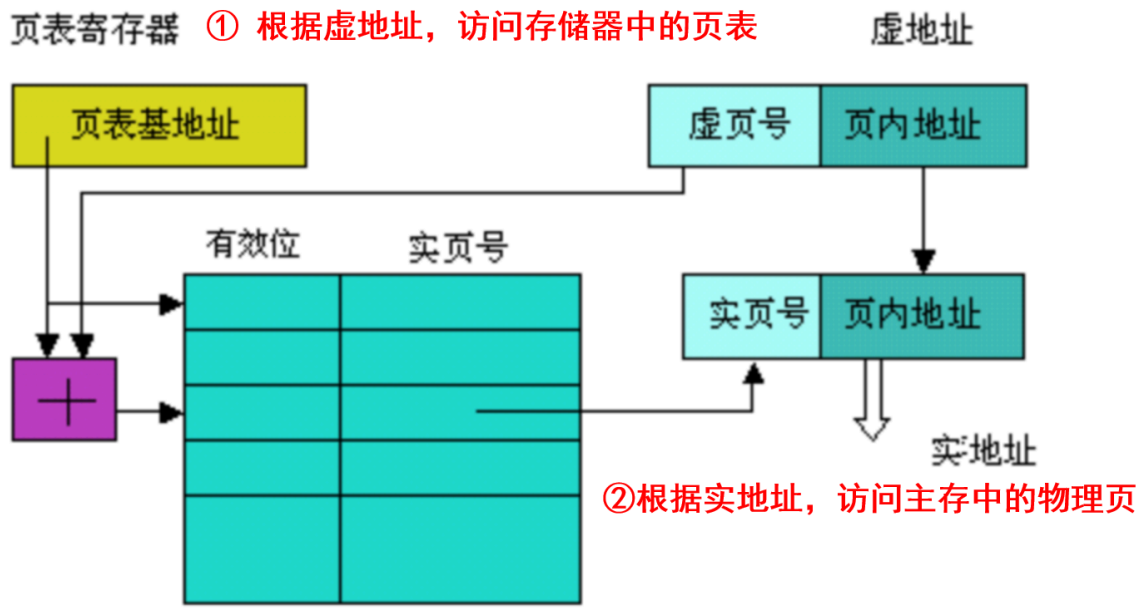
优点

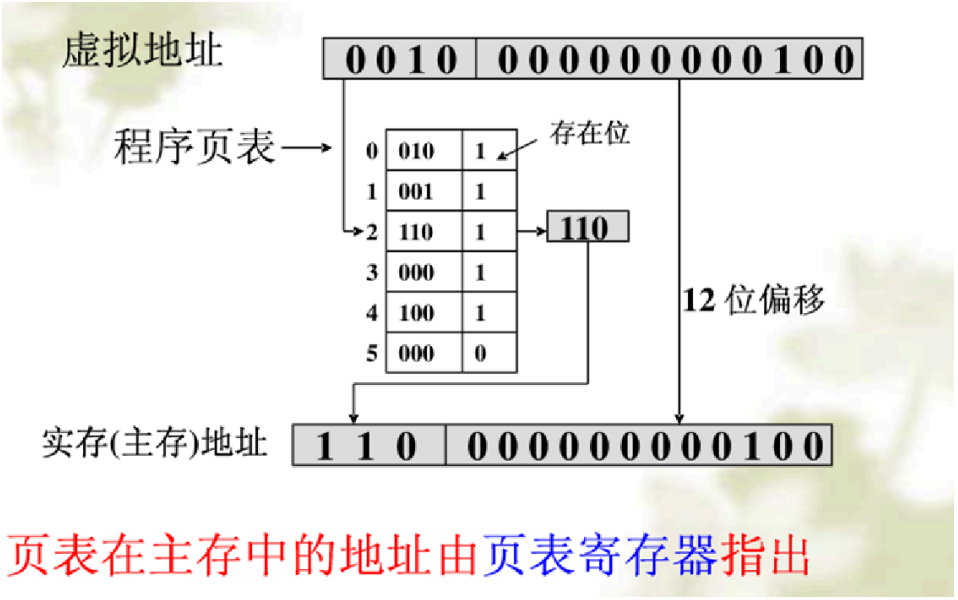
优点：
每个进程有**独立**的虚拟地址空间
简化了**链接**：固定地址，和代码、数据的物理存储器中最终的位置无关
简化了程序**加载**：
为代码和数据分配虚拟页 = 创建 PTE 页表项，设置为 invalid
PTE 指向磁盘目标文件中合适的位置
访问到的页按需加载
简化了**存储空间分配**：连续的虚页可映射到任意的实页
方便进程间**共享**数据和代码
方便了**存储器管理**：PTE 中可扩展许可位

缺页怎么办？（虚页不在内存中）

- 法1：如果有效位为1，则页表保存虚页到实页的映射；如果有效位为0，则页表保存续页的磁盘地址
- 法2：页表只提供虚页到实页的映射，由外页表实现虚页到磁盘地址的映射

虚实地址转换





页表在主存中的地址由页表寄存器指出

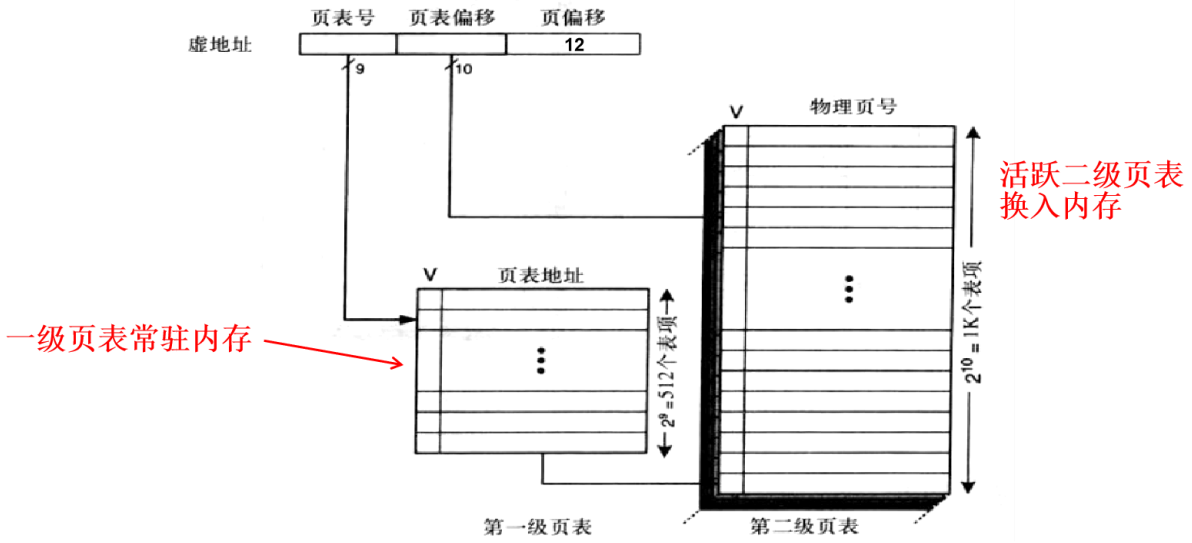
解决页表占据内存过大的问题

- 一级页表：动态扩充，限制大小
- 一级页表：分两个独立的段
- 二级或多级页表：一级为段、二级为页
- 将页表分页，当前使用的页的页表项所在页表在内存，其他在外存，页表也要调进调出

❖ 多级页表 2^{12} 2^{31}

➢ 页面大小为4KB的2GB虚拟存储器，每个页表项占4个字节（32位）：$2^{19}*2^2=2MB$。

512*4+2*1K*4 = 10KB （假定2个活跃页表换入内存时，实际占用的内存）



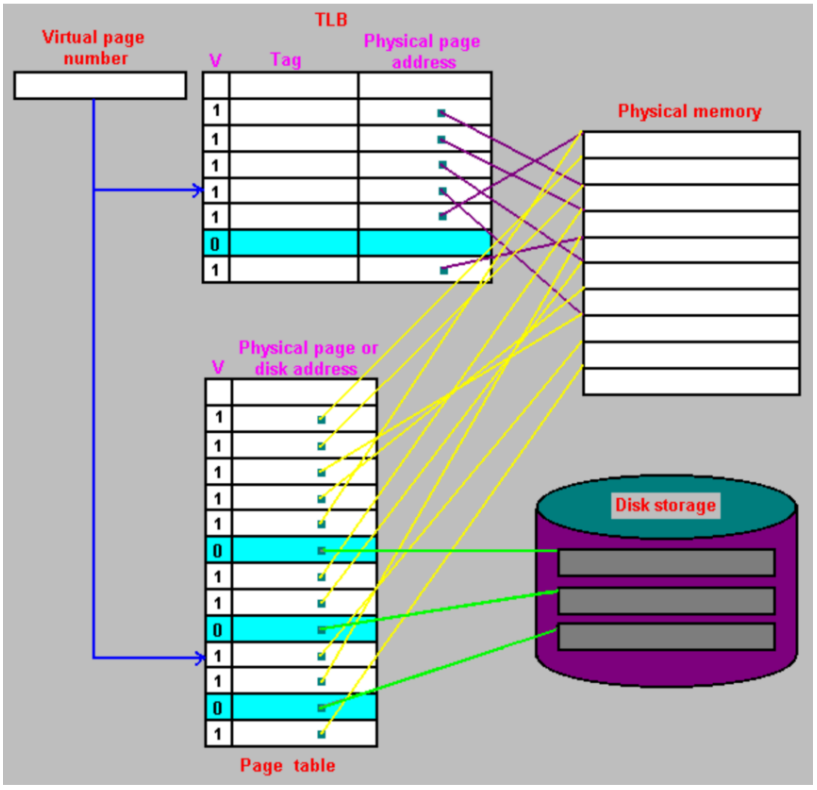
TLB

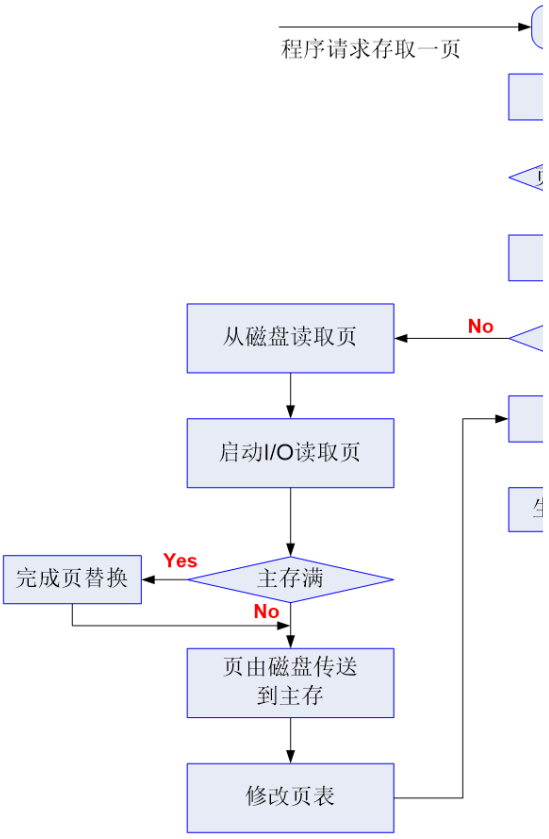
❖ 快表TLB（Translation Lookaside Buffer，转换后备缓冲器）

- VM存在问题：每次虚拟存储器的访问带来两次内存访问，一次访问页表，一次访问所需数据（或指令），简单的虚拟存储器速度慢。
- 解决办法：使用一个小的高速缓存存储部分活跃的页表项，称为TLB（快表），它包含了最近使用的那些页表项。
- TLB内容：标记（全相联结构为虚页号）、数据块（实页号）、有效位、修改位。
- TLB一般采用全相联（或组相联）

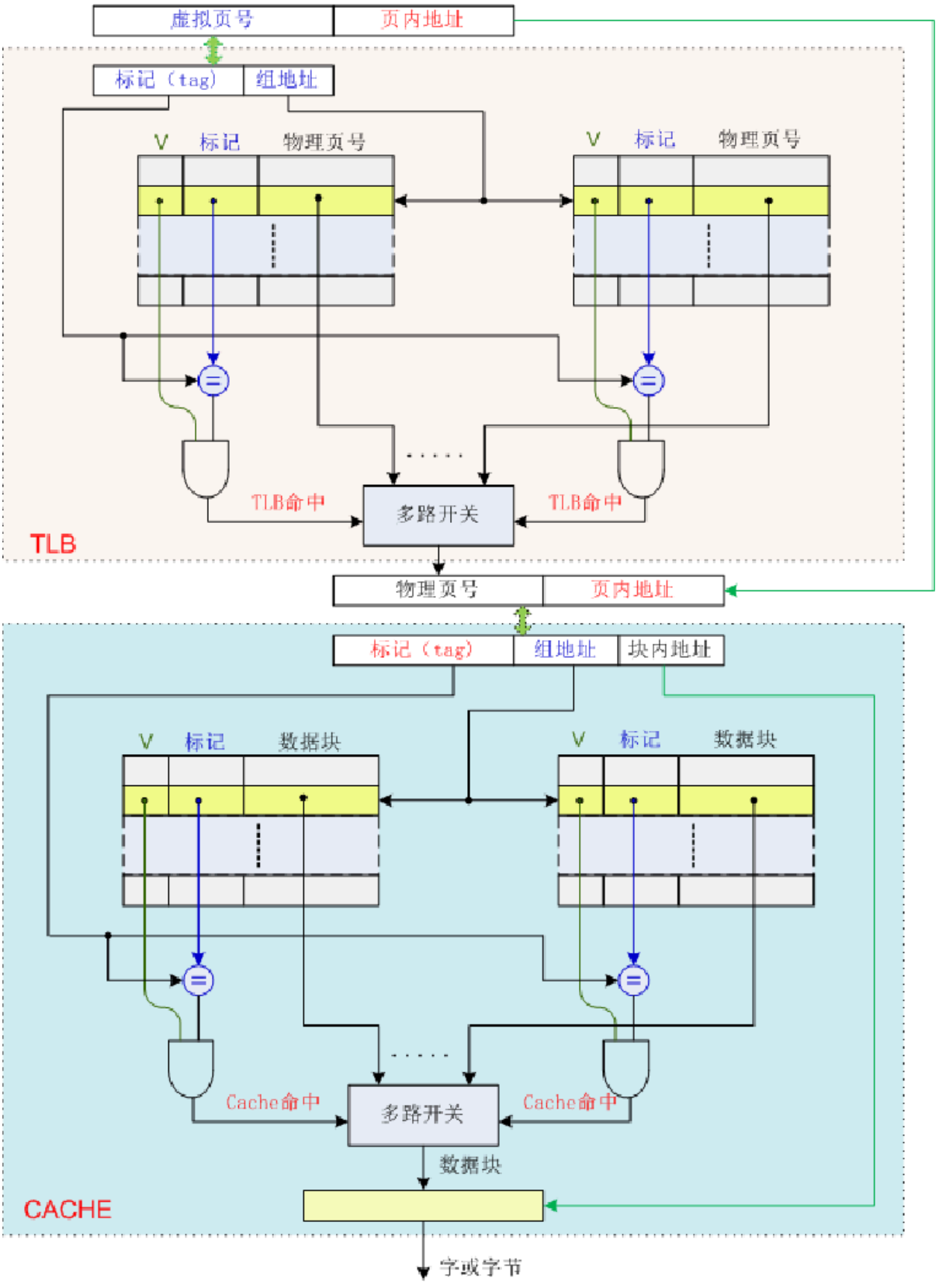
快表（TLB）（全相联）

有效位	修改位	标记（tag）	数据
		虚页号	实页号
		虚页号	实页号
		虚页号	实页号
		虚页号	实页号

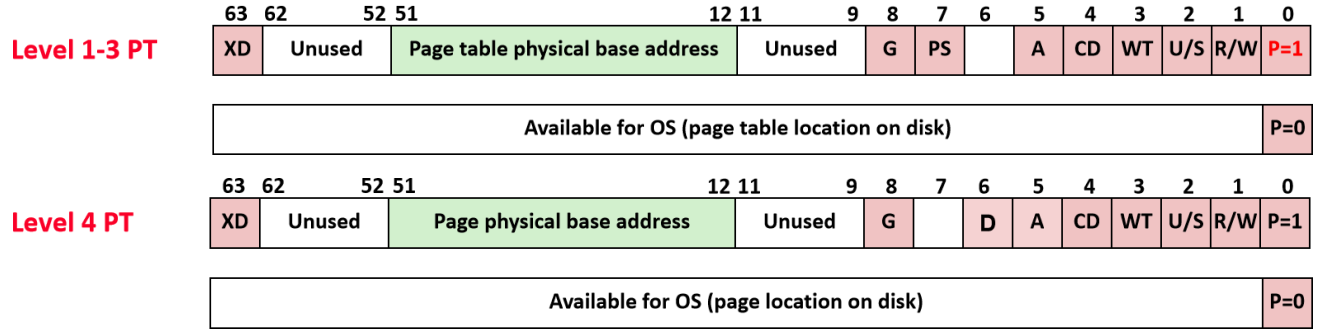




TLB和Cache的联动


虚拟页号
页内地址
标记（tag）
组地址
V
标记
物理页号
V
标记
物理页号
TLB命中
多路开关
TLB命中
TLB
物理页号
页内地址
标记（tag）
组地址
块内地址
V
标记
数据块
V
标记
数据块
Cache命中
多路开关
Cache命中
数据块
CACHE
字或字节

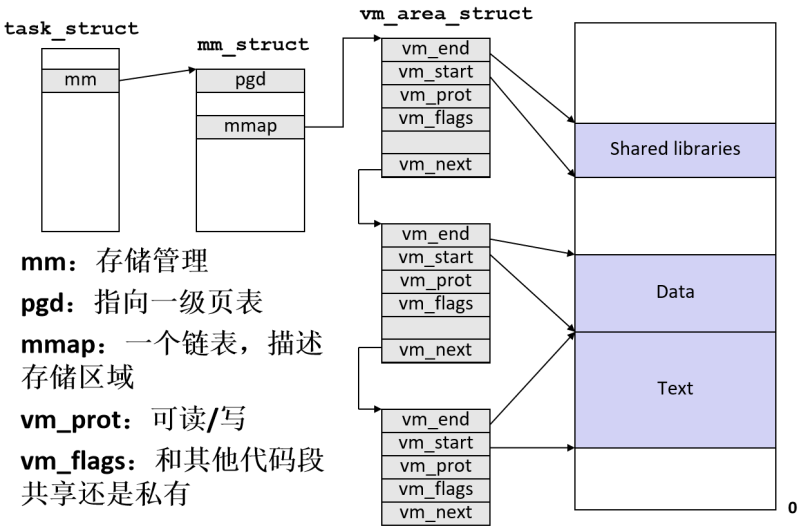
页式虚拟存储器示例



段式管理

❖段式管理：

- ➢Linux 将虚拟存储器组织为不同区域的集合：
  - 为每个区域设置访问权限



mm：存储管理

pgd：指向一级页表

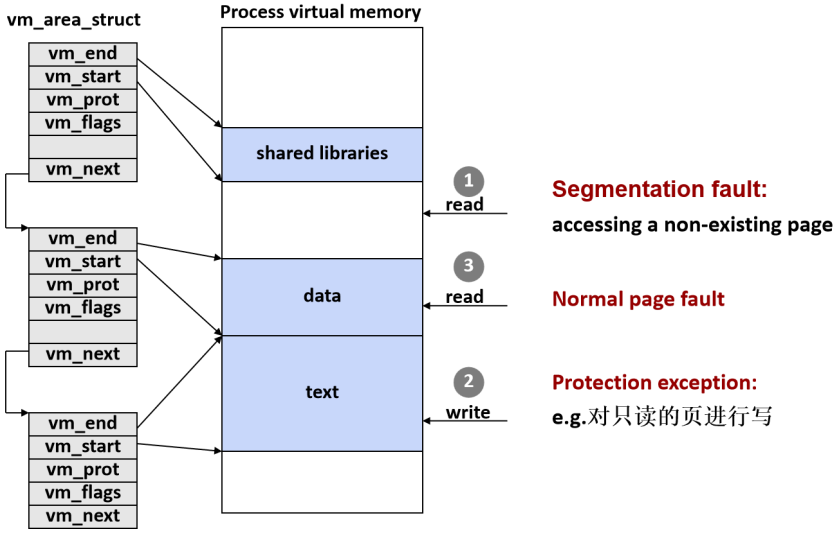
mmap：一个链表，描述存储区域

vm_prot：可读/写

vm_flags：和其他代码段共享还是私有

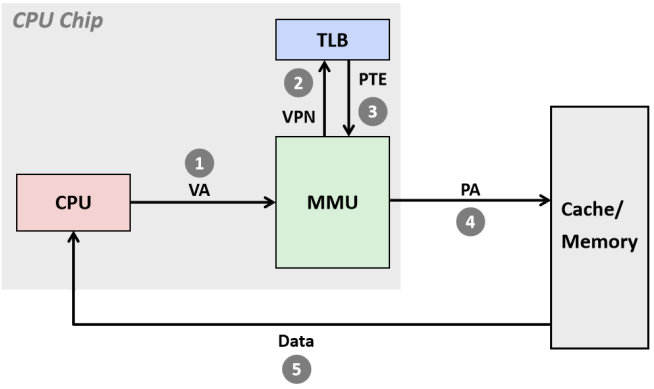
❖段式管理：

- ➢Linux 中的 Page Fault 页错误处理



TLB命中

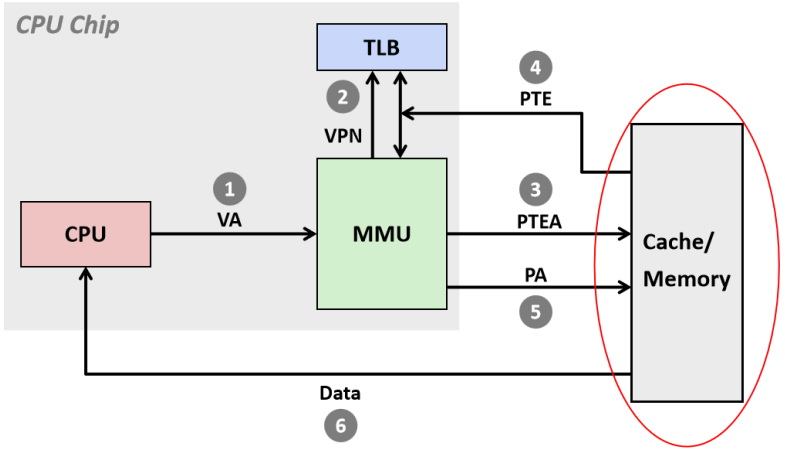
TLB命中



A TLB hit eliminates a memory access

TLB缺失

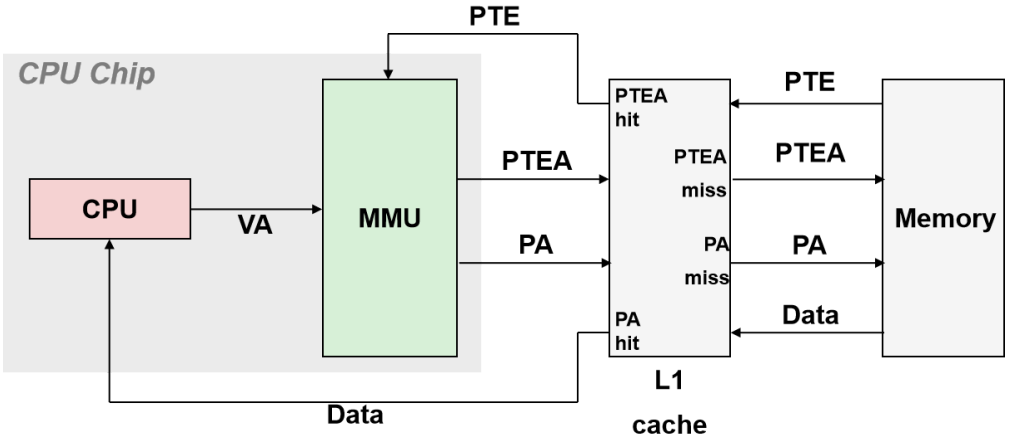
TLB缺失



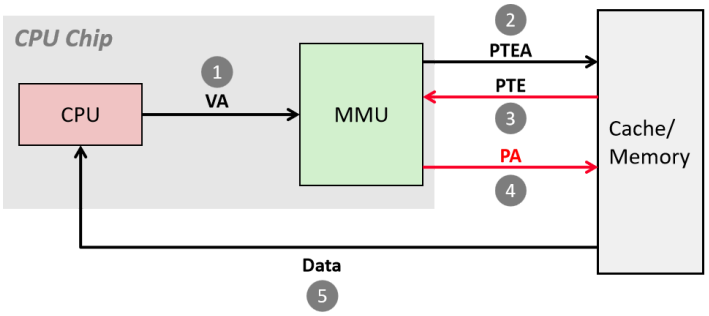
A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare.

页表缓存



页命中

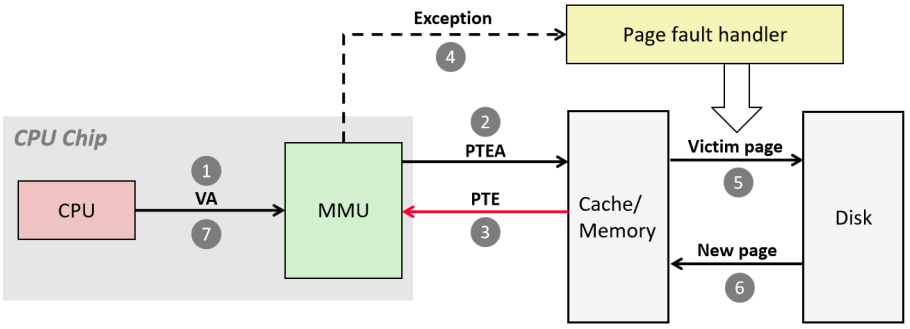


MMU 获得的页表项 PTE 中有效位为 1

则页命中（该页已在主存中）

从 PTE 中得到实页号，获得完整物理地址

页缺失



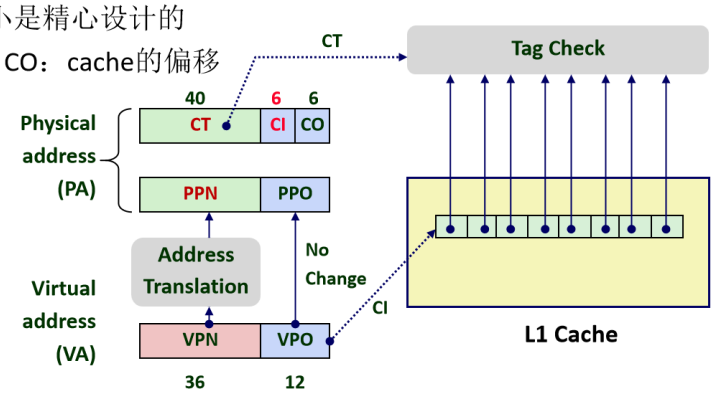
MMU 获得的页表项 PTE 中有效位为 0

则页缺失（该页不在主存中，在辅存）

缺页异常

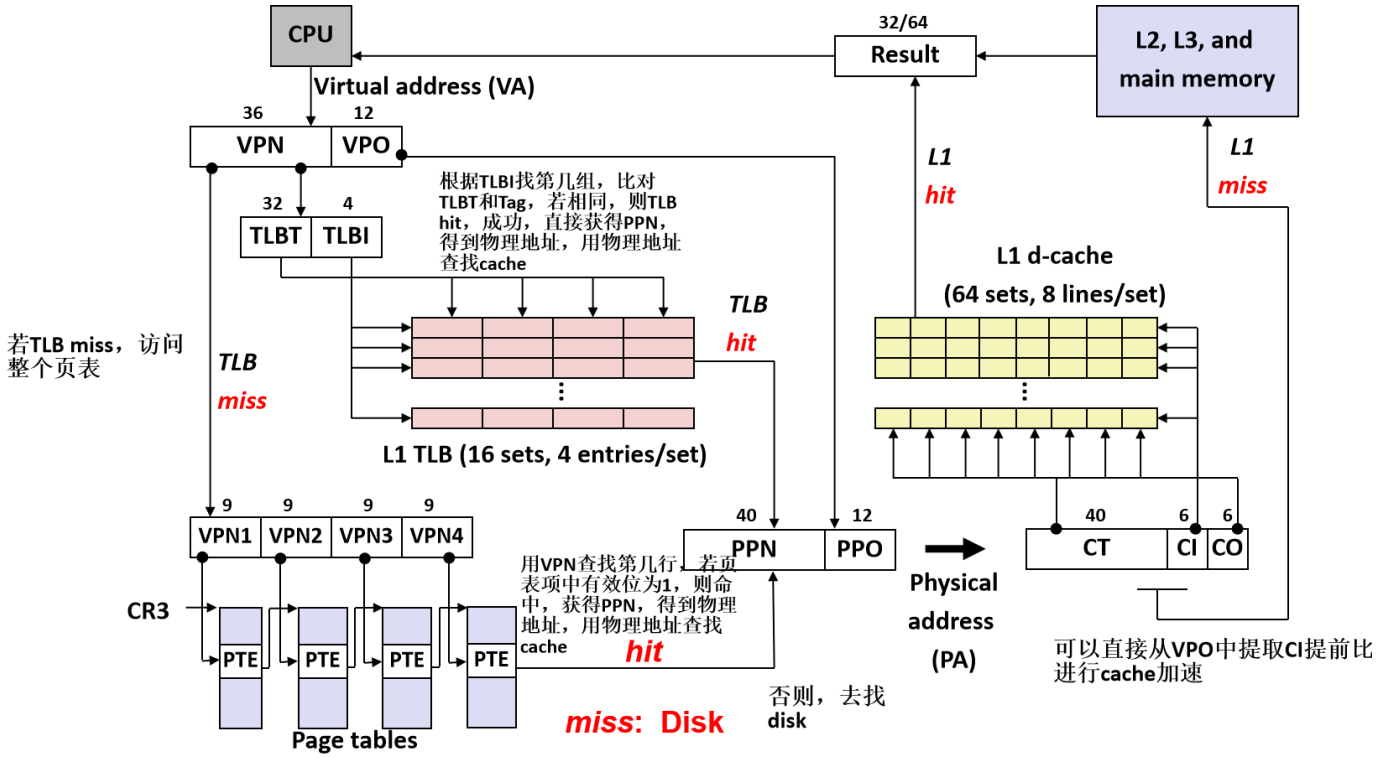
L1 cache访问加速

- **L1 Cache 访问加速**
  - CI 在虚拟地址和物理地址中完全相同。故虚实地址转换的同时，就可以进行cache的组索引。因此一般会先直接用CI去找Cache，等转换完成获得PPN再去和tag进行比对。
  - 大部分情况下TLB会命中，立刻能得到 PPN (即CT)
  - 虚拟地址索引，物理地址标记
  - Cache 结构和大小是精心设计的
  - CI：cache主索引 CO：cache的偏移



L1 i-cache and d-cache:
32 KB, 8-way,
64 sets

总结



TLB命中与否 与Cache是否命中 **无关**

- TLB，页表，Cache三种缺失的可能性

TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	可能，TLB命中则页表一定命中，但实际上不会查页表
miss	hit	hit	可能，TLB缺失但页表可能命中，信息在主存，就可能在Cache
miss	hit	miss	可能，TLB缺失但页表可能命中，信息在主存，但可能不在Cache
miss	miss	miss	可能，TLB缺失页表可能缺失，信息不在主存，一定也不在Cache
hit	miss	miss	不可能，页表缺失，说明信息不在主存，TLB中一定没有该页表项
hit	miss	hit	同上
miss	miss	hit	不可能，页表缺失，说明信息不在主存，Cache中一定也没有该信息

最好的情况应该是hit、hit、hit，此时，访问主存几次？ 不需要访问主存！

以上组合中，最好的情况是什么？hit、hit、miss和miss、hit、hit 只需访问主存1次

以上组合中，最坏的情况是什么？ miss、miss、miss 需访问磁盘、并访存至少2次

介于最坏和最好之间的是什么？ miss、hit、miss 不需访问磁盘、但访存至少2次

例题

2. 假定一个计算机系统中有一个 TLB和一个 L1 Data Cache。该系统按字节编址，虚拟地址16位，物理地址 12位；页大小为 128 字节，TLB采用4路组相联映射，共有16个页表项；L1 Data Cache采用直接映射方式，块大小为 4字节，共 16行。在系统运行到某一时刻。 TLB、页表和 L1 Data Cache中的部分内容（用十六进制表示）如下图所示。

请回答下列问题：

（1）虚拟地址中哪几位表示虚拟页号、哪几位表示页内偏移量？虚拟页号中哪几位表示TLB标记？哪几位表示 TLB组索引？

（2）物理地址中哪几位表示物理页号、哪几位表示页内偏移量？在访问 Cache时，物理地址如何划分成标记字段、行索引字段和块内地址字段？

（3） CPU 从地址067AH中取出的值是多少？要求对 CPU读取地址067AH中内容（一个字节）的过程进行详细说明。

组号	标记	实页号	有效位	标记	实页号	有效位	标记	实页号	有效位	标记	实页号	有效位
0	03	—	0	09	1D	1	00	—	0	07	10	1
1	13	2D	1	02	—	0	04	—	0	0A	—	0
2	02	—	0	08	—	0	06	—	0	03	—	0
3	07	—	0	63	12	1	0A	34	1	72	—	0

(a) TLB 内容(4 路组相联，4 组，16 个页表项)

虚页号	实页号	有效位
000	08	1
001	03	1
002	14	1
003	02	1
004	—	0
005	16	1
006	—	0
007	07	1
008	13	1
009	17	1
00A	09	1
00B	—	0
00C	19	1
00D	—	0
00E	11	1
00F	0D	1

（b）部分页表内容（前 16 项）

行索引	标记	有效位	字节3	字节 2	字节 1	字节 0
0	19	1	12	56	C9	AC
1	—	0	—	—	—	—
2	1B	1	03	45	12	CD
3	—	0	—	—	—	—
4	32	1	23	34	C2	2A
5	0D	1	46	67	23	3D
6	—	0	—	—	—	—
7	10	1	12	54	65	DC
8	24	1	23	62	12	3A
9	—	0	—	—	—	—
A	2D	1	43	62	23	C3
B	—	0	—	—	—	—
C	12	1	76	83	21	35
D	16	1	A3	F4	23	11
E	33	1	2D	4A	45	55
F	—	0	—	—	—	—

（C） L1 Data Cache 内容（直接映射，16 行，块大小 4 字节）

❖请回答下列问题：

➢(1)虚拟地址中哪几位表示虚拟页号、哪几位表示页内偏移量？虚拟页号中哪几位表示TLB标记？哪几位表示TLB组索引？

- TLB分为4组，所以TLB组索引为2位
- 16位虚拟地址中低7位为页内偏移量，高9位为虚页号；虚页号中高7位为TLB标记，低2位为TLB组索引

<table>
<tr><td>7</td><td>2</td><td>7</td></tr>
<tr><td>TLB标记</td><td>TLB组索引</td><td rowspan="2">页内偏移</td></tr>
<tr><td colspan="2">虚页号</td></tr>
</table>

❖请回答下列问题：

➢(2)物理地址中哪几位表示物理页号、哪几位表示页内偏移量？在访问Cache时，物理地址如何划分成标记字段、行索引字段和块内地址字段？

- Cache有16行，所以Cache行索引为4位
- 12位物理地址中低7位为页内偏移量，高5位为物理页号。12位物理(主存)地址中，低2位为块内地址，中间4位为Cache行索引，高6位为标记

<table>
<tr><td>5</td><td colspan="3">7</td></tr>
<tr><td>物理页号</td><td colspan="3">页内偏移</td></tr>
<tr><td>6</td><td>4</td><td colspan="2">2</td></tr>
<tr><td>Cache标记</td><td>Cache行索引</td><td colspan="2">块内地址</td></tr>
</table>

区号　　区内索引

(3)CPU从地址067AH中取出的一个字节值是多少？要求对CPU读取地址067AH中内容的过程进行详细说明。

<table>
<tr><td>7</td><td>2</td></tr>
<tr><td>TLB标记</td><td>TLB组索引</td></tr>
<tr><td colspan="2">虚页号</td></tr>
</table>

- 地址067AH=0000 0110 0111 1010B，所以，虚页号为0000 0110 0B，映射到TLB的第00组
- 将0000 011B=03H与TLB第0组的四个标记比较，虽然和其中一个相等，但对应的有效位为0，其余都不相等，所以TLB缺失，需要访问主存中的页表
- 直接查看0000 0110 0B=00CH处的页表项，有效位为1，取出物理页号19H=1100 1B，和页内偏移111 1010B拼接成物理地址：1100 1111 1010B，继续转化为直接相联内存地址格式：

1100 1111 1010B

<table>
<tr><td>5</td><td></td></tr>
<tr><td>物理页号</td><td>页</td></tr>
<tr><td>6</td><td>4</td></tr>
<tr><td>Cache标记</td><td>Cache行</td></tr>
</table>

区号　　区内索

- 根据中间4位1110直接找到Cache第14行(即第E行)，有效位为1，且标记为33H=11 0011B，正好等于物理地址高6位，故命中
- 根据物理地址最低两位10，取出字节2中的内容4AH=0100 1010B

08 虚拟存储示例

1. 给定一个32位的虚拟地址空间和一个24位的物理地址，对于下面不同的分页大小P，请确定虚拟页号（VPN）、虚拟页内偏移量（VPO）、物理页号（PPN）和物理页内偏移量（PPO）的位数。

P	#VPN位数	#VPO位数	#PPN位数	#PPO位数
1KB	22	10	14	10
2KB	21	11	13	11
4KB	20	12	12	12
8KB	19	13	11	13

举例

2^{14}

假定页式虚拟存储系统按字节编址，逻辑地址**36**位，页大小**16KB**，物理地址**32**位，页表中包括有效位和修改位各**1**位、使用位和存取方式位各**2**位，且所有虚拟页都在使用中。请问：

（**1**）每个进程最多可以有多少虚页？页表大小为多少？

（**2**）如果所使用的快表（**TLB**）总表项数为**256**项，且采用**2**路组相联**Cache**实现，则快表大小至少为多少？

解答（**1**）

页面大小：**16KB=2^{14}，页内偏移14位**

虚地址**36**位：**虚页号 = 36-14 = 22位**

每个进程最多可有：**2^{22}个虚页**

实地址**32**为：**实页号 = 32-14 = 18位**

每个页表项：**1 + 1 + 2 + 2 + 18 = 24位**

每个页表所占空间：**2^{22}×24/8 = 12MB**

（**2**）

TLB：**256**个表项，**2**路组相联，所以共有**128**组

22位虚页号：**15位Tag**，**7**位组地址

TLB每个表项：**15 + 24 = 39**位

TLB容量：**39×256 = 9984**位 = **1248**字节

8.链接

连接器程序（ld）做了什么

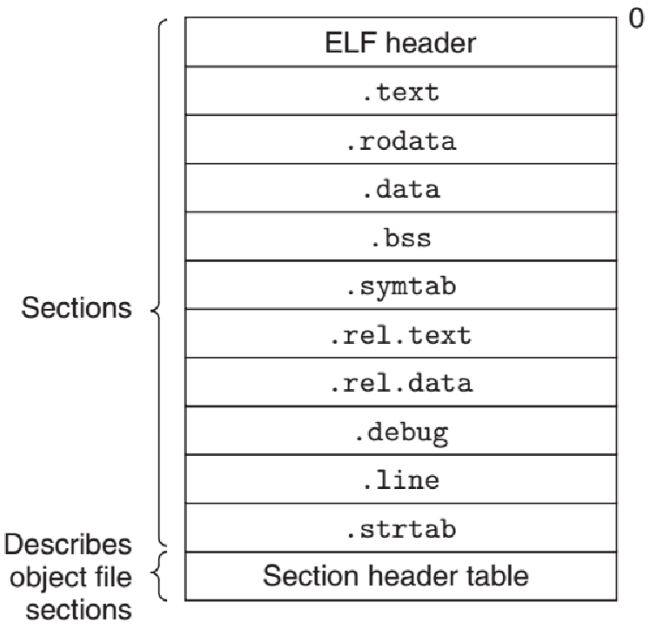
1. **符号**解析
   - 程序定义和引用符号（函数、全局变量、静态变量）
   - 符号在哪：符号的定义由编译器保存在.o文件的**符号表**中
   - 链接器的任务：把每个符号引用和定义**关联**起来
   - **非静态的临时变量**不由链接器处理，而是放到**寄存器或栈**上；**静态变量由链接器进行管理**
2. 重定位
   - 将独立的代码和数据节**合并到单个节**（section）中：可以分配绝对地址

- 将符号从.o文件中的**相对**位置**重新定位**到可执行文件中的最终**绝对**内存位置：符号**定义**有了绝对地址
- 将这些符号的所有**引用**更新到其新位置：符号引用有了绝对地址

目标文件三种形式

- 可重定位目标文件（relocatable object file: .o ）
  - 包含**二进制**代码和数据
  - 其形式可以在**编译时**与其他可重定位目标文件**合并**起来，创建一个可执行目标文件
  - **每一个 .c 源文件产生一个对应的 .o 文件**
- 可执行目标文件（executable object file）
  - Linux：可以没有扩展名或者叫 a.out
  - Windows：.exe
  - 包含**二进制**代码和数据，与可重定位目标文件不同的是**已经分配的具体的内存地址**
  - 其形式可以被**直接复制到内存并执行**
- 共享目标文件（shared object file：.so）
  - Linux：.a(静态链接库) .so(动态链接库)
  - windows：.lib (静态链接库) .dll （动态链接库）
  - 一种特殊类型的**可重定位目标文件**
  - 可以在加载或者运行时被**动态地加载进内存**并链接
- 不同系统的目标文件格式都不相同
  - Windows 使用可移植可执行 (Portable Executable, PE) 格式
  - MacOS-X 使用 Mach-O 格式
  - 现代 x86-64 Linux 和 Unix 系统使用可执行可链接格式 ELF (Executable and Linkable Format)

ELF文件结构



- Elf 头
  - 以一个16字节的序列开始，这个序列描述了生成该文件的系统的**字的大小和字节顺序（大小端存储模式）**。
  - ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息。包括 ELF 头的大小、目标文件的类型（如可重定位、可执行或者共享的）、机器类型（如 x86-64 / MIPS）、节头部表 (section header table) 的文件偏移，以及节头部表中条目的大小和数量。
- 节头部表(Section header table)
  - 描述**各节的位置（偏移）和大小**
  - 其中目标文件中每个节都有一个固定大小的条目 (项，entry)
- .text: 已编译程序的机器代码（Code）
- .rodata(read only): 只读数据，**代码段中（不属于数据段）**
- .data: **已初始化的全局和静态 C 变量**
- .bss: Block Started by Symbol
  - **未初始化的全局和静态 C 变量**（默认初始化为0），以及所有被初始化为 0 的全局或静态变量
  - 在目标文件中这个节不占据实际的空间，它仅仅是一个**占位符**
  - 局部 C 变量在运行时被保存在栈中，既不出现在 .data 节中，也不出现在 .bss 节中
- .symtab:(symbol table) 符号表
  - 存放在程序中**定义和引用的函数和全局变量**的信息
  - 不包含局部变量的条目
- .rel.text(relocate): .text 节的重定位信息
  - 在合并生成可执行文件时**需要修改的指令的指针**
  - 如：任何调用外部函数或者引用全局变量的指令
- .rel.data（.rel.data.rel）：.data节的重定位信息
  - 在合并生成可执行文件时**需要修改的数据的指针**

- 任何已初始化的全局变量，如果它的初始值是一个**全局变量地址或者外部定义函数的地址**，都需要被修改

- .debug、.line等:（gcc -g）
  - 调试符号表
  - 原始 C 源程序中的行号和 .text 节中机器指令之间的映射
- .strtab: 字符串表
  + 以 null 结尾的字符串的序列
  + 包括 .symtab 和 .debug 节中的符号表
  + 节头部中的节名字
  不同编译选项生成的.o文件中，可能有不同的节、不同的顺序

符号

- 全局符号：**非静态的 C 函数和全局变量**
- 外部符号： 在其他模块中定义的非静态 C 函数和全局变量
- 局部符号：**静态的 C 函数、全局变量和局部变量**
  - 静态变量只本文件可见

全局符号分为强符号和弱符号

- 强符号：**函数**和**已初始化**的全局变量
- 弱符号：**未初始化**的全局变量
- 强弱符号**不考虑外部符号**（只考虑**定义**，不考虑引用），只考虑**全局符号**，**不考虑加static的局部符号**

连接器符号规则

- 不允许有多个同名的强符号，否则链接错误
- 如果有**一个强符号和多个弱符号**同名，那么**选择强符号**
  对弱符号的引用会被解析到强符号
- 如果有**多个弱符号**同名，那么从这些弱符号中**任意选择一个**
  用 gcc –fno-common 命令可以在遇到多重定义的全局符号时引发错误提示
- **同名符号被解析到同一个符号**

什么时候链接

- 静态：编译之后
- 动态：程序**加载/执行**过程中加载和链接共享库

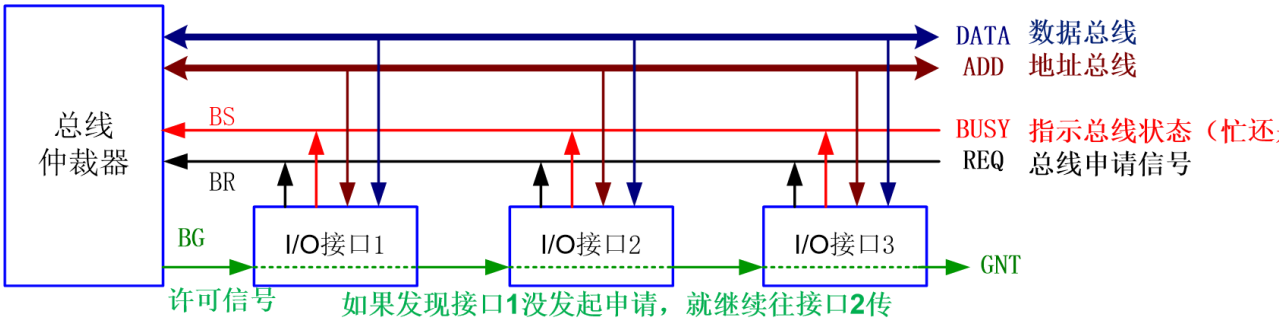
9.总线与IO

总线：计算机主要部件连接到一起
IO：CPU与外部设备通讯的接口

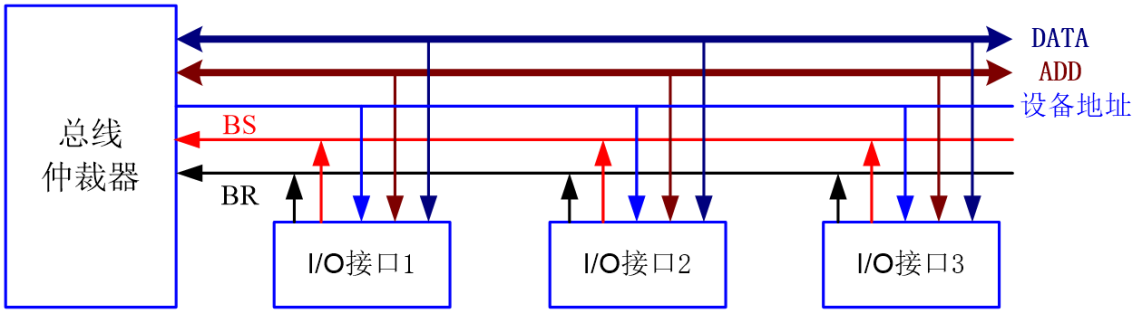
1 总线的仲裁

总线的仲裁方式

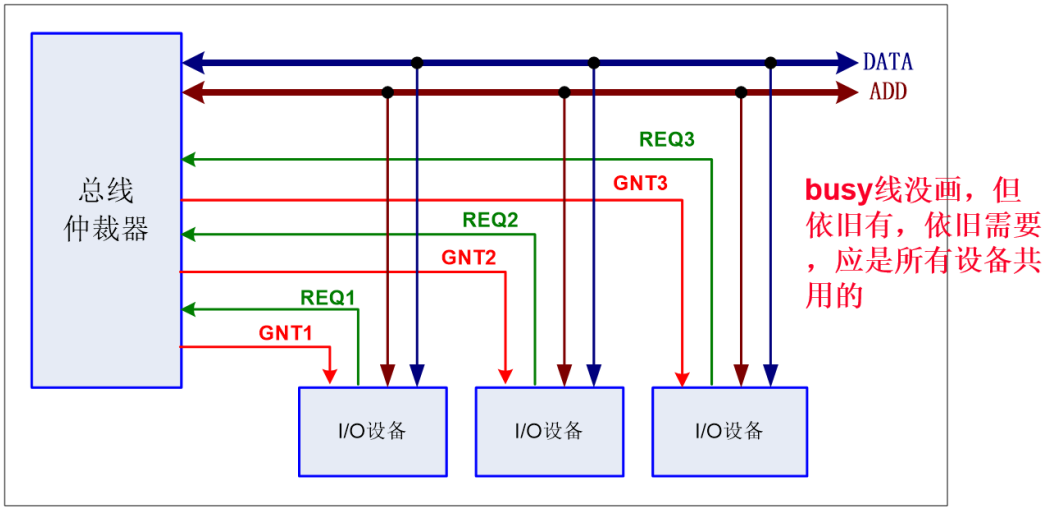
- 集中式仲裁方式（集中有一个地方来判断总线的使用权归谁）
  - 链式查询方式：从第一个到最后一个**依次**看有没有总线申请，如果有就立刻停止；优先级由**硬件连接顺序**决定；硬件简单，易于扩展，但对故障敏感，优先级不灵活



  - 计数器定时查询方式：当某个有总线申请的设备地址与**计数器**一致，便获得总线使用权；优先级与计数器计数方式有关；硬件较复杂，但对故障不敏感，优先级定义灵活



  - 独立请求方式：每个设备有**独立**的请求信号和总线同意信号；总线控制器根据设备的优先级决定将总线的使用权交给哪个设备；连线最多成本高，但速度最快，对故障不敏感，优先级定义灵活



- *分布式仲裁方式

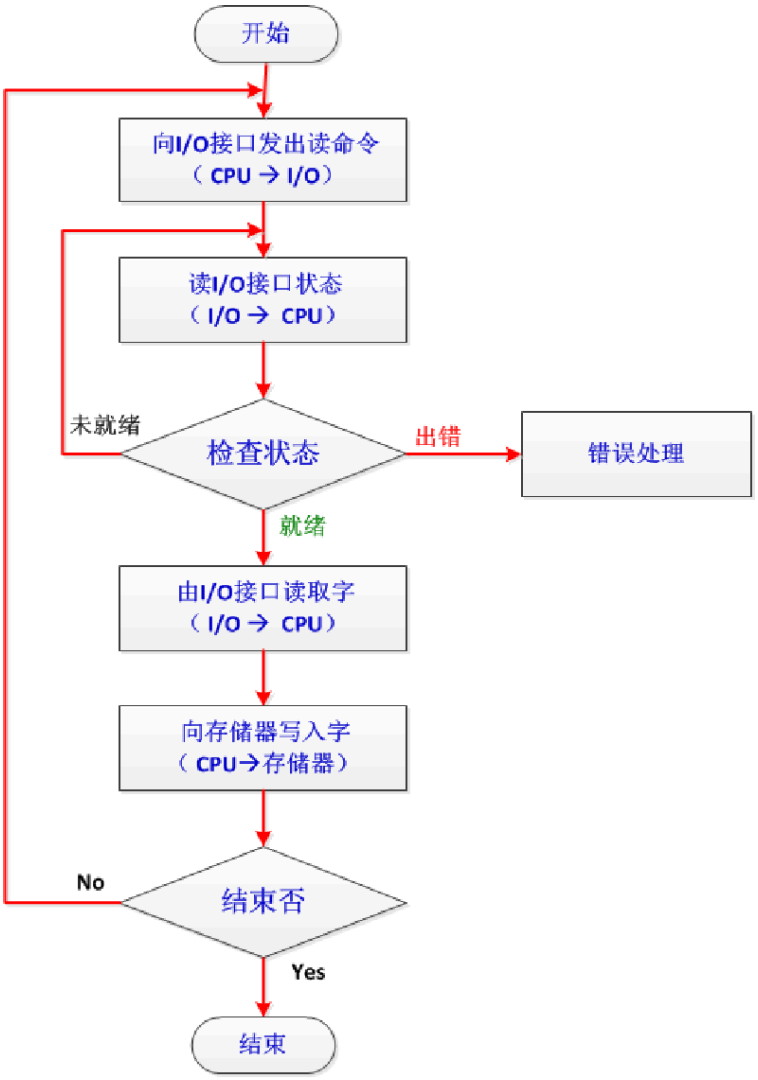
2 I/O

程序查询 I/O 方式

- I/O操作由CPU直接完成（通过**执行I/O指令**完成）,包括：

- 检测设备状态
- 发送读写命令（处理器发送I/O命令后，必须等待，直到I/O接口状态就绪）
- 传送数据

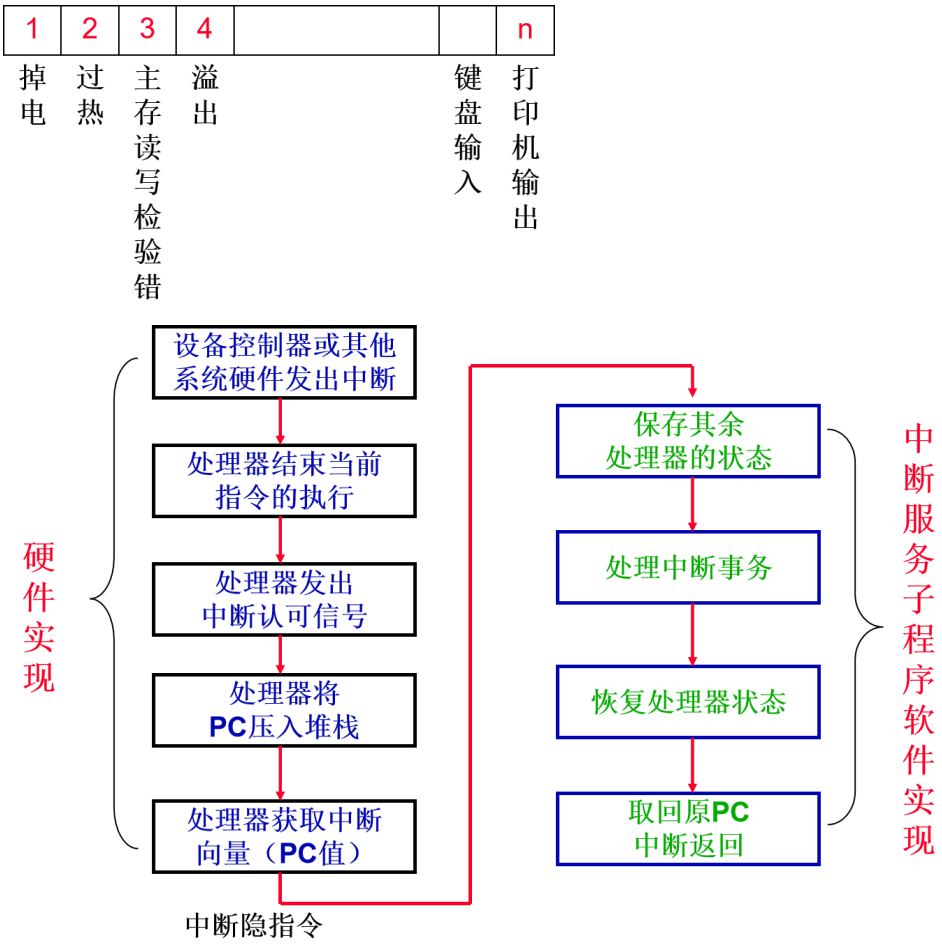
- 外设速度慢，CPU速度快，在外设准备过程中，CPU处在不断的查询之中，CPU的效率**浪费严重。**
- 外设与CPU完全串行工作。



中断与中断 I/O 方式

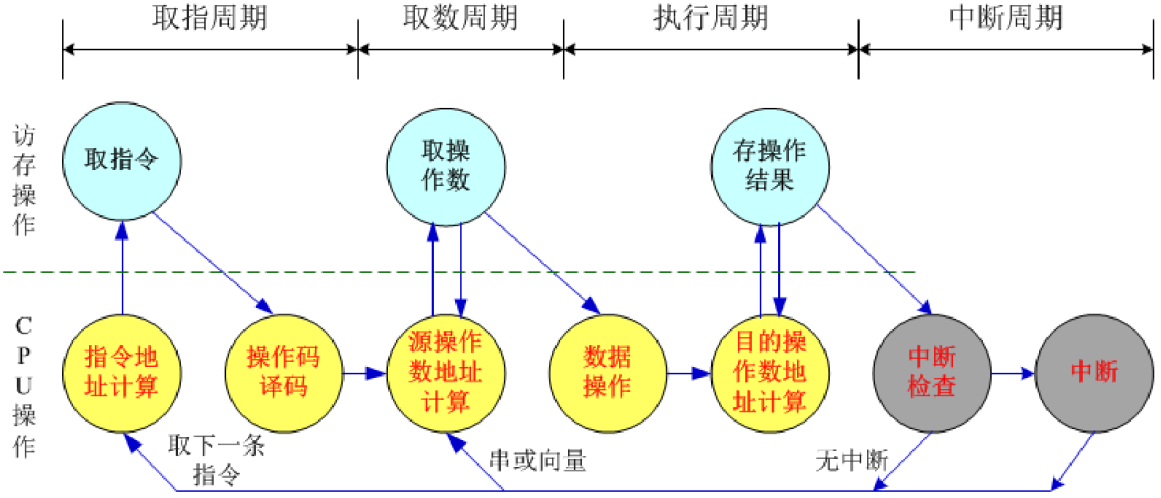
中断请求

- 中断请求触发器（INTR）：每个中断源配置一个中断请求触发器；
- 中断请求标记寄存器：各中断源的请求触发器组成中断请求标记寄存器；
  哪一位为一：哪里导致了中断



有中断的指令周期图

中断不会打断指令的执行，会等到指令执行结束再中断



DMA I/O 方式

DMA过程

1. DMA请求：当接口做好数据传输的准备，通过有关逻辑向CPU发出DMA请求信号。
2. CPU初始化DMA控制器
    1. 设置数据传送方向：是请求读还是请求写（对存储器而言）
    2. 设置I/O接口地址：DMA操作所涉及的I/O接口的地址

3. 设置存储器起始地址：读或写存储器的起始单元地址
4. 设置传送的数据数量：传送数据的字数
5. 有关中断方式的设置：DMA结束后通过中断方式请求CPU处理

3. DMA操作

DMA控制器接到DMA应答信号后，通过控制逻辑向系统总线发送存储器地址信号、存储器读写控制信号、I/O接口读写控制信号等，完成一次数据传送。这些操作完全由DMA控制器完成。

- 若是单字传输，一般仅需要一个总线周期，所以这种方式称为**周期窃取**（cycle-stealing，或者叫周期挪用）方式。
- 若是成组传输，需要多个总线周期来完成。

  所有数据传送结束后，通过中断方式告知CPU进行善后处理。**CPU仅在开始DMA操作之前和完成DMA操作之后参与I/O处理，在DMA过程中，CPU可以运行原来的程序（或其他程序）**

DMA传送方式

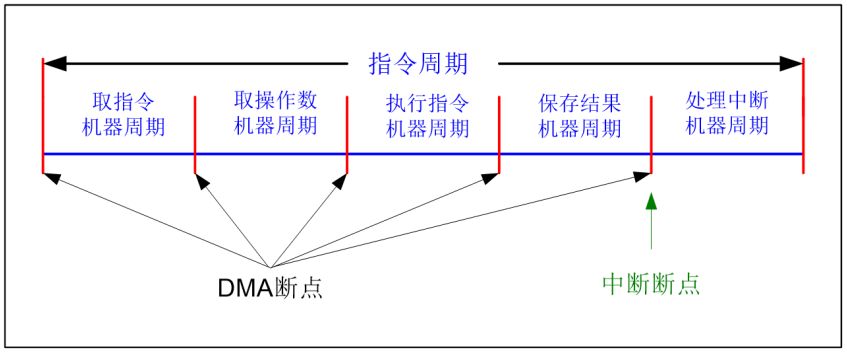
- 停止CPU访问内存（成组传送方式）

  一次DMA请求得到响应后，DMA控制器完全占用总线，进行块数据（多字）传送，直到所有数据传送完毕才释放总线，这段时间完全停止CPU访问内存。

  适应高速外设与存储器交换数据的情况。
- 周期窃取方式（单字传送方式，DMA和CPU交替使用总线）

  每次DMA请求得到响应后，DMA控制器窃取一个总线周期完成一次数据传送，然后释放总线，CPU接着使用一个总线周期，然后DMA再窃取一个周期，这样持续循环下去，直到数据传输结束。一般情况下，CPU 不访问存储器时释放总线

  一般适应存储器速度远高于I/O设备速度的情况。

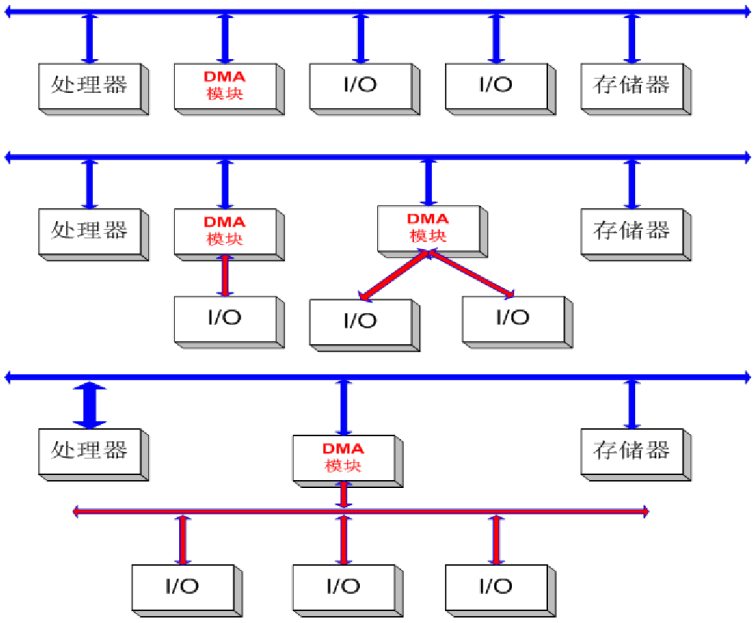


指令周期中的DMA和中断断点

DMA断点：每个断点看CPU现在用不用总线，不用的话就窃取一个周期，DMA来占用总线（因为一般情况下，CPU 不访问存储器时释放总线，所以不会冲突而且不会影响CPU的效率）

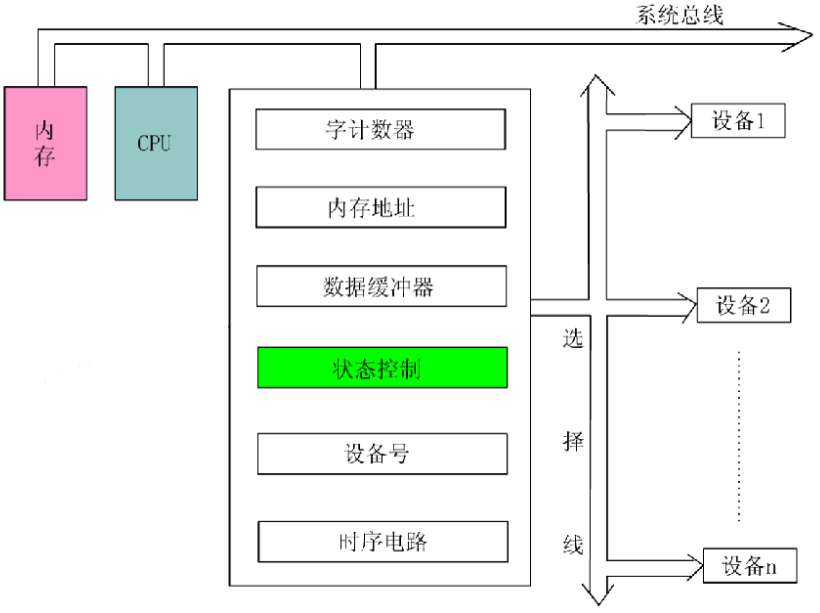
DMA控制器的结构

三种DMA结构



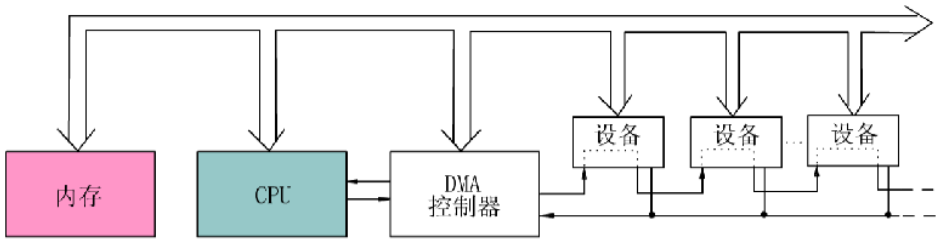
DMA控制器的类型

- 选择性
  - 物理上可以连接多个I/O接口（外设）；
  - 逻辑上只能连接一个设备，即在某一时间段只能为其中一台外设服务。
  - 适应于数据传输率很高（接近于内存）的外设数据传输服务。



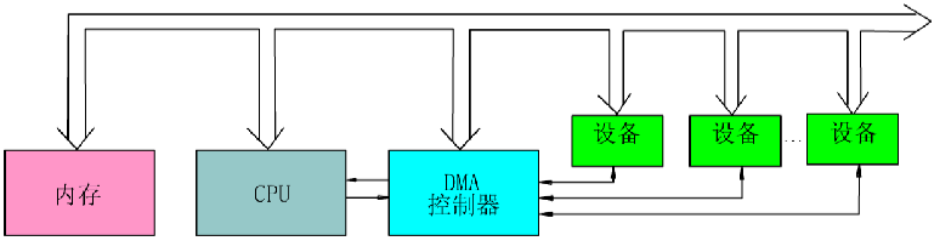
- 多路型DMA控制器
  - 物理上可以连接多个I/O接口（外设）；
  - 逻辑上也可连接多个设备，可通过交叉服务的方式为多台外设服务；
  - 多路型DMA控制器内部应包括多个DMA通道；

- 适应于多个慢速（相对）外设的数据传输服务。



链式多路型DMA

多路型DMA控制器



独立请求多路型DMA