顺序表

⑥按位查找

●获取表 L 中的第 i 个位置的元素的值。

●和访问普通数组方法一致：$return\ L.data[i-1]$;

●时间复杂度：$O(1)$

顺序表元素在内存中连续存放，可以根据起始地址和元素大小立即找到第 i 个元素——随机存取。

静态链表操作

(1)查找

从头结点出发依次往后遍历查找目标结点。

(2)插入

（位序为 i）找到一个空结点，存储数据元素，从头结点找到位序为 $i-1$ 的结点，修改新结点的 $next$，修改 $i-1$ 号结点的 $next$。

(3)删除

从头结点出发找到前驱结点，修改前驱结点的游标，被删除结点 $next$ 设为−2。表示该结点是一个空闲结点。

$Catalan$(卡特兰)数（真题 2015-2）

对于 n 个不同元素进栈，出栈序列的个数为$\frac{1}{n+1}C_{2n}^{n}$。

先序序列为a,b,c,d 的不同二叉树的个数、对于 n 个不同元素进栈，出栈序列的个数、以及 n 个节点的二叉树有多少种形态等问题都是考$Catalan$数。

10）由遍历序列构造二叉树☆

只通过一棵二叉树的 前序/中序/后序/层次 遍历序列，无法唯一的确定一棵二叉树。

(1)前序+中序遍历序列

前序：D A E F B C H G I　　**中序：**E A F D H C B G I

通过前序可知 D 为根，将中序分为 EAF**D**HCBGI，可知 EAF 为 D 的左子树结点，$HCBGI$ 为右子树的结点。分析 EAF，根据前序可知 A 为根，E 为 A 的左子树结点，F 为 A 的右子树结点，根据中序可知 EAF，也符合左中右。分析 $HCBGI$，根据前序可知 B 为根，HC 为 B 的左子树结点，GI 为 B 的右子树结点，根据前序为 CH，后续为 HC 可知，C 为根，H 为 C 的左子树。GI 同理。

(2)后序+中序遍历序列

后序：E F A H C I G B D　　**中序：**E A F D H C B G I

通过后序可知 D 为根，将中序分为 EAF**D**HCBGI，可知 EAF 为 D 的左子树结点，$HCBGI$ 为右子树的结点。分析 EAF，后序 EFA(左右根)可知 A 为根。分析 $HCBGI$，可知 B 为根，HC 和 GI 为左右子树。同上分析。

(3)层次+中序遍历序列

层次：D A B E F C G H I　　**中序：**E A F D H C B G I

根据层序遍历序列第一个结点确定根结点，可知 D 为根；根据根结点在中序遍历序列中分割出左右子树的中序序列；根据分割出的左右子树的中序序列从层序序列中提取出对应的左右子树的层序序列；对左子树和右子树分别递归使用相同的方式继续分解。

(4)前序+后序遍历序列

前序为根左右，后序为左右根，所以我们不能直接划分左右的顺序，无法唯一确定一棵树。

如果他们先序遍历和后序遍历的**相对顺序不一样**，则他们是在**同一棵子树**中，且为父子关系。若他们

的**相对顺序相同**，则他们在共同父节点（对他们的父节点而言的）的不同子树中。

通过先序遍历和后序遍历可以确定两个节点的相对位置关系，但无法构造出一棵确定的树。

(5)结论

①*前(or 后)序序列和中序序列可以唯一确定一棵二叉树。*

②*前序序列和后序序列不能唯一确定一棵二叉树，但可以确定二叉树中结点的祖先关系。当两个相邻结点的前序序列与后序序列顺序不同时为父子；相同为兄弟。*

线索二叉树存储结构

```
typedef struct ThreadNode{
   ElemType data;
   struct ThreadNode *lchild, *rchild;
   int ltag, rtag;          //新增的左、右线索标志
}ThreadNode, *ThreadTree;
```

2）B 树与 B+树对比

①**关键字的不同：**B+树中，n 个关键字的结点含有 n 棵子树，每个关键字对应一棵子树；B 树中，具有 n 个关键字的结点含有 $n+1$ 棵子树。☆

②**关键字个数范围：**B+树中，每个结点(除根)关键字个数 n 的范围是 $\lceil m/2 \rceil \leq n \leq m$ ，(根结点：$2 \leq n \leq m$)；B 树中，每个结点关键字个数范围是 $\lceil m/2 \rceil - 1 \leq n \leq m-1$，(根结点：$1 \leq n \leq m-1$)。☆

③**非叶子结点：**B+树中，非叶结点仅起索引作用，每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含关键字对应记录存储地址；B 树中，非叶结点包含全部信息。

④**叶子结点的不同：***B+树中，叶结点包含了全部关键字*，即在非叶结点中出现的关键字也会出现在叶结点中；在 B 树中，叶结点包含的关键字和其他结点包含的关键字是不重复的。

⑤**叶结点链接不同：**B+树中，有指向关键字最小叶子结点的指针，所有叶子结点按照大小顺序链接成线性链表；而 B 树没有。

⑥**查找不同：**B+树中，若非终端结点上的关键字等于给定值，不终止查找过程，继续向下直至到叶子结点，不管查找是否成功，每次查找都是走了一条从根到叶子结点的完整路径；B 树则不同，找到就终止，查找成功了。

⑦**应用：**B+树是应文件系统所需而产生的 B 树的变形，B+树比B 树更加适用于实际应用中的操作系统的文件索和数据库索引，B+树只要遍历叶子结点就可以遍历整棵树的遍历，磁盘读写代价更低，查询效率更加稳定。

原反补转换

(1)原码→反码

符号位保持不变，数值位按位取法。

(2)原码→补码

原码先转换为反码，再用反码转变为补码。

正数：原码、反码、补码表示都一样。

负数：符号位保持不变，而数值位按位取反，变为反码。末位加 1 就可以得到补码。

小技巧：从右往左找到第一个 1，在 1 的左边，所有的数值位都按位取反。

(3)补码→原码

负数：符号位不变，数值位取反，再在末尾加 1。（从右往左找到第一个 1，以这个 1 作为分界线，所有的数值位按位取反。）

IEEE754 特殊情况

①阶码全 0，尾数不全 0：非规格化的小数

阶码全 0 时，隐藏位为 0，并视为乘以 2^{-126}，$\mp(0.xx\ldots x)_2 \times 2^{-126}$

②阶码全 0，尾数全 0：真值 0

表示正 0 和负 0 两种情况，具体要看数符位。

③阶码全 1，尾数不全 0：非数值数 NaN

$NaN = Not\ a\ Number$，根据尾数最高位的不同决定是否进行异常处理。

④阶码全 1，尾数全 0：无穷大

表示正无穷大和负无穷大两种情况，具体要看数符位。表示正上溢或者负上溢。

冗余磁盘阵列——基本不考，可不看，当作拓展

RAID 0：无冗余，适合容量和速度要求高的非关键数据存储场合。当两个 I/O 请求访问不同盘上的数据，可并发发送。同一个 I/O 请求可能并行传送其不同的数据块(条区)。具体较快的 I/O 响应能力和较高的数据传输率。

RAID 1：镜像盘实现一对一冗余。一个读请求可由其中一个定位时间更少的磁盘提供数据。一个写请求可对两个磁盘进行并行更新。一个磁盘损坏，可以从另一个磁盘读取。

RAID 2：用海明校验法生成多个冗余盘，实现纠错一位，检测两位错误。读操作多盘并行性能高，写操作要同时写数据盘和校验盘。

RAID 3：采用奇偶校验法生成单个冗余盘。用于大容量 I/O 请求场合。

RAID 4：用一个冗余盘存放相应块的奇偶校验位。独立存取，每个磁盘操作独立进行，可同时响应多个 I/O 请求，

RAID 5：奇偶检验块分布在各个磁盘中（RAID3 和 RAID4 是在同一个磁盘中），提高容错性，小数据量的操作可以多个磁盘并行操作。

RAID 6：冗余信息均匀分布在所有磁盘上，数据以交叉方式存放。

RAID 7：带$Cache$的盘阵列，写入时，先写入$Cache$，读出时，先从$Cache$中读。

标志位总位数计算

例：主存地址 32 位，块大小为 16 字节，$Cache$总大小为$4K$块，求标志位总位数。

块内地址为 4 位。

直接映射方式：每组 1 块，共$4K(2^{12})$组，标志位占 32−4−12=16 位，总位数$4K \times 16 = 64K$位。

二路组相联映射：每组 2 块，共$2K(2^{11})$组，标志位 32−4−11=17 位，总位数$4K \times 17 = 68K$位。

全相联映射：每组$4K$块，共 1 组，标志位占 32−4=28 位，总位数$4K \times 28 = 112K$位。

Cache的总容量包括：存储容量、标记阵列容量（有效位、标记位、一致性维护位和替换算法控制位）。

标记阵列中的**有效位**和**标记位**是一定有的，而一致性维护位(脏位)和替换算法控制位的取舍标准要看题意，若题中说明了采用写回法，则一定包含一致性维护位；若关于替换算法的词眼题目中未提及，则不予考虑。若为直写方式，则不需要脏位。

标志位示例

若某主存地址空间大小为$256MB$，按字节编址，每个块大小 64 字节，$Cache$大小为$512B$。

则主存包含$256MB/64B = 2^{22}$个块，$Cache$有2^3个块。

(1)直接映射

前 22 位作为主存块号，直接映射每个主存块只能映射到唯一固定的位置，*Cache*中一共有 8 块，则主存块号对 8 取余后为*Cache*行号（如 1 和 9 对 8 取余都等于 1，都对应*Cache*的第 1 行），因为后 3 位是固定的，所以标记位不需要记录，硬件不需要取余操作，仅需把后 3 位截下来，所以标记位仅需 19 位即可。后 3 位反应每个主存块号应该存储在哪个*Cache*行。

如主存地址“0...0010010000000”意味着标记位 19 位“0...0010”，存储在“010”(第 2 行)。

(2)全相联映射

前 22 位作为主存块号，后 6 为作为块内地址，块内地址从 000000～111111。假设要把主存第 1 块（主存地址 0...0001000000~0...0001111111）放到*Cache*中，可以放入 Cache 中任何位置，假设放入第 2 块，则把其有效位置为“1”，标记位(22 位)记录为“0...0001”。

访问时，比如要访问的主存地址为“0...0010000000”，首先取出主存地址的前 22 位(主存块号)，与*Cache*中每一行进行对比。若相同，则检查有效位，若为 1，则命中。再根据后 6 位(块内地址)找到相应的字节。

(3)组相联映射

每一个主存块可以被放到特定分组当中的任意一个位置。前 22 位作为主存块号，对分组数进行取余操作，来确定把它放在哪一组。

假设采用二路组相联映射（2 块为一组，共 4 组），如主存地址“0...00100 10 000000”可以直接看主存块号的后两位为“10”，表明它可以放在第三(10)组的任意一个位置，因为同一组的主存块的末尾两位都是相同的，没有必要记录这两位，所以标记位只需记录主存块号的前 20 位。

访问时，假如要访问主存地址“1...10100 11 001100”，取出主存块号的后两位 11，可知他在*Cache*中的副本只能存放在编号为 11 的分组中，接下来在 11 分组中挨个对比标记位和我们鬼畜的地址标记是否匹配，若匹配且有效位为 1 则*Cache*命中。

虚拟存储器容量限制

虚拟存储器容量既不受外存容量限制，也不受内存容量限制，而由 CPU 的寻址范围决定的。

虚拟内存**最大容量**：地址结构确定；

虚拟内存**实际容量**：*min*(内存+外存，CPU 寻址)

TLB/页表/Cache 缺失可能性分析

	TLB	页表	Cache	
①	√	√	×	可能，TLB命中则页表一定命中，但实际上不会查页表
②	×	√	√	可能，TLB缺失但页表可能命中，信息在主存，就可能在*Cache*
③	×	√	×	可能，TLB缺失但页表可能命中，信息在主存，但可能不在*Cache*
④	×	×	×	可能，TLB缺失则页表可能缺失，信息不在主存，一定也不在*Cache*
⑤	√	×	×	不可能，页表缺失，说明不在主存，TLB中一定没有该页表项
⑥	√	×	√	同上
⑦	×	×	√	不可能，页表缺失，说明不在主存，*Cache*中一定也没有该信息

情况①和②，只需访问主存 1 次；情况④需访问磁盘、并访存至少 2 次。

情况③不需访问磁盘、但访存至少 2 次。

Cache 经典真题(2013-16)

某计算机主存地址空间大小为 256 *MB*，按字节编址。虚拟地址空间大小为 4 *GB*，采用页式存储管理，页面大小为 4 *KB*，*TLB* (快表)采用全相联映射，有 4 个页表项，则对虚拟地址 03FF F180H 进行虚实地址变换的结果是

有效位	标记	页框号
1	03FFFH	0153H
…	…	…

虚地址有 32 位（$4\,GB = 2^{32}\,B$），页面大小为$4\,KB\ (2^{12}B)$，那么虚页号有 20 位。页内地址有 $32-20=$

12 位。实地址有 28 位($256MB = 2^{28}B$)，实地址由页框号和页内地址组成，其页内地址与虚地址的页内地址相同都为 12 位不变，则页框号有 16 位。

虚拟地址为$03FF\ F180H$，其中页号为$03FFFH$，页内地址为$180H$，根据题目中给出的页表项可知页标记为$03FFFH$所对应的页框号为$0153H$，且有效位为 1，页框号与页内地址之和即为物理地址$015\ 3180H$。若有效位为 0，则 TLB 缺失。

局部性原理题型(考过类似的，且难度更大)

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

←←←高级语言源代码

对应的汇编语言程序↓↓↓

```
I0:          sum <-- 0
I1:          ap  <-- A          A 是数组 a 的起始地址
I2:          i   <-- 0
I3:          if (i >= n) goto done
I4:   loop:  t   <-- (ap)       数组元素 a[i]的值
I5:          sum <-- sum + t    累计在 sum 中
I6:          ap  <-- ap + 4     计算下 1 个数组元素的地址
I7:          i   <-- i + 1
I8:          if (i < n) goto loop
I9:   done:  v   <-- sum        累计结果保存至地址 v
```

主存的布局：

$0x0FC$	I0	指令
$0x100$	I1	
$0x104$	I2	
$0x108$	I3	
$0x10C$	I4	
$0x110$	I5	
$0x114$	I6	
	…	
$0x400$	$a[0]$	A 数据
$0x404$	$a[1]$	
$0x408$	$a[2]$	
$0x40C$	$a[3]$	
$0x410$	$a[4]$	
$0x414$	$a[5]$	
	…	
$0x7A4$		v

BACK

程序段 A：

```
int sumarrayrows(int A[M][N]){
    int i, j, sum = 0;
    for(i = 0; i < M; i++)
        for(j = 0 ;j < N; j++)
            sum += A[i][j];
    return sum;
}
```

程序段 B：

```
int sumarraycols(int A[M][N]){
    int i, j, sum = 0;
    for(j = 0; j < N; j++)
        for(i = 0 ;i < M; i++)
            sum += A[i][j];
    return sum;
}
```

每条指令占 4 个字节，每个数组元素占 4 个字节。sum、ap、i、t 为寄存器，A、V 为内存地址。

1）例一

问： 指令和数据的时间局部性和空间局部性各自体现在哪里？

指令： $0x0FC$（I0）→ $0x108$（I3）→ $0x10C$（I4）→ $0x11C$（I8）→ $0x120$（I9）

执行到 I8 会跳转到 I4。循环内指令的时间局部性好。指令按顺序存放，空间局部性好。

数据： 数组元素按顺序存放，按顺序访问，空间局部性好。

2）例二

问： 假定数组在存储器中以行优先顺序存放，比较程序A和B中，数组A、变量sum和for循环体的空间局限性和时间局限性。($M = N = 2048$)

程序A数组A： 访问顺序为$A[0][0], A[0][1] \ldots A[0][2047], A[1][0] \ldots$与存放顺序一致，空间局部性好。每个$A[i][j]$都只访问一次，时间局部性差。

程序B数组A： 访问顺序为$A[0][0], A[1][0] \ldots A[2047][0], A[0][1] \ldots$与存放顺序不一致，空间局部性差。每个$A[i][j]$都只访问一次，时间局部性差。

变量sum： 单个变量无需考虑空间局部性，每次循环都需访问sum，时间局部性好。

for循环体： 指令是按序存放的，空间局部性好。循环体被重复执行，时间局部性好。

定长/扩展操作码指令经典例题

假设字长为 16 位，操作数的地址码为 6 位，指令有零地址、一地址、二地址三种格式。
①设操作码定长，若零地址指令 P 种，一地址指令有 Q 种，则二地址指令最多多少种？
②采用扩展操作码技术，若二地址指令 X 种，零地址指令 Y 种，则一地址指令最多多少种？

①地址码为 6 位，则二地址指令操作码位数$16-6-6=4$，可有$2^4=16$种操作，除去零地址和一地址的种类数，则二地址指令最多有$16-P-Q$种。

②二地址、一地址、零地址操作码长度分别为 4 位、10 位、16 位。二地址指令 X 种，则一地址指令最多$(2^4-X)\times 2^6$种。设一地址指令有 M 种，则零地址指令最多有$[(2^4-X)\times 2^6-M]\times 2^6$种，则 $Y=[(2^4-X)\times 2^6-M]\times 2^6$，解出一地址指令$M=(2^4-X)\times 2^6-Y\times 2^{-6}$

寻址方式题型

1）常见隐含条件

①存储器直接寻址空间为 128 字→每个地址码为 7 位（$128=2^7$）
②32 个通用寄存器可作为变址寄存器→寄存器编号为 5 位（$32=2^5$）
③变址时位移量为$-128\sim+127$→形式地址 A 取 8 位（$256=2^8$）
④能完成 100 种操作→操作码 7 位（$2^6<100<2^7$）
⑤寻址方式有 5 种→寻址特征位 3 位（$2^2<5<2^3$）

2）寻址范围

例：某机器有基址寄存器和变址寄存器，可完成 105 种操作，采用一地址格式的指令系统，允许直接和间接寻址，指令字长、机器字长、存储字长均 16 位。

寻址方式有 4 种，则寻址特征位占 2 位；105 种操作，操作码占 7 位。
形式地址 A 占：$16-7-2=7$位。
直接寻址范围：$2^7=128$；一次间址寻址范围：$2^{16}=64K$

3）求操作数内容

例：假设变址寄存器 R 的内容为 1000H，指令中的形式地址为 2000 H;地址 1000H 中的内容为 2000H，地址 2000H 中的内容为 3000H，地址 3000H 中的内容为 4000H，则变址寻址方式下访问到的操作数是多少？（真题 2013-17）

变址寻址是把变址寄存器的内容与指令中给出的形式地址 A 相加，形成操作数有效地址，即$EA=($变址寄存器$)+A$。操作数 S 与地址码和变址寄存器的关系为$S=((变址寄存器)+A)$。
(变址)+(形式)为实际地址，有效地址$=(R)+A=3000H$，操作数为$(3000H)=4000H$

CISC/RISC 详细比较——了解/拓展

(1)指令系统

CISC：计算机指令系统比较丰富，有专用指令来完成特定的功能。处理特殊任务效率较高。

RISC：设计者把主要精力放在那些经常使用的指令上，尽量使它们具有简单高效的特色。对不常用的功能，常通过组合指令来完成。因此，在 RISC 机器上实现特殊功能时，效率可能较低。但可以利用流水技术和超标量技术加以改进和弥补。

(2)存储器操作

CISC：机器的存储器操作指令多，操作直接。

RISC：对存储器操作有限制，使控制简单化。

(3)程序

CISC：汇编语言程序编程相对简单，科学计算及复杂操作的程序设计相对容易，效率较高。

RISC：汇编语言程序一般需要较大的内存空间，实现特殊功能时程序复杂，不易设计。

(4)中断

CISC：机器是在一条指令执行结束后响应中断。

RISC：机器在一条指令执行的适当地方可以响应中断。

(5)CPU

CISC：CPU 包含有丰富的电路单元，因而功能强、面积大、功耗大。

RISC：CPU 包含有较少的单元电路，因而面积小、功耗低。

(6)设计周期

CISC：微处理器结构复杂，设计周期长。

RISC：微处理器结构简单，布局紧凑，设计周期短，且易于采用最新技术。

(7)用户使用

CISC：微处理器结构复杂，功能强大，实现特殊功能容易。

RISC：微处理器结构简单，指令规整，性能容易把握，易学易用。

(8)应用范围

CISC：$CISC$ 机器更适合于通用机。

RISC：由于$RISC$指令系统的确定与特定的应用领域有关，故 $RISC$ 机器更适合于专用机。

寄存器

①**普通寄存器：**一定会有控制信号控制读/写（可能分开—2 bit；可能合并—1 bit）

②**暂存寄存器：**单总线结构下，总线只能有一个输入，但是ALU需有两个数据，故要通过先把一个数据放入暂存寄存器中，然后以一个数据从总线输入，一个数据从暂存寄存器输入的方式实现ALU运算。

③**通用寄存器组：**需要有控制信号用于选择是哪个通用寄存器。

④**带有移位功能的寄存器：**每左移一位，真值*2；每右移一位，真值/2。

⑤**带自增功能的寄存器：**PC寄存器可以通过自增功能自动实现PC + “1”。

(1)用户可见的寄存器

①**通用寄存器：**可由程序设计者指定许多功能，可用于存放操作数，也可作为满足某种寻址方式所需的寄存器。

②**数据寄存器：**用于存放操作数，位数满足多数数据类型的数值范围，允许使用两个连读的寄存器存放双倍字长的值。

③**地址寄存器：**用于存放地址，具有通用性，可用于特殊的寻址方式。位数必须足够长，以满足最大的地址范围。

④**条件码寄存器：**存放条件码，对用户来说是部分透明的。条件码是CPU根据运算结果由硬件设置的位。将条件码放到一个或多个寄存器中，就构成了条件码寄存器。

⑤**指令计数器（PC）：**放现行指令的地址，通常具有计数功能。当遇到转移类指令时，PC的值可被修改。

⑥**程序状态字寄存器（PSW）：**用于记录当前处理器的状态和控制指令的执行顺序，并且保留与运行程序相关的各种信息，主要作用是实现程序状态的保护和恢复。

(2)用户不可见的寄存器

①**存储器地址寄存器（MAR）：**用于存放将被访问的存储单元的地址。

②**存储器数据寄存器（MDR）：**用于存放欲存入存储器的数据或最近从存储器读出的数据。

③**指令寄存器（IR）：**存放当前欲执行的指令。

④**高速缓存$Cache$**。

(3)对汇编程序员可见☆

①**中断字寄存器：**可以修改中断的优先级。　②**基址寄存器：**基址寻址。

③**变址寄存器：**变址寻址，如数组的访问需要。　④用户可见寄存器中的所有寄存器。

常见执行周期——了解/拓展

(1)非访存指令

CLA——清除 ACC——$0 \rightarrow ACC$

COM——取反——$\overline{ACC} \rightarrow ACC$

SHR——算术右移——$L(ACC) \rightarrow R(ACC)$，$ACC_0 \rightarrow ACC_0$

CSL——循环左移——$R(ACC)L(ACC)$，$ACC_0 \rightarrow ACC_n$

STP——停机指令——$0 \rightarrow G$（运行标志触发器）

(2)访存指令

加法指令 $ADD\ X$——$Ad(IR) \rightarrow MAR$，$1 \rightarrow R$，$M(MAR) \rightarrow MDR$，$(ACC) + (MDR) \rightarrow ACC$

存数指令 $STA\ X$——$Ad(IR) \rightarrow MAR$，$1 \rightarrow W$，$ACC \rightarrow MDR$，$MDR \rightarrow M(MAR)$

取数指令 $LDA\ X$——$Ad(IR) \rightarrow MAR$，$1 \rightarrow R$，$M(MAR) \rightarrow MDR$，$MDR \rightarrow ACC$

(3)转移指令

无条件转移 $JMP\ X$——$Ad(IR) \rightarrow PC$

有条件转移 $BAN\ X$（负数则转移）——$A_0 \cdot Ad(IR) + \overline{A_0}(PC) \rightarrow PC$，通过最高位$A_0$判正负。

1）非访问主存指令 CLA 的功能

对累加器 AC 清零。

机器周期	取指译码	指令执行
第 1 个机器周期	取指令→PC+1→指令译码	
第 2 个机器周期		执行指令

(1)取指令阶段（第一个CPU周期）：

①$PC \rightarrow MAR$；②$PC + 1$；③$MAR \rightarrow ABUS$；④$M \rightarrow DBUS \rightarrow MDR$；⑤$MDR \rightarrow IR$

⑥指令译码、测试；⑦CPU 识别出 CLA 指令。

(2)执行指令阶段（第二个CPU周期）：

①控制器送一控制信号让 ALU 工作；②$0 \rightarrow AC$

2）直接访主存指令 ADD 30 的功能

将累加器 AC 的内容加上地址为 30 存储单元的内容。

机器周期	取指译码	指令执行	
第 1 个机器周期	取指令→PC+1→指令译码		
第 2 个机器周期		送操作数地址	
第 3 个机器周期			取操作数→执行指令

(1)取指令阶段（第一个CPU周期）：同上

(2)执行指令阶段

送操作数地址（第二个 CPU 周期）：IR（30） → MAR

两操作数相加（第三个 CPU 周期）：①$MAR \rightarrow ABUS$；②$M \rightarrow DBUS \rightarrow MDR$；③$ACC + MDR \rightarrow ACC$；

3）间接访主存指令 STA 40 的功能

将累加器的值存储到地址为 40 的存储单元。

机器周期	取指译码	指令执行		
第 1 个机器周期	取指令→PC+1→指令译码			
第 2 个机器周期		送地址指针		
第 3 个机器周期			取出地址	
第 4 个机器周期				取操作数→执行指令

(1)取指令阶段（第一个CPU周期）：同上。

(2)执行指令阶段

送地址指示器（第二个 CPU 周期）：$IR(40) \rightarrow MAR$

存储和数（第三个 CPU 周期）：①$ACC \rightarrow MDR$；②$MAR \rightarrow ABUS$；③$MDR \rightarrow DBUS$；④$DBUS \rightarrow M$。

4）程序控制指令 NOP 的功能

空操作。

(1)取指令阶段（第一个CPU周期）：同上。

(2)执行指令阶段（第二个CPU周期）：CPU 空转，不发出任何控制信号。

5）程序控制指令 JMP 21 的功能

程序无条件的转移到存储单元地址为 21 的指令开始执行。

机器周期	取指译码	指令执行
第 1 个机器周期	取指令→PC+1→指令译码	
第 2 个机器周期		送转移地址

(1)取指令阶段（第一个CPU周期）：同上。

(2)执行指令阶段（第二个CPU周期）：$IR(21) \rightarrow PC$

定时和分类

总线定时：双方交换数据时间上配合关系，实质上是一种协议和规则。

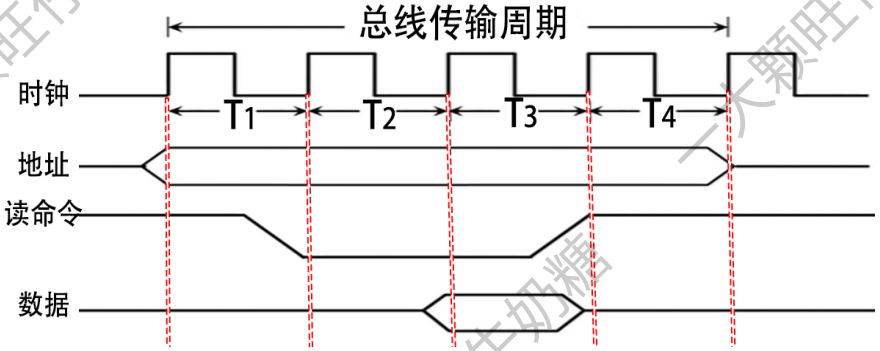
主设备：具有控制总线的能力，能主动发起总线传输操作的设备。可根据时钟信号控制总线上的数据传输和操作。可以发送命令、读取/写入数据操作，并控制从设备的访问。

从设备：被动等待主设备控制和指令的设备。从设备无法主动发起总线传输操作，它们只能在接收到来自主设备的命令或请求后才能进行相应的响应。

1）同步通信(同步定时方式)

由统一时钟控制数据传送定时关系。每个操作和信号都在固定的时间点。有定宽定距的时钟，来控制整个数据传输的过程。

(1)读命令/数据输入



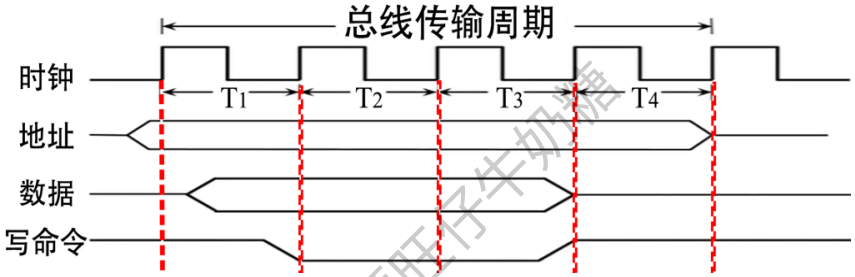
①T_1上升沿，主设备(CPU)给出**地址信号**，即PC发出命令，$PC \rightarrow MAR$，存储体→MDR→IR。T_1时间段内完成。

②T_2上升沿，IR给出**读命令信号**，在T_2时间段内，主设备告诉从设备要从从设备读入数据。

③T_3上升沿，从设备通过数据总线给出要传输的**数据信号**，T_3时间段内，主设备读取数据。

④T_4上升沿，撤销数据信号和控制信号，T_4时钟周期结束，地址信号撤销。

(2)写命令/数据输出



①T_1上升沿，给出**地址信号**，T_1下降沿，给出**数据**。

②T_2上升沿，给出**写命令**，向从设备进行数据写入。T_3仍进行写入操作。

③T_4上升沿，主设备(CPU)撤销数据和写命令。T_4结束，撤销地址信息。

(3)Notes

①当主设备和从设备速度差异较大，从设备可能无法跟上在特定时间内完成读/取操作→所接部件存取时间接近。适用于总线长度比较短，各模块存取时间一致的情况。

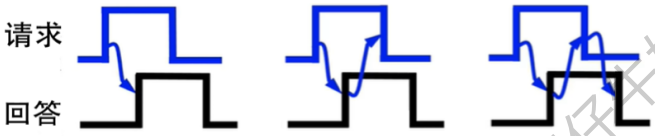
总线长度比较短：任何两个设备之间的通信都给予同样的时间安排。所以必须按距离最长的两个设备的传输延迟来设计公共时钟。总线过长会降低传输频率。

②可靠性差：可能没有多余时间进行数据检验→适用总线长度短（出错概率低）。

③一个周期最左边竖线为**上升沿**；中间的竖线为**下降沿**。

2）异步通信(异步定时方式)

采用应答方式、设有公共时钟标准。不会有统一的节拍。通过请求和回答两个握手信号来完成联络。没有定宽定距的时钟，但需要增加两条线：请求线和应答线。



（上面为主设备，下面为从设备）

①**不互锁方式**：主设备发出请求后，不必等待从设备的回答信号，经过一段时间便撤销请求信号。通信的可靠性存在问题。

②**半互锁方式**：主设备发出请求信号后，必须接到从设备的回答信号后才撤销请求信号；而从设备发出回答信号后不必等到主设备的请求信号撤回，经过一段时间后自动撤销回答信号。

③**全互锁方式**：主设备发出请求信号后，必须接到从设备的回答后才撤销请求，从设备发出回答信号后，必须等到主设备请求撤销后才撤销回答信号。

优点：总线周期长度可变，能保证两个工作速度相差很大的部件或设备之间可靠地进行信息交换，自动适应时间的配合。

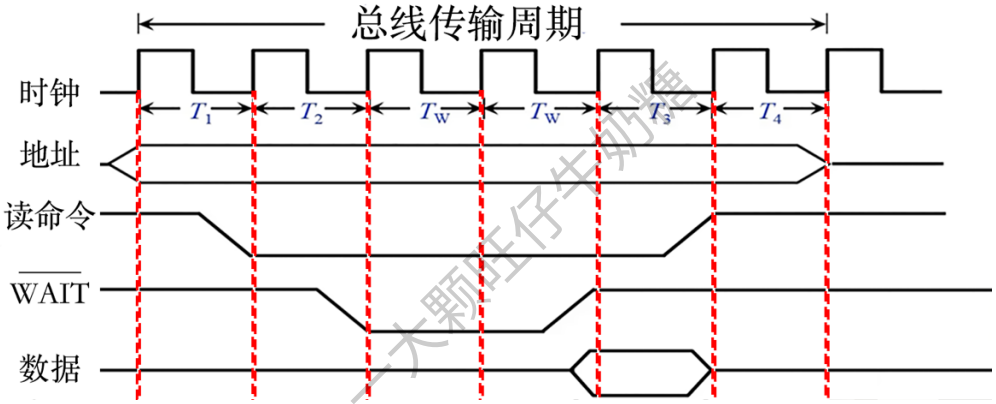
缺点：比同步控制方式稍复杂一些，速度比同时定时方式慢。

3）半同步通信（同步和异步思想结合）

同步/异步结合，根据WAIT信号（从设备发出）动态调节统一时钟节拍，解决不同速度的两个模块/设备之间进行通讯的问题。

(1)过程

假设主设备(CPU)要从某个外部设备/内存单元中读入数据。



①T_1上升沿，主设备通过地址总线发送地址信号。T_2上升沿，主模块发送读信号命令。
②T_3周期开始，从设备不能把数据准备好，通过 WAIT 信号给出低电平，告诉主设备等待。
③主设备(CPU)检测WAIT信号，如果是低电平，则在T_3之前，插入一个WAIT周期。
④检测到WAIT变为高电平，表明数据准备好，且在数据总线上，主设备进行数据接收。
⑤T_4周期开始，撤销读命令信号和数据信号。T_4结束，地址信号撤销，半同步传输结束。

(2)Notes

①$\overline{\text{WAIT}}$信号低电平有效。
②**同步体现在：**有一个定宽定距的时钟来管理整个通信过程。
③**异步体现在：**不要求两个设备以相同速度进行工作，允许不同速度的设备之间协调工作。

4）分离式通信

(1)概念及优化

充分挖掘系统总线每瞬间潜力，利用总线空闲的时间将总线分配给其他设备。

从设备接收到地址和读命令之后，会准备数据，此时不需要使用总线，但主从设备依然会占据总线使用权。**优化：**主设备和总线之间的连接断开，主设备放弃总线的使用权。

从设备准备好数据之后，从模块变为主模块，发起总线的占用请求，进行数据传输。

(2)Notes

①各模块有权申请占用总线：所有模块都可以从从模块变成主模块。

②采用同步方式通信，不等对方回答。从模块准备好数据之后，后面的通讯过程，是通过一个定宽定距的时钟来完成的。

③各个模块准备数据时，不占用总线。

④总线被占用时，无空闲，总线被占用的时候，一定在进行数据通信，或是在传输/控制命令。充分利用了带宽。

⑤一个总线传输周期分为两个子周期：

子周期 1：主模块申请占用总线，使用完后，放弃总线的使用权。
子周期 2：从模块申请占用总线，将各种信息送至总线上。

IO 接口——Notes

(1)*I/O 总线的数据线传输的信息包括：命令字、状态字、中断类型号*

①通道的方式允许 **I/O 命令**通过数据线传输。

②状态字与设备状态容易混淆，状态字不仅仅是一位，其包含了设备的许多工作状态信息，需要用数据线传输。

③中断类型号是指中断向量的地址，即中断服务程序的入口地址，其需要数据线传输。中断类型号是固定的，中断类型号来自中断源。

(2)传输线

I/O 接口与*CPU*之间的 I/O 总线有数据线、命令线和地址线。命令线和地址线都是单向传输的，从*CPU*传送给 I/O 接口，而 I/O 接口中的命令字、状态字以及中断类型号均是由 I/O 接口发往*CPU*的，只能通过 I/O 总线的数据线传输。

I/O 方式类比

例：假设你正在做家务，需要将一堆衣服从洗衣机中取出、晾晒和折叠。

程序查询方式：你会不断地检查洗衣机是否已经完成洗涤程序。每隔一段时间，你会去检查一次洗衣机的状态。如果洗衣机已经完成，你会立即将衣服取出来进行下一步的处理。这种方式需要你不断地关注和查看洗衣机的状态，然后根据结果来决定下一步的操作。

程序中断方式：你可以设置一个定时器。当定时器到达设定的时间时，它会发响铃提醒你去处理洗衣机中的衣服。定时器就相当于一个中断信号，它打断了你当前正在做的事情，并提示你去处理优先级更高的任务。

***DMA*方式：**你可以让你的小助理来帮助你处理家务。小助理会帮你从洗衣机中取出衣服，并晾晒折叠。你只需要告诉小助理如何做，比如衣服的数量和晾衣架的位置，然后让小助理完成任务。这样你可以同时进行其他的家务活动，而不需要亲自去处理每一件衣服。

程序查询方式拓展细节——基本不考

①程序查询方式要完成**内存**与**外设**之间数据的输入输出，需要借助*CPU*中的某一个**寄存器**对数据进行暂存：外设将数据传输到*CPU*指定的寄存器中，然后*CPU*再将数据从寄存器中读取出来，并将其写入内存中的指定位置（从内存写入外设同理）

②通过设置计数器的值控制传输的数据量。设置主存缓冲区地址，缓冲区用于暂存数据的内存区域。在完成一次数据传输后，准备输入或输出下一个数据块，要修改主存缓冲区的地址和计数器。

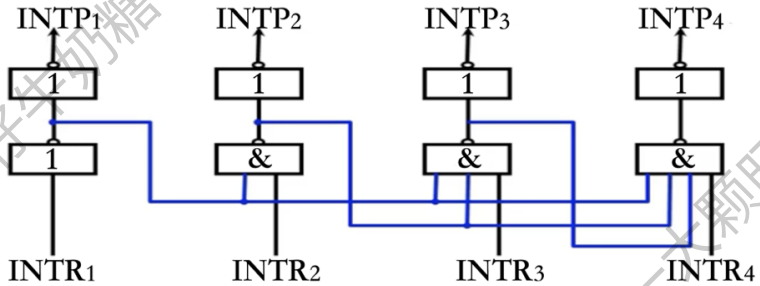
③在数据准备好之前，*CPU*会原地踏步查询状态，直到设备把数据通过数据线放到数据缓冲寄存器*DBR*中。

中断判优——了解/拓展

当某一时刻有多个中断源提出中断请求时，中断系统必须按其优先顺序予以响应，成为中断判优。

(1)硬件实现

链式排队器：下图 $INTR_1$ 到 $INTR_4$ 优先级是降序的。



当 $INTR_1$ 有信号，发出高电平，并把低电平信号传到 $INTR_2$ 的硬件电路中，此时 $INTR_2$ 的信号与从 $INTR_1$ 传过来的“0”相与以后一定会为“0”，会使其低优先级中断源输出 *INTP* 为低电平，实现屏蔽。*PS：*图中的小圈圈代表信号取反。

(2)软件实现

利用程序查询的方式来控制中断源的优先级。

如，优先级A、B、C降序，首先程序查询A是否请求中断，是则转到A的服务程序入口地址，否则程序查询B是否请求中断...依次再查询C。

DMA 方式和中断方式的区别☆☆

①*中断驱动方式在每个数据需要传输时中断 CPU，而 DMA 控制方式则是在所要求传送的一批数据全部传送结束时才中断 CPU；*

②中断方式需要保护和恢复现场，*DMA* 方式除预处理和后处理其他时候不占用 *CPU* 资源。

③对中断请求的响应只能发生在每条指令的中断周期；而对 *DMA* 请求的响应可以发生在每个机器周期结束时（即在取指、间址、执行周期后均可）

④*中断驱动方式数据传送是在中断处理时由 CPU 控制（软件）完成的，而 DMA控制方式则是在DMA 控制器的控制下完成的。*中断传送过程需要 *CPU* 的干预；*DMA* 方式不需要。

⑤*DMA* 请求优先级高于中断请求。

⑥中断方式可以处理异常事件；*DMA* 方式只能用于 I/O 操作。

⑦中断方式靠程序(CPU 执行中断服务程序)传送；*DMA* 方式靠硬件(DMA 控制器)传送。

⑧*DMA* 本质是程序中断方式扩展：*DMA* 每个**块**发出中断，程序中断方式每个**字**发出中断。

⑨程序查询方式和程序中断方式可能发生数据丢失；但是 *DMA* 不可能发生数据丢失。

⑩中断方式用于低速设备；*DMA* 方式用于高速设备（多路型 DMA 控制器可同时为多个低速外围设备服务。）

切换/调度开销

多道程序环境系统要付出额外的开销来组织作业和切换作业，管理内存以及处理各种任务。

①**作业调度开销：**操作系统需要选择哪个作业或进程在 CPU 上执行。这涉及到作业调度算法的开销，例如轮转调度、最短作业优先调度等。操作系统必须不断地评估各个作业的状态，并决定下一个要运行的作业，这会耗费一定的 CPU 时间。

②**上下文切换开销：**当操作系统决定切换到另一个作业或进程时，必须进行上下文切换。这包括保存当前进程的状态（寄存器值、程序计数器等），加载下一个进程的状态，并进行切换。上下文切换本身需要时间，并且会增加系统的开销。

③**内存管理开销：**多道程序环境需要有效地管理内存，确保各个作业或进程能够共享和使用内存资源。涉及到内存分配、分页/分段、虚拟内存管理等。内存管理需要额外的计算和控制。

④**资源竞争开销：**多道程序环境中，多个作业或进程可能竞争系统资源，如 CPU 时间、内存、I/O 设备等。操作系统要进行处理，确保资源分配的公平性和效率，这也会引入一些开销。

⑤**进程间通信和同步开销：**在多道程序环境中，不同的进程可能需要相互通信或同步。这需要一些机制，如信号量、消息传递等，这些机制的实施和维护也需要开销。

子程序调用

(1)概念

●系统调用是用户程序调用系统程序，子程序调用是用户程序调用用户程序。

●过程调用和函数调用都是子程序调用。（函数调用详见 P127）

(2)子程序调用 与 中断处理 区别（2012. 24 真题）

子程序调用只需保护程序断点，即该指令的下一条指令的地址；中断处理不仅要保护断点（*PC* 的内容），还要保护程序状态字寄存器的内容 *PSW*。

无论中断处理和子程序调用都不需要保存通用数据寄存器和通用地址寄存器的内容。在中断处理过程中程序计数器(*PC*)和程序状态字(*PSW*)寄存器的内容都要压栈保存。在子程序调用时，由系统硬件限制只保存程序计数器的内容，如果需要保存 *PSW* 内容可由软件来实现。

程序状态字(*PSW*)寄存器用于记录当前处理器的状态和控制指令的执行顺序，并且保留与运行程序相关的各种信息，主要作用是实现程序状态的保护和恢复。子程序调用在进程内部执行，不会更改 *PSW*。

线程补充

(4)线程共享/独立环境

①**同一进程内线程共享的环境：**进程代码段、进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)、进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户 *ID*与进程组 *ID*。

②**同一进程内线程独立的环境：**线程 *ID*、寄存器组的值、线程的堆栈、错误返回码、线程的信号屏蔽码、线程的信号屏蔽码。

(5)线程与进程

	进程	线程
资源分配	进程是资源分配和拥有的基本单位	线程自己基本不拥有系统资源，但它可访问所属进程所拥有的全部资源
调度	在没有引入线程的操作系统中，进程是独立调度和分派的基本单位。	引入线程后的操作系统中，线程是独立调度和分派的基本单位。
地址空间	进程之间相互独立，使用独立的内存空间。	同一进程的各进程共享进程的内存空间。

(6)进程和线程内存空间的互斥

①同一进程中不同线程之间的局部变量不会相互影响，且不同线程之间有各自的栈空间，不需要互斥。

②进程有各自的地址空间，不同进程之间内存空间不同，不相互影响，故不需要互斥。

③进程的全局变量会相互影响，要互斥；进程下的全局变量定义在堆空间中，仅此一份，故需要互斥访问。

*Peterson*算法

●**算法思想：**

●防止两个进程为进入临界区而无限期等待，设置了变量 *turn*，用于进程间互相“谦让”。每个进程在先设置自己的标志后再设置 *turn*标志。。

●同时检测另一个进程状态标志和不允许进入标志，以便保证两个进程同时要求进入临界区时，只允许一个进程进入临界区。

●$bool\ flag[2]$;//表示进入临界区意愿的数组，初始值都是$false$。

●$int\ turn = 0$;//turn 表示优先让哪个进程进入临界区。

一般情况下，如果进程P_0试图访问临界资源，设置$flag[0] = true$，表示希望访问。此时如果进程P_1还未试图访问临界资源，则$flag[1]$在进程上一次访问完临界资源退出临界区后已设置为$false$。所以进程P_0在执行循环判断条件时，第一个条件不满足，进程P_0可以正常进入临界区，且满足互斥条件。

两个进程同时试图访问临界资源的情况。注意 *turn*变量的含义：进程在试图访问时，首先设置自己的$flag$变量为 *true*，表示希望访问；但又设置 *turn*变量为对方的进程编号，表示“谦让”，因为在循环判断条件中 *turn*变量不是自己编号时就循环等待。这时两个进程就会互相“谦让”一番，但是这不会造成饥饿的局面，因为 *turn*变量会有一个最终值，所以必定有进程可以结束循环进入临界区。

例：先作出“谦让”的进程先进入临界区，后作出“谦让”的进程则需要循环等待。其实这里可以想象为两个人进门，每个人进门前都会和对方客套一句“你走先”。如果进门时没别人，就当和空气说句废话，然后大步登门入室；如果两人同时进门，就互相请先，但各自只客套一次，所以先客套的人请完对方，就等着对方请自己，然后光明正大进门。

管程补充

4）实现进程互斥

①***局部于管程的数据只能被局部于管程的过程所访问：***同一管程内的过程仅能访问同一管程内的数据结构，同一管程内的数据结构也只能被同一管程内的过程访问。（完全封闭的）

②一个进程只有通过调用管程内的过程才能进入管程访问共享数据；

③每次***仅允许一个进程在管程内执行某个内部过程。只能串行执行管程内的过程***（操作管程内的临界资源），实现进程互斥。

④管程类型提供了一组由程序员定义的、在管程内互斥的操作，确保一次只有一个进程在管程内活动。

⑤进程通过管程请求临界资源未满足时，管程将其阻塞加入等待队列，直到阻塞的原因解除时，如果该进程不释放管程，那么其他进程无法进入管程。

5）Notes

①管程是进程同步工具，解决信号量机制大量同步操作分散的问题；
②管程每次只允许一个进程进入管程；
③管程是被进程调用的，管程是语法范围，无法创建和撤销；
④管程中定义的变量只能被管程内的过访问；函数的具体实现外部不可见;
⑤管程是由编程语言支持的进程同步机制;
⑥管程不仅能实现进程间的互斥（编译器负责保证），而且能实现进程间的同步（设置条件变量及等待/唤醒操作）；
⑦管程的 signal 操作与信号量机制中的 V 操作不同；
⑧管程是中不仅有数据，而且有对数据的操作。

7）多级页表

(1)多级页表

在两级页表的基础上拓展更多层，页表和页面一样也进行分页。

(2)特点

①多级页表不会加快地址的变换速度，因为增加更多的查表过程，会使地址变换速度减慢；
②多级页表不会减少缺页中断的次数，反而如果访问过程中多级的页表都不在内存中，会大大增加缺页的次数；
③多级页表不会减少页表项所占的字节数，页表项所占字节是个常数；
④多级页表能够减少页表所占的连续内存空间，即当页表太大时，将页表再分级，可以把每张页表控制在一页之内，减少页表所占的连续内存空间。

(3)例子

题目： 某系统按字节编址，采用40位逻辑地址，页面大小$4KB$，页表项大小$4B$，则采用（）级页表，页内偏移量为（）位。

解： 页面大小 $= 4\text{KB} = 2^{12}\text{B}$，页内偏移量为 12 位；页号 $= 40 - 12 = 28$ 位；每个页面可存放$2^{12}/4 = \text{x} = 2^{10}$个页表项；每级页表对应页号为 10 位，总共 28 位页号至少要分三级。

(4)类比

①假设有一本书，书每一页固定大小，最多一页有1000个字（页面大小）；
②每一页对应有一个目录中的条目，每个条目用2个字表示（页表项大小）；
③那么一页纸就可以写下$1000/2 = 500$个目录条目；
④现在我整本书有1000页（逻辑地址空间大小），每一页对应一个目录条目，所以有1000个目录条目；
⑤所以需要1000 个目录条目/500 个目录条目 $= 2$ 页（一页可写500个条目）。所以需要2个页面来保存目录条目。

地址变换机构

分段系统的逻辑地址A到物理地址E之间的地址变换过程：

①从逻辑地址A中取出前几位为段号S，后几位为段内偏移量W，（段式存储管理的题目中，逻辑地址一般以二进制给出，而在页式存储管理中，逻辑地址一般以十进制给出）。

②比较段号S和段表长度M，若$S \geq M$，则产生越界异常，否则继续执行。

③段表中段号S对应的段表项地址=段表起始地址F +段号S ×段表项长度，取出该段表项的前几位得到段长C。若段内偏移量≥C，则产生越界异常，否则继续执行。段表项实际上只有两部分，前几位是段长，后几位是起始地址。

④取出段表项中该段的起始地址b，计算$E = b + W$，用得到的物理地址E去访问内存。

PS：这部分考题涉及的常见异常(真题 2016-28)

段缺失异常：根据表中状态栏，要访问的地址不在内存中。

越权异常：对只读，进行写操作，越权。

越界异常：段内地址超过段长。

软硬链接——真●例题

例：设文件 $F1$的当前引用计数值为 1，先建立$F1$的符号链接(软链接)文件 $F2$，再建立 $F1$ 的硬链接文件 $F3$，然后删除 $F1$。此时，$F2$ 和 $F3$ 的引用计数值分别是（）——2009，T31

解：当建立 $F2$ 时，$F1$ 和$F2$的引用计数值都为 1。当再建立 $F3$ 时，$F1$ 和 $F3$ 的引用计数值就都变成了 2。当后来删除 $F1$ 时，$F3$ 的引用计数值为 2-1=1，$F2$ 的引用计数值一直不变。

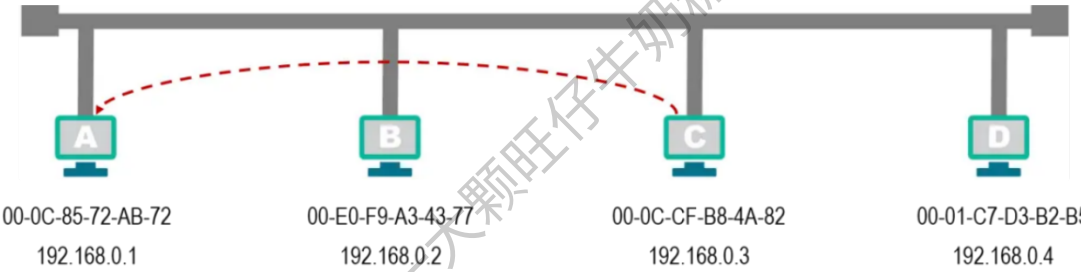
①建立符号链接(软链接)时，引用计数值直接复制；建立硬链接时，引用计数值加 1。软链接建立以后互不影响，硬链接建立以后相互影响。

②删除文件时，删除操作对于符号链接是不可见的，这并不影响文件系统，当以后再通过符号链接访问时，发现文件不存在，直接删除符号链接；

③对于硬链接则不可以直接删除，引用计数值减 1，若值不为 0，则不能删除此文件，因为还有其他硬链接指向此文件。

ARP 协议

在一个总线以太网上有四台主机，各主机的 MAC 地址和 IP 地址都标注在各自的下面，主机 C 要给主机 A 发送一个 IP 数据报（假设主机 C 知道主机 A 的 IP 地址，否则无法发送）。



主机 C 的网际层(IP 层)在封装 IP 数据报时，在 IP 数据报首部的源地址字段填写主机 C(自己)的 IP 地址 192.168.0.3，目的地址字段填写主机 A 的 IP 地址 192.168.0.1。IP 数据报会封装在以太网帧中发送，主机 C 的数据链路层（主要是 MAC 子层）在封装以太网帧时，在帧首部的源地址字段填写主机 C 的 MAC 地址 00-0C-CF-B8-4A-82，目的地址字段应填写主机 A 的 MAC 地址。但主机 C 只知道主机 A 的 IP 地址，而不知道主机 A 的 MAC 地址，这样就无法将 IP 数据报封装成以太网帧进行发送。

主机 C 的 ARP 进程在自己的 ARP 高速缓存表中查找主机 A 的 IP 地址的相关记录，假设主机 C 的 ARP 高速缓存表是空的，无法查找到主机 A 的 IP 地址的相关记录，于是主机 C 的 ARP 进程构建并发送一个 ARP 广播请求（封装成以太网帧时目的 MAC 地址为广播地址 FF-FF-FF-FF-FF-FF），该广播请求的内容是：“我的 IP 地址是 192.168.0.3，我的 MAC 地址是 00-0C-CF-B8-4A-82，我想知道 IP 地址为 192.168.0.1 的主机的 MAC 地址是什么”。

总线上的其他各主机都会收到该 ARP 广播请求，主机 B 和 D 的 ARP 进程解析该请求的内容后发现:“这不是在问我，不用理会”；主机 A 的 ARP 进程解析该请求的内容后发现：“这是在问我，需要做出响应”。

主机 A 的 ARP 进程一方面将主机 C 的 IP 地址和 MAC 地址作为记录存储到自己 ARP 高速缓存；另一方面构建并发送给主机 C 的 ARP 单播响应（封装成以太网帧时目的 MAC 地址为主机 C 的 MAC 地址 00-0C-CF-B8-4A-82），该单播响应内容是：“我是 IP 地址 192.168.0.1 的主机，我 MAC 地址为 00-0C-85-72-AB-72”。

主机 B 和 D 的网卡会丢弃收到的封装有 ARP 单播响应的以太网帧，这是因为该帧的目的 MAC 地址是主机 C 的 MAC 地址，与主机 B 和 D 的 MAC 地址都不匹配；主机 C 的网卡会接收该单播帧，主机 C 的 ARP 进程对单播帧中的 ARP 单播响应进行解析，这样主机 C 就知道了主机 A 的 IP 地址所对应的 MAC 地址。

主机 C 的 ARP 进程将所获取到的主机 A 的 IP 地址与 MAC 地址的映射关系写入自己的 ARP 高速缓存表，之后就可以将之前待发送给主机 A 的 IP 数据报封装成以太网帧进行发送了。