# 数据结构

## 《王道计算机》

### 树与二叉树

#### 树的基本概念

树中的结点数等于所有结点的度+1

度为m的树第i层最多有mi-1个结点

高度为h的m叉树至多有 个结点

有n个结点的m叉树的最小高度为

总分支数=1\*n1+2\*n2+…+m\*nm度为m的结点引出m条分支

总结点数=总分支数+1

#### 二叉树的概念

满二叉树：高度h，含2h-1个结点

标号i，若有双亲，则为

##### 完全二叉树性质

若，则i为分支结点，否则为叶子结点

若有度为1的点，仅一个且有左孩子

按层编号，一旦某i结点为叶子结点或仅有左孩子，则编号大于i的结点均为叶子结点

N为奇数，每个分支结点都有左右孩子，n为偶数，则编号最大的分支结点n/2只有左孩子  
即n为奇时，不含度为1的点

##### 二叉树性质

非空时，第k层最多有2k-1个结点

高度为h，最多有2h-1个结点

按层编号后：

* + i>1时，双亲，若i为偶，为左孩子，i奇则为右孩子
  + 时，结点i的左孩子为2i，否则无左孩
  + 时，结点i的右孩为2i+1，否则无
  + I所在层次（深度）为

具有n个结点（n>0）的完全二叉树高度为  
或

##### 习题知识点

设高度为h的二叉树上只有度为0,2则结点数至少为2h-1

设二叉树右2n个结点，则度为1的结点有奇数个  
N=2n=N1+2N2+1  
N1=2(N-N2)-1



高度为h的完全二叉树最少有2h-1个结点

第n层有叶结点的完全二叉树，高度为n或n+1层  
比如，第六层有8个叶结点，则最少有25-1+8个结点  
最多有：7层，(25-8)\*2+26-1个结点

完全二叉树,n个结点

* + n为奇数不含度为1的点，即N1=0

N0=N2+1=(n+1)/2

N2=(n-1)/2

n=2N0-1=2N2+1

* + n为偶数，含一个度为1的点，即N1=1

N0=n/2=N2+1

N2=n/2-1

n=2N0=2(N2+1)

* + N0已知，则nmax=2N0 此时N1=1  
    nmin=2N0-1 此时N1=0

n个结点的二叉树采用二叉链存储，则空指针数量为n+1  
空=2\*结点数-非空=n+1

高度为h的满m叉树

* + 第i层结点个数：mi-1
  + 编号为i的结点，若其双亲结点存在，则为
  + 结点i的第一个子女编号：j=(i-1)\*m+2
  + 结点i的第k个子女编号：(i-1)\*m+k+1
  + 结点i的第m个子女编号：i\*m+1
  + 编号i的结点有右兄弟的条件：
    - 当结点i不是双亲的第m个子女才有右兄弟，设其双亲结点编号为j:  
      j的第m个孩子  
      (j-1)\*m+m+1=j\*m+1=  
      所以，的时候才有右兄弟  
      或者(i-1)%m!=0（满足不为第m个孩子）

#### 二叉树的遍历和线索二叉树

可唯一确定二叉树

非线索二叉树中，如有n个结点，则有n+1个空指针

二叉树中序遍历的最后一个点一定是从根结点开始沿右子女指针链走到底，可能是叶子结点，也可能是分支结点

若a有左孩子b，右孩子c，则无论先中后序序列，b都在c前面

先序和后序序列正好相反的话，则该二叉树只有一个叶结点

中序：n在m前的条件：n在m左方，可以不同层

后序：n在m前的条件：n是m的子孙或者同层的左方

先序中序后序序列中，叶子结点的先后顺序相同

先序遍历序号要借助栈。先序和中序的关系相当于以先序序列为入栈顺序，中序序列为出栈顺序

N个结点的线索二叉树有n+1个线索（存疑）

一棵左子树为空的二叉树在先序线索话后，其中空的链域个数为2个

* + 根的左子树为空且无前驱结点
  + 先序的最后一个元素为叶子结点，无后继结点

线索二叉树是利用二叉树的n+1个空指针来存放结点的前驱和后继

并非每个结点都可以通过线索找到前驱和后继

后序线索树的遍历仍不能有效求解

先序序列确定，n个结点，则有个不同的二叉树

#### 树、森林

等价关系

|  |  |  |
| --- | --- | --- |
| 树 | 森林 | 二叉树 |
| 先根遍历 | 先序遍历 | 先序遍历 |
| 后根遍历 | 中序遍历 | 中序遍历 |

若树中的任两个叶子结点都不存在相同的双亲，则树中的叶子数才有可能与其对应的二叉树中的叶子数相等

设F是一个森林，B是F变来的二叉树，若F中有n个非终端结点，则B中右指针为空的结点个数有n+1个



将森林F转化为对应的二叉树T，F中叶结点的个数等于T左孩子指针为空的结点个数

#### 树与二叉树的应用

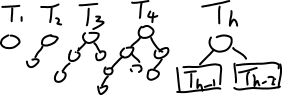
中序遍历二叉排序树，可得到有序数列

Huffman树，N个结点构造，过程中新建了N-1个结点（双分支结点），因此Huffman树中系欸但总数为2N-1

Huffman树中不含度为1的结点

具有n个结点的二叉排序树，最理想深度

平衡二叉树最少结点数递推公式：  
N0=0,N1=1,N2=2,Nh=1+ Nh-1+Nh-2  
h为二叉树高度，Nh为构造此高度的平衡二叉树所需最少结点数；所有非叶结点的平衡因子均为1



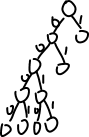
Huffman树权值最小的两个结点一定是兄弟结点

Huffman树中任一非叶结点的权值一定不小于下一层任一结点的权值

Huffman树不一定是一棵完全二叉树

没有一个编码是另一个编码的前缀，则称这样的编码为前缀编码

设Huffman编码长度不超过4，若已对两个字符编码为1和01，则还可以对（4）个字符编码  
长度4,则高度最高为5，已有0和01，说明第2，3层各有一个叶子，为使从第3层起编码更多字符，余下的二叉树应为满二叉树,4个叶子



度为m的Huffman树，只有度为0和m的结点，叶子结点有N0个，度为m的结点有Nm个，总数N，N=N0+Nm。

* + 有N个结点的Huffman树有N-1条分支
  + m\* Nm=N-1= Nm+ N0-1
  + (m-1)\* Nm= N0-1
  + Nm=( N0-1)/(m-1)

合并排序，长度分别为m,n,最坏情况下需比较m+n-1次

N个顶点，无向完全图N(N-1)/2条边，有向完全图N(N-1)

### 图

#### 图的基本概念

若n个顶点的图，小于n-1条边，则一定不是连通图

无向图：连通，连通图，连通分量，极大连通子图，极小连通子图

有向图：强连通图，强连通分量

无向图，n顶点e条边

有向图：

如果一个图有n个顶点，并且边数大于n-1，则一定有环

路径长度：路径上边的数目

简单路径：顶点不重复出现

简单回路：除第一个和最后一个顶点，其余顶点不重复出现

有向树：有一个顶点入度为0，其余顶点的入度均为1的有向图

路径：由顶点和相邻顶点序偶成的边形成的序列

稀疏图，应用邻接表而非邻接矩阵

无向连通图边数顶点数-1

无向连通图至少有一个顶点的度为1 错  
完全图，或n边n顶环图

强连通图，任何顶点到其他顶点都有路径，并不要求直接有弧

N个顶点，强连通图边至少为n个

在有n个顶点的有向图，每个顶点的度最大可达2n-2

具有n个顶点的图是一个环，则有n棵生成树

一个具有n个顶点,e条边的无向图是一个森林，则该森林必有n-e棵树

邻接矩阵：

* + 有向图：第i行表示第i个顶点的出度；第i列表示入度
  + 无向图：第i行/列表示第i个顶点的度

#### 图的存储及基本操作

邻接矩阵是图的顺序存储结构，邻接表是图的链接存储结构

##### 邻接矩阵

设G的邻接矩阵为A，An的元素An[i][j]等于由顶点i到顶点j的长度为n的路径数目

邻接矩阵空间复杂度为O(|v|2)

##### 邻接表

邻接表空间复杂度

* + 无向：O(|v|+2|e|)
  + 有向：O(|v|+|e|)

##### 习题知识点

若一个有向图的邻接矩阵，对角线以下元素为0，则一定存在拓扑排序

一个图的邻接矩阵表示唯一，邻接表表示不唯一

有向图的邻接表存储结构中，顶点v在边表中出现的次数时顶点v的入度

N个顶点的无向图邻接表最多由n(n-1)个边表结点

假设由n个顶点e条边，有向图用邻接表表示，则删除与某个顶点相关的所有边的时间复杂度为O(n+e)

求有向图结点的度，需要遍历整个邻接表

#### 图的遍历

##### BFS：

空间：O(|v|)，

时间：邻接表：O(|v|+|e|)；邻接矩阵O(|v|2)

##### DFS:

空间：O(|v|)

时间：邻接表：O(|v|+|e|)；邻接矩阵O(|v|2)

对于同样一个图，基于邻接矩阵的遍历所得到的DFS序列和BFS序列时唯一的，基于邻接表的DFS/BFS序列不唯一

对于无向图，调用BFS/DFS次数为该图连通分量的个数

##### 习题知识点

各边权值相等时，BFS可解决单源最短路径；权值不等时，BFS解决不了

利用深度优先遍历可以判断G中是否存在回路：

* + 对于无向图：DFS中遇到回边必存在环
  + 对于有向图，这条回边可能是指向DFS森林另一棵生成树上顶点的弧；但如果从有向图的某个顶点v出发进行深度优先遍历，若在DFS(V)结束前出现一条从顶点u到v的子孙，则有向图必存在包含顶点v和u的环

BFS生成树的高度DFS树的高度

一个无向图G是一棵树的条件：G须是无回路的连通图或者有n-1条边的连通图

当图G中各边权值不同时，G的最小生成树就是唯一的

MST（最小生成树）权值唯一

MST的边数=顶点数-1

Prim:

* + 时间：O(|v|2)，不依赖|e|，适用于边稠密的图

Kruskal

* + 时间：O(|e|log|e|)，使用边稀疏而点多的图

最短路径：

* + 单源：Dijkstra，时间O(|v|2)
  + 互相：Floyd-Warshall，时间O(|v|3)

Floyd:

* + 递推产生一个n阶方阵A(-1), A(0),…A(k),…A(n-1),其中A(k)[i][j]表示从顶点Vi到顶点Vj的路径长度，k表示绕行第k个顶点的运算步骤  
    A(-1)[i][j]=arcs[i][j]  
    A(k)[i][j]=Min{ A(k-1)[i][j], A(k-1)[i][k]+ A(k-1)[k][j]} k=0,1,2…n-1
  + A(0)[i][j]是从顶点Vi到顶点Vj，中间顶点是V0的最短路径长度
  + A(k)[i][j]是从Vi到Vj，中间顶点的序号不大于k的最短路径的长度

Flody算法允许图中带负权值的边，但不允许有包含负权值的边组成的环路

Floyd同样适用于带权无向图

DAG:有向无环图

拓扑排序时间复杂度O(|v|+|e|)

对于一般图，如果他的邻接矩阵是三角矩阵，则存在拓扑排序，反之不一定对

若有向图具有有序的拓扑排序序列，则邻接矩阵为三角

时间Vk的最早发生时间Ve(k)指从开始顶点V到Vk的最长路径长度

* + Ve(源点)=0
  + Ve(k)=Max{Ve(j)+weight(Vj,Vk)}
  + 计算Ve(k)时，从前往后

时间Vk的最迟发生时间Vl(k)：在不推迟整个工程下，即保证它所指的事件Vi在Ve(i)时刻能够发生，该事件必须发生

* + Vl(汇点)=Ve(汇点)
  + Vl(j)=Min{Vl(k)-weight(vj,Vk)}
  + 计算Vl(k)时，从后往前计算

活动ai的最早开始时间e(i)：指该活动的起点所表示的时间最早发生时间，如果边<Vk,Vj>表示活动ai,则e(i)=Ve(k)

活动a­i最迟开始时间l(i)：指该活动的终点所表示的事件最迟发生事件与该活动所需事件之差。如果<Vk,Vj>表示活动ai，则l(i)=Vl(j)-weight(Vk,Vj)

一个活动ai,l(i)与e(i)的差额d(i)=l(i)-e(i)，称l(i)-e(i)=0的活动ai是关键活动

所有权值最小的边一定会出现在MST中 x

Prim从不同顶点开始得到的MST不一定相同

最短路径一定是简单路径

Dijkstra不适合求带负权值的最短路径

Floyd求亮点最短路径时，当最短路径发生改变，Pathk-1就不是Pathk的子集

判断有向图是否有环

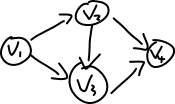
* + DFS
  + 拓扑排序
  + 求关键路径
  + 求最短路径不行

拓扑排序，时间复杂度：邻接表O(n+e)，邻接矩阵O(n2)

顶点数目>1的强连通分量中必然存在回路

拓扑有序序列唯一，则图中每个顶点的入度和出度最多为1 ×

如果有向无环图拓扑排序唯一则可以唯一确定该图 ×



求关键路径以拓扑排序为基础

### 查找

#### 顺序查找和折半查找

##### 顺序查找

一般顺序表查找

* + ASL成功=
  + ASL不成功=n+1

有序顺序表查找

* + ASL不成功=

若有序序列有n个元素，则对应折半查找判定树有n个圆形分支结点和n+1个方形的叶结点

* + 方形：查找不成功
  + 圆形：一个记录

##### 折半查找

ASL成功=

时间：O(log2n)

存储要求时顺序存储，有序，链式不行

##### 分块查找：

块内无序，块间有序

第k个块的最大关键字小于第k+1个块的所有记录的关键字

索引表：表中每个元素含有各块的最大元素和各块第一个元素的地址，表有序

将长度n的查找表均分为b块，每块s个记录，块内块间都用顺序查找，则

若，ASL取最小值

若对索引表采用折半查找则

##### 习题知识点

顺序查找：成功时有序无序ASL一样，失败时ASL有序<ASL无序

折半查找对应的判定树时平衡二叉树AVL

折半查找失败时，比较次数最多为树的高度  
或者  
判定树中方形结点时虚构，不计入比较次数

折半ASL和最大查找长度：O()  
二叉排序树最大O(n)，最短O()

折半，高度最低  
因为是AVL，分支高度差。最高，所以最低

K分查找法：高度

#### B树和B+树

##### B树

一棵m阶B树，或为空，或为满足如下特性的m叉树：

* + 每个结点至多有m棵子树，即每个结点最多含m-1个关键字
  + 若根结点不是终端结点，则至少有两棵子树
  + 除根结点外的所有非叶结点至少有棵子树，即至少有个关键字
  + 非叶结点结构

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | P0 | K1 | P1 | K2 | P2 | … | Kn | Pn |

* + - Ki为关键字，K1<K2<…<Kn
    - Pi为指向字根结点的指针且Pi-1所指的子树中所有的结点均小于Ki
    - Pi所指的子树中所有的结点均大于Ki
    - n()为节点中关键字的个数
  + 所有叶结点都在同一层，无任何信息（像折半查找中失败的点，实际上叶子结点不存在，指向这些结点的指针为空）
  + 所有结点平衡因子=0

B树的高度不包括最后一层叶子结点

B树n个关键字，高度h，m阶

##### B+树

一棵m阶B+树满足如下特性：

* + 每个分支结点最多有m棵子树
  + 非叶根结点至少有2棵子树，其他分支结点至少有棵子树
  + 结点的子树个数与关键字个数相等
  + 所有叶结点包含全部关键字及指向相应记录的指针，叶结点中关键字按大小排列，相邻叶结点相互链接
  + 所有分支结点（可看成是索引的索引）中仅包含它的各个子节点（即下一级的索引块）中关键字的最大值及指针

m阶B树和B+树的区别

* + B+中，具有n个关键字的结点只含有n棵子树  
    B中，n个关键字，有n+1个子树
  + B+:每个非根结点关键字个数：  
     根结点：  
    B: 根：
  + B+:叶结点包含信息，非叶结点仅起索引作用。非叶结点的每个索引项：对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址
  + B+：叶结点包含了全部关键字，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

B+两种查找方式

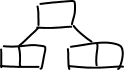
* + 从最小关键字开始的顺序查找
  + 从根结点开始，多路查找

具有n个关键字的m阶B树，应有n+1个叶结点

##### 习题知识点

m叉树，高h，结点数=

5阶B树，并非存在至少有一个有4个关键字的结点，各个结点只有两个关键字也可以



有15个关键字的4阶B树，含关键字的结点个数最多为15个，此时每个结点含关键字应最少才可以:

含有n个非叶结点的m阶B树至少有个关键字

B,B+树都能有效支持随机查找

B,B+都是平衡的多叉树

B,B+都可以用于文件的索引结构

#### 散列表

##### 基本散列函数

直接定址法：H(key)=a\*key+b；不会产生冲突，适合关键字基本连续分布的情况

除留余数法：p为不大于散列表长度m，但最接近或等于的质数：H(key)=key%p

##### 处理冲突的方法

开放定址法：Hi=(H(key)+di)%m；m表示散列表长，d表示增量序列

* + 线性探测法：di=1,2,3..m-1;会“聚集”(“堆积”)
  + 平方探测法：di=12,-12, 22,- 22…k2,-k2,其中km/2，m必须使一个可以表示成4k+3的质数：可以避免“堆积”，但不能探测到散列表上所有的元素
  + 再散列法：di=Hash2(Key)；最多经过m-1次探测就会遍历表中所有位置，回到H0，不易产生聚集
  + 伪随机序列法：di=伪随机数序列。
  + 注意：开放定制法，不能随便删除表中已有元素。可以做删除标记，进行逻辑删除

拉链法：把同义词存储在一个线性链表中

散列表查找效率取决于

装填因子:，定义为一个表的装满程度  
散列表的平均查找长度依赖于，不直接依赖于n或m

##### 题目中知识点

折半查找法只能在顺序存储结构上

“堆积”问题是由于同义词之间或非同义词之间发生冲突引起的，同时解决冲突的方法选择不当

同义词冲突不等于聚集

链地址法（拉链法）不会引起聚集现象

产生冲突的概率与装填因子的大小成正比

堆积现象，对存储效率，散列函数和装填因子都不会有影响，平均查找长度会因为堆积现象而增大

在计算查找失败时的平均查找长度时，既不是根据表中元素个数，也不是根据表长来决定的，而是根据散列函数（Mod后面的数字，如Mod7）来计算平均查找长度。王道p258第5题

#### 字符串模式匹配KMP

next求值的算法

1. void get\_next (char T[ ], int next[ ]){
2. int i, j = 0;
3. next[1] = 0;
4. while(i <= T[0]){ //T[0]用于保存字符串的长度
5. if( j == 0 || T[i] == T[j]) {
6. ++i; ++j; next[i] = j;
7. }else
8. j = next[j];
9. }
10. }

KMP算法

1. int KMP ( char S[ ] , char T[ ] , int next[ ] , int pos){
2. //利用模式串T的next函数求T在主串S中
3. //第pos个字符之后的位置的KMP算法
4. //T非空，1<=pos<=strlen(S)
5. i=pos;
6. j=1;
7. while(i<=S[0]&&j<=T[0]){
8. If(j==0||S[i]==T[j]){
9. ++i; ++j;
10. }else
11. j=next[j]
12. }
13. if(j>T[0]) return i-T[0];
14. else return 0;
15. }

### 排序

#### 排序

拓扑排序不属于内部排序

不是所有的内部排序都要经过比较操作，基数排序就不是

算法的稳定性与算法优劣无关

链表不适用于折半排序

对同一线性表使用不同的排序方法，得到的排序结果可能不同

对任意序列进行基于比较的排序，求最少的比较次数，应考虑在最坏的情况下，对任意n个关键字排序的比较次数至少为，比如7个关键字，至少比较13次

#### 插入排序

##### 直接插入排序

比较和移动次数取决于待排序表的初始状态

适用于顺序存储和链式存储

边比较边移动元素

##### 折半插入排序

将比较和移动元素分离出来，即先折半查找元素的待插入的位置，然后再统一移动

折半插入排序仅仅减少了比较元素的次数，该比较次数与待排序表的初始状态无关，仅取决于表中的元素个数n

元素的移动次数没有改变，依赖于待排序表的初始状态

##### 希尔排序

先将待排序表分割成若干形如L[i,i+d,i+2d,…,i+kd]的特殊子表，分别进行直接插入排序，当整个表都已经基本有序时，再对全体记录进行一次直接插入排序

只适用于当线性表为顺序存储的情况

希尔排序，总的比较次数和移动次数比直接插入排序小得多

##### 习题知识点

不考虑与哨兵比较，直接插入排序最坏的情况下，要做n(n-1)/2次比较

在待排序序列基本有序的前提下，直接插入排序效率最高

堆排序，冒泡排序，快排，简单选择排序每一回合都会有元素放在最终位置，希尔排序，直接插入排序没有这个属性

直接插入排序有局部有序属性，即如果从后向前插，前几个元素是有序排列的

#### 交换排序

##### 冒泡排序

最好时，只需要比较n-1次，移动0次；

最差时，初始状态为逆序，需要进行n-1趟排序，第i趟需要进行n-i次比较，每次比较都必须移动元素三次来交换元素位置

冒泡排序也会局部有序，并且是全局有序（比直接插入厉害点）

##### 快速排序

需要借助递归工作栈，容量应与递归调用的最大深度一致。最好：，最坏：要进行n-1次调用即O(n)

时间效率和划分是否对称有关，如果划分两个区域分别包含n-1和0个元素，就最差；能均分就最好

快排不产生有序子序列，即没有局部有序性

##### 习题知识点

插入排序，第i趟之后前i+1个元素应该时有序的

快速排序在要排序的数据已基本有序的情况下最不利

冒泡和选择排序经过i趟，将有i个元素处于最终位置（最左或最右）

可能时执行第一趟快排之后所得到的序列的判断：如果存在一个元素，其左/其右全都大于/小于自身，即可

快排递归次数与每次划分后得到的分区的处理顺序无关

#### 选择排序

##### 简单选择排序

简单选择排序中元素移动的操作次数很少，不会超过3(n-1)，最好的情况下移动0次，此时对应表已经有序；但是元素之间的比较的次数与序列初始状态无关，始终是n(n-1)/2次

##### 堆排序

构造初始堆，就是一个反复筛选的过程。N个结点的二叉树，最后一个结点是第个结点的孩子。对第个结点为根的子树进行筛选（大根堆，若根结点的关键字小于左右子女中的较大者，则交换），之后依次对各结点（-1~1）为根的子树进行筛选。交换的时候可能破坏下一级的堆，需要采用上述方法对下一级构造。

向下调整的时间与树高有关，为O(h)。在元素个数为n的序列上建堆，其时间复杂度为O(n)，即在线性时间内，就可以将一个无序数组建成一个堆

堆排序的时候，堆顶元素就是最大值（大顶堆），输出堆顶元素后，将堆底元素送入堆顶，再向下调整，再输出堆顶元素

删除堆顶操作，将堆底元素放到堆顶，然后向下调整

插入操作，先插到堆的末尾，再向上调整

时间效率：建堆时间为O(n)，之后有n-1次向下调整操作，每次调整的时间复杂度是O(h)，故时间复杂度为O(nlogn)

##### 习题知识点

小根堆，关键字最大的结点一定存储在这个堆所对应的完全二叉树的叶子结点中，二叉树中最后一个非叶子结点，所以关键字最大的结点存储范围+1~n

插入排序，快排，冒泡排序，排序过程中的比较次数与序列初始状态有关，选择排序比较次数始终是n(n-1)/2

插入排序，快速排序和归并排序，只有将所有元素排序完成后，才能得到前k小的元素序列。

冒泡排序，堆排序和简单选择排序，每一趟都会确定一个最小/大的元素。

堆排序，n个元素，建立初始堆，时间不超过4n，取得第k小的元素之前的排序序列所化的时间为klogn，总时间为4n+klogn，冒泡和简单选择排序完成此功能所花的时间为kn，当k5时，堆排序最优

#### 归并排序和基数排序

##### 归并排序

整个归并排序进行趟

每一趟时间复杂度为O(n)

对于n个元素进行k-路归并排序，排序的趟数m满足从而又考虑到m为整数，所以

##### 基数排序

最高位优先MSD，最低位优先LSD

假设线性表由结点序列a0­,a1,…an-1组成，每个结点aj的关键字由d元组（）组成，其中

一趟排序需要辅助空间为r（r个队列），所以空间复杂度O(r)

基数排序与序列的初始状态无关

基数排序需要进行d趟分配和收集，一趟分配需要O(n),一趟收集O(r),所以基数排序时间复杂度O(d(n+r))

##### 习题知识点

基数排序不需要进行关键字的比较

归并排序的比较次数的数量级与序列的初始状态无关

选择排序的比较次数与序列初始状态无关

将两个各含有n个元素的有序表合并成一个有序表，最少比较n次，最多比较2n-1

#### 各种内部排序算法的比较

##### 时间复杂度

简单选择排序，直接插入排序和冒泡排序平均复杂度O(n2)，直接插入排序和冒泡排序最好的时候可以达到O(n)，且简单选择排序与序列的初始状态无关。

快速排序基于分治的思想，最坏会达到O(n2)但实际应用中常常优于其他算法。

归并同样基于分治思想，其分割子序列与初始序列的排列无关

| 算法种类 | 时间复杂度 | | | 空间复杂度 | 稳定？ |
| --- | --- | --- | --- | --- | --- |
| 最好情况 | 平均情况 | 最坏情况 |
| 直接插入排序 | O(n) | O(n2) | O(n2) | O(1) | √ |
| 冒泡排序 | O(n) | O(n2) | O(n2) | O(1) | √ |
| 简单选择排序 | O(n2) | O(n2) | O(n2) | O(1) |  |
| 希尔排序 |  |  |  | O(1) |  |
| 快速排序 | O(nlog2n) | O(nlog2n) | O(n2) | O(log2n) |  |
| 堆排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) |  |
| 2-路归并排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(n) | √ |
| 基数排序 | O(d(n+r)) | O(d(n+r)) | O(d(n+r)) | O(r) | √ |

##### 排序算法小结

若n较小（n），则可以采用直接插入排序或简单选择排序。由于直接插入排序所需的记录移动操作比简单选择排序多，因此，当记录本身信息量较大时，用简单选择排序比较好

若文件的初始状态已经基本有序，则选用直接插入排序或者冒泡排序

若n比较大，则应采用时间复杂度为O(nlog2n)的算法：快排，堆排或归并排序。

* + 快排是基于比较的内部算法中最好的，当待排序的关键字是随机分布时，快排的平均时间更短。
  + 堆排所需要的辅助空间小于快排，并且不会出现快排可能出现的最坏的情况
  + 归并算法是稳定的，快排和堆排都不稳定。通常将它和直接插入排序结合起来一块用：先利用直接插入排序求得较长的有序子文件，然后再两两归并，因为两者都稳定，所以这个方法也稳定

任何基于“比较”的排序算法，至少需要O(nlog2n)

若n很大，记录的关键字位数较少且可以分解时，采用基数排序比较好

当记录本身信息量较大时，为避免耗费大量时间移动记录，可用链表作为存储结构

##### 习题知识点

交换类的排序，其趟数和原始序列有关，所以冒泡排序与初始序列有关；直接插排，趟数固定n-1，简单选择，趟数固定n-1；基数，趟数固定d

简单选择排序，快排，堆排每一趟都至少可以确定一个元素最终位置

堆是用于排序的，所以对其中的元素进行查找的时候，是无序的，效率不高

基数排序元素的移动次数与关键字的初始排列次数无关

#### 外部排序

##### 外部排序的方法

外部排序通常采用归并排序的方法，包括两个相对独立的阶段

* + 首先，根据内存缓冲区大小，将外存上含n个记录的文件分成若干个长度为h的文件，依次读入内存并利用有效的内部排序方法对他们进行排序，并将排序后得到的有序子文件重新写回外存，称这些有序子文件为 归并段 或 顺串
  + 然后，对这些归并段进行逐趟归并，使归并段（有序的子文件）逐渐由小到大，直至得到整个有序文件

外部排序的时间=内部排序所需的时+外存信息读写的时间+内部归并所需的时间  
TES=r\*TIS+d\*TIO+S\*(n-1)\*Tmg  
其中r是初始归并段个数，TIS是对每一个初始归并断进行内部排序的时间，d是访问外存块的次数，TIO是每一个块的存取时间，S是归并趟数，n是每趟参加二路归并的记录个数，Tmg是每作一次内部归并，取得一个关键字最小记录的时间。

要提高外排序的速度，应着力减少d，即I/O次数

增大归并路数，可减少归并趟数，从而减少总的磁盘I/O次数

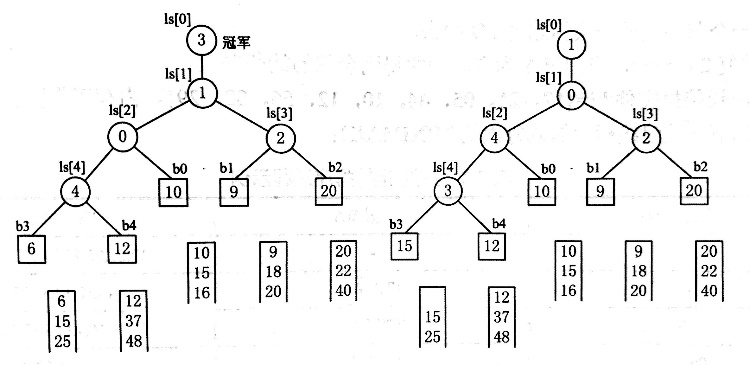
一般地，对r各初始归并段，作m路平衡归并，归并树可用严格m叉树（即只有度为m和度为0的结点的m叉树）

* + 第一趟可将r个初始归并段归并为个归并段
  + 以后每趟归并将l个归并段归并成个归并段
  + 直到最后形成一个大的归并段
  + 树的高度=归并的趟数S=
  + 只要增大归并路数，或减少初始归并段个数r，都能减少归并趟数S，以减少读写磁盘数d，达到提高外排速度的目的

##### 多路平衡归并与败者树

归并趟数S=，从而增加归并路数可以减少归并趟数S，进而减少访问外存的次数。但这样会增加内部排序的时间。

* + 内部排序，m个元素选择关键字最小的元素需要比较m-1次。
  + 每趟归并n个元素需要作(n-1)\*(m-1)次比较
  + S趟归并总共需要比较次数为
  + 其中的在初始归并段个数r与记录个数n一定时是常数。但随着m的增长而增长，则内部归时间也随着m增长。这将抵消增大m而减少外存访问次数所得到的效益，因此，不能使用普通的内部归并排序算法

为了使内部排序不受m增大的影响，引入败者树

* + 因为m归并的败者树深度为，因此m个记录中选择最小关键字，最多需要次比较，所以总比较次数为：  
    可见，使用败者树后，内部归并的比较次数与m无关了。
  + 因此，只要内存空间允许，增大归并路数m将有效减少归并树的高度，从而减少I/O次数d，提高效率

归并路数m并不是越大越好，归并路数m增加，相应地需要增加输入缓冲区个数，如果可供使用的内存空间不变，势必要减少每个输入缓冲区的容量，使得内外存交换数据的次数增大。当m过大时，虽然归并趟数会减少，但读写外存的次数仍然会增加

##### 置换-选择排序（生成初始归并段）

减少初始归并段个数r也可以减少归并趟数S。

设初始待排文件FI，初始归并段文件为FO，内存工作区为WA，内存工作区可容纳m个记录

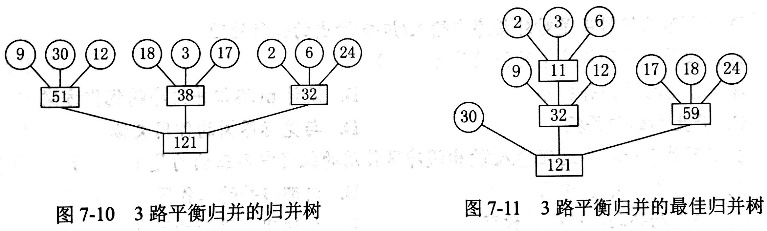
1. 从待排文件FI输入w个记录到工作区WA
2. 从内存工作区WA中选出其中关键字取最小的记录，即为MINIMAX（以后再选出关键字比它大的记录归入本归并段，比他小的归入下一归并段）
3. 将MINIMAX记录输出到FO中取
4. 若FI未读完，则从FI输入下一个记录到WA
5. 从WA中所有关键字比MINIMAX记录的关键字大的记录中选出最小的关键字记录，作为新的MINIMAX
6. 重复3-5，直到WA中选不出新的MINIMAX记录为止，由此得到一个初始归并段，输出一个归并段d额结束标志到FO中去
7. 重复2-6，直到WA为空。由此得到全部初始归并段

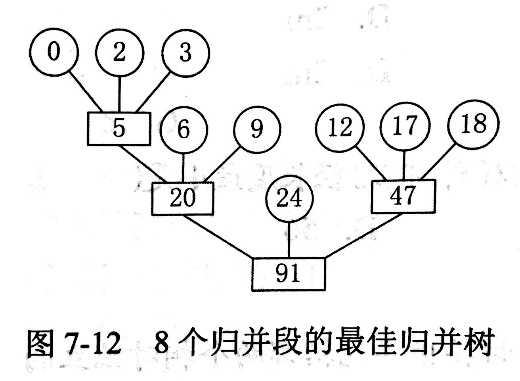
以上算法中，选择MINIMAX记录的过程利用败者树实现

##### 最佳归并树

图示：

* + 叶结点表示参加归并的一个初始归并段
  + 叶结点上的权值表示该初始归并段中的记录数
  + 根结点表示最终生成的归并段
  + 叶结点到根结点的路径长度表示在归并过程中的归并趟数
  + 各非叶结点代表归并成新的归并段
  + 归并树的带权路径长度WPL即为归并过程中的总读记录数。
  + 总的I/O次数=2Xwpl

初始归并段为满m叉树时，利用Huffman树的思想创造最佳归并树。如三叉最佳归并树

如果初始归并段不足构成一棵严格m叉树时，需添加长度为0的虚段  


* + 判定添加虚段的数目：
  + 设度为0的结点有n0(=n)个，度为m的结点有nm个，则对严格m叉树有n0=(m-1)nm+1，由此可以得nm=(n0-1)/(m-1)
  + 如果(n0-1)%(m-1)=0，则说明这n0个结点（初始归并段）正好可以构造m叉归并树。此时，内节点有nm个
  + 如果(n0-1)%(m-1)=u0，则说明对于这n0个叶结点，其中有u个多余，再加上m-u-1个空归并段，就可以建立归并树

##### 习题知识点

置换-选择排序时外排中生成初始归并段的算法

用置换-选择排序得到的初始归并段的长度不是等长的，其长度平均是传统等长初始归并段的2倍，从而使初始归并段数减少到原来的近二分之一

最佳归并树的作用在外排中的作用是设计m路归并排序的优化方案，而不是完成m路归并排序

外排中输入/输出缓冲区作用：

* + 暂存输入/输出记录
  + 内部归并的工作区
  + 产生初始归并段的工作区
  + 就是排序的内存工作区

在作m路平衡排序的过程中，为实现输入/内部排序/输出的并行处理，需要设置2m个输入缓冲区和2个输出缓冲区，以便执行内部排序时，能同时输入/输出操作。若仅设置m个输入缓冲区，则仅能进行串行操作

# 计算机原理

## 《王道计算机组成原理》

### 重要的名词解释

时钟周期、节拍、T周期：计算机操作最小时间单位，主频的倒数

工作脉冲：控制器的最小时间单位，起定时触发作用，一个时钟周期有一个工作脉冲

指令周期：由多个CPU周期组成

CPU周期，机器周期：包含若干时钟周期

存取周期：存储器进行两次独立的存储器操作所需最小时间间隔

CPU时钟周期，节拍脉冲，T周期，时钟周期

时钟周期总线时钟周期

机器周期由存取周期决定

总线时钟周期

总线周期

总线时钟周期

总线宽度：总线位宽，总线上能同时传输的数据位数，通常为总线根数

总线带宽：字节/秒

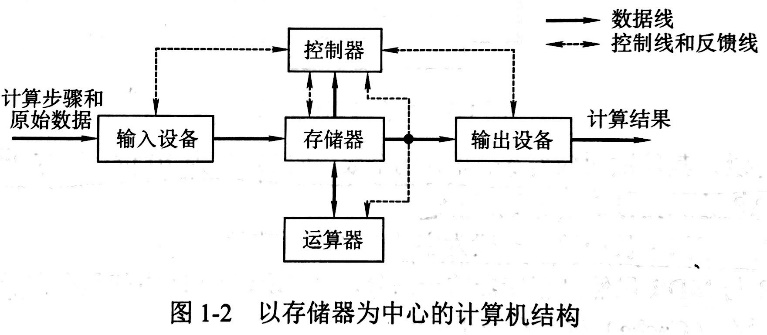
猝发传输：在一个总线周期内传输存储地址连续的多个数据字的总线传输方式

### 计算机系统概述

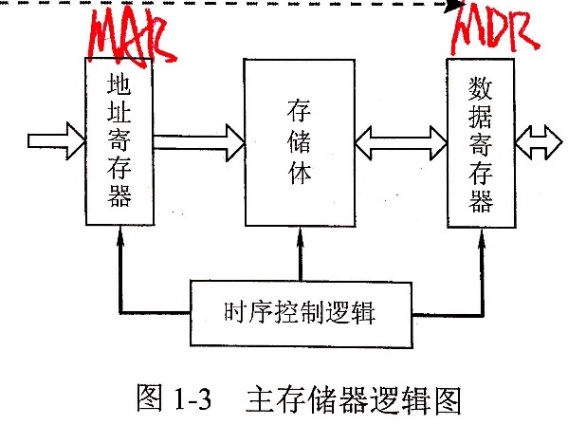
#### 计算机发展历程

机器语言是计算机唯一可以直接执行的语言

##### 计算机层次结构

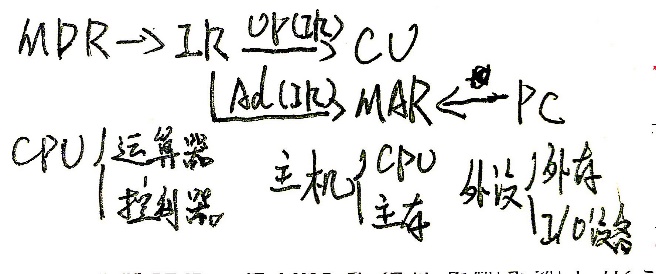
早期计算机以运算器为中心，现代计算机以存储器为中心，使I/O操作尽量绕过CPU

存储器

* + 主存储器：简称主存，也成内存储器，CPU可直接访问
  + 辅助存储器：辅存，也成外存储器，信息须调入内存然后才能被CPU访问
  + **存储单元**可存储**存储字**，一个存储单元有一个存储字，存储字的位数称为**存储字长**，存储字长一般是字节(8bit)或者字节的偶数倍
  + 主存按存储单元的地址进行存取，即**按地址存取方式**
  + **相联存储器**使按内容访问的
  + 主存基本组成
  1. 存储体
  2. **MAR**（地址寄存器）：存放访问地址，经过地址译码后找打所选的存储单元。用于**寻址**，其**位数**对应着**存储单元的个数**，如MAR为10位，则有210=1024个存储单元。**MAR长度**与**PC的长度**相等
  3. **MDR**（数据寄存器）：使主存和其他部件的中介机构，用于暂存要从存储器中读或者写的信息。**MDR的位数**和**存储字长**相等，一般为字节的二次幂整倍
  4. 时序控制逻辑用于产生存储器操作所需的各种时序信号
  + MAR，MDR虽然使存储器一部分，但在现代却是存在于CPU中的，Cacah(高速缓存)也是
  + 运算器核心使ALU（算数逻辑单元）
    - 包含若干通用寄存器
      * 累加器（ACC）（必有）
      * 乘商寄存器（MQ）（必有）
      * 操作数寄存器（X）（必有）
      * 变址寄存器（IX）
      * 基址寄存器（BR）
    - 程序状态寄存器（PSW），保留各类运算指令或测试指令的结果的各类状态信息，以表征系统运行状态

控制器

* + 组成
    - 程序计数器（PC）：存放当前**欲**执行指令的地址，可以自动+1形成下一条指令的地址，与主存的**MAR**有**直接通路**
    - 指令寄存器（IR）：存放当前指令，内容来自主存的**MDR**。指令中的操作码OP(IR)送至CU，用以分析指令并发出各种**微操作命令**序列；而地址码Ad(IR)送到MAR来取操作数
    - 控制单元（CU）：分析指令并发出各种**微操作命令**序列

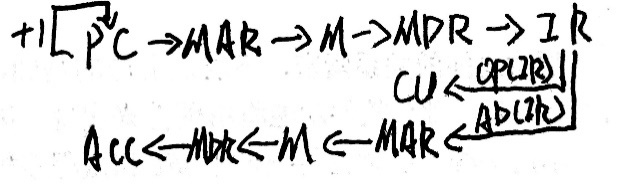
运算器和控制器集成到CPU上，CPU和主存构成主机

##### 软件分类

* + 系统软件：操作系统（os）,数据库管理系统（DBMS），语言处理程序，分布式软件系统，网络软件系统，标准库程序，服务型程序，编译程序
  + 数据库系统（DBS）包含DBMS，但不是系统软件

##### 工作过程

取数指令：将指令地址码知识的存储单元中的操作数取出后送至运算器的ACC中。信息流

* + 取指令：PCMARMMDRIR
  + 分析指令：OP(IR)CU
  + 执行指令：Ad(ID)MARMMDRACC
  + 然后(PC)+1PC
  + 

注意

* + PCMAR
  + (PC)MAR
  + (PC)+1PC

##### 习题知识点

冯诺依曼机的最根本特征是采用”存储程序”原理,基本工作方式:控制流驱动方式;基本特点是按地址访问并顺序执行指令

指令和数据都存放在存储器中,CPU可以根据指令周期的不同阶段来区分:取指阶段:指令；执行阶段：数据。

CPU只有在确定取出的是指令后才会将操作码送去译码

IR,PC,MAR,MDR

* + IR:存放当前欲执行的指令
  + PC:存放下一条指令的地址
  + MAR存放欲访问的存储单元地址
  + MDR存放从存储单元取来的数据

地址译码器在主存中

地址寄存器（MAR）虽然一般属于主存，但是现代计算机中绝大多数CPU内集成了地址寄存器，但不在运算器和控制器中

MAR位数：地址码长度；MDR位数：存储字长

运算器中有：

* + ACC,MQ,X,IX,BR,PSW,ALU,数据总线

速度：寄存器>Cache>内存

编译程序和解释程序的作用都是将高级语言程序转化成机器语言程序

编译程序编译时间长，运行速度快

解释程序方法简单，运行速度慢

解释程序将源程序翻译成机器语言，翻译一条，执行一条

寄存器对汇编语言不透明，可直接操作

软件和硬件在逻辑功能上是等效的，不是等价的

相联存储器既可以按地址寻址也可以按内容寻址

#### 计算机的性能指标

##### 主要性能指标

机器字长

* + 是指计算机进行一次整数运算（定点整数运算）所能处理的二进制数据的位数。与CPU的**寄存器位数**，**加法器**有关。一般等于内部寄存器的大小，字长越长，范围越大，精度越高。通常位字节（Byte,8位）的整数倍。不同计算机，字长可不同

数据通路带宽

* + 数据总线一次能够并行传送信息的位数。这里所说的数据通路带宽是指**外部数据总线**的宽度，它与**CPU内部的数据总线**宽度（**内部寄存器**的大小）有可能不同

主存容量

* + 主存储器的容量，字节或者用字数x字长（512Kx16位）表示。**MAR**的位数反映了存储单元的个数，MAR的位数反映了可寻址范围的最大值（而不一定是实际存储器的存储容量）
  + 例如，MAR16位，表示此存储体内有216=65536个存储单元（可称作64K内存，1K=1024），若MDR为32位，表示存储容量位64Kx32位

运算速度

* + 主频和CPU时钟周期
    - CPU时钟周期：通常为节拍脉冲或T周期，主频倒数，是CPU中最小的时间单位，每个动作至少需要一个时钟周期。MHz
    - 主频（CPU时钟频率）：机器内部主时钟的频率
  + CPI：执行一条指令所需时钟周期数
  + CPU执行时间时间，指运行一条指令所花费的时间  
    CPU执行时间=CPU时钟周期数/主频=（指令条数xCPI）/主频  
    上式表明，CPU性能取决于
    - 主频（时钟频率）
    - CPI
    - 指令条数
  + MIPS=指令条数/（执行时间\*106）=主频/CPI  
    每秒执行多少百万条指令
  + MFLOPS=浮点数操作次数/（执行时间\*106）  
    每秒执行多少百万次浮点运算
  + GFLOPS=浮点数操作次数/（执行时间\*109）  
    每秒执行多少十亿次浮点运算
  + TFLOPS=浮点数操作次数/（执行时间\*1012）  
    每秒执行多少万亿次浮点运算

##### 习题知识点

能缩短程序执行时间：

* + 提高CPU时钟频率
  + 优化数据通路结构
  + 对程序进行编译优化

数据通路的功能是实现CPU内部的运算器和寄存器以及寄存器之间的数据交换

寄存器由触发器构成

计算机的机器字长是指数据运算的基本单位长度

机器字长，指令字长和存储字长，三者在数值上可以相等也可以不等

数据字长是数据总线一次能够并行传送信息的位数，他可以不等于MDR的位数

计算机的位数，即机器字长，也就是计算机一次能够处理的二进制数的长度。如32位微机

操作系统的位数是操作系统可寻址的位数，与机器字长是不一样的。

一般可以通过寄存器判断机器字长

CPU的寄存器

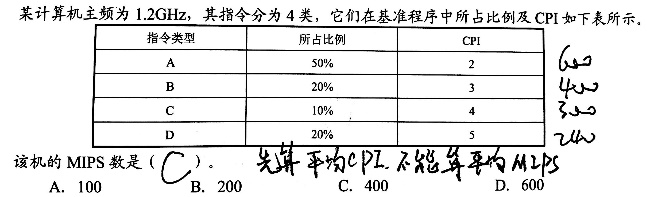
* + 对用户透明：IR,MAR,MDR
  + 不透明：PC，PSW，通用寄存器

**时钟周期**即CPU频率的倒数，是最基本的时间单位。**CPU周期**又称**机器周期**，由多个时钟周期组成

CPI影响因子

* + 系统结构
  + 指令集
  + 计算机组织

决定计算机计算精度的主要技术是计算机的字长

在计算多个指令的综合MIPS的时候，应先算平均CPI，不能分别求出MIPS再求均值

* + CPI=2\*0.5+3\*0.2+4\*0.1+5\*0.2=3  
    主频1.2GHz=1200MHz  
    故MIPS=1200/3=400

CPU提速50%并不是总时间减少50%，而是T=T0/1.5

如果主存储器容量位64K\*32位，且指令字长，机器字长，存储字长三者相等则

* + 216=64K，地址总线宽度为16位
  + 32位表示数据总线宽度
  + MAR=PC=16位
  + MDR=32位
  + 因为三者相等，所以IR,ACC,MQ,X均为32位

#### 易混清单

主频高的CPU不一定快

机器语言和汇编语言与机器指令对应，而高级语言不与指令直接对应

**机器字长**：计算机能直接处理的二进制数据的位数，机器字长一般等于内部寄存器的大小，决定了计算机的运算精度  
等于计算机位数

**操作系统位数**是操作系统可寻址位数，与机器字长不同，与MDR长度相等

**指令字长**：一个指令字中包含二进制代码的位数

**存储字长**：一个存储单元存储二进制代码的长度。

指令字长一般都取存储字长的整数倍，如果指令字长等于存储字长的两倍，就需要2次访存来取出一条指令，因此，取指周期为机器周期的2倍，如果指令字长等于存储字长，则取指周期等于机器周期

**数据字长**：数据总线一次能够并行传送信息的位数

64位机器既可以使用64位操作系统，也可以使用32位操作系统；32位机器不可以使用64位操作系统

### 数据的表示和运算

#### 数制和编码

##### 进位计数制及其相互转换

一个r进制数(KnKn-1…K0K-1…K-m)的数值可以表示为：  
Knrn+Kn-1rn-1+…+ K0r0 +K-1r-1+…+K-mr-m=

在计算机中，整数可以连续的表示，但小数并不是每一个十进制小数都可以准确地用二进制表示。任意的一个二进制小数都可以用十进制小数表示

##### 真值和机器数

带+-号的数称为**真值**。真值是机器数所代表的实际值

##### BCD码

二进制编码的十进制数（BCD），采用4位二进制数来表示以为十进制数中的0~9，故有6种冗余状态

8421码

* + 有权码，最常用。权值从高到低依次为8421
  + 如果两个8421码相加，之和，即(9)10，则不需要修正
  + 如果即(10)10,则需要+(0110)2/(6)10修正（从1010到1111这6个为无效吗），并向高位进位

余3码：

* + 无权码，在8421基础上加(0011)2

2421码

* + 有权码，2421
  + 特点是5的4位二进制数种最高位为1，<5的最高位为0。如510=(1011)2而不是(0101)2

##### 字符与字符串

主存字由2个或4个字节组成的时候，在同一个主存字中：

* + 小端模式： 按先存储低位字节、后存储高位字节（即从低位字节到高位字节顺序）存放字符串的内容
  + 大端模式：按先存储高位字节、后存储低位字节（即从高位字节到低位字节顺序）存放字符串的内容

##### 校验码

数据校验码的码距：通常某种编码都有许多码字组成，任意两个合法码之间最少变化的二进制位数

* + 码距2的数据校验码，开始有**检**错能力
  + 码距越大，检、纠错能力越强
  + 检错能力总是纠错能力

L-1=D+C,

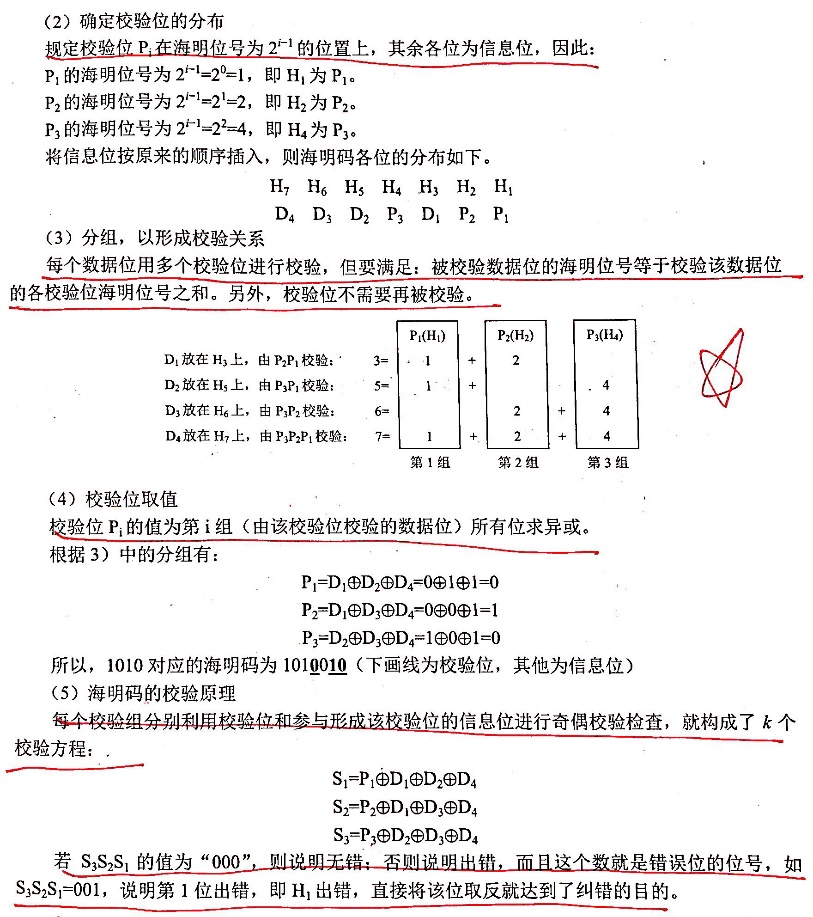
* + 编码最小码距越大，其检测错误的位数D越大，纠正错误的位数C越大
  + 纠错能力检错能力

奇偶校验码

* + 奇校验：整个校验码（有效信息位和校验位）中1的个数为奇数
  + 偶校验
  + 只能发现数据代码中奇数位出错情况，不能纠正错误

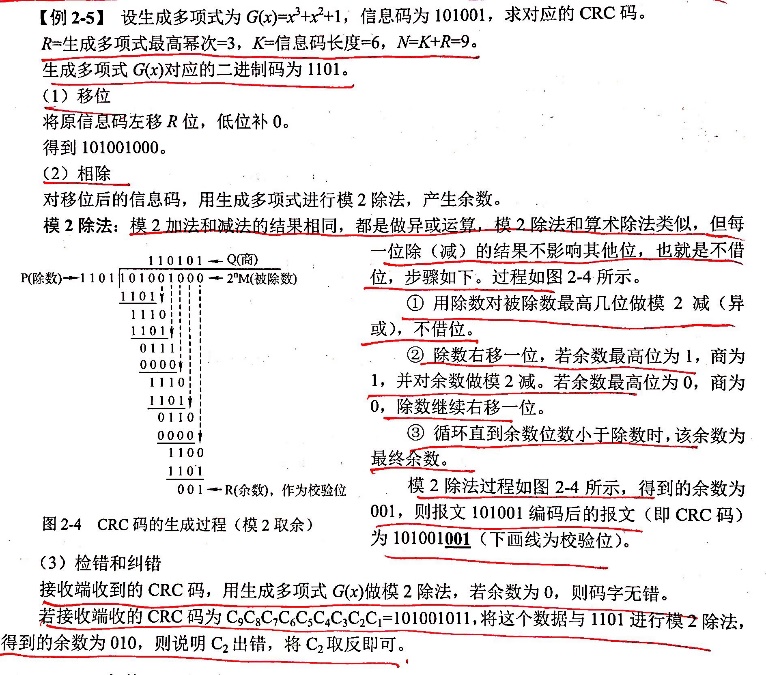
海明校验码

* + 是多重奇偶校验码
  + 求海明码的步骤
  + n=4,k=3，求1010的海明码：
  + 确定海明码位数
    - 为达到检测和纠正1位错：设n为有效信息的位数，k为校验位的位数，则n,k须满足：(若要检测两位错，则需再增加1位校验位，即k+1位)  
      海明码位数为成立，则nk有效。  
      设信息位D4D3D2D1(1010),校验位P3P2P1,对应的海明码为H7H6H5H4H3H2H1



循环冗余校验(CRC)码

* + 再K位信息位信息码后再拼接R位的校验码,整个编码的长度为N,这种编码又称为(N,K)
  + 在发送端,先将K位信息位左移R位,与生成的多项式G(x)做模2运算，生成R位校验码附在信息位后面形成（K+R）位的新的二进制码
  + G(x)的最高次幂为R,共R+1位。  
    如x3+x2+1对应的为1101



##### 习题知识点

(137.5)10=(10001001.1)2=(010001001.100)2=(211.4)8

ASCII码由7位二进制代码表示，从0000000到1111111,共128种编码。在计算机中仍以一个字节存入，最左位为0

字节编制的计算机，小端模式，01234567H，首先存放67H

增加奇偶校验位的位数，并不会提高正确性

CRC码是通过除法运算来建立数据和校验位之间的约定关系的

奇偶校验的ASCII码，校验位存放在最左端那一位

海明码

* + 发现1位，纠正1位：
  + 发现2位，纠正2位：

CRC中，接收端检测出错误后，可以：请求重发，删除数据，通过余数自动纠正

检测纠错1位，8位信息位，需要4个校验位

汉明码求一长串异或时，是不是可以相当于

* + 偶数个1则结果0
  + 奇数个1则结果1

全校验就是偶校验，比如0 10111最左位全校验，右边有偶数个1，所以全校验为0

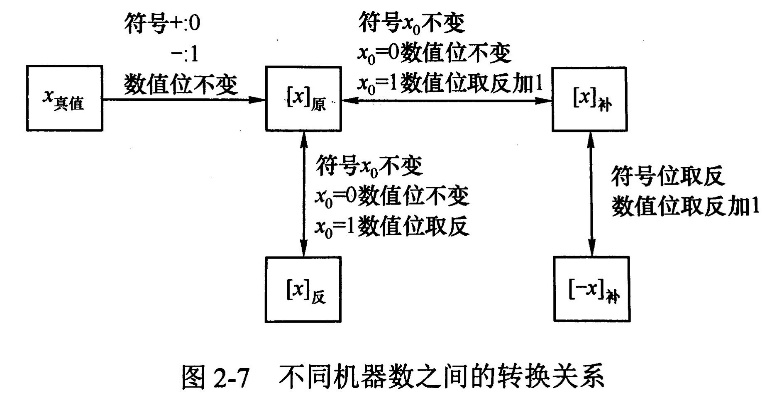
#### 定点数的表示与运算

##### 定点数的表示

设字长n+1位，其中一位符号位

* + 原码
    - 小数：
    - 整数：
    - x1=+0.1101，x2=-0.1101 字长8位  
      [x1]原码=0.1101000，[x2]原码=1.1101000
    - x1=+1110，x2=-1110 字长8位  
      [x1]原码=0,0001110，[x2]原码=1,0001110
  + 补码
    - 小数： [0]补码=0.0000…  
       [-1]补码=1.000…
    - 整数： [-]补码=1000…
    - x1=+0.1001，x2=-0.0110 字长8位  
      [x1]补码=0.1001000，[x2]补码 =1.1010000
    - x1=+1010，x2=-1101 字长8位  
      [x1]补码=0,0001010，[x2]补码 =1,1110011
  + 反码
    - 小数：  
       [+0]反码=0.00…，[-0]反码=1.111…
    - 整数： [-()]反码=1000….
    - x1=+0.0110，x2=-0.0110 字长8位  
      [x1]反码=0.0110000，[x2]反码=1.1001111
    - x1=+1011，x2=-1011 字长8位  
      [x1]反码=0,0001011，[x2]反码=1,1110100
  + 移码：
    - [0]移=1000..(n个0)
    - [-2n]移=0000
    - [2n-1]移码=1111…
    - x1=+10101，x2=-10101 字长8位  
      [x1] 移=1,0010101，[x2]移=01101011
    - 移码特性：
      * 移码就是在真值x上加一个常数（偏置值），通常取2n,相当于x在数轴上向正方向偏移了若干单位
      * [x]移=2n+x ()
      * 移码中0表示的唯一
      * 一个真值的移码和**补码**仅差一个符号位，符号位求反即是移码
      * 移码保持了数据原有的大小顺序，移码大真值就大，移码小真值就小

补码比原码，小数多个-1，整数多个-2n



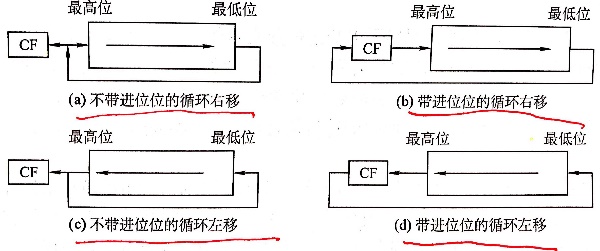
##### 定点数的运算

定点数的移位操作：

* + 算数移位：对有符号数的移位称为算数移位
    - 不论是正数还是负数，移位后符号位均不变，且移位后都相当于对真值补0。所以对于正数移位都是补0，负数则有区别
    - 对于原码，左移移位若不产生溢出，相当于\*2，右移一位，若不考虑舍去的末尾尾数，相当于/2

|  |  |  |
| --- | --- | --- |
|  | 码制 | 添补代码 |
| 正数 | 原码、补码、反码 | 0 |
| 负数 | 原码 | 0 |
| 补码 | 左移添0 |
| 右移添1 |
| 反码 | 1 |

* + 逻辑移位：操作对象时逻辑代码，可以视为无符号数
    - 逻辑左移，高位移丢，低位添0；
    - 逻辑右移，低位移丢，高位添0
  + 循环移位：
    - 带进位标志位CF的循环移位（大循环）
    - 不带进位标志位的循环移位（小循环）



原码定点数的加/减法

* + 加法：判符号位，
    - 相同：绝对值相加，符号位不变
    - 不同，绝对值大的减绝对值小的，符号位和绝对值大的相同
  + 减法：先将减数符号取反，两者按照加法运算
  + 左边出现溢出时，将溢出位丢掉

补码定点数加/减法（设机器字长n+1）

* + 符号位与数值位按同样规则一起参与运算，符号位运算产生的进位要丢掉，结果的符号位由运算得出
  + [A+B]补=[A]补+[B]补
  + [A-B]补=[A]补+[-B]补
  + 溢出位丢掉

符号扩展

* + 正数：原符号位移动到新符号位，其余补0
  + 负数：
    - 原码，同正数
    - 补码：原符号位移动到新符号位，新表示形式的所有附加位，整数补1，小数补0
    - 反码：附加为都用1填充

溢出概念和判别方法

* + 上溢：大于机器所能表示的最大整数
  + 下溢：小于机器所能表示的最小负数
  + 定点小数的表示范围 |x|<1
  + 仅两个符号相同的数相加，或两个符号相异的数相减才可能会溢出
  + 补码定点数加/减法溢出判断3种方法：
    - 采用一位符号位
      * 减法运算在机器中也是加法，所以只要参加操作的两个符号相同，结果又与员操作数不同，就是溢出
      * 设A的符号为As,B的符号位位Bs，运算结果符号位Ss  
        若V=0，无溢出；V=1，有溢出
    - 采用双符号位
      * 双符号位法也称模4补码。两个符号位相同，表示未溢出；不同，表示溢出
      * SS1SS2=00：表示结果为正数，无溢出
      * SS1SS2=01：表示结果正溢出
      * SS1SS2=10：表示结果负溢出
      * SS1SS2=11：表示结果为负数，无溢出
      * V=SS1SS2：V=0，无溢；V=1：溢出
    - 一位符号位根据进位来判断
      * 如果符号位的进位CS与最高数位的进位C1相同，没有溢出；否则溢出
      * V=CSC1

定点数的乘法运算（暂时略过）

定点数的除法运算（暂时略过）

##### 强制类型转换

强制类型转换的结果保持位值不变，知识改变了解释这些位的方便

当大字长变量向小字长变量强制类型转换时，系统把多余的高位字长部分直接截掉，低位直接赋值，这也是一种保持位值的方法

短字长到长字长变量的转换，不仅是把相应的位相等，高位部分还会拓展为原数字的符号位

##### 习题知识点

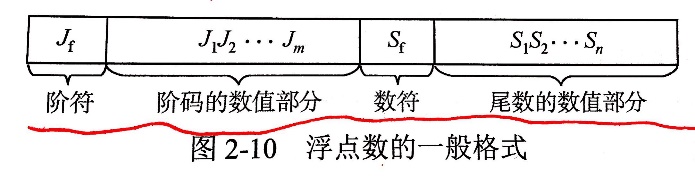
C语言的数据在内存中为补码形式

32位机器：char,1字节；short,2字节；int,4字节；long,4字节；float,4字节；double,8字节

数据转换

* + 有符号和无符号之间的转换。Signed转型化为等长的unsigned型数据，符号位成为数据的一部分，即负数转化为无符号数数值将发生变化。由unsigned转化为signed时，最高位作为符号位，也可能发生数值变化
  + 数据的截取和保留。当一个浮点数转化为整数时，浮点数的小数部分全部舍去，并按整数形式存储。但浮点数的整数部分不能超过整型数允许的最大范围，否则溢出
  + 数据转换中的精度丢失。四舍五入会丢失一些精度，截取小数也会丢失一些精度。数据由long型转化为float或double型，有可能在存储时不能准确的表示该长整数的有效数字，精度也会收到影响
  + 数据转换结果的不确定性。当较长的整数转化为较短的整数时，要将高位截取。如，long型转化称short型，只将低16位送过去，这样就会产生很大的误差。付i单数降格时，如double型转化位float型，当数值超过了float型的表示范围时，所得到的结果时不确定的

边界对齐

1. struct A{
2. int a;
3. char b;
4. short c;
5. };
6. struct B{
7. char b;
8. int a;
9. short c;
10. };
    * sizeof(strcut A)值为8字节；sizeof(struct B)的值却是12字节

16位无符号整数最大216-1=65535

对于相同位数（设为N位，不考虑符号位）的二进制补码和十进制小数，二进制小数能表示的数的个数/十进制小数能表示的个数位(0.2)N

补码表示时，符号位相同，数值位越大，码值越大（用移码理解，移码相当于，补码右移）

不带进位位循环左移，若有溢出(原数值位最高位1)，则CF变为1

[-|x|]补码=[-x]补码，当且仅当x=0或为正

三种溢出判别方法，均须有溢出判别电路，可用“异或”们来实现

8421码是十进制数的编码，不能说就是二进制数

正数的原码、反码、补码都相同

模4补码具有模2补码的全部优点且更容易检查加减运算中的溢出问题

每个模4补码存储时只需要一个符号位，只在把两个模4补码的数送往ALU完成加减运算时，才把每个数的符号位的值同时送到ALU的双符号位中，即只在ALU中采用双符号位

8位定点补码范围：-128~127

补码一位乘法中，最多需要n次位移，n+1次加法运算

原码乘法移位和加法运算最多均为n次

在原码一位乘法中，符号位不参加运算，单独处理，同号正，异号负

N位（不包含符号位）补码一位乘运算过程中一共右移位N次，加上原先的N位，一共是2N位数值位，再加上符号位，即2N+1位

原码不恢复余数除法即加减交替法，只在最终余数为负时（最后一步不够减），才需要恢复余数

补码不恢复余数除法中，异号相除时，够减商0，不够减商1

计算机中，通常用来表示主存地址的是无符号数

由三个1，五个0组成的8位二进制补码，能表示的最小整数是：[10000011]补码=[1,1111101]原码=(-125)10记住补码表示时，符号位相同，数值位越大，码值越大

[X]补=0.1011，[Y]补=1.1110，[-Y]补=0.0010  
[X+Y]补=10.1001=0.1001  
[X-Y]补=[X]补+[- Y]补=0.1101

变形补码就是模4补码00.0000,双符号位

产生进位不一定溢出

#### 浮点数的表示与运算

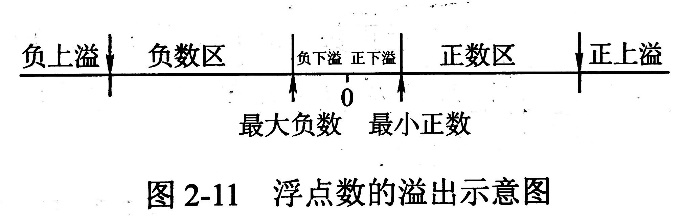
##### 浮点数的表示

浮点数被表示为,r是浮点数阶码的底（隐含），与尾数的基数相同，通常r=2。E和M都是带符号的定点数，E称为阶码，M称为尾数。可见浮点数由**阶码**和**尾数**两部分组成

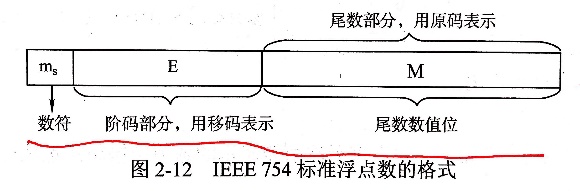
* + 阶码是整数，阶符Jf和阶码的位数m合起来反映浮点数的表示范围及小数点的实际位置；
  + 数符Sf代表浮点数的符号；
  + 尾数的位数n反映浮点数的精度

规格化浮点数

* + 规定尾数的最高数位必须是一个有效值
  + 所谓规格化就是通过调整一个非规格化浮点数的尾数和阶码的大小，使非零的浮点数尾数的最高数位上保证是一个有效值
  + **左规**：将尾数算数左移一位，阶码-1(基数为2时)，左规可能需要进行多次
  + **右规**：当浮点数运算的结果出现溢出(双符号位01,10)时，将尾数算数右移一位，阶码+1(基数为2时)。右规只需进行一次
  + 规格化浮点数的尾数M应该满足：，如果r=2，则有
  + 原码规格化后：
    - 正数为0.1xxxx形式  
      最大值表示为0.11..1；最小值0.100..0  
      尾数的表示范围
    - 负数为1.1xxxx形式  
      最大值表示为1.10..0；最小值1.11..1  
      尾数的表示范围
  + 补码规格化后：
    - 正数为0.1xx形式  
      最大值:0.11..1；最小值:0.100..0  
      尾数的表示范围
    - 负数为1.0xxx形式  
      最大值:1.01..1；最小值:1.00..0  
      尾数的表示范围
  + 基为2：
    - 原码尾数最高位一定是1
    - 补码尾数最高位与符号位相反
  + 基为4：
    - 原码尾数最高两位不全为0
    - 补码(正)尾数最高两位不全为0，(负)不全为1
  + 基为8：
    - 原码尾数最高三位不全为0
    - 补码(正)尾数最高三位不全为0，(负)不全为1

运算结果大于最大正数：正上溢；小于绝对值最大负数为负上溢，正上溢和负上溢统称上溢。产生上溢，计算机中断处理  
运算结果在0至最小正数之间称正下溢；在0至绝对值最小负数之间称负下溢。下溢计算机当作0处理

IEEE754标准

* + 

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 类型 | 数符 | 阶码 | 尾数数值 | 总位数 | 偏置值 | |
| 十六进制 | 十进制 |
| 短浮点数(单精度,float) | 1 | 8 | 23 | 32 | 7FH | 127 |
| 长浮点数(双精度double) | 1 | 11 | 52 | 64 | 3FFH | 1023 |
| 临时浮点数 | 1 | 15 | 64 | 80 | 3FFFH | 16383 |

* + IEEE754标准的浮点数（除临时浮点数外），是尾数用采取隐藏位策略的原码表示，且阶码用移码表示的浮点数
  + 对于规格化的二进制浮点数，数值的最高位总是1，IEEE754将这一位**隐藏**，因此实际上尾数数值是24位(短浮点数)。隐含的1是一位整数。在浮点格式中表示出来的23位尾数是纯小数。例如(12)10=(1100)2,规格化后1.1\*23,其中整数部分的1不存储在23位尾数中
  + 短浮点数和长浮点数都有一个隐含位，临时浮点数没有
  + 存储浮点数阶码部分之前，偏置值要先加到阶码真值上。如1.1\*23阶码值为3，故在短浮点数中，移码表示的阶码为127+3=130(82H);长浮点数中为1023+3=1026(402H)
  + IEEE754标准中，规格化的**短浮点数的真值**为：  
     (-1)sx1.Mx2E-127
  + 规格化的**长浮点数真值**为：  
     (-1)sx1.Mx2E-1023
  + S=0表示正数；  
    短浮点数E(1~254)  
    长浮点数E(1~2046)

|  |  |  |
| --- | --- | --- |
| 格式 | 最小值 | 最大值 |
| 单精度 | E=1,M=0, 1.0x21-127=2-126 | E=254,M=.11… , 1.11…1x2254-127=2127x(2-2-23) |
| 双精度 | E=1,M=0, 1.0x21-1023=2-1022 | E=2046,M=.11… , 1.11…1x22046-1023=21023x(2-2-52) |

定点数和浮点数区别

* + 若定点数和浮点数的字长相同，浮点表示法所表示的数值范围将远远大于定点表示法
  + 若定点数和浮点数的字长相同，浮点数精度低
  + 浮点数包括阶码和尾数，两个运算都要做，并且要求结果规格化，所以运算复杂
  + 定点运算中，运算结果超过数的表示范围，就溢出；浮点运算中，运算结果超出尾数表示范围不一定溢出，只有规格化后阶码超出所能表示的范围时，才发生溢出

偏置值为127（而不是128），是空出8位全1来表示无穷大（若偏置值选128，则无法区分无穷大）。阶码值E(1~254)，空出全0表示非规格化数

##### 浮点数的加减运算

浮点数运算，阶码和尾数分开进行。浮点数的加/减运算一律采用补码

1. 对阶
   * 目的让两个操作数小数点对齐，即让**阶码**相等。
   * 先求阶差，然后**小阶向大阶**看齐，将阶码小的尾数右移一位(基2)，阶+1，直到阶码相等
   * 尾数右移时，舍弃掉有效位会差生误差，影响精度
2. 尾数求和
3. 规格化
   * 以双符号位为例，尾数>0时，补码规格化形式为  
      [S]补码=00.1xx  
     尾数<0时，补码规格化形式为  
      [S]补码=11.0xx
   * **左规**：当尾数出现00.0xx或11.1xx时，需左规，即尾数左移1位，和的阶码-1，直到尾数为00.1xx或11.0xx
   * **右规**：当尾数求和结果溢出(10.xx或01.xx)时，需右规，即尾数右移一位，和的阶码+1
   * 左规一次相当于乘2，右规一次相当于除2；  
     [-1/2]补=1.100不是规格化数，需左规一次，[-1]补=1.00才是规格化数
4. 舍入
   * 在对阶和右规过程中，可能会出现将尾数低位丢失，引起误差，影响精度
   * **0舍1入法**：类似于十进制数中的四舍五入，即尾数右移时，被移去的最高数值位为0，则舍去；被移去的最高数值位为1，则在尾数的末位+1，这样做可能会使尾数又溢出，此时需再做一次右规
   * **恒置1法**：尾数右移时，不论丢掉的最高数值位是1还是0，都使右移后的尾数末位恒置1。这种方法同样有使尾数变大变小两种可能
5. 溢出判断
   * 当尾数之和/差出现01.xx或10.xx时，并不代表溢出，只有将次数右规后，再根据阶码来判断浮点数运算结果是否溢出
   * 浮点数的溢出与否是由**阶码的符号**决定的。以双符号位补码为例，
     + 当阶码符号位出现01时，即阶码大于最大阶码，表示上溢，进入中断处理
     + 当阶码符号位出现10时，即阶码小于最小阶码，表示下溢，按机器0处理
6. 强制类型转换
   * Char-int-long-double;float-double从前往后时范围和精度都从小到大，转换过程没有损失
   * Char为8位ASCII码整数，转化为int，在前面补0
   * Int和unsigned int可以互相转化，但彼此都可能因溢出而造成数据丢失，如8位int和unsigned int  
     a=-1,(unsigned int)a=255;  
     (unsigned int)a=128,(int)a=-128
   * Int和float转换，如果float是小数，转化为int可能会发生精度损失（小数仅保留整数部分）和溢出，从int转float，虽然不会发生溢出，但int可以保留32位，float保留24位，可能有数据舍入，double则不会出现  
     int<224-1时，转float就没问题（存疑）

##### 习题知识点

IEEE754单精度表示-8.25  
(8.25)10=(1000.01)2=(1.00001)2\*23E=3+127=130=(10000010)2 M=(00001)2  
1,10000010,000010000000…=C104 0000H  
此时不用进行规格化M=(00001)2

定义浮点数格式：7位阶码，1位数符，8位尾数。阶码用移码，尾数用补码，则浮点数能表示的范围：  
7位移码，除掉1个符号位，可表示范围  
   
8位不带符号的补码，可表示范围  
   
所以，浮点数能表示的范围

对阶操作不存在阶码减小，都是小阶对大阶

IEEE754短精度，阶码的偏置值127  
如阶码为(10001100)2，则快捷计算  
 (0001100)2+1=(1101)2=13  
普通计算：  
 (10001100)2=128+8+4-127=13

IEEE754长精度则是阶码的后10位（共11位）+1

和非规格化的浮点数相比，采用规格化的浮点数主要是为了增加数据的表示精度

设浮点数共12位，其中阶码含1位阶符共4位，2为底，补码；尾数含1位数符共8位，补码表示，规格化。则该浮点数能表示的最大正数为：  
阶码MAX：23-1=7，尾数：0.1111111，即1-2-7  
 27\*(1-2-7)=27-1

已知X=-0.875\*21，Y=0.625\*22，设浮点数格式为阶符1位，阶码2位，数符1位，尾数3位，通过补码求出Z=X-Y的二进制浮点数规格化结果  
 补码：(-0.875)10=-(0.111)2=(1.001)2  
 X=001,1001 Y=010,0101  
 X对齐：010,1100  
 尾数部分X-Y：10,111 需要右归  
 右规后 Z=011,1011，符合规格化

求-0.875\*21的补码的较快方式：  
 -0.875\*2=-1.75=(-1.11)2=(10.01)2=1.001\*21

假设机器数用反码表示，当机器数为负时，左移时最高数位丢0，结果出错；右移时，最低数位丢0，影响精度

在算数移位的情况下，反码、补码左移的前提条件时其原最高有效位与原符号位要相同

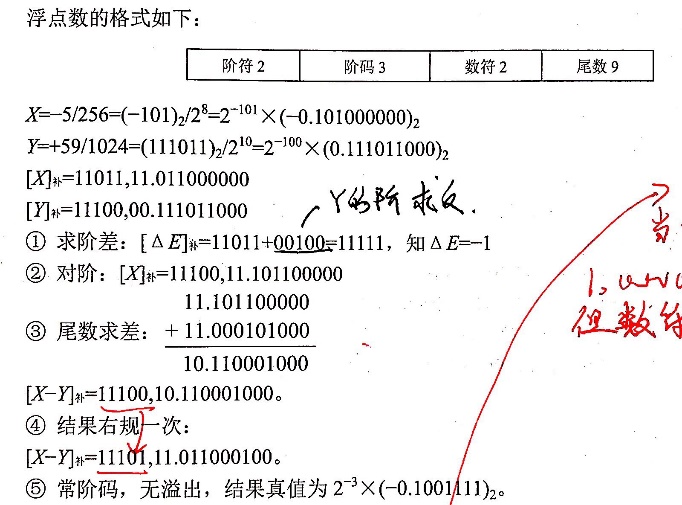
在算数移位的情况下，双符号位的移位操作只有低符号位需要参加移位操作

在采用舍入到最接近且可表示的值时，若要舍入成两个有效数字形式，(12.5)D应该舍入为12  
由于最后一位是5，采用取偶数的方式，即有两个最接近值时，选择偶数，此题有12，13两个数，取12

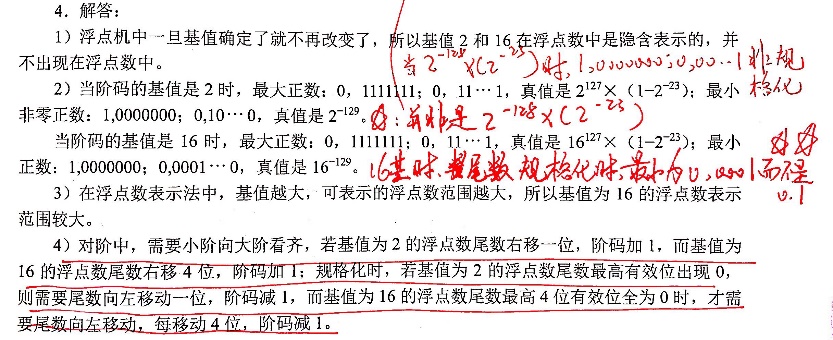
舍入时浮点数的概念，定点数没有舍入

浮点数舍入的情况有两种：对阶、右规格化

舍入不一定产生误差

已知十进制数X=-5/256，Y=+59/1024，按机器补码浮点运算规则计算X-Y,结果用二进制表示，浮点数格式如下：阶符取2位，阶码取3位，数符取2位，尾数取9位

设浮点数长32位，其中阶码部分8位（含一位阶符），尾数部分24位（含一位数符），当阶码的基值分别是2和16时：

* + 说明基值2和16在浮点数中如何表示
  + 当阶码和尾数均用补码表示，且尾数采用规格化形式时，给出两种情况下所能表示的最大正数真值和非零最小真值
  + 在哪种情况下，数的表示范围大
  + 两种基值，对阶和规格化操作有何不同

Float型变量在计算机中都被表示成IEEE754单精度格式

IEEE754在进行运算时，对阶操作，尾数右移，不要忘了隐藏的”1”,也是很要跟着右移的

IEEE754在初始化的时候，要求数值第一位必须是1，然后隐藏；其他浮点数格式定义，按照原码/补码规格化定义来，如补码就要求符号位和第一数值位相反

两个规格化浮点数进行加减运算，最多右规一次；左规次数无法确定，但不会超过尾数的字长n位次

IEEE754

* + 阶码全0 & 符号位1 & 尾数全0：-0
  + 阶码全0 & 符号位0 & 尾数全0：+0
  + 阶码全1 & 符号位1 & 尾数全0：负无穷大
  + 阶码全1 & 符号位0 & 尾数全0：正无穷大
  + 阶码全0 & 尾数非全0：非规格化数
    - 非规格化数，无隐含的”1”，只有规格化数才隐含”1”，如
    - 1,00000000,10000000…实际值为  
      (-0.1)2\*2-126=-2-127 (20-127=-126)

#### 算数逻辑单元ALU

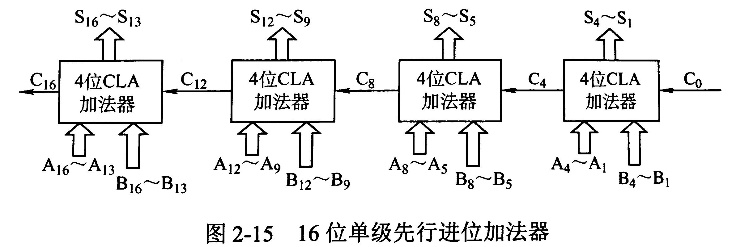
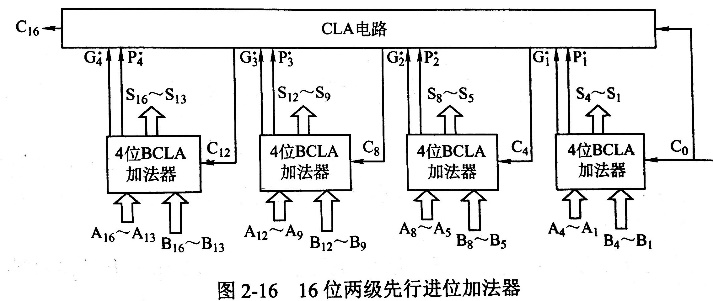
##### 串行加法器和并行加法器

串行加法器只有一个全加器。如果操作数长n位，加法就要分n次进行，每次产生一位和，并且串行逐位地送回寄存器

并行加法器

* + 串行进位：把n个全加器串接起来，就可可以进行两个n位数相加，即串行进位地并行加法器。串行进位又称**行波进位**，每一级直接依赖于前一级地进位，即进位信号是逐级形成地。最长运算时间主要是由进位信号地传递时间决定的。
  + 并行进位：又称**先行进位**，同时进位。特点是各级进位信号同时产生

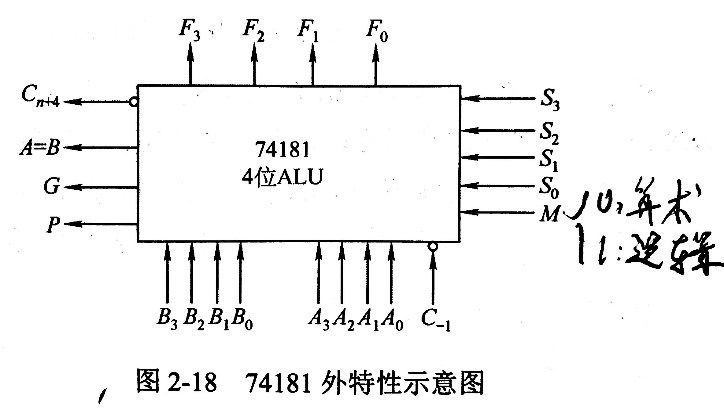
分组并行进位方式

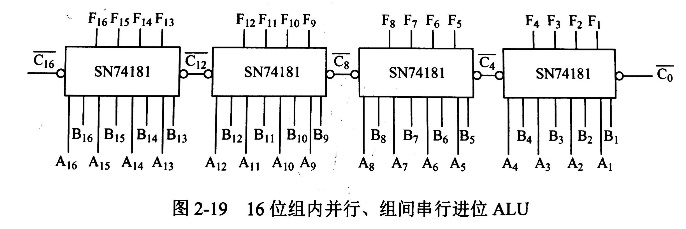
* + **单级先行进位**方式又称组内并行、组间串行
    - CLA：先行进位电路
  + **多级先行进位**方式又称组内并行、组间并行
    - BCLA：成组先行进位电路

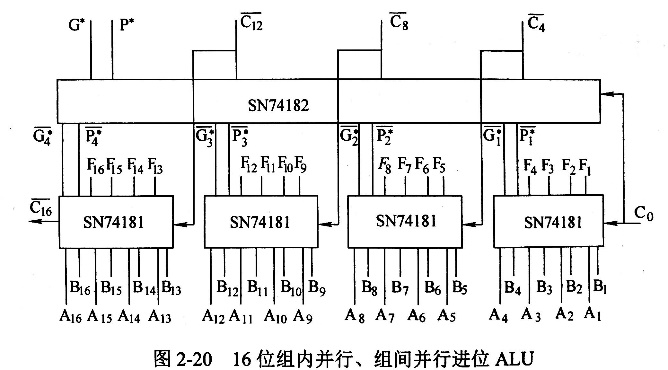
##### 算术逻辑单元的功能和结构

ALU的核心首先应当是一个并行加法器，同时也能执行与或非等逻辑运算

4位ALU芯片（74181，4位并行加法器）

74181能执行16种算数运算，16种逻辑运算，可工作在正负逻辑下。M用来区分算数运算和逻辑运算，S3~S0的不同取值实现不同操作

74182先行进位芯片



##### 习题知识点

ALU是由组合逻辑电路构成的，最基本的部件是并行加法器。单纯的ALU不能够存储运算结果和中间变量，往往将ALU和寄存器或暂存器相连

串行进位的并行加法器，影响加法器运算速度的关键因素是进位传递延迟

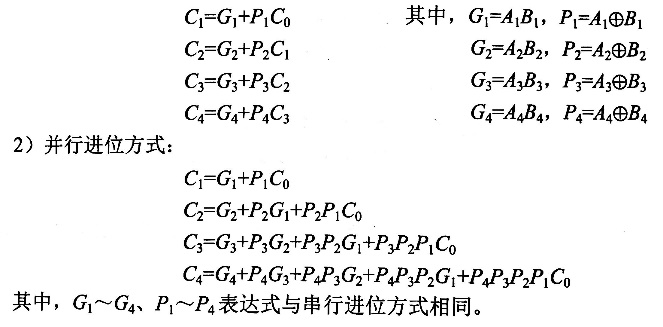
在设计多维加法器时，为了加快运算速度而采用了快速进位链，即对加法器每一位都生成两个信号：进位信号g和进位传递信号p，

用8片74181和两片74182可组成二级先行进位结构的32位ALU  
74181：4位内部先行进位的ALU  
74182：4位先行进位ALU  
每4片74181与1片74182相连，可组成一个两级先行进位结构的16位ALU，两个这种结构的16位ALU串行进位构成两级先行进位的32位ALU

状态寄存器，ALU，数据总线都是组成运算器的部件

某加法器进位链小组信号为，低位来的进位信号为，请分别写出的逻辑表达式

* + 串行进位方式
  + 并行进位方式



#### 易混知识点

字长相同时，浮点数和定点数表示范围与精度

* + 浮点数范围大，定点数精度大

现代计算机用IEEE754标准表示浮点数

* + Float:32位单精度浮点数
  + Double:64位双精度浮点数

C语言中的int和unsigned型变量的存储方式没有区别，都是按照补码的形式存储，在不溢出的范围内的加减法运算也是相同的

对无符号定点整数来说，若寄存器尾数不够，计算运算过程中一般保留低n位，舍弃高位

* + 保留的低n位数不能正确表示运算结果，即有效位超过了n位，发生溢出现象
  + 保留的低n位能正确表达计算结果，即高位的舍去并不影响其运算结果

如何判断一个数是否时规格化数

* + 规格化浮点数的尾数小数点后的第一位一定是个非零数。因此，
  + 对于原码编码的尾数来说，只要看尾数的第一位是否为1就行
  + 对于补码表示的尾数，只要看符号位和尾数最高位是否相反。
  + IEEE754是原码表示的尾数，且1隐藏

对于尾数相同的定点数和浮点数，可表示的浮点数个数和定点数个数应该一样多

* + 可表示的个数取决于编码所采用的位数，编码数一定，则编码出来的数据个数就是一定的。N位编码只能表示2n个数
  + 对于相同位数的浮点数和定点数，可表示的数据个数应该一样多（有时可能由于一个值由两个或多个编码对用，编码个数会有少量差异，如补码和原码在0的表示）

IEEE754四种舍入方式：

* + 就近舍入：舍入为最近可表示的数，若结果值正好落在两个可表示数的中间，一般选择偶数
  + 正向舍入：朝方向舍入，即取右边的那个数
  + 负向舍入：朝方向舍入，即取左边的那个数
  + 截去：朝0方向舍入，即取绝对值较小的那个数

原码加/减法运算可以有以下两种实现方式：

* + 转化为补码后，用补码加减法实现，再转化为原码
  + 直接用原码进行加/减运算，符号和数值部分分开进行

长度为n+1的定点数，按照不同的编码方式，表示的数值范围：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编码方式 | 最小值编码 | 最小值 | 最大值编码 | 最大值 | 数值范围 |
| 无符号定点整数 |  |  |  |  |  |
| 无符号定点小数 |  |  |  |  |  |
| 原码定点整数 |  |  |  |  |  |
| 原码定点小数 |  |  |  |  |  |
| 补码定点整数 |  |  |  |  |  |
| 补码定点小数 |  |  |  |  |  |
| 反码定点整数 |  |  |  |  |  |
| 反码定点小数 |  |  |  |  |  |
| 移码定点整数 |  |  |  |  |  |
| 移码定点小数 | 小数没有移码定义 | | | | |

设阶码和位数均用补码表示，阶码部分共K+1位（含1位阶符），位数部分共n+1位（含1位数符），则这样的浮点数的表示范围：

|  |  |  |  |
| --- | --- | --- | --- |
| 浮点数 | 浮点表示 | | 真值 |
| 阶码 | 尾数 |
| 最大正数 | 01…1 | 0.11..11 |  |
| 绝对值最大负数 | 01…1 | 1.00..00 |  |
| 最小正数 | 10…0 | 0.00..01 |  |
| 规格化的最小正数 | 10…0 | 0.10..00 |  |
| 绝对值最小负数 | 10…0 | 1.11..11 |  |
| 规格化的绝对值最小负数 | 10…0 | 1.01..11 |  |

### 存储系统

#### 存储器的层次结构

随机存储器RAM

制度存储其ROM，它与RAM可共同作为主存的一部分，统一构成主存的地址域

串行访问存储器

CDROM即光盘，串行存取方式，是只读型光盘存储器，其访问方式是顺序访问，不属于只读存储器（ROM）

EPROM,DRAM,SRAM都采用随机存取方式

存储器存取周期：存储器进行连续读或写操作所允许的最短时间间隔

某计算机系统，其操作系统保存在硬盘上，其内存储器应该采用RAM和ROM。引导程序放在ROM

CPU可直接访问：内存，Cache，寄存器

数据传输率中的M指的是106，而非220

ns=10-9s

#### 存储器的层次化结构

主存和Cache之间的数据调动是由硬件自动完成的，对所有程序员都透明

主存和辅存之间的数据调动是硬件和操作系统共同完成的，对应用程序员透明

Cache-主存，主存-辅存 层次中，上一层中的内容都只是下一层中内容的副本

Cache-主存 结构是为了解决主存与CPU速度不匹配

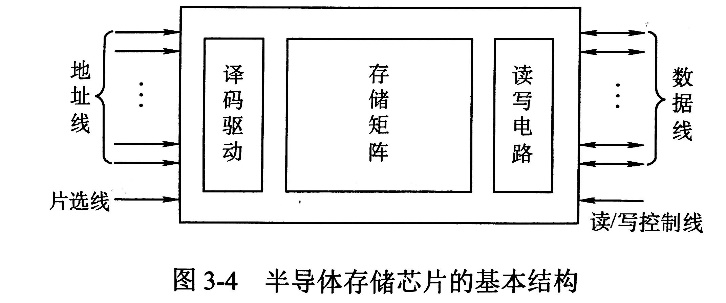
速度由快到慢：寄存器-Cache-主存-辅存

CPU与主存可直接进行数据交换

Cache主存系统的效率  
e=访问Cache的时间/平均访存时间

#### 半导体随机存储器

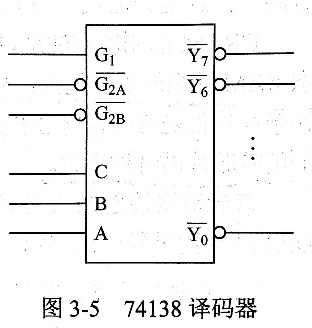
##### 半导体存储芯片



* + 存储矩阵
  + **译码驱动**:将来自**地址总线**的地址信号翻译成对应存储单元的选通信号,该信号在**读写电路**的配合下完成对被选中单元的读/写操作
  + **读写电路**:包括读出放大器和写入电路,用来完成读写操作
  + **读/写控制线**:决定芯片进行读/写操作
  + **片选线**:决定哪个存储**芯片**被选中
  + **地址线**:**单向**输入,其位数与存储字的**个数**有关
  + **数据线**:**双向**,其位数与读出或写入的**数据位数**有关,数据线数和地址线数共同反映存储芯片容量的大小.如地址线10根，数据线8根，则芯片容量=210\*8=8K位

高速缓冲存储器大多由静态RAM实现，动态RAM广泛应用于主存

74138译码器，又称3/8译码器

* + 输入3位二进制码，8个输出端对应8种输入状态
  + 
  + ABC为译码地址输入端；为选通端，为译码输出端（低电平有效）。当选通端为高电平，为低电平时，才可将地址端ABC的二进制编码在一个对应的输出端以低电平溢出

##### SRAM存储器和DRAM存储器

静态随机存储器(SRAM)的存储元是用双稳态触发器（六管MOS）来记忆信息的。

* + 非破坏性读出。
  + 易失性半导体存储器。
  + 存取速度快，集成度低，功耗大，一般用于高速缓冲存储器
  + 不需要刷新

动态随机存储器(DRAM)利用栅极电容上的电荷来存储信息。

* + **地址复用技术**，地址线是原来的1/2，且地址信号分行、列两次传送
  + 一般用于组成大容量主存系统
  + DRAM电容电荷只能维持1~2ms，需隔一段时间就刷新一次，通常2ms，这个时间称作刷新周期。
    - **集中刷新**：在一个刷新周期内，利用一段固定的时间，依次对**所有行**进行逐一再生，在此期间停止对存储器的读写操作，称为“死时间”，又称访存“死区”  
      优点：读写操作时不受刷新工作影响，存取速度比较高；  
      缺点在集中刷新期间，不能访问存储器
    - **分散刷新**：把对每一行的刷新分散到各个**工作周期**中去。  
      工作周期分为两部分：前半部分正常读写或保持；后半部分用于刷新某一行。  
      这种刷新方式增加了系统的存取周期，如存储芯片的存取周期时0.5,则系统的存取周期应为1  
      优点：没有死区；缺点：加长了系统的存取周期
    - **异步刷新**：将刷新周期除以行数，得到两次刷新操作之间的时间间隔t，利用逻辑电路每隔时间t产生一次刷新请求，**缩短**了“死时间”，减少刷新次数  
      如果将刷新安排在不需要访问存储器的**译码**阶段，则既不会加长存取周期，也不会产生“死时间”，称作“透明刷新”
  + 刷新对CPU时透明的，即刷新不依赖于外部的访问
  + DRAM的刷新单位是行，故刷新操作时仅需要行地址
  + 刷新操作类似于读操作，仅是给栅极电容补充电荷，不输出信息
  + 刷新时不需要片选，即整个存储器中的所有芯片都刷新

读写周期（暂时不记笔记，因为没有题）

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

##### 只读存储器ROM

SRAM,DRAM都是易失性存储器

掩膜式只读存储器（MROM）

一次可编程只读存储器（PROM）

可擦除可编程只读存储器（EPROM）

闪速存储器（Flash Memory）

固态硬盘（SSD）

##### 习题知识点

某SRAM芯片，容量为1024\*8位，出电源和接地端外，该芯片的引脚最少数目为：10+8+1+2=21

* + 容量1024B，以字节为单位存取，地址线需要10根（1024=210）
  + 8位说明数据线需要8根
  + 片选线
  + 读/写控制线(读控制为RD，写WE，2根)

DRAM刷新以**行**为单位

集中刷新和异步刷新都有死时间，分散刷新没有

DRAM，电容上有电荷1，无电荷0

某DRAM芯片，采用地址复用技术，容量为1024\*8位，除电源和接地端外，该芯片引脚最少为（读写控制线为2根）：17

* + 数据线8根
  + 地址线5根：通过行选通和列选通分行列两次传送地址信号
  + **行选通（代替片选线）**1
  + 列选通1
  + 读写控制线2

某容量为256MB的存储器由若干4M\*8位的DRAM芯片构成，该DRAM芯片的地址引脚和数据引脚总数为：19

* + 4M=222，地址线22/2=11
  + 数据线8

闪存（Flash Memory）

* + 存储元由MOS管组成，是一种半导体存储器
  + 读快写慢

U盘属于只读存储器

随机存取和随机存取存储器(RAM)是不同的，只读存储器ROM也是随机存取的。支持随机存取的存储器并不一定是随机存取存储器

EPROM是可多次改写，但不能作为随机存储器

DRAM每隔一定时间，需要根据原存内容重新写入一遍

刷新时，读并不是把信息读入CPU，写也并不是从CPU写入主存，它只是把信息读出，通过一个**刷新放大器**后又重新存回到存储单元中去，刷新放大器集成在RAM上，只进行一次访存，也就是只占用**一个存取周期**

提高刷新存储器带宽，可采用：

* + 使用高速DRAM芯片
  + 采用多体交叉存储结构
  + 刷新存储器至显示控制器的内部总线带宽加倍
  + 采用双端口存储器将刷新端口和更新端口分开

一个1K\*4位的DRAM芯片，若其内部结构排列成64\*64形式，且存取周期位0.1。

* + 若采用异步刷新，刷新信号周期
    - 64\*64，需在2ms内将64行刷新一遍，则刷新信号的时间间隔为2ms/64=31.25，故取刷新周期31
  + 若采用集中刷新，则对该芯片刷新一遍需要时间，死时间率为
    - 0.1\*64行=6.4，即在2ms内集中6.4刷新
    - 死时间率为：6.4/2000=0.32%

|  | SRAM | DRAM |
| --- | --- | --- |
| 数据线 | √ | √ |
| 地址线 | √ | 复用 |
| 行选通 |  | √ |
| 列选通 |  | √ |
| 片选线 | √ |  |
| 读写控制 | √两根 | √两根 |

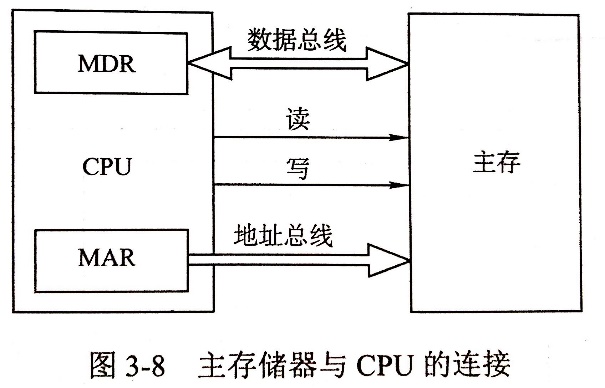
#### 主存储器与CPU的连接

##### 连接原理

主存通过**数据线**，**地址总线**和**控制总线**和CPU连接

数据宗信的位数与工作频率的乘机正比于数据传输率

地址总线的位数决定了可寻址的最大内存空间

控制总线（读/写）支出总线周期的类型和本次输入/输出操作完成的时刻

##### 主存容量的扩展

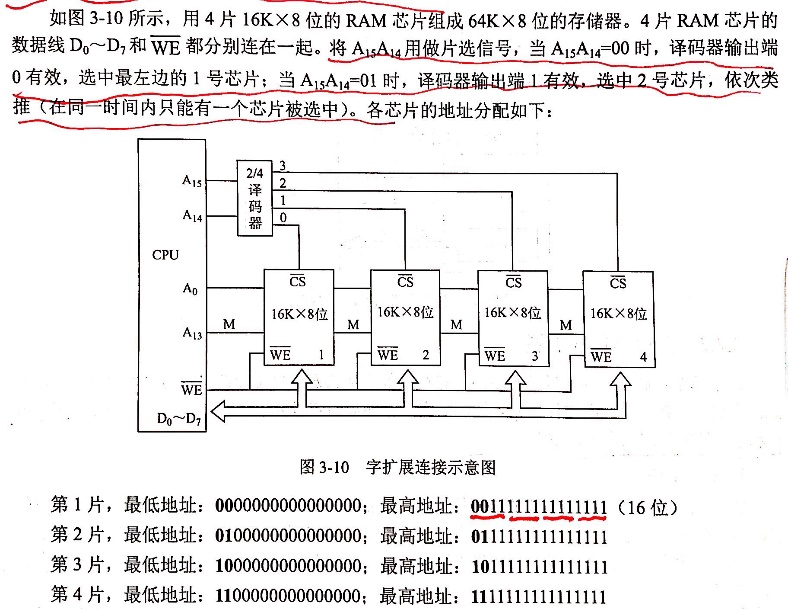
位扩展法

* + CPU数据线数与存储芯片数据线位数不一致，则需位扩展，使其位数与CPU数据线数相等
  + 多个芯片的**地址端**、**片选端**()和**读写控制端**()并联，**数据端**分别引出
  + 仅采用位扩展时，各芯片连接**地址线**的方式相同，连接**数据线**的方式不同。
  + 在某一时刻选中所有芯片，片选信号()要连接到所有芯片



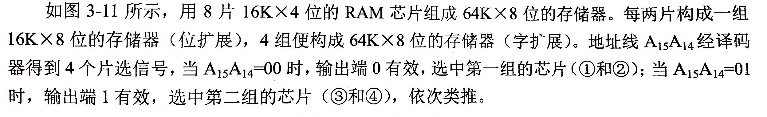
字扩展法

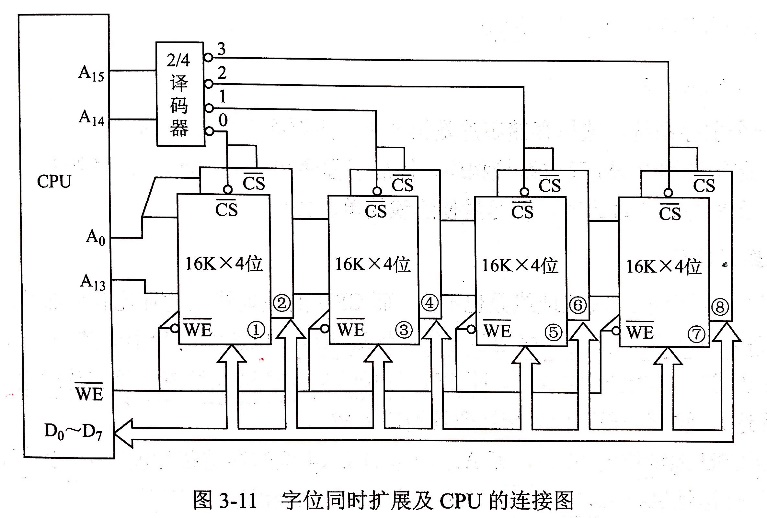
* + **地址线**，**数据线**，**读写控制线**()相应并联，由**片选信号**()来区分各芯片的地址范围
  + 仅采用字扩展时，各芯片连接**地址线**的方式相同，连接**数据线**的方式也相同
  + 某一时刻只需选中部分芯片，所以通过片选信号()或采用译码器设计连接到相应的芯片



字位同时扩展法

* + 采用字位同时扩展时，各芯片连接地址线的方式相同，但连接数据线的方式不同，而且需要通过片选信号()或采用译码器设计连接到相应的芯片





##### 存储芯片的地址分配和片选

**片选**：选择存储芯片

**字选**：为选中的芯片依地址码选择相应的存储单元，以进行数据的存取

片内的**字选**通常是由CPU送出的N条低位地址线完成的，地址线直接接到所有存储芯片的地址输入端（N由片内存储容量2N决定）

片选信号的产生分为**线选法**和**译码片选法**

线选法

* + 用除片内寻址外的高位地址线直接（或经反相器）分别接至各个存储芯片的片选端，当某地址线信息为”0”，就选中与之对应的存储芯片
  + 一次只能有一个有效
  + 假设4片2K\*8位存储芯片用线选法构成8K\*8位存储器，低位地址线A10~A0作为字选线，用于片内寻址。片选信号：

| 芯片 | A14~A11 |
| --- | --- |
| 0# | 1110 |
| 1# | 1101 |
| 2# | 1011 |
| 3# | 0111 |

* + 优点：不行还要地址译码器，线路简单
  + 缺点：地址空间不连续，选片的地址线必须分时为低电平（否则不工作），不能充分利用系统的存储空间，造成地址资源的浪费

译码片选法

* + 译码片选法用除片内寻址外的高位地址线通过地址译码器芯片产生片选信号
  + 如8片8k\*8位的存储芯片组成64K\*8位存储器（地址线为16位，数据线为8位），需要8个片选信号；若采用线选法，出去片内寻址的13位地址线，仅剩下3个高位，不足产生8个片选引号。采用74LS138作为地址译码器，A15 A14 A13=000时选中第一片，A15 A14 A13=001时选中第二片

##### 存储器与CPU的连接

通常ROM存放系统程序、标准子程序和各类常数

RAM则是为用户编程而设置的

通常将CPU地址线的低位与存储芯片的地址线相连，完成**字选**；CPU地址线的高位则在扩展存储芯片时使用，完成**片选**，这部分译码由外接译码器逻辑完成

CPU数据线与存储芯片的数据线不相等时，必须对存储芯片扩位，使之相等

CPU读/写命令线一般可直接与存储芯片的读/写控制端相连，通常高电平为**读**，低电平为**写**。

有些CPU的读/写命令线是分开的（读为写为，均为低电平有效）。此时CPU的读/写命令线应分别与存储芯片的允许读/写控制端相连

片选线的连接是CPU与存储芯片连接的关键。哪一片被选中完全取决于该存储芯片的片选控制端能否接收到CPU的片选有效信号

片选有效信号与CPU的访存控制信号（低电平有效）有关，因为只有当CPU要求访存时，才要求选中芯片；若CPU访问I/O则为高，表示不要求存储器工作

##### 习题知识点

假定用若干个2K\*4位的芯片组成一个8K\*8位的存储器，则地址0B1FH所在芯片的最小地址是：0800H

* + 1：0000H~07FFH
  + 2：0800H~0FFFH
  + 3：1000H~17FFH
  + 4：1800H~1FFFH

某计算机存储器按字节编址，主存地址空间大小为64MB，现用4M\*8位的RAM芯片组成32MB的主存储器，则MAR的位数至少是26位：

* + 应按照地址空间64MB来计算，不能按照主存实际容量来计算。64MB=226B

某计算机字长16位，存储器容量为256KB，CPU按字寻址，其寻址范围是0~217-1

设地址总线为A0(高位)~A15(低位)，用4K\*4位的存储芯片组成16KB存储器，则产生片选信号的译码器的输入地址线应该是 A2A3

* + 接入各芯片地址段的是地址线的低12位（4K=212）A4~A15
  + 将两个4K\*4位芯片组成4K\*8位的芯片，16KB/4KB=4组，即4组，片选4=22，A2A3

若内存地址区间4000H~43FFH,每个存储单元可存储16位二进制数，该内存区域用4片存储芯片组成，则构成该内存所用的存储器芯片的容量是256\*16bit

* + 0100,00**00**,0000,0000  
    0100,00**11**,1111,1111  
    说明每个芯片片内地址8位，28=256
  + 43FF-4000+1=400H,即内存区域为1K个单元，总容量为1K\*16位。现由4片存储芯片构成，则构成该内存芯片的容量为1K\*16位/4=256\*16位

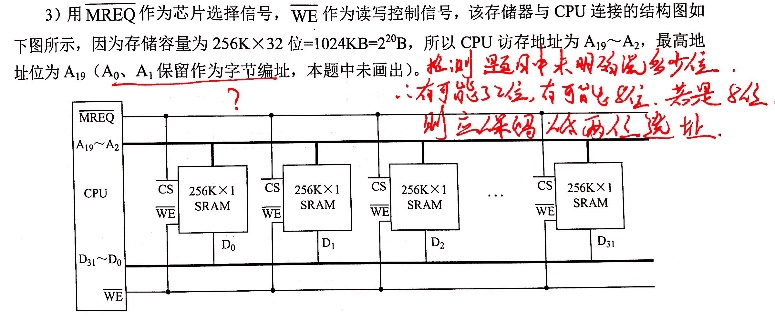
内存按字节编址，地址从90000H到CFFFFH，若用存储容量为16K\*8位芯片构成该内存，至少需要的芯片数是：16

* + CFFFFH-90000H+1  
    =D0000H-90000H  
    =40000H=4\*24\*24\*24\*24=218个单元
  + 218/16K=16片

若地址寄存器为18位，则编址范围：

* + 起始地址00000H
  + 末地址00000H+218-1=40000H-1=3FFFFH

32片256K\*1位SRAM芯片，画出该存储器与CPU连接的结构图，设CPU的接口信号有地址信号、数据信号和控制信号、



王道p106 5、6题没看，太长了，今天很困，以后看

#### 双口RAM和多模块存储器

##### 双端口RAM

**空间**并行

两个端口有两组相互独立的地址线、数据线和读写控制线，允许两个独立的控制器同时异步地访存储单元。当两个端口地地址线不同时，在两个端口上进行读写一定不会发生冲突

不会出错：

* + 两个端口不同时对同一地址单元取元素
  + 两个端口同时对同一地址单元读出数据

写入错误：

* + 两个端口同时对同一地址单元写入数据

读出错误：

* + 两个端口同时对同一地址单元，一个写入数据，一个读出数据

解决办法：置“忙”信号为0，由逻辑判断决定暂时关闭一个端口（即被延时）

##### 多模块存储

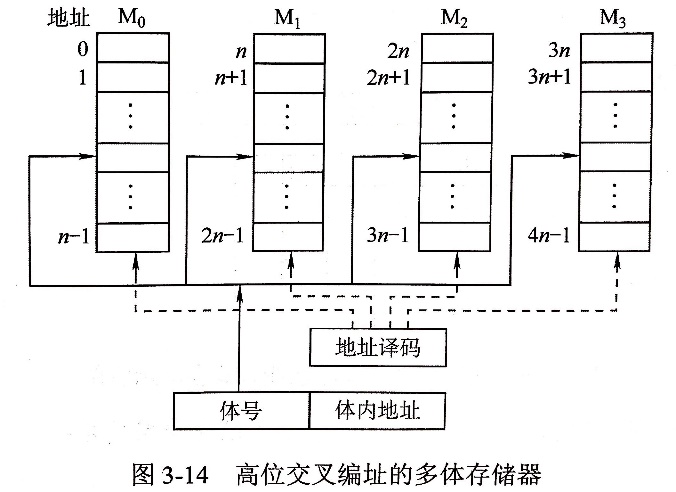
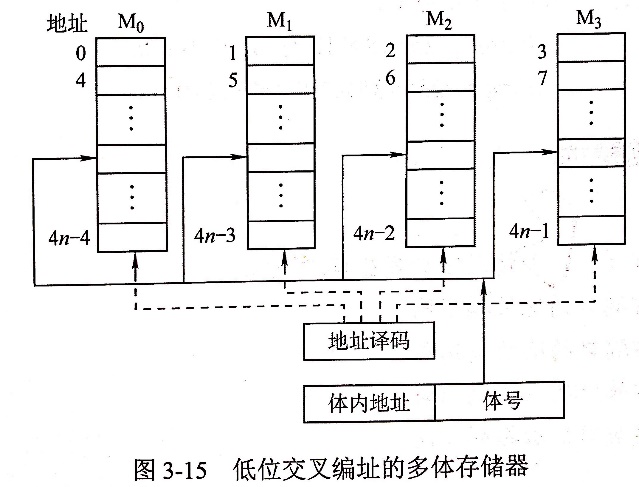
**时间**并行

常用：**单体多字**存储器和**多体低位交叉**存储器

单体多字存储器

* + 存储器中只有一个存储体，每个存储单元存储m个字，总线宽度也为m个字，地址必须顺序排列并除以同一存储单元
  + 单体多字存储系统在一个存取周期内，从同一地址取出m条指令，然后逐条将指令送至CPU执行，即每隔1/m存取周期，CPU向主存取一条指令
  + 缺点：指令和数据在内存内必须是连续存放地，一旦遇到转移指令，或者操作数不能连续存放，这种方法地效果就不明显

多体低位交叉存储器

* + 各模块都有独立地读写控制电路、地址寄存器和数据寄存器。既能并行操作，又能交叉工作
  + **高位**交叉编址（顺序方式）：高位地址表示体号，低位地址为体内地址；仍是顺序存储器
  + 
  + **低位**交叉编址（交叉方式）：低位表示体号，高位表示体内地址。程序连续存放在相邻体内
  + 多体模块结构地存储器采用低位交叉编址后，可以在不改变每个模块存储周期的情况下，采用**流水线**方式并行存取
  + 设模块字长=总线宽度=b位，模块存取一个字的周期为T，总线传送周期为r，为实现流水线方式存取，则存储器交叉模块数应，m称为交叉存取度。每经r时间延迟后启动下一模块。交叉存储器要求其模块数必须m  
    这样连续存取m个字所需的时间为：  
     t1=T+(m-1)r  
    前T时间内，可传输b位数据，然后接下来时间每r时间，都可以传输b位数据

##### 习题知识点

ns = 10-9s

**高位**多体交叉存储器不能保证程序的局部性原理，**低位**多体交叉存储器可以

高位四体交叉存储器**可能**在一个存储周期内连续访问四个模块

双端口存储器可以同时访问同一区间、同一单元

多体并行存储器，在r时间内，整个存储器可向CPU提供共b位信息，但每个存储体必须经过T时间才能提供b位信息

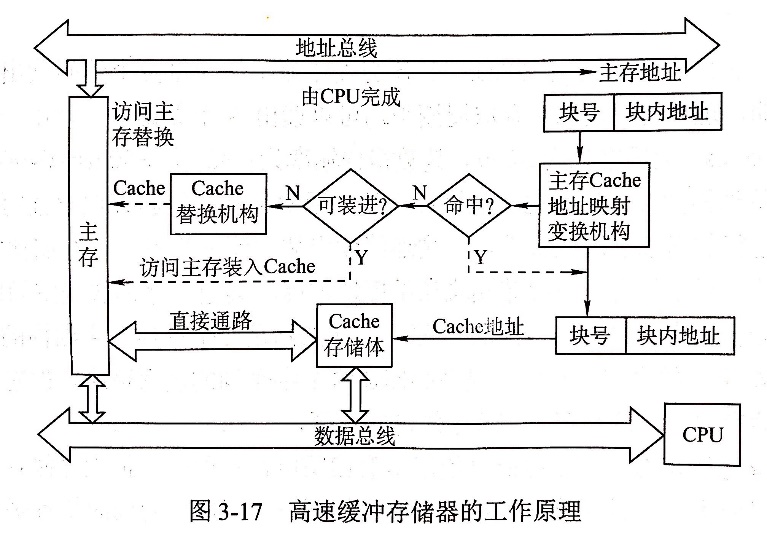
4体低位地址交叉的存储器和4端口读出的存储器比较：

* + 多端口存储器是对同一个存储体使用多套读写电路实现的，扩大存储容量的难度显然比多体结构的存储器大，而且不能对多端口存储器的同行一个存储单元同时执行多个写入操作，而多体结构的存储器允许在同一个存储周期对几个存储体执行写入操作

#### 高速缓冲存储器

##### Cache基本工作原理

Cache位于存储器层次结构的顶层，通常由SRAM构成



Cache和主存都被分为若干大小相等的块（Cache块又称Cache行），每块由若干字节组成，块的长度称为块长（Cache行长）

当CPU发出读请求，如果访存地址在Cache命中，就将此地址转成Cache地址，直接对Cache操作，与主存无关

如果Cache不命中，访问主存，并将**此字所在的块**一次从主存调入Cache。若Cache满，则根据某替换算法替换

Cache与CPU之间交换以**字**为单位，Cache与主存之间以**Cache块**为单位

某些计算机也采用同时访问Cache和主存的方式，若Cache命中，则主存访问终止；否则访问主存并替换Cache

Cache的总命中次数Nc，访问主存的总次数为Nm，命中率：H=Nc/(Nc+Nm)

tc命中时的Cache访问时间，tm未命中时访问时间，则平均访问时间：Ta=H\*tc+(1-H)\* tm

##### Cache和主存的映射方式

在Cache中为每一块加标记，指明是主存中哪一块的副本，相当于主存中块的编号；加有效位，1时表示Cahce映射的主存数据块有效，否则无效；

直接映射

* + 主存数据块只能装入Cache中的唯一位置。若这个位置已有内容，则产生块冲突，原来的块无条件被替换（无需替换算法）
  + 块冲突概率最高，空间利用率最低
  + j=i mod 2c  
    j是Cache块号（行号）；i是主存的块号；2c是Cache中的总块数  
    主存的第0块、第2c块、第2c+1块，只能映射到Cache的第0行

|  |  |  |
| --- | --- | --- |
| 主存字块标记 | Cache字块地址 | 字块内地址 |

* + 主存字块标记：主存/Cache的位数
  + Cache字块地址：共有几个字块的位数
  + 字块内地址：字块大小位数

全相联映射

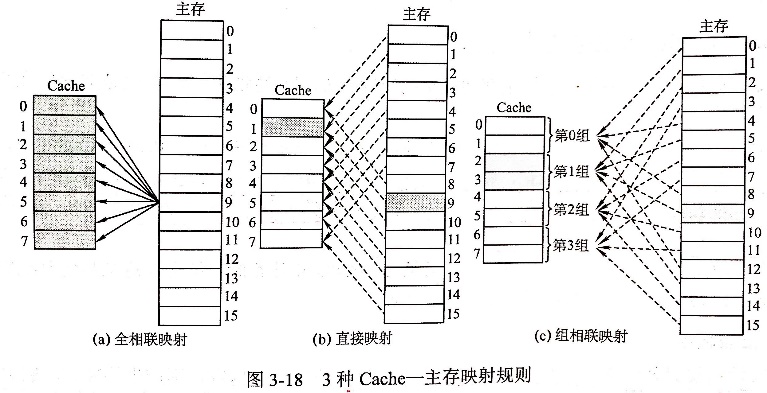
* + 通常采用**相联存储器**

|  |  |
| --- | --- |
| 主存字块地址 | 字块内地址 |

组相联映射

* + 将Cache空间分成大小相同的组，主存的一个数据块可以装到**一个组**内的任何一个位置，即组间直接映射，组内全相联映射。
  + j = i mod Q  
    j是缓存的组号，i是主存的块号，Q是Cache的组数  
    Q=1时，变为全相联映射，当Q=Cache块数 时变为直接映射
  + 地址结构：

|  |  |  |
| --- | --- | --- |
| 主存字块标记 | 组地址 | 字块内地址 |



假设某个计算机的主存地址空间大小为256MB，按字节编址，其数据Cache有8个Cache行，行长64B

* 1. 若不考虑用于Cache的一致维护性和替换算法控制位，采用**直接映射**方式，则该数据Cache的**总容量**为多少
  2. 若该Cache采用直接映射方式，则主存地址为3200（十进制）的主存块对应的Cache**行号**是多少？若采用二路组相联映射又是多少？
  3. 以直接映射方式为例，简述访存过程（设访存地址为0123456H）

1. 数据Cache总容量为4256bit。

（这标记阵列中的**一致性维护位**和Cache数据一致性维护方式相关，**替换算法控制位**和替换算法相关，这里不计算）

* + 每一个Cache行对应一个标记项
  + 而在**组相联**中，将每一组的标记项排成一行，将各组从上而下排列，成为一个二维的标记阵列（**直接映射**一行就是一组）。查找Cache就是查找标记阵列的标记项是否符合要求
  + Cache每块都有个标记项，无论直接映射还是组相联映射，所以Cache的标记阵列中需要有行数个标记项
  + 标记阵列示意图

|  |  |
| --- | --- |
| 标记项 | 标记项 |
| 标记项 | 标记项 |
| 标记项 | 标记项 |
| 标记项 | 标记项 |

故本题中，每行相关的存储器容量如图

|  |  |  |
| --- | --- | --- |
| 有效位1bit | 标记位 19bit | 每行存储的数据64B=512bit |

* + **标记字段长度**计算：主存地址有28位（256MB=228B），其中6位为块内地址（26B=64B），3位为行标记号（23=8），其余28-6-3=19位为标记字段，总容量为8\*(1+19+512)=4256bit
  + **标记字段长度**另解：主存256MB，Cache64B，所以258M/64=219，长度为19

1. 直接映射方式中，主存按照块的大小划分，主存地址3200对应的字块号为3200B/64B=50,50mod8=2,故对应的Cahce行号为2
   * 二路组相联映射（一组有两个Cache）方式，实质上就是将两个Cache行合并，内部采用全相联方式，外部采用直接映射方式，50 mod 4 = 2，故对应的Cache行号为4或5
2. 直接映射方式中，28位主存地址可分为19位的主存标记位，3位的块号，6位的块内地址，即

0000 0001 0010 0011 010为主存标记位

001为块号，010110为块内地址

* + 首先根据块号，查Cache（即001号Cache行）中对应主存标记位，如果相同，再看Cache行中的装入有效位是否为1，如果是则有效，称访问命中，按块内地址010110读出Cache行对应的但还远送入CPU，完成访存；如果出现标记位不相等或有效位为0的情况，则是不命中，访问主存将数据取出送往CPU和Cache的对应块中，把主存的最高19位存入001行的Tag中，并将有效位置1

##### Cache中主存块的替换算法

**全相联映射**和**组相联映射**需要置换算法置换Cache行，**直接映射**时，不需要替换算法

1. 随机算法
2. 先进先出
3. 近期最少使用算法。LRU算法，依据程序访问的**局部性**原理选择近期内长久未访问过的存储行作为替换的行，平均命中率比FIFO高，是**堆栈类**算法。LRU算法对每行设置一个计数器，Cache每命中一次，命中行计数器清0，而其他各行均+1
4. 最不经常使用算法

##### Cache写策略

全写法（写直通法）

* + 当CPU对Cache写命中时，必须把数据同时写入Cache和主存。
  + 当某一块需要替换时，不必把这块写回，直接将新调入的块覆盖即可
  + 缺点：增加了访存次数，降低了Cache的效率
  + 写缓冲：改进措施，在Cache和主存之间加一个写缓冲。CPU同时写数据到Cache和写缓冲中，写缓冲再控制将内容写入主存。写缓存是一个FIFO队列

写回法

* + 当CPU对Cache写命中时，只修改Cache的内容，而不立即写入主存，只有当此块被换出时才写回内存
  + 采用此策略时，每个Cache行必须设置一个标志位（脏位），以反映此块是否被CPU修改过

全写法和写回法都对应于Cache**写命中**

如果Cache写**不命中**时

* + **非写分配法**只写入内存，不进行块调换。非写分配法通常与**全写法**合用
  + **写分配法**除了要写入内存外，还要将该块从内存调入Cahce，通常和**写回法**配合使用

##### 习题知识点

在高速缓存系统中，主存容量为12MB，Cache容量为400KB，则该存储系统的容量为12MB。因为Cache中内容为主存的副本

某计算机的Cache共有16块，采用二路组相联映射方式（每组2块）。每个主存块大小是32字节，按字节编址，主存129号单所在主存块应装入到的Cache组号是：

* + Cache组号从0开始编号
  + 129/4=32 32/8组=4组

某主存单元装入的**Cache地址**，包括行号和行内地址

* + 32位计算机的Cache容量为16KB，Cache行的大小是16B，采用直接映像方式，则主存地址0x1234E8F8的单元装入Cache地址是
    - 16K=214，则标记位长度32-14=18，块号+块内地址=14
    - 0001 0010 0011 0100 1110 1000 1111 1000

有效容量为128KB的Cache，每块16字节，采用**8路组相联**。字节地址为1234567H的单元调入该Cache，则其Tag应为

|  |  |  |
| --- | --- | --- |
| 主存字块标记14 | 组地址10 | 字块内地址4 |

* + 块大小16B，所以**块内地址字段**4位
  + Cache容量为128KB，采用8路组相联，共有  
    128KB/(16B\*8)=1024组，**组号字段**10位
  + 剩下的为**标记字段**28-4-10=14  
    0001 0010 0011 0100 0101 0110 0111  
    0001 0010 0011 01=0000 0100 1000 1101=048DH

主存容量为1MB，Cache容量为16KB，每字块有8个字，每字32位，采用**直接地址映像**方式，若主存地址为35301H，且CPU访问Cache命中，则在Cache的第 字块中（Cache起始字块为0）

* + 难点：每字块有8个字，每字32位，块内地址仍然按照8x32位=32B=25B，字段长5来记录，而不是按照23=8,字段长3来记录

|  |  |  |
| --- | --- | --- |
| 主存字块标记6 | Cache字块地址9 | 字块内地址5 |

* + 主存字块标记：1MB/16KB=64=26，字段长度6
  + Cache字块地址(块号)：16KB/32B=29，字段长9
  + 字块内地址段长度：5

若由高速缓存-主存-硬盘构成的三级存储体系，则CPU访问该存储系统时发送的地址为：主存物理地址  
A.高速缓存地址 B.虚拟地址 C.主存物理地址 C.磁盘地址

* + 只有在虚拟存储器中，CPU发出的才是虚拟地址，并没有指出是虚拟存储系统
  + CPU不可能直接访问磁盘

采用指令Cache与数据Cache分离的主要目的是减少指令流水线资源冲突

某计算机的主存地址位数为32位，按字节编址。假定数据Cache最多存放128个主存块，采用**四路组相联**方式，块大小64B，每块设置了1位有效位。采用一次性写回策略，为此每块设置1位“脏”位

|  |  |  |
| --- | --- | --- |
| 主存字块标记21 | 组地址5 | 字块内地址6 |

* + 字块内地址6位：64B=26B
  + 组地址（组号）5：128/4=32=25组
  + 主存字块标记：32-6-5=21
  + 该数据Cache总位数：  
    ((1+21+1)+64x8)x**128**

某计算机有8个数据Cache行，每个Cache行大小为64B，直接映射方式，int型数据用32位补码表示，程序编译时，i,j,sum都在寄存器中,数组a用行优先方式存放

1. int a[256] [256];
2. …
3. Int sum\_array(){
4. int i , j , sum = 0;
5. For( i = 0 ; i < 256 ; i++)
6. For( j = 0 ; j < 256 ; j++)
7. Sum += a[i][j];
8. Return sum;
9. }
   * 程序的数据访问命中率
     + 数据a的大小为256\*256\*4B=218B，占用218/64=212个主存块，按行优先存放，程序逐行访问数组a
     + 共需访问次数216次：256\*256=216
     + 未命中次数212次：每个字块的第一个数未命中
     + 命中率：(216-212)/ 216x100%=93.75%
   * 命中率另解：
     + 每个字块存放16个int型数据，除访问第一个不命中，随后的15个全部命中，访问全部字块都符合这一规律，且数组大小为字块大小的整数倍，故程序的命中率15/16=93.75%

一个Cache系统，字长16位，主存容量为16字x256块，Cache容量为16字x8块，采用全相联映射

* + 主存的**字**地址位数=8+4=12位
    - 8: 256块=28块
    - 4: 16字=24字

|  |  |
| --- | --- |
| 主存块号8位 | 块内地址4位 |

* + Cache的**字**地址位数=3+4=7位
    - 3: 8块=23块
    - 4: 16字=24字

|  |  |
| --- | --- |
| Cache块号3位 | 块内地址4位 |

* + 如果原先已经装入了5块信息，问字地址为338H所在的主存块将装入Cache块的块号及在Cache中的字地址
    - 全相联映射，原先已经装入Cache的5个块一次在0~4号块，因此要装入第5块，对应的Cache字地址为101 1000B （58）

Cache每行都有个标记项，无论直接映射还是组相联映射，所以Cache的标记阵列中需要有行数个标记项

某个Cache的容量大小为64KB，行长128B，且是四路相联Cache，主存使用32位地址

* + Cache共29行（64KB/128B=29），27组（29/4=27）
  + 块内地址位7：128B=27B
  + 标记位长度18：32-7-7=18

|  |  |  |
| --- | --- | --- |
| 主存字块标记18 | 组地址7 | 字块内地址7 |

* + Cache标记阵列需要29（29个Cache行）个标记项
  + 四路组相联Cache使用LRU算法的替换控制位为2位
  + 该Cache采用LRU替换算法，若当该Cache为**写直达**式Cache时，标记阵列总共需要
    - 无脏位，所以每个标记项位数是18+2+1=21位
    - 总大小为29x21=10752位
  + 写回式
    - 加一个脏位（一致性控制位）
    - 29x22=11264位

某32位计算机，CPU时钟周期是1.25ns。 Cache命中时CPI为4，Cache块大小是32B，主存采用8体交叉存储方式，每个体的存储字长为32位，存储周期为40ns；存储器总线宽度为32位，总线时钟周期是5ns，支持突发传输总线事务。每次读突发传送总线事务的过程包括：**送首地址和命令**、**存储器准备数据**、**传送数据**。每次突发传送32B，传送地址或32为数据均需要一个总线时钟周期（表示传数据也需要时间，不包括在存储周期内）

* + Cache缺失时，需要用1个读突发传送总线事务来完成一个主存块的读取
    - 因为Cache块32B，每次突发传送32B
  + 存储器总线完成一次读突发传送总线事务所需时间：
    - 一次读突发传送总线事务包括一次地址传送和32B数据传送：
    1. 用1个总线时钟周期传输地址
    2. 每隔40ns/8=5ns启动一个体工作（各进行1次存取），第一个体读数据花费40ns，之后数据存取与数据传输重叠
    3. 用8个总线时钟周期传送数据
    - 读突发传送总线事务时间：5ns+40ns+8x5ns=85ns
    - 存取时间并不包括把数据传到CPU的时间
  + 若程序执行过程中，共执行了100条指令，平均每条指令需要进行1.2次访存，Cache缺失率为5%，不考虑替换开销，则程序的CPU执行时间是：
    - 程序的CPU执行时间包括
      * 指令执行时间
      * Cache缺失时带来的额外开销
    - 命中Cache时无额外开销
    - 指令执行时间
      * 100x4x1.25ns=500ns
      * 为什么不乘0.95，因为不管Cache是否命中，最终都是要进行CPU执行
    - Cache缺失时带来的额外开销
      * 1.2x100x5%x85ns=510ns
    - 程序的CPU执行时间：500ns+510ns=1010ns

#### 虚拟存储器

主存和联机工作的辅存共同构成了虚拟存储器

对应用程序员，虚拟存储器是透明的

##### 虚拟存储器的基本概念

虚拟存储器将主存或辅存的地址空间统一编址

用户编程允许涉及到的地址称为**虚地址**或者**逻辑地址**，虚地址对应的存储空间称为虚拟空间或程序空间

实际的主存单元称为**实地址**或者**物理地址**，实地址对应的是主存地址空间

CPU使用虚地址时，由**辅助硬件**找出虚地址和实地址之间的对应关系

* + 若已在主存中，则通过**地址变换**，CPU可直接访问主存指示的实际单元
  + 若不在主存中，则把包含这个字的一页或一段调入主存**后**再由CPU访问
  + 如果主存已满，则采用替换算法置换主存中的一段或一页

##### 页式虚拟存储器

以**页**为基本单位的虚拟存储器称为页式虚拟存储器

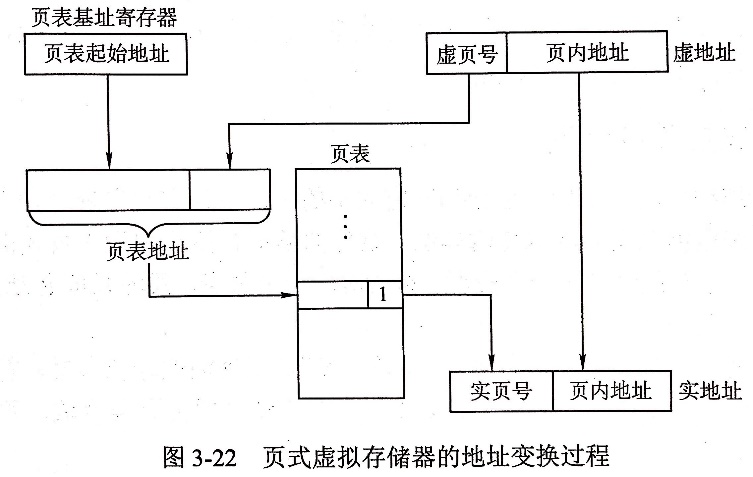
虚拟空间和主存空间都被划分为同样大小的页，主存的页称为**实页**，虚存的也称为**虚页**

虚拟地址分为两个字段：**虚页号**、**页内地址**

虚地址到实地址的变换是由**页表**来实现的。页表是一张存放在**主存**中的虚页号和实页号的对照表。页表一般长久地保存在内存里

**页表基址寄存器**存放当前运行程序的**页表**的起始地址，它和**虚页号**拼接成**页表项**地址，每一个页表项记录了与某个虚页对应的**虚页号**、**实页号**和**装入位**等信息。

* + 若装入位为1，表示该页面在主存中，将对应的**实页号**和虚地址中的**页内地址**拼接就得到了完成的实地址
  + 若装入位为0，表示该页面不再主存中，启动I/O系统，把该页从辅存调入主存**后**再供CPU使用



CPU访存时，先要查**页表**，为此需要访问主存一次。若不命中，还要进行页面替换和页表修改，则访问主存的次数就多了

##### 段式虚拟存储器

段式虚拟存储器的段是按**程序的逻辑结构**划分的

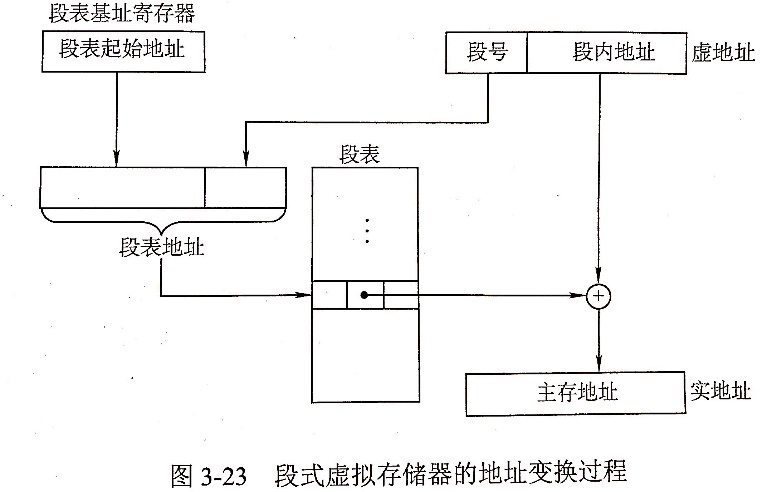
虚拟地址分为两个部分：**段号**和**段内地址**

虚拟地址到实地址之间的变换是通过**段表**来实现的

段表是程序的逻辑段和在主存中存放位置的对照表。

段表的每一行记录了与某个段对应的**段号**、**装入位**、**段起点**和**段长**等信息

CPU根据虚拟地址访存时，首先根据**段号**与**段表起始**地址拼接成对应的**段表行**，根据该段表行的**装入位**判断是否已调入主存（装入位1，则该段已在主存；0，该段不在主存）。若已调入主存，从段表读出该段在**主存**中的**起始位置**，与**段内地址**（偏移量）相加，得到对应的主存**实地址**

****

##### 段页式虚拟存储器

把程序按逻辑结构分段，每段再划分为固定大小的页，主存空间也划分成大小相等的页，程序对主存的调入、调出仍以**页**为基本传送单位

虚地址分为**段号**、**段内页号**、**页内地址**

CPU根据虚地址访存时，

* + 首先根据**段号**得到**段表地址**;
  + 然后从**段表**中取出该段的**页表起始地址**，与虚地址**段内页号**合成，得到**页表地址**
  + 最后从**页表**中取出**实页号**，与**页内地址**拼接形成主存**实地址**
  + 优点：兼具页式和段式虚拟存储器的优点，可以按段实现共享和保护
  + 缺点：在**地址变换**过程中需要**两次查表**，系统开销大

##### 快表TLB

虚拟存储器中，必须先访问一次主存去查页表，再访问主存才能取得数据，相当于访存速度降低了一半。在段页式虚拟存储器中，纪要查段表也要查页表

局部性原理，把经常访问的页存放在高速缓冲器组成的**快表**（TLB）中。相应的，把存放在主存中的页表称为**慢表**（Page）。

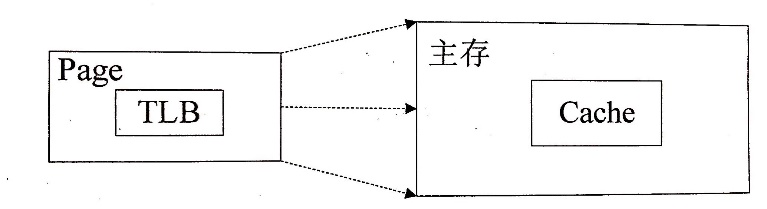
* + 快表只是慢表的一个副本

查找时，快表和慢表**同时**进行，快表根据内容指定地址，一般使用**相联存储器**。若快表中有此逻辑页号，则能很快地找到物理页号，送入**实主存地址寄存器**，并使慢表的查找作废

TLB命中则Page一定命中

在同时具有虚拟页式存储器（有TLB）和Cache的系统中，访问顺序为：TLB→页表→Cache→主存

* + CPU发出访存命令（逻辑地址），先查找TLB和Page,将逻辑地址转化为物理地址
  + 再查找相应的Cache块（与主存查找**并行**进行）
  + 若Cache命中，则说明所需页面已调入内存，Page必然命中，但TLB不一定命中；
  + 若Cache不命中，并**不能说明**所需页面未调入主存，和TLB和Page命中与否**没有联系**
  + 若**Page**不命中，说明所需页面未调入内存，当然Cahce和主存不会命中，需要执行调页策略



##### 虚拟存储器与Cache比较

相同

* + 目标都是为了提高性能，都有容量、速度、价格的梯度
  + 都把数据划分为小信息块，并作为基本的传输单位，虚存系统的信息块更大
  + 都有地址的映射、替换算法、更新策略等问题
  + 根据程序访问的局部性原理应用“快速缓存思想”，将相对活跃的数据放在相对高速的部件中

不同

* + Cache主要解决系统速度，而虚拟存储器确实为了解决主存容量
  + Cache全由硬件实现，是硬件存储器，对所有程序员透明；而虚拟存储器由OS和硬件共同实现，是逻辑上的存储器，对系统程序员不透明，但对应用程序员透明
  + 对于不命中性能影响，因为CPU的速度约为Cache的10倍，主存的速度为硬盘的100倍，所以虚拟存储器系统不命中对系统影响更大
  + CPU与Cache和主存都建立了直接访问的通路，而辅存与CPU没有直接通路。也就是说在Cache不命中时主存能和CPU直接通信，同时将数据调入Cache；而虚拟存储器系统不命中时，只能先由硬盘调入主存，而不能直接和CPU通信

##### 习题知识点

虚拟存储管理系统的基础是程序访问的局部性原理，此理论的基本含义是：在程序的执行过程中，程序对主存的访问是不均匀的

* + 局部性原理的含义就是一个程序执行过程中，其大部分情况下是**顺序执行**的，某条指令或数据使用后，在最近一段时间内较大可能再次被访问（时间局部性）；某条指令或数据使用后，其临近的指令或数据可能在近期被使用（空间局部性）。
  + 虚拟存储管理系统中，程序只能访问主存获得指令和数据
  + 空间局部性、时间局部性和代码的顺序执行都只是局部性原理的一方面

段页式先按照逻辑分段，然后分页，以**页**为单位和主存交互

虚拟存储器中，地址映射由操作系统完成，但需要一部分硬件基础的支持，如快表（TLB）、地址映射系统等

段式虚拟存储器是按照程序的逻辑性来设计的，具有易于实现程序的编译、管理和保护，也便于多到程序共享的优点

某计算机存储器按字节编址，虚拟（逻辑）地址空间大小为16MB，主存（物理）地址空间大小为1MB，页面大小为4KB；Cache采用直接映射方式，共8行；主存与Cache之间的交换的块大小为32B。系统运行到某时刻，页表的部分内容和Cache的部分内容分别为：

| 虚页号 | 有效位 | 页框号 | … |
| --- | --- | --- | --- |
| 0 | 1 | 06 | … |
| 1 | 1 | 04 | … |
| 2 | 1 | 15 | … |
| 3 | 1 | 02 | … |
| 4 | 0 | - | … |
| 5 | 1 | 2B | … |
| 6 | 0 | - | … |
| 7 | 1 | 32 | … |

| 行号 | 有效位 | 标记 | … |
| --- | --- | --- | --- |
| 0 | 1 | 020 | … |
| 1 | 0 | - | … |
| 2 | 1 | 01D | … |
| 3 | 1 | 105 | … |
| 4 | 1 | 064 | … |
| 5 | 1 | 14D | … |
| 6 | 0 | - | … |
| 7 | 1 | 27A | … |

* + **虚拟地址**共24位（224B=16MB），高12位是**虚页号**（页大小4KB=212，24-12=12）

|  |  |
| --- | --- |
| 12 | 12 |

* + **物理地址**共20位（220B=1MB）,高8位是**页框号**（**物理页号**）

|  |  |
| --- | --- |
| 8 | 12 |

* + 使用物理地址访问Cache时，物理地址划分成

|  |  |  |
| --- | --- | --- |
| 主存字块标记12 | Cache字块标记3 | 字块内地址5 |

* + - 字块内地址：块大小32B=25B
    - Cache共8行：23=8
    - 主存字块标记：20-5-3=12
  + 虚拟地址001C60H所在的页面是否在主存？若在主存，则该虚拟地址对应的物理地址是什么？访问该地址时是否Cache命中？
    - 虚拟地址001C60H，001H为虚页号，查001H号的页表项，有效位为1，页框号为04H，拼接成物理地址04C60H
    - 物理地址04C60H：0000 0100 1100 **011**0 0000B，主存块只能映射到001B号Cache，该行有效位1，但标记105H != 04CH，故未命中
  + 假定为该机配置一个四路组相联的TLB，共可放8个页表项，若其当前内容（十六进制）如下，则此时虚拟地址024BACH所在的页面是否存在主存中

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 组号 | 有效位 | 标记 | 页框号 |  | 有效位 | 标记 | 页框号 |  | 有效位 | 标记 | 页框号 |  | 有效位 | 标记 | 页框号 |
| 0 | 0 | - | - |  | 1 | 001 | 15 |  | 0 | - | - |  | 1 | 012 | 1F |
| 1 | 1 | 013 | 2D |  | 0 | - | - |  | 1 | 008 | 7E |  | 0 | - | - |

* + - 虚拟地址低12位为页内地址，前12位为虚拟页号
    - 由于TLB采用4路组相联，故TLB被分为8/4=2组，因此虚页号中，高11位为TLB标记、最低1位为TLB组号

|  |  |  |
| --- | --- | --- |
| TLB标记11 | TLB组号1 | 页内地址12 |

* + - 024BACH=0000 0010 0100 1011 1010 1100B
    - TLB组号0，TLB标记0000 0010 010B=024H命中，则虚拟地址024BACH所在的页面在主存中

#### 小结

存取周期和存取时间的区别

* + 存取时间仅为完成一次操作的时间，而存取周期不仅包含操作时间，还包含操作后线路的恢复时间
    - 存取周期=存取时间+恢复时间

虚拟存储器中，页面设置的大一点号还是小一点好

* + 页面太小时，平均页内剩余空间小，可节省存储空间，但会让页表变大，而且页面太小时不能充分利用访存的空间局部性提高命中率
  + 页面太大时，可减少页表空间，但平均页内剩余空间较大，浪费较多存储空间，页面太大还使页面调入调出的时间较长

存取周期为执行一次读操作**或**写操作的时间，分为读出时间和写入时间

* + 读出时间为从主存接收到有效地址开始到数据稳定为止的时间
  + 写入时间是从主存接收到有效地址开始到数据被写入单元为止的时间

Cache行的大小和命中率之间的关系

* + 行的长度较大，可充分利用程序访问的空间局部性原理，使一个较大的局部空间被一起调入到Cache中，因而可以增加命中机会。但行也不能太大
    - 行长大使失效损失变大。如果未命中，需要更多时间从主存读块
    - 行长太大，则Cache项数变少，因而，命中的可能性变小

发生取指令Cache缺失的处理过程：

* + 程序计数器恢复当前指令的值
  + 对主存进行读的操作
  + 将读入的指令写入Cache中更改有效位和标志位
  + **重新执行当前指令**

指令和数据可以分别存储在不同的Cache中，这种结构称为哈佛Cache，其特点是允许CPU在同一个Cache存储周期内同时提取指令和数据，由于指令执行过程中取指和取数据都有可能访问Cache，这一特性可保证不同的指令同时访存

### 指令系统

#### 指令格式

指令（又称机器指令）是计算机运行的最小功能单位。

指令系统是计算机的主要属性，位于硬件和软件的交界面上

##### 指令的基本格式

指令长度和机器字长没有固定关系，可以等于机器字长，也可以小于或大于

通常，把指令长度=机器字长的指令称为**单字长指令**；把指令长度=0.5\*机器字长的指令称为**半字长指令**；把指令长度=2\*机器字长的指令称为**双字长指令**

定长指令字结构：所有指令的长度都是相等的

变长指令字结构：各种指令的长度随指令功能而异

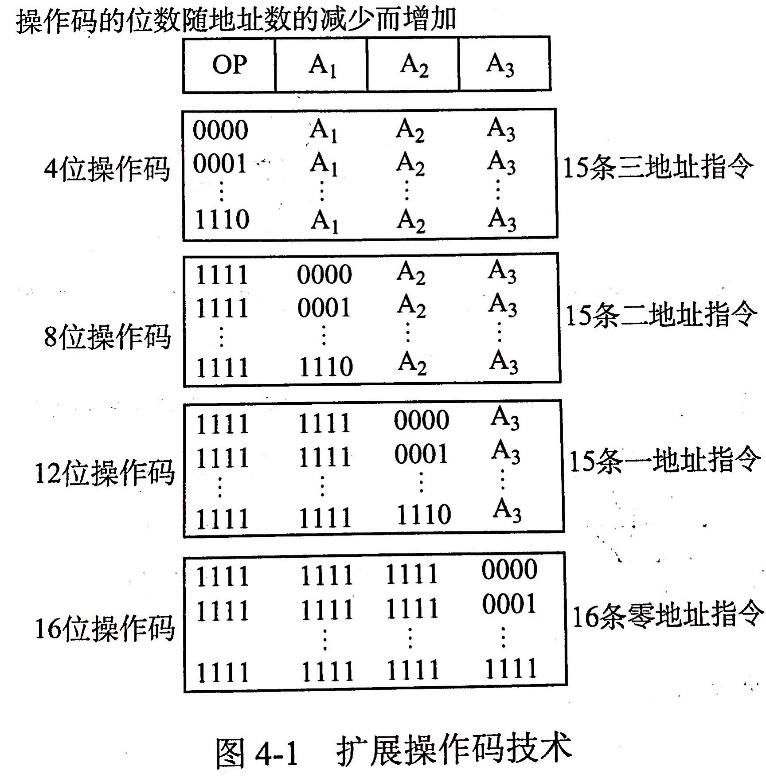
三地址指令，若地址字段均为主存地址，则完成一条三地址需要4次访问存储器：取指1次，取两个操作数2次，存放结果1次

##### 定长操作码指令格式

指令字的最高位部分分配固定的若干位表示操作码。

一般n位操作码字段的指令系统最大能够表示2n条指令

##### 扩展操作码指令格式

常见的变长操作码方法是扩展操作码，使操作码的长度随地址码的减少而增加

* + 图中指令字长16位，其中4位为基本操作码字段OP，另有3个4位长的地址字段A1、A2和A3。
  + 4位基本操作码若全部用于三地址指令，则有16条
  + 图中所示的三地址指令为15条，1111留作扩展操作码之用
  + 二地址指令为15条，1111 1111留作扩展操作码之用
  + 一地址指令为15条，1111 1111 1111留作扩展操作码之用
  + 零地址指令为16条

设计扩展操作码指令格式时，必须注意：

* + 不允许短码是长码的前缀
  + 各指令的操作码不能重复

通常对使用频率高的指令，分配较短的操作码；对频率低的指令，分配较长的操作码，从而尽可能减少指令译码和分析的时间

##### 习题知识点

指令系统是一台机器硬件能执行的指令全体

任何程序运行前都要先转化成机器语言程序

指令系统是计算机软件、硬件的界面

指令系统是计算机硬件的语言系统，和机器语言有关

程序控制类指令用于改变程序执行的顺序，并使程序具有测试、分析、判断和循环执行的能力

程序控制类指令主要包括

* + 无条件转移
  + 有条件转移
  + 子程序调用
  + 返回指令
  + 循环指令

中断隐指令是由硬件实现的，并不是指令系统中存在的指令，更不可能属于程序控制类指令

特权指令是指仅用于操作系统或其他系统软件的指令。不让应用程序使用

零地址的运算类指令又称堆栈运算指令，参与的两个操作数来自栈顶和次栈顶单元

堆栈指令的**访存次数**，取决于采用的是**软堆栈**还是**硬堆栈**

* + 软堆栈（堆栈区由内存实现），对于双目运算，需要访问4次内存：取指、取源数1、取源数2、存结果
  + 硬堆栈（堆栈区由寄存器实现），则只需在取指令时访问一次内存

单地址指令可能有一个操作数，也可能有两个操作数

指令的地址个数与指令的长度是否固定没有必然联系

单地址指令中只有一个地址码，在完成两个操作数的算数运算时，一个操作数由地址码给出，另一个操作数通常存放在累加寄存器（ACC）中，属于隐含寻址

设机器字长32位，一个容量为16MB的存储器，CPU按半字寻址，其寻址单元数是223

* + 16M/2=8M 8M=223

在指令格式中，采用扩展操作码设计方案的目的是：保持指令字长度不变而增加指令的数量

一个计算机系统采用32位单字长指令，地址码为12位，如果定义了250条二地址指令，那么还可以有\_\_条单地址指令

* + 地址码12位，则二地址指令的操作码长度为32-12-12=8位，已经定义了250条二地址指令，  
    28-250=6，即可以设计出单地址指令6x212=24K条

一处理器中共有32个寄存器，使用16位立即数，指令系统结构中共有142条指令。在某个程序中，20%的指令带有一个输入寄存器和一个输出寄存器；30%的指令带有两个输入寄存器和一个输出寄存器；25%的指令带有一个输出寄存器、一个输入寄存器、一个立即数寄存器；其余的25%指令带一个立即数输入寄存器和一个输出寄存器

* + 对于以上指令，需多少位？假定指令系统结构要求所有指令长度必须是8的整数倍
    - 142条指令，故至少需要8位才能确定各条指令的操作码（28=256）；有32个寄存器，需要5位来对寄存器ID进行编码；每个立即数需要16位
    - 8+5+5=18位，对齐到24位
    - 其余略
  + 采用定长编码时，长度应为最长的那个指令编码长度

假设指令字长为16位，操作码的地址码为6位，指令有零地址、一地址、二地址3中格式

* + 设操作码固定，若零地址指令有M种，一地址指令有N种，则二地址指令最多有几种
    - 操作数地址码6位，则二地址指令中操作码的位数为16-6-6=4，这4种操作码可有16种操作。由于操作码固定，剩下的二地址最多有16-M-N
  + 采用扩展操作码技术，二地址指令最多有几种
    - 16-6-6=4
    - 24-1=15种
  + 采用扩展操作码技术，若二地址有P条，零地址指令有Q条，则一地址指令最多有几种
    - 二地址、一地址、零地址的操作码长度分别为4、10、16位
    - 二地址指令操作码每减少一个，就可以多构成26条一地址指令操作码；
    - 一地址指令操作码每减少一个，就可以多构成26条零地址指令操作码；
    - 设一地址指令有R条，则一地址指令最多有  
      (24-P)x26条
    - 零地址指令最多有[(24-P)x26-R]x26条
    - 根据题中给出零地址指令为Q条，即  
       Q=[(24-P)x26-R]x26则 R=(24-P)x26-Qx2-6

在一个36位长的指令系统中，设计一个扩展操作码，使之能表示下列指令

* + 7条具有两个15位地址和一个3位地址的指令
  + 500条具有一个15位地址和一个3位地址的指令
  + 50条无地址指令

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 15 | 15 | 3 |
| OP | Addr1 | Addr2 | Addr3 |

* + OP: 000~110

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 15 | 15 | 3 |
| OP1 | OP2 | Addr1 | Addr2 |

* + OP1: 111   
    OP2: 000000000000000~000000111110011 (499)

|  |  |  |
| --- | --- | --- |
| 3 | 15 | 18 |
| OP1 | OP2 | 18个0 |

* + OP1: 111   
    OP2: 000000111110100~000001000100101 (549)

P151 4题需要后面知识点，暂时略过

#### 指令寻址方式

寻址方式分为指令寻址和数据寻址

##### 指令寻址和数据寻址

指令寻址

* + 顺序寻址方式：(PC)+1
  + 跳跃寻址通过转移类指令实现。
    - 由本条指令给出下调指令地址的计算方式。
    - 能否跳跃可能受到状态寄存器和操作数的控制
    - 跳跃到的地址分为绝对地址（由标记符直接得到）和相对地址（对于当前指令地址的偏移量）
    - 跳跃的结果是修改PC值，所以下一条指令仍然是**PC**给出

数据寻址

|  |  |  |
| --- | --- | --- |
| 操作码 | 寻址特征 | 形式地址A |

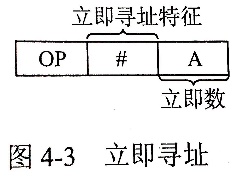
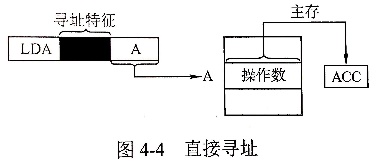
##### 常见的数据寻址方式

| 寻址方式 | 有效地址 | 访存次数 |
| --- | --- | --- |
| 隐含寻址 | 程序指定 | 0 |
| 立即（数）寻址 | A即是操作数 | 0 |
| 直接寻址 | EA=A | 1 |
| 一次间接寻址 | EA=(A) | 2 |
| 寄存器寻址 | EA=Ri | 0 |
| 寄存器间接一次寻址 | EA=( Ri) | 1 |
| 相对寻址 | EA=(PC)+A | 1 |
| 基址变址 | EA=(BR)+A | 1 |
| 变址寻址 | EA+(IX)+A | 1 |

隐含寻址

* + 在指令中隐含着操作数的地址
  + 

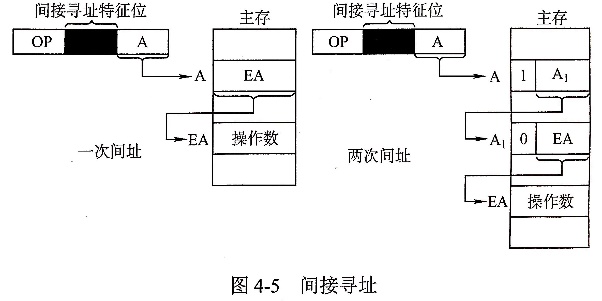
立即（数）寻址

* + 地址字段给出的是操作数本身，称为立即数。
  + 补码
  + 不访存，指令执行时间最短；但A的位数限制了立即数的范围
  +  

直接寻址

* + EA=A
  + 缺点：A的位数决定了该指令操作数的寻址范围，操作数的地址不易修改

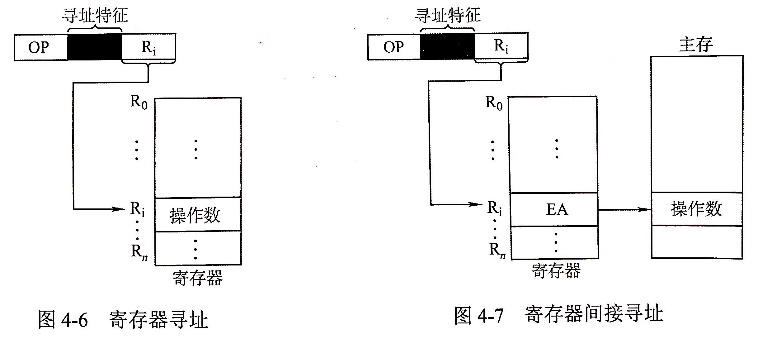
间接寻址



* + EA=(A)
  + 主存字的第一位为1时，表示取出的仍不是操作数的地址；第一位为0，表示取出的是操作数地址
  + 优点：可扩大寻址范围（有效地址EA的尾数大于A的位数），便于编制程序（用间接寻址可以方便的完成**子程序返回**
  + 缺点：指令在**执行**阶段需多次访存，访问速度慢。一般扩大寻址范围通常用寄存器间接寻址

寄存器寻址

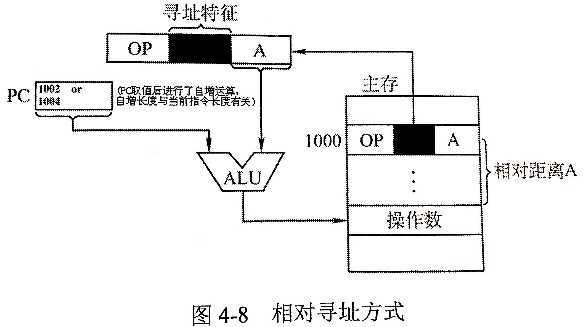
* + EA=Ri
  + 指令字中直接给出操作数所在的寄存器编号，操作数就在Ri所指的寄存器内
  + 优点：执行阶段不访存，因寄存器数量少，对应地址码长度小，使得指令字短缺不访存，执行速度快，支持**向量/矩阵**运算



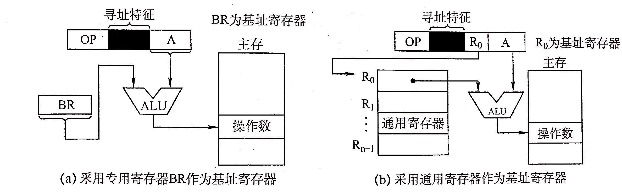
寄存器间接寻址

* + EA=( Ri)

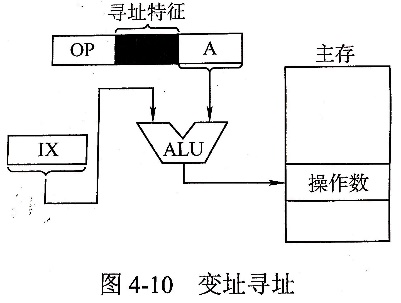
相对寻址

* + 
  + EA=(PC)+A
  + A是相对于当前指令地址的位移量，可正可负，补码
  + A的位数决定操作数的寻址范围
  + 相对寻址广泛应用于**转移指令**
  + 对于转移指令**JMP** A，当CPU从存储器中取出一个字节时，会**自动执行(PC)+1→PC**
    - 若执行的转移指令的地址为X，占2个字节，在取出该指令后，PC的值会增2，即(PC)=X+2，这样在执行完该指令后，会自动跳转到X+2+A的地址继续执行

基址寻址

* + 
  + EA=(BR)+A
  + 基址寄存器既可以采用专用寄存器，也可以用通用寄存器
  + 基址寄存器是面向操作系统的，内容由操作系统或管理程序确定。
  + 程序执行过程中，基址寄存器的内容不变（作为基地址），形式地址可变（作为偏移量）
  + 当采用通用寄存器作为基址寄存器时，可由**用户**决定那个寄存器为基址寄存器，但**内容**仍然由**操作系统**确定
  + 基址寻址的优点是可以扩大寻址范围（基址寄存器的位数大于A的位数）；用户不必考虑自己的程序于主存的哪一空间区域，故有利于多道程序设计，以及可用于编制**浮动程序**

变址寻址

* + 
  + EA+(IX)+A
  + IX为变址寄存器（专用），也可以用通用寄存器为编址寄存器
  + 变址寄存器是面向**用户**的，编址寄存器的内容可**由用户改变**（作为偏移量），形式地址A不变（作为基地址）
  + 优点：
    - 扩大可寻址范围
    - 在数组处理过程中，可设定A为数组的首地址，不断改变IX的内容，便可很容易形成数组中任一数据的地址，特别适合编制**循环程序**
  + 变址寻址和基址寻址的区别
    - * 基址寻址主要用于为多道程序或数据分配存储空间，故基址寄存器的内容通常由操作系统或管理程序确定，在程序执行过程中其值不可变，而指令字中的A是可变的
      * 变址寻址主要用于处理**数组**问题，在变址寻址中，变址寄存器的内容由用户设定，在程序执行过程中其值可变，而指令字中的A是不可变的

堆栈寻址

* + 堆栈是存储器（或专用寄存器组）中一块特定的按后进先出LOFO原则管理的存储区
  + 该存储区中被读/写单元的地址是用一个特定的寄存器给出的，该寄存器称为堆栈指针（SP）
  + 硬堆栈：寄存器堆栈
  + 软堆栈：从主存中划出一段区域来做堆栈
  + 在采用堆栈结构的计算机系统中，大部分指令表面上都表现为无操作数指令的形式，因为操作数地址都隐含使用了SP。通常，在读/写堆栈中的一个单元的前后都伴有自动完成对SP内容的增量或减量操作

##### 习题知识点

采用不同寻址方式的目的是为了缩短指令字长，扩大寻址空间，提高编程的灵活性，这也提高了指令译码的复杂度。实现程序控制是靠转移指令实现的，而不是寻址方式

无条件转移指令是程序转移到新的地址后继续执行，因此必须给出下一条指令的地址，并送入PC

为了缩短指令中某个地址段的位数，有效的方法是采取寄存器寻址

简化地址结构的基本方法是尽量采用隐地址

设指令中的地址码为A，变址寄存器为X，，则**变址间址寻址**方式：EA=((X)+A)

基址寻址：**多道**程序设计和编制**浮动**程序

变址寻址：**数组**问题和编制**循环**程序

相对寻址方式中，指令所提供的相对地址实质上是一种以**下条**指令在内存中首地址为基准位置的偏移量

实现**程序浮动**：相对寻址，基址寻址

相对寻址和基址寻址都常用于：**浮动**程序，**多道**程序设计

**跳跃**寻址通过转移类指令（如相对寻址）来实现，可用来实现程序的**条件或无条件转移**

某机器有一个标志寄存器，其中有进位/借位标志CF、零标志ZF、符号标志SF和溢出标志OF，条件转移指令bgt（无符号整数比较大于时转移）的转移条件是

* + 若A>B，则A-B肯定无进位/借位，也不为0，故而CF和ZF均为0.
  + 用不到SF,OF

某计算机有16个通用寄存器，采用32位定长指令字，操作码字段（含寻址方式）为8位，Store指令的源操作数和目的操作数分别采用寄存器直接寻址和基址寻址方式。若基址寄存器可使用任一通用寄存器，且偏移量用补码表示，则Store指令中偏移量的取指范围是：

* + 32位定长指令字，操作码8位，两个地址码共24位
  + 寻址寄存器需4位
  + 源操作数的寄存器直接寻址用掉4位，目的操作数采用基址寻址也要制定一个寄存器，同样用掉4位
  + 留给偏移地址的位数24-4-4=16
  + 16位补码范围：-32768~+32767

算术与逻辑运算指令用于完成对一个（如自增、取反）或两个数据的算术运算或逻辑运算

移位操作用于把**一个操作数**左移或右移一位或多位

转移指令、子程序调用与返回指令用于解决变动程序中指令执行次序的需求，而不是数据调用次序的需求

某计算机指令系统采用**定长操作码**，**变长指令码**格式

* + 通常采用寄存器寻址方式和寄存器间接寻址方式的**指令码长度最短**。
  + 立即寻址方式、直接寻址方式和间接寻址方式的**指令码长度最长**
  + 寄存器寻址方式**执行速度最快**
  + 间接寻址方式**执行速度最慢**
  + 若指令系统采用**定长指令码**格式，所有指令（包括采用立即寻址方式的指令）所包含的二进制位数均相同，则**立即寻址方式**执行速度最快
  + 若指令系统采用变长指令码格式，由于要表示一定范围内的立即数，包含立即数的指令通常需要较多的二进制位，取指令时，可能需要不止一次地读内存来完成取指令。因此，采用变长指令码格式时，寄存器寻址方式执行速度最快

某机字长16位，形式地址A的位数为8位，下列各种寻址方式的寻址范围

|  |  |  |
| --- | --- | --- |
| 寻址方式 | EA计算公式 | 寻址范围 |
| 直接寻址 | EA = A | 28 |
| 间接寻址 | EA = (A) | 216 (机器字长16位) |
| 变址寻址 | EA = (IX) + A | 216 (IX 16位) |
| 相对寻址 | EA = (PC) + A | 28 (PC附近的28) |

P164 8题需要后面知识点，暂时不看

#### CISC和RISC的基本概念

|  |  |  |
| --- | --- | --- |
|  | CISC | RISC |
| 指令系统 | 复杂，庞大 | 简单，精简 |
| 指令数目 | 一般大于200条 | 一般小于100条 |
| 指令字长 | 不固定 | 定长 |
| 可访存指令 | 不加限制 | 只有Load/Store |
| 各种指令执行时间 | 相差较大 | 绝大多数在一个周期内完成 |
| 各种指令频度 | 相差很大 | 都比较常用 |
| 通用寄存器数量 | 较少 | 多 |
| 目标代码 | 难以用优化编译生成高效的目标代码程序 | 采用优化的编译程序，生成代码较为高效 |
| 控制方式 | 绝大多数为微程序控制 | 绝大多数为组合逻辑控制（硬布线） |
| 指令流水线 | 可以通过一定方式实现 | 必须实现 |

RISC更能充分利用VLSI芯片的面积。CISC控制存储器在CPU芯片内所占的面积为50%以上，RISC的硬布线逻辑只占CPU芯片面积的10%左右

##### 习题知识点

RISC一定采用流水技术，但采用流水技术的也可能是CISC

指令功能尽可能强是CISC的特点

RISC的指令基本上都被CISC指令包含，但RISC并不是为了实现兼容才挑选的，而是选择最常用的简短的指令

RISC的指令功能简单，通过简单指令的组合来实现复杂指令的功能

#### 本章小结

边界对齐，边界不对齐

* + 设存储字长32位，可按字节、半字和字寻址。对于机器字长32位的计算机，数据对齐方式，**半字地址**一定是2的整数倍，**字地址**一定是4的整数倍，这样无论所取的数据是字，半字还是字节，都可以一次访存取出。当所存储的数据不满足要求时，通过填充空白字节满足要求
  + 边界不对齐：

|  |  |  |  |
| --- | --- | --- | --- |
| 字节1 | 字节2 | 字节3 | 半字1-1 |
| 半字1-2 | 半字2 | | 半字3-1 |
| 半字3-2 | 字1-1 | | |
| 字1-2 |  |  |  |

边界对齐

|  |  |  |  |
| --- | --- | --- | --- |
| 字节1 | 字节2 | 字节3 | 填充 |
| 半字1 | | 半字2 | |
| 半字3 | | 填充 | |
| 字1 | | | |

RISC如ARM采用边界对齐方式，CISC如x86对齐和不对齐都支持。因为对齐方式取指令时间相同，故而能适应指令流水线

现代计算机都是采用字节编址的方式，即一个内存单元只能存放一个字节的信息

* + 大端方式：指令中给出的地址是操作数最高有效字节(MSB)所在的地址
  + 小端方式：指令中给出的地址是操作数最低有效字节(LSB)所在的地址

### 中央处理器

#### CPU的功能和基本结构

##### CPU的功能

指令控制：完成取指令、分析指令和执行指令的操作，即程序的顺序控制

操作控制

时间控制：时间控制要为每条指令按时间顺序提供应有的控制信号

数据加工：对数据进行算数和逻辑运算

中断处理：异常和特殊请求

##### CPU的基本结构

运算器

* + **算术逻辑单元(ALU)**
  + **暂存寄存器**：用于暂存从主存读来的数据，这个数据不能存放在通用寄存器，否则会破坏其原有的内容。暂存寄存器对应用程序员透明
  + **累加寄存器(ACC)**：他是一个通用寄存器，用于暂存ALU运算的结果信息，可以作为加法运算的一个输入端
  + **通用寄存器组**：AX,BX,CX,DX,SP，用于存放操作数（包括源操作数、目的操作数及中间结果）和各种地址信息。SP是堆栈指针，用于只是栈顶的地址
  + **程序状态字寄存器(PSW)**:保留由算数逻辑运算指令或测试指令的结果而建立的各种状态信息，如溢出标志OP，符号标志SF，零标志ZF，进位标志CF。PSW中的这些参与并决定微操作的形成
  + **移位器**：对操作数或运算结果进行移位运算
  + **计数器(CT)**：控制乘除运算的操作步数

控制器

* + 控制器的基本功能就是执行指令，每条指令的执行是由控制器发出的一组**微操作**实现的
  + 控制器有**硬布线**控制器和**微程序**控制器
  + 控制器组成：
    - **程序计数器(PC)**
    - **指令寄存器(IR)**：用于保存当前正在执行的那条指令
    - **指令译码器**：仅对操作码字段进行译码，向控制器提供特定的操作信号
    - **存储器地址寄存器(MAR)**：用于存放所要访问的主存单元的地址
    - **存储器数据寄存器(MDR)**：用于存放向主存写入的信息或从主存中读出的信息
    - **时序系统/电路**：用于产生各种时序信号，他们都是由统一时钟(CLOCK)分频得到
    - **微操作信号发生器**：根据IR的内容（**指令**）、PSW（**状态信息**）及**时序信号**，产生控制整个计算机系统所需的各种控制信号，其结构有组合逻辑型和存储逻辑型两种
  + 控制器的工作原理是根据**指令操作码**、**指令的执行步骤（微命令序列）**和**条件信号**来形成当前计算机各部件要用到的控制信号

用户可见寄存器：

* + 通用寄存器组
  + 程序状态字寄存器

用户不可见寄存器：

* + MAR,MDR,IR

##### 习题知识点

汇编程序员可见PC

条件转移指令执行时所依据的条件来自：**标志寄存器**。条件转移指令执行时，需对标志寄存器的内容进行测试，判断是否满足转移条件

N位CPU，这里的N是指**数据总线线数**。数据总线的位数与处理器位数相同，也就表示了CPU一次能够处理的数据的位数，即CPU的位数

PC的值会根据CPU在执行指令的过程中修改（确切说是取指周期末），或自增或转移到程序的某处

转移指令时，需要判别转移是否成功，若成功则PC修改为转移目标指令的地址，否则下一条指令的地址仍然是为PC自增后的地址

在一条无条件跳转指令的指令周期内，PC的值被修改2次：取指周期结束后，PC自增；执行周期中，PC值修改为要跳转到的地址

程序计数器的位数取决于存储器的容量

指令寄存器的位数取决于指令字长

CPU中通用寄存器的位数取决于机器字长

CPU中的通用寄存器可以存放数据和地址

控制器的全部功能是从主存中取出指令、分析指令并产生有关的操作控制信号

指令译码是对指令的操作码字段进行译码

CPU中不包括地址译码器，**地址译码器**是主存等存储器的组成部分，作用是根据输入的地址码唯一选定一个存储单元

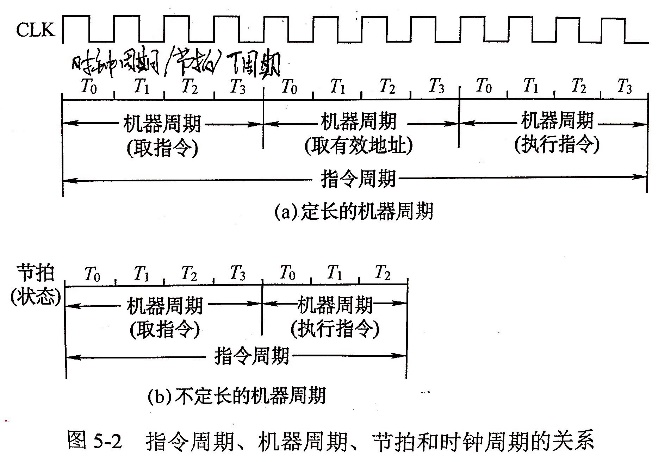
CPU中决定指令执行顺序的是程序计数器

间址周期结束时，CPU内寄存器MDR中的内容是操作数地址

#### 指令执行过程

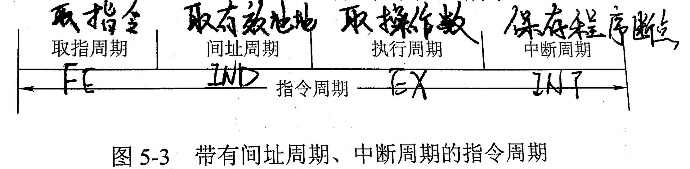
##### 指令周期

指令周期常常用若干机器周期来表示，一个机器周期又包含若干时钟周期

每个指令周期内机器周期可以不等，每个机器周期内的节拍数也可以不等

对于无条件转移指令JMP X，在执行时不需要访问主存，只包含取指阶段（包括取指和分析）和执行阶段，所以其指令周期仅包括取指周期和执行周期

对于间接寻址的指令，还需要包括间址周期，介于取指周期和执行周期之间。

当CPU采用中断方式实现主机和I/O设备交换信息时，CPU在每条指令结束前，都要发中断查询信号，如果有中断请求，CPU则进入中断相应阶段，又称中断周期

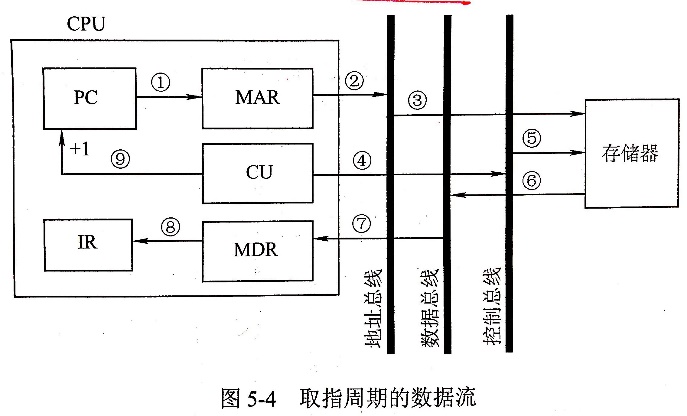
上述4个工作周期都有CPU访存操作

中断周期中进栈操作是将SP减1，和传统意义上的进栈操作相反，原因是计算机的堆栈中都是向低地址增加，所以进栈操作是减1，而不是加1

##### 指令周期的数据流

取指周期

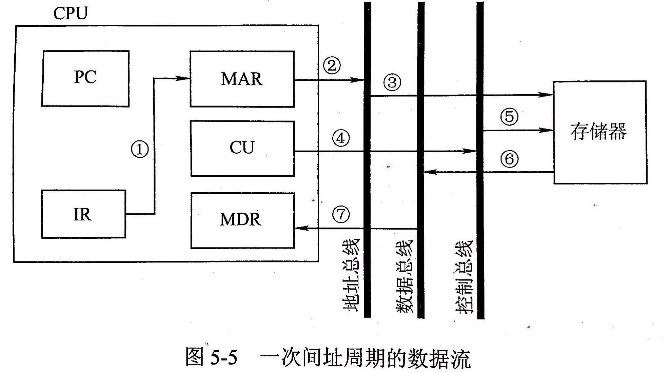
* + 根据PC的内容，从主存中取出指令代码并存放在IR



* + 数据流：
    1. PC ① MAR ② 地址总线 ③ 主存
    2. CU发出控制信号 ④ 控制总线 ⑤ 主存
    3. 主存 ⑥ 数据总线 ⑦ MDR ⑧ IR(存放指令)
    4. CU发出读命令 ⑨ PC内容加1

间址周期

* + 取操作数的有效地址



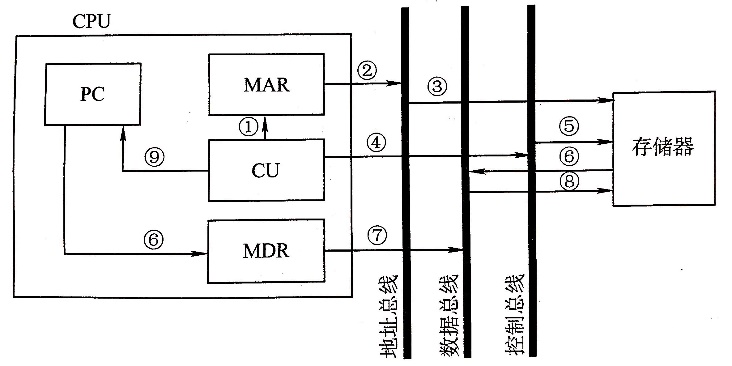
* + 数据流：
    1. Ad(IR)(或MDR) ① MAR ② 地址总线 ③ 主存
    2. CU发出读命令 ④ 控制总线 ⑤ 主存
    3. 主存 ⑥ 数据总线 ⑦ MDR（存放有效地址）
    - 其中Ad(IR)表示取出IR中存放的指令字的地址字段

执行周期

* + 没有统一的数据流向

中断周期

* + 处理中断请求。
  + 假设程序断点存入堆栈中，并用SP只是栈顶地址，而且进栈操作是先修改栈顶指针，后存入数据



* + 数据流
    - CU控制将SP减1，SP ① MAR ② 地址总线 ③ 主存
    - CU发出写命令 ④ 控制总线 ⑤ 主存
    - PC ⑥ MDR ⑦ 数据总线 ⑧ 主存（程序断点存入主存）
    - CU（中断程序的入口地址） ⑨ PC

##### 指令执行方案

单指令周期

* + 对所有指令都选用相同的执行时间来完成
  + 指令周期取决于执行时间最长的指令的执行时间
  + 指令之间串行

多指令周期

* + 指令之间串行
  + 可选用不同个数的时钟周期来完成不同指令的执行过程

流水线方案

##### 习题知识点

冯诺依曼计算机根据指令周期的不同阶段来区分从存储器中取出的是指令还是数据

不采用指令预取技术，每个指令周期都需要取指令，而不采用Cache技术，则每次取指令都至少要访存一次（当指令字长与存储字长相等且按边界对齐时）

空操作指令，在取指操作结束后，PC也会自动增加1

机器处于“开中断”状态，在每条指令执行结束时都可能被外部中断打断

**时钟周期**是计算机操作的最小单位时间，由计算机的主频确定，是主频的倒数

**工作脉冲**是控制器的最小时间单位，起定时触发作用，一个时钟周期有一个工作脉冲

采用DMA方式传送数据时，每传送一个数据就要占用**存取周期**

**存取周期**是指存储器进行两次独立的存储操作（连续两次读或写操作）所需的最小时间间隔

**指令周期**是指CPU从主存中取出一条指令加上执行这条指令的时间

指令总是根据**PC**从主存中读出

在取指操作**后**，PC中存放的是下一条指令的地址

取指操作是自动进行的，控制器不需要得到相应的指令。取指操作是控制器固有的功能，不需要在操作码控制下完成

指令周期的第一个机器周期是取指周期，即从主存中取出指令字

存储器进行一次读或写操作所需的时间称为存储器的**访问时间**（或**读写时间**），而连续启动两次独立的读或写操作（如连续的两次读操作）所需的最短时间称为**存取周期**。**机器周期**通常是由**存取周期**确定

通常把通过一次总线事务访问一次主存或I/O的时间定为一个**机器周期**

在指令的执行周期完成后，处理器会判断是否出现中断请求，只有在出现中断请求时，才会进入中断周期。所以并不是每个指令周期都包含一个中断响应机器周期

执行各条指令的机器周期数可变，各机器周期的长度可变

在指令长度相同的情况下，所有指令的取指操作是相同的

控制器可区分存储单元中存放的是指令还是数据

如果**指令字长**等于**存储字长**（不是机器字长），**取指周期**就等于**机器周期**

指令字长和机器字长的长度没关系

#### 数据通路的功能和基本结构

##### 数据通路的功能

* + 数据通路：数据在功能部件之间传送的路径
  + 中央处理器的内部数据通路：运算器和各寄存器之间的传送路径

##### 数据通路的基本结构

CPU内部单总线方式

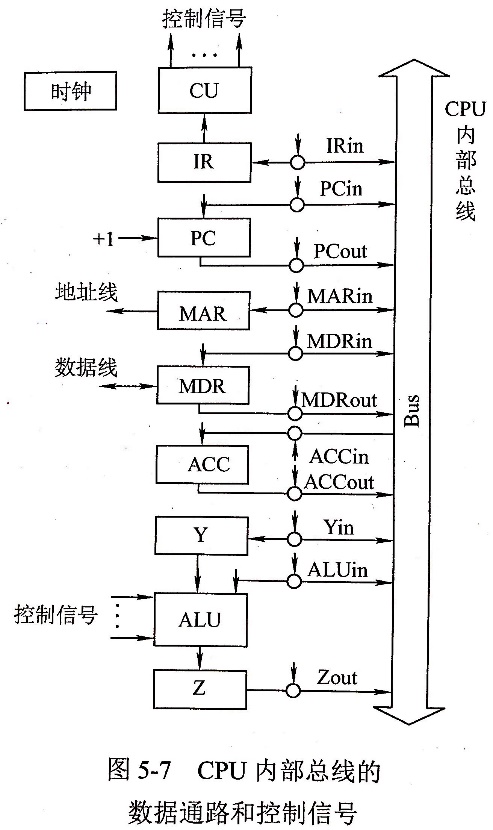
* + 将所有寄存器的输入端和输出端都连接到一条公共的通路上。
  + 数据传输存在较多的冲突现象，性能较低

CPU内部三总线方式

* + 将所有寄存器的输入端和输出端都连接到多条公共的通路上

专用数据通路方式

* + 根据指令执行过程中的数据和地址的流动方向安排连接线路，避免使用共享的总线，性能高，硬件量大



图中，规定各部件用大写字母表示，字母加in，表示该部件允许输入控制信号；字母加out，表示该部件允许输出控制信号

**内部总线**是指同一部件，如CPU内部连接各寄存器及运算部件之间的总线；**系统总线**是指同一台计算机系统的各部件，如CPU,内存，通道和各类I/O接口之间相互连接的总线

寄存器之间数据传送

* + 可通过CPU内部总线完成
  + 如把PC内容送到MAR，流程及控制信号：

PC→Bus PCout有效，PC内容送总线

Bus→MAR MARin有效，总线内容送MAR

主存与CPU之间的数据传送

* + 也要借助CPU内部总线完成。
  + 如CPU从主存读命令：

PC→Bus→MAR PCout和MARin有效，现行指令地址→MAR

1→R CU发出读命令

MEM(MAR) →MDR MDRin有效

MDR→Bus→IR MDRout和IRin有效，现行指令→IR

* + 由第三条看出主存和MDR之间直接连通，无需经过Bus

执行算数或逻辑运算

* + ALU本身是没有内部存储功能的组合电路，因此如要执行加法运算，被相加的两个数必须在ALU的两个输入端同时有效。上图中的暂存器Y即用于该目的。先将一个操作数经CPU内部总线送入暂存器Y保存起来，Y的内容在ALU的左输入端始终有效，再将另一个操作数经总线直接送到ALU的右输入端。这样两个操作数就都送入了ALU，运算结果暂存在暂存器Z中

Ad(IR)→Bus→MAR MDRout和MARin有效

1→R CU发出读命令

MEM→数据线→MDR 操作数从存储器→数据线→MDR

MDR→Bus→Y MDRout和Yin有效，操作数→Y

(ACC)+(Y)→Z ACCout和ALUin有效，CU向ALU发加命令，结果→Z

Z→ACC Zout和ACCin有效，结果→ACC

##### 习题知识点

CPU数据通路

* + 总线结构
    - 单总线结构
    - 双总线结构
    - 多总线结构
  + 专用数据通路结构

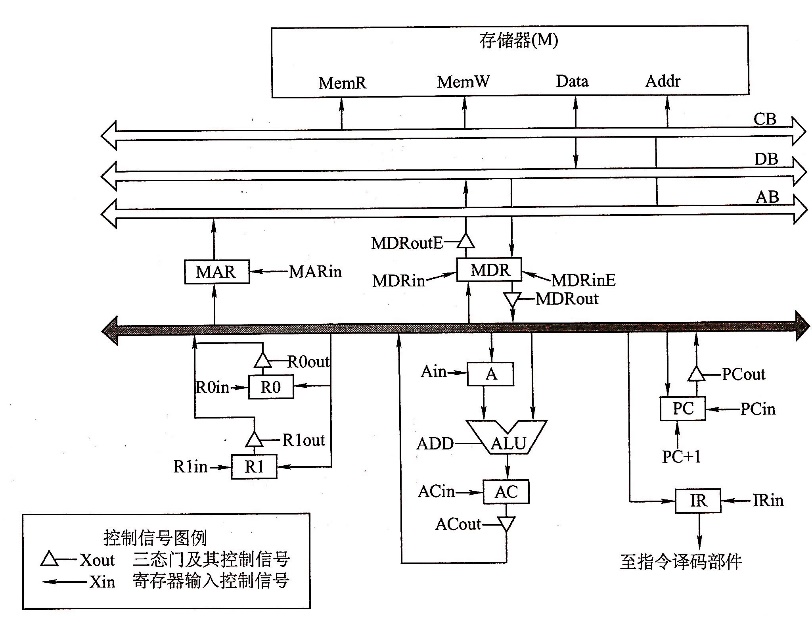
ALU只能有一个输入端可与总线相连，另一个输入端需通过暂存器与总线相连。ALU输出端也不能直接与内部总线相连，也需要一个暂存器与总线相连

采用CPU内部总线方式的数据通路特点：结构简单，实现容易，性能较低，存在较多的冲突现象；不采用CPU内部总线方式的数据通路特点：结构复杂，硬件量大，不易实现，性能高，基本不存在数据冲突现象

CPU的读/写控制信号的作用是：

* + 决定数据总线上的数据流向
  + 控制存储器操作的读/写类型
  + 控制流入、流出存储器信息的方向

某计算机字长16字，采用16位定长指令字结构。图中所有控制信号为1时有效。假设MAR的输出一直处于是能状态（所以没有MARout）。加法指令“ADD (R1),R0”的功能为(R0)+((R1))→(R1)



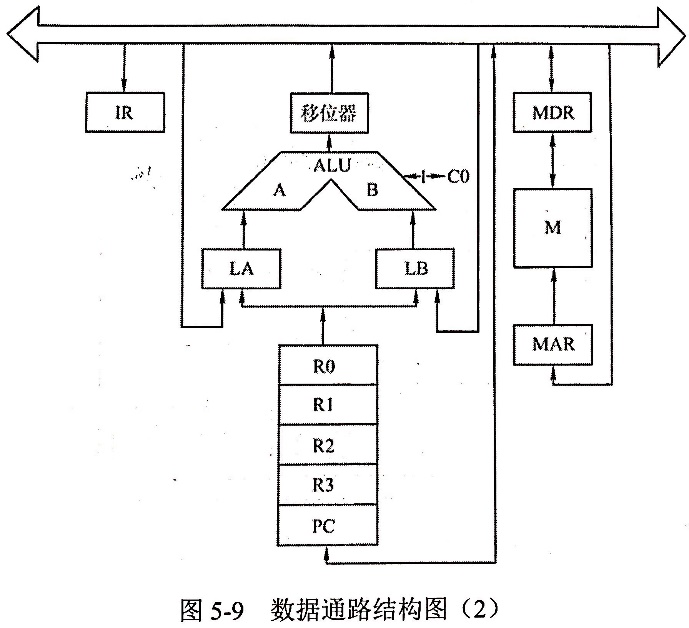
* + 用表格列出指令取指、译码和执行阶段每个节拍的功能和有效控制信号

|  |  |  |
| --- | --- | --- |
| 时钟 | 功能 | 有效控制信号 |
| C1 | (PC)→MAR | PCout, MARin |
| C2 | M(MAR)→MDR  (PC)+1→PC | MemR, MDRinE, PC+1 |
| C3 | (MDR)→IR | MDRout, IRin |
| C4 | 指令译码 | 无 |
| C5 | (R1)→MAR | R1out, MARin |
| C6 | M(MAR)→MDR | MemR, MDRinE |
| C7 | (MDR)→A | MDRout, Ain |
| C8 | (A)+(R0)→AC | R0out, Add, ACin |
| C9 | (AC)→MDR | ACout, MDRin |
| C10 | (MDR)→M(MAR) | MDRoutE, MemW |

* + C5 (R1)→MAR
  + C6 M(MAR)→MDR
    - 读出R1的内容所指的主存单元：先将R1的内容送到MAR，读出的数据必须经过MDR
  + C7 (MDR)→A
    - ALU一端是寄存器A，MDR或R0中必须有一个先写入A中
  + C8 (A)+(R0)→AC
  + 方案二：

|  |  |  |
| --- | --- | --- |
| C5 | (R1)→MAR | R1out, MARin |
| C6 | M(MAR)→MDR  (R0)→A | MemR, MDRinE, R0out, Ain |
| C7 | (A)+(MDR)→AC | MDRout, Add, ACin |
| C8 | (AC)→MDR | ACout, MDRin |
| C9 | (MDR)→M(MAR) | MDRoutE, MemW |

即在方案一C6执行M(MAR)→MDR的同时，完成(R0)→A，并不会发生总线冲突，可节省一个节拍

某计算机的数据通路结构如图所示，写出实现  
ADD R1, (R2)的微操作序列（含取指令及确定后继指令地址）

|  |  |
| --- | --- |
| 微操作 | 控制信号 |
| (PC)→MAR | PC→BUS, BUS→MAR |
| M→MDR | READ |
| (PC)+1→PC | +1 |
| MDR→IR | MDR→BUS, BUS→IR |
| (R1)→LA | R1→LA |
| (R2)→MAR | R2→BUS, BUS→MAR |
| M→MDR | READ |
| MDR→LB | MDR→BUS, BUS→LB |
| (LA)+(LB)→R1 | +, 移位器→BUS, BUS→R1 |

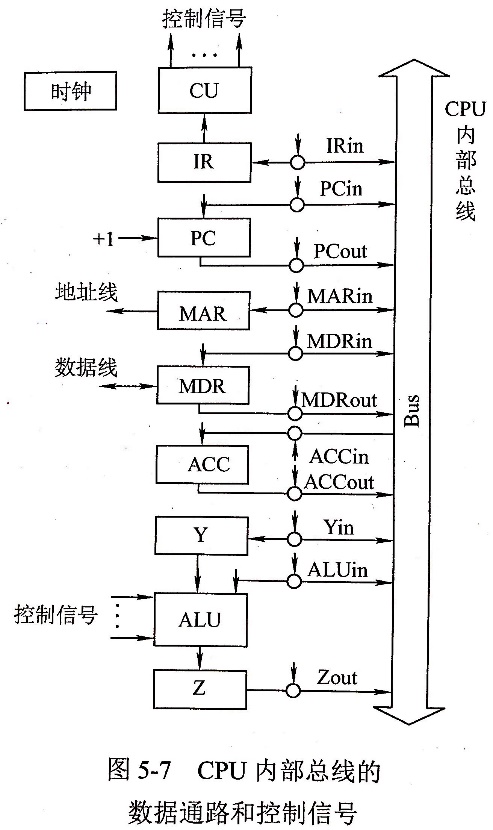
CPU内部结构如下图，此外还设有BCDEHL这6个寄存器（未画出），假设ALU的结果直接送入Z寄存器，要求从取指令开始，写出完成下列指令的微操作序列及所需的控制信号

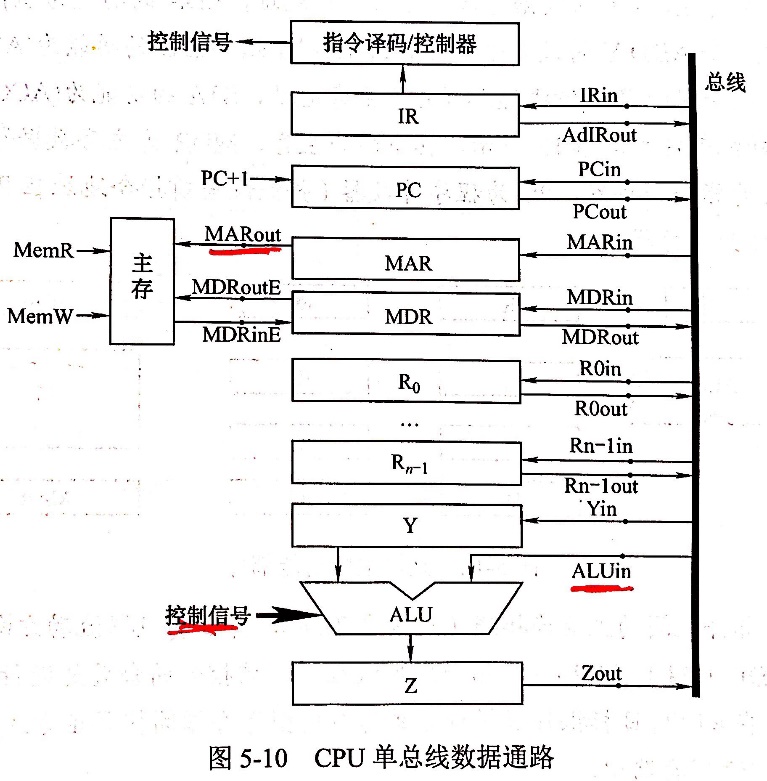
* + ADD B,C (B)+(C)→B

|  |  |
| --- | --- |
| 微操作 | 控制信号 |
| (PC)→MAR | PCout, MARin |
| (PC)+1→PC | +1 |
| M(MAR)→MDR→IR | MDRout, IRin |
| B→Y | Bout, Yin |
| (Y)+(C)→Z | Cout, ALUin, “+” |
| (Z)→B | Zout, Bin |

* + SUB A,H (AC)-(H)→AC

|  |  |
| --- | --- |
| 微操作 | 控制信号 |
| (PC)→MAR | PCout, MARin |
| (PC)+1→PC | +1 |
| M(MAR)→MDR→IR | MDRout, IRin |
| ACC→Y | ACCout, Yin |
| (Y)-(H)→Z | Hout, ALUin, “-” |
| (Z)→ACC | Zout, ACCin |

* + Y是与ALU的A端相连接的暂存器

设系统为单总线结构，分析指令ADD @R0,R1的指令流程和控制信号

* + ((R0)) + (R1) → (R0)
  + 取指周期：公共操作

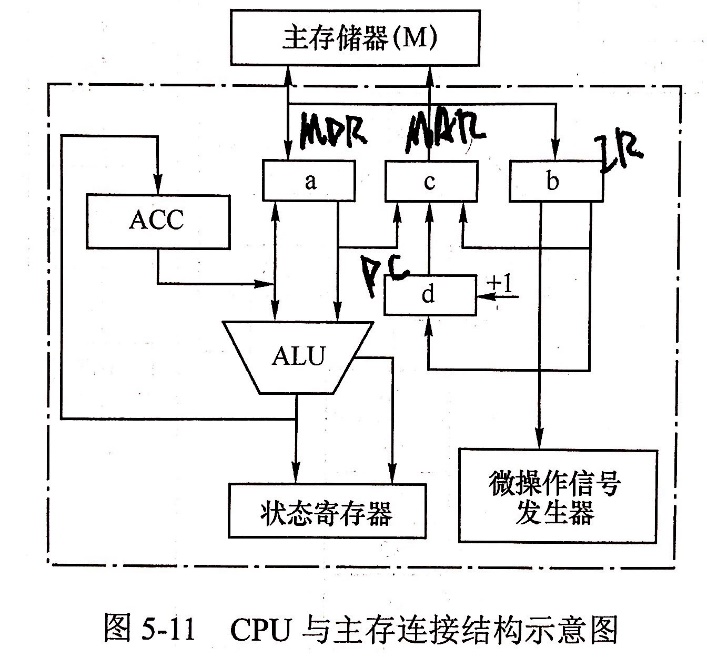
|  |  |
| --- | --- |
| 微操作 | 有效控制信号 |
| (PC) →MAR, Read | PCout, MARin |
| M(MAR) →MDR | MemR, MARout, MDRinE |
| (MDR) →IR | MDRout, IRin |
| 指令译码 | - |
| (PC+1) →PC | - |

* + 间址周期：完成取操作数，被加数在主存，家数已经放在寄存器R1中

|  |  |
| --- | --- |
| 微操作 | 有效控制信号 |
| (R0) →MAR | R0out, MARin |
| M(MAR) →MDR | MemR,MARout,MDRinE |
| (MDR) →Y | MDRout, Yin |

* + 执行周期，完成加法运算，并将结果返回主存

|  |  |
| --- | --- |
| 微操作 | 有效控制信号 |
| (R1)+(Y) →Z | R1out, ALUin, CU向ALU发出ADD控制信号 |
| (Z) →MDR | Zout, MDRin |
| (MDR) →M(MAR) | MemW, MDRoutE |

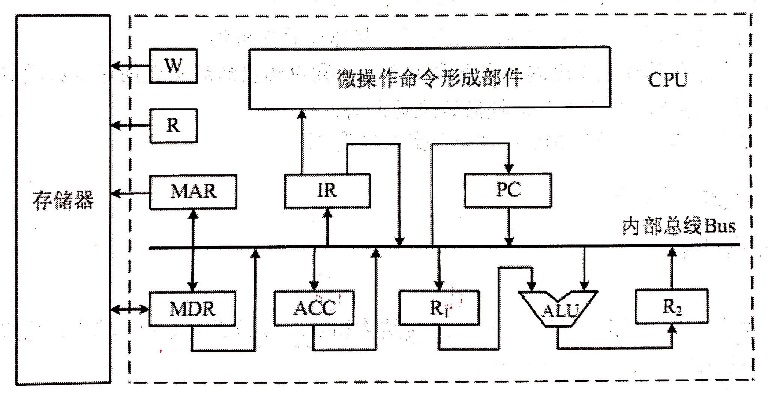
图为简化的CPU与主存连接结构示意图

* + 简述图中取指令的数据通路
    - PC→MAR→主存→MDR→IR
  + 存储器读数据的数据通路为（读取的数据存放在ACC中）
    - MAR(先置数据地址)，主存M→MDR→ALU→ACC
  + 存储器读数据的数据通路为（被写的数据存放在ACC中）
    - MAR(先置数据地址)，ACC→MDR→主存M
  + LDA X的数据通路（X为主存地址，LDA的功能为(X) →ACC）
    - X→MAR→主存 →MDR→ALU→ACC
  + ADD Y的数据通路（Y为主存地址，ADD:(ACC)+(Y) →ACC）
    - Y→MAR→主存→MDR, MDR和ACC→ALU→ACC
  + STA Z的数据通路（Z为主存地址，STA:(ACC) →Z）
    - Z→MAR, ACC→MDR→主存

单总线计算机结构如图，M为主存，XR为变址寄存器，EAR为有效地址寄存器，LATCH为暂存器。假设指令地址已在PC，给出ADD X,D指令周期信息流程和相应的控制信号

* + ADD X,D指令字中，X为变址寄存器XR,D为形式地址
  + 寄存器的输入输出均采用控制信号控制，如PCi表示PC的输入控制信号

|  |  |  |
| --- | --- | --- |
| 周期 | 微操作 | 有效控制信号 |
| 取指 | (PC) →MAR | PCo,MARi |
| M(MAR) →MDR  (PC)+1→PC | MARo, R/W, MDRi, +1 |
| (MDR) →IR | MDRo,IRi |
| 执行 | (XR)+Ad(IR) →EAR | XRo, IRo, +, EARi |
| (EAR) →MAR | EARo,MARi |
| M(MAR) →MDR | MARo,R/W, MDRi |
| (MDR) →X | MARo, Xi |
| (ACC)+(X) →LATCH | ACCo, Xo, Ki=+, LATCHi |
| (LATCH) →ACC | LATCHo,ACCi |

设CPU中各部件机器相互连接关系如图，W是写控制标志，R是读控制标志，R1，R2是暂存器

* + 假设要求在取指周期由ALU完成(PC)+1→PC的操作（即ALU可以对它的一个源操作数完成+1运算），要求用最少节拍写出取指周期全部微操作及节拍安排
    - 由于(PC)+1→PC需要由ALU完成，而取指周期不需要ALU，因此二者可以在同一节拍内实现，PC作为ALU的一个输入的同时作为MAR的输入，ALU计算结果作为PC的后值

|  |  |
| --- | --- |
| C1 | (PC) →MAR, (PC) →R1 |
| C2 | (PC)+1→R2, M(MAR) →MDR |
| C3 | (MDR) →IR |
| C4 | R2→PC |

* + - C3,C4总线冲突，不可放在同一时钟周期
  + 写出ADD #a(#为立即寻址特征，隐含的操作数在ACC中)在执行阶段所需的微操作命令及节拍安排
    - 一个操作数隐藏在ACC中

|  |  |
| --- | --- |
| C1 | ad(IR) →R1 |
| C2 | (ACC)+(R1) →R2 |
| C3 | (R2) →ACC |

P195 9题略过

#### 控制器的功能和工作原理

##### 硬布线控制器(组合逻辑控制器)

时钟周期：用时钟信号控制节拍发生器，可以产生节拍，每个节拍的宽度正好对应一个时钟周期。在每个节拍内机器可完成一个或多个需同时执行的操作

机器周期：机器周期可以看作是所有指令执行过程中的一个基准时间。通常用存取周期作为基准时间，即内存中读取一个指令字的最短时间作为机器周期。在指令字长等于存储字长时，取指周期也可看作是机器周期。在一个机器周期内可完成若干微操作，可用时钟信号来控制产生每一个微操作命令

取指周期的微操作命令：无论是什么指令，取指周期都需有下列微操作命令

* + PC→MAR 现行指令地址→MAR
  + 1→R 命令存储器读
  + M(MAR)→MDR 现行指令从存储器中读至MDR
  + MDR→IR 现行指令→IR
  + OP(IR) →CU 指令的操作码→CU译码
  + (PC)+1→PC 形成下一条指令的地址

间址周期：

|  |  |
| --- | --- |
| Ad(IR) →MAR | 将指令字中的地址码（形式地址）→MAR |
| 1→R | 命令存储器读 |
| M(MAR) →MDR | 将有效地址从存储器读至MDR |

执行周期

* + 非访存指令

|  |  |  |
| --- | --- | --- |
| CLA | 清ACC | 0→ACC |
| COM | 取反 |  |
| SHR | 算数右移 | L(ACC) →R(ACC)  ACC0→ACC0 |
| CSL | 循环左移 | R(ACC) →L(ACC)  ACC0→ACCn |
| STP | 停机指令 | 0→G |

* + 访存指令
    - 加法指令： ADD X
      * Ad(IR) →MAR
      * 1→R
      * M(MAR) →MDR
      * (ACC)+(MDR) →ACC
    - 存数指令： STA X
      * Ad(IR) →MAR
      * 1→W
      * ACC→MDR
      * MDR→M(MAR)
    - 取数指令： LDA X
      * Ad(IR) →MAR
      * 1→R
      * M(MAR) →MDR
      * MDR→ACC
  + 转移指令

|  |  |  |
| --- | --- | --- |
| JMP X | 无条件转移 | Ad(IR) →PC |
| BAN X | 条件转移（负则转） |  |

CPU控制方式

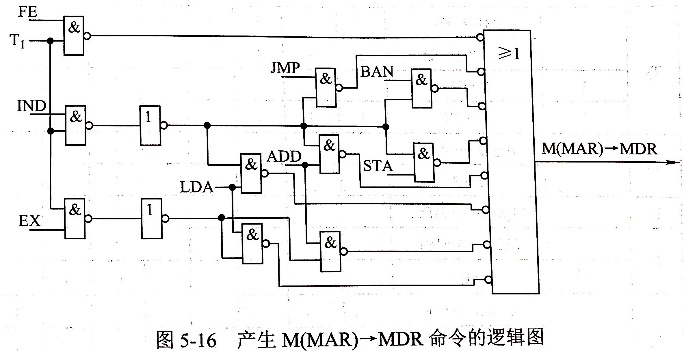
* + 同步控制方式
  + 异步控制方式
  + 联合控制方式

硬布线控制单元设计

* + I为间址标志，在取指周期的T2时刻，若测得I=1，则IND触发器置1，标志进入间址周期；若I=0，则EX触发器置1，进入执行周期。
  + 若IND=1（表示多次间接寻址），则继续间接寻址。
  + 在执行周期T2时刻，CPU要向所有中断源发中断查询信号，若检测到有中断请求且满足相应条件，则INT置1，标志进入中断周期

| 工作周期标记 | 节拍 | 状态条件 | 微操作命令信号 | CLA | COM | SHR | CSL | STP | ADD | STA | LDA | JMP | BAN |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FE取指 | T0 |  | PC→MAR | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1→R | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| T1 |  | M(MAR)→MDR | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| (PC)+1→PC | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| FE取指 | T2 |  | MDR→IR | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| OP(IR)→ID | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| I | I→IND |  |  |  |  |  | 1 | 1 | 1 | 1 | 1 |
|  | I→EX | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| IND  间接寻址 | T0 |  | Ad(IR)→MAR |  |  |  |  |  | 1 | 1 | 1 | 1 | 1 |
| I→R |  |  |  |  |  | 1 | 1 | 1 | 1 | 1 |
| T1 |  | M(MAR)→MDR |  |  |  |  |  | 1 | 1 | 1 | 1 | 1 |
| T2 |  | MDR→Ad(IR) |  |  |  |  |  | 1 | 1 | 1 | 1 | 1 |
|  | I→EX |  |  |  |  |  | 1 | 1 | 1 | 1 | 1 |
| EX执行 | T0 |  | Ad(IR)→MAR |  |  |  |  |  | 1 | 1 | 1 |  |  |
| 1→R |  |  |  |  |  | 1 |  | 1 |  |  |
| 1→W |  |  |  |  |  |  | 1 |  |  |  |
| T1 |  | M(MAR)→MDR |  |  |  |  |  | 1 |  | 1 |  |  |
| AC→MDR |  |  |  |  |  |  | 1 |  |  |  |
| T2 |  | (AC)+(MDR)→AC |  |  |  |  |  | 1 |  |  |  |  |
| MDR→M(MAR) |  |  |  |  |  |  | 1 |  |  |  |
| MDR→AC |  |  |  |  |  |  |  | 1 |  |  |
| 0→AC | 1 |  |  |  |  |  |  |  |  |  |
| →AC |  | 1 |  |  |  |  |  |  |  |  |
| L(AC)→R(AC)  ACo不变 |  |  | 1 |  |  |  |  |  |  |  |
|  |  |  |  | 1 |  |  |  |  |  |  |
| Ad(IR)→PC |  |  |  |  |  |  |  |  | 1 |  |
| A0 | Ad(IR)→PC |  |  |  |  |  |  |  |  |  | 1 |
|  | 0→G |  |  |  |  | 1 |  |  |  |  |  |

* + 根据微操作时间表，写出各微操作控制信号的逻辑表达式，并进行适当简化
  + 微操作控制信号=机器周期^节拍^脉冲^操作码^机器状态条件
  + 如根据表可写出M(MAR)→MDR微操作命令的逻辑表达式：
    - M(MAR) →MDR  
      =FE·T1+IND·T1(ADD+STA+LDA+JMP+BAN)  
       +EX·T1(ADD+LDA)  
      =T1{FE+IND(ADD+STA+LDA+JMP+BAN)  
       +EX(ADD+LDA)}
  + 画出逻辑图



##### 微程序控制器

微程序控制器才会用存储逻辑实现，也就是把微操作信号代码化，使每条机器指令转化称为一段微程序并存入一个专门的存储器（控制存储器）中个，微操作控制信号由微指令产生

微程序设计思想：将每一条**机器指令**编写成一个**微程序**，每一个**微程序**包含若干条**微指令**，每一条**微指令**对应一个或多个**微操作命令。**这些微程序可以存放到一个控制存储器中，用寻址用户程序机器指令的方法来寻址每个微程序中的微指令

**微命令**和**微操作：**一条机器指令可以分解成一个微操作序列，这些微操作是计算机中最基本的、不可再分解的操作。

* + 微命令和微操作是一一对应的。
  + 微命令是微操作的控制信号
  + 微操作是微命令的执行过程
  + 相容性微命令：那些可以同时产生、共同完成某一些微操作的命令
  + 互斥性微命令：在机器中不允许同时出现的微命令
  + 在组合逻辑控制器中也存在微命令和微操作这两个概念

**微指令**和**微周期**

* + 微指令是若干微命令的集合。存放微指令的控制存储器的单元地址称为**微地址**
  + 微指令通常至少包括：操作控制字段，顺序控制字段
    - **操作控制字段**：微操作码字段，用于产生某一步操作所需要的各种操作控制信号
    - **顺序控制字段**：微地址码字段，用于控制产生下一条要执行的微指令地址
  + **微周期**通常指从**控制存储器**读取一条微指令并执行相应的微操作所需的时间

**主存储器**和**控制存储器**

* + 主存储器用于存放程序和数据，在CPU外部，用RAM实现
  + 控制存储器(CM)用于存放微程序，在CPU内部，用ROM实现

**程序**和**微程序**

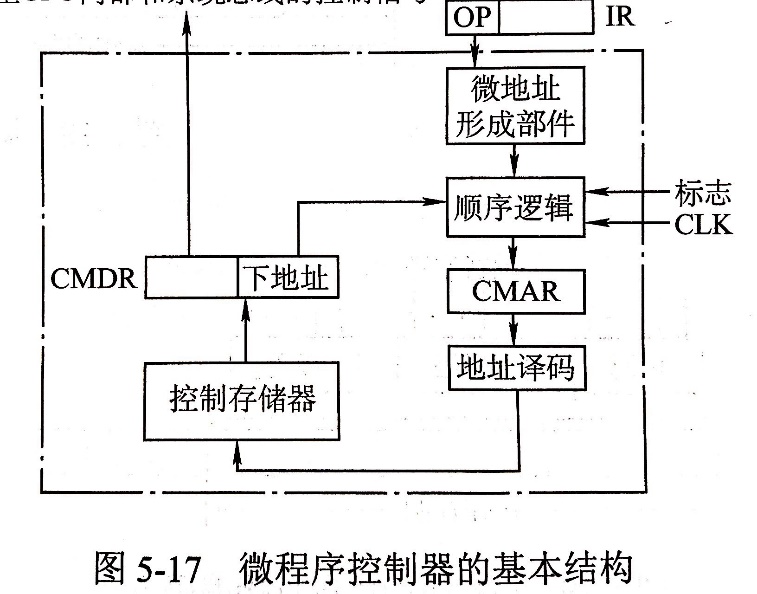
* + 程序是指令的有序集合，用于完成特定的功能
  + 微程序是**微指令**的有序集合，一条指令的功能由一段微程序来实现
  + 微程序用于描述机器指令。
  + 微程序实际上是机器指令的实时解释器，是由计算机设计者实现编制好存放在**控制存储器**中的，一般不提供给用户
  + 程序最终由**机器指令**组成，是由软件设计人员实现编制好并存放在主存或辅存中的

**地址寄存器**（MAR）：用于存放主存的读/写地址

**微地址寄存器**（CMAR）：用于存放控存的读/写微指令的地址

**指令寄存器**（IR）:用于存放从主存中读出的指令

**微指令寄存器**（CMDR或）:用于存放从控存中读出的微指令



**控制存储器CM**：存放各指令对应的**微程序**，ROM

**微指令寄存器CMDR**：存放从CM中取出的**微指令**，位数同**微指令**字长

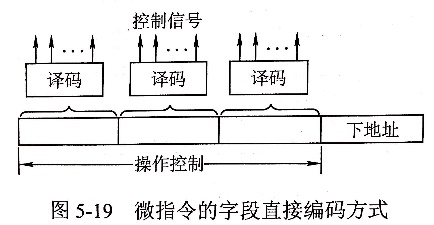
**微地址形成部件**：用于产生初始微地址和后继微地址，以保证**微指令**的连续执行

**微地址寄存器CMAR**：接受微地址形成部件送来的微地址，为在CM中读取**微指令**作准备

通常一条机器指令对应一个微程序

若指令系统中具有n种机器指令，则控制存储器中的微程序数至少是n+1个（1为公共的取指微程序）

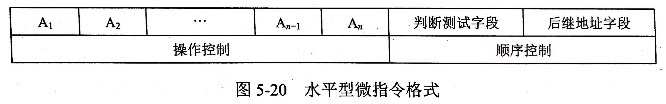
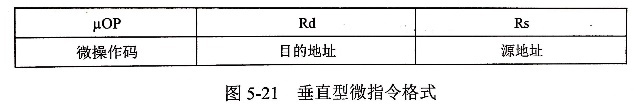
微指令的编码方式：

* + 直接编码（直接控制法）
    - 直接编码法无须进行译码，微指令的微命令字段的每一位都代表一个微命令
  + 字段直接编码方法
    - 
    - 将微指令的微命令字段分成若干小字段，把互斥性微命令组合在同一字段中，把相容性微命令组合在不同的字段中，每个字段单独编码
    - 一般每个小段还要留出一个状态，表示本字段不发出任何微命令
  + 字段间接编码方式

微指令的地址形成方式

* + 直接由微指令的下地址字段指出，又称断定方式
  + 根据机器指令的操作码形成。当机器指令取至到指令寄存器后，微指令的地址由操作码经微地址形成部件形成
  + 其他：
    - 增量计数器法：(CMAR)+1→CMAR，适用于后继微指令的地址是连续的情况
    - 根据各种标志决定微指令分支转移的地址
    - 通过网络测试形成
    - 有硬件直接产生微程序入口地址

微指令的格式

* + 水平型微指令
    - 从编码方式看，直接编码、字段直接编码、字段间接编码和混合编码都属于水平型微指令。
    - 一条水平型微指令定义并执行集中并行的基本操作
    - 优点：**微程序**短，执行速度快；
    - 缺点：**微指令**长，编写微程序较麻烦
  + 垂直型微指令
    - 采用类似机器指令操作码的方式，在微指令中设置微操作码字段，采用微操作码编译法，由微操作码规定微指令的功能
    - 一条垂直型微指令只能定义并执行一种基本操作
    - 优点：**微指令**短、简单、规整，便于编写微程序；
    - 缺点是**微程序**长，执行速度慢，工作效率低
  + 混合型微指令
  + 水平型微指令和垂直型微指令的比较：
    - 水平型微指令并行操作能力强、效率高、灵活性强；
    - 水平型微指令执行一条指令的时间短
    - 由水平型微指令解释指令的微程序，具有**微指令字**比较长但**微程序**短的特点；垂直型微指令则相反，其微指令字短而微程序长
    - 水平型微指令用户难以掌握，而垂直型微指令与指令比较相似，相对容易掌握

微程序控制单元的设计步骤

* + 写出对应机器指令的微操作命令及节拍安排
    - 如取指阶段：

|  |  |
| --- | --- |
| T0 | PC→MAR, 1→R |
| T1 | M(MAR) →MDR, (PC)+1→PC |
| T2 | MDR→IR,  OP(IR) →微地址形成部件 |

* + - 与硬布线控制单元相比，只在T2不同。硬布线控制单元在T2节拍内要将指令的操作码送至指令译码器，以控制CU发出相应的微命令，即OP(IR) →ID
    - 在取指微程序中，除第一条微指令外，其余微指令的地址均有上一条微指令的下地址字段直接给出，因此上述每一条微指令都需增加一个将微指令下地址字段送至CMAR的微操作，记作Ad(CMDR) →CMAR
    - 综上，取指微操作

|  |  |
| --- | --- |
| T0 | PC→MAR, 1→R |
| T1 | Ad(CMDR) →CMAR |
| T2 | M(MAR) →MDR, (PC)+1→PC |
| T3 | Ad(CMDR) →CMAR |
| T4 | MDR→IR |
| T5 | OP(IR) →微地址形成部件→CMAR |

* + - 执行阶段的微操作，与硬布线控制单元相比，多了将下一条微指令地址送至CMAR的微操作命令，即Ad(CMDR) →CMAR。其余的微操作命令与硬布线控制单元相同
  + 确定个微指令格式
  + 编写微指令码点

微程序控制器与硬布线控制器的对比

|  | 微程序控制器 | 硬布线控制器 |
| --- | --- | --- |
| 工作原理 | 微操作控制信号以微程序的形式存放在控制存储器中，执行指令时读出即可 | 微操作控制信号由组合逻辑电路根据当前的指令码、状态和时序，即时产生 |
| 执行速度 | 慢 | 快 |
| 规整性 | 较规整 | 繁琐、不规整 |
| 应用场合 | CISC CPU | RISC CPU |
| 易扩充性 | 易扩充修改 | 困难 |

##### 习题知识点

取指令操作是控制器固有的功能，不需要在操作码控制下进行

CU的输入信号来源如下：

* + 经指令译码器译码产生的指令信息
  + 时序系统产生的机器周期信号和节拍信号
  + 来自执行单元的反馈信息即标志
  + 前两者是主要因素

在微程序控制器中，形成微程序入口地址的是：机器指令的操作码字段。当执行完公共的取指微程序从主存中取出机器指令后，由机器指令的操作码字段指出各个微程序的入口地址（初始微地址）

微指令的设计目标和指令结构的设计目标类似，都是基于执行速度、灵活性和指令长度这三个主要方面考虑的

微程序控制器的速度比硬布线控制器慢，主要是因为：增加了从控制存储器读取微指令的时间

微程序控制器属于CPU的一部分

微指令计数器决定指令执行顺序 x  
微指令计数器决定微指令执行顺序

一条微指令存放在控制器的一个控存单元内

硬布线控制器需要结合各微操作的节拍安排，综合分析，写出逻辑表达式，再设计成逻辑电路，因此**时序系统**比较复杂；而微程序只需按照节拍的安排，顺序执行微指令，时序系统比较简单

在微程序控制器中，控制部件向执行部件发出的控制信号称为微命令，微命令执行的操作称为微操作。微指令则是若干微指令的集合，若干条微指令的有序集合称为微程序

在微程序控制器中，机器指令与微指令的关系是：每一条机器指令由若干微指令组成的微程序来解释执行

一条水平型微指令能定义并执行集中并行的基本操作；一条垂直型微指令只能定义并执行一种基本操作

某计算机采用微程序控制方式，微指令中的操作控制字段采用字段直接编码法，共有33个微命令，构成5个互斥类，分别包含7,3,12,5,6个微命令，则操作控制字段至少有15位

* + 每个字段要留出一个状态，表示本字段不发出任何微命令，需3，2，4，3，3共15位

微程序控制器，有32条指令，公共的取指令微程序包含2条微指令，各指令对应的微程序平均由4条微指令组成，采用断定法（下地址字段法）确定下条微指令地址，则微指令中下地址字段的位数至少是：8

* + 计算机有32条指令，各个指令对应的微程序平均为4条，则指令对应的微指令为32\*4=128条，公共微指令2条，所以整个系统中微指令的条数一共为128+2=130条，所以需要位

某带中断的计算机，若指令系统中具有n中机器指令，则控制存储器中的微程序数至少是n+2个，增加的一个为公共的取指微程序，另一个对应中断周期的微程序

兼容性微命令是指几个微命令可以同时出现

微程序控制方式中,每一条机器指令由一段微程序来解释执行

微程序控制方式中，直接编码方式，编码效率低，但执行效率高

微程序控制方式中，水平型微指令能充分利用数据通路的并行结构

通常情况下，一个微程序的周期对应一个机器周期

CPU控制器主要由3个部件组成：指令寄存器、程序计数器和操作控制器。状态寄存器通常属于运算器部件

一台计算机基本上可以划分为两大部分：控制部件和执行部件。控制器就是控制部件，而运算器、存储器、外围设备相对控制器来说就是执行部件

若存储器容量为64KBx32位，指出主机中各个寄存器的位数：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ACC | MQ | ALU | X | IR | MDR | PC | MAR |
| 32 | 32 | 32 | 32 | 32 | 32 | 16 | 16 |

写出硬布线控制器完成STA X（X为主存地址）指令的全部微操作命令及节拍安排

|  |  |
| --- | --- |
| T0 | PC→MAR, 1→R |
| T1 | M(MAR) →MDR, (PC)+1→PC |
| T2 | MDR→IR, OP(IR) →ID |
| T0 | Ad(IR) →MAR, 1→W |
| T1 | ACC→MDR |
| T2 | MDR→M(MAR) |

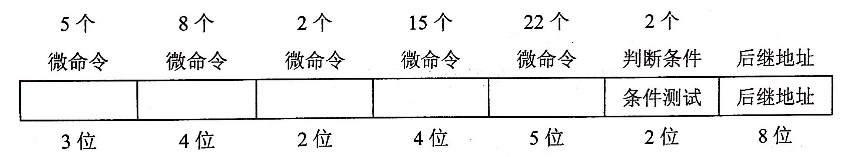
* + 若采用微程序控制，还需增加微操作：
  + 取指周期：
    - Ad(CMDR) →CMAR
    - OP(IR) →CMAR
  + 执行周期：
    - Ad(CMDR) →CMAR

假设某机器有80条指令,平均每条指令由4条微指令组成，其中有一条取指微指令是所有指令公用的。已知微指令长度为32位，请估算控制存储器CM容量

* + 总的微指令条数=(4-1)\*80+1=241条，每条微指令占一个控存单元，控存CM容量为2n，241刚好小于256，所以CM的容量=256\*32位=1KB

某机共有52个微操作控制信号，构成5个相斥类的微命令组，各组分别包含5、8、2、15、22个微命令。一直可判定的外部条件有2个，微指令字长28位

* + 按水平型微指令格式设计微指令，要求微指令的下地址字段直接给出后继微指令地址。
    - 每组还应包含不发命令的情况，条件测试字段应包含一种不转移的情况，则5个控制字段分别需给出6、9、3、16、23种状态，对应3、4、2、4、5位（共18位），条件测试字段取2位。
    - 根据微指令长度为28位，下地址字段28-18-2=8位
    - 其微指令格式如下图：

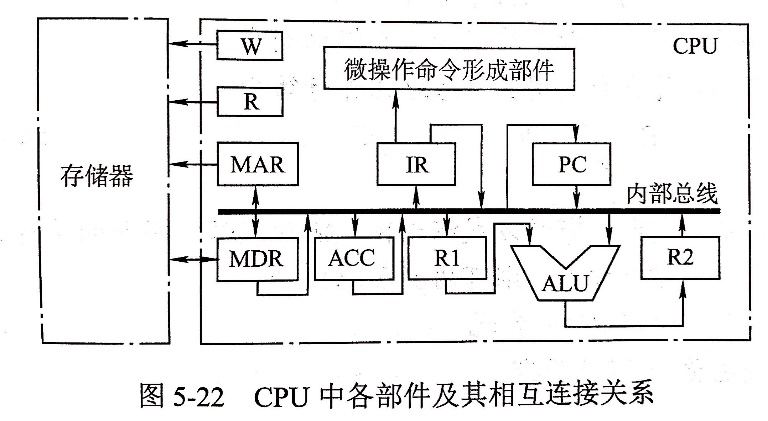


* + 控制存储器的容量：
    - 根据后继地址字段为8位，微指令字长为28位，得出控制存储器的容量为28x28位

某微程序控制器中，采用水平型直接控制（编码）方式的微指令格式，后继微指令地址由微指令的下地址字段给出。已知机器共有28个微命令，6个互斥的可判定的外部条件，控制存储器的容量为512Bx40位。设计其微指令格式

* + 水平型微指令由控制字段、判别测试字段和下地址字段三部分构成。
  + 因为微指令采用直接控制（编码）方式，所以其操作控制字段的位数等于微命令数，28位。
  + 又由于后继微指令地址由下地址字段给出，故其下地址字段的位数可根据控制存储器的容量（512B\*40位）确定为9位（512=29）
  + 当微程序出现分支时，后继微指令地址的形成取决于状态条件—6个互斥的可判定外部条件，因此状态位应编码3位。非分支时的后继微指令由微指令的下地址字段直接给出

|  |  |  |
| --- | --- | --- |
| 操作控制字段 | 判别测试字段 | 后继地址字段 |
| 28位 | 3位 | 9位 |

设CPU中各部件机器相互连接关系如图，其中W是写控制标志；R是读控制标志；R1,R2是暂存器

* + 写出指令ADD #a（#为立即寻址特征，隐含的操作数在ACC寄存器中）在执行阶段所完成的微操作命令及节拍安排

|  |  |  |
| --- | --- | --- |
| T0 | Ad(IR) →R1 | 立即数→R1 |
| T1 | (R1)+(ACC) →R2 | ACC通过总线送ALU |
| T2 | (R2) →ACC | 结果→ACC |

* + 假设要求在取指周期实现(PC)+1→PC，且由ALU完成此操作（ALU可以对它的一个源操作数完成+1运算）。以最少节拍写出取指周期全部微操作命令及节拍安排
    - 由于(PC)+1→PC需由ALU完成，因此PC的值可作为ALU的一个源操作数，在ALU做+1得到(PC)+1后，结果送至与ALU输出端相连的R2，然后送至PC

|  |  |
| --- | --- |
| T0 | PC→MAR, 1→R |
| T1 | M(MAR) →MDR, (PC)+1→R2 |
| T2 | MDR→IR, OP(IR) →微操作命令形成部件 |
| T3 | (R2) →PC |

#### 指令流水线

##### 指令流水线的基本概念

流水线方法的特点：

把一个任务（一条指令或一个操作）分解为几个有联系的子任务，每个子任务由一个专门的功能部件来执行，并依靠多个功能部件并行工作来缩短程序的执行时间

流水线每一个功能段部件后面都要有一个缓冲寄存器，或称为锁存器，用来保存本流水段的执行结果，提供给下一流水段使用

流水线中各功能段的时间应该尽量相等，否则将引起堵塞、断流

只有连续不断地提供同一种任务时才能发挥流水线的效率，所以在流水线中必须处理连续任务

装入时间：第一个任务进入流水线到输出流水线的时间。

排空时间：最后一个任务进入流水线到输出流水线的时间

##### 流水线分类

部件功能级、处理机级和处理机间级流水线

单功能流水线和多功能流水线

动态流水线和静态流水线

线性流水线和非线性流水线

##### 影响流水线的因素

**结构相关**（资源冲突）

* + 多条指令在同一时刻争用同一资源
  + 解决方法：
    - 前一指令访存时，使后一条相关指令（以及其后继指令）暂停一个时钟周期
    - 单独设置数据存储器和指令存储器，使两项各自在不同的存储器中进行，这属于资源重复配置

**数据相关**（数据冲突）

* + 存在必须等待前一条指令执行完才能执行后一条指令的情况
  + 解决方法：
    - 把遇到数据相关的指令及其后继指令都暂停一个至几个周期，可分为硬件阻塞（stall）和软件插入”NOP”指令两种方法
    - 设置相关专用通路，即不等前一条指令把计算结果写回寄存器组，下一条指令也不读寄存器组，而是直接把前一条指令的ALU的计算结果作为自己的输入数据开始计算过程，使本来需要暂停的操作变得可以继续执行，称为**数据旁路技术**
    - 通过编译器对数据相关的指令编译优化的方法，调整指令顺序来解决数据相关

**控制相关**（控制冲突）

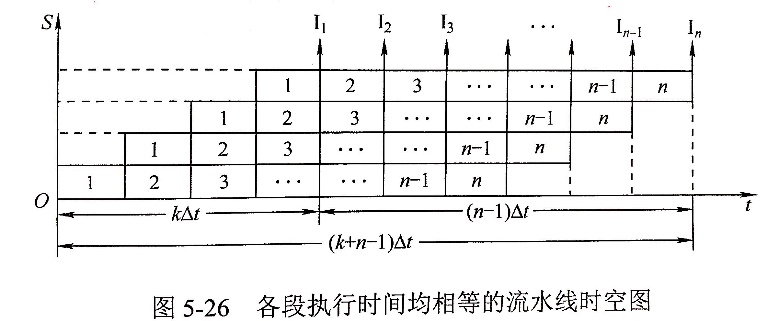
* + 当流水线遇到转移指令和其他改变PC值的指令而造成断流
  + 解决方法：
    - 对转移指令进行**分支预测**，尽早生成转移目标地址。分支预测分为简单（静态）预测和动态预测。静态预测总是预测条件不满足，即继续执行分支指令的后继指令。动态预测根据程序执行的历史情况，进行动态预测调整，有较高的预测准确率
    - 预取转移成功和不成功两个控制流方向上的目标指令
    - 加快和提前形成条件码
    - 提高转移方向的猜准率

Cache缺失的处理过程也会引起流水线阻塞

##### 流水线的性能指标

吞吐率

* + ，n是任务数；Tk是处理完成n个任务所用的时间。
  + 一条k段线性流水线能够在k+n-1个时钟周期完成n个任务。K为流水线段数；t为时钟周期。诗句吞吐率为
  + 当时，



加速比

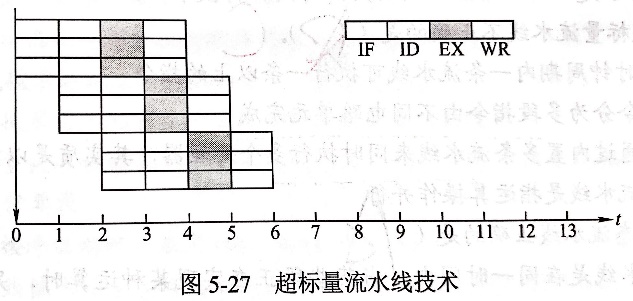
* + T0表示不适用流水线的执行时间，即顺序执行；Tk表示使用流水线的执行时间
  + 加速比：
  + 如果流水线各段执行时间都相等，则一条k段流水线完成n个任务所需的时间为Tk=(k+n-1)t。不使用流水线所需时间T0=knt。得加速比
  + 当时，

效率

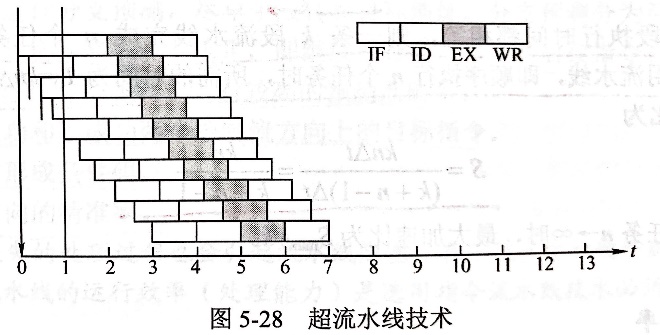
* + 流水线设备利用率。
  + 包含了时间和空间两个因素
  + 当时，

##### 超标量流水线的基本概念

超标量流水线技术

* + 每个时钟周期可并发多条独立指令，即以并行操作方式将两条或多条指令编译并执行，为此需配多个功能部件
  + 不能调整指令的执行顺序，通过编译优化技术，把可并行指令搭配起来
  + 

超流水线技术

* + 在一个时钟周期内再分段，在一个时钟周期内一个功能部件使用多次
  + 不能调整指令的执行顺序，靠编译程序解决优化问题

超长指令字

* + 靠编译程序挖掘出指令间潜在的并行性，将多条能并行操作的指令组合成一条具有多个操作码字段的超长指令字，需采用多个处理部件

##### 习题知识点

流水CPU利用的是**时间**并行性

RISC都采用流水线技术。大部分CISC也采用了流水线基数

流水CPU和多媒体CPU无联系

超标量流水线是指在一个时钟周期内一条流水线可执行一条以上的指令。一条指令分为多段指令，由不同电路单元完成。超标量是通过内置多条流水线来同时执行多个处理器，其实质是以**空间换时间**

静态流水线上下段连接方式固定，而动态流水线的连接方式是可变的。动态流水线在同一时间内，当某些段正在实现某种运算使，另一些段却正在进行另一种运算，这样对提高流水线的效率有好处，但会使流水线控制变得复杂

一个m段流水线稳定时的CPU吞吐性能，与m个并行部件的CPU的吞吐能力相比，相等

设指令由取指、分析、执行三个子部件完成，并且每个子部件的时间均为，若采用度为4的超标量流水线处理机，连续执行20条指令，共需：7

设指令流水线把一条指令分为取指、分析、执行3个部分，且3部分的时间分别是2ns,2ns,1ns，则100条指令全部执行完毕需：203ns

* + =204ns，但最后一步，是1ns，不是2ns，所以203ns

某CPU主频为1.03Ghz，采用4级指令流水线，每个流水段的执行需1个时钟周期。假设CPU执行了100条指令，无任何阻塞，求此时流水线吞吐率：

* + 100条指令共103个时钟周期
  + CPU每秒有1.03G个时钟周期
  + 1.03Gx100/103=1.0x109条指令/秒

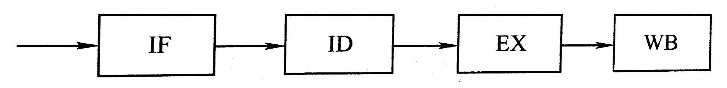
在分支指令加入若干空操作可以避免**控制冒险**

超标量技术需要配置多个功能部件和指令译码电路

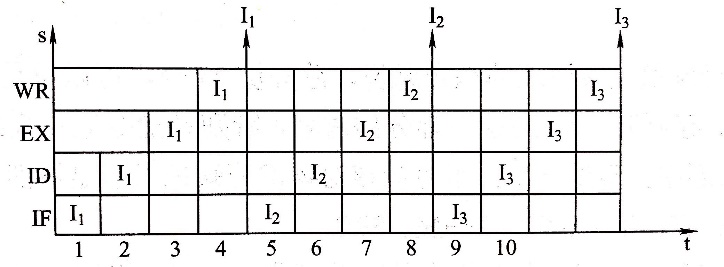
流水线按序流动时，不会出现先读后写？？（WAR）和写后写（WAW）相关

假设指令流水线分为取指（IF），译码（ID），执行（EX），回写（WB）4个过程，共有10条指令连续进入此流水线

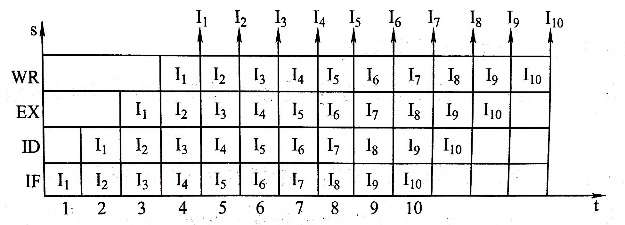
* + 画出指令周期流程图



* + 画出非流水线时空图



* + 画出流水线时空图

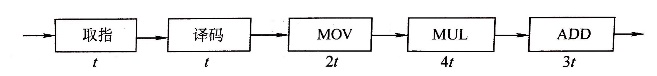


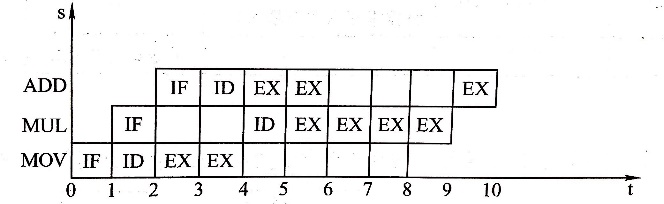
* + 假设时钟周期100ns，求流水线的实际吞吐量
    - 由上图，在13个时钟周期结束时，CPU执行完10条指令，故实际吞吐率

某台单流水线多操作部件处理机，包含有取指、译码、执行3个功能段，在该机上执行以下程序：取指和译码功能段各需要1个时钟周期，MOV操作需要两个时钟周期，ADD操作需要3个时钟周期，MUL操作需要4个时钟周期，每个操作都是在第一个时钟周期接收数据，在最后一个时钟周期把结果写入通用寄存器

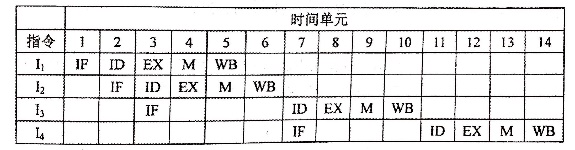
|  |  |  |
| --- | --- | --- |
| K | MOV R1,R0 | (R0) →R1 |
| K+1 | MUL R0,R1,R2 | (R1)\*(R2) →R0 |
| K+2 | ADD R0,R2,R3 | (R2)+(R3) →R0 |

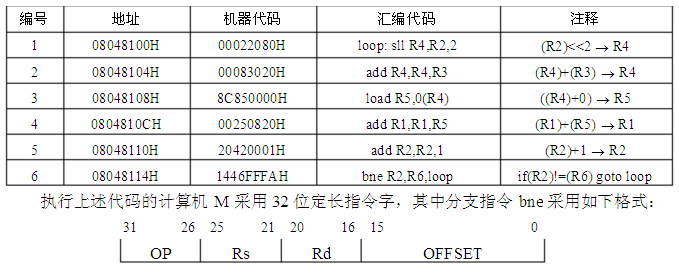
* + 画出流水线功能段结构图



* + 画出指令执行过程流水线的时空图
    - 图中MUL的ID阶段推迟，是因为ID阶段要将R1取至MDR，且MUL与前面的MOV存在数据相关
    - 第5~6段时间中ADD的EX与MUL的EX不会冲突，因其不再同一执行部件，而ADD的最后一个EX需要等MUL执行完之后才能执行

知识点补充：

* + **IF**:取指，所有指令共有的部分，访存取指到IR
  + **ID**:译码，所有指令共有的部分，将IR的指令译码，此时也有默认取寄存器数的过程
    - IF,ID不受指令类型影响
  + **EXE**:运用ALU，其中，加法取反等运算指令直接对操作数进行运算，访存指令用于计算下址
  + **MEM**:访存指令用以访存过程，不访存的指令直接跳过
  + **WB**：运算指令将结果输送到寄存器，load指令访存写回寄存器
  + 整个指令执行过程中各段时间是相同的，统一受时钟沿控制。整个执行过程只有EXE部分使用ALU，避免了资源冲突。由于IF,MEM都有可能访存，通常设置指令Cache和数据Cache避免资源冲突
  + 发生阻塞的情况：
    - 如过上一条指令的WB写回的寄存器与本指令对应寄存器相同，会发生资源冲突，此时有3个时钟周期的阻塞，使本指令ID应在上一条WB之后
    - 跳转指令(JMP)：由于流水线默认直接提取下一条指令，如果指令为JMP或者JC(根据情况预判跳转结果)，在没有分支预测的情况下，默认有3个时钟周期的阻塞使本指令ID应在上一条WB后。
  + bne: branch not equal 条件跳转指令

某程序中有如下循环代码段p：“for(int i = 0; i < N; i++) sum+=A[i];”。假设编译时变量sum和i分别分配在寄存器R1和R2中。常量N在寄存器R6中，数组A的首地址在寄存器R3中。程序段P起始地址为0804 8100H，对应的汇编代码和机器代码如下表所示。 OP为操作码;；Rs和Rd为寄存器编号；OFFSET为偏移量，用补码表示

* + M的存储器编址单位是什么
    - 已知计算机M采用32位定长指令字，即一条指令占4B，表中每条指令的地址差为4个地址单位，即4个地址单位代表4B，一个地址单位代表了1B，所以按字节编址
  + 已知sll指令实现左移功能，数组A中每个元素占多少位？
    - 在二进制中某数左移二位相当于乘4，由该条件可知，树组件的数据间隔为4个地址单位，而计算机按字节编址，所以每个元素占4B
  + 表中bne指令的OFFSET字段的值是多少？已知bne指令采用相对寻址方式，当前PC内容为bne指令地址，通过分析表中指令地址和bne指令内容，推断出bne指令的转移目标地址计算公式。
    - 由表可知，bne指令的机器代码为1446FFFAH，根据题目中给出的指令格式，后2B的内容为OFFSET字段，所以该指令的OFFSET字段为FFFAH，用补码表示，值为-6.  
      当系统执行到bne指令时，PC自动+4，PC的内容就是08048118H，跳转目标是08048100H，两者差了18H，即24个单位的地址间隔，所以偏移址的一位即是真是跳转地址的-24/-6=4位。可知bne指令的转移目标地址计算公式：(PC)+4+OFFSET\*4
  + 若M采用如下“按序发射、按序完成”的5级指令流水线：IF（取值）、ID（译码及取数）、EXE（执行）、MEM（访存）、WB（写回寄存器），且硬件不采取任何转发措施（数据旁路），分支指令的执行均引起3个时钟周期的阻塞，则P中哪些指令的执行会由于数据相关而发生流水线阻塞？哪条指令的执行会发生控制冒险？为什么指令1的执行不会因为与指令5的数据相关而发生阻塞？
    - 由于数据相关而发生阻塞的指令为第2，3，4，6条，因为第2，3，4，6条指令都与各自前一条指令发生数据相关。
    - 第6条指令会发生控制冒险。
    - 当前循环的第5条指令与下次循环的第一条指令虽然有数据相关，但由于第6条指令后有3个时钟周期的阻塞，因而消除了该数据相关

假设对于上题中的计算机M和程序P的机器代码，M采用页式虚拟存储管理；P开始执行时，(R1)=(R2)=0，(R6)=1000，其机器代码已调入主存但不在Cache中；数组A未调入主存，且所有数组元素在同一页，并存储在磁盘同一个扇区。请回答下列问题并说明理由。

* + P执行结束时，R2的内容是多少？
    - 1000
  + M的指令Cache和数据Cache分离。若指令Cache共有16行，Cache和主存交换的块大小为32字节，则其数据区的容量是多少？若仅考虑程序段P的执行，则指令Cache的命中率为多少？
    - Cache共16行，每块32字节，所以Cache数据区的容量为16\*32B=512B
    - P共有6条指令，占24B，小于主存块32B，其起始地址为08048100H，对应一个块的开始位置，由此可知所有指令在一个主存块内。读取第一条指令发生Cache缺失，故将P所在的主存块调入Cache某一行，以后每次读取指令时，都能在指令Cache中命中。因此在1000次循环中，只会发生一次指令访问缺失，
    - 命中率：(1000\*6-1)/(1000\*6)=99.89%
  + P在执行过程中，哪条指令的执行可能发生溢出异常？哪条指令的执行可能产生缺页异常？对于数组A的访问，需要读磁盘和TLB至少各多少次？
    - 指令4为加法指令，即对应sum+=A[i]，当数组A中元素的值过大时，则会导致这条加法指令发生溢出异常；而指令2、5虽然都是加法指令，但他们分别为数组地址的计算指令和存户变量i的寄存器进行自增的指令，而i最大到达1000，所以他们不会产生溢出
    - 只有访存指令可能出现缺页异常，即指令3可能产生缺页异常。因为数组A在磁盘的一页上，而一开始数组并不在主存中，第一次访问数组时会导致访盘，把A调入内存，而以后数组A的元素都在内存中，则不会导致访盘，所以该程序一共访盘一次
    - 没访问一次内存数据就会查TLB一次，共访问数组1000次，所以此时访问TLB1000次，还要考虑第一次访问数组A，即访问A[0]时，会多访问一次TLB：第一次访问A[0]，会先查一次TLB，然后产生缺页，处理完缺页中断后，会重新访问A[0]，此时又查TLB。所以一共访问TLB1001次

#### 小结

指令周期是否是固定值：

* + 指令周期对于不同指令来说，不是一个固定值

指令相关、数据相关

* + 两条连续的指令读取相同的寄存器，就会产生RAR（读后读）相关，这种相关不会影响流水线
  + 当某条指令要读取上一条指令所写入的寄存器时，就会产生RAW（写后读）相关，这种称**数据相关**或真相关，影响流水线
  + 当某条指令的上条指令要读/写该指令的输出寄存器时，就会产生WAR（读后写）和WAW（写后写）相关

对流水线影响最严重的指令相关是数据相关

### 总线