计算机组成与结构
—指令系统1

计算机科学与技术学院

段式虚拟存储器

将程序按逻辑意义分成段，按段进行调入、调出和管理。

主要优点：

1、程序的模块化性能好。——**（将程序按逻辑意义划分）**

2、便于多道程序共享主存中的某些段。——**（例如共用子函数或者数据）**

3、程序的动态链接和调度比较容易。——**（动态链接库.dll）**

4、便于按逻辑意义实现存储器的访问方式保护。——**（段表支持存储不同的访问方式）**

主要缺点：

1、地址变换所花费的时间长。——**（两次加法）**

2、段映象表庞大，地址、段长字段太长。——（**起址可以使任意一个主存地址，段表起址与主存地址位数相同**）

3、主存储器的利用率往往比较低。——存储管理复杂；**段间“零头”**

4、对辅存（磁盘存储器）的管理比较困难。——**磁盘整块存取(512Kb),段大小可变，“零头”为1Kb，存取管理困难。**

页式虚拟存储器

将物理空间和VM空间都等分成相同大小的页面(Page)

主要优点：

① 主存储器的利用率比较高。——段内分页，按页为大小存储到主存

② 页表相对比较简单，使用硬件少。——页号、主存页号

③ 地址变换的速度比较快。——一次加法，直接拼接

④ 对磁盘的管理比较容易。——支持按页存取，“零头”存储在一页

主要缺点：

① 程序的模块化性能不好。——每段内分页

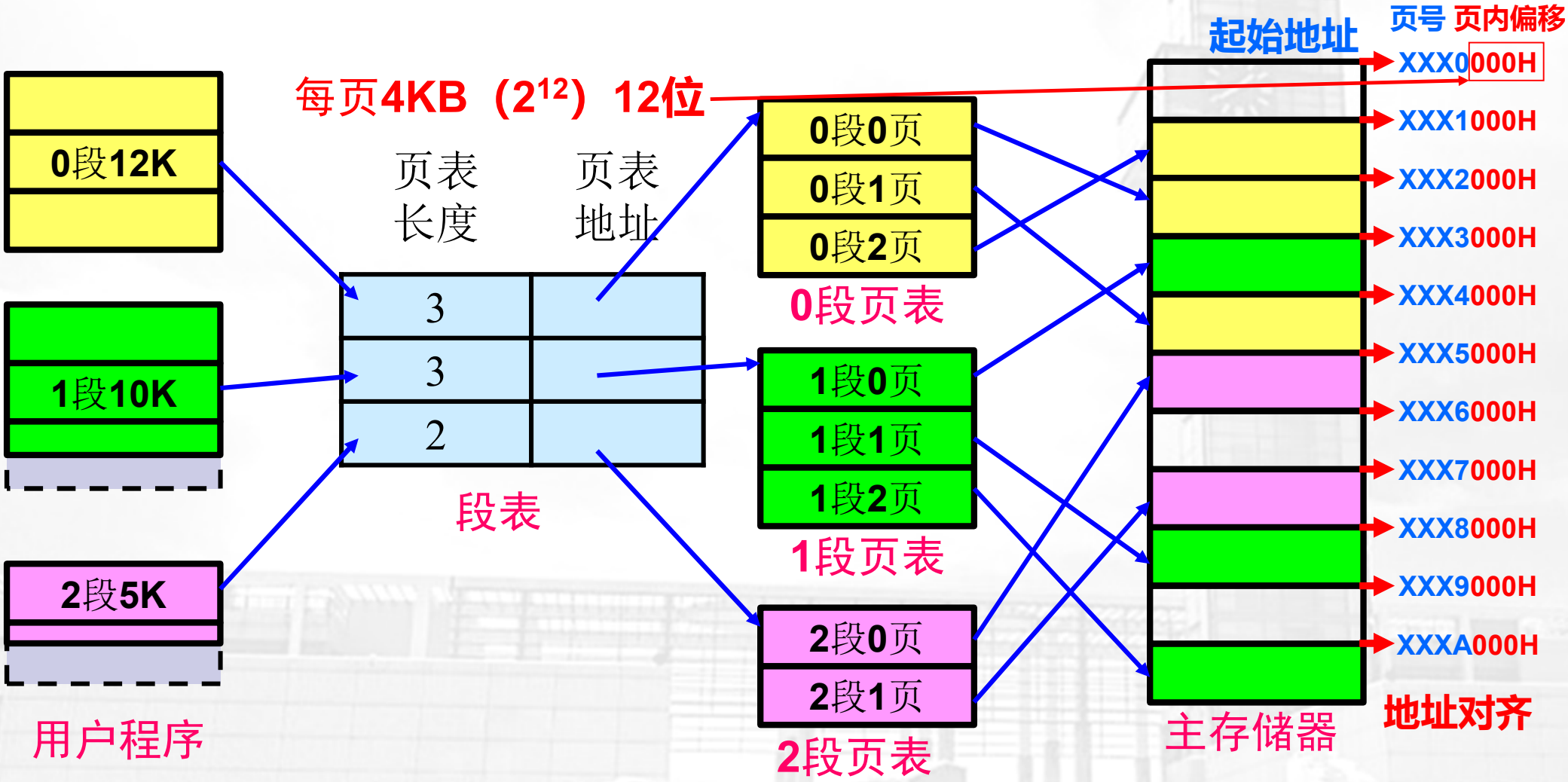
② 页表很长，需要占用很大的存储空间。——页空间小，页数量大

例如：虚拟存储空间4GB，页大小1KB，则页表的容量为4M存储字。如果每个页表存储字占用4个字节，则页表的存储容量为16MB。

段页式虚拟存储器

地址映象方法：

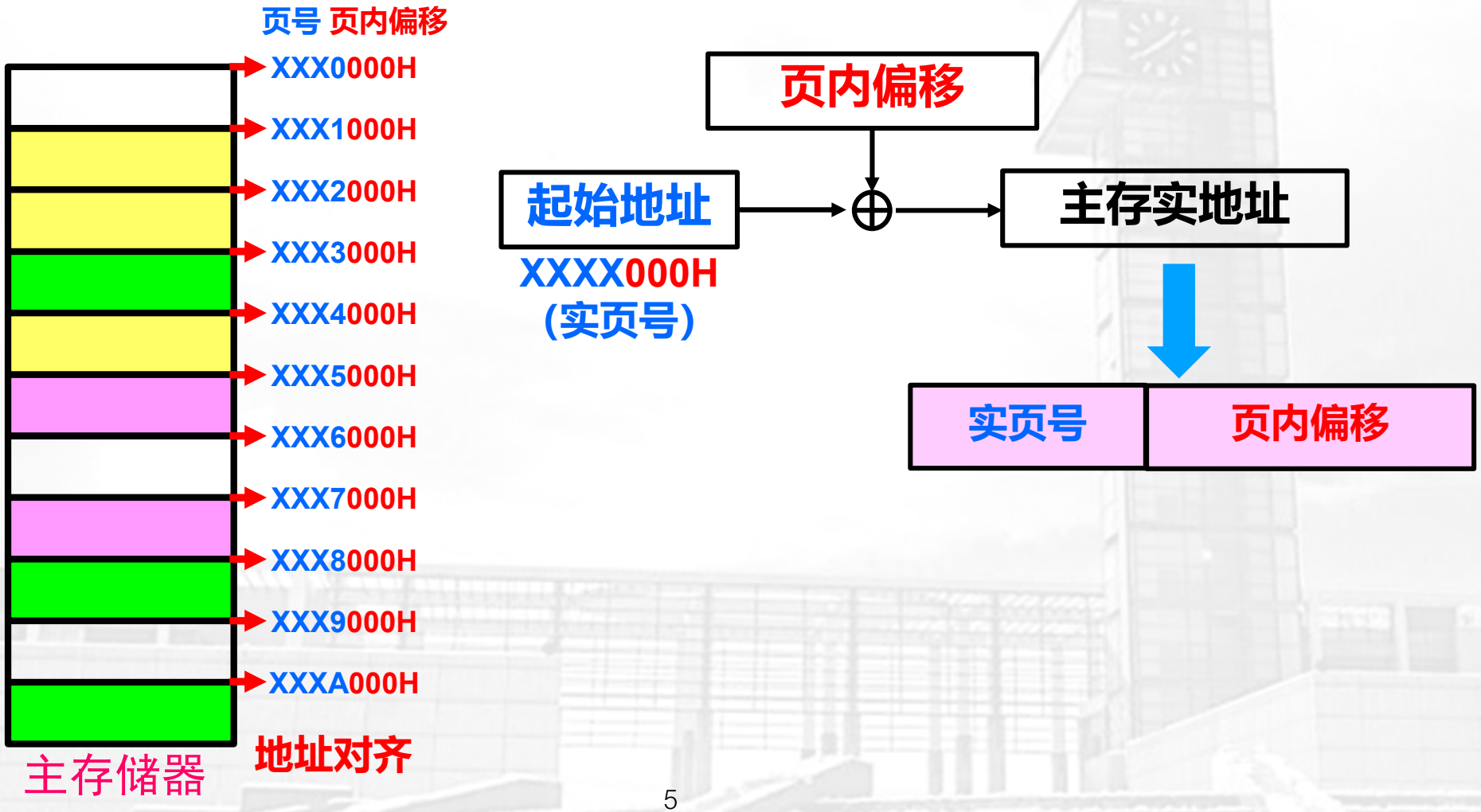
每个程序段在段表中占一行， 在段表中给出页表长度和页表的起始地址，页表中给出每一页在主存储器中的实页号。

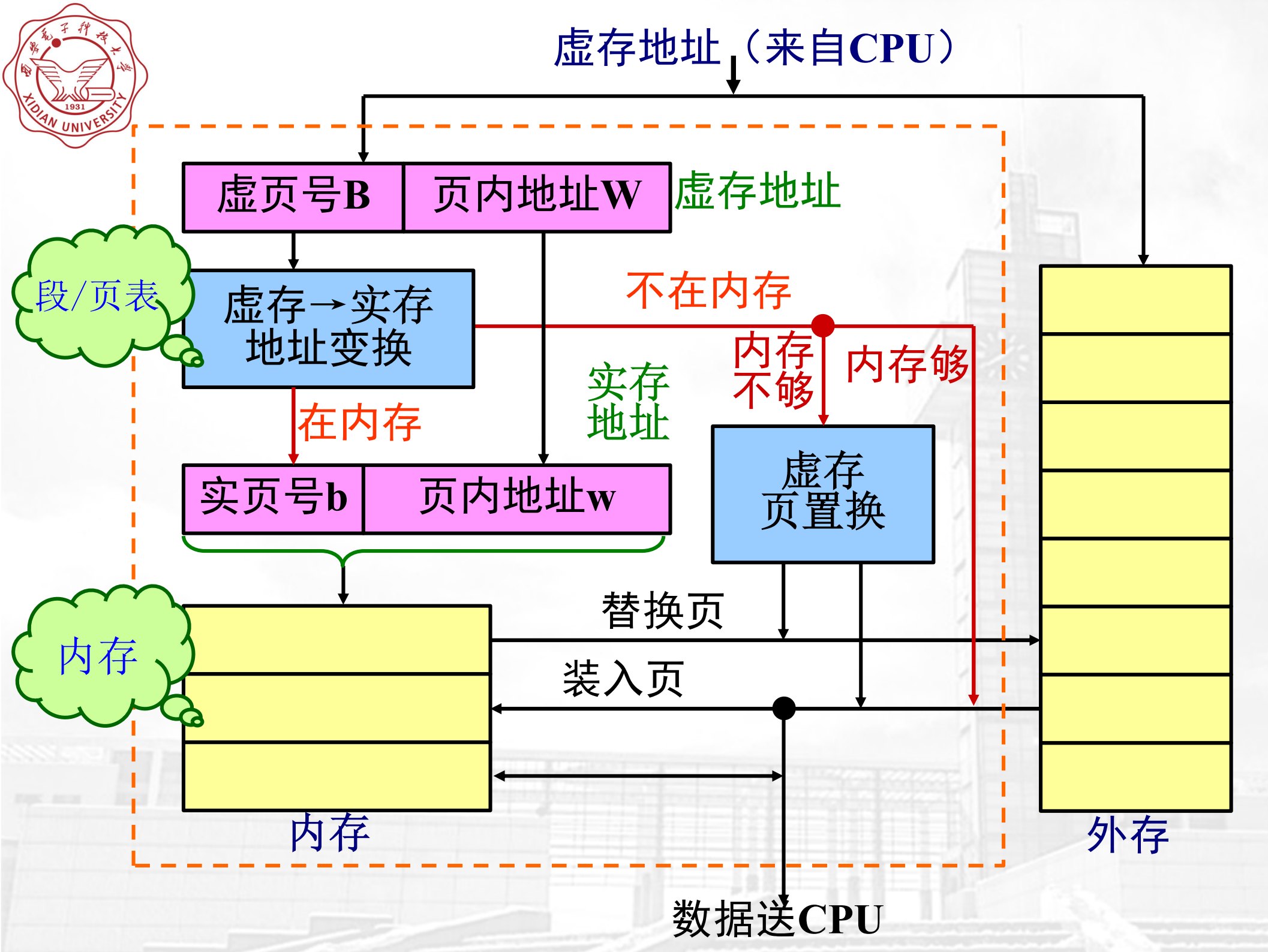


段页式虚拟存储器

地址映象方法：

每个程序段在段表中占一行， 在段表中给出页表长度和页表的起始地址，页表中给出每一页在主存储器中的实页号。




虚存地址（来自CPU）
虚页号B
页内地址W
虚存地址
段/页表
虚存→实存
地址变换
不在内存
内存
不够
内存够
实存
地址
在内存
虚存
页置换
实页号b
页内地址w
替换页
内存
装入页
内存
外存
数据送CPU

VM访存流程

- CPU给出虚拟地址
  - 查段/页表
    - 如果在内存(有效位=0)，访问
    - 如果所需页面不在内存(有效位=0)…
      - CPU发出Page Fault(PF)中断
      - 启动操作系统注册的FP中断处理程序
      - 中断处理程序从外存调数据进内存（文件系统等…）
      - 操作系统更新页表（有效位=1）
      - 操作系统让切换回原进程

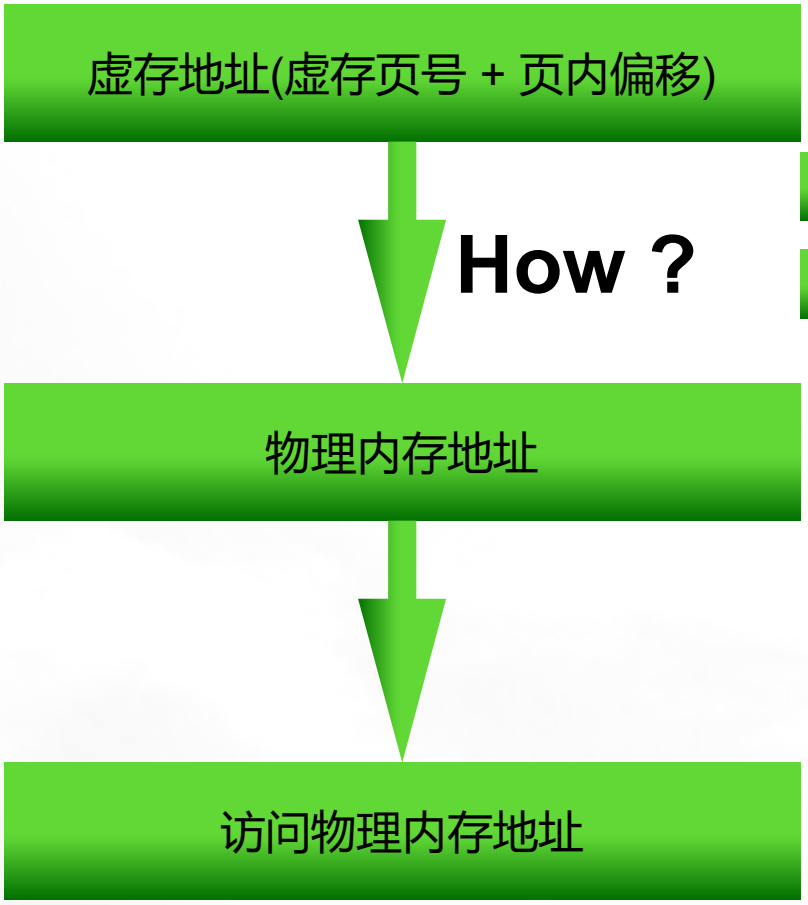
VM页替换

- 物理内存不足时，将暂时不用的物理页替换到外存
- 虽然慢，但可以维持系统运行
- 常规策略
  - LRU，最近最少使用
    - 需要为每一页记录时间戳
  - FIFO，先进先出

Big Picture of VM

- 映射方式
  - 段、页、段页
- 替换：主存空间不够时置换部分页到外存
- VM实现机制
  - 查表 + Page Fault缺页中断
- 访存过程：映射、查找、读写、替换

VM模型下的内存访问



1. 找到当前进程的页表/段页表
2. 查表 + 计算

不难推测几点：

1. **地址转换(or 计算)成为CPU极频繁的工作；**
2. **页表被极频繁的访问；**
3. **so, 大量页表内容进驻cache；**
4. **cache容量有限，进程指令/数据的cache被挤占；**
5. **更可能的情况是，页表体积常远大于cache，使页表本身都无法快速访问；**
6. **系统性能严重下降**
7. **极端情况：页表都无法完全放在内存中，可能被置换到外存**

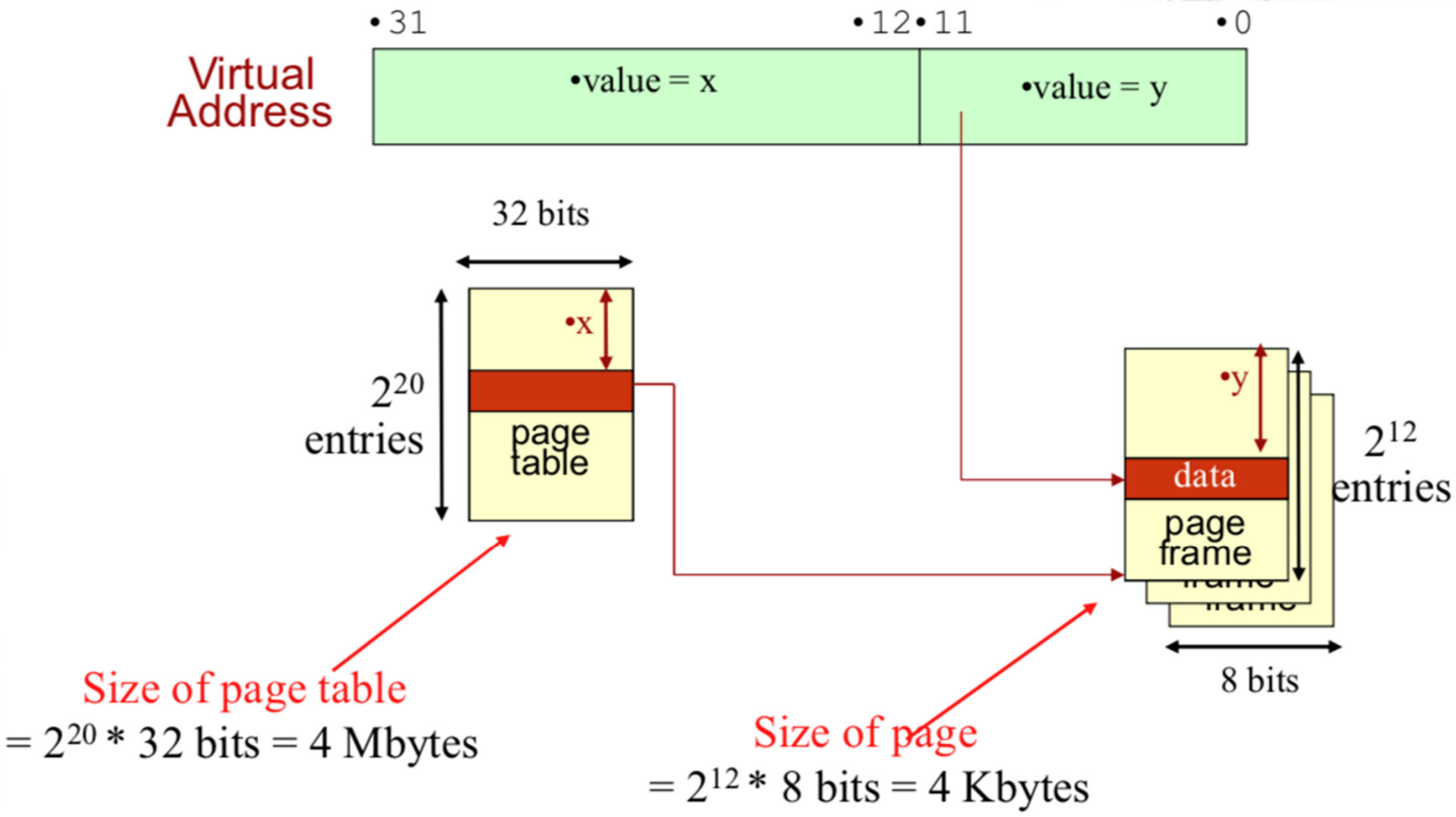
So How to accelerate ?

VM模型下的内存访问

- **优化一：用专用的内存管理单元(MMU)硬件处理VM地址变换**
  - **MMU是专用硬件**
  - **页表被存储在特殊内存位置PTBR**

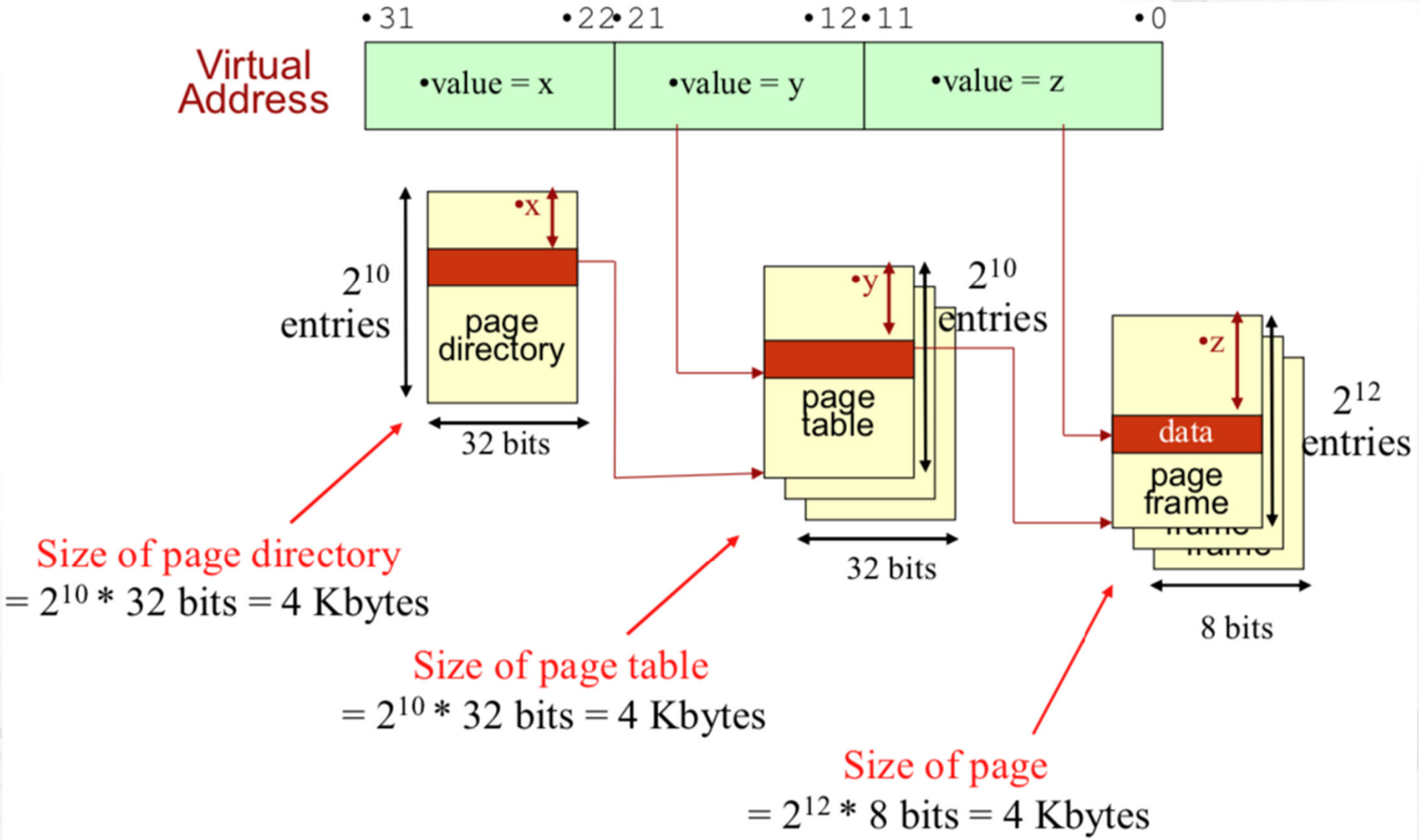
VM模型下的内存访问

- **优化二：单极页表太大，怎么办?**



VM模型下的内存访问

- **优化二：两级页表！**

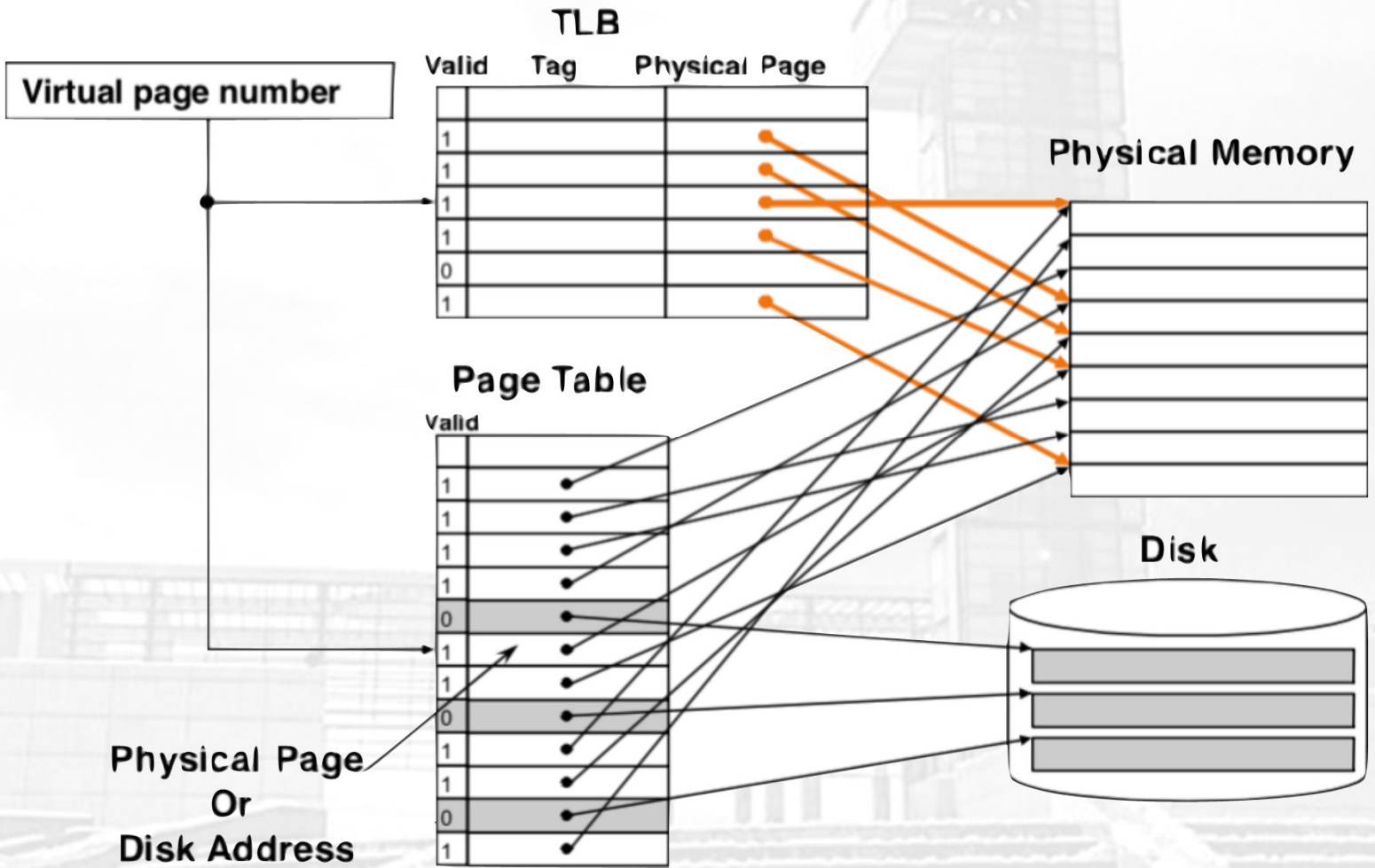


VM模型下的内存访问

- **优化三：页表在内存访问慢，在Cache挤程序空间，怎么办?**

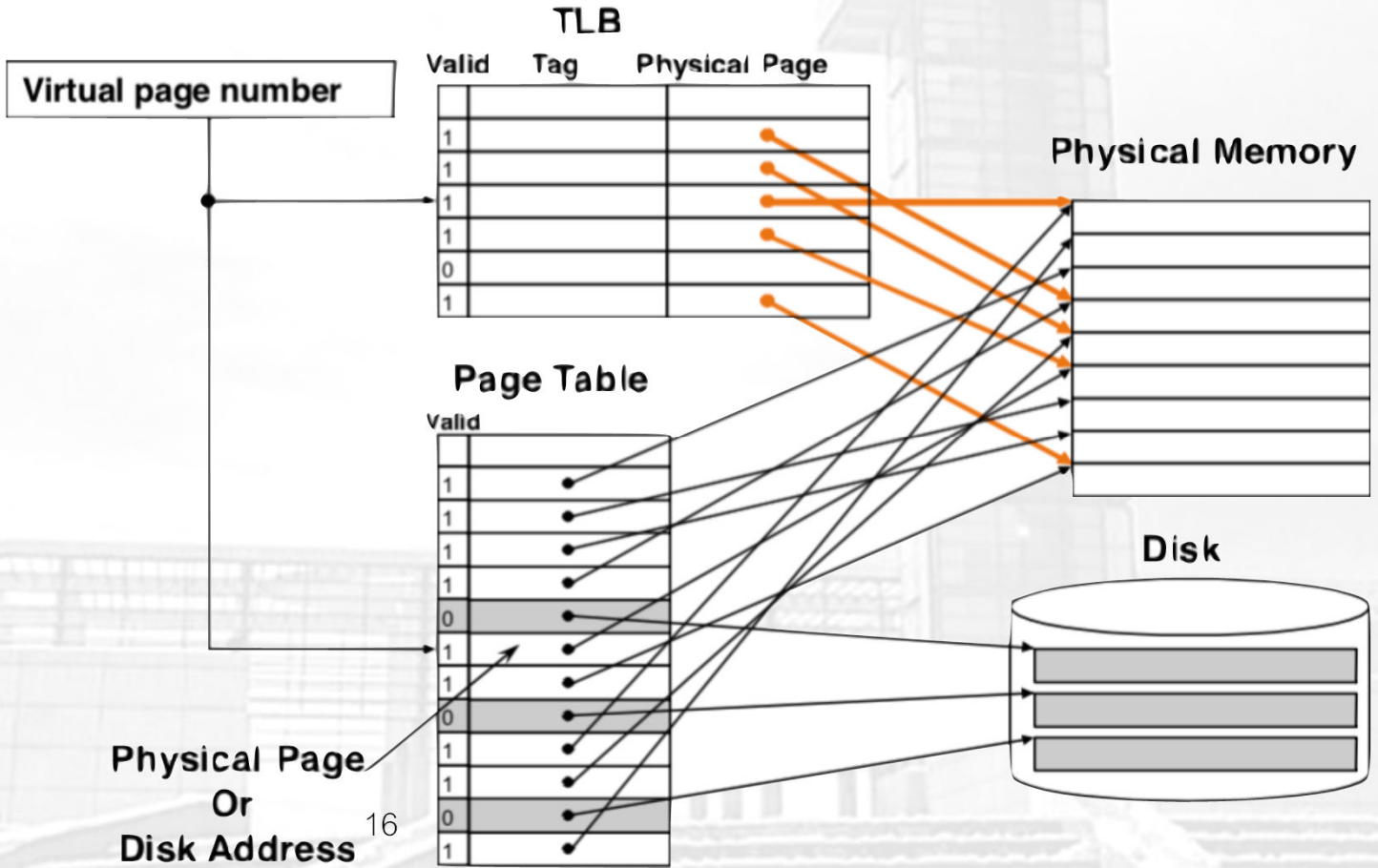
VM模型下的内存访问

- 优化三：给页表加一个专用的Cache——TLB
  - Translation Look-aside Buffers
  - 比L1 Cache的级别更高！
    - 在CPU和Cache之间
  - 同样有hit rate/ miss rate的指标



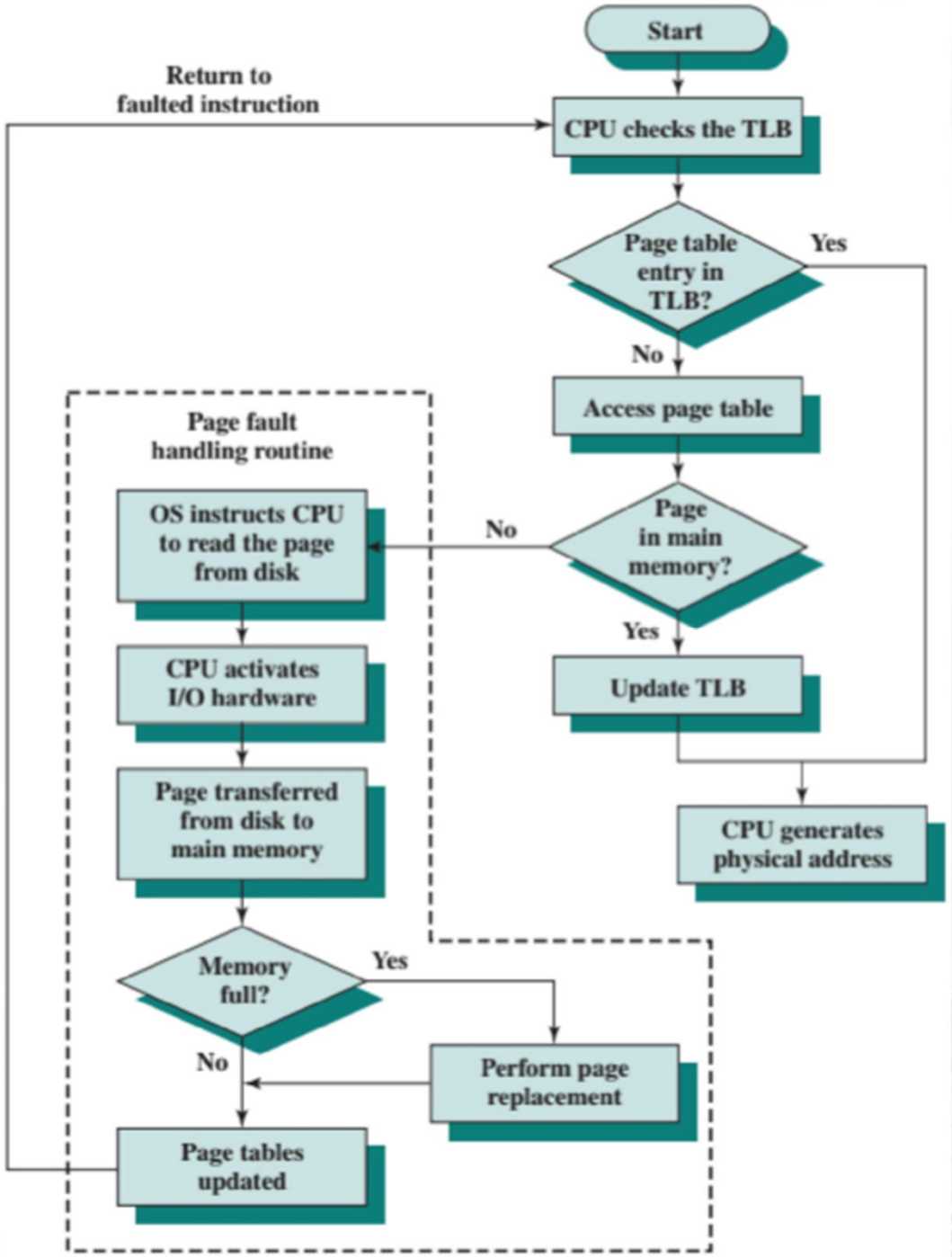
VM模型下的内存访问

- 优化三：给页表加一个专用的Cache——TLB
  - 因为TLB是个Cache…
    - TLB的映射方式：1/2/4/8路组相联
    - TLB的替换策略：随机、FIFO、LRU
    - TLB也搞成多级TLBs….
    - TLB又优化为I-TLB, D-TLB…



VM模型下的内存访问

- **最终的内存访问流程~**

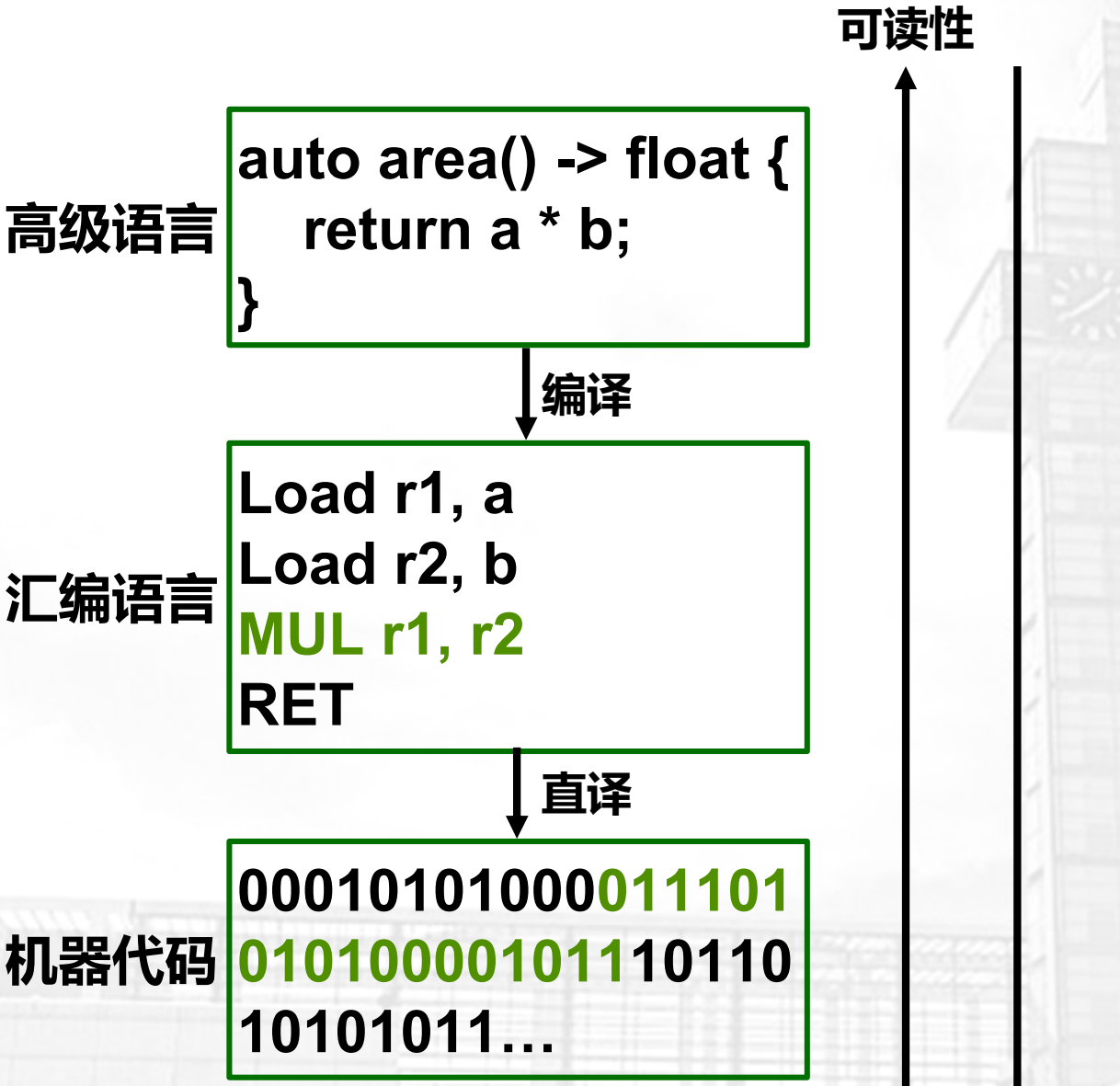


第七章作业

- 6
- 16

指令系统序言

Code, 2 Code


可读性
高级语言
auto area() -> float {
return a * b;
}
编译
汇编语言
Load r1, a
Load r2, b
MUL r1, r2
RET
直译
机器代码
00010101000011101
01010000101110110
10101011…
执行速度

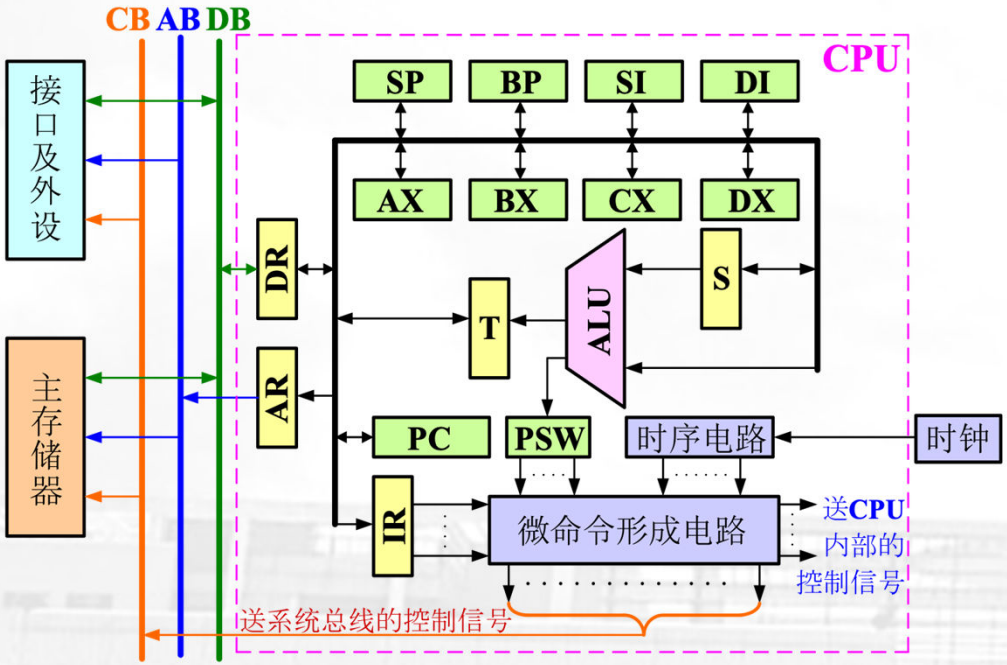
本章内容简介

操作本身，作为指令，存放在内存

高级语言

```
auto area() -> float {
    return a * b;
}
```

一般情况下a和b也在内存



试分析一下，为了执行乘法，至少需要哪些原子操作?

1. 需要从PC所指的内存中，取出当前指令
2. 识别这个指令，发现它有两个操作数
3. 从内存&a地址取a，放在ALU的输入寄存器S;
4. 从内存&b地址取b，放在一个靠近ALU的某个临时寄存器，如AX;
5. 把AX的值送入ALU
6. ALU计算乘法，结果存在临时寄存器T;
7. 把T赋值给其它寄存器
8. 结束函数/子程序，返回

本章内容简介

试分析一下，为了执行乘法，至少需要哪些原子操作?

1. **需要从PC所指的内存中，取出当前指令**
2. **识别这个指令，发现它有两个操作数**
3. **从内存&a地址取a，放在ALU的输入寄存器S;**
4. **从内存&b地址取b，放在一个靠近ALU的某个临时寄存器，如AX;**
5. **把AX的值送入ALU**
6. **ALU计算乘法，结果存在临时寄存器T;**
7. **把T赋值给其它寄存器**
8. **结束函数/子程序，返回**

4.1 机器指令的格式与编码

本章内容简介

试分析一下，为了执行乘法，至少需要哪些原子操作?

1. 需要从PC所指的内存中，取出当前指令
2. 识别这个指令，发现它有两个操作数
3. 从内存&a地址取a，放在ALU的输入寄存器S；
4. 从内存&b地址取b，放在一个靠近ALU的某个临时寄存器，如AX;
5. 把AX的值送入ALU
6. ALU计算乘法，结果存在临时寄存器T；
7. 把T赋值给其它寄存器
8. 结束函数/子程序，返回

- 4.1 机器指令的格式与编码
- 4.2 操作数的取值/取址

本章内容简介

试分析一下，为了执行乘法，至少需要哪些原子操作?

1. 需要从PC所指的内存中，取出当前指令
2. 识别这个指令，发现它有两个操作数
3. 从内存&a地址取a，放在ALU的输入寄存器S;
4. 从内存&b地址取b，放在一个靠近ALU的某个临时寄存器，如AX;
5. 把AX的值送入ALU
6. ALU计算乘法，结果存在临时寄存器T;
7. 把T赋值给其它寄存器
8. 结束函数/子程序，返回

- 4.1 机器指令的格式与编码
- 4.2 操作数的取值/取址
- 4.3 8086系统的指令系统

本章内容简介

试分析一下，为了执行乘法，至少需要哪些原子操作?

1. 需要从PC所指的内存中，取出当前指令
2. 识别这个指令，发现它有两个操作数
3. 从内存&a地址取a，放在ALU的输入寄存器S;
4. 从内存&b地址取b，放在一个靠近ALU的某个临时寄存器，如AX;
5. 把AX的值送入ALU
6. ALU计算乘法，结果存在临时寄存器T;
7. 把T赋值给其它寄存器
8. 结束函数/子程序，返回

- 4.1 机器指令的格式与编码
- 4.2 操作数的取值/取址
- 4.3 8086系统的指令系统
- 4.4 汇编语言程序设计

本章内容简介

试分析一下，为了执行乘法，至少需要哪些原子操作?

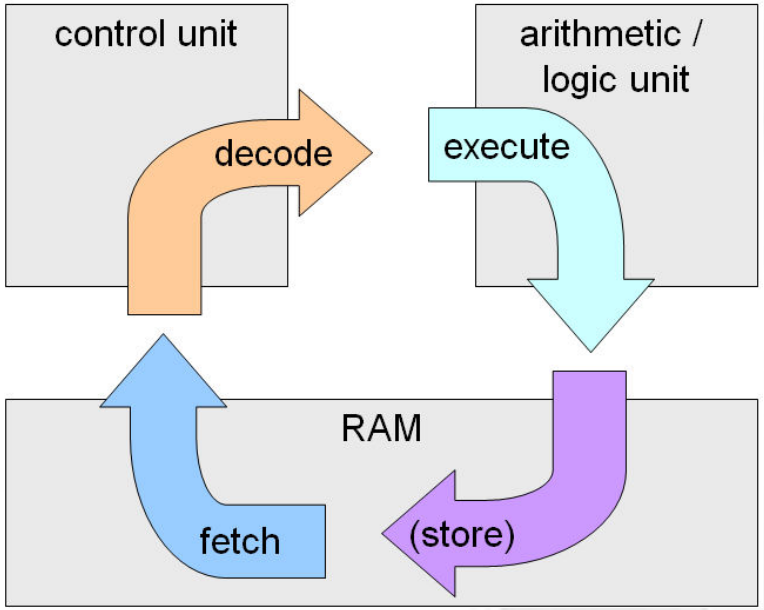
1. 需要从PC所指的内存中，取出当前指令
2. 识别这个指令，发现它有两个操作数
3. 从内存&a地址取a，放在一个靠近ALU的某个临时寄存器，如AX;
4. 从内存&b地址取b，放在ALU的输入寄存器S;
5. 把AX的值送入ALU
6. ALU计算乘法，结果存在临时寄存器T;
7. 把T赋值给其它寄存器
8. 结束函数/子程序，返回

本章所有内容:

- 4.1 机器指令的格式与编码
- 4.2 操作数的取值/取址
- 4.3 8086系统的指令系统
- 4.4 汇编语言程序设计
- 4.5 RISC/CISC科普

本章内容简介

CPU执行指令的3(4)流程循环:
取指令
译码
执行
（存储）



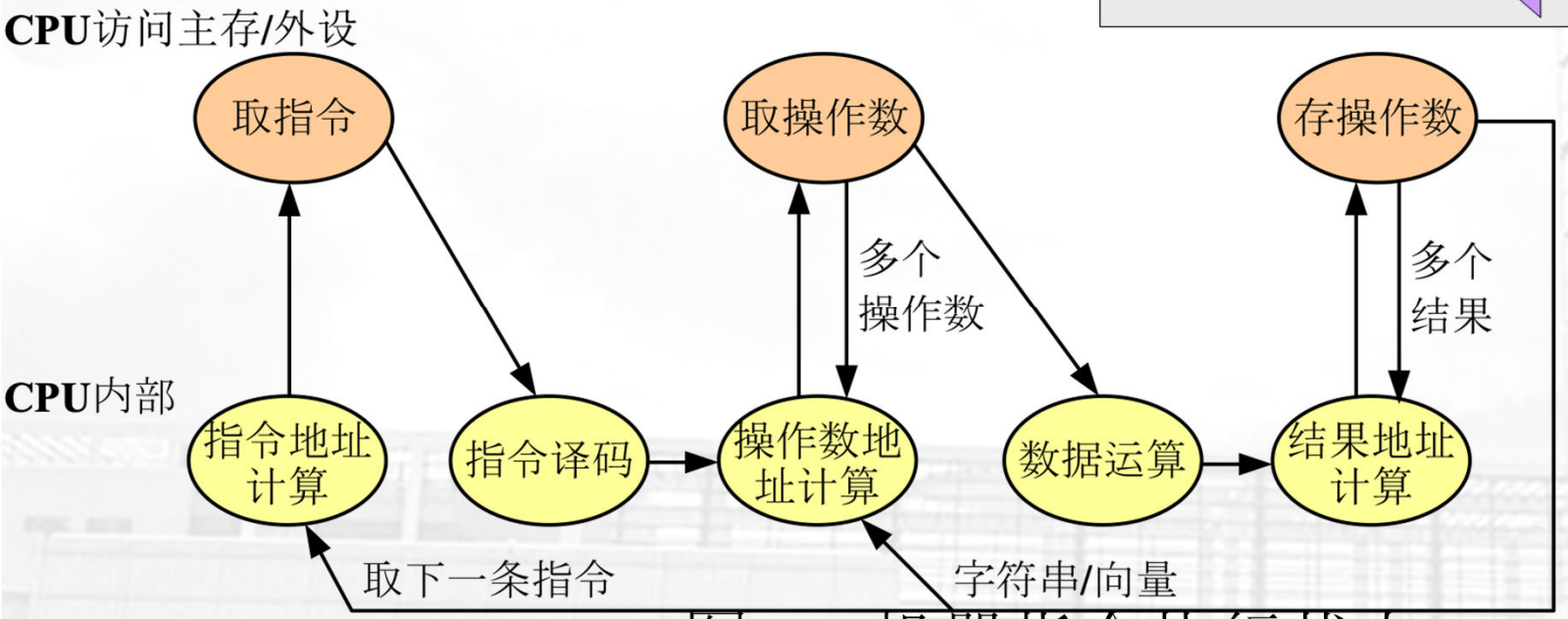


图5.1 机器指令执行状态

指令格式

指令格式

高级语言

```
auto area() -> float {
    return a * b;
}
```

编译

汇编语言

```
Load r1, a
Load r2, b
MUL r1, r2
RET
```

直译

机器代码

```
00010101000011101
01010000101110110
10101011…
```

MUL r1, r2被转译为

“0111010101000010 11” 二进制

这一条指令包含以下4条信息：

- MUL指令的编号
- MUL指令能支持两个操作数
- 操作数1指向r1寄存器
- 操作数2指向r2寄存器

指令格式

- 指令 = 操作码Opcode + 操作数 Operand (0~N)
- 操作数，也叫地址码

地址码的内容可以是：地址、数值

地址码的目标可以是：

源/目的操作数，结果，下一条指令地址（顺序/跳转）

例如，某定长指令格式

指令格式

■常用的地址码数目：

- 三地址：ADD Y, A, B ；Y=A+B （RISC）
- 双地址：ADD Y, A ；Y=Y+A （CISC）
- 单地址：ADD C ；AC=AC+C
- 零地址

指令格式

例如：计算一个典型的算术表达式：

$$x = \frac{a \times b + c - d}{e + f}$$

用三地址指令编写的程序如下：

```
MUL   X, A, B      ;X←(A)×(B)
ADD   X, X, C      ;X←(X)+(C)
SUB   X, X, D      ;分子的计算结果在X中
ADD   Y, E, F      ;计算分母,存入Y
DIV   X, X, Y      ;最后结果在X单元中
```

指令格式

$$x = \frac{a \times b + c - d}{e + f}$$

用普通二地址指令编写的程序：

```
MOVE  X, A      ;复制临时变量到X中
MUL   X, B
ADD   X, C
SUB   X, D      ;X中存放分子运算结果
MOVE  Y, E      ;复制临时变量到Y中
ADD   Y, F      ;Y中存放分母运算结果
DIV   X, Y      ;最后结果在X单元中
```

指令格式

$$x = \frac{a \times b + c - d}{e + f}$$

用多寄存器结构的二地址指令编写程序：

```
MOVE  R1, A     ;操作数a取到寄存器R1中
MUL   R1, B
ADD   R1, C
SUB   R1, D     ;R1中存放分子运算结果
MOVE  R2, E
ADD   R2, F     ;R2中存放分母运算结果
DIV   R1, R2    ;最后结果在R1中
MOVE  X, R1     ;最后结果存入X中
```

指令格式

$$x = \frac{a \times b + c - d}{e + f}$$

用一地址指令编写的程序：

```
LOAD   E     ;先计算分母，
             ;取一个操作数到累加器中
ADD    F     ;分母运算结果在累加器中
STORE  X     ;保存分母运算结果到x中
LOAD   A     ;开始计算分子
MUL    B
ADD    C
SUB    D     ;累加器中是分子运算结果
DIV    X     ;最后运算结果在累加器中
STORE  X     ;保存最后运算结果到x中
```

指令格式

用0地址指令编写程序：

```
PUSH   A      ;操作数a压入堆栈
PUSH   B      ;操作数b压入堆栈
MUL           ;栈顶两数相乘，结果压回堆顶
PUSH   C
ADD
PUSH   D
SUB           ;栈顶是分子运算的结果
PUSH   E
PUSH   F
ADD
DIV           ;栈顶是最后运算的结果
POP    X      ;保存最后运算结果
```

$$x = \frac{a \times b + c - d}{e + f}$$

```
ab*c+d-ef+/
```

指令格式

关于地址码个数结论：

- 对于一般商用处理机，采用多寄存器结构的二地址指令是最理想的。
- 如果强调硬件结构简单，并且以连续运算（如求累加和等）为主，宜采用一地址结构。
- 对于以向量、矩阵运算为主的处理机，最好采用三地址结构。
- 部分RISC处理机也采用三地址指令。
- 对于解决递归问题为主的处理机，宜采用零地址结构。编程容易、节省程序存储量。

指令编码

指令编码

- **指令操作码(Op code)编码方式:**
- **编码的考量因素**
- **编码方法**
  - **定长编码**
  - **不定长编码**
    - **霍夫曼Huffman Coding**
    - **扩展操作码**

指令编码

构成一条指令的二进制的位数称为指令长度，一般都是字节的整数倍。

- 内存大小与组织——内存地址位32位，采用直接寻址，需要32位编码
- CPU数据总线宽度——指令长度为数据总线宽度整数倍，避免浪费
- CPU内部寄存器的数量——数量越多，需要的编码越长，功能越强
- 寻址方式——方式越多，编码越长，功能越强
- 指令数量——数量越多，编码越长

指令编码

定长指令编码：

- **所有指令的编码Opcode长度相同！**
  - **不管几操作数的指令，opcode都一样长！**
- **优点**
  - **操作码构造简单、硬件设计简单、译码速度块**
- **缺点**
  - **存储空间大、指令扩充有限**

指令编码

Huffman指令编码：

- **让高频指令的opcode更短！（不关心操作数长度！）**

指令编码

例：假设一台模型计算机共有7种不同的操作码，已知各种操作码在程序中出现的概率如下表，利用固定长度编码法进行操作码编码。

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.45	0.30	0.15	0.05	0.03	0.01	0.01

解：由于N=7 因此，指令操作码固定长度为

[lbN]=[lb7]=3

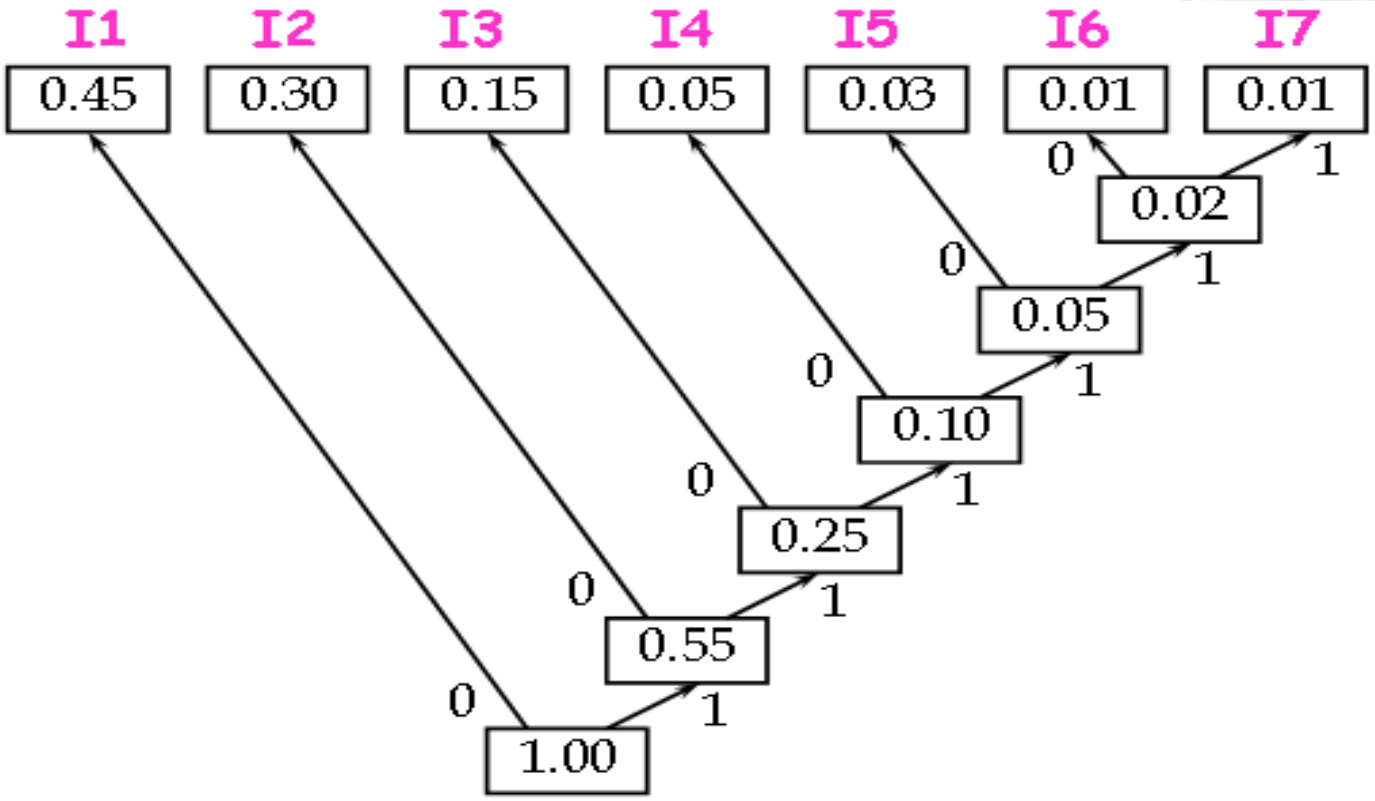
编码结果：

指令序号	概率	编码	操作码长度
I_1	0.45	000	3位
I_2	0.30	001	3位
I_3	0.15	010	3位
I_4	0.05	011	3位
I_5	0.03	100	3位
I_6	0.01	101	3位
I_7	0.01	110	3位

指令编码

例：利用Huffman 编码法进行操作码编码。

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.45	0.30	0.15	0.05	0.03	0.01	0.01



Huffman编码树生成过程

指令编码

编码结果:

指令序号	概率	Huffman编码法	操作码长度
I_1	0.45	0	1位
I_2	0.30	10	2位
I_3	0.15	110	3位
I_4	0.05	1110	4位
I_5	0.03	11110	5位
I_6	0.01	111110	6位
I_7	0.01	111111	6位

采用Huffman编码法的操作码平均长度:

$0.45 \times 1 + 0.30 \times 2 + 0.15 \times 3 + 0.05 \times 4 + 0.03 \times 5 + 0.01 \times 6 + 0.01 \times 6$

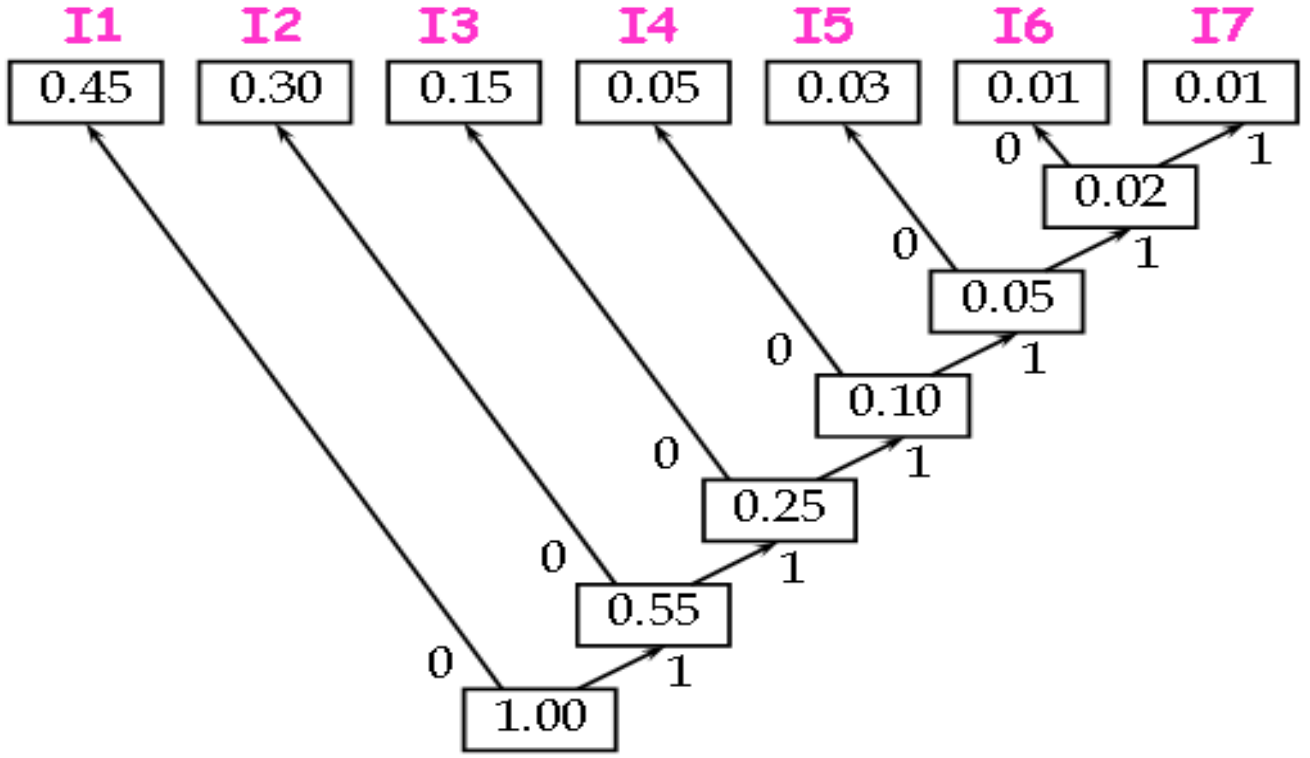
$= 1.97$（位）

指令编码

Huffman操作码的主要缺点：

1) 操作码长度很不规整，硬件译码困难

2) 与地址码共同组成固定长的指令比较困难



Huffman编码树生成过程

特性：短码不能是长码的前缀

解决方法之一：　指令操作码的扩展技术

指令编码

扩展指令编码：

- **让操作数多的指令opcode更短**

操作码	地址码1	地址码2

- **操作数少的指令opcode最长**

操作码	地址码1

短码(a)：1110

长码(b)：1110 0000

歧义?

原则：短码不能是长码的前缀。

扩展指令编码

15-15-15 编码方法

15(A)
- 0000
- 0001
- ……
- 1110

15 (B)
- 1111 0000
- 1111 0001
- ……
- 1111 1110

15 (C)
- 1111 1111 0000
- 1111 1111 0001
- ……
- 1111 1111 1110

8-64-512 编码方法

8 (A)
- 0000
- 0001
- ……
- 0111

64 (B)
- 1000 0000
- 1000 0001
- ……
- 1111 0111

512 (C)
- 1000 1000 0000
- 1000 1000 0001
- ……
- 1111 1111 0111

指令编码

- 短的操作码与多种地址码配合
- 长的操作码与简单地址码组合
- 指令长度一般设计为总线宽度的整数倍
- 指令长度为最小可寻址单位的整数倍
- 影响寻址位的因素：
  - 寻址方式的个数
  - 操作数数目
  - 寄存器数目
  - 地址范围
  - 地址可寻址最小单元

指令编码

例：假设某型号的计算机共有14条指令，各条指令的使用频率分别为：0.01、0.15、0.12、0.03、0.02、0.04、0.02、0.04、0.01、0.13、0.15、0.14、0.11、0.03。

试给出定长、 Huffman 、只能有两种码长的扩展操作码的三种编码方案，并计算各种方案的平均码长。

【解】

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.15	0.15	0.14	0.13	0.12	0.11	0.04
指令	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}
概率	0.04	0.03	0.03	0.02	0.02	0.01	0.01

指令编码

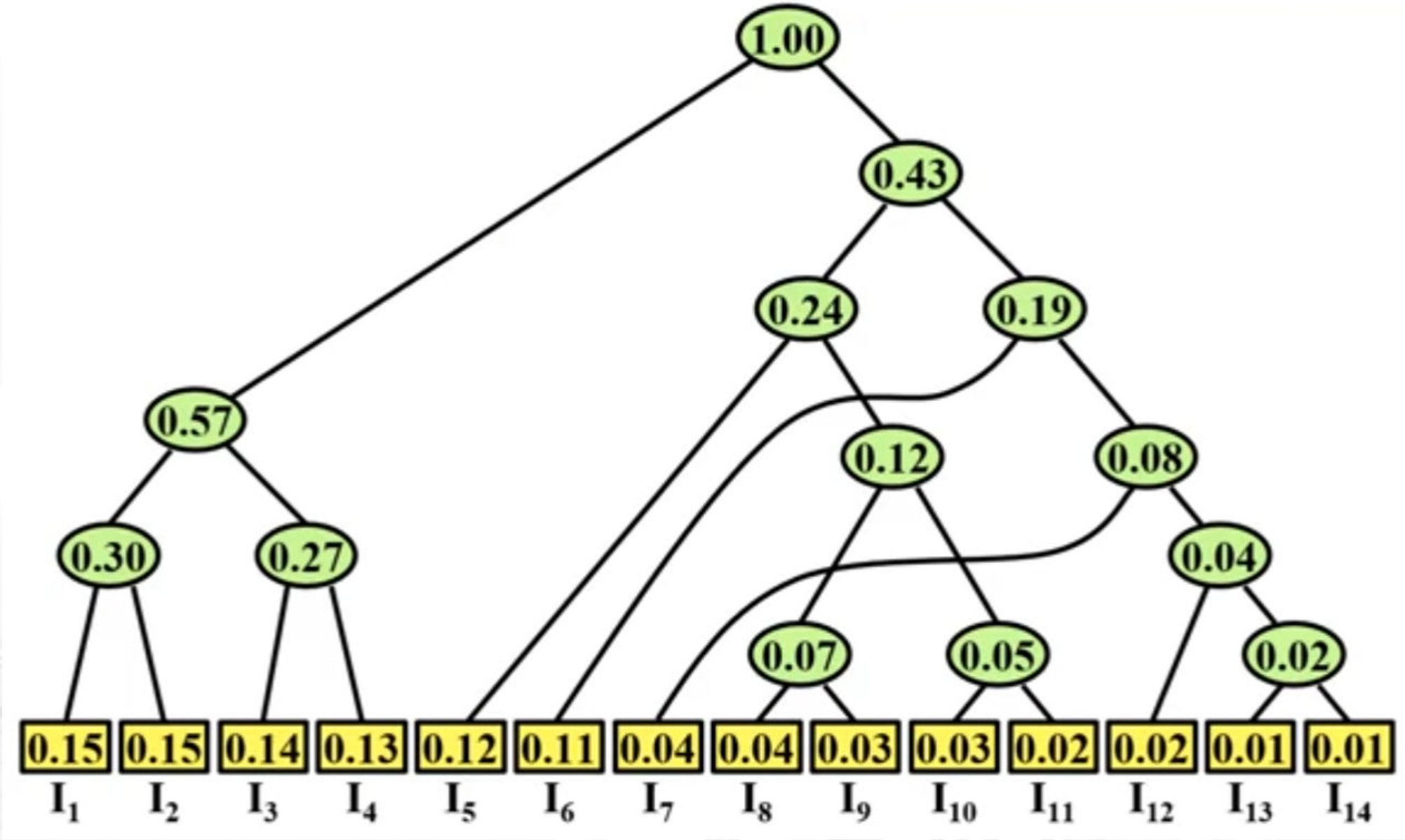
定长编码：

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.15	0.15	0.14	0.13	0.12	0.11	0.04
扩展编码	0000	0001	0010	0011	0100	0101	0110
码长	4	4	4	4	4	4	4
指令	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}
概率	0.04	0.03	0.03	0.02	0.02	0.01	0.01
扩展编码	0111	1000	1001	1010	1011	1100	1101
码长	4	4	4	4	4	4	4

其操作码的平均码长：**H＝4**（位）。

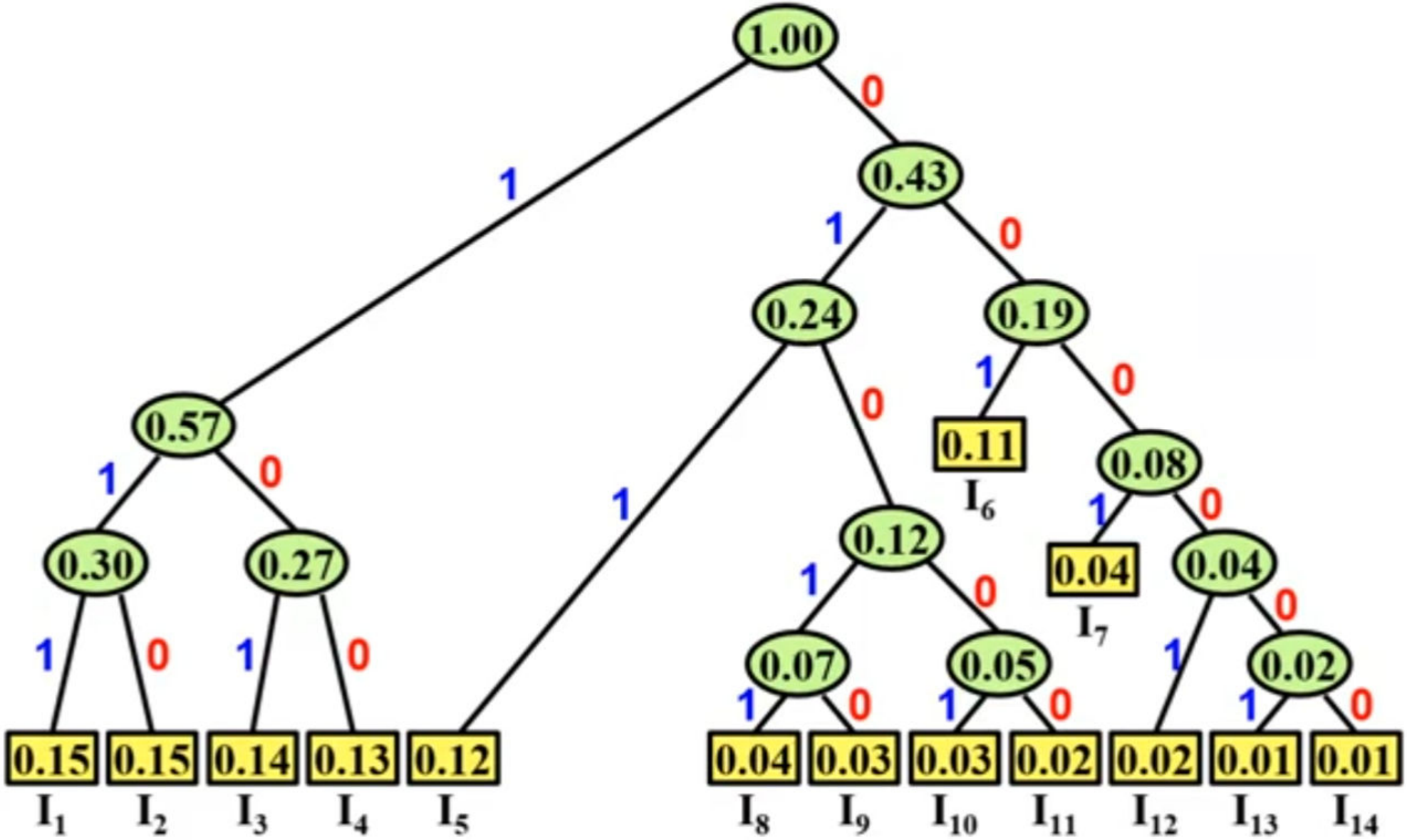
指令编码

Huffman



指令编码

Huffman



指令编码

Huffman编码：

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.15	0.15	0.14	0.13	0.12	0.11	0.04
扩展编码	111	110	101	100	011	001	0001
码长	3	3	3	3	3	3	4
指令	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}
概率	0.04	0.03	0.03	0.02	0.02	0.01	0.01
扩展编码	01011	01010	01001	01000	00001	000001	000000
码长	5	5	5	5	5	6	6

其操作码平均码长为：**H**＝

H＝（0.15＋0.15＋0.14＋0.13＋0.12＋0.11）×3＋ 0.04×4

（0.04＋0.03＋0.03＋0.02＋0.02）×5＋

（0.01＋0.01）×6

＝0.84×3＋0.16×6＝3.38（位）

指令编码

两种码长的扩展操作码：（方法一）

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.15	0.15	0.14	0.13	0.12	0.11	0.04
扩展编码	000	001	010	011	100	101	110
码长	3	3	3	3	3	3	3
指令	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}
概率	0.04	0.03	0.03	0.02	0.02	0.01	0.01
扩展编码	111000	111001	111010	111011	111100	111101	111110
码长	6	6	6	6	6	6	6

其操作码平均码长为：$H=\sum_{i=1}^{n} p_i l_i$

$$H=(0.15+0.15+0.14+0.13+0.12+0.11+0.04)\times 3+(0.04+0.03+0.03+0.02+0.02+0.01+0.01)\times 6$$

$$=0.84\times 3+0.16\times 6=3.48\text{（位）}$$

指令编码

大概率

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.15	0.15	0.14	0.13	0.12	0.11	0.04
扩展编码							
码长							
指令	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}
概率	0.04	0.03	0.03	0.02	0.02	0.01	0.01
扩展编码							
码长							

小概率

指令编码

大概率

两种码长的扩展操作码：（方法二）

指令	I_1	I_2	I_3	I_4	I_5	I_6	I_7
概率	0.15	0.15	0.14	0.13	0.12	0.11	0.04
扩展编码	000	001	010	011	100	101	11000
码长	3	3	3	3	3	3	5
指令	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}
概率	0.04	0.03	0.03	0.02	0.02	0.01	0.01
扩展编码	11001	11010	11011	11100	11101	11110	11111
码长	5	5	5	5	5	5	5

小概率

其操作码平均码长为：$H=\sum_{i=1}^{n} p_i l_i$

$$H=(0.15+0.15+0.14+0.13+0.12+0.11)\times 3+(0.04+0.04+0.03+0.03+0.02+0.02+0.01+0.01)\times 5$$
$$=0.8\times 3+0.2\times 5=3.4\text{（位）}$$

指令格式

8086／8088指令编码由1～6个字节组成，它包括操作码（第一字节）、寻址方式（第二字节）和操作数（第三到第六字节）三部分组成p