

浅谈计算机系统结构与OI中的实用编程技巧

清华大学 计算机科学与技术系 王逸松
wangyisong1996@163.com

概述：卡常数

- 你是否遇到过：
- “这题卡常数诶.....”
“？？？”
- “我怎么都要跑1.5s+”
- “我只要0.3s啊？”

Result

Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed
Time_Limit_Exceed

#40. 【清华集训2014】卡常数

好评 差评 [+48]

描述

提交

> 自定义测试

统计

在某个诡异的地方，有一座智慧之城，那里的人民平均智商为 192，智商低于 150 的人都被称为弱智。智慧之城的市长名叫卡常（Karp-de-Chant），他 12 岁时在智慧之城中心大学 Cross Institute 获得博士学位，两年后发明了一种数列——卡常数（Karp-de-Chant Number），该数列可用来解决或优化数论、图论等领域的多种经典难题。后来，卡常数被 Trajan（智慧之城的副市长）用 spaly 树进行扩展后，威力大大增加，可以在线性时间内解决各种网络流问题和其它一些难题。卡常和 Trajan 因此分别被选为正、副市长，他们和智慧之城内的另一些智者一起，领导人民共同建设人类智慧，发挥创造和改进的能力。

然而某一天，智慧之城突然受到了反人类智慧者的袭击，反人类智慧者在城内设置了 N 个摄像头（由于他们的智商很低，只会用摄像头这种垃圾玩意），企图监视城内的人们。卡常、Trajan 决定找到这些摄像头并摧毁它们。

智慧之城里有一个用扩展卡常数原理设计的发射器，将其放在合适位置并设置半径以后，所有位于球心为这个发射器的位置、半径为指定值的球面上的目标都能被发现。在卡常、Trajan 的带领下，智慧之城的人们用这个发射器进行了若干次实验，并发现了一些摄像头的位置。比较囧的是，每次发射都能且仅能发现一个摄像头，但是，反馈回来的结果貌似有些不对劲……

后来人们终于找到了这 N 个摄像头的位置，并发现在他们用发射器进行实验的过程中，某些摄像头被移位——这就是导致反馈结果不对劲的原因。但是，在对实验结果进行分析的时候，人们却肿么也回忆不起每次实验发现的摄像头是哪个了（可能是遭遇了灵异事件导致脑抽），只知道每次实验时发射器的位置和半径。你的任务就是，根据实验数据（为了防止被反人类智慧者窃取，已经进行了加密）找出每次实验时被发射器找到的摄像头的编号。

概述：实用编程技巧 / 底层优化

- “我们只要这样，
这样， 这样.....
再这样就能AC了”

“好玄学啊.....”



图片来源：清华大学“大学物理2(英)”课程的课件

概述：计算机系统

- 计算机里有啥？
- 高级语言代码怎么被计算机看懂？
- CPU如何实现？
- 存储器有何特性？
-

概述：课程意义

- 装逼
- 学习计算机系统结构
- 加深对程序运行的理解
- 掌握更多OI中的编程技巧
-

概述：课程内容



PART I: 信息的表示和处理

信息的表示和处理

信息的表示

整数的表示

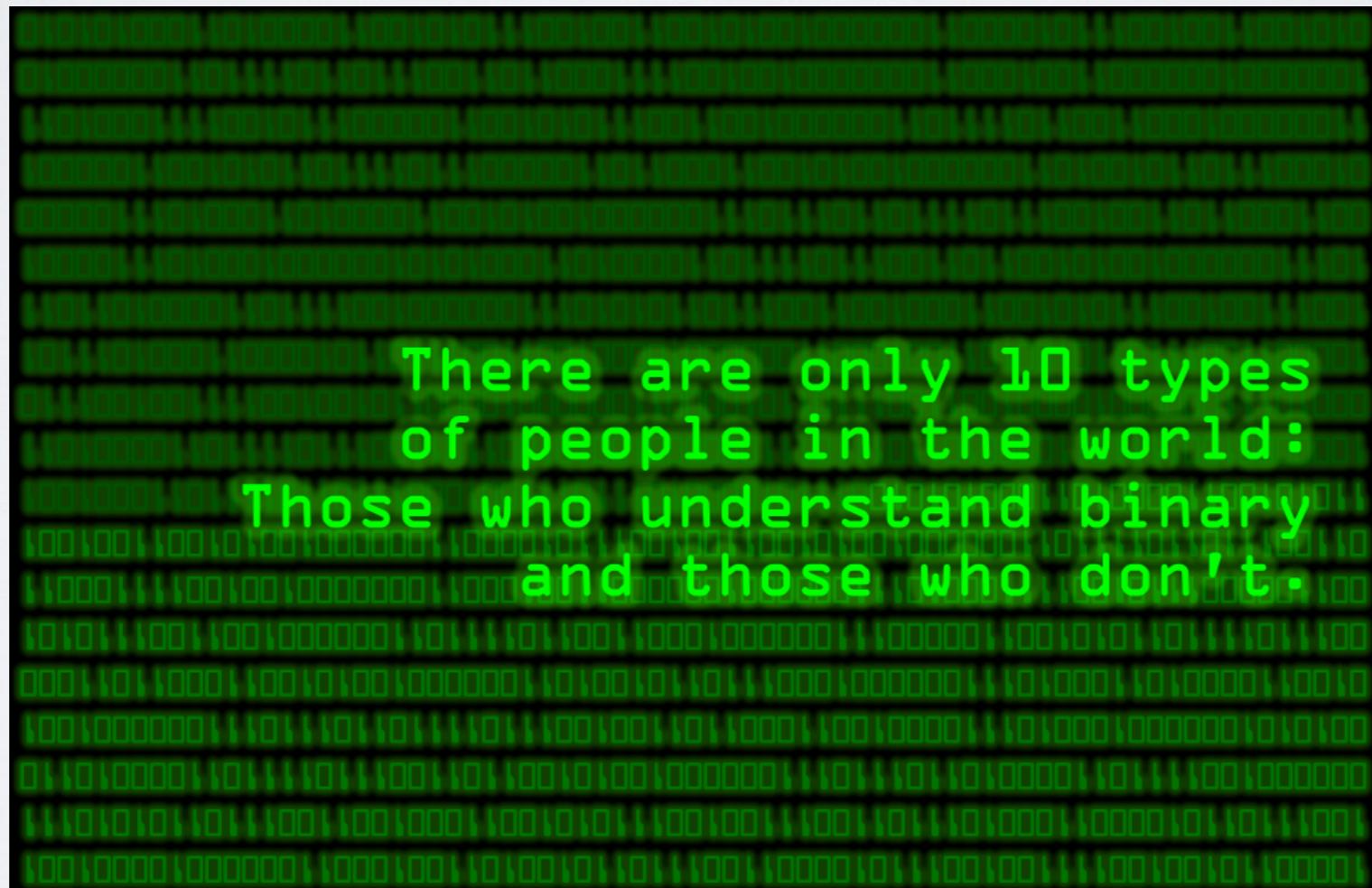
整数的运算

浮点数的表示

编程技巧

信息的表示

- 在现代计算机中，信息都用二进制表示



<https://theshpitz.files.wordpress.com/2010/03/binary.png>

WHY 二进制？

- 二值逻辑： 0 / 1 , True / False
- 数字电路： 低电平 / 高电平
- 存储设备： (磁性材料) 被磁化 / 未被磁化
- 提交代码： Accepted / Not Accepted

整数的表示

- 整数是用一个长度为 w 的位向量表示的
- 例子： $w = 16$, 表示整数 12345

$$12345 = (0101 \ 0000 \ 0101 \ 1001)_2$$

0	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

整数的表示

- 为了书写方便，有时会表示成**16进制**
- 将二进制表示4位一组，转成单个 0~9, a~f 即可



- $12345 = 0x3039$

整数的表示：字长

- 整数的表示中， w 即为字长，反映了CPU同时能处理的数据的位数。由CPU和操作系统决定。
 - 如对于 x86 (i386) 架构的CPU， $w = 32$ 。
如对于 x86_64 (amd64) 架构的CPU， $w = 64$ 。
 - 对于绝大多数操作系统， $w = 64$ 。
对于最新版本的 NOI Linux， $w = 32$ 。
- 注：一般来说，64位CPU可以运行32位操作系统，反过来不行。

整数的表示：字节顺序

- 在存储器（如内存）中，
数据是以**字节**而不是位为单位进行存储的。
 - | **字节**(Byte) = **8 位**(Bit)
- 在内存中一个跨越多个字节的对象该如何存储呢？
- **小端法** (little endian): 从**最低有效字节**到**最高有效字节**存储
大端法 (big endian): 从**最高有效字节**到**最低有效字节**存储

整数的表示：字节顺序

- 举例：在内存的 0x100 地址存储整数 0x1234567

内存地址	0x100	0x101	0x102	0x103
大端法	0x01	0x23	0x45	0x67
小端法	0x67	0x45	0x23	0x01

- 绝大多数 Intel 兼容机（包括几乎所有个人计算机）都只用**小端模式**，即使用**小端法**。

整数的表示：有符号数

- 以上表示方法只能表示 $0 \sim 2^w - 1$ 之间的整数
我们称这种方法表示的数为**无符号数**
- 为了表示负数，我们引入**有符号数**的表示方法：
对于有符号数 x ，令 $y = x$ ($0 \leq x < 2^{w-1}$)
$$x + 2^w (-2^{w-1} \leq x < 0)$$

然后将 y 用位向量表示即可。
此时我们称 y 的最高位为**符号位**。

整数的表示：有符号数

- 举例：

$w = 16, x = 12345, y = 12345$

0	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$w = 16, x = -12345, y = -12345 + 65536 = 53191$

1	0	1	0	1	1	1	1	1	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

发现没有？ $(-\bar{x})$ 的补码表示 = (\bar{x}) 的补码表示 取反 再 +1

- 这种表示方法也叫**补码** (two's complement)

整数的表示：总结

- 位向量、字长、字节顺序、有符号数

我们来看一些编程语言中的整数类型

整数类型	位数	字节数	取值范围
char / char	8	1	-128 ~ 127
unsigned char / byte	8	1	0 ~ 255
short / integer	16	2	-32768 ~ 32767
unsigned short / word	16	2	0 ~ 65535
int / longint	32	4	-2147483648 ~ 2147483647
unsigned int / dword	32	4	0 ~ 4294967295
long long / int64	64	8	$-2^{63} \sim 2^{63} - 1$
unsigned long long / qword	64	8	$0 \sim 2^{64} - 1$

整数的运算

- 在高级语言中，整数的运算分为
四则运算、位运算、逻辑运算和移位运算。
- 四则运算：加、减、乘、除
位运算：或、与、取反、异或
逻辑运算：或、与、非
移位运算：左移、右移

整数的运算：四则运算

- 加、减、乘、除
- 都在模 2^w 意义下定义
- 使用补码表示，在模 2^w 意义下完成
- 可以证明这是符合定义的

整数的运算：位运算

- 一些整数的位级运算
- 包括 或(OR)、与(AND)、取反(NOT)、异或(XOR)

位运算	C / C++	Pascal
OR		or
AND	&	and
NOT	~	not
XOR	!	xor

整数的运算：位运算

- 一些例子

C语言表达式	二进制表达式	二进制结果	十六进制结果
$\sim 0x41$	$\sim [0100\ 0001]$	[1011 1110]	0xbe
$\sim 0x00$	$\sim [0000\ 0000]$	[1111 1111]	0xff
$0x69 \& 0x55$	[0110 1001] & [0101 0101]	[0100 0001]	0x41
$0x69 0x55$	[0110 1001] [0101 0101]	[0111 1101]	0x7d

整数的运算：逻辑运算

- 用来操作布尔变量的一类运算

在 C / C++ 中，所有非0参数都表示True，0表示False

在 Pascal 中，有专门的布尔数据类型 boolean

包括 或(OR)、与(AND)、非(NOT)

逻辑运算	C / C++	Pascal
OR		or
AND	&&	and
NOT	!	not

整数的运算：逻辑运算

- 一些例子

C语言表达式	二进制表达式	二进制结果	十六进制结果
<code>!0x41</code>	<code>![0100 0001]</code>	<code>[0000 0000]</code>	<code>0x00</code>
<code>!0x00</code>	<code>![0000 0000]</code>	<code>[0000 0001]</code>	<code>0x01</code>
<code>0x69 && 0x55</code>	<code>[0110 1001] && [0101 0101]</code>	<code>[0000 0001]</code>	<code>0x01</code>
<code>0x69 0x55</code>	<code>[0110 1001] [0101 0101]</code>	<code>[0000 0001]</code>	<code>0x01</code>

- 注意，二元逻辑运算是**短路**的，当左边的值就能确定表达式的值时不会计算右边的值

整数的运算：移位运算

- 将整数视为位向量进行移位的操作
- 包括 左移 和 右移

移位运算	C / C++	Pascal
左移	<<	shl 或 <<
右移	>>	shr 或 >>

整数的运算：移位运算

- $x \ll k$ ($0 \leq k < w$)

将 x 的位表示的前 k 位删去，并在最后面添加 k 个 0

- $x \gg k$ ($0 \leq k < w$)

将 x 的位表示的后 k 位删去，

无符号数 (逻辑右移)：在最前面添加 k 个 0，

有符号数 (算术右移)：在最前面添加 k 个符号位

- 注意： $k \geq w$ 时移位运算的行为是未定义的(undefined)

整数的运算：总结 & 思考

- 四则运算、位运算、逻辑运算、移位运算
- 位运算有何实际用处？
- 移位运算能否用四则运算来表示？
如果能的话，移位运算有何实际意义？

浮点数的表示

- 由 IEEE 754 标准于1985年规定

- $V = (-1)^s \times M \times 2^E$

s 为符号， M 为尾数， E 为阶码

IEEE 754 浮点数	位数	E的长度	M的长度
单精度浮点数	32	8	23
双精度浮点数	64	11	52

- 限于时间不讲。感兴趣的同学可以参考
“浮点误差与误差复杂度”, 清华大学 陈许旻, WC 2013 讲课

信息的表示和处理

信息的表示 

整数的表示 

整数的运算 

浮点数的表示 

编程技巧

位运算在OI中的应用

magic
bag of
tricks



一些简单的位运算技巧

- 读某个二进制位
- 将某位置为 1 / 0
- 求二进制表示中 1 的个数
- 求二进制表示中 前缀 / 后缀 0 的个数
- 提取 lowbit

读某个二进制位

- u32为32位无符号整数类型

以C++为例

```
inline u32 read_bit(u32 x, int pos) {  
    return (x >> pos) & 1;  
}
```

将某位置为 1 / 0

```
inline u32 set_bit(u32 x, int pos) {  
    return x | (1u << pos);  
}
```

```
inline u32 clear_bit(u32 x, int pos) {  
    return x & ~(1u << pos);  
}
```

求二进制表示中 1 的个数

- 查表法

```
int cnt_table[1 << 16];

void count_pre() {
    cnt_table[0] = 0;
    for (int i = 0; i < 1 << 16; i++) {
        cnt_table[i] = cnt_table[i >> 1] + (i & 1);
    }
}

inline int count(u32 x) {
    return cnt_table[x >> 16] + cnt_table[x & 65535u];
}
```

求二进制表示中后缀 0 的个数

- 二分法

```
inline int count_trailing_zeros(u32 x) {  
    int ret = 0;  
    if (!(x & 65535u)) x >>= 16, ret |= 16;  
    if (!(x & 255u)) x >>= 8, ret |= 8;  
    if (!(x & 15u)) x >>= 4, ret |= 4;  
    if (!(x & 3u)) x >>= 2, ret |= 2;  
    if (!(x & 1u)) x >>= 1, ret |= 1;  
    return ret + !x;  
}
```

求二进制表示中前缀 0 的个数

- 同上，也可用查表法

```
int cls_table[1 << 16];

void cls_pre() {
    cls_table[0] = 16;
    for (int i = 1; i < 1 << 16; i++) {
        cls_table[i] = cls_table[i >> 1] - 1;
    }
}

inline int count_leading_zeros(u32 x) {
    return x >> 16 ? cls_table[x >> 16] :
        16 + cls_table[x & 65535u];
}
```

提取 LOWBIT

- 即一个整数二进制中最右边一个 1 到最低位的部分
以下代码的正确性可由补码的性质证明

```
inline u32 lowbit(u32 x) {  
    return x & -x;  
}
```

将整数用作集合

- 一个 w 位的整数，可以看作一个大小为 w 的集合，即用每一个位来表示一个元素是否存在
- 集合的并、交、补、求大小等运算即可做到 $O(n / w)$ 集合的空间复杂度也是 $O(n / w)$ ，或 n 个 bit
- 这叫“bitset”
在C++中，有同名的库支持这些操作

N维数点

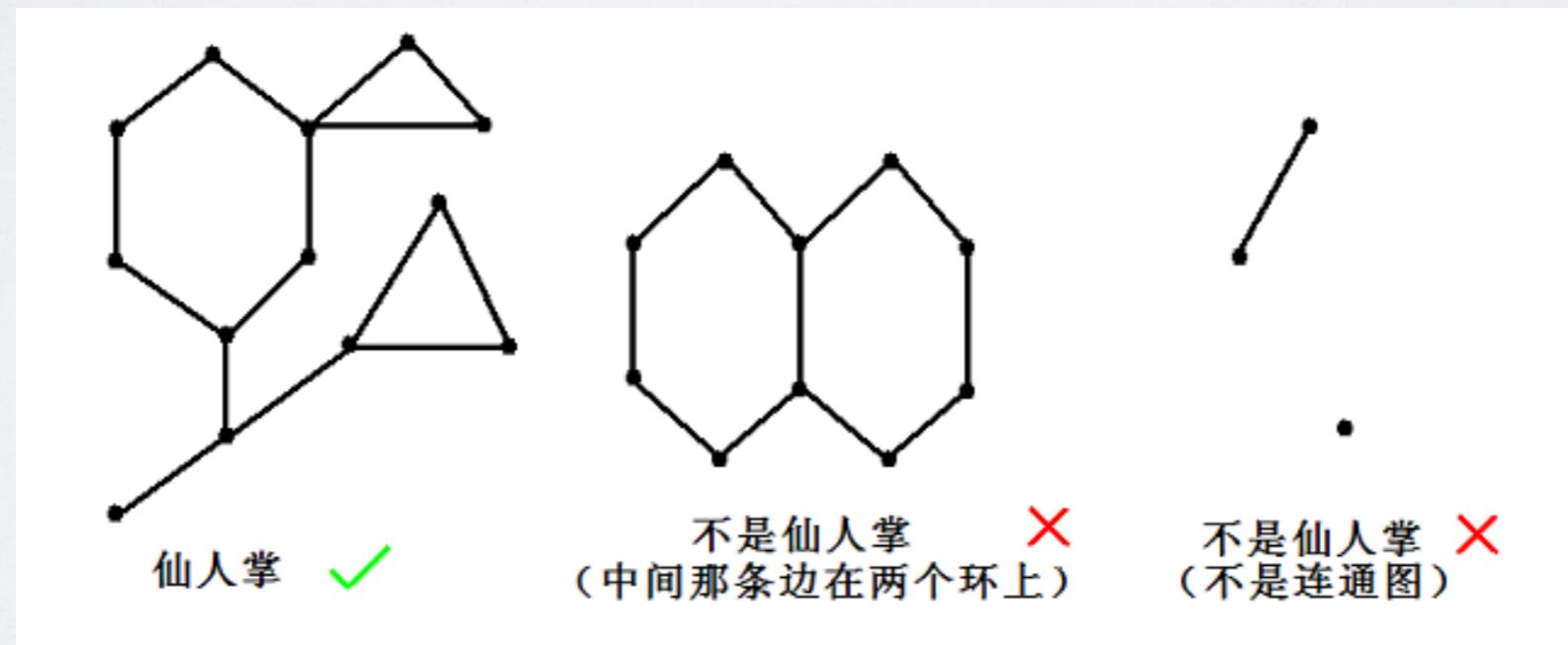
- 给定 n 维空间中的 m 个点，以及 q 个询问，
每个询问给出一个 n 维空间点，
问有多少个点**每一维坐标都不大于该询问点**
- 每个点的每一维坐标都是 $[l, m]$ 之内的整数
// 如果不是则可离散化
- $n \sim 5, m \sim 30000, q \sim 30000$

N维数点

- 对于每一维坐标，保存 m 个集合：
这一维坐标不超过 i 的所有点的编号 ($l \leq i \leq m$)
- 询问时只要将询问点每一维坐标对应的集合求交，
并输出交集的大小即可
- 时间复杂度 $O((m + q) nm / w)$ // 比暴力快 w 倍!
空间复杂度 $O(nm^2 / w)$

静态仙人掌

- 维护一个 n 个点的有根仙人掌，支持三种操作：
 - 将点 x 到根最短路径上所有点的黑白颜色取反
 - 将点 x 到根最长简单路径上所有点的颜色取反
 - 询问点 x 的子仙人掌中黑点的数目



- $n, q \leq 50000$, 内存限制 128 MB

静态仙人掌

- bitset!
- 预处理每个点到根的最短/最长简单路径上的点的编号集合
修改只需 xor, 询问只需要求集合大小
- 空间不够 → 分块+可持久化!

复杂度 $O(n^2/w)$? 跑得过就行!
比圆方树好写多了!

状态压缩动态规划

- 在一些动态规划问题中，状态可用一个较小的集合来表示
- 可以用整数来给集合进行“编码”，
从而用位运算简化动态规划的代码实现
- 在某些问题中，我们需要“枚举子集”，可用以下小技巧：

```
for (int i = S; i; i = (i - 1) & S) {  
    do_something(i);  
}
```

位运算在OI中的应用：总结

- 简单的位运算技巧
- 将整数用作集合 // bitset
 - N维数点
 - 静态仙人掌
- 状态压缩动态规划

信息的表示和处理

信息的表示 

整数的表示 

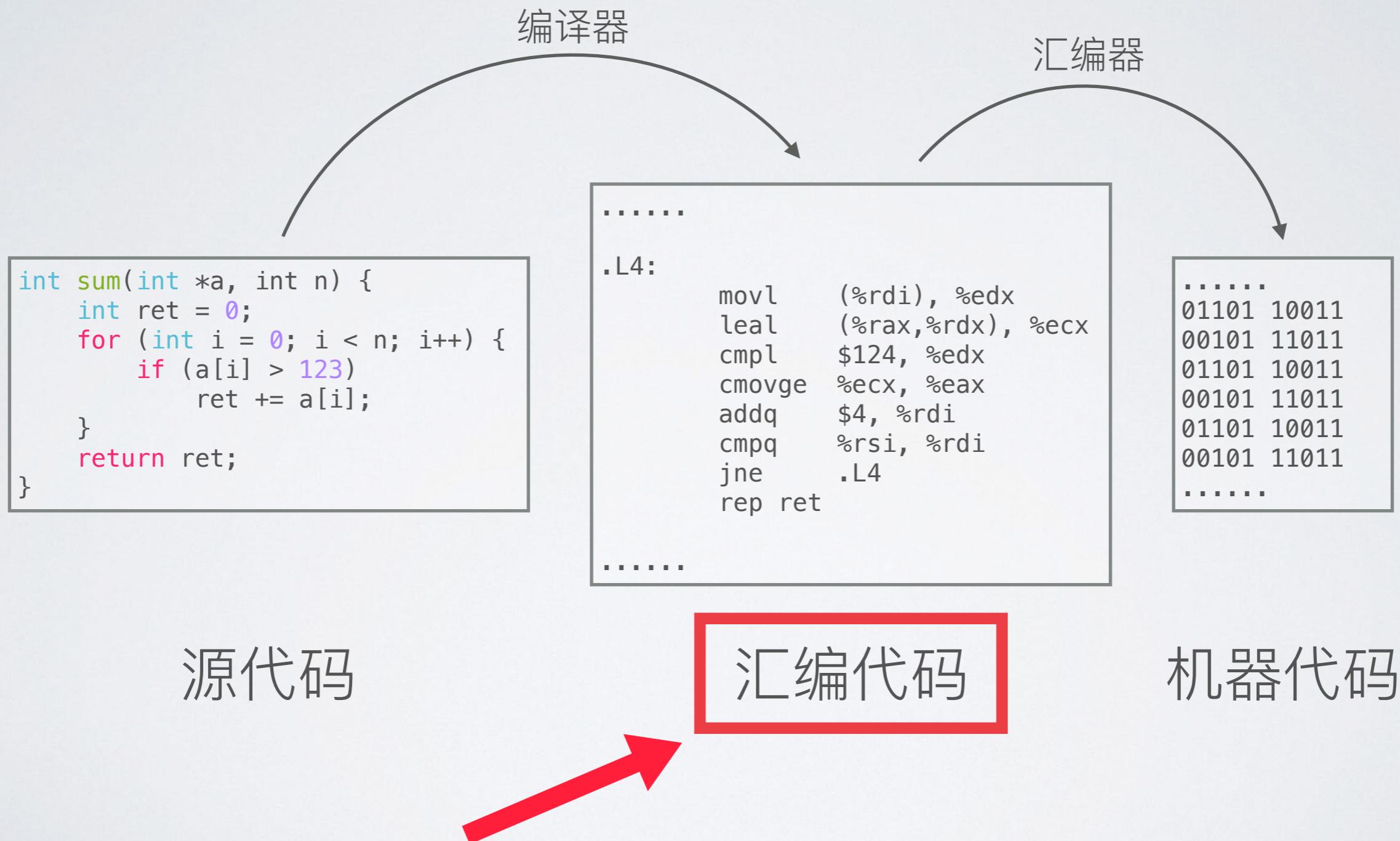
整数的运算 

浮点数的表示 

编程技巧：位运算 

PART2: 程序的底层表示

程序的底层表示：引言



程序的底层表示

x86-64 CPU 架构简介

机器指令的分类

机器指令的性质

编程技巧

X86-64 CPU 架构简介

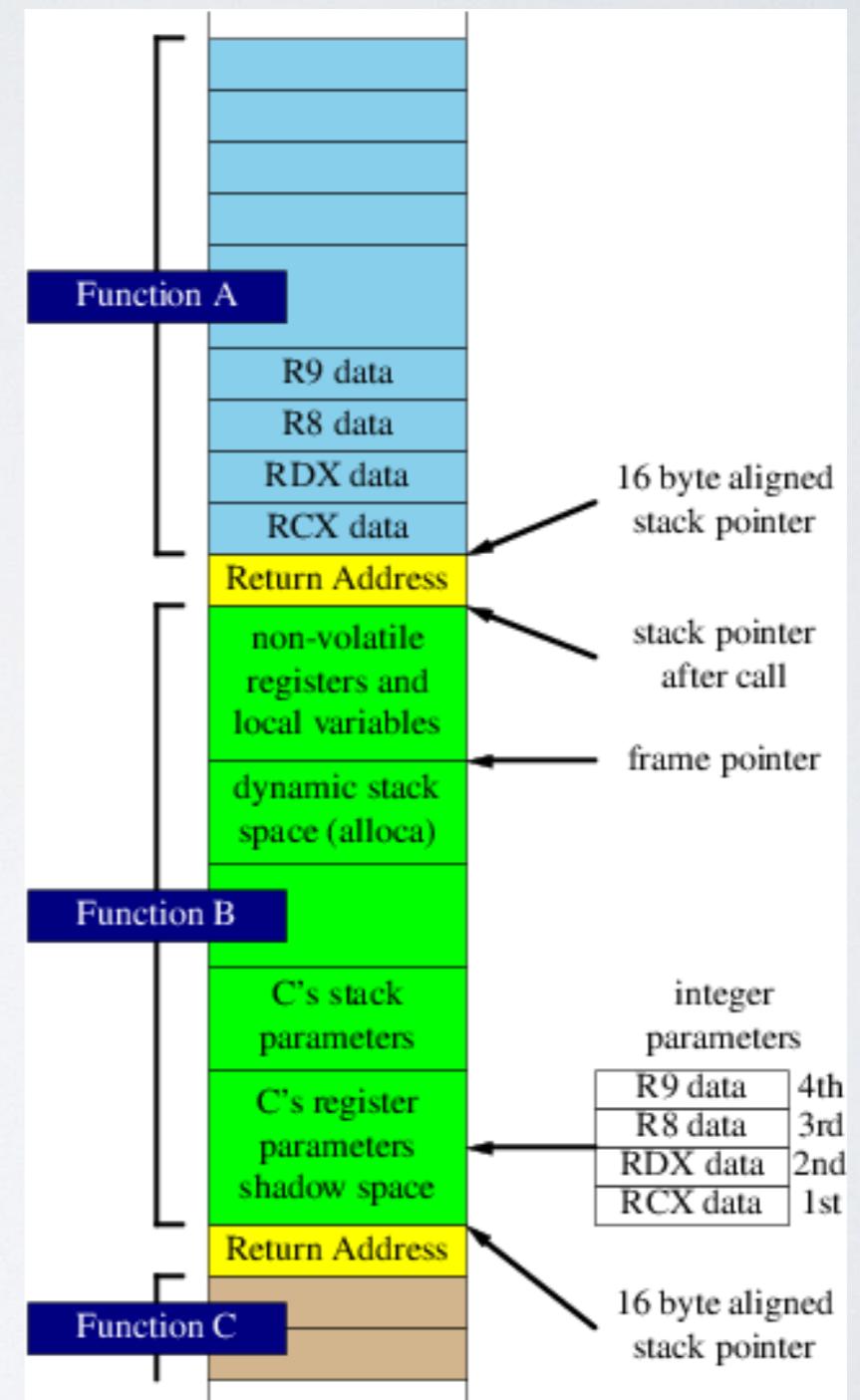
- **程序计数器 (PC)** : 给出下一条将要运行的指令的地址
- **整数寄存器文件**: 16个64位整数值，用来存储各种数据以及CPU的状态 (如%rip存储PC)
- **条件码寄存器**: 保存最近执行的算术或逻辑指令的状态信息，用于实现条件跳转等功能
- **向量寄存器**: 可以存放一个或多个整数或浮点数的值

整数寄存器文件

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

整数寄存器文件

- %rsp 寄存器表示栈顶的地址
- 这个栈存在于内存中，用来实现过程的调用和返回
- 注意：这个栈朝下增长



条件码寄存器

一个特殊的寄存器，叫做%rflags，包含

- CF：进位标志。最近的操作使得最高位产生了进位。
- ZF：零标志。最近的操作得出的结果为 0。
- SF：符号标志。最近的操作得到的结果为负数。
- OF：溢出标志。最近的操作导致一个补码溢出。

CPU的运行过程

- 从PC所指的内存地址获取一条机器指令
 有哪些呢?
- 根据取得的指令进行一个基本的操作
 - 数据传送指令、算术和逻辑指令、控制指令
- 更新PC的值，执行下一条指令

机器指令有哪些呢？

- 这是一本 Intel 的 CPU 手册
- 封面很薄 (.....?)
- 但是有 **2198** 页！
- 可见 x86-64 之复杂！



Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2 (2A, 2B, 2C & 2D):
Instruction Set Reference, A-Z

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of three volumes:
Basic Architecture, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383;
System Programming Guide, Order Number 325384. Refer to all three volumes when evaluating your
design needs.

Order Number: 325383-060US
September 2016

机器指令的主要分类

数据传送指令

算术和逻辑指令

控制指令

机器指令的操作数

- 大多数指令都有一个或多个**操作数** (operand)
有以下几种操作数：
 - **立即数**：一个整数，如 \$12345 或 \$0xFF
 - **寄存器**：某个整数寄存器（或其低 8 / 16 / 32 位）
 - **内存引用**：内存某个地址的内容，有多种**寻址模式**

寻址模式

类型	格式	操作数值	名称
立即数	\$Imm	Imm	立即数寻址
寄存器	r _a	R[r _a]	寄存器寻址
存储器	Imm	M[Imm]	绝对寻址
存储器	(r _a)	M[R[r _a]]	间接寻址
存储器	Imm(r _b)	M[Imm+R[r _b]]	(基址+偏移量) 寻址
存储器	(r _b , r _i)	M[R[r _b]+R[r _i]]	变址寻址
存储器	Imm(r _b , r _i)	M[Imm+R[r _b]+R[r _i]]	变址寻址
存储器	(, r _i , s)	M[R[r _i] · s]	比例变址寻址
存储器	Imm(, r _i , s)	M[Imm+R[r _i] · s]	比例变址寻址
存储器	(r _b , r _i , s)	M[R[r _b]+R[r _i] · s]	比例变址寻址
存储器	Imm(r _b , r _i , s)	M[Imm+R[r _b]+R[r _i] · s]	比例变址寻址

注: s = 1, 2, 4 or 8

数据传送指令

指令	效果	描述
MOV S, D	D \leftarrow S	传送
MOVZ S, R	R \leftarrow 零扩展(S)	以零扩展进行传送
MOVS S, R	R \leftarrow 符号扩展(S)	传送符号扩展的字节
movabsq I, R	R \leftarrow I	传送绝对的四字 (8字节)
cltq	%rax \leftarrow 符号扩展(%eax)	将%eax符号扩展到%rax

表格来源于《深入理解计算机系统》第122~123页

- 注意：两个操作数不能同时为内存引用！

数据传送指令：条件传送

- 当传送条件满足时，将 S 复制到 R

指令	同义名	传送条件	描述
cmove S, R	cmovz	ZF	相等/零
cmovne S, R	cmovnz	~ZF	不相等/非零
cmovs S, R		SF	负数
cmovns S, R		~SF	非负数
cmovg S, R	cmovnle	~(SF^OF)&~ZF	有符号 >
.....

表格来源于《深入理解计算机系统》第147页

- 除此之外还有：有符号 $\geq, <, \leq$, 无符号 $>, \geq, <, \leq$

算术和逻辑指令

指令	效果	描述
leaq S, D	D \leftarrow &S	加载有效地址
INC D	D \leftarrow D + 1	加1
DEC D	D \leftarrow D - 1	减1
ADD S, D	D \leftarrow D + S	加
.....

表格来源于《深入理解计算机系统》第129页

- 除此之外还有：取负、取反、减、乘、各种位运算、各种移位运算
- 注意：leaq常用于简单的算术计算

算术和逻辑指令

- 特殊的算术操作：128位乘法和除法
- `mulq`和`imulq`可将`%rax`和另一个64位整数的128位乘积存入`%rdx:%rax`中
- `divq`和`idivq`可将`%rdx:%rax`表示的128位整数除以一个64位整数，并将商和余数分别存在`%rax`和`%rdx`中

控制指令

- **比较指令** (CMP) 、**测试指令** (TEST) : 设置条件码
- **设置指令** (SET) : 访问条件码
- **跳转指令** (JUMP) : 根据条件码来执行跳转
- **调用指令** (CALL) 、**返回指令** (RET) : 实现过程调用

- 终于把机器指令的主要分类说完了.....
- 真的好多呀!
- 每个操作数有十来种寻址模式，条件码的组合也有十几种，每类指令又有十几条，还有各种特殊情况.....
- 难怪 Intel 的手册那么厚！！
- 作为OI选手 / 程序猿，我们不必关心这些细节，只要了解即可



程序的底层表示

x86-64 CPU 架构简介 

机器指令的分类 

机器指令的性质

编程技巧

机器指令的一些性质

- 在现代CPU中，每条机器指令的运行时间并不是一样的
- 取决于指令类型、指令涉及的数据，甚至是CPU当前的状态
- 加法指令：1个时钟周期
乘法指令：3个时钟周期
除法指令：几十到几十个时钟周期
跳转指令：视具体情况，几十到几十个时钟周期

除法的优化

magic
bag of
tricks



少用除法

```
int sum(int *a, int n, int m) {  
    int ret = 0;  
    for (int i = 0; i < n; i++)  
        ret = (ret + a[i]) % m;  
    return ret;  
}
```



```
int sum(int *a, int n, int m) {  
    long long ret = 0;  
    for (int i = 0; i < n; i++) ret += a[i];  
    return ret % m;  
}
```

除以常数的优化

- 假设我们要计算 x / y , 其中 y 为已知常数
- 如果 $y = 2^k$, 我们只要计算 $x \gg k$, 因为 $x \gg k = \lfloor x / 2^k \rfloor$
- 事实上, 编译器会自动帮你完成这个优化
- 如果 y 是其他常数?

除以常数的优化

- 我们来看一个例子：无符号整数 $x / 6$

```
movabsq $-6148914691236517205, %rdx
movq    %rdi, %rax
mulq    %rdx
movq    %rdx, %rax
shrq    $2, %rax
```

- 其中 -6148914691236517205 的16进制表示为 $0xAAAAAAAAAAAAAAAB$
 $//$ 这个数正好等于 $2^{66} / 6$ 向上取整
- 这段代码相当于计算了 $\lfloor x \cdot 0xAAAAAAAAAAAAAAAB / 2^{66} \rfloor$
- 可以证明这是正确的

除以常数的优化

- 这种优化的意义在于，我们可以将其扩展到运行期
- 即我们可以在代码中实现一个微型的“编译器”，来完成除法的优化
- 当然，这样做的成本太高了……不建议在OI比赛中使用
- 感兴趣的同学可以参考
“论程序底层优化的一些方法和技巧”, 骆可强, WC 2009 论文

$a \times b \bmod c$ 的优化

- 在OI中经常要计算形如 $a \times b \bmod c$ 的表达式，其中 a, b, c 都是32位整数，这时候我们需要使用64位整数类型的乘法，再用64位整数除法，如在32位计算机上，这会被GCC编译成一个内建函数 `_divdi3`
- 总之就是慢！

Time Limit Exceeded

$a \times b \bmod c$ 的优化

- 在汇编语言中，`imul`是不会乘爆的，会将32位乘法结果保存在两个寄存器中，而`idiv`正好能接受`imul`的结果作为输入，进行64位除以32位的操作
- 使用汇编语言直接实现 $a \times b \bmod c$ ，效率高了一倍多！

```
inline int mul_mod(int a, int b, int mo) {
    int ret;
    __asm__ __volatile__ ("\\tmull %%ebx\\n\\tdivl %%ecx\\n" : "=d"(ret):"a"(a),"b"(b),"c"(mo));
    return ret;
}
```

注意：在NOI系列比赛中禁止使用内嵌汇编！

程序的底层表示

x86-64 CPU 架构简介 

机器指令的分类 

机器指令的性质 

编程技巧：除法的优化 

QUESTIONS ?



<https://oracle-base.com/blog/wp-content/uploads/2016/01/Questions.jpg>

HAVE A BREAK ~



PART3: 处理器体系结构

- 听说这是清华计算机系最难的一门必修课

我们要做什么？



奋战三星期

做台计算机

处理器体系结构

一些基础知识

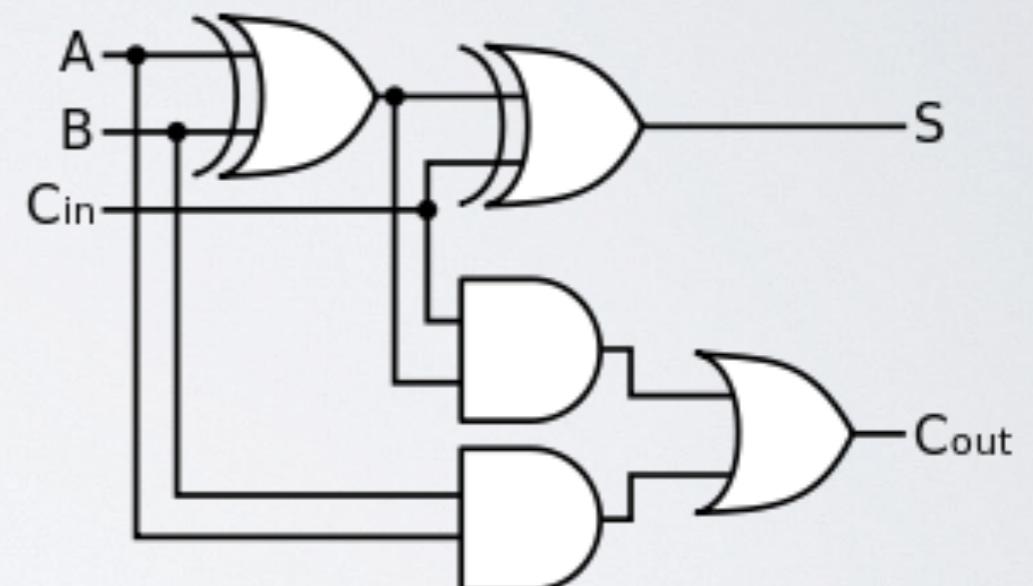
CPU的实现

编程技巧

只讲精神，不讲细节

数字逻辑电路

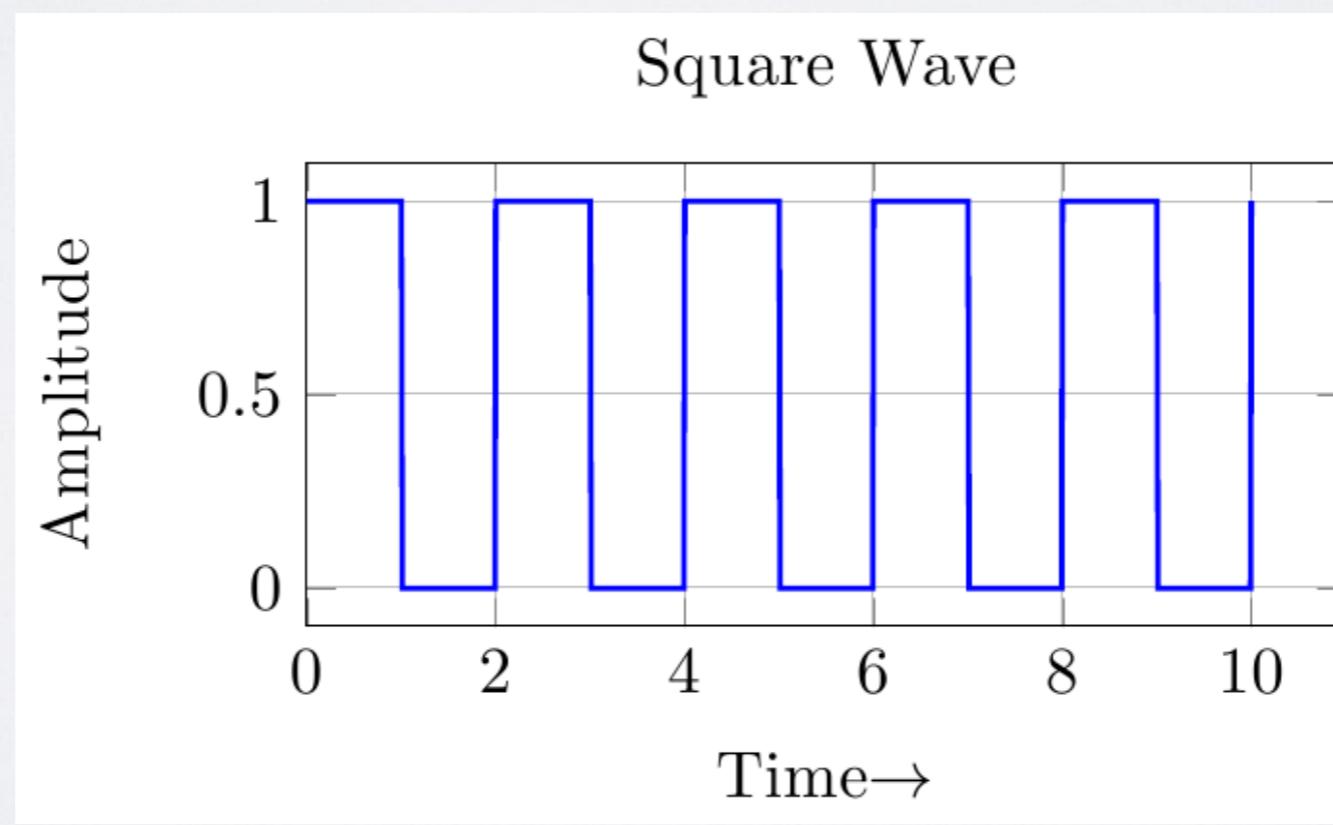
- 现代CPU主要是用**数字逻辑电路**实现的
- 数字： 0 和 1
- 逻辑：布尔运算，布尔表达式
- 电路：编码器、译码器、加法器.....



图片来源：<https://upload.wikimedia.org/wikipedia/commons/thumb/a/a9/Full-adder.svg/300px-Full-adder.svg.png>

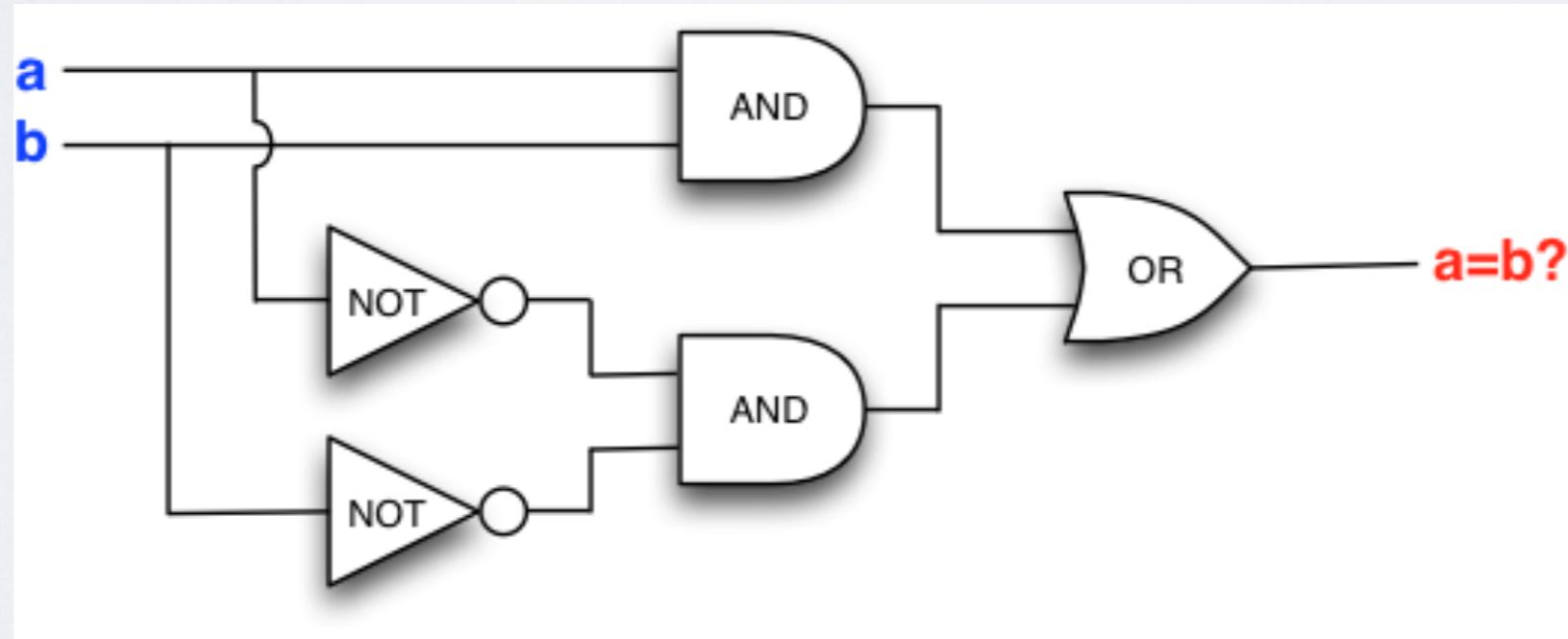
时钟

- 一个电压随时间周期变化的电路，其中电压的取值只有高电压和 0
设其频率为 f ，则一个周期的时间 $T = 1 / f$
- 我们称每个周期开始时，电压变高的瞬间为**上升沿**



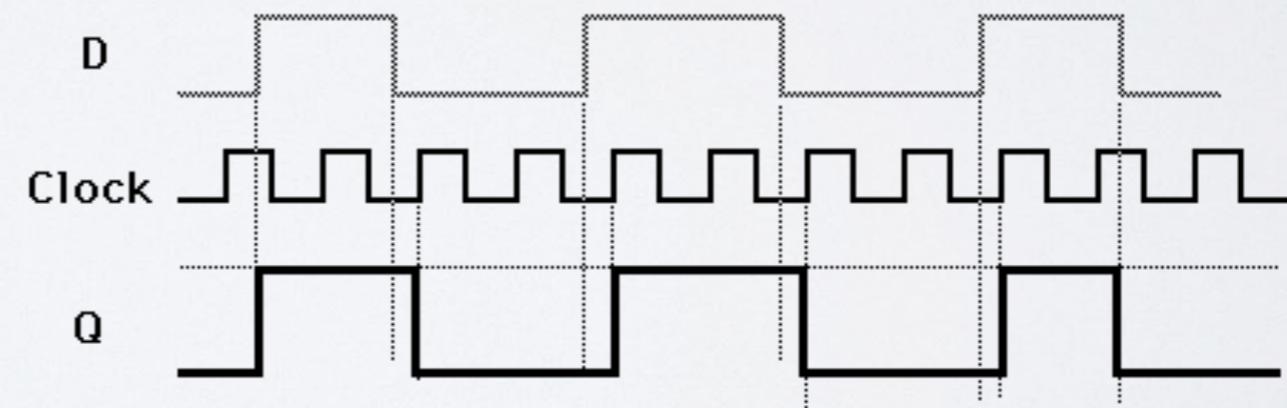
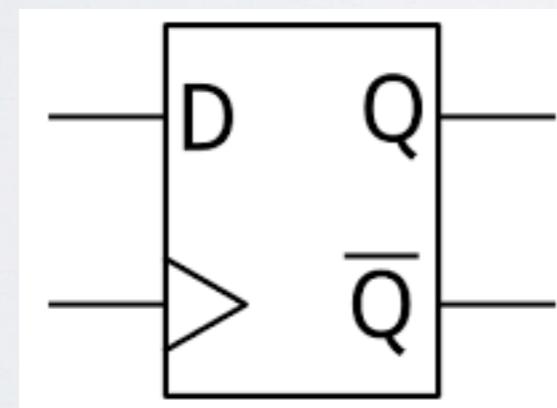
组合逻辑电路

- 由与、或、非门构成的电路，输出只跟当前输入状态有关
当输入变化的时候输出也会在很短的时间内更新
可以实现任何逻辑函数



触发器

- 一种输出跟当前输入状态和过去输入状态有关的电路
在时钟变化的时候会改变状态
- 如**D触发器**会在每个时钟的上升沿将输出设置为输入，并保持到下一个时钟上升沿



King

- 构造一个组合逻辑电路，输入 n 位二进制整数 A, B ，要求实现：
 1. $(A + 1) \bmod 2^n$
 2. $(A + B) \bmod 2^n$
 3. $(A \times B) \bmod 2^n$
- 对电路使用的逻辑门个数和电路的延迟都有一定要求
- 加法器：个数 $O(n)$ ，延迟 $O(\log n)$
- 乘法器：个数 $O(n^2)$ ，延迟 $O(\log n)$

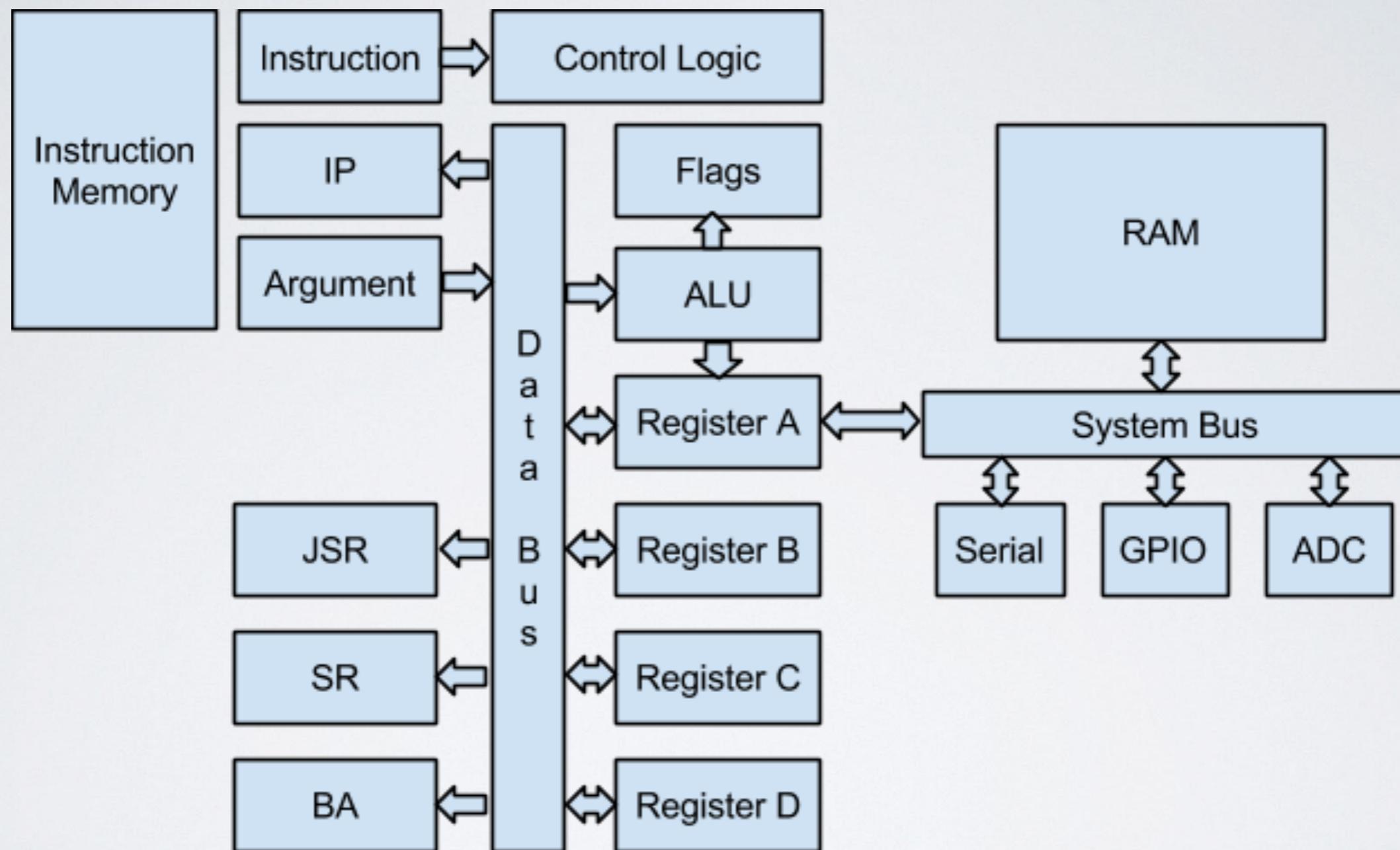
CPU的实现

顺序实现的CPU

流水线实现的CPU

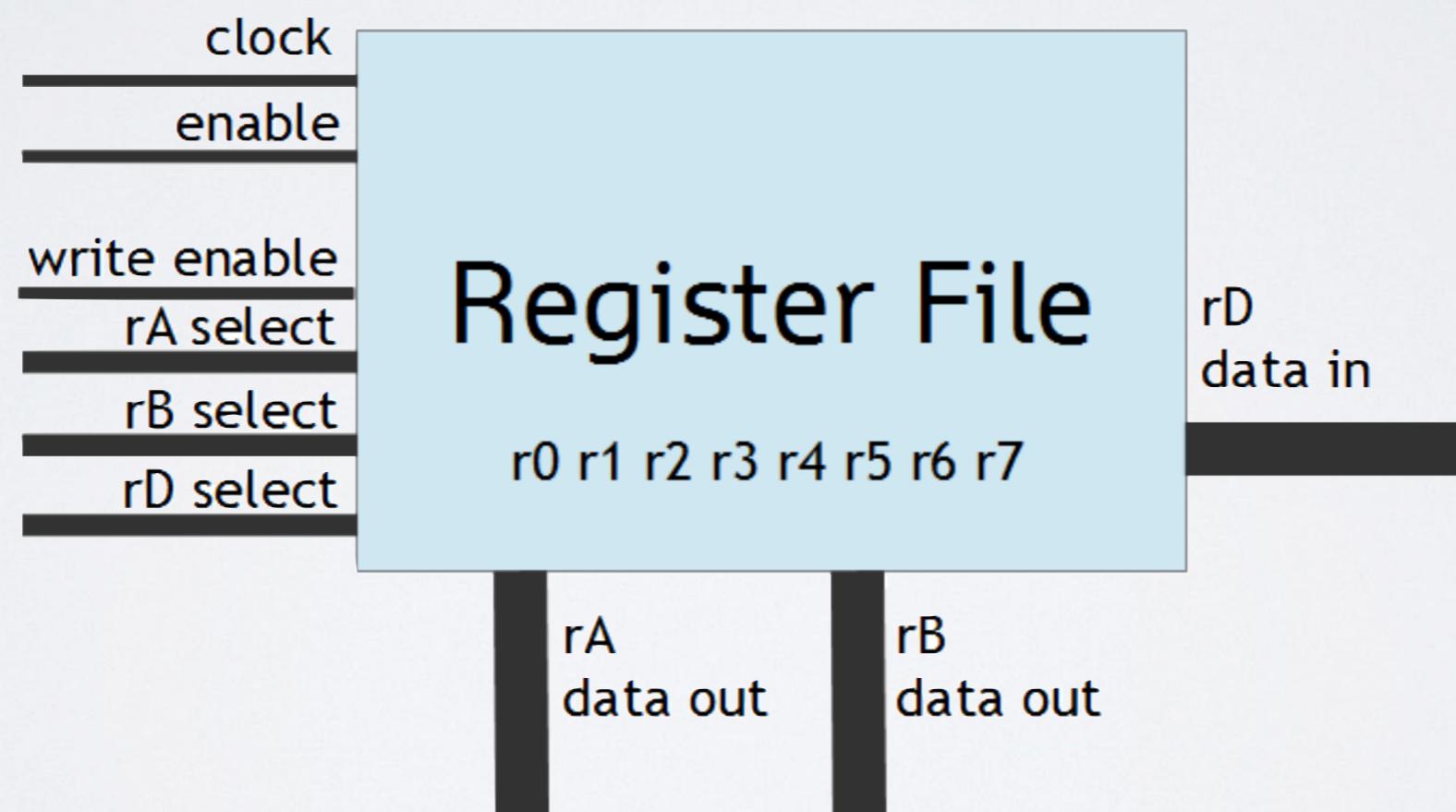
现代CPU的实现

- CPU的结构很复杂，我们在这省略很多细节



“黑盒子”

- 不考虑寄存器文件的细节，也不考虑CPU和内存交换数据的细节等



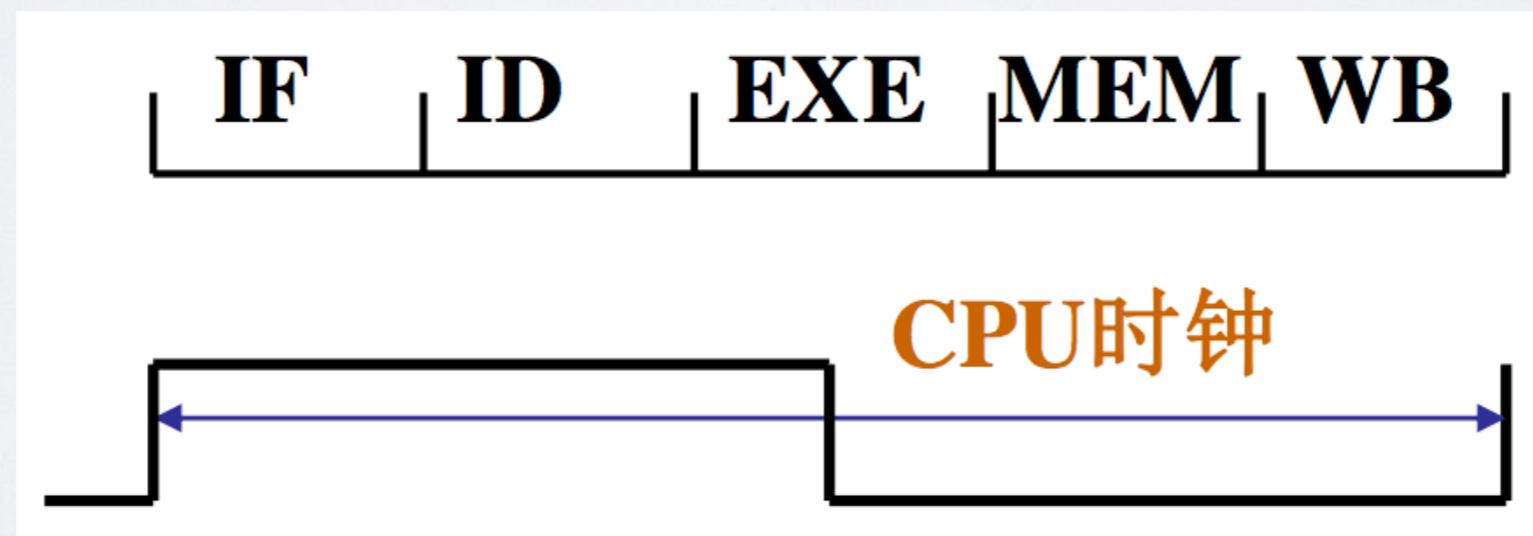
指令的执行步骤

所有指令都能分成这些阶段：

- **取指**: 从内存的PC位置读取指令，并计算下一条指令的地址
- **译码**: 根据指令的类型，从寄存器文件读取最多两个操作数
- **执行**: 由算术逻辑单元(ALU)根据指令进行相应的操作和计算
- **访存**: 将数据写入内存，或是从内存中读出数据
- **写回**: 将最多两个结果写回寄存器文件
- **更新PC**: 将PC的值更新为下一条指令的地址

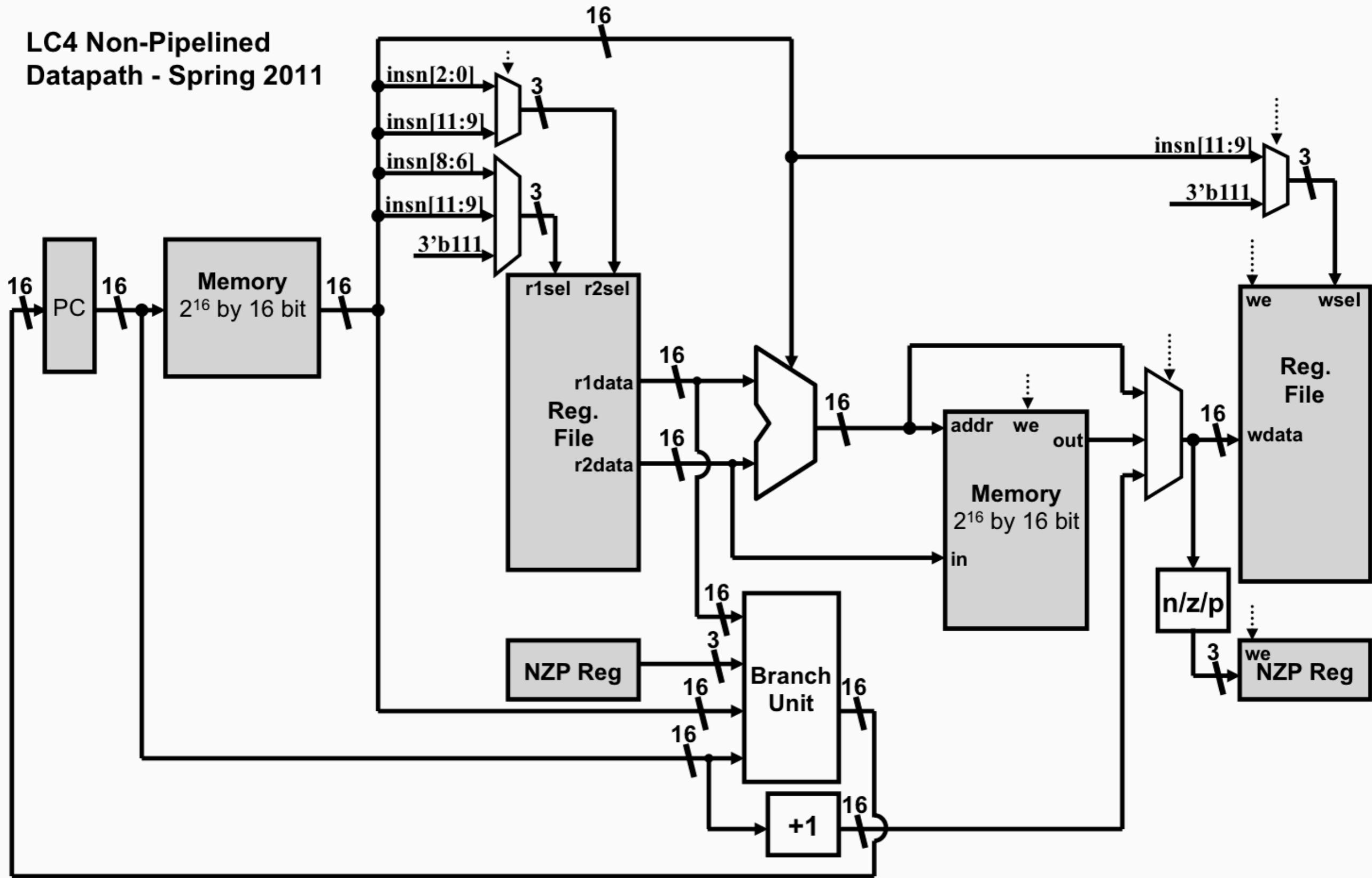
顺序实现的CPU

- 所有指令都在一个时钟周期内完成
- 性能和资源利用率都很低



图片来源：清华大学“计算机组成原理”课程的课件

LC4 Non-Pipelined Datapath - Spring 2011



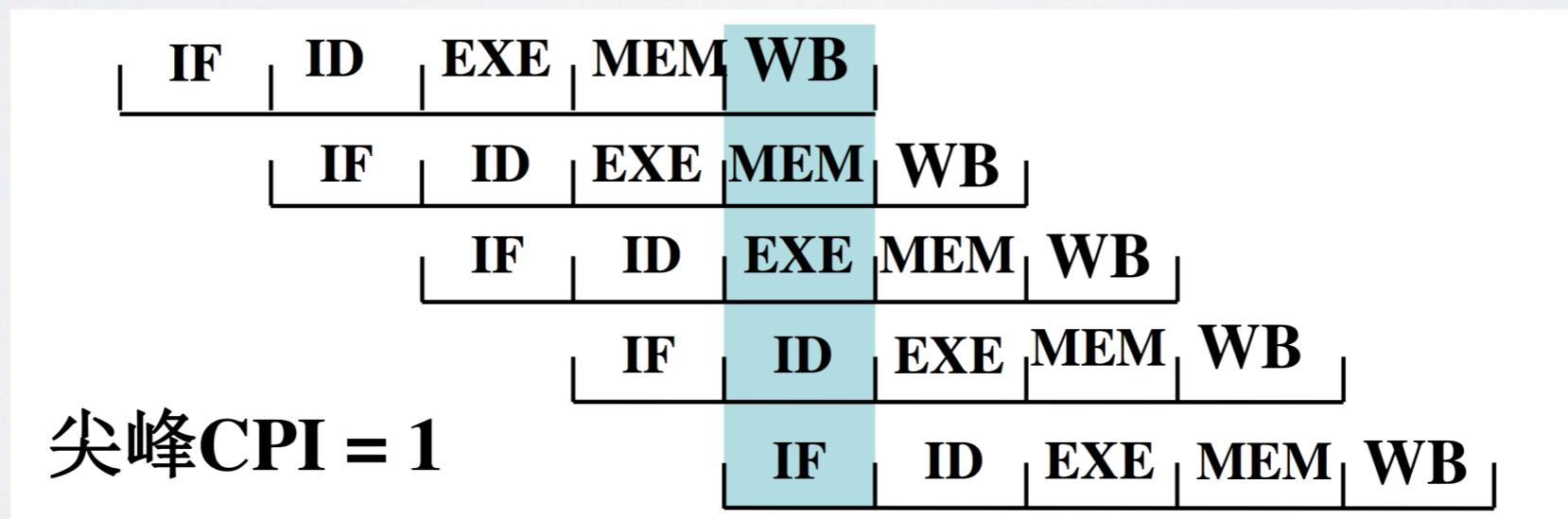
一个现实中的例子

- 你要给OI比赛出题
- 包括**想idea、写题面、写标程、造数据**这些阶段
- 因为是一个人出题，所有操作都按顺序完成
- 出一套题？只能一道一道出.....



流水线实现的CPU

- 顺序实现的CPU，同时只有一条指令的一个阶段在运行
时间效率和电路资源的利用率都不高
- 流水线实现的CPU，每条指令在每个阶段之后会被暂存到寄存器，
然后在下一个时钟周期进入下一个阶段，同时有多条指令在运行



图片来源：清华大学“计算机组成原理”课程的课件

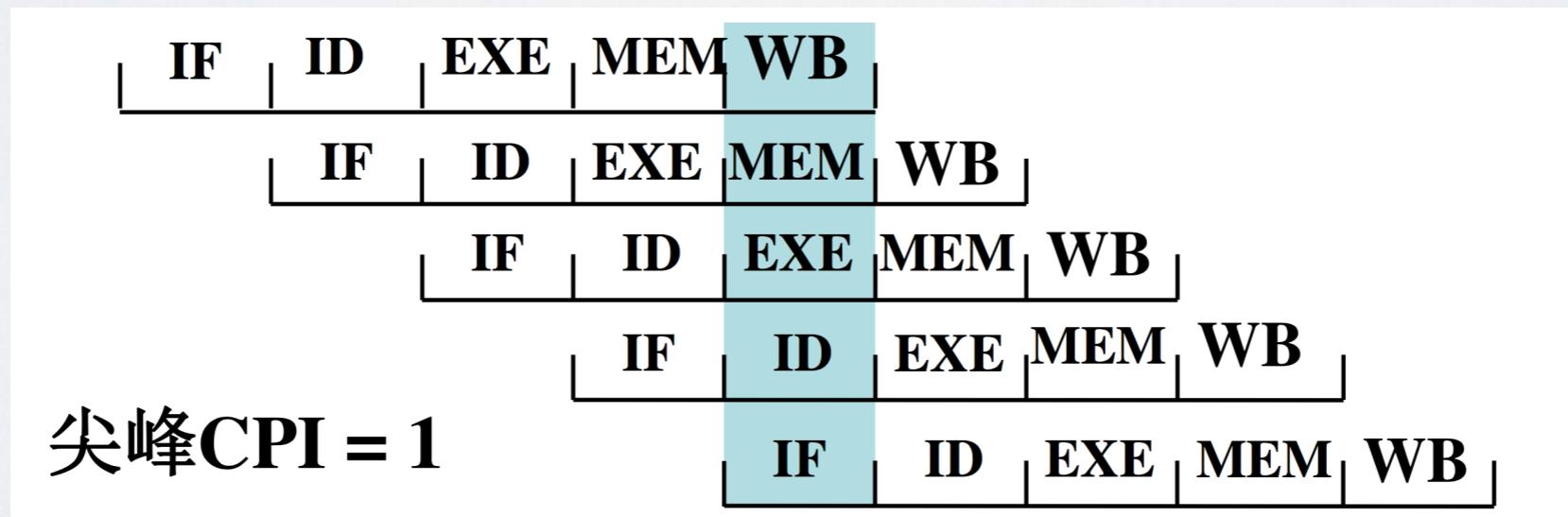
一个现实中的例子

- 你要出很多题，每题包括想idea、写题面、写标程、造数据
- 你手下有4个人，每个人专门负责出题的一部分 → 使用流水线技术



数据冒险

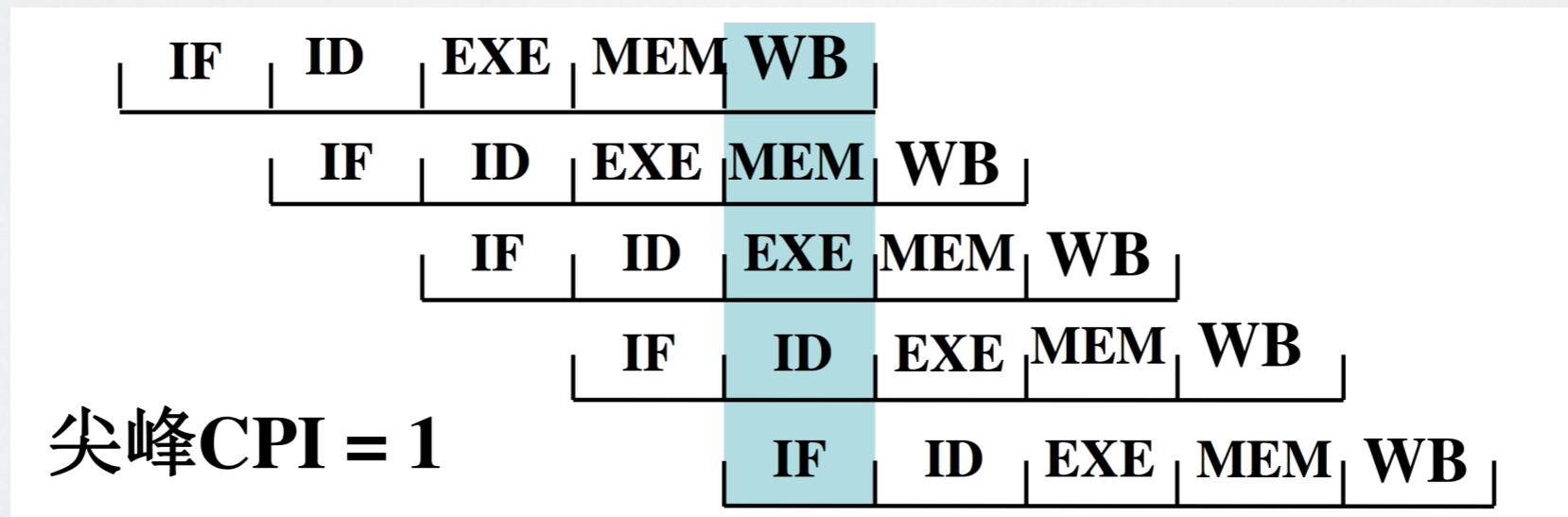
- 一条指令的结果可能会影响后面几条指令所需的数据，而在后面的指令译码的时候，这些数据还没被这条指令写回，导致得到了错误的数据
- 下图中，如果第一条指令写回的数据要被后几条指令使用.....



图片来源：清华大学“计算机组成原理”课程的课件

控制冒险

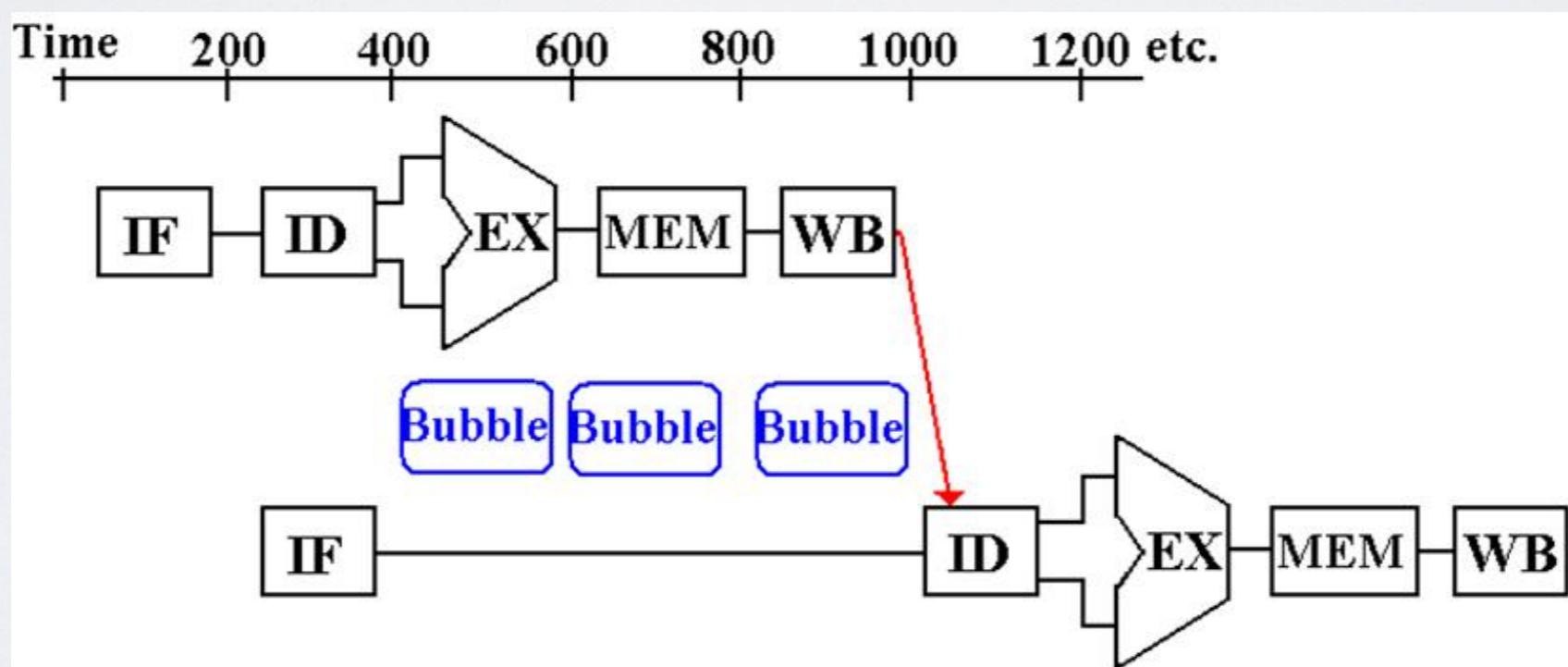
- 当CPU无法确定下一条指令的PC时，就发生了控制冒险
- 下图中，如果第一条指令是跳转指令.....



图片来源：清华大学“计算机组成原理”课程的课件

暂停(STALLING)

- 当下一条指令需要的数据还没被计算完时，向流水线添加气泡，暂停流水线的运行，直到数据被写回
- 简单但是效率不高



一个现实中的例子

- 你要出题，包括想idea、写题面、写标程、造数据
你还要验题，包括验题面、写暴力、写标程、验数据
- 出题之后立即验题 → 验题人需要等待题面写好 → 加气泡



想idea

写题面

写标程

造数据

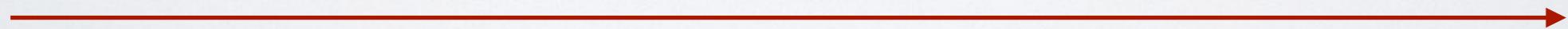
bubble

验题面

写暴力

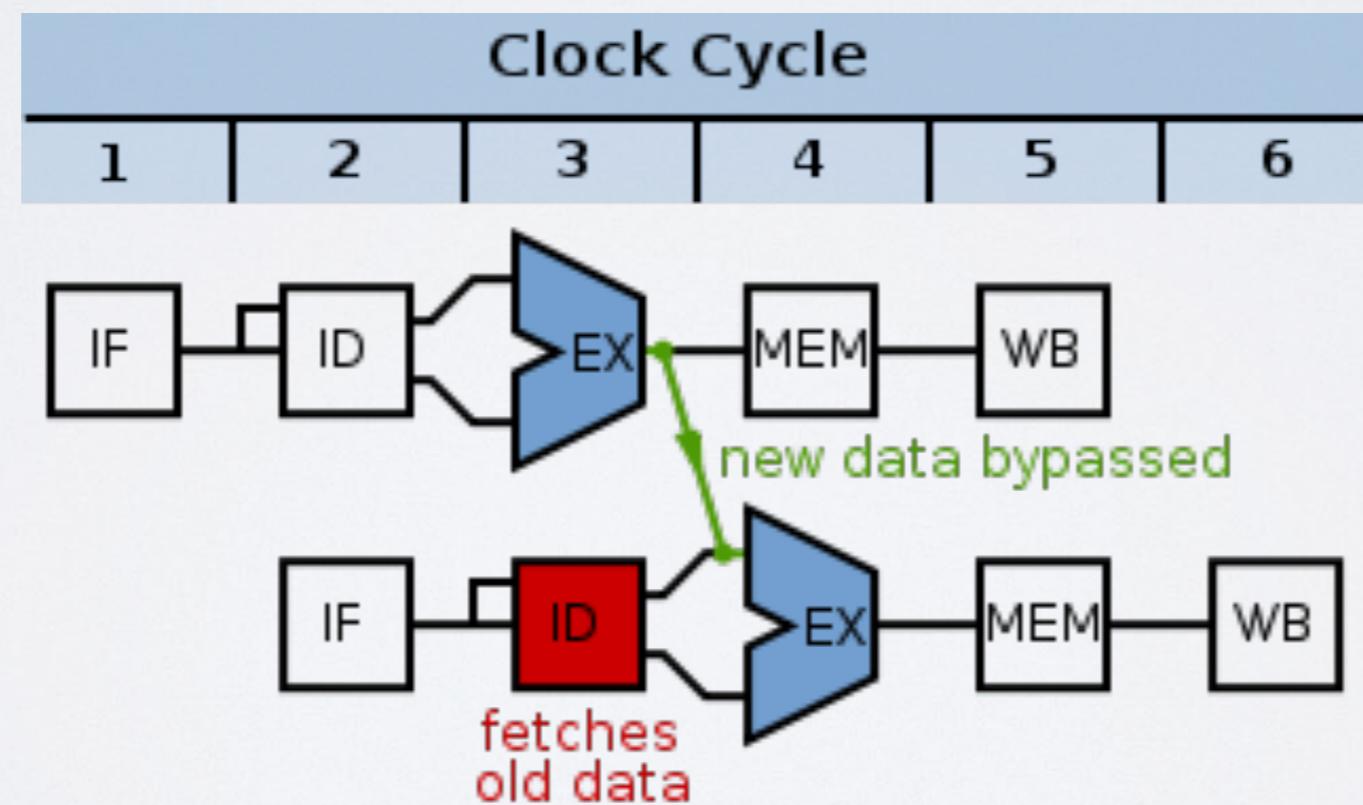
写标程

验数据



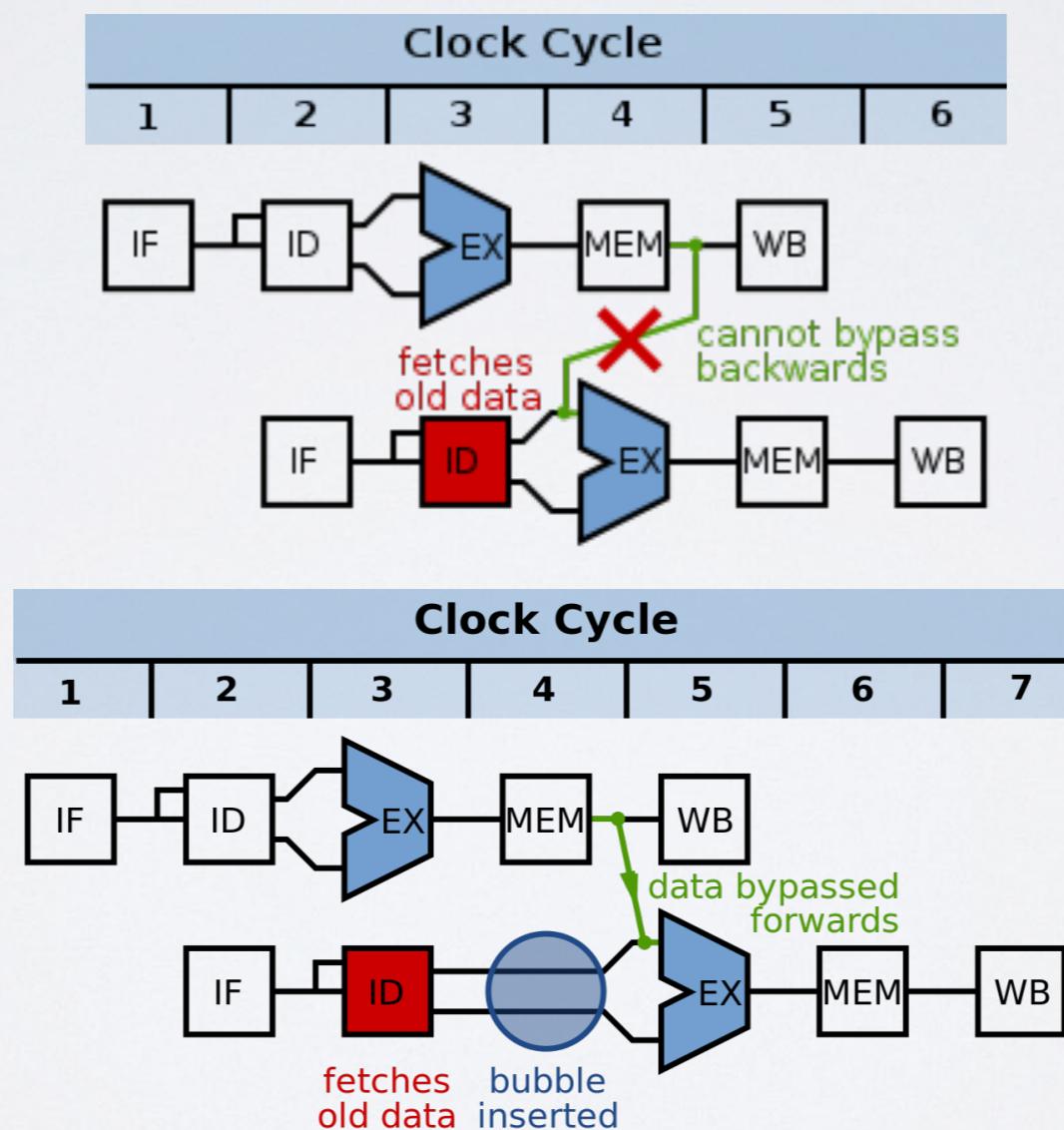
转发(FORWARDING)

- 在写回之前，就将数据直接传给后面的指令



暂停+转发

- 转发不一定总是能实现，有时候需要配合暂停来使用

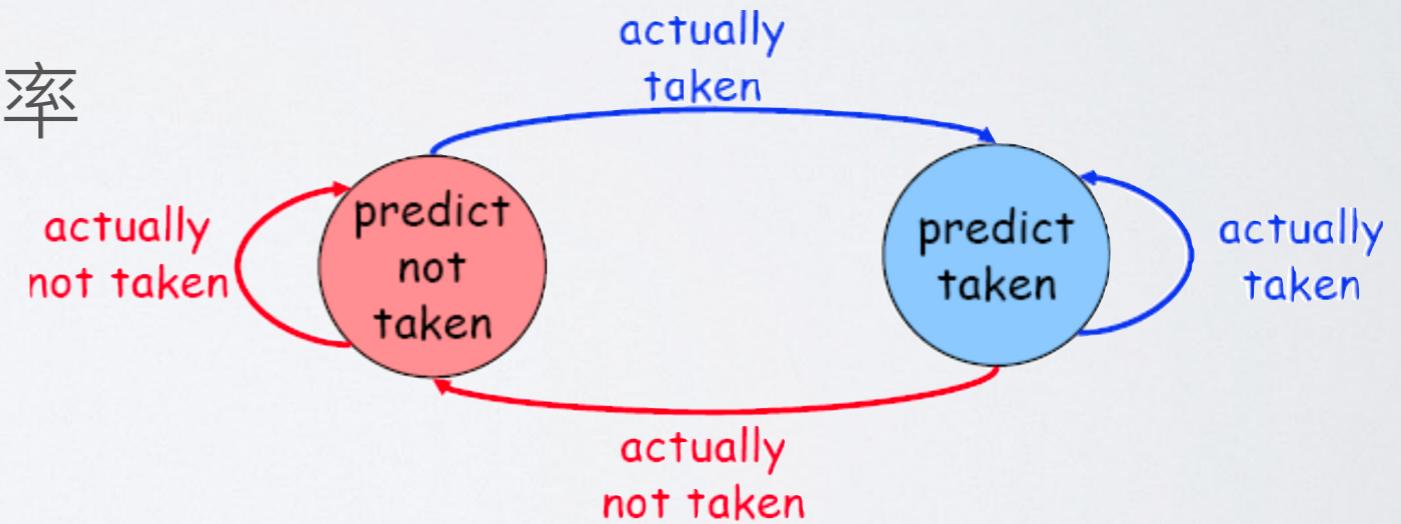


[https://upload.wikimedia.org/wikipedia/commons/thumb/1/15/Data_Forwarding_\(Two_Stage,_error\).svg/320px-Data_Forwarding_\(Two_Stage,_error\).svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/1/15/Data_Forwarding_(Two_Stage,_error).svg/320px-Data_Forwarding_(Two_Stage,_error).svg.png)

[https://upload.wikimedia.org/wikipedia/commons/thumb/d/d0/Data_Forwarding_\(Two_Stage\).svg/1280px-Data_Forwarding_\(Two_Stage\).svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/d/d0/Data_Forwarding_(Two_Stage).svg/1280px-Data_Forwarding_(Two_Stage).svg.png)

分支预测

- 当发生控制冒险时，可以用暂停+转发来解决，但是效率不高
- 如对于条件跳转指令，我们可以**预测**其是否发生跳转
预测正确 → 不会造成性能损失 | 预测错误 → 需要清除错误的部分
- 用一些方法来提高预测的准确率
- 现代CPU的分支预测，
已经能猜出很多分支模式
- 注意：分支预测错误的代价很大，在现代CPU上是数十个时钟周期

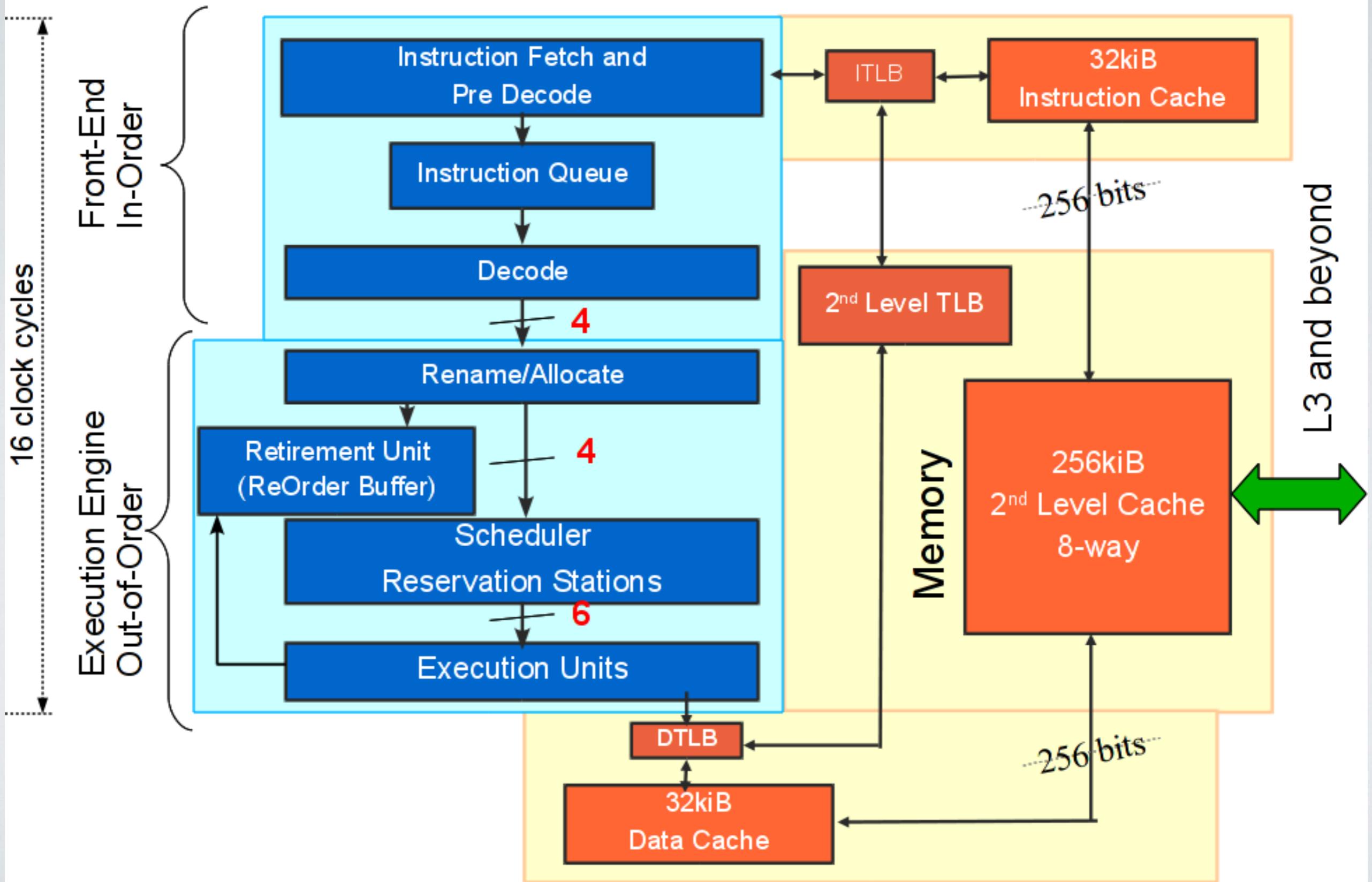


图片来源：清华大学“计算机组成原理”课程的课件

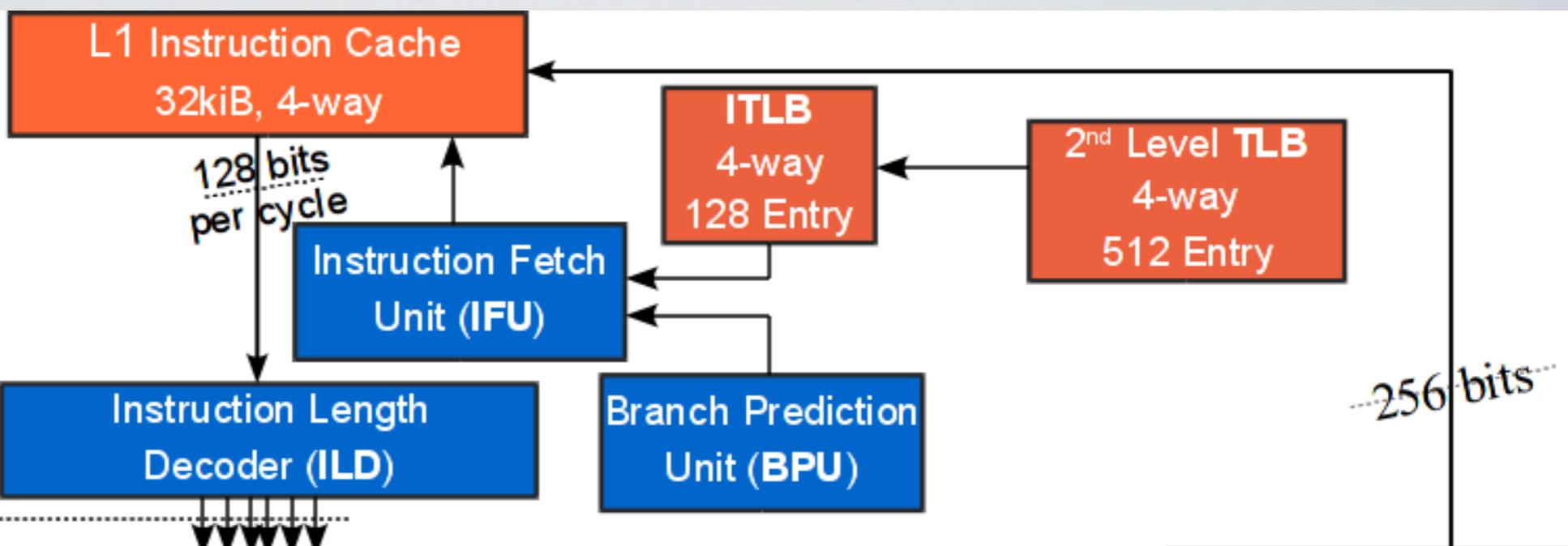
现代CPU的实现

- **超标量**: 在一个时钟周期内执行多个指令
乱序执行: 指令执行的顺序不一定按机器代码的顺序
- **微指令**: 一个指令在CPU中，被译码成多个基本的微指令
指令控制单元: 从内存中获取指令并译码，然后交给执行单元
执行单元: 能完成一些微指令的功能，并把结果交给退役单元
退役单元: 按顺序将指令的结果写回

Nehalem Core Pipeline



Intel64 CISC
macro-instructions



Nehalem
Front-End
In-order Pipeline

Nehalem RISC
micro-operations

6 macro
instructions
per cycle
max

4 macro
instructions
per cycle
max

4 micro
instructions
per cycle
max

MS ROM for
complex instructions

complex instructions
4 μ-ops or more

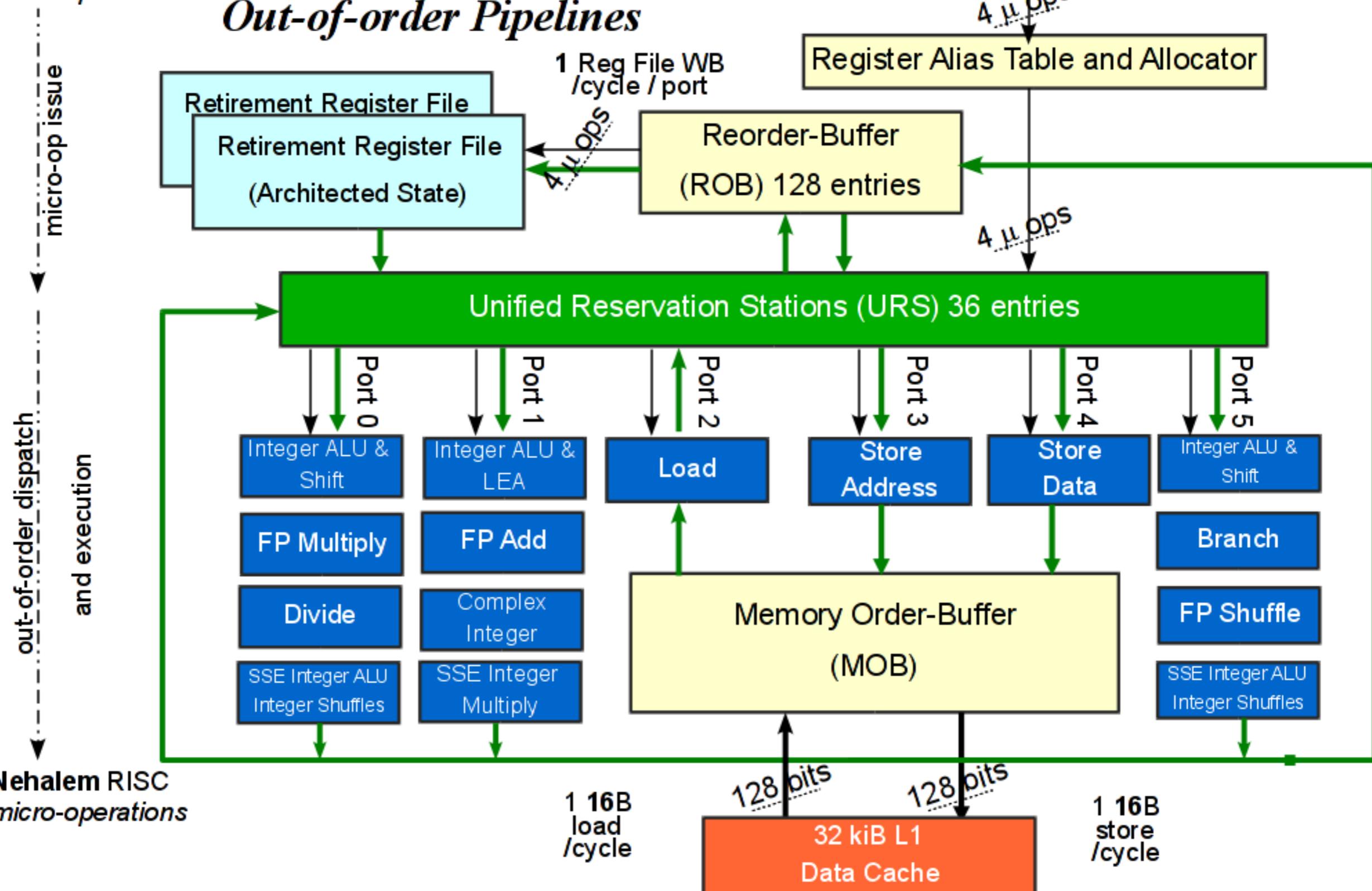
256kiB
2nd Level Cache
8-way

L3, remote
mem, etc.

4 micro-ops issued
per cycle (max)

Nehalem RISC
micro-operations

Nehalem Execution Engine Out-of-order Pipelines



一个现实中的例子

- 你是一个命题组的Boss
- 你要做很多命题、验题工作 →
- 你手下有4个人，每个人都
能做一部分类型的命题工作 →
- 你将任务分配给他们做，
然后你来管理他们即可

命题:	想idea	写题面	写标程	造数据
验题:	验题面	写暴力	写标程	验数据



想idea 想idea 写题面 写标程
写题面 写题面 写标程 造数据
验题面 写暴力 验数据

处理器体系结构

一些基础知识 

CPU的实现 

顺序实现的CPU 

流水线实现的CPU 

现代CPU的实现 

编程技巧

在CPU上优化程序

magic
bag of
tricks



在CPU上优化程序

减少不必要的计算

消除条件跳转

循环展开

减少不必要的计算

- 优化高维数组的寻址：用指针保存上一次使用的地址，直接加偏移

$$a[x][y][z] = a[x - 1][y][z] + b[x][y][z - 2]$$

- 对于同一个值的重复计算，存入临时变量中：

```
if (dis(i, j) < mi) mi = dis(i, j);
```

- 一阶线性递推的矩阵快速幂（带取模）：
使用 2×2 矩阵乘法 → 每步8次取模
经过观察，矩阵中只有两个数有用 → 每步2次取模

总结：精细地实现程序

消除条件跳转

- 这段程序的功能是将较小值放入 a 数组，较大值放入 b 数组
在第6代 Intel Core i5 上进行了测试
对于适合分支预测的数据，测得平均一次循环需要 4.0 个时钟周期
对于随机数据，测得平均一次循环需要 12.8 个时钟周期
- 可见，分支预测错误的**惩罚**为 17.6 个时钟周期

```
void minmax1(long *a, long *b, int n) {
    for (int i = 0; i < n; i++) {
        if (a[i] > b[i]) {
            long t = a[i];
            a[i] = b[i], b[i] = t;
        }
    }
}
```

消除条件跳转

- 用**三元运算符重写**，让编译器生成一种基于**条件传送**的汇编代码
测得不论数据如何，平均一次循环都只需要 4.1 个时钟周期

```
void minmax2(long *a, long *b, int n) {
    for (int i = 0; i < n; i++) {
        long mi = a[i] < b[i] ? a[i] : b[i];
        long ma = a[i] < b[i] ? b[i] : a[i];
        a[i] = mi, b[i] = ma;
    }
}
```

- 可能的使用情景：二路归并

循环展开

- 考虑以下程序
- 测得平均每个元素需要 3.65 个时钟周期

```
double sum(double *a, int n) {  
    double s = 0;  
    for (int i = 0; i < n; i++) {  
        s += a[i];  
    }  
    return s;  
}
```

循环展开

- 把程序变成这样
- 测得平均每个元素需要 1.36 个时钟周期

```
double sum(double *a, int n) {
    double s0 = 0, s1 = 0, s2 = 0, s3 = 0;
    for (int i = 0; i < n; i += 4) {
        s0 += a[i];
        s1 += a[i + 1];
        s2 += a[i + 2];
        s3 += a[i + 3];
    }
    return s0 + s1 + s2 + s3;
}
```

- 当展开次数过多时，性能反而下降，因为寄存器不够用 → 寄存器溢出
- 注意处理非展开次数的倍数的部分！

在CPU上优化程序

减少不必要的计算



消除条件跳转



循环展开



处理器体系结构

一些基础知识



CPU的实现



在CPU上优化程序



PART4: 存储器层次结构

存储器层次结构

现代存储技术

局部性与缓存

CPU的高速缓存

编程技巧

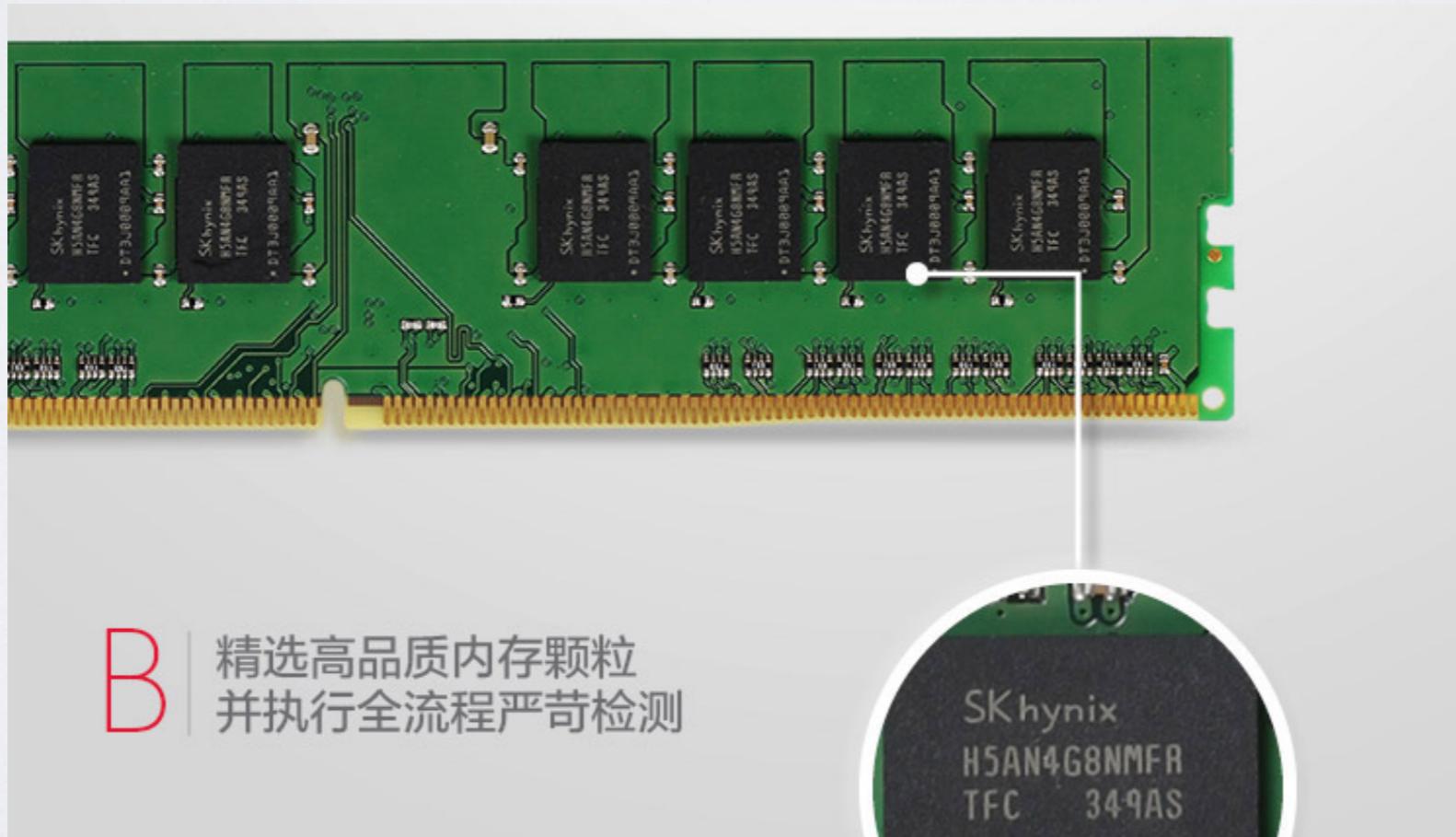
现代存储技术



<https://cnet4.cbsistatic.com/img/ONG0nOFT4pqSUEUqU2DlkmJ2NCM=/770x578/2014/04/24/5c7bea3b-a3a6-441f-b17c-86ee060127d4/dsc0065.jpg>

随机访问存储器 (RAM)

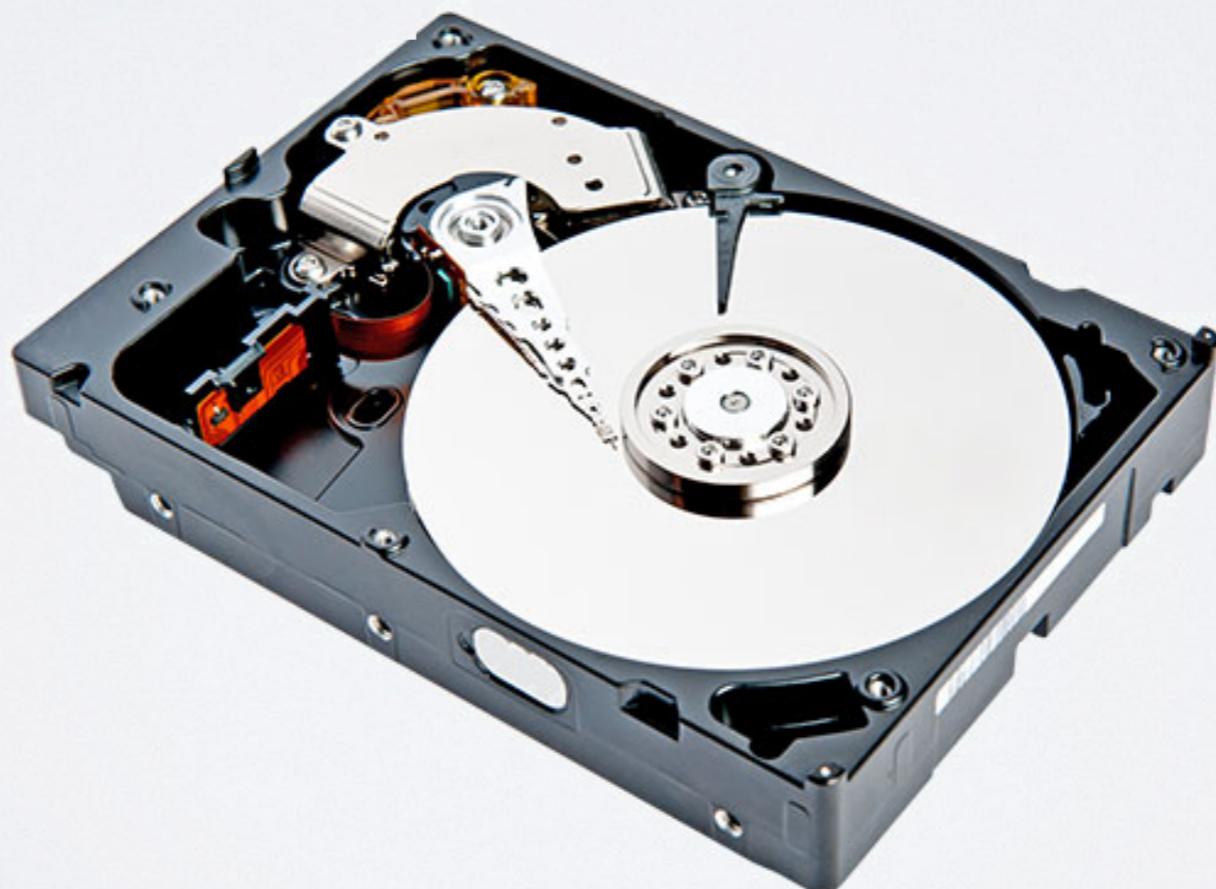
- 分为SRAM和DRAM，分别用晶体管和电容来实现
容量分别在**几M字节** (SRAM) 和**几十G字节** (DRAM)
前者用于CPU寄存器和缓存，后者用于计算机的主存 (即“内存”)



图片来源 <https://item.jd.com/1358804.html>

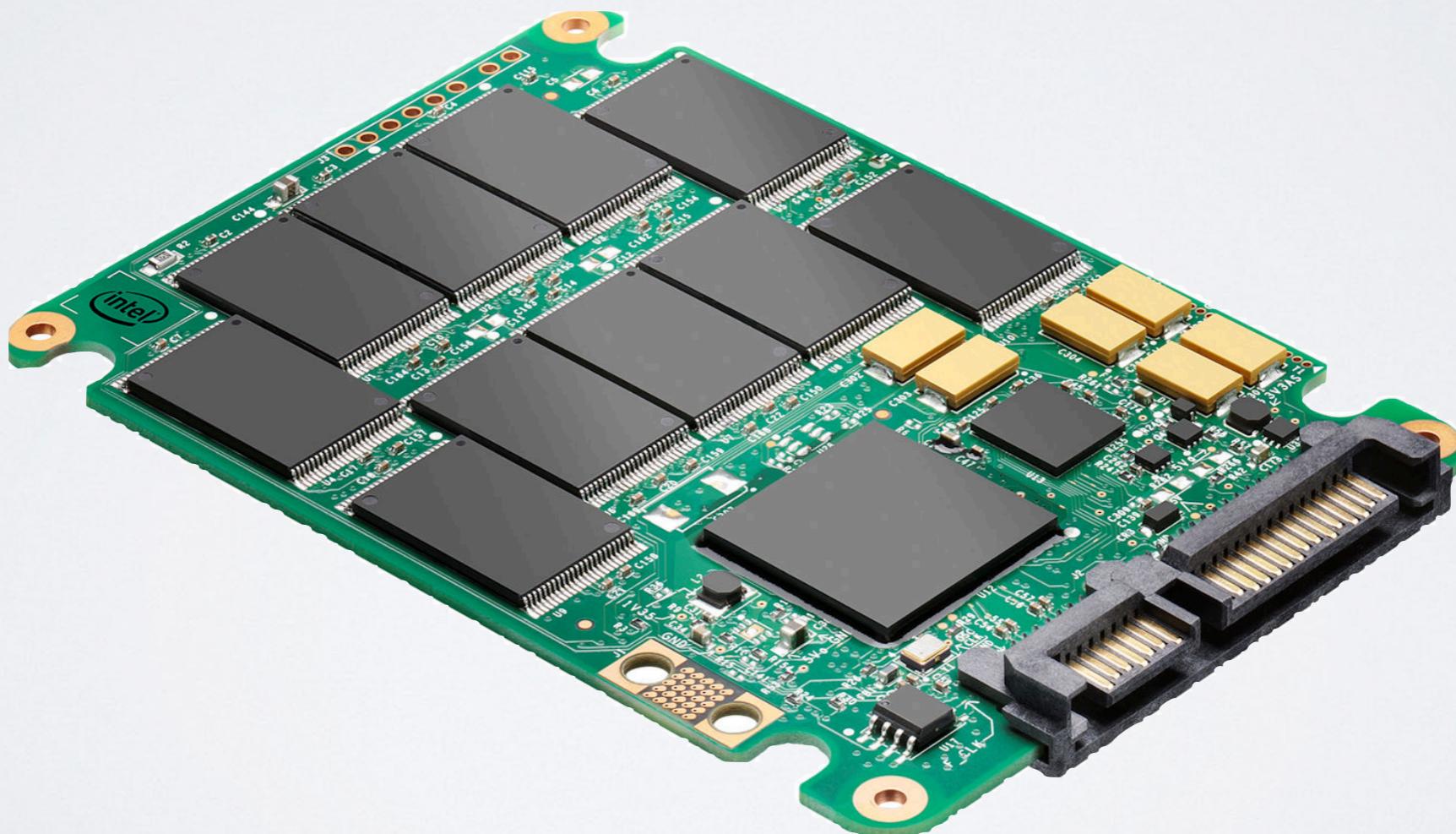
磁盘 (HDD)

- 一种基于磁性材料的存储设备，由若干个**高速旋转的盘片**构成
有一个可移动的**磁性读写头**，用来读取盘片上的数据
容量很大（数百GB到数TB），读写延迟也很大（数毫秒）



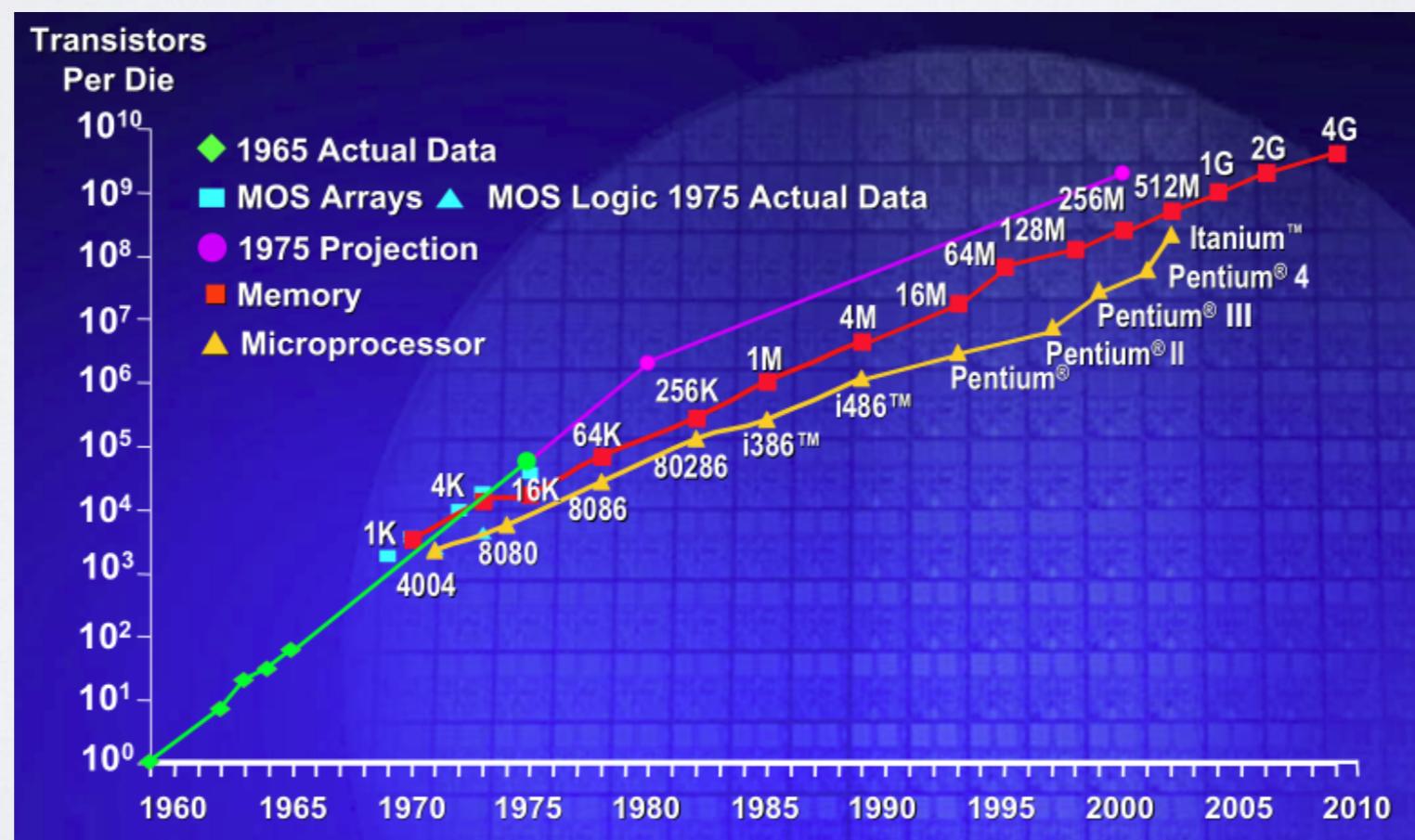
固态硬盘 (SSD)

- 一种基于**闪存**的存储设备，没有任何可移动的部件（“**固态**”），
读写速度比磁盘**快**一个数量级，价格也比磁盘**贵**一个数量级



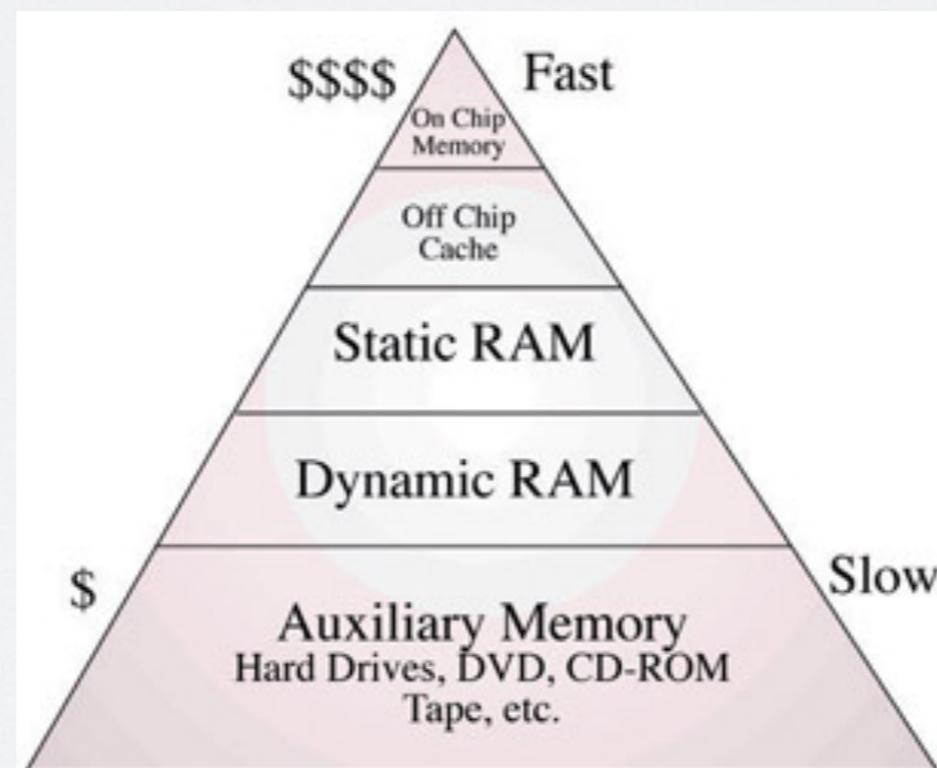
造价决定一切

- 如果CPU的寄存器可以有无限多，那么内存和硬盘就不再需要了？
- 如果内存能有无限大，那么把操作系统和文件直接装进内存可好？



存储器层次结构

- 不同类型的存储器的造价和速度的差异，必然使得计算机拥有一种“**存储器层次结构**”，离CPU越近的存储器越快，越小，越贵
- 如何在这种结构中提高计算机的性能？



局部性

- 程序先后访问的数据在某种意义上有关联
- **时间局部性：**
一个被访问过的内存地址，在短时间内可能被再次访问
- **空间局部性：**
一个被访问过的内存地址，其附近的地址在短时间内很可能被访问

空间局部性

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < m; j++) {  
        sum += a[i][j];  
    }  
}
```

```
for (int j = 0; j < m; j++) {  
    for (int i = 0; i < n; i++) {  
        sum += a[i][j];  
    }  
}
```

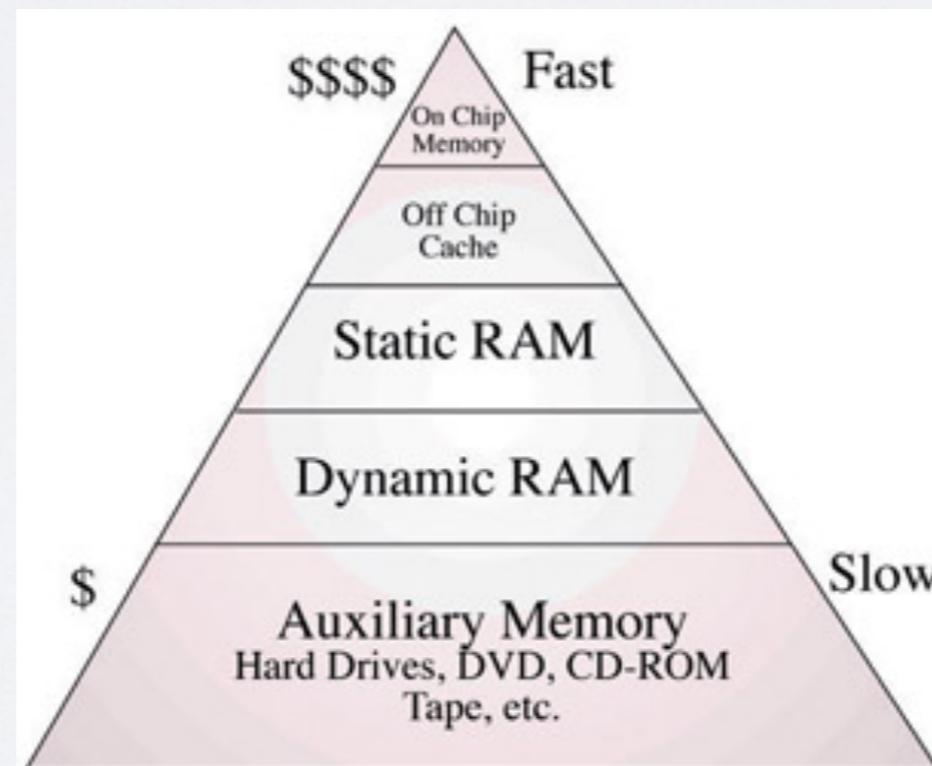
时间局部性

```
for (int i = 0; i < n; i++) {  
    sum += a[i] * (long long)i % 100;  
}
```

```
for (int i = 0; i < n; i++) {  
    sum += a[i] * (long long)i % 1000000;  
}
```

缓存

- 每一层存储器都作为下一层存储器的**缓存** (cache, 读作“cash”) , 包含下一层存储器的一部分内容。
访问某一层存储器时, 先访问上一层存储器,
如果**缓存命中**就可直接返回, 否则**缓存不命中**, 在本层查找。



计算机都有哪些缓存

类型	缓存什么	被缓存在何处	延迟（周期数）	由谁管理
CPU寄存器	4或8字节字	芯片上的CPU寄存器	0	编译器
TLB	地址翻译	芯片上的TLB	0	硬件MMU
L1高速缓存	64字节块	芯片上的L1高速缓存	4	硬件
L2高速缓存	64字节块	芯片上的L2高速缓存	10	硬件
L3高速缓存	64字节块	芯片上的L3高速缓存	50	硬件
虚拟内存	4KB页	主存	200	硬件+OS
缓冲区缓存	部分文件	主存	200	OS
磁盘缓存	磁盘扇区	磁盘控制器	100000	控制器固件
网络缓存	部分文件	本地磁盘	10000000	网络文件系统客户
浏览器缓存	Web页	本地磁盘	10000000	Web浏览器
Web缓存	Web页	远程服务器磁盘	1000000000	Web代理服务器

表格来源：《深入理解计算机系统》第425页

存储器层次结构

现代存储技术 ✓

局部性与缓存 ✓

CPU的高速缓存

编程技巧

CPU的高速缓存

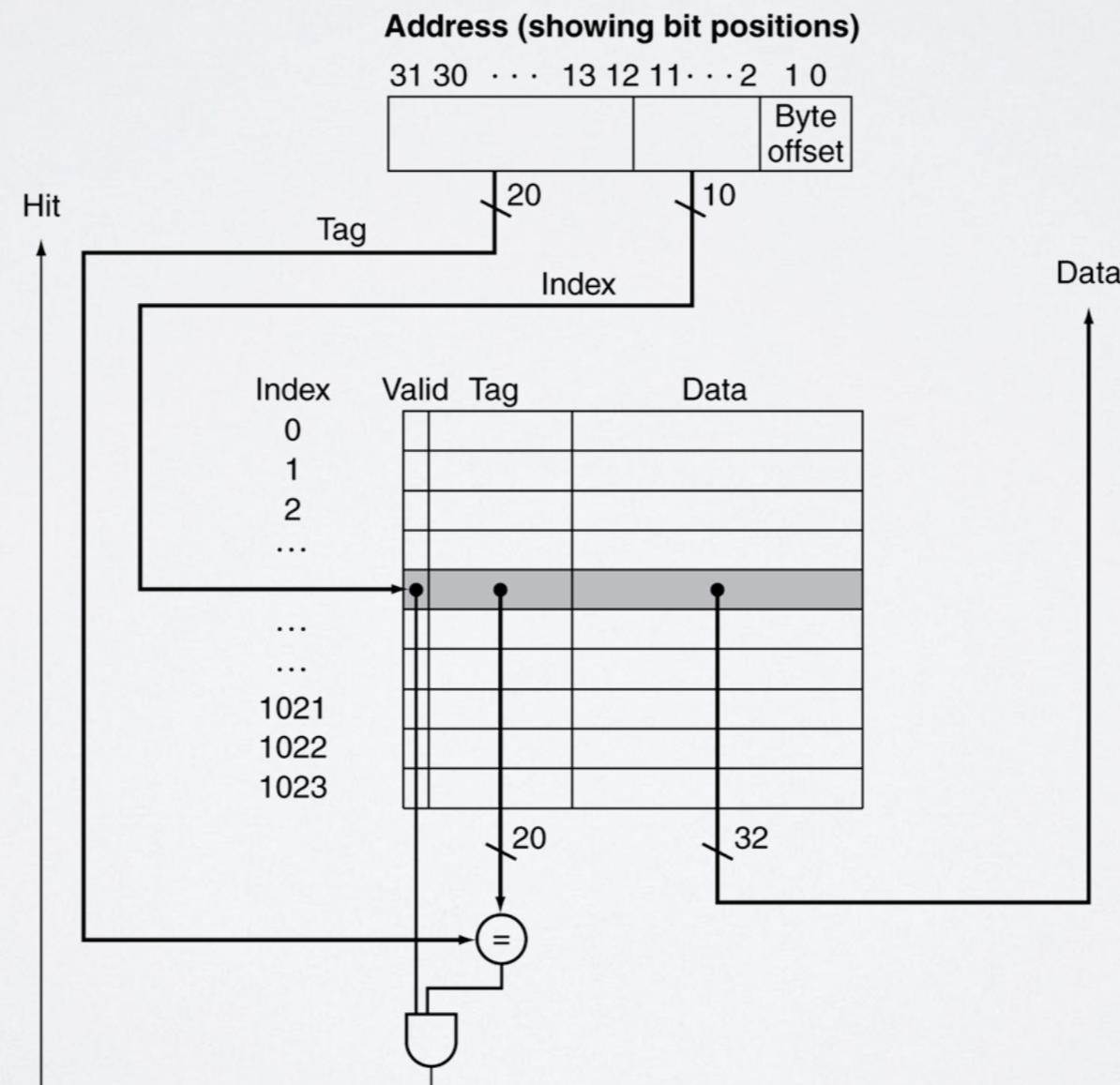
直接映射高速缓存

全相联高速缓存

组相联高速缓存

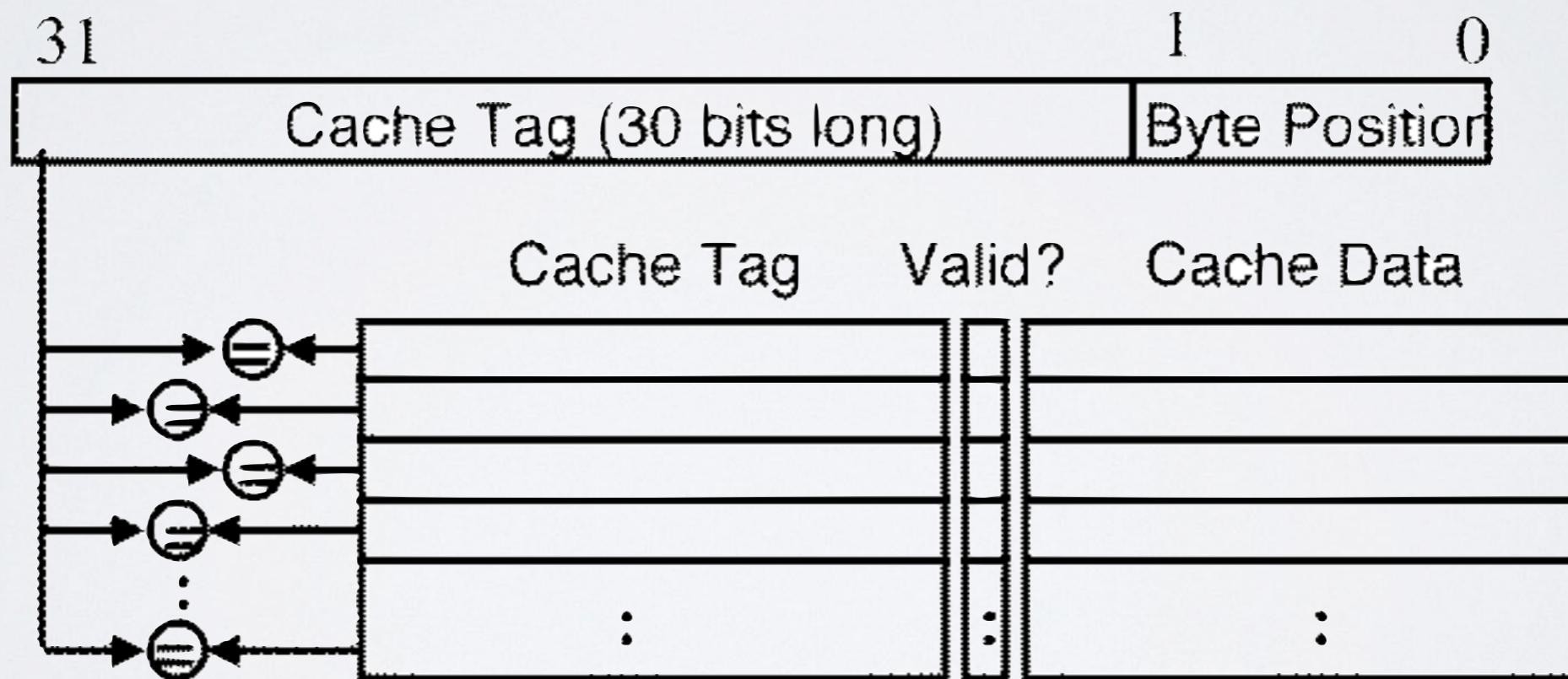
直接映射高速缓存

- 数据以 $B=2^b$ 大小的**块**为单位在不同存储器间传送。缓存分为 $S=2^s$ 个**组**，每个组只有一个块，根据内存地址后 $b + s$ 位确定**组编号**和**组内偏移**



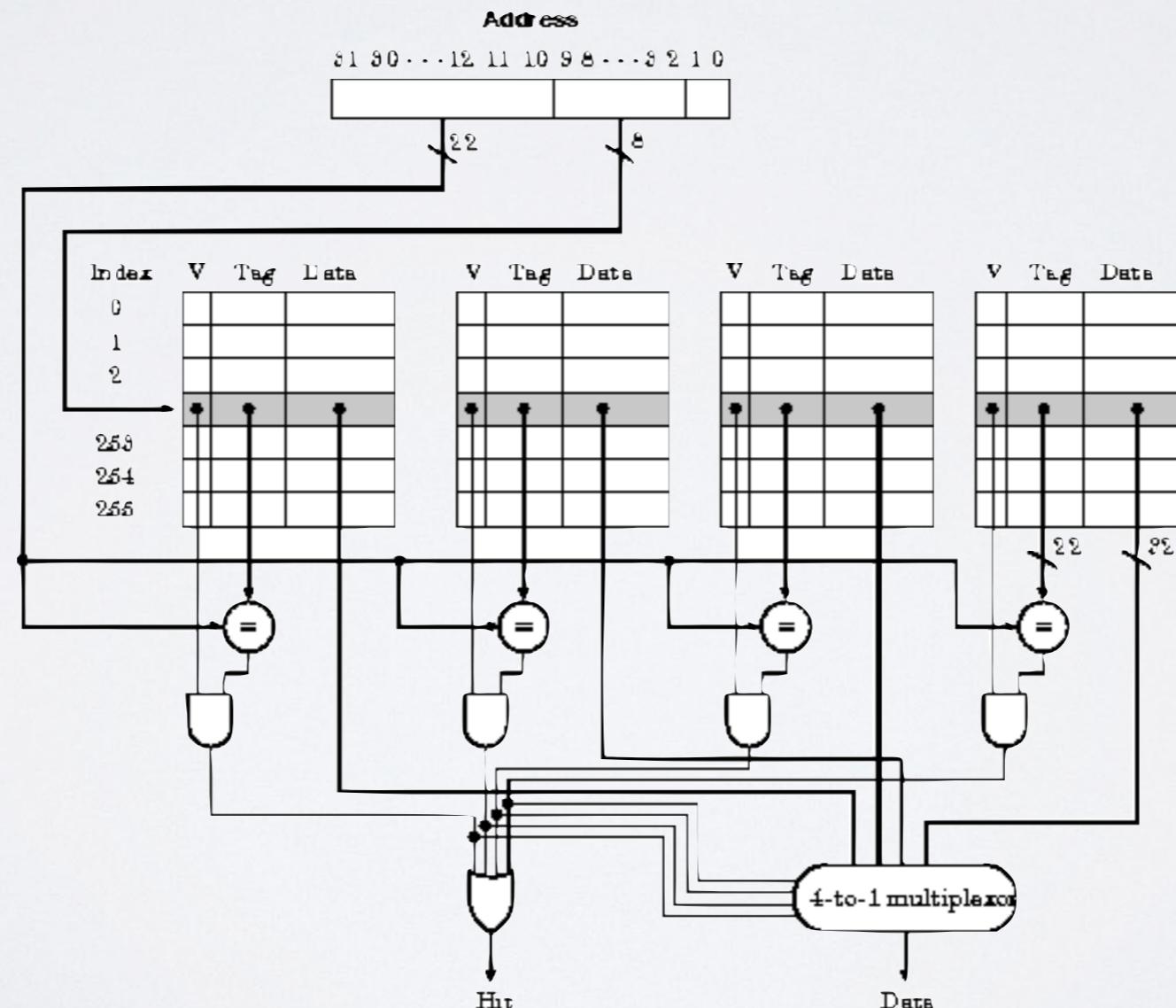
全相联高速缓存

- 只有一个组，分为 E 个行，每行一个块。任意内存地址都能映射到任意行
当缓存满时使用**最近最少使用** (LRU) 策略进行替换



组相联高速缓存

- 分为 $S=2^s$ 个组，每组又分为 E 个行。同一个内存地址只能映射到同一个组内的任意行。使用**最近最少使用** (LRU) 策略。



- **直接映射高速缓存**: 速度快, 造价低, 命中率低
 - 没啥用?
- **全相联高速缓存**: 速度慢, 造价高, 命中率高
 - 用于一些小的缓存, 如虚拟内存系统的TLB
- **组相联高速缓存**: 兼具前两者优点, 速度快, 命中率高
 - 在CPU的高速缓存中得到广泛应用

一个真实的例子

- 在某 Intel Core i5 CPU 上，有这些高速缓存：

高速缓存类型	访问时间 (周期)	高速缓存大小 (C)	相联度 (E)	块大小 (B)	组数 (S)
L1 I-Cache	4	32KB	8	64B	64
L1 D-Cache	4	32KB	8	64B	64
L2 Cache	约12	256KB	4	64B	512
L3 Cache	约50	6MB	12	64B	8192

存储器层次结构

现代存储技术 

局部性与缓存 

CPU的高速缓存 

编程技巧

编写高速缓存友好的代码

magic
bag of
tricks



一个例子

- 对于不同的 n 和 d , 反复调用这个程序, 具有不同的时空局部性
- 容易得知, n 越小, 时间局部性越好, d 越小, 空间局部性越好

```
long sum(long *a, int n, int d) {  
    long s = 0;  
    for (int i = 0; i < n; i++) s += a[i * d];  
    return s;  
}
```

空间局部性

- 我们在样机上测试了这个程序 (n 足够大) , 结果如下
- 与理论相符

d	1	2	3	4	8	16	32	64
周期数	1.50	2.34	3.46	4.73	9.70	15.00	19.76	20.26

时间局部性

- 我们在样机上测试了这个程序 ($n = 200$) , 结果如下

d	2^{19}	$2^{19} + 1$
周期数	159	1.18

- WHY? ? ?
- 200个整数, 显然能在L1缓存装得下?

时间局部性

- 对于 $d = 2^{19}$ ，每次内存访问时，地址的后 19 位都是一样的
- 根据CPU高速缓存的原理，这些地址必然会被映射到同一个组
- 因此，.....
- 缓存变成只有一组（12个行），159周期就是内存访问的速度啊！

d	2^{19}	$2^{19} + 1$
周期数	159	1.18

注：后19位一样的是虚拟地址，在映射成物理地址之后，由于操作系统分页的特性，也至少有后12位是一样的

如何编写缓存友好的代码

- **空间局部性好：**尽量使用步长为 1 的访问模式
 - 在遍历高维数组时很重要
- **时间局部性好：**使内存访问的工作集尽量小
 - 在统计整数二进制表示中 1 的个数时，分两段查表有时不如分三段好
- **避免使用步长为较大的 2 的幂的访问模式：**避免缓存冲突
 - 在状态压缩动态规划、使用高维数组时很重要
解决方法：把数组稍微开大一些

存储器层次结构

现代存储技术 

局部性与缓存 

CPU的高速缓存 

编写缓存友好的代码 

完结撒花！

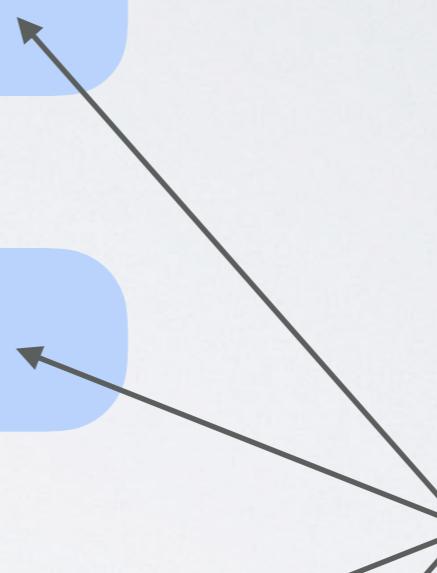
信息的表示和处理 ✓

程序的底层表示 ✓

处理器体系结构 ✓

存储器层次结构 ✓

编程
技巧



QUESTIONS ?



<https://oracle-base.com/blog/wp-content/uploads/2016/01/Questions.jpg>

THANKS!

- 感谢CCF给我提供这个交流的机会
感谢大家的认真聆听
- 祝大家优化愉快!
祝大家明天考试顺利!