

计算机系统导论笔记

ICS Note

作者: Sam

邮箱: 2300012435@stu.pku.edu.cn

创建: 2025 年 9 月 29 日

更新: 2025 年 10 月 31 日

目录

1 信息的表示和处理	3
1.1 信息存储	3
1.2 信息处理	4
1.3 整数	5
1.3.1 整数表示	5
1.3.2 整数运算	6
1.4 浮点数	8
1.4.1 浮点数表示	8
1.4.2 浮点数运算	9
1.4.3 浮点数转换	10
2 程序的机器级表示	11
2.1 汇编代码基础	11
2.1.1 寄存器	11
2.1.2 操作数	11
2.1.3 基础指令	12
2.2 控制	14
2.2.1 条件码	14
2.2.2 跳转指令	15
2.2.3 条件分支	16
2.2.4 循环	17
2.2.5 switch 语句	18
2.3 过程	19
2.3.1 运行时栈	20
2.3.2 转移控制	20
2.3.3 数据传送	20
2.4 数据结构	21
2.4.1 数组	21
2.4.2 结构	23
2.4.3 联合	23
2.4.4 浮点数	24
2.5 机器级程序进阶	25
2.5.1 类型声明	25
2.5.2 内存布局	26
2.5.3 缓冲溢出攻击	26
2.5.4 缓冲溢出保护	28

3 处理器体系架构	30
3.1 Y86-64 指令集体系结构	30
3.1.1 Y86-64 指令	30
3.1.2 Y86-64 异常	32
3.1.3 Y86-64 程序	32
3.2 逻辑设计	33
3.2.1 逻辑门	33
3.2.2 组合电路	34
3.2.3 字级的组合电路	35
3.2.4 存储器	38
3.3 Y86-64 的顺序实现	40
3.3.1 取指阶段	43
3.3.2 译码阶段	44
3.3.3 执行阶段	45
3.3.4 访存阶段	46
3.3.5 写回阶段	46
3.3.6 更新 PC 阶段	47
3.3.7 SEQ 的时序	47
3.4 Y86-64 的流水线实现	48
3.4.1 PC 选择和取指阶段	48
3.4.2 译码和写回阶段	48
3.4.3 执行阶段	49
3.4.4 访存阶段	49
3.4.5 控制逻辑	49
4 存储器层次结构	50
4.1 存储技术	50
4.1.1 随机访问存储器	50
4.1.2 磁盘存储	53
4.1.3 固态硬盘	57
4.2 缓存	58
4.2.1 存储器层次结构	58
4.2.2 存储器层次结构中的缓存	58
4.3 高速缓存存储器	59
4.3.1 通用的高速缓存存储器组织结构	59
4.3.2 直接映射高速缓存	60
4.3.3 组相联高速缓存	62
4.3.4 全相联高速缓存	64

4.3.5	高速缓存的写操作	65
4.3.6	高速缓存参数的性能影响	65
4.4	高速缓存友好的代码	66
4.4.1	存储器山	66
4.4.2	一维数组	66
4.4.3	二维数组	66
4.4.4	矩阵乘法	67
4.4.5	分块	68

1 信息的表示和处理

1.1 信息存储

进制转换

十六进制	十进制	二进制
0x0	0	0000
0x1	1	0001
0x2	2	0010
0x3	3	0011
0x4	4	0100
0x5	5	0101
0x6	6	0110
0x7	7	0111
0x8	8	1000
0x9	9	1001
0xA	10	1010
0xB	11	1011
0xC	12	1100
0xD	13	1101
0xE	14	1110
0xF	15	1111

基本数据类型的存储

数据类型	32 位	64 位
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

C 语言中字符串被编码为一个以 null (其值为 0) 字符结尾的字符数组。每个字符

都由 ASCII 字符码来表示。注意，十进制数字 x 的 ASCII 码正好是 $0x3x$ ，而终止字节的十六进制表示为 $0x00$ 。

寻址和字节顺序

多字节对象会被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。按照从最低有效字节到最高有效字节的顺序存储对象的方法称为小端法，反之则称为大端法。

- 小端法：x86, ARM 等。
- 大端法：Sun, Internet, PPC Mac 等。

1.2 信息处理

位级运算

- 按位与（AND）
- 按位或（OR）
- 按位异或（XOR）
- 按位取反（NOT）

位级运算的完备性

位级运算中的按位与（AND）、按位或（OR）和按位取反（NOT）是完备的，即可以通过它们的组合实现任何其他位级运算。例如，按位异或（XOR）可以通过以下表达式实现：

$$x \oplus y = (x \& \sim y) | (\sim x \& y)$$

常见的完备集有：

- {AND, NOT}
- {OR, NOT}
- {NAND}
- {NOR}

逻辑运算

- 逻辑与（AND）

- 逻辑或（OR）
- 逻辑非（NOT）

短路机制：在逻辑与（AND）和逻辑或（OR）运算中，如果第一个操作数已经能够确定整个表达式的值，则第二个操作数将不会被计算。

移位运算

- 左移
- 逻辑右移
- 算术右移

位运算的优先级

位运算符的优先级顺序从高到低依次为：

1. 按位取反 (`~`)
2. 加减乘除模 (`+ - * / %`)
3. 左移 (`<<`) 和右移 (`>>`)
4. 按位与 (`&`)
5. 按位异或 (`^`)
6. 按位或 (`|`)

细节可参看[C-CPP](#)。

1.3 整数

1.3.1 整数表示

类型

- 无符号整数 (`unsigned int`)

$$x = \sum_{i=0}^{w-1} b_i 2^i$$

$$UMin = 0, UMax = 2^w - 1$$

- 有符号整数 (signed int)

$$x = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

$$TMin = -2^{w-1}, TMax = 2^{w-1} - 1$$

处理同样字长的有符号数和无符号数之间相互转换的一般规则是：数值可能会改变，但是位模式不变。

拓展

- 零拓展 (zero extension): 用于无符号数的拓展，高位补 0。
- 符号拓展 (sign extension): 用于有符号数的拓展，高位补符号位。

截断

截断是指将一个较长字长的整数转换为较短字长的整数。截断时，只保留低位的若干位，高位被丢弃。

C 语言中的隐式类型转换

在 C 语言中，不同类型整数进行比较时，所有比 int 类型小的整数类型（如 char、short 等）会首先被提升为 int 类型。如果两个数宽度不同（例如 int 和 long），宽度较小的类型会转换为宽度较大的类型。最后，如果比较涉及有符号类型和无符号类型，有符号类型会被转换为无符号类型再进行对比。

1.3.2 整数运算

加减法

无符号数：

$$x +_w^u y = \begin{cases} x + y & x + y < 2^w \\ x + y - 2^w & x + y \geq 2^w \end{cases}$$

$$-_w^u x = \begin{cases} x & x = 0 \\ 2^w - x & x > 0 \end{cases}$$

符号数：

$$x +_w^t y = \begin{cases} x + y - 2^w & 2^{w-1} \leq x + y \\ x + y & -2^{w-1} < x + y < 2^{w-1} \\ x + y + 2^w & x + y \leq -2^{w-1} \end{cases}$$

$$-_w^t x = \begin{cases} TMin_w & x = TMin_w \\ -x & x > TMin_w \end{cases}$$

乘除法

无符号数:

- 乘法

$$x *_w^u y = (x \cdot y) mod 2^w$$

- 乘 2 的幂

$$x *_w^u 2^k = x << k$$

- 除以 2 的幂 (逻辑右移, 向下舍入)

$$x /_w^u 2^k = x >> k$$

符号数:

- 乘法

$$x *_w^t y = U2T_w((x \cdot y) mod 2^w)$$

- 乘 2 的幂

$$x *_w^t 2^k = x << k$$

- 除以 2 的幂 (算术右移, 向下舍入)

$$x /_w^t 2^k = x >> k$$

- 除以 2 的幂 (算术右移, 向上舍入)

$$x /_w^t 2^k = (x + (1 << k) - 1) >> k$$

注意, C 语言中的 / 运算符对于整数除法采用向零舍入方式。

整数乘法的二进制低位结果

假设有两个 n 位的二进制数, 其位模式分别为 A 和 B 。作为无符号整数, 它们的值即为 A 和 B 。它们的数学乘积是 $A \cdot B$ 。在计算机中, 仅保留低 n 位, 相当于计算 $(A \cdot B) \bmod 2^n$ 。若将 A 和 B 解释为有符号整数 (补码), 设 $a = A - 2^n \cdot \text{sign}(A)$, $b = B - 2^n \cdot \text{sign}(B)$, 其中 $\text{sign}(X)$ 在 X 为负数时为 1, 否则为 0。有符号乘积的低 n 位为 $(a \cdot b) \bmod 2^n$, 与无符号乘积的低 n 位相同。推导如下:

$$\begin{aligned} (a \cdot b) \bmod 2^n &= [(A - 2^n \cdot \text{sign}(A)) \cdot (B - 2^n \cdot \text{sign}(B))] \bmod 2^n \\ &= (A \cdot B) \bmod 2^n \end{aligned}$$

1.4 浮点数

1.4.1 浮点数表示

二进制小数

$$b = \sum_{k=-j}^i b_k \cdot 2^k$$

IEEE 754 标准

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

- 符号 (sign): s 表示数值的正负。
- 阶码 (exponent): E 的作用是对浮点数加权，这个权重是 2 的 E 次幂。
- 尾数 (mantissa): M 是一个二进制小数。

将浮点数的位表示划分为三个字段，分别对这些值进行编码：

- 一个单独的符号位 s 直接编码符号 s。
- k 位的阶码字段 $exp = e_{k-1} \dots e_1 e_0$ 编码阶码 E。
- n 位小数字段 $frac = f_{n-1} \dots f_1 f_0$ 编码尾数 M，但是编码出来的值也依赖于阶码字段的值是否等于 0。

类型	总位数	符号位 (s)	阶码位 (k)	尾数位 (n)
单精度 (float)	32	1	8	23
双精度 (double)	64	1	11	52

分类

- 规格化的值 (Normalized): 阶码字段不全为 0，尾数字段有有效数字。

$$E = exp - Bias$$

$$M = 1.f_{n-1}f_{n-2} \dots f_0$$

其中， $Bias = 2^{k-1} - 1$ 。

- 非规格化的值 (Denormalized): 阶码字段全为 0 , 尾数字段有有效数字。

$$E = 1 - Bias = -(2^{k-1} - 2)$$

$$M = 0.f_{n-1}f_{n-2}\dots f_0$$

- 无穷大 (Infinity): 阶码字段全为 1 , 尾数字段全为 0 。
- NaN (Not a Number): 阶码字段全为 1 , 尾数字段非全 0 。

1.4.2 浮点数运算

舍入

- 向零舍入
- 向上舍入
- 向下舍入
- 向偶数舍入 (默认)

乘法

$$(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$$

1. 符号计算: 结果的符号位为两个操作数符号位的异或。

$$s = s_1 \oplus s_2$$

2. 阶码相加: 将两个操作数的阶码相加。

$$E = E_1 + E_2$$

3. 尾数相乘: 将两个操作数的尾数相乘。

$$M = M_1 \times M_2$$

4. 规格化: 如果结果的尾数不在规范化范围内, 则对其进行规格化处理。
5. 溢出检查: 如果结果的阶码超出表示范围, 则将结果设为无穷大。
6. 舍入: 根据所选的舍入模式对结果进行舍入。

加法

$$(-1)^{s_1} M_1 2^{E_1} + (-1)^{s_2} M_2 2^{E_2}, E_1 \geq E_2$$

1. 对阶：将阶码较小的数的尾数右移，直到两个数的阶码相等。
2. 尾数相加：根据符号位对尾数进行加减运算。

$$M = (-1)^{s_1} M_1 + (-1)^{s_2} M_2 2^{E_2 - E_1}$$

3. 规格化：如果结果的尾数不在规范化范围内，则对其进行规格化处理。
4. 溢出检查：如果结果的阶码超出表示范围，则将结果设为无穷大。
5. 舍入：根据所选的舍入模式对结果进行舍入。

1.4.3 浮点数转换

- int - float：数字不会溢出，但是可能被舍入。
- int/float - double：数字不会溢出，也不会舍入。
- double - float：可能会溢出，也可能会舍入。
- float,double - int：会向零舍入，可能会溢出。

2 程序的机器级表示

2.1 汇编代码基础

2.1.1 寄存器

寄存器				备注
64 位	32 位	16 位	8 位	
%rax	%eax	%ax	%al	函数返回值 (accumulator)
%rbx	%ebx	%bx	%bl	基址寄存器 (base)
%rcx	%ecx	%cx	%cl	计数器 (counter)
%rdx	%edx	%dx	%dl	数据寄存器 (data)
%rsi	%esi	%si	%sil	源变址寄存器 (source index)
%rdi	%edi	%di	%dil	目的变址寄存器 (destination index)
%rsp	%esp	%sp	%spl	堆栈指针寄存器 (stack pointer)
%rbp	%ebp	%bp	%bpl	基址指针寄存器 (base pointer)
%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	/

当指令以寄存器作为目标时，生成小于 8 字节结果时，寄存器有两条规则：

- 生成 1 字节和 2 字节数字的指令会保持剩下的字节不变。
- 生成 4 字节数字的指令会把高位 4 个字节置为 0。

2.1.2 操作数

操作数类型	格式	数值
寄存器	r_a	$R[r_a]$
立即数	$\$Imm$	Imm
存储器	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$

注意：

- 比例因子 s 必须是 1、2、4 或者 8。
- 基址和变址寄存器都必须是 64 位寄存器。

2.1.3 基础指令

数据传送指令

指令	效果	描述
MOV S, D	$D \leftarrow S$	传送
moveb		传送字节
movew		传送字
movel		传送双字
moveq		传送四字
moveabsq I, R	$R \leftarrow I$	传送绝对四字
MOVZ S, R	$R \leftarrow$ 零扩展 (S)	以零扩展进行传送
movezbw		将做了零扩展的字节传送到字
movezbl		将做了零扩展的字节传送到双字
movezwl		将做了零扩展的字传送到双字
movezbq		将做了零扩展的字节传送到四字
movezwq		将做了零扩展的字传送到四字
MOVS S, R	$R \leftarrow$ 符号扩展 (S)	以符号扩展进行传送
movesbw		将做了符号扩展的字节传送到字
movesbl		将做了符号扩展的字节传送到双字
moveswl		将做了符号扩展的字传送到双字
movesbq		将做了符号扩展的字节传送到四字
moveswq		将做了符号扩展的字传送到四字
moveslq		将做了符号扩展的双字传送到四字
cltq	$\%rax \leftarrow$ 符号扩展 ($\%eax$)	把 $\%eax$ 符号扩展到 $\%rax$

注意：

- movl 指令以寄存器作为目的时，它会把该寄存器的高位 4 字节设置为 0。
- 常规的 movq 指令只能以表示为 32 位补码数字的立即数作为源操作数，然后把这个值符号扩展得到 64 位的值，放到目的位置。
- movabsq 指令能够以任意 64 位立即数值作为源操作数，并且只能以寄存器作为目的。

压栈弹栈指令

指令	效果	描述
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8$ $M[R[\%rsp]] \leftarrow S$	将四字压入栈
popq D	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	将四字弹出栈

算术运算指令

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
neg D	$D \leftarrow -D$	取负
not D	$D \leftarrow \sim D$	取补
add S, D	$D \leftarrow D + S$	加
sub S, D	$D \leftarrow D - S$	减
imul S, D	$D \leftarrow D * S$	乘
xor S, D	$D \leftarrow D \oplus S$	异或
and S, D	$D \leftarrow D \& S$	与
or S, D	$D \leftarrow D S$	或
sal k, D	$D \leftarrow D << k$	左移
shl k, D	$D \leftarrow D << k$	左移
sar k, D	$D \leftarrow D >>_A k$	算术右移
shr k, D	$D \leftarrow D >>_L k$	逻辑右移
imulq S	$R[\%rdx] : R[\%rax] \leftarrow S * R[\%rax]$	有符号全乘法
mulq S	$R[\%rdx] : R[\%rax] \leftarrow S * R[\%rax]$	无符号全乘法
cqto	$R[\%rdx : \%rax] \leftarrow$ 符号扩展 ($R[\%rax]$)	转换为八字
idivq S	$R[\%rdx] \leftarrow R[\%rdx : \%rax] \bmod S$ $R[\%rax] \leftarrow R[\%rdx : \%rax] \div S$	有符号除法
divq S	$R[\%rdx] \leftarrow R[\%rdx : \%rax] \bmod S$ $R[\%rax] \leftarrow R[\%rdx : \%rax] \div S$	无符号除法

注意：

- 移位指令只有一个操作数 D 时，代表将 D 的值左移或右移 1 位。

- 位移指令的移位量只能存放在单字节寄存器%cl 中，或者作为立即数给出。

2.2 控制

2.2.1 条件码

条件码定义

标志	描述	含义
ZF	Zero Flag	结果为 0 时置 1，否则为 0
SF	Sign Flag	结果为负时置 1，否则为 0
OF	Overflow Flag	有符号溢出时置 1，否则为 0
CF	Carry Flag	无符号溢出时置 1，否则为 0

注意：

- leaq 指令不改变任何条件码，因为它是用来进行地址计算的。
- 对于 XOR, 进位标志和溢出标志会设置成 0。
- 对于移位操作，进位标志将设置为最后一个被移出的位，而溢出标志设置为 0。
- INC 和 DEC 指令会设置溢出和零标志，但是不会改变进位标志。

条件码访问

指令	效果		描述
CMP S_1, S_2	$S_2 - S_1$		比较
TEST S_1, S_2	$S_1 \& S_2$		测试
SET D			设置单字节
sete/setz D	$D \leftarrow ZF$		相等/零
setne/setnz D	$D \leftarrow \sim ZF$		不等/非零
sets D	$D \leftarrow SF$		负数
setns D	$D \leftarrow D \sim SF$		非负数
setg/setnle D	$D \leftarrow \sim (SF \oplus OF) \& \sim ZF$		有符号大于
setge/setnl D	$D \leftarrow \sim (SF \oplus OF)$		有符号大于等于
setl/setnge D	$D \leftarrow SF \oplus OF$		有符号小于
setle/setng D	$D \leftarrow (SF \oplus OF) ZF$		有符号小于等于
seta/setnbe D	$D \leftarrow \sim CF \& \sim ZF$		无符号大于
setae/setnb D	$D \leftarrow \sim CF$		无符号大于等于
setb/setnae D	$D \leftarrow CF$		无符号小于
setbe/setna D	$D \leftarrow CF ZF$		无符号小于等于

2.2.2 跳转指令

指令	跳转条件		描述
jmp Label	1		直接跳转
jmp *Operand	1		间接跳转
je/jz Label	ZF		相等/零
jne/jnz Label	$\sim ZF$		不等/非零
js Label	SF		负数
jns Label	$\sim SF$		非负数
jg/jnle Label	$\sim (SF \oplus OF) \& \sim ZF$		有符号大于
jge/jnl Label	$\sim (SF \oplus OF)$		有符号大于等于
jl/jnge Label	$SF \oplus OF$		有符号小于
jle/jng Label	$(SF \oplus OF) ZF$		有符号小于等于
ja/jnbe Label	$\sim CF \& \sim ZF$		无符号大于
jae/jnb Label	$\sim CF$		无符号大于等于
jb/jnae Label	CF		无符号小于
jbe/jna Label	$CF ZF$		无符号小于等于

2.2.3 条件分支

条件控制

```

1 t = ...; // test-expr
2 if (!t)
3     goto false;
4 ... // then-statement
5 goto done;
6 false:
7     ... // else-statement
8 done:
9 ...

```

条件传送

```

1 /* result = test-expr ? then-expr : else-expr; */
2 t = ...; // test-expr
3 result = ...; // then-expr
4 eval = ...; // else-expr
5 if (!t) result = eval;

```

注意，无论测试结果如何，then-expr 和 else-expr 都会被求值。如果这两个表达式中的任意一个可能产生错误条件或者副作用，就会导致非法的行为。

指令	传送条件	描述
cmove/cmovz cmovne/cmovnz	S, R $\sim ZF$	相等/零 不等/非零
cmovs cmovns	S, R $\sim SF$	负数 非负数
cmovg/cmovnle cmovge/cmovnl cmovl/cmovnge cmovle/cmovng	$\sim (SF \oplus OF) \& \sim ZF$ $\sim (SF \oplus OF)$ $SF \oplus OF$ $(SF \oplus OF) ZF$	有符号大于 有符号大于等于 有符号小于 有符号小于等于
cmova/cmovnbe cmovae/cmovnb cmovb/cmovnae cmovbe/cmovna	S, R $\sim CF \& \sim ZF$ $\sim CF$ CF $CF ZF$	无符号大于 无符号大于等于 无符号小于 无符号小于等于

源和目的的值可以是 16 位、32 位或 64 位长，不支持单字节的条件传送。

2.2.4 循环

do-while 循环

```
1 loop:  
2     ... // body-statement  
3     t = test-expr;  
4     if (t)  
5         goto loop;
```

while 循环

```
1 /* jump to middle */  
2     goto test;  
3 loop:  
4     ... // body-statement  
5 test:  
6     t = test-expr;  
7     if (t)  
8         goto loop;
```

```
1 /* guarded-do */  
2 t = test-expr;  
3 if (!t)  
4     goto done;  
5 loop:  
6     ... // body-statement  
7     t = test-expr;  
8     if (t)  
9         goto loop;  
10 done:  
11    ...
```

for 循环

```
1     init-expr;  
2     goto test;  
3 loop:  
4     ... // body-statement
```

```

5     update-expr;
6 test:
7     t = test-expr;
8     if (t)
9         goto loop;

```

```

1     init-expr;
2     t = test-expr;
3     if (!t)
4         goto done;
5 loop:
6     ... // body-statement
7     update-expr;
8     t = test-expr;
9     if (t)
10        goto loop;
11 done:
12 ...

```

2.2.5 switch 语句

switch 语句可以根据一个整数索引值进行多重分支，通过使用跳转表这种数据结构使得实现更加高效。

```

1     cmpl    $0, %rdi           ; if index < 0 -> default
2     jl      .Ldefault
3     cmpl    $2, %rdi           ; if index > 2 -> default
4     jg      .Ldefault
5     jmp     *.%Ljumptable(%rdi,8) ; indirect jump via table
6 .Lcase0:
7     ; Case 0 actions
8     ;
9     jmp .Ldone
10 .Lcase1:
11    ; Case 1 actions
12    ;
13    jmp .Ldone
14 .Lcase2:
15    ; Case 2 actions
16    ;

```

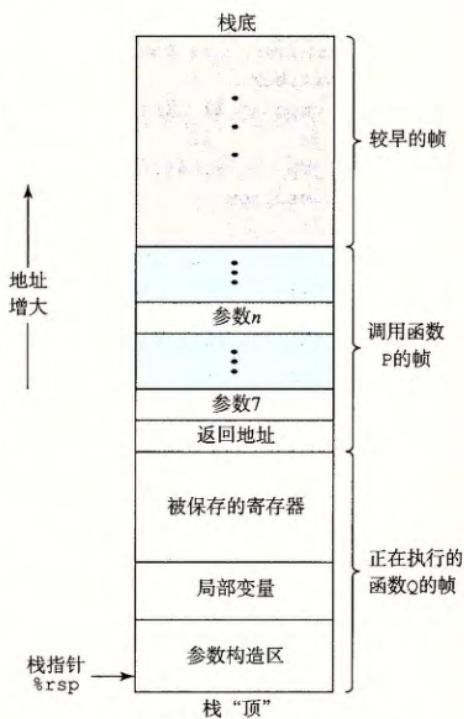
```
17    jmp .Ldone
18 .Ldefault:
19     ; Default actions
20     ; ...
21     jmp .Ldone
22 .Ldone:
23     ; ...
24
25     .section .rodata
26     .align 8
27 .Ljumptable:
28     .quad .Lcase0
29     .quad .Lcase1
30     .quad .Lcase2
31     .quad .Ldefault
```

2.3 过程

过程是软件中一种很重要的抽象。它提供了一种封装代码的方式，用一组指定的参数和一个可选的返回值实现了某种功能。假设过程 P 调用过程 Q，Q 执行后返回到 P。这些动作包括下面一个或多个机制：

- 传递控制：在进入过程 Q 的时候，程序计数器必须被设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 P 中调用 Q 后面那条指令的地址。
- 传递数据：P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 返回一个值。
- 分配和释放内存：在开始时，Q 可能需要为局部变量分配空间，而在返回前，又必须释放这些存储空间。

2.3.1 运行时栈



2.3.2 转移控制

将控制从函数 P 转移到函数 Q 只需要简单地把程序计数器设置为 Q 的代码的起始位置。不过，当稍后从 Q 返回的时候，处理器必须记录好它需要继续 P 的执行的代码位置。

指令	描述
call Label	过程调用
call *Operand	过程调用
ret	从过程调用中返回

2.3.3 数据传送

当调用一个过程时，除了要把控制传递给它并在过程返回时再传递回来之外，过程调用还可能包括把数据作为参数传递。`x86-64` 中，可以通过寄存器最多传递 6 个整型（例如整数和指针）参数，大部分过程间的数据传送是通过寄存器实现的。

寄存器的使用是有特殊顺序的，寄存器使用的名字取决于要传递的数据类型的小。

操作数大小 (位)	参数数量					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

如果一个函数有大于 6 个整型参数，超出 6 个的部分就要通过栈来传递。

栈上的局部存储

有些时候，局部数据必须存放在内存中，常见的情况包括：

- 寄存器不足够存放所有的本地数据。
- 对一个局部变量使用地址运算符 ‘&’，因此必须能够为它产生一个地址。
- 某些局部变量是数组或结构，因此必须能够通过数组或结构引用被访问到。

寄存器中的局部存储空间

寄存器组是唯一被所有过程共享的资源，我们需要确保当一个过程（调用者）调用另一个过程（被调用者）时，被调用者不会覆盖调用者稍后会使用的寄存器值。

根据惯例，寄存器 rbx、rbp 和 r12-r15 被划分为被调用者保存寄存器。当过程 P 调用过程 Q 时，Q 必须保存这些寄存器的值，保证它们的值在 Q 返回到 P 时与 Q 被调用时是一样的。所有其他的寄存器，除了栈指针 rsp，都分类为调用者保存寄存器。

2.4 数据结构

2.4.1 数组

对于数据类型 T 和整型常数 N，数组声明如下：

```
1 T A[N];
```

它在内存中分配一个 $L \cdot N$ 字节的连续区域，这里 L 是数据类型 T 的大小（单位为字节）。并且，它引入了标识符 A，可以用 A 来作为指向数组开头的指针，这个指针的值就是 x_A 。可以用 $0 \sim N - 1$ 的整数索引来访问该数组元素。数组元素 i 会被存放在地址为 $X_A + L \cdot i$ 的地方。

指针运算

C 语言允许对指针进行运算，而计算出来的值会根据该指针引用的数据类型的大小进行伸缩。也就是说，如果 p 是一个指向类型为 T 的数据的指针， p 的值为 x_p ，那么表达式 $p + i$ 的值为 $x_p + L \cdot i$ ，这里 L 是数据类型 T 的大小。

多维数组

以二维数组为例，声明如下：

```
1 T D[R][C];
```

其元素 $D[i][j]$ 的内存地址为

$$\&D[i][j] = x_D + L \cdot (C \cdot i + j)$$

这里 L 是数据类型 T 的大小， x_D 是数组 D 的起始地址。

定长数组

C 语言编译器能够优化定长多维数组上的操作代码。

```
1 /* Get element a[i][j] */
2 int fix_ele(fix_matrix a, size_t i, size_t j) {
3     return a[i][j];
4 }
```

```
1 ; a in %rdi, i in %rsi, j in %rdx
2 salq    $6, %rsi           ; %rsi = i * 64
3 addq    %rsi, %rdi         ; %rdi = a + 64*i
4 movl    (%rdi,%rdx,4), %eax ; %eax = M[a + 64*i + 4*j]
5 ret
```

变长数组

历史上，C 语言只支持大小在编译时就能确定的多维数组，直到 ISO C99 引入了一种功能，允许数组的维度是表达式，在数组被分配的时候才计算出来。

```
1 /* Get element a[i][j] */
2 int var_ele(size_t n, int a[n][n], size_t i, size_t j) {
3     return a[i][j];
4 }
```

```

1 ; n in %rdi, a in %rsi, i in %rdx, j in %rcx
2 imulq    %rdx, %rdi          ; n*i
3 leaq     (%rsi,%rdi,4), %rax ; a + 4*n*i
4 movl     (%rax,%rcx,4), %eax ; a + 4*n*i + 4*j
5 ret

```

2.4.2 结构

C 语言的 struct 声明创建一个数据类型，将可能不同类型的对象聚合到一个对象中，用名字来引用结构的各个组成部分。结构的所有组成部分都存放在内存中一段连续的区域内，而指向结构的指针就是结构第一个字节的地址。

编译器维护关于每个结构类型的信息，指示每个字段的字节偏移。它以这些偏移作为内存引用指令中的位移，从而产生对结构元素的引用。

```

1 struct s {
2     int i;
3     int j;
4     int a[2];
5     int *p;
6 };

```

这个结构包括 4 个字段：两个 4 字节 int，一个由两个类型为 int 的元素组成的数组和一个 8 字节整型指针，总共是 24 个字节。

数据对齐

许多计算机系统对基本数据类型的合法地址做出了一些限制，要求某种类型对象的地址必须是某个值 K（通常是 2、4 或 8）的倍数。这种对齐限制简化了形成处理器和内存系统之间接口的硬件设计。

对齐原则是任何 K 字节的基本对象的地址必须是 K 的倍数。

2.4.3 联合

C 语言的 union 声明创建一个数据类型，它的所有字段都存放在同一段内存区域中。一个联合的总的大小等于它最大字段的大小。

```

1 union u {
2     int i;
3     float f;
4     char c;
5 };

```

2.4.4 浮点数

在 AVX 浮点体系结构中，允许数据存储在 16 个 YMM 寄存器中，它们的名字为%ymm0-%ymm15。每个 YMM 寄存器都是 256 位（32 字节）。汇编代码用寄存器的 SSE XMM 寄存器名字%xmm0-%xmm15 来引用它们，每个 XMM 寄存器都是对应的 YMM 寄存器的低 128 位（16 字节）。

传送指令

指令	描述
vmovss M ₃₂ , X	传送单精度数
vmovss X, M ₃₂	传送单精度数
vmovsd M ₆₄ , X	传送双精度数
vmovsd X, M ₆₄	传送双精度数
vmovaps X, X	传送对齐的封装好的单精度数
vmovapd X, X	传送对齐的封装好的双精度数

转换指令

指令	描述
vcvttss2si X/M ₃₂ , R ₃₂	用截断的方法把单精度数转换成整数
vcvttsd2si X/M ₆₄ , R ₃₂	用截断的方法把双精度数转换成整数
vcvttsd2si X/M ₆₄ , R ₃₂	用截断的方法把双精度数转换成整数
vcvttss2siq X/M ₃₂ , R ₆₄	用截断的方法把单精度数转换成四字整数
vcvttsd2siq X/M ₆₄ , R ₆₄	用截断的方法把双精度数转换成四字整数

指令	描述
vcvtssi2ss M ₃₂ /R ₃₂ , X, X	把整数转换成单精度数
vcvtssi2sd M ₃₂ /R ₃₂ , X, X	把整数转换成双精度数
vcvtssi2ssq M ₆₄ /R ₆₄ , X, X	把四字整数转换成单精度数
vcvtssi2sdq M ₆₄ /R ₆₄ , X, X	把四字整数转换成双精度数

运算指令

每条指令有一个 (S_1) 或两个 (S_1, S_2) 源操作数，和一个目的操作数 D 。第一个源操作数 S_1 可以是一个 XMM 寄存器或一个内存位置，第二个源操作数和目的操作数都必须是 XMM 寄存器。

指令				效果	描述
vaddss	vaddsd	vaddps	vaddpd	$D \leftarrow S_2 + S_1$	浮点数加
vsubss	vsubsd	vsubps	vsubpd	$D \leftarrow S_2 - S_1$	浮点数减
vmulss	vmulsd	vmulps	vmulpd	$D \leftarrow S_2 \times S_1$	浮点数乘
vdivss	vdivsd	vdivps	vdivpd	$D \leftarrow S_2 / S_1$	浮点数除
vmaxss	vmaxsd	vmaxps	vmaxpd	$D \leftarrow \max(S_2, S_1)$	浮点数最大值
vminss	vminsd	vminps	vminpd	$D \leftarrow \min(S_2, S_1)$	浮点数最小值
sqrtss	sqrtsd	sqrtps	sqrtpd	$D \leftarrow \sqrt{S_1}$	浮点数平方根
vxorss	vxorsd	vxorps	vxorpd	$D \leftarrow S_2 \oplus S_1$	位级异或
vandss	vandsd	vandps	vandpd	$D \leftarrow S_2 \& S_1$	位级与

比较指令

指令	描述
ucomiss S_1, S_2	比较单精度值
ucomisd S_1, S_2	比较双精度值

浮点比较指令会设置三个条件码：零标志位 ZF、进位标志位 CF 和奇偶标志位 PF。对于整数操作，当最近的一次算术或逻辑运算产生的值的最低位字节是偶校验的（即这个字节中有偶数个 1），就会设置 PF 标志位。不过对于浮点比较，当两个操作数中任一个时 NaN 时，会设置该位。

顺序 $S_2 : S_1$	CF	ZF	PF
无序的	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

2.5 机器级程序进阶

2.5.1 类型声明

右左法则：首先从标识符看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。

```

1 int * (* (*a) (int) ) [10];
2 /*

```

3 阅读步骤:

4 1. 从变量名开始: a

5 2. 往右看, 什么也没有, 碰到了')', 因此往左看, 碰到一个'*': 一个指针

6 3. 跳出括号, 碰到了(int): 一个带一个int参数的函数

7 4. 向左看, 发现一个'*': (函数) 返回一个指针

8 5. 跳出括号, 向右看, 碰到[10]: 一个10元素的数组

9 6. 向左看, 发现一个'*': 指针

10 7. 向左看, 发现int: int类型

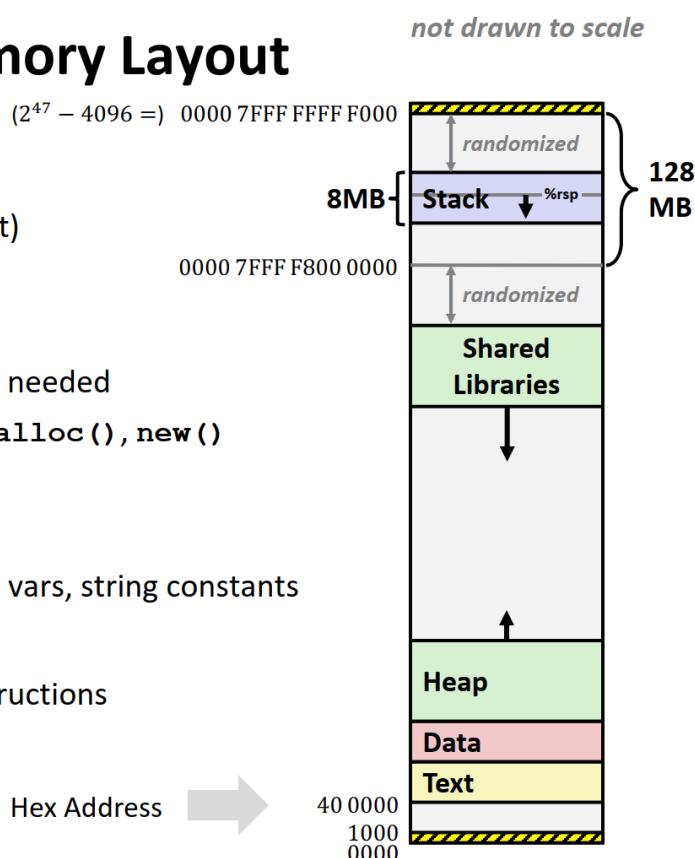
11 总结: a被声明成为一个函数的指针, 该函数返回指向指针数组的指针

12 */

2.5.2 内存布局

x86-64 Linux Memory Layout

- Stack
 - Runtime stack (8MB limit)
 - e.g., local variables
- Heap
 - Dynamically allocated as needed
 - When call `malloc()`, `calloc()`, `new()`
- Data
 - Statically allocated data
 - e.g., global vars, `static` vars, string constants
- Text / Shared Libraries
 - Executable machine instructions
 - Read-only



2.5.3 缓冲溢出攻击

C 对于数组引用不进行任何边界检查, 而且局部变量和状态信息(例如保存的寄存器值和返回地址)都存放在栈中。对越界的数组元素的写操作可能会破坏存储在栈中的状态信息。

一种特别常见的状态破坏称为缓冲区溢出。

```

1  /* Implementation of library function gets() */
2  char *gets(char *s)
3  {
4      int c;
5      char *dest = s;
6      while ((c = getchar()) != '\n' && c != EOF)
7          *dest++ = c;
8      if (c == EOF && dest == s)
9          /* No characters read */
10         return NULL;
11     *dest++ = '\0'; /* Terminate string */
12     return s;
13 }
14 /* Read input line and write it back */
15 void echo()
16 {
17     char buf[8]; /* Way too small! */
18     gets(buf);
19     puts(buf);
20 }
```

```

1 ; void echo()
2 echo:
3 subq    $24, %rsp
4 movq    %rsp, %rdi
5 call    gets
6 movq    %rsp, %rdi
7 call    puts
8 addq    $24, %rsp
9 ret
```

栈粉碎攻击

汇编代码显示，函数调用的参数和存储的返回指针之间的 16 个字节是未被使用的。只要用户输入不超过 7 个字符，gets 返回的字符串（包括结尾的 null）就能够放进为 buf 分配的空间里。不过，长一些的字符串就会导致 gets 覆盖栈上存储的某些信息。随着字符串变长，下面的信息会被破坏：

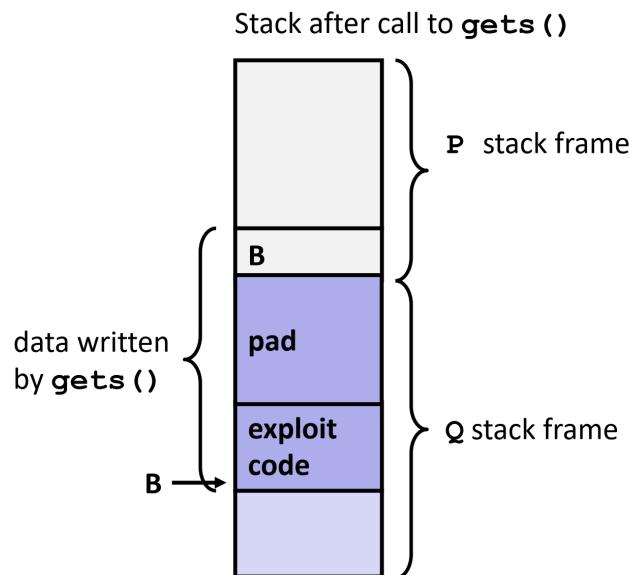
输入的字符数量	附加的被破坏的状态
0 到 7	无
9 到 23	未被使用的栈空间
24 到 31	返回地址
32 及以上	caller 中保存的状态

字符串到 23 个字符之前都没有严重的后果，但是超过以后，返回指针的值以及更多可能的保存状态会被破坏。如果存储的返回地址的值被破坏了，那么 ret 指令会导致程序跳转到一个完全意想不到的位置。

代码注入攻击

缓冲区溢出的一个更加致命的使用就是让程序执行它本来不愿意执行的函数。这是一种最常见的通过计算机网络攻击系统安全的方法。

通常，输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码。另外，还有一些字节会用一个指向攻击代码的指针覆盖返回地址。那么，执行 ret 指令的效果就是跳转到攻击代码。



2.5.4 缓冲溢出保护

代码级保护

通常，使用 `gets` 或其他不检查目标缓冲区大小的函数很容易导致溢出。为了避免这类问题，应优先使用带长度限制的替代函数，例如用 `fgets(buf, sizeof buf, stdin)` 或 POSIX 的 `getline` 代替 `gets`，用 `snprintf` 代替 `sprintf`，用 `strncat/strncpy` 或更推荐的 BSD 扩展 `strlcpy/strlcat` 来进行字符串复制和拼接。

栈随机化

为了在系统中插入攻击代码，攻击者既要插入代码，也要插入指向这段代码的指针，这个指针也是攻击字符串的一部分。

产生这个指针需要知道这个字符串放置的栈地址，栈随机化的思想使得栈的位置在程序每次运行时都有变化。因此，即使许多机器都运行同样的代码，它们的栈地址都是不同的。实现的方式是：程序开始时，在栈上分配一段 0 到 n 字节之间的随机大小的空间。

限制可执行代码区域

在典型的程序中，只有保存编译器产生的代码的那部分内存才是可执行的，其他部分应当被限制为只允许读和写。随着处理器的内存保护引入了“NX”位，将读和执行访问模式分开，栈可以被标记为可读和可写，但是不可执行。由此，攻击代码即使被插入到栈中，也无法被执行。

栈破坏检测

在 C 语言中，没有可靠的方法来防止对数组的越界写。但是，我们能够在发生了越界写的时候，在造成任何有害结果之前，尝试检测到它。

较新版本的 GCC 在产生的代码中加入了一种栈保护者机制。该机制通过在栈帧中插入一个金丝雀值（canary）来检测栈溢出。在函数调用时，GCC 会生成代码来保存当前的金丝雀值，并在函数返回时检查它是否被修改。如果金丝雀值被篡改，程序会立即终止，从而防止潜在的攻击。

```
1 my_function:
2     pushq    %rbp          ; 保存旧的基指针
3     movq    %rsp, %rbp      ; 建立新的栈帧
4     subq    $32, %rsp       ; 为局部变量与保存区留出空间
5     movq    %fs:40, %rax      ; 段寻址读取金丝雀值
6     movq    %rax, -8(%rbp)    ; 将金丝雀值保存到栈帧
7     ; ... 函数的实际主体 ...
8     movq    -8(%rbp), %rax      ; 从栈中取出保存的金丝雀值
9     cmpq    %fs:40, %rax      ; 将其与当前金丝雀值比较
10    jne     .Lstack_chk_fail    ; 若不相等，跳转到失败处理
11    addq    $32, %rsp        ; 清理栈上分配
12    popq    %rbp          ; 恢复旧的基指针
13    ret
14 .Lstack_chk_fail:
15     callq   __stack_chk_fail@plt
```

3 处理器体系架构

3.1 Y86-64 指令集体系结构

我们首先定义一个简单的指令集，作为我们处理器实现的运行示例。因为受 x86-64 指令集的启发，所以我们称我们的指令集为 Y86-64 指令集。与 x86-64 相比，Y86-64 指令集的数据类型、指令和寻址方式都要少一些。它的字节级编码也比较简单，机器代码没有相应的 x86-64 代码紧凑，不过设计它的 CPU 译码逻辑也要简单一些。

- 寄存器：rax、rcx、rdx、rbx、rsp、rbp、rsi、rdi 和 r8 到 r14。
- 条件码：ZF（零标志）、SF（符号标志）、OF（溢出标志）
- 程序计数器：存放目前正在执行指令的地址。
- 状态码：AOK（正常）、HLT（停止）、ADR（地址错误）、INS（无效指令）
- 内存：字节寻址的内存空间，小端法储存。

3.1.1 Y86-64 指令

指令编码

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	rA	rB						
irmovq V, rB	3	0	F	rB			V			
rmmovq rA, D(rB)	4	0	rA	rB			D			
mrmovq D(rB), rA	5	0	rA	rB			D			
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
cmoveq rA, rB	2	fn	rA	rB						
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

整数操作指令	分支指令	传送指令
addq [6 0]	jmp [7 0] jne [7 4]	rrmovq [2 0] cmoveq [2 4]
subq [6 1]	jle [7 1] jge [7 5]	cmovele [2 1] cmovge [2 5]
andq [6 2]	jl [7 2] jg [7 6]	cmoveq [2 2] cmovgt [2 6]
xorq [6 3]	je [7 3]	cmove [2 3]

- x86-64 的 movq 指令分成了 4 个不同的指令:irmovq、rrmovq、mrmovq 和 rmmovq。分别显式地指明源和目的的格式。
- 内存引用方式是简单的基址和偏移量形式。在地址计算中，不支持第二变址寄存器 (second index register) 和任何寄存器值的伸缩 (scaling)。
- 4 个整数操作指令 (OPq)：addq、subq、andq 和 xorq。它们只对寄存器数据进行操作，会设置 3 个条件码 ZF、SF 和 OF。
- 7 个跳转指令 (jXX)：jmp、jle、jl、je、jne、jge 和 jg。它们根据条件码的值来决定是否跳转。
- 有 6 个条件传送指令 (cmovXX)：cmovle、cmovl、cmove、cmovne、cmovge 和 cmovg。它们根据条件码的值来决定是否执行传送操作。
- call 指令将返回地址入栈，然后跳到目的地址。ret 指令从这样的调用中返回。
- pushq 和 popq 指令实现了入栈和出栈操作。
- halt 指令停止指令的执行。
- nop 指令什么也不做，只是简单地前进到下一条指令。

每条指令需要 1 到 10 个字节不等，这取决于需要哪些字段。每条指令的第一个字节表明指令的类型。这个字节分为两个部分，每部分 4 位：高 4 位是代码 (code) 部分，低 4 位是功能 (function) 部分。

寄存器编码

数字	寄存器名字	数字	寄存器名字
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	无寄存器

15 个程序寄存器中每个都有一个相对应的范围在 0 到 OxE 之间的寄存器标识符 (register ID)。当需要指明不应访问任何寄存器时，就用 ID 值 OxF 来表示。

3.1.2 Y86-64 异常

值	名字	含义
1	AOK	正常操作
2	HLT	遇到器执行 halt 指令
3	ADR	遇到非法地址
4	INS	遇到非法指令

Y86-64 中的状态码描述程序执行的总体状态。正常执行时，状态码为 AOK。当程序执行 halt 指令时，状态码变为 HLT。如果程序试图访问无效的内存地址，状态码变为 ADR。如果程序试图执行无效指令，状态码变为 INS。对于 Y86-64，当遇到这些异常的时候，我们就简单地让处理器停止执行指令。在更完整的设计中，处理器通常会调用一个异常处理程序 (exception handler)，这个过程被指定用来处理遇到的某种类型的异常。

3.1.3 Y86-64 程序

```

1 ; Execution begins at address 0
2     .pos 0
3         irmovq stack, %rsp      ; Set up stack pointer
4         call main             ; Execute main program
5         halt                  ; Terminate program
6
7 ; Array of 4 elements
8 array: .align 8
9     .quad 0x000d000d000d
10    .quad 0x00c000c000c0
11    .quad 0xb000b000b00
12    .quad 0xa000a000a000
13
14 main:
15     irmovq array, %rdi
16     irmovq $4, %rsi
17     call sum                 ; sum(array, 4)
18     ret
19
20 ; long sum(long *start, long count)
21 ; start in %rdi, count in %rsi

```

```

22 sum:
23     irmovq $8, %r8          ; Constant 8
24     irmovq $1, %r9          ; Constant 1
25     xorq %rax, %rax        ; sum = 0
26     andq %rsi, %rsi        ; Set CC
27     jmp    test             ; Goto test
28 loop:
29     mrmovq (%rdi), %r10    ; Get *start
30     addq %r10, %rax         ; Add to sum
31     addq %r8, %rdi          ; start++
32 test:
33     subq %r9, %rsi          ; count--. Set CC
34     jne    loop             ; Stop when 0
35     ret                  ; Return
36
37 ; Stack starts here and grows to lower addresses
38 .pos 0x200
39 stack:

```

注意到：

- 由于 Y86-64 是一个简化的指令集，所以相较于 x86-64，Y86-64 的汇编代码会更加复杂。
- 程序从地址 0 处开始。
- 初始化栈指针，指明地址 0x200 处是栈的起始位置，低地址增长。须保证栈不会增长得太大以至覆盖了代码或者其他程序数据。

两个特别的指令

在 Y86-64 和 x86-64 中：

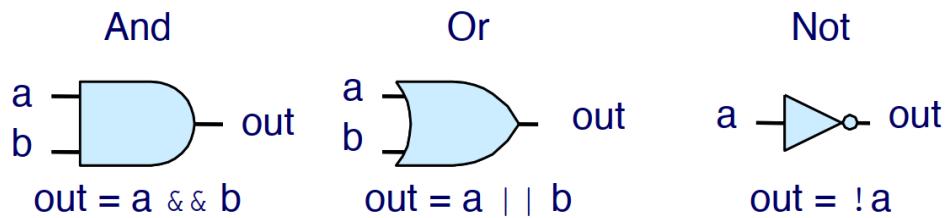
- pushq %rsp 会压入%rsp 的原始值。
- popq %rsp 会将%rsp 置为从内存中读出的值。

3.2 逻辑设计

3.2.1 逻辑门

逻辑门是数字电路的基本计算单元。它们产生的输出，等于它们输入位值的某个布尔函数。辑门总是活动的，一旦一个门的输入变化了，在很短的时间内，输出就会相应

地变化。

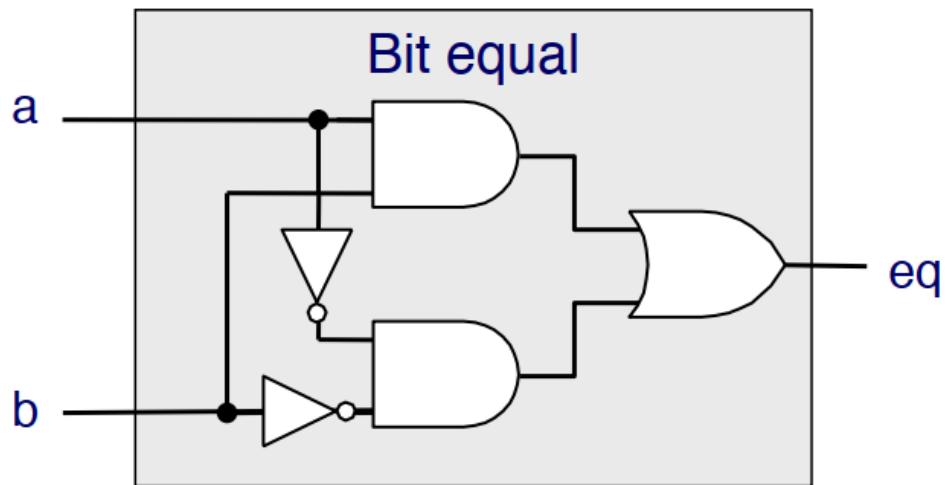


3.2.2 组合电路

将很多的逻辑门组合成一个网，就能构建计算块，称为组合电路。

位相等电路

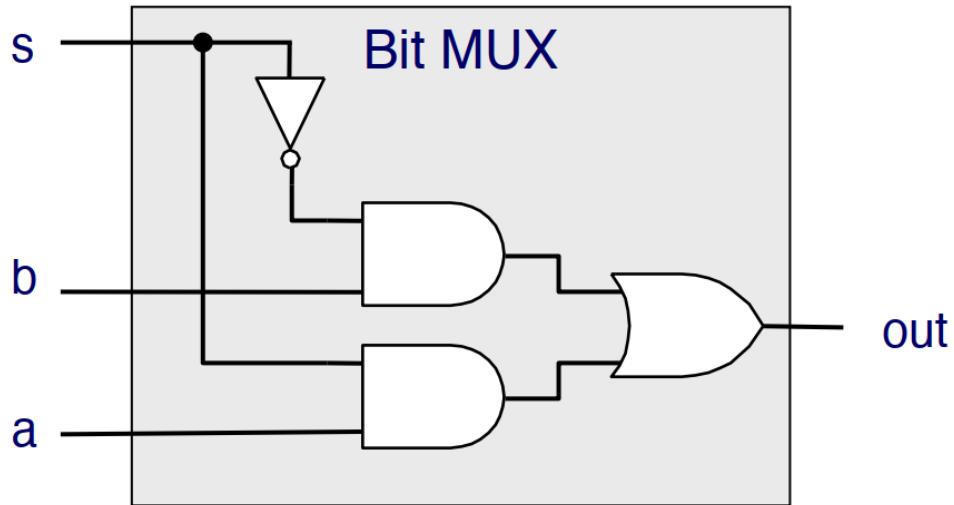
位相等电路有两个输入 a 和 b，有唯一的输出 eq, 当 a 和 b 都是 1 或都是 0 时，输出为 1。



```
1 bool eq = (a && b) || (!a && !b);
```

多路复用器

多路复用器根据输入控制信号的值，从一组不同的数据信号中选出一个。在这个单个位的多路复用器中，两个数据信号是输入位 a 和 b，控制信号是输入位 s。当 s 为 1 时，输出等于 a；而当 s 为 0 时，输出等于 b。



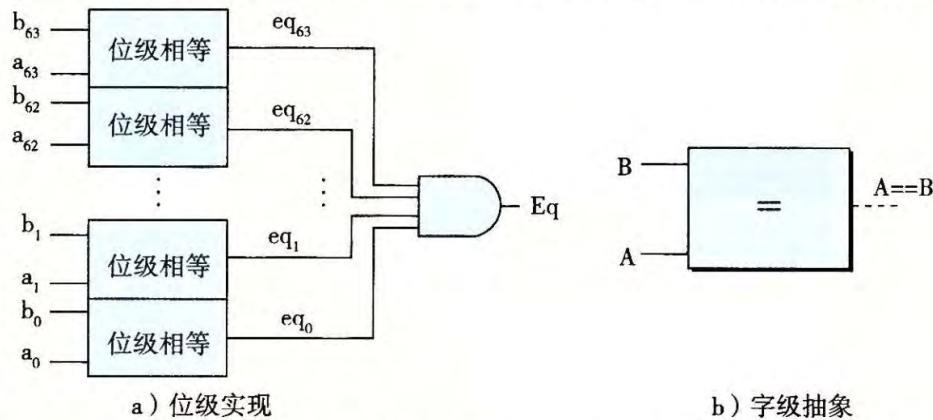
```
1 bool out = (s && a) || (!s && b);
```

3.2.3 字级的组合电路

通过将逻辑门组合成大的网，可以构造出能计算更加复杂函数的组合电路。

字相等电路

字相等电路当且仅当 A 的每一位都和 B 的相应位相等时，输出才为 1。



```
1 bool Eq = (A == B);
```

字级的多路复用器电路

字级的多路复用器电路根据控制输入位 s ，产生一个 64 位的字 Out ，等于两个输入字 A 或者 B 中的一个。处理器中会用到很多种多路复用器，使得我们能根据某些控

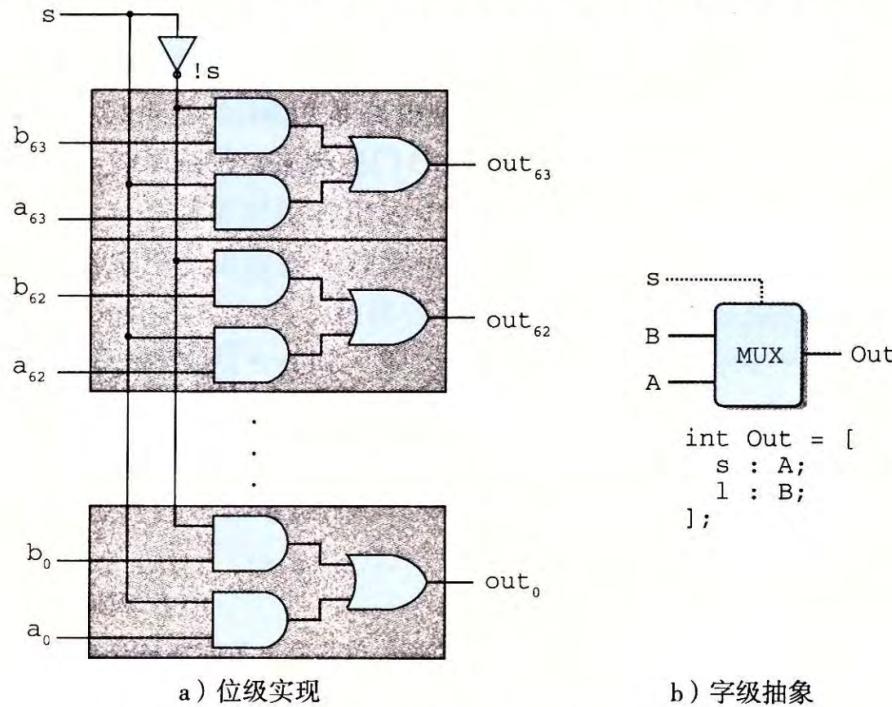
制条件，从许多源中选出一个字。在 HCL 中，多路复用函数是用情况表达式来描述的。情况表达式的通用格式如下：

```

1 [
2     select1 : expr1;
3     select2 : expr2;
4     .
5     .
6     .
7     selectk : exprk;
8 ]

```

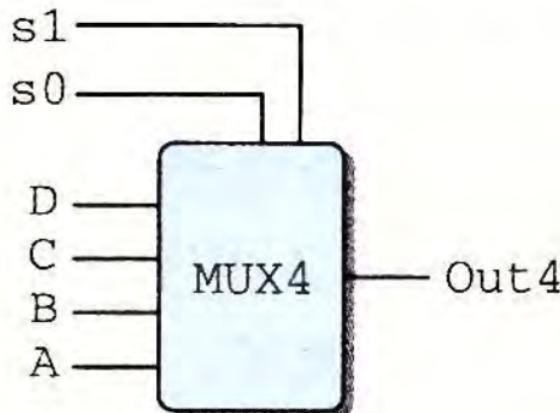
我们不要求不同的选择表达式之间互斥。从逻辑上讲，这些选择表达式是顺序求值的，且第一个求值为 1 的情况会被选中。



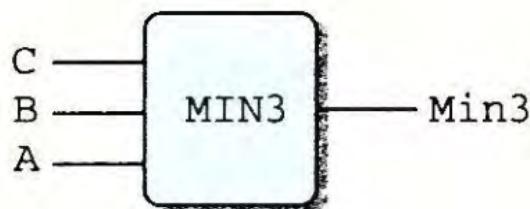
```

1 word Out = [
2     s: A;
3     1: B;
4 ] ;

```



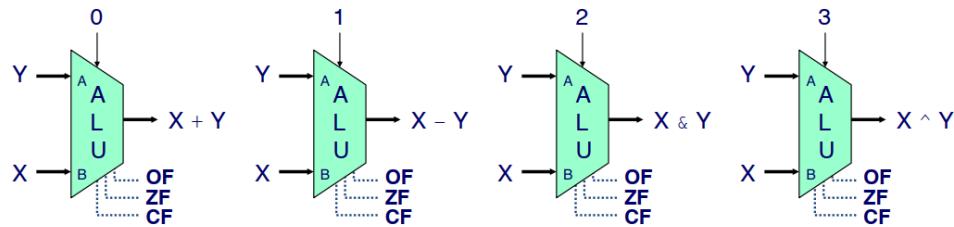
```
1 word Out4 = [
2     !s1 && !s0 : A; # 00
3     !s1         : B; # 01
4     !s0         : C; # 10
5     1           : D; # 11
6 ];
```



```
1 word Min3 = [
2     A <= B && A <= C : A;
3     B <= A && B <= C : B;
4     1                 : C;
5 ];
```

算术逻辑单元

算术逻辑单元 (ALU) 是一种很重要的组合电路。有标号为 A 和 B 的两个数据输入，以及一个控制输入。根据控制输入的设置，电路会对数据输入执行不同的算术或逻辑操作。



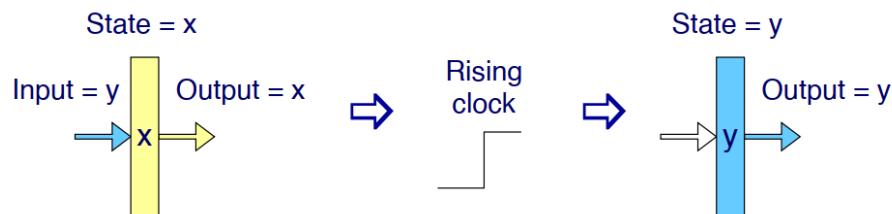
3.2.4 存储器

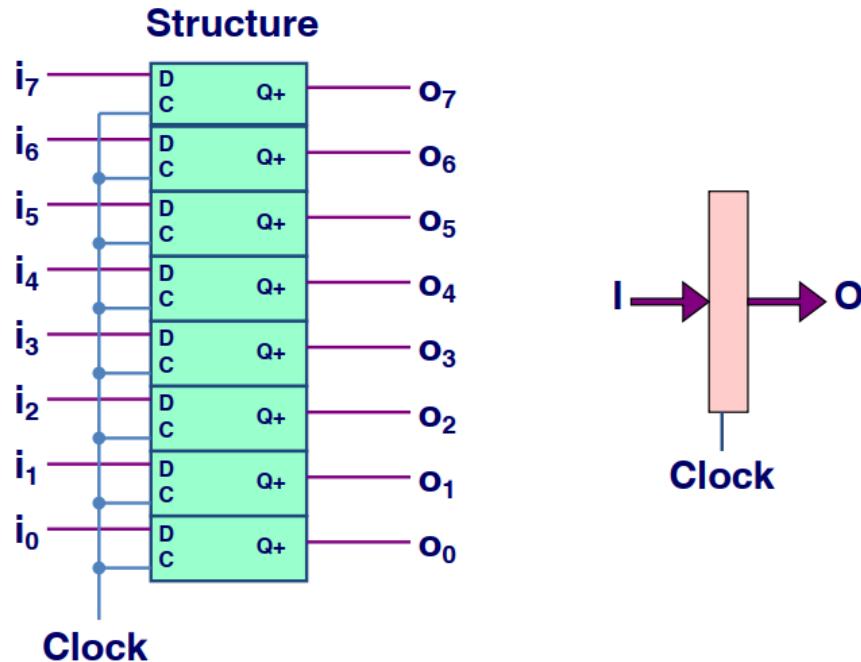
组合电路从本质上讲，不存储任何信息。为了产生时序电路，我们必须引入按位存储信息的设备。存储设备都是由同一个时钟控制的，时钟是一个周期性信号，决定什么时候要把新值加载到设备中。有两类存储器设备：

- 时钟寄存器（简称寄存器）：存储单个位或字。时钟信号控制寄存器加载输入值。
- 随机访问存储器（简称内存）：存储多个字，用地址来选择该读或该写哪个字。
 - 处理器的虚拟内存系统：硬件和操作系统软件结合起来使处理器可以在一个很大的地址空间内访问任意的字。
 - 寄存器文件：用于存储处理器操作数的小型、快速的存储单元集，寄存器标识符作为地址。

寄存器

大多数时候，寄存器都保持在稳定状态（用 x 表示），产生的输出等于它的当前状态。信号沿着寄存器前面的组合逻辑传播，这时，产生了一个新的寄存器输入（用 y 表示），但只要时钟是低电位的，寄存器的输出就仍然保持不变。当时钟变成高电位的时候，输入信号就加载到寄存器中，成为下一个状态 y ，直到下一个时钟上升沿，这个状态就一直是寄存器的新输出。



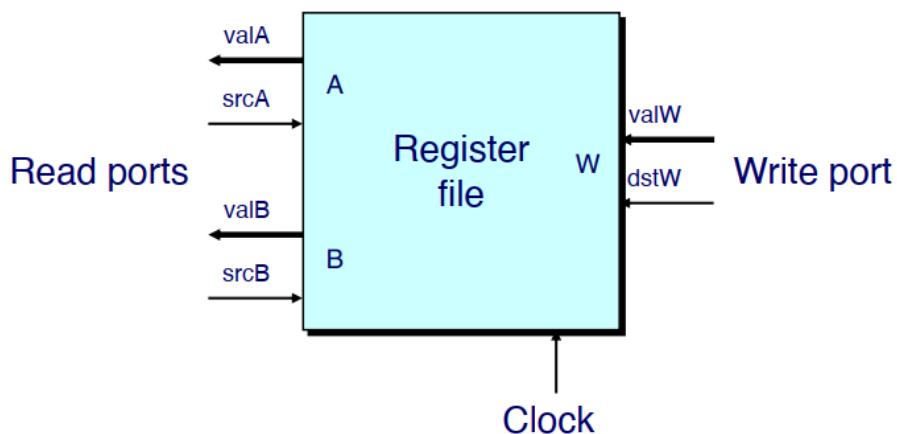


寄存器文件

寄存器文件有两个读端口 (A 和 B)，还有一个写端口 (W)。这样一个多端口随机访问存储器允许同时进行多个读和写操作。

当 srcA 或 srcB 被设成某个寄存器 ID 时，在一段延迟之后，存储在相应程序寄存器的值就会出现在 valA 或 valB 上。

向寄存器文件写入字是由时钟信号控制的，控制方式类似于将值加载到时钟寄存器。每次时钟上升时，输入 valW 上的值会被写入输入 dstW 上的寄存器 ID 指示的程序寄存器。当 dstW 设为特殊的 ID 值 0xF 时，不会写任何程序寄存器。

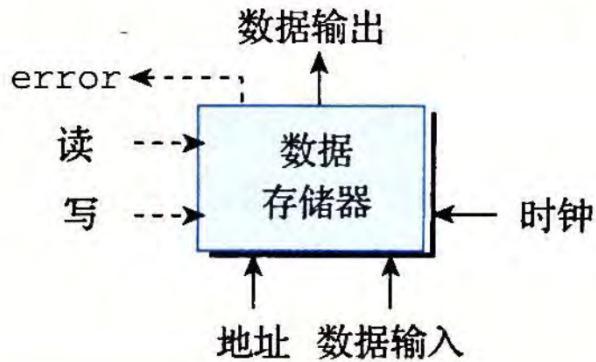


内存

内存有一个地址输入，一个写的数据输入，以及一个读的数据输出。

同寄存器文件一样，从内存中读的操作方式类似于组合逻辑：如果我们在输入 address 上提供一个地址，并将 write 控制信号设置为 0，那么在经过一些延迟之后，存储在那个地址上的值会出现在输出 data 上。如果地址超出了范围，error 信号会设置为 1，否则就设置为 0。

写内存是由时钟控制的：我们将 address 设置为期望的地址，将 data in 设置为期望的值，而 write 设置为 1。然后当我们控制时钟时，只要地址是合法的，就会更新内存中指定的位置。

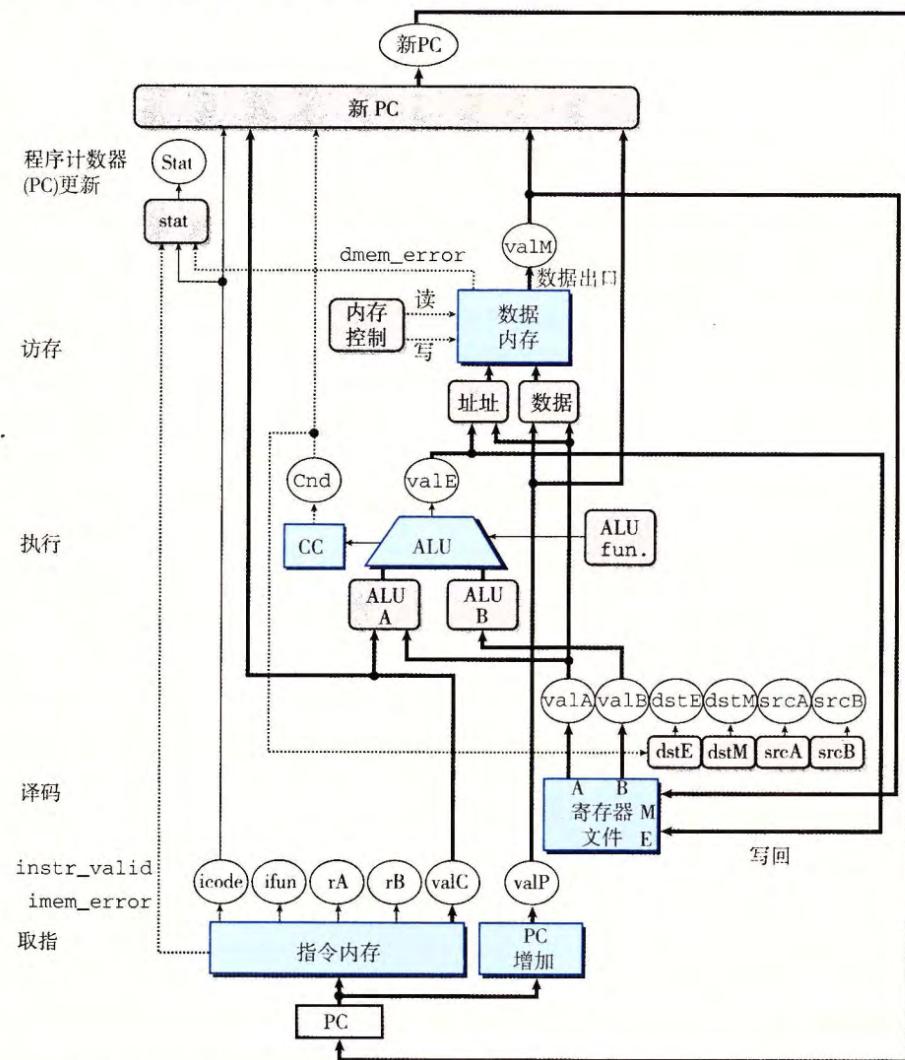
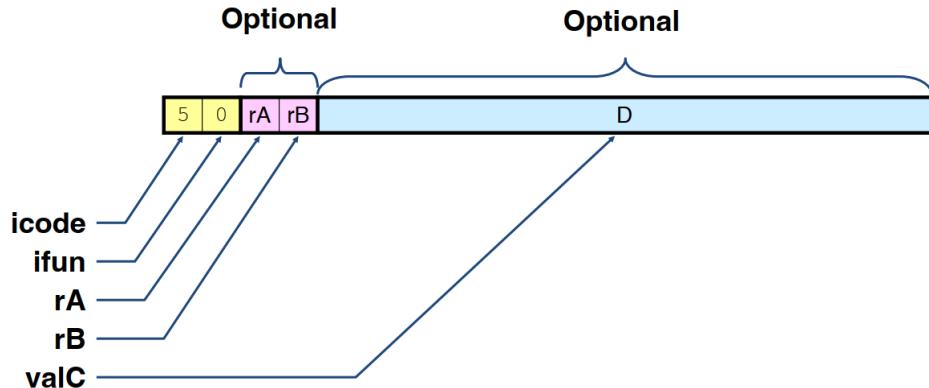


3.3 Y86-64 的顺序实现

通常，处理一条指令包括很多操作。将它们组织成某个特殊的阶段序列，即使指令的动作差异很大，但所有的指令都遵循统一的序列。创建这样一个框架，我们就能够设计一个充分利用硬件的处理器。

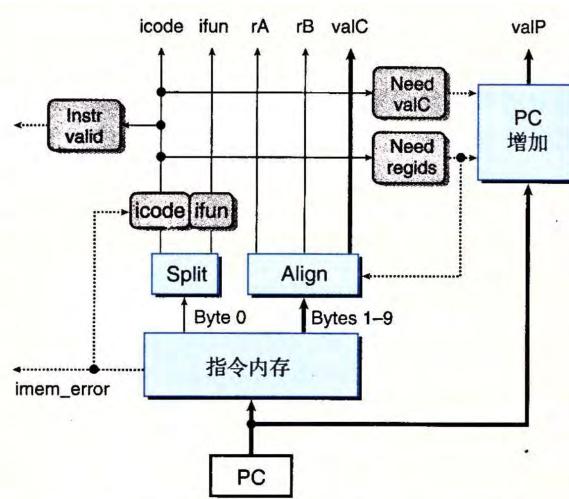
- 取指 (fetch): 将程序计数器寄存器作为地址，读取指令的字节。PC 增加器 (PC incrementer) 计算 valP，即增加了的程序计数器。
- 译码 (decode): 寄存器文件有两个读端口 A 和 B，从这两个端口同时读寄存器值 valA 和 valB。
- 执行 (execute): 执行阶段会根据指令的类型，将 ALU 用于不同的目的。对整数操作，它要执行指令所指定的运算。对其他指令，它会作为一个加法器来计算增加或减少栈指针，或者计算有效地址，或者只是简单地加 0，将一个输入传递到输出。条件码寄存器 (CC) 有三个条件码位。ALU 负责计算条件码的新值。当执行条件传送指令时，根据条件码和传送条件来计算决定是否更新目标寄存器。同样，当执行一条跳转指令时，会根据条件码和跳转类型来计算分支信号 Cnd。
- 访存 (memory): 在执行访存操作时，数据内存读出或写入一个内存字。
- 写回 (write back): 寄存器文件有两个写端口。端口 E 用来写 ALU 计算出来的值，而端口 M 用来写从数据内存中读出的值。

- 更新 PC (PC update): 程序计数器的新值选择自: 下一条指令的地址 valP ; 调用指令或跳转指令指定的目标地址 valE ; 从内存读取的返回地址 valM 。



OPq rA, rB		rmmovq rA, D(rB)	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valP $\leftarrow PC+2$	Read instruction byte Read register byte Compute next PC Read operand A Read operand B Perform ALU operation Set condition code register	Fetch icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valC $\leftarrow M_8[PC+2]$ valP $\leftarrow PC+10$
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	Decode valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	Read displacement D Compute next PC Read operand A Read operand B
Execute	valE $\leftarrow valB \text{ OP } valA$ Set CC	Execute valE $\leftarrow valB + valC$	Compute effective address
Memory		Memory $M_8[valE] \leftarrow valA$	Write value to memory
Write back	R[rB] $\leftarrow valE$	Write back	
PC update	PC $\leftarrow valP$	PC update PC $\leftarrow valP$	Update PC
cmoveXX rA, rB		popq rA	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valP $\leftarrow PC+2$	Read instruction byte Read register byte Compute next PC Read operand A	Fetch icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow 0$	Pass valA through ALU (Disable register update)	valP $\leftarrow PC+2$
Execute	valE $\leftarrow valB + valA$ If ! Cond(CC,ifun) rB $\leftarrow 0xF$	Execute valE $\leftarrow valB + 8$	Read stack pointer Read stack pointer Increment stack pointer
Memory		Memory $valM \leftarrow M_8[valA]$	Read from stack
Write back	R[rB] $\leftarrow valE$	Write back R[%rsp] $\leftarrow valE$	Update stack pointer
PC update	PC $\leftarrow valP$	PC update R[rA] $\leftarrow valM$	Write back result
		PC update PC $\leftarrow valP$	Update PC
jXX Dest		call Dest	
Fetch	icode:ifun $\leftarrow M_1[PC]$ valC $\leftarrow M_8[PC+1]$ valP $\leftarrow PC+9$	Read instruction byte Read destination address Fall through address	Read instruction byte
Decode		Take branch? Update PC	Read destination address Compute return point
Execute	Cnd $\leftarrow Cond(CC,ifun)$		Read stack pointer Decrement stack pointer
Memory			Write return value on stack
Write back			Update stack pointer
PC update	PC $\leftarrow Cnd ? valC : valP$		Set PC to destination
ret			
Fetch	icode:ifun $\leftarrow M_1[PC]$	Read instruction byte	
Decode	valA $\leftarrow R[%rsp]$ valB $\leftarrow R[%rsp]$	Read operand stack pointer Read operand stack pointer Increment stack pointer	
Execute	valE $\leftarrow valB + 8$	Read return address Update stack pointer	
Memory	valM $\leftarrow M_8[valA]$		
Write back	R[%rsp] $\leftarrow valE$		
PC update	PC $\leftarrow valM$	Set PC to return address	

3.3.1 取指阶段



以 PC 作为第一个字节（字节 0）的地址，这个单元一次从内存读出 10 个字节。第一个字节被解释成指令字节，标号为”Split” 的单元）分为两个 4 位的数。然后，标号为”icode” 和”ifun” 的控制逻辑块计算指令和功能码，或者使之等于从内存读出的值，或者当指令地址不合法时（由信号 imem_error 指明），使这些值对应于 nop 指令。根据使之等于从内存读出的值，或者当指令地址不合法时（由信号 imem_error 指明），使这些值对应于 nop 指令。根据 icode 的值，我们可以计算三个一位的信号：

- need regids : 当指令需要寄存器 ID 字节时，该信号为 1 。
- need valC : 当指令需要常数字段时，该信号为 1 。
- instr valid : 当指令是有效指令时，该信号为 1 。

```

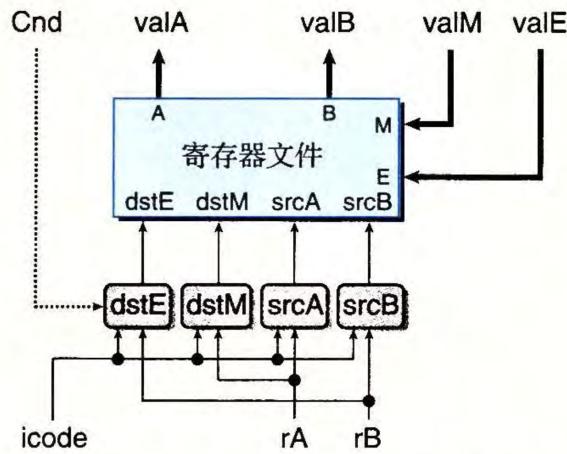
1 bool need_regids =
2     icode in { IRRMDVQ, IOPQ, IPUSHQ, IPOPQ,
3                 IIRMDVQ, IRMMOVQ, IMRMDVQ };

```

从指令内存中读出的剩下 9 个字节是寄存器指示符字节和常数字的组合编码。标号为”Align” 的硬件单元会处理这些字节，将它们放入寄存器字段和常数字中。当被计算出的信号 need regids 为 1 时，字节 1 被分开放入寄存器指示符 rA 和 rB 中。否则，这两个字段会被设为 OxF，表明这条指令没有指明寄存器。根据信号 need regids 的值，要么根据字节 1-8 来产生 valC，要么根据字节 2-9 来产生。

PC 增加器硬件单元根据当前的 PC 以及两个信号 need regids 和 need valC 的值，产生信号 valP 。

3.3.2 译码阶段



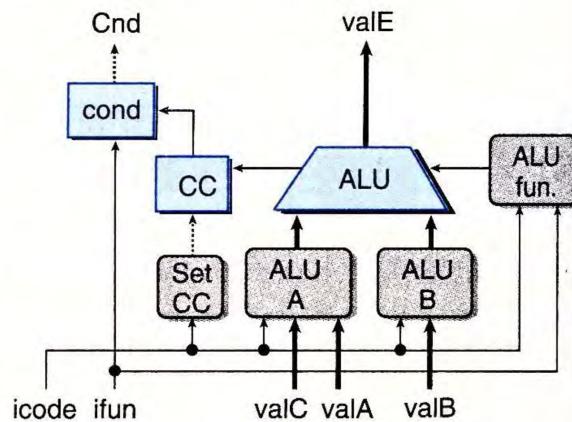
根据指令代码 icode 以及寄存器指示值 rA 和 rB , 可能还会根据执行阶段计算出的 Cnd 条件信号 , 产生出四个不同的寄存器文件的寄存器 ID 。两个读端口的地址输入为 srcA 和 srcB , 两个写端口的地址输入为 dstE 和 dstM 。

```

1 word srcA = [
2     icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
3     icode in { IPOPQ, IRET } : RRSP;
4     1 : RNONE; # Don't need register
5 ];
6 # WARNING: Conditional move not implemented correctly here
7 word dstE = [
8     icode in { IRRMDVQ } : rB;
9     icode in { IIRMOVQ, IOPQ } : rB;
10    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
11    1 : RNONE; # Don't write any register
12 ];

```

3.3.3 执行阶段



ALU 单元根据 alufun 信号的设置，对输入 aluA 和 aluB 执行 ADD 、SUBTRACT 、AND 或 EXCLUSIVEOR 运算，输出为 valE 信号。

```

1 word aluA = [
2     icode in { IRRMOVQ, IOPQ } : valA;
3     icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
4     icode in { ICALL, IPUSHQ } : -8;
5     icode in { IRET, IPOPQ } : 8;
6     # Other instructions don't need ALU
7 ];
8 word alufun = [
9     icode == IDPQ : ifun;
10    1 : ALUADD
11];

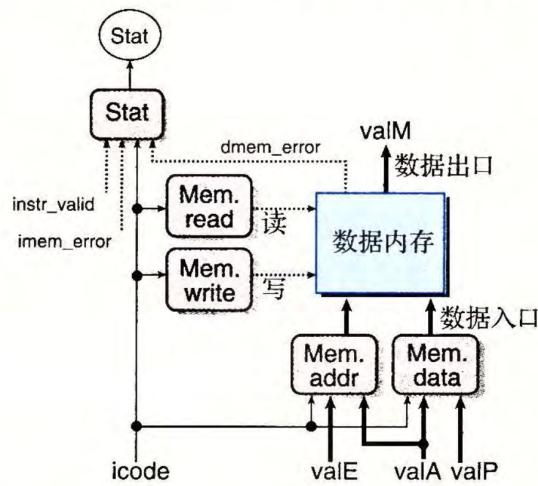
```

执行阶段还包括条件码寄存器。每次运行时，ALU 都会产生三个与条件码相关的信号零、符号和溢出。不过，我们只希望在执行 OPq 指令时才设置条件码。因此产生了一个信号 set cc 来控制是否该更新条件码寄存器。

```
1 bool set_cc = icode in { IDPQ };
```

标号为”cond”的硬件单元会根据条件码和功能码来确定是否进行条件分支或者条件数据传送。它产生信号 Cnd，用于设置条件传送的 dstE，也用在条件分支的下一个 PC 逻辑中。

3.3.4 访存阶段



访存阶段的任务就是读或者写程序数据。两个控制块产生内存地址和内存输入数据的值。另外两个块产生表明应该执行读操作还是写操作的控制信号。当执行读操作时，数据内存产生值 valM。

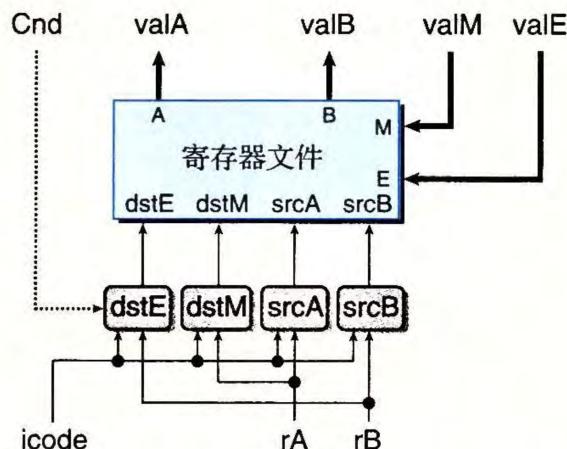
```

1 word mem_addr = (
2     icode in { IRMMOVQ , IPUSHQ, ICALL, IMRMOVQ } : valE;
3     icode in { IPOPQ, IRET } : valA;
4     # Other instructions don't need address
5 );
6 bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };

```

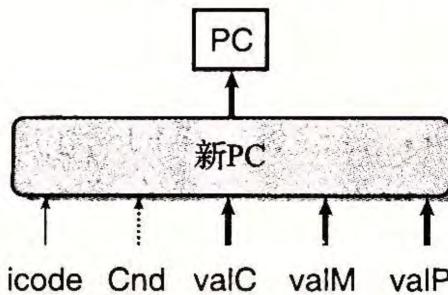
访存阶段最后的功能是根据取值阶段产生的 icode、imem_error、instr valid 值以及数据内存产生的 dmem error 信号，从指令执行的结果来计算状态码 Stat。

3.3.5 写回阶段



写回阶段的任务是将执行阶段产生的值 valE 或者访存阶段产生的值 valM 写入寄存器文件。

3.3.6 更新 PC 阶段



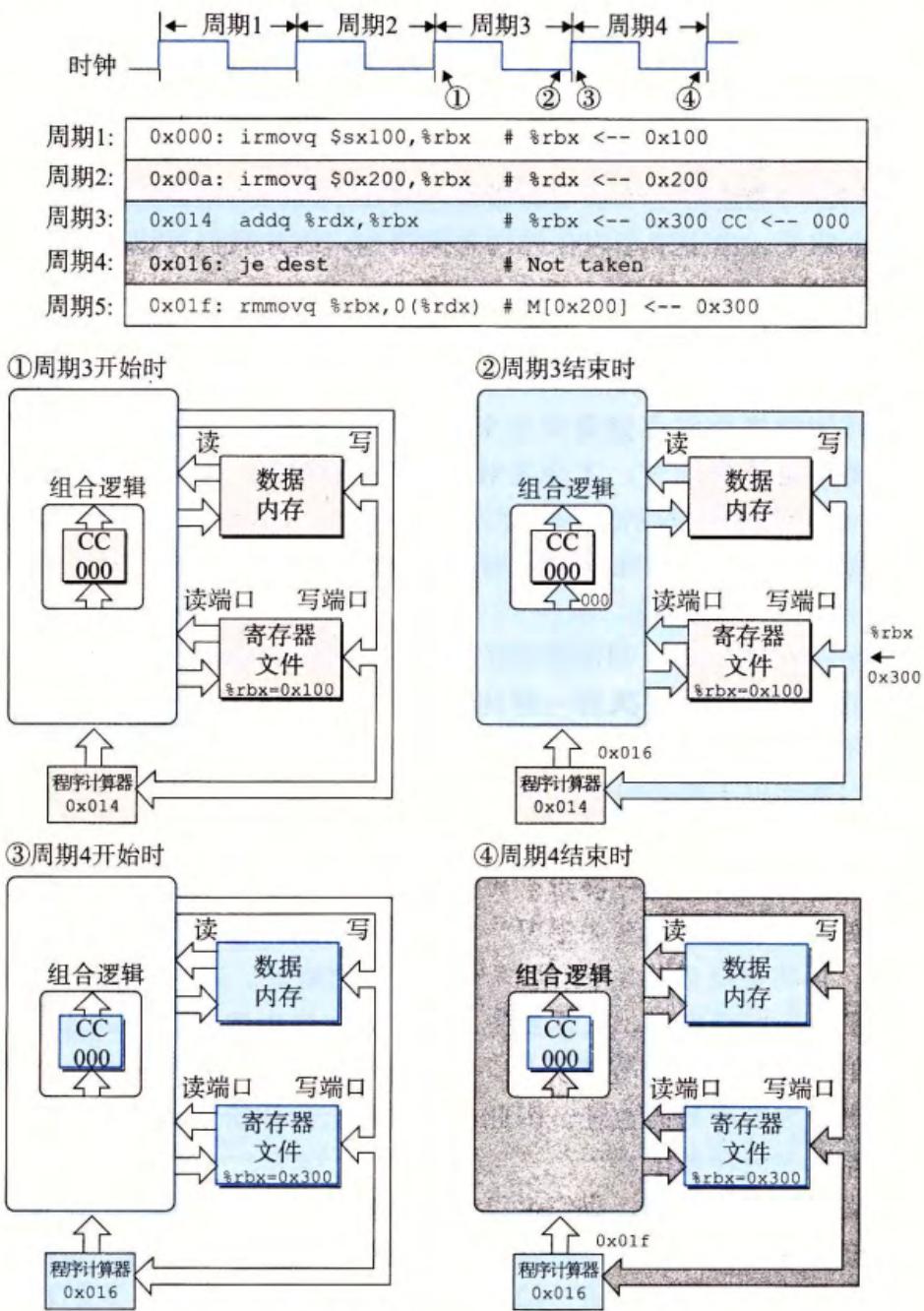
SEQ 中最后一个阶段会产生程序计数器的新值。依据指令的类型和是否要选择分支，新的 PC 可能是 valC 、 valM 或 valP 。

```

1 word new_pc = [
2     # Call. Use instruction constant
3     icode == ICALL : valC;
4     # Taken branch. Use instruction constant
5     icode == IJXX && Cnd : valC;
6     # Completion of RET instruction. Use value from stack
7     icode == IRET : valM;
8     # Default: Use incremented PC
9     1 : valP;
10 ];
  
```

3.3.7 SEQ 的时序

SEQ 的实现包括组合逻辑和两种存储器设备：时钟寄存器（程序计数器和条件码寄存器），随机访问存储器（寄存器文件、指令内存和数据内存）。组合逻辑不需要任何时序或控制，只要输入变化了，值就通过逻辑门网络传播。存储器设备通过一个时钟信号来控制，它触发将新值装载到寄存器以及将值写到随机访问存储器中。



3.4 Y86-64 的流水线实现

3.4.1 PC 选择和取指阶段

分支预测

控制冒险

3.4.2 译码和写回阶段

数据冒险

3.4.3 执行阶段

3.4.4 访存阶段

3.4.5 控制逻辑

控制机制

触发条件

处理

控制条件的组合

4 存储器层次结构

4.1 存储技术

4.1.1 随机访问存储器

SRAM 和 DRAM

- **静态随机存取存储器 (SRAM, Static Random Access Memory):** SRAM 将每个位存储在一个双稳态的 (bistable) 存储器单元里。每个单元由一个六晶体管电路实现。
- **动态随机存取存储器 (DRAM, Dynamic Random Access Memory):** DRAM 的每个单元由一个电容和一个访问晶体管组成。它对干扰非常敏感，当电容的电压被扰乱之后，它就永远不会恢复了。

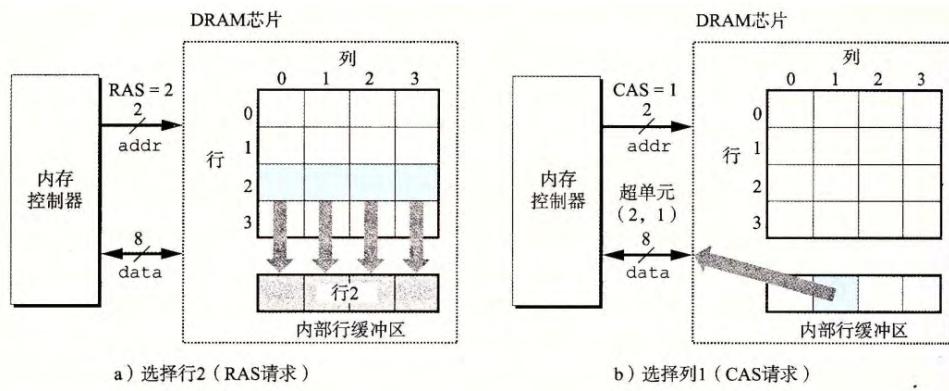
只要有供电，SRAM 就会保持不变。与 DRAM 不同，它不需要刷新。SRAM 的存取比 DRAM 快得多。SRAM 对诸如光和电噪声这样的干扰不敏感。代价是 SRAM 单元比 DRAM 单元使用更多的晶体管，因而密度低，而且更贵，功耗更大。

	每位晶体管数	相对访问时间	持续的？	敏感的？	相对花费	应用
SRAM	6	1×	是	否	1000×	高速缓存存储器
DRAM	1	10×	否	是	1×	主存，帧缓冲区

传统的 DRAM

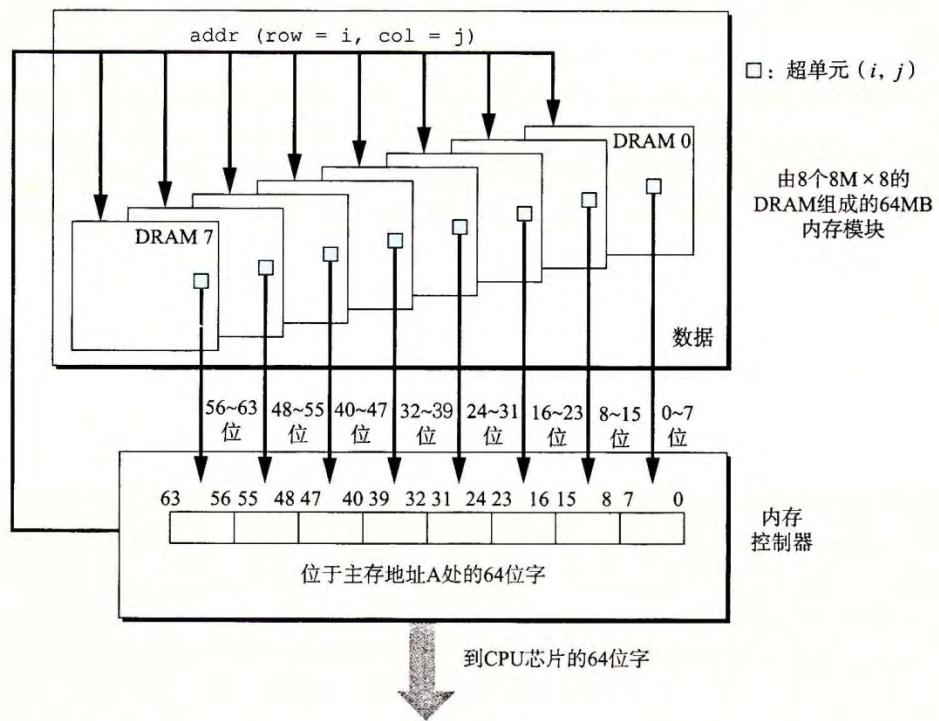
DRAM 芯片中的单元（位）被分成 d 个超单元 (supercell)，每个超单元都由 w 个 DRAM 单元组成。一个 $d \times w$ 的 DRAM 总共存储了 $d \times w$ 位信息。超单元被组织成一个 r 行 c 列的长方形阵列，这里 $r \times c = d$ 。每个超单元有形如 (i,j) 的地址，这里 i 表示行，而 j 表示列。

每个 DRAM 芯片被连接到某个称为内存控制器 (memory controller) 的电路，这个电路可以一次传送 w 位到每个 DRAM 芯片或一次从每个 DRAM 芯片传出 w 位。为了读出超单元 (i,j) 的内容，内存控制器将行地址 i 发送到 DRAM，然后是列地址 j 。DRAM 把超单元 (i,j) 的内容发回给控制器作为响应。行地址请求称为 RAS (Row Access Strobe, 行访问选通脉冲)，列地址请求称为 CAS (Column Access Strobe, 列访问选通脉冲)。注意，RAS 和 CAS 请求共享相同的 DRAM 地址引脚。



内存模块

DRAM 芯片封装在内存模块 (memory module) 中，插入到主板的扩展插槽上。要取出内存地址 A 处的一个字，内存控制器将 A 转换成一个超单元地址 (i, j) 并将它发送到内存模块，然后内存模块再将 i 和 j 广播到每个 DRAM 芯片。作为响应，每个 DRAM 输出它的 (i, j) 超单元的 8 位内容。模块中的电路收集这些输出，并把它们合并成一个 64 位字，再返回给内存控制器。



增强的 DRAM

- 快页模式 DRAM (Fast Page Mode DRAM, FPM DRAM)
- 拓展数据输出 DRAM (Extended Data Out DRAM, EDO DRAM)
- 同步 DRAM (Synchronous DRAM, SDRAM)

- 双数据速率同步 DRAM (Double Data Rate Synchronous DRAM, DDR SDRAM)
- 视频 RAM (Video RAM, VRAM)

非易失性存储器

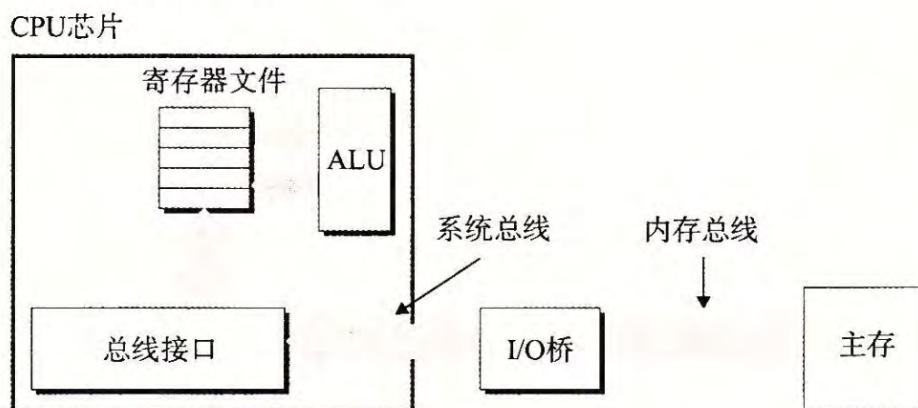
如果断电，DRAM 和 SRAM 会丢失它们的信息，从这个意义上说，它们是易失的 (volatile)。非易失性存储器 (nonvolatile memory) 即使是在关电后，仍然保存着它们的信息。

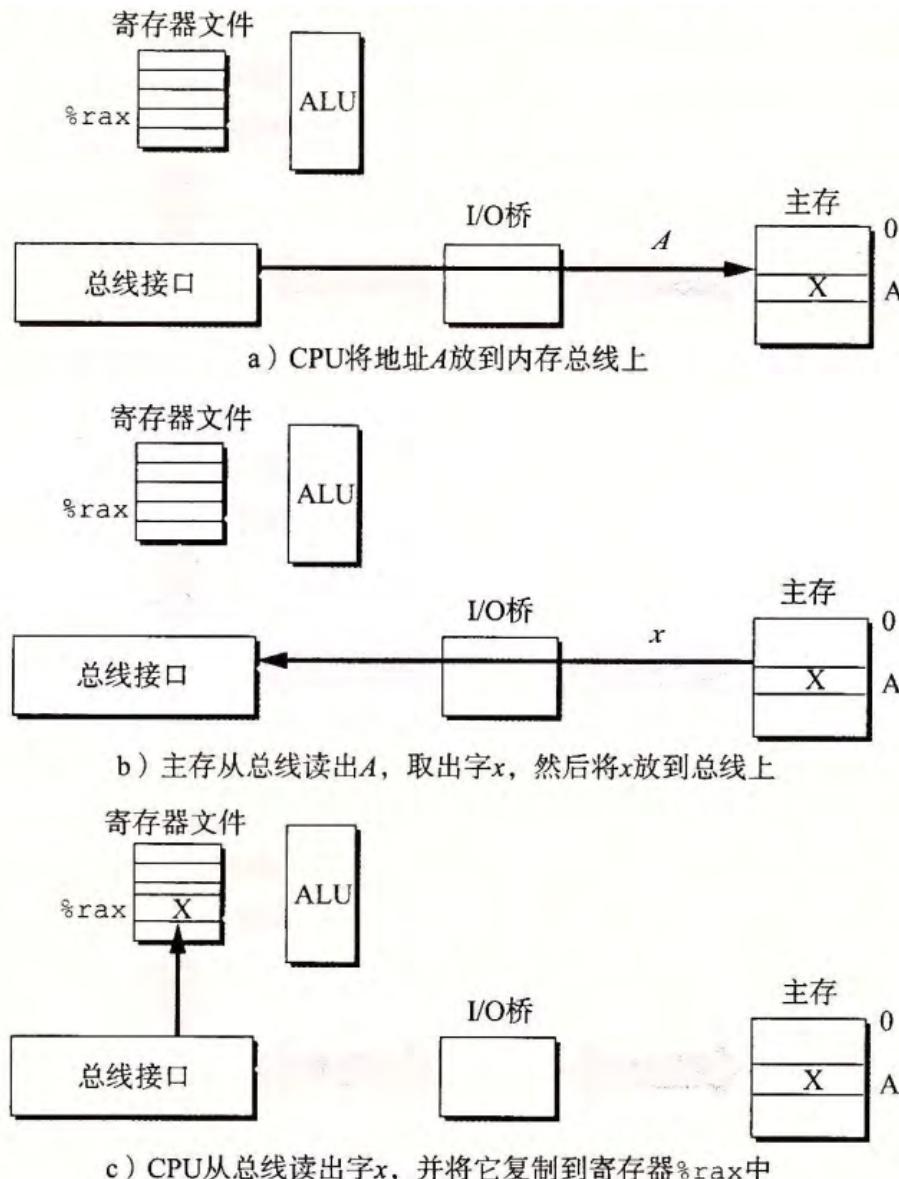
- 只读存储器 (Read-Only Memory, ROM)
- 可编程只读存储器 (Programmable Read-Only Memory, PROM)
- 可擦除可编程只读存储器 (Erasable Programmable Read-Only Memory, EPROM)
- 电可擦除可编程只读存储器 (Electrically Erasable Programmable Read-Only Memory, EEPROM)
- 闪存 (Flash Memory)

访问主存

数据流通过称为总线 (bus) 的共享电子电路在处理器和 DRAM 主存之间来来回回。每次 CPU 和主存之间的数据传送都是通过一系列步骤来完成的，这些步骤称为总线事务 (bus transaction)。读事务 (read transaction) 从主存传送数据到 CPU。写事务 (write transaction) 从 CPU 传送数据到主存。

下图的主要部件是 CPU 芯片、I/O 桥接器 (I/O bridge) ，以及组成主存的 DRAM 内存模块。这些部件由一对总线连接起来，其中一条总线是系统总线 (system bus)，它连接 CPU 和 I/O 桥接器，另一条总线是内存总线 (memory bus)，它连接 I/O 桥接器和主存。I/O 桥接器将系统总线的电子信号翻译成内存总线的电子信号。I/O 桥也将系统总线和内存总线连接到 I/O 总线，像磁盘和图形卡这样的 I/O 设备共享 I/O 总线。





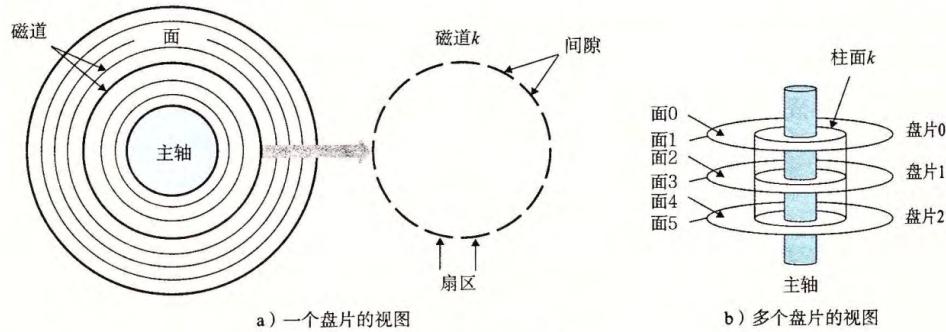
4.1.2 磁盘存储

磁盘构造

磁盘是由盘片 (platter) 构成的。每个盘片有两面，称为表面 (surface)，表面覆盖着磁性记录材料。盘片中央有一个可以旋转的主轴 (spindle)，它使得盘片以固定的旋转速率 (rotational rate) 旋转，通常是每分钟几千转 (revolutions per minute, RPM)。

每个表面是由一组称为磁道 (track) 的同心圆组成的。每个磁道被划分为一组扇区 (sector)。扇区之间由一些间隙 (gap) 分隔开，这些间隙中不存储数据位，间隙用于存放标识扇区的格式化信息。

磁盘是由一个或多个叠放在一起的盘片组成的，它们被封装在一个密封的包装里，整个装置通常被称为磁盘驱动器 (disk drive)，我们通常简称为磁盘 (disk)。



磁盘容量

- 记录密度 (recording density): 每英寸磁道上的位数。
- 轨道密度 (track density): 每英寸径向轨道的数量。
- 面密度 (areal density): 每平方英寸上的位数，记录密度和轨道密度的乘积。

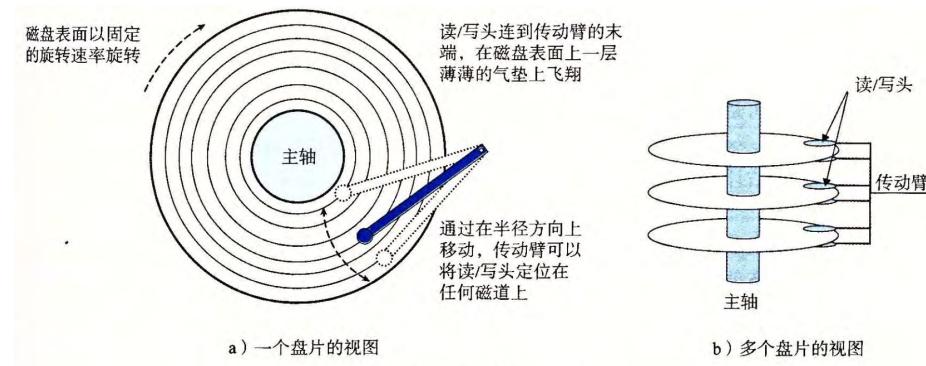
磁盘容量 = 字节数/扇区 × 平均扇区数/磁道 × 磁道数/表面 × 表面数/盘片 × 盘片数/磁盘

一千兆字节有多大

对于与 DRAM 和 SRAM 容量相关的计量单位，通常 $K = 2^{10}$, $M = 2^{20}$, $G = 2^{30}$ 而 $T = 2^{40}$ 。对于与像磁盘和网络这样的 I/O 设备容量相关的计量单位，通常 $K = 10^3$, $M = 10^6$, $G = 10^9$, 而 $T = 10^{12}$ 。

磁盘操作

磁盘用读写头 (read/write head) 来读写存储在磁性表面的位，而读写头连接到一个传动臂 (actuator arm) 一端，通过沿着半径轴前后移动这个传动臂，驱动器可以将读写头定位在盘面上的任何磁道上。这样的机械运动称为寻道 (seek)。一旦读写头定位到了期望的磁道上，那么当磁道上的每个位通过它的下面时，读写头可以感知到这个位的值 (读该位)，也可以修改这个位的值 (写该位)。有多个盘片的磁盘针对每个盘面都有一个独立的读写头。读写头垂直排列，一致行动。在任何时刻，所有的读写头都位于同一个柱面上。



对扇区的访问时间 (access time) 有三个主要的部分：

- 寻道时间 (seek time): 将读/写头移动到正确磁道所需的时间。
- 延迟时间 (latency time): 等待所需扇区旋转
- 传输时间 (transfer time): 读写扇区所需的时间。

因为寻道时间和旋转延迟大致相等，所以将寻道时间乘 2 是估计磁盘访问时间的简单而合理的方法。

磁盘逻辑块

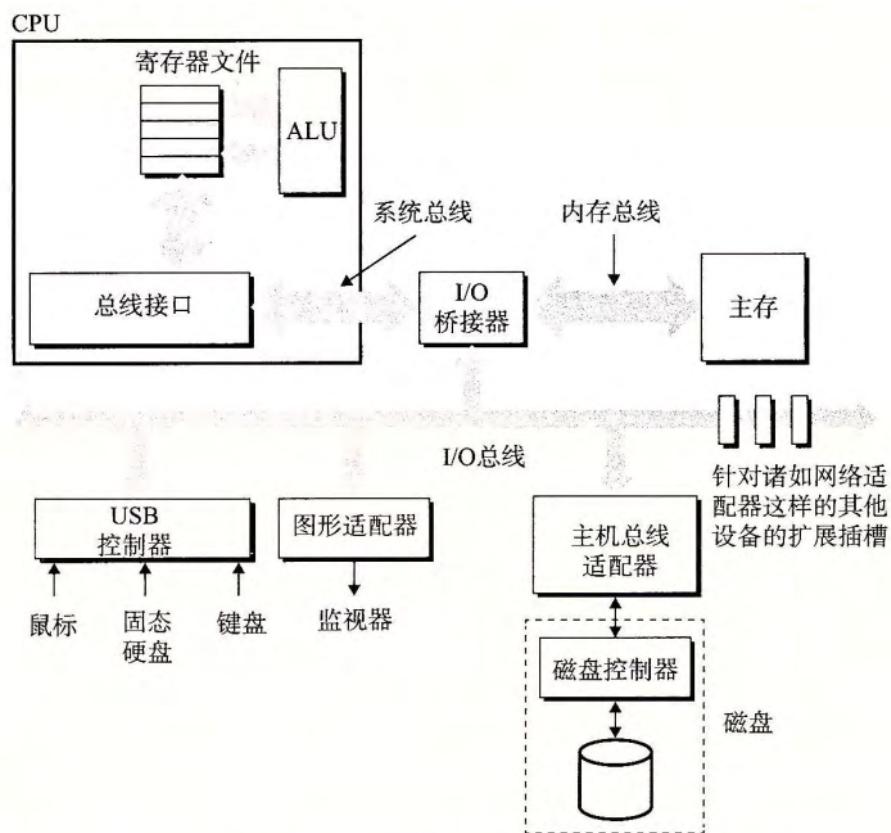
磁盘封装中有一个小的硬件/固件设备，称为磁盘控制器，维护着逻辑块号和实际（物理）磁盘扇区之间的映射关系。

连接 I/O 设备

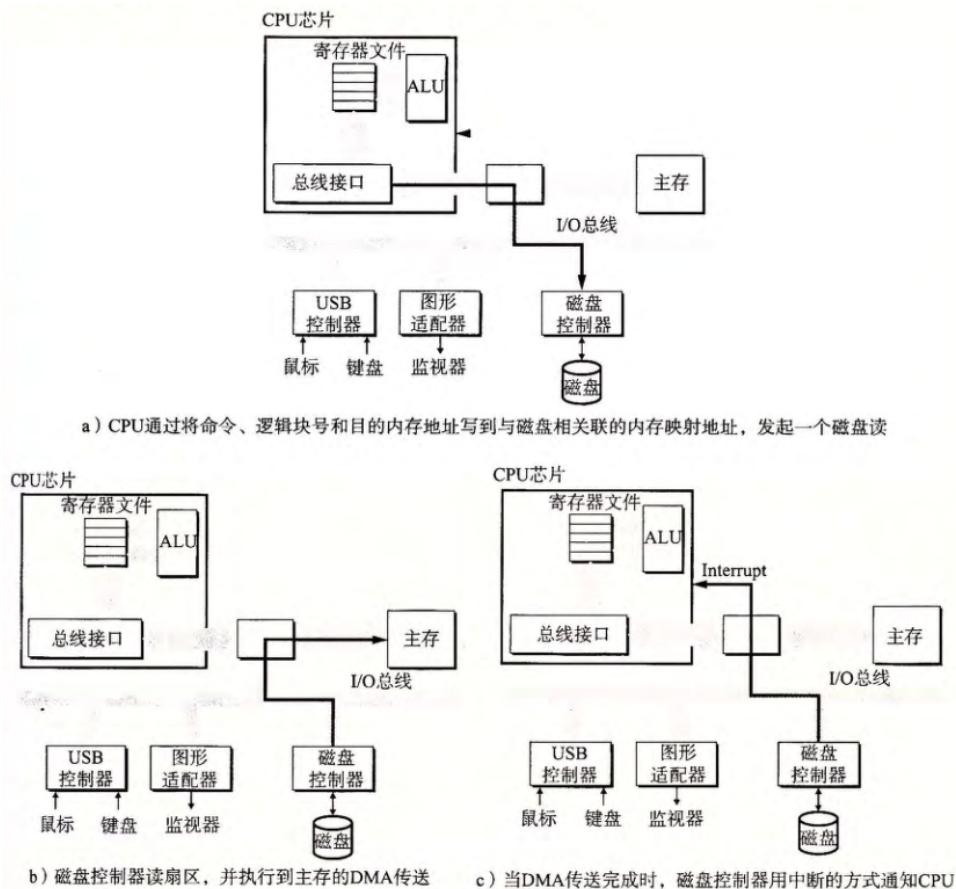
例如图形卡、监视器、鼠标、键盘和磁盘这样的输入/输出 (I/O) 设备，都是通过 I/O 总线连接到 CPU 和主存的。

虽然 I/O 总线比系统总线和内存总线慢，但是它可以容纳种类繁多的第三方 I/O 设备：

- 通用串行总线 (Universal Serial Bus, USB) 控制器
- 图形卡 (或适配器) (Graphics Card/Adapter)
- 主机总线适配器 (Host Bus Adapter, HBA)
- 网络适配器 (Network Adapter)

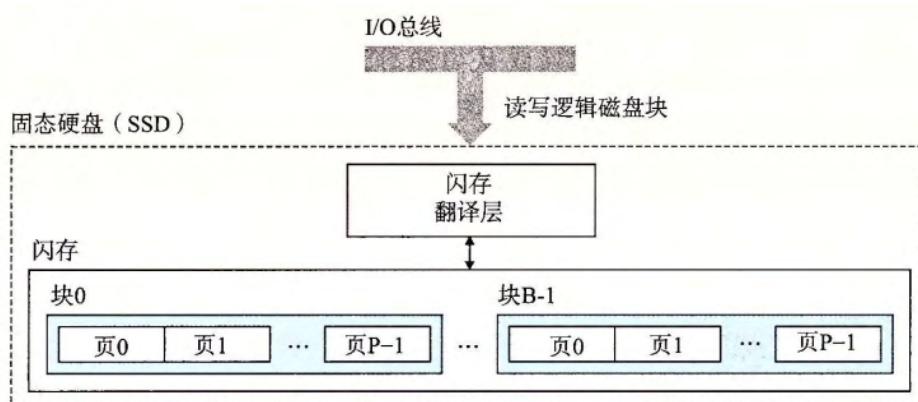


访问磁盘



4.1.3 固态硬盘

固态硬盘 (Solid State Disk, SSD) 是一种基于闪存的存储技术。一个 SSD 封装由一个或多个闪存芯片和闪存翻译层 (flash translation layer) 组成，闪存芯片替代传统旋转磁盘中的机械驱动器，而闪存翻译层是一个硬件/固件设备，扮演与磁盘控制器相同的角色，将对逻辑块的请求翻译成对底层物理设备的访问。



一个闪存由 B 个块的序列组成，每个块由 P 页组成。数据是以页为单位读写的。只

有在一页所属的块整个被擦除之后，才能写这一页（通常是指该块中的所有位都被设置为 1）。不过，一旦一个块被擦除了，块中每一个页都可以不需要再进行擦除就写一次。

局部性原理

局部性原理 (Locality Principle) 是计算机科学中的一个重要概念，指的是在程序执行过程中，访问的数据和指令往往集中在某些特定的区域。这种现象可以分为两种类型：时间局部性和空间局部性。

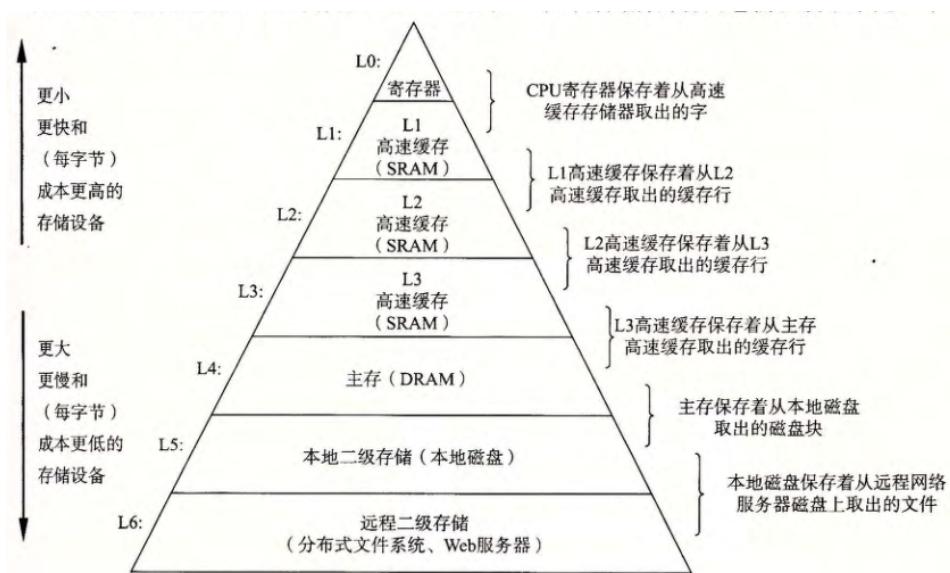
时间局部性 (Temporal Locality) 指的是如果一个数据项在某个时间点被访问，那么在不久的将来它很可能会再次被访问。换句话说，最近使用过的数据很可能会再次被使用。

空间局部性 (Spatial Locality) 指的是如果一个数据项在某个时间点被访问，那么与它相邻的数据项也很可能会在不久的将来被访问。换句话说，程序倾向于访问存储器中相近的地址。

局部性原理是设计高效存储器层次结构（如缓存、主存和辅助存储器）的基础。通过利用局部性原理，计算机系统可以显著提高数据访问速度，减少延迟，从而提升整体性能。

4.2 缓存

4.2.1 存储器层次结构



4.2.2 存储器层次结构中的缓存

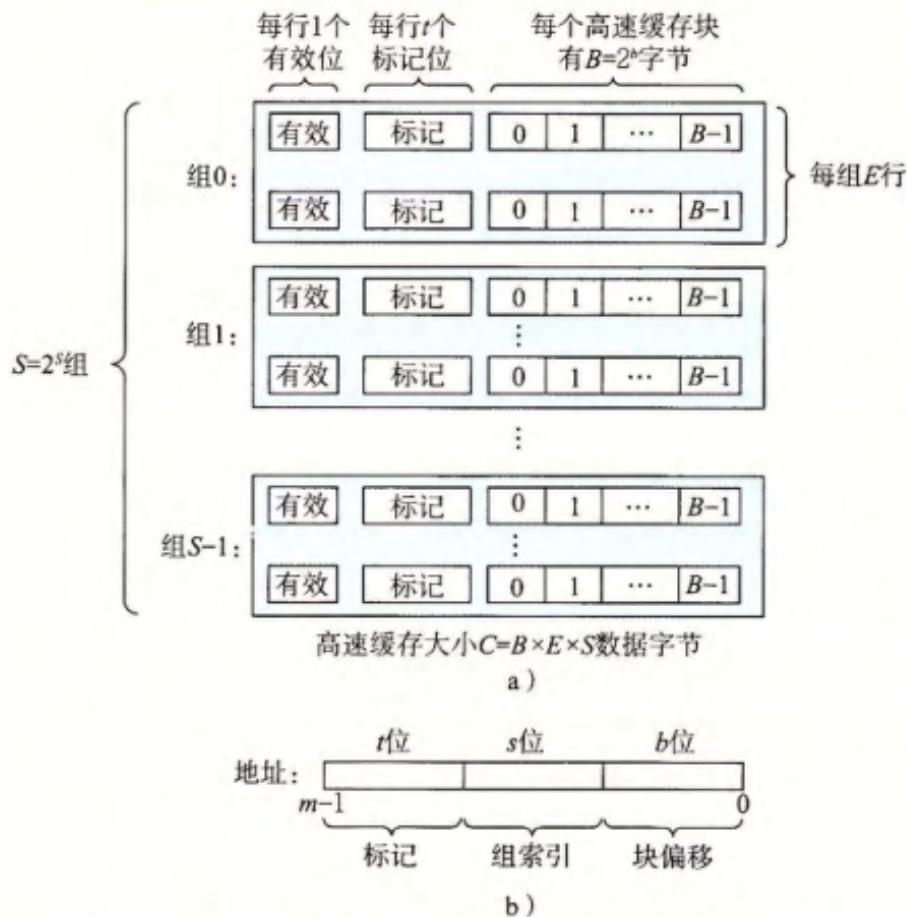
一般而言，高速缓存 (cache) 是一个小而快速的存储设备，它作为存储在更大、也更慢的设备中的数据对象的缓冲区域。存储器层次结构的中心思想是，对于每个 k，位于 k 层的更快更小的存储设备作为位于 k+1 层的更大更慢的存储设备的缓存。

- **缓存命中** (cache hit): 当处理器请求的数据已经存在于缓存中时，就发生了缓存命中。
- **缓存未命中** (cache miss): 当处理器请求的数据不在缓存中时，就发生了缓存未命中。
 - **冷未命中** (cold miss): 也称为初始未命中 (compulsory miss)，当数据第一次被访问时发生，因为它还没有被加载到缓存中。
 - **容量未命中** (capacity miss): 当缓存的容量不足以容纳工作集 (working set) 时发生，即使缓存使用了最优的替换策略，也会发生这种未命中。
 - **冲突未命中** (conflict miss): 当多个数据块映射到缓存中的同一位置时发生，即使缓存有足够的容量，也会发生这种未命中。
 - **一致性未命中** (coherence miss): 在多处理器系统中，当一个处理器修改了缓存中的数据，而另一个处理器试图访问该数据时发生。

4.3 高速缓存存储器

4.3.1 通用的高速缓存存储器组织结构

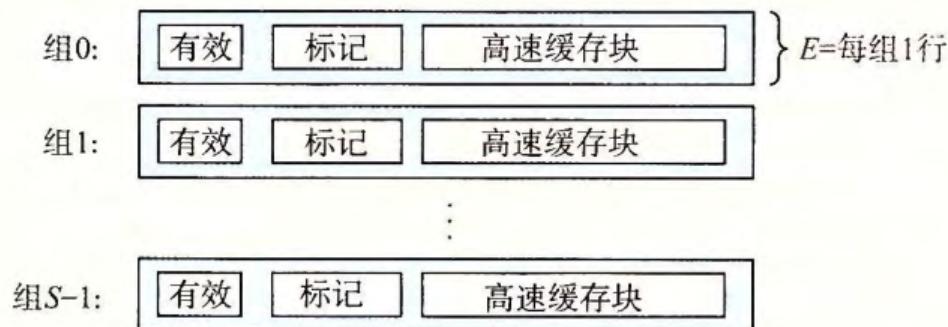
考虑一个计算机系统，其中每个存储器地址有 m 位，形成 $M = 2^m$ 个不同的地址。一个机器的高速缓存被组织成一个有 $S = 2^s$ 个高速缓存组 (cache set) 的数组。每个组包含 E 个高速缓存行 (cache line)。每个行是由一个 $B = 2^b$ 字节的数据块 (block) 组成的，一个有效位 (valid bit) 指明这个行是否包含有意义的信息，还有 $t = m - (h + s)$ 个标记位 (tag bit)，它们唯一地标识存储在这个高速缓存行中的块。



一般而言，高速缓存的结构可以用元组 (C_S, E, B, m) 来描述。高速缓存的大小（或容量） C 指的是所有块的大小的和。标记位和有效位不包括在内。因此， $C = S \times E \times B$ 。

4.3.2 直接映射高速缓存

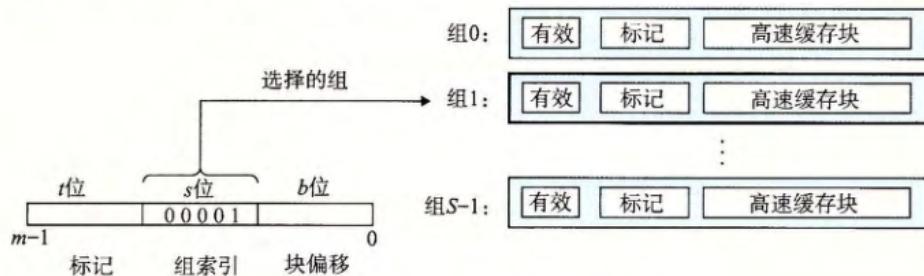
根据每个组的高速缓存行数 E ，高速缓存被分为不同的类。每个组只有一行 ($E = 1$) 的高速缓存称为直接映射高速缓存 (direct-mapped cache)。



高速缓存确定一个请求是否命中，然后抽取出被请求的字的过程，分为三步：组选择；行匹配；字抽取。

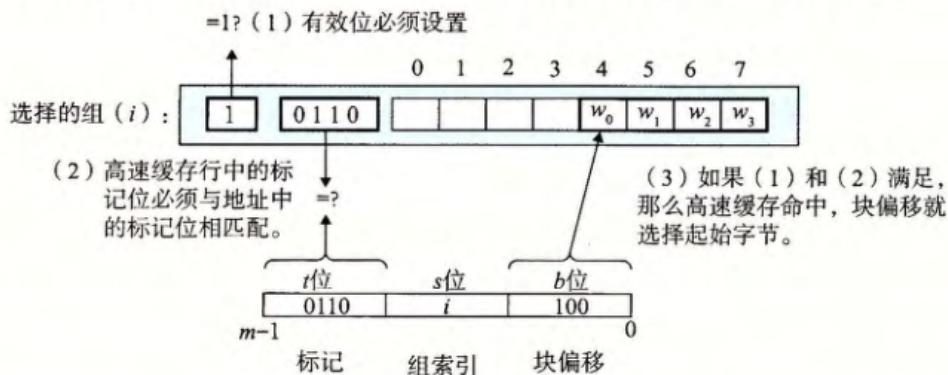
直接映射高速缓存中的组选择

高速缓存从 w 的地址中间抽取出 s 个组索引位。这些位被解释成一个对应于一个组号的无符号整数。



直接映射高速缓存中的行匹配

确定是否有字 w 的一个副本存储在组 i 包含的一个高速缓存行中。当且仅当设置了有效位，而且高速缓存行中的标记与 w 的地址中的标记相匹配时，这一行中包含 w 的一个副本。



直接映射高速缓存中的字选择

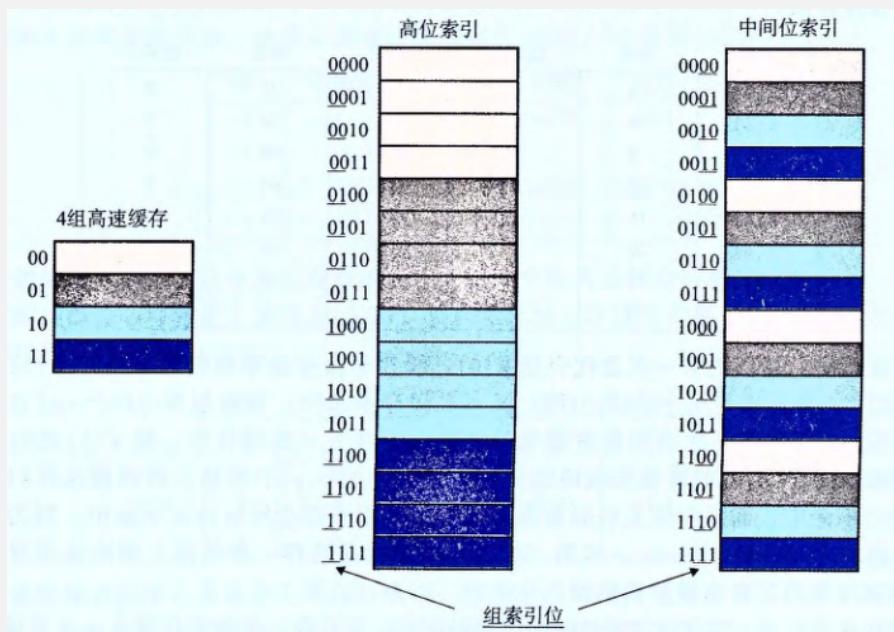
一旦命中，我们知道 w 就在这个块中的某个地方。最后一步确定所需要的字在块中是从哪里开始的。块偏移位提供了所需要的字的第一个字节的偏移。

直接映射高速缓存中不命中时的行替换

如果缓存不命中，那么它需要从存储器层次结构中的下一层取出被请求的块，然后将新的块存储在组索引位指示的组中的一个高速缓存行中。一般而言，如果组中都是有效高速缓存行了，那么必须要驱逐出一个现存的行。对于直接映射高速缓存来说，每个组只包含有一行，替换策略非常简单：用新取出的行替换当前的行。

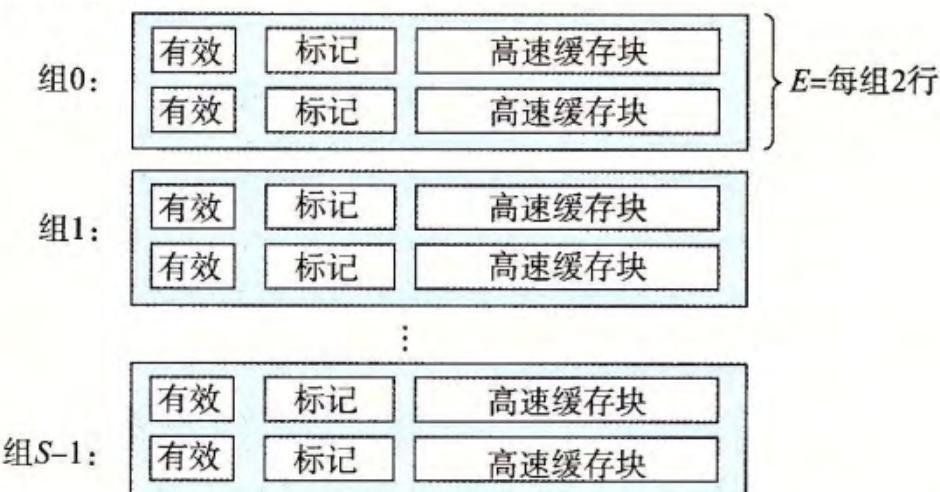
为什么用中间的位来索引

如果高位用做索引，那么一些连续的内存块就会映射到相同的高速缓存块。例如，在图中，头四个块映射到第一个高速缓存组，第二个四个块映射到第二个组，依此类推。如果一个程序有良好的空间局部性，顺序扫描一个数组的元素，那么在任何时刻，高速缓存都只保存着一个块大小的数组内容。这样对高速缓存的使用效率很低。相比较而言，以中间位作为索引，相邻的块总是映射到不同的高速缓存行。



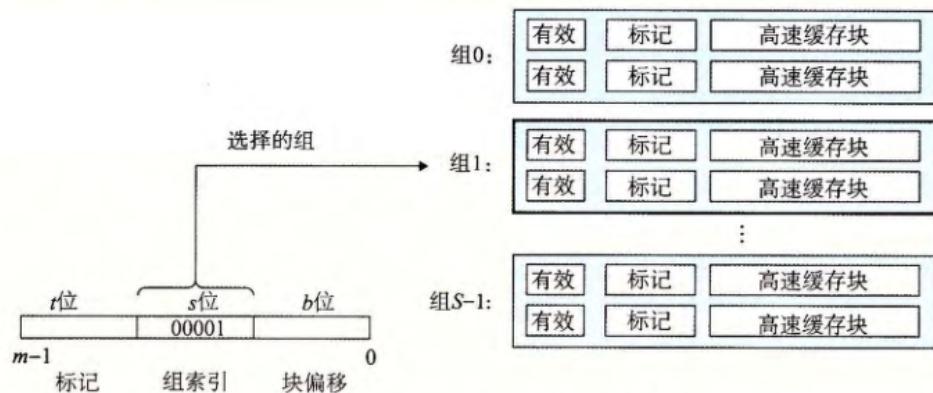
4.3.3 组相联高速缓存

直接映射高速缓存中冲突不命中造成的问题源于每个组只有一行（或者，按照我们的术语来描述就是 $E=1$ ）这个限制。组相联高速缓存 (set associative cache) 放松了这条限制，所以每个组都保存有多个高速缓存行，通常称为 E 路组相联高速缓存。



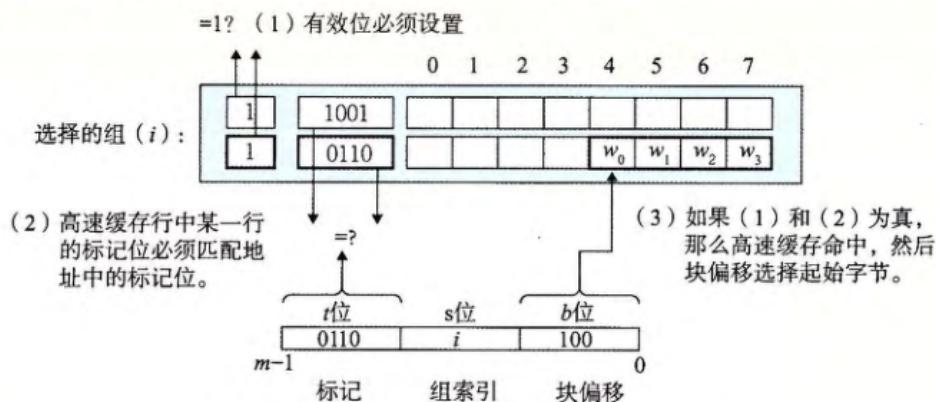
组相联高速缓存中的组选择

组选择与直接映射高速缓存的组选择一样，组索引位标识组。



组相联高速缓存中的行匹配和字选择

组相联高速缓存中的行匹配比直接映射高速缓存中的更复杂。高速缓存必须搜索组中的每一行，寻找一个有效的行，其标记与地址中的标记相匹配。如果高速缓存找到了这样一行，那么我们就命中，块偏移从这个块中选择一个字。



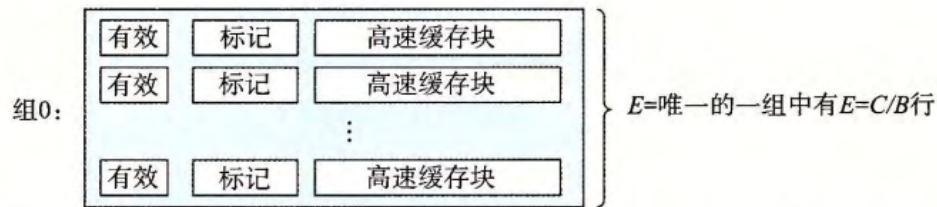
组相联高速缓存中不命中时的行替换

如果 CPU 请求的字不在组的任何一行中，那么就是缓存不命中，高速缓存必须从内存中取出包含这个字的块。不过，一旦高速缓存取出了这个块，该替换哪个行呢？当然，如果有一个空行，那它就是个很好的候选。但是如果该组中没有空行，那么我们必须从中选择一个非空的行，希望 CPU 不会很快引用这个被替换的行。

最简单的替换策略是随机选择要替换的行。其他更复杂的策略利用了局部性原理，以使在比较近的将来引用被替换的行的概率最小。例如，最不常使用 (Least-Frequently-Used, LFU) 策略会替换在过去某个时间窗口内引用次数最少的那一行。最近最少使用 (LeastRecently-Used, LRU) 策略会替换最后一次访问时间最久远的那一行。

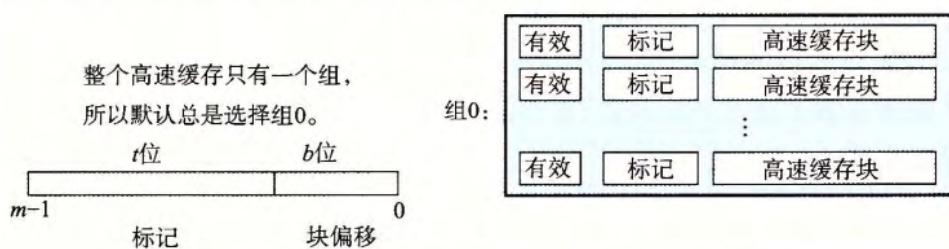
4.3.4 全相联高速缓存

全相联高速缓存 (fully associative cache) 是由一个包含所有高速缓存行的组组成的高速缓存。



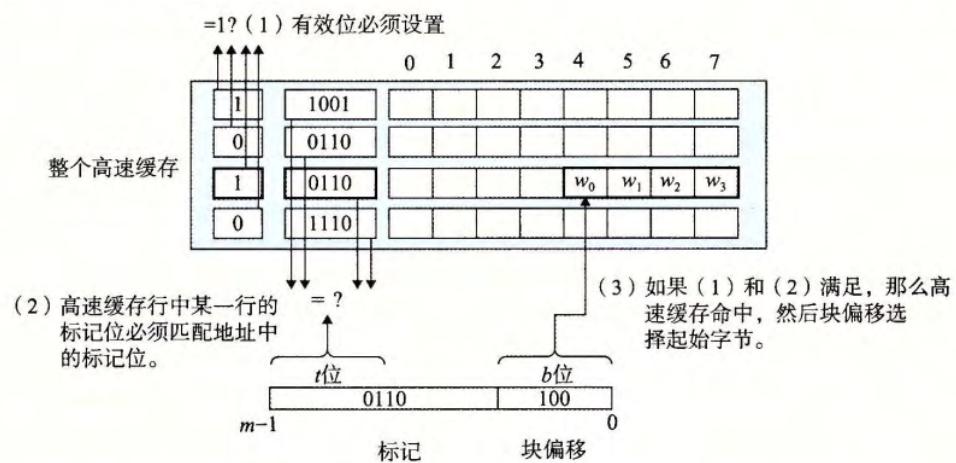
全相联高速缓存中的组选择

注意，全相联高速缓存地址中没有组索引位，地址只被划分成了一个标记和一个块偏移，因为只有一个组。



全相联高速缓存中的行匹配和字选择

全相联高速缓存中的行匹配和字选择与组相联高速缓存中的是一样的，它们之间的区别主要是规模大小的问题。



因为高速缓存电路必须并行地搜索许多相匹配的标记，构造一个又大又快的相联高速缓存很困难，而且很昂贵。因此，全相联高速缓存只适合做小的高速缓存。

4.3.5 高速缓存的写操作

假设我们要写一个已经缓存了的字 w (write hit)。

最简单的方法，称为直写 (write-through)，就是立即将 w 的高速缓存块写回到紧接着的低一层中。虽然简单，但是直写的缺点是每次写都会引起总线流量。

另一种方法，称为写回 (write-back)，尽可能地推迟更新，只有当替换算法要驱逐这个更新过的块时，才把它写到紧接着的低一层中。由于局部性，写回能显著地减少总线流量，但是它的缺点是增加了复杂性。高速缓存必须为每个高速缓存行维护一个额外的修改位 (dirty bit)，表明这个高速缓存块是否被修改过。

另一个问题是处理写不命中。

一种方法，称为写分配 (write-allocate)，加载相应的低一层中的块到高速缓存中，然后更新这个高速缓存块。写分配试图利用写的空间局部性，但是缺点是每次不命中都会导致一个块从低一层传送到高速缓存。

另一种方法，称为非写分配 (not-write-allocate)，避开高速缓存，直接把这个字写到低一层中。

直写高速缓存通常是非写分配的。写回高速缓存通常是写分配的。

4.3.6 高速缓存参数的性能影响

有许多指标来衡量高速缓存的性能：

- 命中率 (hit rate): 命中次数与总访问次数之比。
- 未命中率 (miss rate): 未命中次数与总访问次数之比， $\text{未命中率} = 1 - \text{命中率}$ 。
- 命中时间 (hit time): 从高速缓存传送一个字到 CPU 所需的时间，包括组选择、行确认和字选择的时间。
- 未命中惩罚 (miss penalty): 由未命中所需要的额外的时间。从低一层存储器传送一个块到高速缓存所需的时间，包括传输时间和处理未命中的时间。

优化高速缓存的成本和性能的折中是一项很精细的工作，但也有一些定性的折中考量：

- 高速缓存大小：增大高速缓存的大小通常会提高命中率，但可能会增加命中时间。
- 块大小：增大块大小可以利用空间局部性，帮助提高命中率，但可能会增加未命中惩罚。对于给定的高速缓存大小，块越大就意味着高速缓存行数越少，这会损害时间局部性比空间局部性更好的程序中的命中率。
- 组相联度：增大组相联度通常会降低高速缓存由于冲突不命中出现抖动的可能性。不过，较高的相联度会造成较高的成本。较高的相联度会增加命中时间，因为复杂性增加了，另外，还会增加不命中处罚，因为选择牺牲行的复杂性也增加了。

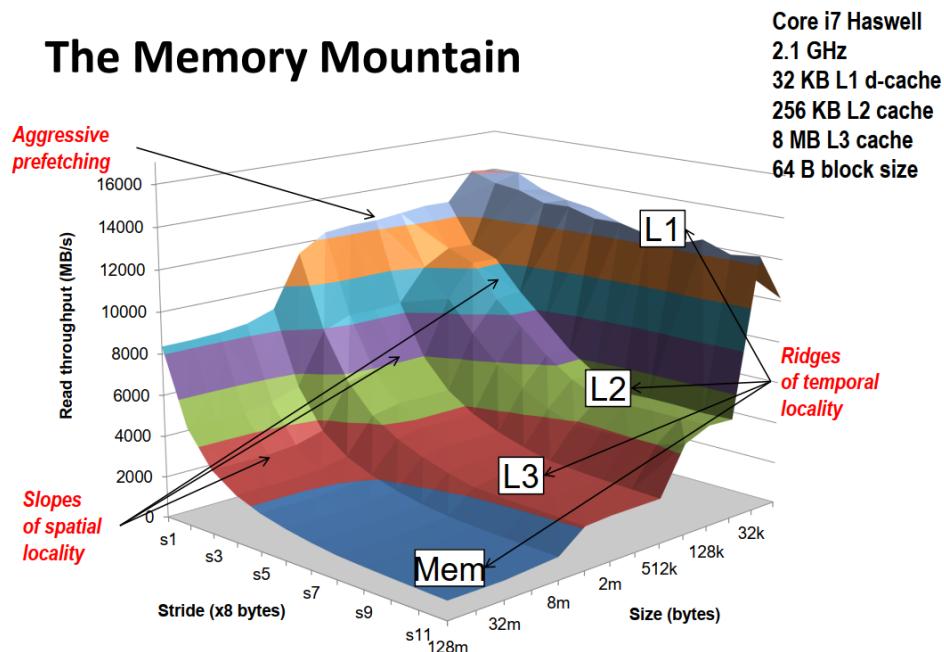
- 写策略：写回通常比直写更复杂，但它能显著地减少总线流量，从而降低未命中惩罚。

4.4 高速缓存友好的代码

确保代码高速缓存友好有一些基本方法：

- 让最常见的情况运行得快。把注意力集中在核心函数里的循环上，而忽略其他部分。
- 尽量减小每个循环内部的缓存不命中数量。

4.4.1 存储器山



4.4.2 一维数组

一般而言，如果一个高速缓存的块大小为 B 字节，那么一个步长为 k 的引用模式（这里 k 是以字为单位的）平均每次循环迭代会有 $\min(1, (\text{wordsize} \cdot k)/B)$ 次缓存不命中。所以步长为 1 的引用确实是高速缓存友好的。

4.4.3 二维数组

行优先顺序：等同于步长为 1 的访问模式。

列优先顺序：如果足够幸运，整个数组都在高速缓存中，那么也会同行优先一样。如果数组比高速缓存要大，那么每次对 $a[i][j]$ 的访问都会不命中！

4.4.4 矩阵乘法

矩阵乘法函数通常是由 3 个嵌套的循环来实现的，分别用索引 i 、 j 和 k 来标识。如果改变循环的次序，对代码进行一些其他的小改动，我们就能得到矩阵乘法的 6 个在功能上等价的版本。为了分析，我们做如下假设：

- 每个数组都是一个 double 类型的 $n \times n$ 的数组。
- 只有一个高速缓存，其块大小为 32 字节 ($B=32$)。
- 数组大小 n 很大，以至于矩阵的一行都不能完全装进 L1 高速缓存中。
- 编译器将局部变量存储到寄存器中，因此循环内对局部变量的引用不需要任何加载或存储指令。

```
----- code/mem/matmult/mm.c ----- code/mem/matmult/mm.c
1  for (i = 0; i < n; i++)           1  for (j = 0; j < n; j++)           1  for (i = 0; i < n; i++) {
2    for (j = 0; j < n; j++) {        2    for (i = 0; i < n; i++) {
3      sum = 0.0;                   3    sum = 0.0;
4      for (k = 0; k < n; k++) {    4    for (k = 0; k < n; k++) {
5        sum += A[i][k]*B[k][j];   5        sum += A[i][k]*B[k][j];
6        C[i][j] += sum;          6        C[i][j] += sum;
7      }                         7      }
----- code/mem/matmult/mm.c ----- code/mem/matmult/mm.c
a) ijk版本                                b) jik版本
----- code/mem/matmult/mm.c ----- code/mem/matmult/mm.c
1  for (j = 0; j < n; j++) {                1  for (k = 0; k < n; k++) {
2    for (k = 0; k < n; k++) {              2    for (j = 0; j < n; j++) {
3      r = B[k][j];                      3    for (i = 0; i < n; i++)
4      for (i = 0; i < n; i++)            4      C[i][j] += A[i][k]*r;
5      C[i][j] += A[i][k]*r;             5
6    }                                     6
----- code/mem/matmult/mm.c ----- code/mem/matmult/mm.c
c) jki版本                                d) kji版本
----- code/mem/matmult/mm.c ----- code/mem/matmult/mm.c
1  for (k = 0; k < n; k++) {                1  for (i = 0; i < n; i++) {
2    for (i = 0; i < n; i++) {              2    for (k = 0; k < n; k++) {
3      r = A[i][k];                      3    r = A[i][k];
4      for (j = 0; j < n; j++) {          4    for (j = 0; j < n; j++)
5        C[i][j] += r*B[k][j];          5        C[i][j] += A[i][k]*r;
6    }                                     6
----- code/mem/matmult/mm.c ----- code/mem/matmult/mm.c
e) kij版本                                f) ikj版本
```

类 AB 例程的内循环以步长 1 扫描数组 A 的行。因为每个高速缓存块保存四个 8 字节的字，A 的不命中率是每次迭代不命中 0.25 次。另一方面，内循环以步长 n 扫描数组 B 的一列。因为 n 很大，每次对数组 B 的访问都会不命中，所以每次迭代总共会有 1.25 次不命中。

类 AC 例程的内循环每次迭代执行两个加载和一个存储（相对类 AB 例程，它们执行 2 个加载而没有存储）。内循环以步长 n 扫描 A 和 C 的列。结果是每次加载都会不命中，所以每次迭代总共有两个不命中。

BC 例程使用了两个加载和一个存储，它们比 AB 例程多需要一个内存操作。内循环以步长为 1 的访问模式按行扫描 B 和 C，每次迭代每个数组上的不命中率只有 0.25 次不命中，所以每次迭代总共有 0.75 个不命中。

矩阵乘法版本 (类)	每次迭代					
	加载次数	存储次数	A未命中次数	B未命中次数	C未命中次数	未命中总次数
$ijk \& jik (AB)$	2	0	0.25	1.00	0.00	1.25
$jki \& kji (AC)$	2	1	1.00	0.00	1.00	2.00
$kij \& ijk (BC)$	2	1	0.00	0.25	0.25	0.50

4.4.5 分块

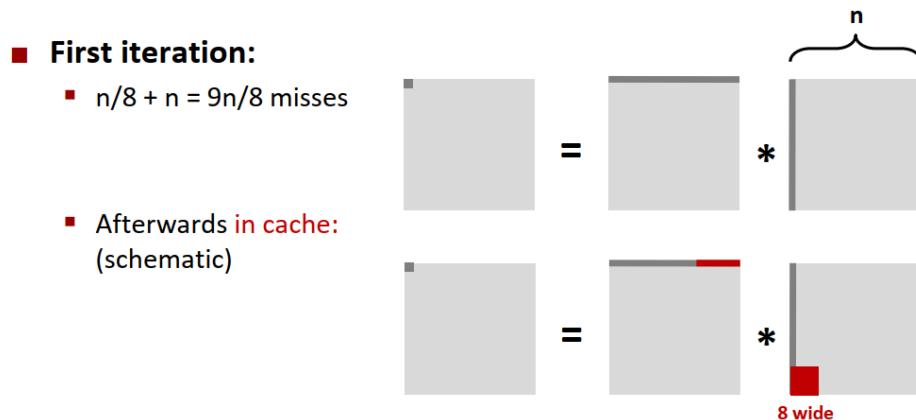
分块的大致思想是将一个程序中的数据结构组织成大的片，称为块 (block)。这样构造程序，使得能够将一个片加载到 L1 高速缓存中，并在这个片中进行所需的所有的读和写，然后丢掉这个片，加载下一个片，依此类推。

与为提高空间局部性所做的简单循环变换不同，分块使得代码更难阅读和理解。由于这个原因，它最适合于优化编译器或者频繁执行的库函数。

假设：

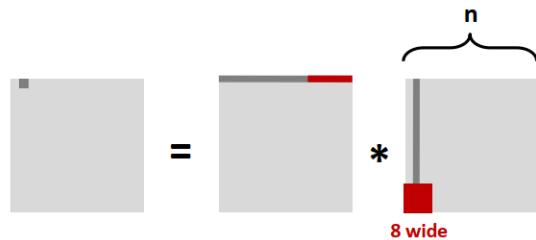
- 数组都一个 double 类型的 $n \times n$ 的数组。
- $\text{block} = 8 \text{ doubles} \times 8 \text{ doubles}$ 。
- 缓存容量 C 远小于 n。

若不采用分块策略：



■ **Second iteration:**

- Again:
 $n/8 + n = 9n/8$ misses



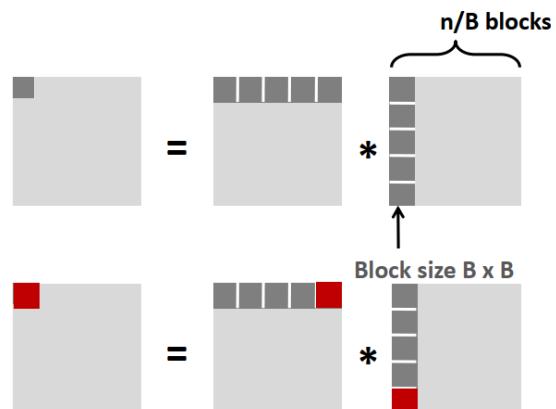
■ **Total misses:**

- $9n/8 * n^2 = (9/8) * n^3$

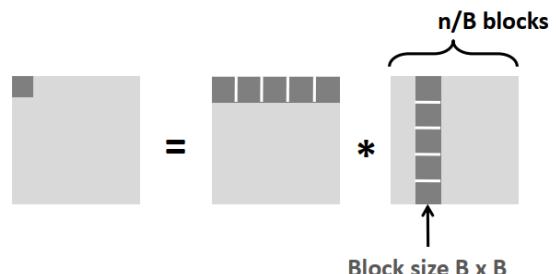
若采用分块策略:

■ **First (block) iteration:**

- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$
 (omitting matrix c)



- Afterwards in cache
 (schematic)



■ **Second (block) iteration:**

- Same as first iteration
- $2n/B * B^2/8 = nB/4$

■ **Total misses:**

- $nB/4 * (n/B)^2 = n^3/(4B)$

比较发现，块大小 B 越大，未命中次数越少，但要求 $3B^2 < C$ 。