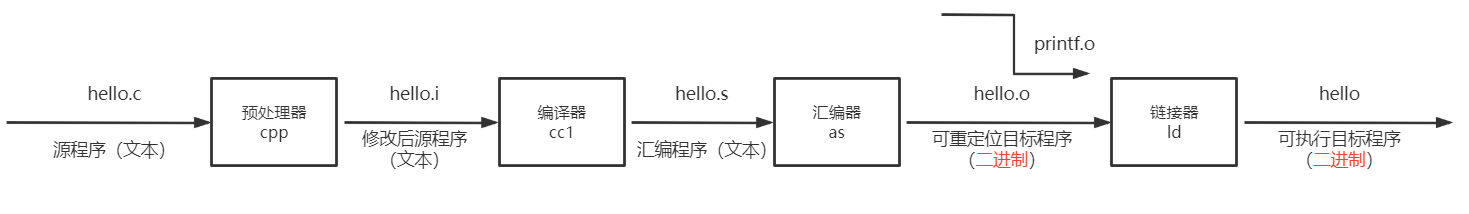
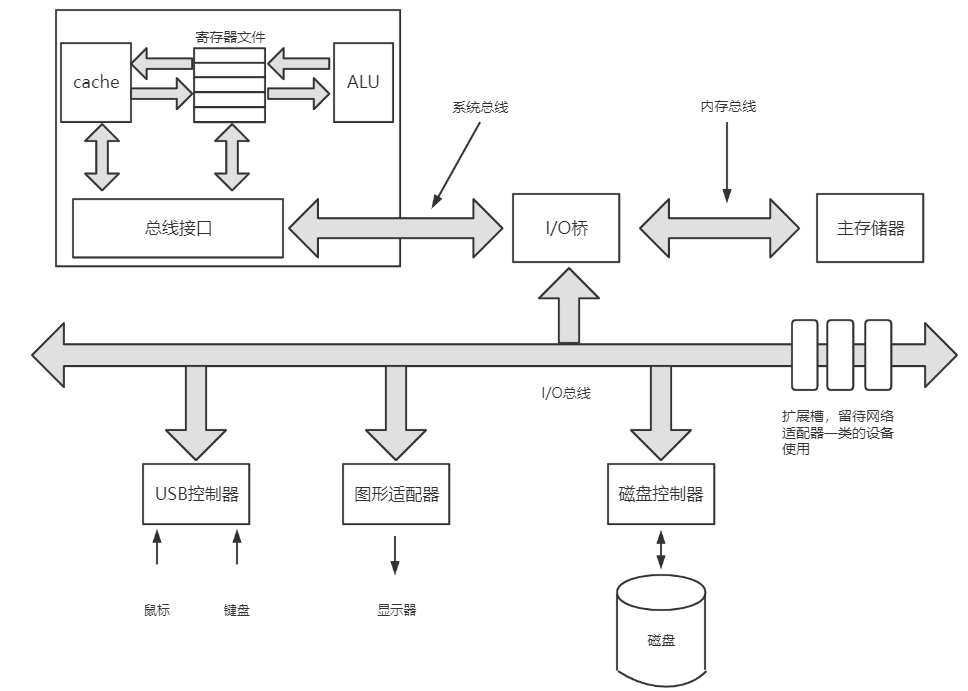
深入理解计算机系统

**一、计算机系统漫游**

**1.1 编译系统：**



**1.2 系统的硬件组成**



1、总线：贯穿整个系统的一组电子管道，携带信息字节并负责在各个部件间传递。 字：总线中传送的定长字节块，字长：字中的字节数，现在系统通常是8字节64位

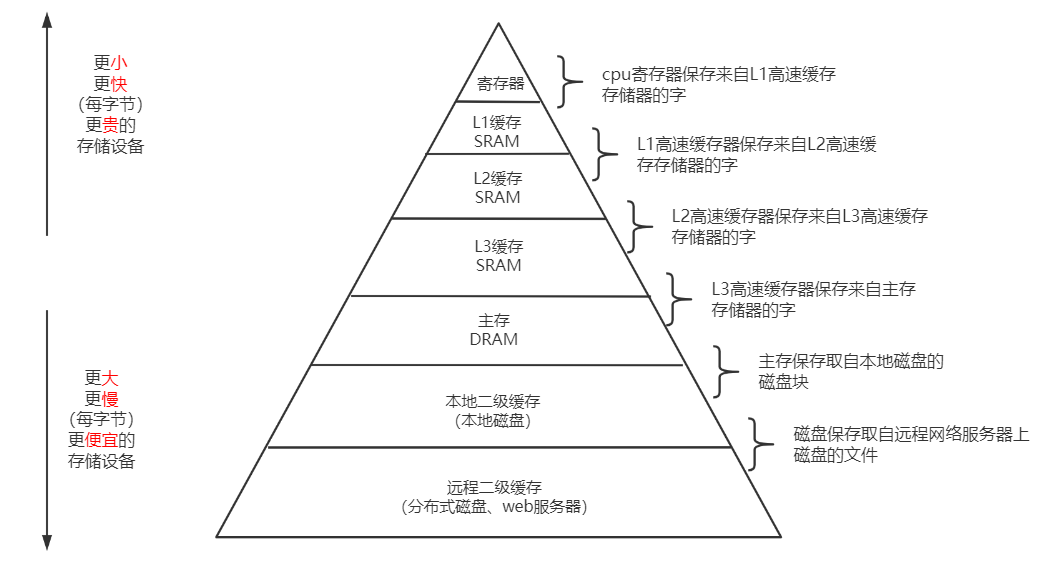
2、I/O设备：输入\输出设备是系统与外界世界的联系通道。通过适配器或控制器与总线相连

3、主存：临时存储设备，在处理器执行程序时，用来存放程序代码和数据

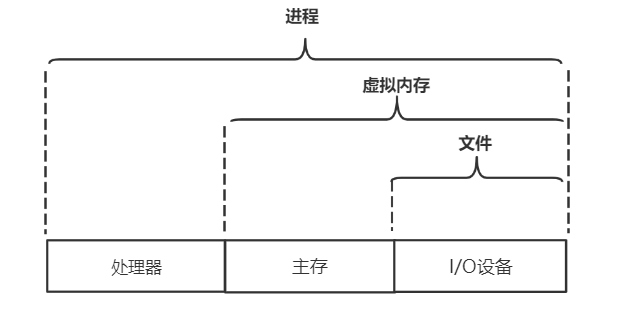
4、处理器(CPU)：中央处理单元，用于处理主存中存放的指令的执行引擎。

5、高速缓冲器(cache)：弥补CPU与主存之间的速度差异，存放处理器近期执行时可能需要的信息

**1.3 存储器层次体系 ★**



**1.4 操作系统管理硬件**



**1.5 Amdahl定律**

a：系统某部分所需执行时间与该时间比例

k：该部分性能提升比例

**二、信息的表示和处理**

**2.1 信息位的表示**

**2.1.1 进制表示法**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16进制 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 10进制 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2进制 | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16进制 | 8 | 9 | A | B | C | D | E | F |
| 10进制 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 2进制 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

★假设数字 x = a0r0 + a1r1 + a2r2 + a3r3 + ……+ anrn ai 表示对应位的系数，r：表示对应进制

十进制：123 =1 \* 102 + 2 \* 101 + 3\*100

十六进制：123 = 7 \* 161+ 12 \* 160 Ox 7B

二进制：123 = 1 \* 26 + 1 \* 25 +1 \* 24 +1 \* 23 + 0 \* 22 + 1 \* 21 + 1 \* 20 0111 1011

**2.1.2 字数据大小**

1、概念：字长指明指针数据的大小，对于字长为 w bit的机器而言，虚拟地址的范围为 0 ~ 2w - 1

1. C基本数据类型大小：

|  |  |  |  |
| --- | --- | --- | --- |
| C声明 | | 字节数 | |
| 有符号 | 无符号 | 32位 | 64位 |
| [signed] char | unsigned char | 1 | 1 |
| short | unsigned short | 2 | 2 |
| int | unsigned | 4 | 4 |
| long | unsigned long | 4 | 8 |
| int32\_t | uint32\_t | 4 | 4 |
| int64\_t | uint64\_t | 8 | 8 |
| char \* |  | 4 | 8 |
| float |  | 4 | 4 |
| double |  | 8 | 8 |

存储顺序：

* 大端法：最高有效字节在最前面
* 小端法：最低有效字节在最前面

例如：Ox 12 34 56 78 小端法：78 56 34 12 大端法：12 34 56 78

**2.2 Bit-level manipulations**

**2.2.1 C语言中的位级运算**

例如：a = 0101 b = 0110

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ~a | ~b | a & b | a | b | a ^ b | a | b |
| a |  |  | 0101 | 0101 | 0101 | 0101 |
| b |  |  | 0110 | 0110 | 0110 | 0110 |
| result | 1010 | 1001 | 0100 | 0111 | 0011 | 0111 |

2.2.2 C语言中逻辑运算

例如：a = 0101 b = 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ！a | ！b | a && b | a || b | ！！a |
| a | 0 |  | 0101 | 0101 | 1 |
| b |  | 1 | 0 | 0 |  |

**2.2.3 C语言中的移位运算（与 java 不同）**

1、左移操作 <<

* 算术左移 = 逻辑左移

2、右移操作 >>

* 无符号数 算术右移 = 逻辑右移
* 有符号数，
  + 如果数 x > 0 算术右移 = 逻辑右移
  + 如果数 x < 0 算术右移 ≠ 逻辑右移，例如：x = 1001 1001 x >> 4 = 1111 1001

**2.3 数据表示 ★:**

**2.3.1 整数定义**

1、无符号定义：

对向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]

2、有符号：有符号数计算机通过补码(Two's-complement)表示

对向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]

3、重要数字表示

★注：|TMinw| = TMaxw + 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数 | 字长w | | | |
| 8 | 16 | 32 | 64 |
| UMaxw | 0xFF | 0xFFFF | 0xFFFF FFFF | 0xFFFF FFFF FFFF FFFF |
| 255 | 65536 | 4 294 967 295 | 18 446 744 073 709 551 615 |
| TMinw | 0x80 | 0x8000 | 0x8000 0000 | 0x8000 0000 0000 0000 |
| -128 | -32768 | -2 147 483 648 | -9 223 372 036 854 775 808 |
| TMaxw | 0x7F | 0x7FFFF | 0x7FFF FFFF | 0x7FFF FFFF FFFF FFFF |
| 127 | 32767 | 2 147 483 647 | 9 223 372 036 854 775 807 |
| -1 | 0xFF | 0xFFFF | 0xFFFF FFFF | 0xFFFF FFFF FFFF FFFF |
| 0 | 0x00 | 0x0000 | 0x0000 0000 | 0x0000 0000 0000 0000 |

注：还有两种表示方法

反码（Ones' Complement）：

对向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]

原码 (Sign-Magnitude)：

对向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]

★★推论：无符号数→补码

对向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]

**2.3.2 C语言中的有符号数和无符号数**

1、C语言创建无符号常量，后缀添加 U 即可

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 类型 | 求值 | 原因 |
| 0 == 0 U | 无符号 | 1 |  |
| -1 ＜ 0 | 有符号 | 1 |  |
| -1 ＜ 0 U | 无符号 | 0 | 0x FFFF FFFF ＜ 0x 0000 0000 False |
| 2147483647 ＞ –2147483647 -1 | 有符号 | 1 |  |
| 2147483647 U ＞ –2147483647- 1 | 无符号 | 0 | 0x 7FFF FFFF ＞ 0x8000 0000 False |
| 2147483647 ＞ (int)2147483648 U | 有符号 | 1 | 无符号强制转有符号 (int)2147483648 = -1 |
| -1 ＞ -2 | 有符号 | 1 |  |
| （unsigned）-1 ＞ -2 | 无符号 | 1 | 有符号强制转无符号 (unsigned)-1 = 0xFFFF FFFF |

★2、如果执行运算两个数中一个数是有符号数，一个数是无符号数，C语言会默认将有符号数强制转换成无符号数

**2.3.3 数据的扩展与截断**

1、无符号数的扩展：当无符号数转换为一个更大的数时，只需要在开头添加 0 ，这种被称为零扩展。

定义宽度为的w位向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]和宽度为w'位向量 =[xw-1，xw-2，xw-3，……，x2，x1，x0]位向量，其中w'＞w，则:

2、补码数的扩展：当补码数转换为一个更大的数时，可以对最高位执行符号扩展

定义宽度为w的位向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]和宽度为w’位向量=[xw-1，xw-2，xw-3，……，x2，x1，x0]，其中w'＞w，则：

例如：

short x = -12345 Ox CF C7 → int x = -12345 Ox FF FF CF C7 (符号扩展)

unsigned short ux = 53191 Ox CF C7 → unsigned x = 53191 Ox 00 00 CF C7 (零扩展)

3、无符号数的截断：将一个w位的数强制截断为一个k位数字时，丢弃高w-k位，即：x' = x mod 2k ，有符号数的截断类似，只不过需要把最高位转换为符号位。

例如：

int x = 53191 Ox 00 00 CF C7 → short x = -12345 Ox CF C7

unsigned ux = 53191 Ox 00 00 CF C7 → unsigned short x = 53191 Ox CF C7

**2.3.4 整数运算**

1、加法：

①无符号数：

假设 0 ≤ u，v ≤ 2w - 1，则 0 ≤ u+v ≤ 2w+1 - 2 ，需要w+1位表示，因此丢弃进位s = UAddw(u,v) = (u + v) mod 2w，(如果 s ＜ u，或者 s < v，则存在溢出)

②有符号数：

假设 -2w-1 ≤ x，y ≤ 2w-1 - 1，则 -2w ≤ x+y ≤ 2w - 2 ，需要w+1位表示，因此像之前一样截断到w位。

对满足，-2w-1≤x，y≤2w-1-1；

2、乘法：

①无符号数：

假设 0 ≤ u，v ≤ 2w - 1，则 0 ≤ u\*v ≤ 22w -2w+1 + 1 ，需要w+1位表示，因此丢弃进位

s = (u \* v) mod 2w

②有符号数：

假设 -2w-1 ≤ x，y ≤ 2w-1 - 1，则 2w-1-22w-2 ≤ x，y ≤22w-2 - 2 ，可能需要2w位表示，因此像之前一样截断到w位，再把无符号数转换为补码。

对满足-2w-1 ≤ x，y ≤ 2w-1 – 1：

3、取非：

①无符号数：

对满足0≤x，y≤2w-1的任意x，其位的无符号逆元给出：

②有符号数：

对满足，TMinw ≤x，y≤TMaxw的任意x，其位的无符号逆元由给出：

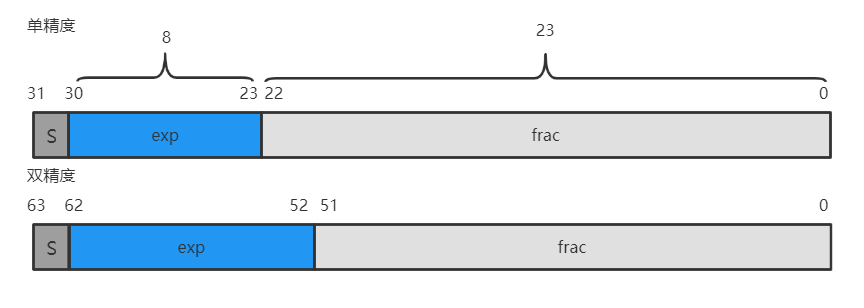
**2.4 IEEE 浮点数★**

**2.4.1 IEEE浮点表示法**

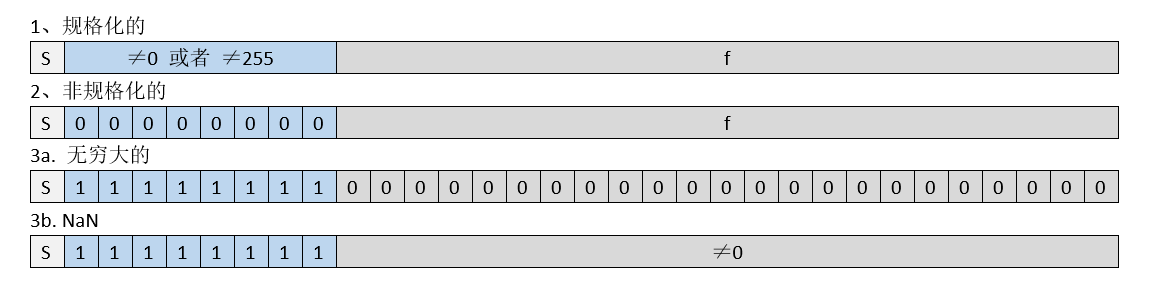
IEEE浮点标准用 V = (-1)s×M×2E 的形式表示一个数：

* 符号位（sign）：s决定这数是正数（s=0）还是负数（s=1），对于数值0的符号位解释另作处理
* 尾数（significand）：M是一个二进制小数，它的范围是1~ 2- ε，或者0 ~ 1 - ε
* 阶码（exponent）：E的作用是对浮点数加权，这个权重是2的E次幂。

1、C语言中的常见浮点数格式



2、IEEE编码表示：



①情况1：规格化的值

当exp ≠ 0 或者exp ≠ 255 时，这是最普遍的情况

* E = e - Bias 单精度的Bias = 127 (2K-1 - 1)，其中e是无符号数，表示为ek-1……e1e0，K = 阶码位数
* M = 1 + f 二进制表示 1.fn-1……f1f0，由于1是固定位可以省略不写

②情况2：非规格化的值，提供了表示数值0的方法，表示接近数值0的方法

当exp = 0时

* E = 1 - Bias 单精度中，E = -126
* M = f 不包含1

③情况3：特殊值，当exp = 255时，出现

* 如果f = 0，表示无穷大
* 如果f ≠ 0，表示NaN

注：例子如下，当 e = 0 时 将E 定义为 1 - bias，而不是-bias，可以补偿非规格数的尾数没有隐含的开头的1。

假设 k = 3 的阶码位数，n = 2 的小数位，bias = 3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 描述 | 位表示 | 指数 | | | 小数 | | 值 | | |
| e | E | 2E | f | M | 2E×M | V | 十进制 |
| 0 | 0 000 00 | 0 | -2 |  |  |  |  | 0 | 0.0 |
|  | 0 000 01 | 0 | -2 |  |  |  |  |  | 0.0625 |
|  | 0 000 10 | 0 | -2 |  |  |  |  |  | 0.125 |
|  | 0 000 11 | 0 | -2 |  |  |  |  |  | 0.1875 |
| 最小的规 | 0 001 00 | 1 | -2 |  |  |  |  |  | 0.25 |
| 格化数 | 0 001 11 | 1 | -2 |  |  |  |  |  | 0.4375 |
|  | 0 010 00 | 2 | -1 |  |  |  |  |  | 0.5 |
|  | 0 010 01 | 2 | -1 |  |  |  |  |  | 0.625 |
| 最大的规 | … |  |  |  |  |  |  |  |  |
| 格化数 | 0 110 11 | 6 | 3 | 8 |  |  | 14 | 14 | 14.0 |
| 无穷大 | 0 111 00 | —— | —— | —— | —— | —— | —— | ∞ | —— |

**2.4.2 舍入**

1、小数舍入方式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 方式 | 1.40 | 1.60 | 1.50 | 2.50 | -1.50 |
| 向偶数舍入 | 1 | 2 | 2 | 2 | -2 |
| 向零舍入 | 1 | 1 | 1 | 2 | -1 |
| 向下舍入 | 1 | 1 | 1 | 2 | -2 |
| 向上舍入 | 2 | 2 | 2 | 3 | -1 |

2、二进制小数舍入方式

二进制（向1/4 舍入）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方式 | 11.001002 | 11.001102 | 11.000102 | 11.101002 |
| 向偶数舍入 | 11.002 | 11.012 | 11.002 | 11.102 |
| 解释 | 11.00 100  是中间值因此向偶数舍入 | 11.00 110  不是中间值，向上舍入 | 11.00 010  不是中间值，向下舍入 | 11.00 010  是中间值，向偶数舍入 |

**2.4.3 浮点运算**

1、浮点数乘法：

假设两个数：（-1）S1 M12E1×（-1）S2 M22E2

步骤：

|  |  |  |  |
| --- | --- | --- | --- |
| 符号位 | 尾数 | 指数 | 结果 |
| S = S1 ^ S2 | M = M1×M2 | E = E1+E2 | （-1）SM·2E |

规格化：

* 如果M ≥ 2，右移M使M在1~2的范围，同时增加E的数值
* 如果E超过范围，则溢出，该结果是无穷大
* 如果M位数很多，必须使用向偶数舍入的方式进行四舍五入。

例：假设阶码位数k=3，小数位n=2，则bias = 23-1 – 1 = 3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 符号位 | 尾数 | 指数 | 值 |
| 0 011 10 | 0 | 1+0.5=6/4 | 23-3 | 6/4=1.5 |
| 1 010 01 | 1 | 1+0.25 =5/4 | 22-3 | -5/8 = -0.625 |
| 计算 | S = S1 ^ S2 = 1 | M = 6/4 \* 5/4 = 15/8 | E = E1+E2=2-1 | -15/16 |

规格化：

尾数15/8 = 1.111过多，需要向偶数舍入，M = 2.00；

此时，M≥2，需要将M右移1位，指数+1，M = 1.00，E = 20

综上所述：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 符号位 | 尾数 | 指数 | 值 | 值 |
| 1 | 1.00 | 23-3 | -1.00 | 1 011 00 |

2、浮点加法：

假设两个数：（-1）S1 M12E1 +（-1）S2 M22E2，E1＞E2

|  |  |  |  |
| --- | --- | --- | --- |
| 符号位 | 尾数 | 指数 | 结果 |
| S = S1 | M = M1+M2>>（E1-E2） | E = E1 | （-1）SM·2E |

规格化：

* 如果M ≥ 2，右移M使M在1~2的范围，同时增加E的数值
* 如果M < 1，左移M使M在1~2的范围，同时减少E的数值
* 如果E超过范围，则溢出，该结果是无穷大
* 如果M位数很多，必须使用向偶数舍入的方式进行四舍五入。

浮点是不满足结合律的

三、程序的机器级表示

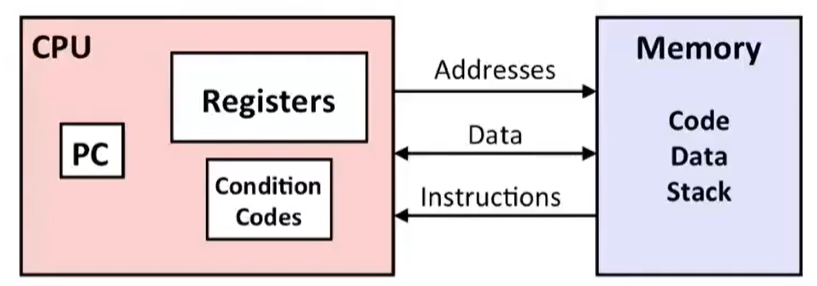
**3.1 概论**

**3.1.1 历史**

1、CISI （Complex instruction set computer）Intel，AMD

2、RISI：ARM

**3.1.2 机器代码**



PC：程序计数器，Program counter

* 下一条指令的地址
* 在X86-64中称为RIP
* Register file：寄存器文件，临时保存ALU运算需要的数据
* Condition codes：条件码寄存器
* 存储最近一次算术运算的状态
* 应用于条件分支

Memory：（1）可寻址字节数组（2）代码和用户数据（3）栈

**3.2 汇编基础 ★★★**

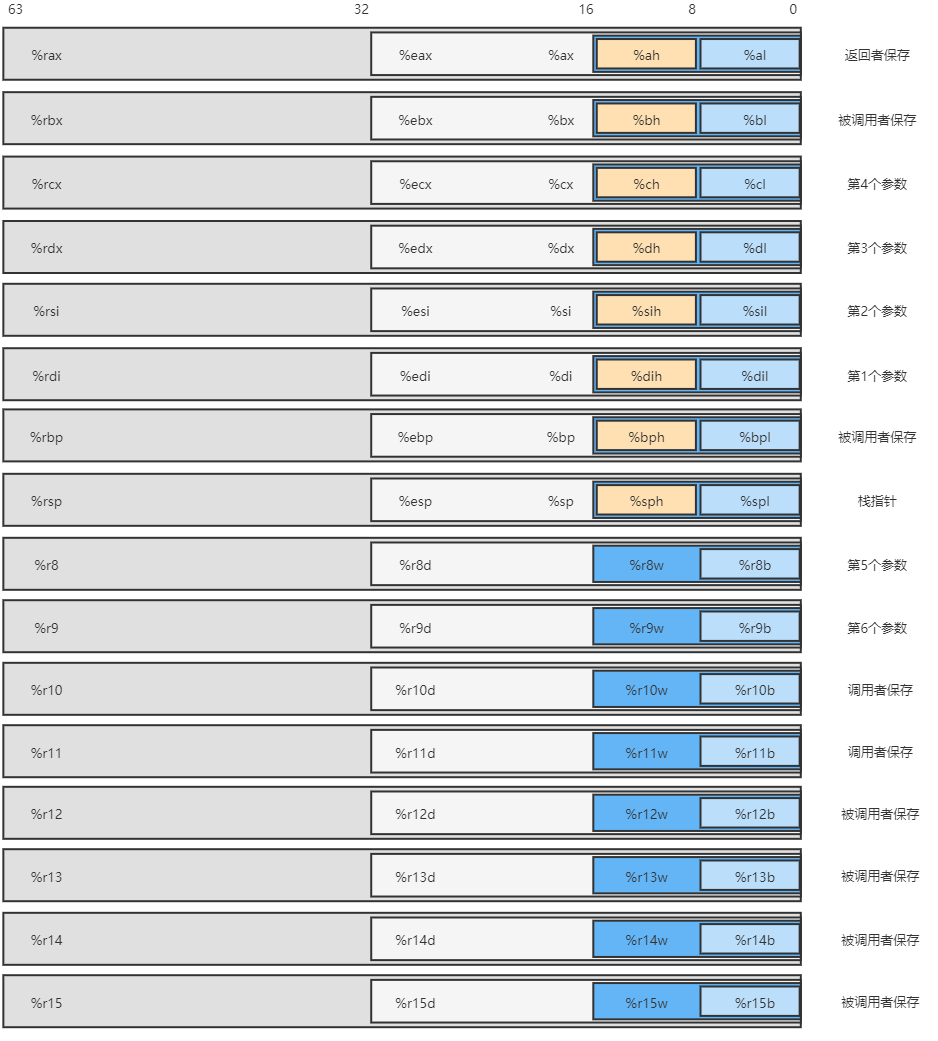
**3.2.1 x86-64中，C语言数据格式**

Intel是从16位体系结构扩展成32位，Intel用“字（word）”表示16位数据类型。（★与第二节中字定义的含义不同）

|  |  |  |  |
| --- | --- | --- | --- |
| C声明 | Intel数据集 | 汇编代码后缀 | 大小（字节） |
| char | 字节 | b | 1 |
| short | 字 | w | 2 |
| int | 双字 | l | 4 |
| long | 四字 | q | 8 |
| char \* | 四字 | q | 8 |
| floar | 单精度 | s | 4 |
| double | 双精度 | l | 8 |

**3.2.2 x86-64信息访问**

1、x86寄存器★★



2、操作数指示符

★注：Imm(rb,ri,s) 是最常用模式，rb：基址寄存器，ri：变址寄存器，寄存器必须是64位，s：比例因子，1，2，4或者8。有效地址被计算为Imm + R[rb] + R[ri] \* s

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 格式 | 操作数值 | 名称 |
| 立即数 | $*Imm* | *Imm* | 立即数寻址 |
| 寄存器 | ra | R[ra] | 寄存器寻址 |
| 存储器 | *Imm* | M[*Imm*] | 绝对寻址 |
| 存储器 | (ra) | M[R[ra]] | 间接寻址 |
| 存储器 | *Imm*(rb) | M[R[rb] + *Imm*] | (基址+偏移量)寻址 |
| 存储器 | (rb,ri) | M[R[rb] + R[ri]] | 变址寻址 |
| 存储器 | *Imm* (rb,ri) | M[R[rb] + R[ri] + *Imm*] | 变址寻址 |
| 存储器 | (,ri,s) | M[R[ri] \* s] | 比例变址寻址 |
| 存储器 | *Imm* (,ri,s) | M[R[ri] \* s + *Imm*] | 比例变址寻址 |
| 存储器 | (rb,ri,s) | M[R[rb] + R[ri] \* s] | 比例变址寻址 |
| 存储器 | *Imm* (rb,ri,s) | M[R[rb] + R[ri] \* s + *Imm*] | 比例变址寻址 |

例子：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址 | 值 |  | 寄存器 | 值 |
| 0x100 | 0xFF |  | %rax | 0x100 |
| 0x104 | 0xAB |  | %rcx | 0x1 |
| 0x108 | 0x13 |  | %rdx | 0x3 |
| 0x10C | 0x11 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 操作数 | 值 | 操作数 | 值 |
| %rax | 0x100 | 260(%rcx,%rdx) | M[264 = 0x108] = 0x13 |
| 0x104 | 0xAB | 0xFC(,%rcx,4) | M[FC+1\*4] = 0xFF |
| $0x108 | 0x108 | (%rax,%rdx,4) | M[0x100+4\*0x3] =0x11 |
| (%rax) | 0xFF |  |  |
| 4(%rax) | 0xAB |  |  |
| 9(%rax,%rdx) | 0x11 |  |  |

3、数据传送指令

★★★注：b：字节 w：字（16bit） l：双字（32bit）q：四字（64bit）

①从源位置复制到另一个位置，不做任何变化

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOV S D | D ← S | 传送，S可以是立即数，内存地址，寄存器中数D可以是寄存器，内存地址，且S和D不能都指向内存中位置 |
| movb，movw，movl，movq |  | 最后一个字母表示传送大小 |
| movabsq |  | 传送绝对四字 |

②将较小的数复制到较大的目的地，MOVZ类把目的剩余的字节填充为0，MOVS类则通过符号进行填充。

最后两个字符都是大小指示符，第一个指示源大小，第二个指示目的大小，源包含1个和2个字节，目的包含1字，2字，四字情况

零扩展：

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOVZ S,R | R ←零扩展S | 以零扩展进行传送 |
| movzbw，movzbl，movzbq  movzwl，movzwq |  | 将做了零扩展的数据传送到目的地 |

符号扩展：

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOVS S,R | R ←符号扩展S | 传送符号扩展的字节 |
| movsbw，movsbl，movsbq  movswl，movswq，movslq |  | 将做了符号扩展的数据传送到目的地 |
| cltq | %rax ←符号扩展（%eax） | 把%eax扩展到%rax |

注：cltq只作用用于%eax和%rax，l是双字，q是四字

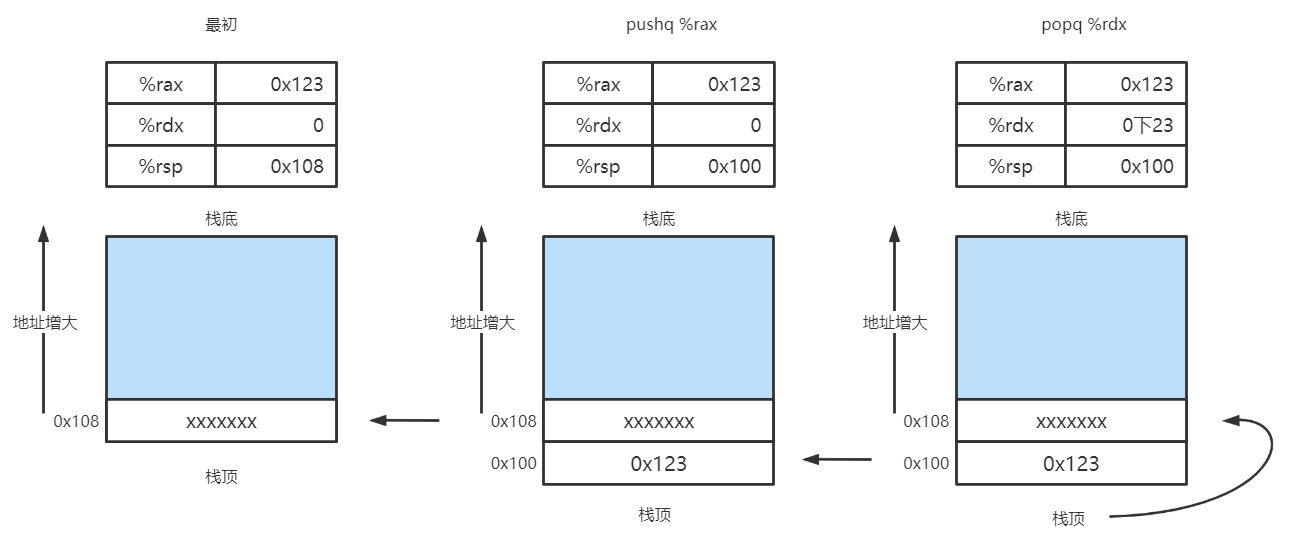
例：

|  |  |  |
| --- | --- | --- |
| movabsq | $0x0011 2233 4455 6677, %rax | %rax = 0011 2233 4455 6677 |
| movb | $0xAA, %dl | %dl = AA |
| movb | %dl, %al | %rax = 0011 2233 4455 66AA |
| movsbq | %dl, %rax | %rax = FFFF FFFF FFFF FFAA |
| movzbq | %dl, %rax | %rax = 0000 0000 0000 00AA |

4、压入和弹出栈数据

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| pushq S | R[%rsp] ← R[%rsp] – 8  M[R[%rsp]] ← S | 将四字压入栈中 |
| popq R | R ← M[R[%rsp]]  R[%rsp] ← R[%rsp] + 8 | 将四字弹入寄存器 |

x86-64中，栈是向低地址方向增长



**3.2.3 算术和逻辑操作**

1、普通的操作

这些操作被分为四类：加载有效地址，一元操作，二元操作，移位操作

注：除了leaq指令外，其他指令都有b，w，l，q四种不同大小数据指令。

移位量：可以是一个立即数。或者存放在单字节%cl中。在x86-64中，移位操作对w位长的数据值进行操作，由%cl的低m位决定的，这里2m = w ，高位会被忽略，例如%cl中的数是0xFF时，salb会移7bit，salw会移15bit，sall会移动16bit，salq会移动63bit

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| leaq S，D | D ← &S | 加载有效地址  leaq (%rdi,%rsi,4), %rax %rax= %rdi + 4 \* %rsi |
| inc D  dec D  neg D  not D | D ← D + 1  D ← D - 1  D ← -D  D ← ~D | +1  -1  取相反数  取反 |
| add S，D  sub S，D  imul S，D  xor S，D  or S，D  and S，D | D ← D + S  D ← D - S  D ← D \* S  D ← D ^ S  D ← D | S  D ← D & S | 加  减  乘  异或  或  与 |
| sal k，D  shl k，D  sar k，D  shr k，D | D ← D << k  D ← D << k  D ← D >>A k  D ← D >>L k | 算术左移  逻辑左移（等同于sal）  算术右移  逻辑右移 |

2、特殊的算术操作

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| imulq S  mulq S | R[%rdx] : R[%rax] ← S×R[%rax]  R[%rdx] : R[%rax] ← S×R[%rax] | 有符号全乘法  无符号全乘法 |
| cqto R | R[%rdx] : R[%rax] ← 符号扩展（R[%rax]） | 转换为八字 |
| idivq S  divq S | R[%rdx] ← R[%rdx] : R[%rax] mod S  R[%rax] ← R[%rdx] : R[%rax] ÷ S | 有符号除法  无符号除法 |

**3.2.4 控制指令**

1、条件码 CF：进位标志。最近的操作使最高位产生了进位。可用来检测无符号操作的溢出。 ZF：零标志。最近的操作得出的结果为0。 SF：符号标志。最近的操作得出的结果为负数。 OF：溢出标志。最近的操作导致一个补码溢出——正溢出和负溢出

2、比较和测试指令

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| CMP S1，S2  cmpb(w,l,q) | S1 – S2 | 比较  比较字节(字，双字，四字) |
| TEST S1，S2  testbb(w,l,q) | S1 & S2 | 测试  测试字节 |

3、访问条件码

4、分支跳转指令

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 同义名 | 效果 | 描述 |
| jmp Label |  | 1 | 直接跳转 |
| jmp \*Operand |  | 1 | 间接跳转 |
| je Label | jz | ZF | 相等/零 |
| jne Label | jnz | -ZF | 不等/非零 |
| js Label |  | SF | 负数 |
| jns Label |  | -SF | 非负数 |
| jg Label | jnle | ~(SF^OF) & ~ZF | 大于（有符号 ＞） |
| jge Label | jnl | ~(SF^OF) | 大于或等于（有符号 ≥ ） |
| jl Label | jnge | SF^OF | 小于（有符号 ＜） |
| jle | jng | (SF^OF) | ZF | 小于等于（有符号 ≤） |
| ja Label | jnbe | ~CF & ~ZF | 超过（无符号 ＞） |
| jae Label | jnb | ~CF | 超过相等（无符号 ≥） |
| jb Label | jnae | CF | 低于（无符号 ＜） |
| jbe Label | jna | CF | ZF | 低于等于（无符号 ≤） |

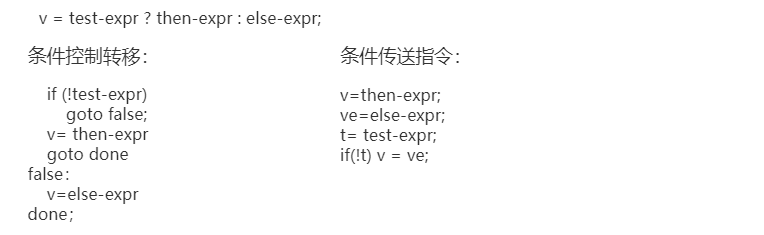
确定分支预测惩罚：预测错误的概论是p，执行代码的时间的TOK，而预测错误的处罚是TMP

Tavg(p) = (1-p) TOK + p(TOK + TMP) = TOK + pTMP (Tran : p = 0.5的平均时间)

5、条件传送指令（为了减少分支预测失败带来的惩罚）

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 同义名 | 效果 | 描述 |
| cmove S，R | cmovz | ZF | 相等/零 |
| cmovne S，R | cmovnz | -ZF | 不等/非零 |
| cmovs S，R |  | SF | 负数 |
| cmovns S，R |  | -SF | 非负数 |
| cmovg S，R | cmovnle | ~(SF^OF) & ~ZF | 大于（有符号 ＞） |
| cmovge S，R | cmovnl | ~(SF^OF) | 大于或等于（有符号 ≥ ） |
| cmovl S，R | cmovnge | SF^OF | 小于（有符号 ＜） |
| cmovle S，R | cmovng | (SF^OF) | ZF | 小于等于（有符号 ≤） |
| cmova S，R | cmovnbe | ~CF & ~ZF | 超过（无符号 ＞） |
| cmovae S，R | cmovnb | ~CF | 超过相等（无符号 ≥） |
| cmovb S，R | cmovnae | CF | 低于（无符号 ＜） |
| cmovbe S，R | cmovna | CF | ZF | 低于等于（无符号 ≤） |

条件传送指令与条件控制指令对比：



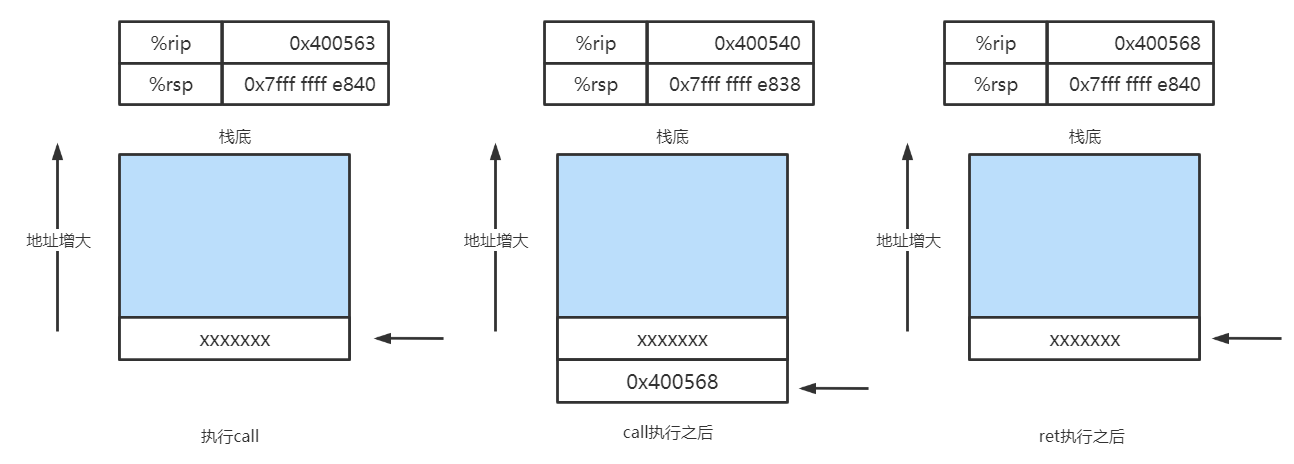
6、转移控制指令

注：在x86-64中末尾会加“q”，这是表示该版本是64位版本，语义没有改变

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| call Label | 1. 将call跳转的地址压入%rip中 2. 将call的下一条地址压入栈中 | 过程调用 |
| call \* Operand | 过程调用 |
| ret | 将栈中的返回地址弹入%rip中 | 从过程调用中返回 |

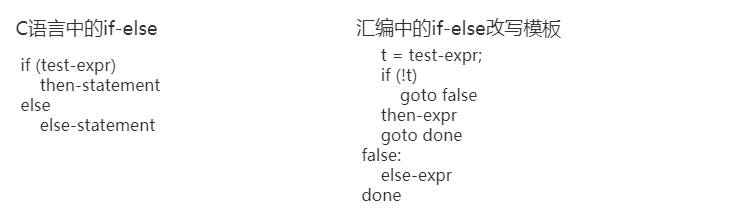
举例：

 400563：E8 D8 FF FF FF     callq     400540 <multstone>  
 400568：48 8B 54 24 08       mov     0x8(%rsp)，%rdx

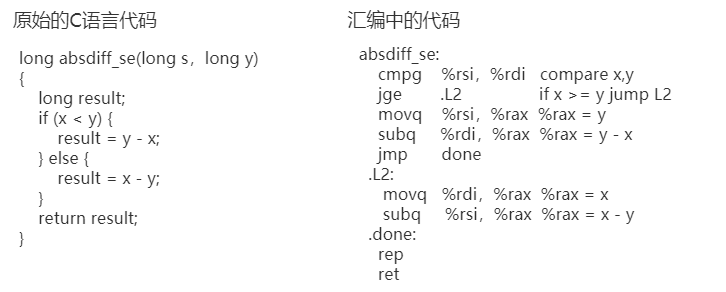


3.2.5 C语言翻译成汇编语言例子

1、if-else语句改写：



例子：



注：rep，repz，当ret指令通过跳转指令到达时，处理器不能正确预测ret指令的目的，因此这里的rep指令是一种空操作·

2、循环语句改写

①do-while循环：

|  |  |
| --- | --- |
| do-while通用形式 | 翻译成goto语句 |
| do{      body-statement  }while（test-expr） | loop      body-statement      t = test-expr      if （t）          goto loop； |

②while循环：

|  |  |  |
| --- | --- | --- |
| while通用形式 | 翻译成goto语句 | 当采用较高优化级别时 |
| while（test-expr）{      body-statement  } | goto test：  loop:  body-statement  test:      t = test-expr      if （t）          goto loop； | t = test-expr;  if（!t）goto done;  loop:  body-statement      t = test-expr      if （t）      goto loop；  done |

③for循环：

|  |  |  |
| --- | --- | --- |
| for循环通用形式 | while通用形式 | 翻译成goto语句 |
| for (init-expr;test-expr;  update-expr){  } | init-expr  while（test-expr）{  body-statement  update-statement  } | init-expr  goto test：  loop:  body-statement  update-statement  test:      t = test-expr      if （t）        goto loop； |

3、switch开关：使用跳转表

|  |  |
| --- | --- |
| switch C语言代码 | 翻译成汇编语句 |
| void switch\_eg(long x，long n，long \*dest)  {  long val = x;  switch(n){  case 100:  val \*= 13;  break;  case 102:  val += 10;  (Fail through)  case 103:  val += 11;  break;  case 104:  case 106:  val \*= val;  break;  default:  val = 0;  }  \*dest = val;  } | .section .rodata # .rodata表示只读数据  .align 8 # .align表示对齐操作，地址按8的倍数对齐  .L4: # 表示跳转代码的地址数组，.quad表示8字节  .quad .L3 # case 100  .quad .L8 # case 101 跳转到默认位置  .quad .L5 # case 102  .quad .L6 # case 103  .quad .L7 # case 104  .quad .L8 # case 105 跳转到默认位置  .quad .L7 # case 106  switch\_eg:  subq $100,%rsi # %rsi = n - 100  cmpq $6,%rsi # compare %rsi和6  ja .L8 # 无符号if %rsi > 6则跳转到 .L8位置  jmp \* .L4(,%rsi,8) # if %rsi ≤ 6 则跳转到 .L4 + 8\*%rsi的位置  # \*表明这是一个间接跳转  .L3: # case 100  leaq (%rdi,%rdi,2),%rax # %rax = 3\*x  leaq (%rdi,%rax,4),%rdi # %rdi = 4\*%rax + x  jmp .L2  .L5: # case 102  addq 10,%rdi # %rdi = x+10  .L6: # case 103  addq 11,%rdi # %rdi = x+11  jmp .L2  .L7: # case 104 or 106  imulq %rdi,%rdi # %rdi = x\*x  jmp .L2  .L8: # case default  movq $0,%rdi # %rdi = 0  .L2 # case 104 or 106  movq %rdi,(%rdx) # \* %rdx = val  ret |

**3.3 过程（转移控制指令详见3.2.4 6）★**

**3.3.1 参数传递**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 | %r9w | %r9w |
| 8 | %dil | %sil | %dl | %cl | %r8b | %r9b |

如果一个函数超过6个参数需要用栈来传递。

**3.3.2 被调用者保存**

寄存器%rbx,%rbp,%r12~%r15被划分为被调用者保存寄存器。当P调用Q时，Q必须保存这些值。

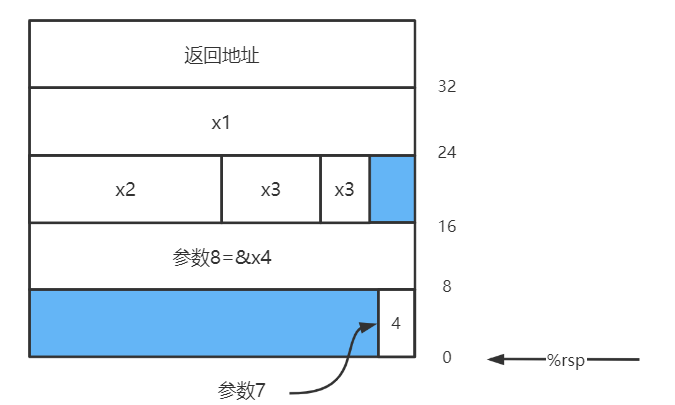
* 过程Q要么不更改这些寄存器的值
* 要么就把这些值放入栈中

**3.3.3、栈上的局部存储**

以下这些情况必须保存在内存中：

* 寄存器不够存放所有的本地数据
* 对一个局部变量使用“&”，
* 某些局部变量是数组或者结构体，需要通过引用访问

|  |  |
| --- | --- |
| call\_proc C语言代码 | 翻译成汇编语句 |
| long call\_proc()  {  long x1 =1; int x2 = 2;  short x3 = 3; char x4 = 4;  proc(x1,&x1,x2,&x2,x3,&x3,x4,&x4);  return (x1+x2)\*(x3-x4);  } | call\_proc:  subq $32,%rsp # 在栈上分配32字节的空间  movq $1,24(%rsp) # 在 %rsp+24 处存储x1  movq $2,20(%rsp) # 在 %rsp+20 处存储x2  movq $3,18(%rsp) # 在 %rsp+18 处存储x3  movq $4,17(%rsp) # 在 %rsp+17 处存储x4  leaq 17(%rsp),%rax # %rax = &x4  movq %rax,8(%rsp) # %rsp+8存储参数8  movq $4,(%rsp) # %rsp处存储参数7 x4  #因为通过寄存器只能传递6个参数，多余的参数需要保存在栈  leaq 18(%rsp),%r9 # 将 &x3保存在 %r9（第六个参数）  movq $3,%r8 # 将 3保存在 %r8（第五个参数）  leaq 20(%rsp),%rcx # 将 &x2保存在%rcx（第四个参数）  movq $2,%rdx # 将 2 保存在 %rdx（第三个参数）  leaq 24(%rsp),%rsi # 将 &x1 保存在%rsi（第二个参数）  movq $1,%rdi # 将 1传递给 %rdi（第一个参数）  call proc |

运行时栈数据： 

**3.4 数据结构**

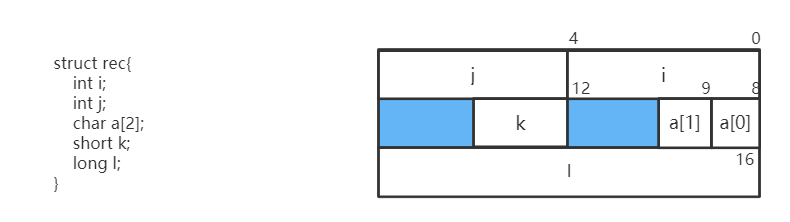
**3.4.1 数组的访问：通过地址访问**

假设数组 E 起始地址和索引 i 放在%rdx和%rcx，则相关操作如下：

|  |  |  |
| --- | --- | --- |
| 表达式 | 类型 | 汇编代码 |
| E | int \* | movq %rdx, %rax |
| E[0] | int | movq (%rdx), %rax |
| E[i] | int | movq (%rdx,%rcx,4), %rax |
| &E[2] | int \* | leaq 8(%rdx), %rax |
| E+i-1 | int \* | leaq -4(%rdx,%rcx,4), %rax |
| \*(E+i-3) | int | movq -12(%rdx,%rcx,4),%rax |
| &E[i] - i | long i | movq %rcx, %rax |

**3.4.2 结构(struct)**

* 结构体在栈上分配空间
* 元素的起始地址必须按照元素的大小倍数对齐



**3.4.3 联合(union)**

在union中，各成员变量共享一段内存空间，长度等于各成员中最长的长度，且要为最长元素类型的倍数。

例如：union { ​

int x; ​

char a[5];

} u

sizeof(u) = 8

**3.5 机器级程序中控制与数据相结合**

**3.5.1 指针**

* 每个指针都对应一个类型，void \* 表示通用指针
* 每个指针都有一个值，表示对应对象的地址
* “&”操作符创建指针，“\*”操作符引用指针
* 指针也可以指向函数，函数指针的值是该函数机器代码的第一条指令的地址

int func(int x, int *p);*

*int (*fp)(int , int \*);

fp = func;

然后调用这个指针来使用函数

int y = 1;

int result = fp(y,&y);

**3.5.2 缓冲区溢出**

1、缓存区溢出：局部变量和状态信息（保存的寄存器值和返回地址）存放在栈中，对越界的数组元素进行写操作会破坏存储在栈上的信息。

对抗缓存区溢出：

①栈随机化：栈的位置在每次程序运行时都有变化。

②栈破坏检测：在局部缓冲区与返回地址之间插入一个哨兵值，在恢复寄存器和函数返回时，检测这个值是否被更改过。★

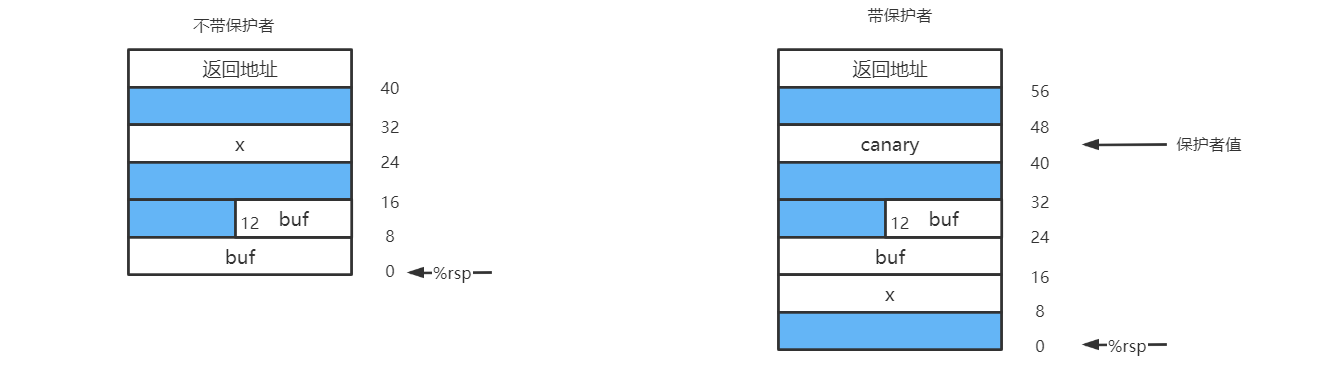
例如：函数intlen使用了带保护和不带保护者的编译版本

 int intlen(long x){  
     long v = x;  
     char buf[12];  
     iptoa(buf,&v);  
     return len(buf);  
 }

对应的汇编代码：

|  |  |
| --- | --- |
| 不带保护者 | 带保护者 |
| intlen:  subq $40,%rsp # 在栈上分配32字节  movq %rdi,24(%rsp) # %rsp+24存储参数x  leaq 24(%rsp),%rsi # 将 &x 保存在%rsi  movq %rsp,%rdi # 将buf地址保存在%rdi  call iptoa | intlen:  subq $56,%rsp # 在栈上分配56字节的空间  movq %fs:40,%rax  # %fs:40 从内存中获取金丝雀值  movq %rax,40(%rsp)  # 在 %rsp+40 处存储金丝雀值  xorl %eax,%eax # 将低位清0  movq %rdi,8(%rsp)# %rsp+8存储参数x  leaq 8(%rsp),%rsi  # 将&x保存在%rsi（第二个参数）  leaq 16(%rsp),%rdi  # 将 buf保存在%rdi（第一个参数）  call iptoa |

栈内存示意图：



③限制可执行代码区域：限制哪些内存区域存放可执行代码（使用硬件支持多种形式的内存保护）

**3.5.3 支持变长栈帧**

C语言中可能会出现变长数组，编译器无法预先知道具体值，不能预先在栈上分配具体值。因此可以用%rbp记录栈顶的值，然后用%rsp动态分配地址。

**3.6 浮点机器代码（了解）**

C语言中可能会出现变长数组，编译器无法预先知道具体值，不能预先在栈上分配具体值。因此可以用%rbp记录栈顶的值，然后用%rsp动态分配地址。

**四、处理器体系结构 ★★（重点，难点）**

**4.1 Y86-64指令集体系结构 ★★★**

**4.1.1 程序员可见状态**

1、寄存器文件 ★

|  |  |  |  |
| --- | --- | --- | --- |
| 数字 | 寄存器名字 | 数字 | 寄存器名字 |
| 0 | %rax 返回值 | 8 | %r8 第五个参数 |
| 1 | %rbx 被调用者保存 | 9 | %r9 第六个参数 |
| 2 | %rcx 第四个参数 | A | %r10 调用者保存 |
| 3 | %rdx 第三个参数 | B | %r11 调用者保存 |
| 4 | %rsp 栈顶地址 | C | %r12 被调用者保存 |
| 5 | %rbp 栈帧 | D | %r13 被调用者保存 |
| 6 | %rsi 第二个参数 | E | %r14 被调用者保存 |
| 7 | %rdi 第一个参数 | F | 无寄存器 |

2、条件码：OF，ZF，SF

3、PC：RIP

4、stat：标识程序是否正常运行

**4.1.2 Y86-64指令 ★★**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | | | | | | | |  |
| cmovXX | 2 fn | rA rB |  |  |  |  |  |  |  |  |
| call Dest | 8 0 | Dest | | | | | | | |  |
| ret | 9 0 |  |  |  |  |  |  |  |  |  |
| pushq rA | A 0 | rA F |  |  |  |  |  |  |  |  |
| pop rA | B 0 | rA F |  |  |  |  |  |  |  |  |

* 4个movq指令：irmovq，rrmovq，mrmovq，rmmovq。指令第一个字母表示源，第二个字母表示目的。i：立即数，r：寄存器，m：内存
* 4个OPq指令：addq，subq，andq，xorq。只对寄存器操作，同时还会对3个条件码ZF，SF，OF进行设置
* 7个跳转指令：jmp，jle，jl，je，jne，jge，jg，分支条件和x86-64一样
* 6个条件传送指令：cmovle，cmovl，cmove，cmovne，cmovge，cmovg
* call指令将返回地址入栈，然后跳到目的地址。ret从这样的调用中返回
* pushq和popq实现了入栈和出栈
* halt指令停止指令的执行，会将状态码设置成HLT

①每条指令的第一个字节高4位是代码部分，低4位是功能部分。

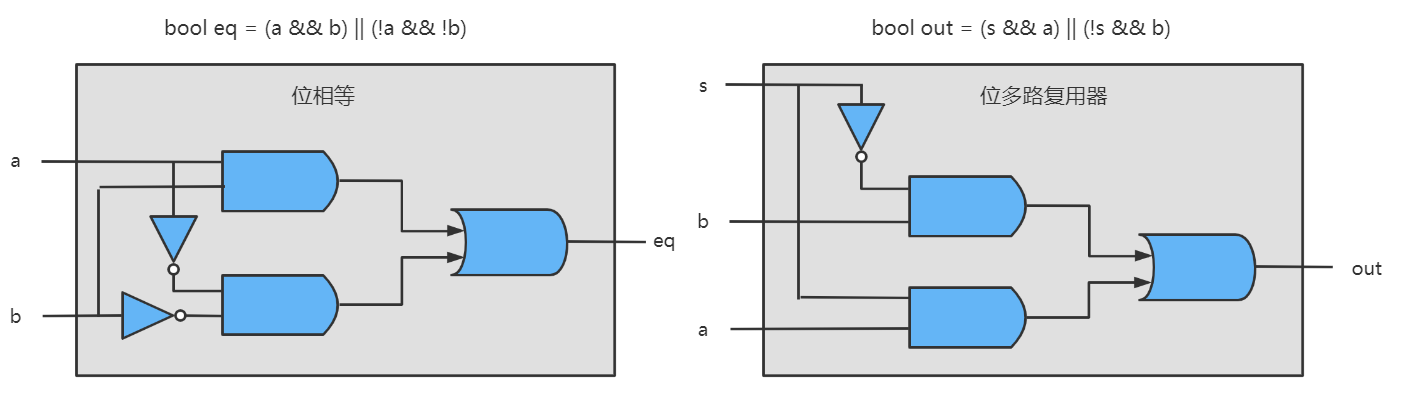
②只需要一个寄存器的会将另一个寄存器设置成0xF

**4.2 逻辑设计和硬件控制语言HCL**

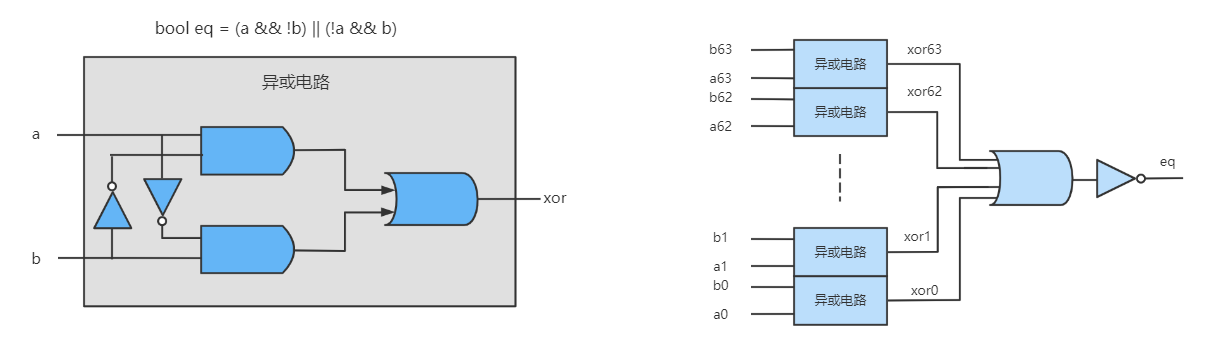
**4.2.1 组合电路和HCL布尔表达式**

将多个逻辑门组合成一个网，这些称为组合电路。如何构建这些网有限制：

* 每个逻辑门的输入必须连接到下述选项之一：1、一个系统输入 2、某个存储器单元的输出 3、某个逻辑门输出
* 两个或多个逻辑门的输出不能连接到一起
* 这个网必须是无环的



练习：使用异或电路设计一个64位字的相等电路。

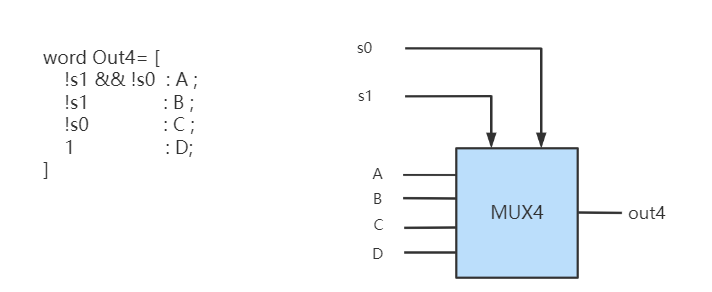


**4.2.2 多路复用函数**

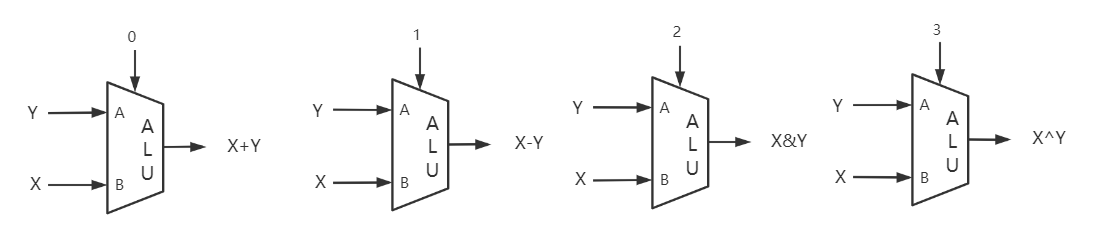


* 与C语言switch不同，不要求表达式互斥
* 遇到第一个表达式为1的情况就会被选中
* 最后一个默认为1，表示前面没选中就会选择这种情况

四路复用器：



★算术逻辑单元：



**4.3 Y-86顺序实现 ★★★**

**4.3.1 处理组成阶段 ★★**

* 取指阶段：从内存中读取指令字节，地址为PC的值。指令分为两个四位部分，icode和ifun，可能读出一个寄存器指示符，指明rA或rB，还可能取出一个四字节常数valC，并计算当前指令下一条指令的valP的值
* 译码阶段：从寄存器文件中最多读取两个操作数，rA和rB
* 执行阶段：根据ifun值执行，ALU执行对应的操作，
  + 计算内存有效引用地址，增加或减少栈指针，这些值称为valE。在此，也可能设置条件码
  + 检验条件码和传送条件
* 访存阶段：将计算的数据写入内存，或者从内存中读数据，读出的值为valM
* 写回阶段：将最多两个结果写回到寄存器文件
* 更新阶段：将PC设置成下一条指令的地址

1、OPq、rrmovq，irmovq

|  |  |  |  |
| --- | --- | --- | --- |
| 阶段 | OPq rA，rB | rrmovq rA，rB | irmovq V，rB |
| 取指 | icode:ifun ← M1[PC] | icode:ifun ← M1[PC] | icode:ifun ← M1[PC] |
| rA，rB ← M1[PC+1] | rA，rB ← M1[PC+1] | rB ← M1[PC+1] |
|  |  | valC ← M1[PC+2] |
| valP ← PC + 2 | valP ← PC + 2 | valP ← PC + 10 |
| 译码 | valA ← R[rA] | valA ← R[rA] |  |
| valB ← R[rB] |  |  |
| 执行 | valE ← valB - valA | valE ← 0 + valA | valE ← 0 + valC |
| set CC |  |  |
| 访存 |  |  |  |
| 写回 | R[rB] ← valE | R[rB] ← valE | R[rB] ← valE |
|  |  |  |
| 更新PC | PC ← valP | PC ← valP | PC ← valP |

2、rmmovq，mrmovq

|  |  |  |
| --- | --- | --- |
| 阶段 | rmmovq rA，D(rB) | mrmovq D(rB)，rA |
| 取指 | icode:ifun ← M1[PC] | icode:ifun ← M1[PC] |
| rA，rB ← M1[PC+1] | rA，rB ← M1[PC+1] |
| valC ← M8[PC + 2] | valC ← M8[PC+2] |
| valP ← PC + 10 | valP ← PC + 10 |
| 译码 | valA ← R[rA] |  |
| valB ← R[rB] | valB ← R[rB] |
| 执行 | valE ← valB + valC | valE ← valB+ valC |
|  |  |
| 访存 | M8[valE] ← valA | valM ← M8[valE] |
|  |  |
| 写回 |  | R[rA] ← valM |
| 更新PC | PC ← valP | PC ← valP |

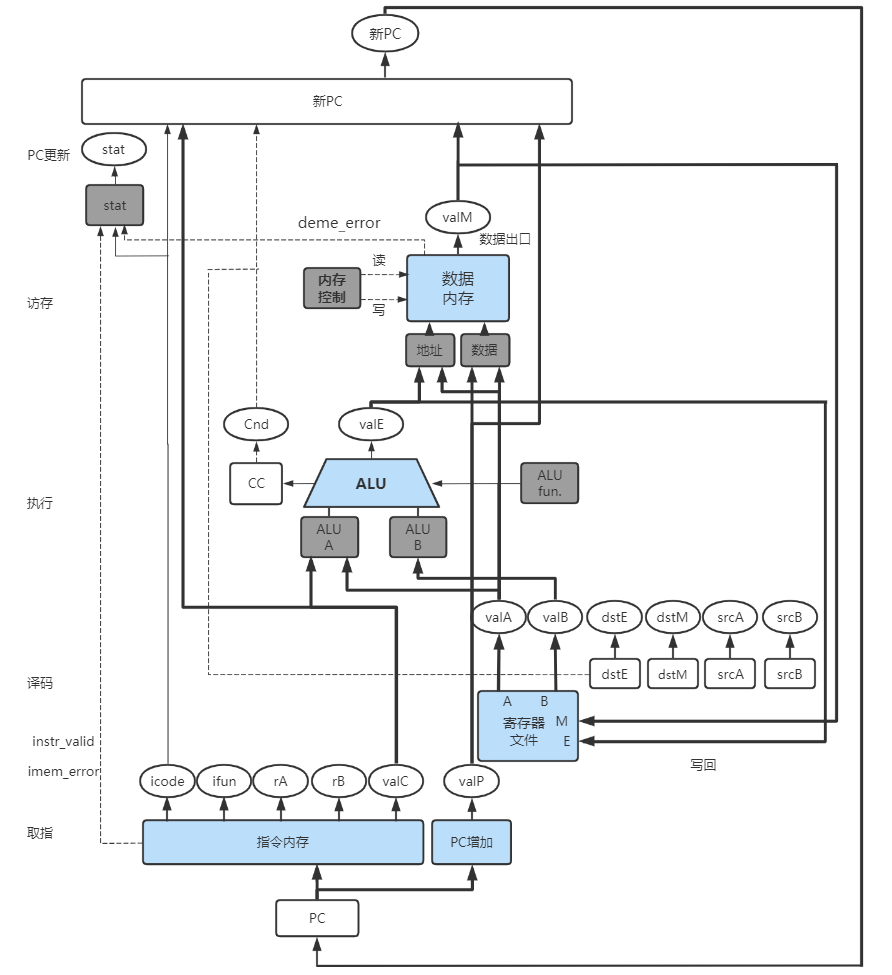
3、pushq，popq

|  |  |  |
| --- | --- | --- |
| 阶段 | pushq rA | popq rA |
| 取指 | icode:ifun ← M1[PC] | icode:ifun ← M1[PC] |
| rA，rB ← M1[PC+1] | rA，rB ← M1[PC+1] |
| valP ← PC + 2 | valP ← PC + 2 |
| 译码 | valA ← R[rA] | valA ← R[%rsp] |
| valB ← R[%rsp] | valB ← R[%rsp] |
| 执行 | valE ← valB - 8 | valE ← valB + 8 |
|  |  |
| 访存 | R[valE] ← valA | valM ← M8[valA] |
|  |  |
| 写回 | R[%rsp] ← valE | R[%rsp] ← valE |
|  | R[rA] ← valM |
| 更新PC | PC ← valP | PC ← valP |

4、控制转移指令：jxx，call，ret

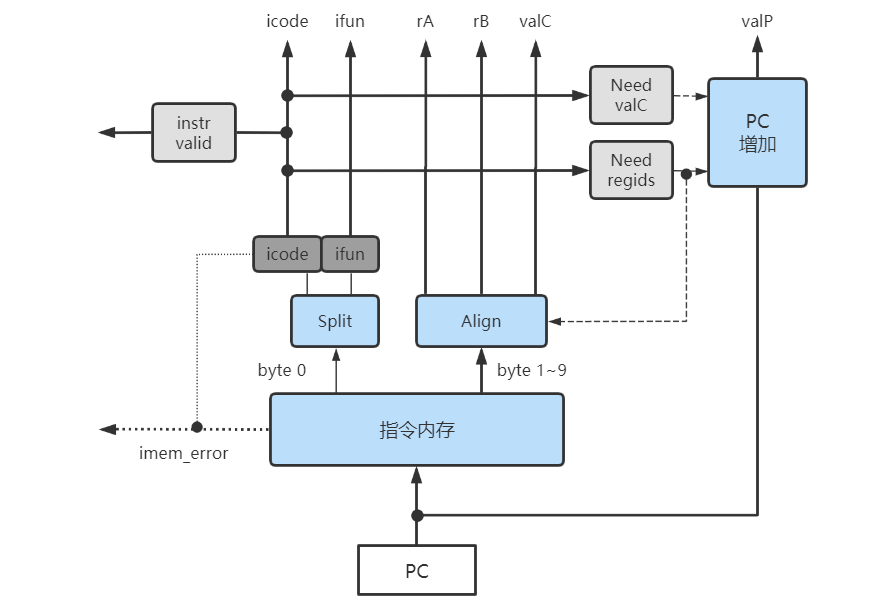
|  |  |  |  |
| --- | --- | --- | --- |
| 阶段 | jxx Dest | call Dest | ret |
| 取指 | icode:ifun ← M1[PC] | icode:ifun ← M1[PC] | icode:ifun ← M1[PC] |
| valC ← M1[PC+1] | valC ← M1[PC+1] |  |
| valP ← PC + 9 | valP ← PC + 9 | valP ← PC + 1 |
| 译码 |  | valB ← R[%rsp] | valA ← R[%rsp] |
|  |  | valB ← R[%rsp] |
| 执行 |  | valE ← valB - 8 | valE ← valB + 8 |
| Cnd← Cond(CC,ifun) |  |  |
| 访存 |  | M8(valE) ← valP | valM ← M8[valA] |
| 写回 |  | R[%rsp] ← valE | R[%rsp] ← valE |
| 更新PC | PC ← Cnd?valC : valP | PC ← valC | PC ← valM |

4.3.2 SEQ 硬件结构 ★★★



4.3.3 SEQ阶段的实现 ★★★★

1、取指阶段



①取出10个字节，第一个字节分为两个4位 icode : ifun。当地址不合法的时候发出 imem\_error信号

②根据icode指令发出三个信号：instr\_valid；Need valC；Need regids

* instr valid用来发现不合法的指令
* Need regids 发现这个指令包括寄存器指示符？
* Need valC 发现这个指令包括一个常数吗？

③当指令发生越界时，instr\_valid，imem\_error在访存阶段产生状态码

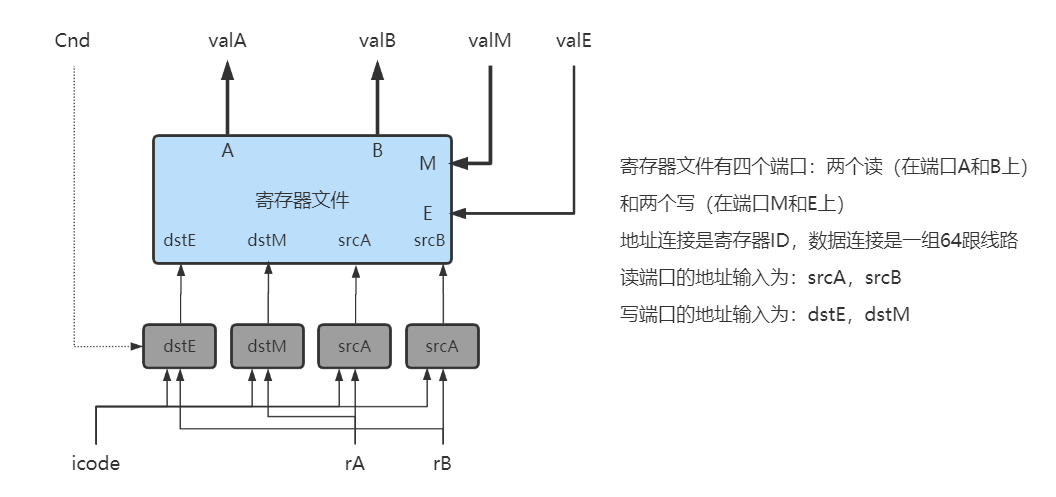
★对应信号的HCL表达式

|  |  |
| --- | --- |
| bool Need\_regids =  icode in { IOPQ，IRRMOVQ，IIRMOVQ，IRMMOVQ，IMRMOVQ，IPOPQ，IPUSHQ } | bool Need\_valC =  icode in { IIRMOVQ，IRMMOVQ，IMRMOVQ，IJXX，ICALL} |

注：PC增加valP根据当前PC值，need\_regids、need\_valC信号决定，假设当前PC值为 p，need\_regids 为r，need\_valC 为i

valP = p + 1 + r + 8i

2、译码和写回阶段

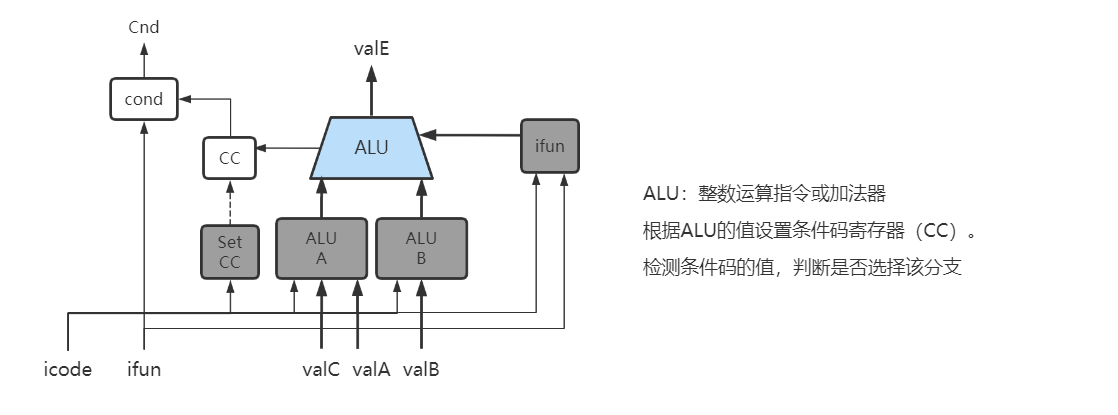


SEQ的译码和写回阶段。指令字段译码，产生寄存器文件使用的四个地址的寄存器标识符。从寄存器文件中读出的值为valA，valB，写回值valM，valE作为写操作的数据。

对应端口的HCL描述：

|  |  |
| --- | --- |
| word srcA = [  icode in {IOPQ，IRRMOVQ，IRMMOVQ，IPUSHQ}：rA；  icode in {IPOPQ，IRET}：RRSP；  1 ：RNONE；  ] | word srcB = [  icode in {IOPQ，IRMOVQ，IMRMOVQ }：rB；  ​icode in {IPUSHQ，ICALL，IPOPQ，IRET}：RRSP；  ​1：RNONE；  ] |
| word dstE = [  icode in {IRRMOVQ，IIRMOVQ，IOPQ}：rB；  ​icode in {IPUSHQ，ICALL，IPOPQ，IRET}：RRSP；  ​1：RNONE；  ] | word dstM = [  icode in {IPOPQ，IMRMOVQ}：rA；  ​1：RNONE；  ] |

3、执行阶段

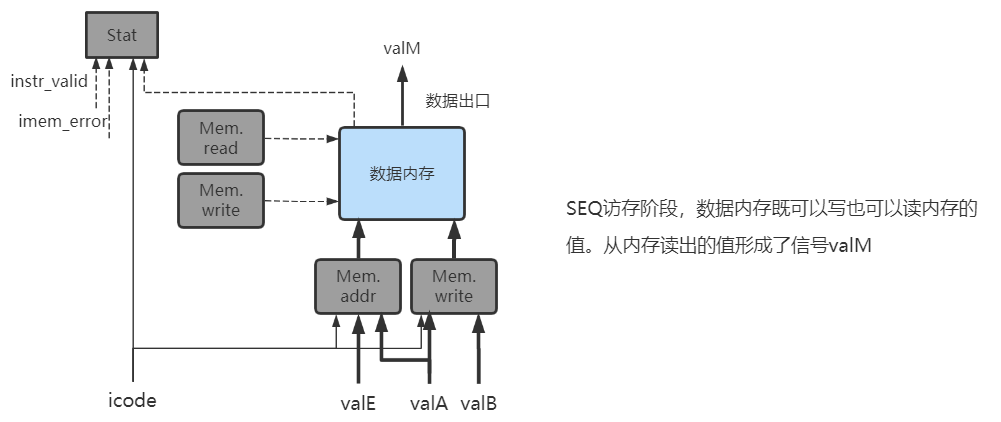


ALU输出是valE，这个单元的功能根据ifun设置：ADD，SUB，AND，EXCLUSIVE-OR

ALU端口HCL描述：

|  |  |
| --- | --- |
| word aluA = [  icode in {IOPQ，IRRMOVQ }：valA；  icode in {IPOPQ，IRET}：valC；  icode in {ICALL，IPUSHQ}：-8；  icode in {IRET，IPOPQ}：8；  # 其他指令不需要ALU  ] | word aluB = [  icode in {IOPQ，IRROVQ，IRMMOVQ，IMRMOVQ，IRET，ICALL，IPOPQ，IPUSHQ，  }：valB；  icode in {IRRMOVQ，IIRMOVQ}：0；  ​# 其他指令不需要ALU  ] |
| word alufun = [  icode == IOPQ：ifun；  ​1：ALUADD；  ] | bool set\_cc = icode in {IOPQ}; |
| 增加Cnd码后实现cmovxx指令：word dstE = [  icode in {IRRMOVQ } && Cnd：rB；  icode in { IIRMOVQ，IOPQ }：rB；  ​icode in {IPUSHQ，ICALL，IPOPQ，IRET}：RRSP；  ​1：RNONE；  ] | |

4、访存阶段



访存阶段的任务是或者写程序数据，两个控制块产生内存地址和内存输入数据的值，两个块产生表明应该执行读操作还是写操作的控制信号。访存最后阶段是根据取值阶段产生的icode，imem\_error，instr\_valid，dmem\_error来计算stat值

|  |  |
| --- | --- |
| word mem\_addr= [  icode in { IRMMOVQ，IMRMOVQ，IPUSHQ，ICALL}：valE；  icode in {IPOPQ，IRET}：valA；  # 其他指令不需要内存地址  ] | word mem\_data = [  icode in {ICALL }：valP；  ​icode in {IPUSHQ，IRMMOVQ }：valA；  ​# 其他指令不需要内存数据；  ] |
| bool mem\_read = icode in {IMRMOVQ，IPOPQ，IRET} | bool mem\_write = icode in {IRMMOVQ,ICALL,IPUSHQ} |
| word stat = [  imem\_error || dmem\_error ：SADR；  !instr\_valid ：SINS；  icode == IHALT ：SHLT；  1 ：SAOK；  ] | |

5、更新PC



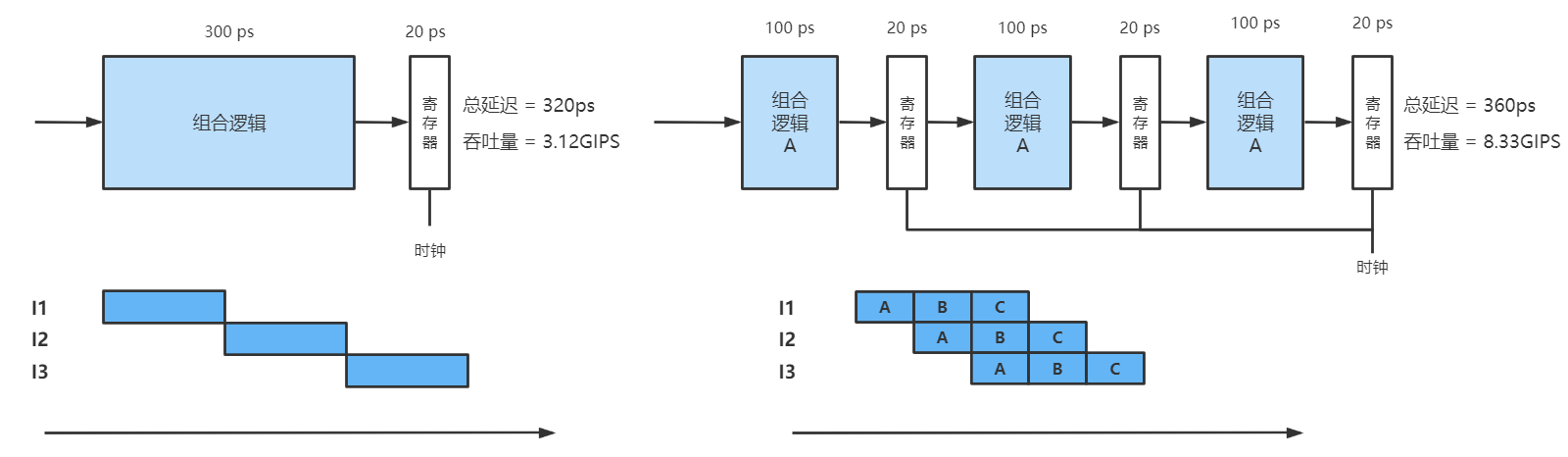
HCL描述：

|  |
| --- |
| word new\_pc = [  icode == ICALL ：valC；  icode == IJXX ：valC；  icode == IRET ：valM；  1：valP；  ] |

**4.4 流水线通用原理**

**4.4.1 计算流水线吞吐量**

1、流水线的重要特性是提高了系统的吞吐量



非流水线吞吐量：

流水线吞吐量：

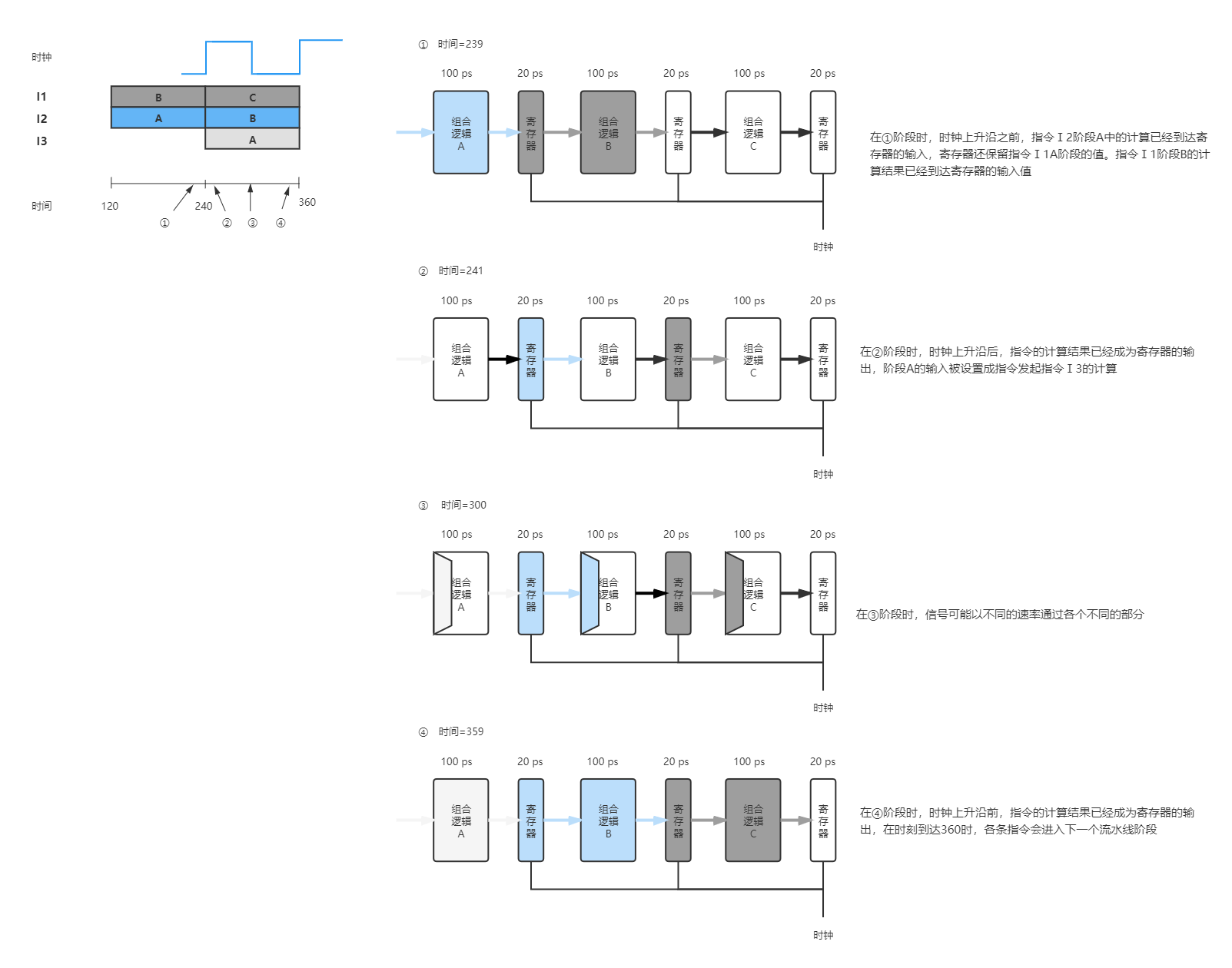
2、跟踪时间240~360之间的电路活动

在①阶段时，时钟上升沿之前，指令Ⅰ2阶段A中的计算已经到达寄存器的输入，寄存器还保留指令Ⅰ1A阶段的值。指令Ⅰ1阶段B的计算结果已经到达寄存器的输入值。

在②阶段时，时钟上升沿后，指令的计算结果已经成为寄存器的输出，阶段A的输入被设置成发起指令Ⅰ3的计算

在③阶段时，信号可能以不同的速率通过各个不同的部分

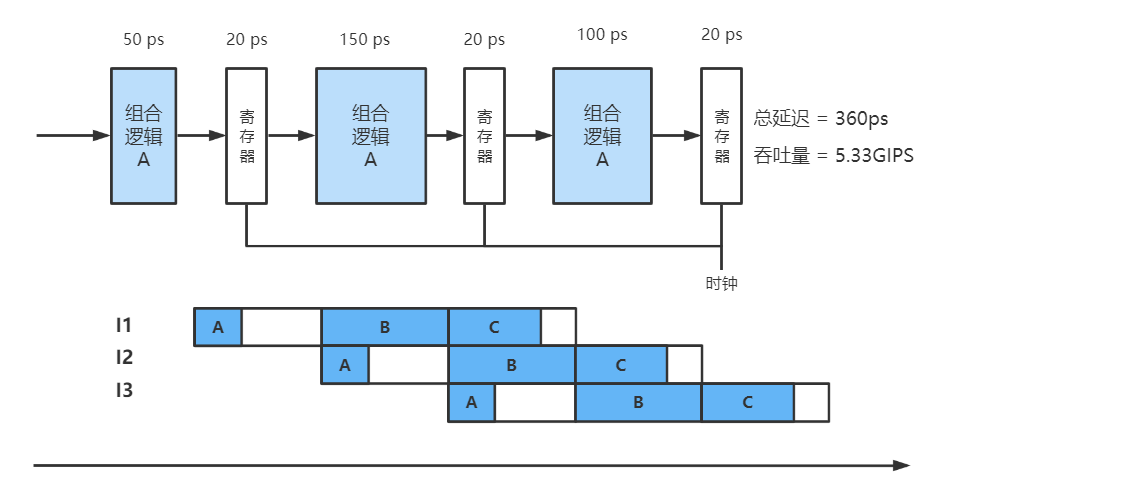
在④阶段时，时钟上升沿前，指令的计算已经成为寄存器的输出，在时刻到达360时，各条指令会进入下一个流水线阶段



**4.4.2 流水线的局限性**

1、不一致的划分

依然可以将上面计算划分为三阶段，不过依次为50ps到150ps不等，通过所有流水线的延迟依然为300ps。运行的时间由最慢的阶段限制，运行时间图表明，每个A阶段都会处于一段空闲时间，B阶段会一直处于活跃。因此必须设置时钟周期为150+20 = 170ps，得到的吞吐量为5.33GIPS



2、流水线过深，收益反而下降

将上述计算过程划分为6阶段，每阶段50ps，将时钟周期设置为70ps，则吞吐量为14.29GIPS，性能提高了14.29/8.33 = 1.71，将计算时钟缩短了两倍，但是由于流水线寄存器的延迟，吞吐量并没有加倍。同时延迟占到了整个时钟周期的120/420 = 28.6%。

#### 4.4.3 流水线性能分析

通过计算一条指令所需要的平均周期数的估计值，来量化处罚对整体性能的影响，这种衡量方法称为CPI。假设一个程序处理了Ci条指令和Cb个气泡，则这个程序的CPI：

因为只有三种指令会导致气泡的产生，因此这里可以将处罚项分成三个部分：

对CPI的计算，使用下面这组频率：

* 加载指令占所有指令的25%，其中20%会导致加载/使用冒险
* 条件分支指令占所有执行指令的20%。其中60%会选择分支，40%不选择分支
* 返回指令占所有执行指令的2%

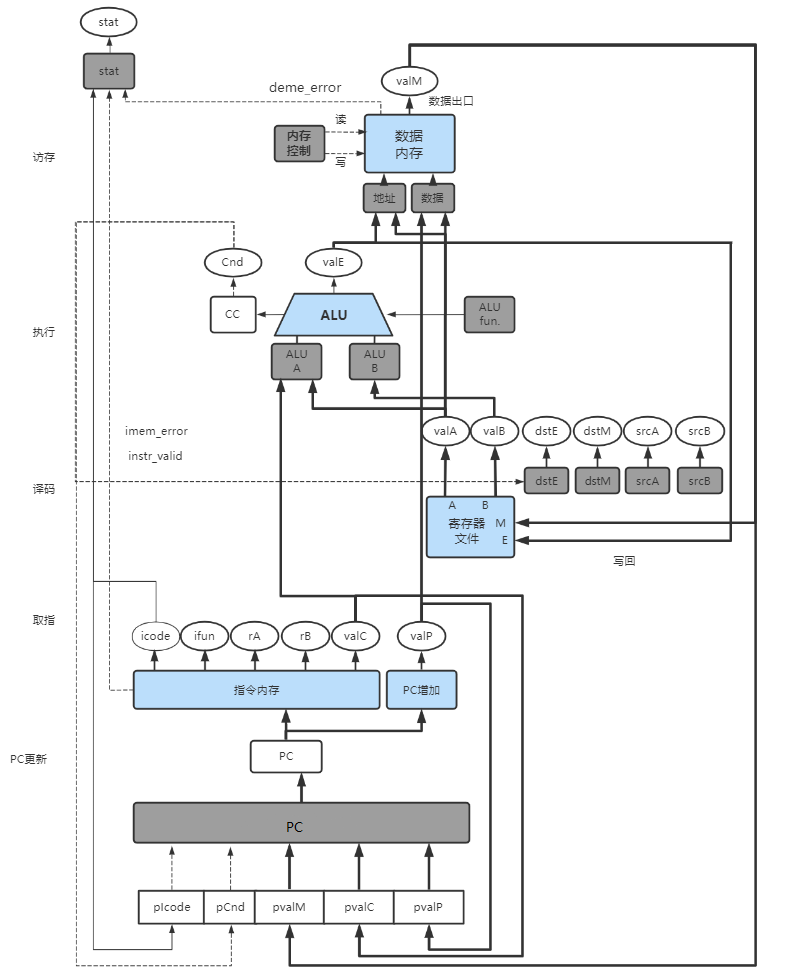
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 原因 | 名称 | 指令频率 | 条件频率 | 气泡 | 乘积 |
| 加载/使用 | lp | 0.25 | 0.20 | 1 | 0.05 |
| 预测错误 | mp | 0.20 | 0.40 | 2 | 0.16 |
| 返回 | rp | 0.02 | 1 | 3 | 0.06 |
| 总处罚 |  |  |  |  | 0.27 |

因此得到的CPI为1.27，可以发现分支预测错误占比较高。

**4.5 Y86-64的流水线实现 ★★★**

#### 4.5.1 SEQ+：重新安排计算阶段

移动PC阶段，使它在逻辑时钟开始时，计算当前指令的PC值。在SEQ中PC的指令根据当前周期内计算的信号值得出，而在SEQ+中，PC的计算是通过上一条指令的信号值给出，同时将寄存器标号改为“pIcode”，“pCnd”，“pvalM”，“pvalC”，“pvalP”



#### 4.5.2 插入流水线寄存器

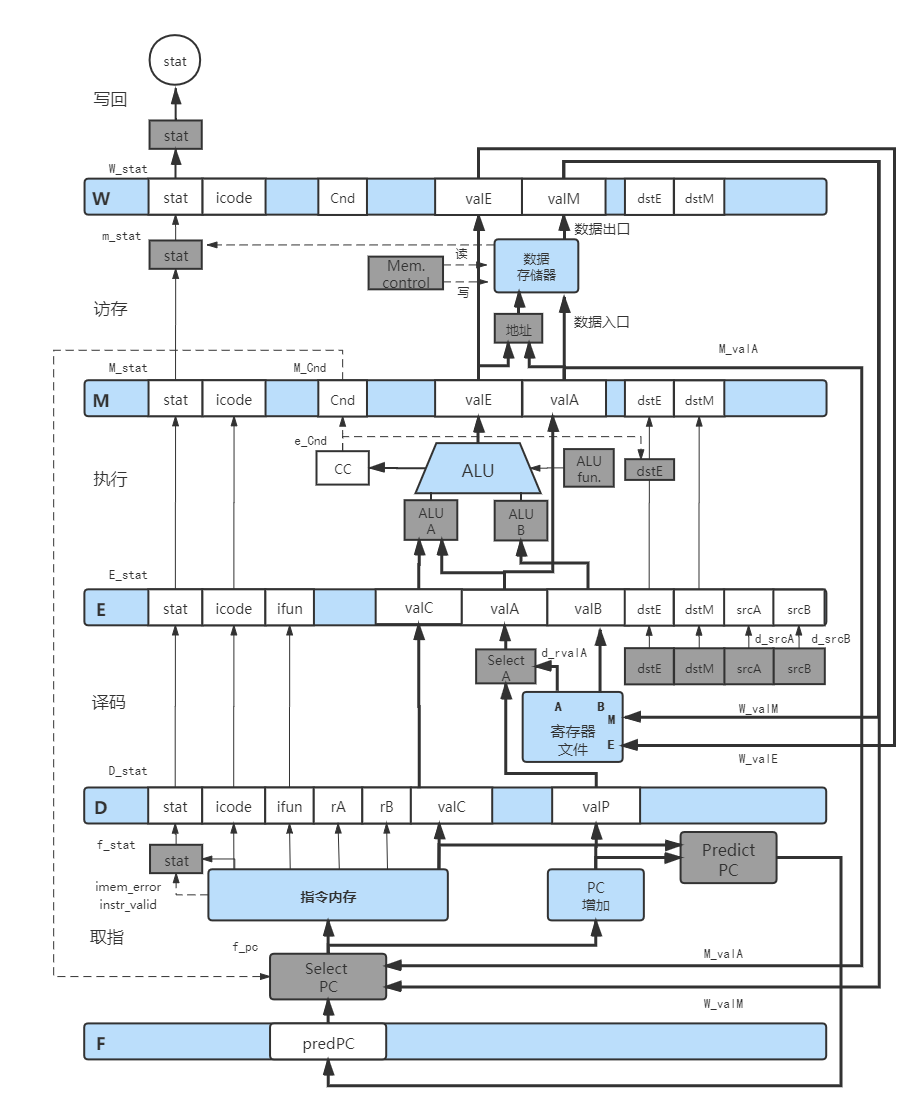
F：保存程序计数器的预测值

D：位于取指和译码阶段之间。保存关于最新取出的指令的信息，即将由译码阶段进行处理

E：位于译码和执行阶段之间。保存关于最新译码的指令和从寄存器文件读出的值的信息，即将由执行阶段进行处理

M：位于执行和访存阶段之间。保存最新执行的结果，还保存用于处理条件转移的分支条件和分支目标的信息

W：位于访存和写回阶段之间。将计算出来的结果写回寄存器文件中，如果是ret指令还需要向PC提供返回地址



旁注：信号M\_stat和m\_stat的差别

在命名系统中，大写的前缀“D”、“E”，“M”，“W”表示的是==流水线==寄存器，小写字母“e”，“m”，“w”表示的是==流水线阶段==寄存器

“Select A”：这个块会从寄存器文件A端口中读出的值和流水线寄存器DvalP中选择一个，只有在执行call和jxx指令不跳转时才会在执行阶段用到valP，因此合并这两个信号，减少流水线寄存器的状态数量。这样就减少了SEQ+中标号为==数据==的硬件结构

#### 4.5.3 预测下一个PC

1、call和jxx指令

* 对于call和jmp指令，下一条指令是valC
* 对于条件跳转来说下一条指令要么是valP，要么是valC，在这里PIPE一中总是预测选择了条件分支，即预测新PC值是valC。

分支预测策略：

在流水线中，我们总是选择条件分支的预测策略。研究表明这个策略的成功率大约为60%，则从不选择（NT）的成功率为40%。

反向选择、从不正向选择（BTFNT）：当分支地址比下一条地址低时就预测条件分支，而分支地址比较高时就预测不选择。这成功率在65%左右

2、ret指令

ret指令会在访存阶段才知道具体要返回的地址，因此流水线不会对这个地址进行预测。

预测栈的返回地址：

对大多数程序来说，预测返回值很容易，因为过程调用和返回总是成对出现的。大多数函数调用会返回到调用后的那条指令。高性能处理器运用了这个特性，在取指单元中放入一个硬件栈，每次执行过程调用指令时，将返回值压入栈中。当取出一个返回指令时就弹出一个地址，作为预测的返回值。这个硬件栈对程序员不可见。

3、Predict PC，Select PC

标号为“Predict PC”的块会从PC增加器中计算的valP和取出指令中得到的valC中进行选择。这个值存放在流水线寄存器F中作为程序计数器的预测值。

标号为“Select PC”的块会从三个值中选择一个作为指令内存的地址：预测的PC值（这里实际为valC），条件跳转不选择分支的valP值（这个存放在M\_valA中），ret指令到达W阶段时的返回值（这个值存储在W\_valM中）。

#### 4.5.4 流水线冒险

数据相关：下一条指令需要用到这一条指令的计算结果

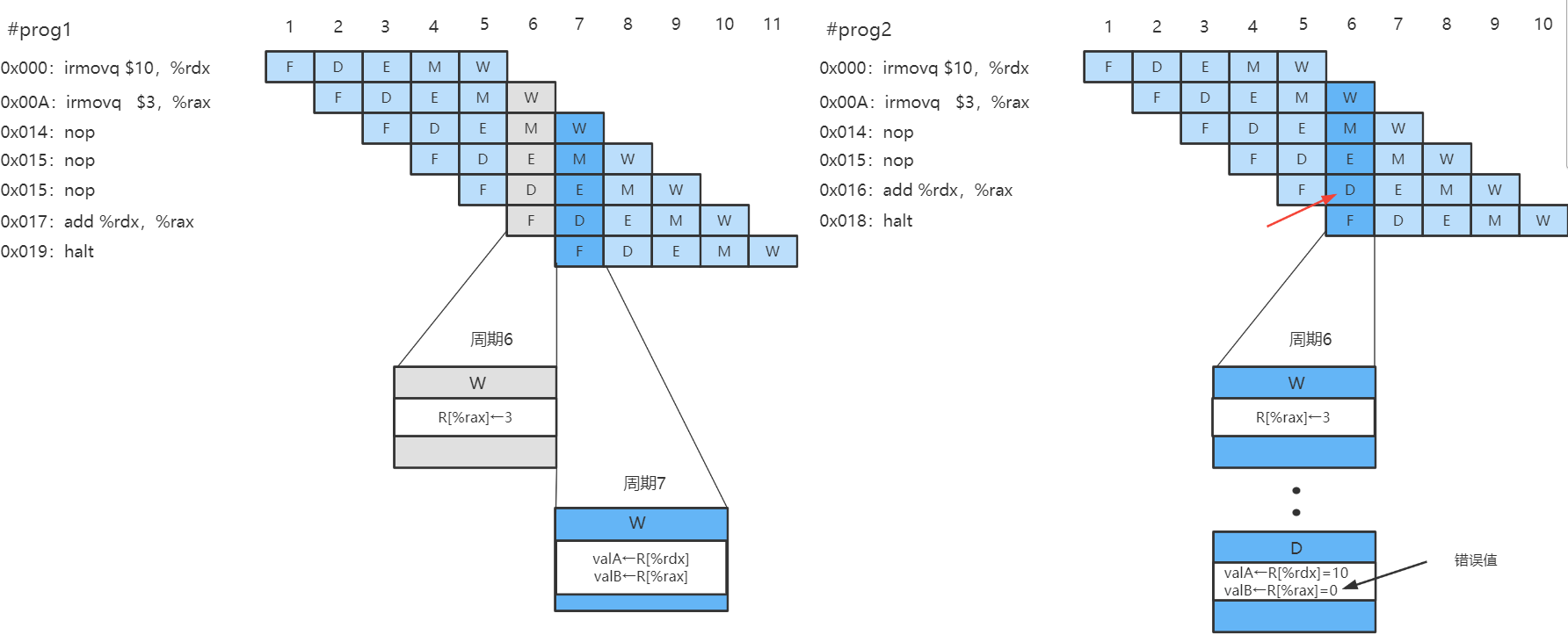
控制相关：一条指令要确定下一条指令的位置，例如在执行跳转、调用或返回指令时。

这些相关可能会导致流水线产生计算错误，与相关类似，冒险也分成两类。数据冒险和控制冒险。

下面是描述PIPE一处理prog1的情况：

prog1中在周期7初始的时钟上升沿时就将值写入%rax中，在周期7译码阶段开始时就可以获取正确的值。这是由于3条nop指令造成的延迟使得流水线没有发生数据冒险

prog2中在周期6初始的时钟上升沿时%rax的值为0，这是因为指令2的写回阶段还没有结束，valE值并没有写入寄存器文件中，因此在周期6时钟上升沿读取的是默认%rax值，由于前面指令未对%rax进行更改，因此为0。



数据冒险类型：

当一条指令更新，后面指令会读到的那些程序状态值时，就会出现数据冒险。

程序寄存器：寄存器文件的读写在不同阶段进行时，就有可能出现

程序计数器：更新和读取程序计数器之间的冲突导致了控制冒险

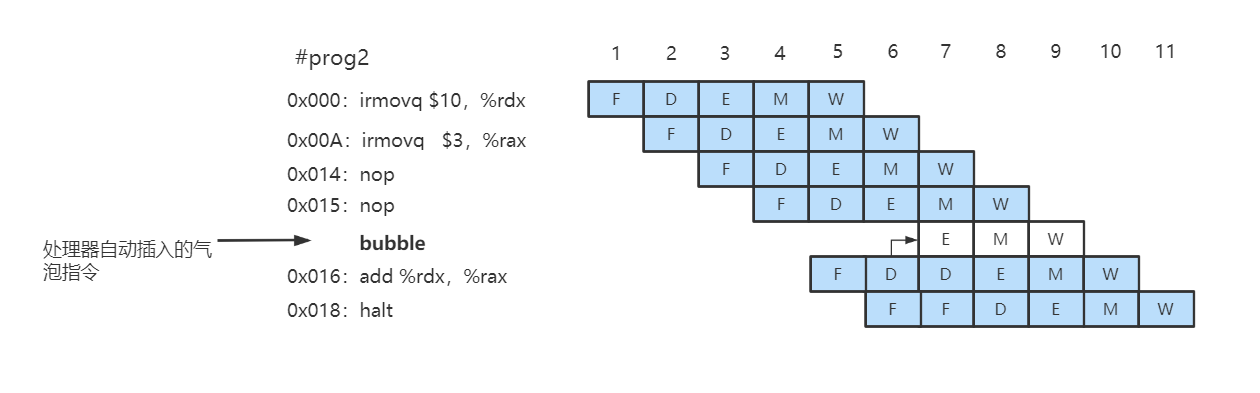
内存：假设程序不会自我修改。在一条读指令到达这个阶段前，前面写内存的指令完成就会发生数据冒险

条件码寄存器：这里不会发生数据冒险

状态码寄存器：每条指令都与一个状态码寄存器相关，因此不会发生数据冒险

1、用暂停来避免数据冒险

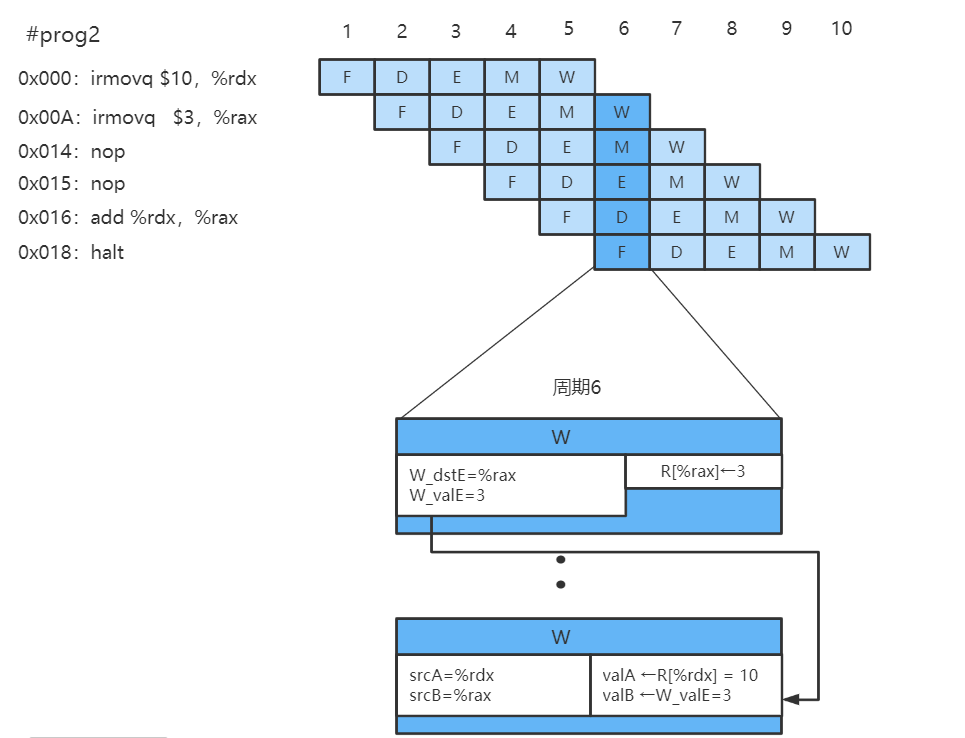
暂停时，处理器会停止流水线中一条或多条指令，直到冒险条件不再满足。暂停技术就是让一组指令阻塞在它们所处在的阶段，而允许前面运行的指令继续通过流水线，类似于自动产生的nop指令。



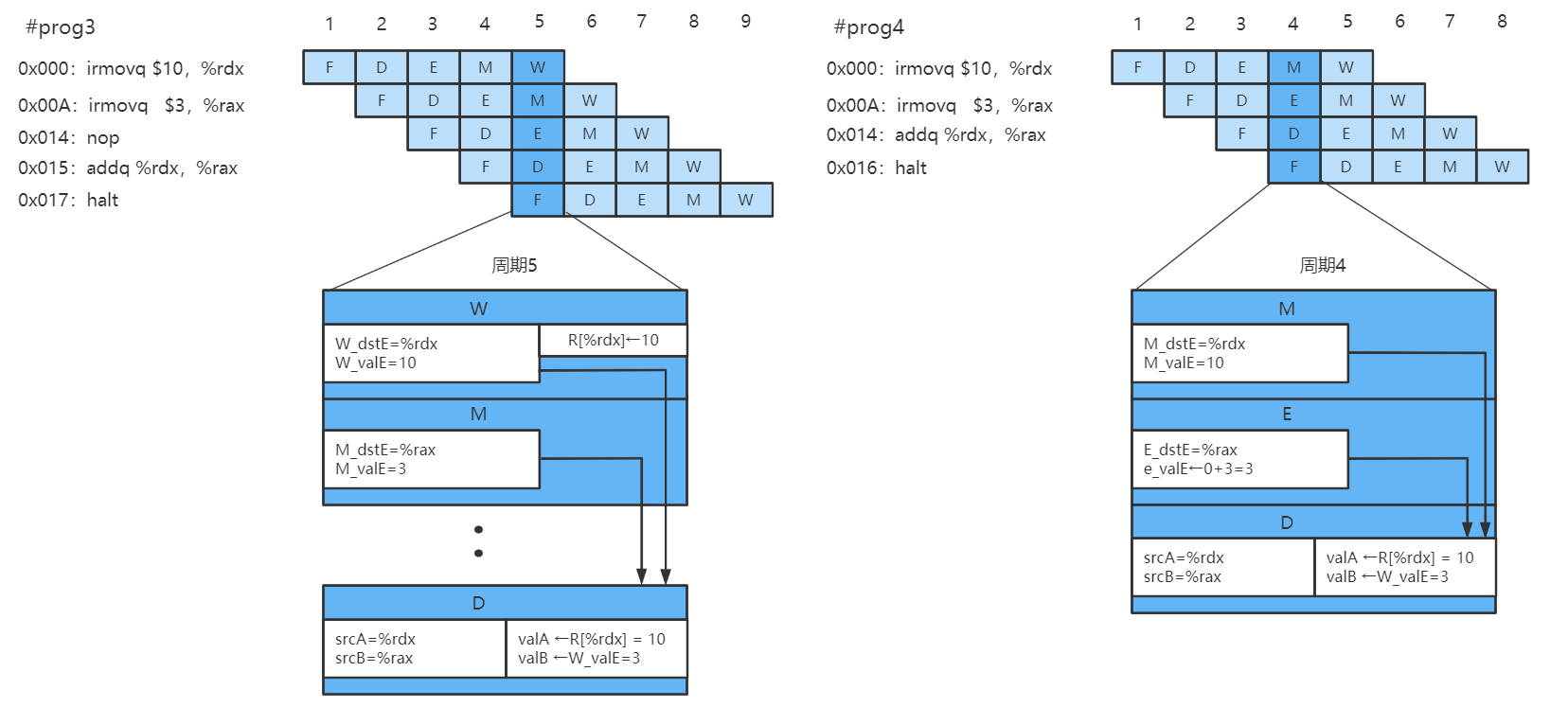
当插入bubble后，add指令一直处于译码阶段，halt指令一直重复取指令阶段

2、用转发来避免数据冒险

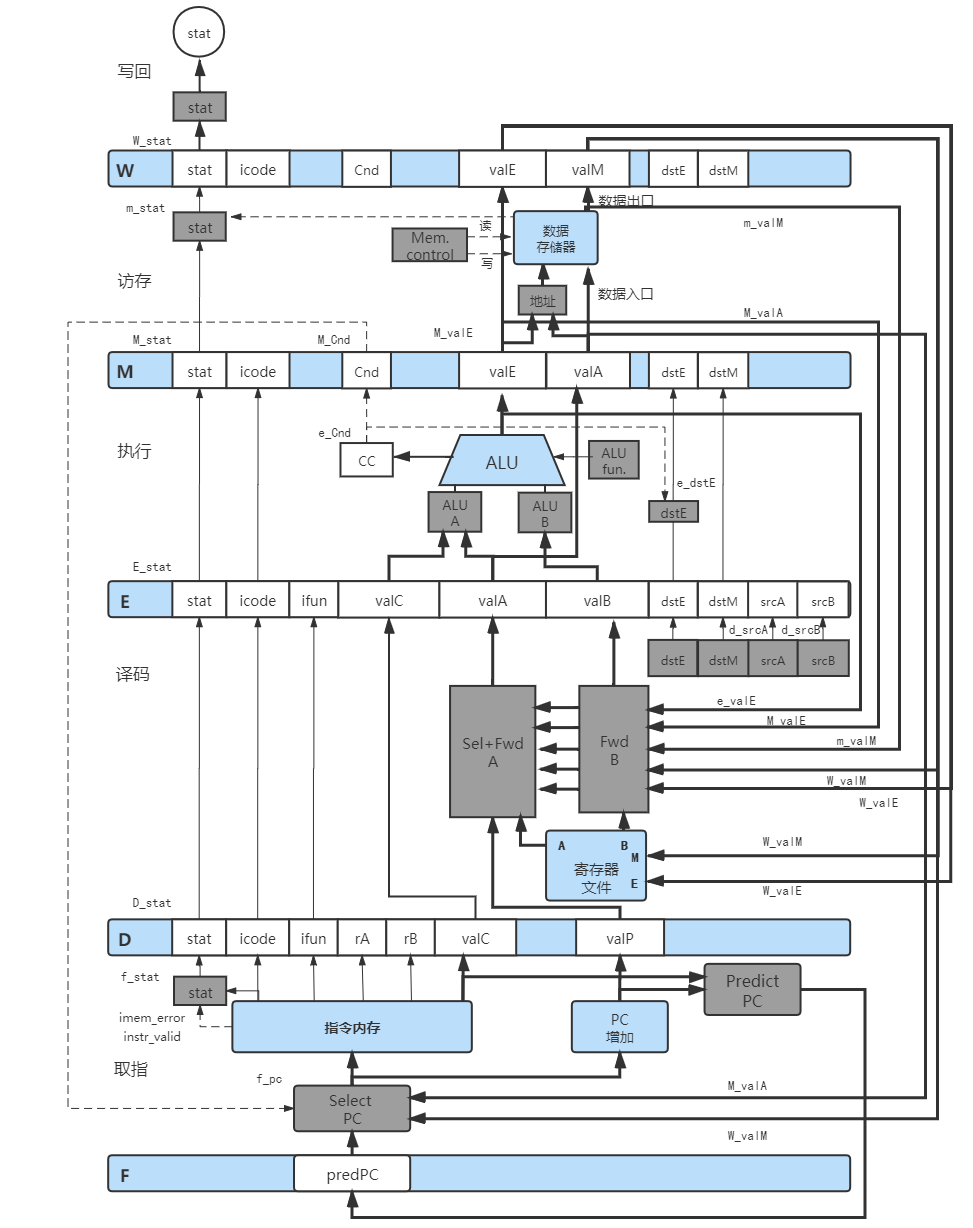
将结果直接从一个流水线阶段传到较早阶段的技术称为数据转发，或称为旁路技术。



在译码阶段发现，寄存器%rax是valB的值，而在dstE处有一个对%rax未进行的写。因此，这里只需要将W\_valE作为操作数valB的值，就能避免数据冒险。



当访存阶段和执行阶段有数据暂停时也可以使用转发技术，避免程序的暂停，例如prog4。译码阶段访存阶段有对%rdx未进行的写，执行阶段ALU正计算的值稍后也会写入%rax，因此可以使用转发技术，对访存阶段的值M\_valE作为操作数A，执行阶段的值e\_valE作为操作数B。根据以上分析，获得最终的PIPE硬件结构：

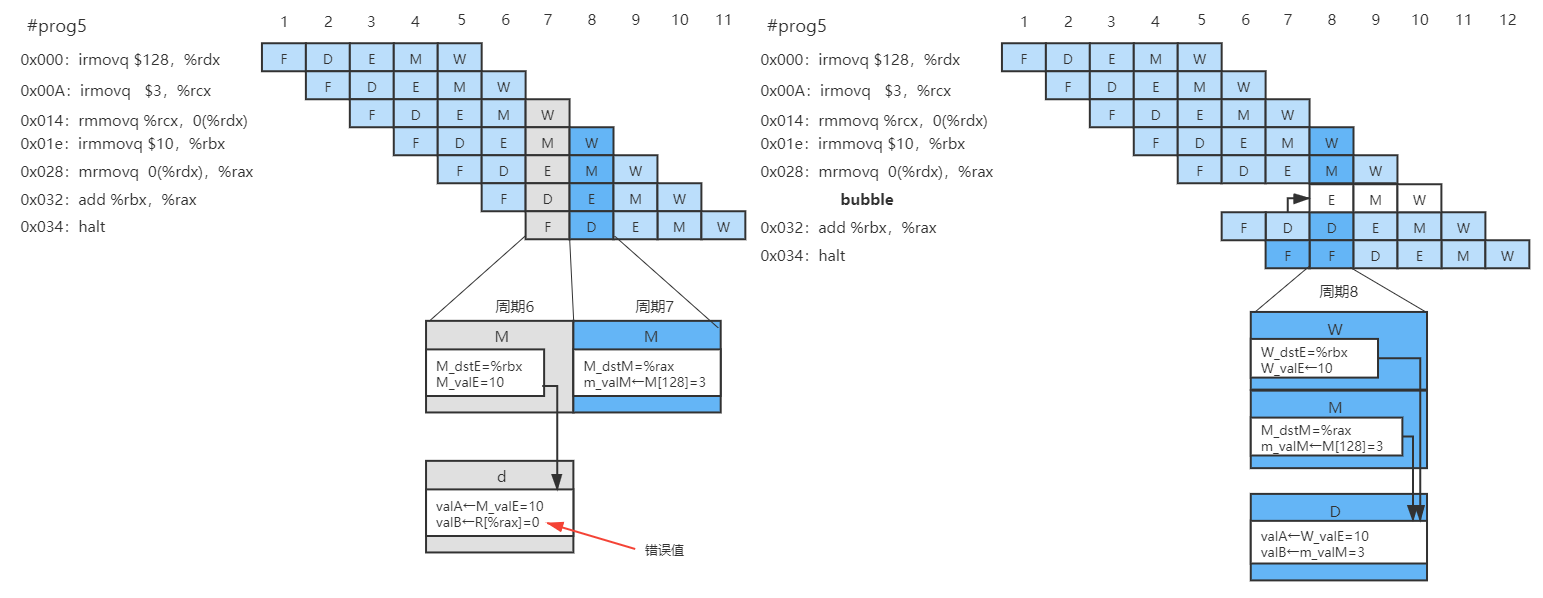


Sel+FwdA：这个块是SelectA+FwdA的结合，它允许从寄存器端口A读出的值，流水线寄存器EvalP已增加的值，某个转发过来的值

FwdB：这个块实现是源操作数valB的转发逻辑。

3、加载/使用数据冒险

这类数据冒险不能单纯的用转发解决，因为内存的读在流水线发生的比较晚。这里可以将暂停+转发结合起来使用。



4、避免控制冒险

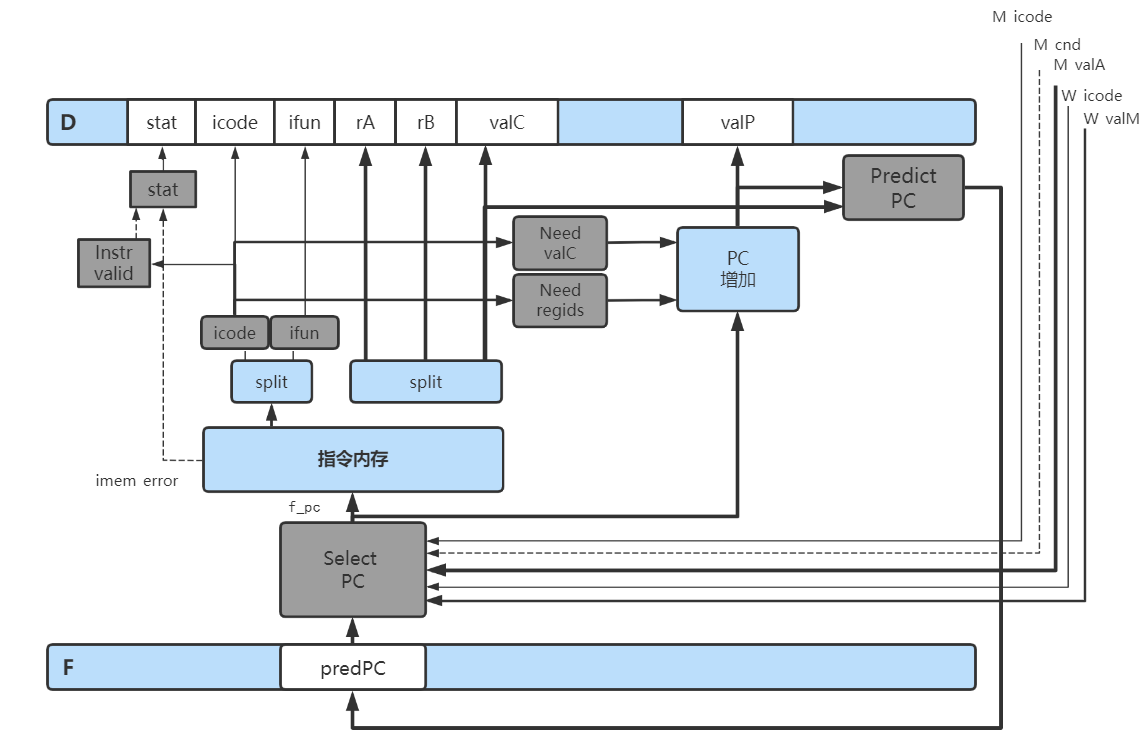
这类数据冒险不能单纯的用转发解决，因为内存的读在流水线发生的比较晚。这里可以将暂停+转发结合起来使用。

异常处理：

当流水线中有一个或多个阶段出现异常时，会禁止流水线中后面的指令更新程序员可见的状态，异常事件不会对流水线中的指令流有任何影响，直到异常指令到达最后的流水线状态。因为指令到达写回阶段的顺序与它们在非流水化的处理器执行顺序相同，所以可以保证第一条遇到异常的指令第一个到达写回阶段，此时程序会执行停止，流水中W状态码会被记录

#### 4.5.5 PIPE各阶段的实现

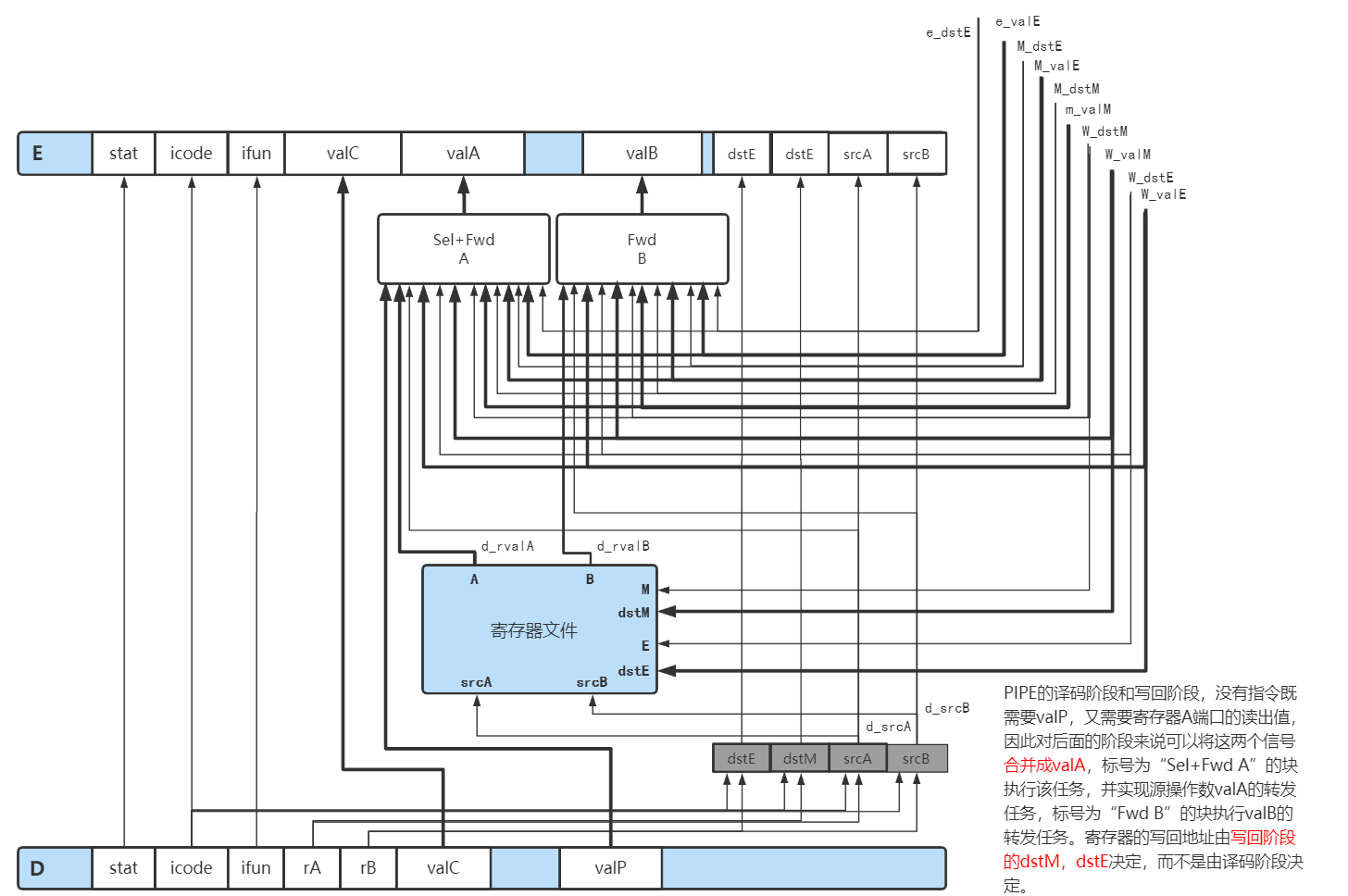
1、PC选择和取指阶段



在PIPE中大写开头表示源于哪里，比如D源于译码阶段流水线寄存器，小写字母表示结果值，表明它是来自哪里的，比如d\_来自译码阶段

|  |  |
| --- | --- |
| SEQ+中  word srcA = [  icode in {IOPQ，IRRMOVQ，IRMMOVQ，IPUSHQ}：rA；  icode in {IPOPQ，IRET}：RRSP；  1 ：RNONE；  ] | PIPE中：  word srcA= [  D\_icode in {IOPQ，IRMMOVQ，IRRMOVQ，IPUSHQ }：rA；  ​D\_icode in {IPOPQ，IRET}：RRSP；  ​1：RNONE；  ] |
| word f\_pc = [  #使用不选择条件分支默认的valP存放在访存阶段中的M\_valM中  M\_icode == IJXX && !Cnd：M\_valA；  ​W\_icode == IRET：W\_valM；#RET指令使用写回阶段内存的valM  ​1：F\_predPC；#其他情况使用预测的PC  ] | word F\_predPC = [  icode in {JXX，ICALL}：valC；  ​1：valP；  ] |
| word f\_stat = [  imem\_error：SADR；  ​!instr\_valid：SINS；  f\_icode == IHALT：SHLT；  1：SAOK；  ] | |

2、译码和写回阶段



合并信号valA和valP的依据是：只有call和jxx指令在后面阶段会使用valP，而这些指令不需要从寄存器A端口读出的值，这个选择是有译码阶段的icode控制的。当信号D\_icode与call和jxx指令匹配时，这个块就会把D\_valP作为它的输出。

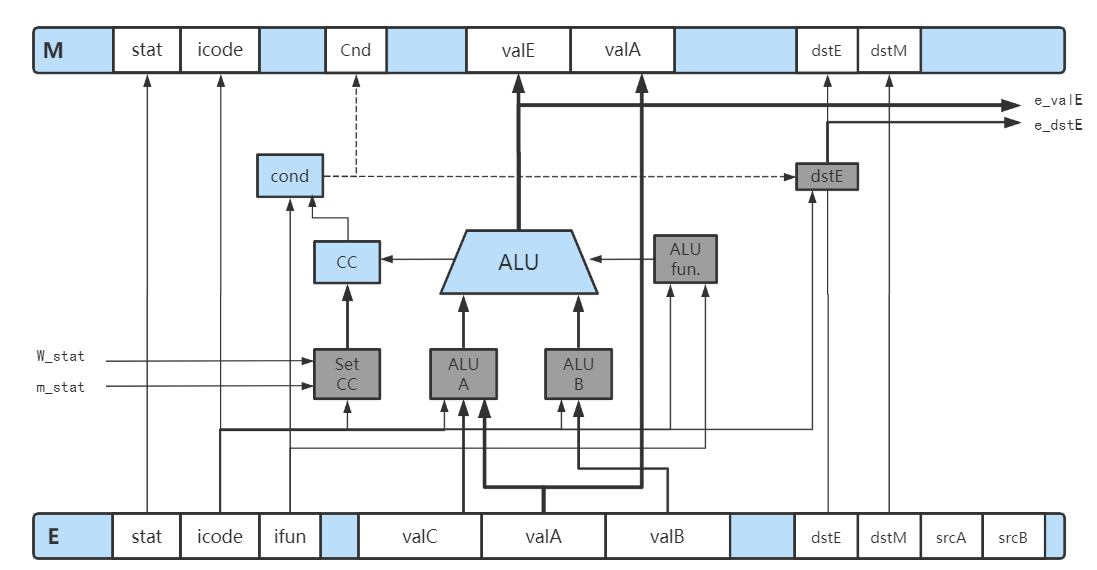
剩余复杂的部分主要是转发逻辑：

|  |  |  |
| --- | --- | --- |
| 数据字 | 寄存器ID | 源描述 |
| e\_valE | e\_dstE | ALU |
| m\_valM | M\_dstM | 内存输出 |
| M\_valE | M\_dstE | 访存阶段中对端口E未进行的写 |
| W\_dstM | W\_dstM | 写回阶段中对端口M未进行的写 |
| W\_dstE | W\_dstE | 写回阶段中对端口E未进行的写 |

目前不关心实现条件传送的逻辑

|  |  |
| --- | --- |
| word d\_valA = [  icode in {IJXX，ICALL}：D\_valP；  #使用增加的PC值  d\_srcA == e\_dstE ：e\_valE；  #使用执行阶段ALU计算的值  d\_srcA == M\_dstM ：m\_valM；  #使用访存阶段从内存读取的值  d\_srcA == M\_dstE ：M\_valE；  #使用访存阶段valE的值  d\_srcA == W\_dstM ：W\_valM；  #使用写回阶段valM的值  d\_srcA == W\_dstE ：W\_valE；  #使用写回阶段valE的值  1 ：d\_rvalA；#使用从寄存器文件端口A读出的值  ] | word d\_valB = [  d\_srcB == e\_dstE ：e\_valE；  #使用执行阶段ALU计算的值  d\_srcB == M\_dstM ：m\_valM；  #使用访存阶段从内存读取的值  d\_srcB == M\_dstE ：M\_valE；  #使用访存阶段valE的值  d\_srcB == W\_dstM ：W\_valM；  #使用写回阶段valM的值  d\_srcB == W\_dstE ：W\_valE；  #使用写回阶段valE的值  1 ：d\_rvalB；  #使用从寄存器文件端口B读出的值  ] |
| word d\_dstE = [  icode in {IRRMOVQ，IIRMOVQ，IOPQ} ：D\_rB；  icode in {IPOPQ，IPUSHQ，IRET，ICALL} ：RRSP；  1：RNONE；  ] | word Stat = [  W\_stat == SBUB：SAOK；#当写回阶段有气泡指令时，这里希望流水线状态是AOK的；  1：W\_stat；  ] |

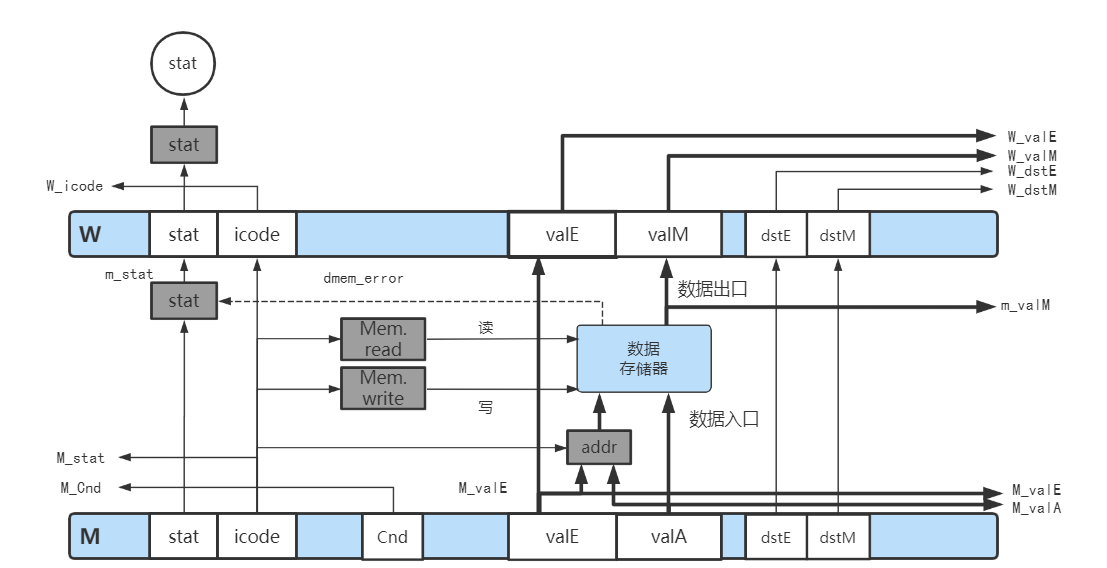
3、执行阶段



这个阶段与SEQ阶段非常相似，使用的信号适当的做重命名。e\_valE，e\_dstE作为转发源，标号为“Set CC”的块以信号m\_stat和W\_stat作为输入。

4、访存阶段

PIPE中并没有SEQ中标号为Data的块。这个块在SEQ中是用与对数据源中valP和valA中进行选择的，但是这个选择现在在译码阶段中标号为“SEL+Fwd A”的块来执行。这个阶段中的块与其他部分都相同，因此对部分信号进行适当命名。



|  |
| --- |
| word m\_stat=[  imem\_error：SADR；  1：M\_stat；  ] |

#### 4.5.6 流水线控制逻辑

流水线控制逻辑必须处理下面**四种情况**，这些情况是数据转发和分支预测不能处理的：

**加载\使用**：在一条从内存中读出一个值的指令和一条使用该值的指令之间，流水线必须暂停一个周期。

处理**ret指令**：流水线必须暂停直到ret指令到达写回阶段。

**预测错误**的分支：在分支逻辑发现不应该选择分支之前，分支目标处的几条指令已经进入流水线。必须取消这些指令，从跳转指令后面的那条指令开始取指。

**异常**：当一条指令导致异常，需要禁止后面的指令更新程序员可见的状态，并且在异常指令到达写回阶段时，停止执行。

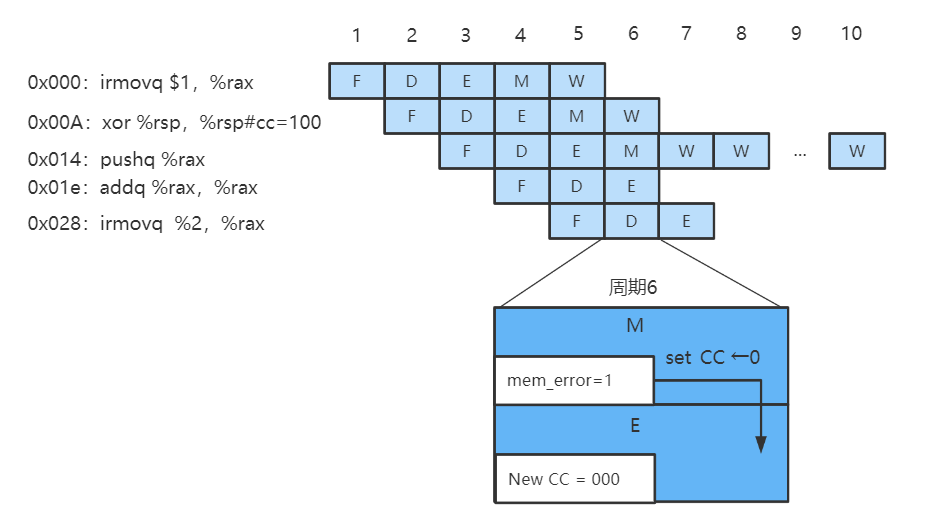
1、特殊控制情况的所期望的处理

只有mrmovq和popq指令会从内存中读数据，因此当这两条指令的任一条指令处于执阶段，并且需要该目的寄存器的指令正处于译码阶段时，要将这条指令阻塞在译码阶段，并在下一个执行阶段中插入一个气泡。此后转发会解决这个数据冒险。

处理ret指令需要添加三个气泡，由于无法在取指令阶段前插入气泡。当译码阶段发现是ret指令后，流水线会先取出下一条指令，然后下一条指令暂停在取指阶段。暂停3个时钟周期后，会在译码阶段会将该指令替换成气泡。

当分支预测错误发生时，跳转指令到达执行阶段就可以检测到预测错误，然后在一个时钟周期，控制逻辑就会在译码阶段和执行阶段插入气泡，取消两条不正确的已取指令。

对于导致异常的指令：（1）异常在两个阶段（取指和访存）被发现；（2）程序状态在三个阶段（执行，访存，写回）被更新。当异常指令到达访存阶段时，会采取措施防止后面的指令修改程序员可见的状态：（1）禁止执行阶段设置条件码；（2）向访存阶段中插入气泡，以禁止向内存中写入数据；（3）当写回阶段有异常指令时，暂停写回阶段，整个流水线停止。



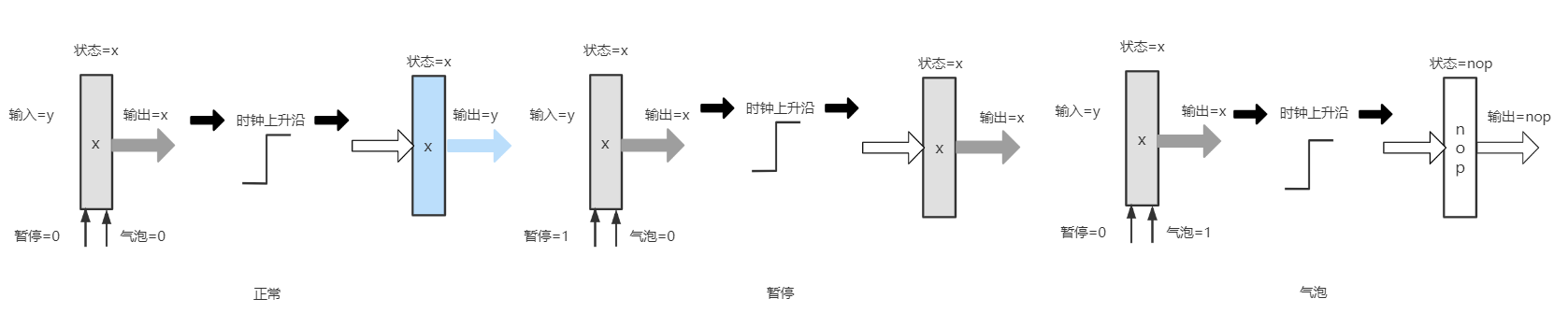
在周期6中，pushq到达访存阶段，产生一个内存错误，同一阶段addq指令产生一个新的条件码。当访存阶段或写回阶段有异常指令时（通过检查信号W\_stat和m\_stat，然后将信号设置为0），则执行阶段会禁止设置条件码。同时，处理器向访存阶段插入气泡，后面的指令都没有通过执行阶段。

2、发现特殊控制条件

|  |  |
| --- | --- |
| 条件 | 触发条件 |
| 处理ret指令 | RET ∈{D\_icode，E\_icode，M\_icode}； |
| 加载/使用冒险 | E\_icode ∈{IMRMOVL，IPOP} && E\_dstM∈{d\_srcA，d\_srcB}； |
| 预测错误的分支 | E\_icode = IJXX && !e\_Cnd |
| 异常 | m\_stat ∈ {SADR，SHLT，SINS} | W\_stat ∈ {SADR，SHLT，SINS} |

3、流水线控制机制

下图是一些低级控制机制，它们使得流水线指令阻塞在寄存器中，或是往流水线中插入一个气泡。假设每个流水线寄存器有两个控制输入：暂停和气泡。这些信号决定了当时钟上升如何更新流水线寄存器。（1）当暂停=1时，禁止更新状态，寄存器保持它以前的值；（2）当气泡=1时，寄存器状态会设置某个固定的复位配置，得到一个等效nop指令的状态。



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 合并 | 流水线寄存器 | | | | |
| F | D | E | M | W |
| 处理ret | 暂停 | 气泡 | 正常 | 正常 | 正常 |
| 加载\使用冒险 | 暂停 | 暂停 | 气泡 | 正常 | 正常 |
| 预测错误的分支 | 正常 | 气泡 | 气泡 | 正常 | 正常 |

4、控制条件的组合

因为加载\使用冒险需要执行阶段的指令是RMOVQ和POPQ，预测错误的分支需要执行阶段的指令是JXX，因此只有两种组合：处理ret和加载\使用冒险，处理ret和预测错误的分支。

* 处理ret和预测错误的分支：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 合并 | 流水线寄存器 | | | | |
| F | D | E | M | W |
| 处理ret | 暂停 | 气泡 | 正常 | 正常 | 正常 |
| 预测错误的分支 | 正常 | 气泡 | 气泡 | 正常 | 正常 |
| 组合 | 暂停 | 气泡 | 气泡 | 正常 | 正常 |

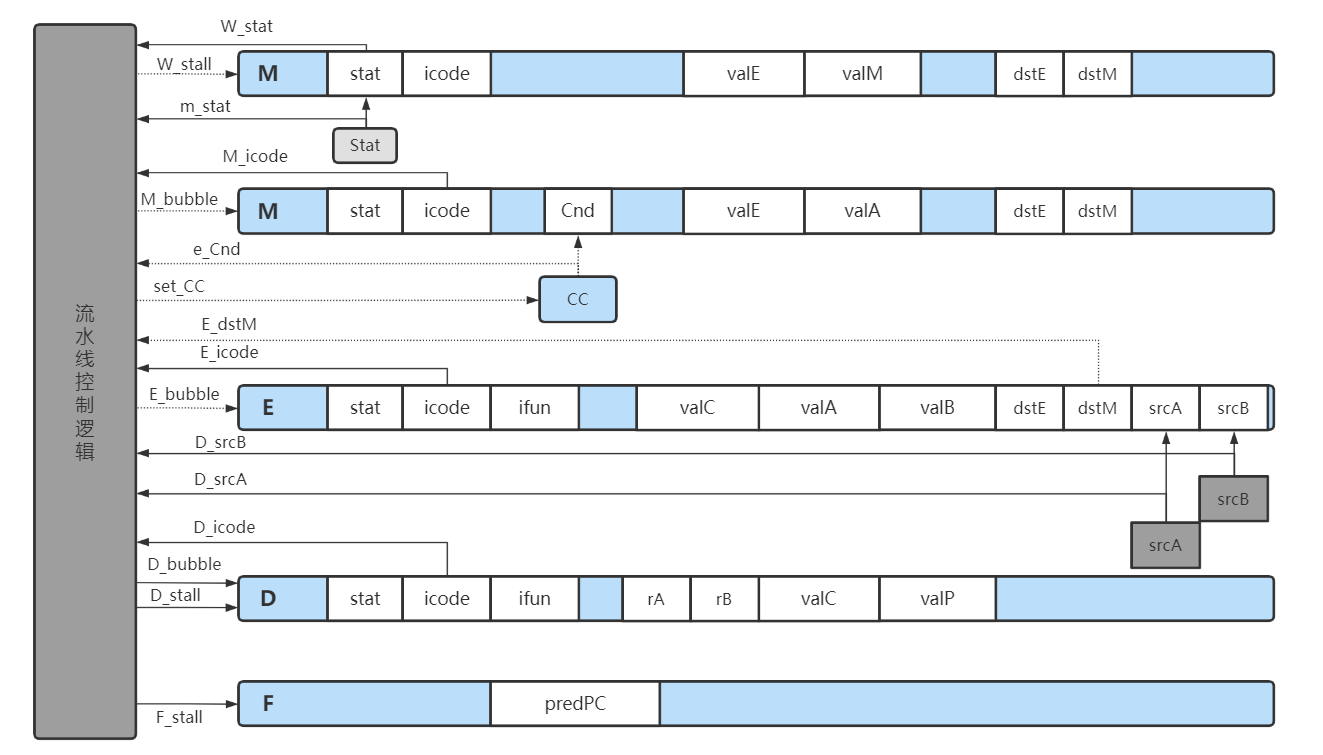
在下一个周期，会将后面那条ret指令取消，PC选择逻辑会选择跳转jxx后面那条指令的地址，而不是预测的程序计数器值，因此流水线F阶段发生了暂停并不会影响。

* 处理ret和加载\使用冒险：

如果同时触发两组动作，控制逻辑会试图暂停ret指令来避免加载/冒险指令，同时ret指令又会往译码阶段执行气泡，这显然不合理，因此这里只希望它针对加载/冒险指令。处理ret指令应该延迟一个周期。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 合并 | 流水线寄存器 | | | | |
| F | D | E | M | W |
| 处理ret | 暂停 | 气泡 | 正常 | 正常 | 正常 |
| 加载/使用 | 暂停 | 暂停 | 气泡 | 正常 | 正常 |
| 组合 | 暂停 | 气泡+暂停 | 气泡 | 正常 | 正常 |
| 期望的情况 | 暂停 | 暂停 | 气泡 | 正常 | 正常 |

5、控制逻辑实现



|  |  |
| --- | --- |
| bool F\_stall=  E\_icode in {IMRMOVQ，IPOPQ}  && E\_dstM in {d\_srcA,d\_srcB}  || IRET in {D\_icode，E\_icode，M\_icode}；  #当执行阶段是IMRMOVQ，IPOPQ而且E\_dstM的目标是译码阶段中寄存器的读的一个端口时，取指阶段需要暂停；  当译码阶段，执行阶段，访存阶段的指令是IRET的时候，取指阶段需要暂停。 | bool D\_stall=  E\_icode in {IMRMOVQ，IPOPQ}  && E\_dstM in {d\_srcA,d\_srcB}  #当执行阶段是IMRMOVQ，IPOPQ而且E\_dstM的目标是译码阶段中寄存器的读的一个端口时，译码阶段需要暂停； |
| bool D\_bubble=  (E\_icode == IJXX && !e\_Cnd) #分支预测错误  ||!(E\_icode in {IMRMOVQ，IPOPQ}  && E\_dstM in {d\_srcA,d\_srcB})  && IRET in {D\_icode，E\_icode，M\_icode}；  #如果出现加载/使用冒险后面跟IRET指令这里不能使用气泡，应该使用暂停。 | boole E\_bubble=  (E\_icode == IJXX && !e\_Cnd) #分支预测错误  ||E\_icode in {IMRMOVQ，IPOPQ}  && E\_dstM in {d\_srcA,d\_srcB} |
| bool setCC=  E\_icode == IOPQ && !m\_stat in {SADR，SINS，SHLT}  && !W\_stat in {SADR，SINS，SHLT} | bool M\_bubble=  m\_stat in {SADR，SINS，SHLT}|| W\_stat in {SADR，SINS，SHLT} |
| bool M\_bubble=  W\_stat in {SADR，SINS，SHLT} | |

**五、处理器体系结构 ★★（重点，难点）**

**六、存储器体系结构 ★★（重点）**

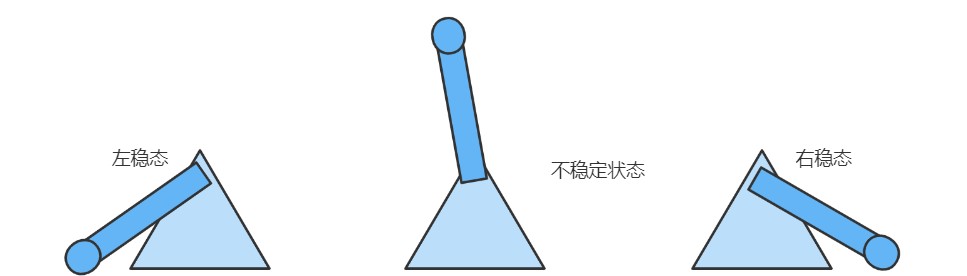
**6.1 存储技术**

**6.1.1 随机访问存储器**

1 静态RAM

SRAM将每个位存储在一个双稳态的存储器里。每个单元用一个六晶体管电路实现。这个电路可以无限期地保持在两个不同的电压配置或状态之一。其他任何状态都是不稳定的。

当钟摆倾斜到最左边或最右时，它是稳定的，在其他任何位置，钟摆都会倒向一边或另一边。由于SRAM存储起单元的双稳态特性，只要有电，它就会永远地保持它的值。



2 动态RAM

DRAM将每个位存储为对一个电容的充电。DRAM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。由于很多原因会导致漏电，DRAM单元在10~100ms时间内会失去电荷。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 每位晶体管数 | 相对访问时间 | 是否持续 | 是否敏感 | 相对花费 | 应用 |
| SRAM | 6 | 1× | 是 | 否 | 1000× | cache |
| DRAM | 1 | 10× | 否 | 是 | 1× | 主存 |

3 传统的DRAM

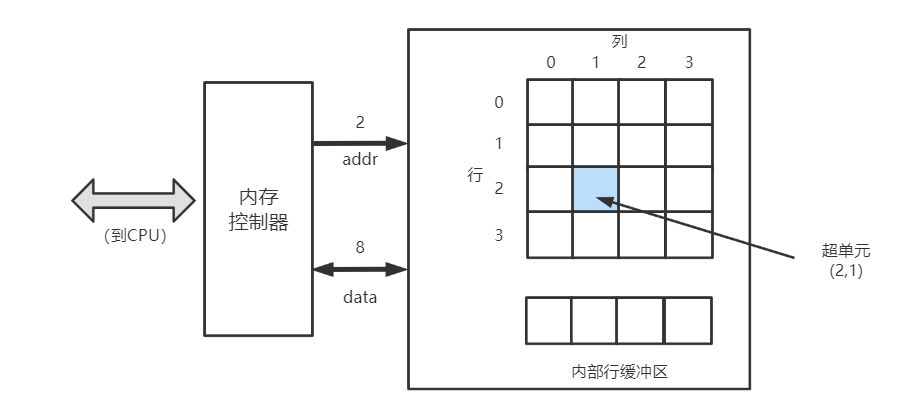
DRAM芯片被分为d个超单元，每个超单元都由w个DRAM单元组成。一个d×w的DRAM总共存储了dw位信息。r行c列的超单元组织成了方形阵列，rc = d。每个超单元有（i，j）的地址。

每个DRAM芯片被连接到某个称为内存控制器的电路，电路一次可以传送w位。DRAM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）的内容作为响应。行地址i称为RAS请求，列地址j称为CAS请求。这两个请求共享相同的DRAM地址引脚。

DRAM的响应会先将整个行复制到一个内部行缓存区。接下来会根据列地址从行缓存区中复制出超单元的wbit，并把它们发送到内存控制器中。

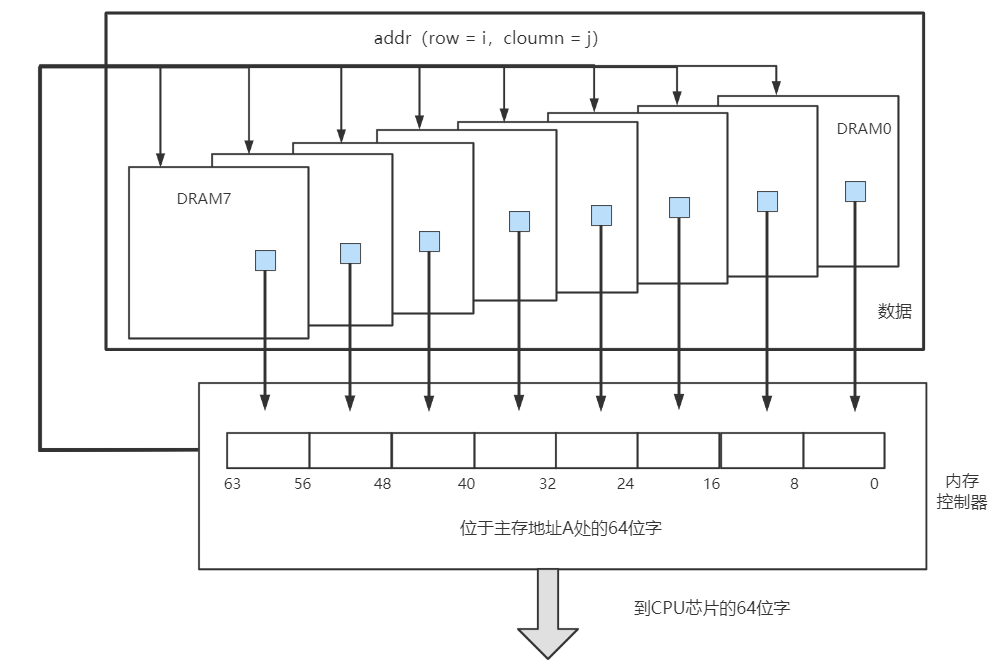
电路设计者将DRAM设计成二维阵列是降低芯片上地址引脚的数量。例如，128位的DRAM被组织成一个16个超单元的线性数组，地址为0~15，那么芯片需要4个地址引脚而不是2个。二维阵列的缺点是必须分两步发送地址，这增加了访存时间。

如下图所示，是一个16×8的DRAM芯片的组织，有d=16个超单元，每个超单元有w=8位。信息通过引脚的外部连接流入和流出芯片。图中给出两组引脚：8个data引脚；2个addr引脚：



4 内存模块

DRAM封装在内存模块中，插到主板的扩展槽上。Core i7使用240个引脚的双列直插内存模块，以64位为块传送数据到内存控制器和从内存控制器传出数据。

如图展现了8M×8的DRAM芯片，共64MB，8个芯片编号为0~7。每个超单元存储一个字节，而用相应超单元地址（i，j）的8个超单元来表示主存中A处64位字。DRAM0存储第一个字节，DRAM1存储下一个字节，依次类推。要取出内存地址A处的一个字，内存控制器将A地址转换为（i，j），然后将i，j广播到每个DRAM上。模块中电路收集DRAM上的输出，并把它们合并成一个64字，再返回给内存控制器。

r：DRAM阵列的行数；C：列数；br：行寻址所需位数；bc：列寻址所需位数

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 规格 | r | c | br | bc | max(br, bc) |
| 16×1 | 4 | 4 | 2 | 2 | 2 |
| 16×4 | 4 | 4 | 2 | 2 | 2 |
| 128×8 | 16 | 8 | 4 | 3 | 4 |
| 512×4 | 32 | 16 | 5 | 4 | 5 |
| 1024×4 | 32 | 32 | 5 | 5 | 5 |

5 增强的DRAM

* 快页模式DRAM（FPM DRAM）：传统DRAM将超单元的一整行复制到内部缓存区中，使用一个然后丢弃全部。FPM DRAM允许对同一行连续访问可以直接从缓存区得到服务。
* 扩展数据输出DRAM（EDO DRAM）：FPM的增强形式，允许各个CAS信号在时间上靠的更紧密一些。
* 同步DRAM（SDRAM）：使用同步控制信号，能更快速传输超单元内容
* 双倍数据速率同步DRAM（DDR SDRAM）：通过使用两个时钟沿作为控制信号，从而使DRAM速度翻倍。类型使用预取缓存区的大小区分：DDR（2位），DDR2（4位），DDR3（8位）
* 视频RAM（VRAM）：用在图形系统的帧缓存区中。

6 非易失性存储器

非易失性存储器在关电后仍然保持信息，通常用作ROM（只读存储器）。

* 可编程ROM（PROM）：只能被编程一次。每个存储器单元有一种熔丝，只能用电流熔断一次。
* 可擦写可编程ROM（EPROM）：有一个透明的石英窗口，当紫光照射窗口时，单元就被清0，重编程次数达到1000次。
* 电子可擦除PROM（EEPROM）：类似EPROM，它可以直接在印刷电路卡上编程，能够编程次数的数量级可以达到105次。
* 闪存（flash memory）：基于EEPROM，可用作固态硬盘

7 访问主存

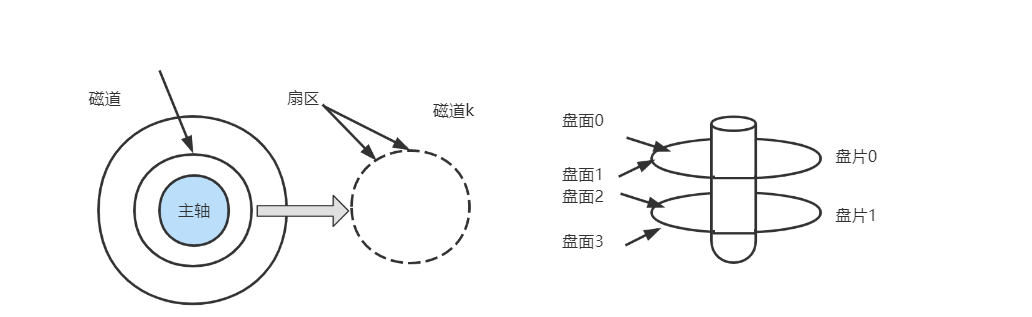
数据流通过bus的共享电子电路在处理器和DRAM主存之间传输。每次CPU和主存之间的数据传送都是通过一系列步骤来完成的，这些步骤称为总线事务。

* 读事务：主存传数据到CPU
* 写事务：CPU传数据到主存

#### 6.1.2 磁盘存储

1 磁盘构造

磁盘由盘片构造，每个盘片有两面或称为表面。每个表面有一组磁道；每个磁道被划分为1组扇区；每个扇区包含相等数量位，通常为512字节。到中心轴等距的磁道统称为柱面。



2 磁盘容量

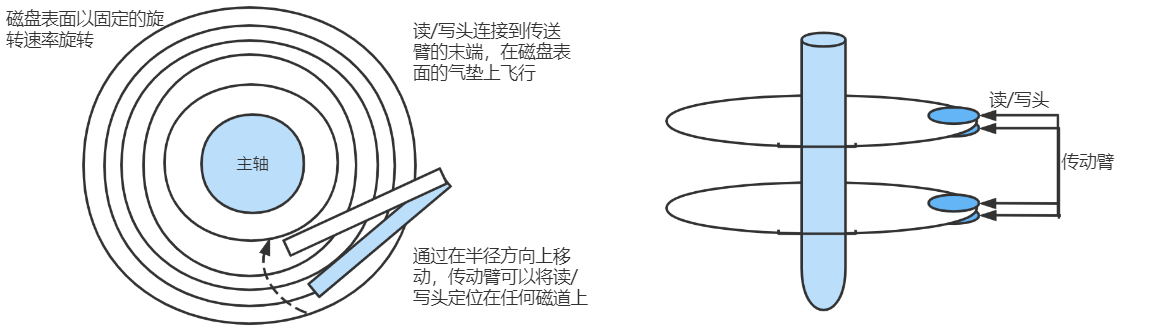
磁盘容量由以下因素决定：

* 记录密度（位/英寸）：磁道一英寸的段中可放入的位数
* 磁道密度（道/英寸）：从盘片众筹出发半径上一英寸的段内的磁道数
* 面密度（位/平方英寸）：记录密度与磁道密度的乘积。

例如，有一个磁盘，有5个盘片，每个扇区512字节，每个面20000条磁道，每条磁道平均300个扇区，磁盘的容量是：

这里1GB = 109B

3 磁盘操作



对扇区访问主要有三个操作：

* 寻道时间：为了读取某个目标扇区的内容，传动臂首先将读/写头定位到包含目标扇区的磁道上，移动传动臂的时间一般称为寻道时间。Tavg seek通常为3~9ms。
* 旋转时间：读/写头到达期望的磁道，需要通过等待磁盘的旋转，到达目标扇区。如果读/写头刚好错过目标扇区，则需要等待磁盘转一圈，因此最大旋转延迟：

平均旋转时间Tavg seek是Tmax rotation的一半

* 传送时间：当读/写头位于扇区下时就可以开始传送了，因此传送时间依赖于旋转速度和磁道上的扇区数量。

例如，考虑有如下参数的磁盘：

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 旋转速率 | Tavg seek | 每条磁道上平均扇区数 |
| 值 | 7200RPM | 9ms | 400 |

平均旋转延迟：

平均传送时间：

总时间：Taccess = 9 + 4 + 0.02 = 13.02

SRAM访问一个64位字大概是4ns，DRAM大概是60ns。因此，读512字节扇区大小的块SRAM是256ns，DRAM是3840ns，磁盘访问时间是10ms

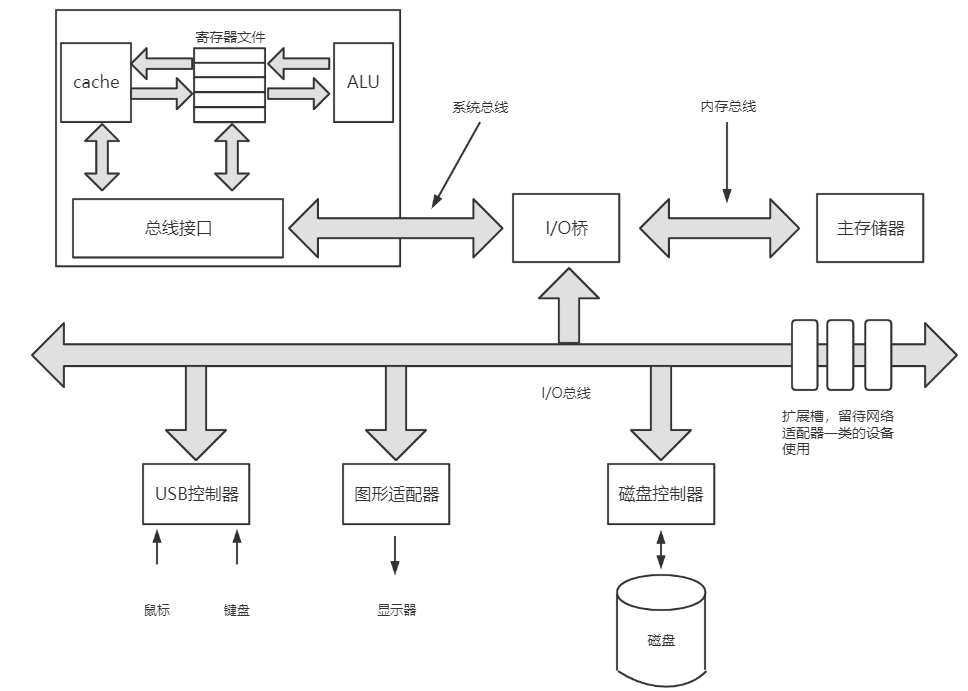
4 逻辑磁盘块

现代磁盘将有一个固件称为磁盘控制器，维护实际块和逻辑块的关系。操作系统进行I/O操作的时候，会向控制器发送读信号，控制器会将该逻辑块号翻译成3元组（盘面，磁道，扇区）

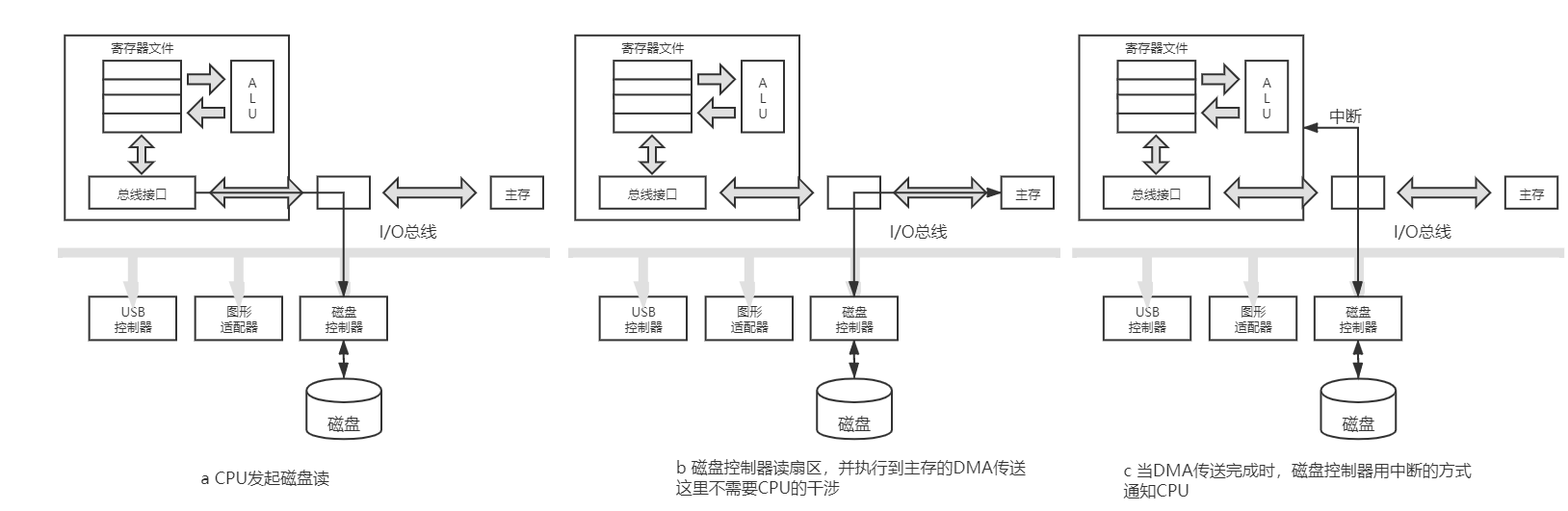
5 连接I/O设备

I/O设备通过I/O总线进行互联，有三种不同类型的设备：

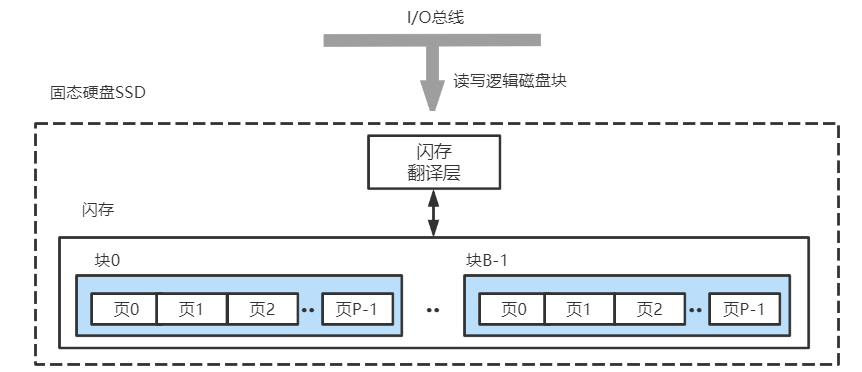
* 通用串行总线（Universal Serial Bus USB）：USB总线是一个广泛的使用标准，连接各种外围I/O设备。
* 图形卡：负责代表CPU在显示器上画像素
* 主机总线适配器：将磁盘连接到I/O总线



6 访问磁盘



#### 6.1.3 固态硬盘存储



固态硬盘是一种基于闪存的存储技术。它行为类似于硬盘处理来自CPU的读写逻辑磁盘的要求。SSD封装了多个闪存和闪存翻译层。一个闪存由B个块组成，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后，才能写这一页。在大约进行100000次重复写后，这个块就会磨损。

SSD的随机写很慢，首先擦除块需要相对较长的时间，1ms级；其次，如果写操作试图修改一个包含已有数据的页P，那么这个块所有带数据的页必须复制到一个新的块。

**6.2 局部性**

局部性分为两种：时间局部性和空间局部性

**时间局部性**：引用过一次的内存块在不久的将来有很大的概率将被多次引用（循环）。

**空间局部性**：如果内存被引用，那么内存附近的位置可能在将来被引用。

良好局部性的简单原则：

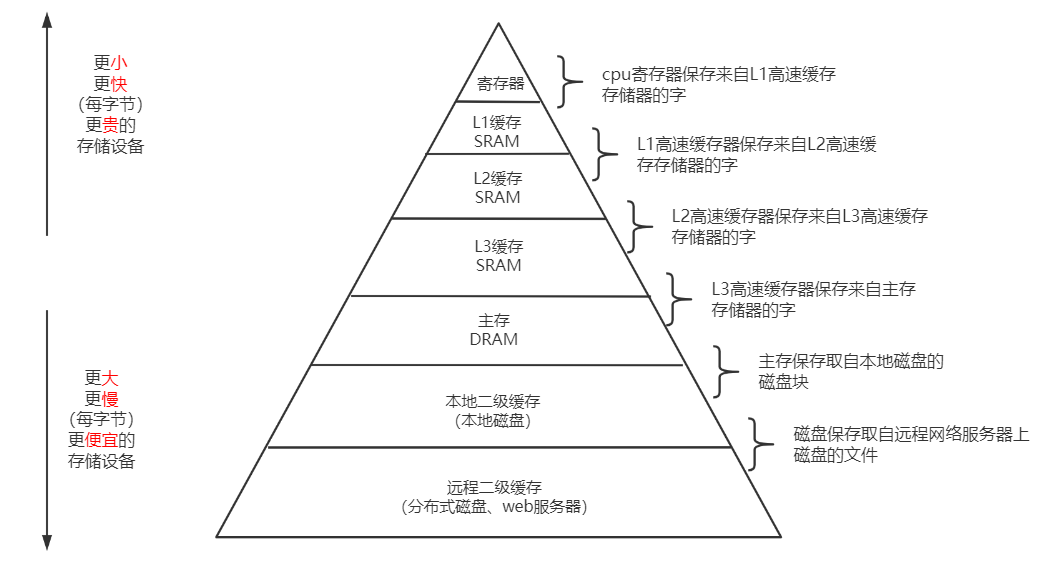
* 重复引用相同变量的程序具有良好的时间局部性
* 对于步长为k的引用模式的程序，步长越小，空间局部性越好
* 对于取指令，循环有好的时间局部性和空间局部性。循环体越小，循环迭代次数越多。

**6.3 存储器层次结构 ★★**

总结前两节：

存储技术：不同存储技术的访问时间差异很大。速度较快的技术每字节成本要高。

计算机软件：编写良好的程序倾向于展示出良好的局部性



#### 6.3.1 存储器层次结构的缓存

存储器结构的中心思想：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。即层次结构中的每一层都缓存来自较低一层的数据对象

1 缓存命中

当程序需要k+1层的某个数据对象d时，CPU首先访问k层，如果d刚好在k层，则称为缓存命中。

2 缓存不命中

如果第k层中没有缓存数据，则为缓存不命中。

3 缓存不命中的种类

* 强制性不命中：如果k层的缓存是空的，则对任何数据访问都会不命中，这有时会称为冷不命中
* 冲突不命中：缓存足够大，能够保存数据对象，但由于数据对象一直映射到同一个块，缓存会一直不命中。

4 缓存管理

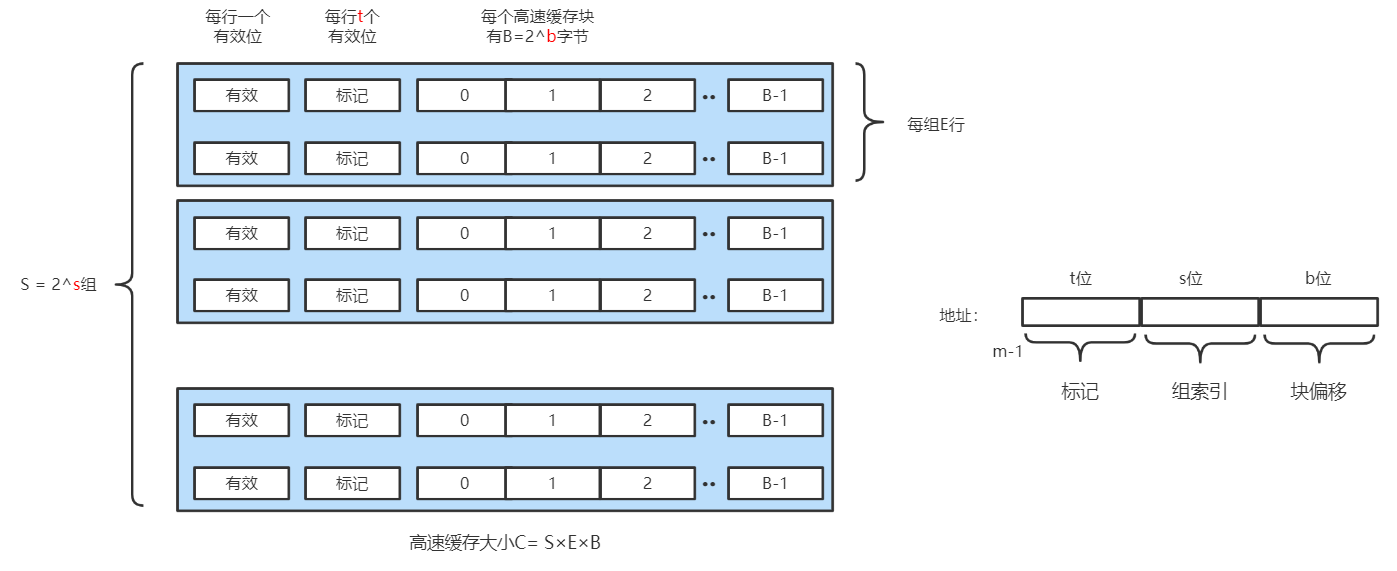
通过内存管理单元将缓存划分成块，在不同层之间进行传送，判定命中还是不命中，并处理它们。管理缓存的逻辑可以是硬件、软件或者两者的结合。

#### 6.3.2 存储器层次结构总结

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 B或8 B | CPU寄存器 | 0 | 编译器 |
| TLB | 地址翻译 | TLB | 0 | 硬件MMU |
| L1高速缓存 | 64 B块 | L1高速缓存 | 4 | 硬件 |
| L2高速缓存 | 64 B块 | L2高速缓存 | 10 | 硬件 |
| L3高速缓存 | 64 B块 | 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 | 100 000 | 控制器组件 |
| 网络缓存 | 部分文件 | 本地磁盘 | 10 000 000 | NFS客户 |
| 浏览器缓存 | Web页 | 本地磁盘 | 10 000 000 | Web浏览器 |
| Web缓存 | Web页 | 远程服务器磁盘 | 1 000 000 000 | Web服务器 |

**6.4 高速缓存存储器**

#### 6.4.1 高速缓存存储器组织结构



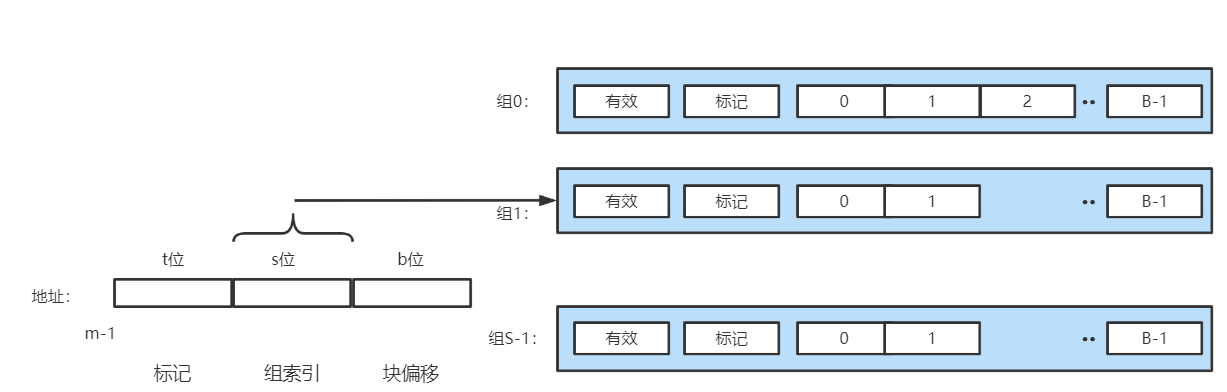
高速缓存通用形式（S，E，B，m）。每个组包含一个或多个行，每个行包含一个有效位，一些标记位，以及一个数据块；高速缓存m个地址位被划分为t个标记位、s个组索引和b个块偏移位。

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 描述 | 参数 | 描述 |
| S | 组数 | M = 2m | 内存地址的最大数量 |
| E | 每组行数 | s=log2S | 组索引位数量 |
| B | 块大小 | b=log2B | 块偏移位数量 |
| m | 物理地址位数 | t = m – (s+b) | 标记位数量 |
|  |  | C=B×E×S | 高速缓存大小 |

#### 6.4.2 直接映射高速缓存

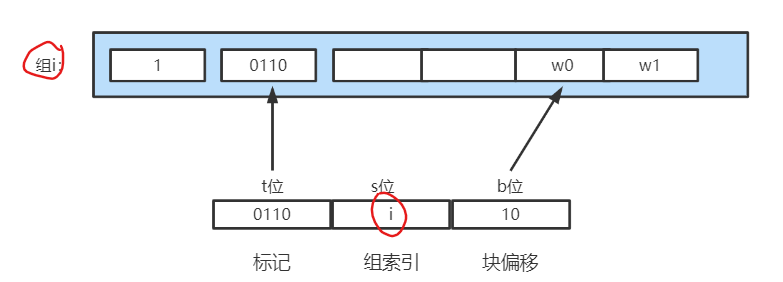
每个组只有一行（E=1）的高速缓存称为直接映射高速缓存。CPU请求一个字w所需的步骤：（1）组选择（2）行匹配（3）字抽取

1 组选择



高速缓存从w的地址中间抽取出s个位做组索引，这个位被解释成一个对应的无符号整数。

2 行匹配



上一步中确定了组i，接下来确定组中是否包含字w的副本。当有效位为1，且w中的标志位与组中的标志位相等时，则该组包含字w。如果有效位不为1或者标志位不相等，则为缓存不命中。

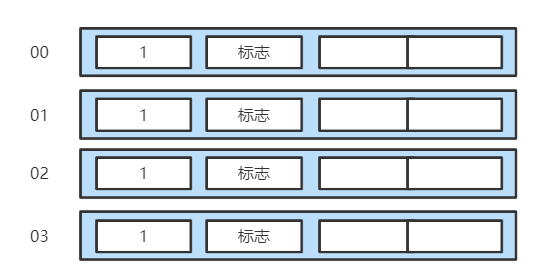
3 字选择

一旦命中，就需要知道字w在数据块的起始位置，通过b的块偏移可以确定。

4 缓存不命中时的行替换

如果缓存不命中，需要从存储器层次结构中的下一层取出被请求的块，对于直接映射高速缓存来说，只需要替换当前行即可。

假设高速缓存 (S,E,B,m)=(4,1,2,4)，即高速缓存有4个组，每个组一行，每个块2个字节，地址是4位的。



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址  （十进制） | 地址位（m=4） | | | 块号  （十进制） |
| 标记位（t=1） | 索引位（s=2） | 偏移位（b=1） |
| 0 | 0 | 00 | 0 | 0 |
| 1 | 0 | 00 | 1 | 0 |
| 2 | 0 | 01 | 0 | 1 |
| 3 | 0 | 01 | 1 | 1 |
| 4 | 0 | 10 | 0 | 2 |
| 5 | 0 | 10 | 1 | 2 |
| 6 | 0 | 11 | 0 | 3 |
| 7 | 0 | 11 | 1 | 3 |
| 8 | 1 | 00 | 0 | 4 |
| 9 | 1 | 00 | 1 | 4 |
| 10 | 1 | 01 | 0 | 5 |
| 11 | 1 | 01 | 1 | 5 |
| 12 | 1 | 10 | 0 | 6 |
| 13 | 1 | 10 | 1 | 6 |
| 14 | 1 | 11 | 0 | 7 |
| 15 | 1 | 11 | 1 | 7 |

接下来模拟一系列CPU读，初始时，高速缓存是空

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 组 | 有效位 | 标记位 | 块[0] | 块[1] |
| 00 | 0 |  |  |  |
| 01 | 0 |  |  |  |
| 10 | 0 |  |  |  |
| 11 | 0 |  |  |  |

(1)读地址0的字。有效位是0，发生缓存不命中

(2)读地址13（1101）的字。组2有效位是0，发生缓存不命中

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 组 | 有效位 | 标记位 | 块[0] | 块[1] |
| 00 | 1 | 0 | m[0] | m[1] |
| 01 | 0 |  |  |  |
| 10 | 1 | 1 | m[12] | m[13] |
| 11 | 0 |  |  |  |

(3)读地址8的字(1000)，由于组0的标记位不匹配，发生缓存不命中

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 组 | 有效位 | 标记位 | 块[0] | 块[1] |
| 00 | 1 | 1 | m[8] | m[9] |
| 01 | 0 |  |  |  |
| 10 | 1 | 1 | m[12] | m[13] |
| 11 | 0 |  |  |  |

(4)接下来读地址0的字，由于标记位又不匹配，这就是冲突不命中的例子，有足够的高速缓存空间，却交替替换映射到同一个组的块。

练习：假想高速缓存，用地址的高s位做索引，那么内存连续块被映射到同一高速缓存组。

A 每个这样的连续的数组片中有多少块？2t块，头2t块会被映射到组0，接下来2t块会被映射到组1，一次类推

B 高速缓存描述形式(S,E,B,m)=(512,1,32,32)，考虑下列程序

int array[4096];

for (I = 0;i < 4096;i++)

sum += array[i];

在任意时刻，存储在高速缓存中的数组块的最大数量？

t = 32 – (9+5) = 18，因此数组的头218块会映射到组0。

数组共有：4096\*4/32 = 512块 < 218，数组所有块都会映射到组0

因此，任意时刻高速缓存中只有组0，一个数据块。

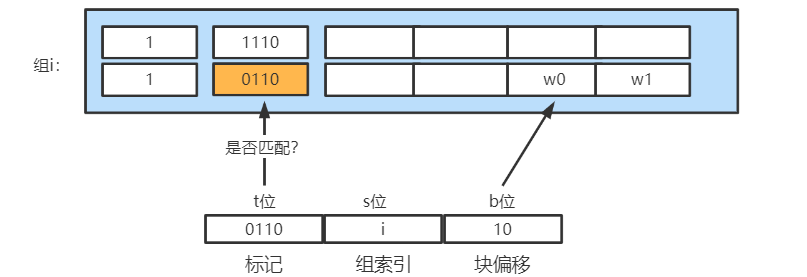
练习解释了为什么组索引在中间位，因为使用中间位作组索引，相邻的数据会映射到不同的高速缓存行。

#### 6.4.3 组相连高速缓存

当1＜E＜C/B的高速缓存，通常称为E路组相联高速缓存。

1 组选择（同直接映射高速缓存类似）

2 行匹配和字选择



组相联存储器是一个（key，value）键值对，key是标记和有效位，value是块的内容。在组相连高速缓存中，组中的任何一行都可以包含任何一个映射到这个组的内存块。因此高速缓存必须搜索组中每一行，寻找一个有效行。字选择和前面类似。

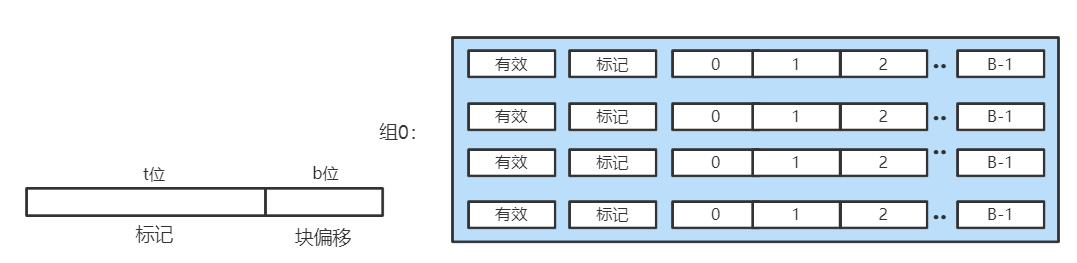
3 缓存不命中时的行替换

替换策略：随机替换；最不常使用（LFU）；最近最少使用（LRU）

#### 6.4.4 全相联高速缓存

当高速缓存行的组E = C/B时，被称为全相联高速缓存，此时高速缓存只有一个组，因此没有组索引。

1 组选择



因为只有一个组，地址被划分为标记和块偏移。

2 行匹配和字选择（同组相联类似）

因为高速缓存电路必须并行搜索许多相匹配的标记，构造一个又大又块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。例如，翻译备用缓存器TLB

有关写的问题：

在高速缓存更新副本后，有两种方法更新低一层的副本：

直写：立即将w的高速缓存块写回到紧接着的低一层中，简单但会引起总线流量。

回写：尽可能推迟更新，只有当替换算法要驱逐这个块时，才把它写到低的一层。

#### 6.4.5 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：

* 不命中率/命中率：不命中数/引用数量；1-不命中率
* 命中时间：从高速缓存传送一个字到CPU所需的时间
* 不命中处罚：由于不命中所需要的额外的时间。

1 高速缓存大小的影响

较大的高速缓存提高命中率，但可能会增加命中时间。

2 块大小的影响

较大的块能利用程序中可能存在的空间局部性。不过，对于给定的高速缓存大小，数据块越大，表示缓存行越少，则容易损害程序的命中率。同时数据块越大，传送时间也越长。

3 相联度影响

这是参数E的选择，较高的相联度降低了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带来较高的成本，而且很难使之速度变快。同时较高的相联度，每一行需要更多的标记位，需要额外的LRU状态位和额外的控制逻辑，实现复杂。

4 写策略影响

直写高速缓存比较容易实现，而且能使用独立于高速缓存的写缓冲区来更新内存。此外越往层次下面走，传送时间增加，减少传送的数量就变得更重要。一般而言高速缓存往下走，使用写回而不是直写。

练习：

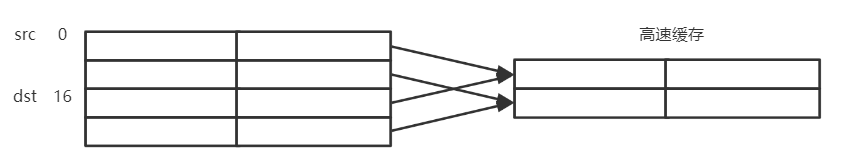
例如，考虑下面转置函数：

 typedef int array[2][2];  
 void transpose(array dst,array src){  
     int i,j;  
     for (i = 0; i < 2; i++){  
         for (j = 0; j < 2; j++){  
        dst[j][i] = src[i][j];  
    }  
    }  
 }

假设：

* sizeof(int) = 4;
* src数组从0地址开始，dst地址从16开始；
* 高速缓存大小16字节，一开始为空；

A：访问命中是h，不命中是m



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| dst数组 | | |  | src数组 | | |
|  | 列0 | 列1 |  |  | 列0 | 列1 |
| 行0 | m | m |  | 行0 | m | m |
| 行1 | m | m |  | 行1 | m | h |

首先访问src[0][0]，得到miss，然后访问dst[0]会将src[0]驱逐出去，因此访问src[0][1]时，需要重新加载并把dst[0]驱逐；

接下来加载dst[1][0]，会加载到第二个缓存行，发生冷不命中；

访问src[1][0]，会把第二缓存行（即dst[1][]）替换出去，发生不命中；访问dst[0][1]，会把src[0]驱逐出去，发生不命中。

访问src[1][1]，由于上一步骤已经缓存了，此时缓存中有数据块，发生命中；访问dst[1][1]把第二缓存行(src[1][])替换出去，发生不命中

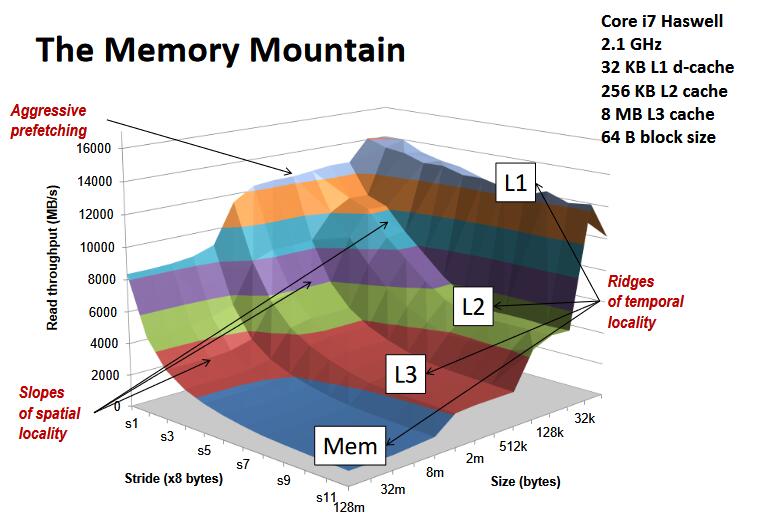
B：如果将缓存调整至32字节

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| dst数组 | | |  | src数组 | | |
|  | 列0 | 列1 |  |  | 列0 | 列1 |
| 行0 | m | h |  | 行0 | m | h |
| 行1 | m | h |  | 行1 | m | h |

由于缓存可以容纳两个数组，不需要进行替换，因此只有第一次加载会造成缓存不命中

**6.5 综合**

#### 6.5.1 存储器山

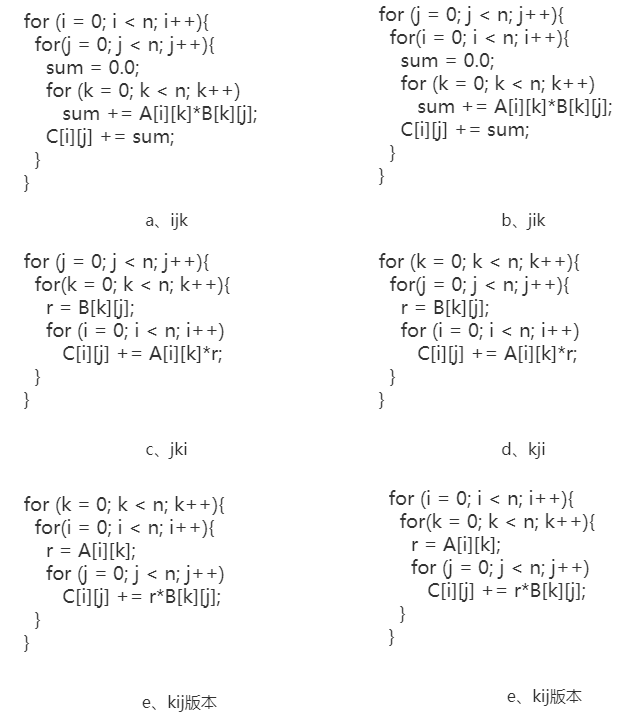


#### 6.5.2 重新排列循环提高空间局部性

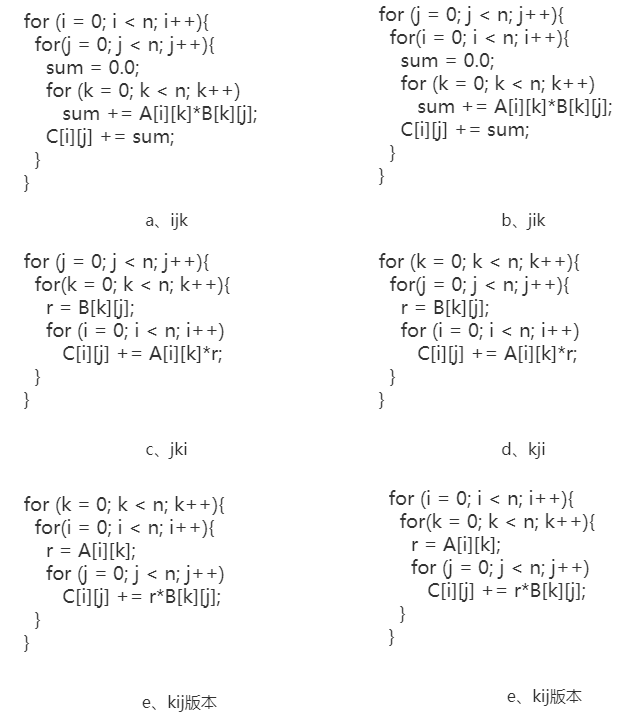
n×n矩阵相乘问题：C = A×B，例如n=2

下面有三个版本的矩阵计算，每个版本都是o(n3)时间，A和B的n2个元素中的每一个都要读n次。为了分析最里层循环迭代的行为，做如下假设：

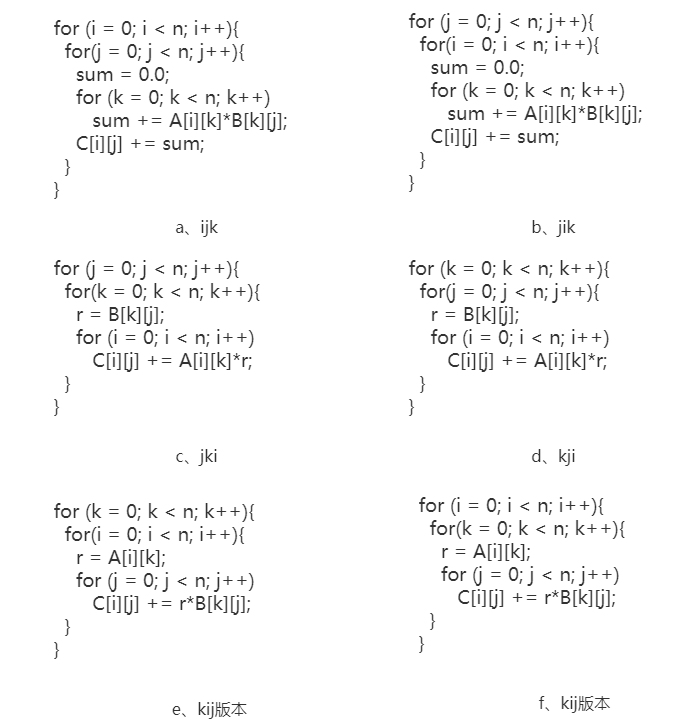
* 每个数组都是double类型的n×n数组，sizeof(double) = 8;
* 只有一个高速缓存，其块大小为32字节
* 数组n很大，以至于矩阵的一行都不能完全装进L1高速缓存
* 编译器将局部变量放到寄存器中



a、b：a、b情况类似可以放在一起讨论，以步长为1扫描数组A的一行，因此每4个就有1个冷不命中，未命中为0.25；由于高速缓存数据块大小不能放下一行，内循环以步长为n扫描数组B，因此每次访问都是不命中，未命中为1。未命中和为1.25。



c、d：c、d情况类似，可以一起讨论。这里发生两个加载和一个存储操作，内循环都是以步长为n扫描数组A和C，因此每次访问都是不命中，未命中和为2。



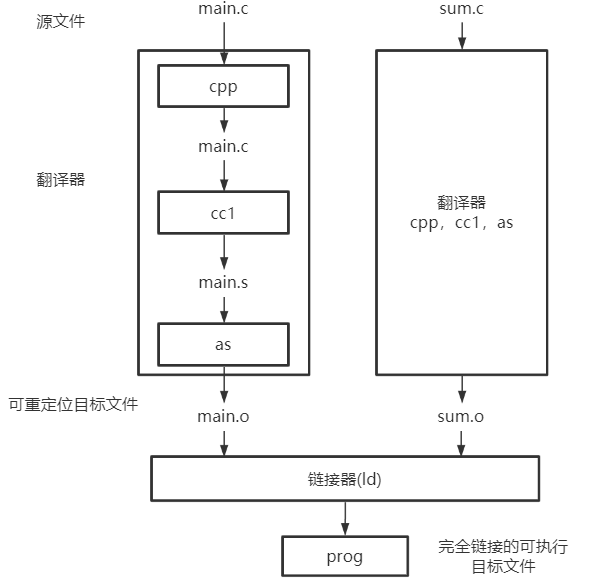
e、f：e、f情况类似，可以一起讨论。这里发生两个加载和一个存储操作，内循环都是以步长为1扫描数组B和C，每4次访问有1次冷不命中，因此总未命中为0.5。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 矩阵乘法  版本 | 每次迭代 | | | | | |
| 加载次数 | 存储次数 | A未命中数 | B未命中数 | C未命中数 | 总未命中数 |
| ijk&jik | 2 | 0 | 0.25 | 1.00 | 0.00 | 1.25 |
| jki&kji | 2 | 1 | 1.00 | 0.00 | 1.00 | 2.00 |
| kij&ikj | 2 | 1 | 0.00 | 0.25 | 0.25 | 0.50 |

**七、链接 ★★（重点）**

**7.1 概述**

#### 7.1.1 生成可执行目标文件



编译：驱动程序首先运行cpp将C的源程序翻译成ASCⅡ码的中间文件main.i，接着运行cc1将main.i编译成汇编语言的main.s，最后运行as，将main.s编译成可重定位目标文件main.o.

链接：运行链接器程序ld，将main.o和sum.o以及一些必要的系统文件组合起来，创建一个可执行目标文件。

shell调用操作系统的加载器，将可执行文件prog中的代码和数据部分复制到内存，然后将控制转移到这个程序的开头

#### 7.1.2 静态链接

为了构造可执行文件，链接器必须完成两个主要任务：

* 符号解析：目标文件定义和引用符号。
* 重定位：编译器和汇编器生成从地址0开始的代码和数据节。链接器通过把每个符号定义与一个内存位置关联起来，从而重定位这些节，然后修改所有对这些符号的引用，使得它们指向这个内存位置。

**7.2 目标文件**

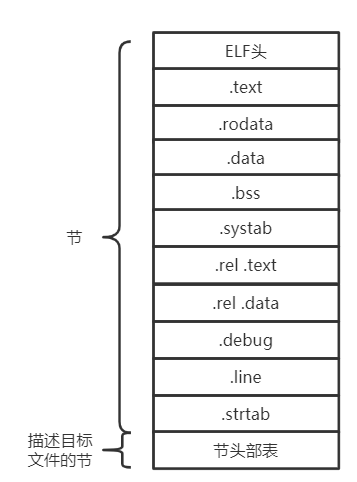
目标文件有三种形式：

可重定位目标文件：包含二进制代码和数据，其形式可以在编译时与其他可重定位目标文件合并起来，创建一个可执行目标文件。

可执行目标文件：包含二进制代码和数据，其形式可以被直接复制到内存并执行。

共享目标文件：一种特殊类型的可重定位目标文件，可以在加载或者运行时动态地加载进内存并链接。

#### 7.2.1 可重定位目标文件

.text：已编译的程序机器代码

.rodata：只读数据，比如printf语句中的格式串和switch语句的跳转表

.data：已初始化的全局和静态C变量

.bss：未初始化的全局和静态C变量，以及所有被初始化为0的全局或静态变量。在目标文件中这个节不占据实际空间，仅仅只是一个占位符。区分初始化变量和未初始化变量是为了空间效率，这个可以在运行时，内存分配这些变量。

.symtab：符号表，存放程序中定义和引用的函数和全局变量信息。

.rel.text：一个.text节中位置的列表，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。一般而言任何调用外部函数或引用全局变量的指令都需要修改。可执行目标文件不需要重定位信息，一般可以省略。

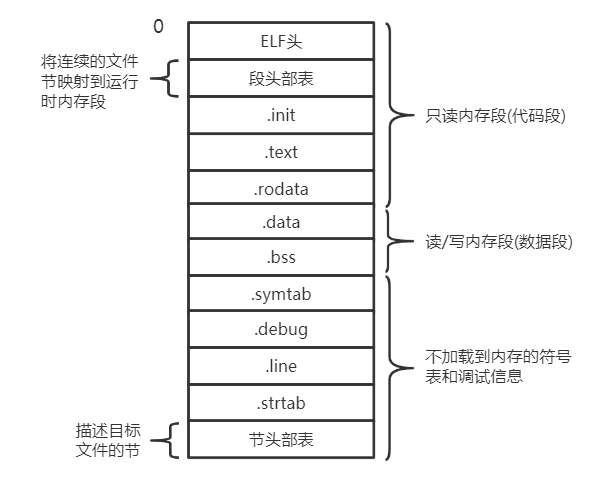
.rel.data：被模块引用或定义的所有全局变量的重定位信息。

.debug：调试符号表

.line：C源程序中的行号和.text节中机器指令之间的映射。

.strtab：字符串表，包含.symtab和.debug节中的符号表，以及节头部中的节名。

#### 7.2.2 可执行目标文件



ELF头部描述了总体的文件格式，以及程序的入口点。

.init节定义了一个函数，叫做\_init，程序的初始化代码会调用它。

由于可执行目标文件是完全链接的，所以不需要.rel节。

对于任何段s，链接器必须选择一个起始地址vaddr，使得：

vaddr mod align = off mod align

**7.3 符号和符号表**

全局符号：由本模块定义并能被其他模块引用的。对应于非静态的C函数和全局变量

外部符号：由其他模块定义并被本模块引用的符号，对应于其他模块中定义的非静态C函数和全局变量。

局部符号：对应于带static属性的C函数和全局变量。

（局部变量通常运行时保存于栈上，不属于上面的任何一种类型）

符号表条目：

|  |
| --- |
| typedef struct{ |
| int name; // 字符串表中的字节偏移，指向字符串名字 |
| char type：4， // 表示符号类型，函数或数据 |
| binding：4; // 表示符号是本地的，还是全局的 |
| char reserved; // 表示未使用 |
| short section； // 节头部表的索引 |
| long value; // 距定义目标节的起始位置偏移，对于ELF来说是绝对运行地址 |
| long size； // 目标的大小 |
| } ELF64\_Symbol; |
| int sum(int \*a, int n); |
| int array[2] = {1,2}; |
| int main(){ |
| int val = sum(array,2); |
| return val; |
| } |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Num: | value | Size | Type | Bind | Vis | Ndx | Name |
| 8: | 0000 0000 0000 0000 | 24 | FUNC | GLOBAL | DEFAULT | 1 | main |
| 9: | 0000 0000 0000 0000 | 8 | OBJECT | GLOBAL | DEFAULT | 3 | array |
| 10: | 0000 0000 0000 0000 | 0 | NOTYPE | GLOBAL | DEFAULT | UND | sum |

section有三个特殊的伪节，ABS表示不该被重定义的符号；UND代表未定义的符号，在本模块引用但在其他地方定义；COMMON表示还未被分配位置的未初始化的数据目标。1表示.text节；3表示.data节。

练习：

|  |  |
| --- | --- |
| m.c | swap.c |
| void swap(); | extern int buf[]; |
| int buf[2] = {1,2}; | int \*bufp0 = &buf[0]; |
| int main(){ | int \*bufp1; |
| swap(); | void swap(){ |
| return 0; | int temp; |
| } | bufp1 = & buf[1]; |
|  | temp = \*bufp0; |
|  | \*bufp0 = \*bufp1; |
|  | \*bufp1 = temp; |
|  | } |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 符号 | .symtab条目？ | 符号类型 | 在哪个模块中定义 | 节 |
| buf | 是 | 外部符号 | m.o | .data |
| bufp0 | 是 | 全局符号 | swap.o | .data |
| bufp1 | 是 | 全局符号 | swap.o | COMMON |
| swap | 是 | 全局符号 | swap.o | .text |
| temp | 否 |  |  |  |

**7.4 符号解析**

#### 7.4.1 链接器解析多重定义的全局符号

函数和已初始化的符号是强符号，未初始化的全局变量是弱符号

Linux链接器使用下面的规则处理多重定义的符号名：

* 规则1：不允许有多个同名的强符号
* 规则2：如果有一个强符号和多个弱符号同名，那么选择强符号
* 规则3：如果有多个弱符号同名，从这些弱符号中任意选择一个

例1，在这个情况下链接器会生成一条错误的信息，因为有两个强符号main

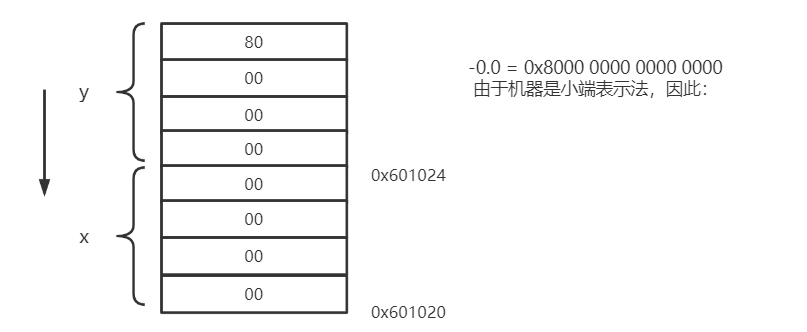
|  |  |
| --- | --- |
| // foo1.c | //bar1.c |
| int main (){ | int main (){ |
| return 0; | return 0; |
| } | } |

例2，在这情况模块有一个x未被初始化，那么链接器默认选择另一个模块中定义的强符号。运行时，函数f将x的值由15213改为15212。链接器通常不会表明它检测多个x的定义：

|  |  |
| --- | --- |
| // foo3.c | //bar3.c |
| #includ <stdio.h> | int x; |
| void f(); | void f(){ |
| int x = 15213; | x = 15212 |
| int main (){ | } |
| f();  printf(“x = %d\n”,x); |  |
| return 0; |  |
| } |  |

例3，在这种情况下有两个弱符号，容易产生不易察觉的错误

|  |  |
| --- | --- |
| // foo5.c | //bar5.c |
| #includ <stdio.h> | double x; |
| void f(); | void f(){ |
| int x = 15212; | x = -0.0; |
| int y = 15213; | } |
| int main (){ |  |
| f();  printf(“x = 0x%x y = 0x%x \n”,x,y); |  |
| return 0; |  |
| } |  |



假设系统中x的地址为0x601020，y的地址为0x601024，bar5.c中对double类型的x赋值-0.0（0x8000 0000 0000 0000）会覆盖内存中x和y的位置，导致最终打印结果是x = 0x0，y=0x8000 0000。

#### 7.4.2 与静态库链接

在没有静态库的情况下，编译器开发人员需要将所有标准C函数放在一个单独的可重定位目标模块中，应用程序员可以将这个模块链接到它们的可执行文件中：

优点：将编译器的实现与标准函数的实现分离

缺点：

* 空间浪费：系统中每个ELF都包含函数集合的完全副本，这是对磁盘空间的巨大浪费；每个正在运行的程序都将它这些函数副本放在内存中，这是对内存的浪费
* 时间浪费：对任何标准函数的任何改变，都要求库的开发人员重新编译整个源文件

静态库：将所有相关的目标模块打包成为一个单独的文件，称为静态库，可以用做链接器的输入。在linux系统中，静态库以一种存档的特殊文件格式存放在磁盘中。

#### 7.4.3 链接器解析引用

在符号解析阶段，链接器从左到右按照命令行出现的顺序来扫描可重定位目标文件和存档文件。在这次扫描中链接器维护一个可重定位目标文件的集合E，一个未解析（已引用未定义）的符号U，以及在输入文件中已定义的符号集合D。解析步骤：

* 初始时，三个集合都是空
* 对于命令行上每个文件f，链接器会判断文件是目标文件还是存档文件。如果是目标文件，则添加到E；修改U和D反映f中的符号定义和引用
* 如果f是存档文件，链接器就尝试匹配U中未解析的符号和由存档文件定义的符号。如果存档目标文件成员m定义了一个符号解析U中的一个引用，那么就将m添加到E中，并且修改U和D来反映成员m的符号定义和引用。对所有存档文件成员都依次进行这个过程，直到U和D都不再发生变化。不包含在E中的成员将会被丢弃，链接器会处理下个文件。
* 如果链接器完成输入文件扫描后，U是非空的，则输出一个错误并终止。否则，链接器会合并和重定位E中的目标文件，构建输出的可执行文件。

**7.5 重定位**

一旦链接器完成符号解析就开始重定位步骤，为每个符号分配运行时地址。重定位由两步组成：

* 重定位节和符号定义：链接器将所有相同类型的节合并为一个新的节；将运行时内存地址赋给新的节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号。
* 重定位节中的符号引用：修改代码节和数据节中对每个符号的引用，使他们指向正确的 运行时地址。

#### 7.5.1 重定位条目

代码的重定位条目放在.rel.text中，已初始化数据的重定位条目放在.rel.data中。

这里只关心两种基本的重定位类型，这两种重定位类型支持X86-64小型代码模型，该模型假设ELF文件中代码和数据＜2GB：

* R\_X86\_64\_PC32。重定位使用32位PC相对地址（距PC当前运行值的偏移量）的引用
* R\_X86\_64\_32：重定位使用32位绝对地址的引用。

#### 7.5.2 重定位符号引用

假设每个节s是一个字节数组，重定位条目r是一个类型为ELF64\_Rela的结构，链接器为每个节和符号分配运行时地址，分别为ADDR(s)和ADDR(r.symbol)表示。

下面展示了链接器重定位算法的伪代码：

|  |
| --- |
| foreach section s{ |
| foreach relocation entry r{ |
| refptr = s + r.offset; |
| if (r.type == R\_X86\_64\_PC32){ //重定位PC相对地址引用 |
| refaddr = ADDR(s) + r.offset; |
| \*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr); |
| } |
| if (r.type == R\_X86\_64\_ 32){ // 重定位绝对地址引用 |
| \*refptr = (unsigned) (ADDR(r.symbol) + r.addend ); |
| } |
| } |

用下面代码说明：

|  |
| --- |
| 0000 0000 0000 0000 <main>: |
| 0：48 83 ec 08 sub %0x8,%rsp |
| 4：be 02 00 00 00 mov %0x2,%esi  9：bf 00 00 00 00 mov %0x0,%edi %edi=array  a：R\_X86\_64\_32 array Relocation entry  e：e8 00 00 00 00 callq 13<main+0x13> 调用sum  f：R\_X86\_64\_PC32 sum-0x4 Relocation entry  13：48 83 c4 08 add %0x8,%rsp 调用sum  17：c3 retq |

1 重定位PC相对引用

call指令开始于相对于节偏移0xe，包括一字节操作码0xe8，后面跟着对目标sum的32位PC相对引用。

重定位条目r由4个字符组成：

|  |  |  |  |
| --- | --- | --- | --- |
| r.offset | r.symbol | r.type | r.addend |
| 0xf | sum | R\_X86\_64\_PC32 | -4 |

假设链接器设定ADDR(s) = ADDR(.text) = 0x4004d0和ADDR(r.symbol) = ADDR(sum) = 0x4004e8，

refaddr = ADDR(s) + r.offset = 0x4004d0 + 0xf = 0x4004df

\*refptr = unsigned (ADDR(r.symbol) + r.addend - refaddr)

= unsigned (0x4004e8 – 4 – 0x4004df) = 0x5

对call指令重定位如下

|  |
| --- |
| e：e8 05 00 00 00 callq 13<main+0x13> |

2 重定位绝对地址引用

|  |  |  |  |
| --- | --- | --- | --- |
| r.offset | r.symbol | r.type | r.addend |
| 0xa | array | R\_X86\_64\_32 | 0 |

假设链接器设定ADDR(r.symbol) = ADDR(array) = 0x601018

\*refptr = unsigned (ADDR(r.symbol) + r.addend)

= unsigned (0x601018 + 0) = 0x601018

因此第9行内容改写为

|  |
| --- |
| 9：bf 18 10 60 00 mov $0x601018,%edi %edi = &array |

在加载的时候，加载器会将这些字节直接复制到内存中，不再进行任何修改。

加载ELF

Linux通过调用execve函数来调用加载器，加载器将可执行目标文件中的代码和数据从磁盘复制到内存中，然后通过跳转到程序的入口点来运行该程序。这种将程序复制到内存并运行的过程叫做加载。

**7.6 动态链接共享库**

静态库仍然有一些缺点：

* 必须定期维护和更新。应用程序员必须显式了解情况，然后显式将程序与更新后的库重新链接。
* 每个C程序都使用标准I/O函数，在运行时这些函数的代码会被复制到每个运行进程的文本段中。这是对内存的极大浪费

共享库的出现解决了静态库缺陷，共享库是一个目标模块，在运行或加载时，可以加载到任意的内存地址，并和任一个内存中进程链接。这个过程称为动态链接。Linux中通常以.so结尾。微软中称为DDL。

#### 7.6.1 加载后执行前链接

当加载器加载和运行可执行文件时，加载部分链接的可执行文件。接着注意到程序中包含.interp节，这一节包含动态链接器的路径名，加载器不会将控制传递给应用，而是传递给这个控制器。然后，动态链接器通过执行重定位完成任务：

* 重定位libc.so的文本和数据到某个内容段
* 重定位libvector.so的文本和数据到另一个内存段
* 重定位prog21中所有对libc.so和libvector.so定义的符号和引用

最后动态链接器将控制传递给引用，进入程序入口点。

#### 7.6.2 运行中加载和链接共享库

将每个生成动态内容的函数打包在共享库中。当一个请求到来时，动态地加载和链接适当的函数，然后直接调用它，而不是使用fork和execve在子进程的上下文中运行函数。函数会一直存在内存的地址空间。这对web来说至关重要，即可以在运行时，更新已存在的函数，以及添加新的函数。

**7.7 位置无关代码**

现代系统以一种方式编译共享模块的代码段，使得可以把他们加载到内存的任何位置而无需链接器修改。这种代码段称为位置无关代码（PIC）。

1 PIC数据引用

代码段与数据段的距离总是保持不变，因此代码段中任何指令和数据段中变量都是一个运行时内存常量。

GOT全局偏移量表：在数据段开始的地方创建。在GOT中每个被目标模块引用的全局变量都有一个8字节条目。在加载时，动态链接会重定位GOT中的每个条目，使得它包含目标的正确的绝对地址。每个引用模块都有自己的GOT。

如图：addvec通过GOT[3]间接加载全局变量addcnt地址，然后在内存中加1、这里的关键思想是对GOT[3]的PC相对引用偏移量是一个运行时常量。



2 PIC函数调用

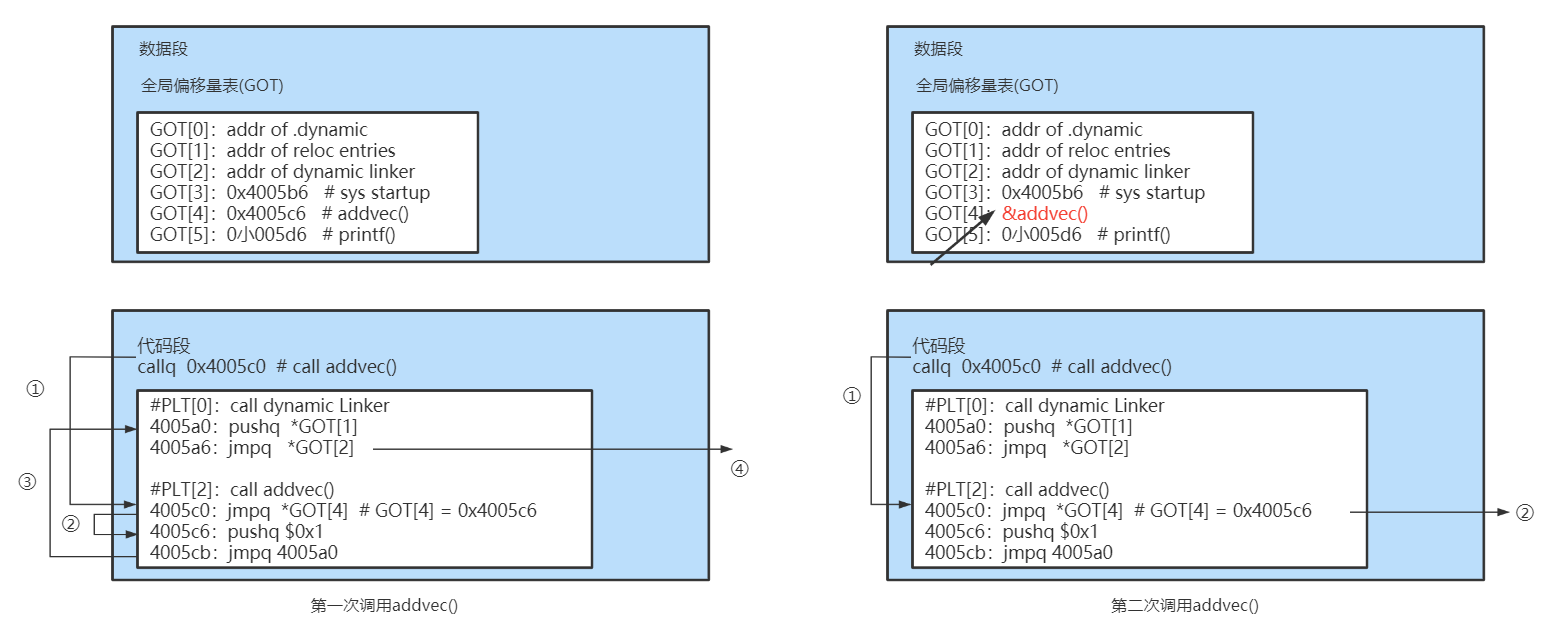
延迟绑定：将函数地址的绑定推迟到第一次调用时。（libc.so这样的共享库有大量的函数，应用程序一般只会使用少量函数，把函数地址的解析推迟到实际调用的地方，能避免动态链接器在加载时对不必要函数进行重定位。）

延迟绑定的构成：GOT和过程链接表 PLT。GOT是数据段的一部分，PLT是代码段的一部分。

过程链接表：PLT是一个数组，每个条目是16字节代码。PLT[0]跳转到动态链接器，PLT[1]调用系统启动函数。从PLT[2]开始的条目调用用户代码调用的函数。

GOT：和PLT联合使用时，GOT[0]和GOT[1]包含动态链接器在解析函数时会使用的信息。GOT[2]是动态链接器在ld-linux.so模块中的入口点。其余每个条目对应于一个被调用的函数，其地址需要在运行时解析。

下面是PIC函数调用实例：



addvec第一次调用：

①：程序调用进入PLT[2]，这是addvec的PLT条目

②：第一条指令通过GOT[4]间接跳转到4005c6，即PLT[2]的第二条指令。

③：把addvec的ID(0x1)压入栈中，然后PLT[2]跳转到PLT[0]

④：PLT[0]首先通过GOT[1]间接把参数压入栈，然后通过GOT[2]跳转进动态链接器中。动态链接器使用两个栈条目来确定addvec的运行时位置，然后重写GOT[4]

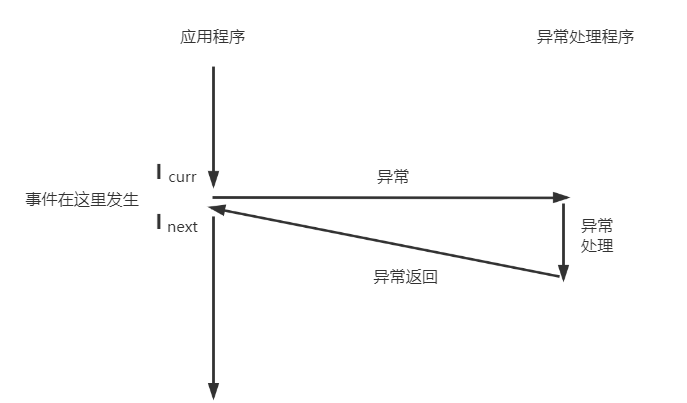
addvec第二次调用：

①：和先前一样跳转至PLT[2]

②：这次通过GOT[4]的间接跳转会将控制权转移到addvec

**八、异常控制流（选择题重点）**

**8.1 异常**

异常是控制流中的突变，用来响应处理器状态中的某些变化。一部分是由硬件实现，一部分由操作系统实现。

发生异常时，会通过异常表的跳转表，进行一个间接过程调用，到一个专门设计用来处理这类事件的操作系统子程序。处理完异常后，会根据引起异常的事件的类型，发生以下3种情况的一种：

①：处理程序将控制返回给当前指令Icurr，

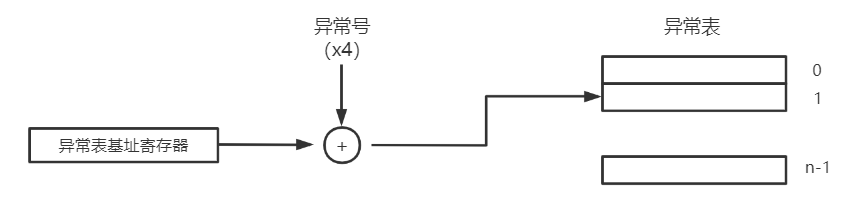
②：处理程序将控制返回给Inext，没有发生异常将要执行的下一条命令

③：终止被中断的程序。

#### 8.1.1 异常处理

处理异常需要软件和硬件紧密合作。

系统中可能的每种类型的异常都分配了一个唯一的非负整数的异常号。其中一些由处理器设计者分配，其他号码由操作系统内核的设计者分配。在系统启动时，操作系统分配和初始化一张称为异常表的跳转表，使得表目k包含异常k的处理程序地址。异常表的起始地址放在异常表基址寄存器中。



与过程调用区别：

* 异常根据类型返回地址要么是当前指令，要么是下一条指令
* 处理器会把额外的状态压入栈中
* 如果控制从用户转到内核，所有这些项目都被压倒内核栈中，而不是用户栈
* 异常处理程序运行在内核模式下

硬件触发异常，剩下的工作由异常处理程序在软件中完成。

#### 8.1.2 异常的类别（★★）

异常可以分为四类：中断，陷阱，故障和终止。

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | 原因 | 异步/同步 | 返回行为 |
| 中断 | 来自I/O设备的信号 | 异步 | 总是返回到下一条指令 |
| 陷阱 | 有意的异常，向内核请求服务 | 同步 | 总是返回到下一条指令 |
| 故障 | 潜在可恢复的错误，例如缺页故障 | 同步 | 可能返回到当前指令 |
| 终止 | 不可恢复的错误 | 同步 | 不会返回 |

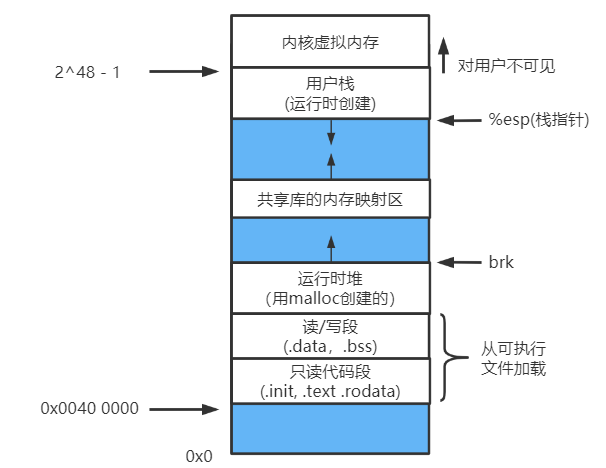
**8.2 进程**

进程：一个执行中程序的实例。

关键抽象：

* 一个独立的逻辑控制流，使得程序好像独占CPU
* 私有的地址空间，使得程序好像独占内存系统

#### 8.2.1 私有空间



在一台n位机器上，地址的可能集合数是2n个。

地址空间底部是保留给用户程序，包括代码、数据堆栈。代码通常从0x400 000开始，顶部留给内核。

#### 8.2.2 基础概念

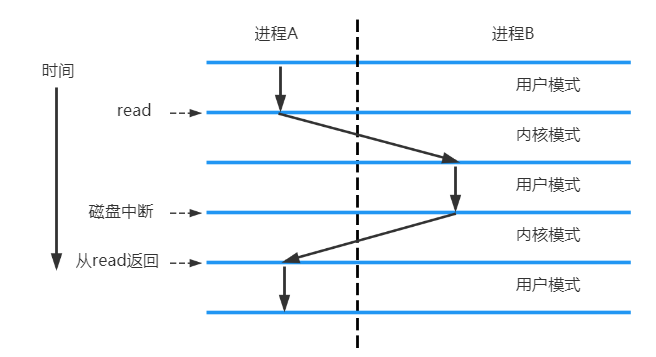
1 用户模式和内核模式

处理器使用控制寄存器的模式位来控制应用可执行的指令以及访问的地址空间范围。当设置模式位时，进程就运行在内核模式，可以**执行任何指令**，访问系统中的**任何内存位置**；当没有设置模式位时，进程就运行在用户模式。用户模式不允许执行特权指令，直接引用内核区的代码和数据。

2 上下文切换

上下文：内核重新启动一个被抢占的进程所需要的状态。

上下文切换：保存当前进程的上下文；恢复先前某个被抢占进程保存的上下文；将控制传递给这个新恢复的进程



系统调用错误处理

当Unix系统级函数遇到错误时，通常会返回-1，并设置全局整数变量errno表示出错原因。

void unix\_error(char \*msg){

fprintf(stderr, “%s：%s\n”，msg，strerror(errno))；

exit(0);

}

**8.3 进程控制**

从程序员的角度，进程总是处于下面三种状态之一：

* 运行：进程要么在CPU上执行，要么在等待被执行且最终会被内核调度
* 停止：进程的执行被挂起，且不会被调度
* 终止：进程永远停止了，进程会因为三种原因终止：收到一个终止信号；从main返回；调用exit()函数

#### 8.3.1创建进程函数

pid\_t fork(void); //创建进程，调用一次返回两次；父进程中返回子进程PID，子进程中返回0

新创建的子进程几乎与父进程相同：

* 相同但是独立的地址空间：两个进程的地址空间是相同的，包括代码段、数据段、堆，但是父子进程是独立的进程，有自己的私有空间。
* 共享文件：子进程继承了父进程所有的打开文件

父子进程最大的区别在于它们有不同的PID。

#### 8.3.2 子进程回收函数

当进程由于某种原因终止时，内核不会立即回收它，进程会保持一种已终止的状态，直到被它的父进程回收，一个终止但未被回收的进程称为僵死进程。如果一个进程终止，内核会安排init进程（是系统启动时创建，不会终止）回收它的僵死子进程。

pid\_t waitpid(pid\_t pid, int \*statup, int options);

返回：如果成功则为子进程pid；如果WNOHANG，则为0；其他错误则为-1；

当options=0（默认情况），waitpid会挂起调用进程的执行，直到它的等待集合中的一个子进程终止；如果等待集合中一个进程在刚调用时终止，waitpid立即返回。此时终止的子进程已经被回收。

1 判定等待集合的成员

pid > 0，等待集合是一个单独的子进程，进程ID = pid

pid = -1，等待集合由父进程中所有子进程构成。

2 options

options可以设置为常量WNOHANG，WUNTRACED，WCONTINUED

* WNOHANG：如果等待集合中的任何子进程都没有终止，立即返回。
* WUNTRACED：挂起调用进程的执行，直到等待进程的中一个进程变为已终止或被停止
* WCONTINUED：挂起调用进程的执行，直到等待进程的中一个正在运行的进程终止或等待集合中一个被停止的进程收到SIGCONG信号重新开始执行。

3 检查已回收子进程的退出状态

* WIFEXITED(status)：如果子进程通过调用exit或者返回一个正常终止，返回true
* WEXITSTATUS(status)：返回一个正常终止的子进程的退出状态
* WIFSIGNALED(status)：如果子进程是因为一个未被捕获的信号终止的，返回真
* WTERMSIG(status)：返回导致该子进程终止信号的编号，如果WIFSIGNALED(status) 为真才定义
* WIDSTOPPED(status)：如果引起返回的子进程当前是停止的，返回真
* WSTOPSIG(status)：返回引起该子进程停止的信号的编号，如果WIDSTOPPED(status)为真才定义
* WIFCONTTINUED(status)：如果子进程收到SIGCONT信号重新启动，返回真

4 错误条件

如果调用进程没有子进程，那么返回-1，设置errno为ECHILD；如果waitpid函数被一个信号中断，返回-1，并设置errno为EINTR

5 wait函数

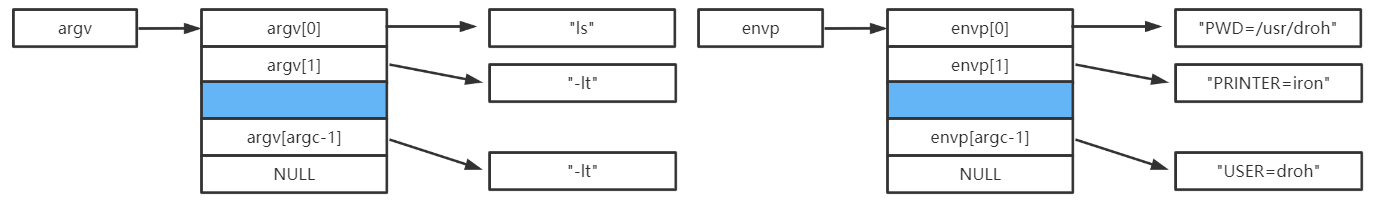
wait函数是waitpid函数的简单版本，等价于waitpid(-1，&status，0)

pid\_t wait(int \*statusp);

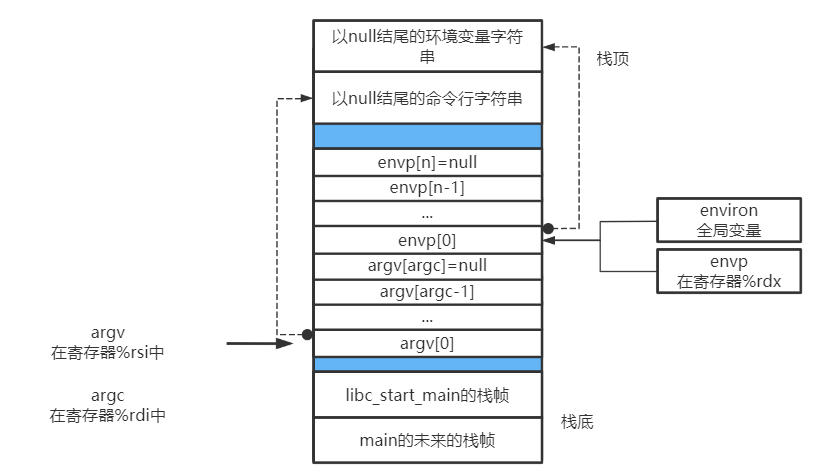
#### 8.3.3 执行函数

int execve(const char \*filename，const char \*argv[]，const char \*envp[])；

execve函数加载并运行可执行目标文件filename，调用一次从不返回。（只有出了错误才返回）。



main函数有三个参数：argc，给出argv[]数组中非空指针的数量；argv指向argv[]数组中的第一个条目；envp，指向envp[]第一个条目。



#### 8.3.4 进程相关函数

unsigned int sleep(unsigned int secs);

// sleep函数将一个进程挂起一段指定时间，如果时间已到，返回0；如果被信号中断提前返回，返回值是剩下的要休眠秒数。

int pause(void); //让调用函数休眠，直到该进程收到一个信号

pid\_t getpid(void); // 获取进程ID，PID是一个int类型

pid\_t getppid(void); // 获取父进程ID

**8.4 信号**

|  |  |  |  |
| --- | --- | --- | --- |
| ID | 名称 | 默认行为 | 相应事件 |
| 2 | SIGINT | 终止 | 来自键盘的中断 |
| 9 | SIGKILL | 终止 | 杀死程序，该信号不能被捕获，被忽略 |
| 11 | SIGSEGV | 终止 | 无效的内存引用（段错误） |
| 14 | SIGALRM | 终止 | 来自alarm函数的定时器信号 |
| 17 | SIGCHLD | 忽略 | 一个子进程停止或者终止 |

信号：软件形式的异常，允许进程和内核中断其他进程

传送信号到目的进程由两个不同步骤组成：

* 发送信号：内核通过更新目的进程上下文的某个状态，发送一个信号给目的进程。发送信号可能有两种原因：内核检测到一个系统事件，比如除0或子进程终止；进程调用了kill函数。
* 接收信号：目的进程被内核强迫以某种方式对信号的发送做出了反应时，它就接收了信号。

一个发出而没有被接收的信号叫做待处理信号。任何时刻，一种类型至多只会有一个待处理信号，多余的会被丢弃。进程可以有选择性地阻塞接收某种信号，信号被阻塞时，进程仍然可以发送，但是待处理信号不会被接收。

#### 8.4.1 发送信号

1 进程组

每个进程都只属于一个进程组，进程组是由一个正整数ID标识。

pid\_t getpgrp(void); //函数返回当前进程的进程组ID

默认情况下，一个子进程和它的父进程同属于一个组，一个进程可以通过使用setpgid函数改变自己或其他进程的进程组。

int setpgid(pid\_t pid, pid\_t pgid);

将pid的进程组改为pgid，如果pid是0，则使用当前进程的PID；如果pgid是，使用pid的进程PID作为进程组ID。

2 用kill程序发送信号

可以使用kill程序向另外的进程发送任意信号，kill -9 15213，向进程15213发送信号9，一个负的PID，会发送给进程组PID中的每个进程。

3 从键盘发送信号

shell使用作业这个概念表示对命令行求值而创建的进程。在任何时候只有一个前台作业和0个或多个后台作业

* ctrl+C表示向前台进程组的每个进程发送SIGINT信号，默认情况是终止前台作业。
* ctrl+Z会发送SIGTSTP信号到前台进程组的每个进程，默认是挂起前台作业。

4 用kill函数发送信号

int kill(pid\_t pid，int sig)；

* 如果pid > 0，则发送信号给进程pid
* 如果pid = 0，那么发信号给调用进程所在进程组的每个进程，包括自己
* 如果pid < 0，发送信号给进程组|pid|中的每个进程

5 用alarm函数发送信号

unsigned int alarm(unsigned int secs);

进程可以通过调用alarm函数想自己发送SIGALRM信号

#### 8.4.2 接收信号

当内核把进程p从内核模式切换到用户模式，会检查进程p的未被阻塞的待处理信号的集合。

* 如果这个集合为空，则将控制转移到p；
* 如果集合非空，内核选择集合中的某个信号k（通常是最小的），并且强制p接收k。进程收到信号会采取某种行为，一旦完成行为就会将控制传递回p的逻辑控制流中的下一条命令。每个信号类型都有以下默认行为：进程终止；进程终止并转储内存；进程停止直到被SIGCONT信号重启；进程忽略该信号。

typedef void (\*sighandler\_t)(int);

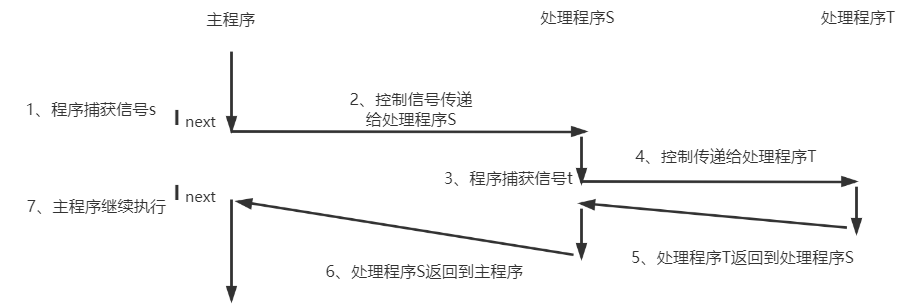
sighandler\_t signal(int signum，sighandler\_t handler);

如果成功则返回前次处理程序的指针，如果失败返回SIG\_ERR不设置errno

signal函数通过下列三种方式改变和信号signum相关联的行为：

* 如果handler是SIG\_IGN，忽略类型为signum的信号
* 如果handler是SIG\_DFL，类型为signum的信号恢复默认行为
* 如果handler是用户自定义的函数指针，只要进程接收到类型为signum的信号，就会调用这个程序。当处理程序执行完return后，会返回到原进程被信号中断位置。

信号处理程序可以被其他信号处理程序中断：



#### 8.4.3 阻塞和解除阻塞信号

Linux提供阻塞信号的模式：

隐式阻塞机制：内核默认阻塞任何当前处理程序正在处理的待处理信号。

显示阻塞机制：通过sigprocmask函数和它的辅助函数，明确地阻塞和解除阻塞选定的信号。

|  |
| --- |
| int sigprocmask(int how, const sigset\_t \*set, sigest\_t \*oldest); |

how值：

SIG\_BLOCK：把set中的信号添加到blocked中 blocked = blocked | set

SIG\_UNBLOCK：从blocked中删除set中的信号 blocked = blocked & ~set

SIG\_SETMASK：blocked = set

#### 8.4.5 编写信号处理程序

1 安全的信号处理

* 处理程序尽可能简单
* 在处理程序中只调用异步信号安全的函数。异步信号安全的函数：只访问局部变量；不能被信号处理程序中断。
* 保存和恢复errno
* 当处理程序和主程序共享一个全局变量时，阻塞所有的信号，保护对共享全局数据结构的访问
* 用volatile声明全局变量。volatile定义一个变量会通知编译器不要缓存这个变量。
* sig\_atomic\_t声明标志。C提供一种整型数据类型sig\_atomic\_t，对它的读写保证会是原子的。

2 正确的信号

未处理的信号从不排队。因为pending位向量中每种类型只有一位，所以每种类型最多只有一种未处理的信号，多余的未处理信号会被丢弃。

|  |
| --- |
| 例：下面这个程序输出什么  F:\pdf阅读\Personal-Study-Materials\post-graduate\csapp\picture\信号屏蔽练习.png  213  一开始使用printf打印2  首先父进程fork后，子进程一直停在while循环；  父进程用Kill函数发送信号，然后父进程进入阻塞，子进程接收信号，counter--，输出1  子进程结束，父进程唤醒，使用printf 打印counter++，由于子进程有独立的副本，父进程的counter值依然为2，输出3 |

子进程结束，父进程唤醒，使用printf 打印counter++，由于子进程有独立的副本，父进程的counter值依然为2，输出3

**8.5 非本地跳转**

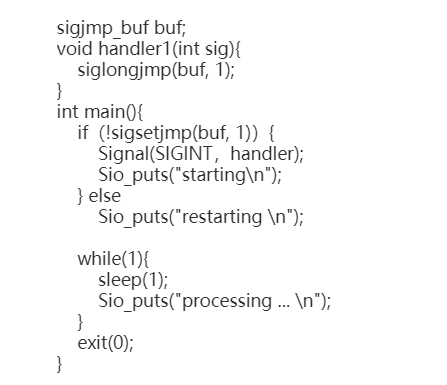
C语言提供了一种用户级异常控制，非本地跳转。

|  |
| --- |
| int setjmp (jmp\_buf env)；  int sigsetjmp（sigjmp\_buf env, int savesigs）;  int longjmp (jmp\_buf env, int retval);  int siglongjmp (sigjmp\_buf env, int retval); |

setjmp函数在env缓存区中保存当前调用环境，以供后面的longjmp使用。setjmp返回值只可以用在条件测试或者switch语句中。一次调用，多次返回

longjmp函数从缓存区中恢复调用环境，然后触发一个最近一次初始化env的setjmp调用返回。然后setjmp返回有非0的retval。一次调用，从不返回。

sigsetjmp和siglongjmp是可以被信号处理的版本。

例如

在程序第一次启动时，sigsetjmp函数初始调用保存环境信息。然后主函数进入循环。当用户键入ctrl+c时，内核发送一个SIGINT信号给这个进程，这个进程捕获SIGINT信号使用siglongjmp跳转，跳转到sigsetjmp函数，sigsetjmp获取siglongjmp的retval(即1)，取非后，则进入另一个分支，输出restarting，随后又进入无限循环。

**九、虚拟内存**

**9.1寻址**

#### 9.1.1 地址空间

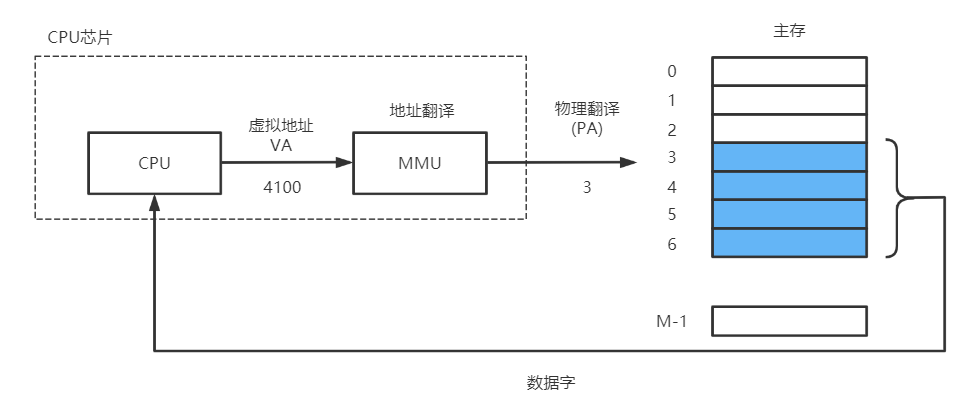
地址空间是一个非负整数地址的有序集合：{0，1，2……}

CPU从一个有N = 2n个地址的地址空间生成虚拟地址：{0，1，……N-1}

一个系统还有物理空间，对应于物理内存的M个字节：{0，1，……，M-1}

|  |  |  |
| --- | --- | --- |
| 虚拟地址位数（n） | 虚拟地址数（N） | 最大可能的虚拟地址 |
| 8 | 28 = 256 | 256-1=255 |
| 16 | 216 = 64k | 216-1 = 64 k -1 |
| 32 | 232 = 4G | 232 -1 = ？G - 1 |
| 64 | 264 = 16384 p | 264 – 1 = 16384 p - 1 |

#### 9.1.2 虚拟寻址

CPU生成一个虚拟地址，虚拟地址（VA）通过内存管理单元MMU转换成适当的物理地址（PA），然后通过内存总线传递给主存。主存取出物理地址4字节，并返回给CPU。

**9.2虚拟内存作为缓存的工具**

VM系统将虚拟内存分割成虚拟页，每个虚拟页大小为P = 2*P*字节。类似物理内存分割成物理页（PP），大小也为P。任意时刻，虚拟页面的集合都分为三个不相交的子集：

* 未分配的：VM系统还分配的页
* 缓存的：当前已缓存在物理内存中的已分配页
* 未缓存的：未缓存在物理内存中已分配页。

DRAM缓存的组织结构：

SRAM表示位于CPU和主存之间的L1，L2，L3高速缓存

DRAM表示虚拟内存系统的缓存，在主存中缓存虚拟页。

#### 9.2.1 页表

页表就是一个页表条目（PTE）的数组，放在内存中。

每个PTE是由一个有效位和一个n位地址组成。有效位表明该虚拟页是否被缓存在DRAM中，如果设置了有效位，地址字段表示DRAM相应的物理页起始地址；如果没有设置，空地址表示虚拟页还未分配；否则，这个地址指向虚拟页在磁盘的起始位置。

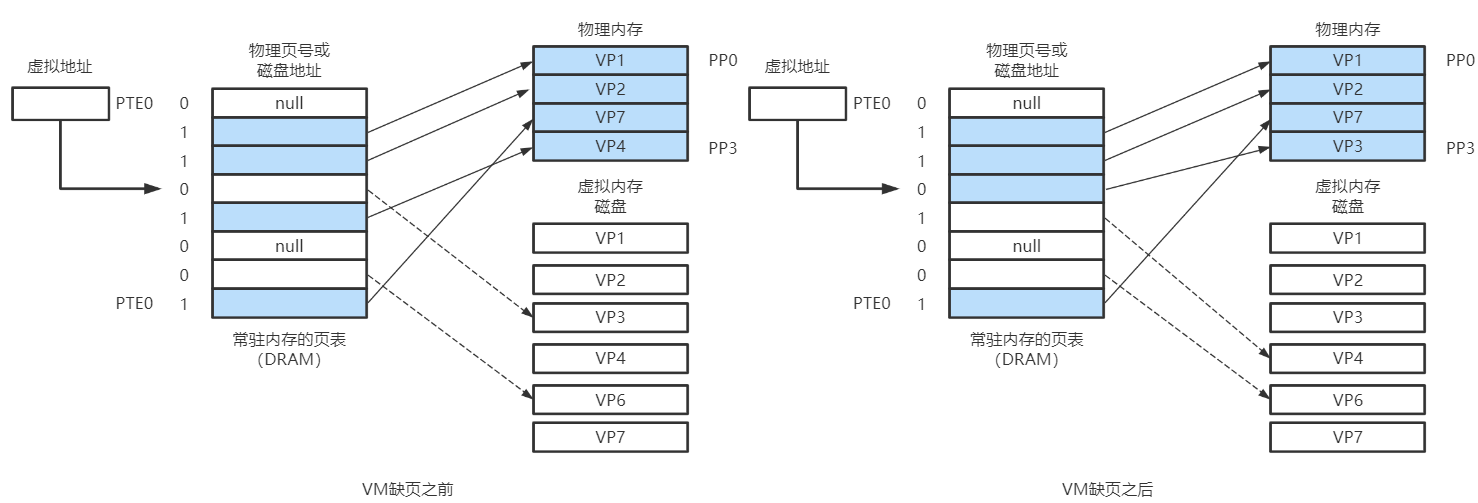
练习：确定下列虚拟地址n和页大小P的组合所需要的PTE数量

|  |  |  |
| --- | --- | --- |
| n | P = 2P | PTE数量 |
| 16 | 4K | 16 |
| 16 | 8K | 8 |
| 32 | 4K | 1 M = 220 |
| 32 | 8K | 512 K=219 |

#### 9.2.2 页命中和缺页

当CPU想要读包含VP2中的虚拟内存的一个字时，如果VP2缓存在DRAM中，就称为页命中。如果DRAM缓存不命中则称为缺页。

CPU引用VP3中的一个字，VP3并未缓存在DRAM中。地址翻译硬件从内存中读取PTE3，从有效位推断出VP3未被缓存，触发一个缺页异常。调用缺页异常处理程序会选择一个牺牲页，在这个例子中就是存放在PP3中的VP4。如果VP4已经被修改，内核就会将它复制回磁盘。



接下来，内核从磁盘复制VP3到内存PP3，更新PTE3，随后返回。

局部性：

在整个运行过程中，程序引用的不同页面的总数可能超出物理内存总大小，但是局部性原则保证在任意时刻，程序将趋向于在一个较小的活动页面集合上工作，这个集合是工作集合。

如果工作集的大小超出物理内存大小，程序将产生抖动。

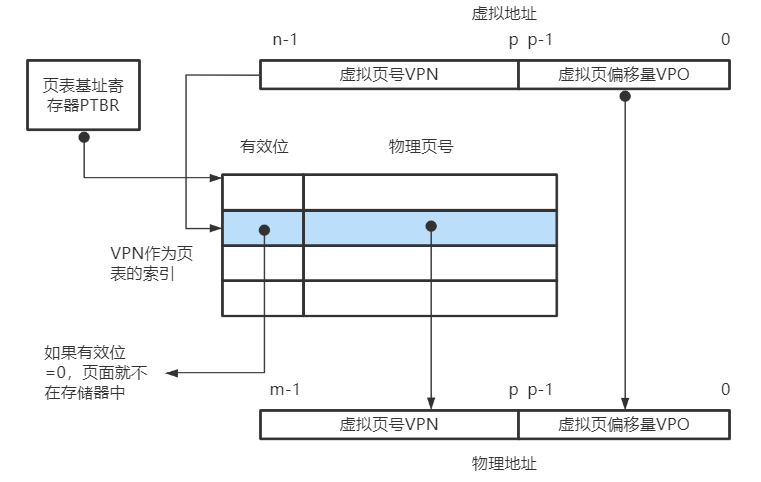
**9.3 地址翻译**

地址翻译是一个N个元素的虚拟地址空间VAS中的元素和一个M元素的物理地址空间PAS中元素之间的映射：MAP：VAS→PAS ∪Ø

这里：

|  |  |  |  |
| --- | --- | --- | --- |
| 基本参数 | | | |
| 符号 | | 描述 | |
| N = 2n | | 虚拟地址空间中的地址数量 | |
| M = 2m | | 物理地址空间中的地址数量 | |
| P = 2p | | 页的大小 | |
| 虚拟地址的组成部分 | | 物理地址的组成部分 | |
| 符号 | 描述 | 符号 | 描述 |
| VPO | 虚拟页面偏移量 | PPO | 物理页面偏移量 |
| VPN | 虚拟页号 | PPN | 物理页号 |
| TLBI | TLB索引 | CO | 缓冲块内字节偏移 |
| TLBT | TLB标记 | CI | 高速缓冲索引 |
|  |  | CT | 高速缓冲标记 |

#### 9.3.1 利用页表实现映射

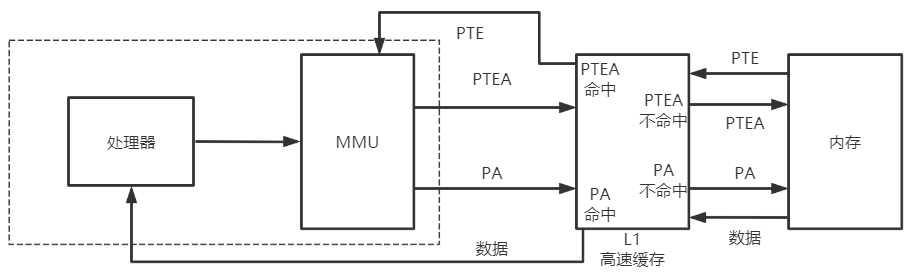


练习：给定一个32位的虚拟地址和24位的物理地址，确定VPN，VPO，PPN和PPO的位数。

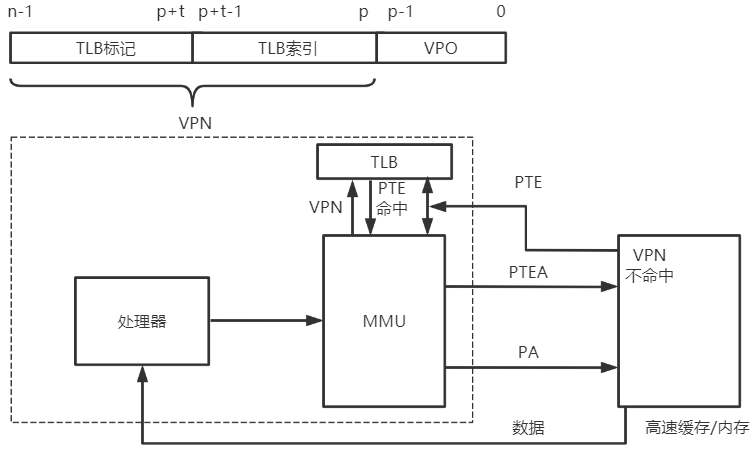
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P | VPN位数 | VPO位数 | PPN位数 | PPO位数 |
| 1KB | 22 | 10 | 14 | 10 |
| 2KB | 21 | 11 | 13 | 11 |
| 4KB | 20 | 12 | 12 | 12 |
| 8KB | 19 | 13 | 11 | 13 |

#### 9.3.2 虚拟内存系统改进

1 将VM系统与高速缓存结合起来，地址翻译发生在高速缓存查找之前：

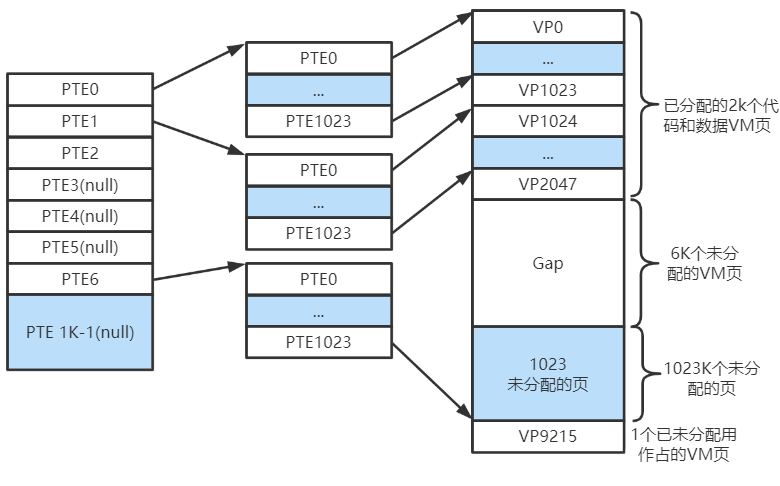


2 利用TLB加速翻译

在MMU中包括一个关于PTE的小的缓存，被称为TLB。由于TLB在芯片上的MMU执行，因此TLB的速度通常快于L1cache，同时TLB具有高度的相联度：

* CPU产生一个虚拟地址
* MMU从TLB中取出相应的PTE，如果命中则继续下面步骤，如果不命中则从L1cache中取。
* MMU将这个虚拟地址翻译成物理地址，并将它发送至高速缓存
* 高速缓存/主存将请求的数据字返回

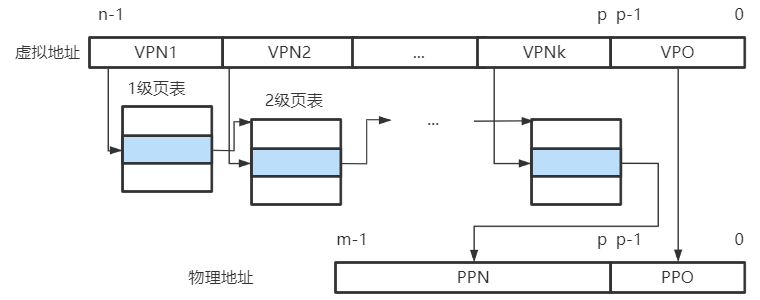
多级页表：

假设32位虚拟空间被分为4KB的页，每个PTE是4字节，一级页表每个PTE负责映射虚拟地址空间中一个4MB的片，这里每个片都是由1024个PTE组成。假设地址空间是4GB，1024个PTE足够覆盖。

二级页表每个PTE负责映射一个4KB的虚拟内存页面，每个一级和二级页表都是4KB大小。这种方法从两个方面节约了内存：

* 如果一个一级页表PTE是空的则相应的二级页表就根本不存在，对于一个程序，4GB的虚拟地址大部分是未分配的。
* 只有一级页表总是在内存中；虚拟内存系统可以在需要时创建、页面调入或调出二级页表。

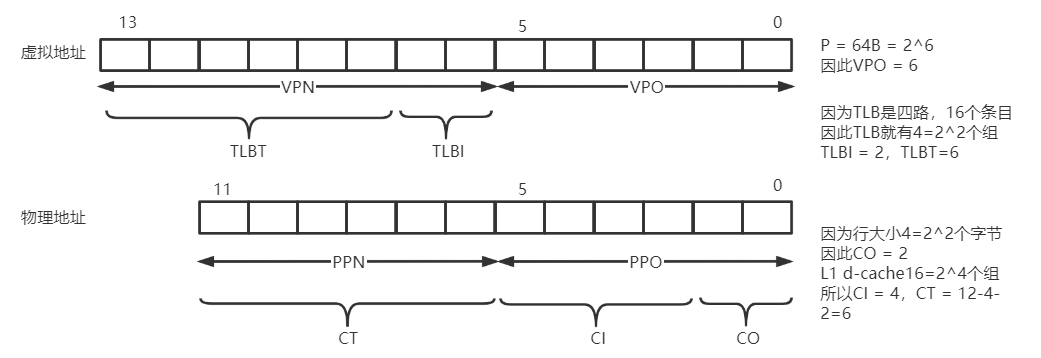
k级页表地址翻译类似：



#### 9.3.3 综合：端到端的地址翻译

为了保证可管理性，现做如下假设：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 寻址大小 | 虚拟地址 | 物理地址 | 页面大小 | TLB相联度 | L1 d-cache |
| 内存访问针对1字节 | n = 14 | m = 12 | P = 64B | 4路，共有16个条目 | 直接映射，行4字节，16组 |

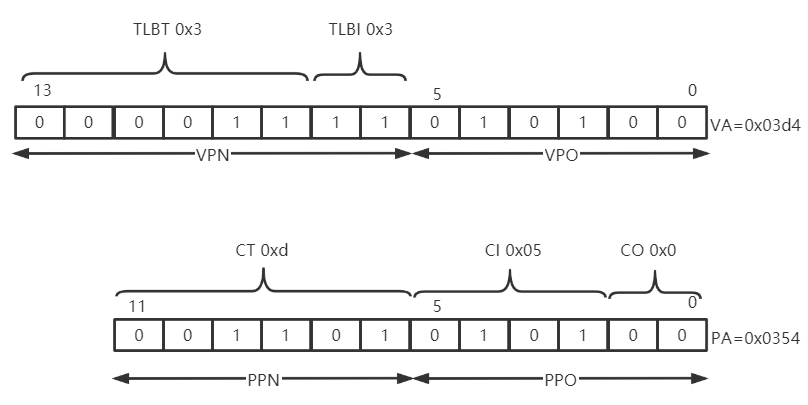


|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 组 | 标记位 | PPN | 有效位 | 标记位 | PPN | 有效位 | 标记位 | PPN | 有效位 | 标记位 | PPN | 有效位 |
| 0 | 03 | - | 0 | 09 | 0D | 1 | 00 | - | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | - | 0 | 04 | - | 0 | 0A | - | 0 |
| 2 | 02 | - | 0 | 08 | - | 0 | 06 | - | 0 | 03 | - | 0 |
| 3 | 07 | - | 0 | 03 | 0D | 1 | 0A | 34 | 1 | 02 | - | 0 |

**TLB：四组，16个条目，四路组相连**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VPN | PPN | 有效位 | VPN | PPN | 有效位 | 索引 | 标记位 | 有效位 | 块0 | 块1 | 块2 | 块3 |
| 00 | 28 | 0 | 08 | 13 | 1 | 00 | 19 | 1 | 99 | 11 | 23 | 11 |
| 01 | - | 1 | 09 | 17 | 1 | 01 | 15 | 0 | - | - | - | - |
| 02 | 33 | 0 | 0A | 09 | 1 | 02 | 1B | 1 | 00 | 02 | 04 | 08 |
| 03 | 02 | 0 | 0B | - | 0 | 03 | 36 | 0 | - | - | - | - |
| 04 | - | 0 | 0C | - | 0 | 04 | 32 | 1 | 43 | 6D | 8F | 09 |
| 05 | 16 | 1 | 0D | 2D | 1 | 05 | 0D | 1 | 36 | 72 | F0 | 1D |
| 06 | - | 0 | 0E | 11 | 1 | 06 | 31 | 0 | - | - | - | - |
| 07 | - | 0 | 0F | 0D | 1 | 07 | 16 | 1 | 11 | C2 | DF | 03 |
|  |  |  |  |  |  | 08 | 24 | 1 | 3A | 00 | 51 | 89 |
| **页表：只展示了16个PTE（上）** | | | | | | 09 | 2D | 0 | - | - | - | - |
| **高速缓存：16个组，4字节块，直接映射**  **(右)** | | | | | | 0A | 2D | 1 | 93 | 15 | DA | 3B |
| 0B | 0B | 0 | - | - | - | - |
|  |  |  |  |  |  | 0C | 12 | 0 | - | - | - | - |
|  |  |  |  |  |  | 0D | 16 | 1 | 04 | 96 | 34 | 15 |
|  |  |  |  |  |  | 0E | 13 | 1 | 83 | 77 | 1B | D3 |
|  |  |  |  |  |  | 0F | 14 | 0 | - | - | - | - |

CPU执行一条读地址0x3d4的加载指令。

开始时MMU从虚拟地址中抽取VPN（0x0F）。

TLB从VPN中抽取出TLBI（0x03）和TLBT（0x03），进行匹配，命中，然后将缓存的PPN（0x0D）返回给MMU。

MMU将TLB返回的PPN和虚拟地址的VPO组合，形成物理地址0x354

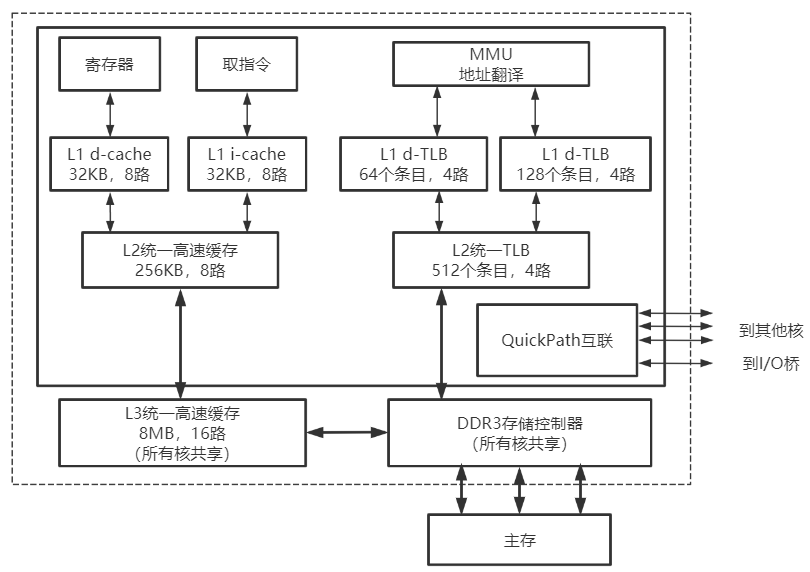
MMU发送物理地址给缓存，缓存从物理地址中抽取CO（0x00）CI（0x5）以及CT（0x0D）

由于组0x5中的标记与CT相匹配，缓存检测到命中，返回数据字节0x36。

练习：给出虚拟地址0x3d7，填空：

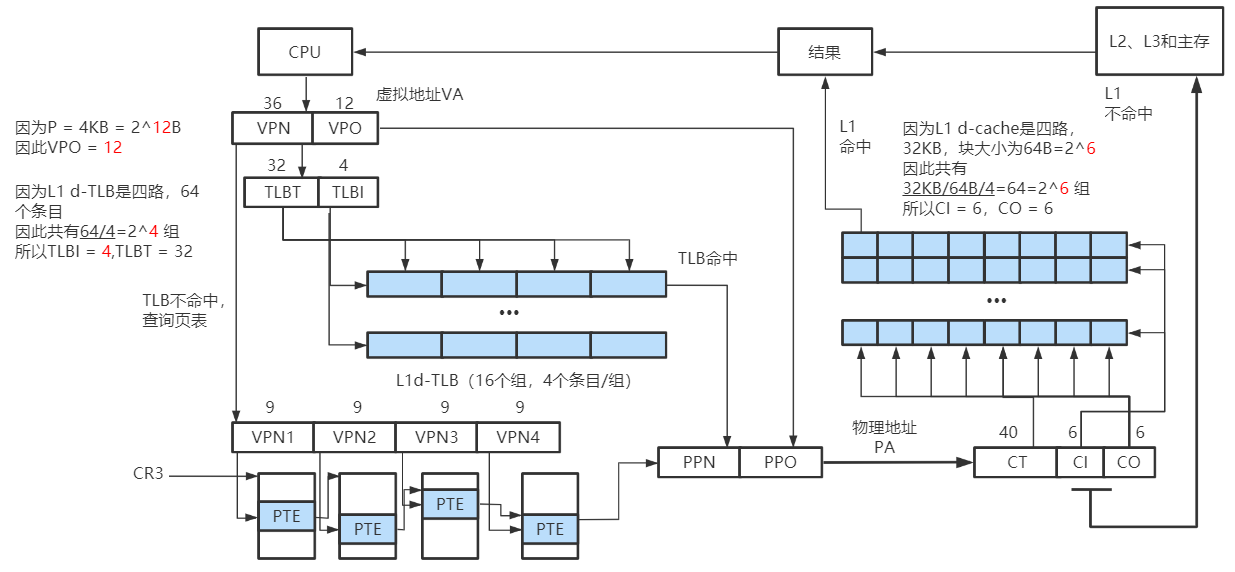
|  |  |  |  |
| --- | --- | --- | --- |
| 地址翻译 | | 物理内存引用 | |
| 参数 | 值 | 参数 | 值 |
| VPN | 0x0f | 字节偏移 | 0x03 |
| TLB索引 | 0x03 | 缓存索引 | 0x05 |
| TLB标记 | 0x03 | 缓存标记 | 0x0d |
| TLB命中 | 是 | 缓存命中 | 是 |
| 缺页 | 否 | 返回的缓存字节 | 0x1D |
| PPN | 0D |  |  |

**9.4 Intel Core I7/Linux内存系统**

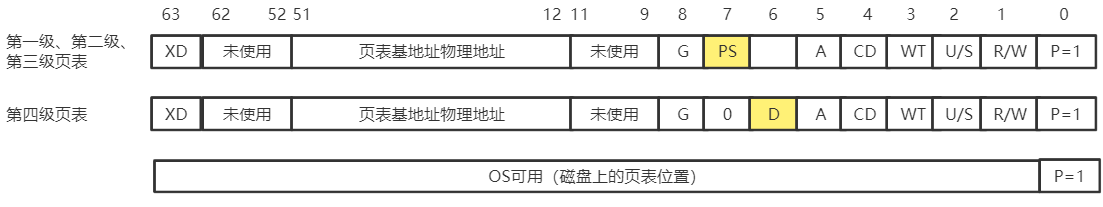
CoreI7实现支持48位虚拟地址和52位物理地址空间。左图给出了CoreI7内存系统的重要部分：

L1，L2，L3高速缓存是物理寻址的，块大小是64字节；TLB是虚拟寻址的，四路组相联

#### 9.4.1 Core I7地址翻译



CoreI7采用四级页表的结构，因此每个页表29=512个PTE，CR3控制寄存器指向第一级页表L1的起始位置。下图给出了页表中条目的格式：



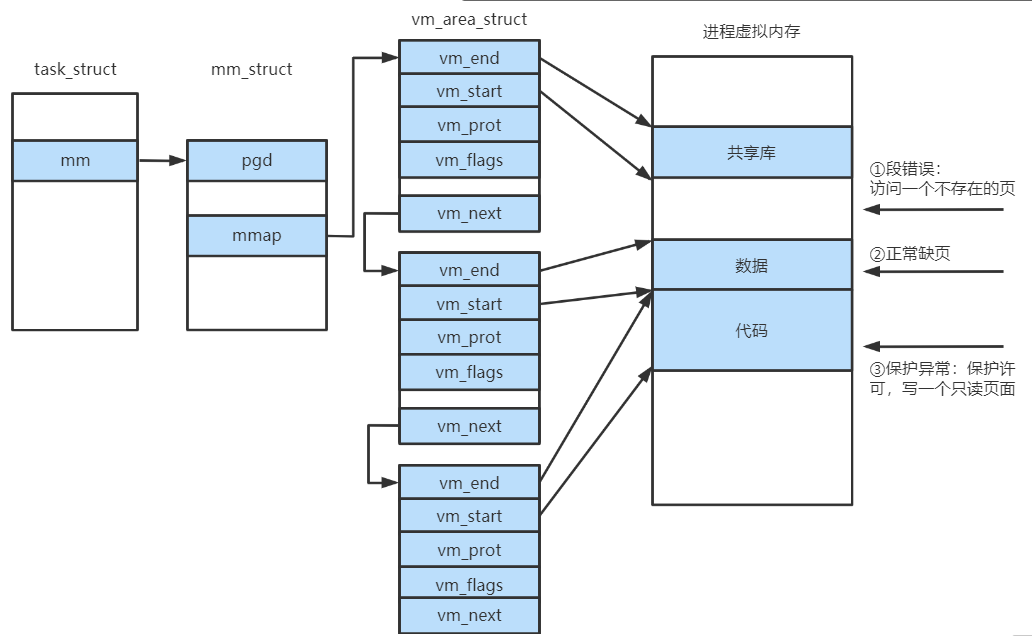
|  |  |
| --- | --- |
| 字段 | 描述 |
| P | 子页表是否在物理内存中 |
| R/W | 读写访问权限 |
| U/S | 用户或超级用户访问权限 |
| WT | 子页的直写或者写回策略 |
| CD | 能/不能缓存 |
| A | 引用位 |
| D | 修改位，告诉内核在复制替换页时是否要写回（只对第4级页表定义） |
| PS | 页大小为4KB或者4MB（只对第1级页表定义） |
| G | 全局页，不从TLB中替换出去 |
| Base addr | 物理基地址最高40位 |
| XD | 能/不能从这个页中取指令 |

PTE有3个权限位，控制对页访问，R/W确定读写权限，U/S确定用户权限，XD能不能从这个子页中取数据。

优化地址翻译：

仔细观察可以发现，VPO = 12，刚好等于L1缓存CI + CO的和，这是因为CPU发送VPN给MMU，同时发送VPO给L1 d-cache。当MMU向TLB请求PTE时，L1高速缓存正忙着利用VPO位查找相应的组，读出8个对应的标记位和数据字。当MMU获得PPN时，缓存已经可以试把这个PPN与8个标记中一个进行匹配。

#### 9.4.2 Linux虚拟内存系统



1 Linux虚拟内存区域

任务结构中的一个entry指向mm\_struct，描述了虚拟内存的当前状态。pgd指向一级页表的基址，mmap指向一个vm\_area\_struct链表，当内核运行这个程序时，pgd就放在CR3中。

* vm\_start：指向这个区域的起始处
* vm\_end：指向这个区域的结束处
* vm\_prot：描述这个区域包含的所有读写权限
* vm\_flags：描述这个区域是共享的，还是私有的
* vm\_next：指向链表中下一个区域结构

2 Linux缺页异常（如图所示）

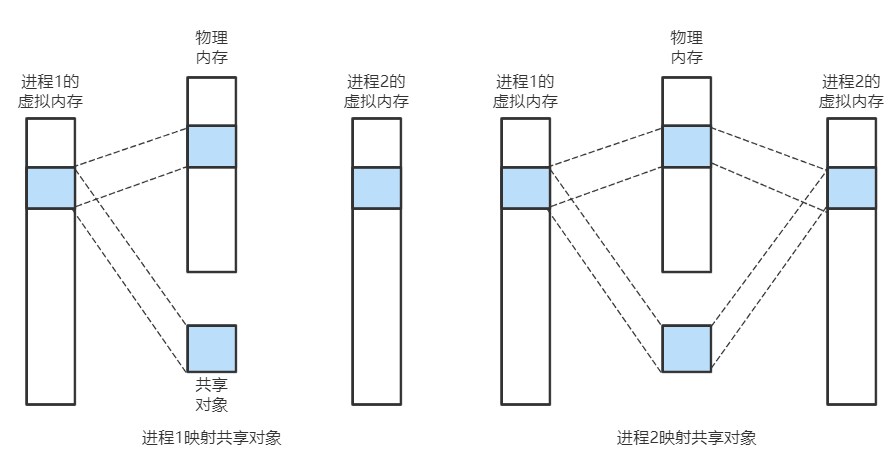
**9.5 内存映射**

内存映射：将一个虚拟内存区域与磁盘上的对象关联起来，以初始化这个虚拟内存区域的内容。虚拟内存区域可以映射两种对象：普通文件；匿名文件。

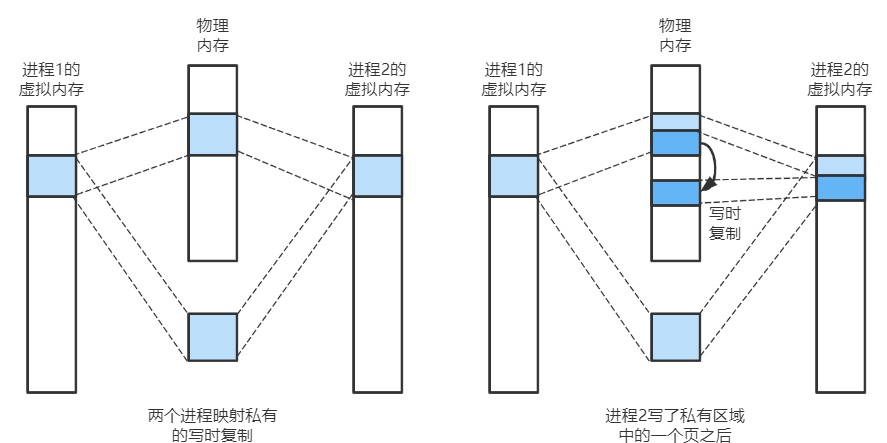
#### 9.5.1 共享对象

一个对象被映射到虚拟内存的某个区域，要么是共享对象，要么是私有对象。

假设进程1将一个共享对象映射到内存中，进程2将同一个共享对象映射到地址中。关键点在于即使对象被映射到了多个共享区域，物理内存也只需要存放共享对象的一个副本。



私有对象开始时生命周期和共享对象一样。只要有一个进程试图写私有区域的某个页面，就会触发一个保护故障。故障处理程序就会在物理内存中创建这个页面的新副本，更新页表条目指向副本。通过延迟私有对象中的副本直到对象更改时，这种技术称为写时复制，可以充分的利用内存。



#### 9.5.2 回看进程控制函数

1 fork函数

当fork函数被调用时，内核为新进程创建各种数据结构，并分配一个唯一PID。同时给这个新进程创建了mm\_struct，区域结构和页表的副本，并将这两个进程的区域结构都标记为私有对象的写时复制。

2 execve函数

假设execve函数执行execve(a.out，null，null);，a.out代替当前程序。加载运行a.out需要以下步骤：

* 删除已存在的用户区域。
* 映射私有区域：为新程序代码，数据段，bss和栈区域创建新的结构。
* 映射共享区域：如果a.out与共享对象链接，这些对象都是动态链接到这个程序，然后映射到用户虚拟内存空间中。
* 设置程序计数器：设置上下文中的程序计数器。

#### 9.5.3 使用函数操作内存映射

1 mmap函数

|  |
| --- |
| void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset); |

创建一个新的虚拟内存区域，地址从start开始，，并将文件描述符fd指定的对象的一个连续的片映射到这个新的区域，连续片的长度是length，从距文件开始处偏移量offset字节的地方开始。参数prot包含权限位相等于vm\_prot:

* PROT\_EXEC：这个区域内的页面由可以被CPU执行的指令组成
* PROT\_READ：这个区域内的页面可读
* PROT\_WRITE：这个区域内的页面可写
* PROT\_NONE：这个区域内的页面不能被访问

flags表示对象类型：MAP\_ANON表示被映射对象是一个匿名对象；MAP\_PRIVATE表示对象是一个私有的、写时复制对象；MAP\_SHARED表示对象是一个共享对象。

2 munmap函数

|  |
| --- |
| int munmap(void \*start, size\_t length); |

删除从虚拟地址start开始，接下来length字节组成的区域。

**9.6 动态内存分配**

当运行时需要额外虚拟内存时，linux使用动态内存分配器。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。

分配器有两种类型：

显示分配器：显示释放任何已分配的块，比如C中的malloc

隐式分配器：分配器检测一个已分配的块何时不再引用，就释放这个块，也叫垃圾收集器。

#### 9.6.1 malloc和free函数

|  |
| --- |
| void \*malloc(size\_t size); |

malloc函数返回一个指针，指向大小至少为size字节的内存块。动态内存分配器malloc通过mmap和munmap显示分配和释放内存，使用sbrk函数也可以达到目的。

|  |
| --- |
| void \* sbrk(intptr\_t incr); |

sbrk函数通过将brk指针增加incr来扩展和收缩堆，如果成功返回brk旧值，失败返回-1；如果incr = 0，返回当前brk；如果incr < 0，返回值指向距新堆顶abs(incr)字节处

|  |
| --- |
| void free(void \*ptr); |

free函数释放已分配的堆块，ptr必须指向一个从malloc、calloc、realloc获得的已分配块的起始位置。

#### 9.6.2 分配器的要求和目标

显示分配器必须在下述约束条件下工作：

* 处理任意请求序列：不能假设所有的分配都有相匹配的释放。
* 立即响应请求：不允许分配器为了提高性能重新排列或者缓存请求。
* 只使用堆：分配器使用的任何非标量数据必须保存在堆里。
* 对齐块：分配器必须对齐块。
* 不修改已分配的块：分配器只操作或者改变空闲块。

分配器试图实现两个目标：吞吐量最大化，内存使用率最大化。

目标1：最大化吞吐率，一个分配请求的最糟运行时间与空闲块成正比，释放请求的运行时间是常数

目标2：最大化内存利用率，如果一个应用程序请求一个p字节的块，得到的已分配的有效载荷是p字节。在请求Rk完成后，聚集有效载荷表示为Pk，为当前已分配载荷之和，而Hk表示堆的当前大小前k+1个峰值利用率，表示为Uk，可以通过下式得到：

分配器的目标是在整个序列中使峰值利用U最小。

碎片：造成堆利用率很低的主要原因，有内部碎片和外部碎片

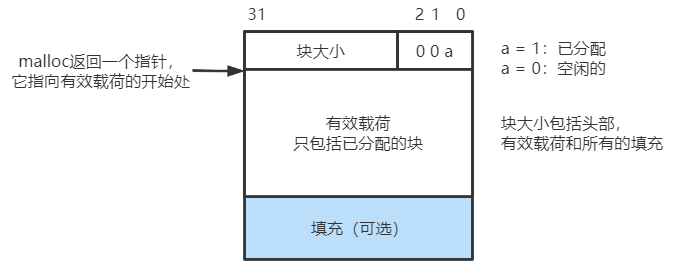
内部碎片：已分配块比有效载荷大时，比如为了满足对齐要求。内部碎片量化是已分配块大小和它们的有效载荷之差。

外部碎片：当空闲内存合计起来足够满足一个分配请求，但是没有一个单独的块可以处理这个请求。外部碎片难以量化且不可预测。因此分配器总是试图维持少量的大空闲块，而不是大量的小空闲块。

一个分配器要在吞吐率和利用率之前把握平衡，就必须考虑以下问题：

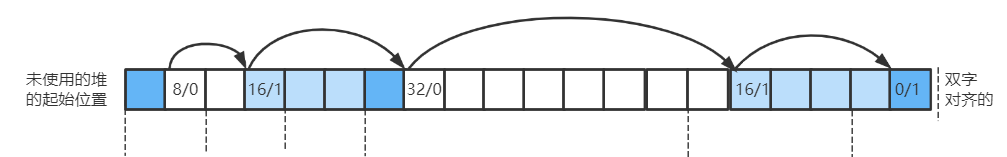
* 空闲块组织：如何记录空闲块
* 放置：如何选择一个合适的空闲块来放置一个新分配的块
* 分割：将一个新分配的块放置到某个空闲块后，如何处理这个空闲块的剩余部分。
* 合并：如何处理一个刚被释放的块。

#### 9.6.3 隐式空闲链表

头部编码了这个块的大小，如果强加一个双字对齐的约束条件，块大小就总是8的倍数，且块大小的最低3位总是0，因此只需要29位，剩余的3位来编码其他信息。

头部后面就是应用调用malloc时请求的有效载荷。

可以将堆组织为一个连续的已分配块和空闲块的序列，这种结构称为隐式空闲链表。分配器可以通过头部块大小来间接遍历堆中所有的块。分配后需要某种特殊标记的结束块，这里就是已分配但大小为0的块。



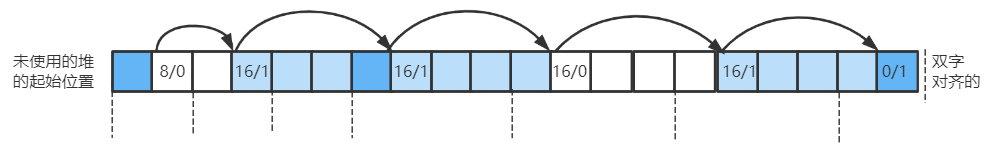
隐式空闲链表优点是简单，缺点是：搜索的时间与堆中已分配块和空闲块的总数呈线性关系，同时每次分配最少需要两个块。

#### 9.6.4 放置已分配的块和分割空闲块

应用请求一个k字节块，分配器搜索空闲链表，查找一个足够大可以放置所请求的块，这中搜索方式称为放置策略：一些常见的策略有

* 首次适配：从头开始搜索空闲链表，选择第一个合适的空闲块。
* 下一次适配：和首次适配类似，不过是从上一次查询结束的地方开始
* 最佳适配：检查每个空闲块，选择适合所需请求大小的最小空闲块。

如果分配器匹配情况不太好，通常会将这个块分割成两个部分，而剩下的一个部分会变成新的空闲块。



如果分配器不能为请求的块找到合适的空间，分配器就会通过sbrk函数，向内核申请额外的堆堆内存空间，然后将这个空间转化成空闲块中，插入到空闲链表中。

#### 9.6.5 合并空闲块

当分配器释放一个空闲块后，可能有其他空闲块与这个空闲块相邻，这些称为假碎片。为了解决假碎片，分配器必须合并相邻的空闲块，这个过程称为合并。有两种合并策略：立即合并；延迟合并。

* 立即合并：在空闲块释放后，分配器就合并相邻的空闲块，如果反复申请、释放，这可能会造成抖动。
* 延迟合并：分配器直到某个分配请求失败后，合并所有空闲的块。

分配器通过当前块的头部指向下一个块的头部来判断是否需要合并，如果是修改当前块中头部大小，这两个块可以在常数时间内进行合并；为了保证可以向前合并块，可以在每个块的结尾添加一个脚部。脚部是头部的一个副本，分配器可以通过检查脚部来判断前一个块是否需要合并。

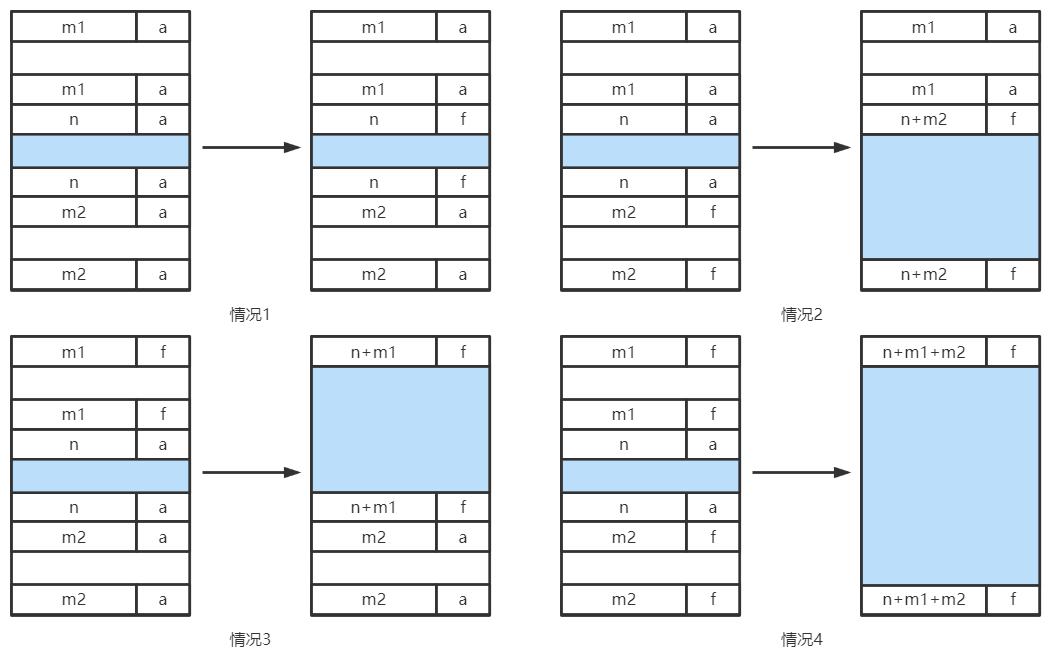
考虑分配器释放当前块可能出现的情况：

1） 前面的块和后面的块都是已分配的

2） 前面的块都是已分配的，后面的块都是空闲的

3） 前面的块都是空闲的，后面的块都是已分配的

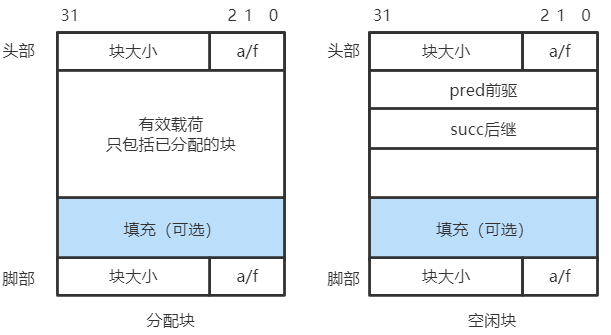
4） 前面和后面的块都是空闲的



如果每个块都需要脚部会浪费内存空间，因此只有空闲块才分配脚部，已分配块不需要脚步。

#### 9.6.6 显示空闲链表

由于块分配与堆块的总数呈线性关系，不太适合通用分配器，因此一种更好的方法是将空闲块组织为某种显式数据结构。



数据结构的指针放在空闲块里，可以使首次适配的分配时间从块总数的线性时间减少到空闲块总数的线性时间。释放一个块可能是线性也可能是常数，这取决于空闲块排列策略。

LIFO后进先出，新释放的块放置在链表的开始处。释放一个块是常数时间，如果使用边界标记，合并一个块也可以是常数时间完成。

按地址顺序，链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间搜索定位合适的前驱，但是有更高的内存利用率。

#### 9.6.7 分离的空闲链表

分离的空闲链表：维护多个空闲链表，其中每个链表中的块有大致相等的大小。一般的思路就是将所有可能的块分成一些等价类。

1 简单分离存储

每个等价类的空闲链表包含大小相等的块，每个块的大小是这个等价类中最大元素的大小。例如定义某个等价类为{17~32}，那么这个类的空闲链表全由大小为32的块组成。

分配一个给定大小的块，分配器会检查相应的空闲链表，如果链表非空，则简单分配其中第一块的全部；如果链表为空，则向内核申请一个额外大小的片。要释放一个块，则会简单的将这个块插入到相应的空闲链表的前部。

优点：分配和释放都是常数时间操作；每个块只有很少的内存开销；因为没有合并，所以不需要头部/脚部，已分配标记，且链表是单向，因此链表只需要单个指针。

缺点：容易造成内部碎片和外部碎片。

2 分离适配

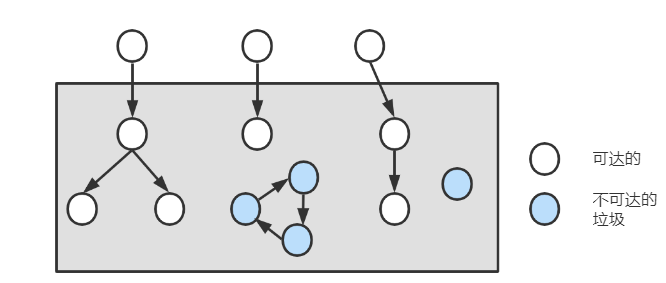
分配器维护一个空闲链表的数组，每个空闲链表是和等价类相关联的，并且被组织成某种类型的显示或隐式链表。每个链表包含潜在的大小不同的块。

为了分配一个块，必须确定请求的等价类，并且对适当的空闲链表做首次适配。如果找到，就分割它，然后将剩余的部分插入适当的链表中；如果找不到，就搜索下一个更大的等价类空闲链表，直到找到；如果空闲链表没有块，则向操作系统请求额外的堆内存。

优点：内存利用率高；搜索时间也相应的减少。

**9.7 垃圾收集**

垃圾收集器是一种动态内存分配器，自动释放程序不再需要的已分配块。

垃圾收集器将内存视为一张有向可达图。该图的节点被分成一组根节点和一组堆节点。每个堆节点对应于堆中的一个已分配块。有向边p→q意味着块p中的某个位置指向块q中的某个位置，根节点对应于不在堆中的位置，包含指向堆中的指针。这些可以是寄存器、栈里的变量，或者是虚拟内存中读写数据区域内的全局变量。

当存在一条从根节点出发并达到p的有向路径时，p是可达的。对于java的垃圾收集器，对应用如何创建和使用指针有很严格的控制，能够精确地维护可达图的表示，能够返回所有垃圾。而C这样的语言收集器不能维持可达图的精确表示，这称为保守的垃圾搜集器。

无论何时需要推空间时，应用都会调用malloc，如果malloc找不到一个合适的空闲块，就调用垃圾收集器，并通过调用free返回给堆。关键：收集器代替应用调用free。

**十、系统级I/O**

**10.1 Unix I/O**

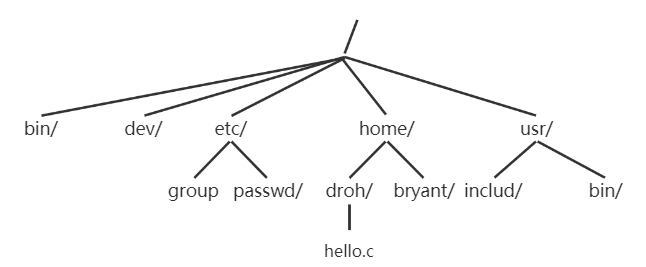
Linux 文件就是一个m字节的序列，所有的I/O设备都被模型化为文件，所有的输入和输出都被当作对相应文件的读和写来执行。这种将设备映射为文件，允许Linux内核统一一个低级、简单的应用接口，称为Unix I/O。包括以下操作：打开文件 ；Linux shell创建的每个进程都有三个打开的文件：标准输入（描述符0），标准输出（描述符1）和标准错误（描述符2）；改变文件当前指针位置；读写文件；关闭文件。

#### 10.1.1 文件

Linux文件有多种类型，这里只讨论三种：

* 普通文件：包含任意数据。应用程序通常分为文本文件（只含有ASCⅡ和Unicode字符的普通文件）和二进制文件。
* 目录：包含一组链接的文件，每个链接都将文件名映射到一个文件，这个文件可能是目录。每个目录至少包含两个文件：“.”是到该目录的链接；“..”是到父目录的链接。
* 套接字：用来与另一个进程进程跨网络通信的文件。

Linux内核将所有文件组织成一个目录层次结构，由名为/的根目录确定。



目录层次结构中的位置用路径名来指定：

* 绝对路径名：以一个斜杆开始，表示从根节点开始的路径，例如/home/droh/hello.c
* 相对路径名：表示从当前工作目录开始的路径，如果/home/bryant是当前工作目录，则../droh/hello.c表示访问hello.c

#### 10.1.2 打开和关闭文件

1 打开文件

一个进程通过要求打开相应的文件，宣告它想访问一个文件。Linux返回一个文件描述符。进程是通过调用open函数打开文件：

|  |
| --- |
| int open(char \*filename, int flags, mode\_t mode); |

open函数将filename转换为一个文件描述符，并返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。

flags参数指明进程访问文件方式：O\_RDONLY只读；O\_WRONLY 只写；O\_RDWR读写；O\_CREATE如果文件不存在则创建；O\_TRUNC如果文件已存在则截断；O\_APPEND设置文件指针位置到文件的结尾处。

mode参数指定了新文件的访问权限位(读、写、执行)。新文件的权限位会被设置为mode & ~umask。

2 关闭文件

进程通过调用close函数关闭一个打开的文件

|  |
| --- |
| int close (int fd); |

关闭一个已关闭的描述符会出错。

#### 10.1.3 读和写文件

|  |
| --- |
| ssize\_t read(int fd, void \*buf, size\_t n);  返回若成功为读取字节数，若出错为-1，若EOF为0  ssize\_t write(int fd, const void \*buf , size\_t n)  返回若成功则为写的字节数，若出错则为-1 |

read函数从描述符为fd的当前文件指针位置处复制最多n个字节到内存buf中。write函数从内存buf位置复制至多n个字节到描述符fd的当前文件位置。通过调用lseek函数可以显示地修改当前文件指针的位置。

在某些情况下read和write传送的字节比应用程序要求的少，这些不足值不表示有错误，出现这种情况的原因有：

* 读时遇到EOF：假设读一个文件，只含有20个字节，而应用以50字节进行读取，下一个read函数返回不足值为20，以后都返回0
* 从终端读文本行：如果打开文件是与终端相关联，则read函数每次传送一个文本行。
* 读和写网络套接字：如果打开文件是网络套接字，那么内部缓冲约束和较长的网络延迟会引起read和write返回不足值。

**10.2 RIO健壮读写**

RIO提供了两类不同函数：无缓冲的输入输出函数；带缓冲的输入函数

1 无缓冲的输入输出函数

|  |
| --- |
| ssize\_t rio\_readn(int fd, void \*usrbuf, size\_t n);  ssize\_t rio\_writen(int fd, void \*usrbuf, size\_t n); |

这些函数直接在内存和文件间传送数据，应用于二进制文件读写网络和网络读写到二进制文件。

rio\_readn函数从fd的当前文件位置传送n个字节到内存位置usrbuf，遇到EOF时会返回一个不足值。rio\_writen函数从位置usrbuf传送n个字节到描述符fd。对同一个fd，可以任意交错调用rio\_readn和rio\_writen。

2 带缓冲的输入函数

高效地从文件读取文本行和二进制函数。

|  |
| --- |
| void rio\_readinitb(rio\_t \*rp, int fd);  ssize\_t rio\_readlineb(rio\_t \*rp, void \*usfbuf, size\_t maxlen);  ssize\_t rio\_readnb(rio\_t \*rp, void \*usfbuf, size\_t n); |

每次打开一个描述符，都需要调用一次rio\_readinitb函数。它将fd和rp处的一个类型为rio\_t的读缓冲区联系起来。

rio\_readlineb函数从文件rp处读出下一个文本行，将它复制到内存位置usrbuf，并且用NULL字符来结束这个文本行。最多读maxlen-1个字节。

rio\_readnb函数从rp处最多读n个字节到内存位置usrbuf。

3 rio\_read函数

rio\_read函数是RIO读程序的核心，是Linux内核read函数带缓冲的版本。当调用rio\_read要求读n个字节的时，如果缓冲区为空，会先调用read函数填满它。一旦缓冲区非空，rio\_read就从读缓冲区复制n和rp->rio\_cnt中较小值个字节到用户缓冲区，并返回复制的字节数。

**10.3 标准I/O**

**10.4 读取文件元数据**