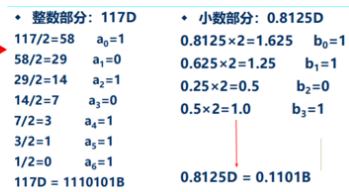
计算机系统期末：sdf老师部分

- 1. 信息处理与表示
  - 1.0. 进制转换

转换：左→右	B	O	D	H
B	\	与 B→H 的区别在于分成三个二进制数一组	$D=\sum_{i=-m}^{n-1} B_i * 2^i$	1110101. 11B= ? H 0111 0101 . 1100 7 5 . C ∴ 1110101. 11B=75. CH
O	与 H→B 的区别在于一位八进制数转为三位二进制数	\	$D=\sum_{i=-m}^{n-1} O_i * 8^i$	
D	这是降幂法；PS：除法，见右 例： 27D = ? B 27 11 3 3 1 -↓ -↓ -↓ -↓ -↓ 16 8 4 2 1 1 1 0 1 1 ∴ 27D = 11011B		\	
H	例如：EA. 11H= ? B E A . 1 1 1110 1010 0001 0001 ∴ EA. 11H=11101010. 00010001B		$D=\sum_{i=-m}^{n-1} H_i * 16^i$	\



  - 1.1. 信息储存
    - 1.1.1. 字数据大小

      1. 字长决定虚拟地址空间的最大值。字长w位则虚拟地址范围
      2. int32_t/int64_t分别为4/8个字节，不随编译环境变化
    - 1.1.2. 位级运算(C)：`~`取反，`|`或，`&`与，`^`异或

C 的表达式	二进制表达式	二进制结果
~0x41	~ [0100 0001]	[1011 1110]
~0x00	~ [0000 0000]	[1111 1111]
0x69&0x55	[0110 1001]&[0101 0101]	[0100 0001]
0x69|0x55	[0110 1001] | [0101 0101]	[0111 1101]

    - 1.1.3. 位移运算(C)：左结合 `x<<j<<k` = `(x<<j)<<k`

操作	值
参数 x	[01100011] [10010101]
x << 4	[0011*0000*] [0101*0000*]
x >> 4（逻辑右移）	[*0000*0110] [*0000*1001]
x >> 4（算术右移）	[*0000*0110] [*1111*1001]

      `x<<k`：左移k位，丢弃高k位，在右端补k个0

      `x>>k`：(逻辑右移)舍弃低k位，在左边补0；(算数右移)舍弃低k位，在左边补原最高位
  - 1.2. 整数表示
    - 1.2.1. 整数类型：注意一下几点即可

      负数的取值范围比正数的大1，默认有符号

- 1.2.2. 无符号数的编码：唯一性

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \qquad B2U_4([0001]) = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1$$

每一个无符号数都有唯一的编码，就是上图的B2U函数

- 1.2.3. 补码表示：唯一性
  - (1)表示规则

正数的补码: 符号 - 绝对值（与正数的原码相同）

$[+1]_{补码}$ = 0000 0001 = 01H

$[+127]_{补码}$ = 0111 1111 = 7FH

$[+0]_{补码}$ = 0000 0000 = 00H

负数的补码: 负数 X 用 2^n-|X| 表示

$[-1]_{补码}$ = 2^8-1 = 1111 1111 = FFH

$[-127]_{补码}$ = 2^8-127 = 1000 0001 = 81H

  - (2)求法

一种简单方法：

（1）写出与该负数相对应的正数的补码

（2）按位求反

（3）末位加1

[-1]补=?

[+1]补= 0000 0001

1111 1110

[-1]补= 1111 1111

  - (3)补码符号扩展问题

机器字长8位，$[-46]_{补码}$ = 1101 0010B = D2H

机器字长16位，$[-46]_{补码}$ = 1111 1111 1101 00010B = FFD2H

正数前边补0，复数前边补1

  - (5)表示范围

n位补码的表数范围： $-2^{n-1} \le N \le 2^{n-1}-1$

n=8 $-128 \le N \le 127$ **2^8个数**

n=16 $-32768 \le N \le 32767$ **2^{16}个数**

- 1.2.4. 符号数/无符号数转换

(1)无符号有符号

无符号高位为0，有符号=无符号；无符号高位为1，取补码为有符号数。

(2)有符号无符号

有符号高位为0，无符号=有符号；有符号高位为1，取补码为无符号数。

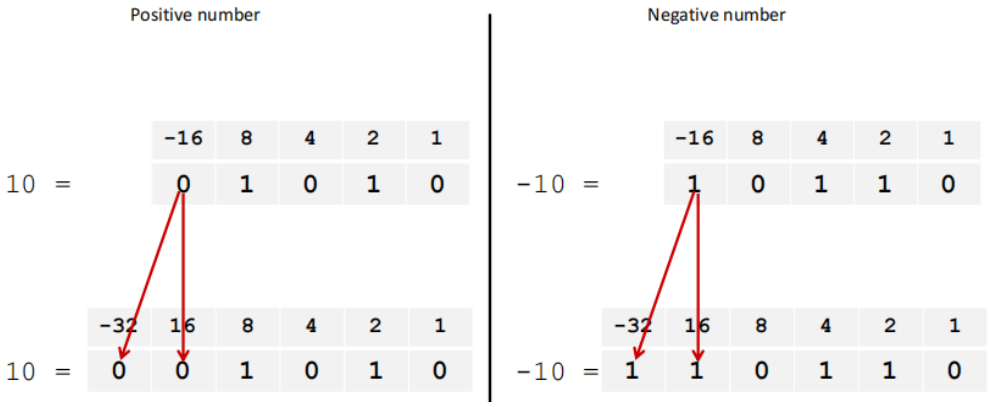
- 1.2.5. C语言中的有无符号数

```
int a = 12345;              // 这是一个有符号整数，默认
unsigned int c = 12345U;    // 这是一个无符号整数，U是代表无符号的后缀
```

```
int e = 12345U;                 // 编译器警告
```

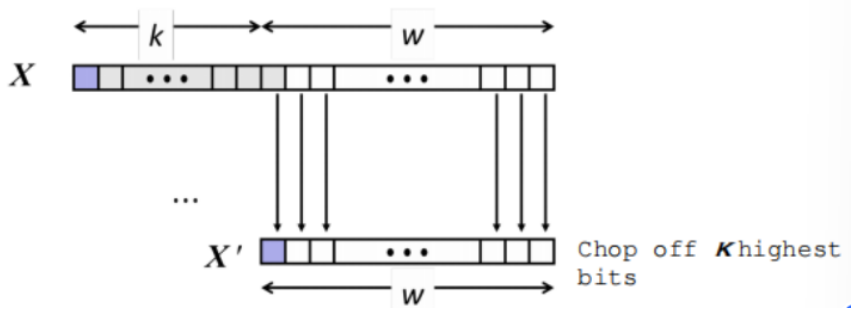
有/无符号混合运算中，有符号会被隐式强制转换为无符号

- 1.2.6. 扩展一个数字的位表示



1. 扩展无符号数(0扩展)：扩展k位，高位加k个0
2. 扩展有符号数(补码数)(符号扩展)：扩展k位，最高位加k个最高位(上图示例)

- 1.2.7. 截断数字



1. 截断无符号数：直接丢弃高位，保留低位
2. 截断有符号数(补码数)：直接丢弃高位，保留低位，但将结果视为补码，再得到相应结果。或者如上图

- 1.3. 整数运算
  - 1.3.1. 无符号加法/乘法：注意如何处理溢出

```
unsigned char     1110 1001      E9      233
                + 1101 0101    + D5    + 213
                1 1011 1110     1BE      446
                  1011 1110      BE      190

                  1110 1001      E9      233
*                 1101 0101    *  D5    *  213
 1100 0001 1101 1101            C1DD     49629
                  1101 1101      DD      221
```

  - 1.3.2. 补码的加法/非

1. 加减法： ，
2. 取反：x的补码取反，得到的就是-x的补码
3. 其它：

- 1.3.3. C语言位移运算：实现乘除2的幂

  `x<<a` 左移就是: 丢弃最高位，0补最低位

  `x>>a` 右移就是：丢弃最低位，对于无符号数左边补0，对于有符号数左边补符号位

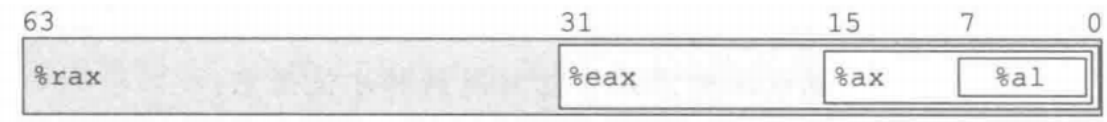
2. X86-64汇编

2.1. 数据格式

1. 字=16位，双字(长字)=32位，四字=64位
2. 汇编代码中的后缀表示数据的大小：movb(传送字节)/movw(传送字)/movl(传送双字)/movq(传送四字)

2.2. 访问信息(寄存器/内存)

- 2.2.0. X86-64 CPU的结构：包含16个64位通用寄存器



返回值	%rax	%eax	%r8	%r8d	第5个参数
被调用者保存	%rbx	%ebx	%r9	%r9d	第6个参数
第4个参数	%rcx	%ecx	%r10	%r10d	调用者保存
第3个参数	%rdx	%edx	%r11	%r11d	调用者保存
第2个参数	%rsi	%esi	%r12	%r12d	被调用者保存
第1个参数	%rdi	%edi	%r13	%r13d	被调用者保存
被调用者保存	%rsp	%esp	%r14	%r14d	被调用者保存
栈指针	%rbp	%ebp	%r15	%r15d	被调用者保存

1. 字节/16位/32位/64位操作，可以访问最低/最低2个/最低4个/整个字节
2. 针名%rsp用来指明运行时栈的结束位置

- 2.2.1. 操作数指示符

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

数据为立即数/来自寄存器/来自内存结果存到寄存器/内存中

1. 立即数：是常数，书写方式Dollar+整数，如$0x1F
2. 寄存器：表示寄存器内容，表示任意寄存器a，**表示其值**
3. 内存引用：根据计算的有效地址访问某块内存
   中是立即数偏移，为基址/变址寄存器(64位)，s为比例因子
   存在

- 2.2.2. 数据移动指令：以MOV为例，MOVZ/MOVS同理

指令		效果	描述
MOV	S, D	D←S	传送
movb			传送字节
movw			传送字
movl			传送双字
movq			传送四字
movabsq	I, R	R←I	传送绝对的四字

```
1   movl $0x4050,%eax       Immediate--Register, 4 bytes
2   movw %bp,%sp            Register--Register,  2 bytes
3   movb (%rdi,%rcx),%al    Memory--Register,    1 byte
4   movb $-17,(%rsp)        Immediate--Memory,   1 byte
5   movq %rax,-12(%rbp)     Register--Memory,    8 bytes
```

1. S：立即数/在寄存器中/内存中；D：寄存器/内存；不允许内存内存(需要寄存器做中介)
2. movl指令D为寄存器时，寄存器高32位置0

- 2.2.3. 数据传送指令示例

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

```
    long exchange(long *xp, long y)
    xp in %rdi, y in %rsi
1   exchange:
2     movq    (%rdi), %rax    Get x at xp. Set as return value.
3     movq    %rsi, (%rdi)    Store y at xp.
4     ret                     Return.
```

xp是第一个参数所以在%rdi，y是第二个参数所以在%rsi，x为返回值所以在%rax中
其中xp实际是内存中的一个地址*xp才是内存中的值，(%rdi)相当于一个指针

- 2.3. 算术和逻辑操作
  - 2.3.1. 加载有效地址：类似于C中取地址&
    - 2.3.1.1. 含义

指令	效果	描述
leaq S, D	D ← &S	加载有效地址

指令不引用内存，不读内存数据，而是将有效地址写入到目的操作数

- 2.3.1.2. 示例：说明 `leaq` 可以描述简单的算术操作

```
long scale(long x, long y, long z) {
    long t = x + 4 * y + 12 * z;
    return t;
}
```

```
    x in %rdi, y in %rsi, z in %rdx
scale:
    leaq    (%rdi,%rsi,4), %rax     x + 4*y
    leaq    (%rdx,%rdx,2), %rdx     z + 2*z = 3*z
    leaq    (%rax,%rdx,4), %rax     (x+4*y) + 4*(3*z) = x + 4*y + 12*z
    ret
```

- 2.3.2. 一元和二元操作：熟悉一下，考试时会给出

INC D	$D \leftarrow D+1$	加1
DEC D	$D \leftarrow D-1$	减1
NEG D	$D \leftarrow -D$	取负
NOT D	$D \leftarrow \sim D$	取补
ADD S, D	$D \leftarrow D+S$	加
SUB S, D	$D \leftarrow D-S$	减
IMUL S, D	$D \leftarrow D*S$	乘
XOR S, D	$D \leftarrow D \hat{} S$	异或
OR S, D	$D \leftarrow D \mid S$	或
AND S, D	$D \leftarrow D \& S$	与
SAL k, D	$D \leftarrow D << k$	左移
SHL k, D	$D \leftarrow D << k$	左移（等同于SAL）
SAR k, D	$D \leftarrow D >>_A k$	算术右移
SHR k, D	$D \leftarrow D >>_L k$	逻辑右移

- 2.3.3. 特殊算术操作

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

64位乘法生成128位，%rdx存高64位，%rdi存低64位；除法则是128→64位

- 2.4. 程序控制
  - 2.4.1. 条件码
    - 2.4.1.1. 条件码寄存器：描述最近的算数/逻辑操作属性

      CF:进位标志———最近的操作溢出
      ZF:零标志————最近的操作得出的结果为0
      SF:符号标志———最近的操作得到负数
      OF:溢出标志———最近的操作导致补码溢出
    - 2.4.1.2. 条件码设置指令(注意标黄的顺粗别反了)

指令		基于	描述
CMP	S_1，S_2	S_2-S_1	比较
cmpb			比较字节
cmpw			比较字
cmpl			比较双字
cmpq			比较四字
TEST	S_1，S_2	$S_1 \& S_2$	测试
testb			测试字节
testw			测试字
testl			测试双字
testq			测试四字

CMP：根据两个操作数之差来设置条件码

TEST：根据相与结果设置条件码

- 2.4.2. 访问条件码(考试会给出，重在理解

指令	同义名	效果	设置条件
sete D	setz	D ← ZF	相等/零
setne D	setnz	D ← ~ZF	不等/非零
sets D		D ← SF	负数
setns D		D ← ~SF	非负数
setg D	setnle	D ← ~(SF ^ OF) & ~ZF	大于（有符号>）
setge D	setnl	D ← ~(SF ^ OF)	大于等于（有符号>=）
setl D	setnge	D ← SF ^ OF	小于（有符号<）
setle D	setng	D ← (SF ^ OF) | ZF	小于等于（有符号<=）
seta D	setnbe	D ← ~CF & ~ZF	超过（无符号>）
setae D	setnb	D ← ~CF	超过或相等（无符号>=）
setb D	setnae	D ← CF	低于（无符号<）
setbe D	setna	D ← CF | ZF	低于或相等（无符号<=）

- 2.4.3. 跳转指令：跳转目的地有一个标号label

指令	同义名	跳转条件	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	不相等/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 *Label*	jng	(SF ^ OF) | ZF	小于或等于（有符号<=）
ja *Label*	jnbe	~CF & ~ZF	超过（无符号>）
jae *Label*	jnb	~CF	超过或相等（无符号>=）
jb *Label*	jnae	CF	低于（无符号<）
jbe *Label*	jna	CF | ZF	低于或相等（无符号<=）

- 2.4.4. 跳转指令的编码方式

- 2.4.4.1. PC-相对：目标编码=目标指令地址—跳转指令后条指令地址

```
1    movq    %rdi, %rax
2    jmp     .L2
3  .L3:
4    sarq    %rax
5  .L2:
6    testq   %rax, %rax
7    jg      .L3
8    rep; ret
```

```
1    0:   48 89 f8
2    3:   eb 03
3    5:   48 d1 f8
4    8:   48 85 c0
5    b:   7f f8
6    d:   f3 c3
```

```
mov    %rdi,%rax
jmp    8 <loop+0x8>
sar    %rax
test   %rax,%rax
jg     5 <loop+0x5>
repz retq
```

第一条目标编码=0x03(第二字节)，其下一条指令的地址0x5，相加得0x8，即第4行跳转目标地址
第二条目标编码=0xf8(第二字节)，其下一条指令的地址0xd，相加得0x5，即第3行跳转目标地址
注意正常情况下每行编码相差2

- 2.4.4.2. 绝对地址：用4字节直接指定目标

- 2.4.5. 循环

- 2.4.5.1. `do-while`

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

```
    long fact_do(long n)
    n in %rdi
1   fact_do:
2     movl    $1, %eax        Set result = 1
3   .L2:                    loop:
4     imulq   %rdi, %rax      Compute result *= n
5     subq    $1, %rdi        Decrement n
6     cmpq    $1, %rdi        Compare n:1
7     jg      .L2             If >, goto loop
8     rep; ret                Return
```

- 2.4.5.2. `while`

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

```
  long fact_while(long n)
  n in %rdi
fact_while:
  movl    $1, %eax        Set result = 1
  jmp     .L5             Goto test
.L6:                    loop:
  imulq   %rdi, %rax      Compute result *= n
  subq    $1, %rdi        Decrement n
.L5:                    test:
  cmpq    $1, %rdi        Compare n:1
  jg      .L6             If >, goto loop
  rep; ret                Return
```

- 2.4.5.3. `for`：处理思路，转化成while处理

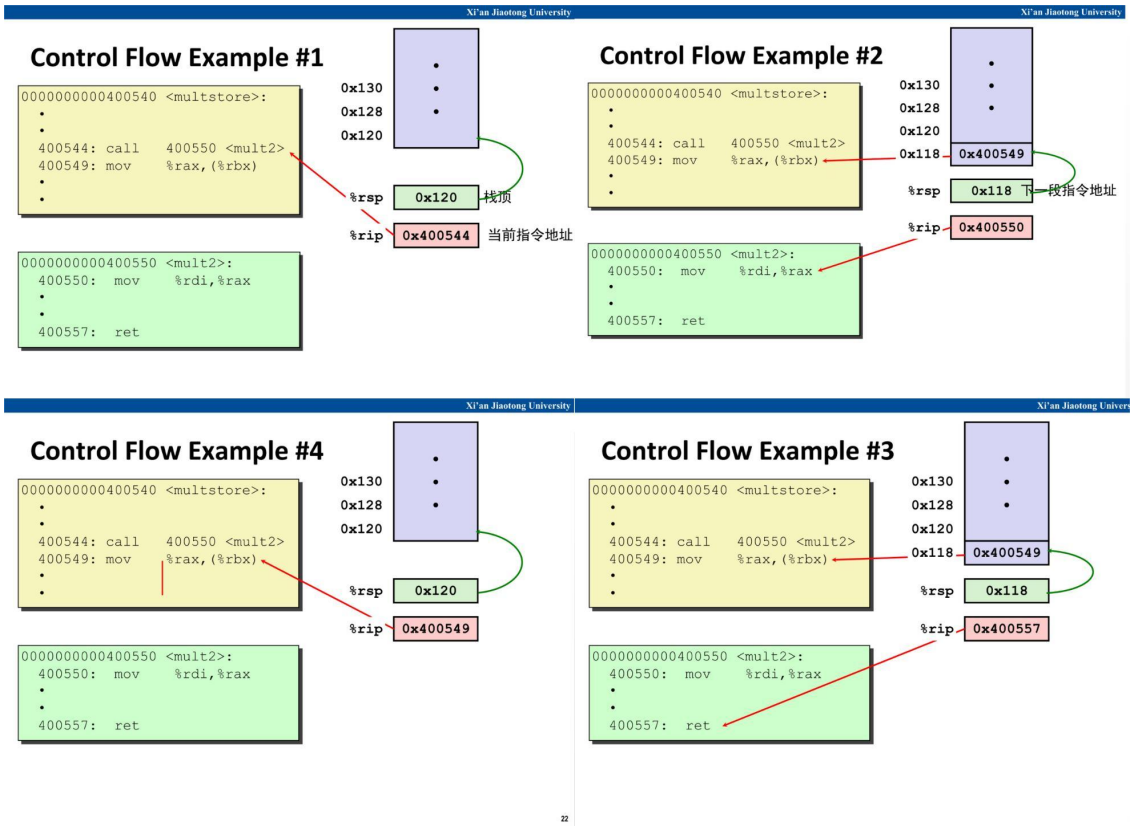
```
long fact_for(long n)
{
    long i;
    long result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

```
long fact_for_while(long n)
{
    long i = 2;
    long result = 1;
    while (i <= n) {
        result *= i;
        i++;
    }
    return result;
}
```

```
long fact_for_jm_goto(long n)
{
    long i = 2;
    long result = 1;
    goto test;
loop:
    result *= i;
    i++;
test:
    if (i <= n)
        goto loop;
    return result;
}
```

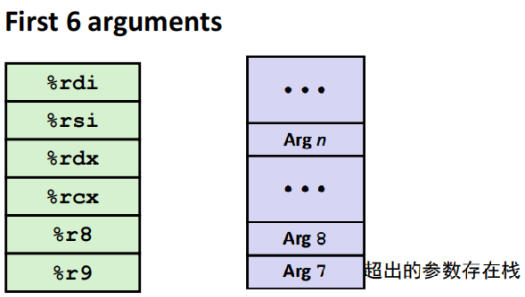
```
long fact_for(long n)
n in %rdi
fact_for:
  movl    $1, %eax      Set result = 1
  movl    $2, %edx      Set i = 2
  jmp     .L8           Goto test
.L9:                    loop:
  imulq   %rdx, %rax    Compute result *= i
  addq    $1, %rdx      Increment i
.L8:                    test:
  cmpq    %rdi, %rdx    Compare i:n
  jle     .L9           If <=, goto loop
  rep; ret              Return
```

- 2.5. 过程控制：调用原理
  - 2.5.1. X86-64栈结构：记住栈的top在下面地址小，%rsp指向top
  - 2.5.2. 调用：控制权的转移



1. 过程调用：`call label  #跳转到label`，将当前的下一位地址当作**返回地址**压入栈
2. 过程返回：`ret`，让返回地址出栈，跳转到返回地址

  - 2.5.3. 调用：数据的传递



函数参数在内存中的位置：前六个在寄存器中，多出的存在栈中
例如`void multstore(long x, long y, long *dest)`中`x in %rdi, y in %rsi, dest in %rdx`

- 2.5.4. 内存管理：只有调用的时候才为被调用函数分配空间，调用完就销毁

- 2.6. 数组分配与访问

  PS：关于指针：注意 `a=*&a` ， `A[i]=*(A+i)`

  - 2.6.1. 指针-数组-汇编

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

  - 2.6.2. 嵌套数组：$D[i][j] = s_D + Size(Colum * i + j)$

行	元素	地址
A[0]	A[0][0]	x_A
	A[0][1]	x_A+4
	A[0][2]	x_A+8
A[1]	A[1][0]	x_A+12
	A[1][1]	x_A+16
	A[1][2]	x_A+20
A[2]	A[2][0]	x_A+24
	A[2][1]	x_A+28
	A[2][2]	x_A+32

  - 2.6.3. 数组与指针的示例：以下Comp表示编译通过，Bad表示可能的错误指针引用

    - 2.6.3.1. 示例1

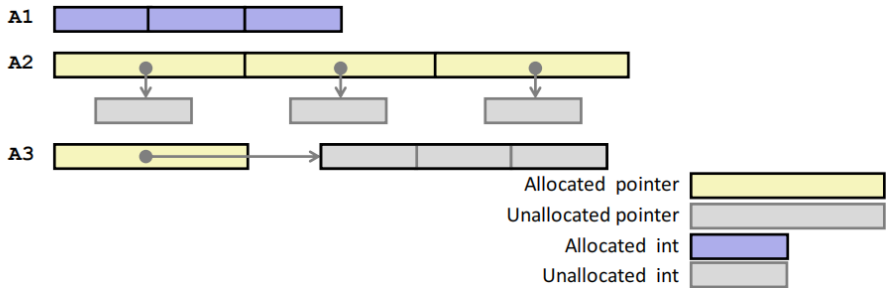
Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
int A1[3]	Y	N	12	Y	N	4
int *A2	Y	N	8	Y	Y	4



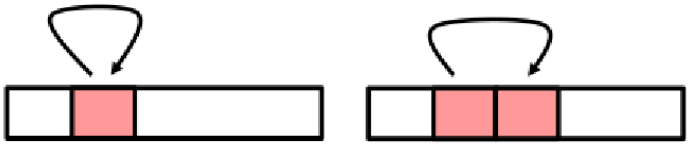
*A1没事是因为一开始就给数组分配了空间
*A2会出问题在于没有让数组指向某个值，是个野指针

    - 2.6.3.2. 示例2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4	N	-	-
int *A2[3]	Y	N	24	Y	N	8	Y	Y	4
int (*A3)[3]	Y	N	8	Y	Y	12	Y	Y	4

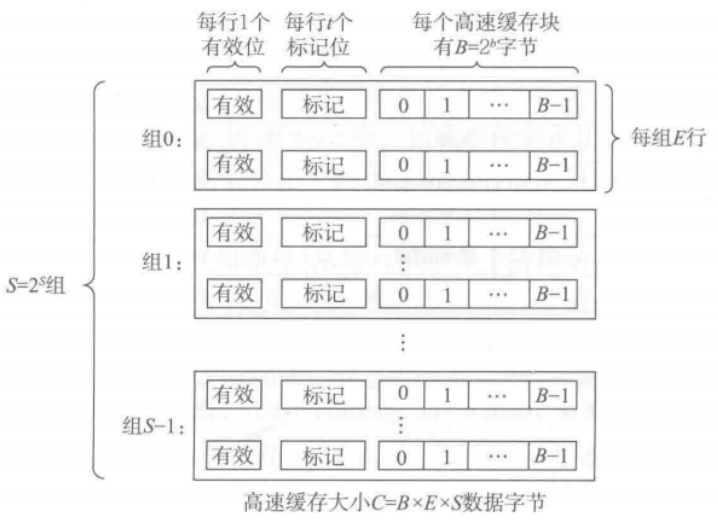


- 3. 高速缓存
  - 3.1. 局部性



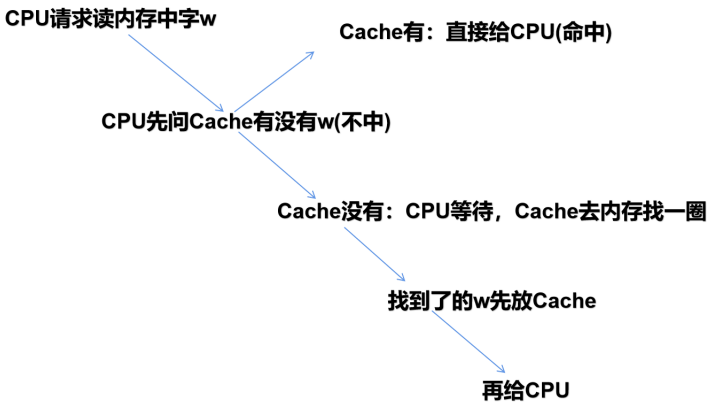
1. 概念：程序倾向于引用附近刚刚被引用的数据
2. 分类：时间局部性(一个地方引用后一段时间再引用)，空间局部性(被应用位置的附件马上会被引用)

  - 3.2. 高速缓存结构：元组$(S_{组}, E_{行}, B_{块}, m_{位})$

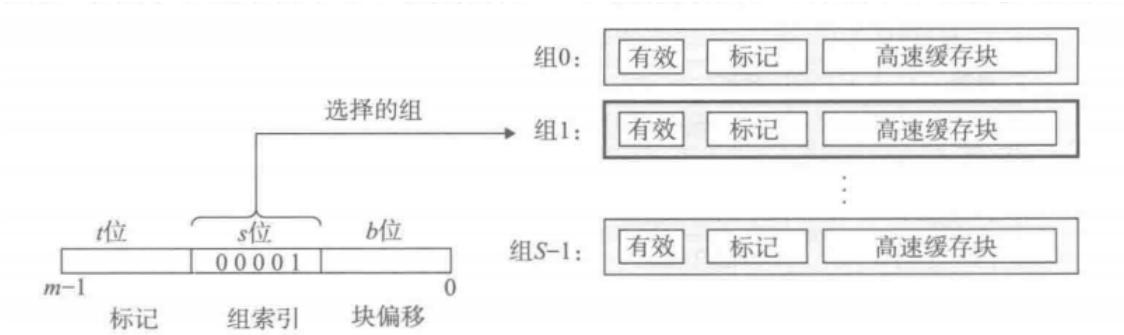


1. 高速缓存有个组，每组行，每一行的成分是：有效位+标记+快
2. 其中块大小为；有效位表示这一行是否有意义； 标记位大小为地址位数-(s+b)，是每一块cache的ID
3. 高速缓存大小：，

  - 3.3. 直接映射高速缓存(E==1,S>1的特殊情形)

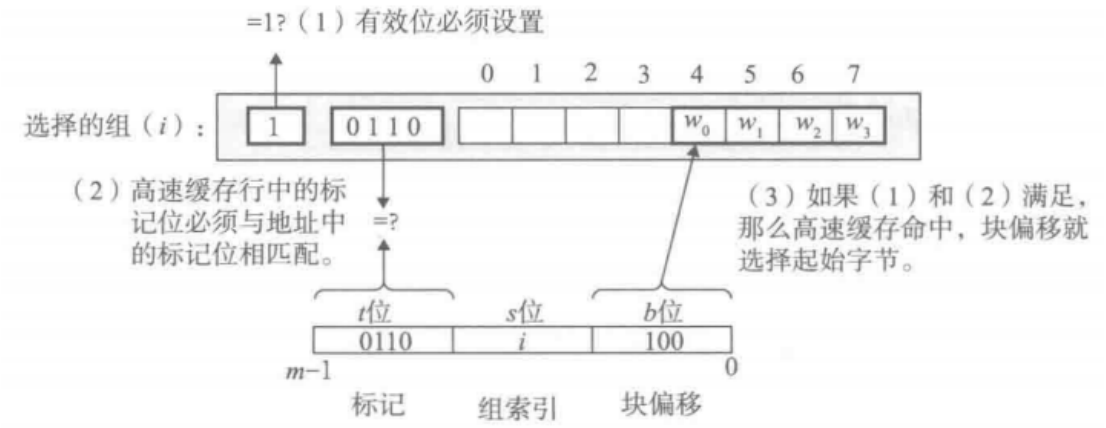


- 3.3.1. 组选择



高速缓存从w的地址中间抽取出s个组索引位

- 3.3.2. 行匹配



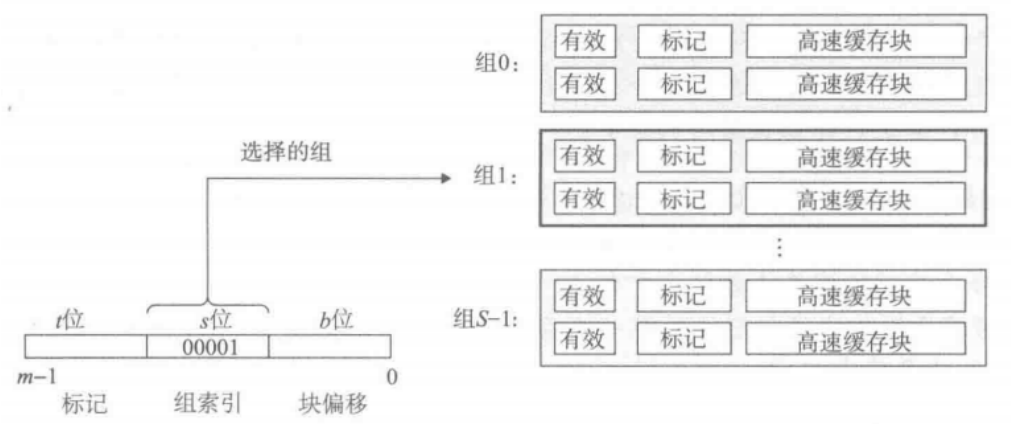
- 3.3.3. 字选择

1. 块偏移位提供了w的第一个字节的偏移
2. 在上图的示例中，块偏移位是100B(二进制16)，它表明w的副本是从块中的字节4开始的(我们假设字长为4字节)。

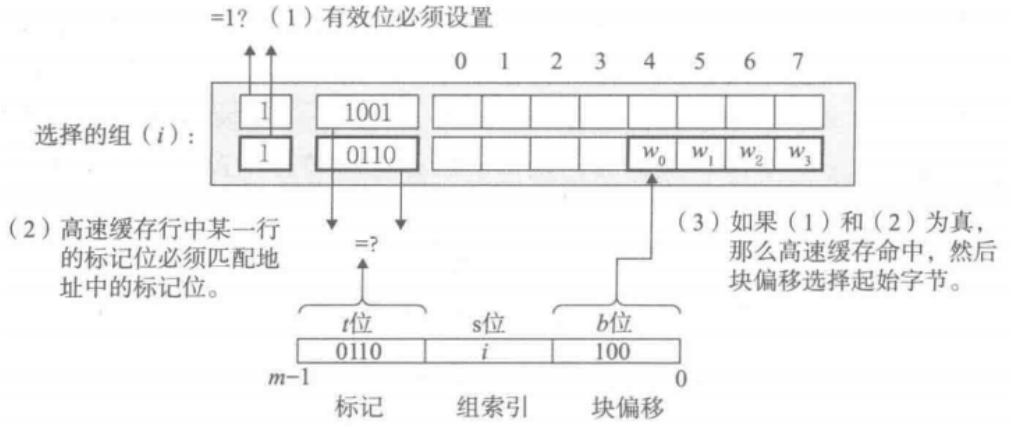
- 3.3.4. 不命中时的行替换

1. 不命中时，请求下一层缓存取出被请求的块
2. **来到组选择所指向的组，由于只有一行，直接用新取出的行替换当前的行**

- 3.4. 组相联高速缓存(1<E<C/B，S>1的特殊情形)
- 3.4.1. 组选择



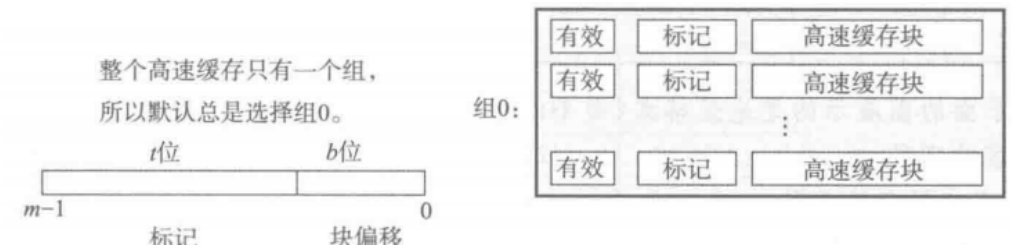
- 3.4.2. 行匹配和字选择



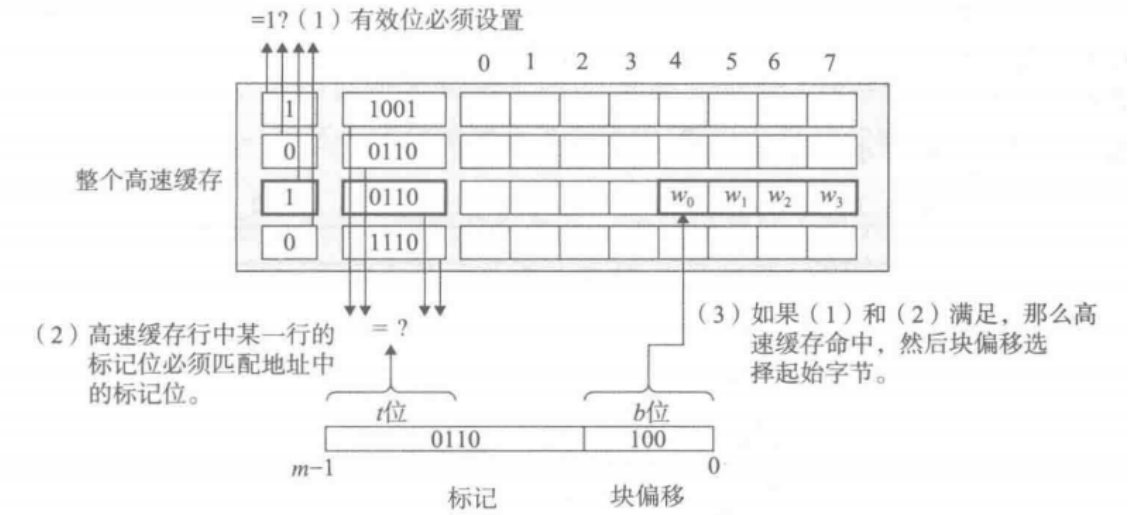
- 3.4.3. 不命中时的行替换

1. 优先替换空行
2. 没有空行的时候替换**最后一次访问时间最久远的那一行**(最近最少使用策略)
3. 或者是替换在过去某个时间窗口内引用次数最少的那一行(最不常使用策略

- 3.5. 全相联高速缓存：只需要行匹配与字选择，替换策略不变
- 3.5.2. 组选择(只有一个组选择个Der)



- 3.5.3. 全相联高速缓存中的行匹配和字选择



以上内容整理于 幕布文档