

计算机系统基础考点总结

徐逸辰

2019 年 12 月 24 日

目录

1 整数的表示与运算	1	3.3 数据大小格式	3
1.1 大纲	1	3.4 寄存器	3
1.2 无符号数的表示 (Unsigned)	1	3.5 地址格式	3
1.3 有符号数的表示 (Two-complement)	1	3.6 数据传送与算数指令	3
1.4 无符号数的扩展	1	3.7 算数指令	4
1.5 有符号数的扩展	1	3.8 128 位数据算数指令	4
1.6 截断整数	1	4 机器指令：控制	4
1.7 有符号数与无符号数之间的转换	1	4.1 大纲	4
1.8 整数的加法	1	4.2 条件码	5
1.9 有符号数的加法逆元	1	4.3 条件码的访问	5
1.10 整数乘法	2	4.4 条件跳转指令	5
1.11 整数乘以常数的优化方式	2	4.5 用条件跳转实现条件分支	5
1.12 整数除以 2 的幂次	2	4.6 用条件跳转实现循环	6
2 浮点数的表示	2	4.6.1 do while	6
2.1 大纲	2	4.6.2 while	6
2.2 符号、尾数、阶码	2	4.6.3 for	6
2.3 浮点数表示的三种情况	2	5 IO	6
2.3.1 规范化的值	2	5.1 大纲	6
2.3.2 非规范化的值	2	5.2 文件描述符 (File Descriptor)	7
2.3.3 特殊情况	3	5.3 打开的文件在内核层面的表示	7
3 机器指令：数据传送与运算	3	5.4 fork 与文件	7
3.1 大纲	3	5.5 IO 重定向	7
3.2 大端法与小端法	3	5.6 Unix IO 与标准 IO	7

1 整数的表示与运算

1.1 大纲

1. 整数的表示（无符号，有符号）
2. 整数的扩展与截断
3. 有符号数与无符号数之间的转换
4. 整数运算

1.2 无符号数的表示 (Unsigned)

位向量 $\vec{x} = [x_{w-1}, \dots, x_0]$ 所表示的无符号整数

$$B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i 2^i.$$

w 位无符号数的表示范围为 $[0, 2^w - 1]$ 。

1.3 有符号数的表示 (Two-complement)

位向量 $\vec{x} = [x_{w-1}, \dots, x_0]$ 所表示的无符号整数

$$B2U_w(\vec{x}) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i.$$

w 位有符号数的表示范围为 $[-2^{w-1}, 2^{w-1} - 1]$ 。

与无符号数表示的唯一区别是最高位的权重为负。

1.4 无符号数的扩展

对位向量 $\vec{x} = [x_{w-1}, \dots, x_0]$ 表示的无符号数进行拓展到 w' 位得到的结果为

$$[0, \dots, 0, x_{w-1}, \dots, x_0].$$

也即高位补 0。

1.5 有符号数的扩展

对位向量 $\vec{x} = [x_{w-1}, \dots, x_0]$ 表示的有符号数进行拓展到 w' 位得到的结果为

$$[x_{w-1}, \dots, x_{w-1}, x_{w-1}, \dots, x_0].$$

也即高位补符号位。

1.6 截断整数

对位向量 $\vec{x} = [x_{w-1}, \dots, x_0]$ 表示的有符号数或无符号数截断到 w' 位的结果为

$$[x_{w'-1}, \dots, x_0].$$

也即无论是有符号数还是无符号数，截断操作直接丢弃高位。

1.7 有符号数与无符号数之间的转换

在 C 语言中，当一个运算符两端同时涉及有符号与无符号数时，默认将有符号数转化为无符号数。这会导致一些不容易发现的错误。

此外，在同时涉及符号与长度的转换时，总是先转化长度，再转化符号。

```
char a = -1;
unsigned short b = a; // b = 65535 而不是 255
```

1.8 整数的加法

无论是有符号还是无符号整数，在机器层面的加法都是等同的。唯一的区别是溢出问题。（或者说对运算结果的解读方式）

无符号溢出

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

第二种情况即为溢出的情况。

有符号溢出

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

第一种情况为正溢出，第二种情况为负溢出。

1.9 有符号数的加法逆元

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

1.10 整数乘法

无论是无符号数还是有符号数，乘法在机器层面都是等同的。唯一不同的仍然是解读结果的方式。

对于无符号数：

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

对于有符号数：

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

1.11 整数乘以常数的优化方式

假设要计算 $x * K$ ，其中 K 可以表示为如下形式：

$$(0 \cdots 0)(1 \cdots 1)(0 \cdots 0),$$

也即从 n 位置到 m 位置有连续的 1，且其余位置都为 0。可以利用位运算进行如下优化：

$$x * K = (x \ll n) + (x \ll (n-1)) + \cdots + (x \ll m),$$

或

$$x * K = (x \ll (n+1)) - (x \ll m).$$

1.12 整数除以 2 的幂次

无符号数的情况

$$\lfloor \frac{x}{2^k} \rfloor = x \gg k.$$

$$\lceil \frac{x}{2^k} \rceil = \lfloor \frac{x + 2^k - 1}{2^k} \rfloor = (x + (1 \ll k) - 1) \gg k.$$

有符号数的情况 有符号数与无符号数基本等同，唯一不同的是右移操作 \gg 应当使用算数右移，也即高位补符号位。

2 浮点数的表示

2.1 大纲

1. 符号、尾数、阶码
2. 三种情况
3. C 语言中的单精度与双精度浮点

2.2 符号、尾数、阶码

符号 s 决定表示的浮点数是正数还是负数。

尾数 M 一个二进制小数。

阶码 E 对尾数表示的二进制小数进行加权。
于是浮点数可以被表示为

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

相应地，浮点数的表示也被分为三个部分：

1. 1 位符号位，表示 s ；
2. k 位阶码字段 $\text{exp} = e_{k-1} \cdots e_0$ 编码阶码 E ；
3. n 位小数字段 $\text{frac} = f_{n-1} \cdots f_0$ 编码尾数 M 。

2.3 浮点数表示的三种情况

2.3.1 规范化的值

当 exp 既不全为 0，也不全为 1 时，即为该种模式。此时

阶码 $E = e - \text{Bias}$ ，其中 $\text{Bias} = 2^{k-1} - 1$ 。

尾数 $M = 1.f_{n-1} \cdots f_0$ 。

2.3.2 非规范化的值

当 exp 全为 0 时，即为此种情况，此时

阶码 $E = 1 - \text{Bias}$ ，其中 $\text{Bias} = 2^{k-1} - 1$ 。

阶码没有设置为 $-\text{Bias}$ ，是为了与规范化的值平滑过渡。考虑某种长度的浮点数表示中，规范化的正值的最小值：

$$0.0001000 \cdots$$

假设仍然设置 $E = e - \text{Bias} = -\text{Bias}$ ，则非规范化值的最大值：

$$0.0000011 \cdots$$

而调整 $E = 1 - \text{Bias}$ 之后，非规范化值的最大值：

$$0.0000111 \cdots$$

尾数 $M = 0.f_{n-1} \cdots f_0$ 。

注：
非规范化的值让我们能够表示 0，事实上是两种不同的 0：
+0.0 和 -0.0。

2.3.3 特殊情况

当阶码 exp 全为 1 时，即为此种情况，此时

- 1. frac全为 0，则表示 $+\infty$ 与 $-\infty$ ；
- 2. frac不全为 0，则表示 NaN。

3 机器指令：数据传送与运算

3.1 大纲

- 1. 大端法与小端法
- 2. 数据大小格式
- 3. 寄存器
- 4. 地址格式
- 5. 数据传送与算数指令
- 6. 128 位数的算数指令

3.2 大端法与小端法

大端法：高位字节在前，低位字节在后。
小端法：与大端法相反。

3.3 数据大小格式

C 类型	数据类型	汇编代码后缀	大小 (Bytes)
char	字节	b	1
short	字 (word)	w	2
int	双字	l	4
long	四字	q	8
char*	四字	q	8

注：
在 64 位系统中，所有指针都是四字的，也即有 8 个字节。

3.4 寄存器

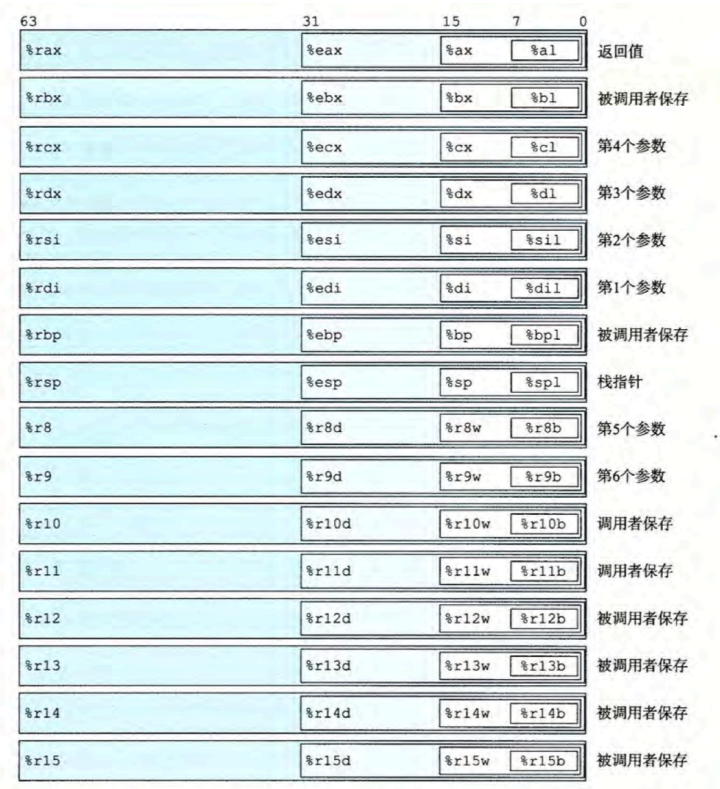


Figure 1: 寄存器信息

3.5 地址格式

格式	数值
$\$Imm$	Imm
r_a	$M[r_a]$
Imm	$M[Imm]$
(r_a)	$M[R[r_a]]$
$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$

注：
1. 最后一种情形不一定每一个参数都要给出，缺少参数可以构成许多种形式，不再全部列出。
2. $s \in \{1, 2, 4, 8\}$ 。

3.6 数据传送与算数指令

MOV 指令 MOV S, D 的作用是将操作数 S 的值传送到 D。

movb, movw, movl, movq。

- S 可以为立即数，寄存器与内存地址。D 只能为寄存器或内存地址。
- S 与 D 不能同时为内存地址。
- S 与 D 数据大小相同。

注：

movabsq 能够将 64 位立即数传送到指定目标，这意味着：普通的 MOV 指令只能传送 32 位立即数。

3.7 算数指令

指令	效果	描述
leaq S, D	$D \leftarrow \&S$	加载有效地址
INC D	$D \leftarrow D + 1$	加1
DEC D	$D \leftarrow D - 1$	减1
NEG D	$D \leftarrow -D$	取负
NOT D	$D \leftarrow \sim D$	取补
ADD S, D	$D \leftarrow D + S$	加
SUB S, D	$D \leftarrow D - S$	减
IMUL S, D	$D \leftarrow D * S$	乘
XOR S, D	$D \leftarrow D \wedge S$	异或
OR S, D	$D \leftarrow D S$	或
AND S, D	$D \leftarrow D \& S$	与
SAL k, D	$D \leftarrow D \ll k$	左移
SHL k, D	$D \leftarrow D \ll k$	左移（等同于SAL）
SAR k, D	$D \leftarrow D \gg_A k$	算术右移
SHR k, D	$D \leftarrow D \gg_L k$	逻辑右移

Figure 2: 算数指令

MOVZ 指令 MOVZ S, D 的作用是将操作数 S 进行零扩展（高位补 0）后传送到 D。

movzbw, movzbl, movzbq, movzwl, movzwq。

- 指令名称的最后两个字符分别指定了 S 与 D 的类型。
- 不存在 movzql，因为 movl 会自动把寄存器高位置 0。

MOVS 指令 MOVS S, D 的作用是将操作数 S 进行符号扩展（高位补符号位）后传送到 D。

movsbw, movsbl, movsbq, movswl, movswq, movslq。

- 与 MOVZ 基本一致，做符号扩展，且存在 movslq。

注：

altq 将 %eax 上的数据符号扩展到整个 %rax。

注：

- leaq 事实上不进行内存访问，只是计算地址。也可用于算数运算。
- 注意操作数顺序。
- 注意算数右移（高位补符号位）与逻辑右移（高位补 0）的区别。

3.8 128 位数据算数指令

指令	效果	描述
imulq S	$R[\%rdx] \leftarrow R[\%rax] \times S \times R[\%rax]$	有符号全乘法
mulq S	$R[\%rdx] \leftarrow R[\%rax] \times S \times R[\%rax]$	无符号全乘法
cltq	$R[\%rdx] \leftarrow \text{符号扩展}(R[\%rax])$	转换为八字
idivq S	$R[\%rdx] \leftarrow R[\%rdx] \div R[\%rax] \bmod S$ $R[\%rdx] \leftarrow R[\%rdx] \div R[\%rax]$	有符号除法
divq S	$R[\%rdx] \leftarrow R[\%rdx] \div R[\%rax] \bmod S$ $R[\%rdx] \leftarrow R[\%rdx] \div R[\%rax]$	无符号除法

Figure 3: 128 位算数运算

4 机器指令：控制

4.1 大纲

1. 条件码的含义与访问

2. 条件跳转与条件传送

3. 实现 C 语言中的条件分支

4.2 条件码

- CF: 进位标志。与无符号操作溢出有关。
- ZF: 零标志。最近一次算数操作得到的结果是否为 0。
- SF: 符号标志。最近一次操作的符号位。
- OF: 溢出标志。最近一次操作是否导致补码意义下的溢出。正溢出或负溢出。

图 2 中列出的算数指令会设置条件码，除此以外，下图的 CMP 指令与 TEST 指令也会在不改变任何寄存器的条件下设置条件码：

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

Figure 4: CMP 和 TEST 指令

4.3 条件码的访问

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

Figure 5: SET 指令

注:

基于的含义是执行运算，丢弃结果，只保留标志位。

4.4 条件跳转指令

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
j _e $Label$	jz	ZF	相等/零
j _{ne} $Label$	jnz	$\sim ZF$	不相等/非零
j _s $Label$		SF	负数
j _{ns} $Label$		$\sim SF$	非负数
j _g $Label$	jnle	$\sim(SF \wedge OF) \& \sim ZF$	大于（有符号>）
j _{ge} $Label$	jnl	$\sim(SF \wedge OF)$	大于或等于（有符号>=）
j _l $Label$	jnge	$SF \wedge OF$	小于（有符号<）
j _{le} $Label$	jng	$(SF \wedge OF) \mid ZF$	小于或等于（有符号<=）
j _a $Label$	jnbe	$\sim CF \& \sim ZF$	超过（无符号>）
j _{ae} $Label$	jnb	$\sim CF$	超过或相等（无符号>=）
j _b $Label$	jnae	CF	低于（无符号<）
j _{be} $Label$	jna	$CF \mid ZF$	低于或相等（无符号<=）

Figure 6: JUMP 指令

4.5 用条件跳转实现条件分支

原始代码：

```
if (a > b)
    return a;
else
    return b;
```

翻译后的版本:

```
if (a ≤ b)
    goto false;
return a;
goto done;
false:
    return b;
done:
```

等价机器码:

```
; a in %rdi, b in %rsi
cmpq %rsi, %rdi ; Compare a:b
jle .L2
movq %rdi, %rax
jmp .L3
.L2:
    movq %rsi, %rax
.L3:
    ret
```

4.6 用条件跳转实现循环

4.6.1 do while

原始代码:

```
do {
    do_something();
} while (condition);
```

翻译后的版本:

```
loop:
    do_something()
    if (condition)
        goto loop;
```

4.6.2 while

对于一段 while 循环

```
while (condition) {
    do_something();
}
```

共有两种常用的翻译(实现)方法:

Jump to middle

```
goto test;
loop:
    do_something();
test:
    if (condition)
        goto loop;
```

Guarded do

```
if (!condition)
    goto done;
loop:
    do_something()
    if (condition)
        goto loop;
done:
```

这一种方法的全称应当为 guarded do-while, 因为它事实上是一个带初始判断条件的 do-while 循环。

4.6.3 for

对于 for 循环:

```
for (init_expr; cond_expr; update_expr) {
    body_expr;
}
```

直接翻译成对应的 while 循环:

```
init_expr;
while (cond_expr) {
    body_expr;
    update_expr;
}
```

再进行进一步翻译即可。

5 IO

5.1 大纲

1. 文件描述符
2. 内核中文件的表示方式

3. fork与文件

4. IO 重定向

5. Unix IO 与标准库 IO

5.2 文件描述符 (File Descriptor)

- C 语言程序中用来标示打开的文件的一种方式
- open 这样的 Unix IO 函数会返回文件描述符。(若打开失败则会返回 -1)
- 每个由 Linux 终端启动的进程都有三个已有的文件描述符: 0, 1, 2, 分别代表 stdin, stdout, stderr。

5.3 打开的文件在内核层面的表示

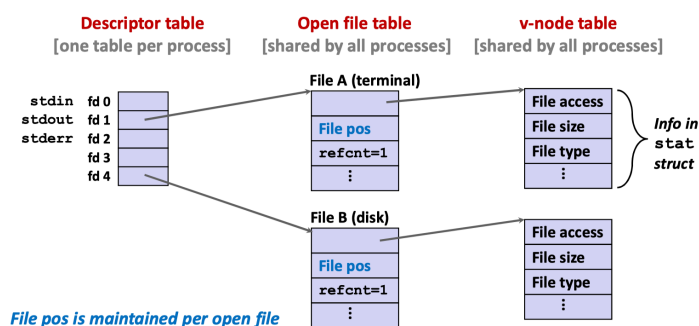


Figure 7: 描述符、打开文件表与 v-node 表

特别的，如果打开同一个文件两次：

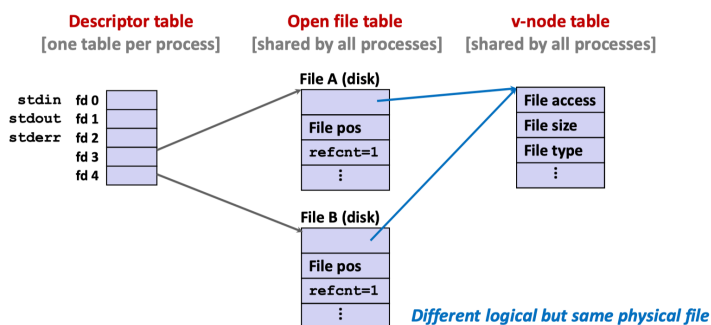


Figure 8: 描述符、打开文件表与 v-node 表

5.4 fork与文件

fork 出来的进程事实上会共享父进程所有的信息：运行时栈和所有的寄存器，当然也包括描述符表。

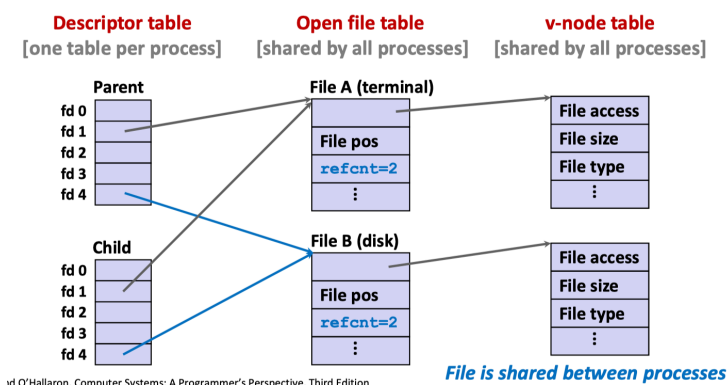


Figure 9: 描述符、打开文件表与 v-node 表

5.5 IO 重定向

IO 重定向所做的事情事实上是让多个描述符指向同一个打开的文件。dup2(oldfd, newfd) 事实上会让 newfd 改为指向 oldfd 所指向的打开的文件。

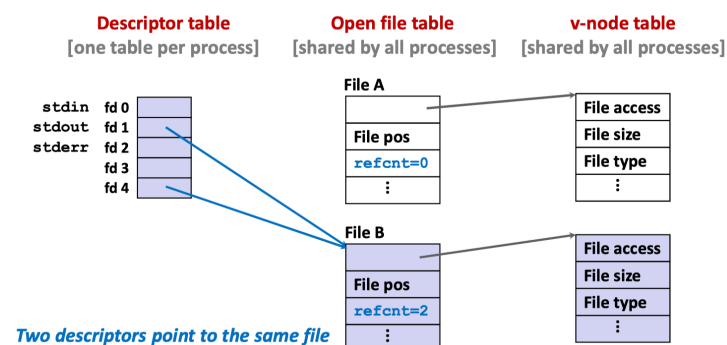


Figure 10: 调用 dup2(4, 1) 之后的描述符、打开文件表与 v-node 表

5.6 Unix IO 与标准 IO

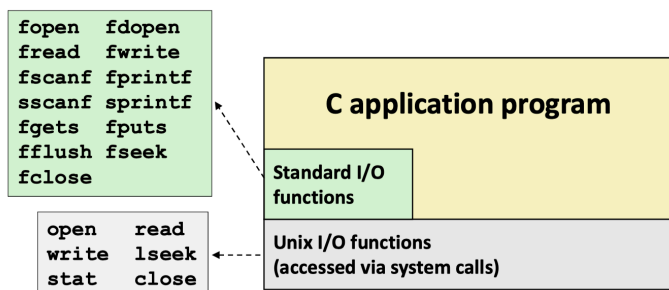


Figure 11: 标准 IO 与 Unix IO 的关系