

ICS 期中考试复习细节

有用的链接

- [第三程序的机器级表示笔记](#)
- [第四章全考点，CMU 完整 HCL 手册](#)
- [第五章优化程序性能详细笔记](#)
- [第六章存储器层次结构详细笔记](#)
- [条件码、条件控制和条件传送讲解](#)
- [在线汇编器](#)
- [条件跳转中， \$l = SF \wedge OF\$ 的证明](#)
- [C 语言声明解析器](#)

第二章 信息的表示和处理

$T_{min} = -2^{31} = -2147483648$ ，位级表示为 1000 0000 0000 0000 0000 0000 0000 0000，十六进制表示为 0x80000000

对于 T_{min} 没有 $-x = \sim x + 1$ ，因为 T_{min} 的 $-x$ 会溢出，即 $-T_{min} = T_{min}$ 。其他补码整数都有 $-x = \sim x + 1$

$T_{max} = 2^{31} - 1 = 2147483647$ ，位级表示为 0111 1111 1111 1111 1111 1111 1111 1111，十六进制表示为 0x7fffffff

反码表示：正数的反码是其本身，负数的反码是其除符号位外的其他位取反。0 有两个表示：0000 ... 0000 和 1111 ... 1111，即 +0 和 -0

补码整数移位向下舍入，除法向零舍入

第三章 程序的机器级表示

寄存器后缀

$r[a-d]x - e[a-d]x - [a-d]x - [a-d]l$ ，记忆：只有 $abcd-x$ 是这样的

$r[si/di/bp/sp] - e[si/di/bp/sp] - [si/di/bp/sp] - [si/di/bp/sp]l$ ，记忆：

- $l = \text{low}$ 低位（1 个字，2 字节的低位，即 1 个字节）

- e = extended 扩展（扩展一个字，即 4 个字节）
- r = register 寄存器（寄存器的值，即 8 个字节）

r[8-15] - r[8-15]d - r[8-15]w - r[8-15]b，记忆：dwb 是个颜文字

被调用者保存寄存器：rbx、rbp、r12-r15

强制转换小到大为先改大小再改类型（位级表示不变，是原格式的扩展，即根据原格式是否为有符号数来决定高位补 0 还是 1）

强制转换大到小为直接截断

SAR：算数（Arithmetic）右移，高位补符号位

SHR：逻辑（Logical，或者按照 Hex 记忆，直接移动十六进制）右移，高位补 0

变址寻址表达式的立即数前不加 \$ 符号，比例因子为 1, 2, 4, 8，其中 1 可以省略

mov 要注意寄存器部分（即所有的直接的 %rXX），都需要和指令大小对齐

movX 的大小：b = byte 一个字节，w = word 一个字（两个字节），l = longword 两个字（语言学家-双语-两个字，记住就行），q 四个字就不说了（quad）

movX D, S 要求 D、S 不全为内存地址

movl 设置高 4 字节（32 位）为 0

movq Imm, %rax 会将 Imm 符号扩展为 64 位，然后存入 %rax

movabsq Imm, %rax 会将 Imm（64 位补码）直接存入 %rax，不用扩展，该指令目标地址必须是寄存器

movz 零 Zero 扩展（没有 movzbl）、movs 符号 Symbol 扩展

PUSH 的运算过程：

- $RRSP \leftarrow RRSP - 8$
- $M[RRSP] \leftarrow S$

POP 的运算过程：

- $D \leftarrow M[RRSP]$
- $RRSP \leftarrow RRSP + 8$

imulq S : $rdx:rax \leftarrow S * rax$

idivq S : $rdx \leftarrow rdx:rax \bmod S$, $rax \leftarrow rdx:rax/S$

imulq/idivq 有符号，mulq/divq 没有符号

cltq : eax 符号扩展到 rax

INC/DEC 指令不改变 CF（可以通过这两个指令的结尾都是 C 来记忆），但是可以设置 ZF、OF

逻辑运算指令如 andq、orq、xorq 设置 CF、OF 为 0，但可以设置 ZF 为 1

指针运算，加减常数，要倍乘指针指向的类型的大小；两个相同类型的指针相减，得到的是两个指针之间的元素个数，即要除以指针指向的类型的大小

```
int* p, * p2 = p + 2; // p = 0x100, p2 = 0x108
// 加减常数，要倍乘指针指向的类型的大小，对于 int 就是 * 4
printf("%p\n", p ); // 0x100
printf("%p\n", p+1 ); // 0x104
// 减法，两个相同类型的指针相减，得到的是两个指针之间的元素个数，即要除以指针指向的类型的大小
printf("%ld\n", p2 - p ); // 2
```

特别地，对于数组下标 $A[i]$ 的计算，实际上是 $*(A+i)$ ，即 $A+i$ 是一个指针，指向 A 的第 i 个元素，所以 $A[i]$ 实际上是 $*(A+i)$ ，即 A 的第 i 个元素的值。

解码一个复杂的表达式，要从外向内解码，如

```
int *(*p[2])[3];
// 1
*(*p[2])[3] = int
// 2
*(*p[2]) = int[3] // 即左式整体是一个指针，指向包括 3 个 int 的数组
// 3
*p[2] = * int[3] // 即左式整体是一个指针，指向上一步所述的左式中的整体指针
// 4
p[2] = **int[3] // 由于运算时后缀运算符[]优先，所以反向解的时候，前缀运算符*优先，即左式整体是一个指针
// 5
p = (**int[3])[2] // 即 p 是一个数组名（指针），其指向的数组中的每个元素都是上一步所述的左式中的整体
// summary
// 所以，p 是一个数组名（指针），其指向的数组中的每个元素都是一个指针(i)，这个指针(i)指向一个指针(ii)，
// 严谨版：声明p为指向3个整型指针的2维数组
// 你可以在 https://cdecl.org/ 尝试。
```

同时，注意当 A 是数组名时，调用 $\text{sizeof}(A)$ 时，实际返回的是指针指向的内容的大小：

```
int main() {
    int A[5][3];
    cout << sizeof(&A) << endl; // 8
    cout << sizeof(A) << endl; // 60
    cout << sizeof(*A) << endl; // 12
    cout << sizeof(A[0]) << endl; // 12
    cout << sizeof(**A) << endl; // 4
    cout << sizeof(A[0][0]) << endl; // 4
    cout << &A << " " << (&A + 1) << endl; // 0x8c 0xc8
}
```

又如对于一个 `int* p`，`sizeof(p)` 返回的不是指针的大小（8），而是指针指向的類型的大小（4）。

`jXX` 指令的跳转范围，在相对跳转时，是相对于下一条指令的起始地址，且可以向上跳转（即偏移量为负数，对应的字节码表示为一个负数的补码），注意大小端。

Condition Code 在 x86-64 中可以直接使用指令修改。

符号运算优先级：请参考 [C-CPP](#)。

重点关注：后缀运算符是最高优先级、加减法优先于移位运算符。

`setl = SF^OF`，证明如下：

1. 考虑 `CMP b, a`。假设有 `a < b` 即 `a - b < 0`，那么：
2. 如果有 `a - b < 0`，那么 `a - b` 的符号位为 1，即 `SF=1`。同时也没有发生溢出，即 `OF=0`。
3. 如果有 `a - b > 0`，那么 `a - b` 的符号位为 0，即 `SF=0`。同时说明发生了溢出（下溢），即 `OF=1`。

第四章 处理器体系结构

读字节的时候要写读了多少个，如 `M8`

$$PC + 1$$

三个特殊情况判断条件：

1. 返回：`IRET in {D_icode, E_icode, M_icode}`
2. 加载使用冒险：`E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB}`
3. 预测失败：`E_icode == IJXX && !e_Cnd`

这部分建议对照 CMU 的完整 HCL 手册复习：<http://csapp.cs.cmu.edu/3e/waside/waside-hcl.pdf>

RISC 指令长度不一定比 CISC 短（书 P249）

考虑一段代码的执行时间时，需要考虑结尾的执行周期，即一段代码总是从第一条代码进入 F 到最后一条代码执行完毕 W 为止

第五章 优化程序性能

第六章 存储器层次结构

对于写入，有如下几种策略：

1. HIT 命中时
 - i. 直写（Write-through）：同时写入高速缓存和内存
 - ii. 写回（Write-back）：只写入高速缓存，内存中的数据不变，只有在修改后的高速缓存被驱逐时才写入内存
2. MISS 未命中时
 - i. 写分配（Write-allocate）：将块读入高速缓存，再写入高速缓存
 - ii. 不写分配（No-write-allocate）：直接写入内存，不读入（也不写）高速缓存

对于 DRAM，访问过程如下：

1. 从地址总线发起行地址（RAS，Row Address Strobe）请求，将对应的行数据读入行缓冲器
2. 从地址总线中发起列地址（CAS，Column Address Strobe）请求，从行缓冲器中读取对应的列数据

DRAM 以二维数组的形式组织，因而对于 RAS 和 CAS，可以共享引脚。

SRAM: 访问比 DRAM 快，但是成本更高，所以一般用于高速缓存

DRAM: 访问比 SRAM 慢，但是成本更低，所以一般用于主存，需要刷写

FPM DRAM: Fast Page Mode DRAM，快页模式 DRAM，可以连续读取同一行的数据，不需要每次都发起 RAS 请求

EDO DRAM: Extended Data Out DRAM，扩展数据输出 DRAM，可以在 CAS 请求后继续读取下一列的数据，不需要等待 CAS 请求结束，让 CAS 信号更加紧密

SDRAM: Synchronous DRAM，同步 DRAM，时钟信号和数据信号同步，可以在时钟信号上升沿读取数据，速度更快

DDR SDRAM: Double Data Rate SDRAM，双倍数据率 DRAM，可以在上升沿和下降沿都进行数据传输，即每个时钟周期传输两次数据，速度更快

对于 S/B/t，三个参数均要求是 2 的幂（可以理解为需要从地址解析他们，而地址是二进制表示的，所以任何一部分都必须是 2 的幂，从而可以建立一个满射），且 $S * B = C$ ，即 S 为组数，B 为每组的块数，C 为高速缓存的大小。

对于内存及以上存储， $1GB=2^{30} B$ ， $1MB=2^{20} B$ ， $1KB=2^{10} B$ ， $1B=8b$ ，即是按照 2 的幂来计算的。

对于硬盘， $1GB=10^9 B$ ， $1MB=10^6 B$ ， $1KB=10^3 B$ ， $1B=8b$ ，即是按照 10 的幂来计算的。

$1 \text{ RPM} = 1 \text{ 转/分} = 1 \text{ 转/60 秒}$ ，所以计算的时候要除以 60。

动态分配内存中，对齐指的是块大小（含头脚）为对齐字节数的整数倍，有效载荷的起始字节对齐，这个过程可以通过调整内部碎片来实现。比如，16 字节对齐+4 字节头部/脚部的设定下，头部的起始位置可以是 0xb(12)，而对应的有效载荷的起始位置就是 0x10(16)，如果分配一个 12 字节的有效载荷，则以内部碎片填充至 32 字节（含头脚 8 字节），使得整个块大小为 32 字节。

第七章 链接

`ccp -> .i -> cc1 -> .s -> as -> .o -> ld -> prog(exec)`

动态链接器负责链接共享库，运行时加载共享库

位置无关代码：可以被加载到内存的任何位置，不需要重定位，共享库的编译总是需要加上 `-fPIC` 选项

编译时打桩：需要访问源代码

链接时打桩：需要访问可重定向目标文件

运行时打桩：只需要访问可执行文件