CSAPP 笔记

adamanteye

1. Representing and Manipulating Information

1.1. Integer

`char` 类型是 `signed` 还是 `unsigned`,取决于编译器(大部分编译器认为是 `signed`).如果需要明确指定,可以使用 `signed char` 或者 `unsigned char`.

移位运算是从左到由结合的.

- `<< k` 表示左移k位.
- `>> k` 表示右移k位.如果是逻辑右移,填充 `0`.如果是算术右移,填充最高位.

C 标准没有规定有符号数的右移是算术的还是逻辑的,但几乎所有编译器-平台对有符号数执行算术右移.

无符号数的表示:

$$\mathrm{B2U}_w(\vec{x}) := \sum_{i=0}^{w-1} x_i 2^i$$

有符号数(补码)的表示:

$$\mathrm{B2T}_w(\vec{x}) := -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

两者间的转换:

$$\mathrm{T2U}_{w(x)} = x + x_{w-1}2^w$$

$$\mathrm{U2T}_{w(x)} = x - x_{w-1}2^w$$

无符号数必须执行逻辑右移.

微妙的特例 补码表示中,只有 `0` 和 `0x80000000` 的负数表示是它自身.

1. 可以依靠符号位判断是 `0` 或者不是 `0`.

```cpp
int logicalNeg(int x) {
    int a = x | (~x + 1);
    return (a >> 31) + 1; // 等价于 !x
}
```

2. 对于 `x`, `y` 同号的减法(肯定不会溢出),为了处理 `x` 是 `0x80000000` 的边界情况,需要写成`~x+y-1`.

常量乘法的优化: 编译器会将 `x * 14` 这样的常量乘法优化为`(x >> 4) - (x >> 1)`.后面会提到 LEA 指令,更详细地讨论这个问题.

1.2. Floating Point

IEEE 754 标准规定了浮点数与浮点运算,这套体系被称为 IEEE 浮点数.

二进制浮点数只能精确表示可以写成$x \times 2^y$形式的小数.

1.2.1. IEEE Floating-Point Representation

IEEE 浮点数使用以下表示:

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

- s是符号位,对于表示 `0` 的情况则有特殊处理
- M的范围是$[0, 1-\varepsilon)$或$[1, 2-\varepsilon)$,位数记为n
- E的位数记为k
- 规定$\mathrm{Bias} = 2^{k-1} - 1$

32 位浮点数 `23` 位M, `8` 位E

64 位浮点数 `52` 位M, `11` 位E

如何解读上述表述,分为 3 种情况:

Normalized Values `exp` 不是全 `0`,也不是全 `1`

$$\begin{aligned} E &= e - \mathrm{Bias} \\ &= e_{k-1}\cdots e_1 e_0 - (2^{k-1} - 1) \end{aligned}$$

因此对于 32 位和 64 位浮点数,E的范围分别是$[-127, +128]$以及$[-1023, +1024]$.但考虑到这里 `exp` 不能是全 `0` 或者全 `1`,因此最终为$[-126, +127]$以及$[-1022, +1023]$.

`frac` 解释为$M = 1 + f = 1.f_{n-1}\cdots f_1 f_0$.这是为了无开销地增加一位精度.

Denormalized Values `exp` 全 `0`

$$E = 1 - \mathrm{Bias}, M = f$$

这种情况可以表示$+0.0$与-0.0,也利于表示非常接近0.0的数.

Special Values `exp` 全 `1`

- `frac` 全 `0`,表示$\pm\infty$

- `frac` 不是全 `0`,称为 `NaN`(Not a number)

从最大的 Denormalized Value 到最小 Normalized Value 的过渡是平滑的.此外,如果将浮点数解读为无符号整数,仍然保持原先的大小关系.这是有意设计,使得浮点数的排序可以转化为整数,然后进行排序.

1.2.2. Rounding

IEEE 浮点数标准定义了 4 种修约(Rounding)模式:

Mode	Description
Round-to-even	find a closest match, or round either upward or downward such that the least significant digit of the result is even
Round-toward-zero	downward if greater than zero, upward otherwise
Round-down	$x^- \leq x$
Round-up	$x^+ \geq x$

表 1　Rounding Modes

Round-to-even 是最常用的模式,不会引入统计误差.

2. Machine-Level Representation of Programs

如果想查看详细的机器指令,可以参考 coder64 edition | X86 Opcode and Instruction Reference 1.12.

2.1. Historical Perspective

8086(1978, 29K 晶体管)是第一代 x86 系列处理器,16 位寄存器. i386(1985, 275K 晶体管)扩展到 32 位,成为第一个可以运行 UNIX 的 x86 处理器. Pentium 4E(2004, 125M 晶体管)引入超线程技术与 EM64T(现在称为 x86-64). Core i7, Sandy Bridge(2011, 1.16B 晶体管)引入了 AVX 指令集.

2.2. Program Encodings

对 `gcc` 或 `clang`, `-S` 编译选项输出汇编文件(以`.s` 结尾). `-c` 编译选项输出目标文件(以`.o` 结尾). `-o` 编译选项指定输出文件名.

ATT 格式(`gcc`, `objdump` 的默认格式)与 Intel 格式(Intel, 微软的默认格式)都是汇编语言的表示方式. Intel 格式中操作数逆向排列.

如果要在 C 中使用汇编代码,可以通过 `asm` 关键字内联使用,也可以链接汇编文件.

2.3. Data Formats

C declaration	Intel data type	Assembly-code suffix	Size(bytes)
`char`	Byte	b	1
`short`	Word	w	2
`int`	Double word	l	4
`long`	Quad word	q	8
`char *`	Quad word	q	8
`float`	Single precision	s	4
`double`	Double precision	l	8

表 2　Size of C data types in x86-64

x86 历史上实现过 `10` 字节的浮点数扩展,在 C 中通过声明 `long double` 使用.[1]但如果不是 x86 平台,那么 `long double` 可能回退到 `double`,并且 `10` 字节浮点数的性能也不如 `float` 或 `double`.

后缀 `l` 既表示 `int` 又表示 `double`,不过因为浮点运算所用的指令,寄存器都和整数运算不同,所以不会混淆.

2.4. Accessing Information

x86-64 架构有 16 个 64 位的通用寄存器,它们既存储整数,也存储指针.

[1] long double - Wikipedia

最初的 8086 处理器只有 8 个 16 位寄存器(从 %ax 到 %bp).IA32 扩展到 32 位,原先的 8 个 16 位寄存器变为 8 个 32 位寄存器(从 %eax 到 %ebp).到了 x86-64,所有寄存器扩展为 64 位,且增加了 8 个新的寄存器(从 %r8 到 %r15).

63	31	15	7	Purpose
%rax	%eax	%ax	%al	Return value
%rbx	%ebx	%bx	%bl	Callee saved
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	Callee saved
%rsp	%esp	%sp	%spl	Stack pointer
%r8	%r8d	%r8w	%r8b	5th argument
%r9	%r9d	%r9w	%r9b	6th argument
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	Callee saved
%r13	%r13d	%r13w	%r13b	Callee saved
%r14	%r14d	%r14w	%r14b	Callee saved
%r15	%r15d	%r15w	%r15b	Callee saved

表 3　Integer registers

寄存器可以按照 1,2,4,8 字节的方式被使用,对应不同的指令后缀.其中对于目标寄存器的更新,有这样的规则:

- 使用 1,2 字节的指令,目标寄存器的高位字节保持不动.
- 使用 4 字节的指令,目标寄存器的高位字节会被清零.

%rsp 寄存器指向栈顶,其他 15 个寄存器的使用更为灵活.

2.4.1. Operand Specifiers

Type	Form	Operand value	Name
Immediate	$Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[\text{Imm}]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$\text{Imm}(r_b)$	$M[\text{Imm}+R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b]+R[r_i]]$	Indexed
Memory	$\text{Imm}(r_b, r_i)$	$M[\text{Imm}+R[r_b]+R[r_i]]$	Indexed
Memory	(r_b, r_i, s)	$M[R[r_b]+R[r_i]\cdot s]$	Scaled indexed
Memory	$\text{Imm}(r_b, r_i, s)$	$M[\text{Imm}+R[r_b]+R[r_i]\cdot s]$	Scaled indexed

表 4　Operand forms

表 4 中的s只能是 1, 2, 4, 8 中的一个.

最通用的内存寻址模式是$\text{Imm}(r_b, r_i, s)$,这个表达式的值称为 **effective address**. LEA 指令可以计算这样的地址,并将地址加载到其他寄存器.

2.5. Data Movement Instructions

Instruction	Discription
movb S, D	Move byte
movw S, D	Move word
movl S, D	Move double word
movq S, D	Move quad word
movabsq S, D	Move quad word

表 5　Simple data movement instructions

表中为 MOV 系列. 例如:

```assembly
1  movl $0x4050,%eax
2  movq %rax,-12(%rbp)
```

x86-64 规定,移动指令的目标和源不能都是内存地址,如果确实有这样的需求,应当先将内存地址上的值加载到寄存器中,再从寄存器中存入内存地址.

注意 movl 如果以寄存器作为目标,会将高 4 字节置零,这是之前提到的规则.

除此之外还有 MOVZ 和 MOVS 系列,分别为 zero-extending 以及 sign-extending.

2.6. Pushing and Popping Stack Data

x86-64 中,栈由高位向低位增长,也就是栈顶的地址反而更小.栈指针为 %rsp.

PUSH 和 POP 指令的作用如下:

```assembly (x86_64)
pushq %rbp
```

相当于

```assembly (x86_64)
subq $8,%rsp
movq %rbp,(%rsp)
```

关于代码和数据的地址空间是否独立,以及哈佛架构,可以参考这些文章:

- Memory, Pages, mmap, and Linear Address Spaces

2.7. Arithmetic and Logical Operations

Instr	Operand	Effect	Description
leaq	S, D	D <- &S	Load effective address
INC	D	D <- D + 1	Increment
DEC	D	D <- D - 1	Decrement
NEG	D	D <- -D	Negate
NOT	D	D <- ~D	Complement
ADD	S, D	D <- D + S	Add
SUB	S, D	D <- D - S	Substract
IMUL	S, D	D <- D * 1	Multiply
XOR	S, D	D <- D - 1	Exclusive-or
OR	S, D	D <- D | S	Or
AND	S, D	D <- D & S	And
SAL	k, D	D <- D << k	Left shift
SHL	k, D	D <- D << k	Left shift(same as SAL)
SAR	k, D	D <- D >> k	Arithmetic right shift
SHR	k, D	D <- D >> k	Logical right shift

表 6 Integer arithmetic operations

2.8. Control

2.8.1. Condition Codes

CF Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow yielded zero.

ZF Zero flag. The most recent operation yielded zero

SF Sign flag. The most recent operation yielded a negative value.

OF Overflow flag. The most recent operation caused a two's-complement overflow: either negative or positive.

注意 leaq 不对上面四个标志产生影响,因为操作的是"内存地址".

Instr	Operand	Based on
CMP	S1, S2	S2 - S1
TEST	S1, S2	S2 & S1

表 7 Comparsion and test instructions

CMP 与 TEST 只会操作 4 个标志,不修改操作数.

TEST 常用于测试单个寄存器是否满足某条件,例如:

```assembly (x86_64)
testq %rax,%rax
```

2.8.2. Accessing the Condition Codes

Instr	Synonym	Effect	Set condition
sete D	setz	D <- ZF	Zero
setne D	setnz	D <- ZF	Not zero
sets D		D <- SF	Negative
setns D		D <- ~SF	Nonnegative
setg D	setnle	D <- ~(SF^OF)&~ZF	Greater (Signed)
setge D	setnl	D <- ~(SF^OF)	Greater or Equal (Signed)
setl D	setnge	D <- SF^OF	Less (Signed)

Instr	Synonym	Effect	Set condition
`setle D`	`setng`	`D <- (SF^OF)|ZF`	Less or Equal (Signed)
`seta D`	`setnbe`	`D <- ~CF&~ZF`	Above (Unsigned)
`setae D`	`setnb`	`D <- ~CF`	Above or equal (Unsigned)
`setb D`	`setnae`	`D <- CF`	Below (Unsigned)
`setbe D`	`setna`	`D <- CF|ZF`	Below or equal (Unsigned)

表 8　The SET instructions

注意 SET 操作 1 字节大小的寄存器或内存.

2.8.3. Jump Instructions

Instr	Synonym	Jump condition	Description
`jmp Label`		`1`	Direct jump
`jmp *Operand`		`1`	Indirect jump
`je Label`	`jz`	`ZF`	Equal or zero
`jne Label`	`jnz`	`~SF`	Not equal or not zero
`js Label`		`SF`	Negative
`jns Label`		`~SF`	Nonnegative
`jg Label`	`jnle`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge Label`	`jnl`	`~(SF^OF)`	Greater or equal (Signed)
`jl Label`	`jnge`	`SF^OF`	Less (Signed)
`jle Label`	`jng`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja Label`	`jnbe`	`~CF&~ZF`	Above (Unsigned)
`jae Label`	`jnb`	`~CF`	Above or equal (Unsigned)
`jb Label`	`jnae`	`CF`	Below (Unsigned)
`jbe Label`	`jna`	`CF|ZF`	Below or equal (Unsigned)

表 9　The JUMP instructions

2.9. Procedures

2.9.1. The Run-Time Stack

如果要在栈上分配或释放空间:

```assembly
subq $16, %rsp
addq $16, %rsp
```

注意 x86-64 约定,进入函数后,`%rsp` 应该对齐到 16 字节.而 CALL 指令会将 8 字节的返回地址压栈,所以调用 CALL 之前,应该保证 `%rsp` 模 16 余 8.

如果不会调用其他函数,那么恢复 `%rsp` 即可,没有其他限制.

2.9.2. Data Transfer

x86-64 中,最多有 6 个寄存器可以传参.如果函数有多于 6 个参数,剩余的将通过栈传参.

为了调用函数所做的准备是,首先将参数拷贝到寄存器中,推入最后一个参数,重复进行直推入第 7 个参数(如果有多于 6 个参数).推入返回地址.

以上调用约定等是 ABI(Application Binary Interface)规范的一部分,不同的平台(Linux, Windows)以及不同的架构都有自己的 ABI 规范.

2.9.3. Local Storage on the Stack

一般局部变量会尽可能地分配在寄存器上,但在下面三种情况下需要分配在栈上:

- 局部变量的地址要被使用,例如传递给其他要调用的函数
- 寄存器不够用
- 需要通过指针访问的数组,结构体等

2.9.4. Local Storage in Registers

寄存器 `%rbx`, `%rbp` 以及 `%r12` 至 `%r15` 都是被调用者保存(callee-saved).其他除了 `%rsp` 的寄存器都是调用者保存(caller-saved).

2.10. Array Allocation and Access

2.11. Floting-Point Code

3. 附录

北大一位学长写了 15 年版本的 Lab,代码很值得学习.