

Machine Programming

元培数科 王恩博

Let's get started →

2025/9/24



浮点数表示

floating point

IEEE 754 标准

本质是对实数的近似

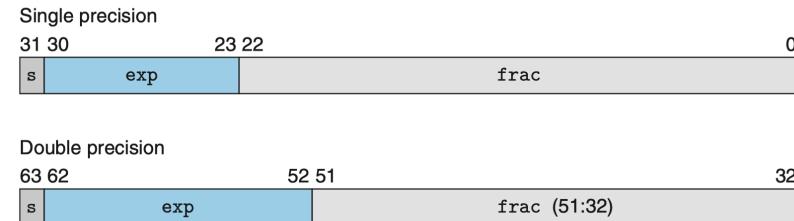
- 符号位: 1 位
- 指数位: e 位
- 尾数位: m 位

实际表示:

- 单精度浮点数 (`float`) : $e = 8, m = 23$
- 双精度浮点数 (`double`) : $e = 11, m = 52$

各个位长多少要牢牢记清楚! 考试会考!

注意, 0 的表示有两种, +0 和 -0



理解 IEEE 754

underlying IEEE 754

规格化值

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

- s 为符号位, M 为尾数, E 为阶码
- $E = [e] - Bias$, 其中 $[e]$ 为阶码处这 e 位的实际值, $Bias = 2^{e-1} - 1$
- 尾数隐含的 1 (Implied leading 1), 可以获得 1 位额外精度

非规格化值

$$V = (-1)^s \times (0.M) \times 2^{1-Bias}$$

- $E = 1 - Bias$, 虽然此时 $[e] = 0$, 但如此规定可以实现规格化值和非规格化值的平滑过渡
- 尾数没有隐含的 1

非规格化值的看似反直觉的定义实现了**规格化值和非规格化值的平滑过渡**

理解 IEEE 754

underlying IEEE 754

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

偏置 : $Bias = 2^{k-1} - 1$

指数: $E = e - Bias$

特殊值

- 最小的非规格化数 $not_{min} = 2^{1-Bias} * 2^{-n} = 2^{2-2^{k-1}} * 2^{-n} = 2^{2-2^{k-1}-n}$
- 最大的规格化数 $not_{max} = 2^{1-Bias} * (1 - 2^{-n}) = 2^{2-2^{k-1}} * (1 - 2^{-n})$
- 最小的规格化数 $yes_{min} = 2^{1-Bias} * (1) = 2^{2-2^{k-1}}$
- 最大的规格化数 $yes_{max} = 2^{(2^k-1)-Bias-1} * (2 - 2^{-n}) = 2^{2^{k-1}-1} * (2 - 2^{-n})$

理解 IEEE 754

underlying IEEE 754

无穷 ($\pm\infty$)

- 符号位 s 区分正无穷/负无穷
- 阶码 $[e] = 1\dots1$ (全1) , 尾数 $M = 0$
- 表示无穷大，在计算中可以用于表示溢出或未定义的结果

零

- 符号位 s 任意，阶码 $[e] = 0\dots0$ (全0) , 尾数 $M = 0$

非数值 (NaN, Not a Number)

- 符号位 s 任意, 阶码 $[e] = 1\dots1$ (全1) , 尾数 $M \neq 0$
- NaN 用于表示未定义或无法表示的值，例如 $0/0$ 或 $\sqrt{-1}$
- 通常在比较中被认为不等于任何值，包括自身

特殊值的应用

- 处理异常情况：如除零错误、溢出、下溢等
- 提高计算的鲁棒性，避免程序崩溃
- 在数值算法中用于标记和处理特殊情况

理解 IEEE 754

underlying IEEE 754

平滑过渡

考慮 $e = 2, m = 3$

0 01 000

0 00 111

$$Bias = 2^{e-1} - 1 = 2^{2-1} - 1 = 1$$

$$1.000_2 \times 2^{01_2-1} = 1.000_2$$

$$0.111_2 \times 2^{1-1} = 0.111_2$$

浮点舍入

rounding

由于浮点数是对实数的近似，浮点数可以精确表示的实数是有限的，所以舍入是不可避免的。

实数舍入到浮点数

- 向偶数舍入 (round-to-even) (round-to-nearest)
- 类比“四舍六入五成双”，**避免统计偏差**

$$1.234 \Rightarrow 1.0$$

$$1.678 \Rightarrow 2.0$$

$$1.500 \Rightarrow 2.0$$

* 实际上，这一过程发生在浮点数精度最末位

浮点数转整型

- 如果舍入，向零舍入
- 如果溢出，C 语言未规定 (undefined behavior)，各自处理 (Intel: T_{\min}^w ，即舍入到限定最接近的数)

浮点类型转换

- 舍入规则
 - 一般情况看舍入位，0 舍1入
 - 类似“四舍六入五成双”，若舍入位为1且后续位全0，向偶舍入
- int/float,double 互转
 - int 转 float
 - 可能发生舍入
 - double 转 float
 - 可能溢出，可能舍入
 - float,double 转 int
 - 如果需要舍入，向零舍入
 - 如果发生溢出，未定义行为，一般得到 T_{min}
 - 其余情况，得到精确值
- 和 long 之间的转换类似，但需另作讨论

浮点运算

floating point arithmetic

- 加法可交换，加法不可结合，`Nan` 没有加法逆元
- **浮点加法单调性**，如果 $a \geq b$ ，那么对于任何 a, b 以及 x 的值，除了 `Nan` 都有 $x + a \geq x + b$
- 乘法可交换，乘法不可结合，乘法在加法上不可分配
- 小心特殊值：`+inf`, `-inf`, `Nan` (`Nan` 不能通过任何运算变为其它值)

* 这些东西说了没用，得你自己做题踩坑才会知道。

- Invalid operation: 零乘无穷、零除零、无穷除无穷、无穷绝对值相减...
- Division by zero: 有穷数除零结果为无穷
- Overflow: 无穷
- Underflow: 舍入结果
- Inexact: 舍入结果

更多详情请见 [IEEE754 异常处理](#)

浮点运算

floating point arithmetic

01 加法注意事项

Mathematical Properties of FP Add

■ Compare to those of Abelian Group

- Closed under addition? **Yes**
 - But may generate infinity or NaN
- Commutative? **Yes**
- Associative? **No**
 - Overflow and inexactness of rounding
 - $(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14$
- 0 is additive identity? **Yes**
- Every element has additive inverse? **Almost**
 - Yes, except for infinities & NaNs

■ Monotonicity

- $a \geq b \Rightarrow a+c \geq b+c?$ **Almost**
 - Except for infinities & NaNs

02 乘法注意事项

Mathematical Properties of FP Mult

■ Compare to Commutative Ring

- Closed under multiplication? **Yes**
 - But may generate infinity or NaN
- Multiplication Commutative? **Yes**
- Multiplication is Associative? **No**
 - Possibility of overflow, inexactness of rounding
 - Ex: $(1e20*1e20)*1e-20 = \text{inf}, 1e20*(1e20*1e-20) = 1e20$
- 1 is multiplicative identity? **Yes**
- Multiplication distributes over addition? **No**
 - Possibility of overflow, inexactness of rounding
 - $1e20*(1e20-1e20) = 0.0, 1e20*1e20 - 1e20*1e20 = \text{NaN}$

■ Monotonicity

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c?$ **Almost**
 - Except for infinities & NaNs

Machine Basics

数据格式

- 基本的数据格式，`b, w, l, q`，对应的 `size` 和 `C declaration`

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

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

寄存器

- %rax: accumulator
- %rbx: base
- %rcx: count
- %rdx: data
- %rsi: source index
- %rdi: destination index
- %rsp: stack pointer
- %rbp: base pointer
- %rip(PC): instruction pointer
- 注意名称由长到短变化规律



Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

寻址模式

$$Imm(r_b, r_i, s) \Rightarrow Imm + R[r_b] + R[r_i] * s$$

- 注意基址和变址寄存器必须是 64 位寄存器，比例因子必须是 1,2,4,8
 - (%eax) 不合法
 - (%rax,3)不合法

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

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

mov指令

- 具有指令后缀 b/w/l/q
- 两个操作数不能都是内存，寄存器大小必须和指令后缀匹配
- 扩展数据传送指令 movz/movs, z 表示零扩展，s 表示符号扩展
- movl 指令以寄存器为目的时，会将高位 4 字节清零，因此没有 movzlq
- 特殊指令 movabsq 和 cltq
 - movabsq 将 64 位立即数传送至寄存器
 - cltq 将 %eax 有符号扩展到 %rax
- 计算机的底层是很机械的，所以很多指令都可执行，但要注意匹配

Instruction	Effect	Description
MOV S, D	$D \leftarrow S$	Move
movb		Move byte
movw		Move word
movl		Move double word
movq		Move quad word
movabsq I, R	$R \leftarrow I$	Move absolute quad word

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
movzbw		Move zero-extended byte to word
movzbl		Move zero-extended byte to double word
movzwl		Move zero-extended word to double word
movzbq		Move zero-extended byte to quad word
movzwq		Move zero-extended word to quad word

Instruction	Effect	Description
MOVS S, R	$R \leftarrow \text{SignExtend}(S)$	Move with sign extension
movsbw		Move sign-extended byte to word
movsbl		Move sign-extended byte to double word
movswl		Move sign-extended word to double word
movsbq		Move sign-extended byte to quad word
movswq		Move sign-extended word to quad word
movslq		Move sign-extended double word to quad word

cltq	$\%rax \leftarrow \text{SignExtend}(\%eax)$	Sign-extend %eax to %rax
------	---	--------------------------

mov指令

■ Legal instruction

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

■ illegal instruction

<code>movb \$0xF, (%ebx)</code>	<i>Cannot use %ebx as address register</i>
<code>movl %rax, (%rsp)</code>	<i>Mismatch between instruction suffix and register ID</i>
<code>movw (%rax),4(%rsp)</code>	<i>Cannot have both source and destination be memory references</i>
<code>movb %al,%sl</code>	<i>No register named %sl</i>
<code>movl %eax,\$0x123</code>	<i>Cannot have immediate as destination</i>
<code>movl %eax,%dx</code>	<i>Destination operand incorrect size</i>
<code>movb %si, 8(%rbp)</code>	<i>Mismatch between instruction suffix and register ID</i>

push&pop指令

- push: 先将 %rsp 减 8, 再压栈
- pop: 先弹栈, 再将 %rsp 加 8
- 先这么理解, 原理会在第四章学习 (与表象并不完全一致)
- call: push + jmp, 可进行间接跳转
- ret: pop + jmp

pushq %rsp

效果: 让 %rsp 的值入栈
入栈的值是 %rsp 还是 %rsp - 8?

popq %rsp

效果: 弹出栈顶的值, 存储到 %rsp 中
存储到 %rsp 中的值是 %rsp + 8 还是 (%rsp)?

Instruction	Effect	Description
pushq <i>S</i>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Push quad word
popq <i>D</i>	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Pop quad word

Figure 3.8 Push and pop instructions.

常见算术操作

- leaq 进行优化运算
- sub 指令是后减前
- 移位操作的第二个操作数必须是立即数或 %cl

Instruction		Effect	Description
leaq	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \wedge S$	Exclusive-or
OR	S, D	$D \leftarrow D \vee S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D \ll k$	Left shift
SHL	k, D	$D \leftarrow D \ll k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D \gg_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D \gg_L k$	Logical right shift

Figure 3.10 Integer arithmetic operations. The load effective address (leaq) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation \gg_A and \gg_L to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

特殊算术操作

- 乘法：高位 %rdx，低位 %rax
- 除法：模 %rdx，商 %rax
- 带 i 为有符号

Instruction	Effect	Description
imulq S	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
mulq S	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply
cqto	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word
idivq S	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
divq S	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide

Figure 3.12 Special arithmetic operations. These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers %rdx and %rax are viewed as forming a single 128-bit oct word.

解码复杂表达式

decode complex expression

1. 从变量名开始
2. 往右读直到读到右括号或到底
3. 往左，忽略读过的
4. 上面的括号均不包括成对的括号
5. `*` 读作“指针，指向”
6. `[x]` 读作“长为 x 的数组，元素类型为”
7. `(...)` 读作“函数，返回值为” + 上面提的成对括号
(参数列表可选)
8. 如果为匿名类型，手动补充变量名 (参数列表里会出现)

```
int *(*p[2])[3];
```

1. `p` : 变量名
2. `[2]` : 往右读，这是一个长度为 2 的数组，元素类型为...
3. `)` : 往左读，忽略读过的
4. `(*p[2])` : 这是一个指针，指向...
5. `(*p[2])[3]` : 一个长度为 3 的数组
6. `;` : 到底了，往左读
7. `*(*p[2])[3]` : 一个指针，指向...
8. `int *(*p[2])[3]` : `int` 类型

合并：这是一个长度为 2 的数组，元素类型为指针，指向一个长度为 3 的数组，数组元素类型为指针，指向 `int`。

解码复杂表达式

decode complex expression

1. 从变量名开始
2. 往右读直到读到右括号或到底
3. 往左，忽略读过的
4. 上面的括号均不包括成对的括号
5. `*` 读作“指针，指向”
6. `[x]` 读作“长为 x 的数组，元素类型为”
7. `(...)` 读作“函数，返回值为” + 上面提的成对括号
(参数列表可选)
8. 如果为匿名类型，手动补充变量名 (参数列表里会出现)

```
int func();
```

函数，返回值为 `int`

```
void func(int a, float b);
```

函数，返回值为 `void`，参数列表为 `int a` 和 `float b`

```
int* func();
```

函数，返回值为 `int*`

```
int (*func)();
```

函数指针，指向返回值为 `int` 的函数

解码复杂表达式

decode complex expression

Quiz

```
int *(*p[2])[3];
```

你可以在 cdecl.org 验证，或者尝试其他表达式。

让我们逐步解码：

1. p : 变量名
2. p[2] : 一个长度为 2 的数组，元素类型为...
3. *p[2] : 一个指针，指向...
4. (*p[2])[3] : 一个长度为 3 的数组，元素类型
为...
5. *(*p[2])[3] : 一个指针，指向...
6. int *(*p[2])[3] : int 类型

合并：声明 p 为一个包含 2 个元素的 数组，每个元素是 指向一个包含 3 个元素的 数组 的指针， 这些数组的元素是指向 int 类型的指针。

解码复杂表达式

- 指针运算动图：阅读方法：先找变量名，一层一层括号往外读

Tips

- 国庆前剩余的三节课是汇编的关键部分，Control本节做一些练习
- Procedures 和 Data 考前来不及复习做题，十一注意自己复习，做题，尤其是对函数调用传参与返回的理解，往年题中的汇编填空涉及递归等调用方法。
- 复习时整体理解和细节两手抓，参考课本和往年题，汇编部分细节更多更杂，**注意各种特例**

作业讲评

作业讲评

1. 对于非规格化数, $E = 1 - \text{Bias}$
2. 正确理解二进制值、十进制值, 尤其注意 0
3. $(dx+dy)+dz == dx+(dy+dz)$
4. $(dxdy)dz == dx(dydz)$

注意datalab截止时间!

习题试炼

习题试炼1

6. 下列寻址模式中，正确的是：

- A. (%eax, , 4)
- B. (%eax, %esp, 3)
- C. 123
- D. \$1(%ebx, %ebp, 1)

习题试炼1

6. 下列寻址模式中，正确的是：

- A. (%eax, , 4)
- B. (%eax, %esp, 3)
- C. 123
- D. \$1(%ebx, %ebp, 1)

选择 C. A 中不能省略 Ei, B 中比例因子错误, D 中偏移地址表示错误。

习题试炼2

4. 在 x86-64 下，以下哪个选项的说法是错误的？
- A) movl 指令以寄存器作为目的时，会将该寄存器的高位 4 字节设置为 0
 - B) cltq 指令的作用是将%eax 符号扩展到%rax
 - C) movabsq 指令只能以寄存器作为目的
 - D) movswq 指令的作用是将零扩展的字传送到四字节目的

习题试炼2

4. 在 x86-64 下，以下哪个选项的说法是错误的？
- A) movl 指令以寄存器作为目的时，会将该寄存器的高位 4 字节设置为 0
 - B) cltq 指令的作用是将%eax 符号扩展到%rax
 - C) movabsq 指令只能以寄存器作为目的
 - D) movswq 指令的作用是将零扩展的字传送到四字节目的

答案： D

movswq 应该是符号扩展

习题试炼3

4. 以下关于 x86-64 指令的描述，说法正确的有几项？
- a) 有符号除法指令 `idivq S` 将`%rdx`（高 64 位）和`%rax`（低 64 位）中的 128 位数作为被除数，将操作数 S 的值作为除数，做有符号除法运算；指令将商存在`%rdx` 寄存器中，将余数存在`%rax` 寄存器中。
 - b) 我们可以使用指令 `jmp %rax` 进行间接跳转，跳转的目标地址由寄存器`%rax` 的值给出。
 - c) 算术右移指令 `shr` 的移位量既可以是一个立即数，也可以存放在单字节寄存器`%cl` 中。
 - d) `leaq` 指令不会改变任何条件码。
- A. 1
 - B. 2
 - C. 3
 - D. 4

习题试炼3

4. 以下关于 x86-64 指令的描述，说法正确的有几项？
- a) 有符号除法指令 `idivq S` 将`%rdx`（高 64 位）和`%rax`（低 64 位）中的 128 位数作为被除数，将操作数 S 的值作为除数，做有符号除法运算；指令将商存在`%rdx` 寄存器中，将余数存在`%rax` 寄存器中。
 - b) 我们可以使用指令 `jmp %rax` 进行间接跳转，跳转的目标地址由寄存器`%rax` 的值给出。
 - c) 算术右移指令 `shr` 的移位量既可以是一个立即数，也可以存放在单字节寄存器`%cl` 中。
 - d) `leaq` 指令不会改变任何条件码。
- A. 1
B. 2
C. 3
D. 4

答案：A

本题考察 x86-64 中的一些基本指令，答案为 A。a 项错误，原因是 `idivq` 将余数存在`%rdx` 中，将商存在`%rax` 里。b 项错误，间接跳转的正确书写格式应为 `jmp *%rax`。c 项错误，算术右移指令应为 `sar`。

习题试炼4

8. 下列关于条件码的描述中，不正确的是（）
- A) 所有算术指令都会改变条件码
 - B) 所有比较指令都会改变条件码
 - C) 所有与数据传送有关的指令都会改变条件码
 - D) 条件码一般不会直接读取，但可以直接修改

习题试炼4

8. 下列关于条件码的描述中，不正确的是（）
- A) 所有算术指令都会改变条件码
 - B) 所有比较指令都会改变条件码
 - C) 所有与数据传送有关的指令都会改变条件码
 - D) 条件码一般不会直接读取，但可以直接修改

答案：C，`leaq` 是 `movq` 指令的变形，它只传送地址，不改变条件码。

答案修订：ABCD 均给分。

AB 没有指明是 x86 指令系统；x86 指令系统中有 SIMD 类指令不改变条件码；

C 数据传送一般不改变条件码；

D 正确，`SAHF`、`STC`、`CLC`、`CMC` 等指令均可以直接写条件码。

习题试炼5

- 加法指令：ADD、ADC、INC、XADD除了INC不影响CF标志位外，都影响条件标志位。 CF、ZF、SF、OF CF最高位是否有进位 DF若两个操作数符号相同而结果符号与之相反OF=1，否则OF=0.
- 减法指令：SUB、SBB、DEC、NEG、CMP、CMWXCHG、CMWXCHG8B 前六种除了DEC不影响CF标志外都影响标志位。 CMWXHG8B只影响ZF。 CF说明无符号数相减的溢出，同时又确实是被减数最高有效位向高位的借位。 OF位则说明带符号数的溢出 无符号运算时，若减数>被减数，有借位CF=1，否则CF=0. OF若两个数符号相反，而结果的符号与减数相同则OF=1.否则OF=0.
- 乘法指令：MUL、IMUL MUL：如果乘积高一半为0，则CF和OF位均为0，否则CF和OF均为1. IMUL：如果高一半是低一半符号的扩展，则CF位和OF位均为0，否则就均为1.
- 除法指令：DIV、IDIV 对所有条件位均无定义。
- 逻辑指令：AND、OR、NOT、XOR、TEST NOT不允许使用立即数，其它4条指令除非源操作数是立即数，至少要有一个操作数必须存放在寄存器中。另一个操作数则可以使用任意寻址方式。 NOT不影响标志位，其余4种CF、OF、置0，AF无定义，SF、ZF、PF位看情况而定。

习题试炼6

6. X86-64 指令提供了一组条件码寄存器；其中 ZF 为零标志，ZF=1 表示最近的操作得出的结构为 0；SF 为符号标志，SF=1 表示最近的操作得出的结果为负数；OF 为溢出标志，OF=1 表示最近的操作导致一个补码溢出（正溢出或负溢出）。当我们在一条 cmpq 指令后使用条件跳转指令 jg 时，那么发生跳转等价于以下哪一个表达式的结果为 1？
- A. $\sim(SF \wedge OF) \wedge \sim ZF$
 - B. $\sim(SF \wedge OF)$
 - C. $SF \wedge OF$
 - D. $(SF \wedge OF) \mid ZF$

习题试炼6

6. X86-64 指令提供了一组条件码寄存器；其中 ZF 为零标志，ZF=1 表示最近的操作得出的结构为 0；SF 为符号标志，SF=1 表示最近的操作得出的结果为负数；OF 为溢出标志，OF=1 表示最近的操作导致一个补码溢出（正溢出或负溢出）。当我们在一条 cmpq 指令后使用条件跳转指令 jg 时，那么发生跳转等价于以下哪一个表达式的结果为 1？

- A. $\sim(SF \wedge OF) \ \& \ \sim ZF$
- B. $\sim(SF \wedge OF)$
- C. $SF \wedge OF$
- D. $(SF \wedge OF) \mid ZF$

答案：A。本题考察 x86-64 条件码，答案为 A。cmpq a, b 相当于通过 $b - a$ 的值来设置条件码。SF \wedge OF 为 1 表示 $b < a$ （减法结果要么负溢出要么为负数），于是 $\sim(SF \wedge OF)$ 表示 $b \geq a$ ，再与上 $b \neq a$ 的条件 ($\sim ZF$)，就可以得到最终结果 ($b > a$)。

习题试炼7

5. 在下列关于条件传送的说法中，正确的是：
- A. 条件传送可以用来传送字节、字、双字、和 4 字的数据
 - B. C 语言中的“?:”条件表达式都可以编译成条件传送
 - C. 使用条件传送总可以提高代码的执行效率
 - D. 条件传送指令不需要用后缀（例如 b, w, l, q）来表明操作数的长度

习题试炼7

5. 在下列关于条件传送的说法中，正确的是：
- A. 条件传送可以用来传送字节、字、双字、和 4 字的数据
 - B. C 语言中的“?:”条件表达式都可以编译成条件传送
 - C. 使用条件传送总可以提高代码的执行效率
 - D. 条件传送指令不需要用后缀（例如 b, w, l, q）来表明操作数的长度

答案：D

解析：A. 条件传送不支持单字节传送；B. 如果“?:”涉及到的两个表达式中有一个出错或者有副作用，用条件传送会导致非法行为；C. 如果被旁路的分支的计算量很大，计算就白做了；D. 从目标寄存器的名字可以推断出条件传送指令的操作数长度

习题试炼8

7. 下列关于程序控制结构的机器代码实现的说法中，正确的是：
- A) 使用条件跳转（conditional jump）语句实现的程序片段比使用条件赋值（conditional move）语句实现的同一程序片段的运行效率高
 - B) 使用条件跳转语句实现的程序片段与使用条件赋值语句实现的同一程序片段虽然效率可能不同，但在 C 语言的层面上看总是有着相同的行为
 - C) 一些 switch 语句不会被 gcc 用跳转表的方式实现
 - D) 以上说法都不正确

习题试炼8

7. 下列关于程序控制结构的机器代码实现的说法中，正确的是：
- A) 使用条件跳转（conditional jump）语句实现的程序片段比使用条件赋值（conditional move）语句实现的同一程序片段的运行效率高
 - B) 使用条件跳转语句实现的程序片段与使用条件赋值语句实现的同一程序片段虽然效率可能不同，但在 C 语言的层面上看总是有着相同的行为
 - C) 一些 switch 语句不会被 gcc 用跳转表的方式实现
 - D) 以上说法都不正确

答案：C。条件跳转本身开销大于条件赋值，但条件赋值会将两个分支中的运算都完成，故分支中的运算较为复杂时，使用条件赋值语句实现的程序效率较低，故 a 错误。分支中的运算带有副作用时，条件跳转语句和条件赋值语句实现的程序行为不同，故 b 错误。switch 语句中的 case 若比较稀疏，则不会被用跳转表的方式实现，故 c 正确。

习题试炼9

3. 在如下代码段的跳转指令中，目的地址是：

400020: 74 F0 je _____

400022: 5d pop %rbp

- A. 400010
- B. 400012
- C. 400110
- D. 400112

习题试炼9

3. 在如下代码段的跳转指令中，目的地址是：

400020: 74 F0 je _____

400022: 5d pop %rbp

- A. 400010 B. 400012 C. 400110 D. 400112

答案：B（有符号跳转，向后跳转-0x10）

习题试炼10

8. 假设某条 C 语言 switch 语句编译后产生了如下的汇编代码及跳转表：

```
movl 8(%ebp), %eax          .L7:  
subl $48, %eax             .long .L3  
cmpl $8, %eax              .long .L2  
ja .L2                     .long .L2  
jmp * .L7(, %eax, 4)       .long .L5  
                           .long .L4  
                           .long .L5  
                           .long .L6  
                           .long .L2  
                           .long .L3
```

在源程序中，下面的哪些（个）标号出现过：

- A. '2', '7'
- B. 1
- C. '3'
- D. 5

习题试炼10

8. 假设某条 C 语言 switch 语句编译后产生了如下的汇编代码及跳转表：

```
movl 8(%ebp), %eax          .L7:  
subl $48, %eax             .long .L3  
cmpl $8, %eax              .long .L2  
ja .L2                     .long .L2  
jmp * .L7(, %eax, 4)       .long .L5  
                           .long .L4  
                           .long .L5  
                           .long .L6  
                           .long .L2  
                           .long .L3
```

在源程序中，下面的哪些（个）标号出现过：

- A. '2', '7'
- B. 1
- C. '3'
- D. 5

选择 C. 标号的取值范围为'0' - , '1'、'2'、'7'、'9'、...等没有出现。

习题试炼11

9. pushq %rbp 的行为等价于以下()中的两条指令。

- A. subq \$8, %rsp movq %rbp, (%rdx)
- B. subq \$8, %rsp movq %rbp, (%rsp)
- C. subq \$8, %rsp movq %rax, (%rsp)
- D. subq \$8, %rax movq %rbp, (%rdx)

习题试炼11

9. pushq %rbp 的行为等价于以下()中的两条指令。

- A. subq \$8, %rsp movq %rbp, (%rdx)
- B. subq \$8, %rsp movq %rbp, (%rsp)
- C. subq \$8, %rsp movq %rax, (%rsp)
- D. subq \$8, %rax movq %rbp, (%rdx)

答案: B。

说明: x86-64 系统中, pushq %rbp 指令将栈指针减 8, 并向其中存入%rbp 寄存器的值 (书 P127)

习题试炼12

9. 对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

```
long f1(long a, long b) {
    return (++a > --b) ? a : b;
}
```

Y N

```
long f2(long *a, long *b) {
    return (*a > *b) ? --(*a) : (*b)--;
}
```

Y N

```
long f3(long *a, long *b) {
    return a ? *a : (b ? *b : 0);
}
```

Y N

```
long f4(long a, long b) {
    return (a > b) ? a++ : ++b;
}
```

Y N

习题试炼12

9. 对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

long f1(long a, long b) { return (++a > --b) ? a : b; }	<input checked="" type="radio"/> Y <input type="radio"/> N
long f2(long *a, long *b) { return (*a > *b) ? --(*a) : (*b)--; }	<input type="radio"/> Y <input checked="" type="radio"/> N
long f3(long *a, long *b) { return a ? *a : (b ? *b : 0); }	<input type="radio"/> Y <input checked="" type="radio"/> N
long f4(long a, long b) { return (a > b) ? a++ : ++b; }	<input checked="" type="radio"/> Y <input type="radio"/> N

【答】f1 由于比较前计算出的 a 与 b 就是条件传送的目标，因此会被编译成条件传送；f2 由于比较结果会导致 a 与 b 指向的元素发生不同的改变，因此会被编译成条件跳转；f3 由于指针 a 可能无效，因此会被编译为条件跳转；f4 会被编译成条件传送，注意到 a 和 b 都是局部变量，return 的时候对 a 和 b 的操作都是没有用的。使用 -O1 可以验证 gcc 的行为。

其他说明

从 C 源代码到汇编代码

- [Linux环境下GCC基本使用详解（含实例）_linux gcc-CSDN博客](#)
- [objdump\(Linux\)反汇编命令使用指南_怎么用objdump反汇编-CSDN博客](#)
- -S 生成汇编代码文件， -c 生成不可执行的二进制文件（未经过链接）
- `gcc x.c -S x.s`
- `gcc -c x.s -o x.o`
- 使用 objdump 可以将二进制汇编文件通过反汇编得到它的汇编代码
- **objdump -d x.o > x.s**
- 右图是 objdump 可能的输出，左侧是汇编代码的二进制编码和对应的存储地址，右侧是汇编代码

```
ubuntu@yaen:~/ICS/HW/02$ gcc -S exam.c -Og
ubuntu@yaen:~/ICS/HW/02$ gcc -c exam.c -Og
ubuntu@yaen:~/ICS/HW/02$ gcc exam.c -o exam -Og
ubuntu@yaen:~/ICS/HW/02$ objdump -s -d exam.o > exam_obj.o.s
ubuntu@yaen:~/ICS/HW/02$ objdump -s -d exam > exam.out.s
```

THANKS

Made by WEB-05

webrun@stu.pku.edu.cn

Reference: [WalkerCH]'s and [Arthals]'s presentations.



博

黑龙江 哈尔滨



扫一扫上面的二维码图案，加我为朋友。