

# Machine Programming

13 元培数科 常欣海

Here we go! →

2024/9/25



# 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 <sub>a</sub>	R[r <sub>a</sub> ]	Register
Memory	Imm	M[Imm]	Absolute
Memory	(r <sub>a</sub> )	M[R[r <sub>a</sub> ]]	Indirect
Memory	Imm(r <sub>b</sub> )	M[Imm + R[r <sub>b</sub> ]]	Base + displacement
Memory	(r <sub>b</sub> , r <sub>i</sub> )	M[R[r <sub>b</sub> ] + R[r <sub>i</sub> ]]	Indexed
Memory	Imm(r <sub>b</sub> , r <sub>i</sub> )	M[Imm + R[r <sub>b</sub> ] + R[r <sub>i</sub> ]]	Indexed
Memory	(,r <sub>i</sub> ,s)	M[R[r <sub>i</sub> ] · s]	Scaled indexed
Memory	Imm(,r <sub>i</sub> ,s)	M[Imm + R[r <sub>i</sub> ] · s]	Scaled indexed
Memory	(r <sub>b</sub> ,r <sub>i</sub> ,s)	M[R[r <sub>b</sub> ] + R[r <sub>i</sub> ] · s]	Scaled indexed
Memory	Imm(r <sub>b</sub> ,r <sub>i</sub> ,s)	M[Imm + R[r <sub>b</sub> ] + R[r <sub>i</sub> ] · 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

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,(%esp)</code>	<i>Immediate--Memory, 1 byte</i>
5	<code>movq %rax,-12(%rbp)</code>	<i>Register--Memory, 8 bytes</i>

- Not `(%esp)` but `(%rsp)`

- 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 $S$	$R[%rsp] \leftarrow R[%rsp] - 8;$ $M[R[%rsp]] \leftarrow S$	Push quad word
popq $D$	$D \leftarrow M[R[%rsp]];$ $R[%rsp] \leftarrow R[%rsp] + 8$	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.

# Machine Control

# 条件码 CC

- 进位标志 **CF** (carry flag) : 无符号溢出 (无论上溢还是下溢)
- 零标志 **ZF** (zero flag) : 零
- 符号标志 **SF** (sign flag) : 负数
- 溢出标志 **OF** (overflow flag) : 有符号溢出
- `leaq` 不会设置任何条件码
- **逻辑操作**会置零 CF 和 OF (**无溢出**)
- **移位操作**会置零 OF, 根据最后一个移出的位改变 CF (无论左移还是右移)
- INC 和 DEC 会改变 ZF 和 OF, 但是**不会改变 CF**

CF	<code>(unsigned) t &lt; (unsigned) a</code>	Unsigned overflow
ZF	<code>(t == 0)</code>	Zero
SF	<code>(t &lt; 0)</code>	Negative
OF	<code>(a &lt; 0 == b &lt; 0) &amp;&amp; (t &lt; 0 != a &lt; 0)</code>	Signed overflow

# 设置条件码

## 直接设置条件码

指令	全称	功能
CLC	clear carry flag	CF清零
STC	set carry flag	CF置位1
CMC	complement carry flag	CF取反
CLD	clear direction flag	DF清零
STD	set direction flag	DF置位1
CLI	clear interrupt endable flag	IF清零, 关闭中断
STI	set interrupt endable flag	IF置位1, 打开中断

# 间接设置条件码

- cmp/test 用于设置 CC, 后缀 b/w/l/q, 行为等同 sub/and

- testq %rax %rax (零测试)
- testq %rax \$MASK (掩码测试)

Instruction		Based on	Description
CMP	$S_1, S_2$	$S_2 - S_1$	Compare
cmpb			Compare byte
cmpw			Compare word
cmpl			Compare double word
cmpq			Compare quad word
TEST	$S_1, S_2$	$S_1 \& S_2$	Test
testb			Test byte
testw			Test word
testl			Test double word
testq			Test quad word

# 访问条件码

- 任意后缀对应的条件码要求熟练掌握
- 不用背但要能快速写出【理解】

Instruction	Synonym	Effect	Set condition
sete $D$	setz	$D \leftarrow ZF$	Equal / zero
setne $D$	setnz	$D \leftarrow \sim ZF$	Not equal / not zero
sets $D$		$D \leftarrow SF$	Negative
setns $D$		$D \leftarrow \sim SF$	Nonnegative
setg $D$	setnle	$D \leftarrow \sim (SF \wedge OF) \& \sim ZF$	Greater (signed >)
setge $D$	setnl	$D \leftarrow \sim (SF \wedge OF)$	Greater or equal (signed $\geq$ )
setl $D$	setnge	$D \leftarrow SF \wedge OF$	Less (signed <)
setle $D$	setng	$D \leftarrow (SF \wedge OF) \mid ZF$	Less or equal (signed $\leq$ )
seta $D$	setnbe	$D \leftarrow \sim CF \& \sim ZF$	Above (unsigned >)
setae $D$	setnb	$D \leftarrow \sim CF$	Above or equal (unsigned $\geq$ )
setb $D$	setnae	$D \leftarrow CF$	Below (unsigned <)
setbe $D$	setna	$D \leftarrow CF \mid ZF$	Below or equal (unsigned $\leq$ )

**Figure 3.14 The SET instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have “synonyms,” that is, alternate names for the same machine instruction.

# 条件跳转-jmp

- 直接跳转 jmp .L1
- 间接跳转 jmp \*%rax 或 jmp \*(%rax)
- 条件跳转只能是直接跳转
- jmp 的二进制编码为相对寻址
  - 如右图, jmp 8 的 8 对应的编码为 03, 因为第二行结束时 PC 为 05, 与 08 相差 03, jmp 5 的 5 对应的编码为 f8, 因为在第五行结束时 PC 为 0d, 与 05 相差 f8 (-8 的十六进制编码)
- 绝对寻址 (只在大程序中使用, 如 > 2M)

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

**Figure 3.15 The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.

# 条件传送-cmov

- 不能传送至内存，不支持单字节传送
- 会自动推断传送数据大小
- 无需预测判断结果，更高效
- 条件跳转可能被优化为条件传送，但以下情形不适用：
  - 表达式求值需要大量运算
  - 表达式求值可能导致错误，如  $\text{val} = p ? *p : 0$
  - 表达式求值可能导致副作用，如  $\text{val} = x > 0 ? (x *= 7) : (x += 3)$

Instruction	Synonym	Move condition	Description
cmove $S, R$	cmovez	ZF	Equal / zero
cmovne $S, R$	cmovnz	$\sim ZF$	Not equal / not zero
cmovs $S, R$		SF	Negative
cmovns $S, R$		$\sim SF$	Nonnegative
cmovg $S, R$	cmovnle	$\sim(SF \wedge OF) \& \sim ZF$	Greater (signed $>$ )
cmovge $S, R$	cmovnl	$\sim(SF \wedge OF)$	Greater or equal (signed $\geq$ )
cmovl $S, R$	cmovnge	SF $\wedge$ OF	Less (signed $<$ )
cmovle $S, R$	cmovng	$(SF \wedge OF) \mid ZF$	Less or equal (signed $\leq$ )
cmova $S, R$	cmovnbe	$\sim CF \& \sim ZF$	Above (unsigned $>$ )
cmovae $S, R$	cmovnb	$\sim CF$	Above or equal (Unsigned $\geq$ )
cmovb $S, R$	cmovnae	CF	Below (unsigned $<$ )
cmovbe $S, R$	cmovna	CF $\mid$ ZF	Below or equal (unsigned $\leq$ )

**Figure 3.18** The conditional move instructions. These instructions copy the source value  $S$  to its destination  $R$  when the move condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.

# 条件分支-switch

- 当 switch 分支数量较多且值的跨度范围较小时会启用跳转表，存放于 .rodata (只读数据区) 中，使用间接跳转
- 不是简单的 if-else，而是空间换时间的策略
- switch 源代码与汇编代码的对应
  - 没有 break 的 case 对应无 jmp 的 label
  - default 有另设 label
  - 多 case 同处理对应同 label 段

## (a) Code

```

void switcher(long a, long b, long c, long *dest)
a in %rsi, b in %rdi, c in %rdx, d in %rcx
1    switcher:
2        cmpq    $7, %rdi
3        ja     .L2
4        jmp     *.L4(%rdi,8)
5        .section .rodata
6        .L7:
7        xorq    $15, %rsi
8        movq    %rsi, %rdx
9        .L3:
10       leaq    112(%rdx), %rdi
11       jmp     .L6
12       .L5:
13       leaq    (%rdx,%rsi), %rdi
14       salq    $2, %rdi
15       jmp     .L6
16       .L2:
17       movq    %rsi, %rdi
18       .L6:
19       movq    %rdi, (%rcx)
20       ret

```

## (b) Jump table

```

1   .L4:
2   .quad   .L3
3   .quad   .L2
4   .quad   .L5
5   .quad   .L2
6   .quad   .L6
7   .quad   .L7
8   .quad   .L2
9   .quad   .L5

```

Figure 3.24 Assembly code and jump table for Problem 3.31.

# 循环语句

do-while/while/for

- 本质都是 goto 的应用
- for 大致等价于初始化 + while (需要仔细处理 continue)
- while 的两种转化，左边称为 jump to the middle，右边称为 **guarded-do**
  - 二者在语义上等价，不过后者可以优化初始的判断（少一次 jmp）

```

if (test-expr)           init-expr;
    then-statement
else                      while (test-expr) {
    else-statement          body-statement
                            update-expr;
}
t = test-expr;           init-expr;           init-expr;
if (!t)                  goto test;           t = test-expr;
    goto false;           if (!t)
    goto done;            goto done;
false:                   then-statement       loop:
    goto done;           body-statement
    update-expr;          update-expr;         loop:
                           body-statement
else-statement          test:                 body-statement
done:                    t = test-expr;        update-expr;
                        if (t)                t = test-expr;
                        goto loop;           if (t)
                                         goto loop;
                                         done;

```

# 作业讲评

# 作业讲评

学号	HW-分数	HW-问题
2300012932	9	存在答案版本错误，注意明确给出你最后认可的唯一答案
2300012935	10	
2300012950	10	
2300012951	9	存在答案版本错误，注意明确给出你最后认可的唯一答案
2300013083	10	
2300013115	10	
2300013158	7	1. error: redeclaration of 'b' with no linkage 2. 审题，63 而不是 31
2300013201	8	答案错误，return $(x * y) \wedge ((y - z) << 63 >> 63);$
2300013222	10	
2300013230	10	
2300013272	10	
2300017788	10	
2300094610	8	答案错误，return $(x * y) \wedge ((y - z) << 63 >> 63);$

# 作业讲评

1. 代码本身有问题
2. 运行结果不正确
3. 编译结果不正确

```

File Edit Selection View Go ... ↵ ⌂ ICS [SSH: 10.129.81.70]
C exam.c ×
HW > 02 > C exam.c > decode2_2300017788(long, long, long)
102 int main()
103 {
104     // rand test x,y,z and exam students result compare to decode2_ans
105     int times = 100;
106     while (times--)
107     {
108         long x = rand();
109         long y = rand();
110         long z = rand();
111         printf("x=%ld, y=%ld, z=%ld\n", x, y, z);
112         long ans = decode2_ans(x, y, z);
113         if (decode2_2300012932(x, y, z) != ans)
114         {
115             printf("decode2_2300012932 error!");
116             printf("ans=%ld, decode2_2300012932=%ld\n", ans, decode2_2300012932(x, y, z));
117         }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS Code
decode2_2300094610 error! ans=-3447141188801390436, decode2_2300094610=2686011723954200040
x=38489372, y=1381455736, z=35095211
decode2_2300012932 error! ans=-385722069870215301, decode2_2300012932=39489172
decode2_2300012951 error! ans=-385722069870215301, decode2_2300012951=385722069870215301
decode2_2300013158 error! ans=-385722069870215301, decode2_2300013158=861634427
decode2_2300013201 error! ans=-385722069870215301, decode2_2300013201=-396366775498890593
decode2_2300094610 error! ans=-385722069870215301, decode2_2300094610=-396366775498890593
x=521595368, y=294702557, z=1726956429
decode2_2300012932 error!, z=1726956429
decode2_2300012932 error! ans=-7470569802193111216, decode2_2300012932=-521595369
decode2_2300013158 error! ans=-7470569802193111216, decode2_2300013158=1930251152
decode2_2300013201 error! ans=-7470569802193111216, decode2_2300013201=153715493884989656
decode2_2300094610 error! ans=-7470569802193111216, decode2_2300094610=153715493884989656
x=536465782, y=-861621536, z=278722862
decode2_2300012932 error! ans=-195923576686178376, decode2_2300012932=-334643782
decode2_2300013158 error! ans=-195923576686178376, decode2_2300013158=-2004938824
decode2_2300013201 error! ans=-195923576686178376, decode2_2300013201=-289764282410286460
decode2_2300094610 error! ans=-195923576686178376, decode2_2300094610=-289764282410286460

```

SSH: 10.129.81.70 ⌂ 0 △ 0 账户

# 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

5. 已知函数 func 的参数超过 6 个。当 x86-64 机器执行完指令 call func 之后，%rsp 的值为 s。那么 func 的第 k( $k > 6$ ) 个参数的存储地址是？
- A.  $s + 8 * (k - 6)$
  - B.  $s + 8 * (k - 7)$
  - C.  $s - 8 * (k - 6)$
  - D.  $s - 8 * (k - 7)$

## 习题试炼12

5. 已知函数 func 的参数超过 6 个。当 x86-64 机器执行完指令 call func 之后，%rsp 的值为 s。那么 func 的第 k( $k > 6$ ) 个参数的存储地址是？
- A.  $s + 8 * (k - 6)$
  - B.  $s + 8 * (k - 7)$
  - C.  $s - 8 * (k - 6)$
  - D.  $s - 8 * (k - 7)$
- · ·

答案：A

本题考察 x86-64 运行时栈帧结构，答案为 A。当执行完 call 指令后，s 处存储的是函数的返回地址；再往上依次是第 7、第 8 个函数参数...故第 k 个函数参数存储在  $s + 8 * (k - 6)$  的地址处。

## 习题试炼13

9. x86 体系结构中，下面哪个说法是正确的？
- A. leal 指令只能够用来计算内存地址
  - B. x86\_64 机器可以使用栈来给函数传递参数
  - C. 在一个函数内，改变任一寄存器的值之前必须先将其原始数据保存在栈内
  - D. 判断两个寄存器中值大小关系，只需要 SF 和 ZF 两个条件码

## 习题试炼13

9. x86 体系结构中，下面哪个说法是正确的？

- A. leal 指令只能够用来计算内存地址
- B. x86\_64 机器可以使用栈来给函数传递参数
- C. 在一个函数内，改变任一寄存器的值之前必须先将其原始数据保存在栈内
- D. 判断两个寄存器中值大小关系，只需要 SF 和 ZF 两个条件码

答案：B

A. leal 指令做普通算术运算；C. caller saved 寄存器才需要；D. 还需要 OF

# 习题试炼14

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14

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。

- x86-64在线系统 Compiler Explorer: <https://godbolt.org/>

# 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
```

# 从 C 源代码到汇编代码

The screenshot displays the ICS (IDA) debugger interface with two windows side-by-side. The left window shows the assembly code for the file `exam.s`, which contains several labels like `.LFB46`, `.LFE46`, `.LFB47`, `.LFE47`, and `.LFB48`, along with various assembly instructions and comments. The right window shows the assembly code for the file `exam_obj.o`, which is the object version of the same source code. Both windows have memory dump panes on the right side. The status bar at the bottom indicates the connection is to `SSH: 10.129.81.70`.

# 作业要求

- 小班分数占比15%，其中作业占比40%左右，课上表现占比60%左右
- 作业从本周开始的评分要求如下：
  - **基本分：60%**，只要按时提交即可得到
    - 提交时间要求：
      - 请于**周二晚18:00**，按照格式提交至邮箱中（我会批改，答案有申诉权利）
        - 课上会给出作业的答案，以及根据学号查看的分数和问题，有问题请在**当天课后提出**
        - 最晚**周三午12:00**，最后DDL（我可能没时间看了，所以会之后直接给出分数）
        - 如果你还想要补交，请于**周三晚23:59**之前提交，会扣除部分分数，再后无需提交。
    - **正确率：40%**，这一部分你可以根据网上答案自行修正，但需明确给出最后你认可的答案
      - 推荐用其他颜色的笔更正，属于你自己写的作业范畴，不会扣分

# THANKS

Made by WalkerCH

[changxinhai@stu.pku.edu.cn](mailto:changxinhai@stu.pku.edu.cn)

Reference: [Weicheng Lin]'s presentation.

