

Machine Prog (Assembly Code) ——Basics & Control section

王善上 贾博暄 倪嘉怡 许珈铭

2023.9.27

Preface

许珈铭

C/C++ 编译

- Pre-processor
- Compiler
- Assembler
- Linker&loader

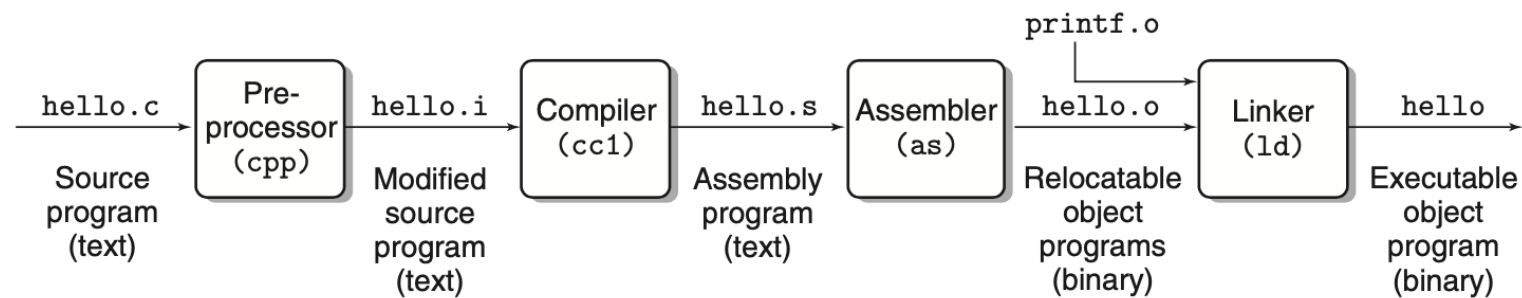


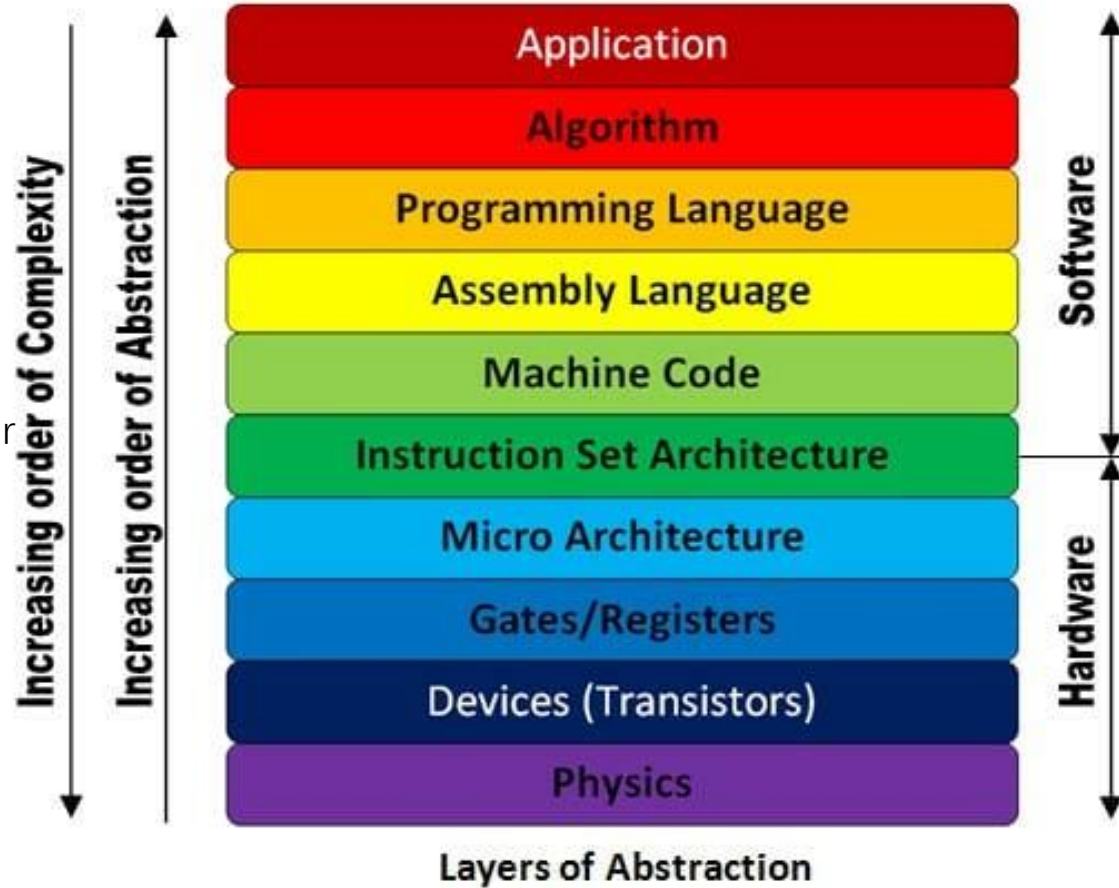
Figure 1.3 The compilation system.

Recall

- C/C++
(.c/.cpp, .h/.hpp)

GCC
↓ Compiler
• Assembly language (.s)
↓ Assembler
• Machine Code (.o, .so, .a)
↓ Linker

CLANG (LLVM)
↓ Compiler
• LLVM IR
↓ LLVM Optimizer
• LLVM Backend
↓ Assembler
↓ Linker



- Java

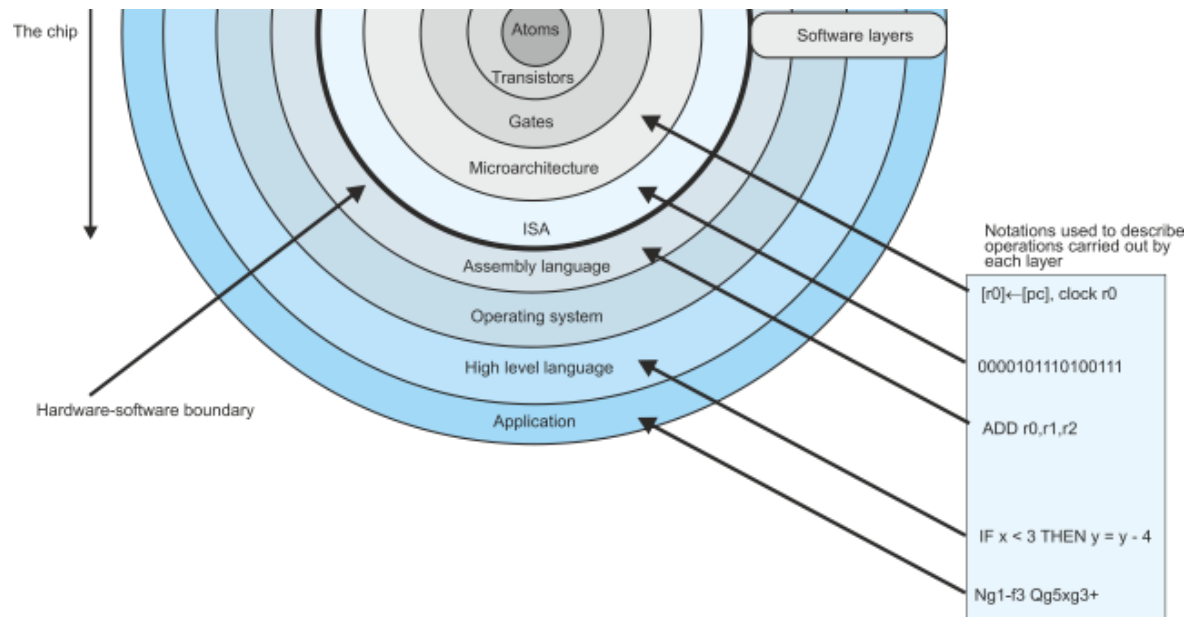
JVM
↓ Compiler
• Java Bytecode
(Java Virtual Machine)
↓ Assembler
↓ Linker
• Machine Code

JIT
↓ Compiler
• Java Bytecode
↓ Just In Time compiler
• Machine code
• (Java Virtual Machine)
↓ Linker
• Machine Code

Two abstractions: ISA

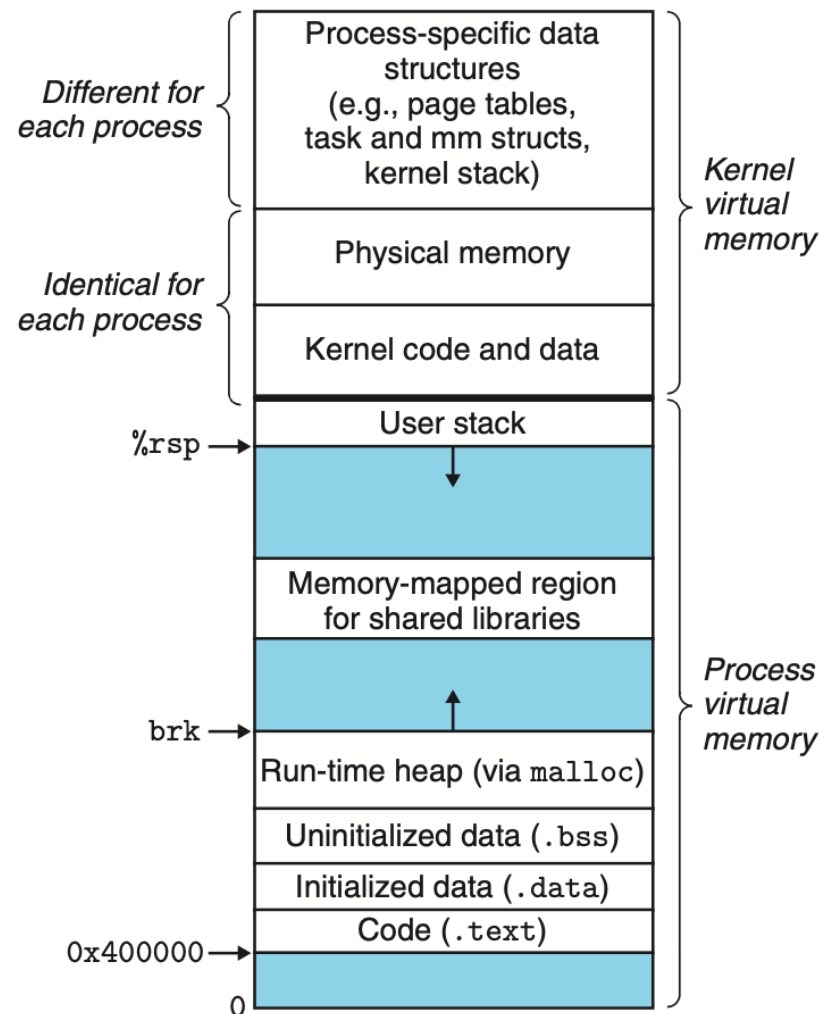
- 定义处理器状态、指令格式和指令对状态的影响
- The hardware/software **interface**
- Examples
 - IA32, **AMD64 (x86-64)**, ARM64, RISC-V, etc.

• ISA \neq 汇编语言



Two abstractions: Virtual Address

- 每个 **进程** 享有同样的虚拟地址空间
- 操作系统的四大抽象之一，随着ICS的学习会逐渐补全拼图



工具链


(1) gcc [-O] -S <file> [-o]

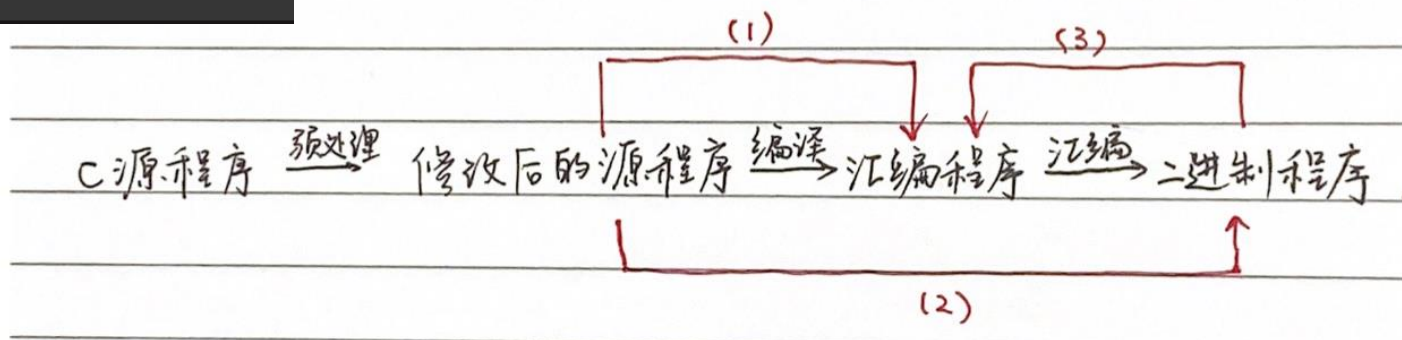
(2) gcc [-O] -c <file> [-o]

(3) objdump -d <file> (> ...)

• gdb and bomblab

(c->s) Compiler Explorer  COMPILER EXPLORER

(o->c) ByteNinja  BINARYNINJA



编译器优化



助教 李翰禹

再具体解释一下优化等级的问题

- O0是几乎所有优化都不做

- O或者-O1是做一部分优化

- Og是-O1优化去除会影响调试器正常工作的优化

- O2是-O1不考虑时间和空间的权衡的情况下做更多优化

- O3是-O2基础上做更多优化，比如一些向量化和循环展开的优化

- Os是-O2优化去除会增加目标代码长度的优化

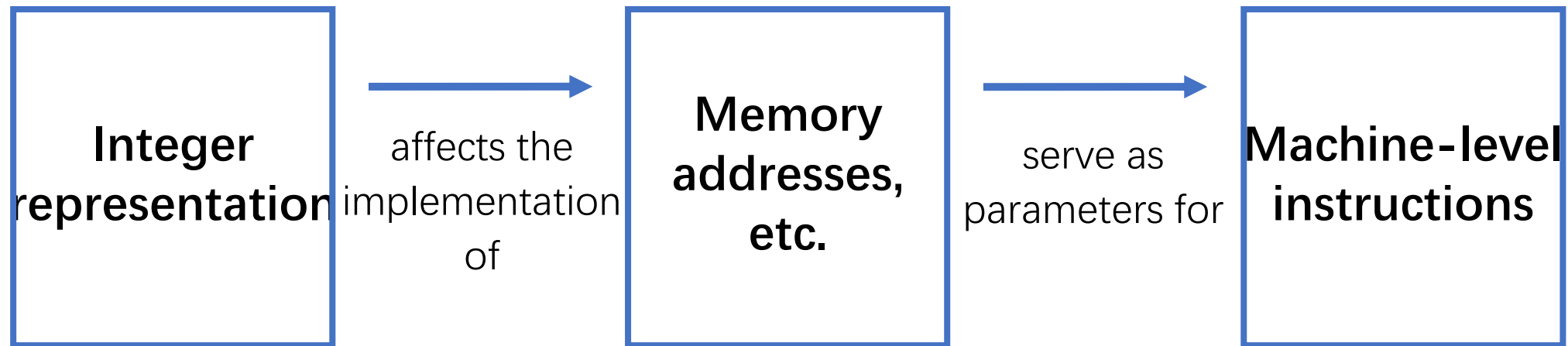
- Ofast是无视严格标准的优化

Basics (CS:APP Ch. 3.1 - 3.5)

贾博暄

Relationships between topics

- (courtesy of TA)



Registers

- Correspondence
- Memory reference in parens: ()
 - **%rax** -> register
 - **(%rax)** -> memory
 - Compare: pointers in C

Note on
naming
b, w, l, q

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.

63	31	15	7	0	
%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

Conventions...

- Instructions that generate 4-byte quantities for a register set the upper 4 bytes of the register to zero

1	<code>movabsq \$0x0011223344556677, %rax</code>	<code>%rax = 0011223344556677</code>
2	<code>movb \$-1, %al</code>	<code>%rax = 00112233445566FF</code>
3	<code>movw \$-1, %ax</code>	<code>%rax = 001122334455FFFF</code>
4	<code>movl \$-1, %eax</code>	<code>%rax = 00000000FFFFFFFF</code>
5	<code>movq \$-1, %rax</code>	<code>%rax = FFFFFFFFFFFFFFFF</code>

- Consequences
 - MOV** instructions get a little weird (e.g. there is no **movz1q**)
 - You might see (for example):
 - movzbl** when your intuition says **movzbq**
Textbook: Problem 3.4
 - Writing to **%eax** on one line and quoting **%rax** on the next

Conventions...

- In a memory reference, the **scaling factor** must be either 1, 2, 4, or 8

$Imm(rb, r_i, s)$

9 (%rax, %rdx, 4)

- Why?
 - Useful when referencing array and structure elements

1、某C语言程序中对数组变量a的声明为“int a[10][10];”，有如下一段代码：

```
for (i=0; i<10; i++)  
    for (j=0; j<10; j++)  
        sum+= a[i][j];
```

假设执行到“sum+= a[i][j];”时，sum的值在%rax中，a[i][0]所在的地址在%rdx中，j在%rsi中，则“sum+= a[i][j];”所对应的指令是（ ）。

- A. addl 0 (%rdx, %rsi, 4), %rax
- B. addl 0 (%rsi, %rdx, 4) , %rax
- C. addl 0 (%rdx, %rsi, 2) , %rax
- D. addl 0 (%rsi, %rdx, 2) , %rax

Conventions...

- x86-64 不允许将除 64 位寄存器以外的寄存器作为寻址模式基地址

1. 判断下列 x86-64 ATT 操作数格式是否合法。

(7) () (%ecx)

- Textbook: Problem 3.3

Here is the code with explanations of the errors:

```
movb $0xF, (%ebx)      Cannot use %ebx as address register
```

Conventions...

- Binary instructions' operands are, in order, _Source_ and _Destination_
 - Intuitive (eg. **MOV**)
 - Counter-intuitive (eg. **SUB**, **CMP**)
- the operand pairs CANNOT be of what type? _Any -> Imm, Mem -> Mem_
 - this applies to **MOV** and Arithmetic

Instruction Classes – Move

- **MOV**

- Operand types: $\$Imm, Reg, Mem$
- Operand pairs: Some are forbidden

- Memory reference

- Full form:

Offset (Base, Index, scale factor)

$$Imm + R[rb] + R[ri] \cdot s$$

- Selected omission yields the other formats
- s limited to 1, 2, 4, 8
- No \$ before Imm here

From\To	Imm	Reg	Mem
Imm		✓	✓
Reg		✓	✓
Mem		✓	

Instruction Classes – Move

- **MOV:** `movl` vs `movq` vs `movabsq`

- `movl`

- Moves double word
 - Higher bits automatically set to 0

- `movq`

- Takes 32-bit value, sign-extends to 64-bit (i.e. quad word)
 - **THEN** moves to destination

- `movabsq`

- Takes 64-bit immediate value and moves to destination
 - **Only allows Imm -> Reg**

Instruction Classes – Move

- “Corner cases”

- **MOVZ**

- there is no `movzlw`

- **MOVS**

- **cltq**

- effectively a compact encoding of

`movslq %eax, %rax`

MOVZ



z: zero-extend
s: sign-extend

MOVS



Instruction Classes – Arithmetic

- **leaq**

- (Important!) **leaq** vs **mov**

- Same format

- **leaq** DOES NOT reference memory; purely numerical

B. `leal (%ecx,%ebx,4), %eax`

C. `movl (%ecx,%ebx,4), %eax`

- Unary: **INC, DEC, NEG, NOT**

- Somewhat self-evident

- Binary: **ADD, SUB, IMUL, XOR, OR, AND**

- 2nd operand (*D*) = Destination

- Mnemonic: **Add** *S* to *D*
Subtract *S* from *D*

`ADD S, D D ← D + S`

`SUB S, D D ← D – S`

Instruction Classes – Shifts

- Shifts
 - **SAL(SHL), SAR, SHR**
 - **sh**ifts (logical) vs **s**hifts (**a**rithmetic)

sh: logical shift

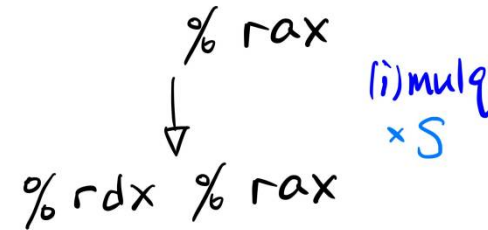
sa: arithmetic shift

- 1st operand (k) = shift amount
 - **must be \$lmm or %c1**
 - actual shift amount (also) depends on type of value to be shifted

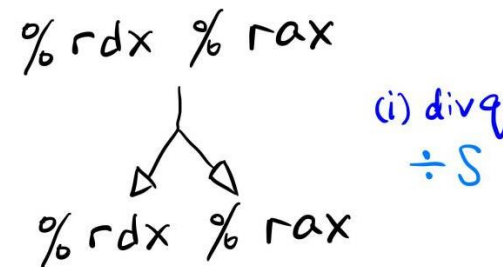
SAL	k, D	$D \leftarrow D \ll k$
SHL	k, D	$D \leftarrow D \ll k$
SAR	k, D	$D \leftarrow D \gg_A k$
SHR	k, D	$D \leftarrow D \gg_L k$

Instruction Classes - 128-bit

- **Single operand**
- **`imulq`, `mulq`**
 - S is multiplicand
 - **`imulq`** S vs **`IMUL`** S, D
- **`idivq`, `divq`**
 - S is divisor
 - Quotient at `%rax`, remainder at `%rdx`
- **`cqto`**
 - Sign extend, presets `%rdx`
 - Compare: **`cltq`**



i: integer



`cltq`:

convert **l** to **q**

`cqto`:

convert **q** to **o**

On to the next section!

Control (CS:APP Ch 3.6)

倪嘉怡

Contents

- 条件码, 跳转指令
- 逆向工程和goto代码
- **if-else, do-while, while, for, switch, ...**
- 条件传送

条件码

- CF Carry Flag：进位；无符号溢出。1进位0不进
- ZF Zero Flag：零。1零0非零
- SF Sign Flag：符号。1负0正
- OF Overflow Flag：溢出；正/负有符号溢出。1溢出0不溢出

- 逻辑运算：进位 CF、溢出 OF 置 0
- 移位运算：溢出 OF 置 0；进位 CF 设置为最后一个被移出的位
- inc, dec：不改变进位标志
- lea：不改变条件码

- cmp = sub -
- test = and &

- <https://www.felixcloutier.com/x86/>
- x86 and amd64 instruction reference

指令	效果	描述
leal 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	$D \leftarrow D \vee S$	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

	CF	ZF	SF	OF	
LEA	✗	✗	✗	✗	INC, DEC
INC	✗	✓	✓	✓	
DEC	✗	✓	✓	✓	
NEG	✓	✓	✓	✓	算术运算
NOT	✗	✗	✗	✗	
ADD	✓	✓	✓	✓	
SUB	✓	✓	✓	✓	逻辑运算
IMUL	✓	✓	✓	✓	
XOR	0	✓	✓	0	
OR	0	✓	✓	0	移位运算
AND	0	✓	✓	0	
SAL	✓	✓	✓	0	
SHL	✓	✓	✓	0	移位运算
SAR	✓	✓	✓	0	
SHR	✓	✓	✓	0	

2、条件码描述了最近一次算术或逻辑操作的属性。下列关于条件码的叙述中，哪一个是不正确的？

- A. `set` 指令可以根据条件码的组合将一个字节设置为 0 或 1
- B. `cmp` 指令和 `test` 指令可以设置条件码但不更改目的寄存器
- C. `leaq` 指令可以设置条件码 CF 和 OF
- D. 除无条件跳转指令 `jmp` 外，其他跳转指令都是根据条件码的某种组合跳转到标号指示的位置

C

`leaq` 不改变条件码

1. 在下列指令中，其执行会影响条件码中的 CF 位的是：

A. `jmp NEXT` B. `jc NEXT` C. `inc %bx` D. `shl $1, %ax`

D

CF: 进位; 无符号溢出

AB. `jump`指令不改变条件码

C. `inc, dec` : 不改变进位标志

D. 移位运算: 进位 CF 设置为最后一个被移出的位

判断两个寄存器中值大小关系，只需 SF（符号）和 ZF（零）两个条件码

- **×**
- 有符号比较：符号SF^溢出OF和零ZF的组合
- 无符号比较：进位CF和零ZF

CMP B,A	CF	ZF	SF	OF
signed				
A=B		1		
A>B		0	0 正	0 无溢出
A>B		0	1 负	1 正溢出
A<B		0	1 负	0 无溢出
A<B		0	0 正	1 负溢出
unsigned				
A=B		1		
A>B	0 无进位	0		
A<B	1 有进位	0		

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 `jb` 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. $(SF \wedge OF) \mid ZF$

A

以 `cmp B, A; jb ...` 为例，跳转 `jb` 需要满足 $A - B > 0$

1) $A - B \neq 0$: $\sim ZF$

2) $OF == 0$ 无溢出: $SF == 0$ 结果为正

3) $OF == 1$ 溢出: $SF == 1$ 结果为负

4) 综合2,3, $\sim (SF \wedge OF)$ 先异或再取反，再与1)取交集

跳转指令

Jump

if Equal

if Not Equal

if Sign

if Not Sign

if Greater

if Not Greater

if Less

if Not Less

if Above

if Not Above

if Below

if Not Below

指令	同义名	跳转条件	描述
jmp <i>Label</i>		1	直接跳转
jmp <i>*Operand</i>		1	间接跳转
jje <i>Label</i>	jz	ZF	相等/零
jne <i>Label</i>	jnz	~ZF	不相等/非零
js <i>Label</i>		SF	负数
jns <i>Label</i>		~SF	非负数
jg <i>Label</i>	jnle	~(SF ^ OF) & ~ZF	大于 (有符号>)
jge <i>Label</i>	jnl	~(SF ^ OF)	大于或等于 (有符号>=)
jl <i>Label</i>	jnge	SF ^ OF	小于 (有符号<)
jle <i>Label</i>	jng	(SF ^ OF) ZF	小于或等于 (有符号<=)
ja <i>Label</i>	jnbe	~CF & ~ZF	超过 (无符号>)
jae <i>Label</i>	jnb	~CF	超过或相等 (无符号>=)
jb <i>Label</i>	jnae	CF	低于 (无符号<)
jbe <i>Label</i>	jna	CF ZF	低于或相等 (无符号<=)

“*” 相当于C语言指针，汇编的括号

%rax 寄存器
(%rax) 内存
寄存器+括号=内存

jmp 的编码

- jmp 指令的机器编码: PC-relative 编码
- 编码 + jmp 后面那条指令的地址 = 目标地址

.....

3: eb 03

5: 48 d1 f8

8: 48 85 c0

.....

(03 + 5 = 8)

jmp 8<loop+0x8>

sar %rax

test %rax ,%rax

jmp 的编码

- 即使链接之后，这些指令重定位到了不同地址，仍有
编码 + jmp 后面那条指令的地址 = 目标地址 成立

.....

4004d3:	eb 03	jmp 8<loop+0x8>
4004d5:	48 d1 f8	%rax
4004d8:	48 85 c0	test %rax ,%rax

.....

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

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

B

$$\text{f0 (dec: -16)} + 400022 = 400012$$



练习题 3.15 在下面这些反汇编二进制代码节选中，有些信息被 X 代替了。回答下列关于这些指令的问题。

A. 下面 je 指令的目标是什么？（在此，你不需要知道任何有关 call 指令的信息。）

```
804828f:      74 05                je      XXXXXXXX
8048291:      e8 1e 00 00 00    call   80482b4
```

B. 下面 jb 指令的目标是什么？

```
8048357:      72 e7                jb      XXXXXXXX
8048359:      c6 05 10 a0 04 08 01  movb    $0x1,0x804a010
```

C. mov 指令的地址是多少？

```
XXXXXXX:      74 12                je      8048391
XXXXXXX:      b8 00 00 00 00    mov     $0x0,%eax
```

D. 在下面的代码中，跳转目标的编码是 PC 相关的，且是一个 4 字节的补码数。字节按照从最低位到最高位的顺序列出，反映出 IA32 的小端法字节顺序。跳转目标的地址是什么？

```
80482bf:      e9 e0 ff ff ff      jmp     XXXXXXXX
80482c4:      90                  nop
```



练习题 3.15 在下面这些反汇编二进制代码节选中，有些信息被 X 代替了。回答下列关于这些指令的问题。

A. 下面 je 指令的目标是什么？（在此，你不需要知道任何有关 call 指令的信息。）

804828f:	74 05	je	XXXXXXX	05 + 8048291 = 8048296
<u>8048291:</u>	e8 1e 00 00 00	call	80482b4	

B. 下面 jb 指令的目标是什么？

8048357:	72 e7	jb	XXXXXXX
8048359:	c6 05 10 a0 04 08 01	movb	\$0x1,0x804a010

C. mov 指令的地址是多少？

XXXXXXX:	74 12	je	8048391
XXXXXXX:	b8 00 00 00 00	mov	\$0x0,%eax

D. 在下面的代码中，跳转目标的编码是 PC 相关的，且是一个 4 字节的补码数。字节按照从最低位到最高位的顺序列出，反映出 IA32 的小端法字节顺序。跳转目标的地址是什么？

80482bf:	e9 e0 ff ff ff	jmp	XXXXXXX
80482c4:	90	nop	



练习题 3.15 在下面这些反汇编二进制代码节选中，有些信息被 X 代替了。回答下列关于这些指令的问题。

A. 下面 je 指令的目标是什么？（在此，你不需要知道任何有关 call 指令的信息。）

804828f:	74 05	je	XXXXXXX	$05 + 8048291 = 8048296$
<u>8048291:</u>	e8 1e 00 00 00	call	80482b4	

B. 下面 jb 指令的目标是什么？

8048357:	72 e7	jb	XXXXXXX	$e7 \text{ (dec: -25)} + 8048359 = 8048340$
<u>8048359:</u>	c6 05 10 a0 04 08 01	movb	\$0x1,0x804a010	

C. mov 指令的地址是多少？

XXXXXXX:	74 12	je	8048391
XXXXXXX:	b8 00 00 00 00	mov	\$0x0,%eax

D. 在下面的代码中，跳转目标的编码是 PC 相关的，且是一个 4 字节的补码数。字节按照从最低位到最高位的顺序列出，反映出 IA32 的小端法字节顺序。跳转目标的地址是什么？

80482bf:	e9 e0 ff ff ff	jmp	XXXXXXX
80482c4:	90	nop	



练习题 3.15 在下面这些反汇编二进制代码节选中，有些信息被 X 代替了。回答下列关于这些指令的问题。

A. 下面 je 指令的目标是什么？（在此，你不需要知道任何有关 call 指令的信息。）

804828f:	74 05	je	<u>XXXXXXX</u>	$05 + 8048291 = 8048296$
<u>8048291:</u>	e8 1e 00 00 00	call	80482b4	

B. 下面 jb 指令的目标是什么？

8048357:	72 e7	jb	<u>XXXXXXX</u>	$e7 \text{ (dec: -25)} + 8048359 = 8048340$
<u>8048359:</u>	c6 05 10 a0 04 08 01	movb	\$0x1,0x804a010	

C. mov 指令的地址是多少？

XXXXXXX:	74 12	je	<u>8048391</u>	$12 + 8048385 = 8048391$
<u>XXXXXXX:</u>	b8 00 00 00 00	mov	\$0x0,%eax	

D. 在下面的代码中，跳转目标的编码是 PC 相关的，且是一个 4 字节的补码数。字节按照从最低位到最高位的顺序列出，反映出 IA32 的小端法字节顺序。跳转目标的地址是什么？

80482bf:	e9 e0 ff ff ff	jmp	XXXXXXX
80482c4:	90	nop	



练习题 3.15 在下面这些反汇编二进制代码节选中，有些信息被 X 代替了。回答下列关于这些指令的问题。

A. 下面 je 指令的目标是什么？（在此，你不需要知道任何有关 call 指令的信息。）

804828f:	74 05	je	<u>XXXXXXX</u>	$05 + 8048291 = 8048296$
<u>8048291:</u>	e8 1e 00 00 00	call	80482b4	

B. 下面 jb 指令的目标是什么？

8048357:	72 e7	jb	<u>XXXXXXX</u>	$e7 \text{ (dec: -25)} + 8048359 = 8048340$
<u>8048359:</u>	c6 05 10 a0 04 08 01	movb	\$0x1,0x804a010	

C. mov 指令的地址是多少？

XXXXXXX:	74 12	je	<u>8048391</u>	$12 + 8048385 = 8048397$
<u>XXXXXXX:</u>	b8 00 00 00 00	mov	\$0x0,%eax	

D. 在下面的代码中，跳转目标的编码是 PC 相关的，且是一个 4 字节的补码数。字节按照从最低位到最高位的顺序列出，反映出 IA32 的小端法字节顺序。跳转目标的地址是什么？

80482bf:	e9 e0 ff ff ff	jmp	<u>XXXXXXX</u>	$fffffe0 \text{ (dec: -32)} + 80482c4 = 80482a4$
<u>80482c4:</u>	90	nop		

逆向工程和goto

- goto代码：描述汇编代码程序控制流的C程序
- 逆向工程：汇编代码->C语言代码

```
x at %ebp+8, y at %ebp+12
1    movl    8(%ebp), %edx    Get x
2    movl    12(%ebp), %eax   Get y
3    cmpl    %eax, %edx       Compare x:y
4    jge     .L2              if >= goto x_ge_y
5    subl    %edx, %eax       Compute result = y-x
6    jmp     .L3              Goto done
7  .L2:                                x_ge_y:
8    subl    %eax, %edx       Compute result = x-y
9    movl    %edx, %eax       Set result as return value
10  .L3:                                done: Begin completion code
```

c). 产生的汇编代码

```
1  int absdiff(int x, int y) {
2      if (x < y)
3          return y - x;
4      else
5          return x - y;
6  }
```

a) 原始的 C 语言代码

goto: 正向视角

```
1 int absdiff(int x, int y) {  
2     if (x < y)  
3         return y - x;  
4     else  
5         return x - y;  
6 }
```

a) 原始的 C 语言代码

```
1 int gotodiff(int x, int y) {  
2     int result;  
3     if (x >= y)  
4         goto x_ge_y;  
5     result = y - x;  
6     goto done;  
7     x_ge_y:  
8     result = x - y;  
9     done:  
10    return result;  
11 }
```

b) 与之等价的 goto 版本

Basic: mov, add, sub...

专用于设置条件码: cmp, test...

跳转: jmp, je, jg...

如何把else转化为汇编?

```
    x at %ebp+8, y at %ebp+12  
1     movl    8(%ebp), %edx    Get x  
2     movl    12(%ebp), %eax   Get y  
3     cmpl    %eax, %edx       Compare x:y  
4     jge     .L2              if >= goto x_ge_y  
5     subl    %edx, %eax       Compute result = y-x  
6     jmp     .L3              Goto done  
7     .L2:                   x_ge_y:  
8     subl    %eax, %edx       Compute result = x-y  
9     movl    %edx, %eax       Set result as return value  
10    .L3:                   done: Begin completion code
```

c) 产生的汇编代码

goto: 逆向视角

```
x at %ebp+8, y at %ebp+12
1    movl    8(%ebp), %edx
2    movl    12(%ebp), %eax
3    cmpl    %eax, %edx
4    jge     .L2
5    subl    %edx, %eax
6    jmp     .L3
7    .L2:
8    subl    %eax, %edx
9    movl    %edx, %eax
10   .L3:
```

c) 产生的汇编

```
1  int gotodiff(int x, int y) {
2      int result;
3      if (x >= y)
4          goto x_ge_y;
5      result = y - x;
6      goto done;
7      x_ge_y:
8      result = x - y;
9      done:
10     return result;
11 }
```

b) 与之等价的 goto 版本

```
1  int absdiff(int x, int y) {
2      if (x < y)
3          return y - x;
4      else
5          return x - y;
6  }
```

a) 原始的 C 语言代码

循环：do-while

```
do
    body-statement
while (test-expr);
```

```
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
```

```
1  int fact_do(int n)
2  {
3      int result = 1;
4      do {
5          result *= n;
6          n = n-1;
7      } while (n > 1);
8      return result;
9  }
```

a) C 代码

```
Argument: n at %ebp+8
Registers: n in %edx, result in %eax
1      movl    8(%ebp), %edx    Get n
2      movl    $1, %eax        Set result = 1
3      .L2:                    loop:
4          imull %edx, %eax      Compute result *= n
5          subl    $1, %edx      Decrement n
6          cmpl    $1, %edx      Compare n:1
7          jg      .L2           If >, goto loop
Return result
```

c) 对应的汇编代码

循环: while

```
while (test-expr)
    body-statement
```

```
t = test-expr;
if (!t)
    goto done;
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
done:
```

```
1  int fact_while(int n)
2  {
3      int result = 1;
4      while (n > 1) {
5          result *= n;
6          n = n-1;
7      }
8      return result;
9  }
```

a) C 代码

```
1  int fact_while_goto(int n)
2  {
3      int result = 1;
4      if (n <= 1)
5          goto done;
6      loop:
7          result *= n;
8          n = n-1;
9      if (n > 1)
10         goto loop;
11     done:
12         return result;
13 }
```

b) 等价的 goto 版本

Argument: n at %ebp+8
Registers: n in %edx, result in %eax

```
1  movl    8(%ebp), %edx    Get n
2  movl    $1, %eax        Set result = 1
3  cmpl    $1, %edx        Compare n:1
4  jle     .L7             If <=, goto done
5  .L10:
6  imull   %edx, %eax       Compute result *= n
7  subl    $1, %edx        Decrement n
8  cmpl    $1, %edx        Compare n:1
9  jg      .L10            If >, goto loop
10 .L7:
    Return result
done:
```

c) 对应的汇编代码

循环: for

```
for (init-expr; test-expr; update-expr)
    body-statement
```

```
1  int fact_for(int n)
2  {
3      int i;
4      int result = 1;
5      for (i = 2; i <= n; i++)
6          result *= i;
7      return result;
8  }
```

```
init-expr;
while (test-expr) {
    body-statement
    update-expr;
}
```

```
init-expr;
t = test-expr;
if (!t)
    goto done;
loop:
    body-statement
    update-expr;
    t = test-expr;
    if (t)
        goto loop;
done:
```

```
1  int fact_for_goto(int n)
2  {
3      int i = 2;
4      int result = 1;
5      if (!(i <= n))
6          goto done;
7      loop:
8          result *= i;
9          i++;
10         if (i <= n)
11             goto loop;
12     done:
13         return result;
14 }
```

Argument: n at %ebp+8

Registers: n in %ecx, i in %edx, result in %eax

```
1  movl    8(%ebp), %ecx    Get n
2  movl    $2, %edx        Set i to 2      (init)
3  movl    $1, %eax        Set result to 1
4  cmpl    $1, %ecx        Compare n:1      (!test)
5  jle     .L14            If <=, goto done
6  .L17:
7  imull    %edx, %eax      Compute result *= i (body)
8  addl    $1, %edx        Increment i      (update)
9  cmpl    %edx, %ecx      Compare n:i      (test)
10 jge     .L17            If >=, goto loop
11 .L14:
done:
```

switch

• 跳转表内的数组引用

```
1  int switch_eg(int x, int n) {
2      int result = x;
3
4      switch (n) {
5
6          case 100:
7              result += 13;
8              break;
9
10         case 102:
11             result += 10;
12             /* Fall through */
13
14         case 103:
15             result += 11;
16             break;
17
18         case 104:
19         case 106:
20             result += result;
21             break;
22
23         default:
24             result = 0;
25     }
26
27     return result;
28 }
```

a) switch 语句

```
1  int switch_eg_impl(int x, int n) {
2      /* Table of code pointers */
3      static void *jt[7] = {
4          &&loc_A, &&loc_def, &&loc_B,
5          &&loc_C, &&loc_D, &&loc_def,
6          &&loc_D
7      };
8
9      unsigned index = n - 100;
10     int result;
11
12     if (index > 6)
13         goto loc_def;
14
15     /* Multiway branch */
16     goto *jt[index];
17
18     loc_def: /* Default case*/
19     result = 0;
20     goto done;
21
22     loc_C: /* Case 103 */
23     result = x;
24     goto rest;
25
26     loc_A: /* Case 100 */
27     result = x * 13;
28     goto done;
29
30     loc_B: /* Case 102 */
31     result = x + 10;
32     /* Fall through */
33
34     rest: /* Finish case 103 */
35     result += 11;
36     goto done;
37
38     loc_D: /* Cases 104, 106 */
39     result = x * x;
40     /* Fall through */
41
42     done:
43     return result;
44 }
```

b) 翻译到扩展的 C 语言

```
x at %ebp+8, n at %ebp+12
1  movl    8(%ebp), %edx          Get x
2  movl    12(%ebp), %eax         Get n
3                                     Set up jump table access
4  subl    $100, %eax            Compute index = n-100
5  cmpl    $6, %eax              Compare index:6
6  ja      .L2                    If >, goto loc_def
7  jmp     *.L7(,%eax,4)          Goto *jt[index]
8                                     Default case
9  .L2:                          loc_def:
10     movl    $0, %eax            result = 0;
11     jmp     .L8                  Goto done
12                                     Case 103
13     .L5:                          loc_C:
14     movl    %edx, %eax          result = x;
15     jmp     .L9                  Goto rest
16                                     Case 100
17     .L3:                          loc_A:
18     leal    (%edx,%edx,2), %eax  result = x*3;
19     leal    (%edx,%eax,4), %eax  result = x+4*result
20     jmp     .L8                  Goto done
21                                     Case 102
22     .L4:                          loc_B:
23     leal    10(%edx), %eax       result = x+10
24                                     Fall through
25     .L9:                          rest:
26     addl    $11, %eax            result += 11;
27     jmp     .L8                  Goto done
28                                     Cases 104, 106
29     .L6:                          loc_D
30     movl    %edx, %eax          result = x
31     imull   %edx, %eax          result *= x
32                                     Fall through
33     .L8:                          done:
34                                     Return result
```

图 3-19 图 3-18 中 switch 语句示例的汇编代码

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

C

1) .L2是default, 说明没有127

2) 0-8中去除127, 还剩034568

3) subl \$48, %eax: 与8比较前先减去48, '0'=48, 说明是字符而非数字

8、对简单的 switch 语句常采用跳转表的方式实现，在 x86-64 系统中，下述最有可能正确的 switch 分支跳转汇编指令为哪个？

- A. `jmp .L3(,%eax,4)`
- B. `jmp .L3(,%eax,8)`
- C. `jmp *.L3(,%eax,4)`
- D. `jmp *.L3(,%eax,8)`

答：()

D

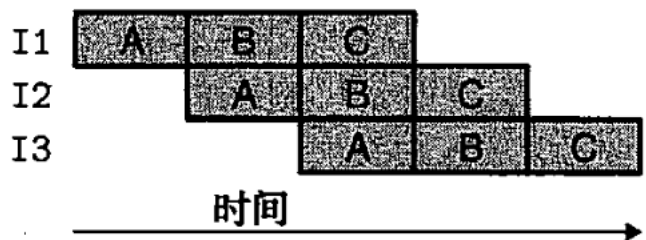
跳转表内存储的数据类型是地址，64位上地址8字节

条件传送

- 条件**控制**转移->条件**数据**传送：可能提高效率
- 计算流水线（p264）：取指—译码—执行—访存—写回

```
1  int absdiff(int x, int y) {  
2      return x < y ? y-x : x-y;  
3  }
```

a) 原始的 C 语言代码



```
1  int cmovdiff(int x, int y) {  
2      int tval = y-x;  
3      int rval = x-y;  
4      int test = x < y;  
5      /* Line below requires  
6          single instruction: */  
7      if (test) rval = tval;  
8      return rval;  
9  }
```

b) 使用条件赋值的实现

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

Bad Performance

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

Unsafe

- Both values get computed
- May have undesirable effects

当test-expr p为False, *p间接引用空指针

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

Illegal

- Both values get computed
- Must be side-effect free

两个分支同时改变全局变量

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

。

<pre>long f1(long a, long b) { return (++a > --b) ? a : b; }</pre>	Y	N
<pre>long f2(long *a, long *b) { return (*a > *b) ? --(*a) : (*b)--; }</pre>	Y	N
<pre>long f3(long *a, long *b) { return a ? *a : (b ? *b : 0); }</pre>	Y	N
<pre>long f4(long a, long b) { return (a > b) ? a++ : ++b; }</pre>	Y	N

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。

<pre>long f1(long a, long b) { return (++a > --b) ? a : b; }</pre>	<div><div>Y</div>N</div>
<pre>long f2(long *a, long *b) { return (*a > *b) ? --(*a) : (*b)--; }</pre>	<div>Y<div>N</div></div>
<pre>long f3(long *a, long *b) { return a ? *a : (b ? *b : 0); }</pre>	<div>Y<div>N</div></div>
<pre>long f4(long a, long b) { return (a > b) ? a++ : ++b; }</pre>	<div><div>Y</div>N</div>

【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。

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

D

A. 16\32\64位，不支持单字节

B. Risks or Side Effects

C. 两个分支需要大量计算时，效率更低

D. 无条件指令操作数的长度显式编码在指令名中，可以从目标寄存器名字推断出条件传送指令的操作数长度，不需后缀表明操作数长度

Practice

王善上

寄存器大小记忆

规律：

- b: 结尾为l
- w: 两个字符
- d: e开头
- r: r开头

%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl

易错点

- 复制double word
- movs与movz
- cltq

T1详解

(1)从%ax复制16b: 0x0...0CDEF

(2)做有符号扩充, %bx最高位是C, 即1100, 于是扩1: 0xF...FCDEF

(3)注意复制32b的时候, 高位也会被清0: 0x00000000FFFCDEF

(4)Clcq = movslq %eax %rax: 0xF...F89ABCDEF

(5)~(8)查表填入对应大小即可。

(5)w

(6)wq

(7)l

T2

- Long
- 一般机器（比如intel & ios）都是小端，题目多数都是在这个环境下的，不过提到网络传输一定要反应过来是大端
- 推导过程

T2详解

转化后的C语言代码如右图

先定位出汇编代码中各jump的位置，然后将两份代码的代码块对应上

然后细化定位，还可以根据比较明显的立即数，和代码中出现较少的位运算辅助定位。

```
y=0;
if(x>1)
    goto L1;
return x;
m=(0x40000000<<32)+0;
goto test;

L2:
b=y+m;
y>>=1;
if(x<b)
    goto L3;
x-=b;
y+=m;
L3:
m>>=2;
test:
if(m!=0)
    goto L2;

return y;
```

T2答案

(1)\$0x0

(2)4004e1

(3)40052f

(4)\$0x40000000

(5)400524

(6)(%rdx,%rax,1)

(7)40051f

(8)\$0x2

(9)4004f1

(10)%rax

*T3附加练习

8. 将下列汇编代码翻译成 C 代码

```
func:
    movl    $1, %eax
    jmp .L2
.L4:
    testb   $1, %sil
    je      .L3
    imulq   %rdi, %rax
.L3:
    sarq    %rsi
    imulq   %rdi, %rdi
.L2:
    testq   %rsi, %rsi
    jg      .L4
    rep ret
```

```
// a in %rdi, b in %rsi
long func(long a, long b) {
    long ans = _____;
    while (_____) {
        if (_____)
            ans = _____;
        b = _____;
        a = _____;
    }
    return ans;
}
```

*T3详解

由汇编第一句可得 $ans=1$ ，并且确定 ans 为 $\%eax$

While结构，则L2为test判断，则为 $b>0$

由L2后的jg .L4推出L4为while正文，正文部分一一对应即可

答案

```
long func(long a, long b) {  
    long ans = 1;  
    while (b > 0) {  
        if (b & 1)  
            ans = ans * a;  
        b = b >> 1;  
        a = a * a;  
    }  
    return ans;  
}
```

The End